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
Prof. dr. hab. Andrzeja Skowrona

Czerwiec 2015

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

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 ([**?**], [**?**]). 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, mo?emy 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.

Ostatnie zagadnienie - najlepiej zobrazowa? 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 [**?**]) 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 [**?**], czyli realizacje modelu neuronu McCullocha-Pittsa [**?**]. 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. Co więcej, w Marvin Minsky oraz Seymour Papert 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? 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. I tutaj pomoc przyszła z najmniej spodziewanej strony, a mianowicie pochodzącej z klasycznego rachunku prawdopodobieństwa reguły łańcuchowej. Algorytm, (Rumelhart et al. [**?**] 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. [**?**, **?**]. 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 skreślono sieci neuronowe z listy obiecujących kierunków rozwoju. 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 ([**?**, **?**], 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 procesie 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 ([**?**]), 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.

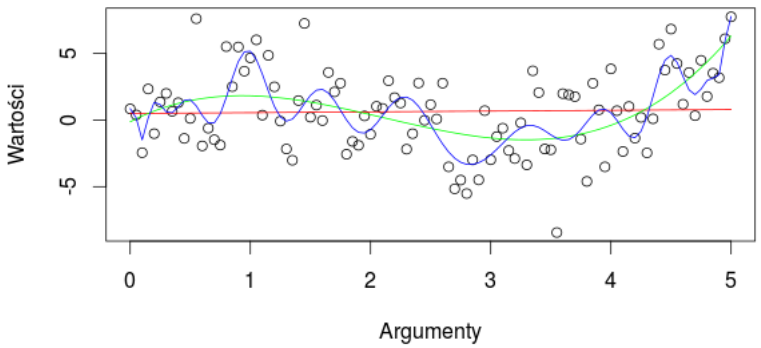
2.1.2. Uczenie z nadzorem

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 *Na poniższych obrazkach możemy zobaczyć efekt działania tzw. algorytmu k-średnich. W tym przypadku przez rodzinę modeli Θ_n będziemy uznawać n-elementowe krotki kolorów RGB, a przez $\theta_{\mathcal{X}}$ te z nich które spełniają własność, że jeśli w każdym z pikseli naszego obrazka, zamienimy jego kolor na jeden z występujących w naszej krotce, to suma kwadratów odległości euklidesowych pomiędzy zanurzeniami faktycznych kolorów pikseli w trójwymiarowej przestrzeni RGB (czyli de facto $[0,1]^3$), a zanurzeniem zastępujących ich kolorów z krotki) będzie najmniejsza z możliwych. Poniższe obraz*

2.1.4. Generalizacja oraz przeuczenie

Jako, że podczas procesu uczenia nasz model wykorzystał tylko pewien skończony zbiór danych $\mathcal{X}$, warto zastanowić się, czy nasz model będzie równie skuteczny w przypadku, gdy zbiór ten zostanie rozszerzony o nowe dane. W przypadku gdy rozkład funkcji kosztu nie różni się znacząco dla nowych danych mówimy, że nasz model posiada zdolność do generalizacji. W przypadku gdy błąd ten jest istotnie większy - mówimy o zjawisku przeuczenia. Zjawiska te obrazuje kolejny przykład.

TODO

Metody wykrywania oraz radzenia sobie ze zjawiskiem przeuczenia (poza szczególnymi przypadkami opisanymi w rozdziale TODO) przekraczają zakres tej pracy. Przykłady postępowania w takich sytuacjach możemy znaleźć w następujących artykułach : [**?**].

2.1.5. Optymalizacja gradientowa

Jak wspominaliśmy w podsekcji [**?**], 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. Przybliżony schemat działania większości z tych metod przybliża poniższy algorytm.

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 *szybkością uczenia* 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 [**?**]. Warunek stopu występujący w warunku pętli natomiast najczęściej ustalany jest jako zejście wartości funkcji kosztu $J_{\mathcal{X}}$ poniżej pewnej ustalonej wartości ϵ lub brak dostatecznej poprawy (względnej lub bezwzględnej) w stosunku do wartości obliczonej w poprzedniej iteracji.

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

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 ([**?**]) 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.

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

TODO

2.2.2. Perceptron

Pierwszym praktycznym zastosowaniem, inspirowanym niesamowitą skutecznością biologicznych układów nerwowych, który przebił się do powszechnej świadomości był opracowany przez Franka Rosenblatta sztuczny model neuronu opracowany według prac McCullocha-Pittsa. Nie wykorzystywał on jeszcze potencjału tkwiącego we współpracy neuronów ze sobą, jednak dzięki bardzo prostemu w zaimplementowaniu algorytmowi uczącemu, a także stosunkowo dużej skuteczności (w połączeniu ze sporą prostotą modelu) odniósł on olbrzymi sukces i znacząco przyczynił się do popularyzacji idei uczenia maszynowego na przełomie lat 50' i 60; XX w. Niestety - nadmierna prostota w połączeniu z nadmiernymi oczekiwaniami przyniosły ogromne rozczarowanie - w 1967 r. Papert i Minsky udowodnili olbrzymie ograniczenia jakie nałożone są na zadania, które efektywnie może rozwiązać perceptron.

Czym jest perceptron?

Oto stosowna definicja:

Definicja 2.2.1 *Perceptronem z funkcją aktywacji ϕ nazywamy funkcję $P_{\theta_0,\theta_1,\ldots,\theta_n} : \mathbb{R}^n \to \mathbb{R}$ postaci :*

$$P_{\theta_0,\theta_1,\ldots,\theta_n}(x_1, x_2, \ldots, x_n) = \phi\left(x_0 + \sum_{i=1}^{n} x_i\theta_i\right).$$

Zauważmy, że matematycznie perceptron to *de facto* złożenie funkcjonału afinicznego z zadaną funkcją aktywacji ϕ. Prostota tej kompozycji okazała się jednocześnie największą wadą i największą zaletą perceptronu. Jednak zanim omówimy szczegóły jego działania oraz algorytmu jego uczenia przedstawmy funkcję aktywacji którą będziemy stosować we wszystkich przykładach w niniejszej sekcji.

Sigmoidalna funkcja aktywacji

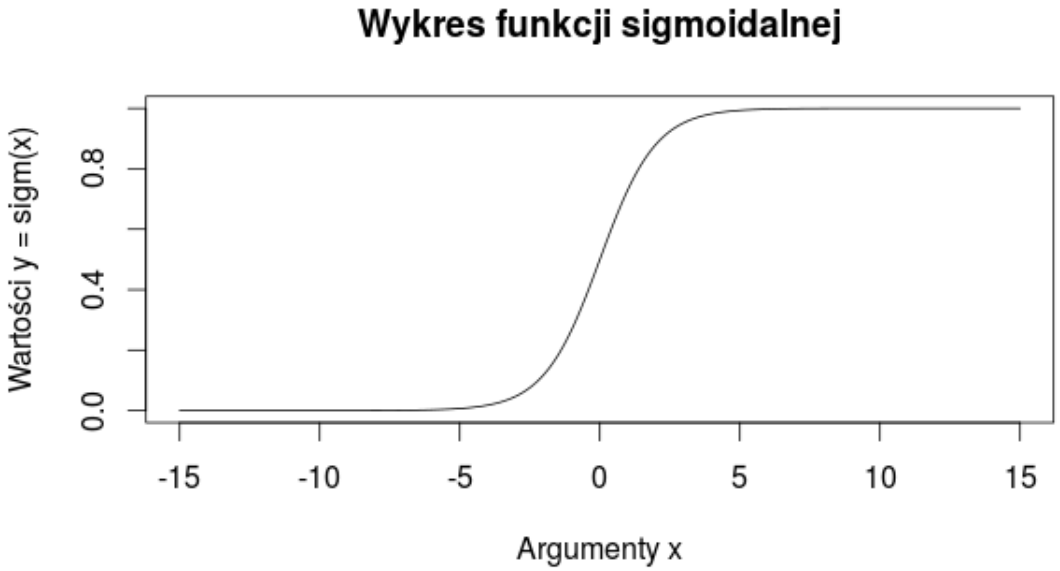
Definicja 2.2.2 *Sigmoidalną (logistyczną) funkcją aktywacji będziemy nazywać funkcję :*

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

W literaturze funkcja ta często określana jest mianem funkcji logistycznej (jako rozwiązanie tzw. równania logistycznego [**?**]. Zauważmy tylko niektóre jej 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 2.1: 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. Począwszy od tego momentu, za funkcję aktywacji ϕ będziemy przyjmować tylko funkcję sigmoidalną.

Klasyfikacja przy użyciu perceptronu.

Podstawowym zadaniem do jakiego najczęściej stosuje się model perceptronu jest zagadnienie klasyfikacji binarnej. Przyjmuje się wówczas, dla ustalenia uwagi, że każdy element zbioru $\mathcal{X}$ możemy przyporządkować do jednej z dwóch klas, które dla prostoty będziemy nazywać klasami 0 oraz 1. Przyjmuje się zatem, że dla dowolnego elementu $x \in \mathcal{X}$, obiekt x należy do klasy 1 wedle perceptronu $P_{\theta_0,\theta_1,\ldots,\theta_n}$ o ile $P_{\theta_0,\theta_1,\ldots,\theta_n}(x) > \epsilon_{threshold}$, a do klasy 0 w przeciwnym przypadku. Oczywiście parametry $\theta_0, \theta_1, \ldots, \theta_n$ a także $\epsilon_{threshold}$ uzyskujemy w trakcie procesu uczenia. Zwyczajowo jednak przyjmuje się wartość $\epsilon_{threshold} = 0.5$ i taką będziemy przyjmować do końca tej sekcji.

Algorytm uczenia perceptronu.

Jedną z przyczyn ogromnej popularności perceptronu był banalny w swojej prostocie algorytm, który pozwalał bardzo szybko znaleźć odpowiednie parametry dla modelu. Poniżej prezentujemy sposób zaprezentowany przez Franka Rosenblatta.

Jak widzimy zasada działania algorytmu jest banalnie prosta. Jeśli dany przykład został dobrze sklasyfikowany - nie zmieniamy wag, jednak gdy tylko w klasyfikacji pojawi się błąd wagi aktualizowane są wedle reguły : jeśli wynik algorytmu jest zbyt mały - dodaj do wag przykład, na którym się pomylił, jeśli za duży - odejmij. Okazuje się, że algorytm ten, znajduje optymalne rozwiązanie (o ile istnieje), a także, że jest bardzo prostym zastosowaniem metody

Data: Zbiór danych $\mathcal{X}$
Result: Wartości parametrów $\theta_1, \theta_2, \ldots, \theta_n$ dla których perceptron osiąga najmniejszy koszt dla średniokwadratowej funkcji kosztu $J_{\mathcal{X}}(\theta_0, \theta_1, \ldots, \theta_n)$.
zainicjuj wagi $\theta_0, \theta_1, \ldots, \theta_n$ w sposób losowy. ;
oblicz $J_{\mathcal{X}}(\theta_0, \theta_1, \ldots, \theta_n)$;
while $J_{\mathcal{X}}(\theta_0, \theta_1, \ldots, \theta_n)$ *nie spełnia warunku stopu* **do**
  forall the $x \in \mathcal{X}$ **do**
    if $P_{\theta_0,\theta_1,\ldots,\theta_n}(x) > 0.5$ *i* x *należy do klasy* 0 **then**
      $\theta_i = \theta_i - x_i$ dla każdego $i = 1, \ldots, n$, $\theta_0 = \theta_0 - 1$,
    end
    if $P_{\theta_0,\theta_1,\ldots,\theta_n}(x) <= 0.5$ *i* x *należy do klasy* 1 **then**
      $\theta_i = \theta_i + x_i$ dla każdego $i = 1, \ldots, n$, $\theta_0 = \theta_0 + 1$,
    end
    ponownie oblicz $J_{\mathcal{X}}(\theta_0, \theta_1, \ldots, \theta_n)$;
  end
end

Algorithm 2: Uproszczony algorytm uczenia perceptronu.

optymalizacji gradientowej - o ile za szybkość uczenia przyjmiemy liczbę 1. Dużą zaletą była także łatwość zaimplementowania go na przełomie lat 50' oraz 60'. W gwoli ścisłości należy dodać, że w wypadku tego sposobu uczenia problemem może okazać się bardzo duże wartości parametru (przy niesprzyjających danych, wartości niektórych z θ_i mogą rozbiegać do nieskończoności). Z problemem tym radzono sobie na wiele możliwych sposobów, począwszy od zmiany wartości stałej uczenia, po normalizację wag po każdej iteracji głównej pętli algorytmu uczącego.

2.3. Klasyczne sieci neuronowe oraz algorytm wstecznej propagacji

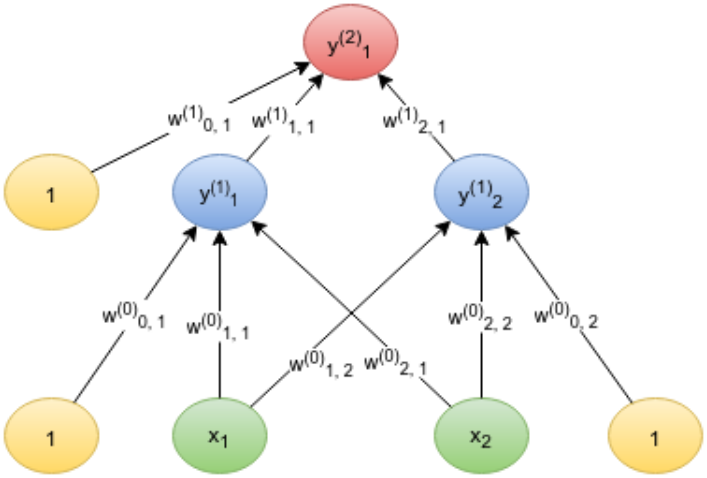
Jak zauważyliśmy w poprzednim rozdziale - prosta perceptronu, choć czyni jego uczenie banalnie prostym, stanowi niebagatelną przeszkodę - potrafi on efektywnie nauczyć się jedynie bardzo prostych w swojej strukturze zbiorów (dokładniej półprzestrzeni afinicznych kowymiaru 1). Szybko jednak zauważono, że tak jak układy nerwowe zwierząt nie składają sie tylko i wyłącznie z jednego neuronu, tylko z ich współdziałającego ze sobą konglomeratu, tak skutecznie działające modele muszą łączyć w sobie moc wielu połączonych ze sobą perceptronów. Tak narodziła się idea *wielowarstwowego perceptronu* (ang. *multi-layered perceptron* czyli w istocie pierwszej sztucznej sieci neuronowej.

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 :



Rysunek 2.2: Przykład architektury wartstwowej sieci typu *feed-forward*. Sieć o tej architekturze jest w stanie nauczyć się logicznej funkcji XOR.

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

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

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

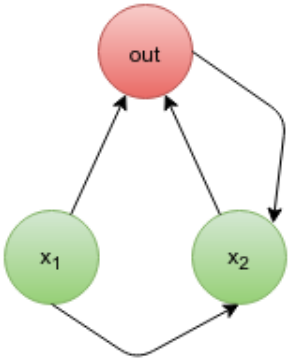
Dla uproszczenia przyjęliśmy, że w obrębie warstwy funkcji aktywacji są takie same, jednak nie stanowi to koniecznego wymogu. Przedstawiona powyżej architektura stanowi przykład architektury warstwowej - kolejne warstwy przyjmują jako swoje argumenty rezultaty obliczeń z poprzednich warstw (stąd nazwa *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. Topologię każdej sieci zwykle przedstawia się w postaci skierowanego grafu złożeń (patrz rysunek [**?**]). 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 aproksymacji przez sieci neuronowe mówi poniższe twierdzenie (dowód można znaleźć [**?**]).

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 :*

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



Rysunek 2.3: Przykład architektury rekurencyjnej. Zwróćmy uwagę na skierowany cykl w grafie połączeń.

Powyższe twierdzenie mówi nam o problemie regresji funkcji gładkich. Analogiczne twierdzenie ([**?**]) 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

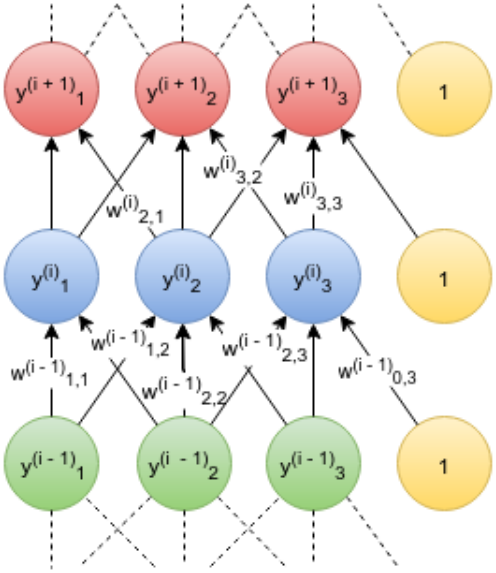
To, że dołożenie kolejnych warstw zwiększa możliwości perceptronu wiedziano już w latach '60 XXw. To co przyczyniło się do przełamania złego trendu, który nastał po rozczarowaniu związanym z ograniczeniami perceptronu był kolejny skuteczny algorytm uczenia. W tym przypadku kluczem stał się algorytm wstecznej 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 :

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



Rysunek 2.4: Rysunek pomocniczy dla procesu wstecznej propagacji.

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) :

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

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.

Zauważmy, że w powyższe rozumowanie z łatwością można rozszerzyć na przypadek sieci w których istnieją połączenia między wagami. 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.

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 :*

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

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

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

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:

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

a zatem logarytm funkcji wiarygodności wyraża się wzorem :

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

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 :

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

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 :

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

gdzie $\mathcal{L}\left(f_\theta(x), 1\right)$ to rozkład Laplace'a z parametrami 0 oraz 1, równoważną metodzie MLE metodą optymalizacji będzie minimalizacja z następującą funkcją błędu :

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

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

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

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:

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

lub stosując zapis wektorowy :

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

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:

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

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

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

ale wówczas dokładając Bayesowskie założenie o wartości parametrów θ otrzymujemy :

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

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 :

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

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

$$\theta \sim \mathcal{L}(0, 1).$$

2.3.6. Czym różnią się powyższe kary?

Warto wspomnieć o dosyć poważnych koncepcyjnych różnicach pomiędzy powyższymi wagami. Udowodniono bowiem numerycznie [**?**], 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, jdowód 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 opisanego w sekcji TODO *przekleństwa wymiarowości*.

Oba powyższe wnioski rozwiniemy w kolejnych rozdziałach.

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. 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.

Głęboka hierarchiczna struktura przyczyną sukcesu algorytmów z rodziny Deep Learning

Dzięki intuicyjnemu zrozumieniu na czym polega schemat działania sieci neuronowych, możemy teraz opisać jedną z przyczyn dla których algorytmy z rodziny algorytmów *deep learning* odniosły w ostatnim czasie tak ogromny sukces. Istnieje bowiem uzasadniona hipoteza [**?**], że wiele problemów, z którymi borykają się specjaliści uczenia maszynowego wymaga stworzenia odpowiedniej *głębokiej* hierarchii cech, koniecznej do stworzenia skutecznego rozwiązania.

Według badań [**?**] problemy związane z zagadnieniami wizji komputerowej oraz rozpoznawania dźwięku wymagają architektury o głębokości conajmniej 7. Bardziej związane

Rozdział 3

Rozwój metod uczenia

W swojej pracy przyjąłem konwencję wedle której zamierzam przedstawiać szczegóły technik *deep learningowych* zgodnie chronologią ich powstania. Uważam taką formę za odpowiednią, ponieważ przedstawia je zgodnie z rosnącym poziomem trudności oraz skomplikowania. Dlatego rozpoczniemy od pierwszej rewolucji, która pchnęła sieci neuronowe w kierunku budowania bardziej realistycznej hierarchii cech - czyli wynalezienia sieci konwolucyjnych, a także omówienia pierwszej zaawansowanej i działającej sieci korzystającej z tego paradygmatu czyli słynnej *LeNet* autorstwa Yana LeCun'a.

Kolejnym etapem będzie przedstawienie rewolucji związanej z nową formą treningu wstępnego sieci, która przyniosła olbrzymi skok w jakości uczenia. Przyjrzymy się zatem powodom, dla których regularyzacja gwarantowana przez inicjację sieci jako autoenkoder lub Ograniczoną Maszynę Boltzmanna pozwoliły przenieść skuteczność działania sieci neuronowych na wyższy poziom.

Kolejne dwie sieci przeniosą nas w świat nowoczesnych implementacji oraz skoku, który był związany z głównie ze wzrostem skuteczności obliczeń, a zakończymy przeglądem innych interesujących sieci, których znajomość z pewnością poszerzy nasze intuicje dotyczące możliwości jakie rozpościera przed badaczami *deep learning*.

3.1. Sieci konwolucyjne oraz LeNet

3.1.1. 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 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.

3.1.2. Sieci konwolucyjne

Sieci konwolucyjne to wynaleziona w TODO 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 obrazujące dwa historyczne przykłady pierwszych topologii konwolucyjnych.

TODO - przykład linia TODO - przykład płaszczyzna

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.

3.1.3. Pooling

Zauważmy, że jedna warstwa sieci neuronowej może składać się z wielu warstw konwolucyjnych. Intuicyjnie odpowiada to uczeniu przez sieć wielu rodzajów inwariantnych cech na danym poziomie. Jednak jeśli dla każdej z tych warstw, moc zbioru rzutowań będzie duża, może to doprowadzić do powstania warstwy z ogromną ilością neuronów. Aby tego uniknąć postanowiono automatycznie rozszerzyć każdą warstwę konwolucyjną o automatycznie złożoną z nią kolejną, zazwyczaj nieparametryczną, warstwę konwolucyjną o znaczenia mniejszej ilości neuronów, a także posiadającą tę własność, że rzutowania ją generujące są parami rozłączne, zwaną warstwą *poolingu*. Pozwala to znacznie zmniejszyć ilość informacji trzymanej w pamięci (obliczenia można dystrybuować tak, aby w danym momencie trzymać informację tylko z rzutowań połączonych z ustaloną jednostką z warstwy *poolingu*), a także zmniejsza ilość neuronów przekazywanych do kolejnej warstwie.

TODO : Rysunek, warstwa poolingu.

3.2. Autoenkodery, sieć Hintona i Salakhudinowa

3.3. Sieć Kryzhevskiego

3.4. GoogLeNet

3.5. Inne sieci

Rozdział 4

Moja implementacja

Bibliografia