Segmenting Retinal Blood Vessels with Deep Neural Networks

1

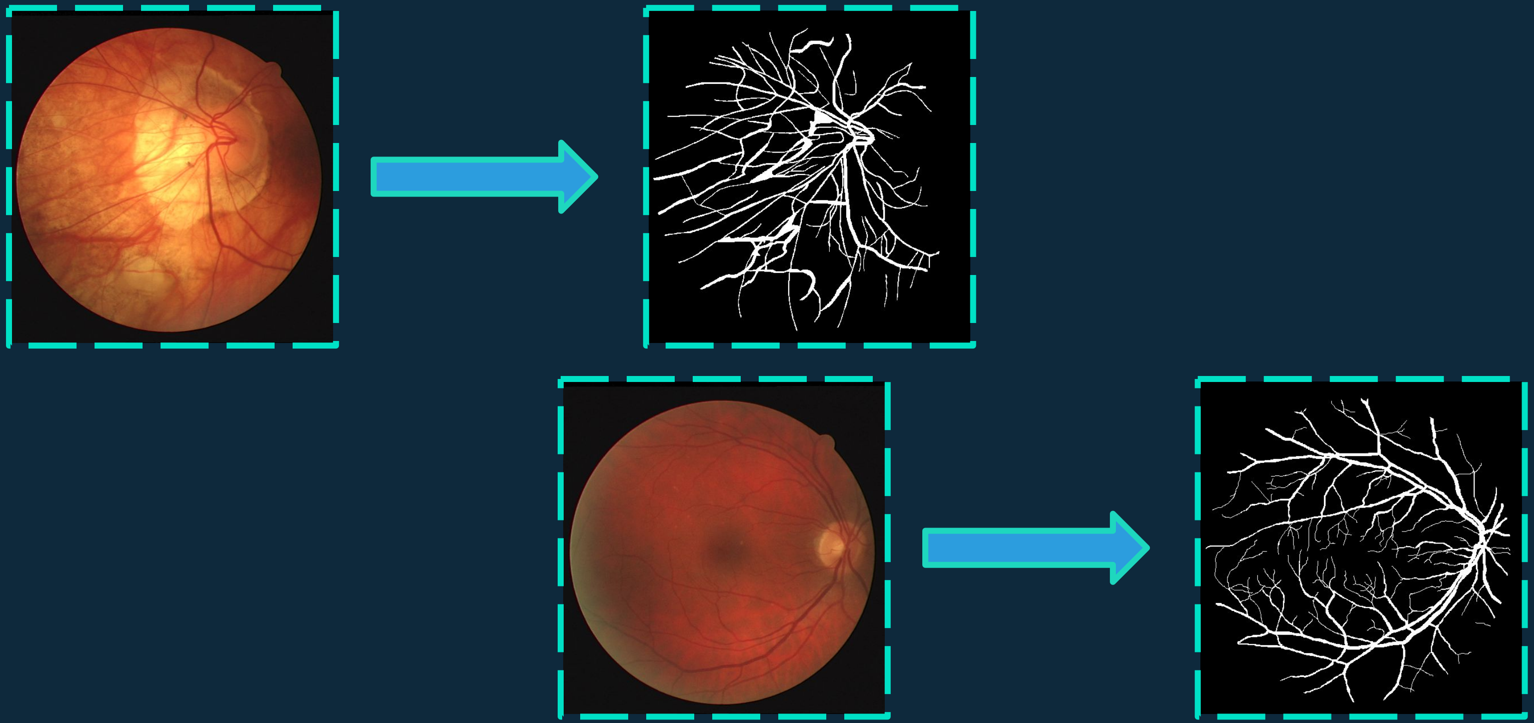
{ Problem statement }

What are trying to do? And why?

Why?

- The condition of the vascular network of human eye is an important diagnostic factor.
- Segmentation is a nontrivial task due to variable size of vessels, low contrast, and potential presence of pathologies.
- Deep Networks are known to perform well in computer vision, but are hard to train on small datasets.
- They are usually inefficient both in terms of training and inference time.

What do we want to achieve?



What data do we have?

DRIVE dataset

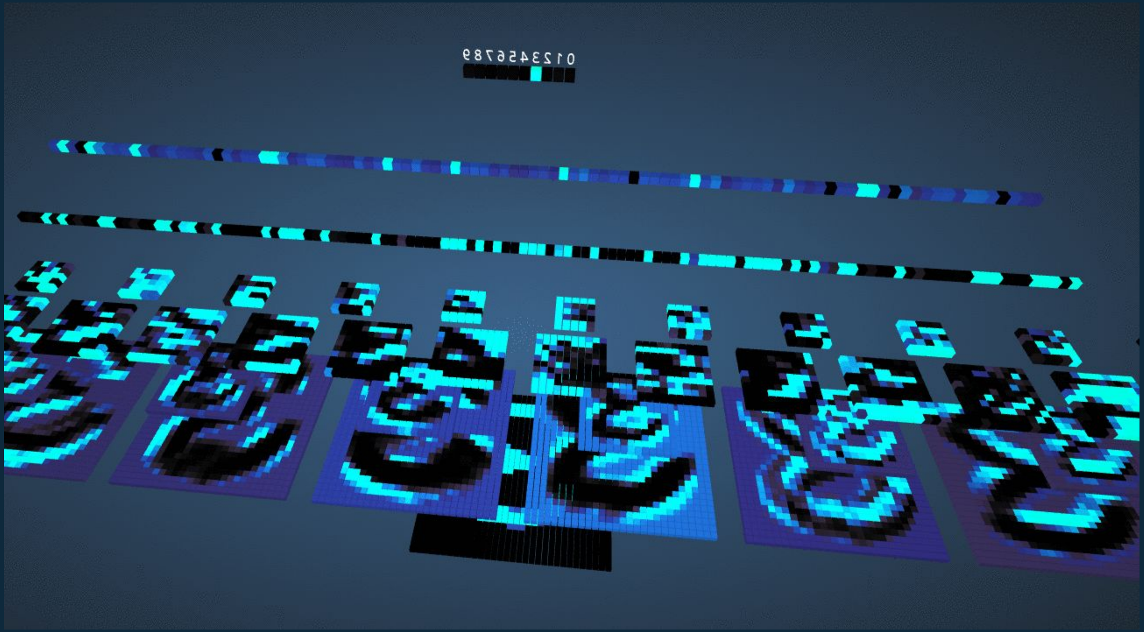
- 20 training images
- 20 test images
- 560 x 560 pixels

HRF dataset

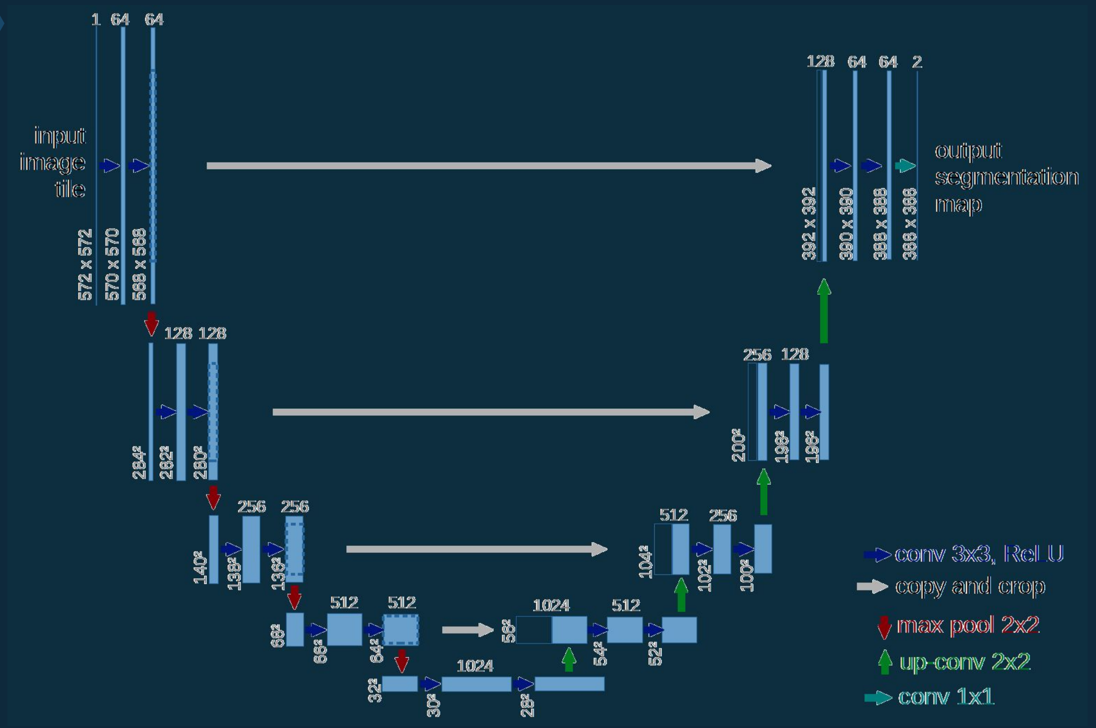
- 15 images (healthy) +
- 15 images (diabetic retinopathy) +
- 15 images (glaucomatous)
- 3504 x 2336 pixels
- 6 random images chosen as a test set

What Methods do we use?

convnets



U-Net



1 64 64
input image tile
572 x 572
570 x 570
568 x 568
128 128
256 256
512 512
1024
512
256
128
128 64 64 2
output segmentation map
392 x 392
390 x 390
388 x 388
388 x 388
conv 3x3, ReLU
copy and crop
max pool 2x2
up-conv 2x2
conv 1x1

2

{ Preprocessing }

How to handle the data?

Overview

Channel-Wise normalization

Compute dataset-wide maen and std for each channel, and then use that to normalize.

Extracting random crops

For each image, random crops of size 64x64 or 128x128 are extracted. We generate between 150 and 500 random patches per image.

Random Rotations

Each image is cloned 3 to 6 times, and each copy is randomly rotated. Rotation angle is chosen between 0 and 270 deg.

Patch-wise normalization

For each patch, and for each channel, subtract its mean and divide by its std

Conversion to grayscale

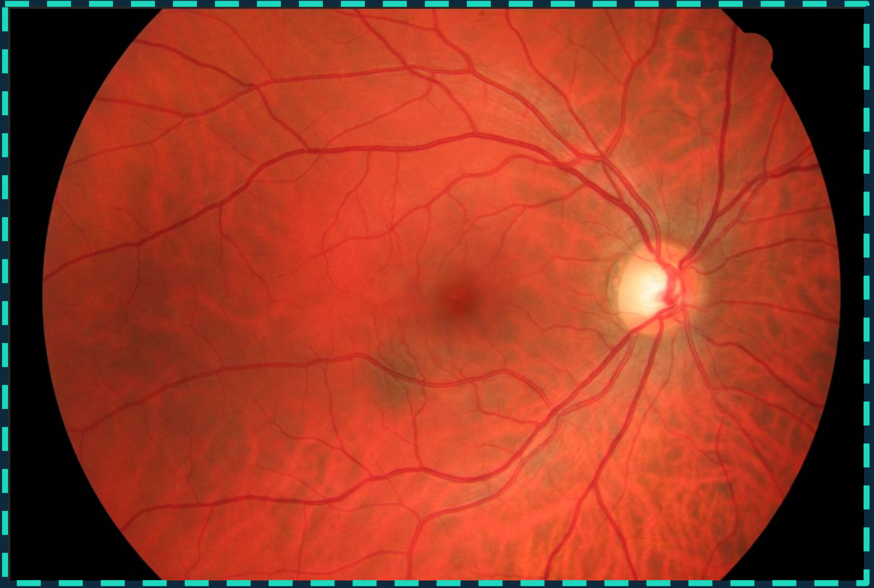
Convert RGB image to grayscale.

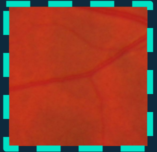
ZCA - whitening

Decorrelates pixels in the image.

Generating random patches

- Essential to obtain enough data points
- Acts as data augmentation
- Combined with random rotations

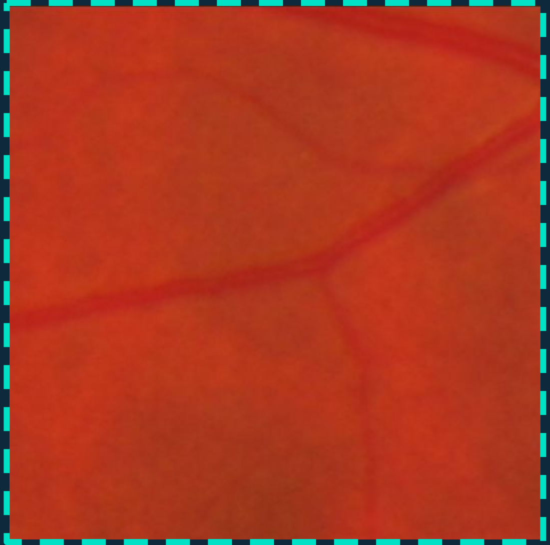


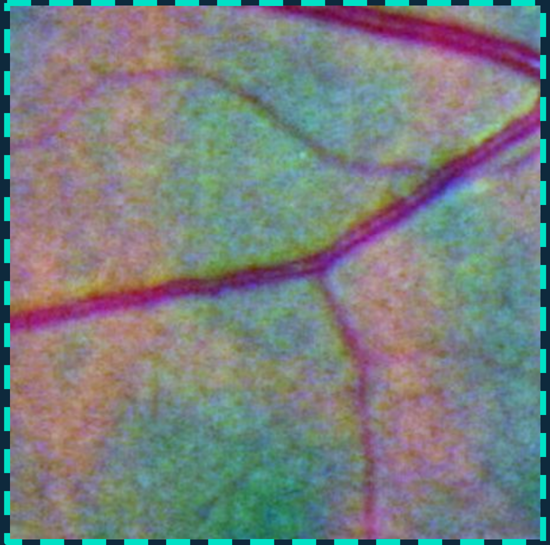


x ~300

Patch-wise normalization

- Proved to be extremely helpful
- Easy to compute
- In the end the only color transformation we used

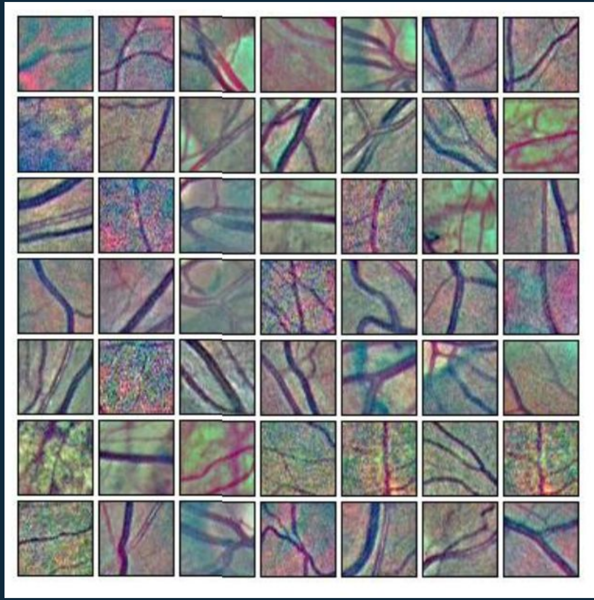




Patch-wise normalization (code)

```python
def ChannelWiseStd(with_mean=True, with_std=True):
    def f(images, masks):
        for i, img in enumerate(tqdm(images, desc="{:<10}".format("Std"))):
            mu    = np.mean(img, axis=(0, 1), keepdims=True)
            sigma = np.std(img, axis=(0, 1), keepdims=True) + 1e-6

            if with_mean:
                img = img - mu
            if with_std:
                img = img / (sigma+1e-6)

            images[i] = img

        return images, masks
    return f
```

ZCA whitening



- Decorelates the features
- Normalizes variance of each feature
- Requires computing SVD -- we couldn't use it due to memory requirements :(
- Others report great results -- prehaps we should try to adopt it after all?

3

{ Neural Network Details }

How exactly our net looks like? How is it trained?

Overview

Batch-Norm

Deals with covariate shift. Greatly speeds up training.

Lots of conv layers

We use several different building blocks

Dropout

Standard way to combat overfitting

Up-scaling with convolution.

Each up-scaling operation if followed by a 1x1 convolution.

U-Net design choices

Lot's of those...

Adam

Because we love adaptive learning rate.

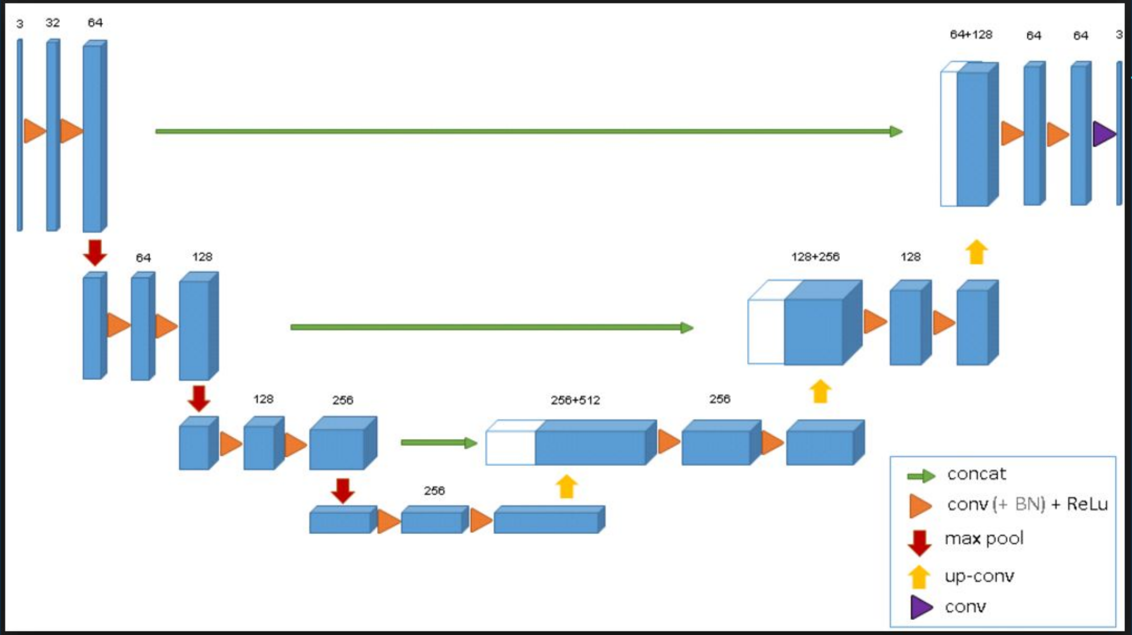
ConvNets

Cool demos:

http://cs231n.github.io/convolutional-networks/
http://scs.ryerson.ca/~aharley/vis/conv/

Design Choices

- 3 levels instead of 4: we use smaller crops
- Upsampling with 1x1 conv to preserve the sh
- Different number of channels
- Dropout only in DoubleConv



Adam

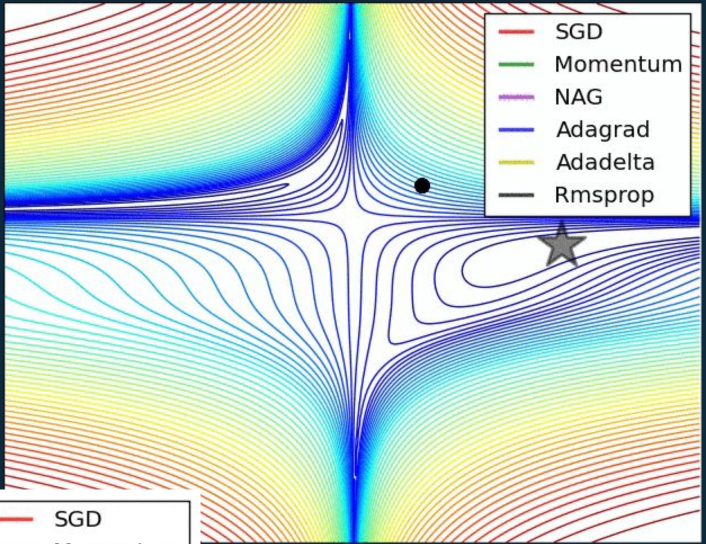
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t .$$

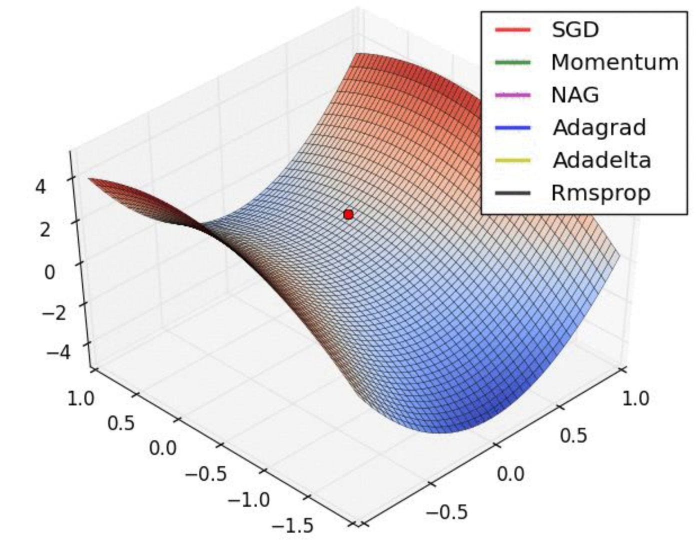
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 .$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} .$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} .$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t .$$





Batch Norm

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

4

{ Results }

How does it work?

Overview

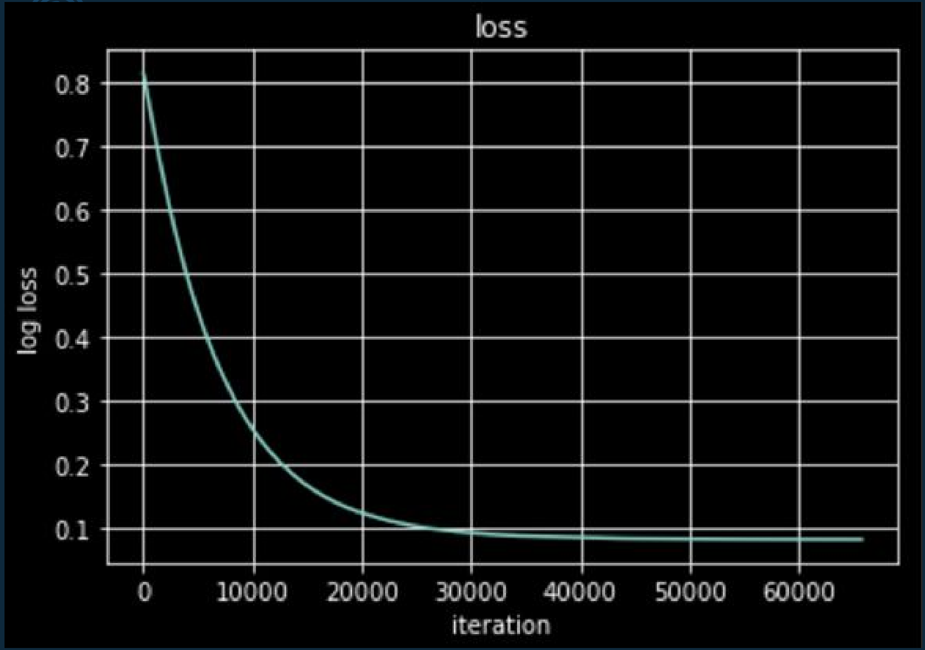
How did we perform?
Can we compare to anyone?

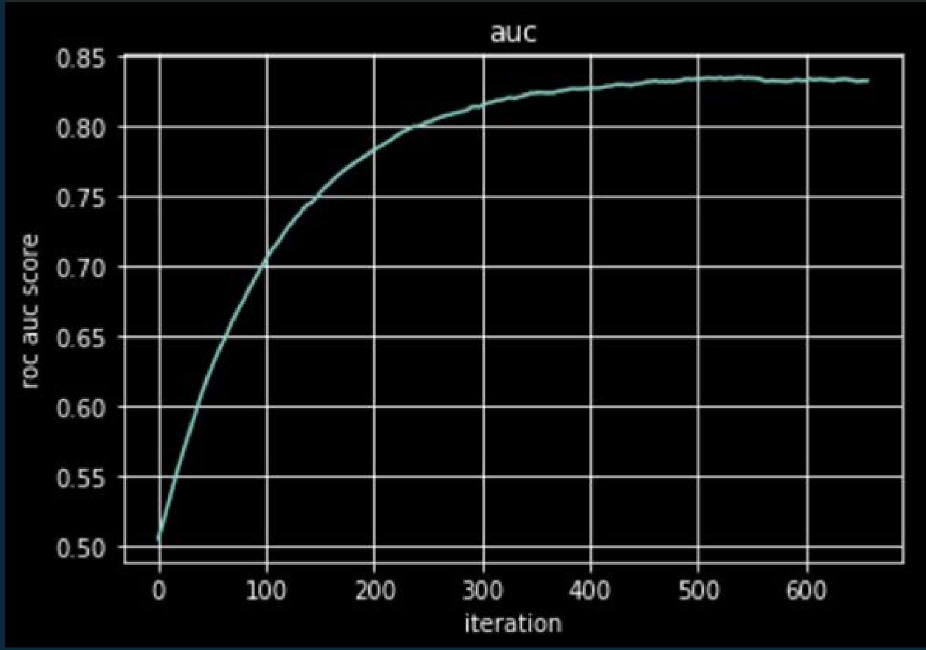
Evaluation measures
How do we measure network's performance?

Prediction on fullsize images
We train on random patches. How do we predict on full images?

Training progress
How to visualize learning?

Training progress (smoothed with IIR 1st order filter)





Performance measures

AUC (ROC)

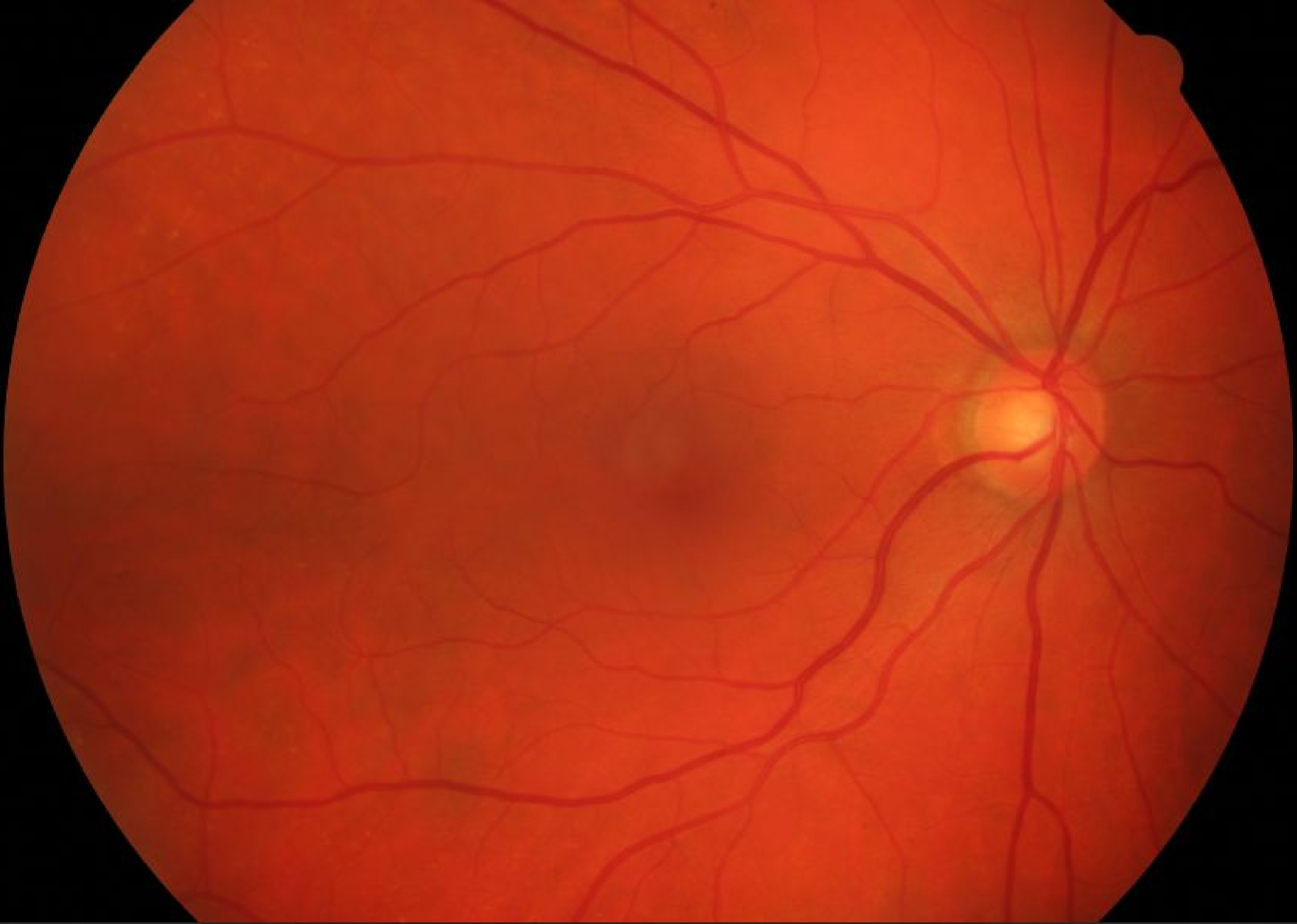
- Robust to class imbalance
- Expensive to compute
- Measured as area under a curve, where we
  Plot TPR vs FPR at various thresholds

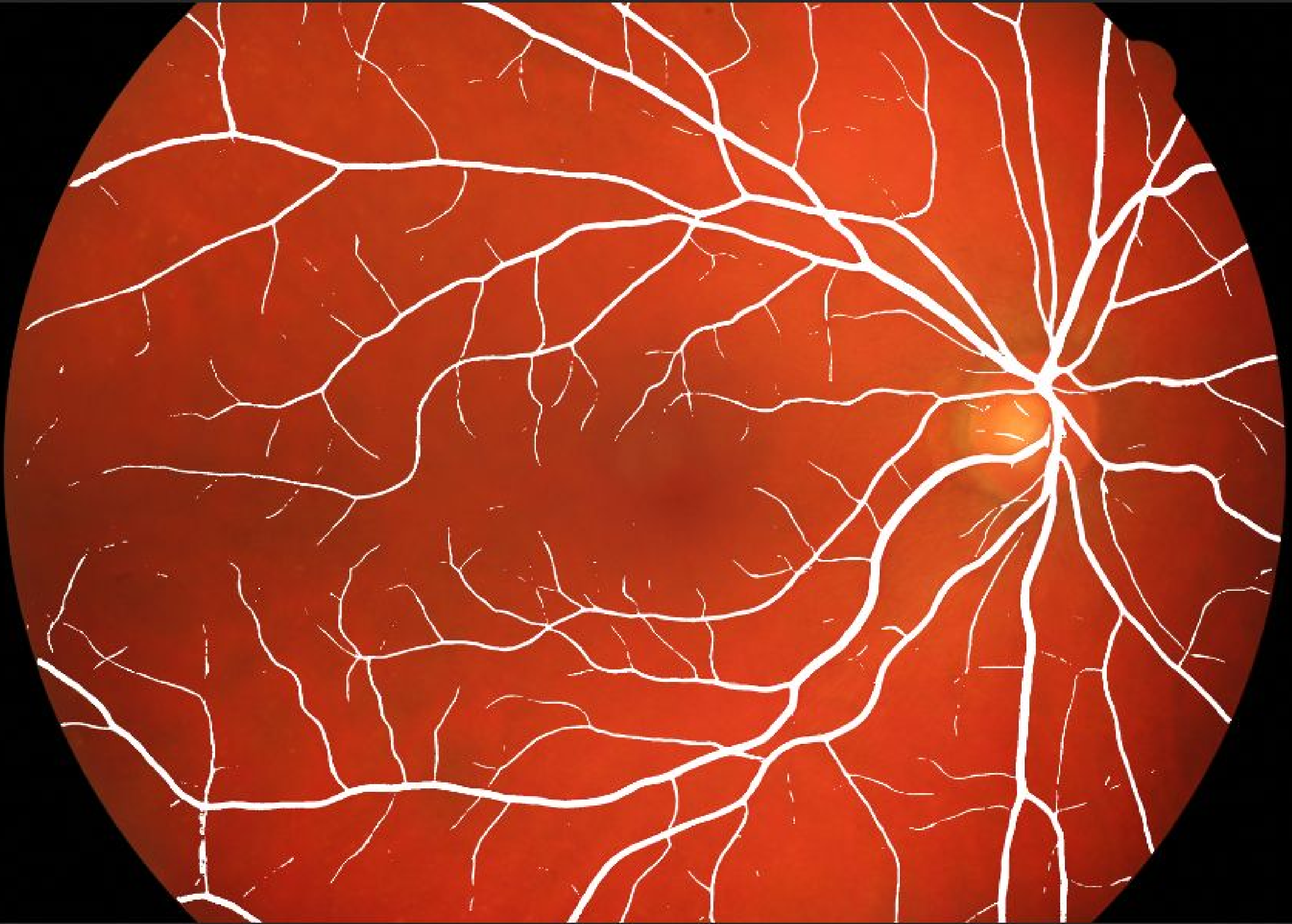
Accuracy

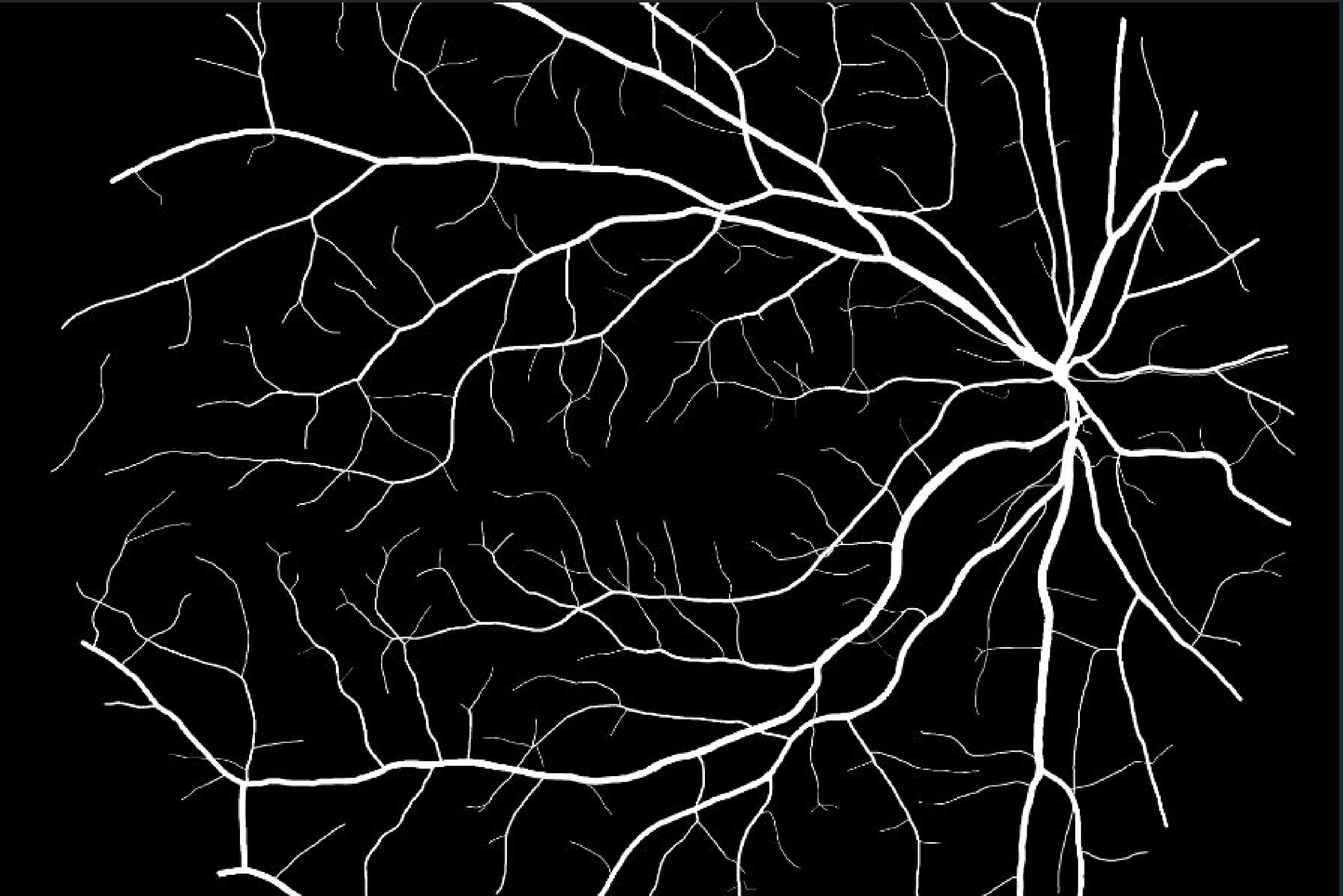
- Very intuitive
- Sensitive to class imbalance

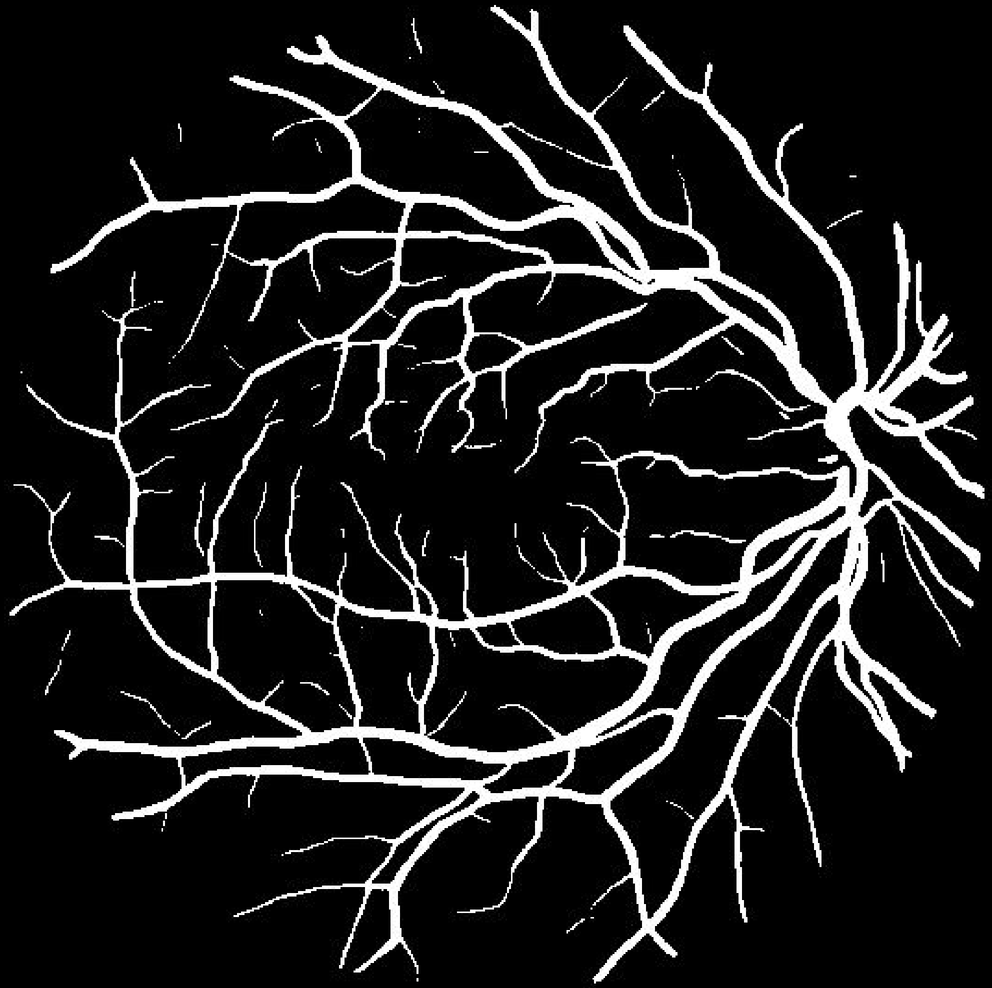
Log Loss

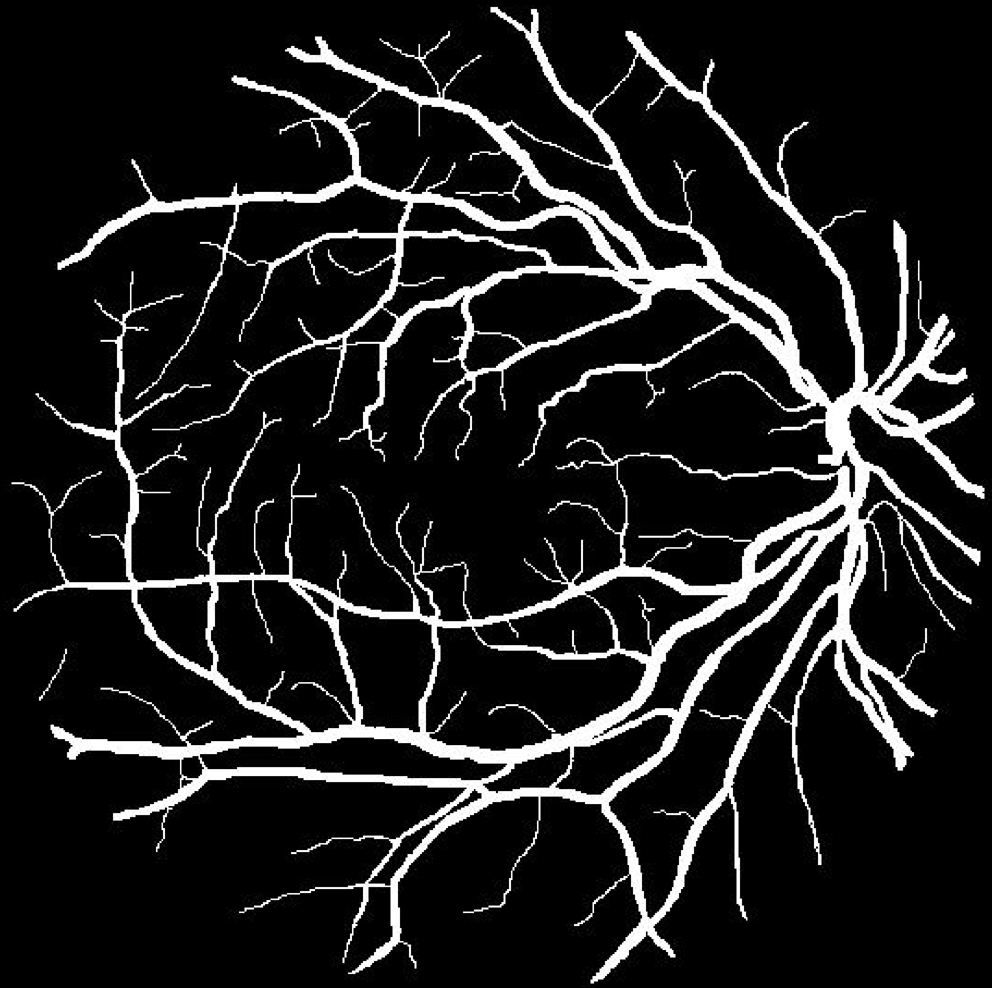
- This is what we directly optimize
- Not really intuitive (KL divergence)









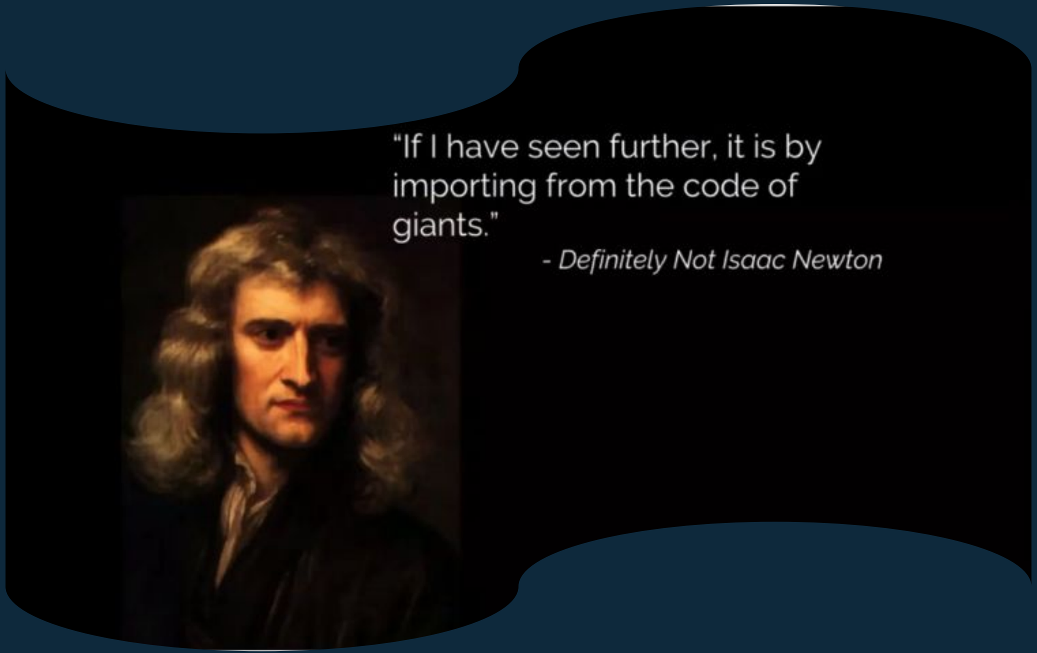


Results

LOSS \ DSET	DRIVE	HRF
Log Loss	0.075	0.084
Accuracy	0.971	0.966
AUCROC	0.911	0.828

5

{ Tools we used }



Overview

Visdom

Facebook's lightweight alternative to Tensorboard.

Supports plotly graphs and video exporting.

Used to track experiments

PyTorch

New deep learning library straight from Facebook AI Research.

Numpy-like, with strong GPU acceleration.

Provides Autograd engine.

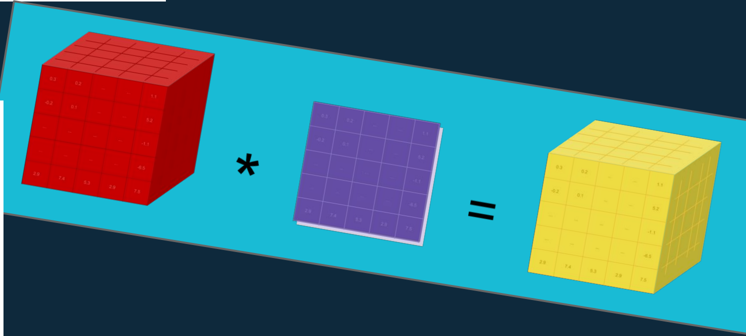
NumFocus Stack

Jupyter + NumPy

PIL.Image

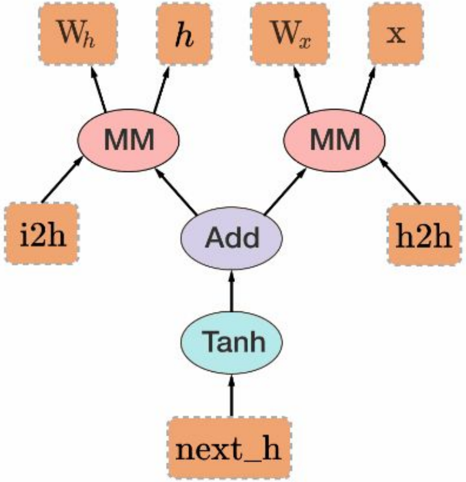
Please, stop using OpenCV when it's not absolutely necessary!

PYTORCH

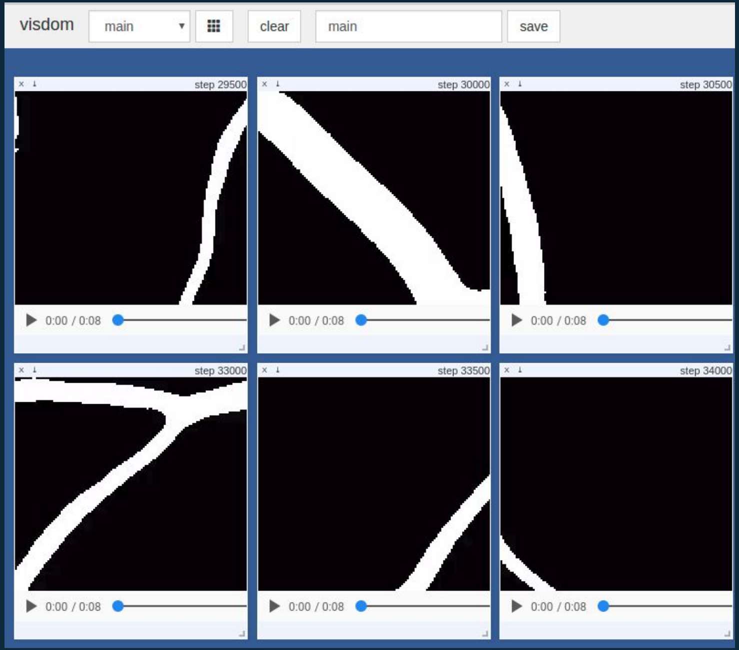


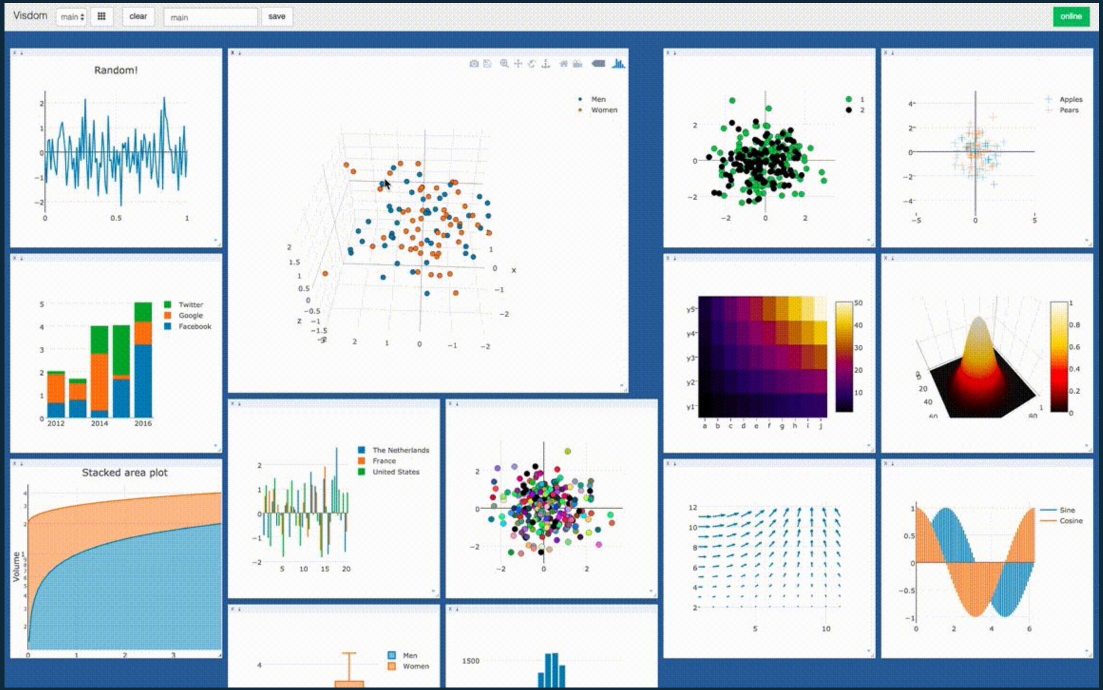
Back-propagation uses the dynamically built graph

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
```



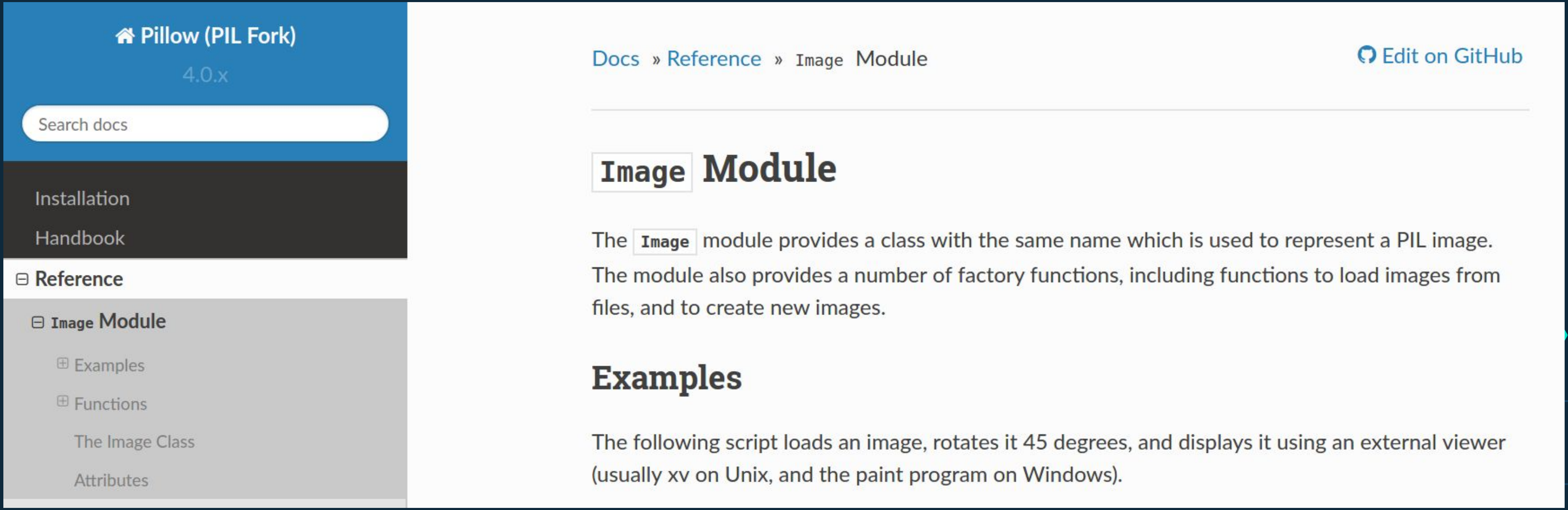
Visdom


visdom
main
clear
main
save
step 29500
step 30000
step 30500
step 33000
step 33500
step 34000
0:00 / 0:08


Visdom
main
clear
main
save
online
Random!
Men
Women
Twitter
Google
Facebook
Apples
Pears
Stacked area plot
Volume
The Netherlands
France
United States
Sine
Cosine

Pillow

- Great alternative to OpenCV
- Clean and up-to-date docs
- Many useful primitives for manipulating images
- pip installable!



6

{ Simple baseline}

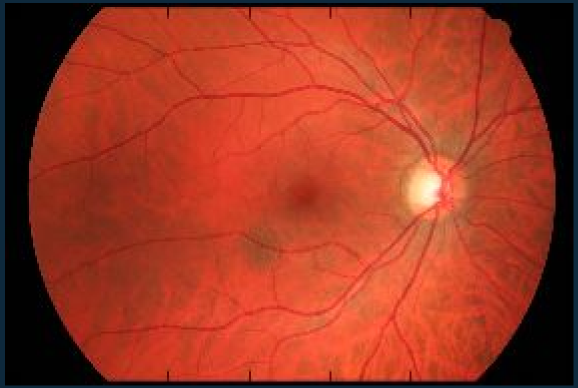
What have we used?

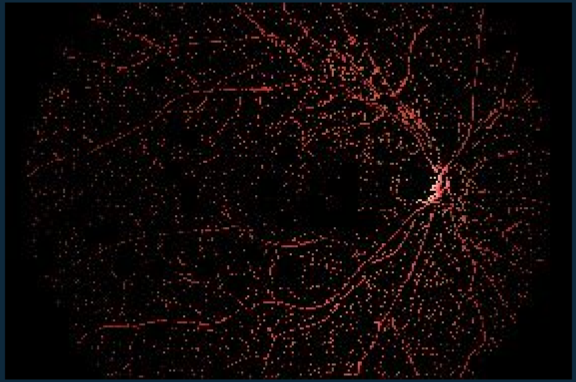
OpenCV

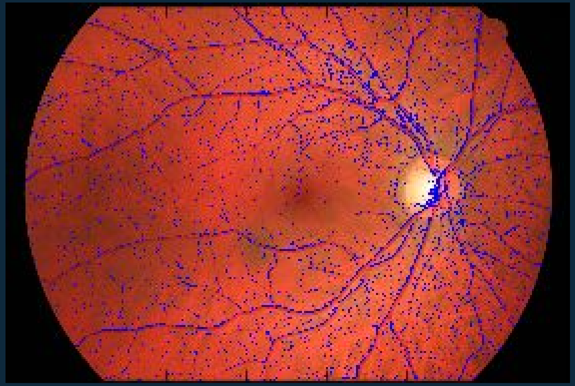
How does it work?

1. Input image preprocessing
2. Algorithm
3. Output image processing
4. Show the result on the input image

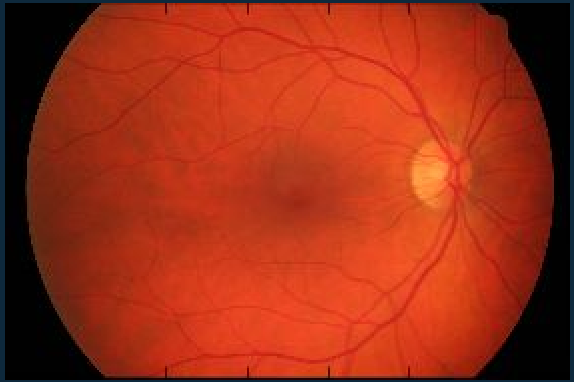
Results

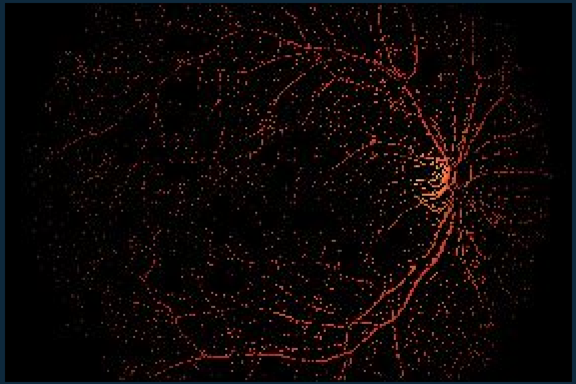


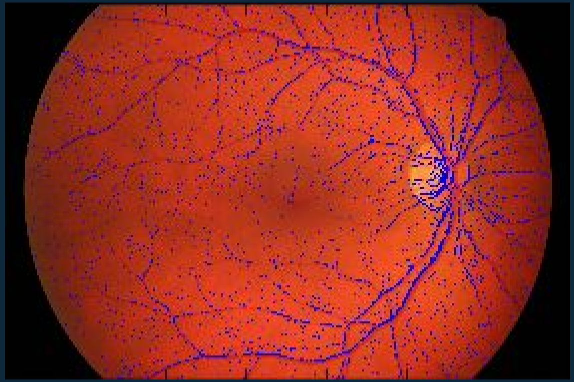




Results







Disadvantages

1. Processing time

2. Ineffective results