SIECI NEURONOWE: KRÓTKI* TUTORIAL

Autor: Radosław Łazarz

Sieci neuronowe to struktury algebraiczne luźno inspirowane strukturą ludzkiego mózgu. Odpowiednio zaprojektowane i poddane procedurze uczenia stają się potężnym narzędziem pozwalającym na modelowanie złożonych procesów i predykcję ich rezultatów.

ROZDZIAŁ PIERWSZY: THROUGH THE LOOKING GLASS...

PRZYGOTOWANIE ŚRODOWISKA DO DALSZEJ PRACY

UWAGA! Rozpocznij przygotowywanie środowiska przed przeczytaniem reszty dokumentu (będą „przestoje", np. na pobieranie/indeksowanie bibliotek).

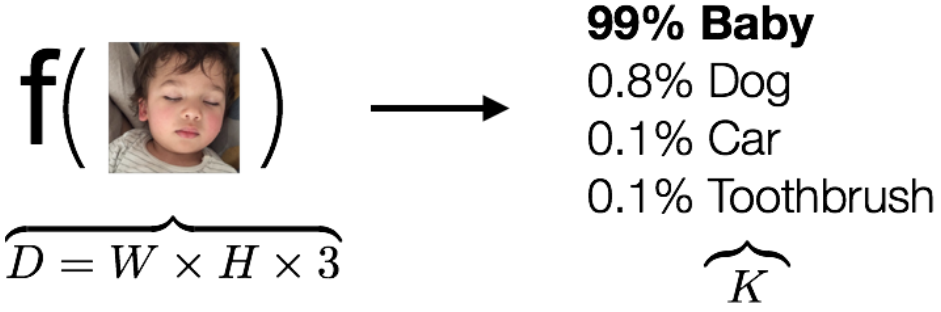
- Jeżeli pracujesz na własnym sprzęcie będziesz potrzebował interpretera języka Python oraz następujących bibliotek: tensorflow, keras, matplotlib, pillow.
- Jeżeli korzystasz z PCoIP zaloguj się na maszynę „Systemy ekspertowe i sztuczna inteligencja".
  - Interpreter języka Python wymieniony w zmiennej środowiskowej Path posiada zainstalowane wszystkie potrzebne biblioteki.
  - Jeżeli chcesz korzystać z IDE PyCharm, to pamiętaj by tworząc nowy projekt wybrać dla niego ten właśnie interpreter (Project Interpreter → rozwiń → Existing interpreter → trzykropek → System Interpreter → trzykropek → znajdź właściwy na dysku). Poczekaj chwilę na ukończenie indeksowania.
  - Wykorzystywane maszyny wirtualne należą do tzw. puli pływającej – w związku z czym nie są czyszczone natychmiast po wylogowaniu. Przed zakończeniem pracy warto posprzątać po sobie i usunąć wszystkie pobrane lub stworzone pliki!

PROBLEM KLASYFIKACJI OBRAZÓW

Wprowadzenie zaczniemy od problemu klasyfikacji obrazów. Pozornie jest on bardzo prosty: przyporządkuj wejściową fotografię do jednej z kilku zadanych kategorii („pies", „małpa", „samolot", itp.).

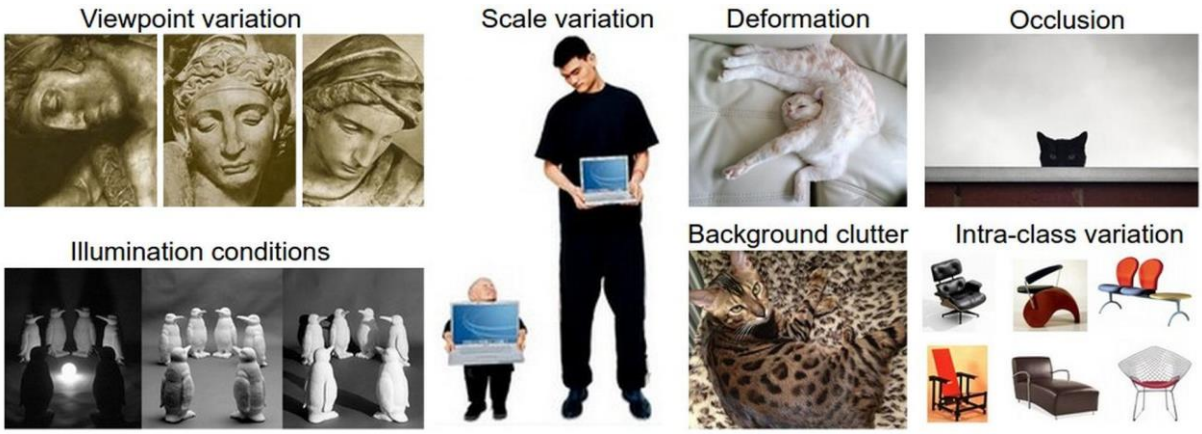
Jego bardziej precyzyjna definicja wygląda następująco. Kolorowe zdjęcie (RGB), szerokie na **W** pikseli i wysokie na **H** pikseli może być reprezentowane w pamięci komputera jako tablica **D = W * H * 3** liczb z zakresu od 0 (minimalna jasność) do 1 (maksymalna jasność. Załóżmy również, że mamy **K** kategorii do których chcemy zaklasyfikować nasz obraz. Naszym zadaniem jest więc znalezienie funkcji, która przyjmie tą tablicę jako wejście i zwróci etykietę właściwej kategorii.

Co jeżeli takie jasne przyporządkowanie nie jest jednak wykonalne? Prostym rozwiązaniem jest rozszerzenie naszej funkcji w ten sposób, by dla każdej kategorii zwracała % prawdopodobieństwa takiego rozwiązania – pozwoli nam to podjąć decyzję jak również zweryfikować jak bardzo możemy na niej polegać. Co do zasady potrzebujemy więc znaleźć funkcję $\mathbf{f} : \mathbb{R}^D \mapsto \mathbb{R}^K$.



* To niestety kłamstwo. Mam nadzieję, że jedyne zawarte w tym dokumencie.

Jak się za to zabrać? Naiwnym podejściem byłaby próba ręcznej specyfikacji pewnych cech (niemowlęta mają duże głowy, szczoteczki są długie, etc.). Szybko jednak stwierdzilibyśmy, że nawet dla niewielkiego zbioru kategorii jest to tytaniczna praca bez gwarancji sukcesu. Co więcej, istnieje wiele czynników zniekształcających zawartość naszych zdjęć. Obiekty mogą być przedstawiane z różnych ujęć, w różnych warunkach oświetleniowych, w różnej skali, częściowo niewidoczne, ukryte w tle…



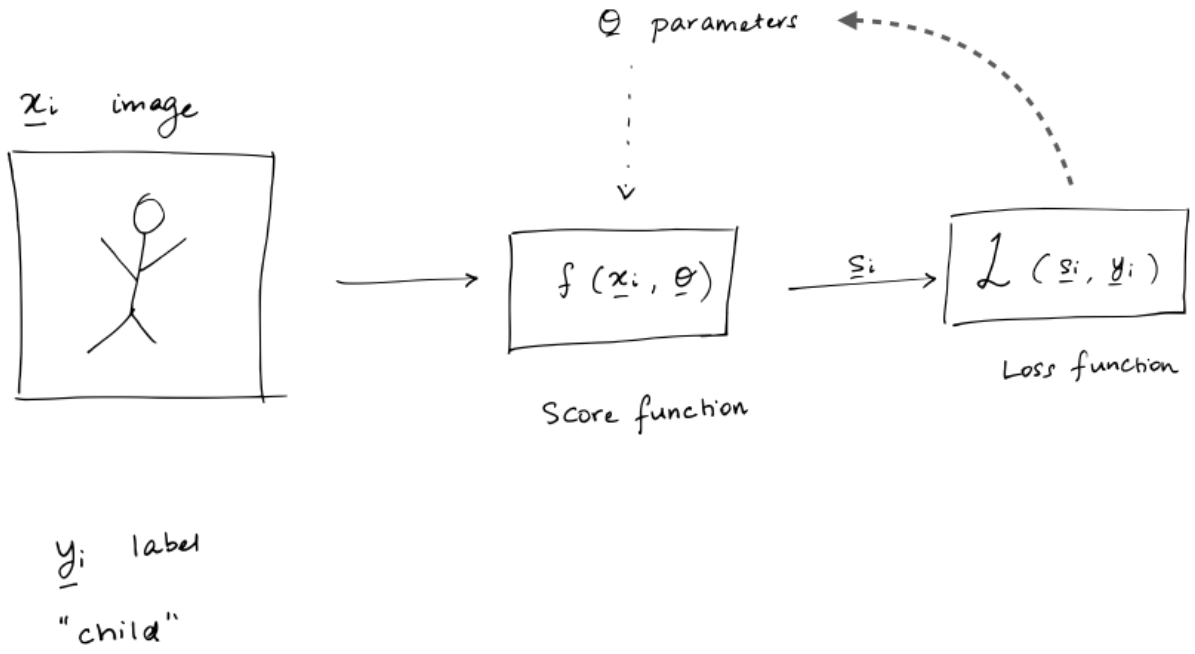
Wszystkie wymienione problemy są skutkiem istnienia semantycznej przepaści między tym jak reprezentowane są nasze dane wejściowe (tablica liczb), a tym czego w nich szukamy (kategorii i cech: zwierząt, nosów, głów, itp.). Zamiast więc próbować samodzielnie napisać funkcję **f** spróbujemy skorzystać z dobrodziejstw uczenia maszynowego by automatycznie skonstruować reprezentację wejścia właściwą dla postawionego sobie zadania.

UCZENIE Z NADZOREM

Skorzystamy w tym celu z poddziedziny uczenia maszynowego zwanej uczeniem z nadzorem – obecnie najpopularniejszego podejścia do uczenia się na podstawie zbiorów danych. Można je opisać w następujących krokach.

- Startujemy posiadając zbiór obrazów o znanych wcześniej klasach (np. przyporządkowanych przez ludzkiego eksperta, tytułowego nadzorcę). Zbiór ten będziemy nazywać zbiorem treningowym (training data), stanowić on będzie bazę wiedzy do przyszłego uczenia systemu.
- Funkcja **f**, którą próbujemy znaleźć jest zazwyczaj zwana funkcją oceny (score function). Zakładamy z góry, że na jej działanie będzie miał wpływ szereg parametrów **θ**.
- Wprowadzamy również dodatkową funkcję **L**, tak zwaną funkcję straty (loss function). Będzie ona opisać jak bardzo wyniki zwracane przez funkcję **f** są zgodne z naszymi założeniami i oczekiwaniami. Duże wartości funkcji **L** oznaczają, iż funkcja **f** nie działa zbyt dobrze.
- Na koniec uruchamiamy algorytm optymalizujący, często nazywany po prostu algorytmem uczącym. Jego zadaniem jest iteracyjne poprawianie wartości **θ** na bazie przykładów uczących tak, by zapewnić jak najniższe **L**.
- Kiedy proces się zakończy, mamy nadzieję że uzyskana funkcja **f** ma wysoką zdolność do generalizacji – to jest, działa porównywalnie dobrze dla danych wejściowych z których nie korzystaliśmy uprzednio (tzw. test data).

Większość wyzwań współczesnego uczenia maszynowego sprowadza się do wyboru jak najlepszych narzędzi realizujących powyższe kroki. W tym tutorialu będziemy sukcesywnie przechodzić od tych najbardziej trywialnych do nieco bardziej skomplikowanych, po drodze poprawiając uzyskiwane rezultaty.



Warto jeszcze zwrócić uwagę na to, jaką formę może mieć opis właściwej, wyznaczonej przez eksperta klasy ($\mathbf{y_i}$). Planujemy porównywać go z oceną (score) $\mathbf{s_i}$, będącą wektorem % szans na przynależenie do danej klasy. W tej sytuacji najprościej by $\mathbf{y_i}$ był wektorem zawierającym zera na wszystkich pozycjach poza jedną, gdzie pojawi się jedynka (oznaczająca 100%). Taką reprezentację nazywa się często „one hot encoding".

KLASYFIKATOR LINIOWY

Niech nasze wejściowe zdjęcie będzie reprezentowane przez tablicę (wektor) **x**. Najprostszym przykładem szukanej funkcji **f** jest funkcja liniowa

$$\mathbf{f}(\mathbf{x}; \mathbf{W}, \mathbf{b}) = \mathbf{W}\mathbf{x} + \mathbf{b},$$

w przypadku której parametry **θ** to wagi **W** i bias **b**.

Pojawia się tu jednak pewien drobny problem. Produkowane przez nią wyniki nie zawsze mieszczą się w zakresie od 0 do 1 (oraz niekoniecznie sumują do 100%) i trudno byłoby je interpretować jako prawdopodobieństwa. By poradzić sobie z tym mankamentem przepuścimy je przez dodatkową funkcję **σ**, zwaną funkcją softmax (miękkie maksimum: zamiast wybierać jeden wynik oblicza % prawdopodobieństwa), która sprowadzi je do oczekiwanego zakresu (zakładamy, że funkcja **f** to tak naprawdę zbiór podfunkcji $\mathbf{f_j}$ – po jednej dla każdej z rozważanych klas).

$$s_j = \sigma(\mathbf{f})_j = \frac{e^{f_j}}{\sum_{k=1}^{K} e^{f_k}}$$

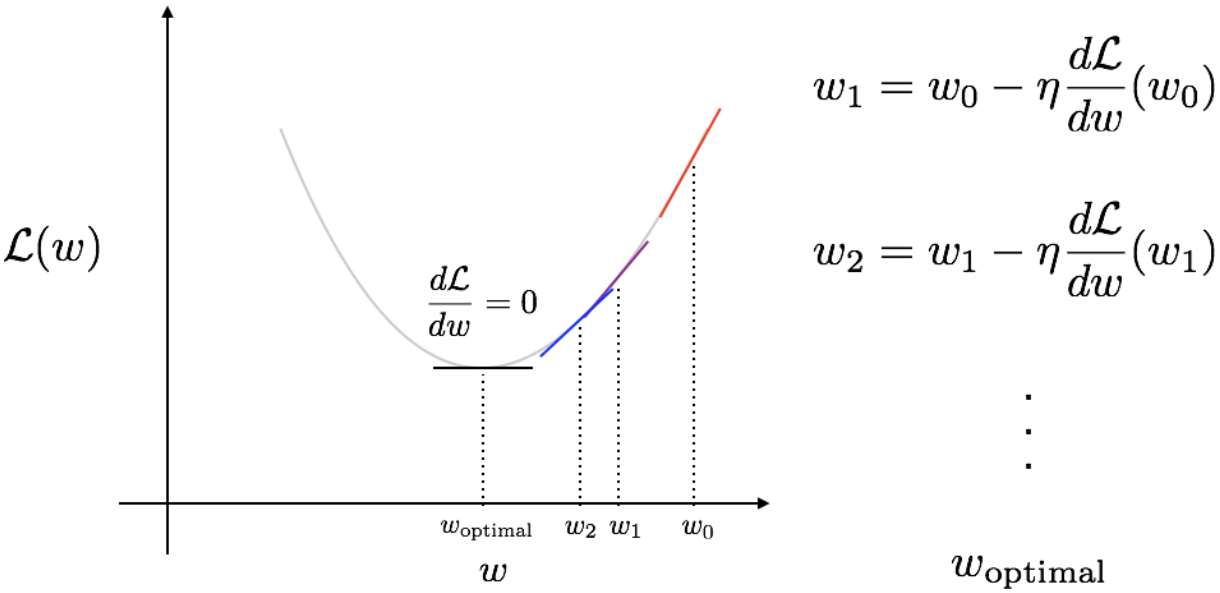
Jak teraz zdefiniować stratę (loss) **L**, która oceni jakość takiego **f**? Zakładając, że dla każdego z treningowych zdjęć $\mathbf{x_i}$ właściwą kategorią jest $\mathbf{y_i}$ (tu rozumiane jako indeks właściwej kategorii) jednym z rozwiązań jest policzenie tzw. entropii krzyżowej (cross entropy).

$$\mathcal{L}(\mathbf{s}) = -\sum_i log(s_{y_i})$$

Jeżeli dobrze przyjrzymy się powyższemu wzorowi, to zauważymy że im bliżej wynik odpowiedniego **f** jest do oczekiwanego, tym bliższy 0 jest jego przyczynek do funkcji straty ($\mathbf{s_{yi}}$ powinno być jak najbliższe 1 – bo to właściwa klasa).

Pamiętaj, że nie sumujemy wszystkich składowych kolejnych wektorów **s**, a tylko te, które miały być zapalone. Postępowanie odwrotne to typowy błąd wstrzymujący proces uczenia (**L** nie zależałoby wtedy w żaden sposób od tego, jak podobne są uzyskane wyniki do oczekiwanych)!

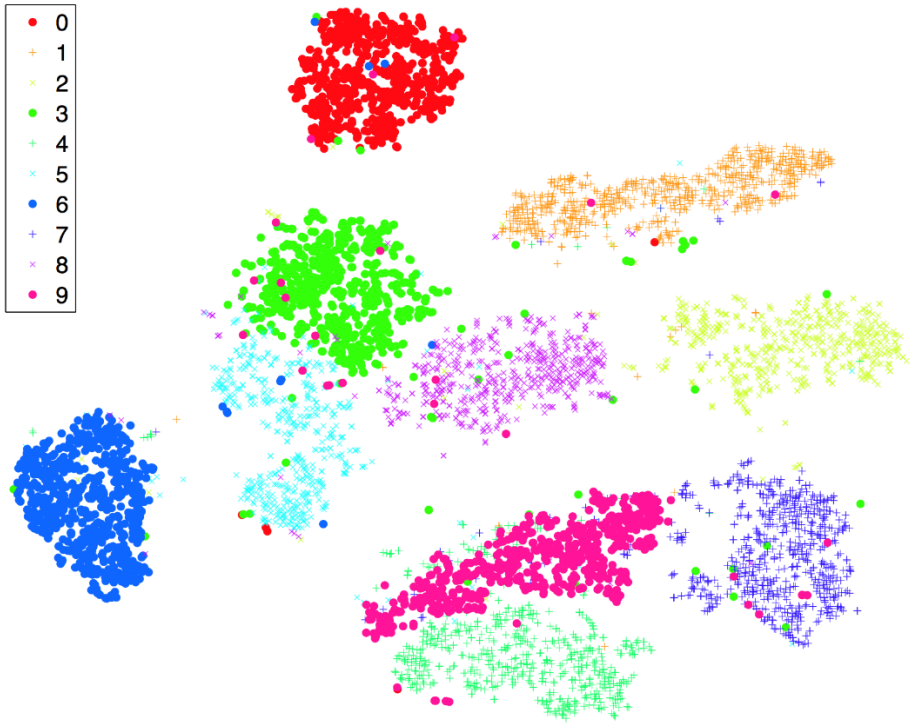
Ostatnim krokiem jaki nam teraz pozostał jest optymalizacja parametrów funkcji **f**, tak by zminimalizować **L**. Istnieje wiele technik minimalizacji, my skorzystamy z jednej z najpopularniejszych: stochastycznego spadku po gradiencie. Nie zagłębiając się w szczegóły (choć warto to zrobić, jeżeli jest się osobą mocniej zainteresowaną tematem) sprowadza się on do poruszania w przestrzeni parametrów małymi krokami w kierunku największego spadku (gradientu), tak jak na tym filmie: https://www.youtube.com/watch?v=kJgx2RcJKZY. Gdyby **L(f(θ))** miała tylko jeden parametr dobrze ilustrowałby to również poniższy przykład.



Ponieważ nie wiemy jaki zestaw parametrów jest tym właściwym, optymalizację zaczniemy od ich losowego wyboru (w praktycznych realizacjach jest to nieco bardziej złożone, niepotrzebnie komplikowałoby jednak przykład). Stała **η** używana na rysunku powyżej określa tzw. tempo uczenia się (learning rate) – długość kroków jakie wykonujemy szukając optimum.

ĆWICZENIE 1 – LINIOWY KLASYFIKATOR CYFR

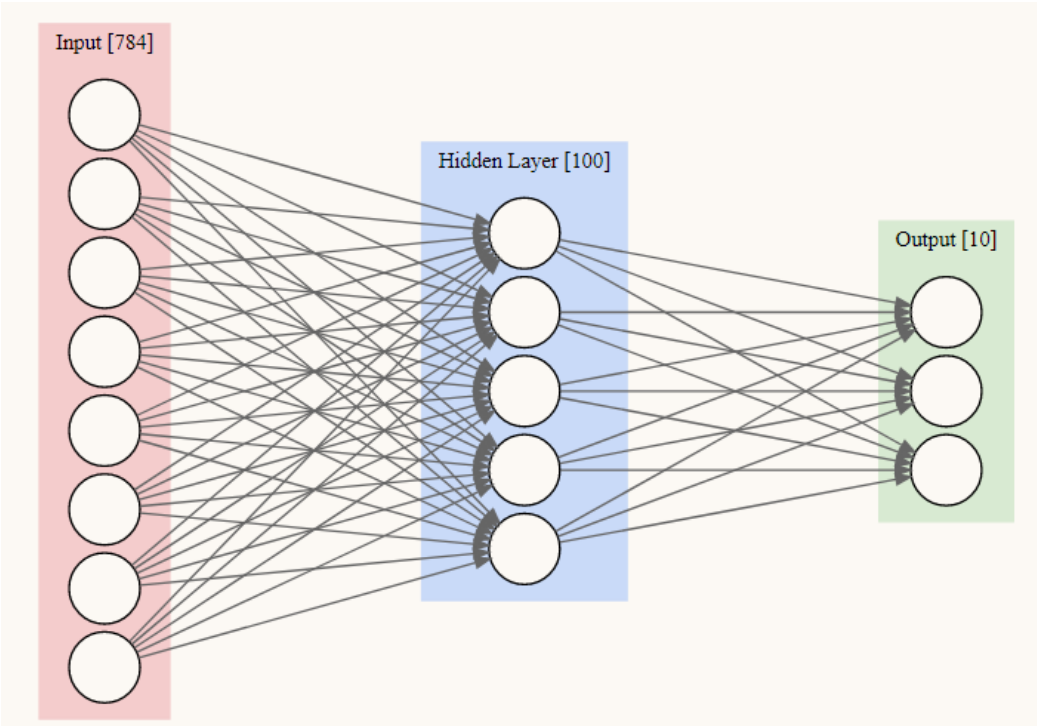
Wymienione wyżej składowe (funkcja liniowa opakowana w funkcję softmax, funkcja entropii krzyżowej i stochastyczny spadek po gradiencie) pozwolą nam na nauczenie naszego pierwszego klasyfikatora! Skocz teraz do załączonego pliku .py i wykonaj intro oraz pierwsze ćwiczenie. Nasza (na razie jednowarstwowa) sieć będzie się uczyć rozpoznawania ręcznie pisanych cyfr ze zbioru MNIST – jego kształt możemy zobaczyć na wizualizacji poniżej – większość cyfr jest łatwa od odróżnienia od pozostałych, ale niektóre mogą być zwodnicze. Niektóre cyfry są też bardzo zbliżone do innych (np. 9 i 4, z intuicyjnych powodów). Wizualizacja została sporządzona poprzez wykorzystanie algorytmu t-SNE (który nie jest tematem dzisiejszych zajęć, ale warto go znać).



Jeżeli wszystkie kroki wykonano poprawnie, to taki prosty klasyfikator powinien uzyskać zupełnie przyzwoitą skuteczność rozpoznawania cyfr (a na pewno znacznie większą niż losowe zgadywanie jednej z dziesięciu).

ĆWICZENIE 2 – PROSTA SIEĆ NEURONOWA

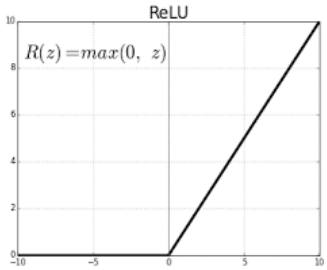
Wynik z poprzedniego ćwiczenia pozostawia wiele do życzenia (zbiór MNIST nie jest trudnym zbiorem!)… Jak go poprawić? Po pierwsze zamienimy naszą jednowarstwową liniową sieć w taką, która posiada ciekawszą, bardziej złożoną architekturę. Jaką? Taką jak na schemacie poniżej.



Zauważmy, że pojawiła się nowa warstwa – tzw. warstwa ukryta. Każdy z widocznych powyżej neuronów to nadal funkcja liniowa, wraz ze swoimi wagami i biasem. Różnica polega na tym, że neurony końcowe nie korzystają z danych wejściowych, a z tego co zwraca warstwa poprzednia.

Choć wydaje się, że taka sieć powinna mieć znacznie większą siłę wyrazu…to tak naprawdę nic jeszcze nie uzyskaliśmy! Gdyby korzystając z zasad algebry liniowej uprościć wykonywane operacje okazałoby się, że powyższa sieć jest dokładnie równoważna takiej zwykłej, jednowarstwowej (przemnożone przez siebie macierze wag dają macierz wynikową).

By skorzystać z dodatkowej warstwy sieci i zwiększyć jej siłę wyrazu musimy znów zmodyfikować wyniki funkcji liniowej i przetworzyć je przez inną, nieliniową funkcję. Funkcję tą zwiemy zwyczajowo funkcją aktywacji. Popularnym przykładem takiej funkcji jest funkcja ReLU, o przebiegu pokazanym poniżej.



Nasza sieć będzie się więc zachowywać następująco:

- bierze wektor wejściowy **x**
- przemnaża go przez funkcje liniowe pierwszej warstwy $\mathbf{y_1 = W_1x + b_1}$
- wynik ten przetwarza nieliniową funkcją ReLU $\mathbf{h_1 = max(0, y_1)}$
- te wyniki przechodzą przez funkcje liniowe warstwy drugiej $\mathbf{y_2 = W_2x + b_2}$
- na końcu zaś są zamieniane w prawdopodobieństwa z użyciem funkcji softmax $\mathbf{s = \sigma(y_2)}$

Zmodyfikuj kod z ćwiczenia drugiego tak, aby spełniał powyższe założenia i sprawdź wyniki.

ĆWICZENIE 3 – ROZWIĄZUJEMY PROBLEM Z AKTYWACJĄ

Uzyskany wynik okazał się być niedorzecznie niski, bliski 10% (odpowiednik losowego zgadywania)? To normalne! Co właściwie poszło nie tak? Wszystkie wagi i biasy w naszej sieci były inicjalizowane wartością 0. W przypadku dwuwarstwowej sieci z funkcją ReLU takie wartości zostały również przestałe na kolejną warstwę (**max(0, 0) = 0**). Ta sytuacja uniemożliwiła sieci wykorzystywanie algorytmu stochastycznego spadku po gradiencie (druga warstwa otrzymywała zawsze takie samo zerowe wejście, niemożliwe było policzenie odpowiednich pochodnych), a więc efektywnie zablokowany został proces uczenia.

Aby rozwiązać ten problem należy zmienić startowe wagi i biasy tak, by zamiast 0 rozpoczynały od losowych niewielkich wartości (np. między -0.1 a 0.1). Udaj się do ćwiczenia 3 i dokonaj tej modyfikacji. Teraz wynik powinien być lepszy niż kiedykolwiek!

ĆWICZENIE 4 – CZY MUSI BYĆ TAK CIĘŻKO?

Wszystkie dotychczasowe ćwiczenia wykonywaliśmy z użyciem biblioteki TensorFlow (wiodącego frameworku do wydajnego wykonywania operacji tensorowych). Pozwala on na pełną kontrolę wszystkich szczegółów danego procesu obliczeniowego – konsekwencją jest jednakże obszerny kod, w którym łatwo o błędy (konieczność dbania o odpowiednie wymiary tensorów i ich zawartość, efekt znany dopiero po zbudowaniu całego grafu obliczeń, niekiedy długie oczekiwanie na wyniki).

W wielu sytuacjach jest to klasyczne „wynajdywanie koła na nowo”. Na szczęście obecnie istnieją już także dobrze zaprojektowane biblioteki operujące na znacznie wyższym poziomie abstrakcji. Jedną z najpopularniejszych i uznawanych za najbardziej intuicyjne jest Keras – od niedawna będący również oficjalnym wysokopoziomowym API dla samego TensorFlow (choć wspierający również inne backendy).

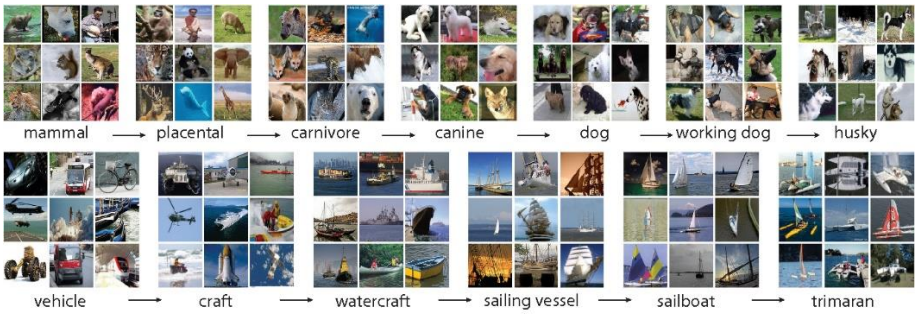
Naszym zadaniem jest skorzystać z zamieszczonych w komentarzach wskazówek i odtworzyć w Kerasie sieć zaimplementowaną wcześniej w ćwiczeniu 3. Tym razem powinno pójść o wiele łatwiej! Warto skorzystać z kultowego ekspresowego tutoriala do Kerasa (https://keras.io/#getting-started-30-seconds-to-keras).

ĆWICZENIE 5 – WYKORZYSTYWANIE JUŻ WYTRENOWANYCH SIECI

Zakres tego laboratorium jest ograniczony – więc powyższe kroki były tylko powierzchownym wstępem do sieci neuronowych i ich architektury. Współczesne rozwiązania korzystają z dziesiątek warstw o różnych sposobach działania, są pretrenowane, odwołują się do siebie rekurencyjnie i wdrażają wiele innych pomysłów, które pojawiły się na przestrzeni ostatnich dekad.

Nasz (tymczasowy) brak wiedzy (i co niestety ważniejsze - mocy obliczeniowej) nie oznacza jednak, że nie możemy w ogóle korzystać z ich dobrodziejstw. Ostatnie ćwiczenie pokazuje jak pobrać dużą, już wyuczoną sieć rozpoznającą elementy zbioru ImageNet (przykłady na zdjęciu poniżej). Sprawdź jej skuteczność w praktyce! Wykonaj polecenia zawarte w komentarzach.

Kolejne etapy tego ćwiczenia pozwalają (orientacyjnie) zaznajomić się z jej budową i działaniem. Będą też świetnym wstępem przez kolejnym rozdziałem – zadaniem domowym.



CHCESZ WIĘCEJ?

Zacznij przerabiać dostępne w sieci tutoriale! Choćby

- https://www.superdatascience.com/blogs/the-ultimate-guide-to-convolutional-neural-networks-cnn
- czy https://www.analyticsvidhya.com/blog/2018/12/guide-convolutional-neural-network-cnn/ .

Uczenie maszynowe oparte o sieci neuronowe jest obecnie niezwykle popularnym tematem – w źródłach wiedzy można wręcz przebierać.

ROZDZIAŁ DRUGI: ...AND WHAT ALICE FOUND THERE

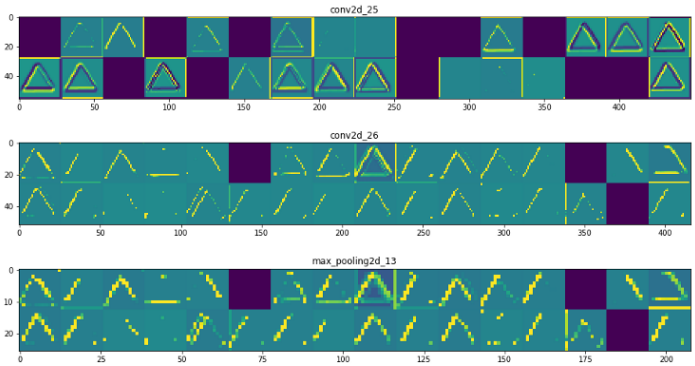
Celem rozdziału drugiego jest poznanie bliżej dużych, współczesnych sieci konwolucyjnych – i wykorzystanie ich do kilku nowych eksperymentów (prawdopodobnie nieco bardziej ekscytujących, niż rozpoznawanie cyfr). Ze względu na nieco większą dowolność (i poziom komplikacji) będą one razem tworzyć zadanie domowe. Dowiemy się przykładowo jak zrealizować pokazaną poniżej transformację.

Jeżeli istnieje taka możliwość zalecana jest instalacja biblioteki TensorFlow w wersji tensorflow-gpu! Zadania domowe są w pełni realizowalne w oparciu o CPU, ale obliczenia będą wtedy trwały zauważalnie dłużej. W tym drugim przypadku dobrym pomysłem jest zajęcie się nimi odpowiednio wcześniej (by mieć zapas czasu na uzyskanie finalnych wyników), cierpliwość (polecane zostawienie obliczeń „na noc", szczególnie w Zadaniu Domowym 3) oraz użycie obrazów o mniejszych rozmiarach.

ZADANIE DOMOWE 1 – AKTYWACJE W SIECIACH GŁĘBOKICH

Jednym z odwiecznych problemów przy interakcjach z sieciami głębokimi jest próba "intuicyjnego" zrozumienia ich działania oraz sensu wyuczonego modelu. Najbliższe ćwiczenia zaprezentują proste sposoby spojrzenia "do wnętrza sieci".

Ostatnie ćwiczenie z pierwszego rozdziału zawierało fragment kodu odpowiedzialny za wyświetlanie przykładowej aktywacji w jednym kanale jednej warstwy sieci na skutek podania na jej wejście zdjęcia z nosaczem. Celem tego ćwiczenia jest przygotowanie programu, który wygeneruje szerszą wizualizację, zawierającą wiele przykładów aktywacji ze wszystkich warstw danej sieci. Kawałek przykładowej wizualizacji tego typu zaprezentowany jest poniżej.



Postaraj się, by wykonana implementacja była uniwersalna (użyjemy jej potem do analizy innych sieci). Sprawdź aktywacje dla różnych fotografii wejściowych. Co obserwujesz? Czym różnią się poszczególne warstwy? Za co prawdopodobnie odpowiadają?

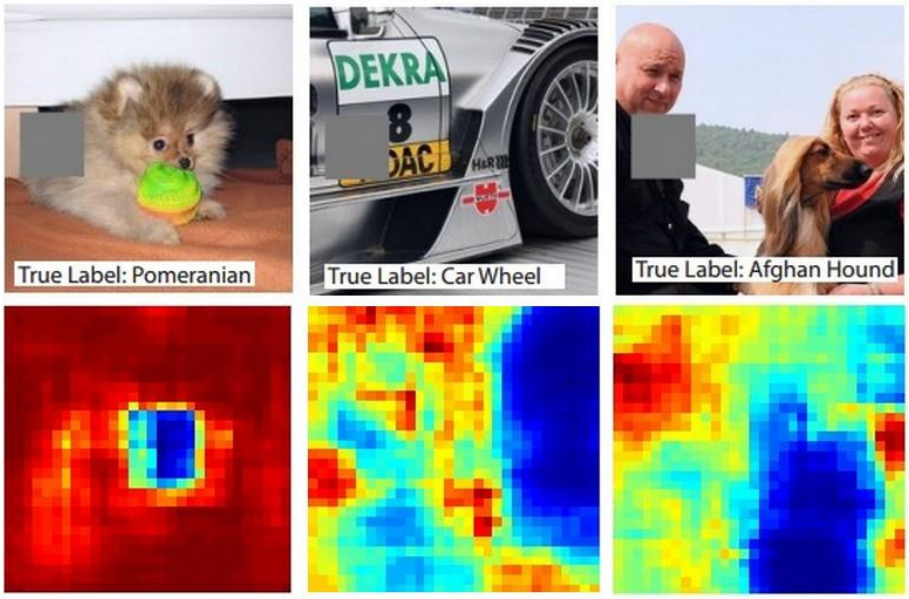
ZADANIE DOMOWE 2 – CO WIDZI SIEĆ, GDY WIDZI BIAŁEGO KRÓLIKA?

W środowiskach zajmujących się uczeniem maszynowym od lat krąży miejska legenda opisująca początki tej dyscypliny. Zgodnie z nią, w czasach zimnej wojny Pentagon zlecił przygotowanie sztucznej inteligencji, która na polu walki potrafiłaby odróżnić z daleka czołgi NATO od tych wykorzystywanych przez Układ Warszawski, dając przewagę rodzimym czołgistom. Model był długo trenowany w oparciu o dostarczone dane i po jakimś czasie zaczął dawać naprawdę obiecujące wyniki. Podekscytowani generałowie natychmiast rozpoczęli tajne nocne testy na próbnym poligonie, które...skończyły się spektakularną porażką. AI rozpoznawała wszystkie maszyny jako wrogie i sugerowała ostrzał. Skąd taka nagła zmiana? Otóż w zbiorze uczącym wszystkie fotografie własnych czołgów zrobiono za dnia, w dobrych warunkach – wrogich zaś nocą, podczas działań szpiegowskich. Tranzystorowy mózg nauczył się więc jedynie rozróżniać dzień od nocy…

Choć historia jest oczywiście przesadzona, to podobne wątpliwości zdarzają się na co dzień. Czy moja sieć naprawdę nauczyła się po czym odróżnić panterę od ocelota? Czy może korzysta z jakiegoś tricku (jedno ze zwierząt fotografowane zwykle na innym tle)?

W tym zadaniu nauczymy się prostej metody rozpoznawania na obrazie obszarów istotnych dla danej sieci. W tym celu wykonamy następujące kroki.

- Usuniemy fragment obrazu zastępując go szarym prostokątem.
- Policzymy % prawdopodobieństwo przynależności zmodyfikowanego zdjęcia do właściwej klasy.
- Sprawdzimy o ile % spadło w porównaniu do niezaciemnionej fotografii.
- Przesuwając prostokąt po całym zdjęciu poznamy ten % dla każdego z jego obszarów.
- Uzyskane wyniki najlepiej zaprezentować jako typową heatmapę – miejsca dużych spadków były dla sieci najbardziej istotne – to w oparciu o nie dokonywała takiej a nie innej klasyfikacji.
- Przykład realizacji i uzyskanych wyników prezentuje obraz poniżej.



W ramach zadania wykorzystaj nauczoną już sieć z Ćwiczenia 5 i przetestuj ją na różnych zdjęciach w opisany powyżej sposób. Czy obszary na których się skupiała miały sens z punktu widzenia rozpoznanej klasy?

ZADANIE DOMOWE 3 – I TY ZOSTANIESZ BOHATEREM MARVELA

By wykonać modyfikację fotografii taką jak przedstawiona na początku rozdziału należy skorzystać z techniki nazywanej potocznie art style transfer. W tym celu skorzystamy z gotowej implementacji, opisanej w tym (https://keras.io/examples/neural_style_transfer/) tutorialu, a bardziej szczegółowo w następującym (http://arxiv.org/abs/1508.06576) artykule.

Ponownie skorzystamy z wcześniej wyuczonej sieci – tym razem jest to VGG19. Wykorzystamy wyrażoną w niej informację o różnych rodzajach cech by zmienić styl jednego obrazu na taki, jaki ma inny obraz. Jak to zrobić? Schemat działania jest taki sam jak w rozdziale pierwszym, zmieniają się tylko szczegóły.

- Funkcja straty (loss) **L** tym razem nie opisuje, jak dobrzy jesteśmy w klasyfikacji (nie taki jest nasz cel), a zamiast tego powinna mówić o tym, jak dobrze udał się transfer stylu.
  - Nie będzie więc działała na wynikowych klasach, a na wynikowym zdjęciu.
  - Będzie też nieco bardziej skomplikowana – złożą się na nią 3 czynniki. Jak dokładnie należy je zaimplementować opisuje podlinkowany kod i publikacja. Tu skupimy się na ich ogólnym sensie.
    - **Style loss** (strata stylu) – oparta o wyjście z początkowych i środkowych warstw sieci (te warstwy opisujące cechy zależące od drobnych szczegółów zdjęcia). Tym niższa im bardziej podobne wyniki w tych warstwach są wyniki zwracane przez się dla modyfikowanego zdjęcia oraz wzorca stylu.
    - **Content loss** (strata zawartości) – oparta o wyjście z końcowych warstw sieci (te warstwy opisują wysokopoziomowe abstrakcyjne cechy – interesuje nas, by zdjęcie nadal przestawiało oryginalny obiekt, nie zaś w jaki sposób to zrobi). Tym niższa, im bardziej wyniki w tych warstwach są podobne dla oryginalnego i modyfikowanego zdjęcia.
    - **Variation loss** (strata wariacji) – tym niższa, im więcej gwałtownych zmian wartości pikseli na modyfikowanym zdjęciu. Chroni nas przed powstawaniem szumu i artefaktów, ma znacznie drugorzędny.
    - **Loss** (strata) **L** jest sumą wszystkich powyższych – jednakże to na którą z nich damy większy nacisk (dając jej większą wagę podczas sumowania) jest już tylko naszym artystycznym wyborem. Oczywiście marzymy o zminimalizowaniu wszystkich trzech, ale nie zawsze jest to możliwe.
- Co teraz? Znów używamy algorytmu optymalizacji, by znaleźć warunki w których **L** jest najmniejsze. Ponownie potrzebna jest drobna modyfikacja.
  - Tym razem nie zmieniamy wag sieci (one pozostają stałe podczas całego procesu), a zamiast nich modyfikujemy...zawartość poszczególnych pikseli. Staramy się zmieniać delikatnie wartości pikseli by **L** spadało – z każdym kolejnym krokiem oryginalne zdjęcie będzie coraz bardziej zmienione.

Celem zadania jest uruchomienie i przetestowanie podlinkowanej implementacji. Dokładnie zaś należy.

- Użyć analizatora aktywacji z Zadania Domowego 1 by sprawdzić, za co odpowiadają wykorzystane we wspomnianym kodzie warstwy.
- Uruchomić program dla własnej pary zdjęcie + wzorzec stylu.
- Prześledzić jak w kolejnych iteracjach spada wartość straty **L** - zarówno w całości jak i podziałem na 3 składowe.
- Wykonać eksperyment dla przynajmniej 2 różnych zestawów wag określających istotność danej składowej i sprawdzić jak różnić będą się wyniki.

ZADANIE DOMOWE 4 – JAK OSZUKAĆ SIEĆ NEURONOWĄ

Implementacja z Zadania Domowego 4 jest dość skomplikowana, ale możemy użyć jej jako inspiracji do innego ciekawego eksperymentu (zgodnie z twierdzeniem Copy'ego-Paste'a). Tym razem chcemy sprawdzić, jak trudno oszukać wyuczoną sieć. W tym celu weźmiemy zdjęcie przestawiające np. welsh corgi pembroke i postaramy się zmodyfikować tak, by AI uznało je za pudla. Przykładowy efekt poniżej (tu wykorzystano dodatkowy trick by wzmocnić zmiany, w praktyce mogą być nawet bardziej subtelne).

pembroke

poodle

(wszystkie klasy rozpoznawane przez tą sieć wraz z numerem wyjścia znajdują się np. w tym pliku: https://github.com/raghakot/kerasvis/blob/master/resources/imagenet_class_index.json).

Jak to zrobić? Większość kodu z Zadania Domowego 3 pozostaje taka sama (nadal modyfikujemy jakąś fotografię). Różnica jest jedna – tym razem funkcja straty **L** (loss) jest znacznie prostsza. Nie ma opisywać tego jak dobrze przeniesiono styl (złożone zjawisko o trzech składowych). Zamiast tego ma tylko jeden cel: zamienić corgiego w pudla! W związku z czym im większy % dostaje w wynikach klasa "pudel" tym mniejsza powinna być wartość funkcji **L**.

Celem zadania jest zaimplementowanie odpowiedniej straty **L**, by oszukać sieć neuronową zmodyfikowanym zdjęciem. Powinno udać się to zrealizować w jednej-dwóch iteracjach.

ZADANIE DOMOWE 5 – MÓJ WŁASNY TRANSFER STYLU

Ostatnie zadanie jest prawdopodobnie najtrudniejszym, stąd też należy je potraktować jako raczej bonusowe. Chcemy powtórzyć eksperyment z Zadania Domowego 3, ale z użyciem innej sieci dostępnej w bibliotece Keras (np. Xception albo MobileNetV2 z początkowych ćwiczeń). W tym celu należy zmodyfikować (wybierając wyjścia z właściwych warstw do wszystkich składowych **L** oraz korygując odpowiednie stałe liczbowe) kod z podanego tutoriala (a potem porównać wyniki). Jest to dowód na dobre zrozumienie tej techniki i umiejętność samodzielnego stosowania tejże.

PSZE PANA, A MUSZEM TO ROBIĆ NA CZYZERO?

Schemat oceniania:

- ukończone ćwiczenia z zajęć na żywo (lub jeszcze mniej) = 2.0 + osobista imienna nagana przesłana do prowadzącego przedmiot z prośbą o negatywny wpływ na ocenę końcową;
- ukończone ćwiczenia z zajęć na żywo + 1 zadanie domowe = 2.0;
- ukończone ćwiczenia z zajęć na żywo + 2 zadania domowe = 3.0;
- ukończone ćwiczenia z zajęć na żywo + 3 zadania domowe = 4.0;
- ukończone ćwiczenia z zajęć na żywo + 4 zadania domowe = 4.5;
- ukończone ćwiczenia z zajęć na żywo + wszystkie zadania domowe = 5.0;
- ukończone ćwiczenia z zajęć na żywo + wszystkie zadania domowe wykonane w staranny sposób = 5.0 + osobista imienna pochwała przesłana do prowadzącego przedmiot z prośbą o pozytywny wpływ na ocenę końcową.

Na koniec dowcipy opowiadane przez sieć neuronową wytrenowaną na zbiorze one-linerów: "What do you call a cat does it take to screw in a light bulb? They could worry the banana.", "What did the new ants say after a dog? It was a pirate.", "Why did the monsters change a lightbulb? And a cow the cough.", "What do you call a pastor cross the road? He take the chicken.", "What do you call a farts of tea? He was calling the game of the dry.". Kurtyna.