

# dno

Maciej A. Czyziewski  
inf136698

Poznan University of Technology  
Poland

## 1 Zastosowany język programowania oraz dodatkowe biblioteki

Python. Lista bibliotek (niektore sluza tylko do debugowania):

- torchvision==0.4.2
- torchsummary==1.5.1
- ipython==7.13.0
- scipy==1.3.1
- torch==1.3.1
- Pillow==7.1.0
- numpy==1.17.2
- tqdm==4.36.1
- matplotlib==3.1.1
- scikit\_image==0.15.0
- scikit\_learn==0.22.2

## 2 Opis zastosowanych metod

Uczenie masznowe. Batchboost. Kfold. Augmentacja. Lamb Optimizer. Unet (wlasna wersja 400tys parametrow + batchnorm w bottlenecku). Taktyka tym razem jest prosta. Pokroj obrazek na male kwadraciki (32x32px, albo podobne warianty). Naucz sie rozpoznawac maske dla takiego malego obrazka. Nie przejmujemy sie skutecznoscia jak wynosi juz okolo 80%. Losujemy bardzo duzo malych wycinkow i usredniajmy predykcje. Dzieki temu powinno to jakos wygladac. Aby dzialaly wieksze wejscia niz 32x32 polecam zwiekszyc ilosc warst sieci. Kfold-y pomogaja przy overfittingu (bardziej zdiagnozowac raka niz go wyleczyc). Dlatego zastosowalem technike batchboost - efektem jest trening ktory zawsze ma wyzszy loss niz walidacja, oraz process stabilizuje sie bedac bardziej niezaleznym od parametrow.

### 2.1 Przetwarzanie obrazów

#### 2.1.1 Poszczególne kroki przetwarzania obrazu (w tym zastosowane filtry)

Kod jest bardzo czytelny - colab.py. Ponizej najwazniejsze wycinki. **Preprocessing**:

```
1 @staticmethod
2     def block(img):
3         # FIXME: grid searchowac ten fragment?
4         img = exposure.equalize_adapthist(img)
```

---

```

5     img = exposure.adjust_gamma(img)
6     img = unsharp_mask(img, radius=3, amount=2)
7     img = ndimage.uniform_filter(img, size=2)
8     return (img * 255).astype(np.uint8)

```

---

## Normalizacja:

---

```

1 @staticmethod
2 def normalize(imgs):
3     _imgs = np.empty(imgs.shape)
4     _imgs = (imgs - np.mean(imgs)) / np.std(imgs)
5     for i in range(imgs.shape[0]):
6         _imgs[i] = (
7             (_imgs[i] - np.min(_imgs[i])) /
8             (np.max(_imgs[i]) - np.min(_imgs[i])))
9         ) * 255
10    return _imgs

```

---

### 2.1.2 Krótkie uzasadnienie zastosowanego rozwiązania

Najważniejsze w tym etapie jest aby wszystkie przykłady z datasetu miały takie same rozkłady jasności pikseli - tak aby człowiek w miarę łatwo potrafił zaznaczyć naczynia krwionosne. Jeśli człowiek potrafi to maszyna też.

## 2.2 Uczenie maszynowe

### 2.2.1 Przygotowanie danych - wyznaczanie wycinków obrazu, ekstrakcja cech z wycinków

Używam sieci konwolucyjnej. Zastosowałem autorski wariant sieci Unet. Sieć sama uczy się enkodowac obrazek a później dekodowac do maski. W środku znajdują się bottleneck który możemy traktować jak wyestrachowane cechy obrazka.

### 2.2.2 Wstępne przetwarzanie zbioru uczącego

Wycięcie małych obrazków (testowałem 3 wersje dla kwadratu o boku: 32px, 48px, 64px) oraz BatchBoost-a jako regularyzacje.

### 2.2.3 Zastosowane metody uczenia maszynowego wraz z informacją o przyjętych parametrach

Opisane w sekcji "Opis zastosowanych metod". Kod jest samo wyjaśniający, a zastosowane podejścia klasyczne. Nizej architektura detektora:

---

Layer (type)	Output Shape	Param #

3	=====		
4	Conv2d-1	[ -1, 64, 64, 64]	640
5	ReLU-2	[ -1, 64, 64, 64]	0
6	Conv2d-3	[ -1, 64, 32, 32]	3,136
7	MaxPool2d-4	[ -1, 64, 16, 16]	0
8	MaxPool2d-5	[ -1, 64, 16, 16]	0
9	Conv2d-6	[ -1, 64, 16, 16]	36,864
10	Conv2d-7	[ -1, 64, 16, 16]	36,864
11	BatchNorm2d-8	[ -1, 64, 16, 16]	128
12	BatchNorm2d-9	[ -1, 64, 16, 16]	128
13	ReLU-10	[ -1, 64, 16, 16]	0
14	ReLU-11	[ -1, 64, 16, 16]	0
15	Conv2d-12	[ -1, 64, 16, 16]	36,864
16	Conv2d-13	[ -1, 64, 16, 16]	36,864
17	BatchNorm2d-14	[ -1, 64, 16, 16]	128
18	BatchNorm2d-15	[ -1, 64, 16, 16]	128
19	ReLU-16	[ -1, 64, 16, 16]	0
20	ReLU-17	[ -1, 64, 16, 16]	0
21	BasicBlock-18	[ -1, 64, 16, 16]	0
22	BasicBlock-19	[ -1, 64, 16, 16]	0
23	Conv2d-20	[ -1, 64, 16, 16]	36,864
24	Conv2d-21	[ -1, 64, 16, 16]	36,864
25	BatchNorm2d-22	[ -1, 64, 16, 16]	128
26	BatchNorm2d-23	[ -1, 64, 16, 16]	128
27	ReLU-24	[ -1, 64, 16, 16]	0
28	ReLU-25	[ -1, 64, 16, 16]	0
29	Conv2d-26	[ -1, 64, 16, 16]	36,864
30	Conv2d-27	[ -1, 64, 16, 16]	36,864
31	BatchNorm2d-28	[ -1, 64, 16, 16]	128
32	BatchNorm2d-29	[ -1, 64, 16, 16]	128
33	ReLU-30	[ -1, 64, 16, 16]	0
34	ReLU-31	[ -1, 64, 16, 16]	0
35	BasicBlock-32	[ -1, 64, 16, 16]	0
36	BasicBlock-33	[ -1, 64, 16, 16]	0
37	Upsample-34	[ -1, 64, 32, 32]	0
38	Conv2d-35	[ -1, 64, 32, 32]	4,160
39	ReLU-36	[ -1, 64, 32, 32]	0
40	BatchNorm2d-37	[ -1, 64, 32, 32]	128
41	Upsample-38	[ -1, 128, 64, 64]	0
42	Conv2d-39	[ -1, 64, 64, 64]	110,656
43	ReLU-40	[ -1, 64, 64, 64]	0
44	Conv2d-41	[ -1, 1, 64, 64]	64
45	=====		
46	Total params:	414,720	
47	Trainable params:	414,720	
48	Non-trainable params:	0	
49	-----		
50	Input size (MB):	0.02	

---

```

51 Forward/backward pass size (MB) : 18.28
52 Params size (MB) : 1.58
53 Estimated Total Size (MB) : 19.88
54 -----

```

---

## 2.3 Wyniki wstępnej oceny zbudowanego klasyfikatora (testy hold-out lub k-fold cross validation)

Podam przykład dla detektora w wersji kwadratu o boku 64px (ostatni epoch).

---

```

1 train: loss: 0.428751 (nie ma overfitowania przez batchboost-a)
2     val: loss: 0.205344
3     test: loss: 0.219781
4 best val loss: 0.193103

```

---

Jako bledu a zarazem metryki uzylem tzw. “Dice Error/Loss”. Czyli tak naprawde ile pixeli sie nie zgadza w przekroju masek (znormalizowana wartosc). Oczywiscie mozna optymalizowac inną funkcje kosztu wzgledem metryki. Dlatego dodalem jeszcze z mala wagą “binary\_cross\_entropy\_with\_logits” (beda ostre krawedzie).

---

```

1 def dice_loss_1(a, b, smooth=1.0):
2     intersection = (a * b).sum()
3     return 1 - ((2.0 * intersection + smooth) /
4                 (a.sum() + b.sum() + smooth))
5
6
7 def dice_loss(input, target, smooth=1.0):
8     iflat = input.view(-1)
9     tflat = target.view(-1)
10    return dice_loss_1(iflat, tflat)
11
12
13 def calc_loss(pred, target, bce_weight=0.2):
14     target = target.type_as(pred)
15     bce = F.binary_cross_entropy_with_logits(pred, target)
16
17     pred = torch.sigmoid(pred)
18     dice = dice_loss(pred, target)
19
20    return bce * bce_weight + dice * (1 - bce_weight)

```

---

Wyniki zbioru testowego (cale obrazki) dla roznych detektorow:

Detector Size	1	2	3	4	5
32x32px	0.297	0.319	0.347	0.265	0.302
48x48px	0.279	0.277	0.304	0.230	0.260
64x64px	<b>0.263</b>	<b>0.252</b>	<b>0.275</b>	<b>0.219</b>	<b>0.238</b>

Table 1: Wplyw rozmiaru na jakosc maski, uzyty dice\_loss.

## 2.4 Krótkie uzasadnienie zastosowanego rozwiązania

Intuicja.

### 3 Wizualizacja wyników działania

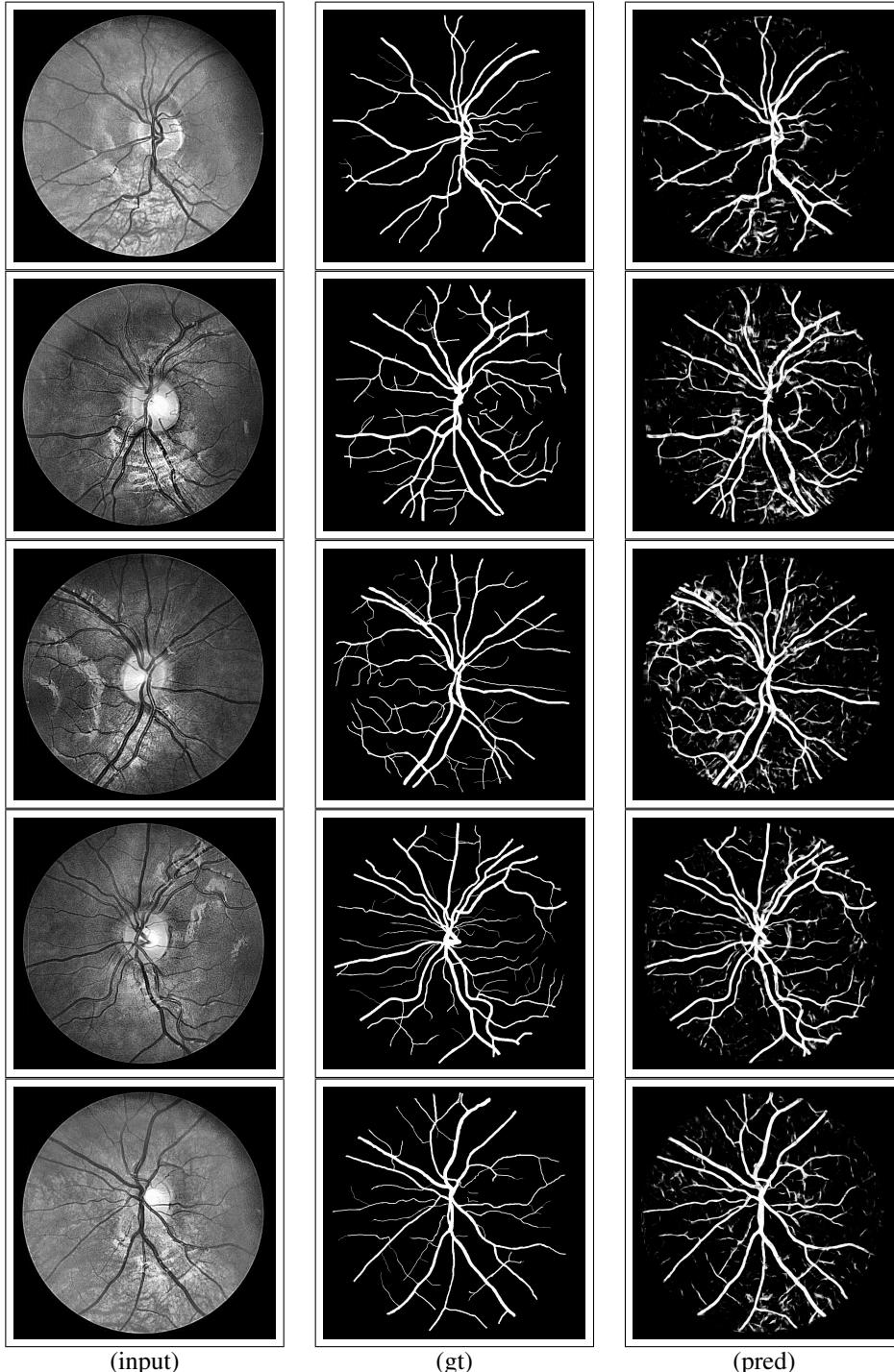


Figure 1: Dla detektora 32x32px.

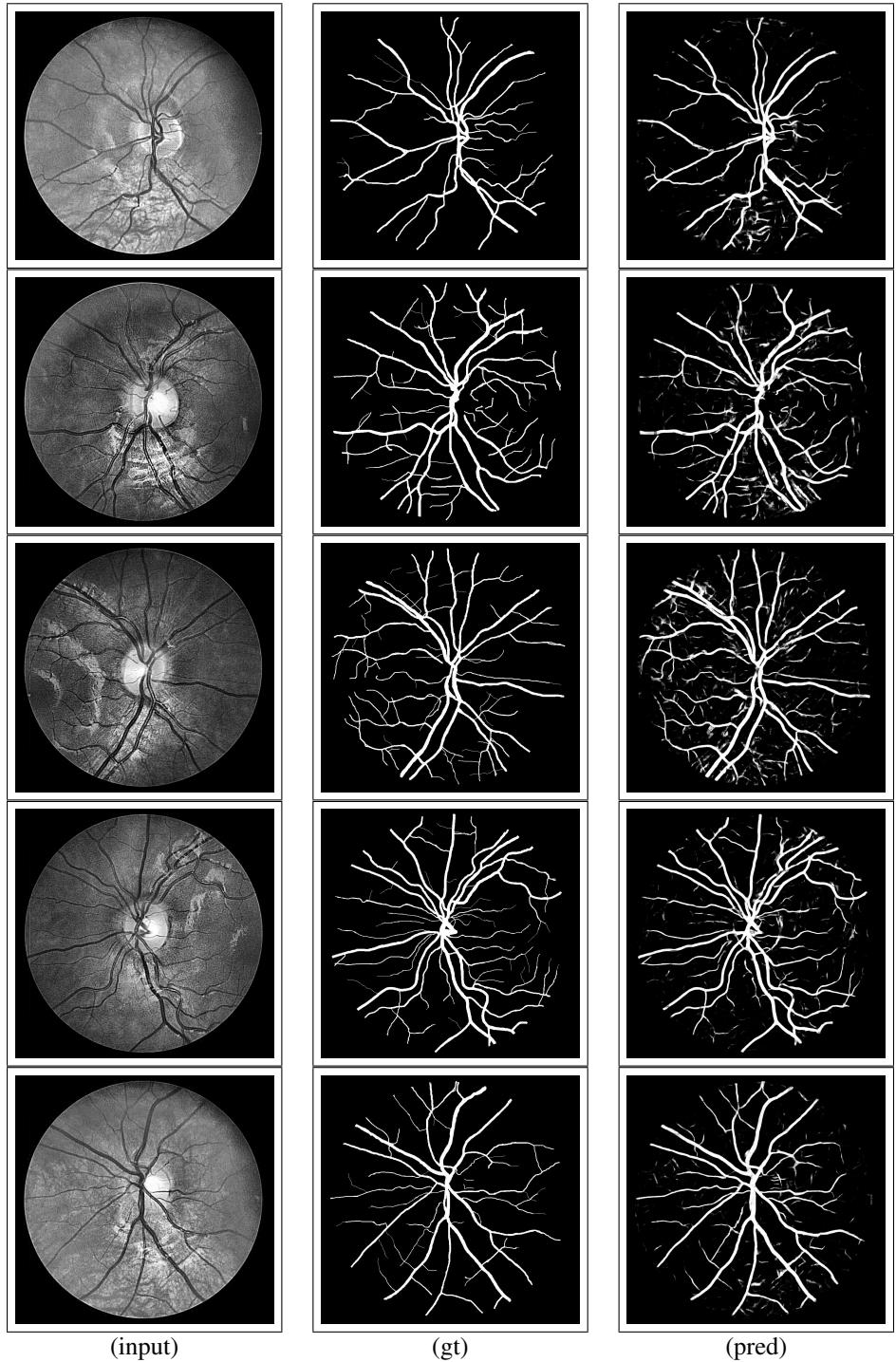


Figure 2: Dla detektora 48x48px.

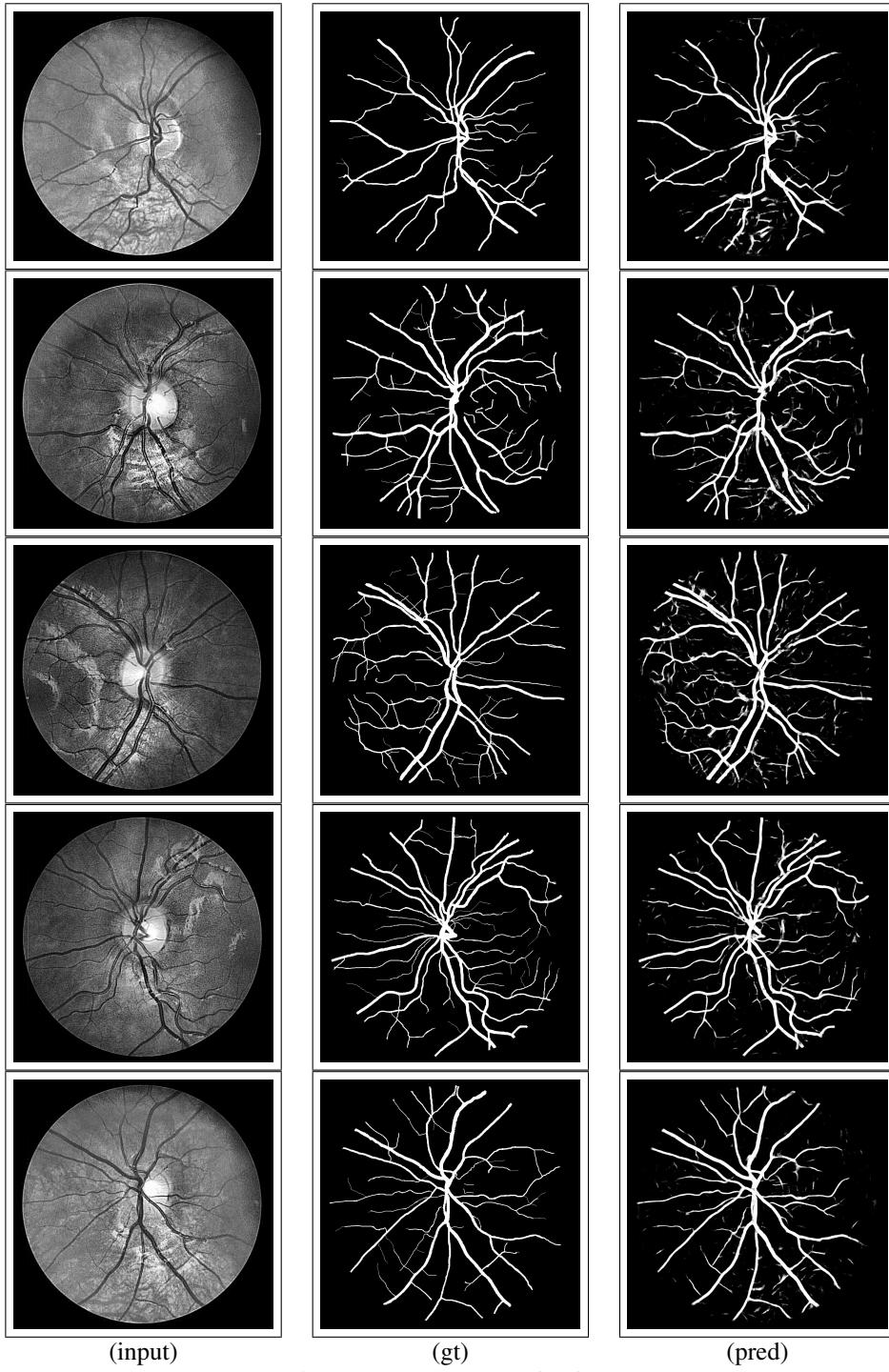


Figure 3: Dla detektora 64x64px.

## 4 Analiza wyników działania

Trzeba tu podkreslic ze detektory byly trenowane mala ilosc epochow a jednak daja calkiem przyzwoite wyniki. Najlepiej dziala dla wersji 64x64px. Przy dobrej augmentacji oraz sprzecie byloby mozliwe wytrenowanie maski 128x128px ktora zapewnie bylaby dobrym komprosimem pomiedzy jakoscia a szybkoscia. Aby poprawic rezultaty moznaby po wygenerowaniu maski, potraktowac ja jeszcze raz jakims innym modelem ktory przyjmowałby 3 wejscia: input, nasza predykcja, wspolrzedne kwadratu wzgledem srodka oka. Taki system zapewne by potrafil usunac szumy wokol naczyn. Aktualanej wersji usuwane sa szumu gdy nalozy sie threshold-a 170 (dla obrazka 0 .. 255). Jednak wtedy przerywa niektore naczynia. Wiec postanowilem zostawic orginalne wyjscia detektorow. Analiza pojedynczych zdjec jest ujeta tabelka w poprzedniej sekcji (z `dice_loss`). Prosze zwrocic uwage ze kazdy przyklad w miejscu nadmiernej ekzpozycji swiatla ma lekkie przebiecie - pojawiaja sie szumy w postaci drobnych plamek/kreseczek (mozna probowac je usunac jakims filtrem, jednak nie przeszakadza to w intrepretacji obrazu).

## 5 Uwage

Aby przetestowac skrypt wystarczy skoplowac go do Google Colab. Instaluje on sam potrzebne pakiety oraz sciaga moje wagi, ktore uzylem w eksperymencie. W innym przypadku wymagany jest ponowny trening.