

Sprawozdanie

7 listopada 2020

Spis treści

1	Wstęp	1
2	Zbieranie danych	1
2.1	Obróbka danych	2
2.2	Augmentacja danych	2
2.2.1	Wycięcie kawałka zdjęcia	4
2.2.2	Pogorszenie jakości zdjęć	4
2.2.3	Lustrzane odbicie zdjęcia	4
2.3	Podział danych	6
3	Wybór modelu	6
3.1	Odrzucone modele	7
4	Trening	8
5	Testy	8
Bibliografia		12

1 Wstęp

Zgodnie z poleceniem spróbowujemy wytrenować sieć rozpoznającą środek twarzy osoby przedstawionej na zdjęciu. Nasz problem podpisuje się pod zadanie znajdowania landmarków twarzy (charakterystycznych punktów w rodzaju oczu, twarzy, ust) dlatego spróbowujemy wykorzystać zbiory danych i zapożyczyć architektury sieci używane przy **Facial landmark detection**.

2 Zbieranie danych

Będziemy bazować na zdjęciach z zbioru [Facial Keypoints \(68\) dataset](#) zawierającego lekko ponad cztery tysiące zdjęć o różnych rozmiarach. Każdemu obrazkowi odpowiada wiersz zawierający 68 współrzędnych odpowiadających kluczowym punktom twarzy osoby przedstawionej na zdjęciu – nas interesuje trzydziesta para odpowiadająca współrzędnym czubka nosa.

Komentarz Na początku chciałem dodać też zdjęcia z innych datasetów ([Biwi Kinect Head Pose Database](#), [Celeb-Faces Attributes \(CelebA\) Dataset](#) – ten pierwszy jest o tyle fajny, że zawiera mniej ”sztywne” przykłady i środek twarzy rzadziej odpowiada środkowi obrazka) ale już dla samych danych z [Facial Keypoints \(68\) dataset](#) mój laptop

Rysunek 1



Rysunek 1: Przykładowe zdjęcie z [Facial Keypoints \(68\) dataset](#)

zaczynał protestować (po augmentacji dostałem około 70 000 zdjęć).

2.1 Obróbka danych

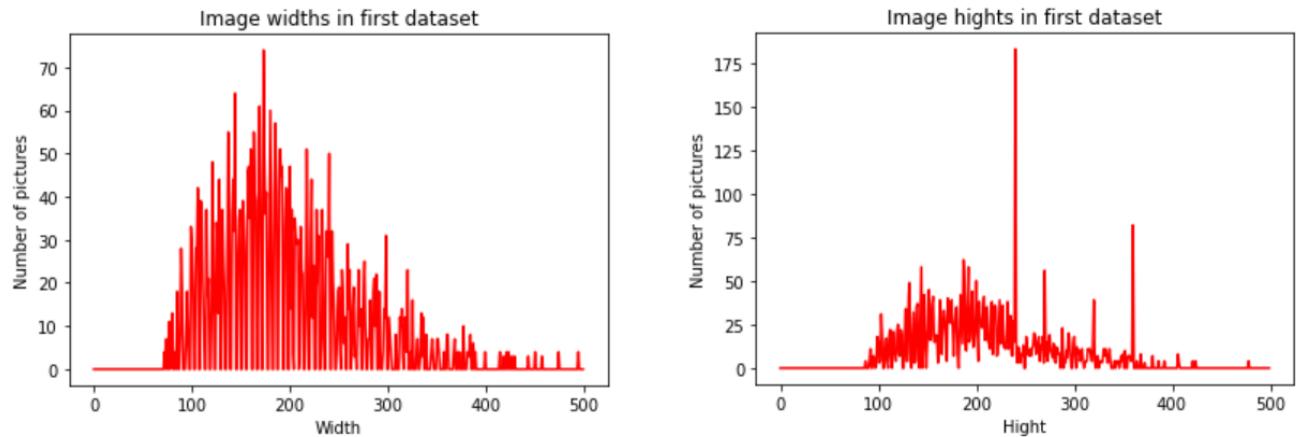
Z reguły sieci neuronowe przyjmują input o ustalonym rozmiarze a wymiary dostępnych zdjęć są rozrzucone w przedziale od ok. 80 do prawie 500. Żeby temu zaradzić zaczniemy od swego rodzaju "ustandardyzowania" – każde zdjęcie będziemy chcieli przekształcić do rozmiaru odpowiadającego wejściu naszej sieci, dokładniej do postaci macierzy o wymiarach [78, 78, 3]. (Podkрадziona przez nas architektura sieci przyjmuje wejście o rozmiarze [39, 39, 3] do których łatwo można przejść od [78, 78, 3]. Dodatkowo wybranie siedemdziesiątki ósemki zamiast większych rozmiarów pozwala nam uniknąć powiększania zdjęć które dość często doprowadza do utraty jakości.) Zróbmy to w trzech krokach:

- 1) Przekształć zdjęcie z RGBA do RGB.
- 2) Przeskaluj zdjęcie z zachowaniem proporcji tak, żeby większy z jego wymiarów wynosił 78.
- 3) Dodaj padding w postaci czarnego paska, u dołu lub z prawej części obrazka, tak, żeby wyrównać jego wymiary do 78×78 . Przy odrobinie szczęścia sieć nauczy się go ignorować.

2.2 Augmentacja danych

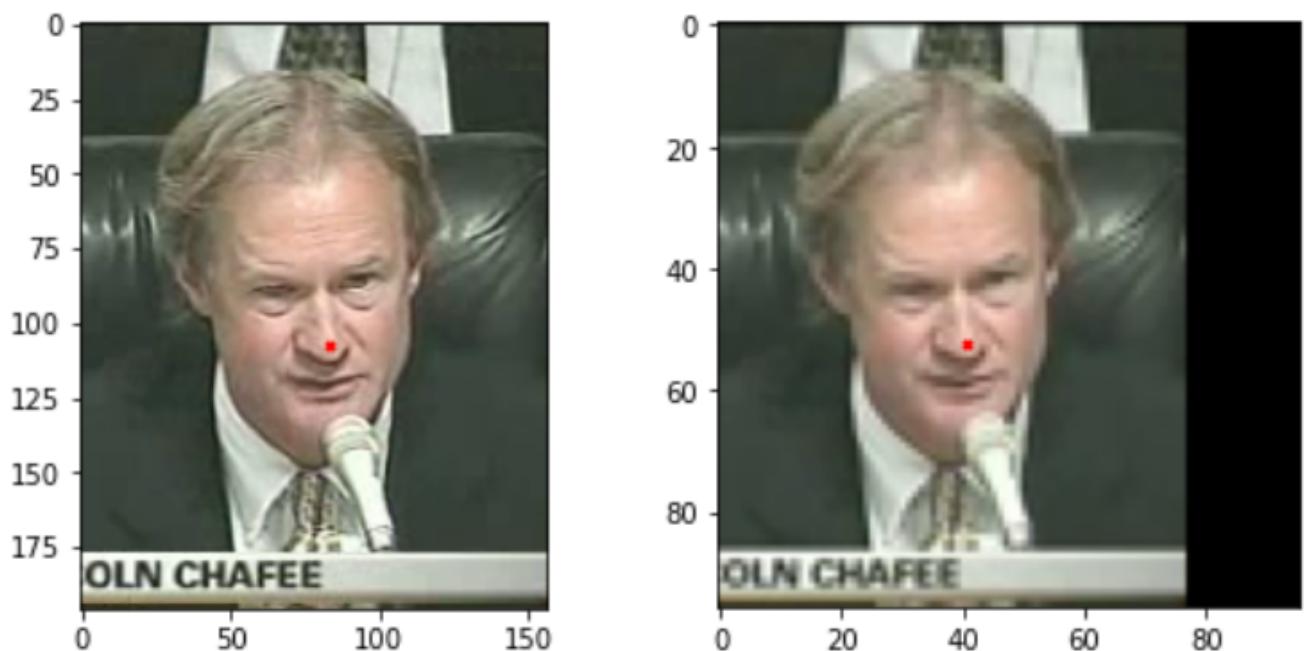
Dużym plusem [Facial Keypoints \(68\) dataset](#) jest to, że zawiera zdjęcia wielu różnych osób. Niestety jeśli chcemy wykorzystać zapożyczone zdjęcia jako dane treningowe rzuca się w oczy kilka wad:

Rysunek 2



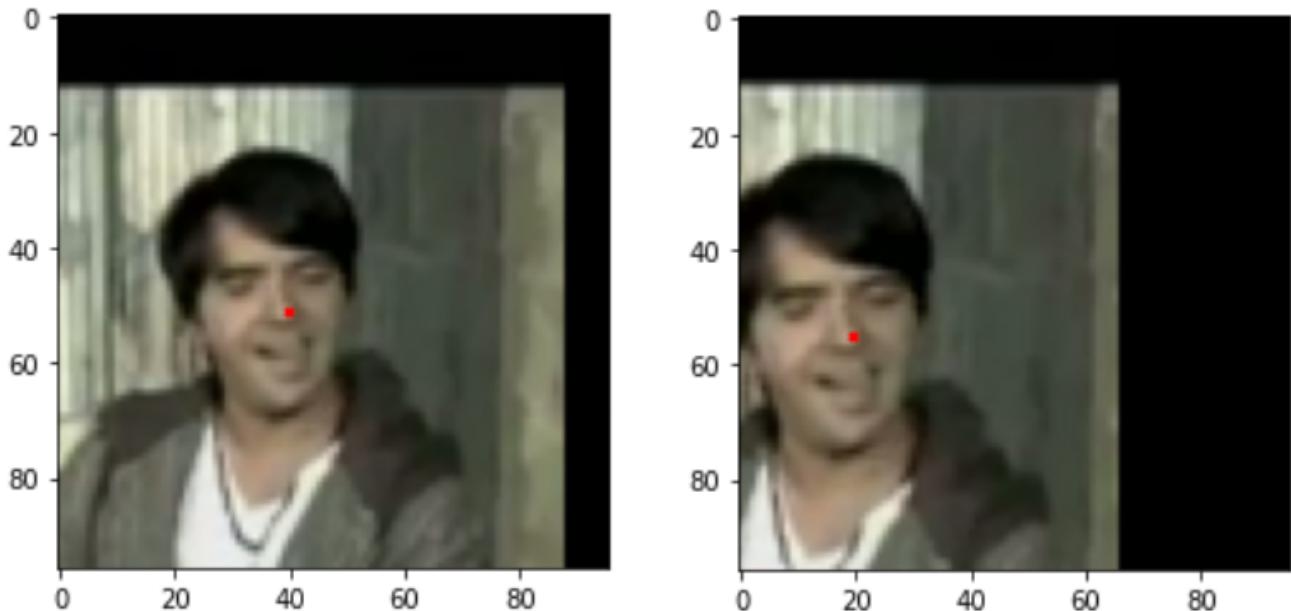
Rysunek 2: Wymiary dostępnych zdjęć.

Rysunek 3



Rysunek 3: Przykład przeskalowania zdjęcia z dodaniem paddingu.

Rysunek 4



Rysunek 4: Przykład wycinania części zdjęcia numer 1

- 1) Po pierwsze i najważniejsze, nosy na zdecydowanej większości zdjęć znajdują się w okolicach środka obrazka, co może ograniczyć trenowaną sieć do zaznaczania punktów o współrzędnych blisko środka.
- 2) Jakość większości dostępnych zdjęć jest dość dobra, przez co wytrenowana sieć może radzić sobie gorzej z rzeczywistymi danymi. (Niestety raczej nie możemy założyć, że użytkownicy będą bardzo się starali przy robieniu zdjęć, żeby tylko ułatwić naszej sieci pracę. :/)

Spróbujemy temu zaradzić, powiększając jednocześnie liczbę dostępnych danych z użyciem trzech drobnych modyfikacji:

2.2.1 Wycięcie kawałka zdjęcia

Najważniejsze dla nas będzie wybranie kawałków oryginalnych zdjęć, przy odrobinie szczęścia pomoże nam to uzyskać zbiór zdjęć w którym nosy sfotografowanych osób znajdują się w punktach o bardziej losowych współrzędnych.

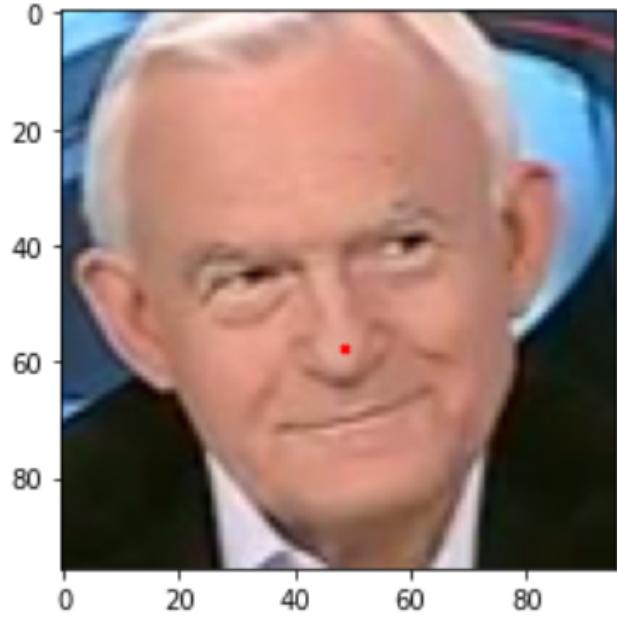
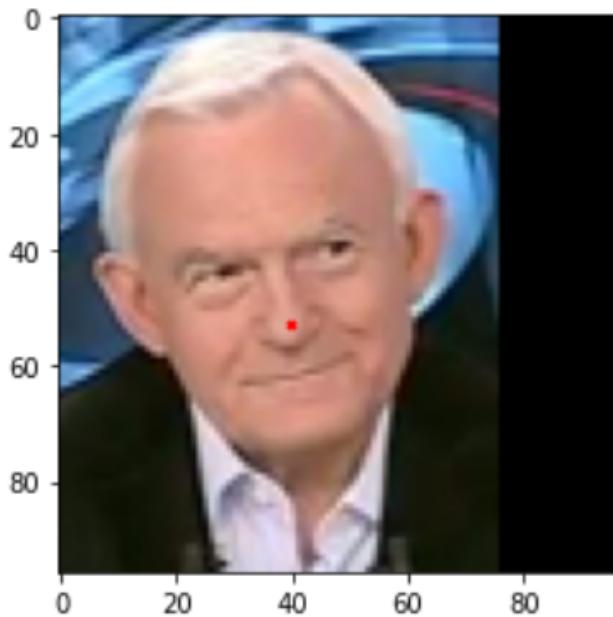
2.2.2 Pogorszenie jakości zdjęć

Żeby poradzić sobie z drugim problemem wprowadzimy szum do wartości RGB, zdjęcia – dokładniej wartość każdej komórki macierzy zmienimy o losową liczbę z przedziału $[-5.0, 5.0]$ pilnując przy tym, żeby wartości nowego zdjęcia dalej mieściły się w przedziale $[0.0, 255.0]$. (Dodatkowo nie zaburzamy komórek odpowiadających czarnemu paskowi dodanemu przy paddingu.)

2.2.3 Lustrzane odbicie zdjęcia

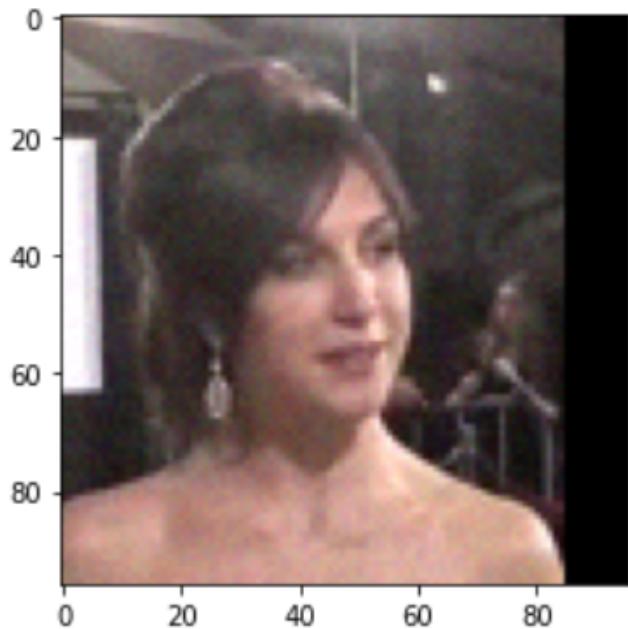
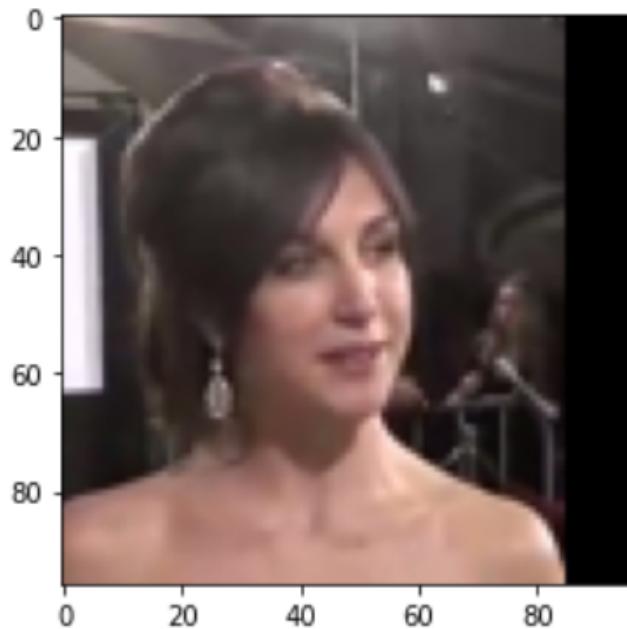
Na koniec dodamy lustrzane odbicie każdego dostępnego zdjęcia, to prosty sposób na podwojenie liczby danych, chociaż tracimy coś nie coś występującej w zbiorze oryginalności.

Rysunek 5



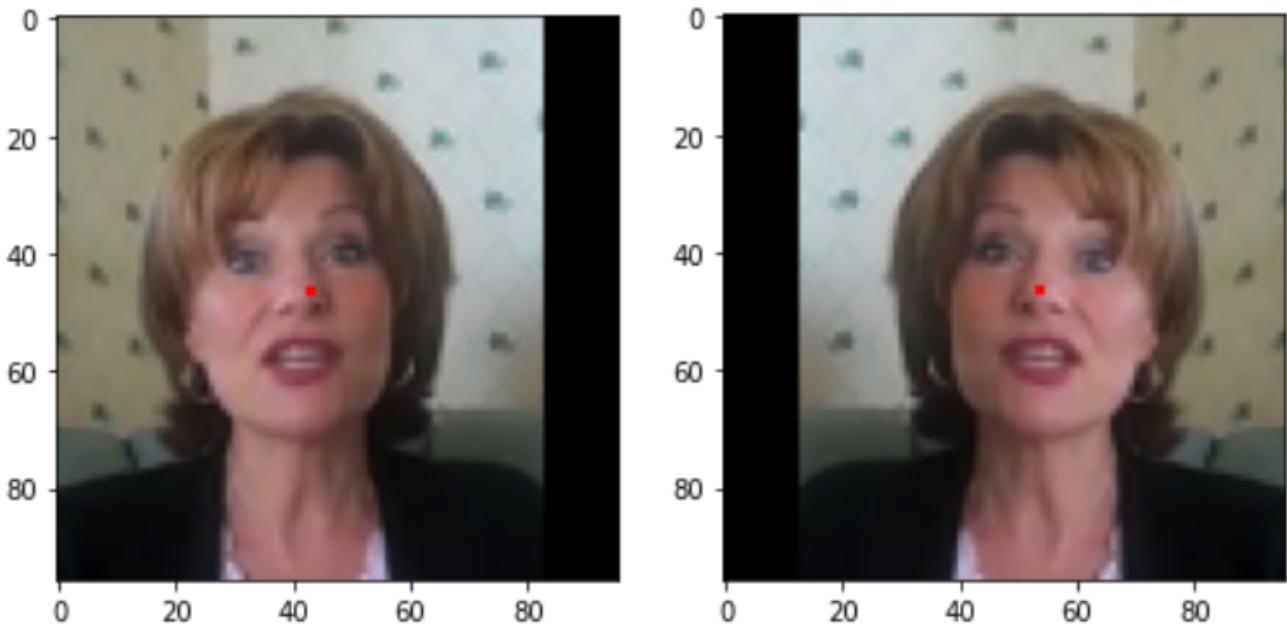
Rysunek 5: Przykład wycinania części zdjęcia numer 2

Rysunek 6



Rysunek 6: Przykład wprowadzania szumu do wartości RGB.

Rysunek 7



Rysunek 7: Obrazek i jego lustrzane odbicie.

Uwaga: Obróbkę zdjęć robiłem "ręcznie" (tzn. z pomocą numpy i PIL), dopiero później trafiłem na klasę [ImageDataGenerator](#) wbudowaną w kerasa która ma wbudowaną większość z użytych transformacji.

2.3 Podział danych

Chociaż wydaje się to lekkim overkillem zebrane dane podzielimy na trzy zbiory:

Training set - to właśnie zawarte w nim dane wykorzystamy do trenowania sieci.

Cross validation set - standardowo używany do ocenienia jak radzi sobie trenowania sieć i do dobierania hiperparametrów (takich jak prędkość uczenia, stopień regularyzacji i rodzaj używanych funkcji aktywacji), chociaż w naszym przypadku nie będziemy próbować dobierać hiperparametrów, wykorzystamy zestaw proponowany przez autorów [1].

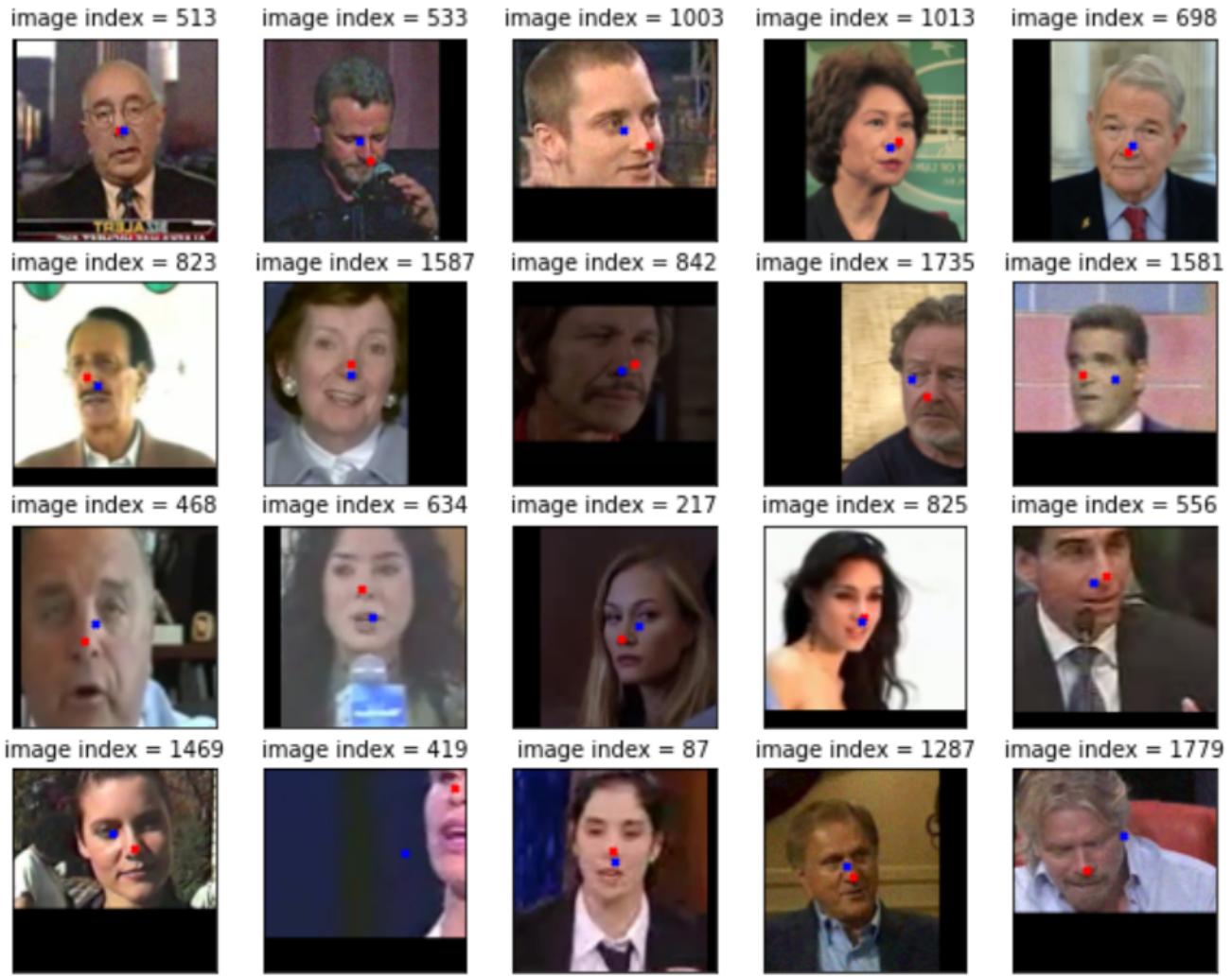
Test set - służy do sprawdzenia na sam koniec jak sprawuje się już wytrenowana z użyciem dobranych parametrów sieć.

3 Wybór modelu

Zgodnie z obietnicą użyjemy gotowej architektury proponowanej przez Yi Sun, Xiaogang Wang, Xiaoou Tang [1]. Dużym plusem zapożyczonej sieci jest jej rozmiar – jest na tyle niewielkie, że spokojnie będziemy w stanie wytrenować ją od zera. W stosunku do oryginalnej sieci wprowadzimy dwie modyfikacje:

Po pierwsze oryginalny model składał się z trzech sieci, po jednej dla każdej z wartości **RGB**. Użyjemy pojedynczej sieci przetwarzającej je jednocześnie.

Rysunek 8



Rysunek 8: Pierwsza próba z maleńkim, prostym modelem. Kolorem niebieskim są zaznaczone punkty wybrane przez sieć a czerwonym oryginalne landmarki.

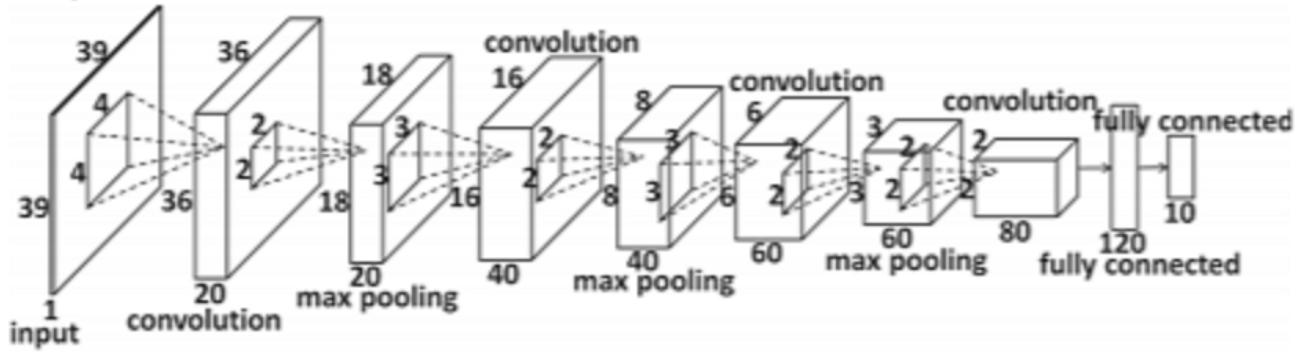
Po drugie podkрадziona przez nasz sieć przyjmowała zdjęcia rozmiaru 39 na 39. Żeby uniknąć utraty jakości przy zbyt dużym przeskalowywaniu my będziemy operować na macierzach $[78 \times 78 \times 3]$ i do oryginalnej architektury dodamy pojedynczy pooling layer gładko przekształcający nasze dane do wielkości na których uczyła się oryginalny model.

3.1 Odrzucone modele

1) Oryginalny model [1] nie korzystała z regularyzacji. Próbowałem dodać do sieci L2 regularyzację i dropout ale taka lekko podrasowana sieć radziła sobie gorzej – nawet po dłuższym treningu uzyskiwała większy training loss, podobny validation loss i co dla nas najważniejsze popełniała wyraźniejsze błędy dla obrazków testowych. (Oczywiście zawsze możemy zmniejszyć parametry regularizacji tak, żeby radziła sobie nie gorzej niż oryginalna sieć ale na razie ten model nie przeszedł próby ognia.)

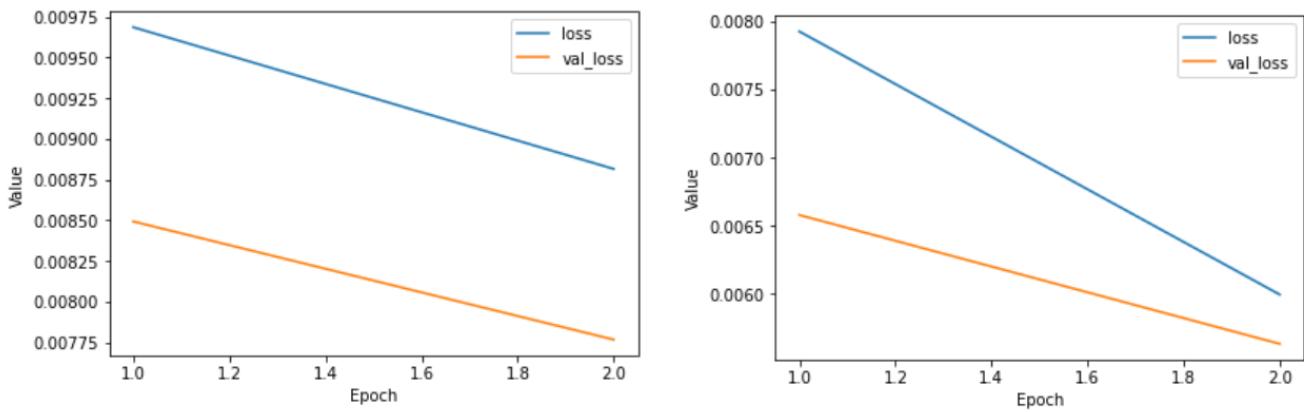
2) Próbowałem wykorzystać wytrenowaną sieć **Resnet** z podmienioną głową ale akurat w tym przypadku transfer

Rysunek 9



Rysunek 9: Struktura sieci pożyczonej z artykułu[1].

Rysunek 10



Rysunek 10: Porównanie prostego modelu z modelem podkрадzionym z [1] dla krótkiego treningu (3 epochs)

learningowy model radził sobie gorzej niż sieć złożona od zera – prawdopodobnie przy odmrożeniu większej ilości poziomów sieć mogłaby dopasować się do naszych danych nieco lepiej ale to z kolei boleśnie wydłużałoby czas potrzebny na trening.

4 Trening

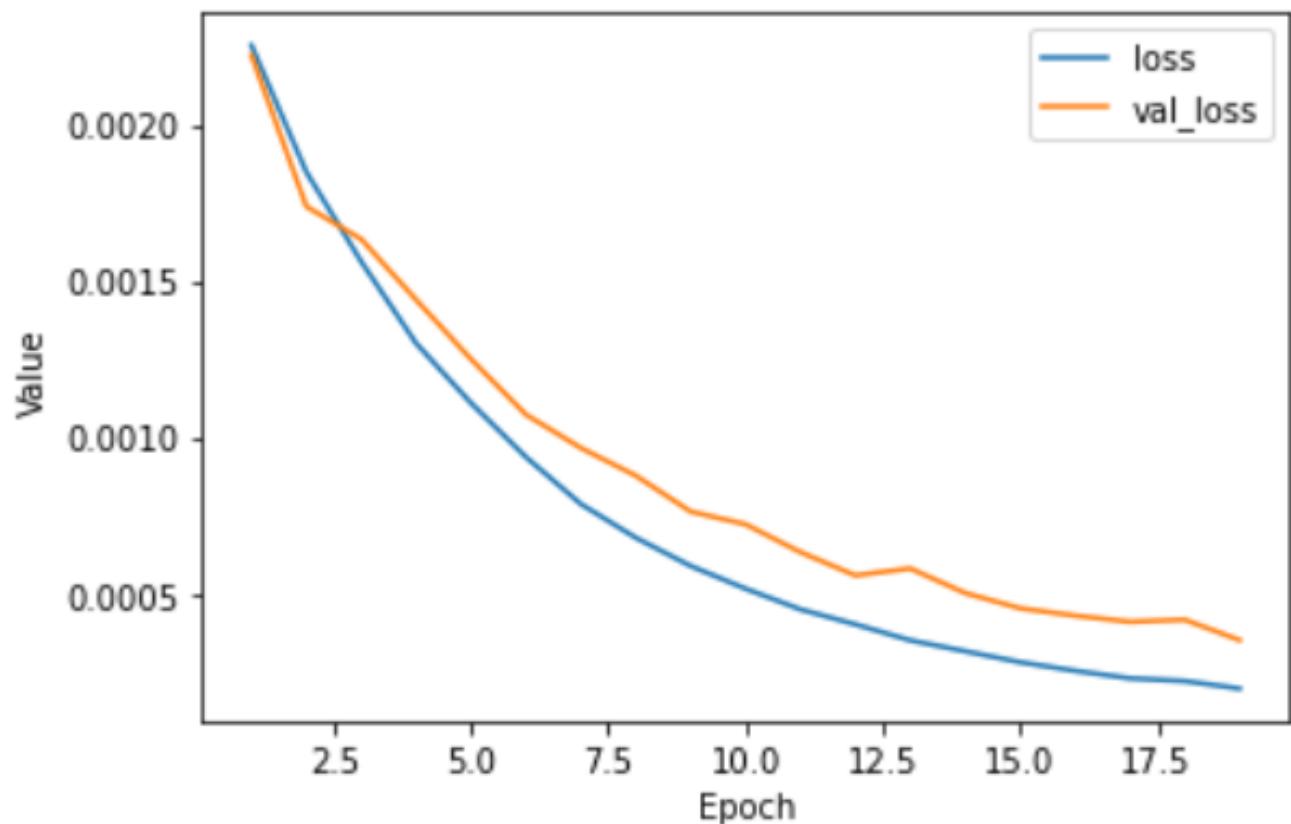
Sieć uczyła się przez 20 epoch z parametrem learning rate równym 10^{-4} (wartość bezczelnie podkradziona z [1]) i później przez kilkanaście dodatkowych przy stopniowym zmniejszaniu learning rate do 10^{-5} . Dla danych z test setu radziła sobie nie najgorzej – udało jej się zejść do błędu rzędu $2.5 \cdot 10^{-4}$. (minimalizowaliśmy mean squared error)

5 Testy

Na sam koniec możemy sprawdzić jak nasz model radzi sobie z rzeczywistymi danymi a przynajmniej z danymi bliskim tym pochodząącym od użytkowników – w naszym przypadku, zobaczymy jak model radzi sobie ze zdjęciami zrobionymi kamerką w laptopie.

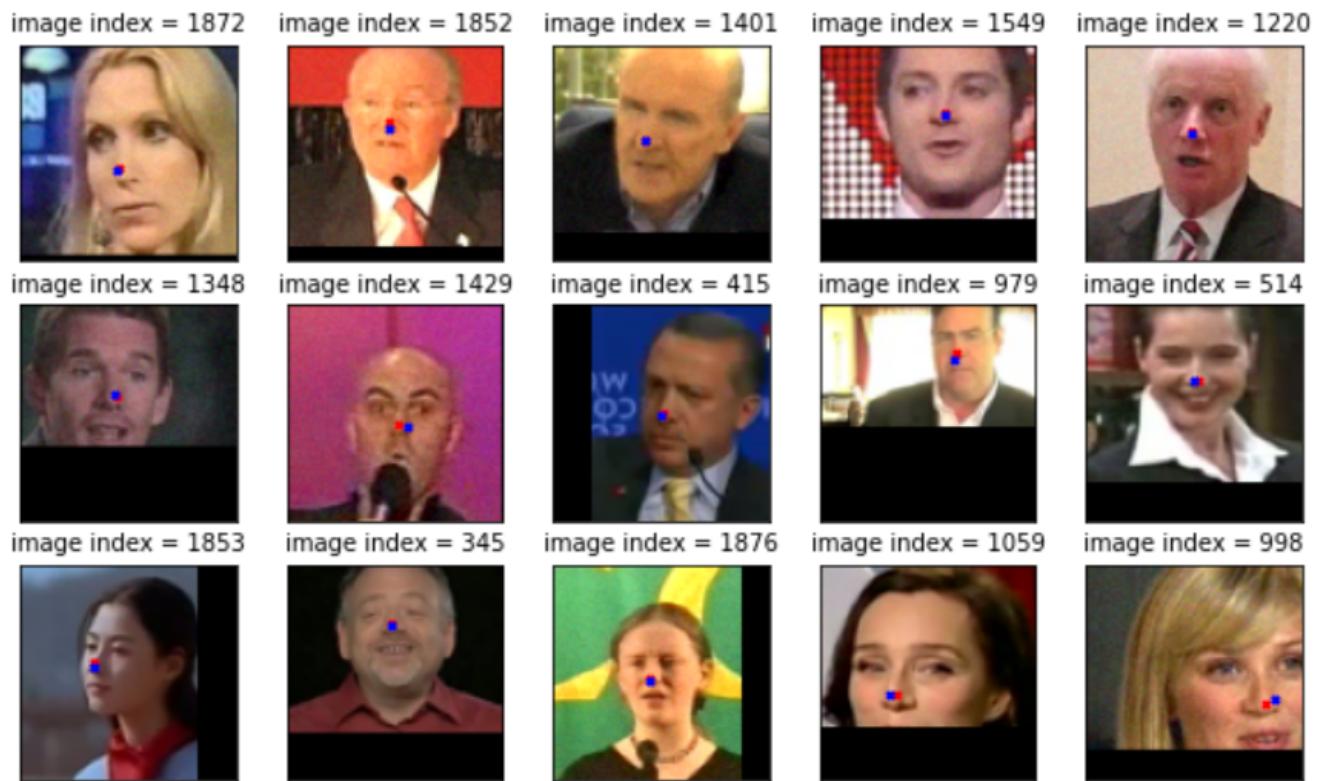
Nasza sieć wydaje się mieć widoczne problemy z obrazkami na których twarz nie jest skierowana prosto w stronę kamery – prawdopodobnie trochę podkłada nam w tym miejscu nogę wykorzystany dataset w którym tego typu zdjęcia nie występowały. Natomiast dla reszty obrazków radzi sobie dość dobrze.

Rysunek 11



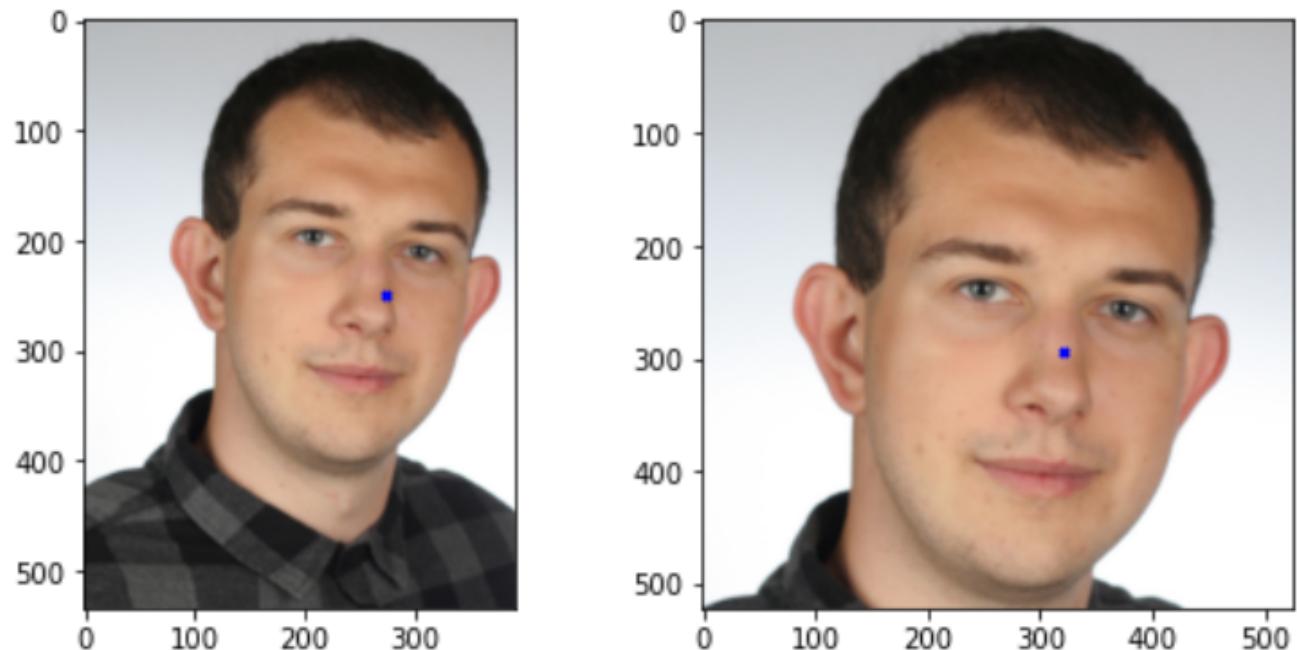
Rysunek 11: Wykres błędów dla trenowania modelu przez 20 epoch.

Rysunek 12



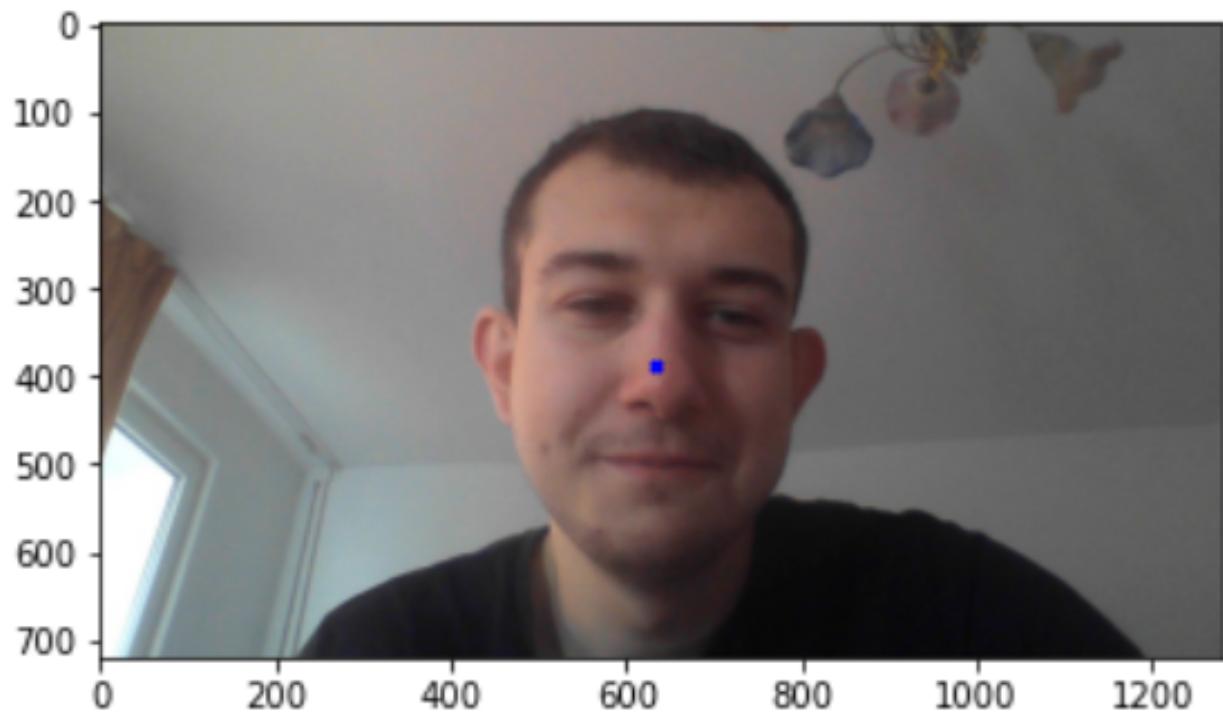
Rysunek 12: Kilka predykcji modelu trenowanego przez 20 epoch.

Rysunek 13



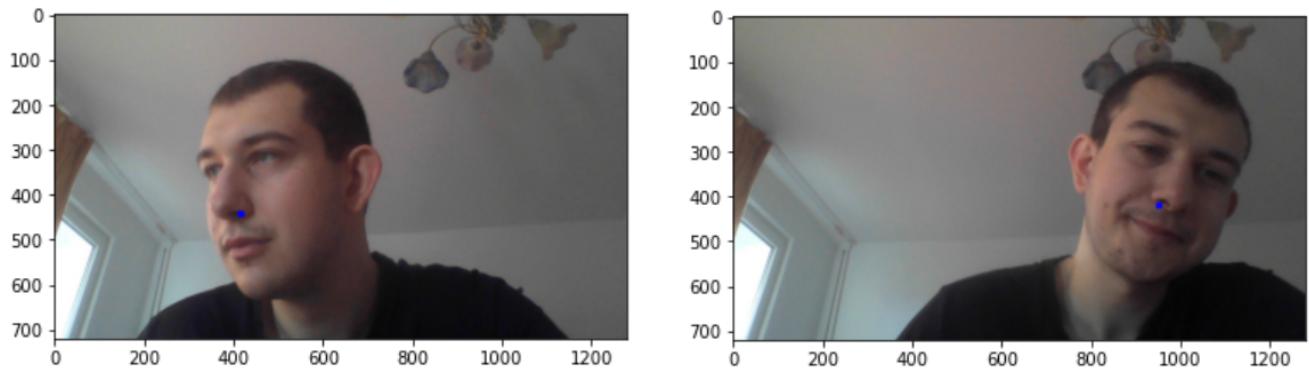
Rysunek 13: Predykcje sieci po skończeniu treningu dla zdjęć "z cv"

Rysunek 14



Rysunek 14: Przewidywanie dla zdjęcia zrobionego kamerką z laptopa.

Rysunek 15



Rysunek 15: Złośliwe przykład, sieć nie radzi sobie najlepiej z głową skierowaną w bok.

Bibliografia

- [1] Y. Sun, X. Wang, and X. Tang. Deep convolutional network cascade for facial point detection. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3476–3483, 2013.