

University of Cincinnati

# U-net: Medical Image Segmentation

Deep learning: Homework 3

Benjamin PHAN

## Contents

INTRODUCTION .....	2
I. DATA.....	2
A. Data preview.....	2
B. Data split.....	3
C. Data preprocess and normalization.....	3
II. Models (U – net).....	4
A. U-Net 2 layers .....	7
B. U-Net 3 layers .....	8
C. U-Net 4 layers .....	9
III. Objective.....	10
IV. Optimization .....	10
V. Model selection .....	10
VI. Model performance.....	11

## INTRODUCTION

The U-Net architecture is characterized by its unique U-shaped structure, which consists of a contracting path followed by an expansive path. This design allows U-Net to capture fine-grained details while retaining contextual information, making it exceptionally well-suited for tasks such as medical image segmentation.

### I. DATA

#### A. Data preview

The "Retina Blood Vessels" dataset is designed for the task of segmenting blood vessels in retinal images. It plays a crucial role in the field of medical image analysis, particularly in the diagnosis and monitoring of various retinal diseases. Here is an overview of the dataset:

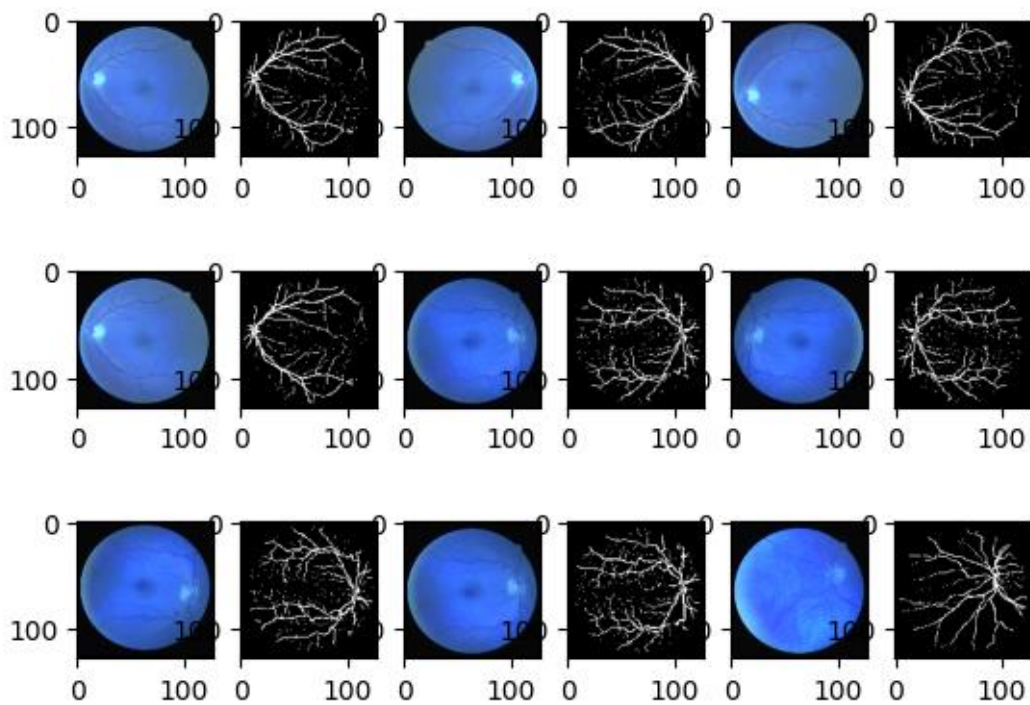


Figure 1 First images

## B. Data split

```
train_image_folder = 'DL_Assignment3/Data/train/image'
train_mask_folder = 'DL_Assignment3/Data/train/mask'
test_image_folder = 'DL_Assignment3/Data/test/image'
test_mask_folder = 'DL_Assignment3/Data/test/mask'
```

In the dataset given for this homework, the training set and testing set are already split into 80 images/masks for the training set and 20 images/masks for the testing set.

## C. Data preprocess and normalization

```
size = 128
for image_file in train_image_files:
    img = cv2.resize(cv2.imread(image_file), (size, size))
    grayscale_images.append(cv2.cvtColor(img,
cv2.COLOR_BGR2GRAY).reshape(size, size, 1))
    if img is not None:
        images.append(img)
train_images = np.array(images)
grayscale_train_images = np.array(grayscale_images)
```

Each image, represented by the 'image\_file' variable, is loaded and resized to a fixed dimension of 128x128 pixels using OpenCV's 'resize' function so I can avoid Out of Memory. Additionally, the images are converted to grayscale using 'cv2.cvtColor' and reshaped to have a single channel, which is essential for some computer vision tasks. The processed color images are stored in the 'train\_images' array, while their grayscale counterparts are saved in the 'grayscale\_train\_images' array. This preprocessing prepares the data for subsequent machine learning or computer vision tasks by ensuring uniform size and color channel dimensions. With that we can later trained our model on 3 channels images or 1 channel images.

For masks we do the same but because we are performing a binary segmentation, we don't need a mask with 3 channels, so our labels will be masks with 1 channel.

Additionally, we compute a normalization on the images and masks.

Images :

```
train_images_n = (train_images - np.min(train_images)) /
(np.max(train_images) - np.min(train_images))
```

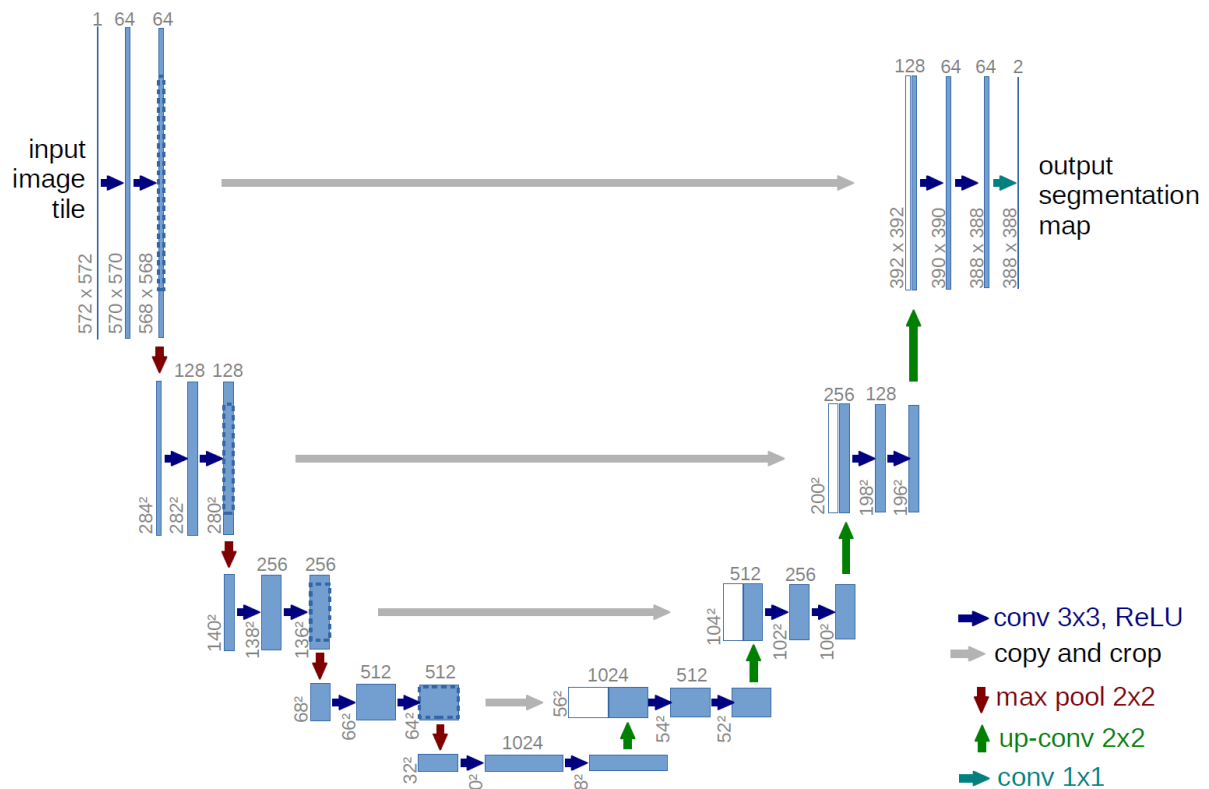
Masks and grayscale images:

```
grayscale_train_masks_n = grayscale_train_masks/255
```

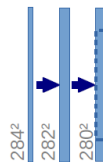
Plus, a threshold is applied for masks (making values 0 or 1):

```
thresholded_grayscale_train_masks_n = np.array([np.where(mask_image <
0.5, 0, 1) for mask_image in grayscale_train_masks_n])
```

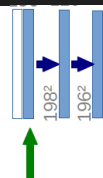
## II. Models (U – net)



To make the U-net model, I create two blocks one for encoder part and one for the decoder part:



```
def blockConv(input, output):
    x = tf.keras.layers.Conv2D(output, kernel_size=3, padding="same")(input)
    x = tf.keras.layers.ReLU()(x)
    x = tf.keras.layers.Conv2D(output, kernel_size=3, padding="same")(x)
    x = tf.keras.layers.ReLU()(x)
    return x
```



```
def blockUpConv(input, skip, output):
    x = tf.keras.layers.Conv2DTranspose(output, 3, 2, padding="same")(input)
    x = tf.keras.layers.concatenate([x, skip])
    x = blockConv(x, output)
    return x
```

```

def UNet(input_shape,num_classes, LR, nblayers):
    model = tf.keras.Sequential()
    input = tf.keras.Input(input_shape)
    x = input
    output = 64
    skipList = []
    #Encoder
    for i in range(nblayers):
        x = blockConv(x, output*(2**i))
        skipList.append(x)
        x = tf.keras.layers.MaxPool2D(2)(x)

    # Bottom of the u-net

    x = tf.keras.layers.Conv2D(output*(2**nblayers), kernel_size=3,
padding="same")(x)
    x = tf.keras.layers.ReLU()(x)
    x = tf.keras.layers.Conv2D(output*(2**nblayers), kernel_size=3,
padding="same")(x)
    x = tf.keras.layers.ReLU()(x)

    # Decoder

    for i in range(nblayers):
        x = blockUpConv(x, skipList[nblayers-(i+1)], output*(2**(nblayers-i)))

    x = tf.keras.layers.Conv2D(num_classes, (1,1), padding="same")(x)

    l = 'binary_crossentropy'
    outputs = tf.keras.activations.sigmoid(x)

    model = tf.keras.Model(input, outputs, name="U-Net")

    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=LR),
        loss = l,
        metrics =
['accuracy',tf.keras.metrics.BinaryIoU(threshold=0.5),tf.keras.metrics.Precisi
on(),tf.keras.metrics.Recall()])

    model.summary()

    return model

```


Here a 2 layers U-Net is 2 descending blocks, the bottom, and 2 ascending blocks.

Dice coefficient (F1 score) :

$$\frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

That's why, to compute the dice coefficient I need the Precision and Recall metrics.

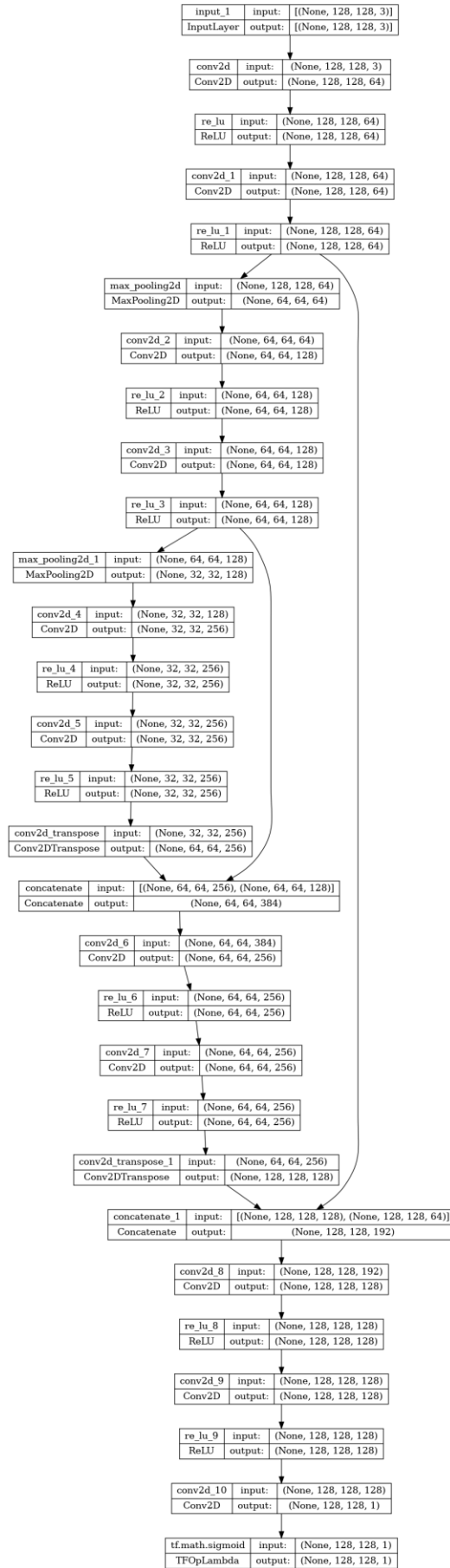
Binary IoU is IoU but applied with a threshold on the output:

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$


In the next part I plotted the model with (to see more detailed of the model we need to zoom but we can clearly see the blocks):

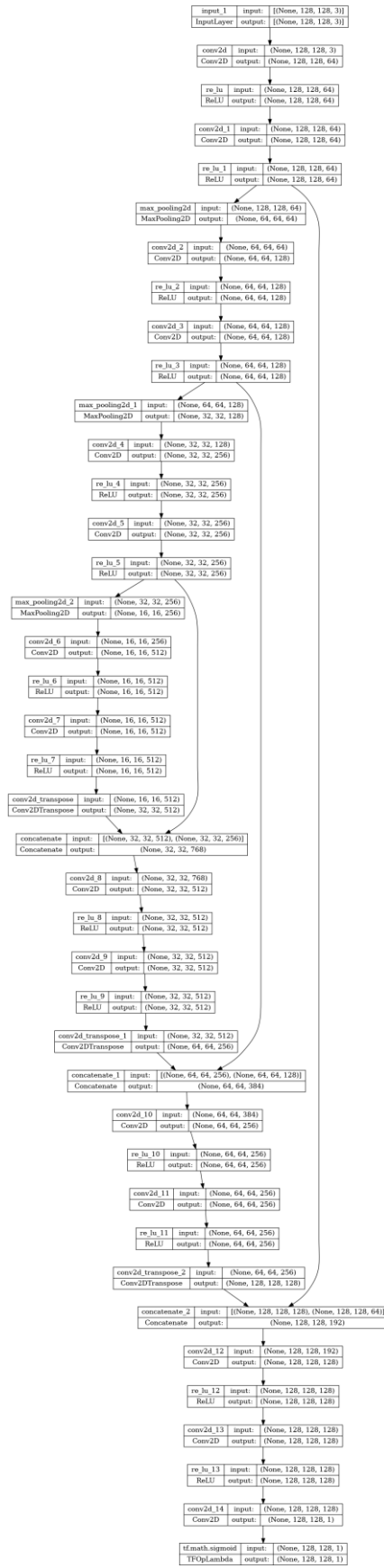
```
tf.keras.utils.plot_model(model, to_file='model_plot.png', show_shapes=True,
show_layer_names=True)
```

## A. U-Net 2 layers

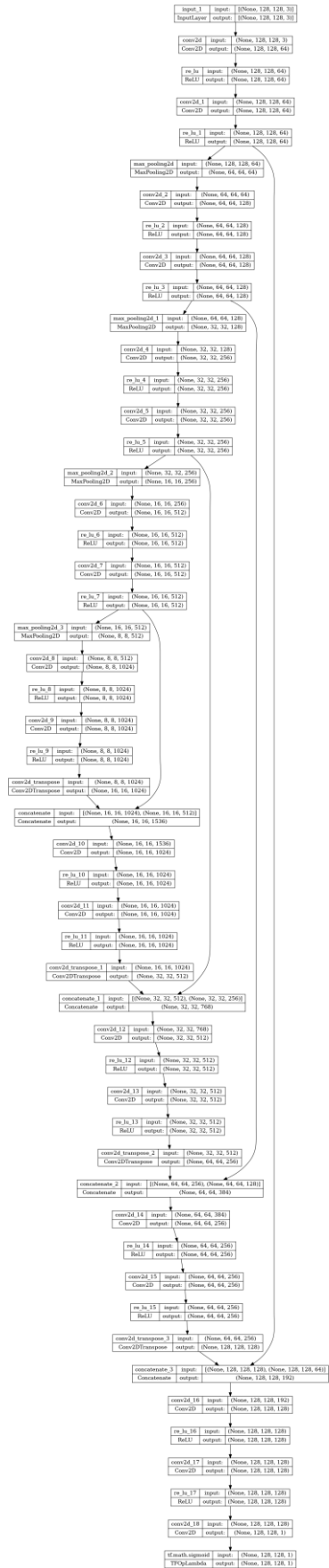




## B. U-Net 3 layers



C. U-Net 4 layers



### III. Objective

Binary Cross-entropy is the loss function used to train the different models.

$$H_p(q) = -\frac{1}{N} \sum_{i=1}^N y_i \cdot \log(p(y_i)) + (1 - y_i) \cdot \log(1 - p(y_i))$$

### IV. Optimization

The Adam optimizer, short for Adaptive Moment Estimation, is a widely adopted optimization algorithm for training deep neural networks. It combines the advantages of two other popular optimizers, namely RMSprop and Momentum. Adam adapts the learning rates of individual model parameters based on their historical gradients and updates, making it highly effective in optimizing models with complex and non-stationary loss surfaces.

The key reasons for choosing Adam as an optimization algorithm is its ability to accelerate convergence during training, typically resulting in faster model training and convergence to a good solution.

### V. Model selection

By comparing accuracy scores between the training and testing datasets, we can effectively gauge whether a model is encountering issues of underfitting or overfitting. For instance, if a model exhibits strong performance on the training data but significantly lower accuracy on the test set, it's indicative of overfitting. On contrary if the model shows low performance, it might be underfitting.

During the training process, even as the accuracy on the training data appears to improve, a drop in accuracy on the validation dataset can be observed. This is often a consequence of overfitting. To avoid this, early stopping can be employed to halt training and preserve the model's best parameters, ensuring better generalization.

## VI. Model performance

Best Model	Dice   IoU
U-Net 2 layers	0.965   0.965
U-Net 3 layers	0.991   0.991
U-Net 4 layers	0.992   0.992

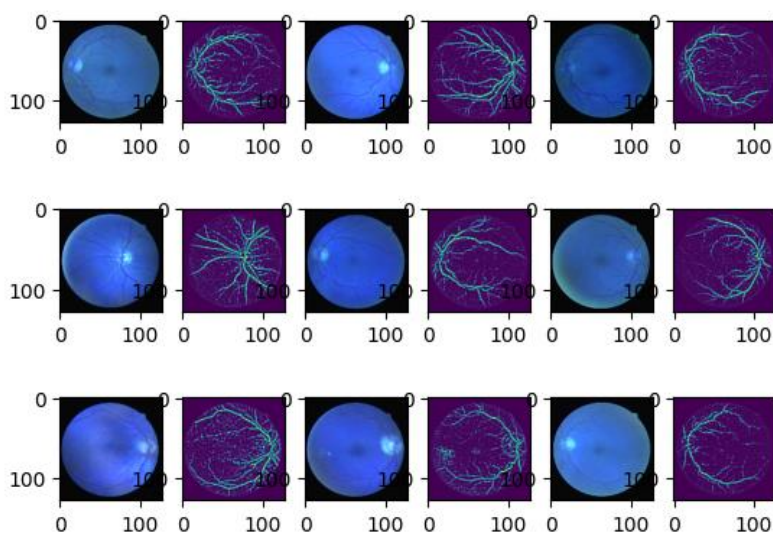
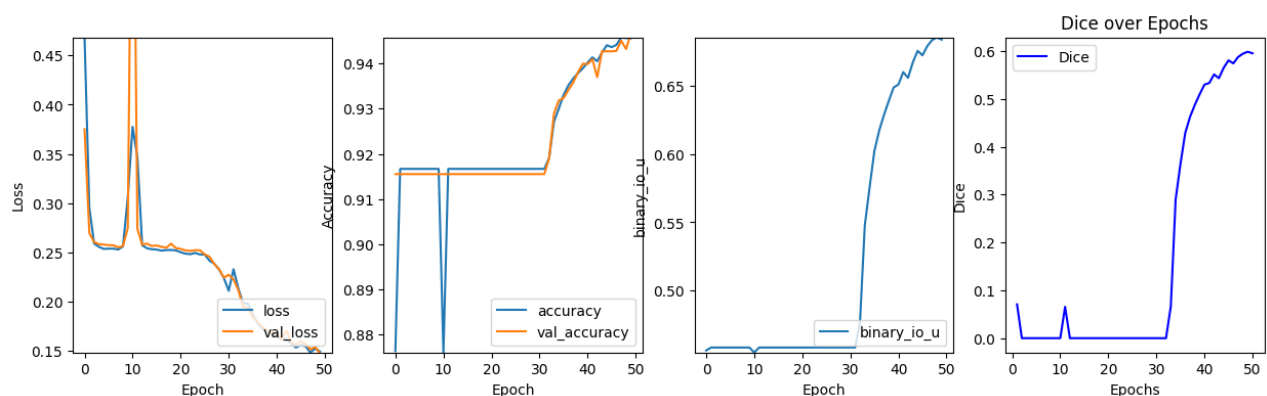
Table 1

	Normalization Dice   IoU	No Normalization Dice   IoU
U-Net 2 layers	0.77   0.80	0.75   0.78
U-Net 3 layers	0.78   0.80	0.78   0.80
U-Net 4 layers	0.62   0.70	0.75   0.78

Table 2

After training the different models with a learning rate of 0.001, they can achieve good performance over the test set. In my case, there is not a significant difference between the data normalized or not and for 1 channel or 3 channels I also get almost the same results. I tried with a learning rate of 0.0001 but I didn't achieve any better performance and the results are similar. Nonetheless with a learning rate of 0.1 or 0.01 training my model results as an error.

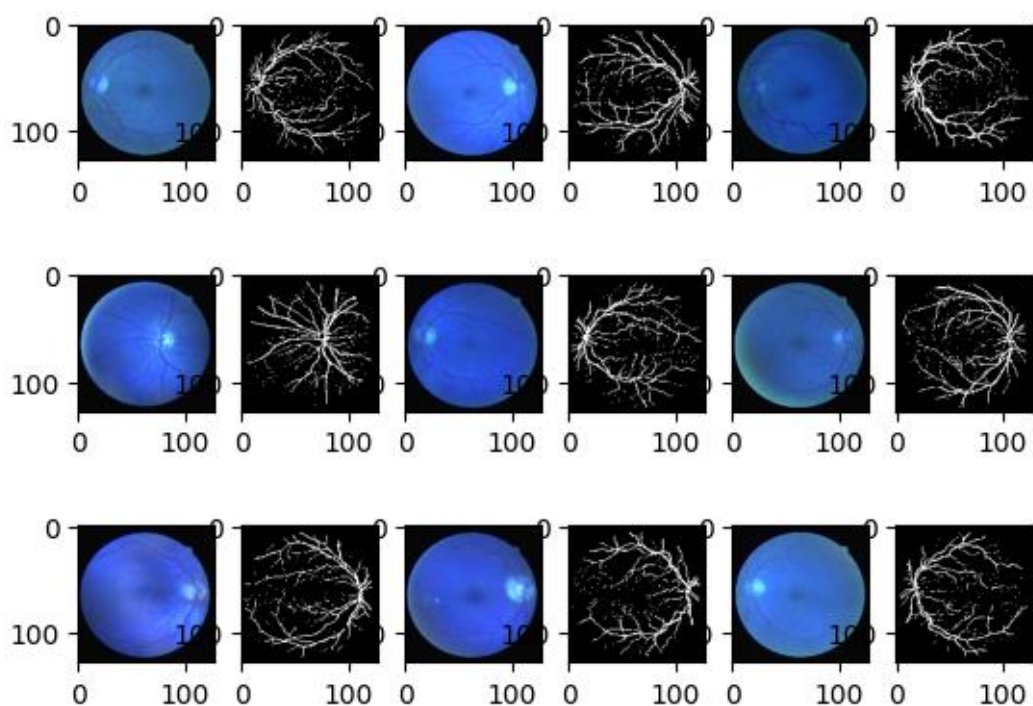
Below are examples of plots and values obtained for a U-net model 4 layers with a learning rate of 0.001 (other screenshots can be find in the "screenshot" folder):



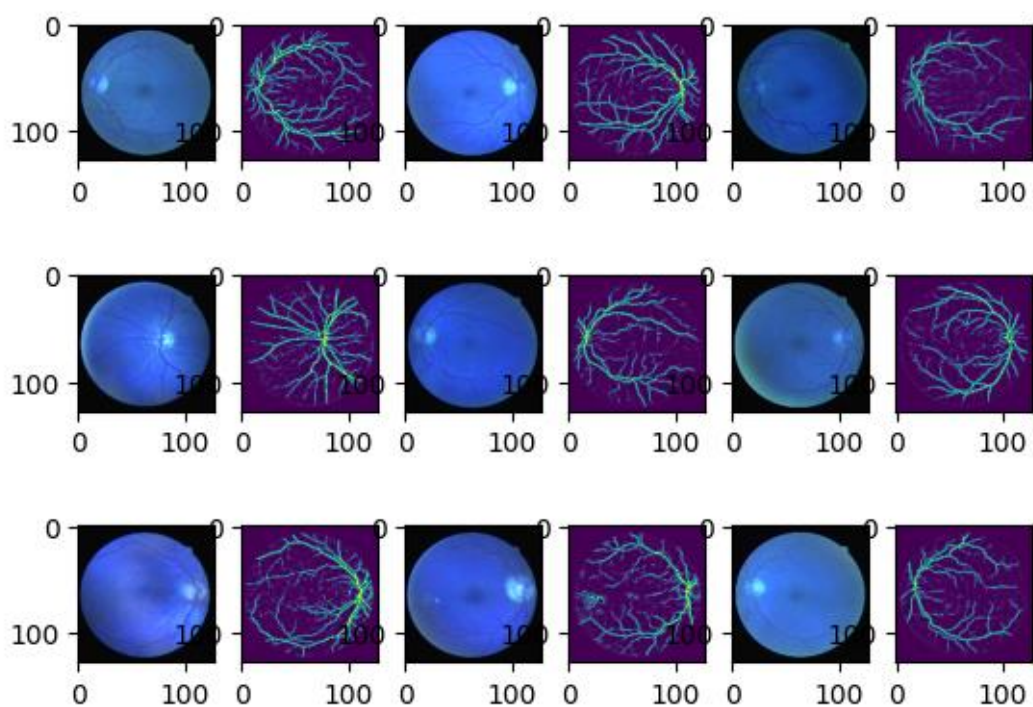
Loss: 0.13794897496700287  
Accuracy: 0.950488269329071  
BinaryIoU: 0.6979999542236328  
Dice: 0.6183845345755317

Finally, below are the results for the 9 first images of the test set:

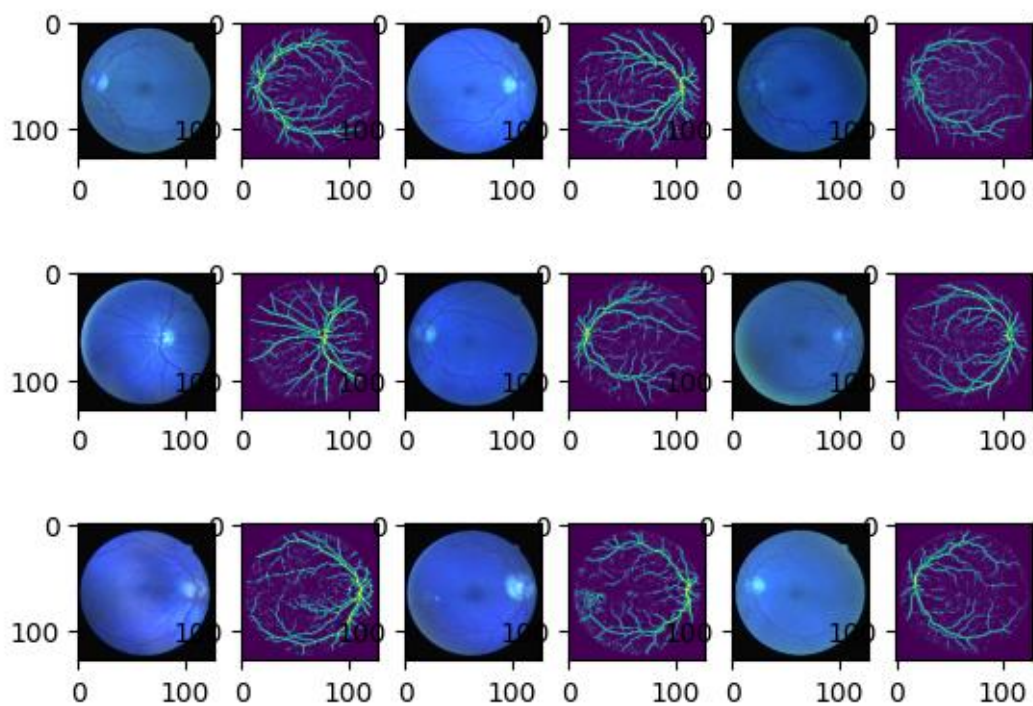
Images | Masks



2 LAYERS



3 LAYERS



4 LAYERS

