Segmentacja komórek na zdjęciach mikroskopowych skóry z użyciem głębokich sieci neuronowych.

Michał Tracewicz

2020-12-25

Spis treści

1 Wstęp

W codziennej pracy wielu lekarzy poświęca czas na manualne liczenie zafarbowanych komórek na zdjęciach. Jest to czasochłonne i monotonne zadanie. Zdecydowanie nie należy ono do takich, które wymagają sześcioletnich studiów. Co za tym idzie, nadaje się ono idealnie do automatyzacji.

W ostatnich latach bardzo mocno rozwija się dziedzina uczenia maszynowego. Technika ta jest inspirowana działaniem ludzkiego mózgu i pozwala na pisanie programów dla, których trudno jest zdefiniować jednoznaczny zbiór reguł postępowania. Człowiek posiada pewną intuicję dzięki, której jest w stanie w dość krótkim czasie nauczyć się w jaki sposób należy obrysować komórkę. Jednakże gdy zostanie poproszony o wyjaśnienie swojego podejścia będzie to trudne lub wręcz nie wykonalne. Możemy się raczej spodziewać odpowiedzi pokroju "No patrzę na komórkę i ją obrysowuje". Biorąc pod uwagę specyfikę tego problemu użycie tej metody wydaję się idealne.

W ramach mojej pracy przeprowadziłem testy architektury U-Net będącej jedną z najpopularniejszych wyborów do zadań związanych z segmentacją obrazów medycznych. Przetestowałem również różne rodzaje obróbki obrazów wyjściowych w celu zwiększenia skuteczności działania sieci.

Dodatkowo stworzyłem również zestaw narzędzi pozwalający na łatwe testowanie różnych architektur oraz funkcji strat dla sieci neuronowych w tym problemie. Rozwiązanie zostało napisane w języku Python[1] z użyciem następujących bibliotek

- Keras[2] - służy do pracy z sieciami neuronowymi i jest interfejsem do biblioteki TensorFlow
- TensorFlow[3] - biblioteka ta pozwala na wykonywanie obliczeń na tensorach
- NumPy[4] - biblioteka do obliczeń numerycznych na n-wymiarowych tablicach
- Pillow[5] - biblioteka pozwalająca na wczytywanie, zapisywanie oraz edycję obrazów.
- Jupyter Notebook[6] - pozwalająca utworzyć interaktywne środowisko uruchomieniowe Python-a w przeglądarce.

Kod źródłowy projektu jest wersjonowany z użyciem narzędzia git[7]
oraz upubliczniony na platformie GitHub[8]. Prezentacja wyników jest dostępna

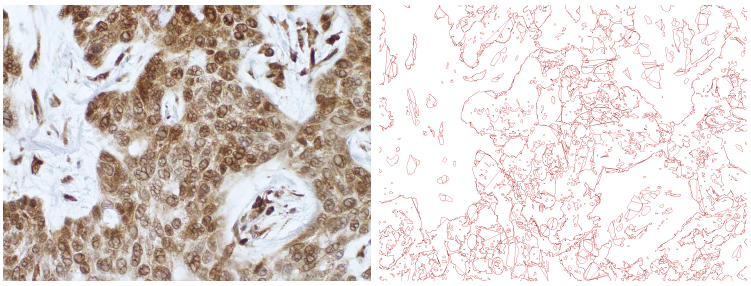
[1] https://www.python.org/
[2] https://keras.io/
[3] https://www.tensorflow.org/
[4] https://numpy.org/
[5] https://python-pillow.org/
[6] https://jupyter.org/
[7] https://git-scm.com/
[8] https://github.com/mtracewicz/CellSegmentation/

online za pośrednictwem platformy GitHub Pages[9] Repozytorium zawierające kod projektu zostało podłączone do platformy CircleCi[10] w celu automatycznego uruchamiania testów jednostkowych (napisane przy użyciu PyTest[11] i Tox[12]) oraz publikacji wyników. W ramach pracy powstały również skrypty w powłoki bash[13] tworzące kontener deweloperski (technologia Docker[14]) zawierający wszystkie niezbędne biblioteki. Całość dopełnia moduł pozwalający na obróbkę zdjęć wejściowych oraz wyjściowych.

Jako, że obrazy medyczne są danymi chronionymi prawnie to szpitale w większości wypadków nie mogą wysłać ich poza własną sieć wewnętrzną. Z tego powodu przetestowałem również możliwość uruchomienia splotowej sieci neuronowej w przeglądarce z użyciem biblioteki TensorFlowJS[15]. Dzięki takiemu podejściu jedynie uczenie modelu przebiega na komputerze nie będącym komputerem klienckim. Model zostaje nauczony na jednym komputerze ale predykcja dla konkretnego obrazu wejściowego wykonywana jest w przeglądarce po stronie klienta (nie następuje wysłanie zapytania do serwera API) i nigdy nie opuszcza sieci.

1.1 Obrazy wejściowe i wyjściowe

W opracowywanym przeze mnie zagadnieniu jako obrazy wejściowe użyte zostały zdjęcia mikroskopowe skóry. Wszystkie zdjęcia są zrobione z przybliżeniem x40 oraz posiadają rozmiar 1200x1600px z trzema kanałami. Obrazy wyjściowe zostały przygotowane za pomocą programu opracowany przez Panią Dominikę Pawłowską. Są to obrazy tego samego rozmiaru jednak zawierają cztery kanały zamiast trzech (RGBA zamiast RGB). Możemy na nich zobaczyć czerwoną otoczkę na około komórek, natomiast cała reszta obrazu jest przezroczysta.



(a) Wejście (b) Wyjście

Rysunek 1: Przykładowe obrazy wejściowy i wyjściowy

[9] https://mtracewicz.ksummarized.com/CellSegmentation.
[10] https://circleci.com/
[11] https://docs.pytest.org/en/latest/
[12] https://tox.readthedocs.io/en/latest/
[13] https://www.gnu.org/software/bash/
[14] https://www.docker.com/
[15] https://www.tensorflow.org/js

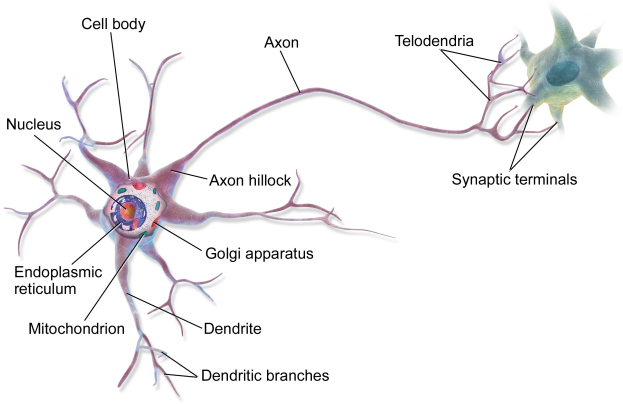
2 Sztuczne sieci neuronowe

Sztuczne sieci neuronowe są próbą stworzenia programów, których działanie jest inspirowane naszym rozumieniem mózgu. Chociaż zagadnienie to podejmowano już od bardzo długiego czasu (1943r.), to dopiero od niedawna posiadamy wystarczającą moc obliczeniową aby realizować tworzone od wielu dziesięcioleci algorytmy.

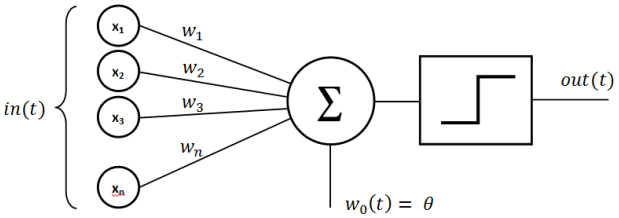
Informacje w tym rozdziale powstały na podstawie książki "Hands-On Machine Learning with Scikit-Learn and TensorFlow" [2] oraz wykładu [5]

2.1 Ogólne zasady działania

Jak wcześniej wspomniałem sztuczne sieci neuronowe są zainspirowane działaniem mózgu. Dlatego w celu ich zrozumienia zaczniemy od przyrównania najprostszej jednostki mózgu do najprostszej jednostki w uczeniu maszynowym, czyli neuron biologiczny i perceptron.



(a) Neuron biologiczny [1]



(b) Perceptron [6]

Neurony przekazują sobie sygnały za pomocą tak zwanych połączeń synaptycznych. Pojedyńczy neuron w momencie otrzymania wystarczająco wielu sygnałów w krótkim czasie sam zaczyna je emitować.

Podobnie działają perceptrony. Gdy suma wejść wymnożonych przez ich wagi przekroczy pewien próg (bias) to perceptron jest aktywny.

Wartość zwracana przez perceptron opiszemy wzorem:

$$O(x) = \sigma(\sum_{i=1}^{n} w_i x_i)$$

gdzie:

- $$\sigma(y) = \begin{cases} -1 & y < \theta \\ 1 & y >= \theta \end{cases}$$
- x - n-elementowy wektor danych wejściowych
- w - n-elementowy wektor wag
- θ - bias

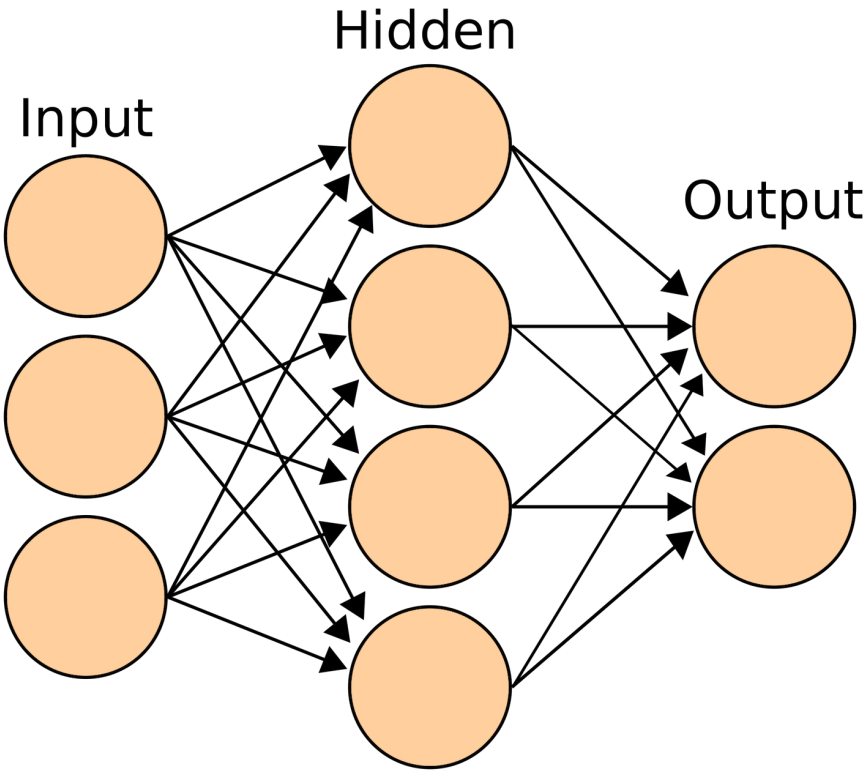
Warto zaznaczyć, że dla ułatwienia obliczeń i jasności wzoru bias jest często dopisywany do wektora wag, natomiast do wektora wejściowego dopisujemy wartość -1. W takim wypadku wzór przyjmuje postać:

$$O(x) = sign(\sum_{i=0}^{n} w_i x_i)$$

Gdzie "sign" to funkcja znaku opisana wzorem

$$sign(x) = \begin{cases} 1 & x >= 0 \\ -1 & x < 0 \end{cases}$$

Głębokie sieci neuronowe składają się z warstwy wejściowej, wyjściowej oraz pewnej liczby warstw ukrytych. Każda warstwa składa się z neuronów, które przyjmują wartość zależnie od wyniku funkcji aktywacyjnej. Liczba oraz typ warstw ukrytych a także ilość zawartych w nich neuronów w dużej mierze determinuje działanie sieci.



Rysunek 3: Głęboka sieć neuronowa z jedną warstwą ukrytą

Omówienie zaczniemy od tego w jaki sposób sieć przeprowadza obliczenia zwracające odpowiedź, a następnie zajmiemy się procesem uczenia.

W uproszczeniu możemy patrzeć na neurony w sieciach jak na perceptrony, które mogą posiadać funkcję aktywacji inną niż progowa. Grupujemy neurony w warstwy a następnie łączymy każdy neuron z warstwy "k" z neuronami z warstwy "k+1". Takie warstwy (posiadają połączenia neuronów każdy z każdym) nazywamy warstwami typu "dense". Każde takie połączenia posiada wagę. Aby obliczyć wartości neuronów w warstwie "k+1" potrzebujemy znać wartości w warstwie k-tej (jako warstwę pierwszą przyjmujemy otrzymany wektor wejściowy). W celu obliczenia wartości konkretnego neuronu postępujemy analogicznie jak przy obliczaniu wartości perceptronu. Traktujemy wartości zwrócone przez neurony z poprzedniej warstwy jako wektor wejściowy i wymarzamy go przez odpowiedni wektor wag.

Wartość dla j-tego neuronu w warstwie k-tej opiszemy wzorem:

$$x_{k,j} = \sigma(\sum_{i=0}^{n-1} w_i x_{(k-1),i})$$

Gdzie (k>=1):

- σ - funkcja aktywacji
- $x_{k,j}$ - wartość j-tego neuronu w k-tej warstwie
- n - liczba neuronów w warstwie k-1
- w_i - i-ta waga wchodząca do neuronu

Obliczając w ten sposób wartości wszystkich neuronów, w ostatniej warstwie otrzymujemy odpowiedź naszego modelu.

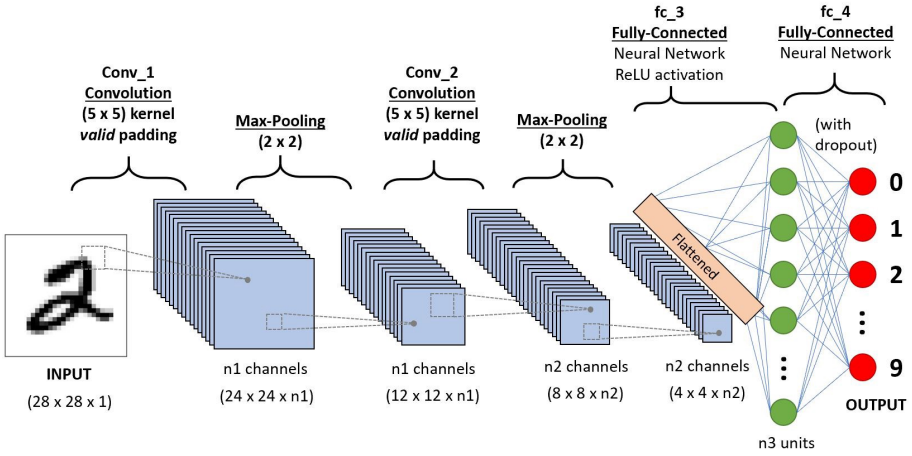
Proces uczenia to wielokrotne przejście przez poprawnie opisane dane wejściowe i poprawianie wag w celu minimalizacji funkcji błędu. W celu poprawy wag używamy algorytmu wstecznej propagacji błędu. Polega on na poprawianiu wag poprzez przejście sieci w przeciwnym kierunku i zmianie wartości wag na podstawie delt wyliczonych przy pomocy algorytmu spadku gradientowego.

2.2 Porównanie do klasycznych metod programowania

Przy standardowych paradygmatach programowania (programowanie zorientowane obiektowo, programowanie proceduralne, programowanie funkcyjne) jako programista piszemy kod, który instruuje komputer na temat tego w jaki sposób ma on przekształcić otrzymane dane wejściowe. Natomiast sieci neuronowe są zdecydowanie bardziej zbliżone do programowania deklaratywnego. Jako programista podajemy dane wejściowe, architekturę i oczekiwane dane wyjściowe i na ich podstawie, chcemy aby komputer samemu "nauczył się" tego jaką drogą ma dojść aby uzyskać oczekiwany efekt (jakie wartości mają przyjmować wagi).

3 Sieci splotowe

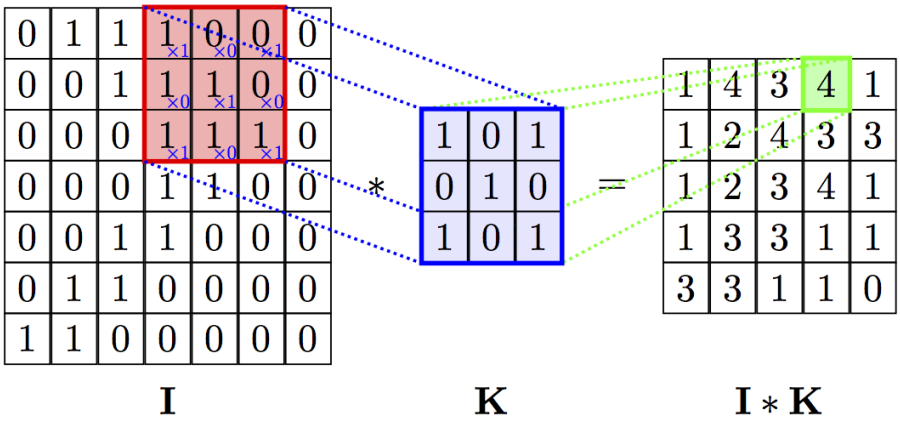
Sieci splotowe to takie głębokie sieci neuronowe, które zawierają jedną lub więcej warstw splotowych. Warstwa splotowa przeprowadza operację splotu na obrazach wejściowych i w wyniku zwraca inne obrazy (w liczbie zależnej od parametrów).



Rysunek 4: Przykład sieci splotowej dla problemu MNIST
[8]

3.1 Operacja splotu

Operacja splotu w kontekście sieci neuronowych jest to przebieg jądra przez wartości pikseli obrazu wejściowego. W przestrzeni dwu wymiarowej jest to równoważne z nałożeniem na obraz wejściowy filtra.



Rysunek 5: Operacja splotu
[4]

3.2 Sieci w pełni splotowe

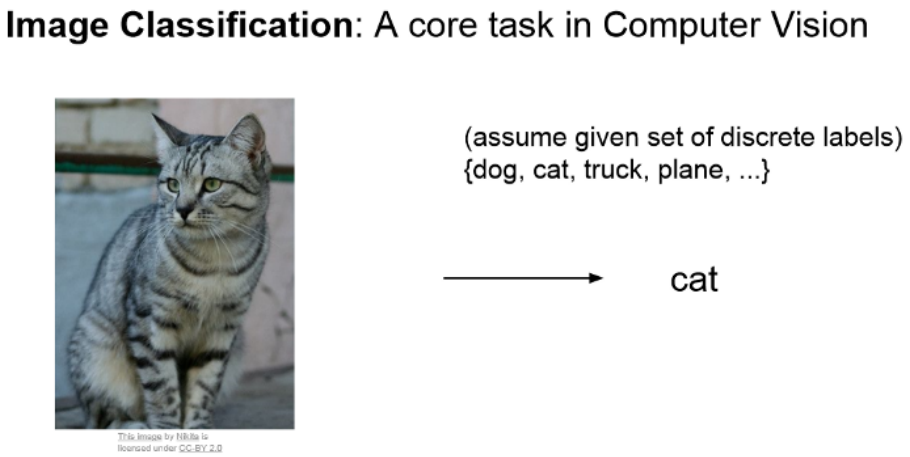
Sieciami w pełni splotowymi nazywamy te głębokie sieci neuronowe, które jako wszystkie warstwy posiadają warstwy splotowe. Sieci tego typu służą głównie do przekształcania obrazów. Jedną z lepiej znanych architektur tego typu jest architektura "U-NET" z, której skorzystałem.

4 Problem segmentacji

Uczenie maszynowe możemy podzielić na różne podtypy. Jednym z nich jest zagadnienie wizji komputerowej. Zaznaczenie konturów komórek na zdjęciach mikroskopowych jest problemem segmentacji instancji, która jest częścią tego zagadnienia. W celu zrozumienia tego pojęcia przeanalizujemy składające się na niego mniejsze problemy z kategorii widzenia komputerowego.

4.1 Klasyfikacja

Jest to najprostsze zagadnienie w dziedzinie widzenia komputerowego. Polega ono na tym, że dla otrzymanego na wejście obrazu sieć ma zwrócić klasę do której przedstawiony na zdjęciu obiekt należy. Za dobry przykład może nam posłużyć "Problem MNiST", polegający na rozpoznawaniu odręcznie napisanych cyfr. Jako wejście sieć otrzymuje czarno-biały obrazek rozmiaru 16x16 pikseli na, którym widnieje odręcznie napisana cyfra. Natomiast na wyjściu sieć zwraca nam predykcję jaka cyfra widnieje na zdjęciu. Ograniczeniem w tym typie problemu jest fakt, że na danym zdjęciu może znajdować się jedynie pojedyńczy obiekt.

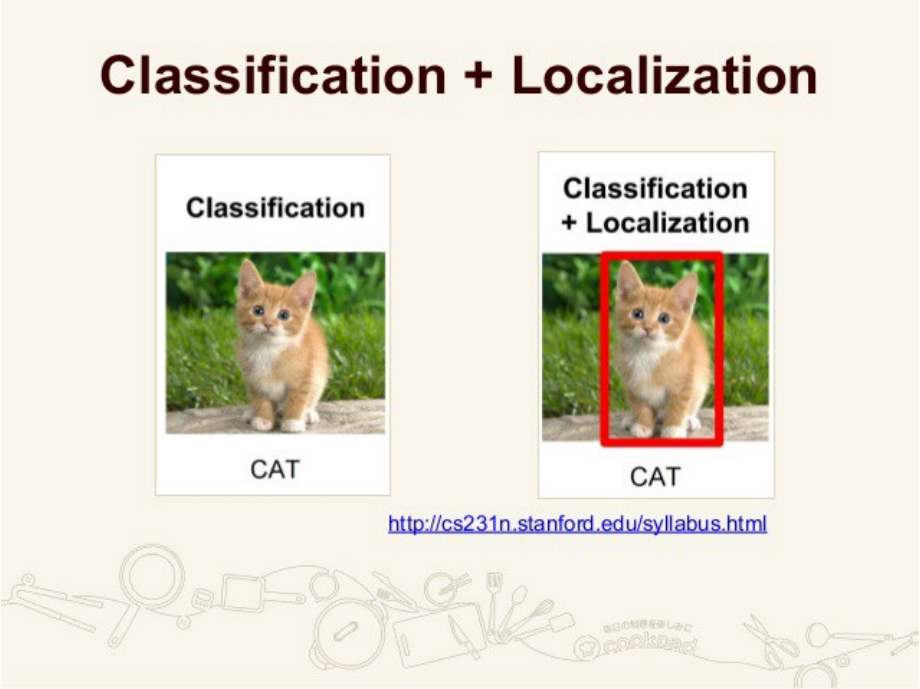


Rysunek 6: Przykład klasyfikacji
[3]

4.2 Klasyfikacja z lokalizacją

Klasyfikacja z lokalizacją to dodanie do poprzedniego problemu zagadnienia odnalezienia klasyfikowanego obrazu na zdjęciu. Przykładowo tworząc sieć rozpoznającą gatunki zwierząt domowych, otrzymując zdjęcie kota siedzącego na kanapie otrzymamy nie tylko informację, że jest to kot ale również informację o tym gdzie na tej kanapie się znajduje. Najczęściej zwracane są w takim wypadku

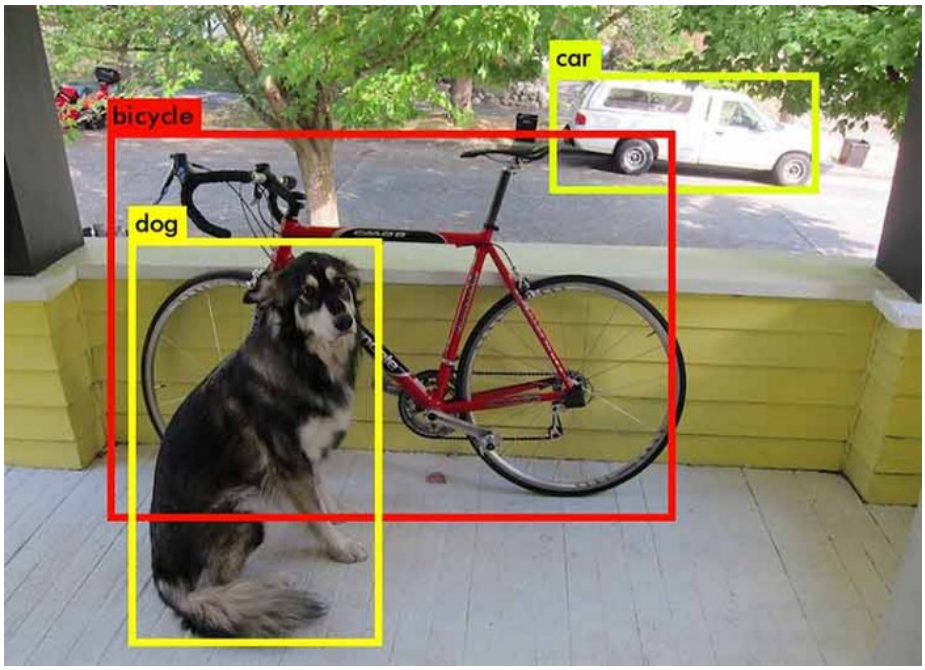
oprócz klasy współrzędne pikseli w których zaczyn i kończy się obiekt. Następnie możemy na tej podstawie narysowanie naokoło obiektu ramkę z podpisem zawierającym nazwę klasy.



Rysunek 7: Przykład klasyfikacji z lokalizacją
[3]

4.3 Wykrywanie obrazów

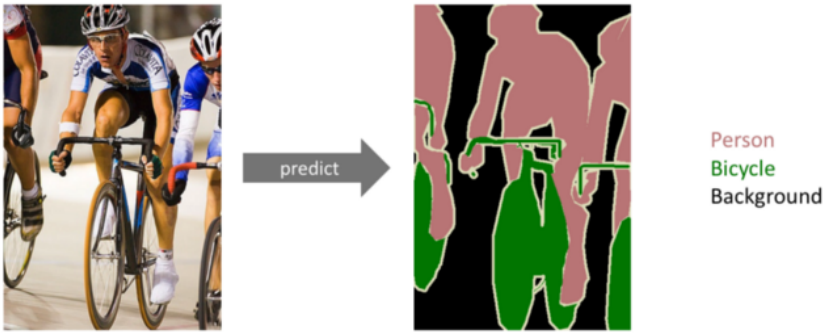
Kolejnym krokiem jest wykrywanie obrazów. Jest to klasyfikacja z lokalizacją dla wielu elementów jednocześnie. Kontynuując poprzedni przykład z rozpoznawaniem zwierząt domowych. W wypadku zdjęcia, które zawiera zarówno psa i kota dostaniemy zestaw współrzędnych oraz odpowiadających im klas.



Rysunek 8: Przykład wykrywania obrazów
[3]

4.4 Segmentacja

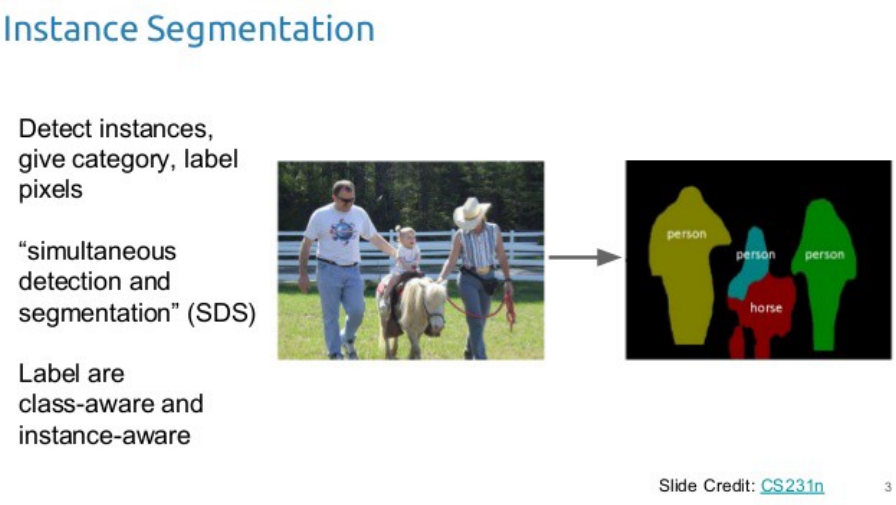
Segmentacja to przypisanie każdemu pikselowi na obrazie wejściowym klasy. Za przykład może nam posłużyć zdjęcie drogi w mieście. Celem naszej sieci neuronowej będzie podział obrazu na klasy: droga, człowiek, budynek, drzewo, niebo etc. Na wyjściu możemy na przykład otrzymać ten sam obrazek z tym, że zależnie od klasy danego piksela będzie miał on różny kolor (czerwony - droga, zielony - człowiek etc.)



Rysunek 9: Przykład segmentacji
[3]

4.5 Segmentacja instancji

Segmentacja instancji to dodanie do segmentacji ograniczenia, że każda jednostka ma być widocznie oddzielona. Dla przykładu gdy mamy zdjęcie czterech przytulających się osób, w wyniku segmentacji wszystkie cztery zostaną zakolorowane na ten sam kolor oraz podpisane raz jako człowiek. Natomiast gdy przeprowadzimy segmentację instancji to w wyniku otrzymamy każdą osobę pokolorowaną na własny kolor, oraz etykiety "człowiek 1.", "człowiek 2." etc.



Rysunek 10: Przykład segmentacji instancji
[3]

5 Architektura U-Net historia i zastosowania

W roku 2015 Olaf Ronneberger, Philipp Fischer, Thomas Brox stworzyli architekturę sieci, której zadaniem jest segmentacja obrazów biomedycznych [7].

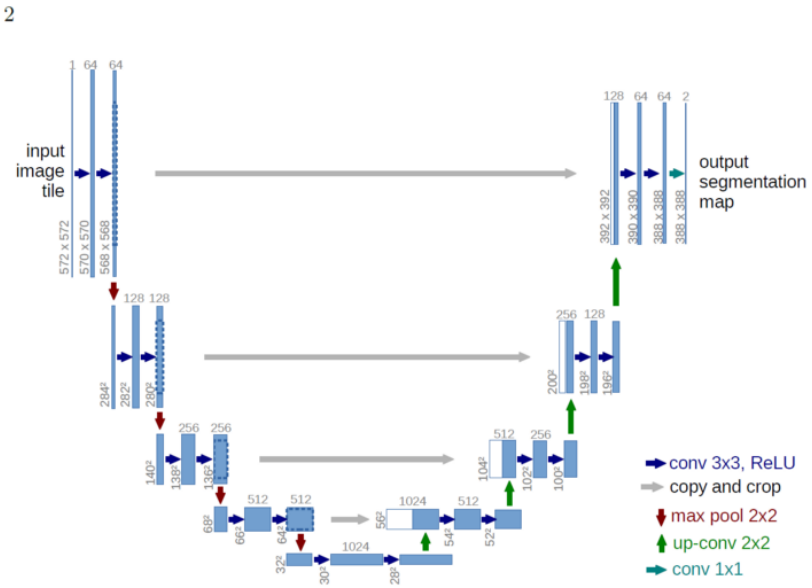


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

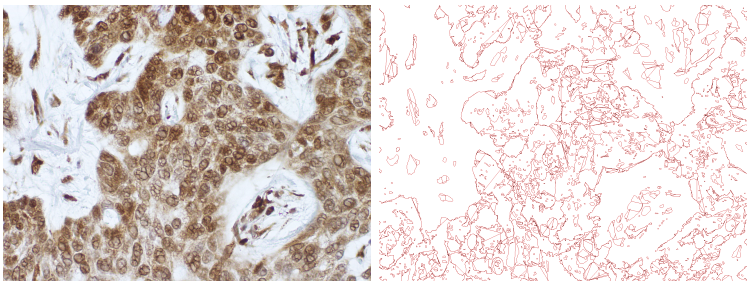
Rysunek 11: Architektura U-NET
[3]

Założeniem sieci jest to aby wyciągnąć informacje o detalach nie tracąc przy tym informacji przestrzennych.

Patrząc na ilustrację architektury możemy ją podzielić na dwie części. Część zmniejszając oraz zwiększającą. Idąc ścieżką zmniejszającą szukamy detali, jest to typowe zastosowanie sieci splotowych. Jednakże druga część sieci, powiększa obraz z powrotem i dzięki temu odzyskujemy informacje o lokalizacji.

6 Przetestowane podejścia

W ramach pracy przetestowałem architekturę U-Net oraz różne podejścia do obróbki danych. Mimo wielu prób sieci które przygotowałem zawsze zwracały niemal w pełni czarne/czerwone obrazy.



Rysunek 12: Przykładowe otrzymane wyniki

6.1 Bazowa sieć U-Net

Pierwszym podejściem, które przetestowałem było uruchomienie trzy poziomowej (dziesięciu warstwowej) sieci neuronowej w architekturze U-Net. Sieć otrzymywała na wejście obrazy rozmiaru 200x200x3 a jako wyjście zwracała obrazy 200x200x4 (oryginalne obrazy były rozmiaru 1200 na 1600 pikseli, więcej na ten temat w kolejnym punkcie). Na tym etapie przeprowadziłem również testy modyfikując opisujące te sieć hiperparametry. Zmieniałem liczbę epok (między wartościami z przedziału od 20 do 40), stałą uczącą (w przedziale od 0.003 do 0.1) oraz wielkość parti (w przedziale od 1 do 25). Dodatkowo modyfikowałem również ilość filtrów używanych w każdej warstwie. Niezależnie od tych zmian wszystkie powyższe eksperymenty zwracały jako predykcję obrazy praktycznie w stu procentach czarne, zawierały maksymalnie dwa/trzy piksele koloru czerwonego. Działo się tak mimo, że sieć uzyskiwała w trakcie uczenia/walidacji wyniki dokładności na poziomie nawet dziewięćdziesięciu czterech procent.

Powyższe testy uświadomiły mi konieczność znalezienia lepszej funkcji strat/metryki mierzącej poprawność lub poprawienie w jakiś sposób danych wejściowych i wyjściowych w celu ułatwienia sieci procesu uczenia.

6.2 Preprocessing obrazów

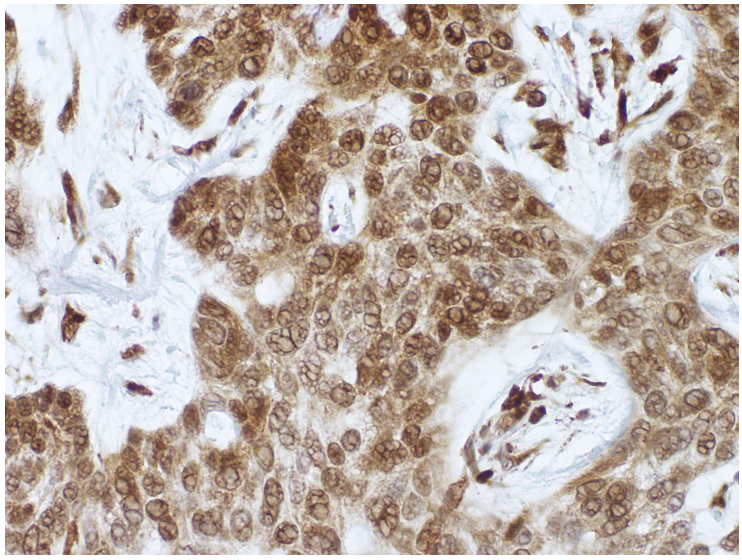
Skupię się teraz na modyfikacji danych, której dokonałem w nadziei na poprawę wyników uczenia.

6.2.1 Cięcie i sklejanie obrazów

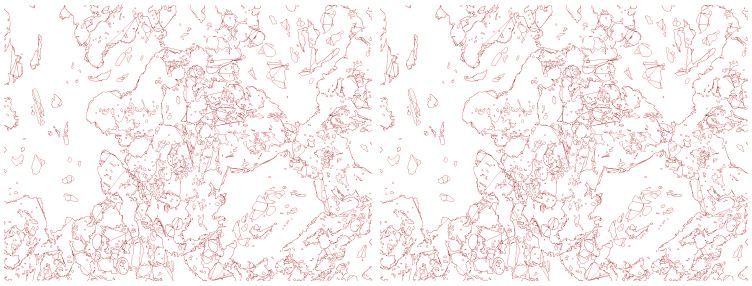
Przy rozwiązywaniu postawionego przede mną problemu natkąłem się na dwa problemy, które miały to samo rozwiązanie. Pierwszym z nich była wielkość obrazów wejściowych, która przy wybranej przeze mnie architekturze sieci była

zdecydowanie zbyt duża. Był to problem na tyle poważny, że na dostępnym dla mnie sprzęcie nie byłem nawet w stanie uruchomić sieci z minimalną ilością warstw i filtrów. Natomiast drugim problemem był niedobór danych uczących. Otrzymałem dziewięć obrazów wejściowych/wyjściowych. Ilość ta jest zdecydowanie zbyt mała aby nauczyć sieć.

Doszedłem do wniosku, że oba te problemy mogę rozwiązać poprzez pocięcie dużych obrazów wejściowych (1200 na 1600 pikseli) na wiele mniejszych (200 na 200 pikseli z zakładką wynoszącą 100 pikseli), a następnie uczenie na nich modelu.



(a) Obraz oryginalny



(b) Lewy górny róg (c) Kolejny obraz na prawo

Rysunek 13: Przykładowe cięcia obrazów wejściowych

Gdy model będzie już wyuczony w celu otrzymania segmentacji obraz wejściowy będzie najpierw dzielony w ten sam sposób, aby był zestawem mniejszych zdjęć, potem nastąpi segmentacja a po niej wyniki zostaną scalone ponownie w jeden obraz wyjściowy (w miejscach, gdzie będzie więcej niż jedna odpowiedź

dla piksela z powodu kieszonki przyjmę wartość maksymalną). Dzięki takiemu rozwiązaniu udało mi się znacząco zmniejszyć ilość pamięci operacyjnej wymaganej dla pojedyńczego przebiegu sieci, dzięki czemu mogę wykorzystać architekturę U-Net oraz znacząco zwiększyć jej skuteczność, ponieważ dostępną pamięć można wykorzystać by dodać dodatkowe warstwy/filtry. Dodatkowo rozmiar 200 na 200 pikseli jest obrazem kwadratowym a z takimi najlepiej radzi sobię ta architektura.

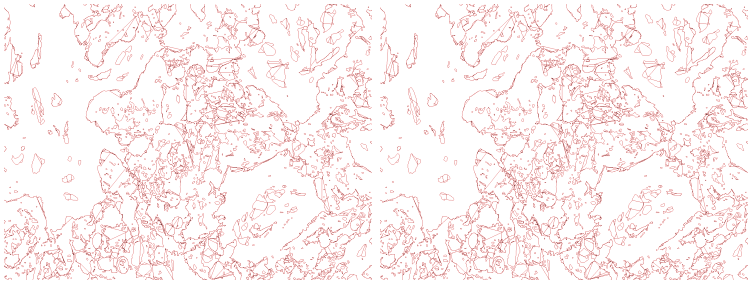
Kolejnym plusem jest to, że w ten sposób ilość moich danych wejściowych drastycznie wzrośnie, ponieważ z każdego jednego obrazu utworzy się trzysta mniejszych, z których każdy będzie oddzielny/niezależnym przykładem uczącym (w tym wypadku obraz oryginalny składa się na siatkę o szerokości dwudziestu i wysokości piętnastu mniejszych obrazów). Dzięki czemu otrzymujemy zamiast jednego przykładu uczącego trzysta.

Wprowadzając opisane powyżej podejście udało mi się uruchomić proces uczenia sieci neuronowej. W wyniku otrzymałem jednak obraz który posiadał około 2 piksele czerwone, podczas gdy wszystkie pozostałe były czarne.

6.2.2 Wybór jednego kanału - czerwony/alfa

Analizując dane wyjściowe zauważyłem, że piksele tła posiadają wartości kanału alpha bliskie zeru natomiast piksele linii obrysowującej komórki przybierają wartości bliskie dwustu pięćdziesięciu pięciu. Niemal identycznie wyglądały wartości kanału czerwonego. Natomiast kanały zielony i niebieski miały niemal zawsze wartość równą zero.

Dlatego wpadłem na pomysł, żeby spróbować potraktować kanały alfa i czerwony jako obrazy monochromatyczne i użyć ich jako dane wyjściowe. W ten sposób w założeniu predykcje tworzyłyby monochromatyczny obraz. W celu ujednolicenia wyników otrzymane tak obrazy zostały potraktowane jako kanał czerwony obrazu wynikowego a kanały niebieski i zielony otrzymały arbitralnie wartość zero dla wszystkich pikseli (otrzymany po takiej modyfikacji obraz był czarno/czerwony, tak jak oryginalne obrazy).



(a) Lewy górny róg (b) Kolejny obraz na prawo

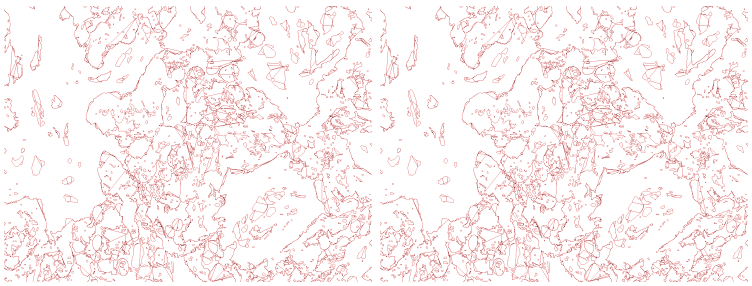
Rysunek 14: Przykładowy otrzymany obraz

Niestety zastosowanie tego rozwiązania nie przyniosło porządnych efektów.

Obrazy wynikowe przyjmowały teraz w całości kolor czerwony (gdy jako wyjście przykładów uczących podaliśmy obraz monochromatyczny wygenerowany z kanału czerwonego) lub czarny (gdy dane wyjściowe były na podstawie kanału alfa).

6.2.3 Redukcja do obrazów trzy kanałowych i redukcja kolorów

Kolejnym pomysłem było przeprowadzenie redukcję wymiarowości danych wyjściowych oraz zmniejszyłem ilość kolorów w obrazie. Wartości kanałów czerwonego i alfa są ze sobą powiązane (dla tła obie zmierzają do zera). Dlatego zdecydowałem się na zmianę obrazów wyjściowych z formatu RGBA do RGB. Zależnie od wartości kanałów czerwonego i alfa zaokrągliłem kolor danego piksela do czystego czerwonego (255, 0, 0) lub czystego czarnego (0, 0, 0). Przeprowadziłem te operację z różnymi wartościami progowymi dla obu kanałów. W wyniku jednak nadal otrzymywałem całkowicie czerwone/czarne obrazy zależnie od tego jak dużo pikseli zostało zaokrąglonych do danego koloru.



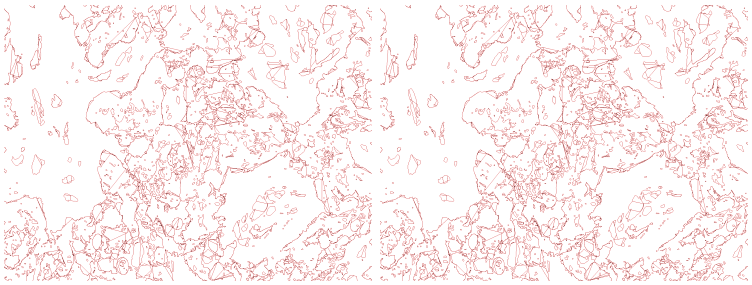
(a) Lewy górny róg (b) Kolejny obraz na prawo

Rysunek 15: Przykładowy otrzymany obraz

6.2.4 Rozmycie Gausa

Otrzymane w poprzednich próbach wyniki skłoniły mnie do myśli, że proporcje koloru czerwonego do czarnego są zbyt nierówne. Z powodu tej nierówności model jest w stanie uzyskać wysoką wartość dokładności (rzędu dziewięćdziesięciu czterech procent) jako predykcję zwracając wszystkie piksele w tym samym kolorze.

Pomysłem na rozwiązanie tego problemu było zastosowanie na obrazach docelowych rozmycia Gaussa przed pokazaniem ich sieci. Filtr ten rozmywa obraz co w wypadku dwu kolorowego obrazu zawierającego linie oraz tło będzie skutkowało poszerzeniem linii.



(a) Przed (b) Po

Rysunek 16: Obraz wyjściowy przed i po zastosowaniu filtra

6.3 Funkcje start

We wszystkich wykonanych do tej pory eksperymentach otrzymywałem jako predykcję jednokolorowe obrazy. Jednakże dla każdego z tych podejść otrzymywałem bardzo wysoką wartość dokładności i małą wartość funkcji strat. Dlatego zdecydowałem się na zbadanie innych sposobów oceny poprawności działania sieci.

6.3.1 Funkcja strat - Współczynnik Sørensena

Często używaną w zagadnieniu segmentacji funkcją strat jest $1 - QS$ gdzie QS to współczynnik Sørensena opisany wzorem:

$$QS = \frac{2 * |X \cap Y|}{|X| + |Y|} \tag{1}$$

Dla:

- X - zbiór wejściowy
- Y - zbiór wyjściowy

Listing 1: Implementacja przy użyciu Keras i TensorFlow

```
from tensorflow.keras import backend as K

def custom_metric(y_true, y_pred):
    y_true = K.flatten(y_true)
    y_pred = K.flatten(y_pred)
    return (2. * K.sum(y_true * y_pred) +1) / (K.sum(y_true) +
        K.sum(y_pred) + 1)
```

Korzystając z tej funkcji strat otrzymałem prawie dokładnie takie same obrazy wyjściowe. Jednakże w przeciwieństwie do użycia dokładności jako metryki otrzymane wartości oscylowały w okolicy dwudziestu procent (zamiast okolic dziewięćdziesięciu procent).

6.3.2 Własna funkcja strat

Kolejny pomysł opierał się na tym aby metryka mierząca poprawność zwracała większą uwagę na poprawnie sklasyfikowane czerwone piksele (gdyż jest ich znacznie mniej). Piksele, które zostaną poprawnie z kategoryzowane a mają kolor czerwony będą "X" razy ważniejsze niż poprawnie sklasyfikowane piksele czarne.

$$\frac{X * \frac{PC}{AC} + \frac{PCZ}{ACZ}}{x + 1} \tag{2}$$

Gdzie:

- X - waga ile razy ważniejsze są piksele czerwone
- PC - liczba poprawnie sklasyfikowanych czerwone pikseli
- AC - liczba wszystkich czerwonych pikseli
- PCZ - liczba poprawnie sklasyfikowanych czarnych pikseli
- ACZ - liczba wszystkich czarnych pikseli

6.4 Obliczenie predykcji na komputerze klienckim

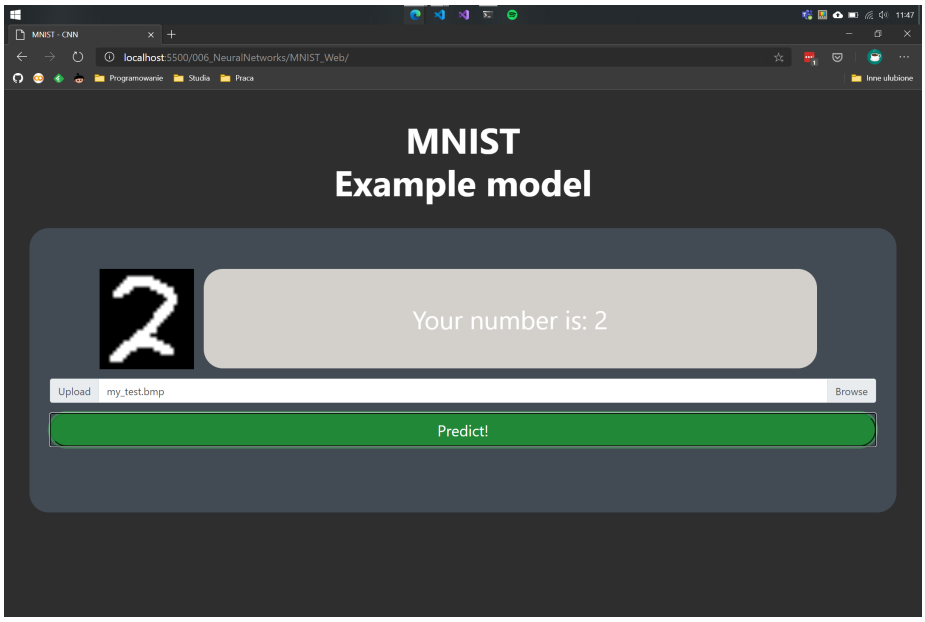
Jednym z problemów przy pracy z danymi medycznymi jest to, że nie mogą one opuścić sieci wewnętrznej szpitala. Dlatego nie możemy uruchomić wytrenowanego modelu na serwerze i zapewnić możliwości odpytania poprzez API.

Rozwiązaniem w tej sytuacji było przeniesienie wytrenowanych w TensorFlow modeli do TensorFlowJS który pozwala na użycie ich na komputerze klienckim. Testy te były dokonane na znacznie prostszy modelu służącym do rozpoznawania odręcznie pisanych cyfr ze zbioru MNIST (stosował on jednak wszystkie warstwy z, których składa się model w architekturze U-Net, jedynie w innej ilości/konfiguracji).

Pierwszym krokiem było przekształcenie modelu z biblioteki Python-owej na bibliotekę javascript-ową. Można to było łatwo wykonać w skrypcie Pythonowym za pomocą metody tfjs.converters.save_keras_model, której przekazujemy model oraz ścieżkę do niego. Największym problemem w tej zmianie było jednak to, że aby użyć biblioteki w javascript operacje przez nią wykonywane musiały być poza wątkiem głównym, w celu nie blokowania UI - jest to narzucane przez samą bibliotekę. Aby to osiągnąć posłużyłem się tak zwanymi webworker-ami. Pozwalają one na wykonywanie zadań równolegle do wątku głównego. W wątku głównym tworzymy obiekt klasy Worker oraz subskrybujemy wydarzenie worker.onmessage. Dzięki temu możemy wykonać operację na danych otrzymanych w wiadomości. Natomiast w celu wysłania zapytania do workera używamy metody worker.postMessgae w której przekazujemy dane. W moim wypadku była to pobrana z wyświetlonej kanwy tablica z danymi o pikselach w obrazku. W skrypcie zawierającym kod źródłowy workera również

subskrybujemy wydarzenie onmessage, w jego obsłudze najpierw preprocesuję dane w ten sposób aby ich format na wejście był taki sam jak danych uczących. Jest to wymagane nawet jeżeli chcielibyśmy sprawdzić model korzystając z tego samego obrazu ponieważ biblioteki PIL oraz NumPy mają inny format danych niż dane zwrócone przy pomocy javascript z obiektu canvas dostępnego w HTML5 na którym wyświetlany był obraz do klasyfikacji. Następnie po doprowadzeniu otrzymanych danych do odpowiedniego formatu jesteśmy w stanie użyć wyuczonego w Pythonie i TensorFlow modelu w celu uzyskania predykcji. Następnie za pomocą metody postMessage jesteśmy w stanie wysłać do wątku głównego wiadomość zawierającą predykcję modelu, który ją wyświetli.

Dzięki powyższemu rozwiązaniu jesteśmy w stanie uruchomić nasz model po stronie klienta w przeglądarce. Dodatkowo zapewniona jest płynność działania ponieważ procesowanie danych oraz predykcja modelu wykonywane są równolegle do wątku głównego, który obsługuje interfejs użytkownika.



Rysunek 17: Przykład klasyfikacji obrazu w przeglądarce

7 Powstałe narzędzia

W trakcie prac powstał szereg narzędzi mających za zadanie ułatwić pracę nad tym zagadnieniem. Kod wraz z historią jest dostępny na platformie GitHub[16]. Wyniki są prezentowane za pomocą platformy GitHub Pages[17].

7.1 Skrypty pozwalające na preprocessing obrazów uczących

W ramach prac stworzyłem cztery skrypty pozwalające na edycję zdjęć wejściowych/wyjściowych. Powstałe skrypty pozwalają na:

- zastosowanie filtra rozmycia Gaussa z wybranym parametrem na pojedyńczym obrazie lub katalogu obrazów
- pocięcie obrazów na mniejsze nachodzące się na siebie obrazki
- zespolenie obrazków utworzonych poprzednim skryptem z powrotem w całość
- zredukowanie kanałów obrazów z RGBA do RGB i zaokrąglenie wartości pikseli do czerwonego (255, 0, 0) i czarnego (0, 0, 0)

7.2 Skrypt do uczenie sieci

W celu umożliwienia automatyzacji procesu uczenia stworzyłem skrypt pozwalający na uruchamiane procesu uczenia z różnymi parametrami. Dzięki takiemu podejściu jesteśmy przykładowo w stanie uruchomić przez noc proces uczenia dla kilku różnych kombinacji architektur, hiperparametrów, danych wejściowych/wyjściowych.

Skrypt przyjmuje na wejściu pod jaką nazwą ma zostać zapisany wytrenowany model oraz katalogi ze zdjęciami wejściowymi i wyjściowymi.

Dodatkowo opcjonalnie możemy podać ścieżkę do skryptu napisanego w języku Python w, którym znajdują się funkcje o nazwach "custom_loss" i "custom_metrics" (w wypadku braku domyślnie użyty jest współczynnik Sørensena), które zostaną użyte jako metryka oraz funkcja strat.

Kolejnym opcjonalny parametr jest odpowiedzialny za użytą architekturę, jest to ścieżka do pliku zawierającego funkcję "get_model", która zwraca obiekt klasy "tf.Keras.Model" (domyślnie używany jest model z moją implementacją architektury U-Net).

Następny parametr to ścieżka do pliku zawierającego hiperparametry jako zmienne w skrypcie języka Python (domyślnie jest to plik "hyperparameters.py" z napisanego przeze mnie moduły "neural_network").

[16] https://github.com/mtracewicz/CellSegmentation
[17] https://mtracewicz.ksummarized.com/CellSegmentation/

Skrypt zaczyta dane do pamięci a następnie wykona uczenie modelu ("tf.keras.Model.fit") na podanych danych ze wskazaną funkcją strat, metryką, hiperparametrami i architekturą. Następnie wynikowy model zostanie zapisany pod wskazaną nazwą w katalogu "trained_models".

7.3 Skrypt pozwalające na testowanie zapisanych modeli

Kolejny skrypt przyjmuje na wejście ścieżkę do zapisanego modelu oraz ścieżki do katalogów zawierających dane wejściowe i wyjściowe. A następnie przeprowadza na nich test sprawdzający jak dobrze model sobie na nich poradził za pomocą metody model.evaluate.

7.4 Skrypt pozwalające na predykcję dla obrazu/folderu obrazów

Stworzyłem również skrypt, który otrzymując na wejście ścieżkę do modelu, którego chcemy użyć oraz ścieżkę do obrazu wejściowego/katalogu obrazów wejściowych zapiszę do pliku/plików nowy obraz będący predykcją modelu.

7.5 Testy jednostkowe

W celu wykazania poprawności działania oraz umożliwienia szybszej modyfikacji bez wprowadzania błędów do modułu pozwalającego na preprocessing zdjęć zostały dodane testy jednostkowe. Zostały napisane przy pomocy modułu PyTest a ich uruchamianie jest obsługiwane, przez moduł Tox. Testy są automatycznie uruchamiane przy wypchnięciu lokalnych migawek (ang. commit), do publicznego repozytorium na platformie GitHub. Funkcjonalność ta została zrealizowana za pomocą platformy CircleCi, pozwala ona na uruchomienie testów po otrzymaniu informacji od portalu GitHub. Status tych testów możemy zobaczyć w łatwy sposób poprzez dodany do pliku README.md obraz. Jest on każdorazowo przy wyświetlaniu wyrenderowanego pliku README pobierany z serwera CircleCi i zależnie od tego czy wszystkie testy jednostkowe z ostatniej migawki zostały poprawnie ukończone czy nie przybiera odpowiednio kolor zielony/czerwony.

7.6 Automatyczne renderowanie i publikacja wyników

W celu prezentacji wyników oraz łatwego i interaktywnego badania ich powstał notatnik Jupyter. W celu interaktywnego przejrzenia go należy pobrać repozytorium a następnie zainstalować wymagane biblioteki oraz uruchomić serwer jupyter za pomocą polecenie "jupyter notebook". Jednakże jest on również dostępny jako strona internetowa w postaci nie interaktywnej (z pokazanymi wyjściami z wszystkich komórek). Strona ta jest automatycznie uaktualniana za każdym razem gdy do repozytorium na serwerze GitHub zostanie wysłana nowa migawka zawierająca modyfikację notatnika. Rozwiązanie to tak samo jak automatyczne testy jednostkowe zostało zrealizowane przy użyciu platformy

CircleCi. Otrzymuje ona informację od serwisu GitHub o nowej migawce a następnie sprawdza, czy został w niej zmieniony plik notatnika. Jeżeli tak to wywołane zostanie polecenie eksportujące go do pliku w formacie HTML a następnie platforma stworzy nową migawkę (jako wiadomość przesłany jest ciąg znaków, który sygnalizuje, że testy jednostkowe nie powinny być dla niej uruchamiane, ponieważ stało się to już przy migawce wywołującej konwersję) i wyśle ją do repozytorium na serwisie GitHub. Gdy ten go otrzyma opublikuje nowo utworzony plik na platformie GitHub Pages. Dzięki tej integracji wszystkie zmiany których dokonam w notatniku prezentującym wyniki moich prac są automatycznie publikowane i dostępne do wglądu.

7.7 Kontener developerski

Stworzyłem również skrypty, które pozwalają na uruchomienie kontenera (przy użyciu technologii Docker) zawierającego wszystkie wymagane biblioteki.

Rozwiązanie to w szczególności w znaczący sposób upraszcza dostęp do bibliotek firmy NVIDIA wymaganych do uruchomienia biblioteki TensorFlow na karcie graficznej tegoż producenta. Dzięki temu osoba zainteresowana uruchomieniem aplikacji z wykorzystaniem karty graficznej w celu przyśpieszenia obliczeń musi jedynie posiadać na swoim urządzeniu aktualne sterowniki oraz Docker.

Skrypt przyjmuje na wejście ścieżkę do katalogu, który zawiera kod programu. Jest on montowany do kontenera. Dzięki czemu zmiany mogą być dokonywane zarówno w kontenerze jaki poza nim.

8 Podsumowanie i możliwe następne kroki

Testy, które przeprowadziłem wskazują na to, że dla tych danych wejściowych/wyjściowych zastosowanie architektury U-Net nie sprawdza się. Jako, że jest to najbardziej rozpowszechniona architektura do segmentacji obrazów medycznych może to również sugerować, że użycie sieci neuronowych do tego zadania na ten moment nie przyniesie żądanych wyników.

Jednakże dzięki powstałym w trakcie testów narzędziom pojawiła się możliwość łatwego testowania nowych podejść, które jak wskazują na to ostatnie lata pojawiają się w dziedzinie uczenia maszynowego i głębokich sieci neuronowych bardzo często.

Dobrym tropem może być również przedstawienie obrazów wyjściowych za pomocą techniki zwanej mapy odległości (ang. distance map). Polega ona na tym aby wartość każdego piksela obrazu była jego odległością od najbliższej krawędzi komórki. W wypadku gdy piksel znajduję się wewnątrz komórki do jego odległość oznaczamy jako liczbę ujemną. Następnie dla obrazu wejściowego obliczamy predykcję i na podstawie otrzymanej mapy odległości generujemy obraz ustawiając piksele o wartości zero na czerwone a pozostałe na kolor czarny.

9 Bibliografia

Literatura

[1] Bruce Blaus. https://en.wikipedia.org/wiki/neuron.

[2] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, 2018.

[3] Harshall Lamba. https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47, 2021.

[4] Md Nasir Uddin Laskar. https://nasirml.wordpress.com/2019/01/08/convnet-in-tensorflow/, 2021.

[5] A. Rutkowski M. Czoków, J. Piersa. Wstęp do sieci neuronowych., 2021.

[6] Mayranna. https://pl.wikipedia.org/wiki/perceptron.

[7] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[8] Sumit Saha. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53, 2021.