Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Marcin Możejko
Nr albumu: 262793

Podstawy metod głębokiego uczenia wraz z przykładami zastosowań

Praca magisterska
na kierunku MATEMATYKA

Praca wykonana pod kierunkiem
Dr Ewy Szczurek

Wrzesień 2017

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

W ostatnim czasie, wśród specjalistów zajmujących się uczeniem maszynowym, bardzo eksplorowanym tematem stało się tzw. głębokie uczenie, czyli uczenie z wykorzystaniem głębokich (tzn. mających wiele warstw) sieci neuronowych. W swojej pracy przedstawie matematyczne podstawy sukcesu stojących za tym algorytmów.

Słowa kluczowe

deep learning, machine learning, uczenie maszynowe, sztuczne sieci neuronowe, sieci konwolucyjne

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.4 Sztuczna inteligencja

Klasyfikacja tematyczna

D. Software
D.127. Blabalgorithms
D.127.6. Numerical blabalysis,
TODO

Tytuł pracy w języku angielskim

Foundations od deep learning methods with examples of applications

Spis treści

Wprowadzenie

W ostatnim czasie, jednym z tematów które najbardziej pochłaniają specjalistów od *Machine Learningu* stały się tzw. metody głębokiego uczenia (*deep learning*). Co chwila możemy usłyszeć informację, że według naukowców wykorzystujących te metody, algorytmy przez nich otrzymane pokonały człowieka w zagadnieniu, które do tej pory wydawało się być tylko w zasięgu ludzkiego umysłu ([1], [2]). To co łączy większość tych zagadnień i jest w mojej opinii kluczowe do zrozumienia fenomenu *deep learningu* to :

- ogromna ilość danych (często bez żadnych etykiet),
- duża problematyczność w zakodowaniu problemu w klasycznym języku algorytmicznym,
- zbiór interesujących nas danych jest niezwykle mały w stosunku do przestrzeni z której pochodzą dane.

Spróbujmy przyjrzeć się tym podpunktom. Rozważając np. problem rozpoznawania przedmiotów na filmach, możemy natrafić na ciekawy symptom. Dzięki wieloletniej historii zapisu cyfrowej, przez lata ludzkość zebrała na rozmaitych nośnikach danych niezmierzone ilości danych z kamer. Dzięki temu współcześni badacze mają ułatwione zadanie przy budowaniu algorytmów do rozpoznawania filmów - mogą wykorzystać tak zdobyta wiedzę do poprawienia rezultatów uzyskiwanych przez ich programy.

Kolejny podpunkt, czyli problematyczność, możemy doskonale zobrazować przez przykład zaprojektowania sieci rozpoznającej czy dana recenzja na portalu społecznościowym zawiera pozytywną opinię o temacie bądź nie. Niech czytelnik pokusi się o zapisanie na kartce rzeczy, które algorytm powinien sprawdzić aby uzyskać poprawną odpowiedź. Krótka inspekcja powinna uświadomić jak karkołomne zadanie staje przed badaczami, którzy się tym zajmują. Wyniki ostatnich algorytmów pokazują jednak, że problem ten nie przekracza zasiegu współczesnych technologii ([18]).

Ostatnie zagadnienie - najlepiej zobrazować liczbami. Załóżmy, że rozważamy obraz który posiada 28 x 28 pikseli oraz, dla uproszczenia, jest czarnobiały. Ilość wszystkich możliwych obrazków tego typu przekracza niewyobrażalnie liczbę atomów we wszechświecie. Natomiast liczba obrazków które prezentują nam coś interesującego (np. znany problem rozpoznawania cyfr [3]) jest liczbą zdecydowanie mniejszą. Można powiedzieć, że interesujący nas zbiór jest niczym galaktyka pośród otaczającego ją szumu i pustki.

W swojej pracy chciałbym przedstawić w jaki sposób poradzono sobie z powyższymi zagadnieniami. Zamierzam zacząć od historii powstania tych algorytmów, którą uważam za bardzo ważną w zrozumieniu istoty ich działania, by potem zgłębić podstawy matematycznych narzędzi stojących za sukcesem *deep learningu*. W kolejnych rozmiarach zamierzam, na podstawie historycznych algorytmów, przedstawić to, co stanowi esencję sukcesu metod głębokiego uczenia. Wydaję mi się, że ta droga wytłumaczy najlepiej, dlaczego niemal każdy szanujący się ośrodek naukowy posiada dziś grupę zajmującą się metodami głębokiego uczenia.

Rozdział 1

Historia

1.1. Początki i perceptron

Aby dobrze zrozumieć historię metod *deep learning* należy w moim mniemaniu choć pobieżnie zgłębić historię powstania i rozwoju metod opartych na tym paradygmacie. Można powiedzieć, że historia ta zaczęła się w momencie odkrycia przez biologów neuronalnej natury mózgu, jednak początki algorytmicznych sieci neuronowych, których najwyższą emanacją są głębokie sieci, datuje się na koniec lat 60' XX w., gdy Frank Rosenblat przygotował mechaniczno - elektryczny perceptron [4], czyli realizacje modelu neuronu McCullocha-Pittsa [5]. Choć teoretyczne podstawy opracowane przez wspomniany duet naukowców powstały ponad 25 lat wcześniej, to rewolucyjnym rozwiązaniem, zastosowanym w przypadku koncepcji Rosenblata był efektywny i skuteczny model uczenia, który przyczynił się do wieloletniego, bujnego rozwoju zastosowań opartych na tej koncepcji. Warto tutaj podkreślić motyw, który będzie powtarzał się przy omawaniu kolejnych punktów, biologiczne inspiracje, które doprowadziły do rozwoju algorytmicznych rozwiązań.

1.2. Zima sztucznej inteligencji

I wtedy, gdy wydawało się, że perceptron to narzędzie niemal idealne do wszystkich zadań, na horyzoncie pojawił się ogromny problem. Jako narzędzie, wynalazek Rosenblata służył min. do rozpoznawania obiektów na obrazkach. Okazało się jednak, że mniemanie o jego skuteczności jest znacznie przesadzone. W większości wypadków bowiem okazało się, że to co zostaje wykrywane to jakaś istotnie prosta cecha, która towarzyszy obrazom interesujących badaczy, a nie obiekty obecność których algorytm miał wykrywać. I tak np. zamiast statków algorytm wykrywał ciemne plamy po środku zdjęcia, a zamiast czołgów - zaciemnienia w kształcie lasu, w których najczęściej maszyny były fotografowane [6]. Co więcej, w Marvin Minsky oraz Seymour Papert [6] pokazali dużo mocniejsze ograniczenia na to, co mogło być efektywnie rozpoznane przez perceptron. Okazało się, że wyuczenie prostej, logicznej funkcji XOR przekracza możliwości wynalazku Rosenblata. Co więcej - uogólnienie tego wyniku pokazało, że w szczególności, niezależenie od rozmiaru oraz typu zbioru treningowego, nie można perceptronu nauczyć rozpoznawania wzoru, który byłby zamknięty na wszystkie możliwe translacje (przesunięcia) na obrazku. Fakt ten wywołał tak ogromny szok wśród badaczy, że ilość naukowców oraz funduszy przeznaczonych na badania drastycznie zmalała, a niski poziom zainteresowania utrzymywał się przez kolejne 10 lat. Dlatego właśnie ten okres nazywa się w historii *machine learningu* zimą sztucznej inteligencji.

1.3. Dlaczego perceptron zawiódł?

Odpowiedź na pytanie dlaczego perceptron zawiódł, należy podzielić na dwa fragmenty : jakie były matematyczne powody dla których algorytm nie podołał oczekiwaniom oraz dlaczego oczekiwania które zawiódł były tak ogromne. Odpowiedź na pierwszą część jest prozaiczna. Model McCullocha-Pittsa zastosowany przez Rosenblata, ogranicza działanie algorytmu do rozważania znaku jaki przyjmuje funkcjonał afiniczny dla zadanego wektora w $\mathbb{R}^n$. Krótka inspekcja tego faktu pokazuje, że istotnie rozróżniane wedle tego algorytmu mogą być tylko zbioru, które są separowalne liniowo, tzn. istnieje podprzestrzeń kowymiaru 1, która je istotnie oddziela. Odpowiedź na drugą część, czyli społeczne rozczarowanie, które pociągnął za sobą krach zaufania do perceptronu do dziś ciężko mi zrozumieć. Być może jest to wynikało to z ogólnego wówczas zachwytu nad nowymi technologiami wynikającego z wynalezienia komputerów i ogólnie elektroniki? ([19]) Z całą pewnością po latach - patrząc na matematyczne podstawy algorytmu Rosenblata - możemy stwierdzić, że nadmierne oczekiwania w stosunku do tak prostego narzędzia okazały się po prostu naiwne. Co więcej - naiwność ta nie leży tylko w matematycznej podstawie algorytmu, ale także w biologicznej nieadekwatności. Skoro naukowcy powoływali się częściowo na naśladowanie mózgu nie sposób dostrzec prostego faktu : organ ten u człowieka posiada ok. 10^{11} neuronów i z tyle razy mniejszej liczby składa się perceptron.

1.4. Algorytm wstecznej propagacji - odrodzenie i ponowna zima

Przez kolejne lata badacze rozwijający algorytmy związane z sieciami neuronowymi próbowali pokonać przedstawione powyżej przeszkody. Naturalnym rozwiązaniem wydawało się zbudowanie sieci składające się z większej ilości - tak neuronów, jak i ich warstw. Rozwiązanie to napotkało jednak na dosyć poważny problem - o ile uczenie perceptronu było zadaniem banalnie prostym, to wyuczenie sieci o większej ilości wartstw okazało się zadaniem o wiele trudniejszym. Podstawowy problem wynika z tego, że aby otrzymać nową jakość i przezwyciężyć stare problemy, należy zastosować inną niż liniową funkcję obliczającą wynik neuronu w zależności od informacji której do niego wprowadzimy. Konieczność ta wypływa ze znanego faktu, że złożenie wielu funkcji liniowych, jest także funkcją liniową. Zatem niezależenie od tego ile neuronów będzie składało się na strukturę naszej sieci - uzyskane narzędzie będzie równoważne klasycznemu perceptronowi.

Jednak wprowadzenie nieliniowych tzw. funkcji aktywacji rodzi inne problemy - jak mianowicie skutecznie przeprowadzić proces uczenia naszego modelu gdy analityczne rozwiązania na ogół są nieosiągalne? I tutaj pomoc przyszła z najmniej spodziewanej strony, a mianowicie pochodzącej z klasycznego rachunku prawdopodobieństwa reguły łańcuchowej. Algorytm, (Rumelhart et al. [7]) którego istota działania polega na obliczeniach gradientu błędu, a następnie aktualizowania odpowiednich parametrów na podstawie ich wpływu na omylność modelu (wpływ ten jest oblicznany właśnie przez regułę łańcuchową) przyniósł fantastyczne rezultaty. Ze względu na to, że błąd obliczany przy działaniu sieci, propaguje się następnie na odpowiednie parametry, nazwano go algorytmem wstecznej propagacji.

Od momentu prezentacji algorytmu (1986 r.) nastąpił ponowny, bujny rozwój sieci neuronowych, który zaowocował powstaniem wielu technik, które są popularne do dziś (np. [7, 10]. Co jednak charakterystyczne, entuzjazm towarzyszący badaniom stanowił tylko cień euforii która towarzyszyła wynalezieniu perceptronu. Po ok. 10 latach (początek lat 90' XXw.), z powodu zbyt wolnego rozwoju technologii i algorytmów, a także nieuzyskiwania oczekiwanych rezultatów, przyszedł czas ponownego rozczarowania, podczas którego ponownie niemal skre-

ślono sieci neuronowe z listy obiecujących kierunków rozwoju. ([20]) Traktowano je bardziej jako wzorowaną na przyrodzie ciekawostkę, niż drogę która może przynieść rozwiązanie dla podstawowych problemów sztucznej inteligencji.

1.5. Dlaczego sieci ponownie zawiodły?

Ponowna zima jednak, podobnie jak i entuzjazm który towarzyszył powrotowi sieci neuronowych do łask, okazała się dużo mniej intensywna niż poprzednia. Spróbujmy jednak przyjrzeć się podstawowym przyczynom, które sprawiły, że modele te zawiodły oczekiwania rozwijających je badaczy.

Pierwszą przyczyną stanowiły ponownie zbyt duże oczekiwania. Jakkolwiek na przełomie lat 90' XX w. udowodniono, że sieć z jedną tzw. warstwą ukrytą jest w stanie nauczyć się niemal każdej funkcji $f : \mathbb{R}^n \to \mathbb{R}$ ciągłej o zwartym nośniku ([11, 12], to nie do końca zdawano sobie sprawe, że zadanie nauczenia chociażby rozpoznawania obrazu lub ludzkiej mowy, choć możliwe, mogą być bardzo trudne. W szczególności, w dowodach powyższego faktu, niedostrzeżono alarmującego wzrostu złożoności problemu wraz ze wzrostem jego skomplikowania. Doprowadziło to do nieuzasadnionego rozczarowania tym, że *de facto* nie możemy nauczyć sieci wszystkiego na pamięć.

Drugą przyczynę, związaną właśnie ze wspomnianym powyżej problemem, stanowiły niedostateczne warunki sprzętowe, które uniemożliwiały wykorzystanie potencjału drzemiącego w algorytmie. Niedostateczność ta nie wynikała tylko ze zbyt małych możliwości obliczeniowych, ale także znacznego niedoboru danych (tak w sensie ich zebrania, jak i przechowywania). Największe zebrane zbiory treningowe nie liczyły, na początku lat 90', więcej niż 100 000 elementów. Z perspektywy wyuczenia chociażby zagadnień związanych z obrazem lub mową ludzką liczba ta jawi się jako niemalże śmiesznie mała. Problemy te jednak rozwiązał czas - wraz z rozwojem technologii pojawiły się tak i nowoczesne komputery, które przyspieszyły proces uczenia oraz ewaluacji, jak i ogromne, zróżnicowane zbiory danych, które sprawiły, że problemy o rozwiązaniu których marzyli eksperci od sztucznej inteligencji, znalazły się jak najbardziej w zasięgu ludzkich możliwości.

Rozdział 2

Matematyczne podstawy algorytmów

2.1. Uczenie

Wszystkie algorytmy opisane w tej pracy należą do rodziny algorytmów uczących. Aby móc o nich mówić musimy zatem wprowadzić formalną definicję uczenia. Zrobimy to w kolejnej podsekcji. Później przedstawimy trzy najczęstsze przykłady uczenia czyli uczenie z i bez nadzoru, a także uczenie z półnadzorem.

2.1.1. Formalna definicja uczenia

Trzy rzeczy są absolutnie niezbędne aby mówić o o algorytmach uczących. Pierwsza z nich to zbiór treningowy. Należy o nim myśleć jako o zbiorze danych, które znamy, a które chcemy wykorzystać jako podstawę do uczenia przy pomocy naszego algorytmu. Drugą z nich jest zbiór modeli. Efektem uczenia ma być efektywnie opisujący nasze dane model, aby jednak móc taki znaleźć, musimy ustalić pewien zbiór możliwych modeli spośród którego będziemy go wybierać. Koniecznym jest także ustalenie, co oznacza, że nasze dane są efektywnie opisywane przez wybrany przez nas opis - do tego służy funkcja kosztu. Intuicyjnie, funkcją kosztu nazywamy nieujemną funkcję rzeczywistą, malejącą wraz ze wzrostem efektywności opisywanych danych. Wszystkie powyższe intuicje składają się na poniższą definicję.

Definicja 2.1.1 *Niech $\mathcal{X}$ będzie dowolnym zbiorem (tzw. **treningowym**), Θ - zbiorem modeli, a*

$$J_{\mathcal{X}} : \Theta \to \mathbb{R}_+ \cup \{0\}$$

*funkcją kosztu. **Uczeniem** będziemy nazywać algorytm mający na celu odnalezienie :*

$$\theta_{\mathcal{X}} = \texttt{argmin}_{\theta \in \Theta} J_{\mathcal{X}}(\theta) .$$

Przy okazji tej definicji należy podnieść kilka istotnych kwestii, które są ściśle związane z procesem uczenia, a które należą do zagadnień zaawansowanej filozofii nauki i daleko przekraczają zakres poniższej pracy. Pierwszą rzeczą na którą należy zwrócić uwagę jest to, że $\theta_{\mathcal{X}}$ to *de facto* rozwiązanie pewnego problemu optymalizacyjnego i nie należy do gestii algorytmu uczącego rozsądzanie czy opis wyznaczany przez wybrany model mieści się w granicach które uznajemy za rozsądne, bądź nie. Wybór rodziny modeli, a także funkcji kosztu musi poprzedzić głęboka analiza, a także refleksja nad problemem, aby dokonany przez nas wybór nie okazał się niesatysfakcjonujący. Należy mieć także w pamięci, że zgodnie z aktualnie przyjętą filozofią ([9]), wybrany przez nas model, nawet w kontekście uzyskania pozytywnych wyników, należy traktować jako niesfalsyfikowany, a nie prawdziwy. Jest to o tyle istotne, że wypracowane przez

nas rozwiązanie zależy w bardzo istotny sposób od zbiory danych, który wykorzystujemy przy uczeniu - napływ kolejnych, może doprowadzić do weryfikacji tak konkrentego $\theta_{\mathcal{X}}$ jak i samej rodziny Θ.

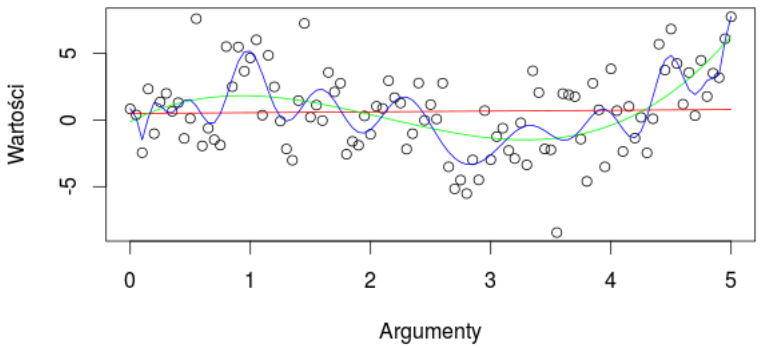
2.1.2. Uczenie z nadzorem

Przez uczenie z nadzorem rozumiemy przypadek, w którym możemy wyróżnić dwa zbiory $\mathbb{X}_{\mathcal{X}}$ (zbiór argumentów) oraz $\mathbb{Y}_{\mathcal{X}}$ (zbiór wartości) takie, że $\mathcal{X} \subset \mathbb{X}_{\mathcal{X}} \times \mathbb{Y}_{\mathcal{X}}$. Oznacza to, że na każdy element zbioru treningowego możemy patrzyć jak na parę argument - wartość, a rodzinę modeli Θ z których wybieramy określić jako rodzinę funkcji najlepiej przybliżającą relację jaką na $\mathbb{X}_{\mathcal{X}} \times \mathbb{Y}_{\mathcal{X}}$ generuje zbiór $\mathcal{X}$. Dobrą ilustrację do tego zagadnienia ilustruje poniższy przykład.

Przykład 2.1.1 *W poniższym przykładzie za zbiór treningowy $\mathcal{X}$ uznamy punkty zaznaczone czarnymi okręgami, przyjmując, że pierwsza współrzędna odpowiada argumentom, a druga - wartościom. Funkcją kosztu będzie funkcja :*

$$J(\theta) = \sum_{x \in \mathbb{X}_{\mathcal{X}}} (y_x - f_\theta(x))^2 ,$$

gdzie y_x to wartość odpowiadająca argumentowi x, a f_θ to funkcja odpowiadająca modelowi θ. Na poniższym obrazku kolorem czerwonym oznaczono $\theta_{\mathcal{X}}$ w przypadku gdy Θ to zbiór funkcji liniowych, zielonym - gdy Θ to zbiór funkcji sześciennych, natomiast niebieskiem - przypadek gdy Θ to zbiór funkcji wielomianowych stopnia co najwyżej 20.



Warto podkreślić, że wybrana przeze mnie ogólność definicji ma swoje głębokie uzasadnienie. Zwyczajowo techniki uczenia z nadzorem dzieli się na dwie grupy : techniki regresji (gdy $\mathbb{Y}_{\mathcal{X}} = \mathbb{R}^n$ dla pewnego $n \in \mathbb{N}$) oraz techniki klasyfikacji (gdy $\mathbb{Y}_{\mathcal{X}}$ jest zbiorem dyskretnym). W swoim opisie podkreślam jednak to, że tym co *de facto* jest szukane, to zależność funkcyjna, ponieważ pozwala nam to na znacznie szerszy dobór struktur wartości (co np. gdy $\mathbb{Y}_{\mathcal{X}}$ to zbiór zdań w języku angielskim), a także unika konfuzji gdy w niektórych przypadkach wybrane przez nas własności $\mathbb{Y}_{\mathcal{X}}$ mogą prowadzić do nieefektywnego uczenia lub rezultatów innych niż oczekiwane.

2.1.3. Uczenie bez nadzoru

Przez uczenie bez nadzoru będziemy rozumieć taką formę uczenia, w której Θ jest zbiorem możliwych, pożytecznych reprezentacji zbioru danych $\mathcal{X}$. Funkcja kosztu $J_{\mathcal{X}}$ określa w tym przypadku jak efektywny opis danych gwarantuje nam model θ. Często zmiana reprezentacji danych stanowi konieczny krok przy analizie danych. Zauważmy np., że z perspektywy komputera obraz z kamery telewizyjnej to ciąg elementów ze zbioru $\mathbb{R}^{6220800}$ (przy założeniu jakości FullHd z kodowaniem RGB), co czyni problem komputerowej analizy takiego obrazu praktycznie nierozwiązywalnym. Na poniższym przykładzie zobaczmy jak wygląda dobór reprezentacji przy uczeniu bez nadzoru.

Przykład 2.1.2 *Poniżej przedstawimy przykład tzw. uczenia rozmaitości (manifold learning [21]) metodą t-SNE. W przykładzie tym:*

$$\mathcal{X} = \{x_1, x_2, \ldots, x_k\} \subset \mathbb{R}^n,$$

oraz

$$\Theta = \{\{\theta_1, \theta_2, \ldots, \theta_k\} : \theta_1, \theta_2, \ldots, \theta_k \in \mathbb{R}^m\}.$$

Aby zdefiniować funkcję błędu potrzebne będą nam pomocnicze wyrażenia:

$$p_{j|i} = \frac{exp\left(-||x_i - x_j||^2/2\sigma_i^2\right)}{\sum_{k\neq i} exp\left(-||x_i - x_k||^2/2\sigma_i^2\right)},$$

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2},$$

$$q_{ij} = \frac{\left(1 + ||\theta_i - \theta_j||^2\right)^{-1}}{\sum_{k\neq l}\left(1 + ||\theta_k - \theta_l||^2\right)^{-1}},$$

gdzie $\sigma_i \in \mathbb{R}^2$ dla $i = 1, \ldots n$ to ustalone wcześniej parametry. Ustalmy funkcję kosztu:

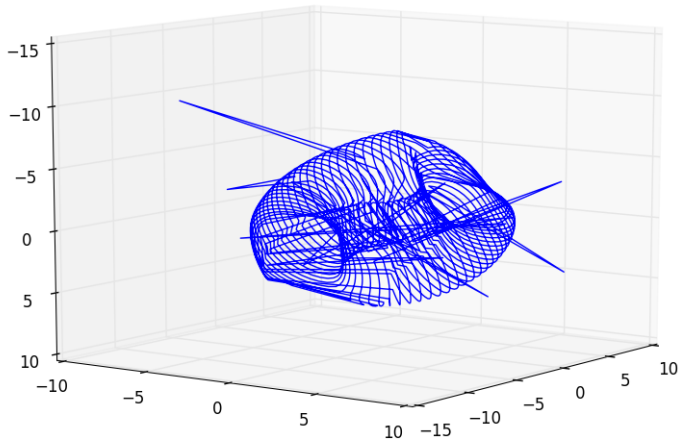
$$J_\theta = \sum_{i\neq j} p_{ij} log \frac{p_{ij}}{q_{ij}}.$$

Rozwiązanie tego typu problemu możemy uznać jako znalezienie odpowiedniej reprezentacji θ_i dla każdego z punktów x_i, gdzie $i = 1, \ldots, n$. Na poniższym obrazku prezentujemy przykładowe rozwiązanie dla następującego zbioru :

$$\mathcal{X} = \left\{(\sin\alpha, \cos\alpha, sin\beta, \cos\beta) : \alpha, \beta = 0, \frac{2\pi}{n}, \frac{4\pi}{n}, \ldots, 2\pi\right\} \subset \mathbb{R}^4,$$

czyli czterowymiarowego zanurzenia torusa w wypadku gdy $m = 3$, $n = 50$ oraz

$$\sigma_1, \sigma_2, \ldots, \sigma_{2500} = 1.$$



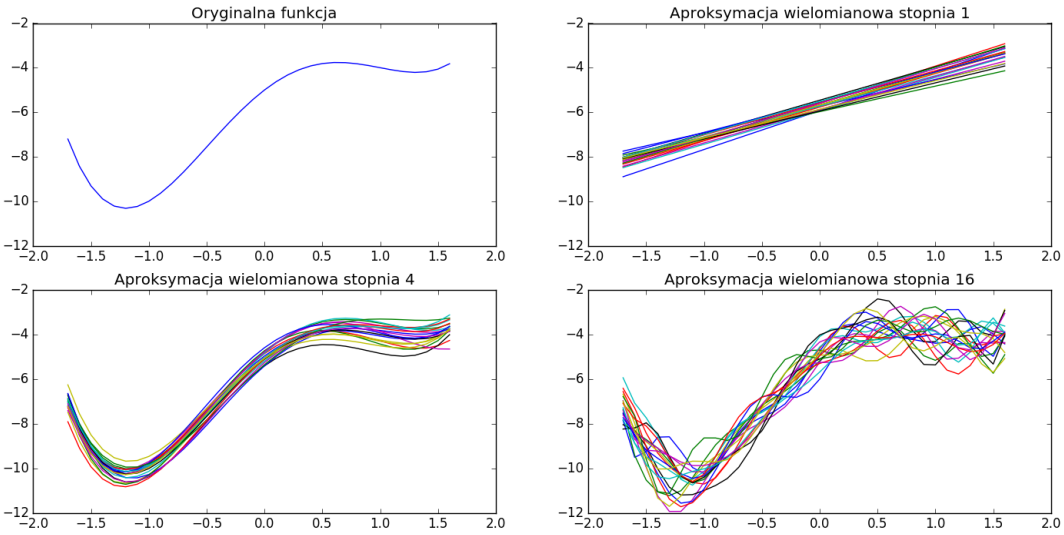
Na obrazku widać, że pączkowata struktura torusa została uchwycona w trójwymiarowej reprezentacji zbioru $\mathcal{X}$.

2.1.4. Generalizacja oraz przeuczenie

W dotychczasowych rozważaniach przyjęliśmy, że głównym celem uczenia jest odnalezienie modelu, który minimalizuje ustaloną przez nas funkcję błędu. Jednak problem uczenia zawiera w sobie jeszcze jedną płaszczyznę. Zbiór posiadanych przez nas danych może się rozszerzyć o nowe punkty - i naturalnym jest oczekiwanie od modelu, że funkcja błędu będzie przybierała porównywalne wartości na rozszerzonym zbiorze. Taką zdolność modelu nazywamy *generalizacją*.

Zjawisko przeciwne, tzn. stan w którym wartość funkcji błędu wzrasta znacząco po dodaniu nowych punktów nazywamy *przeuczeniem*.

Przykład 2.1.3 *Jednym z ciekawych, ale mało praktycznych sposobów sprawdzenia czy nasz model może doświadczyć zjawiska przeuczenia jest analiza wariancji modelów metodą tzw. resamplingu. Intuicyjnie - jeśli model ma uchwycić faktyczną strukturę danych - to dla różnych zbiorów danych pochodzących z tego samego źródła - modele zbudowane wedle tych zbiorów powinny być podobne. Duża wariancja wśród modeli może natomiast wskazywać na to, że nasz model może doświadczać zjawiska przeuczenia. Poniżej prezentujemy przykład takiego sprawdzenia. Oryginalne dane pochodzą z wielomianu 4-go stopnia, którego wykres zaprezentowany jest w prawym górnym rogu obrazka. W kolejnych oknach prezentujemy różne modele wielomianowe wytrenowane na delikatnie zaburzonych danych podstawowych.*



Widzimy, że w przypadku wielomianów stopnia pierwszego mamy do czynienia z małą wariancją modeli - jednakowoż - modele te odległe są od dobrej aproksymacji wyjściowej funkcji. W przypadku aproksymacji wielomianami stopnia 4 - zarówno jakość aproksymacji oraz jednorodność modeli stoją na wysokim poziomie. W wypadku aproksymacji modelami stopnia 16 - widzimy, że zarówno jakość aproksymacji oraz jednorodność modeli uległy znaczącemu obniżeniu. Modele tej rodziny mogą cierpieć z powodu zjawiska przeuczenia.

Zbiory walidacyjne i testowe

Częstym i naturalnym przeznaczeniem modelu jest jego zastosowanie na nowych danych. Dlatego sprawdzenie czy wyniki uzyskiwane na przykładach pochodzących spoza zbioru treningowego są porównywalne do tych uzyskiwanych na zbiorze testowym w wielu przypadkach stanowi kluczowy punkt w ocenie przydatności rozwiązania. W celu zasymulowania napływu nowych danych stosuje się tzw. technikę *zbioru testowego*, w której część danych nie uczestniczy w działaniu algorytmu optymalizacyjnego - lecz mierzy aktualną skuteczność modelu i pozwala np. zaobserwować czy model działa znacznie lepiej na przykładach treningowych niż testowych (co jest zaprzeczeniem tego, że model powinien dobrze radzić sobie dla nowych danych).

Częstym zjawiskiem jest także wyodrębnianie jeszcze jednego zbioru - tzw. *zbioru walidacyjnego*. Celem tego wyodrębnienia jest uniknięcie sytuacji w której rozwijający rozwiązanie wykorzystują informację ze zbioru testowego w trenowaniu. Wówczas nie jest uczciwym powiedzenie, że zbiór testowy *naśladuje* napływ nowych danych. Dlatego ostateczny wynik potwiedzony jest na osobnym *zbiorze walidacyjnym*, na którym rozwiązanie użyte jest dokładnie jeden raz i które wynik jest ostateczną estymatą użyteczności rozwiązania.

Warto podkreślić, że w niektórych źródłach określa się osobny zbiór wykorzystywany podczas trenowania mianem *zbioru testowego*, a ostateczny wynik uzyskiwany jest na zbiorze określanym mianem *zbioru walidacyjnego* (jest to notacja odwrotna do przedstawionej w tej pracy).

Walidacja krzyżowa (ang. *cross-validation*)

Wyrażnymi wadami techniki *zbioru walidacyjnego* są dwa następujące fakty:

- tylko niewielki procent posiadanych danych zostaje wykorzystany do symulowania napłynięcia nowych danych, co czyni estymatę skuteczności modelu mniej dokładną,
- wynik modelu może zostać wyraźnie zaburzony jeśli wydzielony do testowania zbiór będzie miał dystrybucje znacząco różną od dystrybucji zbioru treningowego (tzw. szum podziału - ang. *bias split*).

Aby uniknąć tych dwóch niedogodności stosuję się tzw. metodę *k-walidacji krzyżowej*, w której zbiór treningowy dzielony jest na k róznych części (ang. *folds*), a następnie wybiera się kolejne $k-1$ elementowe zbiory części jako zbiory treningowe, a niewybraną część wykorzystuje się do walidacji. Uzyskane w ten sposób k wyników wykorzystuje się do ostatecznej estymaty (najczęściej badając ich rozkład i biorąc średnią wyników jako ostateczny wynik).

Dzieki tej metodzie każdy element zbioru treningowego uczestniczy w symulowaniu napływu nowych danych, a zjawisko szumu podziału jest mniej szkodliwe, jako, że wielokrotne jego powtórzenie jest zjawiskiem bardzo rzadkim. Wszystko to odbywa się kosztem czasu uczenia - w związku z wielokrotnym uczeniem modelu technika ta wymaga zazwyczaj znacznie więcej czasu niż uczenie wykorzystujący podział na zbiory *treningowy-testowy-walidacyjny*.

Warto dodać - że ze zbioru treningowe często wydziela się osobny zbiór testowy, który uczestniczy w procesie uczenia w sposób opisany w poprzedniej subsekcji.

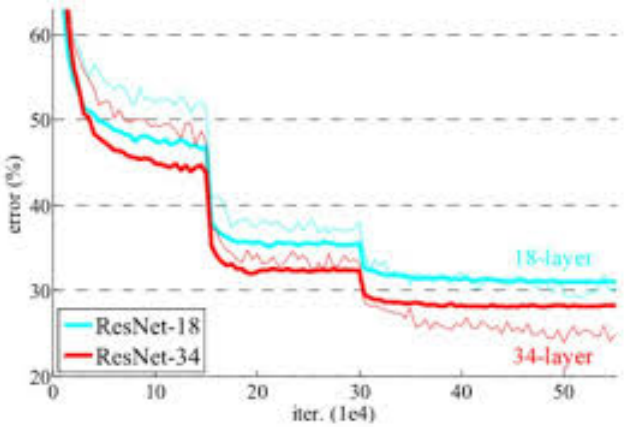
2.1.5. Optymalizacja gradientowa

Jak wspominaliśmy w podsekcji 2.1.1, algorytmy uczenia to de facto algorytmy rozwiązujące pewne zadanie optymalizacyjne. Aktualnie tylko dla niewielkiej części z nich potrafimy odnaleźć rozwiązanie w sposób ściśle analityczny. Dlatego dla dużej grupy problemów uczenia maszynowego stosuje się tzw. metody optymalizacji gradientowej. Schemat działania większości z tych metod przybliża poniższy algorytm ([22]).

Data: Zbiór danych $\mathcal{X}$, rodzina modeli Θ parametryzowana przez argumenty $\theta_1, \theta_2, \ldots, \theta_n$ oraz różniczkowalna funkcja błędu $J_{\mathcal{X}}$.
Result: Wartości parametrów $\theta_1, \theta_2, \ldots, \theta_n$ dla których rozwiązanie jest możliwie najlepsze ze względu na funkcję kosztu $J_{\mathcal{X}}$.
zainicjuj wagi $\theta_1, \theta_2, \ldots, \theta_n$ w sposób losowy. ;
oblicz $J_{\mathcal{X}}(\theta_1, \theta_2, \ldots, \theta_n)$;
while $J_{\mathcal{X}}(\theta_1, \theta_2, \ldots, \theta_n)$ *nie spełnia warunku stopu* **do**
  oblicz $\delta_i = \frac{\partial J_{\mathcal{X}}(\theta_1, \theta_2, \ldots, \theta_n)}{\partial \theta_i}$ dla każdego $i = 1, \ldots, n$;
  zaktualizuj: $\theta_i := \theta_i - \delta_i \eta$ dla każdego $i = 1, \ldots, n$;
  ponownie oblicz $J_{\mathcal{X}}(\theta_1, \theta_2, \ldots, \theta_n)$;
end

Algorithm 1: Uproszczony schemat optymalizacji gradientowej

Oczywiście powyższy schemat został uproszczony w celu uchwycenia głównej idei. Zgodnie z matematycznym faktem, który mówi, że gradient wyznacza kierunek najwyższego wzrostu, podążanie w kierunku przeciwnym do wyznaczonego przez niego jest tożsame podążaniu w kierunku największego spadku. Stała η, która pojawia się w algorytmie nazywana jest *stałą uczenia* (ang. *learning rate*) i powinna być dobierana ostrożnie. W wielu algorytmach może ona zmieniać się wraz z rosnącą liczbą iteracji, a najskuteczniejsze aktualnie metody gradientowe ustalają ją nawet jako funkcję wag oraz historii kolejnych iteracji ([13] [23]). Warunek stopu występujący w warunku pętli natomiast najczęściej ustalany jest jako zejście wartości funkcji



Rysunek 2.1: Przykład zmiany funkcji koszty po redukcji stałej uczenia - za [31].

kosztu $J_{\mathcal{X}}$ poniżej pewnej ustalonej wartości ϵ lub brak dostatecznej poprawy (względnej lub bezwzględnej) w stosunku do wartości obliczonej w poprzedniej iteracji.

Innym często spotykanym schematem jest ustalenie pewnej dużej ilości operacji uczenia (zwanych często *epokami* - ang. *epochs*) i brania jako ostatecznego modelu albo rozwiązania uzyskanego po ostatniej epoce, albo rozwiązania które uzyskało najlepszy wynik na *zbiorze testowym*.

Redukcja stałej uczenia w wypadku wysokiej i stałej wartości funkcji kosztu (ang. *learning rate reduction on plateau*)

Kolejną techniką wykorzystywaną podczas uczenia gradientowego jest tzw. *redukcja stałej uczenia w wypadku stałej i wysokiej wartości kosztu uczenia*. Opiszemy ją szerzej w związku z tym, że zostanie wykorzystana w rozwiązaniu opisanym w kolejnych rozdziałach tej pracy. Metoda ta, często używana podczas uczenia gradientowego, zakłada, że gdy kolejne epoki nie przynoszą znaczącej poprawy - oznacza to, że lokalny kształt funkcji kosztu (w zależności od parametru) zawiera subtelności, które nie są dostatecznie dobrze uwzględniane przez zbyt dużą wartość *stałej uczenia*. Dlatego zwykło się przemnażać ją przez ustaloną stałą $p < 1$ (często przyjmuje się $p = 0.1$) w celu uwrażliwienia metody optymalizacji na znacznie mniejsze, lokalne zmiany gradientu (2.1.5).

2.1.6. Stochastyczna optymalizacja gradientowa

Bardzo często, w przypadku gdy liczność zbioru danych $\mathcal{X}$ jest wyjątkowo duża, obliczenie funkcji błędu, a także jej gradientów stanowi zadanie nieefektywne obliczeniowo. W takich przypadkach często stosowaną techniką jest metoda stochastycznej optymalizacji gradientowej. Najczęściej polega ona na losowym podziale zbioru na mniejsze części i sukcesywnym stosowaniu metody gradientowej na kolejnych elementach tego podziału. Zwyczajowo zbiory dzieli się porcje (ang. *batch*) równej liczności i ze względu na rozmiar tych porcji wyróżniamy:

- uczenie *online* w którym każda porcja składa się z jednego przykładu,
- uczenie *mini-batch*, w którym liczność porcji jest znacząco mniejsza od rozmiaru całego zbioru,
- uczenie *full-batch*, w którym mamy tylko jedną porcję - składającą się z całego zbioru.

Co warte podkreślenia, losowanie tych danych wprowadza do uczenia dosyć sporą dozę losowości. Może to oczywiście znacząco spowolnić proces uczenia lub doprowadzić do tego, że odnaleziony przez nas model będzie daleki od optymalnego. Okazuje się jednak, że w praktyce odpowiedni dobór rozmiaru próbki, w połączeniu z zastosowaniem metod granicznych ([14]) przynosi znaczące przyspieszenie algorytmów uczących przy minimalnej stracie poprawności. Intuicja która za tym stoi, mówi, że pomimo iż dana porcja może delikatnie zaburzyć nasz model, to średnio (ponieważ *de facto* próbkujemy nasz zbiór danych) uzyskamy krok w dobrym kierunku ([24]).

2.1.7. Metaoptymalizacja

W poprzednie podsekcji przedstawiliśmy sposób znalezienia optymalnego rozwiązania w przypadku gdy zależność funkcji błędu od parametrów jest różniczkowalna. Jednak często w tego typu zagadnieniach mamy do czynienia także z parametrami które nie mają tej właściowości. Tradycyjnie nazywamy takie parametry *metaparametrami*. W tego typu przypadkach najczęściej sposób działania wygląda następująco:

1. Podzielmy zbiór paramterów Θ na Θ_g i Θ_m gdzie Θ_g to parametry ze względu na które funkcja błędu jest różniczkowalna, a Θ_m to *metaparametry*. Zakładamy, że $\Theta = \Theta_g \times \Theta_m$, a także, że $\Theta_m = \Theta_m^1 \times \cdots \times \Theta_m^k$ dla pewnego k naturalnego.

2. Dla każdego $i = 1, \ldots, k$ wybierzmy skończony podzbiór $\bar{\Theta}_m^i$. Oznaczmy przed $grid = \bar{\Theta}_m^1 \times \cdots \times \bar{\Theta}_m^k$.

3. Dla każdego $g \in grid$ ustalmy kryterium oceny danego zbioru metaparametrów. Może to być np. wartość funkcji błędu uzyskana przy pomocy gradientowej optymalizacji parametrów z rodziny Θ_g.

4. Dla każdej wartości $g \in grid$ obliczmy wartość kryterium opisanego w podpunkcie 3 i jako ostateczny wybierzmy model który minimalizuje to kryterium.

Powyższy schemat prezentuje tzw. algorytm *grid search*. Jednak w przypadku gdy wartość $|grid|$ jest duża - obliczenia te mogą być nieosiągalne ze względu na naturę obliczeniową. W takich przypadkach stosuje się lekko zmodyfikowany schemat nazywany *random search*. W schemacie tym ustalamy pewną liczbę $k <= |grid|$, a następnie losujemy k krotnie bez zwracania wartości z *grid* wybierając na końcu najlepszą wartość spośród wylosowanych parametrów. O zaskakującej skuteczności tego podejścia można poczytać chociażby w [15].

2.2. Sztuczne Sieci neuronowe

W tym rozdziale zajmiemy się omówieniem podstawowego narzędzia, z jakiego korzystają wszystkie algorytmy głębokiego uczenia, czyli sztucznych sieci neuronowych (ang. *artificial neural network*). Co ciekawe, w literaturze nie istnieje jedna, formalna definicja wspólna dla wszystkich sieci neuronowych. Dlatego w poniższym rozdziale, przeanalizujemy różne przykłady sieci (przyglądając się ich historycznemu rozwojowi) które przybliżą nam podstawowy koncept ich funkcjonowania, a także zgłębimy biologiczną inspirację, która stoi za ich wynalezieniem, czyli układ nerwowy zwierząt z jego ewolucyjnym zwięczeniem (czyli ludzkim mózgiem).

2.2.1. Biologiczna motywacja

Badania naukowców i neurobiologów nad działaniem ludzkiego mózgu trwają od nie z mała 100 lat i nadal skrywa on przed badaczami ogrom tajemnic. Jednak już dzisiaj możemy z całą pewnością stwierdzić, że działanie ludzkiego mózgu opiera się na zorganizowanej współpracy ogromnej ilości komórek nerwowych. Każdy neuron przyjmuje od połączonych z nim innych neuronów nieregularnie w czasie aktywacje elektryczne i na ich podstawie sam emituje (lub nie) impuls elektrycznych do neuronów z którymi jest połączony. Zasada współpracy ze sobą wielu jednostek obliczeniowych (jako taką możemy traktować neuron) stanowi podstawę niemal każdej architektury sieci neuronowych.

2.2.2. Regresja logistyczna

Jednym z najpopularniejszych algorytmów uczenia maszynowego jest aktualnie algorytm tzw. *regresji logistycznej* ([25]). Z naszej strony jest on interesujacy ponieważ możemy interpretować go jako pojedynczy obliczeniowy neuron. Zacznijmy jednak od formalnej definicji:

Definicja 2.2.1 *Modelem regresji logistycznej parametryzowanej wektorem* $(\theta_0, \theta_1, \ldots, \theta_n)$ *będziemy nazywali funkcję* $f : \mathbb{R}^n \to (0, 1)$ *zadaną wzorem:*

$$f_\theta(x) = \phi\left(\theta_0 + \sum_{i=1}^{n} \theta_i x_i\right),$$

gdzie ϕ *to sigmoidalna funkcja aktywacji opisana w paragrafie 2.2.2.*

Widzimy zatem, że jeśli zinterpretujemy $x_1, x_2, \ldots x_n$ wartości impulsów z komórek nerwowych sąsiadujących z naszą, parametry $\theta_1, \theta_2, \ldots, \theta_n$ jako siły połączeń naszego neuronu ze swoimi sąsiadami oraz θ_0 jako wewnętrzną pobudliwość neuronów - to widzimy, że w oparciu o interakcję z sąsiadami - nasz neuron oblicza swą wewnętrzną aktywację, przekazywaną jako wynik funkcji f.

Sigmoidalna funkcja aktywacji

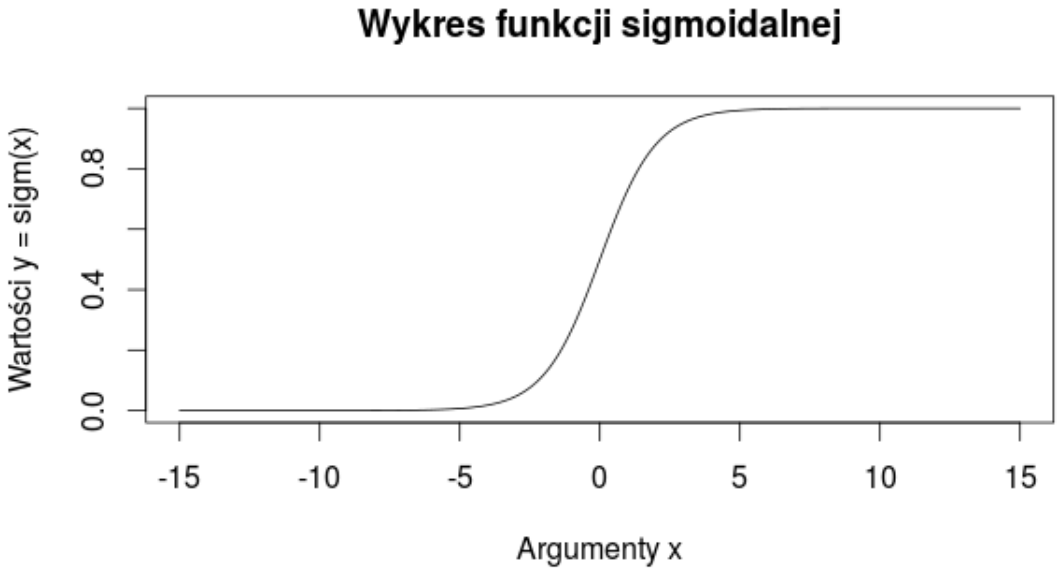
Definicja 2.2.2 *Sigmoidalną (logistyczną) funkcją aktywacji będziemy nazywać funkcję :*

$$S(x) = \frac{1}{1 + e^{-x}}.$$

Oto niektóre z jej ważnych własności:

Uwaga 2.2.1 *Dla funkcji logistycznej zdefiniowanej powyżej zachodzi :*

- $\lim_{x\to+\infty} S(x) = 1$,
- $\lim_{x\to-\infty} S(x) = 0$,
- $S'(x) = S(x)(1 - S(x))$,
- $\lim_{x\to+\infty} S'(x) = 0$ *saturacja prawostronna,*
- $\lim_{x\to-\infty} S'(x) = 0$ *saturacja lewostronna.*



Rysunek 2.2: Wykres funkcji sigmoidalnej.

Zauważmy, że na funkcję sigmoidalną możemy spojrzeć jako na ciągłe przybliżenie funkcji charakterystycznej zbioru $\{x \in \mathbb{R} : x > 0\}$. Kluczowe w zrozumieniu sekcji [**?**] będzie natomiast własność saturacji. Zauważmy, że dla liczb o dużej wartości bezwzględnej gradient funkcji logistycznej praktycznie się zeruje. Może to spowodować znaczące spowolnienie uczenia w wypadku wyboru metody optymalizacji gradientowej.

Wynik regresji logistycznej uznawany jest prawdopodobieństwo przynależności do pewnej ustalonej klasy - zatem rozwiązuje ona zagadnienie klasyfikacji binarnej.

Logistyczna funkcja kosztu - (ang. *log-loss*)

Jedną z podstaw sukcesu regresji logistycznej była nowa funkcja kosztu zadana wzorem:

$$J_{\mathcal{X},\theta} = -\frac{1}{|\mathcal{X}|}\sum_{x\in\mathcal{X}} \log\left(f_\theta(x)\mathbb{I}_{y_x=1} + \log(1-f_\theta(x))\mathbb{I}_{y_x=0}\right),$$

gdzie $y_x \ \{0,1\}$ to prawidłowa klasa przykładu $x \in \mathcal{X}$. Funkcja ta okazała się dalece bardziej skuteczna od klasycznego błędu średniokwadratowego oraz posiada teorioinformacyjną interpretację jako tzw. odległość *Kullbacka-Leiblera* pomiędzy rozkładem wyznaczonym przez f_θ, a uczony rozkładem pochodzącym z danych ([33]). Innym określeniem logistycznej funkcji kosztu jest *binarna entropia krzyżowa* (ang. *binary crossentropy*).

Uogólnienie funkcji *log-loss* do zagadnienia multiklasyfikacji

Istnieje naturalne przedłużenie regresji logistycznej do zagadnienia multiklasyfikacji zwane regresją multinomialną. W tym wypadku zamiast funkcjonału używamy wielowymiarowego przekształcenia liniowego, miast funkcji sigmoidalnej używa się tzw. funkcji *softmax* $Soft : \mathbb{R}^n \to \mathbb{R}^n$, zadanej wzorem:

$$Soft_\theta(x)_i = \frac{e^{(\theta_i x)_i}}{\sum_{j=1}^{n} e^{(\theta_j x)_j}},$$

gdzie θ to macierz przekształcenia liniowego, natomiast funkcja kosztu (nazywana *kategoryczną entropią krzyżową* - ang. *categorical crossentropy* - lub *entropią krzyżową typu softmax* - ang. *softmax crossentropy*) zadana jest wzorem:

$$J_{\mathcal{X},\theta} = -\frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \sum_{i=1}^{k} \left(\log Soft_\theta(x)_i \mathbb{I}_{y_x = i} + \log\left(1 - Soft_\theta(x)_i\right) \mathbb{I}_{y_x \neq i} \right).$$

Uogólnienie powyższych funkcji kosztu do dowolnego modelu

Zarówno dla *logistycznej funkcji kosztu* oraz *kategorycznej entropii krzyżowej* nie jest wymagane aby wyniki używane do jej obliczenia pochodziły z przekształceń liniowych. Regresję logistyczną możemy zastąpić dowolną funkcją $model_b : \mathbb{R}^n \to \mathbb{R}$ złożoną z funkcją sigmoidalną S, natomiast regresję mutlinomialną możemy zastąpić dowolną funkcją $model_c : \mathbb{R}^n \to \mathbb{R}^k$ złożoną z funkcją $Soft$ i nadal uzyskać użyteczną funkcję kosztu. W dalszej części funkcje $model_b$ i $model_c$ będą pewnymi sieciami neuronowymi.

2.3. Klasyczne sieci neuronowe oraz algorytm wstecznej propagacji

Regresja logistyczna - dzięki swojej prostocie - zyskała sobie miano bardzo popularnej metody w dziedzienia uczenia maszynowego. Ogromnym problemem jaki napotykamy jednak przy jej używaniu jest to, że dzięki swojej prostocie - nie jest w stanie efektywnie nauczyć się bardziej skomplikowanych funkcji. Szybko jednak zauważono, że tak jak układy nerwowe zwierząt nie składają sie tylko i wyłącznie z jednego neuronu, tylko z ich współdziałającego ze sobą konglomeratu, tak skutecznie działających modeli nie należy szukać w funkcjach naśladujących pojedyncze neurony, lecz w złożeniu takich funkcji ze sobą. Tak narodziła się idea *sieci wielowarstwowych* które stanowią podstawę także algorytmów *deep-learning*.

2.3.1. Topologie sieci

Aby pozbyć się problemu jaki niesie ze sobą prostota perceptronu, przyjęto, że wyjście z wielu perceptronów, które jako argumenty przyjmują ten sam wektor x czyli :

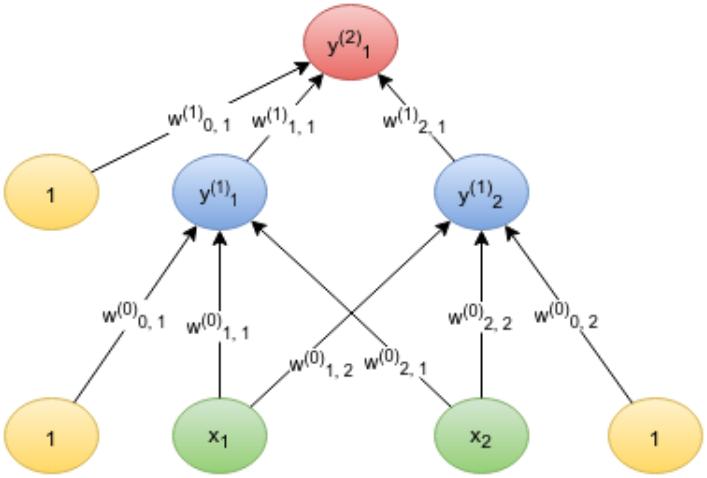
$$y_k^{(1)} = \phi_1 \left(\sum_{i=1}^{n} x_i w_{i,k}^{(1)} + w_{0,k}^{(1)} \right),$$

stanie się automatycznie *wejściem* dla kolejnego perceptronu :

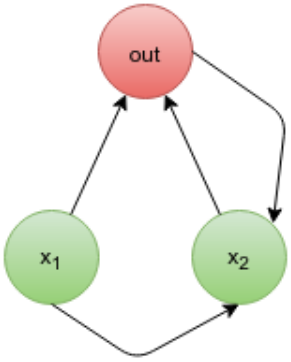
$$y_j^{(2)} = \phi_2 \left(\sum_{k=1}^{m} y_k w_{k,j}^{(2)} + w_{0,j}^{(2)} \right).$$

Oczywiście proces tworzenia takiej sieci można iterować i tak np. trzecia wartstwa miałaby postać :

$$y_l^{(3)} = \phi_3 \left(\sum_{j=1}^{m} y_j w_{j,l}^{(3)} + w_{0,l}^{(3)} \right).$$



Rysunek 2.3: Przykład architektury wartstwowej sieci typu *feed-forward*. Sieć o tej architekturze jest w stanie nauczyć się logicznej funkcji XOR.



Rysunek 2.4: Przykład architektury rekurencyjnej. Zwróćmy uwagę na skierowany cykl w grafie połączeń.

Dla uproszczenia przyjęliśmy, że w obrębie warstwy funkcji aktywacji są takie same, jednak nie stanowi to koniecznego wymogu. Przedstawiona powyżej architektura stanowi przykład architektury warstwowej - kolejne warstwy przyjmują jako swoje argumenty rezultaty obliczeń z poprzednich warstw (stąd nazwa *wielowarstwowy perceptron*). O wektorze wejściowym x zwykło się zazwyczaj mówić jako o warstwie wejściowej (ang.*input layer*), natomiast o ostatniej wartstwie jako warstwie wyjściowej (ang. *output layer*). Oczywiście istnieją także architektury w których dopuszczamy połączenia pomiędzy różnymi warstwami (ang. *skip-connections* [26]). Topologię każdej sieci zwykle przedstawia się w postaci skierowanego grafu złożeń (patrz rysunek 2.3.1). Jeśli w grafie tym nie ma cykli, sieć taką nazywamy siecią *w przód* (ang. *feed-forward neural net*), w przeciwnym przypadku mamy do czynienia z siecią rekurencyjną (ang. *recurent neural net*).

2.3.2. Wielowarstwowa sieć neuronowa jako uniwersalny aproksymator

Opisane powyżej architektury nie są oczywiście jedyną formą złożenia ze sobą wielu perceptronów. Łączą one jednak ze sobą dwie podstawowe ważne cechy - prostotę parametryzacji zbiorem wag $w_i^{(l)}$ wraz z ogromną zdolnością aproksymacji. O niesamowitej możliwości aprok-

symacji przez sieci neuronowe mówi poniższe twierdzenie (dowód można znaleźć np. tu [11]).

Twierdzenie 2.3.1 *Niech $f : \mathbb{R}^n \to \mathbb{R}$ będzie dowolną funkcją ciągłą o zwartym nośniku. Dla każdego $\epsilon > 0$ istnieje wówczas dwuwarstwowa sieć neuronowa $f^* : \mathbb{R}^n \to \mathbb{R}$ z sigmoidalną funkcją aktywacji dla której :*

$$\sup_{x \in \text{supp} f} |f(x) - f^*(x)| \leq \epsilon.$$

Powyższe twierdzenie mówi nam o problemie regresji funkcji gładkich. Analogiczne twierdzenie ([12]) mówi nam, że dowolny problem klasyfikacji da się rozwiązać z dowolną dokładnością przy pomocy trójwarstwowej sieci neuronowej.

Własność uniwersalnej aproksymacji daje nam gwarancję, że dla każdego problemu, znajdziemy sieć neuronową która rozwiązuje go z dowolną dokładnością. Nie należy jednak przeceniać jej znaczenia - ich dowody nie przedstawiają nam żadnej skutecznej metody do znajdowania architektury oraz odpowiednich parametrów sieci.

2.3.3. Algorytm wstecznej propagacji

To, że dołożenie kolejnych warstw zwiększa możliwości perceptronu wiedziano już w latach '60 XXw. To co przyczyniło się do przełamania złego trendu, który nastał po rozczarowaniu związanym z ograniczeniami perceptronu był kolejny skuteczny algorytm uczenia. W tym przypadku kluczem stał się algorytm wstecznej propagacji, działający dla architektur typu *feed-forward*. I ponownie - to co uderza, to już nie prostota samego algorytmu, lecz prostota koncepcji, która stała za jego wynalezieniem. To co okazało się kluczowym w tym przypadku to idea znanej z rachunku różniczkowego - czyli reguła łańcuchowa.

W swej najprostszej postaci algorytm ten jest kolejnym praktycznym zastosowaniem metody optymalizacji gradientowej. W sensie obliczeniowym, algorytm ten wykorzystuje brak cyklów w grafie obliczeń, co znacząco ułatwia obliczenia pochodnej błędu ze względu na każdy parametr $w_i^{(j)}$. Spróbujmy prześledzić jak wygląda obliczanie tych pochodnych, w przypadku gdy za funkcję błędu przyjmiemy błąd średniokwadratowy, a za funkcję aktywacji funkcję sigmoidalną dla sieci z jednym neuronem w wartstwie wyjściowej. Przy zadanym zbiorze $\mathcal{X}$ wartość pochodnej funkcji błędu ze względu na wyniki warstwy wyjściowej $y_x^{(out)} = f^*(x)$, gdy dla ustalonego $x \in \mathcal{X}$ prawidłowa wartość wynosi y_x jest zadana wzorem :

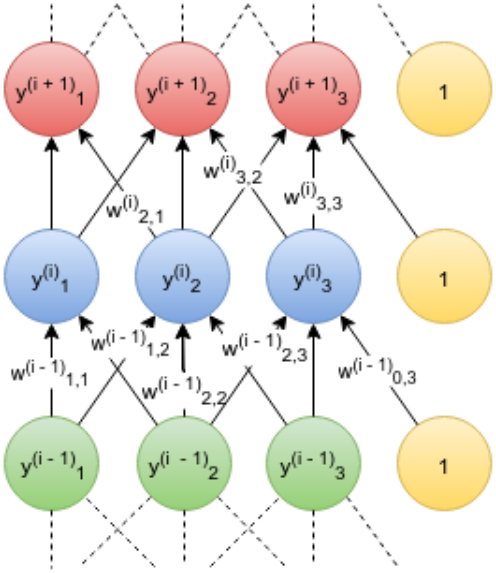
$$\frac{\partial J_{\mathcal{X}}(\Theta)}{\partial y^{(out)}} = \frac{\partial \left(\frac{1}{2|\mathcal{X}|} \sum_{x \in \mathcal{X}} \left(y_x^{(out)} - y_x \right)^2 \right)}{\partial y^{(out)}} = \sum_{x \in \mathcal{X}} \left(y_x^{(out)} - y_x \right).$$

Załóżmy teraz, że dla ustalonej k warstwy znamy pochodną ze względu na każde wyjście $y_i^{(k)}$ czyli :

$$\frac{\partial J_{\mathcal{X}}(\Theta)}{\partial y_i^{(k)}}.$$

Przyjmując wtedy za $x_i^{(k-1)} = \sum_{j=1}^{n_{k-1}} w_{j,i}^{(k-1)} y_j^{(k-1)}$ Mamy wówczas (korzystając z reguły łańcuchowej) :

$$\frac{\partial J_{\mathcal{X}}(\Theta)}{\partial w_{i,j}^{(k-1)}} = \frac{\partial J_{\mathcal{X}}(\Theta)}{\partial y_i^{(k)}} \frac{\partial y_i^{(k)}}{\partial x_i^{(k-1)}} \frac{\partial x_i^{(k-1)}}{\partial w_{i,j}^{(k-1)}}.$$



Rysunek 2.5: Rysunek pomocniczy dla procesu wstecznej propagacji.

Powyższy wzór, korzystając z własności funkcji sigmoidalnej upraszcza się do :

$$\frac{\partial J_{\mathcal{X}}(\Theta)}{\partial w_{i,j}^{(k-1)}} = \frac{\partial J_{\mathcal{X}}(\Theta)}{\partial y_i^{(k)}} \left(1 - x_i^{(k-1)}\right) x_i^{(k-1)} y_j^{(k-1)}.$$

Reguła łańcuchowa ponadto pozwala nam obliczyć wartość:

$$\frac{\partial J_{\mathcal{X}}(\Theta)}{\partial y_j^{(k-1)}} = \sum_{i=1}^{n_{k-1}} \frac{\partial J_{\mathcal{X}}(\Theta)}{\partial y_j^{(k)}} \frac{\partial y_j^{(k)}}{\partial y_i^{(k-1)}} = \sum_{i=1}^{n_{k-1}} \frac{\partial J_{\mathcal{X}}(\Theta)}{\partial y_j^{(k)}} \frac{\partial y_j^{(k)}}{\partial x_i^{(k-1)}} \frac{\partial x_i^{(k-1)}}{\partial y_j^{(k-1)}}. \tag{2.1}$$

Po uproszczeniu (korzystając z własności funkcji sigmoidalnej) otrzymujemy :

$$\frac{\partial J_{\mathcal{X}}(\Theta)}{\partial y_j^{(k-1)}} = \frac{\partial J_{\mathcal{X}}(\Theta)}{\partial y_i^{(k-1)}} \sum_{i=1}^{n_{k-1}} \left(1 - x_i^{(k-1)}\right) x_i^{(k-1)} w_{i,j}^{(k-1)}.$$

Wartości te możemy następnie wykorzystać do obliczenia pochodnych ze względu na wagi z niższych warstw. Zatem na mocy zasady indukcji matematycznej możemy policzyć gradient funkcji kosztu ze względu na dowolny parametr.

Zauważmy, że w powyższe rozumowanie z łatwością można rozszerzyć na przypadek sieci w których istnieją połączenia między warstwami. Wzory te - w ogólnej postaci, można również rozszerzyć na inne przypadki funkcji błędu oraz aktywacji.

Warto również podkreślić, że technika ta stosowana jest także w przypadku rekurencyjnych sieci neuronowych, przez odpowiednie przedstawienie sieci rekurencyjnej, jako sieci typu *feed-forward* z dodatkowym kryterium, aby część wag w sieci miały taką samą wartość.

Zjawisko przesunięcia kowariancji (ang. *covariate shift*) i technika normalizacji porcjowej (ang. *Batch Normalization*)

Dosyć poważną wadą techniki wstecznej propagacji jest występowanie tzw. zjawiska przesunięcia kowariancji w sieciach typu *feed-forward*. Intuicyjnie polega ono na tym, że podążanie w kierunku najmniejszego spadku w dwóch kolejnych warstwach może doprowadzić do tego,

że późniejsza z nich będzie oczekiwała kompletnie innych wartości niż te które dostarcza jej warstwa poprzednia. Często znacznie pogarsza to stabilność oraz efektywność procesu uczenia.

Aby uniknąć tego zjawisko często stosuje się tzw. *technikę normalizacji porcjowej* (ang. *Batch Normalization* [32]) gdzie każda kolejna porcja danych używana w procesie uczenia - jest uśredniana przez swoją średnią porcjową y_p^k:

$$\bar{y}^{(k)} = y^{(k)} - y_p^{(k)},$$

standaryzowana przez porcjowe odchylenie standardowe σ_p^k:

$$\bar{\bar{y}}^{(k)} = \bar{y}^{(k)} - y_p^{(k)},$$

a następnie przesuwana o wyuczalny wektor $m_p^{(k)}$ i skalowana przez wyuczalny czynnik $c_p^{(k)}$:

$$y_b^{(k)} = \bar{\bar{y}}^{(k)} * c_p^{(k)} + m_p^{(k)}.$$

Odbywa się nie tylko na poziomie normalizacji danych wejściowych, ale także pomiędzy każdymi kolejnymi warstwami. Każde wejście do kolejnej warstwy ma taką samą średnią $m_p^{(k)}$ oraz odchylenie standardowe $c_p^{(k)}$ dzięki czemu zjawisko *covariate shift* zostaje znacząco ograniczone. W dniu dzisiejszym *normalizacja porcjowa* uznawana jest za obowiązkową technikę normalizacji większości architektur sieci neuronowych.

2.3.4. Interpretacja probabilistyczna sieci neuronowych

Dosyć interesującym spojrzeniem na zagadnienie poszukiwania optymalnych wartości parametrów sieci neuronowej, jest przedstawienie tego zagadnienia optymalizacyjnego jako maksymalizowanie funkcji wiarygodności (dokładnie jej logarytmu) modeli sposród ustalonej rodziny. Metodę tę, znaną ze statystyki jako estymacja metodą największej wiarygodności (*ang.* maximum likelihood estimation (MLE)). Rozważmy to poniższym przykładzie.

Przykład 2.3.1 *Rozważmy przez f_θ parametryzowaną wagami $\theta \in \Theta$ sieć neuronową o ustalonej architekturze typu $feedforward$. Rozważmy wówczas rodzinę rozkładów warunkowych zadanych wzorem :*

$$\mathbb{P}(y|\theta, x) \sim \mathcal{N}(f_\theta(x), \sigma),$$

dla pewnej ustalonej liczby $\sigma > 0$. Możemy to zinterpretować w ten sposób, że przy ustalonym x, rozkład y przedstawia się jako :

$$y = f_\theta(x) + \mathbf{X},$$

gdzie $\mathbf{X} \sim \mathcal{N}(0, \sigma)$. Załóżmy teraz, że nasz zbiór treningowy $\mathcal{X}$ wyznacza zbiór do zadania regresji i możemy go przedtawić w postaci :

$$\mathcal{X} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \subset \mathbb{R}^n \times \mathbb{R}.$$

Zauważmy, że jeśli przyjmiemy, że elementy naszego zbioru treningowego są niezależne oraz zgodne z powyższym rozkładem prawdopodobieństwa, to wówczas funkcja wiarygodności ustalonego zestawu wag $\theta \in \Theta$ wyraża się wzorem:

$$\mathcal{L}(\theta, \mathcal{X}) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-f_\theta(x))^2}{2\sigma}},$$

a zatem logarytm funkcji wiarygodności wyraża się wzorem :

$$\log \mathcal{L}(\theta, \mathcal{X}) = -\frac{1}{2\sigma} \sum_{i=1}^{n} (y - f_\theta(x))^2 + C,$$

gdzie $C = -\frac{n}{2} \log 2\pi\sigma^2$. *Widzimy zatem, że zadanie maksymalizacji logarytmu funkcji wiarygodności jest równoważne rozwiązaniu zadania minimalizacji średniokwadratowej funkcji błędu dla rodziny funkcji neuronowych* f_θ, *albowiem :*

$$\arg\max_{\theta\in\Theta} -\frac{1}{2\sigma} \sum_{i=1}^{n} (y - f_\theta(x))^2 + C = \arg\min_{\theta\in\Theta} \frac{1}{2n} \sum_{i=1}^{n} (y - f_\theta(x))^2 .$$

Analogia ta pozwala często skorzystać z wygody dualnego patrzenia na zadanie uczenia - jako z jednej strony zagadnienia stricte optymalizacyjnego, a z drugiej - pokrewnego metodą statystycznym. Pozwala to skorzystać z wygodnych intuicji probabilistycznych, a także np. na łatwe korzystanie z metod Bayesowskich w zagadnieniach machine learningu.

Oczywiście funkcja średniokwadratowa nie jest jedyną funkcją błędu, dla której powyższa analogia ma miejsce. Jeśli np. zarządamy aby :

$$\mathbb{P}(y|\theta, x) \sim \mathcal{L}(f_\theta(x), 1),$$

gdzie $\mathcal{L}(f_\theta(x),\ 1)$ to rozkład Laplace'a z parametrami $f_\theta(x)$ oraz 1, równoważną metodzie MLE metodą optymalizacji będzie minimalizacja z następującą funkcją błędu :

$$J_{\mathcal{X}}(\theta) = \sum_{x\in\mathcal{X}} \|f_\theta(x) - y\|_1.$$

Widzimy, że rozumowanie to można uogólnić niemal dla każdej funkcji błędu, dla której istnieje taki rozkład prawdopodobieństwa $p(\theta, x)$, że

$$J_{\mathcal{X}}(\theta) = -C \sum_{x\in\mathcal{X}} \log p(x, \theta) + D,$$

gdzie $C, D \in \mathbb{R}$ to pewne stałe. Przykład zastosowania intuicji probabilistycznej możemy zobaczyć już w poniższym rozdziale.

2.3.5. Regularyzacja sieci neuronowych

W tej podsekcji chciałbym opisać niektóre prostsze metody regularyzacji stosowane w sieciach neuronowych. Przez regularyzację rozumiemy działania mające na celu zmniejszenie skutku procesu przeuczenia. W kolejnych rozdziałach poznamy inne metody regularyzacji (w tym te, które są esencjonalne dla algorytmów *deep learning*), w tym rozdziale jednak przedstawimy elementarne metody radzenia sobie ze zjawiskiem *overfittingu*.

Kara związana z $\|\Theta\|_2$

Podstawową metodą regularyzacji jest zmiana funkcji błędu poprzed dodanie dodatkowego składnika, który zwiększa wartość błędu gdy norma Euklidesowa wektora wag jest duża. Prowadzi to do nowej funkcji błędu zadanej zazwyczaj wzorem:

$$J_{\mathcal{X}}^{R}(\Theta) = J_{\mathcal{X}}(\Theta) + \frac{\lambda}{2}\sum_{\theta\in\Theta}\theta^2$$

lub stosując zapis wektorowy :

$$J_{\mathcal{X}}^{R}(\Theta) = J_{\mathcal{X}}(\Theta) + \frac{\lambda}{2}\Theta\Theta^T.$$

dla pewnego $\lambda \in \mathbb{R}$ Warto zauważyć, że stosując probabilistyczną interpretację sieci neuronowych, minimalizacja powyższej funkcji błędu jest równoważna maksymalizacji funkcji ufności przy założeniu rozkładu *a priori* na zbiorze parametrów Θ:

$$\theta_i \sim \mathcal{N}\left(0, \frac{1}{\lambda}\right), iid.$$

Załóżmy bowiem, że istnieje rozkład $p(\theta, x)$ dla którego :

$$J_{\mathcal{X}}(\theta) = -C\sum_{x\in\mathcal{X}}\log p(x,\theta) + D,$$

ale wówczas dokładając Bayesowskie założenie o wartości parametrów θ otrzymujemy :

$$J_{\mathcal{X}}^{R}(\theta) = -C\sum_{x\in\mathcal{X}}\log\left(\frac{p(x,\theta)}{p(\theta)}\right) + D = -C\sum_{x\in\mathcal{X}}\log\left(p(x,\theta)\right) + \frac{\lambda}{2}\sum_{\theta\in\Theta}\theta^2 + D',$$

co jest równoważne powyższej formie funkcji kosztu.

Kara związana z $\|\Theta\|_1$

Inną często stosowaną metodą regularyzacji jest tzw. metoda *lasso*, polegająca na dodaniu do funkcji błędu dodatkowego składnika zwiększającego funkcję błędu gdy tym razem pierwsza norma wektora wag jest duża. Funkcję tę możemy zapisać w postaci :

$$J_{\mathcal{X}}^{R}(\Theta) = J_{\mathcal{X}}(\Theta) + \sum_{\theta\in\Theta}|\theta| = J_{\mathcal{X}}(\Theta) + \lambda\|\Theta\|_1.$$

Analogicznie jak w poprzednim przypadku, możemy potraktować dodatkowy składnik jako dołożenie Bayesowskiego założenia o rozkładzie parametru wedle wzoru :

$$\theta \sim \mathcal{L}\left(0, \frac{1}{\lambda}\right),$$

gdzie przez $\mathcal{L}(a,b)$ oznaczamy rozkład Laplace'a z parametrami a, b.

2.3.6. Czym różnią się powyższe kary?

Warto wspomnieć o dosyć poważnych koncepcyjnych różnicach pomiędzy powyższymi wagami. Przyjmuje się ([16]), że o ile kara $\|\Theta\|_2$ stara się niedopuszczać aby wartości θ_i były wyraźnie większe od 0 (funkcja kwadratowa rośnie szybko dla liczb $\gg$ 1, o tyle kara $\|\Theta\|_1$ sprawia, że znacznie częściej duża część parametrów zbiór parametrów θ_i jest zaniedbywalnie większa od 0. Dlatego też tę drugą metodę zwykło nazywać się metodą *lasso* i często stosuje się ją do eliminacji zbędnych parametrów, poprzez odrzucenie tych dla których w wyniku uczenia wartości otrzymane spełniały $\theta \approx 0$.

2.3.7. Intuicje dotyczące warstw sieci neuronowych

To, że trójwarstwowa sieć neuronowa może poradzić sobie z praktycznie każdym zadaniem *machine learningowym* jest faktem, udowodnionym teoretycznie, jednakowoż dowód ten jest niepraktyczny w tym sensie, że dla zadanego zadania pokazuje istnienie rozwiązania, bez dokładnego przepisu na jego uzyskanie. Poniżej chciałbym pokazać intuicyjny szkic dowodu tego, że dowolny zbiór otwarty w $\mathbb{R}^n$ można przybliżyć przy pomocy sieci z dwiema warstwami ukrytymi :

1. Jednostki z sigmoidalną funkcją aktywacji, których argumentami są neurony z warstwy wejściowej, są w stanie reprezentować przybliżone funkcje charakterystyczne półprzestrzeni kowymiaru 1 (jeśli argumenty $x \in \mathbb{R}^n$.

2. Pojedynczy perceptron, z sigmoidalną funkcją aktywacji, jest w stanie w przybliżony sposób naśladować operacje logiczne typu AND oraz OR jeśli swoje argumenty potraktuje jako przybliżone wartości *boolowskie*.

3. W związku z powyższym, sieć neuronowa z jedną warstwą ukrytą i sigmoidalną funkcją aktywacji jest w stanie nauczyć się funkcji charakterystycznej dowolnego zbioru będącego przecięciem półprzestrzeni (w szczególności - kostki $k-$wymiarowej, gdzie $k \leq n$).

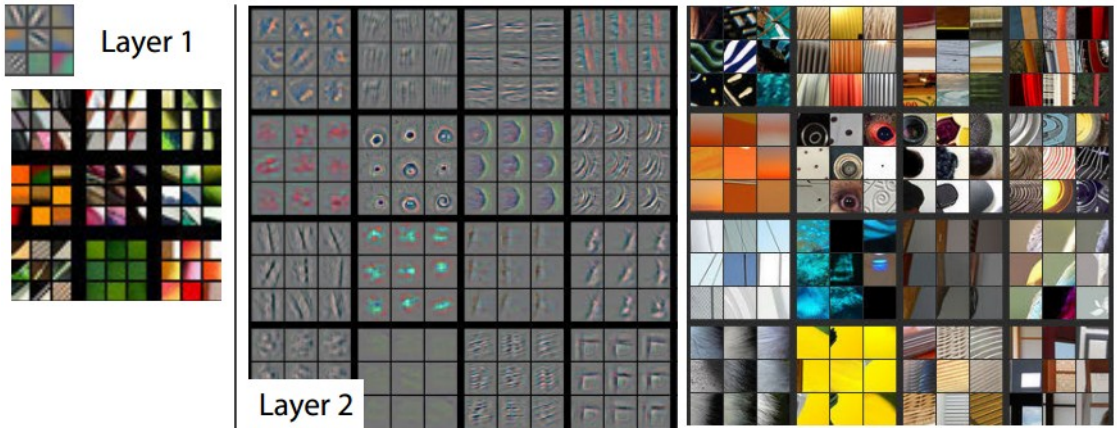
4. Sieci neuronowe z dwiema warstwami ukrytymi potrafią nauczyć się zatem przybliżonej funkcji charakterystycznej dowolnej funkcji będącą sumą, przecięciem zbiorów których funkcje charakterystyczne mogą zostać uzyskane przy pomocy sieci z podpunktu 3. W szczególności - z dowolnym przybliżeniem - dowolny zbiór otwarty $\mathcal{U} \subset \mathbb{R}^n$.

Powyższy dowód uznaję za wartościowy z dwóch powodów :

1. Pokazuje jak wielowarstwowa sieć neuronowa koduje hierarchiczny zbiór cech, który następnie wykorzystuje do budowania ostatecznego rozwiązania.

2. W związku ze swą ideą pokazuje, że niekonstruktywne dowody własności uniwersalnej aproksymacji opierają się li tylko na budowaniu lokalnych przybliżeń ostatecznego rozwiąznia, co koniec końców może doprowadzić do tzw. *przekleństwa wymiarowości* (ang. *curse of dimensionality* [27]).

Oba powyższe wnioski rozwiniemy w kolejnych rozdziałach.



Rysunek 2.6: Wizualizacja cech wyuczonych przez tzw. konwolucyjną sieć neuronową z [29].

2.3.8. Reprezentacja rozproszona

Idea działania sieci opiera się na budowaniu opisanej w poprzedniej podsekcji - struktury *cech* potrzebnych do rozwiązania zadanego problemu. Z jednej strony - stanowi to klucz do zrozumienia ogromnego sukcesu sieci neuronowych w ostatnim czasie. Fakt, że jeden spójny algorytm uczenia może wyręczyć badacza z wyszukiwania cech niezbędnych do rozwiązania danego problemu (np. przy wspomnianym we wstępie zagadnieniu rozpoznawania obrazu ekstrakcja takich cech była przez wiele lat ciężką i dobrze płatną pracą), przerzucając ciężar pracy na zaprojektowanie odpowiedniej architektury sieci, funkcji kosztu dla procesu optymalizacji, a potem na moc obliczeniową współczesnych komputerów. Widzimy zatem, że automatycznie budowany zbiór cech stanowi klucz do zrozumienia działania wielowarstwowych sieci neuronowych.

W klasycznych warstwowych sieciach neuronowych typu *feed-forward* zbiór tych cech stanowi hierarchiczny konglomerat cech, w którym cechy reprezentowane w danej warstwie tworzone są przez transfomacj cech reprezentowanych w warstwie poprzedniej (2.3.8, [29]). Jeden z przykładów takiej hierarchiczości przedstawiłem w szkicu dowodu aproksymacyjnego z poprzedniej podsekcji.

Koncepcyjnie - zrezygnowanie z warstwowości, przy zachowaniu typu *feed-forward* pozwala budować nowe cechy przy pomocy konceptów wyuczonych przez niższe warstwy. Dalsza relaksacja założeń, czyli dopuszczenie cyklów w architekturze pozwala na budowanie cech oraz definicyjnych powiązań między nimi. W praktyce jednak, obliczeniowe własności architektury *feed-forward* sprawiają, że to właśnie te sieci stanowią współcześnie awangardę wśród najskuteczniejszych rozwiązań zadań *machine learningu*. Nawet sieci rekurencyjne prezentuje się z tej przyczyny, w tej formie, dodając odpowiednie ograniczenia i warunki na wartości odpowiednich wag ([28]).

Głęboka hierarchiczna struktura przyczyną sukcesu algorytmów z rodziny Deep Learning

Przyjmuje się, że podstawową przyczyną ogromnego sukcesu głębokich sieci jest efektywna umiejętność uchwycania hierarchicznych struktur wiedzy, która stanowi podstawę kompleksowości wielu z najważniejszych współcześnie problemów *machine learning'owych* ([17]). Prawidłowe zrozumienie rządzących tym prawideł to nadal wielkie wyzwanie dla badaczy zajmujących się sztuczną inteligencją. Możemy jednak na kilku przykładach przyjrzeć się dlaczego

uchwycenie takiej struktury może okazać się zbawienne dla efektywnego nauczenia modelu dla wielu ważnych zagadnień współczesnego *AI* ([30]):

- Wiedza opisująca np. rozpoznawanie twarzy ma hierarchiczną strukturę. Podstawowe pojęcia takie jak kreski, łuki, zabarwienia - służą do budowania bardziej złożonych pojęć takich jak oczy, policzki - które z kolei tworzą rozmaite rysy, twarze charakterystyczne dla rasy, płci, etc.
- Wiedza opisująca rozpoznawanie mowy również ma podobną strukturę - Podstawowe pojęcia tj. głoski biorą udział w tworzeniu fonemów, które składają się na słowa tworzące zdania etc.

Przyjmuje się, że właśnie taka postać wiedzy - naturalnie uchwycana przez model o hierarchicznej strukturze - stanowi podstawę sukcesów *głębokich sieci*. W codziennej praktyce często potwierdza się - że modele takie nie działają tak efektywnie dla problemów o prostszej strukturze - chociażby ze względu na złożoność przestrzeni parametrów, a także - złożoność obliczeniową procesu uczenia.

Rozdział 3

Sieci konwolucyjne

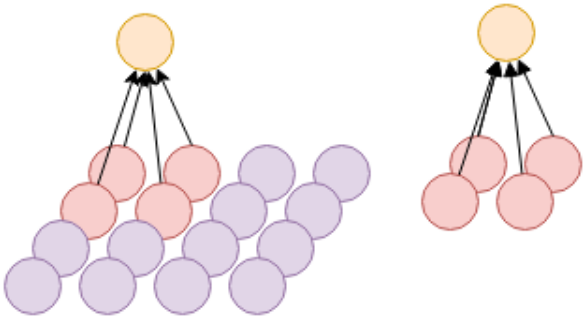
Jednym z najpopularniejszych zastosowań metod *głębokiego uczenia* są tzw. *konwolucyjne sieci neuronowe* (ang. *convolutional neural networks - CNN*). Ze względu na swoją architekturę są one niezwykle skuteczne w niemal wszystkich zagadnieniach wizji komputerowej (ang. *computer vision - CV*). W poniższym rozdziale zaczniemy od opisu zjawiska *inwariancji* - ważnego dla zrozumienia ogromnego sukcesu sieci konwolucyjnych. Następnie dokładnie zdefiniujemy sieci, a także rozmaite pojęcia pojawiające się w nomenklaturze ich architektur. Następnie po krótce opiszemy najważniejsze historycznie sieci - aby na końcu dokładnie opisać tzw. sieci residualne - stanowiące podstawę implementacji sieci neuronowej opisanej w tej pracy.

3.1. Inwariancja

W rozdziale opisującym szczegóły działania sieci neuronowych wspomnialiśmy o tym, że struktura sieci neuronowej wraz z wagami ją parametryzującymi koduje pewną ustaloną hierarchię cech niezbędnych do rozwiązania postawionego przed nią problemu. Powinny one zatem odwzorowywać większość własności, które posiadają rzeczywiste struktury rozważanych problemów. Jedną z takich własności, którą posiada wiele zadań do których rozwiązania wykorzystujemy techniki *deep learning* stanowi tzw. własność *inwariancji*, czyli zamkniętość pewnej cechy na przekształcanie argumentów przy pomocy ustalonej rodziny przekształceń. I tak np. cecha bycia okiem przez fragment obrazu lub zdjęcia jest zamknięta na przekształcenia skalowania, przesunięcia oraz obroty. Kwestia bycia dźwiękiem słowa *mama* jest zamknięta na zmiany tonu głosu, głośność, a także umiejscowanie na ścieżce dźwiękowej. Przykłady można mnożyć - jednak zauważmy, że własność inwariancji pociąga za sobą dwie bardzo poważne konsekwencje :

- Jeśli uda nam się uchwycić tę inwariancje przy pomocy struktury sieci, możemy zyskać ogromną kompresje wiedzy, gdyż ogromna mnogość cech, spośród których każda powstała jako przekształcenie innej, będzie reprezentowana w sieci jako jedna.
- W przypadku niemożności wychwycenia takiej inwariancji, skala trudności dramatycznie ponieważ :
  - Każda *realizacja* danej abstakcyjnej cechy musi być kodowana w sieci osobno.
  - Dla każdej z takich realizacji, musi istnieć co najmniej jeden element w zbiorze treningowym, w którym one występuje.

Aby zrozumieć skalę problemu, spróbujmy sobie wyobrazić przypadek w którym chcielibyśmy nauczyć sieć neuronową rozpoznawania oka na obrazku, w przypadku w którym nasza



Rysunek 3.1: Przykład dwuwymiarowej sieci konwolucyjnej. Obliczenia można interpretować jako złożenie splotu wag $w_1, \ldots, w_n$ z wartościami z dwuwymiarowej macierzy oraz funkcji aktywacji.

sieć nie mogłaby realizować własności *inwariancji* ze względu na przesunięcia na obrazku. W skrajnym przypadku musielibyśmy kodować własność bycia środkiem oka dla każdego piksela z osobno, co sprawia, że problem jest o rząd wielkości trudniejszy, niż w przypadku gdy moglibyśmy cechę bycia okiem zakodować raz i *zamknąć* naszą strukturę ze względu na przekształcenie translacji.

3.2. Architektura sieci konwolucyjnych

Sieci konwolucyjne to opisana w [8] i rozwinięta przez Yana LeCuna oraz Geoffrey'a Hintona architektura, która pozwala uchwycić inawariancję ze względu na translacje w obrębie wektora wejść. Matematyczna struktura stojąca za sieciami konwolucyjnymi jest banalnie prosta. Oznaczmy przez $P_{i_1,i_2,\ldots i_k} : \mathbb{R}^n \to \mathbb{R}^k$, gdzie $n \geq k$ operator rzutowania, tzn.

$$P_{i_1,i_2,\ldots i_k}(x_1, x_2, \ldots, x_n) = (x_{i_1}, x_{i_2}, \ldots, x_{i_k}).$$

Przez *warstwę konwolucyjną* z wagami $w_0, w_1, \ldots, w_n$, funkcją aktywacji ϕ i rzutowaniami:

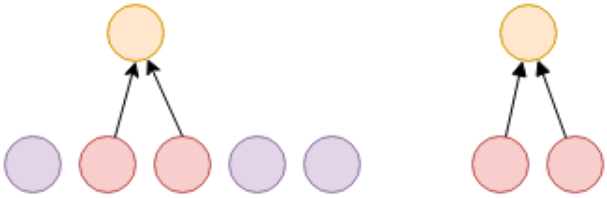
$$\mathsf{P} = \left(\left\{ i_1^{(1)}, i_2^{(1)}, \ldots, i_k^{(1)} \right\}, \left\{ i_1^{(2)}, i_2^{(2)}, \ldots, i_k^{(2)} \right\}, \ldots, \left\{ i_1^{(l)}, i_2^{(l)}, \ldots, i_k^{(l)} \right\} \right).$$

będziemy rozumieć funkcję $f_{\mathsf{P}}^{conv} : \mathbb{R}^n \to \mathbb{R}^k$, dla której $i-$ta współrzędna wektora $[f_{\mathsf{P}}^{conv}(x_1, x_2, \ldots . x_n)]_i$ zadana jest wzorem :

$$[f_{\mathsf{P}}^{conv}((x_1, x_2, \ldots . x_n)]_i = \phi \left(w_0 + \sum_{j=1}^{k} w_i x_{i_j} \right).$$

Zwróćmy uwagę, że powyższą warstwę można zinterpretować jako warstwę w której każdy neuron połączony jest tylko i wyłącznie z pewnym k-elementowym podzbiorem indeksów argumentu, a także że wszystkie neurony współdzielą wagi między sobą. Pozwala to na inwariancje w tym sensie, że wagi $w_0, w_1, \ldots, w_n$ uczestniczą w wykrywaniu pewnej cechy w kodowanych przez rzutowania regionach obrazu.

Przedstawiona przez nas definicja konwolucyjnych sieci neuronowych jest bardzo ogólna. W szczególności nie do końca jasnym może być skąd wzięło się określenie sieci jako *konwolucyjnej*. Pewnym wytłumaczeniem mogą być poniższe przykłady na poniższym obrazku przedstawiające dwa historyczne przykłady pierwszych topologii konwolucyjnych.



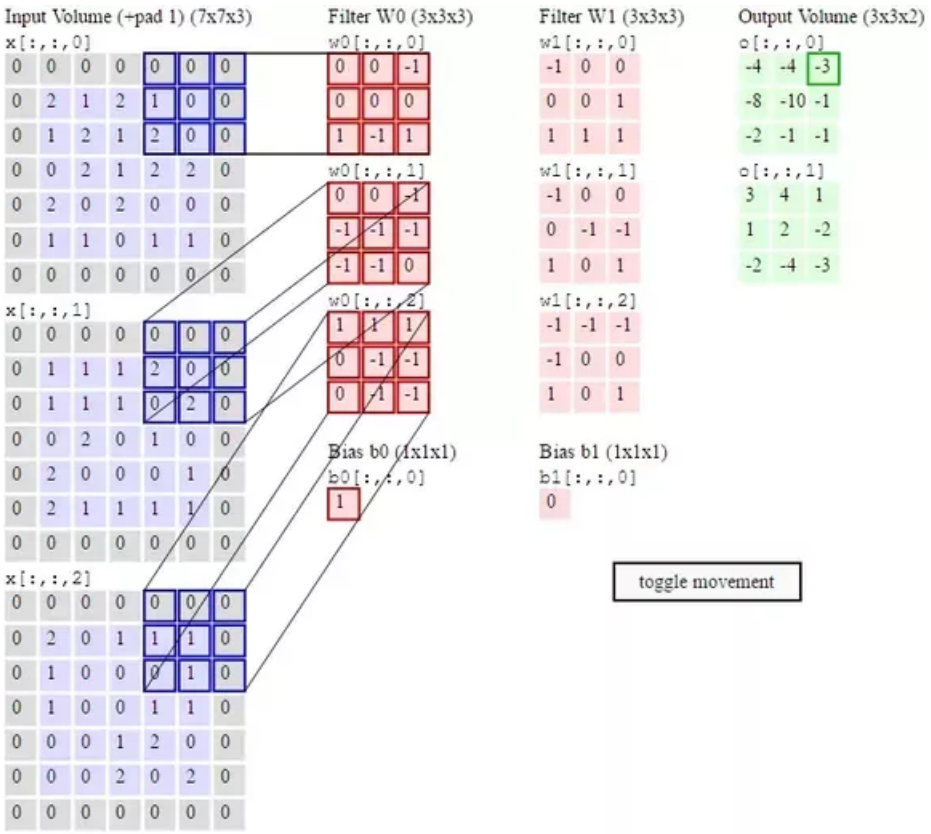
Rysunek 3.2: Przykład dwuwymiarowej sieci konwolucyjnej. Obliczenia można interpretować jako złożenie splotu wag $w_1, \ldots, w_n$ z wartościami z wektora oraz funkcji aktywacji.

Oczywiście warstwy konwolucyjne można ze sobą składać, jedna po drugiej. Prowadzi to do uczenia inwariantnych ze względu na translacje cech z innych inwariantnych na przesunięcia konceptów nauczonych przez niższe warstwy.

3.2.1. Najpopularniejsze rzutowania

Definicja sieci konwolucyjnych przedstawiona powyżej jest bardzo ogólna - w większości przypadków można rzutowania przedstawić w sposób znacznie bardziej zwięzły. W poniższych rozważaniach ograniczymy się do tzw. *konwolucji dwuwymiarowych* które można interpretować jako konwolucje obrazkowe. Przyjmijmy ponadto kilka dodatkowych założeń:

- Wejściem oraz wyjściem z warstwy konwolucyjnej będą macierze o kształcie (w, h, c), gdzie w to szerokość obrazka, h to wysokość, natomiast c to tzw. wymiar cech (lub kanałów - ang. *channels*). Ostatni wymiar możemy interpretować w sposób następujący:
  - W wypadku gdy argumentami całej sieci są obraz *RGB* wymiar kanałów ma wartość 3 i możemy je interpretować jako wektor zawierający wartości poszczególnych kolorów,
  - W wypadku gdy argumentami całej sieci są obraz czarnobiały - wymiar kanałów ma wartość 1 i możemy je interpretować jako jednowymiarowy wektor zawierający wartość odcienia szarości (ang. *grayscale*),
  - W wypadku warstw niewejściowych - wymiar kanałów należy interpretować jako kanał trzymający wartości otrzymane z obliczeń w poprzedniej warstwie.
- Przez *filtr* będziemy rozumieć pojedynczą współrzędną z wektora wyjściowego rzutowań. Jest on tożsamy z trójwymiarową macierzą o wymiarach (f_w, f_h, c) gdzie f_w będziemy określać mianem *szerokości* filtra, f_h będziemy określać mianem *wysokości* filtra, natomiast c jest równe wartości wymiaru kanałów warstwy wejściowej filtra wraz z funkcją aktywacji.
- Rzutowaniem o kroku (ang. *stride*) (s_w, s_h) oraz rozmiarze filtra (f_w, f_h, c) będziemy nazywać zbiór (dla wszystkich poniższych przypadków $c' \in \{0, \ldots, c-1\}$, $k \in \{0, \ldots, f_w - 1\}$ oraz $l \in \{0, \ldots, f_h - 1\}$ zakładamy, że dla wymiarów wykraczających poza rozmiar macierzy $x_{i,j,k} = 0$ oraz ustalamy liczenie jej wymiarów od 0):
  - $\left\{\left(s_w * i + k, s_h * j + l, c'\right) : i \in \{0, \ldots, \lfloor \frac{w}{s_w} \rfloor\}, j \in \{0, \ldots, \lfloor \frac{h}{s_h} \rfloor\}\right)\}$ dla tzw. *braku dopełnienia*, (ang. *valid padding*),



Rysunek 3.3: Przykład obrazujący działanie filtrów konwolucyjnych.

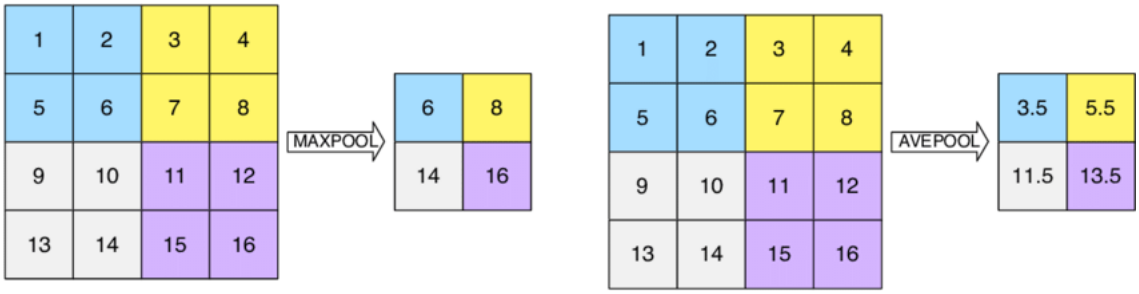
– $\left\{(s_w * i + k, s_h * j + l, c') : i \in \{-\lfloor \frac{f_w}{2} \rfloor, \ldots, \lfloor \frac{w+\lfloor \frac{f_w}{2} \rfloor}{s_w} \rfloor\}, j \in \{-\lfloor \frac{f_h}{2} \rfloor, \ldots, \frac{\lfloor \frac{h+f_h}{2} \rfloor}{s_h}\}\right\}$ dla tzw. *dopełnienia identycznego*, (ang. *same padding*).

Rzutowania te można zinterpretować jako wycinanie prostokątów rozmiaru (f_w, f_h) z obrazka. Na parametry (s_w, s_h) można natomiast spojrzeć ile kroków należy zrobić w lewo (s_w) lub w dół (s_h) aby wyciąć kolejny. Dopełnienia determinują czy wycinane kwadraty będą wycinane tylko i wyłącznie z macierzy wejściowej (brak dopełnienia) lub dodatkowe 0 będą dopisane dookoła obrazka / mapy cech w celu utrzymania rozmiaru wyjściowej mapy cech proporcjonalnego do rozmiaru macierzy wejściowej.

3.2.2. Pooling

W większości zastosowań krokiem większości warstw jest wektor $(1, 1)$, co przy zastosowaniu dopełnienia identycznościowego powoduje, że rozmiary (w, h) macierzy wejściowej i wyjściowej są identyczne. W związku z tym, że rozmiar c obydwu tych macierzy jest często większy od 100 powoduje poważne problemy pamięciowe przy obliczeniach wykonywanych w sieci. Jednym z rozwiązań tego problemu jest zastosowanie tzw. *poolingu*. Możemy zinterpretować go jako filtr konwolucyjny dla którego zachodzi $(f_w, f_h) \sim (s_w, s_h)$. Powoduje to $(s_w * s_h)$-krotne zmniejszenie rozmiaru macierzy wyjściowej. Pozwala to na znaczące ograniczenie pamięci potrzebnej do wykonania obliczeń.

Dwa najczęściej spotykane rodzaje *pooling* to tzw. *max-pooling* w którym funkcją aktywacją jest funkcja max oraz tzw. *sum-pooling* - gdzie jako aktywację stosuję się sumę po



Rysunek 3.4: Przykład obrazujący działanie *poolingu*.

wszystkich elementach rzutowania.

Pooling globalny (ang. *global pooling*)

Specjalnym przypadkiem *poolingu* jest tzw. *pooling globalny* (ang. *global pooling*), w którym $(f_w, f_h) = (w, h)$. Wraz z brakiem dopełnień powoduje to zmniejszenie rozmiaru wyjściowej macierzy do rozmiaru $(w, h) = (1, 1)$ co sprawia, że macierz taką możemy interpretować jako wektor. Jest to szczególnie przydatne w przypadku gdy rozwiązujemy zadanie np. klasyfikacji lub regresji, ponieważ ostatnie warstwy takich sieci muszą mieć naturę wektorową (klasa lub wartości $\mathbb{X}_y$ są najczęściej kodowane wektorowo) - więc w sieci musi istnieć moment, w którym kolejne warstwy zaczynają mieć tylko i wyłącznie naturę wektorową. Przykładem takiej warstwy jest właśnie *pooling globalny*.

3.3. Przegląd najpopularniejszych architektur konwolucyjnych

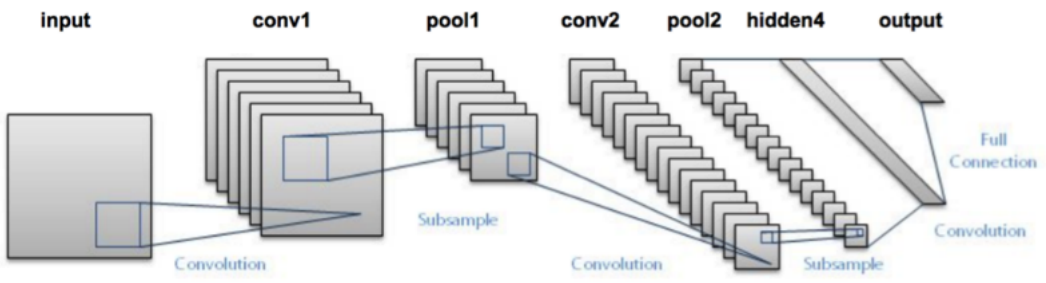
Społeczność badaczy sieci konwolucyjnych jest w dzisiejszych czasach bardzo szeroka i niesposób opisać wszystkie trendy i rozwiązania stosowane przy ich projektowaniu i wykorzystywaniu. Dlatego w tej sekcji przyjrzymy się najważniejszym historycznie sieciom wraz z analizą ich najważniejszych rozwiązań.

3.3.1. Pierwsze sieci konwolucyjne

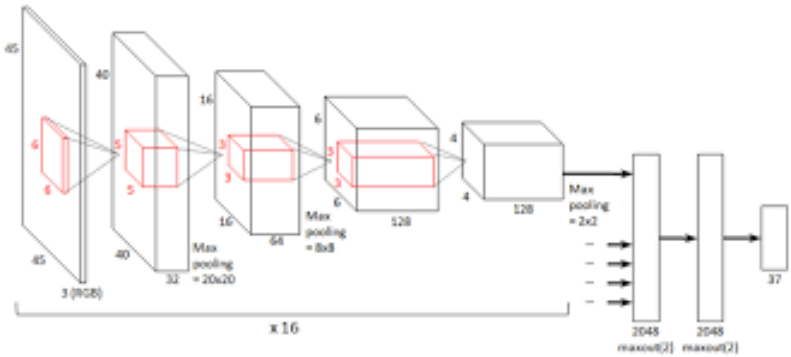
Biologiczą podstawę działania konwolucyjny sieci neuronowych odkryto i sformułowano ok. 1968r. gdy odkryto, że widzenie odbywa się przy pomocy tzw. pól receptywnych (ang. *receptive fields*) ([35]). Jednak za prawdziwą datę ich powstania uznaję się 1986r. i pracę ([7]) (*nota bene* tę samą w której zdefiniowano *wsteczną propagację*). Przez wiele lat jednak brak dostatecznie dużych zbiorów danych oraz mocy obliczeniowej nie pozwalał sieciom konwolucyjnym na pokazanie pełnego potencjału. Chlubnym wyjątkim była sieć LeNet ([34]) autorstwa Yana LeCuna która dzięki wsparciu biologów pracujących nad percepcją wzrokową w mózgu była w stanie uzyskać satysfakcjonujące rezultaty.

3.3.2. AlexNet

Za pierwszą z tzw. *nowoczesnych* sieci konwolucyjnych uznaje się tzw. sieć *AlexNet* ([36]) autorstwa Alexa Krizhevskiego. Osiągnęła ona fantastyczne rezultaty w ImageNet Large Scale Visual Recognition Challange 2012 [37] (przez wielu uznawanym za najważniejszy konkurs rozpoznawania obrazu na świecie) bijąc o kilkanaście procent dotychczasowe rekordy algorytmów opartch o klasyczne metody wizji komputerowej. Rozpaliła tym wyobraźnie badaczy



Rysunek 3.5: Schemat architektury sieci *LeNet-5*.



Rysunek 3.6: Schemat architektury sieci *AlexNet*.

kompletnie zmieniając krajobraz środowiska specjalistów od wizji komputerowej. W ciągu 3 lat - sieci konwolucyjne z interesującej ciekawostki stały się dominującym rozwiązaniem w kategorii rozpoznania obrazu. Przeanalizujmy najważniejsze innowacje wprowadzone przez A. Krizhevskiego w swojej pracy:

Recitified Linear unit - ReLU

Pierwszą z niach było wprowadzenie nowej funkcji aktywacji - tzw. *ReLU* (ang. *Rectified Linear Unit)*. Jest ona zadana niesłychanie prostym wzorem:

$$Relu(x) = x\mathbb{I}_{x>0},$$

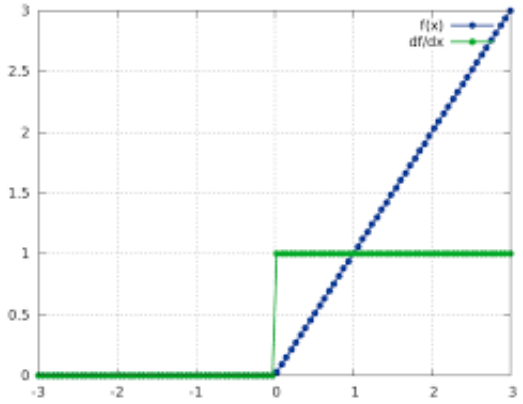
z równie prostym wzorem na słabą pochodną:

$$Relu'(x) = \mathbb{I}_{x>0}.$$

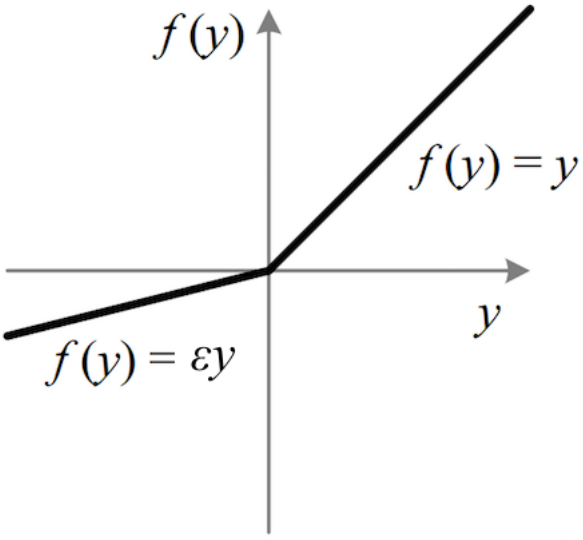
Co więcej - obliczenia potrzebne do ich policzenia są niewyobrażalnie proste - sprowadzają się *de facto* do wykonania pojedynczego porównania - przez co obliczenia wykonywane przy obliczaniu jej wartości, a także przy wstecznej propagacji były znacznie szybsze do popularnych w tym czasie funkcji tanh oraz funkcji sigmoidalnej (które wymagały chociażby obliczenia funkcji *exp*).

Leaky ReLU Jedną z największych wad funkcji *ReLU* była kompletna saturacja dla ujemnych wartości argumentów. Aby temu zaradzić wprowadzono tzw. funkcję *Leaky ReLU*. Jest to tak właściwie rodzina funkcji parametryzowana parametrem $\alpha \in (0, 1)$ zadana wzorem:

$$LeakyReLu_\alpha(x) = x(\mathbb{I}_{x>0} + \alpha\mathbb{I}_{x\leq 0}),$$



Rysunek 3.7: Wykres funkcji i słabej pochodnej *ReLU*.



Rysunek 3.8: Wykres funkcji *Leaky ReLU*.

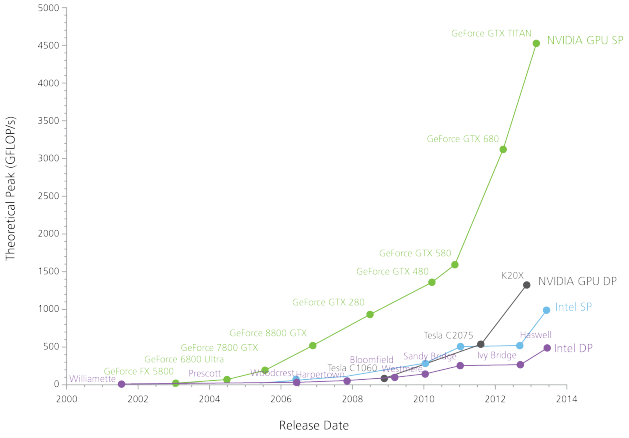
z pochodną:

$$LeakyReLu'_{\alpha}(x) = \mathbb{I}_{x>0} + \alpha\mathbb{I}_{x\leq 0}.$$

Dzięki mniejszej wartości współczynnika kierunkowego funkcja jednocześnie nie posiada własności kompletnej saturacji i nie jest liniowa. Ponadto jej obliczenie jej i jej gradientu nadal jest szybkie gdyż sprowadza się w najgorszym przypadku do porównania i mnożenia przez stałą.

Obliczenia na kartach graficznych

Kolejną przyczyną niewyobrażalnego sukcesu *AlexNet* był fakt, że obliczenia były efektywnie przeniesione na dwie karty graficzne. W związku z tym, że podstawą obliczeń w sieciach neuronowych są mnożenia macierzy - które są również podstawą wyświetlania obrazu przy pomocy jednostek *GPU* (ang. *Graphic Process Unit*) - Krizhevsky postanowił użyć ogromnej mocy obliczeniowej i optymalizacji dostarczanych przez głównych producentów kart graficznych w



Rysunek 3.9: Przyspieszanie kart graficznych oraz zwiększanie różnicy pomiędzy obliczeniami wykonywanymi na GPU / CPU.

celu przyspieszenia trenowania sieci. Ostatecznie trening *AlexNet* trwał ok. 2 tygodni co jak na tamten okres było czasem niewiarygodnie krótkim.

Co warto wspomnieć - od tego czasu wielu producentów kart graficznych (szczególnie NVidia) poświęca bardzo wiele pracy na optymalizację nowoczesnych kart graficznych specyficznie do *Deep Learningu* dzięki czemu obliczenia algorytmów sieci neuronowych z roku na rok przyspieszają coraz bardziej.

Augmentacja obrazów

Kolejną techniką która przyniosła ogromne poprawienie rezultatów była tzw. *augmentacja obrazów*. W kolejnych iteracjach obrazki podawane sieci były poddawane rozmaitym operacjom, w tym m.in.:

- powiększanie - zmniejszanie o losowy czynnik,
- wycinanie losowego prostokąta z obrazu,
- obrót o losowy kąt,
- losowa symetria wzdłuż ustalonych osi,
- przesunięcie kolorów o losowy składnik.

Wymagało to od sieci uchwycenia inwariancji nietylko ze względu na przesunięcia, ale także mnogość innych transformacji - ograniczając w ten sposób zjawisko przeuczenia. Co więcej - w związku z tym, że zbiór potencjalnych operacji jest nieograniczony (np. ze względu na możliwe kąty obrotu czy wartości przesunięcia) - augmentacja czyni zbiór danych treningowych potencjalnie nieskończonym.

Dropout

Sieci neuronowe - w związku ze swą potencjalnie ogromną pojemnością - często doświadczają zjawiska przeuczenia. Jednym z rodzajów takiego przeuczenia jest wyodrębnianie się w sieci konglomeratu jednostek które uczą się *de facto* na pamięć podrodziny przykładów. Uważa się,

Rysunek 3.10: Przykłady obrazków uzyskanych z pojedynczego obrazka ze zbioru treningowego przy pomocy losowych augmentacji.

że zjawisko takie pogarsza generalizację - dlatego w swojej pracy Krizhevsky po raz pierwszy zastosował specjalną metodę stworzoną w celu uniknięcia tego zjawiska. Technika ta zwana *dropoutem* [44] polega na losowym (z prawdopodobieństem $p \in (0, 1)$) wyłączeniu podczas pojedynczej iteracji uczenia każdego neuronu z osobna. Dzięki temu - tworzenie się takiego zespołu jednostek sieci jest utrudniane przez to, że w każdej z iteracji część takiej podsieci jest losowo wyłączana.

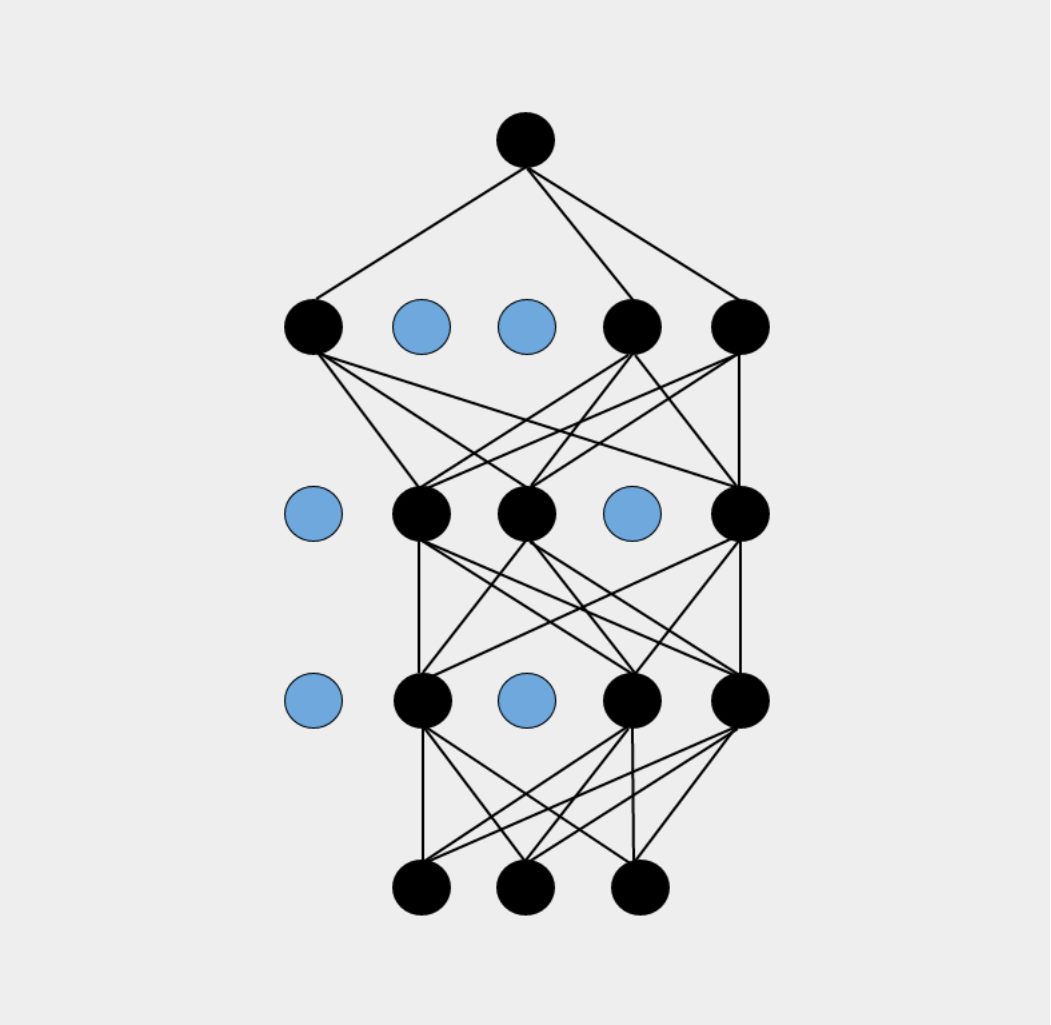
3.3.3. VGG

Kolejną ważną siecią w historii rozwoju nowoczesnych głębokich sieci konwolucyjnych była tzw. sieć *VGG* [38] autorstwa K. Simonyana i A. Zissermana z Oksfordzkiej grupy *Visual Geometry Group* (skąd nazwa). Występuje ona w różnych wersjach - w zależności od ilości warstw - *VGG-16* oraz *VGG-19*. Co ciekawe - znajduje się w tym zestawieniu ze względu na znaczące poprawienie rezultatu - w roku ogłoszenia nie pobiła nawet rekordu w ImageNet 2014 [37]. To co okazało się kluczowym dla jej popularności to niewiarygodnie prosta teleskopowa architektura - w której kolejne filtry mają ten sam rozmiar ($(f_w, f_h) = (3, 3)$), a wystąpienie *poolingu maksymalnego* kompensowane jest podwojeniem ilości filtrów. Układ ten stał się na tyle inspirujący, że po dziś dzień architektura jest zwykle pierwszym wyborem początkujących *developerów* sieci konwolucyjnych oraz naukowców z innych dziedzin próbujących wykorzystać *deep learning* do swoich zastosowań.

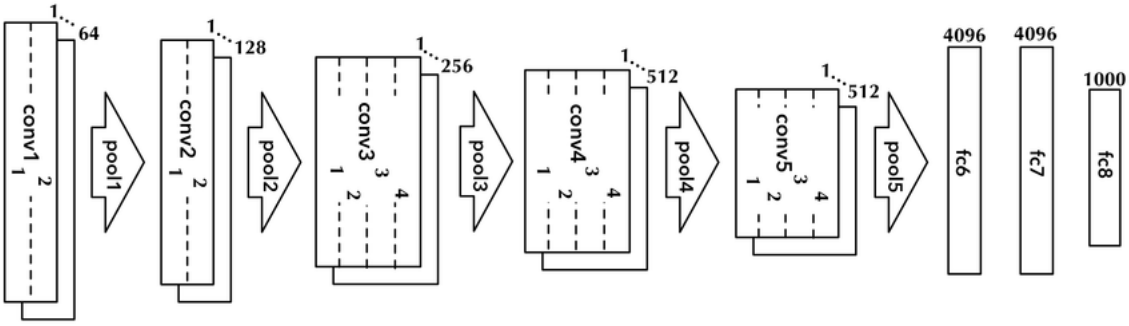
Co warte podkreślenia - aktualnie architektura powoli staje się lekko archaiczna ze względu na dużą ilość obliczeń wykonywanych przy maksymalnej rozdzielczości (w tym przypadku równej $(224, 224)$), co powoduje poważne problemy natury pamięciowej i czasowej.

3.3.4. GoogLeNet

W tym samym roku co *VGG* powstała inna rewolucyjna sieć zwanna *GoogLeNet* (aluzja do *LeNet*) lub *Incepcją* (od tytułu artykułu *Going Deeper with Convolutions* [39] oraz popularnego filmu *Incepcja* [40]). Przyjrzyjmy się bliżej rewolucyjnym zmianą zastosowanym w *Incepcji*:



Rysunek 3.11: Przykłady architektury sieci w której zastosowane technikę *dropout*.



Rysunek 3.12: Schemat architektury sieci *VGG-19*.

Rysunek 3.13: Schemat architektury sieci *Google Inception v1*

Łodyga (ang. *stem*) sieci

Przez *łodygę* (ang. *stem*) sieci rozumiemy początkowe warstwy sieci konwolucyjnej które dokonują znaczącej redukcji wymiarowości na przestrzeni 2-4 warstw. Tak jak wspominaliśmy przy okazji sieci *VGG* początkowe warstwy sieci cierpią z powodów pamięciowych ponieważ rozdzielczość map cech jest wtedy największa. Autorzy zastosowali po kolei:

1. Duże filtry z dwukrotnym *krokiem*: $(f_w, f_h) = (7, 7)$ i $(s_w, s_h) = (2, 2)$,
2. *Max Pooling* z $(f_w, f_h) = (3, 3)$ i $(s_w, s_h) = (2, 2)$,
3. Mały filtry z pojedynczym krokiem: $(f_w, f_h) = (3, 3)$ i $(s_w, s_h) = (1, 1)$,
4. *Max Pooling* z $(f_w, f_h) = (3, 3)$ i $(s_w, s_h) = (2, 2)$.

Dzięki tej technice po zaledwie czterech warstwach rozmiary przestrzenne map cech zostały zmniejszone po 8-kroć (z $(224, 224)$ do $(28, 28)$) co pozwala na znacznie efektywniejsze pamięciowo i czasowe obliczenia w głębszych częściach sieci.

Konwolucja $(f_w, f_h) = (1, 1)$ - redukcja wymiarowości

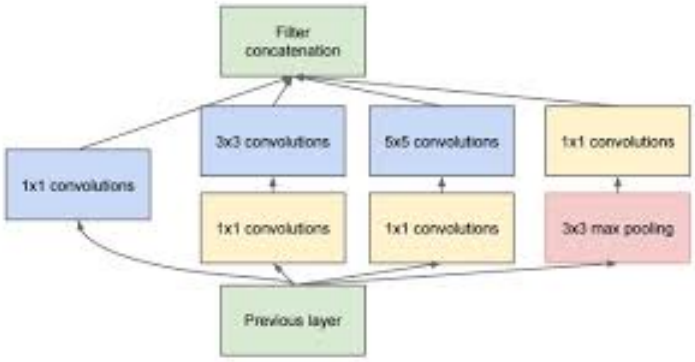
Na pierwszy rzut oka konwolucja z $(f_w, f_h) = (1, 1)$ może nie mieć większego sensu - gdyż jest *de facto* przekształceniem filtrów z pojedynczego piksela mapy cech. Pozwala jednakowoż wykonać dwie bardzo przydatne czynności:

- Pozwala na skuteczną rozmiaru wyjścia z filtrów - mogąc być interpretowana jako swoisty semantyczny *pooling* w wymiarze kanałów.
- Pozwala na adaptowalne dopasowanie wyjścia z wielu warstw sieci konwolucjynch w taki sposób aby posiadały dokładnie taką samą ilość filtrów wyjściowych.

Szczególnie ta druga własność uczyni konwolucję przydatnych zarówno dla *GoogLeNet* jak i opisanego w późniejszych częściach pracy *ResNeta*.

Klasyfikatory pomocnicze

Kolejną bardzo interesującą modyfikacją było wprowadzenie tzw. *klasyfikatorów pomocniczych*. Klasyfikatory pomocnicze to dodatkowe wyjścia z sieci (umieszczane w warstwach bliższych warstwie wejściowej) w których dodawane są pomocnicze warstwy wyjściowe rozmiaru takiego samego jak ostateczne wyjście z sieci - mające optymalizować dokładnie tę samą funkcję kosztu co jednostki wyjściowe sieci (często nadaje się tym kosztom niższą wagę). Celami stosowania klasyfikatorów pomocniczych są:



Rysunek 3.14: Przykład bloku incepcyjnego - *sieci w sieci*.

- Przekazywanie rozsądnego sygnału uczącego (gradientu) dla niższych warstw sieci - dla których *dotarcie* gradientu z ostatnich warstw może okazać się trudne przez co przyspieszane jest uczenie sieci,
- Zmuszenie niższych warstw sieci aby uczyły się *rozsądnych cech*, tj. takich które same w sobie mogą rozwiązać wyjściowy problem.

Blok incepcyjny - *sieć w sieci*

Najważniejszym wkładem *Incepcji* było jednakowoż wprowadzenie specjalnych bloków incepcyjnych - w których różnego rodzaju konwolucje były stosowane do tej samej warstwy wejściowej - po czym wyniki złączane były wzdłuż wymiaru kanałów. Pozwalało to na wykrywanie cech w różnych skalach, a przez to, że obliczenia mogły być wykonywane równolegle - równocześnie zwiększano skuteczność i pojemność sieci bez zbytniego spowalniania obliczeń.

Kolejne wersji Incepcji

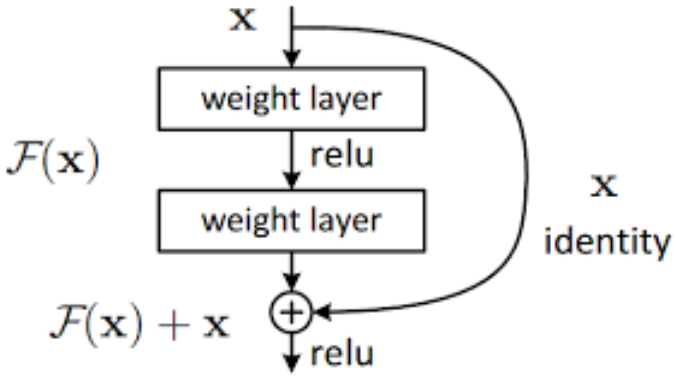
Warto wspomnieć, że sieć ta jest bujnie rozwijana. Obecnie dostępne jest jej już kilkanaście wersji. Opisywanie ich wykracza poza ramy tej pracy - o szczegółach można przeczytać w [32], [41] (warto wspomnieć o udziale naukowca z naszego wydziału - Zbigniewa Wojny) oraz [42].

3.3.5. Sieci residualne ResNet

Ostatnią z nowoczesnych architektur którą opiszemy w tej pracy jest *architektura połączeń residualnych - ResNet* [43]. Opracowana w laboratoriach firmy Microsoft sieć występuje w kilkunastu wersjach w zależności od ilości warstw - *ResNet 18, ResNet 34, ResNet 50, ResNet 101 i ResNet 151*. W architekturze tej wykorzystywane są elementy opisane już wcześniej - czyli klasyfikatory pomocnicze oraz łodyga. Rewolucyjnym elementem architektur residualnych jest natomiast tzw. *połączenie residualne* (ang. *residual connection*) które pozwoliło na osiąganie niewyobrażalnych wcześniej głębokości.

Połączenia residualne

Połączeniem residualnym nazywamy blok warstw konwolucyjnych w którym wyjście z tego bloku zostaje poddane jednej z dwóch następujących operacji:



Rysunek 3.15: Przykład połączenia residualnego.

- dodania do warstwy wejściowej (wymagane jest aby wyjście miało dokładnie taki sam rozmiar (w, h, c) jak wejście,
- dołączenia do warstwy wejściowej (wymagane jest aby wyjście miało takie same w i h jak wejście).

W architekturach *ResNet* wykorzystywany jest blok z operacją dodawania. Intuicyjnym wytłumaczeniem skuteczności działania połączeń residualnych w tym wypadku jest to, że sieć uczy się ciągu kolejnych zaburzeń poprzednich warstw które prowadzą do dobrego wyniku, zamiast ciągu transformacji prowadzącego do celu (transformacje te mogą kompletnie gubić naturę swoich argumentów - często znacznie ułatwiającą rozwiązanie zadania). Dzięki temu - problemy z przekazywaniem sygnału uczącego w niższe warstwy sieci są znacznie mniejsze - co pozwala na znaczące zwiększenie głębokości sieci.

Bibliografia

[1] DeepFace: Closing the Gap to Human-Level Performance in Face Verification, Yaniv Taigman, et al. Conference on Computer Vision and Pattern Recognition (CVPR) 24 czerwca 2014.

[2] Recognizing Traffic Signs Using a Practical Deep Neural Network, Hamed H. Aghdam, et al. Robot 2015: Second Iberian Robotics Conference, 2 grudnia 2015.

[3] Gradient-based learning applied to document recognition, LeCun et al., Proceedings of the IEEE, 86(11):2278-2324, November 1998.

[4] The Perceptron: A probabilistic model for information storage and organization in the brain, Frank Rosenblatt, Psychological Review, Vol.65, No. 6, 1958

[5] A logical calculus of the ideas immament in nervous activity, Warren S.McCulloch and Walter Pitts, Bulletin of Mathematical Biophysics, vol.5 1943

[6] Perceptrons: An Introduction to Computational Geometry, Marvin Minsky i Seymour Papert, The MIT Press, Cambridge MA

[7] Learning representations by back-propagating errors, David E. Rumelhart et al., Nature, 323 (6088) Październik 1986

[8] Decomposition of surface EMG signals into single fiber action potentials by means of neural network, Daniel Graupe, et al., Proc. IEEE International Symp. on Circuits and Systems, 1989

[9] Logika odkrycia naukowego, Karl Popper, PWN, 1977

[10] Generalization of backpropagation with application to a recurrent gas market model, P. J. Werbos, Neural Networks, 1, 1988.

[11] Approximations by superpositions of sigmoidal functions, G. Cybenko, Mathematics of Control, Signals, and Systems 2 (4) (1988), 303-314

[12] Approximation Capabilities of Multilayer Feedforward Networks,Kurt Hornik (1991) K. Hornik, Neural Networks, 4 (2) (1991), 251?257

[13] A Method for Stochastic Optimization, D.P. Kingma, J. Ba, CoRR, 2014

[14] Efficient mini-batch training for stochastic optimization, Mu Li, et al., Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, 661-670

[15] Random Search for Hyper-Parameter Optimization, J. Bergstra, Y. Bengio, Journal of Machine Learning Research 13 (2012) 281-305

[16] Regression Shrinkage and Selection via the lasso, R. Tibshirani, Journal of the Royal Statistical Society. Series B (methodological) 58 (1). Wiley: 267?88

[17] Learning Deep Architectures for AI, Y. Bengio, Foundations and Trends in Machine Learning Vol. 2, No. 1 (2009) 1?127

[18] Domain Adaptation for Large-Scale Sentiment Classification: A Deep Learning Approach, X. Glorot et al., Proceedings of the 28th International Conference on Machine Learning, Bellevue, WA, USA, 2011.

[19] Dialogi, S. Lem, Wydawnictwo Literackie 1957

[20] https://www.wired.com/2014/08/deep-learning-yann-lecun/

[21] Manifold Learning: The Price of Normalization, Y. Goldberg et al., Journal of Machine Learning Research 9 (2008) 1909-1939

[22] An overview of gradient descent optimization algorithms, http://ruder.io/optimizing-gradient-descent/

[23] Incorporating Nesterov Momentum into Adam, T. Dozat, Stanford CS 229 Projects - 2015

[24] Towards Understanding Generalization of Deep Learning: Perspective of Loss Landscapes, L. Wu et al., https://arxiv.org/pdf/1706.10239.pdf

[25] The regression analysis of binary sequences (with discussion), Cox, DR, J Roy Stat Soc B. 20: 215 - 242 1958

[26] Skip Connections Eliminate Singularities, A. E. Orhan, Xaq Pitkow, arXiv:1701.09175

[27] Nearest Neighbour Searches and the Curse of Dimensionality, R. B. Marimont M. B. Shapiro, IMA Journal of Applied Mathematics, Volume 24, Issue 1, 1 August 1979, Pages 59-70

[28] Generalization of Back-Propagation to Recurrent Neural Networks, F. J. Pineda,

[29] Visualizing and Understanding Convolutional Networks, M. D. Zeiler, R. Fergus, arXiv:1311.2901

[30] Learning hierarchical categories in deep neural networks, A. M. Saxe et al., Proceedings of the 35th annual meeting of the Cognitive Science Society. (pp. 1271-1276), 2013

[31] Deep Residual Learning for Image Recognition, K. He et al., arXiv:1512.03385

[32] Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, S. Ioffe, Ch. Szegedy, arXiv:1502.03167

[33] On information and sufficiency, S. Kullback, R.A. Leibler, Annals of Mathematical Statistics 22, 79-86

[34] Gradient-Based Learning Applied to Document Recognition, Yann LeCun et al., Proc. of IEEE, November 1998

[35] Receptive fields and functional architecture of monkey striate cortex, D. H. Hubel, T. N. Wiesel, The Journal of Physiology. 195 (1): 215?243

[36] ImageNet Classification with Deep Convolutional Neural Networks, A. Krizhevsky et al, NIPS 2012

[37] ImageNet Large Scale Visual Recognition Challenge, Russakovsky et al., arXiv:1409.0575

[38] Very Deep Convolutional Networks for Large-Scale Image Recognition, K. Simonyan, A. Zisserman, arXiv:1409.1556

[39] Going Deeper with Convolutions, Ch. Szegedy et al., arXiv:1409.4842

[40] Inception (2010), www.imdb.com/title/tt1375666 IMDB

[41] Rethinking the Inception Architecture for Computer Vision, Ch. Szegedy et al, arXiv:1512.00567

[42] Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning, Ch. Szegedy et al., arXiv:1602.07261

[43] Deep Residual Learning for Image Recognition, K. He, et al., arXiv:1512.03385

[44] Improving neural networks by preventing co-adaptation of feature detectors, G. Hinton, arXiv:1207.0580