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, nie wymagające sześcioletnich studiów. Nadaje się ono idealnie do automatyzacji. W ostatnich latach bardzo mocno rozwija się użycie uczenia maszynowego do rozwiązywania tej klasy problemów.

W ramach mojej pracy powstał zestaw narzędzi pozwalający na łatwe testowanie różnych architektur oraz funkcji strat dla sieci neuronowych w tym problemie. Rozwiązanie zostało napisane w języku python z użyciem biblioteki Keras (służącej do pracy z sieciami neuronowymi) będącej interfejsem do biblioteki TensorFlow (pozwalającej na operacje na tensorach). Dodatkowo wykorzystałem biblioteki NumPy do obliczeń macierzowych, Pillow do manipulacji obrazów oraz Jupyter Notebook do interaktywnej prezentacji wyników. Kod źródłowy projektu jest wersjonowany z użyciem narzędzia git oraz upubliczniony na platformie GitHub. Repozytorium zawierające kod projektu zostało podłączone do platformy CircleCi w celu automatycznego uruchamiania testów jednostkowych (napisane przy użyciu PyTest i Tox) oraz publikacji wyników. W ramach pracy powstały również skrypty w jeżyku bash tworzące kontener deweloperski (technologia Docker) zawierający wszystkie niezbędne biblioteki.

Dodatkowo przeprowadziłem testy dla architektury u-net oraz pewnych wariacji na jej temat. Jako, że zdjęcia tego typu są danymi chronionymi prawnie to szpitale, nie mogą wysłać ich poza własną sieć wewnętrzną. Z tego powodu przetestowałem również możliwość uruchomienia splotowej sieci neuronowej w przeglądarce z użyciem TensorFlowJS. Co pozwala aby wszystkie operacje były wykonane po stornie użytkownika końcowego a dane medyczne nie musiały być wysyłane na serwer.

Całość pracy dopełnia moduł pozwalający na obróbkę zdjęć wejściowych oraz wyjściowych.

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 dostarczonego mi, przez dr. Wiśniewskiego. 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 przyjmuje kolor czarny.

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, to dopiero od niedawna posiadamy wystarczającą moc obliczeniową aby realizować tworzone od wielu dziesięcioleci algorytmy.

2.1 Ogólne zasady działania

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.

Omówienie zaczniemy od tego w jaki sposób sieć przeprowadza obliczenia zwracające odpowiedź, a następnie zajmiemy się procesem uczenia. Neurony kolejnych warstw są ze sobą połączone (zajmę się tu jedynie warstwami, które posiadają połączenia neuronów każdy z każdym, czyli tak zwanymi warstwami typu "dense"). Połączenia te posiadają wagi. Aby obliczyć wynik zwracany przez sieć rozpoczynamy obliczenia od warstwy wejściowej i obliczmy wartości neuronów w kolejnych warstwach. Wartość dla neuronu w warstwie k-tej to wartość funkcji aktywacji na sumie wartości wszystkich neuronów z warstwy poprzedniej przemnożonych przez odpowiednie im wagi połączeń.

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

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 wyjściowe otrzymujemy odpowiedź naszego modelu.

Proces uczenia to wielorazowe 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.

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.

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.

4 Problem segmentacja

Zaznaczenie konturów komórek na zdjęciach mikroskopowych jest problemem segmentacji instancji. 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.

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.

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.

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.)

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.

5 Architektura U-Net historia i zastosowania

6 Przetestowane podejścia

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

6.1 Bazowa sieć u-net

Pierwszym podejściem, które przetestowałem było uruchomienie trzy poziomowej (dziesięcio warstwowej) sieci neuronowej w architekturze u-net. Sieć otrzymywała na wejście obrazy rozmiaru 200x200x3 a jako wyjście zwracała obrazy 200x200x4. Na tym etapie przeprowadziłem testy modyfikując opisujące te sieć hiperparametry. Zmieniałem liczbę epok, stałą uczącą oraz wielkość batchy. 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 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. Natomiast drugim był niedobór danych uczących. Doszedłem do wniosku, że oba te problemy mogę rozwiązać poprzez pocięcie dużych obrazów wejściowych (1200x1600px) na wiele mniejszych (90x90px z zakładką wynoszącą 10px), a następnie uczenie na nich modelu. 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 90x90px 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 20 i wysokości 15 mniejszych obrazów).

6.2.2 Wybór jednego kanału - czerwony/alfa

Analizując dane wyjściowe zauważyłem, że piksele czarne (tło) posiadają wartości kanału alpha bliskie zeru natomiast piksele linii wartości bliskie dwustu pięć dziesięciu 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 z białymi liniami na czarnym tle.

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.

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.

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 $1 - f(x)$ gdzie f(x) to współczynnik Sørensena.

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).

6.4 Zapis wyniku w postaci distance map

6.5 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 (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 python-owym 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.

7 Powstałe narzędzia

W trakcie prac powstał szereg narzędzi znacznie ułatwiających pracę nad tym zagadnieniem. Kod wraz z historią jest dostępny na platformie GitHub. Wyniki są prezentowane za pomocą platformy GitHub Pages i są dostępne pod adresem:.

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 Gaussian Blur 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

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.

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 commitów, 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 ostatniego commita zostały poprawnie ukończone czy nie przybiera odpowiednio kolor zielony/czerwony.

7.6 Automatyczne renderowanie i publikacja wyników

Dodatkowo powstał notatnik Jupyter[1] zawierający prezentację uzyskanych wyników. W celu interaktywnego przejrzenia go należy pobrać repozytorium. Jednakże jest on również dostępny jako strona internetowa w postaci z pokazanymi wyjściami z wszystkich komórek. Strona ta jest uaktualniana za każdym razem gdy do repozytorium na serwerze GitHub zostanie wysłany nowy commit zawierający modyfikację notatnika. Rozwiązanie to tak samo jak automatyczne testy jednostkowe zostało zrealizowane przy użyciu platformy CircleCi. Otrzymuje ona informację od serwisu GitHub o nowym commicie następnie sprawdza, czy plik notatnika został zmieniony. Jeżeli tak to wywoła polecenie eksportujące go do pliku w formacie HTML a następnie sama stworzy commit i wyśle go 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.

[1] interaktywne środowisko uruchomieniowe. Pozwalające na łączenie kodu oraz języka znaczników markdown. Dostępne pod adresem: https://jupyter.org/

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.

9 Bibliografia