OPTIMIZATION

1. Wprowadzenie do optymalizacji

1.1 Dlaczego potrzebujemy optymalizacji ciągłej?

Głębokie sieci neuronowe (ang. *deep neural networks*) pod wieloma względami zrewolucjonizowały podejście do problemów sztucznej inteligencji. Na przykład, jeszcze kilka lat temu stworzenie programu komputerowego wygrywającego w grę Go wydawało się czymś całkowicie nieosiągalnym. Dlaczego? Powodów było wiele, ale nadmieńmy tutaj jedynie fakt, że liczba możliwych stanów w tej grze jest dalece większa od liczby wszystkich atomów w całym wszechświecie! Efektywne przeszukiwanie tak dużej przestrzeni klasycznymi metodami jak np. algorytm alfa-beta jest więc w zasadzie nie możliwe. Z tego powodu gra w Go długo stanowiła niezdobyty szczyt dla algorytmów, pomimo tego że sukcesy komputerów np. w grze w szachy zaczęły się jeszcze w ubiegłym stuleciu. Większość badaczy myślała, że pokonanie mistrzów w Go zajmie jeszcze co najmniej dekadę, kiedy to w 2016 roku system AlphaGo pokonał mistrza Europy a także ówczesnego mistrza świata. Główną składową systemu AlphaGo, obok statystycznych technik przeszukiwania przestrzeni stanów, były właśnie głębokie sieci neuronowe.

Jednak sieci neuronowe nie tylko grają w gry, ale także potrafią samodzielnie prowadzić samochód, rozpoznawać obrazy, tłumaczyć teksty z różnych języków, rozpoznawać emocje w wypowiedziach i wiele, wiele innych. Co się jednak kryje za tak dużym praktycznym sukcesem tych metod? W dużej mierze nowoczesne metody optymalizacji ciągłej.

Jak to możliwe? Cóż, sieć neuronową możemy interpretować jako po prostu pewną funkcję, przyjmującą na wejściu pewien wektor danych np. piksele w obrazku, a na wyjściu zwracającą informację czy na obrazku znajduje się kot czy też inna rzecz. Taka sieć (czy też taka funkcja) ma oczywiście swoje parametry θ, które należy dostosować w taki sposób aby nasza sieć robiła to co trzeba. Ten proces nazywamy *uczeniem się* sieci neuronowej. To, co przed chwilą dumnie nazwaliśmy automatycznym zdobywaniem wiedzy nie jest jednak w rzeczywistości niczym innym jak procesem optymalizacyjnym. Problem ten można nawet bardzo prosto sformułować: „minimalizuj liczbę popełnianych przez sieć

błędów”!

$$minimize_\theta \quad \sum_{i=1}^{N} error_i \qquad \text{gdzie} \qquad error_i = \begin{cases} 1 & \text{jeśli } f(x_i;\theta) \neq y_i \\ 0 & \text{w przeciwnym wypadku} \end{cases}$$

Jak jednak możemy zoptymalizować tak skomplikowaną funkcję, jaką jest przecież głęboka sieć neuronowa, która posiada miliony parametrów[1]? Dla wielu jest to całkowita magia. I o tej magii jest właśnie ten przedmiot ;)

1.2 Organizacja zajęć

1.2.1 Zasady zaliczenia przedmiotu

Zasady obowiązujące studentów są następujące:

- każdemu laboratorium towarzyszy test na platformie Moodle, który studenci wypełniają w domu (dość proste, zwykle obejmujące tylko ostatnie zajęcia, 60% oceny).
- student ma możliwość 3-krotnego podejścia do każdego testu elektronicznego, jednak ostateczne podejście musi zostać zakończone przed kolejnymi laboratoriami. Termin ten jest nieprzekraczalny, a brak wypełnienia testu skutkuje wynikiem 0%. Do oceny wlicza się najwyższy uzyskany wynik.
- Dopuszcza się także zmianę formy testu na klasyczną, papierową kartkówkę, którą studenci piszą na początku zajęć - jednak zmianę formy testu ogłosi prowadzący na swojej stronie internetowej nie później niż na 3 dni przed zajęciami.
- przewiduje się jedno większe zadanie domowe na ocenę (40% oceny)
- w przypadku nieterminowego oddania zadania domowego odejmuje się 10% od ostatecznego wyniku za każdy rozpoczęty dzień spóźnienia
- aby zaliczyć laboratoria należy każde z zajęć *zaliczyć* (ocena binarna) poprzez pokazanie wykonanych zadań i/lub odpowiedzenie na kilka prostych pytań oraz zdobyć wymaganą liczbę punktów procentowych (patrz następny punkt).
- ocena z laboratoriów jest przyznawana na podstawie średniej ważonej testów oraz zadania domowego. Stosuje się następującą skalę ocen: od 51% (3), próg każdej następnej oceny rośnie o 10% (np. 3.5 jest od 61%)
- dopuszcza się 1 nieusprawiedliwioną nieobecność studenta na laboratoriach, jednak nadal student powinien wypełnić w zwykłym terminie test (w przypadku dłuższej choroby uniemożliwiającej podeście do testu, prosimy o jak najszybszy kontakt z prowadzącym)
- student przyłapany na ściąganiu lub plagiatowaniu zadań domowych otrzymuje ocenę niedostateczną, niezależnie od innych ocen

Dodatkowo, zgodnie z Regulaminem Studiów Politechniki Poznańskiej:

- nieobecności studenta, w tym usprawiedliwione, przekraczające 1/3 zajęć są podstawą do niezaliczenia zajęć
- student zobowiązany jest do usprawiedliwienia u prowadzącego nieobecności na zajęciach w ciągu dwóch tygodni

Informacje organizacyjne:

[1] Autorzy skryptu zdają sobie sprawę, że uczenie maszynowe i optymalizacja to nie to samo. Jednak zawiłościami statystycznymi i, konkretnie, uczeniem maszynowym zajmują się inne przedmioty i edukacja w tym zakresie nie jest celem niniejszego skryptu

- w przypadku kierowania korespondencji mailowej do prowadzącego laboratorium (`{michal.kempka,mateusz.lango}@cs.put.poznan.pl`) uprzejmie prosimy o rozpoczynanie tematu maila od skrótu przedmiotu: „[OC]”.
- materiały do przedmiotu oraz ew. ogłoszenia są zamieszczane na stronie internetowej `www.cs.put.poznan.pl/mlango` w zakładce „Teaching”.

1.2.2 Program labratorium

Celem przedmiotu jest zaznajomienie studenta z podstawowymi technikami optymalizacji ciągłej. Omawiane będą metody bezgradientowe jak metoda złotego podziału czy metoda dychotomii, a także metody gradientowe (metoda spadku wzdłuż gradientu, metoda Cauchy'iego) czy metody Newtona (uogólniona metoda Newtona, algorytm Newtona-Raphsona, algorytm Levenberga-Marquarda). Ponadto w programie przedmiotu znalazło się też kilka bardziej zaawansowanych i nowoczesnych technik stosowanych w uczeniu maszynowym, statystyce czy analizie danych jak bp. stochastyczny spadek wzdłuż gradientu, technika momentum czy AdaGrad. Sporą część przedmiotu przeznaczono także na powtórki wymaganych pojęć i operacji matematycznych.

Na laboratoriach studenci będą wykonywać implementacje oraz eksperymenty korzystając z języka Python, a jako przykład biblioteki do optymalizacji wypukłej zostanie pokazana biblioteka `cvxpy`. Jednakże większość laboratoriów będzie się skupiała na własnej implementacji algorytmów, a nie na korzystaniu z gotowych bibliotek. Ponadto podczas laboratoriów omówione zostanie kilka wybranych problemów optymalizacyjnych mających duże znaczenie w informatyce. Wśród nich znajdziemy optymalizację modeli liniowych, sieci neuronowych, technikę MDS (ang. *multi-dimensional scaling*) czy problem rekonstrukcji obrazków.

1.3 Podstawy optymalizacji

W ogólności problem optymalizacyjny możemy zapisać jako

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & x \in \mathbf{X} \end{aligned}$$

gdzie $\mathbf{X}$ nazywamy zbiorem rozwiązań dopuszczalnych. Ponadto problem optymalizacji ciągłej to taki problem w którym optymalizowane zmienne należą do zbioru liczb rzeczywistych czyli $\mathbf{X} \subseteq \mathbb{R}^n$. Problem w którym $\mathbf{X} = \mathbb{R}^n$ nazywamy problemem bez ograniczeń.

Problem 1.1 Podaj przykłady znanych ci problemów optymalizacji ciągłej.

Warto zauważyć, że $f(x)$ wcale nie musi być wyrażona jakimś matematycznym wzorem. Możemy sobie wyobrazić np. optymalizację kąta nachylenia skrzydeł w samolocie w taki sposób, aby własności aerodynamiczne były jak najlepsze. Jak jednak zbadać własności aerodynamiczne? Standardową metodą jest zastosowanie metod obliczeniowej mechaniki płynów (ang. *computational fluid dynamics*) i przeprowadzenie odpowiedniej symulacji takiego samolotu. Nie trzeba chyba mówić, że taka symulacja jest bardzo skomplikowana i wymaga ona dużych zasobów obliczeniowych. Jest więc ona wykonywana przez wiele godzin (jeśli nie dni) na sporych rozmiarów klastrze obliczeniowym. Pomimo tego, z punktu widzenia optymalizacji, możemy powiedzieć, że ta cała symulacja jest po

prostu pewną funkcją zwracającą jakąś miarę jakości właściwości aerodynamicznych. To prawda, nie policzymy po niej pochodnej, ani jej nie zcałkujemy. Jednak mamy możliwość jej zewaluowania i to powinno - przynajmniej niektórym algorytmom - zupełnie wystarczyć.

Trzeba także wspomnieć o tym, że funkcja $f(x)$ nie musi być deterministyczna, a pomimo tego może być optymalizowana. Możemy chcieć np. zoptymalizować mieszankę uprawianych gatunków i stosowanych nawozów tak aby zmaksymalizować zyski rolnika[2]. Jednak ilość plonów (a więc także i zysk) nie zależy tylko od optymalizowanych przez nas czynników, ale chociażby także od warunków pogodowych, które nie mogą być przez nas optymalizowane. Ewaluacja funkcji jest więc tutaj obarczona pewnym błędem losowym. Takimi problemami nie zajmujemy się na tym przedmiocie, ale warto wiedzieć że one istnieją[3].

Zwykle jednak będziemy się zajmować problemami w których wzór funkcji jest znany. Z taką sytuacją najczęściej mamy miejsce gdy problem optymalizacyjny jest przez nas projektowany. Przykład? Chociażby problem uczenia się, który zdefiniowaliśmy wcześniej jako minimalizacja liczby błędów. Taka postać funkcji nie wynika z praw fizyki (jak np. w przykładzie z samolotem) czy nie zależy od czynników zewnętrznych których nie znamy (jak w przykładzie z rolnictwem) to my powiedzieliśmy że w ten właśnie sposób zdefiniujemy uczenie sieci. Moglibyśmy je zaprojektować inaczej np. moglibyśmy przerobić etykietę „na obrazku jest kotek” na wartość numeryczną 1, a decyzję „na obrazku nie ma kotka” reprezentować jako -1. Teraz zamiast zliczać błędy poprzez porównywanie etykiet możemy sumować np. wartość bezwzględną różnic pomiędzy prawdziwym y_i a tym którym uzyskaliśmy z naszej funkcji $f(x_i;\theta)$.

$$minimize_\theta \quad \sum_{i=1}^{N} error_i \qquad \text{gdzie} \qquad error_i = |f(x_i;\theta) - y_i|$$

Możemy też – no bo czemu nie – minimalizować sumę różnic do kwadratu...

$$minimize_\theta \quad \sum_{i=1}^{N} error_i \qquad \text{gdzie} \qquad error_i = (f(x_i;\theta) - y_i)^2$$

Tutaj również możemy powiedzieć, że nasza sieć „uczy się” poprzez rozwiązywanie takiego problemu optymalizacyjnego. To ty projektujesz w jaki sposób będziesz ją uczył czyli jak zdefiniujesz problem optymalizacyjny.

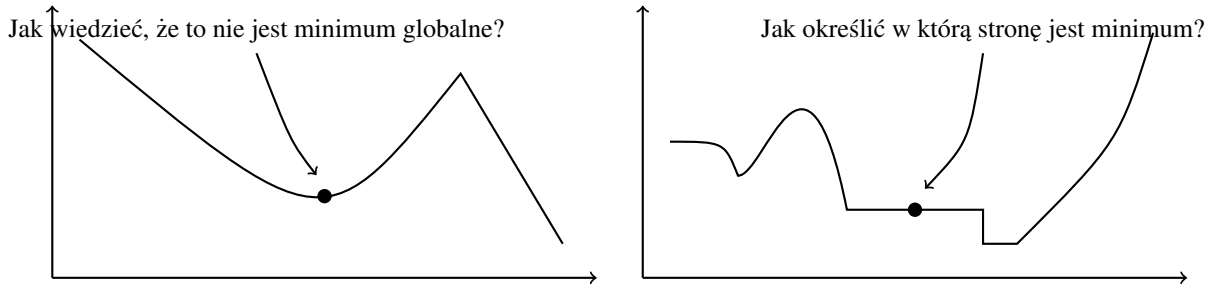
! Nie oznacza to, że te problemy są takie same czy też, że dadzą tę samą jakość rozwiązania.

1.3.1 Łatwe i trudne w optymalizacji

Problem 1.2 Jakie funkcje są twoim zdaniem trudne, a które proste w optymalizacji? Podaj po 3 przykłady takich funkcji i spróbuj nazwać formalnie ich własności. Czy czyni je trudnymi? Czy czyni je prostymi?

[2] Ile trwa ewaluacja takiej funkcji? Jeden rok.

[3] W takich problemach nie możemy się spodziewać że gradient nawet w minimum/maksimum wyniesie dokładnie 0. Tak, stosujemy tutaj testy statystyczne!



Rysunek 1.1: Problemy w optymalizacji

Definicja 1.1 — Stała Lipschitza. Najmniejsza stałą L taką, że

$$|f(x_1) - f(x_2)| \leqslant L|x_1 - x_2|$$

nazywamy *stałą Lipschitza* (ang. *Lipschitz constant*).

Definicja 1.2 — Funkcja wypukła (nieformalnie). Jeżeli linia łącząca dwa dowolne punkty na wykresie funkcji leży zawsze albo na linii tej funkcji, albo nad nią to funkcję taką nazywamy funkcją wypukłą.

Przez długi okres czasu wydawało się, że jedynie problemy programowania liniowego są stosunkowo proste w optymalizacji. Są to problemy w których zarówno funkcja celu jak i ograniczenia są funkcjami liniowymi. Na szczęście okazało się, że postać takich problemów jest trochę ogólniejsza: są to problemy optymalizacji wypukłej.

Definicja 1.3 — Problem optymalizacji wypukłej. Problem optymalizacji wypukłej definiujemy jako

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq b_i,\ i = 1, \ldots, m \\ & h_j(x) = d_j,\ j = 1, \ldots, l \\ & x \in \mathbb{R}^n \end{aligned}$$

gdzie $f(x)$ oraz $g_i(x)$ są funkcjami wypukłymi, a $h_j(x)$ są funkcjami liniowymi.

Jest to dużo szersza gama problemów niż tylko programy liniowe, a problemy wypukłe mają wielkie znaczenie w inżynierii, statystyce czy analizie biznesowej. Ponieważ techniki programowania liniowego poznałeś na Badaniach Operacyjnych ta część optymalizacji ciągłej zostanie tutaj całkowicie pominięta.

Wróćmy na chwilę do naszych uwag dot. formułowania twoich problemów jako problemów optymalizacyjnych. Skoro funkcje wypukłe są stosunkowo proste w optymalizacji to odpowiednie przedefiniowanie twojego problemu na problem optymalizacji wypukłej jest w zasadzie połową sukcesu. Dlatego też nawet dużo problemów nie wypukłych próbuje się przybliżać funkcjami wypukłymi, albo stara się przedstawić proces optymalizacji jako optymalizację serii (zestawu) takich przybliżeń. Z powyższych powodów poświęcimy na dalszych laboratoriach dużo uwagi tym właśnie funkcjom.

1.4 Modyfikowanie funkcji celu

Co tak naprawdę jest naszym poszukiwanym wynikiem? Czy jest to minimalna wartość funkcji $f(x)$ czy też x dla którego $f(x)$ daje wartość najmniejszą? Różnicę tę można dość dobrze uchwycić patrząc na zapis matematyczny:

$$\min(x-2)^2 = 0 \qquad \arg\min(x-2)^2 = 2$$

Powyższa funkcja kwadratowa przyjmuje wartość minimalną dla $x = 2$, a wartość funkcji w minimum wynosi $f(2) = (2-2)^2 = 0$.

Zwykle będzie nas bardziej interesowało w jakim punkcie funkcja osiąga minimum (jak mam ustawić parametry sieci neuronowej, żeby robiła co trzeba?) niż ile to minimum wynosi (jaki jest minimalny błąd dla sieci neuronowej?). Zwróć uwagę, że z chwilą kiedy znasz optymalne x, możesz po prostu obliczyć wartość $f(x)$, gdybyś kiedykolwiek chciał się tego dowiedzieć. Z drugiej strony, gdybyś chciał poznać x znając optymalną wartość funkcji $f(x)$ sprawa jest trochę trudniejsza (problem przeszukiwania).

Tutaj może zapalić ci się czerwona lampka: to wszystko prawda, ale przecież ewaluacja funkcji $f(x)$ może być bardzo kosztowna! Taką sytuację mieliśmy np. w przykładzie z samolotem gdzie ewaluacja funkcji wymagała długotrwałych symulacji na całym klastrze obliczeniowym. Oczywiście masz rację, jednak wykonanie algorytmu optymalizacyjnego zwykle będzie wymagało wielokrotnego ewaluowania funkcji celu, więc ten jeden dodatkowy raz zwykle i tak będzie znikomą częścią całkowitego kosztu optymalizacji.

1.4.1 Czy jest możliwe transformowanie problemów optymalizacyjnych do problemów prostszych?

Powyższe rozważanie uprawnia nas do zadania następującego pytania: czy jest możliwe przetransformowanie problemu optymalizacyjnego do innego (być może łatwiejszego?) w taki sposób aby mieć gwarancję, że optimum będzie takie samo? Zaznaczmy, że mamy tu na myśli, że oba problemy będę osiągać swoje minimum w tym samym punkcie, choć wartość funkcji celu może się różnić.

Problem 1.3 Jak zmienić problem z maksymalizacją funkcji wklęsłej na równoważny problem z funkcją wypukłą?

Dla prostych problemów często jesteśmy w stanie od razu stwierdzić, że mają one optimum w tym samym miejscu. Na przykład minimalizacja $(x-2)^2$ oraz $(x-2)^2 + 5$ będzie miała to samo minimum w $x = 2$. Czy jednak możemy powiedzieć, że do dowolnej funkcji celu możemy dodać piątkę i wynik się nie zmieni? Jakie inne operacje są dozwolone?

W celu wypracowania ogólnej reguły przypomnijmy najpierw, że x^* jest ścisłym minimum globalnym kiedy wartość funkcji $f(x)$ jest w tym punkcie najmniejsza. Inaczej możemy powiedzieć, że wszystkie inne x'y oprócz x^* mają wyższą wartość funkcji.

$$\forall_{x \neq x^*} f(x^*) < f(x)$$

Możemy więc korzystać z jakichkolwiek transformacji które nie zmienią tego faktu. Wracając do naszego przykładu, jeżeli $\forall_{x \neq x^*} f(x^*) < f(x)$ to również $\forall_{x \neq x^*} f(x^*) + 5 < f(x) + 5$.

Wyraźmy to co właśnie zrobiliśmy w sposób ogólniejszy, przy użyciu zapisu funkcyjnego. Naszą operację dodania piątki zapiszmy jako pewną funkcję $g(x) = x + 5$. Wtedy

wcześniej skonstruowaną implikację możemy zapisać jako: jeżeli $\forall_{x \neq x^*} f(x^*) < f(x)$ to również $\forall_{x \neq x^*} g(f(x^*)) < g(f(x))$. W ten sposób doszliśmy do definicji funkcji silnie rosnącej!

Definicja 1.4 — Funkcja silnie rosnąca. Funkcję $g(x)$ nazywamy silnie rosnącą (monotoniczną) gdy

$$x < y \Rightarrow g(x) < g(y)$$

■ **Przykład 1.1** Funkcjami silnie rosnącymi są wszystkie funkcje liniowe, dla których współczynnik kierunkowy jest większy od 0 czyli

- $min.\ 2x^2$ jest tożsama z $min.\ x^2$, bo funkcja którą działamy na problem to $g(x) = \frac{1}{2}x$
- $min.\ 2x^2 + 5$ jest tożsama z $min.\ x^2$, bo funkcja którą działamy na problem to $g(x) = \frac{1}{2}x - 2.5$

■

■ **Przykład 1.2 — Coś idzie nie tak!.** Funkcją silnie rosnącą jest także $g(x) = \sqrt{x}$ czyli

- $min.\ x^2$ jest tożsama z $min.\ |x|$. Jest to prawda: obie te funkcje osiągają minimum dla $x = 0$.
- $min.\ x$ jest tożsama z $min.\ \sqrt{x}$. Hmmmm... minimalną wartość pierwiastka osiągniemy dla $x = 0$ podczas gdy dla pierwszego problemu będzie to $-\infty$ (minimum nie istnieje).

■

Co więc poszło nie tak? Chodzi oczywiście o dziedzinę funkcji modyfikującej funkcję celu. Funkcja $g(x) = \sqrt{x}$ jest zdefiniowana dla liczb nieujemnych, a więc jej zastosowanie dodaje jakby nowe ograniczenie $x \geq 0$ do naszego problemu optymalizacyjnego.

! Zwróć uwagę, że jeżeli aplikujemy funkcję $g(x)$ na funkcji celu – czyli mamy wyrażenie $g(f(x))$ – to ograniczenie dotyczące dziedziny funkcji g ($x \geq 0$) dotyczy tutaj $f(x)$! Powstałe więc ograniczenie to $f(x) \geq 0$.

Dla pierwszego rozważanego problemu optymalizacyjnego ($min.\ x^2$) zastosowanie funkcji silnie rosnącej o ograniczonej dziedzinie nie zmieniło nam wartości minimum, ponieważ powstałe ograniczenie to $f(x) = x^2 \geq 0$ co jest zawsze prawdą! Jednakże dodanie ograniczenia $x \geq 0$ do problemu minimalizacji $f(x) = x$ zmienia go dość znacząco. Podsumowując: uważaj na dziedzinę funkcji silnie rosnącej, którą modyfikujesz problem optymalizacyjny.

Powyższe rozważania dotyczyły funkcji celu, które minimalizujemy – jakie operacje możemy stosować dla funkcji, które maksymalizujemy? Przypomnijmy, że x^* jest maksimum globalnym kiedy $\forall_{x \neq x^*} f(x^*) > f(x)$. Po przepisaniu tej nierówności od prawej do lewej otrzymujemy $\forall_{x \neq x^*} f(x) < f(x^*)$. Jest to taka sama nierówność jak dla problemu minimalizacji, tylko, że x i x^* zamieniły się miejscami! Aby zachować prawdziwość tej nierówności nadal możemy stosować funkcje silnie rosnące!

Czy więc nigdy nie możemy stosować funkcji silnie malejących? Możemy ich używać ale wtedy kierunek optymalizacji musimy zmienić na przeciwny.

Definicja 1.5 — Funkcja silnie malejąca. Funkcję $g(x)$ nazywamy silnie malejącą (monotoniczna) gdy

$$x < y \Rightarrow g(x) > g(y)$$

1.4.2 Transformacje a wypukłość

Wcześniej wyjaśniliśmy, że szczególnie interesować nas będą problemy optymalizacyjne w których funkcja celu jest wypukła bądź wklęsła, ponieważ takie problemy są stosunkowo łatwe w optymalizacji. Czy transformacje funkcjami silnie rosnącymi, mogą doprowadzić nas z problemu niewypukłego do wypukłego? Czasami tak!

■ **Przykład 1.3 — Estymacja maksymalnej wiarygodności.** Aby znaleźć estymator maksymalnej wiarygodności $\hat{p}$ dla rozkładu Bernoulliego należy rozwiązać odpowiedni problem optymalizacyjny (zmaksymalizować wiarygodność). Rozważmy prostą sytuację rzutu monetą. W eksperymencie uzyskaliśmy 4 orły i 1 reszkę. W takiej sytuacji nasz problem sprowadza się do maksymalizacji wyrażenia:

$$p^4(1-p)$$

Problem 1.4 Czy ten problem optymalizacyjny jest problemem z ograniczeniami czy bez ograniczeń?

Wykres tej funkcji jest przedstawiony na rysunku 1.2a. Jak możesz zauważyć funkcja ta nie jest funkcją wklęsłą, ponieważ np. linia łącząca czubek wykresu funkcji z punktem $(0, f(0))$ leży ponad linią funkcji. Jednocześnie nie jest to funkcja wypukła, ponieważ np. linia łącząca czubek funkcji z punktem $(1, f(1))$ leży poniżej wykresu.

Czy jest szansa przetransformować ten problem w taki sposób, żeby optymalizowana funkcja była wklęsła? Okazuje się, że tak. Jak pewnie pamiętasz ze statystyki, aby uniknąć błędów numerycznych czy też aby ułatwić sobie obliczenia, zamiast funkcji wiarygodności zwykle optymalizujemy jej logarytm.

$$\log(p^4(1-p)) = 4\log p + \log(1-p)$$

Funkcja logarytm jest oczywiście silnie rosnąca, jednak ma ograniczoną dziedzinę do $x > 0$. Musimy więc pamiętać, że w naszej optymalizacji pojawiło się jakby dodatkowe ograniczenie $p^4(1-p) > 0$. To ograniczenie nie jest jednak dla nas bardzo problematyczne ponieważ to oznacza, że $p > 0$ oraz $1-p > 0$ czyli $0 < p < 1$. Ponieważ prawdopodobieństwo może przyjmować wartości od 0 do 1, to jedyne co straciliśmy to możliwość by p wynosiło 0% lub 100%. Taka wartość prawdopodobieństwa nie jest jednak możliwa, ponieważ zaobserwowaliśmy zarówno orły jak i reszki.

No dobra, to jak wygląda nasza funkcja celu po potraktowaniu jej logarytmem? Spójrz na wykres 1.2b: jest to funkcja wklęsła! Ponadto patrząc na rysunek 1.3 widzimy, że obie te funkcje mają maksimum w tym samym punkcie dla $p = 0.8$. ■

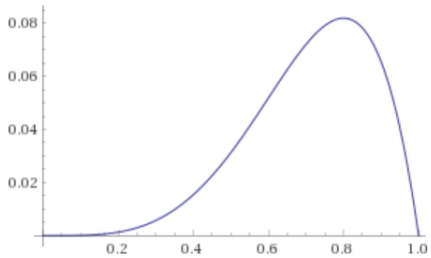
Literatura

Literatura powtórkowa

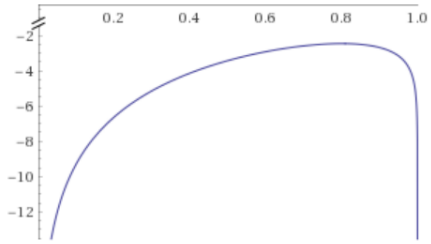
Dobre wprowadzenie do optymalizacji można przeczytać w rozdziale 1 książki [1]. Intuicyjne omówienie podstawowych zagadnień i algorytmów optymalizacji dyskretnej można znaleźć w [**?**].

Literatura dla chętnych

W jednym z zadań tutorialowych stworzyłeś problem optymalizacyjny, którego rozwiązaniem był klasyfikator czyli „reguła” pozwalająca ci na przypisywanie obserwacji do jednej

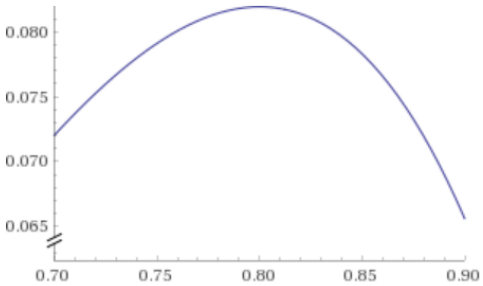


(a) Funkcja wiarygodności

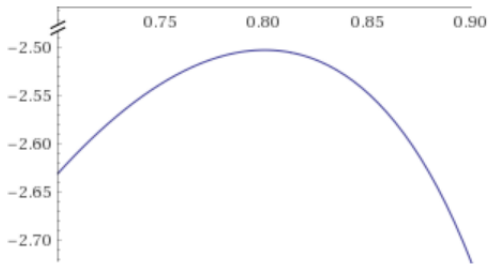


(b) Logarytm funkcji wiarygodności

Rysunek 1.2: Funkcja wiarygodności oraz jej logarytm dla problemu estymacji prawdopodobieństwa w rozkładzie Bernoulliego (4 orły, 1 reszka).



(a) Funkcja wiarygodności



(b) Logarytm funkcji wiarygodności

Rysunek 1.3: Funkcja wiarygodności oraz jej logarytm dla problemu estymacji prawdopodobieństwa w rozkładzie Bernoulliego (4 orły, 1 reszka) w okolicy maksimum.

z dwóch grup. Zaprojektowany klasyfikator był uproszczoną wersją Maszyny Wektorów Wspierających (SVM). Zachęcamy Cię do poszerzenia wiedzy o tym klasyfikatorze, a w szczególności do prześledzenia matematycznej formulacji problemu optymalizacji SVM, który nie jest już uproszczony.

- `https://www.youtube.com/watch?v=v7H5ks5iDEQ`
- `https://www.youtube.com/watch?v=ax8LxRZCORU`

Bibliografia

[1] Stephen Boyd i Lieven Vandenberghe. Convex Optimization. `http://stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf`, 2013.

2. Wypukłość i nie tylko

2.1 Czemu funkcja wypukła?

Pytanie brzmi czemu właściwie interesują nas funkcje wypukłe? Otóż odpowiedź jest prosta: **minimum lokalne** funkcji wypukłej jest zarazem **minimum globalnym**. Wprzypadku funkcji ściśle wypukłej jest także gwarancja, że takie minimum jest tylko jedno. Czyni to funkcje wypukłe prostymi do optymalizacji - prawdziwym wyzwaniem staje się w tym momencie transformacja rzeczywistych problemów do problemów wypukłych.

2.2 Kombinacje wypukły

W świecie optymalizacji popularną rodziną funkcji są **funkcje wypukłe**. Zanim jednak przejdziemy do funkcji wypukłych omówmy kilka innych pomocnych bytów tak jak kombinacja wypukła:

Definicja 2.1 — Kombinacja wypukła. Kombinacją wypukłą punktów $x_i \in \mathbb{R}^d, d \in \mathbb{N}$ nazywamy sumę:

$$\sum_{i=1}^{n} \alpha_i x_i; \alpha_i \in \mathbb{R}+; \sum_{i=1}^{n} \alpha_i = 1$$

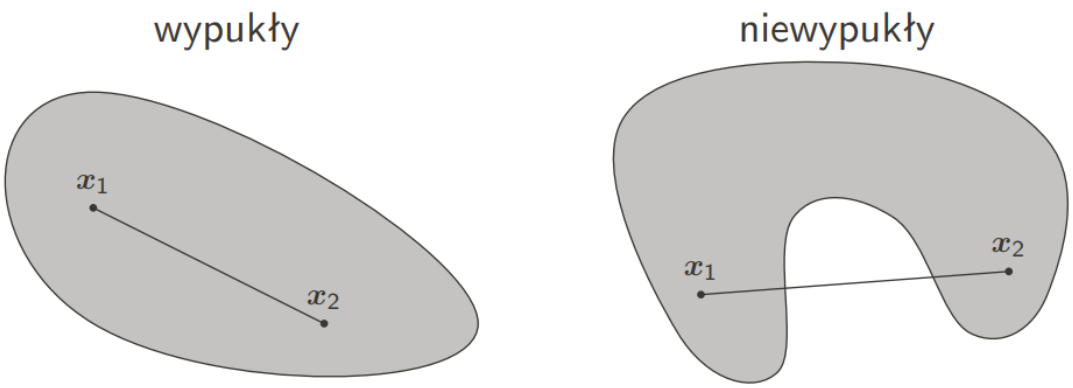
Kombinacja liniowa jest zatem sumą ważoną składowych punktów, gdzie wagi sumują się do 1 i są nieujemne (tworzą zatem rozkład prawdopodobieństwa - rozkład kategoryczny). Wszystkie kombinacje liniowe wybranych punktów tworzą razem zbiór wypukły.

Definicja 2.2 — Zbiór wypukły. Zbiór dowolnych punktów jest wypukły jeśli dowolna kombinacja wypukła tych punktów także znajduje się w tym zbiorze.

Zbiór wypukły możemy zatem zinterpretować geometrycznie jako figurę zawartą pomiędzy tymi punktami. W przyadku dwóch punktów dostajemy odcinek (λ decyduje o konkretnym

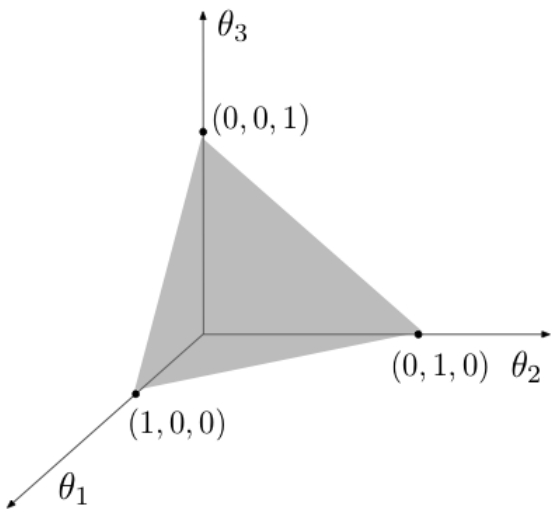
położeniu na tym odcinku), dla 3 punktów mamy trójkąt, dla 4 czworokąt i tak dalej (aż do nieskończoności).

Jedną z bardziej znanych figur wypukłych jest simplex. W przestrzeni (k+1) - wymiarowej, simpleks definiowany jest przez (k+1) niezależnych liniowo punktów. Dla k=1 otrzymujemy 2 punkty - simpleks jest odcinkiem. Dla k=2 otrzymujemy trójkąt równoboczny, dla k=3 otrzymujemy czworościan (tetrahedron - nie jest to piramida!). Szczególnym przypadkiem simplexa jest simplex standardowy (probability/standard simplex):



Definicja 2.3 — Simpleks standardowy. k-wymiarowym simpleksem standardowym nazywamy taki zbiór punktów:

$$\{x \in \mathbb{R}^{k+1} : x_0 + \cdots + x_k = 1, x_i >= 0, i = 0, \ldots k\}$$

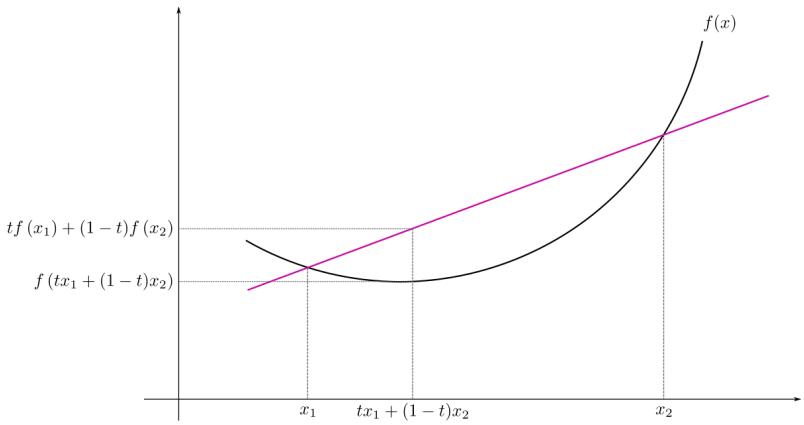


2.2.1 Funkcja wypukła

Definicja 2.4 — Funkcja wypukła. Funkcja f(x) jest **wypukła**, jeśli dla dowolnych dwóch punktów $\mathbf{x_1}$, $\mathbf{x_2}$ i dowolnego $\lambda \in [0,1]$ zachodzi:

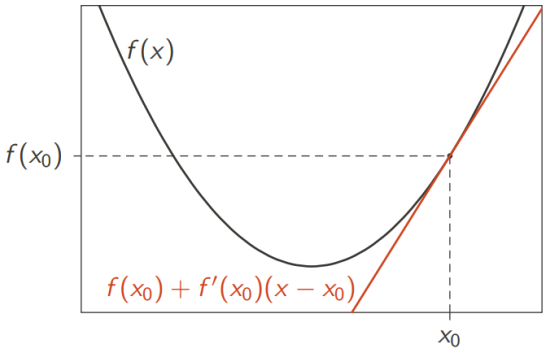
$$f(\lambda x_1(1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda) f(x_2)$$

Jeśli nierówność jest ostra (<), to funkcja jest **ściśle wypukła**.
Zapis ten możemy zinterpretować geoemetrycznie: odcinek łączący dwa dowolne punkty na wykresie (cięciwa /odcinek siecznej) leży w całości powyżej lub na wykresie funkcji. Warto wspomnieć, że punkty leżące ponad wykresem funkcji wypukłej tworzą **zbiór wypukły**.



Jeśli funkcja f(x) jest **różniczkowalna**, to jest **wypukła** wtedy i tylko wtedy gdy dla dowolnych $\mathbf{x_0}$, $\mathbf{x_1}$ zachodzi:

$$f(x) \geq f(x_0) + \nabla f(x_0)^T (x - x_0)$$



Czyli styczna do wykresu leży zawsze pod wykresem.

2.2.2 Operacje zachowujące wypukłość (wybrane):

- złożenie funkcji wypukłej z niemalejącą funkcją wypukłą
- max
- kombinacja **wypukła**

2.2.3 Operacje niekoniecznie zachowujące wypukłość

- mnożenie, dzielenie
- przeciwny znak
- kombinacja liniowa/afiniczna
- złożenie funkcji

2.2.4 Ważne funkcje wypukłe:

- funkcja kwadratowa
- norma euklidesowa (i dowolna inna)
- funkcja wykładnicza
- -logarytm
- błąd zawiasowy (hinge loss)
- entropia krzyżowa (cross entropy, logloss)
- abs

2.3 Normy, metryki i dywergencje

Normy służą nam do oceny **wielkości** wektorów (cokolwiek). Jednak nie każda funkcja moze być użyta jako norma.

2.3.1 Norma

Definicja 2.5 — Norma. Odwzorowanie $||.|| : X \to [0,\infty]$ jest normą jeśli spełnia następujące warunki:

1. $||x|| = 0 \Rightarrow x = 0$
2. $||\alpha x|| = |\alpha|||x||; \alpha \in \mathbb{R}$
3. $||x+y|| \leq ||x|| + ||y||$ (nierówność trójkąta)

Do najbardziej znanych norm należą:

- norma l_p: $||x||_p = \sqrt[p]{\sum_{i=1}^{n} |x_i|^p}$
  w szczególności:
- $l_1(x) = \sum_{i=1}^{n} |x_i|$ (norma Manhaattańska/taksówkowa),
- $l_2(x) = ||x||$ (norma Euklidesowa)
- $l_\infty(x) = \max_i(|x_i|)$ (norma Czebyszewa)

2.3.2 Metryka

Metryki służą do reprezentacji **dystansu** między dwoma wektorami.

Definicja 2.6 — Metryka. Odwzorowanie $d : XxX \to [0,\infty]$ nazywamy metryką jeśli dla każdego $x,y,z \in X$ spełnione są następujące warunki:

1. $d(x,y) \geq 0$
2. $d(x,y) = 0 \Leftrightarrow x = y$
3. $d(x,y) = d(y,x)$ (symetria)
4. $d(x,z) \leq d(x,y) + d(y,z)$ (nierówność trójkąta)

Na pierwszy rzut oka normy i metryki wydają się podobne. Nie jest przypadek gdyż każda norma indukuje metrykę (w drugą stronę to nie zachodzi).

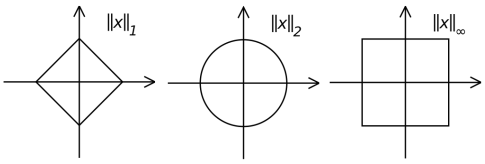
Definicja 2.7 — Metryka indukowana przez normę. Metryka d jest indukowana przez normę $||.||$ jeśli:

$$d(x,y) = ||x-y||$$

Najbardziej znane metryki są indukowane przez najbardziej znane normy, mianowicie metryka euklidesowa, dystans manhattański (odległość taksówkowa) czy dystans Czebyszewa.

2.3.3 Okrąg jednostkowy

Okrąg jednostkowy jest to zbiór wektorów, których norma jest równa jeden - alternatywnie: odległość od środka układu współrzędnych to 1. To jak będzie wyglądał okrąg jednostkowy zależy od normy jakiej użyjemy:



2.3.4 Dywergencja

Nie wchodząc w dalsze szczegóły chcieliśmy tu wspomnieć o dywergencjach, którą służą jako bardziej uogólniona definicja dystansu w stosunku do normy. Dywergencje nie muszą być symetryczne i nie muszą spełniać nierówności trójkąta. Dywergencja **Kullbacka-Leiblera (KL divergence)** jest najbardziej powszechnie używaną dywergencją i służy do oceny rozbieżności dwóch rozkładów prawdopodobieństwa. Jest to kluczowe w przypadku klasyfikacji - próbujemy doprawodzić do sytuacji gdzie rozkład prawdopodobieństwa naszych predykcji i danych mają jak najmniej rozbieżności (mała dywergencja). By dowiedzieć się więcej zachęcamy do zapoznania z Convex Optimization [1].

Literatura

Dość wyczerpującym źródłem na temat optymalizacji wypukłej (i tym co zrobić z wiedzą na ten temat) jest Convex Optimization [1].

Bibliografia

[1] Stephen Boyd i Lieven Vandenberghe. Convex Optimization. `http://stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf`, 2013.

3. Metody bezgradientowe

3.1 Czemu metody bezgradientowe?

Metody gradientowe są to metody optymalizacji, które, jak sama nazwa wskazuje, nie korzystają z gradientu (pochodnej). Może być to przydatne w momencie gdy nie znamy minimalizowanej, zatem nie mamy bezpośredniego dostępu do gradientu.

3.2 Przeszukiwanie jednostajne

Jedną z prostszych method optymalizacji jest przeszukiwanie jednostajne, które polega na wyznaczeniu jednostajnie rozmieszczonych argumentów (np. od 0 do 1 co 0.1) i wybraniu tego, dla którego funkcja ma najmniejszą wartość. Gdy zrezygnujemy z założenia, że argumenty muszą być rozmieszczone jednostajnie otrzymamy metodę **grid search**, która rozpina siatkę na interesujących nas wartościach przeszukiwanej przestrzeni. Grid search jest stosowany najczęściej w doborze hiper parametrów dla algorytmów optymalizacji (np. ilość epok uczenia, wielość populacji etc.). Dość typowym zwyczajem (np. w przypadku doboru prędkości uczenia) jest korzystanie z argumentów skalujących się wykładniczo (np. kolejne potęgi dwójki) zamiast z rozkładu jednostajnego.

Niewątpliwą zaletą przeszukiwania jednostajnego (i grid searcha) jest brak jakichkolwiek założeń na temat optymalizowanej funkcji i jej wymiarowości oraz prostata działania i implementacji. Efektem ubocznym takiego podejścia jest brak jakichkolwiek gwarancji - możemy nie znaleźć nawet przeciętnie dobrego wyniku dla trywialnych funkcji (nawet minimum lokalnego). Co więcej złożność algorytmu efektywnie rośnie wykładniczo z wymiarowością.

3.3 Przeszukiwanie dychotomiczne

Przeszukiwanie dychotomiczne, jak sugeruje nazwa (dychotomia to z greckiego podział na 2 części), polega na dzieleniu przeszukiwanej przestrzeni na pół i odcinaniu połowy, która na pewno nie zawiera minimum. Niestety wspomniana procedura wymaga dziedzina funkcji była **jednowymiarowa**, a sama funkcja była **unimodalna**. W przypadku funkcji multimodalnych, metoda może nie zadziałać poprawnie.

Definicja 3.1 — Funkcja unimodalna. Funkcja f(x) jest jednomodalna jeśli posiada dokładnie jedno ekstremum lokalne.

Poniższy pseudokod pokazuje jak dokładnie działa przeszukiwanie dychotomiczne dla funkcji f na przedziale $[x_1, x_2]$:

Definicja 3.2 — Przeszukiwanie dychotomiczne.

dane:x_1, x_2, δ(mała liczba np 10^{-8}), n (ilość kroków)
$a \leftarrow x_0$ $b \leftarrow x_1$
for k **do** = 1,2,3, n
  $m \leftarrow \frac{a+b}{2}$
  $x_l \leftarrow m - \delta$
  $x_r \leftarrow m + \delta$
  if $f(x_l) \leq f(x_r)$ **then**
    $b \leftarrow x_l$
  else
    $a \leftarrow x_r$
  end if
end for
return $\frac{a+b}{2}$

Algorytm przeszukiwania dychotomicznego znajduje minimum z dokładnością zależną od ilości, którków jakie wykona. Konkretnie, obszar lokalizacji minimum będzie zawężony do $\approx \frac{1}{2}^n (x_2 - x_1)$ gdzie przyblizenie wynika z pominięcia małego δ. Aby uniezależnić się od wyboru ilości kroków można zatrzymać algorytm po uzyskaniu zadanej dokładności dla funkcji celu lub po wyczerpaniu budżetu obliczeniowego.

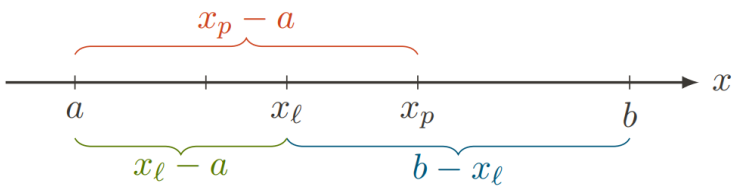
3.3.1 Metoda złotego podziału

Na pierwszy rzut oka wydaje się, że algorytmy dychotomizacji jest najlepszą metodą tego typu jaką można zaproponować – w każdej iteracji redukuje się obszar przeszukiwania o (prawie) połowę. Okazuje się jednak, że można zaproponować algorytm który nie dzieli obszaru o połowę (czyli, aby otrzymać tę samą szerokość przedziału musi wykonać więcej iteracji), a pomimo tego jest szybszy!

Algorytm dychotomizacji wymaga dwóch wywołań funkcji celu w każdej iteracji. Można jednak ograniczyć liczbę odwołań do funkcji i w każdej iteracji ewaluować funkcję tylko jeden raz (czyli iteracja algorytmu trwa dwukrotnie szybciej!) – z takim zamysłem stworzono algorytm złotego podziału. Pomysł na poprawę polega na ponownym użyciu obliczonej wcześniej wartości funkcji dla jednego z wyznaczonych punktów. Podobnie jak w algorytmie dychotomizacji, w każdej iteracji porównujemy ze sobą wartości funkcji w dwóch punktach i zawężamy obszar przeszukiwania. Mamy jednak gwarancję, że w każdej iteracji jednym z dwóch punktów które będą ewaluowane będzie punkt już policzony w

poprzedniej iteracji. Najtrudniejsze pytanie: w jaki sposób możemy wyznaczać takie dwa punkty w każdej iteracji aby mieć taką gwarancję?

Nanieśmy na oś x dwa punkty które ewaluujemy w iteracji algorytmu:



Po wykonaniu iteracji zakres zwęża się do $[a, x_p]$ lub $[x_l, b]$. Ponieważ nie wiemy, który przypadek nastąpi, ustalamy, że długości te są sobie równe tj. w każdej iteracji następuje redukcja przedziału poszukiwań o taki sam procent obszaru.

$$x_p - a = b - x_l \implies x_l - a = b - x_p$$

Gdy $f(x_l) < f(x_p)$ wybieramy obszar $[a, x_p]$, a punkt x_l chcielibyśmy aby został ponownie użyty w kolejnej iteracji algorytmu. Natomiast w następnej iteracji algorytmu znów chcielibyśmy uzyskać takie same proporcje podziału, czyli ustalamy:

$$\frac{x_p - a}{b - a} = \frac{x_l - a}{x_p - a}$$

Podstawiając pierwsze równanie do drugiego otrzymujemy:

$$\frac{x_p - a}{b - a} = \frac{b - x_p}{x_p - a} = \frac{b - a - (x_p - a)}{x_p - a} = \frac{b - a}{x_p - a}$$

Definiując $q = \frac{x_p - a}{b - a}$ (stosunek podziału) otrzymujemy równanie:

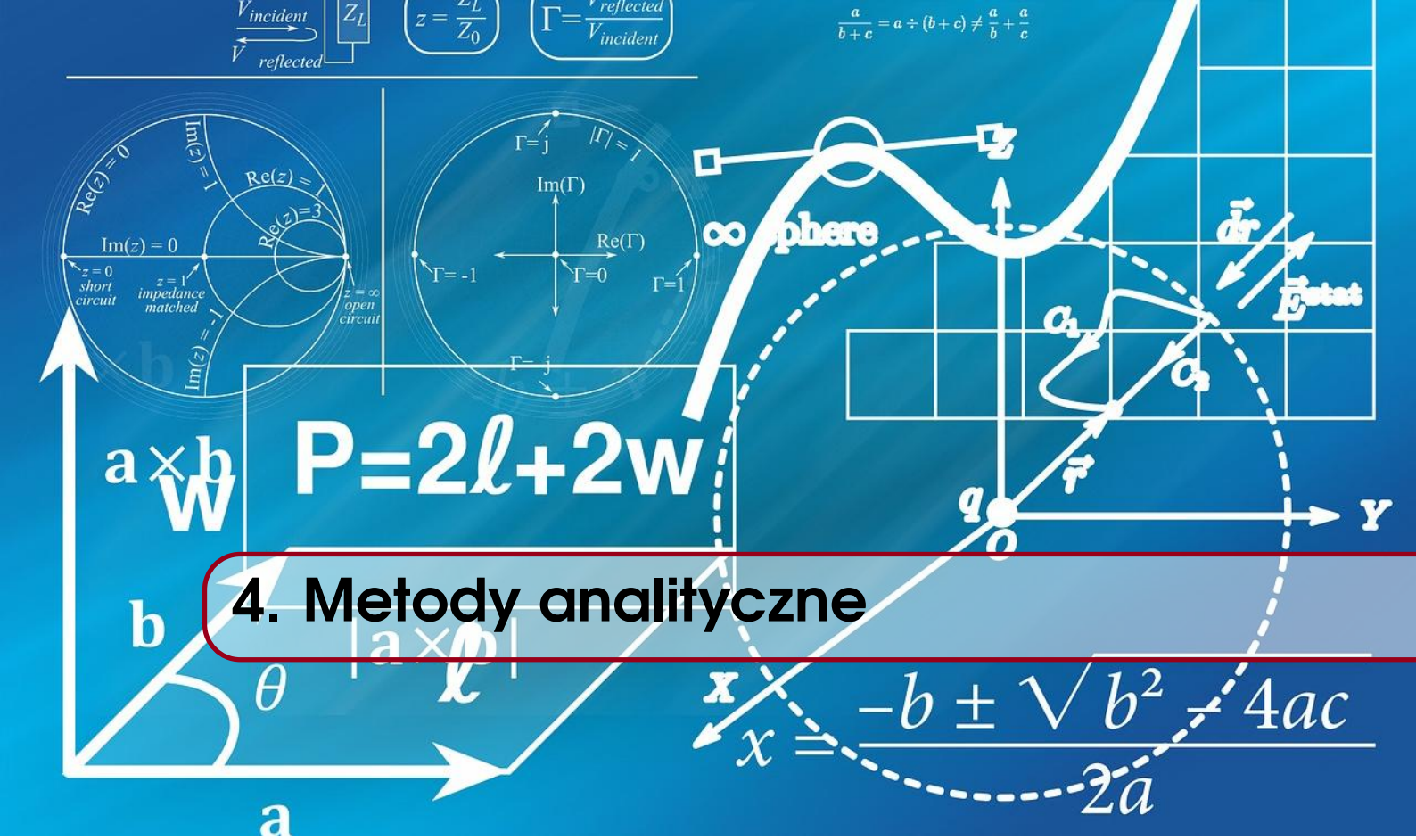
$$q^2 + q - a = 0$$

którego rozwiązaniem jest $q = \frac{\sqrt{(5)} - 1}{2} \approx 0.618$ czyli odwrotnosć *złotej liczby*. Liczba ta która wyznacza złoty podział odcinka (stąd nazwa metody). Poniższy pseudokod pokazuje jak działa metoda złotego podziału dla funkcji f na przedziale $[x_1, x_2]$:

Definicja 3.3 — Metoda złotego podziału.

dane:x_1, x_2, n (ilość kroków)
$\alpha \leftarrow \frac{\sqrt{5}-1}{2}$
$a \leftarrow x_0$ $b \leftarrow x_1$
$x_l \leftarrow \alpha a + (1 - \alpha) b$
$x_r \leftarrow (1 - \alpha) a + \alpha b$
$fx_l \leftarrow f(x_l), fx_r \leftarrow f(x_r)$
for k **do** = 1,2,3,... n
    if $fx_l \leq fx_r$ **then**
        $b \leftarrow x_r$
        $x_r \leftarrow x_l$
        $fx_r \leftarrow fx_l$
        $x_l \leftarrow \alpha a + (1 - \alpha) b$
        $fx_l \leftarrow f(x_l)$
    else

$a \leftarrow x_l$
$x_l \leftarrow x_r$
$fx_l \leftarrow fx_r$
$x_r \leftarrow (1-\alpha)a + \alpha b$
$fx_r \leftarrow f(x_r)$
end if
end for
return $\frac{a+b}{2}$



4. Metody analityczne

Na pierwszych laboratoriach podawaliście różne pomysły na algorytmy optymalizacyjne takie jak narysowanie wykresu funkcji celu (przeszukiwanie jednostajne) czy przeszukiwanie losowe. Algorytmem który jakoś szczególnie przypadł nam do gustu był algorytm wykorzystujący lokalną informację o funkcji. Taki algorytm (w przypadku funkcji 1D) patrzył sobie w prawo i w lewo i przesuwał się tam gdzie było niżej. Tę lokalną informację którą był kierunek najszybszego lokalnego wzrostu, nazywaliśmy gradientem. Nadszedł czas by dokładniej przyjrzeć się temu podejściu.

4.1 Idea spadku wzdłuż gradientu

Spróbujmy sformalizować to podejście używając bardzo prostego pseudokodu. Rozpoczynamy optymalizację od jakiegoś punktu startowego x, który możemy wybrać np. losowo. Punkt ten będzie naszym aktualnym rozwiązaniem problemu optymalizacyjnego, a zadaniem naszego algorytmu będzie przesuwać go w kierunku x^* dla którego funkcja celu osiąga minimum. W każdej iteracji nasz algorytm będzie rozglądać się w prawo i w lewo starając się określić kierunek w którym funkcja lokalnie rośnie[1]. Następnie widząc, że funkcja rośnie w prawo (poprzez zwiększenie x) – przesuwamy się w kierunku przeciwnym, aby funkcję zminimalizować. Wykonujemy więc aktualizację naszego aktualnego rozwiązania x trochę w lewą stronę, mając nadzieję, że zbliżamy je do minimum funkcji. Analogicznie, jeżeli funkcja rośnie poprzez zmniejszanie x to nasz algorytm zwiększy go, przesuwając aktualne x w prawą stroną.

$x \leftarrow$ INICJALIZUJ
while warunek stopu nie jest spełniony **do**
  $v \leftarrow$ ROZGLĄDAJ SIĘ I SPRAWDŹ KIERUNEK WZROSTU W BIEŻĄCEJ OKOLICY(x)
  $x \leftarrow x - v$ ▷ Przesuń x w kierunku minimum

[1] patrzymy gdzie rośnie, a nie gdzie maleje po prostu przez przyjętą konwencję

end while

Nasz kod wygląda w miarę sensownie, jednak nadal wymaga od nas ustalenia jak zaimplementujemy funkcję inicjalizuj, warunek stopu czy funkcję rozglądającą się. Na razie uprościmy sobie rozważania zakładając, że iterujemy w nieskończoność (brak warunku stopu, kiedyś ktoś wciśnie `Ctrl+C` :) a inicjalizacja jest losowa. Nadal jednak musimy zaimplementować najważniejszą część naszego algorytmu czyli funkcję rozglądającą się.

Jak sprawdzić czy funkcja rośnie czy maleje w okolicach naszego aktualnego punktu x? Cóż, możemy dodać/odjąć jakieś małe ε od naszego punktu i sprawdzić ile wynosi funkcja $f(x+\varepsilon)$ oraz $f(x-\varepsilon)$, a następnie porównać te dwie wartości. Jeżeli $f(x+\varepsilon) > f(x-\varepsilon)$ to wiemy[2], że funkcja rośnie poruszając się w prawo, jeśli $f(x+\varepsilon) < f(x-\varepsilon)$ to funkcja lokalnie maleje. Kiedy $f(x+\varepsilon) = f(x-\varepsilon)$ to funkcja jest lokalnie stała i, tak jak dyskutowaliśmy, nasz algorytm utknie w tym punkcie. Choć – jeżeli funkcja jest rzeczywiście stała w okolicach punktu x – to w zasadzie znaleźliśmy lokalne maksimum lub minimum.

W takim razie przepiszmy pseudokod algorytmu wykorzystując nasz świeżo opisany sposób badania zmienności funkcji. Obecną w kodzie funkcję „rozglądaj się...” możemy zamienić po prostu jako różnicę $f(x+\varepsilon) - f(x-\varepsilon)$. Ta różnica będzie dodatnia, kiedy $f(x+\varepsilon) > f(x-\varepsilon)$ i ujemna w przeciwnym wypadku. Dodatkowo, przesuwanie naszego aktualnego rozwiązania x o tę różnicę ma potencjalną dodatkową zaletę. Jeśli ta różnica będzie duża (funkcja bardzo szybko rośnie) to po wykonaniu aktualizacji $x \leftarrow x - v$ (czyli $x \leftarrow x - (f(x+\varepsilon) - f(x-\varepsilon))$ przesuniemy nasze rozwiązanie mocno w lewo, robiąc duży krok. Jeśli zaś funkcja będzie rosła wolno to odpowiednio krok wykonany przez nas będzie mały. Dlaczego to może być potencjalnie dobre? Przypomnij sobie funkcję x^2 – funkcja ta powoli rośnie/maleje w okolicach minimum i bardzo szybko rośnie z daleka od niego. W tym konkretnym przypadku informacja, że funkcja rośnie bardzo szybko jest równoznaczna z informacją „jesteś daleko od minimum” - wykonaj więc duży krok. Jeśli zaś funkcja rośnie wolno $\Rightarrow$ jesteś blisko minimum $\Rightarrow$ rób małe kroczki.

Jest jednak pewien problem: jednostki. Wyobraź sobie, że chcemy np. minimalizować odległość od jakiegoś punktu x_0. Możemy skonstruować dwie ekwiwalentne funkcje celu: jedna z nich będzie wyrażała odległość w centymetrach, a druga w metrach. Zwróć uwagę, że kroki oparte na różnicach w centymetrach będą automatycznie 100 razy większe niż w metrach! Z tego powodu potrzebujemy pewnej stałej η, którą będziemy wymnażać przez różnicę wartości funkcji w celu dodatkowego regulowania wielkości kroku. Stałą tę nazwiemy *szybkością optymalizacji*. Problemem dobierania tej stałej, jak i lepszym wyjaśnieniem dlaczego jej potrzebujemy zajmiemy się na kolejnych laboratoriach.

$x \leftarrow$ INICJALIZUJ
$\eta \leftarrow$pewna stała (szybkość optymalizacji)
while warunek stopu nie jest spełniony **do**
  $v \leftarrow f(x+\varepsilon) - f(x)$
  $x \leftarrow x - \eta v$ ▷ Przesuń x w kierunku minimum
end while

W powyższym zapisie uprościliśmy jeszcze trochę różnicę: tak naprawdę jeśli znamy wartość funkcji w punkcie $f(x)$ to żeby sprawdzić czy funkcja rośnie możemy policzyć jej różnicę z $f(x+\varepsilon)$.

[2]Chyba lepszym słowem byłoby *wydaje nam się* na podstawie naszego prostego sposobu rozglądania się

Ok, to pozostał ostatni problem: jak wybrać ε? Sam wybrany przez nas symbol matematyczny sugeruje, że powinna to być jakaś mała liczba. Jednak dlaczego tak jest? Po pierwsze, chcieliśmy uzyskać lokalną informację o kierunku wzrostu/spadku funkcji i na jej podstawie poruszać się w kierunku minimum. Jeżeli nasz krok rozglądania się będzie zbyt duży ta cała analogia legnie w gruzach. Dodatkowo, funkcja $f(x)$ może być dowolnie skoczna, więc wybierając zbyt duże ε ryzykujemy, że nasze rozglądanie się przegapi istotny spadek funkcji! Jednak jeśli chcemy wybrać tak małe ε jak to tylko możliwe... to biorąc granicę z ε dążącym do 0 otrzymujemy w zasadzie definicję pochodnej funkcji!

Definicja 4.1 — Algorytm spadku wzdłuż gradientu (1D).

$x \leftarrow$ INICJALIZUJ
$\eta \leftarrow$pewna stała – szybkość optymalizacji
while warunek stopu nie jest spełniony **do**
  $x \leftarrow x - \eta f'(x)$ ▷ Przesuń x w kierunku minimum
end while

Algorytm który właśnie opisaliśmy nazywamy algorytmem spadku wzdłuż gradientu (ang. *gradient descent*), a dla maksymalizacji algorytmem wzrostu wzdłuż gradientu (ang. *gradient ascent*). Aby go zaimplementować musimy być w stanie policzyć pochodną z funkcji celu - pora przypomnieć sobie jak to się robi ;)

4.2 Pochodna funkcji

Definicja 4.2 — Pochodna funkcji. Pochodną funkcji w punkcie x nazywamy granicę

$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

■ **Przykład 4.1** Obliczmy ile wynosi pochodna z $f(x) = x^2$

$$\begin{aligned}\frac{df(x)}{dx} &= \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} = \lim_{\Delta x \to 0} \frac{(x+\Delta x)^2 - x^2}{\Delta x} \\ &= \lim_{\Delta x \to 0} \frac{\cancel{x^2} + 2x\Delta x + \Delta x^2 - \cancel{x^2}}{\Delta x} \\ &= \lim_{\Delta x \to 0} (2x + \Delta x) = 2x\end{aligned}$$

■

■ **Przykład 4.2** Pokażmy, że pochodna sumy to suma pochodnych:

$$\begin{aligned}\frac{d}{dx}\big(f(x)+g(x)\big) &= \lim_{\Delta x \to 0} \frac{(f(x+\Delta x) + g(x+\Delta x)) - (f(x)+g(x))}{\Delta x} \\ &= \lim_{\Delta x \to 0} \frac{f(x+h) - f(x) + g(x+\Delta x) - g(x)}{\Delta x} \\ &= \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} + \lim_{\Delta x \to 0} \frac{g(x+\Delta x) - g(x)}{\Delta x} \\ &= f'(x) + g'(x)\end{aligned}$$

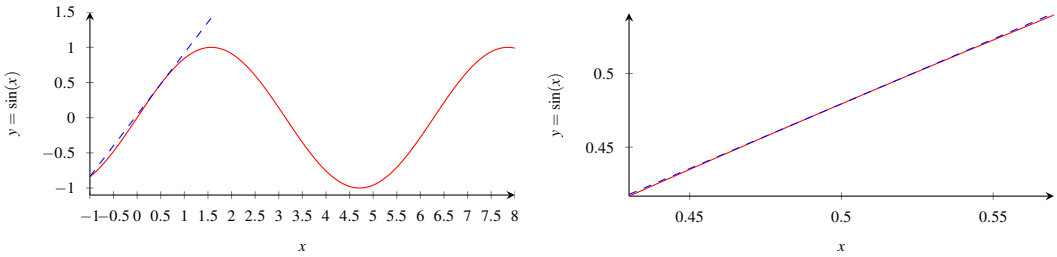
■

Wzory na pochodne:

- $\frac{d}{dx}(f(x)+g(x)) = \frac{d}{dx}(f(x)) + \frac{d}{dx}(g(x))$ – pochodna sumy to suma pochodnych
- $\frac{d}{dx}(a) = 0$ – pochodna ze stałej to 0
- $\frac{d}{dx}(x^n) = nx^{n-1}$
- W szczególności powyższy wzór możemy użyć do obliczenia $\frac{d}{dx}(\frac{1}{x}) = \frac{d}{dx}(x^{-1}) = -1 \cdot x^{-2} = -\frac{1}{x^2}$
- $\frac{d}{dx}(e^x) = e^x$
- $\frac{d}{dx}(\log(x)) = \frac{1}{x}$
- $\frac{d}{dx}(f(x)g(x)) = f'(x)g(x) + f(x)g'(x)$
- $\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{f'(x)g(x)-f(x)g'(x)}{g^2(x)}$
- $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$ – reguła łańcuchowa

4.2.1 Pochodna jako najlepsze liniowe przybliżenie funkcji w punkcie

Zastanawiając się czym jest pochodna często słyszymy, że jest to tangens kąta nachylenia stycznej do funkcji w danym punkcie. W rzeczywistości możemy spojrzeć na tę styczną jako na najlepsze liniowe przybliżenie różniczkowanej funkcji w danym punkcie.



Rysunek 4.1: Funkcja $\sin(x)$ oraz jej liniowe przybliżenie (lewa). Obserwujemy duże błędy przybliżenia dla dużych x, ponieważ funkcja sinus jest mocno nieliniowa. Jednak jeżeli spojrzymy na zbliżenie tej funkcji w okolicach $x = 0.5$ (prawa) widzimy, że funkcję można dobrze lokalnie przybliżyć przez linię prostą.

Załóżmy, że mamy daną (potencjalnie mocno nieliniową) funkcję f, jednak trzymamy kciuki, żeby w małej okolicy wybranego przez nas punktu x_0 była możliwość dobrego przybliżenia tej funkcji linią prostą. Linia prosta ma oczywiście równanie postaci $\hat{f}(x) = ax + b$. Zastanówmy się co dla nas oznacza „dobrze przybliżenie" funkcji f w okolicach punktu x_0.

Po pierwsze chcielibyśmy, żeby różnica $f(x) - \hat{f}(x)$ była jak najmniejsza. Jednak wiemy, że nie możemy tego uzyskać dla całego zakresu x i próbujemy to uzyskać jedynie *lokalnie*, dla okolic naszego wybranego punktu x_0. W związku z tym błąd przybliżenia będziemy skalować względem odległości od naszego punktu x_0:

$$\frac{f(x) - \hat{f}(x)}{x - x_0}$$

im odległość jest większa tym błąd podzielimy przez większą liczbę, jednocześnie go zmniejszając. Co więc ostatecznie rozumiemy przez „funkcja liniowa $\hat{f}(x)$ dobrze lokalnie przybliża funkcję f"? No więc oznacza to, że

(a) funkcja $\hat{f}(x)$ przechodzi dokładnie przez punkt $(x_0, f(x_0))$ czyli popełnia dokładnie zerowy błąd w wybranym przez nas punkcie ($\hat{f}(x_0) = f(x_0)$)
(b) przynajmniej w bardzo maleńkiej okolicy (której wielkość dąży do 0, $x \to x_0$) nasze przybliżenie popełnia zerowy błąd!

$$\lim_{x\to x_0} \frac{f(x) - \hat{f}(x)}{x - x_0} = \lim_{x\to x_0} \frac{f(x) - (ax+b)}{x - x_0} = 0$$

Jeżeli takiej linii nie można znaleźć to najzwyczajniej w świecie uważamy, że nie da się przybliżyć funkcji f w okolicach x_0 linią prostą.

Spróbujmy wyznaczyć tę linię. Skoro wiemy, że linia przechodzi przez punkt $(x_0, f(x_0))$ to linia musi spełniać równanie:

$$f(x_0) = ax_0 + b \qquad \Rightarrow \qquad b = f(x_0) - ax_0$$

Dalej podstawiając do wzoru

$$\begin{aligned}
\lim_{x\to x_0} \frac{f(x) - (ax+b)}{x - x_0} &= \lim_{x\to x_0} \frac{f(x) - (ax + f(x_0) - ax_0)}{x - x_0} \\
&= \lim_{x\to x_0} \frac{f(x) - ax - f(x_0) + ax_0}{x - x_0} \quad \text{/wyciągnijmy } a \text{ przed nawias/} \\
&= \lim_{x\to x_0} \frac{f(x) - f(x_0) - a(x - x_0)}{x - x_0} = \lim_{x\to x_0} \left[\frac{f(x) - f(x_0)}{x - x_0}\right] - a
\end{aligned}$$

Jeżeli więc wymagamy by tak zdefiniowany błąd (w bardzo maleńkim otoczeniu x_0) był zerowy to po przyrównaniu go do zera otrzymujemy:

$$a = \lim_{x\to x_0} \frac{f(x) - f(x_0)}{x - x_0}$$

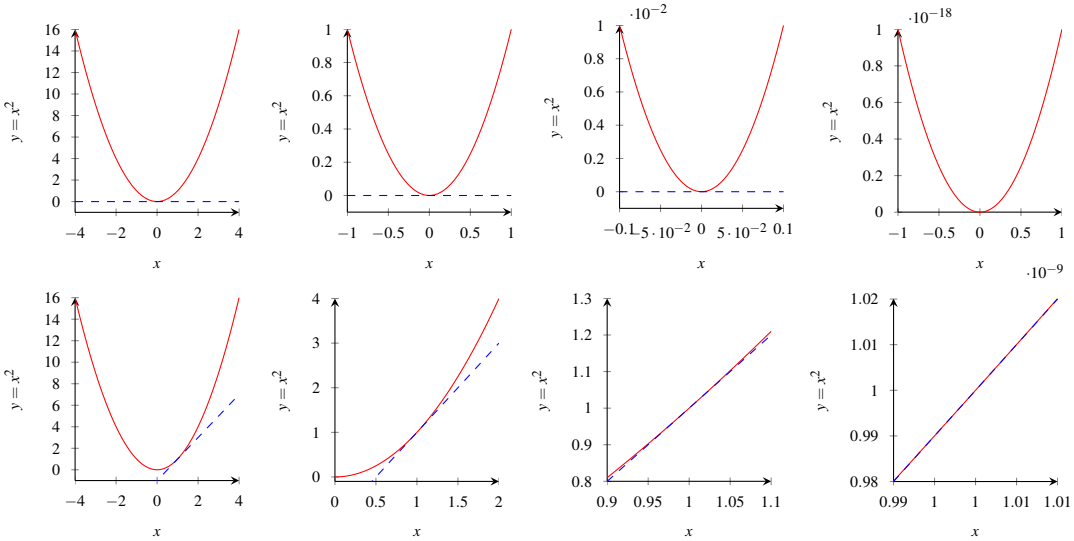
gdzie a to współczynnik kierunkowy naszego przybliżenia. Dokonując zamiany zmiennych $\Delta x = x - x_0$ odkrywamy, że w rzeczywistości wzór po prawej stronie znaku równości to definicja pochodnej!

$$a = \lim_{x\to x_0} \frac{f(x) - f(x_0)}{x - x_0} = \lim_{\Delta x\to 0} \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} = f'(x_0)$$

Można więc intuicyjnie powiedzieć, że pochodna istnieje tylko gdy funkcję można lokalnie przybliżyć linią prostą z zerowym błędem.

Słowa przestrogi

Idąc dalej z tą analogią można sądzić że funkcja jest różniczkowalna jeżeli przy dostatecznie dużym zoomie jest ona linią prostą. Nie jest to prawda. Spójrz na kolejne zbliżenia funkcji x^2 w punkcie 0 i 1. W obydwu punktach funkcja jest różniczkowalna, a pomimo tego niezależnie jak byśmy bardzo zbliżali wykres w punkcie 0, funkcja zawsze wygląda jak parabola. Dlaczego? Cóż, jak matematyk mówi o nieskończenie małej lokalnej okolicy to właśnie ma namyśli *nieskończenie małą* okolicę. Pomimo tego jest to użyteczna intuicja dot. pochodnej i przydaje się myśleć o pochodnej jako o liniowym przybliżeniu funkcji.



Co na to nasz algorytm?

Fakt, że pochodna w rzeczywistości jest rozwiązaniem problemu liniowego przybliżenia funkcji daje nam ciekawą zauważkę (ang. *insight*) dot. algorytmu spadku wzdłuż gradientu. Skoro liczymy w każdej iteracji pochodną to *de facto* w każdej iteracji konstruujemy liniowe przybliżenie naszej funkcji celu. Dalej wiemy, że aby znaleźć minimum[3] funkcji liniowej to przy współczynniku kierunkowym $a > 0$ musimy go szukać jak najbardziej w lewo ($-\infty$), a przy $a < 0$ musimy go szukać jak najbardziej w prawo. W związku z tym wykonujemy krok w kierunku przeciwnym do znaku a (czyli naszego liniowego przybliżenia, pochodnej). Gdyby nasze liniowe przybliżenie było prawidłowe na całej dziedzinie to chcielibyśmy wykonać nieskończenie duży krok w tym kierunku, jednak ponieważ nasze przybliżenie jest dobre jedynie lokalnie to wykonujemy pewien mały kroczek (którego wielkość reguluje η i $|a|$) w kierunku minimum tego przybliżenia.

! Można formalnie pokazać, że krok aktualizacji w algorytmie jest rozwiązaniem problemu minimalizacji „zminimalizuj jak najbardziej funkcję stycznej + nie odchodź za bardzo od okolicy, bo przybliżenie jest lokalne”. Ale o tym na kolejnych laboratoriach ;)

4.2.2 Wzór Taylora

Intuicje, że pochodna jest liniowym przybliżeniem funkcji w okolicach pewnego punktu x_0 można także zbudować na podstawie wzoru Taylora. Rozdział ten jest niezwykle ważny, ponieważ w przyszłości pozwoli nam na konstrukcje algorytmów optymalizacyjnych, które przybliżają funkcję lepiej niż linią prostą!

Definicja 4.3 — Twierdzenie Taylora. Niech $K \geq 1$ będzie liczbą całkowitą, a funkcja f będzie K razy różniczkowalna w punkcie x_0, wtedy istnieje taka funkcja h_K że

$$f(x) = \sum_{k=0}^{K} \left(\frac{(x-x_0)^k}{k!} f^{(k)}(x_0) \right) + h_K(x)(x-x_0)^K$$

[3] Formalnie minimum nie istnieje

a $\lim_{x\to x_0} h_K(x) = 0$ (reszta dąży do 0 gdy x dąży do x_0 –przybliżenie lokalne).

Jak zbudować przybliżenie funkcji korzystając z tego twierdzenia? Na początku musisz ustalić jakiś punkt x_0 wokół którego budujesz przybliżenie. Pamiętaj, że tak zbudowane przybliżenie jest dobre tylko w okolicach tego punktu, powinieneś więc wybrać takie x_0 które jest stosunkowo najbliższe $x$om dla których będziesz chciał przybliżyć sobie wartość funkcji. Po wybraniu tego punktu obliczmy wartość pochodnej w punkcie x_0 oraz tyle pochodnych ile tylko jesteśmy w stanie obliczyć rozbudowując nasz wzór. Im więcej ich dodamy tym nasze przybliżenie będzie lepsze.

$$\begin{aligned}
f(x) &\approx \sum_{k=0}^{K}\left(\frac{(x-x_0)^k}{k!}f^{(k)}(x_0)\right)\\
&= \frac{(x-x_0)^0}{0!}f(x_0)+\frac{(x-x_0)^1}{1!}f(x_0)+\frac{(x-x_0)^2}{2!}f(x_0)+\ldots\\
&= f(x_0)+(x-x_0)f'(x_0)+\frac{1}{2}(x-x_0)^2f''(x_0)+\ldots\\
&= \underbrace{f(x_0)}_{\text{stała}}+x\underbrace{f'(x_0)}_{\text{stała}}-\underbrace{x_0f'(x_0)}_{\text{stała}}+\underbrace{\frac{1}{2}f''(x_0)}_{\text{stała}}x^2-x\underbrace{x_0f''(x_0)}_{\text{stała}}+\underbrace{x_0^2f''(x_0)}_{\text{stała}}+\ldots
\end{aligned}$$

Zwróć uwagę, że po ustaleniu punktu x_0 wokół którego budujemy przybliżenie oraz obliczeniu jego pochodnych otrzymujemy przybliżenie funkcji wielomianem stopnia K! Jak podejrzewasz może być to niezwykle użyteczne, bo dowolną[4] funkcję możesz w taki sposób przybliżyć wielomianem, który najczęściej jest łatwiejszy w obliczeniach (i w optymalizacji, jeśli jest niskiego rzędu!).

Ustalając $K = 1$ otrzymujemy przybliżenie funkcji poprzez pierwszą pochodną:

$$f(x) \approx \sum_{k=0}^{1}\left(\frac{(x-x_0)^k}{k!}f^{(k)}(x_0)\right) = \frac{(x-x_0)^0}{0!}f(x_0)+\frac{(x-x_0)^1}{1!}f'(x_0) = f(x_0)+f'(x_0)(x-x_0)$$

Jest to przybliżeniem wielomianem stopnia 1 czyli funkcją liniową. Warto zauważyć, że jest to dokładnie wzór stycznej do funkcji, który wyprowadzaliśmy na poprzednich laboratoriach!

■ **Przykład 4.3** Przybliżmy sobie funkcję $f(x) = 5x^2+10x+5$ poprzez linię prostą wokół punktu $x_0 = 1$. Z treści zadania wynika, że $K = 1$. Otrzymujemy więc z twierdzenia Taylora

$$f(x) \approx f(x_0)+f'(x_0)(x-x_0) = f(1)+f'(1)(x-1)$$

Obliczamy pochodną oraz wartość funkcji w punkcie $x_0 = 1$

$$f(1) = 5\cdot 1^2+10\cdot 1+5 = 20 \qquad f'(x) = 10x+10 \Rightarrow f'(1) = 10+10 = 20$$

Podstawiając do wzoru otrzymujemy:

$$f(x) \approx 20+20(x-1) = 20+20x-20 = 20x$$

■

[4] St. z o.o. – stwierdzenie z ograniczoną odpowiedzialnością

■ **Przykład 4.4** Przybliżmy sobie tę samą funkcję $f(x) = 5x^2 + 10x + 5$, ale tym razem przyjmując $K = 2$. Z twierdzenia Taylora:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{(x - x_0)^2}{2!} f''(x_0) = f(1) + f'(1)(x - 1) + \frac{(x - 1)^2}{2} f''(1)$$

Obliczmy tylko brakującą wartość drugiej pochodnej:

$$f''(x) = (10x + 10)' = 10 \Rightarrow f''(1) = 10$$

Podstawiając do wzoru otrzymujemy:

$$\begin{aligned} f(x) &\approx f(1) + f'(1)(x - 1) + \frac{(x - 1)^2}{2} f''(1) \\ &= 20 + 20(x - 1) + \frac{(x - 1)^2}{2} \cdot 10 \\ &= \cancel{20} + 20x - \cancel{20} + 5(x - 1)^2 \\ &= 20x + 5(x^2 - 2x + 1) = 20x + 5x^2 - 10x + 5 \\ &= 5x^2 + 10x + 5 \end{aligned}$$

Czyli dostaliśmy oryginalną funkcję $f(x)$! Cóż, najlepsze przybliżenie wielomianu stopnia 2 poprzez wielomian stopnia 2 to ten sam wielomian ;)

■

4.2.3 Pochodne w ekstremach

W lokalnym minimum lub maksimum pochodna funkcji wynosi zawsze 0. Jednak pamiętaj, że zerowa wartość pochodnej może także sygnalizować np. punkt siodłowy - nie jest to więc ostateczny wskaźnik, że funkcja osiąga ekstremum. Typ punktu podejrzanego o bycie ekstremum możemy zweryfikować poprzez obliczenie kolejnych pochodnych

- Jeśli druga pochodna jest dodatnia to funkcja osiąga swoje minimum (pamiętasz warunek wypukłości?)
- Jeśli druga pochodna jest ujemna to funkcja osiąga swoje maksimum
- Jeśli druga pochodna jest równa 0 to... masz kłopoty!!! Musisz policzyć kolejne pochodne, aż znajdziesz pierwszą, która się nie zeruje.
  - Jeśli jest to pochodna rzędu nieparzystego to nie jest to ekstremum
  - Jeśli jest to pochodna rzędu parzystego to interpretujesz ją jak drugą pochodną

■ **Przykład 4.5** Z jakim typem punktu mamy do czynienia w $x = 0$ funkcji $f(x) = x^3$?
Obliczmy pierwszą pochodną

$$f'(x) = 2x^2 \Rightarrow f'(0) = 0$$

Widzimy, że funkcja w danym punkcie ani nie rośnie ani nie maleje. Obliczmy więc drugą pochodną, aby ustalić typ punktu...

$$f''(x) = 6x \Rightarrow f''(0) = 0$$

Co za pech! Druga pochodna się zeruje, więc nie jesteśmy w stanie określić typu punktu... Obliczmy kolejną pochodną

$$f'''(x) = 6 \Rightarrow f''(0) = 6$$

Pierwsza pochodna, która nie wynosi 0 jest trzeciego (czyli nieparzystego) rzędu. Funkcja nie osiąga ekstremum w tym punkcie. ■

Twierdzenie 4.1 Jeśli funkcja $f(x)$ jest wypukła to warunek $f'(x) = 0$ jest wystarczający, by stwierdzić, że jest to lokalne/globalne minimum.

Powyższe twierdzenie dla funkcji ściśle wypukłych jest proste do udowodnienia używając naszych wcześniej zdefiniowanych reguł. Jak dowiedzieliśmy się na drugich laboratoriach, jednym z warunków ścisłej wypukłości jest $f''(x) > 0$, więc wiemy że jeśli $f'(x) = 0$ to $f''(x) > 0$ czyli mamy minimum. Dla funkcji (nieściśle) wypukłych dowód można znaleźć w Internecie.

4.3 Przybliżanie pochodnej numerycznie

Czasami nie możemy policzyć analitycznie pochodnej funkcji (bo np. jej postać nie jest nam znana), a pomimo tego chcielibyśmy użyć algorytmu który takiej pochodnej używa. Okazuje się, że jest na to sposób! Zauważ, że pochodna to zgodnie z definicją $\lim_{\Delta x \to 0} \frac{f(x+\Delta x)-f(x)}{\Delta x}$. Widzimy w niej, że Δx dąży do 0, czyli tak naprawdę jest ono tak małe jak tylko się da. Gdyby stwierdzić że ta mała różnica wynosi $\Delta x = \varepsilon$, to bez problemu obliczylibyśmy pochodną jako

$$f'(x) \approx \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

Powyższym wzorem możemy obliczyć przybliżoną wartość pochodnej poprzez dwukrotne obliczenie funkcji $f(x)$ a wzór na pochodną nie musi być znany! Z tego powodu wzór ten często używamy w praktyce, gdy policzenie pochodnej nie jest możliwe lub gdy np. nie mamy czasu na implementacje funkcji liczącej pochodną. Oczywiście ε powinien być tak mały jak tylko to możliwe, jednak wybranie za niskiej wartości może doprowadzić do trudności numerycznych (dzielenie przez bardzo małą stałą prawdopodobnie i tak małej różnicy). Praktyczną wartością stosowaną w optymalizacji jest np. $\varepsilon = 10^{-5}$.

! Nawisem mówiąc, funkcja nawet nie musi być różniczkowalna żeby obliczenie takiego przybliżenia było możliwe.

■ **Przykład 4.6** Przybliżmy wartość pochodnej dla $f(x) = x^2$ w punkcie $x = 2$ i zakładając $\varepsilon = 10^{-5}$. Obliczmy najpierw wartość funkcji w punkcie x oraz w punkcie $x + \varepsilon$:

$$f(2) = 2^2 = 4$$

$$f(2+\varepsilon) = (2+0.00001)^2 = 2.00001^2 = 4.00004$$

Przybliżmy pochodną:

$$f'(2) \approx \frac{f(2+\varepsilon) - f(x)}{\varepsilon} = \frac{4.00004 - 4}{0.00001} = \frac{0.00004}{0.00001} = 4$$

Zwróć uwagę, że w żadnej chwili nie użyliśmy wzoru na pochodną funkcji, a do wykonania przybliżenia potrzebujemy jedynie czarnej skrzynki, która jest w stanie obliczyć funkcję celu $f(x)$. Oczywiście jest to przybliżenie i zwykle nie wychodzi ono równe dokładnie pochodnej ;) ■

Czy jednak takie przybliżenie ma jakieś poparcie teoretyczne? Okazuje się, że tak! Na przykład twierdzenie Lagrange'a o wartości średniej mówi, że jeżeli dana funkcja f jest ciągła w przedziale $[a,b]$ oraz różniczkowalna w przedziale (a,b) to istnieje taki punkt $c \in (a,b)$, że

$$f'(c) = \frac{f(b)-f(a)}{b-a}$$

Podstawiając $a = x$ oraz $b = x+\varepsilon$ (czyli $b-a = \varepsilon$) otrzymujemy

$$f'(c) = \frac{f(x+\varepsilon)-f(x)}{\varepsilon}$$

gdzie c należy do przedziału $(x, x+\varepsilon)$. Twierdzenie to nie mówi nam o tym jak dobre jest nasze przybliżenie pochodnej w punkcie x. Wiemy jednak, że obliczona przez nas wartość równa się *dokładnie* pochodnej funkcji dla jakiejś wartości c która jest bardzo blisko naszego x.

Na początku mówiliśmy o algorytmie, który szuka minimum funkcji poprzez spoglądanie na wartości funkcji na prawo i na lewo. Teraz jednak dostaliśmy algorytm który patrzy tylko w prawo ($f(x+\varepsilon)$)! W praktyce okazuje się jednak, że nasza początkowa intuicja jest poprawna. Patrzenie również w lewą stronę $f(x-\varepsilon)$ daje stabilniejsze estymacje pochodnej niż tylko spoglądanie tylko w jedną stronę. Nasze przybliżenie uzyskujemy więc zwykle poprzez obliczenie:

$$f'(x) \approx \frac{f(x+\varepsilon)-f(x-\varepsilon)}{2\varepsilon}$$

Do wyjaśnienia dlaczego tak jest wrócimy jeszcze na kolejnych laboratoriach.

4.4 Gradient

Definicja 4.4 Gradient funkcji $f(\mathrm{x})$ w punkcie x to wektor pochodnych cząstkowych

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \frac{\partial f}{\partial x_3} \\ \dots \end{bmatrix}$$

■ **Przykład 4.7** Policzmy sobie gradient dla funkcji $f(x_1,x_2) = x_1^2 + x_2^2$. Pochodne cząstkowe to odpowiednio:

$$\frac{\partial f}{\partial x_1} = 2x_1 \qquad \frac{\partial f}{\partial x_2} = 2x_2$$

a więc gradient wynosi

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 \\ 2x_2 \end{bmatrix}$$

■

Gradient jest więc w zasadzie użytecznym zapisem matematycznym pozwalającym nam na bardziej przejrzysty zapis oraz na operowanie na wszystkich pochodnych cząstkowych na raz. Wyznaczmy kilka użytecznych wzorów na gradient.

■ **Przykład 4.8 — Gradient funkcji liniowej.** Policzmy gradient następującej funkcji:

$$f(x) = \mathsf{a}^T\mathsf{x} = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = a_1x_1 + a_2x_2 + a_3x_3$$

Gradient to wektor składający się z kolejnych pochodnych cząstkowych. Obliczmy pierwszą pochodną cząstkową:

$$\frac{\partial f}{\partial x_1} = \frac{\partial(a_1x_1 + a_2x_2 + a_3x_3)}{\partial x_1} = \frac{\partial(a_1x_1)}{\partial x_1} + \frac{\partial(a_2x_2)}{\partial x_1} + \frac{\partial(a_3x_3)}{\partial x_1} = a_1 + 0 + 0 = a_1$$

A więc pochodną cząstkową po pierwszym x_1 jest tylko odpowiadająca mu waga a_1. Analogiczne rozumowanie można przeprowadzić dla każdego kolejnego x_i.

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \frac{\partial f}{\partial x_3} \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \mathsf{a}$$

■

Problem 4.1 Pokaż, że dla analogicznej funkcji $f(x) = \mathsf{x}^T\mathsf{a}$ jej gradient również wynosi $\nabla f = \mathsf{a}$. Pomimo tego, że wynik jest już teraz dość oczywisty, zwróć uwagę, że w pokazanym wyżej przykładzie a straciło w gradiencie swoją transpozycję – tutaj nie pojawia się żadna dodatkowa transpozycja i przepisujemy a jak jest.

! Zauważ, że powyższy wzór (jak i następne) pozwalają na policzenie gradientu dla dowolnie długiego wektora x! Atakujemy więc tym zapisem nawet funkcje trylionwymiarowe!

■ **Przykład 4.9 — Gradient funkcji kwadratowej.** Policzmy gradient następującej funkcji:

$$f(x) = \mathsf{x}^T\mathsf{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = x_1^2 + x_2^2 + x_3^2$$

Gradient to wektor składający się z kolejnych pochodnych cząstkowych. Obliczmy pierwszą pochodną cząstkową:

$$\frac{\partial f}{\partial x_1} = \frac{\partial(x_1^2 + x_2^2 + x_3^2)}{\partial x_1} = \frac{\partial(x_1^2)}{\partial x_1} + \frac{\partial(x_2^2)}{\partial x_1} + \frac{\partial(x_3^2)}{\partial x_1} = 2x_1 + 0 + 0 = 2x_1$$

Analogiczne rozumowanie można przeprowadzić dla każdego kolejnego x_i.

$$\nabla f = \nabla(\mathsf{x}^T\mathsf{x}) \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \frac{\partial f}{\partial x_3} \end{bmatrix} = \begin{bmatrix} 2x_1 \\ 2x_2 \\ 2x_3 \end{bmatrix} = 2\mathsf{x}$$

■

■ **Przykład 4.10 — Gradient formy kwadratowej.** Czas na coś trudniejszego! Policzmy gradient następującej funkcji:

$$f(x) = \mathsf{x}^T \mathsf{A}\mathsf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Zacznijmy wymnażanie powyższego iloczynu od lewej strony

$$\mathsf{x}^T \mathsf{A} = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{21}x_2 & a_{12}x_1 + a_{22}x_2 \end{bmatrix}$$

Zwróć uwagę, że elementy z pierwszego wiersza $a_{1\cdot}$ są zawsze mnożone przez x_1, a elementy drugiego wiersza są mnożone przez x_2. Dokończmy mnożenie:

$$\mathsf{x}^T \mathsf{A}x = \begin{bmatrix} a_{11}x_1 + a_{21}x_2 & a_{12}x_1 + a_{22}x_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = a_{11}x_1^2 + a_{21}x_2x_1 + a_{12}x_1x_2 + a_{22}x_2^2$$

Znów, zwróć uwagę na ciekawą korespondecję pomiędzy indeksami elementów a wymnażanymi indeksami. Element a_{21} stanowi wagę dla x_2x_1, element a_{12} jest wagą x_1x_2. Z kolei element a_{22} to waga dla $x_2x_2 = x_2^2$.

Jak będą wyglądały pochodne cząstkowe względem x_1 i x_2?

$$\frac{\partial f}{\partial x_1} = \frac{\partial(a_{11}x_1^2 + a_{21}x_2x_1 + a_{12}x_1x_2 + a_{22}x_2^2)}{\partial x_1} = 2a_{11}x_1 + (a_{21} + a_{12})x_2$$

$$\frac{\partial f}{\partial x_2} = \frac{\partial(a_{11}x_1^2 + a_{21}x_2x_1 + a_{12}x_1x_2 + a_{22}x_2^2)}{\partial x_2} = (a_{21} + a_{12})x_1 + 2a_{22}x_2$$

W poprzednich zadaniach byliśmy w stanie dojść do eleganckiego zapisu macierzowego, spróbujmy osiągnąć go także tym razem. Rozważmy na razie tylko pierwszą pochodną cząstkową. Stosunkowo łatwo można zauważyć, że można ją wyrazić w postaci mnożenia dwóch wektorów:

$$2a_{11}x_1 + (a_{21} + a_{12})x_2 = \begin{bmatrix} 2a_{11} & a_{21} + a_{12} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Teraz dokładamy jeden dodatkowy wiersz do wektora po lewej stronie uzyskując dodatkowo pochodną po x_2 i w rezultacie otrzymując cały gradient:

$$\nabla f = \begin{bmatrix} 2a_{11} & a_{21} + a_{12} \\ a_{12} + a_{21} & 2a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Jak zapisać macierz po lewej stronie elegancko w rachunku macierzowym? Jest to po prostu $\mathsf{A}^T + \mathsf{A}$!

$$\begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{bmatrix} + \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 2a_{11} & a_{21} + a_{12} \\ a_{12} + a_{21} & 2a_{22} \end{bmatrix}$$

Ostatecznie otrzymujemy więc:

$$\nabla f = \nabla(\mathsf{x}^T \mathsf{A}\mathsf{x}) = (\mathsf{A}^T + \mathsf{A})\mathsf{x}$$

! W szczególności, gdy macierz jest symetryczna zachodzi $\mathsf{A}^T = \mathsf{A}$ czyli $\nabla f = 2\mathsf{A}\mathsf{x}$

■

Inne ważne wzory na gradient:
- $\nabla(af(\mathsf{x}) + bg(\mathsf{x})) = a\nabla f(\mathsf{x}) + b\nabla g(\mathsf{x})$ – zasada liniowości gradientu
- $\nabla f(\mathsf{A}\mathsf{x}) = \mathsf{A}^T \nabla f$

4.4.1 Przykład regresji liniowej

Rozważmy przykład znalezienia minimum wielowymiarowej funkcji celu metodami analitycznymi. Za przykład posłuży nam tutaj problem znajdowania minimalnej sumy błędów kwadratowych (ang. *least squares*) na przykładzie regresji. Regresja wieloraka to regresja liniowa w której uwzględniamy kilka zmiennych objaśniających.

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots$$

Na statystyce rozważaliśmy głównie przykład z tylko jedną zmienną i – jeśli nadal to pamiętacie – równania prowadzące nas do wzorów na (tylko dwa) współczynniki β były dość długie i żmudne. Spróbujmy zobaczyć jak to wygląda w przypadku wielowymiarowym używając naszej wiedzy o gradiencie.

Korzystając z naszej wcześniejszej wiedzy o mnożeniu wektorów, równanie regresji trochę się upraszcza:

$$\hat{y} = \beta_0 + \mathsf{x}^T \beta$$

Aby jeszcze uprościć ten zapis dodajemy do wektora x sztuczną, dodatkową wartość równą jeden. Pozwala nam to na wpisanie stałej β_0 do wektora β i dalsze uproszczenie zapisu:

$$\hat{y} = \mathsf{x}^T \beta$$

Kontynuujmy upraszczanie: upakujmy kolejne obserwacje x jako wiersze macierzy X zawierającej wszystkie obserwacje ze zbioru danych. Zauważ, że jej kolumny zawierają wartości kolejnych zmiennych modelu. Macierzy tej odpowiada wektor y mający tyle wierszy (elementów) ile jest obserwacji, i zawierający wartości zmiennej zależnej. W takim wypadku możemy zapisać „globalne” równanie regresji obliczające wartość $\hat{y}$ dla każdej obserwacji na raz!

$$\hat{\mathsf{y}} = \mathsf{X}\beta$$

Zauważ, że po wykonaniu tego równania macierzowego otrzymujesz wektor $\hat{\mathsf{y}}$, który zawiera od razu predykcje naszej regresji dla każdej obserwacji w macierzy! Suma kwadratów rezyduów może więc zostać wyrażona wzorem:

$$SSE = (\mathsf{y} - \mathsf{X}\beta)^T (\mathsf{y} - \mathsf{X}\beta)$$

Dlaczego ten wzór jest prawdziwy? Wcześniej pokazaliśmy, że $\mathsf{x}^T\mathsf{x}$ to po prostu suma kwadratów $\sum + ix + i^2$. My chcemy policzyć sumę kwadratów błędów gdzie pojedynczy błąd to różnica między naszą predykcją (wynikającą z regresji liniowej) a tym co powinniśmy uzyskać. Pojedynczą predykcję uzyskujemy poprzez $\hat{y} = \mathsf{x}^T\beta$. Przepisując to na nasz zapis macierzowy musimy z naszej macierzy zawierającej wszystkie dane X wyciągnąć pojedynczy wiersz zawierający dane naszego konkretnego x. Otrzymujemy

więc $\hat{y} = \mathsf{X}_{i,*}\beta$, a błąd na tym pojedynczym przykładzie to $\mathsf{y}_i - \mathsf{X}_{i,*}\beta$. A więc wektor zawierający wszystkie błędy (dla wszystkich przykładów) to $\mathsf{y} - \mathsf{X}\beta$. Suma kwadratów tego wektora uzyskujemy poprzez wyrażenie $\mathsf{x}^T\mathsf{x}$ czyli $(\mathsf{y} - \mathsf{X}\beta)^T(\mathsf{y} - \mathsf{X}\beta)$.

Policzmy teraz gradient po SSE szukając współczynników regresji liniowej. Zacznijmy od wymnożenia nawiasów w SSE.

$$\begin{aligned} SSE &= (\mathsf{y} - \mathsf{X}\beta)^T(\mathsf{y} - \mathsf{X}\beta) && \text{/* transpozycje wciągamy do środka */} \\ &= (\mathsf{y}^T - \beta^T\mathsf{X}^T)(\mathsf{y} - \mathsf{X}\beta) && \text{/* wymnażamy*/} \\ &= \mathsf{y}^T\mathsf{y} - \beta^T\mathsf{X}^T\mathsf{y} - \mathsf{y}^T\mathsf{X}\beta + \beta^T\mathsf{X}^T\mathsf{X}\beta \end{aligned} \tag{4.1}$$

Na takiej postaci *SSE* jest już prosto zastosować poznane przez nas wzory na gradient. Dla uproszczenia zaaplikujmy gradient do każdej z części tego wyrażenia. Zwróć uwagę, że zmienną po której liczmy tutaj gradient jest wektor β.

$$\nabla(\beta^T\mathsf{X}^T\mathsf{y}) = \mathsf{X}^T\mathsf{y} \qquad \nabla(\mathsf{y}^T\mathsf{X}\beta) = (\mathsf{y}^T\mathsf{X})^T = \mathsf{X}^T\mathsf{y} \qquad \nabla(\beta^T\mathsf{X}^T\mathsf{X}\beta) = 2\mathsf{X}^T\mathsf{X}\beta$$

W ostatniej pochodnej skorzystaliśmy z faktu, że $\mathsf{X}^T\mathsf{X}$ jest macierzą symetryczną. Zapisując cały gradient dostajemy:

$$\nabla SSE = 0 - \mathsf{X}^T\mathsf{y} - \mathsf{X}^T\mathsf{y} + 2\mathsf{X}^T\mathsf{X}\beta = -2\mathsf{X}^T\mathsf{y} + 2\mathsf{X}^T\mathsf{X}\beta$$

Z przyrównania tego gradientu do zera (a w zasadzie wektora zer) otrzymujemy:

$$\begin{aligned} -2\mathsf{X}^T\mathsf{y} + 2\mathsf{X}^T\mathsf{X}\beta &= 0 \\ 2\mathsf{X}^T\mathsf{X}\beta &= 2\mathsf{X}^T\mathsf{y} \\ \beta &= (\mathsf{X}^T\mathsf{X})^{-1}\mathsf{X}^T\mathsf{y} \end{aligned}$$

Co bardzo interesujące, takie postawienie problemu pozwala nam na zminimalizowanie sumy kwadratów poprzez jedno równanie macierzowe[5]! Równanie to czasami jest nazwane równaniem normalnym (ang. *normal equation*).

4.4.2 Reguła łańcuchowa

Definicja 4.5 — Reguła łańcuchowa. Pochodna funkcji złożonej $g(x) = f(h_1(x), h_2(x), h_3(x), \ldots)$ dla $x \in \mathbb{R}$ wynosi

$$\frac{dg(x)}{dx} = \sum_{i=1}^{n} \frac{\partial g(x)}{\partial h_i} \frac{dh_i(x)}{dx}$$

■ **Przykład 4.11** Oblicz pochodną $z = x^2y^2$ jeżeli $x = \cos(t)$ a $y = \sin(2t)$. Korzystając z reguły łańcuchowej otrzymujemy

$$\frac{dz}{dt} = \frac{\partial z}{\partial x}\frac{dx}{dt} + \frac{\partial z}{\partial y}\frac{dy}{dt} = 2xy^2 \cdot (-\sin(t)) + 2yx^2 \cdot 2\cos(2t)$$

Włączając wzory na x i y otrzymujemy:

$$\frac{dz}{dt} = 2\cos(t)\sin^2(2t) \cdot (-\sin(t)) + 2\sin(2t)cos^2(t) \cdot 2\cos(2t)$$

Powyższy wynik można oczywiście dalej upraszczać korzystając z własności funkcji trygonometrycznych. ■

[5] W drugim równaniu korzystamy z pomnożenia obustronnie równania z lewej strony przez $(\mathsf{X}^T\mathsf{X})^{-1}$. Ponieważ jest to odwrotność macierzy $\mathsf{X}^T\mathsf{X}$ macierze te się skracają. $(\mathsf{X}^T\mathsf{X})^{-1}\mathsf{X}^T\mathsf{X} = \mathsf{I}$

Powyższy przykład jest dość zastanawiający. Czy nie można było od razu przekształcić funkcję do $z = \cos^2(t)\sin^2(2t)$ i policzyć zwykłej pochodnej? Można było. Jednak reguła łańcuchowa jest przydatna do np. wyrażania ogólnych wzorów gdy funkcja jest nieznana.

■ **Przykład 4.12** Mamy jakąś funkcję $f(x,y)$ zależną od współrzędnych kartezjańskich x i y. Chcielibyśmy policzyć jednak pochodną tej funkcji w zależności od współrzędnych biegunowych (r,α). Przypomnienie: $x = r\cos\alpha$, a $y = r\sin\alpha$. Ogólny wzór na pochodną względem r wynosi

$$\frac{\partial f}{\partial r} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial r} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial r} = \frac{\partial f}{\partial x}\cos\alpha + \frac{\partial f}{\partial y}\sin\alpha$$

■

■ **Przykład 4.13** Rozważmy pochodną następującej funkcji $f(x) = g(x)k(x)$. Znamy wzór na mnożenie pochodnej, jednak załóżmy, że akurat go zapomnieliśmy i mamy pod ręką tylko regułę łańcuchową dla funkcji wielowymiarowej.

Przeróbmy więc naszą funkcję na funkcję wielowymiarową ;)

$$f_{multi}(h_1,h_2) = h_1h_2$$

gdzie odpowiednio $h_1 = g(x)$, a $h_2 = k(x)$. Czyli naszą oryginalną funkcję możemy zapisać jako $f(x) = f_{multi}(g(x),k(x))$. Zauważ, że

$$\frac{\partial f_{multi}(h_1,h_2)}{\partial h_1} = \frac{\partial(h_1h_2)}{\partial h_1} = h_2$$

ponieważ licząc pochodną zakładamy że inne zmienne (czyli h_2) są stałymi[6]! Stosując regułę łańcuchową:

$$\frac{df_{multi}}{dx} = \frac{\partial f_{multi}}{\partial h_1}\frac{dh_1}{dx} + \frac{\partial f_{multi}}{\partial h_2}\frac{dh_2}{dx} = h_2\frac{dh_1}{dx} + h_1\frac{dh_2}{dx}$$

Wracając do naszych oznaczeń $f(x) = f_{multi}(g(x),k(x))$ czyli

$$\frac{df(x)}{dx} = k(x)\frac{dg(x)}{dx} + g(x)\frac{dk(x)}{dx} = k(x)g'(x) + k'(x)g(x)$$

■

Literatura

Literatura powtórkowa

Dokładny opis pochodnych oraz przykłady obliczeń można znaleźć w dowolnej książce do analizy matematycznej np. [1].

Literatura dla chętnych

W tym tygodniu zachęcamy do zapoznania się z algorytmami do automatycznego liczenia pochodnych. Wśród takich technik popularne jest pojęcie grafu obliczeń (ang. *computational graph*) i liczenie pochodnych na takim grafie. Opis grafów obliczeń wraz z algorytmem wstecznej propagacji można znaleźć np. na stronie `http://colah.github.io/posts/2015-08-Backprop/`

[6] Czyli tak jakby ktoś się nas zapytał ile wynosi pochodna z $(ax)' = a$

Bibliografia

[1] M. Gewert i Z. Skoczylas. *Analiza matematyczna 1: Definicje, twierdzenia, wzory*. Oficyna Wydawnicza GiS, 2007.

5. Metoda najszybszego spadku

5.1 Wielowymiarowa funkcja kwadratowa

Na ostatnich zajęciach poznaliśmy podstawy dotyczące funkcji wielowymiarowych. Szczególnie interesującą dla nas klasą funkcji będą funkcje kwadratowe. Wynika to z kilku powodów: po pierwsze możemy w prosty sposób wyznaczyć punkt stacjonarny takiej funkcji:

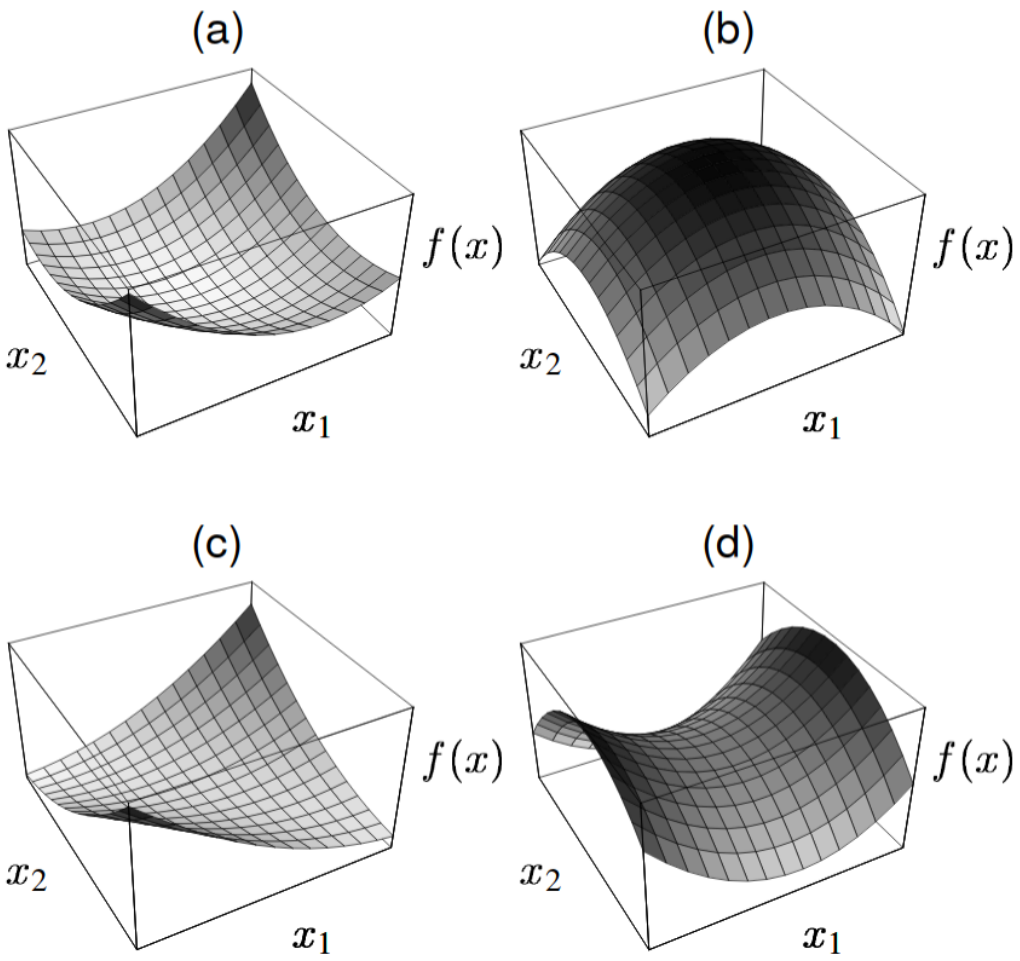
$$f(\mathsf{x}) = \frac{1}{2}\mathsf{x}^\mathsf{T}\mathsf{A}\mathsf{x} - \mathsf{b}^\mathsf{T}\mathsf{x} + c$$

$$\nabla f(\mathsf{x}) = \frac{1}{2}2\mathsf{A}\mathsf{x} - \mathsf{b} + 0 = \mathsf{A}\mathsf{x} - \mathsf{b}$$

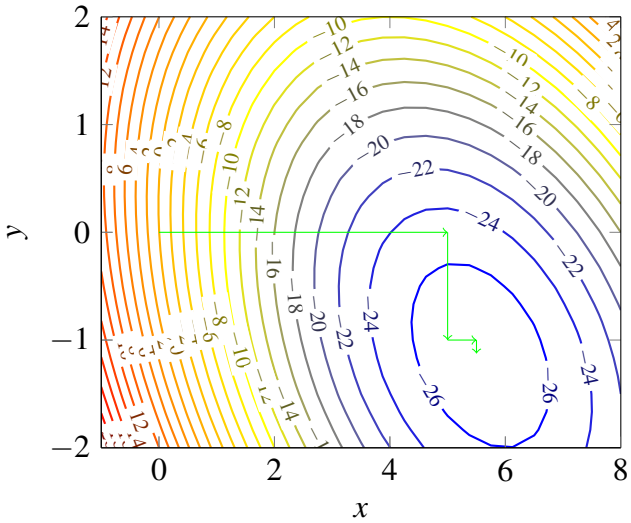
$$\mathsf{A}\mathsf{x} - \mathsf{b} = 0 \quad \Rightarrow \quad \mathsf{x}^* = \mathsf{A}^{-1}\mathsf{b}$$

Dlaczego, więc skoro możemy obliczyć punkt stacjonarny takiej funkcji w prosty, analityczny sposób mielibyśmy stosować dla takich funkcji algorytmy optymalizacyjne? Ponieważ w przypadku problemów o wysokiej wymiarowości macierz A jest bardzo duża, a jej odwracanie ma złożoność sześcienną. Z tego powodu, inżynierowie nawet wtedy gdy ich zadaniem jest policzenie wyrażenia $\mathsf{A}^{-1}\mathsf{b}$ dla dużej macierzy używają algorytmów optymalizacyjnych zamiast np. metody eliminacji Gaussa.

Po drugie, funkcje kwadratowe pozwalają nam na pokazanie różnych trudności na które napotykamy się w optymalizacji. Funkcja kwadratowa może być zarówno funkcją wypukłą (ściśle i nie), funkcją wklęsła czy modelować punkt siodłowy – patrz rysunek 5.1. Po trzecie, jak się później okaże wiele funkcji w okolicy minimum możemy dość dobrze lokalnie przybliżyć funkcją kwadratową, wiec nasze analizy funkcji kwadratowej będą się w naturalny sposób odnosić do większej gamy funkcji. Na koniec warto też podkreślić, że optymalizacja funkcji kwadratowej ma duże znaczenie praktyczne: maszyny wektorów nośnych czy wielowymiarowa regresja liniowa to kilka najprostszych metod uczenia maszynowego, które wymagają optymalizacji funkcji kwadratowych.



Rysunek 5.1: Cztery typy funkcji modelowanej przez funkcje kwadratową.



Rysunek 5.2: Przykład działania metody Gaussa-Seidela

5.2 Jak optymalizować w wielu wymiarach?

5.2.1 Idea dekompozycji na problemy jednowymiarowe

Dotychczas poznane metody optymalizacyjne działają w jednym wymiarze. Czy możemy wykorzystać ich ideę do optymalizacji funkcji wielu zmiennych? Jednym z najprostszych pomysłów jest iteracyjne optymalizowanie każdej współrzędnej problemu z osobna. Tę ideę utożsamia algorytm Gaussa-Seidela, którego działanie można zobaczyć na rysunku 5.2.

W każdej jego iteracji definiowany jest sztuczny, jednowymiarowy problem:

$$\min_{\alpha} g(\alpha) = f(x+\alpha, y, z, \ldots)$$

który zwraca długość najlepszego przesunięcia na danej współrzędnej. Zauważ, że jest to funkcja jednej zmiennej, więc można ją zoptymalizować algorytmami dychotomizacji czy złotego podziału. W kolejnych krokach tego typu problemy są definiowane dla każdej współrzędnej z osobna.

Dużą wadą tego algorytmu jest jego duża złożoność rosnąca z liczbą wymiarów. Aby wykonać jedną pełną iterację algorytmu, należy przeiterować po każdej współrzędnej i wykonać optymalizację funkcji jednej zmiennej dla każdej z nich. Z tego powodu interesujące będą dla nas algorytmy, które optymalizują wszystkie wymiary naraz.

5.2.2 Metoda najszybszego spadku

Algorytm 5.1 — Algorytm najszybszego spadku (Cauchy'ego).

$\mathsf{x}_0 \leftarrow$ INICJALIZUJ

while warunek stopu nie jest spełniony **do**

$$\eta_t = arg \min_{\eta>=0} f(\mathsf{x}_t - \eta \nabla f(\mathsf{x}_t))$$

$$\mathsf{x}_{t+1} = \mathsf{x}_t - \eta_t \nabla f(\mathsf{x}_t)$$

end while

Przy następujących oznaczeniach:

- f - funkcja, którą chcemy optymalizować
- x_t - wektor, argument w iteracji t
- $\nabla f(x_t)$ - gradient z minimalizowanej funkcji
- η - rozmiar kroku (optymalizowany)

Warto zauważyć, ze w algorytmie najszybszego spadku kolejne kierunki poprawy (gradienty) są do siebie ortogonalne.

Ćwiczenie 5.1 Udowodnij powyższą własność. ■

5.2.3 Właściwości gradientu

Jak pokazaliśmy w skrypcie do poprzednich laboratorium, pochodna $f'(x_0)$ jest liniowym przybliżeniem różniczkowanej funkcji wokół danego punktu x_0. Z naszych obliczeń wynikało, że[1]

$$\hat{f}(x) = f(x_0) + f'(x_0)(x - x_0)$$

(co de facto jest rozwinięciem tej funkcji w szereg Taylora z dokładnością do wyrazów pierwszego stopnia). To przybliżenie jest również prawdziwe, gdy przejdziemy na funkcje wielowymiarowe tj. zamienimy pochodną na gradient:

$$\hat{f}(\mathsf{x}) = f(\mathsf{x}_0) + \nabla f(\mathsf{x}_0)^T(\mathsf{x} - \mathsf{x}_0)$$

Wykorzystamy ten wzór, aby pokazać kilka ważnych właściwości gradientu.

Jednak zanim to nastąpi przypomnijmy, że kosinus kąta pomiędzy dwoma wektorami możemy wyznaczyć wzorem:

$$\cos(\mathsf{x},\mathsf{y}) = \frac{\mathsf{x}^\mathsf{T}\mathsf{y}}{||\mathsf{x}||_2||\mathsf{y}||_2} \qquad \Rightarrow \qquad \mathsf{x}^\mathsf{T}\mathsf{y} = \cos(\mathsf{x},\mathsf{y})||\mathsf{x}||_2||\mathsf{y}||_2$$

gdzie $||\mathsf{x}||_2$ to długość wektora x. Jak pamiętamy, kosinus z 90 stopni wynosi zero, więc wektory prostopadłe spełniają własność $\mathsf{x}^\mathsf{T}\mathsf{y} = 0$.

Wróćmy do naszego algorytmu. Załóżmy, że w danej iteracji jesteśmy w punkcie x_0 i zastanawiamy się w którą stronę powinniśmy pójść tak aby uzyskać największy spadek wartości funkcji. Miarą spadku jest oczywiście różnica pomiędzy wartością funkcji w potencjalnym przyszłym punkcie x, a jej wartością w aktualnym punkcie x_0. Różnicę tę możemy wyznaczyć z naszego przybliżenia:

$$f(\mathsf{x}) \approx f(\mathsf{x}_0) + \nabla f(\mathsf{x}_0)^T(\mathsf{x} - \mathsf{x}_0)$$

$$f(\mathsf{x}) - f(\mathsf{x}_0) \approx \nabla f(\mathsf{x}_0)^T(\mathsf{x} - \mathsf{x}_0)$$

$$df \approx \nabla f(\mathsf{x}_0)^T d\mathsf{x} = \cos(\nabla f(\mathsf{x}_0), d\mathsf{x})||\nabla f(\mathsf{x}_0)||_2||d\mathsf{x}||_2$$

W jakim więc kierunku powinniśmy się przesunąć, aby różnica była największa? Ponieważ długości gradientu nie możemy modyfikować, a długości wektora $\mathsf{x} - \mathsf{x}_0$ też nie chcemy modyfikować, ponieważ szukamy kierunku – możemy zmaksymalizować jedynie kosinus

[1] Wróć do poprzedniego skryptu i wyznacz ten wzór.

kąta. Kosinus $\cos(\nabla f(\mathsf{x}_0), d\mathsf{x})$ osiąga maksymalną wartość dla 0 stopni (równe 1), co oznacza że kierunek największej różnicy jest równoległy do gradientu! Innymi słowy gradient *jest kierunkiem najszybszego lokalnego wzrostu*. Analogicznie, ujemny gradient ($\cos(\nabla f(\mathsf{x}_0), d\mathsf{x}) = -1$) jest kierunkiem najszybszego lokalnego spadku.

Z podanej zależności możemy się też zastanowić jak wygląda położenie gradientu względem kierunku warstwicy/poziomicy funkcji – czyli linii łączącej punkty o tej samej wysokości. Otóż, skoro warstwica łączy punkty o takiej samej wartości $f(\mathsf{x})$, to różnica wartości funkcji w jej kierunku wynosi zero.

$$0 = \cos(\nabla f(\mathsf{x}_0), d\mathsf{x})||\nabla f(\mathsf{x}_0)||_2||d\mathsf{x}||_2$$

Zakładając, że nie jesteśmy w punkcie stacjonarnym (minimum, maksimum, punkt siodłowy) to długość gradientu nie jest zerowa. Wynika stąd, że $\cos(\nabla f(\mathsf{x}_0), d\mathsf{x}) = 0$ czyli kąt pomiędzy gradientem a kierunkiem poziomicy wynosi 90 lub 270 stopni. Innymi słowy, *gradient jest prostopadły do poziomicy funkcji*.

5.3 Wektory i wartości własne

Niniejszy rozdział prezentuje powtórkę materiały z algebry liniowej dotyczącą wektorów i wartości własnych macierzy (ang. *eigenvalue, eigenvector*), które są kluczowe do analizy formy kwadratowej (i nie tylko!).

W ogólności, jeśli przemnożymy jakieś wektor x przez macierz A dostaniemy zwykle wektor Ax, który jest w pewien sposób zniekształcony. Jak np. pamiętasz z zajęć o grafice komputerowej istnieją macierze, które obracają wektory o zadany kąt, są macierze które wektory pochylają i wydłużają, odbijają itd. Czasami jednak istnieje specjalna grupa wektorów, których macierz A nie potrafi przekręcić, a jedynie je wydłuża bądź skraca (wliczamy w to tak duże skrócenie, że wektor zmienia swój zwrot). Dlaczego jest to interesujące? Ponieważ cała operacja mnożenia przez macierz (złożoność kwadratowa) może być zniwelowana (bądź zastąpiona) poprzez mnożenie przez zwykłą liczbę (złożoność liniowa)! Przykładowo: jeśli macierz A wydłużyła dany wektor o dwa razy to wystarczy podzielić wszystkie elementy wektora przez 2, żeby odwrócić jej działanie.

Tego typu wektory, które dla rozróżnienia od zwykłych będziemy oznaczali jako q zamiast x. spełniają więc następujące równanie:

$$\mathsf{Aq} = \lambda \mathsf{q}$$

gdzie λ jest zwykłą liczbą. Ta równość to po prostu matematyczny zapis zdefiniowanej przez nas wcześniej właściwości: wektor q pomnożony przez q po prostu się wydłuża-/skraca o stałą λ. Wektory q_i nazywamy wektorami własnymi macierzy A, a skojarzone z nimi wartości λ_i nazywamy wartościami własnymi. Pominę tutaj opis wyznaczania wektorów i wartości własnych macierzy (to zadanie kursu z algebry liniowej) i tylko skupię się na kilku właściwościach i intuicjach.

Przede wszystkim dwa podstawowe fakty: wektor zerowy byłby wektorem własnym każdej macierzy z dowolną λ bo

$$\mathsf{A}\vec{0} = \lambda\vec{0} = \vec{0}$$

z tego powodu nie bierzemy go pod uwagę. Dodatkowo jeśli znaleźliśmy już wektor q który ma tę własność z pewną stałą λ to od razu znaleźliśmy nieskończoną liczbę wektorów

własnych z tę samą wartością własną: $2\mathsf{q}$, $3\mathsf{q}$, $4\mathsf{q}$, $\frac{1}{2}\mathsf{q}$, $-5\mathsf{q}$, ...

$$\mathsf{A}(2\mathsf{q}) = 2\mathsf{A}\mathsf{q} = 2\lambda\mathsf{q} = \lambda(2\mathsf{q})$$

Dla uproszczenia będziemy często zakładać, że wektory własne mają długość 1 np. stwierdzenie „macierz ma dwa wektory własne" oznacza, że ma ona dwa wektory własne o długości 1 (które potencjalnie możemy wydłużać/skracać aby uzyskać nieskończoność wektorów własnych).

Kilka prostych pytań:

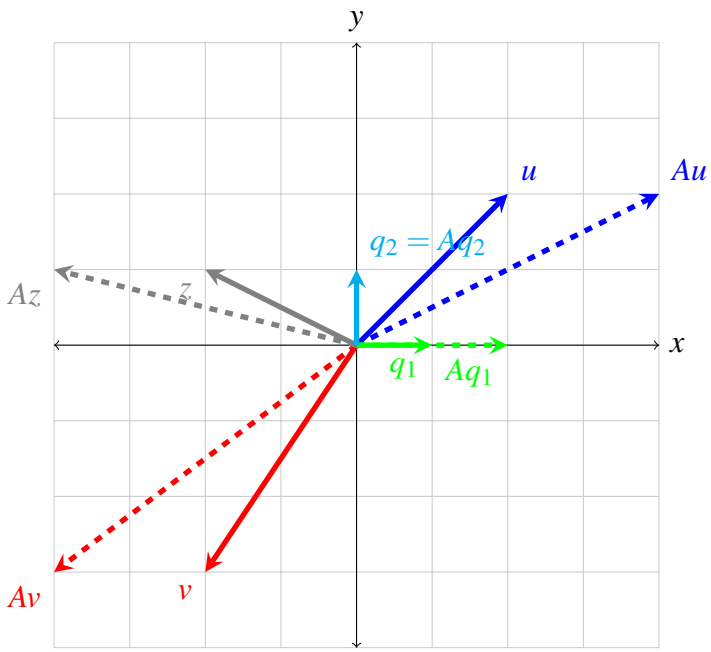
- Jakie wektory q są wektorami własnymi macierzy jednostkowej I?
  Wszystkie, z wartością własną równą $\lambda = 1$. Mnożenie dowolnego wektora przez macierz I w ogóle go nie zmienia.
- Jakie wektory q są wektorami własnymi macierzy $2\mathsf{I} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$
  Wszystkie, z wartością własną równą $\lambda = 2$. Mnożenie dowolnego wektora przez macierz $2\mathsf{I}$ wydłuża go dwukrotnie. To samo rozumowanie możemy odnieść do wszystkich wielokrotności macierzy jednostkowej.
- Jakie wektory q są wektorami własnymi macierzy obracającej o 90 stopni?
  Żaden[2], ponieważ żaden wektor (oprócz zerowego, co wykluczyliśmy) nie ma takiego samego kierunku po obróceniu o 90 stopni.
- Jakie wektory q są wektorami własnymi macierzy obracającej o 180 stopni?
  Wszystkie, z wartością własną równą $\lambda = -1$. Obracanie o 180 stopni nie zmienia kierunku, jedynie zmienia zwrot wektora na przeciwny.
- Jakie wektory są wektorami własnymi dla macierzy, która mnoży pierwszą koordynatę wektora przez 2, a drugą pozostawia bez zmian?
  Są dwa takie wektory: 1) wektor $\mathsf{q}_1 = [1,0]^T$ który po przemnożeniu przez tę macierz równa się $\mathsf{A}\mathsf{q}_1 = [2,0]^T$ czyli jest to wektor własny o wartości własnej $\lambda_1 = 2$. 2) wektor $\mathsf{q}_2 = [0,1]^T$ który po przemnożeniu przez tę macierz równa się $\mathsf{A}\mathsf{q}_2 = [0,1]^T$ czyli jest to wektor własny o wartości własnej $\lambda_2 = 1$. Przy okazji, taka macierz to oczywiście $\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$ i ten przykład został zwizualizowany na rysunku 5.3.
- O ile nieosobliwa (odwracalna) macierz A ma wektor własny q z wartością własną λ, co mogę powiedzieć o jej macierzy odwrotnej A^{-1}?
  Macierz ta z pewnością ma również wektor własny q z wartością własną $\frac{1}{\lambda}$. Skoro pomnożenie przez macierz A spowodowało wydłużenie wektora o λ to odwrócenie tej operacji to po prostu podzielenie przez λ! Formalnie:

  $$\mathsf{A}\mathsf{q} = \lambda\mathsf{q} \quad \Rightarrow \quad \mathsf{A}^{-1}\mathsf{A}\mathsf{q} = \lambda\mathsf{A}^{-1}\mathsf{q} \quad \Rightarrow \quad \mathsf{q} = \lambda\mathsf{A}^{-1}\mathsf{q} \quad \Rightarrow \quad \frac{1}{\lambda}\mathsf{q} = \mathsf{A}^{-1}\mathsf{q}$$

  co jest dokładnie równaniem własności wektorów własnych (czytanej od prawej do lewej)!
- Jeśli istnieje (niezerowy) wektor własny macierzy q, którego własność własna wynosi $\lambda = 0$ – co to nam mówi o tej macierzy?
  Macierz ta jest osobliwa (nieodwracalna). Dlaczego? Jeśli macierz jest odwracalna to mogę zrobić

  $$\mathsf{A}\mathsf{q} = \lambda\mathsf{q} = \vec{0} \qquad \Rightarrow \qquad \mathsf{A}^{-1}\mathsf{A}\mathsf{q} = \mathsf{A}^{-1}\vec{0} \qquad \Rightarrow \qquad \mathsf{q} = \vec{0}$$

[2] wykluczając wektory z wartościami zespolonymi



Rysunek 5.3: Kilka wektorów oraz wyniki ich mnożenia z macierzą A [2 0; 0 1] oraz jej wektory własne.

co nie jest prawdą, bo wektor nie jest zerowy!

Wektory własne dla niektórych macierzy istnieją, dla innych nie, a dla innych wartości własne mogą być zespolone. Na szczęście, na tym przedmiocie będziemy działali tylko na macierzach symetrycznych o których wiemy, że:

- wszystkie wartości własne są rzeczywiste
- wektorów własnych jest dokładnie tyle ile jest wymiarów macierzy
- wektory własne są wzajemnie prostopadłe

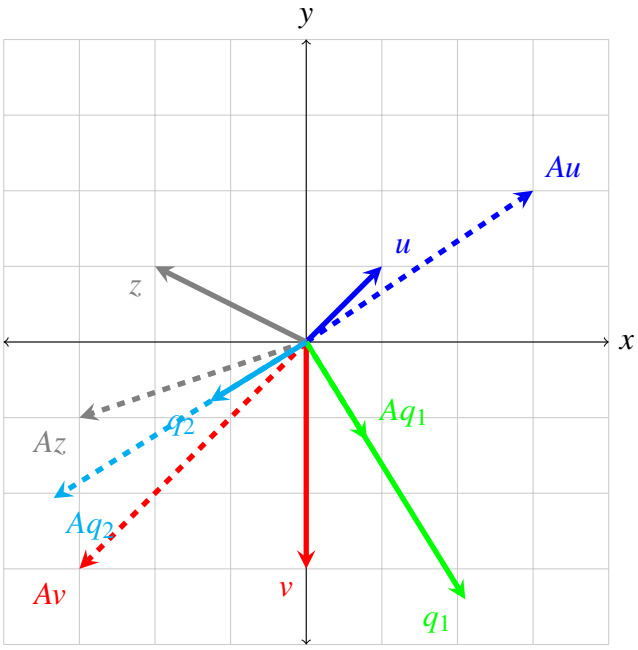
Przykłady wektorów własnych macierzy symetrycznych są narysowane na rysunkach 5.3 i 5.4 – zwróć uwagę, że są one prostopadłe do siebie (i jest ich dwa).

Wróćmy zatem do formy kwadratowej $x^T Ax$. Jak ona się zachowa w kontakcie z wektorem własnym macierzy A?

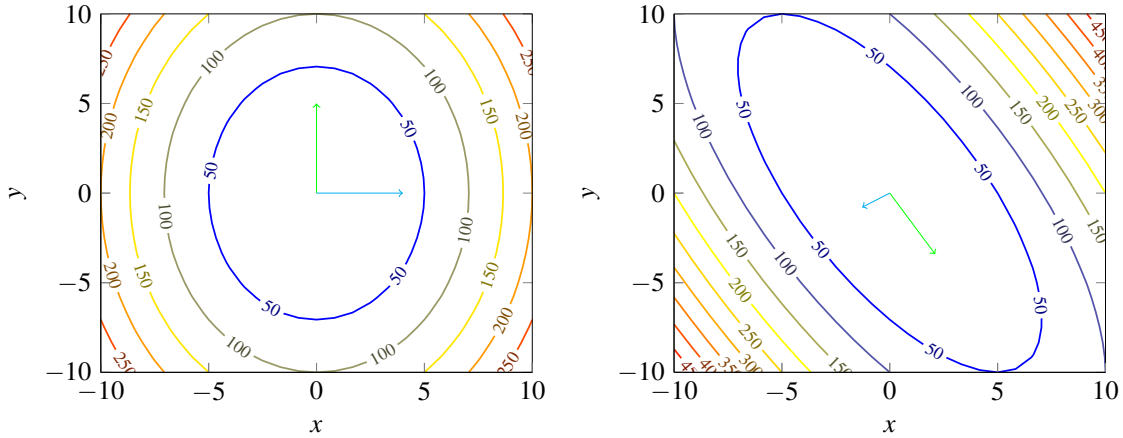
$$\begin{aligned} q^T Aq &= q^T(Aq) && \text{/korzystając z definicji wektora własnego/} \\ &= q^T(\lambda q) && \text{/lambda jest stałą/} \\ &= \lambda q^T q \end{aligned} \tag{5.1}$$

gdzie $q^T q$ to kwadrat długości wektora q. Oznacza to, że wykres formy kwadratowej w kierunku q to zwykła, standardowa parabola pomnożona przez stałą λ. Co więcej, okazuje się, że kierunki wskazywane na wykresie przez wektory własne są osiami głównymi elips/warstwic funkcji na wykresie konturowym (patrz rys. 5.5).

To proste wyprowadzenie daje nam sporo do myślenia w kontekście funkcji definiowanej przez formę kwadratową. Nasza funkcja w kierunku danego wektora własnego q_i wygląda jak zwykła parabola pomnożona przez wartość własną λ_i. Co więcej, takich wektorów jest tyle ile jest wymiarów i są one prostopadłe więc pomyślenie sobie o wykresach funkcji w tych kierunkach pozwala na wyrobienie sobie pewnych intuicji co do wyglądu całej funkcji.



Rysunek 5.4: Kilka wektorów oraz wyniki ich mnożenia z macierzą A [2 1; 1 1] oraz jej wektory własne.



Rysunek 5.5: Wykres konturowy dla form kwadratowych z macierzami [2 0; 0 1] i [2 1; 1 1]. Porównaj zaznaczone wektory oznaczające kierunki główne elips z wektorami własnymi na poprzednich rysunkach (to te same kierunki!).

Jeśli we wszystkich kierunkach parabole są skierowane w górę – inaczej mówiąc wszystkie $\lambda_i > 0$ – to funkcja jest funkcją wypukłą. Analogicznie, jeśli wszystkie parabole na przecięciach są skierowane w dół (wszystkie $\lambda_i < 0$) to funkcja zdefiniowana przez formę kwadratową jest wklęsła. Jeśli jedna z wartości własnych jest równa zero, ale wszystkie pozostałe $\lambda_i > 0$ to we wszystkich kierunkach widzimy parabole w górę, a w jednym kierunku jest zupełnie płasko. Oznacza to, że funkcja jest wypukła, ale nie w sposób ścisły – w danym kierunku funkcja ma nieskończoność punktów, które mają wartość funkcji wynoszącą 0. Ponieważ w innych kierunkach parabole są skierowane w górę, zera te są minimami lokalnymi. Najgorsza sytuacja ma miejsce gdy na niektórych przecięciach parabole są skierowane w górę (niektóre $\lambda_i > 0$), a niektóre w dół $\lambda_i < 0$ – mamy do czynienia z punktem siodłowym.

Macierze, których formy kwadratowe definiują funkcje *ściśle* wypukłe nazywamy macierzami dodatnio określonymi, ponieważ mają wszystkie wartości własne dodatnie ($\lambda > 0$). Z kolei macierze, których formy kwadratowe definiują funkcje wypukłe nazywamy półdodatnio określonymi ($\lambda \geq 0$).

5.4 Analiza działania metody najszybszego spadku

Wykorzystując wartości i wektory własne pokażemy prostą właściwość działania algorytmu najszybszego spadku, kiedy minimalizuje on ściśle wypukłą funkcję kwadratową.

$$f(\mathsf{x}) = \frac{1}{2}\mathsf{x}^\mathsf{T}\mathsf{A}\mathsf{x} - \mathsf{b}^\mathsf{T}\mathsf{x} + c$$

$$\nabla f(\mathsf{x}) = \mathsf{A}\mathsf{x} - \mathsf{b} \quad \Rightarrow \quad \mathsf{x}^* = \mathsf{A}^{-1}\mathsf{b}$$

Wyniki te, jak się okaże podczas kolejnych laboratoriów, można jednak odnieść do szerszej gamy funkcji, które są ściśle wypukłe.

Ponieważ chcemy analizować pozycje rozważanego przez algorytm w k-tej iteracji punktu x_k w stosunku do pozycji minimum x^*, zdefiniujmy wektor błędu

$$\mathsf{e}_k = \mathsf{x}_k - \mathsf{x}^*$$

który jest różnicą pomiędzy optimum a naszą obecną pozycją. Nota bene: gdybyśmy znali wektor e_0 wystarczyłoby go dodać do naszej pozycji startowej i od razu wylądować w minimum. Możemy też wyznaczyć jak zmienia się błąd w kolejnych iteracjach algorytmu:

$$\begin{aligned}
\mathsf{e}_{k+1} &= \mathsf{x}_{k+1} - \mathsf{x}^* \quad \text{/ze wzoru na iteracje algorytmu/} \\
&= \mathsf{x}_k - \eta_k \nabla f(\mathsf{x}_k) - \mathsf{x}^* \quad \text{/szeregując elementy/} \\
&= \mathsf{x}_k - \mathsf{x}^* - \eta_k \nabla f(\mathsf{x}_k) = \mathsf{e}_k - \eta_k \nabla f(\mathsf{x}_k)
\end{aligned} \tag{5.2}$$

W kontekście wektora błędu możemy także zdefiniować gradient funkcji kwadratowej.

$$\begin{aligned}
\nabla f(\mathsf{x}_k) &= \mathsf{A}\mathsf{x}_k - b \quad \text{/pomnóżmy b przez A i jej odwrotność (czyli przez 1)/} \\
&= \mathsf{A}\mathsf{x}_k - \mathsf{A}\underbrace{\mathsf{A}^{-1}b}_{x^*} = \mathsf{A}\mathsf{x}_k - \mathsf{A}\mathsf{x}^* = \mathsf{A}(\mathsf{x}_k - x^*) = \mathsf{A}\mathsf{e}_\mathsf{k}
\end{aligned} \tag{5.3}$$

Zacznijmy od czegoś prostego: jeżeli po k-tej iteracji okaże się, że wektor błędu e_k ma kierunek zgodny z wektorem własnym – jak zareaguje na to nasz algorytm?

$$\begin{aligned}\mathsf{e}_{k+1} &= \mathsf{e}_k - \eta_k \nabla f(\mathsf{x}_k) \quad \text{/ze wzoru na gradient/}\\ &= \mathsf{e}_k - \eta_k \mathsf{A}\mathsf{e}_\mathsf{k} \quad \text{/skoro błąd jest wektorem własnym/}\\ &= \mathsf{e}_k - \eta_k \lambda \mathsf{e}_\mathsf{k} = (1-\eta_k\lambda)\mathsf{e}_\mathsf{k}\end{aligned}$$

Pamiętając, że algorytm wybiera najlepsze η_k jest oczywiste, że wybierze $\eta_k = \frac{1}{\lambda}$ otrzymując:

$$\mathsf{e}_{k+1} = (1-\eta_k\lambda)\mathsf{e}_\mathsf{k} = (1-\frac{1}{\lambda}\lambda)\mathsf{e}_\mathsf{k} = 0$$

Wektor błędu wynosi zero, a więc znaleźliśmy się w optimum! Jeśli więc, w takcie działania algorytmu wektor błędu stanie się (przez przypadek, bo tego nie kontrolujemy) wektorem własnym macierzy to będzie to ostatnia iteracja algorytmu najszybszego spadku.

Dodatkowo, jak wynika z naszej dyskusji, istnieją macierze dla których *każdy* wektor jest wektorem własnym. Taką macierzą była np. macierz jednostkowa, która definiuje funkcje kwadratową $\frac{1}{2}\mathsf{x}^\mathsf{T}\mathsf{x} - \mathsf{b}^\mathsf{T}\mathsf{x} + c$. Tego typu problemy optymalizacyjne zostaną rozwiązane w jednej iteracji przez algorytmu najszybszego spadku!

Literatura

Literatura powtórkowa

Informacje o wektorach i wartościach własnych można znaleźć w każdej książce do algebry liniowej. Opis algorytmu najszybszego spadku można znaleźć w rozdziale 5 książki [**?**].

Literatura dla chętnych

W tym tygodniu zachęcamy do zapoznania się z algorytmami do automatycznego liczenia pochodnych, które są przydatne do implementacji algorytmu najszybszego spadku. Wśród takich technik popularne jest pojęcie grafu obliczeń (ang. *computational graph*) i liczenie pochodnych na takim grafie. Opis grafów obliczeń wraz z algorytmem wstecznej propagacji można znaleźć np. [**?**].

6. Stochastic Gradient Descent

Na drugich laboratoriach poznawaliśmy algorytm dychotomizacji, który wybierał dwa punkty, sprawdzał w tych punktach wartość funkcji celu, a następnie był w stanie zawęzić obszar przeszukiwań poprzez odrzucenie jednego z rogów dziedziny (pomiędzy jednym z punktów a końcem dziedziny). Z tego powodu staraliśmy się tak dobierać sprawdzane punkty, aby były one jak najbliżej środka rozważanego przedziału, aby w każdej iteracji zawęzić przedział poszukiwań o połowę.

Jednakże okazało się potem, że można to robić szybciej używając algorytmu złotego podziału, który w każdej iteracji odrzucał nie połowę zakresu, a... tylko 38%! Jak to możliwe? Bo każda iteracja algorytmu złotego podziału jest dwukrotnie szybsza niż iteracja algorytmu dychotomizacji (tylko jedna ewaluacja wartości funkcji)! W związku z tym, nawet pomimo gorszego ograniczenia zakresu przedziału poszukiwań mogliśmy wykonać dwukrotnie więcej iteracji w tym samym czasie, sprawiając że ostatecznie wychodziliśmy na plus.

Na tym laboratorium poznamy analogiczną technikę w kontekście algorytmu spadku wzdłuż gradientu. Będziemy wykonywać iteracje przesuwając nasze aktualne rozwiązanie w kierunku pewnego, szybko obliczonego, zaszumionego *przybliżenia* gradientu. Jednakże, ponieważ będziemy w stanie wykonać tych iteracji dużo, dużo więcejw tym samym czasie, ostatecznie otrzymamy wynik tej samej jakości, ale w dużo krótszym czasie. Technika ta (i jej rozszerzenia) jest zdecydowanie najpopularniejszą metodą optymalizacyjną w uczeniu maszynowym, a już szczególnie w głębokim uczeniu się sieci neuronowych.

6.1 Algorytm stochastycznego spadku wzdłuż gradientu

Co może stać za wolnym działaniem standardowego algorytmu spadku wzdłuż gradientu? Oczywiście może być to zbyt mała stała szybkości optymalizacji, no ale po ostatnich laboratoriach potrafisz ją wspaniale dobierać, więc to ci raczej nie grozi. Co innego może wolno działać w praktycznie jednej linijce kodu $\mathsf{x} \leftarrow \mathsf{x} - \eta \nabla f(\mathsf{x})$? Jednym z problemów

może być czas konieczny do obliczenia gradientu, który wpływa na czas wykonania każdej iteracji algorytmu!

Czasami jednak funkcja celu może mieć szczególną postać, która pozwala nam na rozbicie obliczenia gradientu na kilka podproblemów. Taką postacią, która wyjątkowo często pojawia się w statystyce czy uczeniu maszynowym jest funkcja celu będą sumą innych funkcji.

$$f(\mathsf{x}) = \sum_{i=1}^{N} f_i(\mathsf{x})$$

Standardowym przykładem jest tutaj np. problem regresji liniowej czy minimalizowania błędu klasyfikacji. Funkcja, którą wtedy minimalizujemy jest zwykle sumą błędów popełnianych na kolejnych przykładach uczących np. błąd kwadratowy $SSE = \sum_{i=1}^{N}(y-\hat{y})^2$. Co dzieje się wtedy z gradientem?

$$\nabla f(\mathsf{x}) = \nabla \left[\sum_{i=1}^{N} f_i(\mathsf{x}) \right] = \sum_{i=1}^{N} \nabla f_i(\mathsf{x})$$

Gradient sumy to suma gradientów, więc iteracja w algorytmie przekształca się do $\mathsf{x} \leftarrow \mathsf{x} - \eta \sum_{i=1}^{N} \nabla f_i(\mathsf{x})$. Taka postać jest dla nas niesamowicie sympatyczna, bo widać z tego wzoru, że obliczanie gradientu możemy potencjalnie zrównoleglić. Każdy wątek oblicza gradient funkcji cząstkowych, a następnie je sumujemy. Jednak nie to jest ideą algorytmu stochastycznego spadku wzdłuż gradientu. Okazuje się... że tej sumy wcale nie trzeba robić!

Jak już wcześnie wspomnieliśmy, w przypadku regresji liniowej minimalizujemy błąd kwadratowy. Konkretnie, rozważmy system dokonujący predykcji oceny filmu przez użytkowników Filmweb'a zanim jeszcze wejdzie on do kin. Takiej predykcji możemy dokonać np. na podstawie informacji o reżyserze, obsadzie, budżecie, wytwórni filmowej czy na podstawie oceny książki (jeśli film kręcony jest na jej podstawie). Z resztą takie predykcje rzeczywiście robią studia w Hollywood zanim zdecydują się na kosztowną inwestycję w nowy film. Aby wytrenować taki system zbieramy dane o wielu poprzednich filmach i na ich podstawie konstruujemy model regresji $f(\mathsf{x})$. Chcemy go stworzyć w taki sposób, aby popełniany przez niego błąd był jak najmniejsze – jest to więc oczywiste zastosowanie metod optymalizacji. Jak taki błąd liczymy? System wykonuje predykcję $\hat{y}_i = f(\mathsf{x}_i)$, porównujemy ją z prawdziwą oceną filmu na portalu y_i, a następnie obliczamy podniesioną do kwadratu różnicę $(\hat{y}_i - y_i)^2$. Taki sposób powtarzamy dla wszystkich filmów otrzymując całkowity błąd (używamy litery L od angielskiego słówka *loss*):

$$L = \sum_{i=1}^{N} (y_i - f(\mathsf{x}_i))^2$$

Tylko... co tutaj optymalizować? Prawdziwą odpowiedź na portalu y_i (może przekupimy jakichś użytkowników ;P) czy może x_i czyli informacje o reżyserze itd.? Nic z tych rzeczy: nasz model regresji $f(\mathsf{x}_i)$ ma pewne parametry θ od których zależy jego działanie i to właśnie te parametry będziemy optymalizować. Takimi parametrami w regresji liniowej były wagi poszczególnych zmiennych wyjaśniających, które tutaj są zapakowane do jednego dużego wektora θ.

$$\min_{\theta} \quad L(\theta) = \sum_{i=1}^{N} (y_i - f(\mathsf{x}_i; \theta))^2$$

No właśnie, skoro o statystyce mowa... tak naprawdę to bylibyśmy zainteresowani optymalizacją nie tylko na tych filmach o których mamy dane (w końcu predykcję będziemy wykonywać dla zupełnie nowych filmów, których nie ma przecież w naszym zbiorze danych!), ale na *całej populacji* filmów. Czyli nasz problem wygląda w następujący sposób (zwróć wagę, że jeżeli podzielisz wcześniejszy problem przez $\frac{1}{N}$ otrzymasz średnią arytmetyczną, jednocześnie nie zmieniając problemu optymalizacyjnego):

$$\min_{\theta} \quad L(\theta) = \mathbb{E}_{\mathsf{x},y\sim p_{dane}}\left[(y - f(\mathsf{x};\theta))^2\right]$$

Optymalizowanie funkcji w których występują zmienne losowe nazywamy optymalizacją stochastyczną. Zwróć uwagę, że wartość oczekiwana jest sumą ważoną[1], więc

$$\nabla_\theta L(\theta) = \mathbb{E}_{\mathsf{x},y\sim p_{dane}}\left[\nabla_\theta (y - f(\mathsf{x};\theta))^2\right] \tag{6.1}$$

Powyższe rozumowanie można łatwo dostosować do wielu innych problemów np. do problemu klasyfikacji (czy na obrazku jest kot?) itd. W szczególności w uczeniu maszynowym czy w statystyce często używamy metody największej wiarygodności (ang. *maximum likelihood estimation*), która sprowadza się maksymalizacji wartości oczekiwanej po empirycznym rozkładzie prawdopodobieństwa (czyli po rozkładzie naszych konkretnych danych), mocno trzymając kciuki że jest ona tożsama z maksymalizacją takie wartości po prawdziwym rozkładzie populacji.

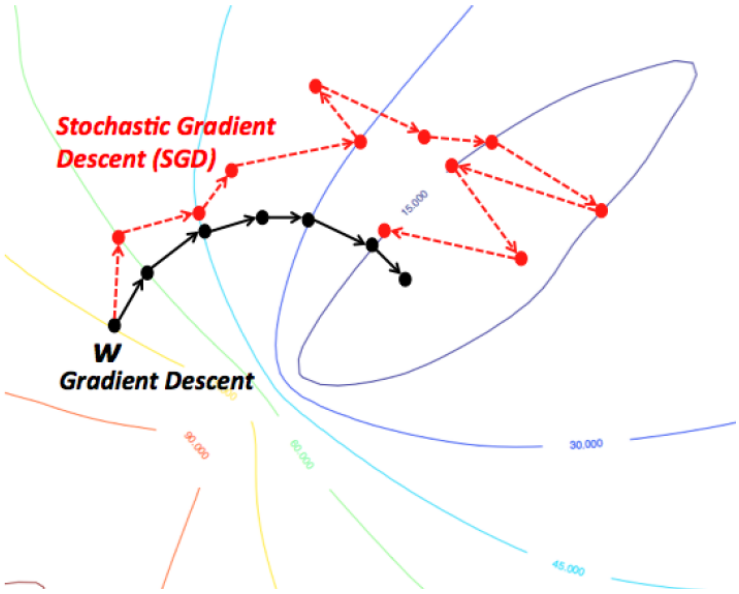
$$\min_{\theta} \quad L(\theta) = \mathbb{E}_{\mathsf{x},y\sim \hat{p}_{dane}}\left[\log P(\mathsf{x},y;\theta)\right]$$

Zauważ ewidentne podobieństwo ze wcześniej analizowanym przez nas problemem regresji.

No ok, tak wyglądające problemy są ważne, pojawiają się często w uczeniu maszynowym, ale gdzie tu algorytm? Pomysł jest bardzo prosty: po kursie ze statystyki jesteś (a przynajmniej powinieneś być) mistrzem w estymowaniu wartości oczekiwanej i wiedzieć, że dobrym estymatorem jest tutaj średnia arytmetyczna. Jak widzisz gradient naszej funkcji (patrz wzór 6.1) jest tak naprawdę wartością oczekiwaną, więc można go wyestymować! Nasz algorytm nie musi się więc poruszać w kierunku prawdziwego, dokładnie policzonego gradientu, ale w kierunku jego wyestymowanej/przybliżonej wartości. Oczywiście taki algorytm nie będzie zachowywał się dokładnie tak samo jak algorytm spadku wzdłuż gradientu, ale jeśli nasze estymacje będą dostatecznie dobre to powinno być to całkowicie wystarczające do efektywnej optymalizacji (patrz Rys. 6.1).

Jest to wspaniała wiadomość, bo nie musisz liczyć całego gradientu, żeby dostać dokładną jego wartość, a jedynie jego przybliżenie. Pewną (szczególnie biedną) formą estymacji wartości oczekiwanej jest średnia z... jednej obserwacji (czyli po prostu ta jedna wartość). W klasycznym algorytmie obliczałbyś błąd predykcji dla wszystkich obserwacji (np. masz ich milion – czyli dużo czasu!), następnie liczyłbyś z tego gradient i wykonywał jeden krok optymalizacji. We właśnie stworzonym algorytmie: bierzesz pojedynczą, losową obserwację z twojego zbioru danych, liczysz dla niej błąd, liczysz gradient (dla niej jednej) i... wykonujesz krok optymalizacji. Wykonujesz krok w kierunku przybliżonym do dobrego (zaszumionym), ale możesz ich wykonać milion w tym samym czasie co jeden krok algorytmu spadku gradientu!

[1] $\mathbb{E}X = \sum_{x\in X} P(x = x)x$



Rysunek 6.1: Standardowy algorytm spadku wzdłuż gradientu (ang. *gradient descent*) zmierza do minimum zgodnie z kierunkiem gradientu. Algorytm stochastyczny używający przybliżonych kierunków gradientu potrzebuje więcej kroków aby dojść w okolice minimum, ale robi je szybciej. [2]

Definicja 6.1 — Algorytm stochastycznego spadku wzdłuż gradientu. Algorytm stochastycznego spadku wzdłuż gradientu (ang. *stochastic gradient descent, SGD*)

$\theta \leftarrow$ INICJALIZUJ
while warunek stopu nie jest spełniony **do**
  Wylosuj obserwację ze zbioru danych x_i, y_i
  Oblicz błąd dla wylosowanej obserwacji $l_i(\mathsf{x}_i, y_i; \theta)$ oraz jego gradient $\nabla_\theta l_i(\mathsf{x}_i, y_i; \theta)$
  $\theta \leftarrow \theta - \eta \nabla_\theta l_i(\mathsf{x}_i, y_i; \theta)$ ▷ Przesuń θ w kierunku minimum
end while

Hmmm... ale czy to zadziała? Okazuje się, że tak oraz że ten algorytm ma bardzo zbliżone gwarancje teoretyczne do algorytm spadku wzdłuż gradientu.

Twierdzenie 6.1 Niech $f(x)$ będzie funkcją wypukłą ze stałą Lipschitza równą L, poprzez x^* oznaczymy jej minimum zakładając, że $||x^*|| \leq B$. Po T iteracjach algorytmu stochastycznego spadku wzdłuż gradientu (z odpowiednią η) otrzymujemy:

$$\mathbb{E}[f(\bar{x})] - f(x^*) \leq \frac{LB}{\sqrt{T}}$$

6.1.1 Przykład: regresja liniowa

Kontynuując nasz przykład z laboratorium 3, zobaczmy jak wygląda algorytm spadku wzdłuż gradientu w wersji zwykłej i stochastycznej. W regresji liniowej minimalizujemy

$$\min_\theta \quad L(\theta) = \sum_{i=1}^{N} (y_i - \mathsf{x}_i^T \theta)^2$$

Obliczając gradient otrzymujemy:

$$\nabla_\theta L(\theta) = \sum_{i=1}^{N} 2(y_i - \mathsf{x}_i^T \theta)\mathsf{x}_i$$

Jeden krok algorytmu spadku wzdłuż gradientu wygląd więc następująco:

$$\theta \leftarrow \theta - 2\eta \sum_{i=1}^{N} (y_i - \mathsf{x}_i^T \theta)\mathsf{x}_i$$

i wymaga przeiterowania po całym zbiorze danych. Z kolei jeden krok algorytmu stochastycznego (dla losowej obserwacji) wygląda następująco:

$$\theta \leftarrow \theta - 2\eta (y_i - \mathsf{x}_i^T \theta)\mathsf{x}_i$$

Zwróć uwagę na ciekawą interpretację tego równania: im większy błąd predykcji tym wykonujesz większy krok by go wyeliminować.

6.1.2 Rozważania praktyczne

Algorytm stochastycznego spadku wzdłuż gradientu ma wiele zalet. Po pierwsze wzór na gradient który liczymy tylko na jednym elemencie ma zwykle bardzo prosty wzór. Dodatkowo, policzenie gradientu błędu, który jest sumą błędów miliona przykładów uczących wymaga od nas przejrzenia tych wszystkich przykładów (wolno!). Tutaj by policzyć gradient potrzebujemy przejrzeć potencjalnie tylko jedną obserwację. Dodatkowo, jak pokażemy w rozdziale 6.3.3 algorytm ten łatwo się rozprasza na wiele maszyn. Zwróć też uwagę, że algorytm ten w naturalny sposób możemy zaaplikować kiedy nie mamy całego zbioru danych zapisanego na dysku, a dane przychodzą do nas w sposób przyrostowy (strumień danych). Wtedy to nie my, a „świat” losuje nam przykłady uczące które wysyła do naszego algorytmu – wszystko inne pozostaje bez zmian. Z powyższych powodów jest to najczęściej stosowany algorytm przy uczeniu sieci neuronowych i nie tylko.

Wybór szybkości optymalizacji

Pewną wadą algorytmu jest za to konieczność wyboru parametru η (szybkość optymalizacji). Zwykle ustalamy go na pewną stałą wartość wybieraną metodą prób i błędów. Z punktu widzenia teoretycznego, aby osiągnąć zbieżność powinniśmy jednak obniżać η z czasem np. $\eta = \frac{\eta_0}{\sqrt{t}}$ gdzie t to numer iteracji (czas). Intuicyjnie: nasze gradienty są zaszumione i do pewnego stopnia losowe, jeśli więc nie zmniejszymy η to nasze rozwiązanie będzie losowo „drgało” w okolicach optimum funkcji. Dopiero zmniejszanie η z czasem spowoduje, że te drgania będą coraz to mniejsze, aż w końcu utkniemy w optimum. W praktyce zbyt szybkie zmniejszanie η może spowodować bardzo wolne zbieganie do minimum. Z kolei zbyt wolne zmniejszanie szybkości optymalizacji spowoduje duże chaotyczne skakanie na początku optymalizacji i tracenie czasu bez rzeczywistej optymalizacji . Dlatego czasami korzysta się ze wzoru:

$$\eta_k = \left(1 - \frac{k}{\tau}\right)\eta_0 + \frac{k}{\tau}\eta_\tau$$

gdzie k jest numerem iteracji, a τ jest pewnym horyzontem czasowym. Powyższy wzór jest kombinacją wypukłą dwóch liczb (parametrów) η_0 i η_τ czyli geometrycznie możemy

go wizualizować jako odcinek. Ponieważ waga w tej kombinacji zależy od numeru iteracji k, łatwo zauważyć, że pierwsza iteracja będzie miała szybkość iteracji η_0 i będzie ona spadała aż do η_τ w sposób liniowy z czasem. Po osiągnięciu horyzontu czasowego $k = \tau$ pozostawiamy η_k stałe (w przeciwnym wypadku po wielu iteracjach szybkość optymalizacji stałaby się ujemna).

W książce [4] proponowany jest następujący wybór parametrów. Wartość końcową η_τ należy ustawić na około 1% wartości początkowej η_0, a horyzont czasowy τ dobrać tak, aby wystarczał on na wykonanie kilkuset iteracji po zbiorze danych. Pozostaje więc wybranie parametru η_0 – jeżeli więc korzystamy z powyższych heurystyk nadal całe nasze zadanie ograniczania się do podania tylko jednego parametru. Początkową wartość optymalizacji należy w zasadzie wybrać tak jak to robiliśmy przy metodzie spadku gradientu. Pamiętaj jednak, że przebieg optymalizowanej funkcji celu, nawet przy dobrze dobranej prędkości optymalizacji nie będzie gładki z powodu zaszumionych kierunków gradientów (patrz Rys. 6.2). Małe wahania są więc czymś naturalnym. Ponieważ nasz wzór będzie liniowo obniżał prędkość optymalizacji, należy wybrać wartość trochę wyższą od najlepiej się zachowującej, ale jednocześnie nie powodującą rozbiegania się algorytmu.

Istnieje oczywiście wiele innych sposobów na zmniejszanie szybkości optymalizacji z czasem. Jak chociażby technika *step decay* polegająca na dzieleniu prędkości optymalizacji co ileś iteracji np. co 50 iteracji zmniejsz prędkość o połowę $\eta \leftarrow \frac{1}{2}\eta$. Są to tylko heurystyki i czuj się całkowicie swobodny by zaprojektować swoją własną, najbardziej ci odpowiadającą.

Czy rzeczywiście trzeba losować dane?

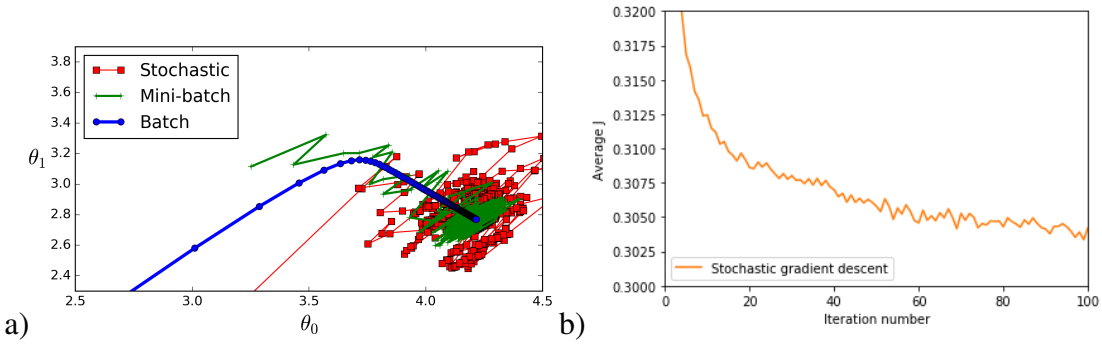
Chociaż nasze rozważania teoretyczne sugerowały, że trzeba losować kolejne obserwacje, w praktyce przed rozpoczęciem optymalizacji losowo sortujemy zbiór danych, a dalej traktujemy je sekwencyjnie.

Technika mini-batch

W sekcjach powyżej przyznaliśmy, że robienie średniej z jednej obserwacji jest dość słabą formą estymacji wartości oczekiwanej. Nadal jednak w pełni działającą i wystarczającą do prawidłowego działania algorytmu. W praktyce jednak wiemy, że jeżeli pochodna z losowej obserwacji ma wariancję σ to średnia z jednej pochodnej ma też taką wariancję, przez co nasz algorytm jak widać na rysunku 6.1 dość mocno skacze po przestrzeni optymalizowanych parametrów. Wiemy jednak, że jeśli wyciągnęlibyśmy średnią z większej liczby elementów np. $n = 100$ to wariancja estymowanej pierwszej pochodnej spadnie dziesięciokrotnie ($\mathbb{D}^2[\bar{X}] = \frac{\sigma}{\sqrt{n}}$) i uzyskamy mniej zaszumiony kierunek optymalizacji. Z tego powodu w praktyce łączymy obserwacje w małe paczki (ang. *mini-batch*), liczymy z nich wszystkich gradient i uśredniamy go, aby otrzymać lepszą estymację gradientu. Następnie wykonujemy krok optymalizacji i zaczynamy przetwarzać kolejną paczkę obserwacji.

Definicja 6.2 — Algorytm stochastycznego spadku wzdłuż gradientu w wersji mini-batch. Algorytm stochastycznego spadku wzdłuż gradientu (ang. *stochastic gradient descent, SGD*) w wersji *mini-batch*.

$\theta \leftarrow$ INICJALIZUJ
while warunek stopu nie jest spełniony **do**
  Wylosuj n obserwacji ze zbioru danych $(\mathsf{x}_1, y_1), (\mathsf{x}_2, y_2), \ldots, (\mathsf{x}_n, y_n)$ ▷ Stwórz



Rysunek 6.2: a) Porównanie standardowego algorytm spadku wzdłuż gradientu (ang. *gradient descent*) z wersją mini-batch i „całkowicie” stochastyczną. Zwróć uwagę, że wachania wersji mini-batch są mniejsze niż stochastycznej. b) Nawet przy dobrze dobranej szybkości optymalizacji algorytmu SGD, nie w każdej iteracji następuje poprawa wypukłej funkcji celu, a wykres nie jest gładki.

paczkę danych
Oblicz błąd dla wylosowanych obserwacji $l_i(\mathrm{x}_i, y_i; \theta)$ oraz ich gradienty $l_i(\mathrm{x}_i, y_i; \theta)$
Oblicz średni gradient $\bar{\mathrm{g}} = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta l_i(\mathrm{x}_i, y_i; \theta)$ ▷ Estymacja wartości oczekiwanej gradientu
$\theta \leftarrow \theta - \eta\bar{\mathrm{g}}$ ▷ Przesuń θ w kierunku minimum
end while

Oczywiście nie można przesadzać z rozmiarem paczki, ponieważ gdy ustawimy n równe liczbie elementów w naszym zbiorze danych dostaniemy dokładny gradient, ale jednocześnie nasz algorytm stanie się zwykłym algorytmem spadku wzdłuż gradientu. Jednocześnie tracąc zalety algorytmu SGD. Jaki jest więc optymalny rozmiar paczki? Jest to trudne pytanie, ponieważ większy rozmiar paczki oznacza dokładniejszą estymację gradientu, pewniejsze kroki algorytmu optymalizacyjnego, ale i wolniejsze jego działanie. Z kolei mały rozmiar paczki może spowodować mocno zaszumione kroki algorytmu optymalizacyjnego, a więc szybsze ich wykonywanie nie musi się nam ostatecznie opłacać. Trzeba więc wybrać jakąś rozsądną wartość po środku.

Warto zauważyć, że w praktyce liczenie gradientu z kilku obserwacji na raz może być znacząco szybsze niż liczenie ich pojedynczo. Dlaczego? Ponieważ licząc kilka gradientów na raz możemy w pełni wykorzystać możliwości standardowych architektów wielordzeniowych procesora i zastosowanie obliczeń równoległych. W szczególności większość algorytmów uczenia maszynowego jak np. sieci neuronowe można wyrazić przy pomocy rachunku macierzowego jako mnożenie (kilku) macierzy. Operacja ta doskonale się zrównolegla, a jeśli dostosujemy wielkość paczki danych do wielkości pamięci cache procesora to praktycznie możemy nie poczuć różnicy pomiędzy obliczeniem jednego gradientu a kilkunastu na raz. Gdyby proces optymalizacji działał na karcie graficznej to tutaj w sposób szczególny dopasowanie wielkości paczki do rozmiaru pamięci w karcie może grać kluczową rolę dla wykonania efektywnie czasowo optymalizacji.

! Ponieważ przy odpowiednim ustawieniu wielkości paczki algorytm SGD staje się zwykłym algorytmem GD często stosuje się następujące nazewnictwo: *stochastic GD* gdzie rozmiar paczki jest równy $n = 1$, *batch GD* jako standardowy algorytm spadku gradientu $n = N$ oraz *mini-batch GD* gdzie $1 < n < N$.

Oprócz techniki mini-batch istnieją także inne specjalizowane metody aby zmniejszać wariancję w algorytmie SGD, które nazywamy (zaskoczenie) metodami redukcji wariancji. Przykładem takiej metody jest np. algorytm SAGA [3] który dodatkowo przechowuje tabelę gradientów dla każdego elementu zbioru danych. Podczas wykonywania aktualizacji bieżącego rozwiązania bierze się pod uwagę nie tylko gradient dla danej obserwacji, ale także buforowane w pamięci historyczne gradienty dla całego zbioru danych. Techniki te potrafią mieć lepsze właściwości teoretyczne (szybsza zbieżność) niż algorytm SGD dla funkcji ściśle wypukłych, a ich udoskonalanie jest nadal aktywnym przedmiotem badań naukowych.

6.2 Rozważania praktyczne: sprawdzanie implementacji gradientu

Kiedy implementujesz gradient dla złożonej funkcji celu ryzyko że popełniłeś błąd w obliczeniach lub po prostu podczas implementacji jest całkiem spore. Z tego powodu zawsze powinieneś przetestować swój kod - tak jakbyś to zrobił implementując jakikolwiek inny program. Fakt, że funkcja celu maleje z kolejnymi iteracjami nie jest *żadną* gwarancją, że gradient został zaimplementowany poprawnie! Jak więc to przetestować?

Jeżeli jesteś na etapie implementacji gradientu to praktycznie na pewno masz (powinieneś mieć) zaimplementowaną funkcję obliczającą wartość funkcji celu. Możesz więc porównać wartość twojego gradientu z jego numerycznym przybliżeniem, wykorzystując wzór poznany na trzecich laboratoriach.

Choć oczywistym rozwiązaniem jest po prostu sprawdzenie różnicy pomiędzy wartością przybliżenia gradientu $\tilde{f}'(x)$ a wartością (potencjalnie) dokładną obliczoną przez twój kod $f'(x)$ to nie jest to rozwiązanie najlepsze. Jeżeli gradient w punkcie jest bardzo mały to nawet bardzo małe różnice powinny być niepokojące, z kolei jeśli wartość gradientu jest duża to również też twoja tolerancja na odchylenia spowodowane przybliżeniem numerycznym powinna być większa. Z tego powodu zwykle obliczamy błąd względny:

$$error = \frac{|f'(x) - \tilde{f}'(x)|}{\max\{|f'(x)|, |\tilde{f}'(x)|\}}$$

Błąd względny wyższy niż 1% prawie na pewno oznacza błędną implementację gradientu, przy prawidłowej implementacji gradientu powinieneś uzyskiwać błąd znacznie niższy.

6.2.1 Przybliżanie gradientu numerycznie

W przypadku pochodnej byliśmy w stanie ją przybliżyć używając wzoru

$$f'(x) \approx \frac{f(x+\varepsilon) - f(x)}{\varepsilon} \tag{6.2}$$

lub trochę lepszego w praktyce

$$f'(x) \approx \frac{f(x+\varepsilon) - f(x-\varepsilon)}{2\varepsilon} \tag{6.3}$$

Wzory te pozwalają nam przybliżenie wartości pochodnej bez jej bezpośredniego liczenia, a jedynie używając wywołania funkcji $f(x)$.

W przypadku gradientu w ten sposób musimy przybliżać każdy jego element, dodając/odejmując ε od wymiaru z którego liczmy pochodną cząstkową. Tak jak w przypadku

funkcji jednowymiarowej potrzebowaliśmy dwóch wywołań funkcji $f(x)$, aby przybliżyć pochodną, w gradiencie mamy d elementów i każdy z nich musi być przybliżony. Wymaga to więc od nas $2d$ ewaluacji funkcji, gdzie d potencjalnie może być dużą liczbą. Dodatkowym problemem był także wybór ε, który powinien być małą stałą np. $\varepsilon = 10^{-5}$ – jednak nie tak małą by dzielenie przez nią zaczęło stwarzać problemy numeryczne.

■ **Przykład 6.1 — Problemy z przybliżaniem pochodnej (za duże ε).** Rozważmy jako przykład gradient w punkcie $(0,0)$ następującej funkcji dwóch zmiennych:

$$f(x,y) = x + y - 100(x^2 + y^2)$$

Obliczmy gradient:

$$\nabla f(x,y) = \begin{bmatrix} \frac{\partial f(x,y)}{\partial x} & \frac{\partial f(x,y)}{\partial y} \end{bmatrix}^T = \begin{bmatrix} 1 - 200x & 1 - 200y \end{bmatrix}^T$$

a więc dla punktu $(0,0)$ gradient wynosi $\nabla f(0,0) = (1,1)$. Załóżmy jednak, że postać badanej funkcji nie jest nam znana i używamy numerycznego przybliżenia poszczególnych elementów gradientu poprzez wzór

$$\frac{\partial f(x,y)}{\partial x} \approx \frac{f(x+\varepsilon,y) - f(x,y)}{\varepsilon}$$

gdzie ε jest pewną wybraną przez nas stałą[2]?

$$\begin{aligned}
\frac{\partial f(x,y)}{\partial x} \approx \frac{f(x+\varepsilon,y) - f(x,y)}{\varepsilon} &= \frac{\left[(x+\varepsilon) + y - 100\left((x+\varepsilon)^2 + y^2\right)\right] - \left[x + y - 100(x^2+y^2)\right]}{\varepsilon} \\
&= \frac{\left[\cancel{x} + \varepsilon + \cancel{y} - \cancel{100x^2} - 200x\varepsilon - 100\varepsilon^2 - \cancel{100y^2}\right] - \cancel{x} - \cancel{y} + \cancel{100x^2} + \cancel{100y^2}}{\varepsilon} \\
&= 1 - 200x - 100\varepsilon
\end{aligned}$$

Z powyższych obliczeń wynika, że dla punktu (0,0) otrzymamy przybliżenie gradientu wynoszące $[1 - 100\varepsilon, 1 - 100\varepsilon]^T$. Fakt, że nie otrzymujemy idealnie spodziewanego gradientu nie jest niczym niezwykłym – jest to przecież pewne przybliżenie. To co jednak może nas zaskoczyć to fakt, że przy nie dość małym ε kierunek przybliżonego gradientu będzie miał kierunek *przeciwny* do prawdziwego. Prawdziwy gradient jest dla obu kierunków dodatni, a aby nasz przybliżony gradient też był dodatni potrzeba by

$$1 - 100\varepsilon > 0 \qquad \Rightarrow \qquad -100\varepsilon > -1 \qquad \Rightarrow \qquad \varepsilon < 0.01$$

Jeżeli więc wybrałeś nie dość małe ε to w tej sytuacji otrzymasz gradient o przeciwnym kierunku! Co potencjalnie pokieruje Twój algorytm optymalizacyjny w kierunku przeciwnym do pożądanego! To oczywiście specyficzny przypadek, zwróć jednak uwagę, że funkcja ma prostą postać oraz posiada tylko jedno optimum lokalne (funkcja wklęsła).

! Powyższe rozważania można uogólnić do dowolnych funkcji $f(x,y) = x + y - a(x^2 + y^2)$ gdzie $a > 0$. W takim wypadku ε musi być wtedy mniejszy niż $\frac{1}{a}$, aby nie otrzymać gradientu o przeciwnym kierunku.

[2] Prezentujemy wzory i wyprowadzenie dla zmiennej x, wyprowadzenia i wzory dla y są analogiczne.

■

Powyższy przykład z powodzeniem można wykorzystać do pokazania, że obliczenie pochodnej na kartce i bezpośrednie jej zaimplementowanie ma duży sens praktyczny! Nie mówiąc o tym, że nawet jeżeli ewaluacja pochodnej trwa tyle samo czasu co ewaluacja oryginalnej funkcji to przybliżanie pochodnej numerycznej duplikuje czas potrzebny na jej obliczenie. Dlaczego? Bo wymagane są dwie ewaluacje funkcji w punkcie $f(x)$ oraz $f(x+\varepsilon)$. Zwykle więc, wraz ze zwiększającą się wymiarowością problemu, zysk obliczeniowy wynikający z wyprowadzenia pochodnej i bezpośredniego jej obliczenia będzie spory.

6.2.2 * Dlaczego w praktyce korzystamy z przybliżenia wycentrowanego?

ADVANCED

Pora na wyjaśnienie dlaczego w praktyce lepsze przybliżenie uzyskujemy poprzez wycentrowany wzór 6.3 niż poprzez wzór 6.2 wymagający tylko jednej dodatkowej ewaluacji funkcji celu, oprócz jej wartości w danym puncie. Zwróć uwagę, że często tak czy tak będziemy chcieli obliczyć $f(x)$, żeby po prostu wiedzieć jak nam idzie proces optymalizacji. Potencjalnie więc możliwość ponownego użycia wyniku tego obliczenia w przybliżeniu pochodnej byłaby bardziej efektywna, zmniejszając konieczną liczbę ewaluacji funkcji celu. Wzór 6.2 także bezpośrednio wynikał z definicji pochodnej. Dlaczego więc się męczymy z wycentrowanym wzorem 6.3?

Aby określić błąd przybliżenia pochodnej użyjemy twierdzenia Taylora. Założymy więc, że nasza funkcja jest trzykrotnie różniczkowalna w okolicach punktu x dla którego przybliżamy wartość pochodnej. Zapiszmy przybliżenie Taylora drugiego rzędu dla $f(x+\varepsilon)$ oraz $f(x-\varepsilon)$

$$f(x+\varepsilon) = f(x) + f'(x)\varepsilon + \frac{1}{2}f''(x)\varepsilon^2 \quad \underbrace{+\frac{1}{6}f^{(3)}(c_1)\varepsilon^3}_{\text{reszta w postaci Lagrange'a}}$$

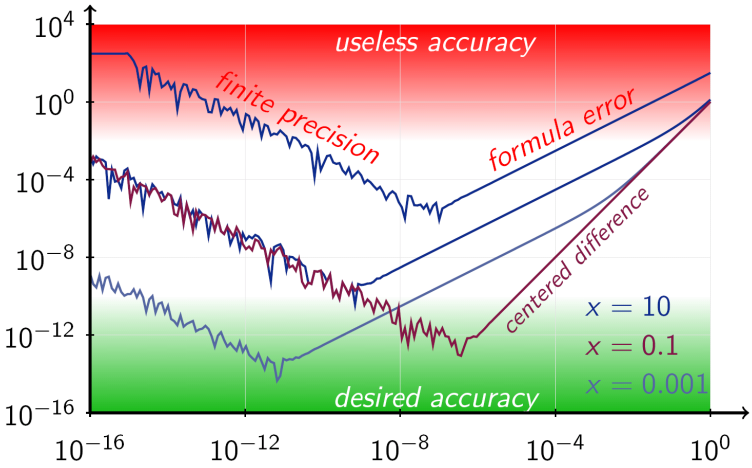
$$f(x-\varepsilon) = f(x) - f'(x)\varepsilon + \frac{1}{2}f''(x)\varepsilon^2 \quad \underbrace{-\frac{1}{6}f^{(3)}(c_2)\varepsilon^3}_{\text{reszta w postaci Lagrange'a}}$$

gdzie składniki postaci $\frac{1}{6}f^{(3)}(c_i)\varepsilon^3$ są resztami pozwalającymi nam na zapis ze znakiem równości (c_1 i c_2 są pewnymi stałymi, których nie znamy, ale wiemy że istnieją). Uzyskajmy więc wzór na różnicę $f(x+\varepsilon) - f(x-\varepsilon)$ poprzez odjęcie od siebie tych dwóch równań.

$$\begin{aligned} f(x+\varepsilon) - f(x-\varepsilon) &= \cancel{f(x)} + f'(x)\varepsilon + \cancel{\frac{1}{2}f''(x)\varepsilon^2} + \frac{1}{6}f^{(3)}(c_1)\varepsilon^3 \\ &\quad - \cancel{f(x)} + f'(x)\varepsilon - \cancel{\frac{1}{2}f''(x)\varepsilon^2} + \frac{1}{6}f^{(3)}(c_1)\varepsilon^3 \\ &= 2f'(x)\varepsilon + \frac{1}{6}\varepsilon^3(f^{(3)}(c_1) + f^{(3)}(c_2)) \end{aligned}$$

Przekształcając, aby otrzymać wzór na pochodną:

$$2f'(x)\varepsilon = f(x+\varepsilon) - f(x-\varepsilon) - \frac{1}{6}(f^{(3)}(c_1) + f^{(3)}(c_2))\varepsilon^3$$



Rysunek 6.3: Porównanie błędu przybliżenia numerycznego gradientu dla funkcji $f(x) = x^3$ dla trzech wartości x w zależności od wyboru wartości ε. Zwróć uwagę, że wybór zbyt małego ε powoduje problemy z arytmetyką zmiennoprzecinkową (błędy wynikające z *finite precision*), a wybór zbyt dużego ε powoduje błędy wynikające z używania przybliżenia numerycznego (*formula error*). Na wykresie widoczne jest także błąd przybliżenia wzorem wycentrowanym dla $x = 0.1$ - zwróć uwagę, że popełnia ono znacznie mniejszy błąd niż standardowy wzór. [1]

$$2f'(x) = \frac{f(x+\varepsilon) - f(x-\varepsilon)}{\varepsilon} - \frac{1}{6}(f^{(3)}(c_1) + f^{(3)}(c_2))\varepsilon^2$$

$$f'(x) = \underbrace{\frac{f(x+\varepsilon) - f(x-\varepsilon)}{2\varepsilon}}_{\text{nasze przybliżenie}} - \underbrace{\frac{1}{12}(f^{(3)}(c_1) + f^{(3)}(c_2))}_{\text{stała (błąd)}}\varepsilon^2$$

Błąd naszego przybliżenia jest więc rzędu $O(\varepsilon^2)$. Pamiętaj, że ze względu na problemy numeryczne nie możemy ustawić ε na bardzo małą liczbę (patrz rysunek 6.3), więc popełnianie błędu który maleje kwadratowo wraz ze zmniejszaniem ε bardzo nas cieszy. Poprzez analogiczną analizę można pokazać[3], że błąd dla przybliżenia wzorem $f'(x) \approx \frac{f(x+\varepsilon)-f(x)}{\varepsilon}$ jest rzędu $O(\varepsilon)$.

Czy można lepiej? To znaczy czy można uzyskiwać przybliżenia dokładniejsze, które mają jeszcze lepszą niż kwadratową resztę? Można, choć wzory robią się wtedy trochę bardziej skomplikowane. Na przykład można pokazać że przybliżenie z resztą rzędu $O(\varepsilon^4)$ można uzyskać wzorem

$$f'(x) \approx \frac{-f(x+2\varepsilon) + 8f(x+\varepsilon) - 8f(x-\varepsilon) + f(x-2\varepsilon)}{12\varepsilon}$$

dla funkcji pięciokrotnie różniczkowalnej. Wzór może i jest bardziej skomplikowany, ale w zasadzie jest on bardzo prosty do implementacji w praktyce. Zła wiadomość jest jednak taka, że w jego środku występują dwie dodatkowe ewaluacje funkcji celu dla

[3] Spróbuj! Masz już gotowe przybliżenie dla $f(x+\varepsilon)$, a $f(x)$ nie trzeba przybliżać, bo to jest wartość funkcji w naszym punkcie. W zasadzie więc od przybliżenia $f(x+\varepsilon)$ odejmujesz $f(x)$ co sprawi, że trzeci term przybliżenia się nie skróci jak to miało miejsce w naszym przypadku!

$f(x \pm 2\varepsilon)$ co zduplikuje czas obliczenia przybliżenia. Wniosek jest więc następujący: można uzyskać lepsze przybliżenie, ale kosztem dłuższych obliczeń... W praktyce jednak najczęściej korzystamy z prostszej formuły wycentrowanej, szczególnie jeśli ewaluacja $f(x)$ jest kosztowna obliczeniowo. Dodatkowo, jeśli funkcja jest z naszego punktu widzenia czarną skrzynką (nie możemy policzyć dla niej pochodnej analitycznie) to robienie coraz mocniejszych założeń o wielokrotnej różniczkowalności jest coraz to bardziej wątpliwe.

! Co ciekawe, takie formuły można również wyznaczyć dla pochodnych kolejnych rzędów! Na przykład drugą pochodną można przybliżyć poprzez

$$f''(x) \approx \frac{f(x+\varepsilon) - 2f(x) + f(x-\varepsilon)}{\varepsilon^2}$$

6.3 Dodatkowe rozszerzenia

6.3.1 Problemy z ograniczeniami

Okazuje się, że algorytm stochastycznego spadku wzdłuż gradientu można stosunkowo łatwo uogólnić do problemu optymalizacji z ograniczeniami.

$$\min \quad f(x) \quad \text{subject to} \quad x \in \mathrm{C}$$

gdzie C jest zbiorem wypukłym. Tak naprawdę ograniczenia nie sprawiają że nasz algorytm nagle przestanie działać, jednak problematyczne jest to, że w czasie optymalizacji algorytm po wykonaniu kroku aktualizacji rozwiązania $x \leftarrow x - \eta \nabla f(x)$ może wyjść poza zbiór rozwiązań dopuszczalnych.

Okazuje się, że rozwiązanie tego problemu jest bardzo proste: jeśli nasz x wyskoczył poza nasz zbiór C to znajdź najbliższy punkt w tym zbiorze i udawaj że nic się nie stało. Pełna aktualizacja w algorytmie będzie więc wyglądała następująco:

$$x \leftarrow x - \eta \nabla f(x)$$

$$x \leftarrow \arg\min_{c \in \mathrm{C}} ||c - x||$$

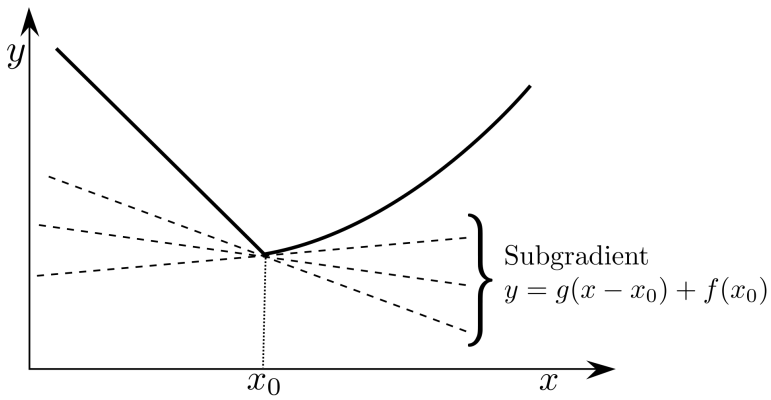
Zwróć uwagę, że jeśli po wykonaniu pierwszej linijki kodu nasz x nadal jest w zbiorze C to drugi krok nic nie zrobi! Stanie się tak, ponieważ punkt jest najbliższy do samego siebie ;) Choć modyfikacja jest bardzo prosta to ma bardzo podobne gwarancje teoretyczne, wymaga ona jednak rozwiązania problemu optymalizacyjnego jako część iteracji innego algorytmu optymalizacyjnego.

6.3.2 Problemy wypukłe nieróżniczkowalne

Algorytm stochastycznego spadku wzdłuż gradientu wymaga, aby funkcja była różniczkowalna tj. aby było możliwe policzenie kierunku spadku $\nabla f(x)$. Okazuje się jednak, że algorytm ten można zastosować także do nieróżniczkowalnych funkcji wypukłych!

Przypomnijmy, że dla funkcji wypukłych styczna do wykresu leży zawsze całkowicie pod nim.

$$f(\mathrm{x}) \geq f(\mathrm{x}_0) + \nabla f(\mathrm{x}_0)^T (\mathrm{x} - \mathrm{x}_0)$$



Rysunek 6.4: Wizualizacja definicji podpochodnej w punkcie nieróżniczkowalnym.

Funkcja wartości bezwzględnej jest nieróżniczkowalna w punkcie $x = 0$, ale... bez problemu znajdziemy wiele linii przechodzących przez ten punkt, aby spełniały powyższą równość! Ta obserwacja prowadzi nas do uogólnienia definicji pochodnej i zdefiniowania pod pochodnej.

Definicja 6.3 — Podpochodna. Podpochodną funkcji wypukłej nazywamy takie g, które spełnia warunek:

$$f(\mathsf{x}) \geq f(\mathsf{x}_0) + \mathsf{g}^T(\mathsf{x} - \mathsf{x}_0)$$

Zwróć uwagę, że jeżeli pochodna istnieje w danym punkcie funkcji wypukłej to automatycznie jest ona jej podpochodną. W punktach nieróżniczkowalnych może istnieć wiele wektorów (liczb w 1D) spełniających to równanie (jak w przykładzie z wartością bezwzględną lub na rysunku 7.7).

Używając podpochodnej modyfikacja algorytmu jest trywialna. W każdej z iteracji zamiast pochodnej wstawiamy jedną z podpochodnych (przypominam: tam gdzie funkcja jest różniczkowalna istnieje tylko jedna podpochodna równa pochodnej, więc w takich punktach algorytm pozostaje bez zmian) i to tyle... działa :)

! To rozumowanie aplikuje się także do standardowego algorytmu spadku wzdłuż gradientu.

6.3.3 Implementacje rozproszone

Algorytm stochastycznego spadku wzdłuż gradientu również można dość prosto rozproszyć. Zarysujemy tutaj zgrubnie podejście które nazywamy asynchronicznym SGD (ang. *Asynchronous Stochastic Gradient Descent*).

Na jednym z komputerów w klastrze tzw. masterze przechowujemy oficjalne, ostateczne rozwiązanie x. Ponieważ cząstkowa funkcja celu jak i jej gradient zależy tylko od konkretnego składnika sumy (w uczeniu maszynowym: od konkretnych obserwacji) możemy ją obliczyć *niezależnie* od innych składników (innych danych).

Możemy więc rozproszyć nasze dane po wszystkich komputerach w klastrze i każdy z nich lokalnie wykonuje algorytm stochastycznego spadku wzdłuż gradientu tylko na swoich obserwacjach. W czasie trwania optymalizacji algorytm oprócz swojego rozwiązania przechowuje także gradienty $\nabla f(x)$ które wyznacza w trakcie swojego działania. Co jakiś

czas (zupełnie asynchronicznie, bez wiedzy o innych komputerach w klastrze) węzeł wysyła swoje policzone gradienty $\nabla f(x)$ do master'a, a master używa ich aby zaktualizować swoje rozwiązanie (zwykły krok SGD). Dalej master przesyła węzłowi aktualne najlepsze rozwiązanie (uaktualniane gradientami nadsyłanymi co jakiś czas przez wszystkie węzły), a węzeł zastępuje nim swoje rozwiązanie i rozpoczyna kolejne iteracje optymalizacji.

Efektywne rozpraszanie algorytmu stochastycznego spadku wzdłuż gradientu jest nadal aktywnym przedmiotem badań w uczeniu maszynowym i optymalizacji.

Literatura

Literatura powtórkowa

Dobry opis algorytmu stochastycznego spadku wzdłuż gradientu, wraz z pogłębioną analizą teoretyczną, można znaleźć w książce [6].

Literatura dla chętnych

Ciekawy opis trików związanych z praktycznym użyciem algorytmu SGD prezentuje Leon Bottou (Microsoft Reserach) w rozdziale książki [5] pod tytułem *Stochastic Gradient Descent Tricks* dostępnego pod adresem `https://www.microsoft.com/en-us/research/wp-content/uploads/2012/01/tricks-2012.pdf`.

Za tydzień rozpoczynamy omawianie technik optymalizacyjnych drugiego rzędu, ponieważ zostały już one omówione na wykładzie, zachęcam do obejrzenia fragmentu wykładu z optymalizacji na Stanfordzie który podsumowuje metody spadku gradientu i opowiada o zaletach metod Newtona (już za tydzień ;) `https://www.youtube.com/watch?v=NPybf7JgQ7I`

Bibliografia

[1] Numerical differentiation – Wikipedia, wolna encyklopedia. `https://en.wikipedia.org/wiki/Numerical_differentiation`. Dostęp: 2018-04-10.

[2] Introduction to auto-encoder. `https://wikidocs.net/3413`. Dostęp: 2018-04-10.

[3] Aaron Defazio, Francis R. Bach, i Simon Lacoste-Julien. SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives. *CoRR*, abs/1407.0202, 2014. URL `http://arxiv.org/abs/1407.0202`.

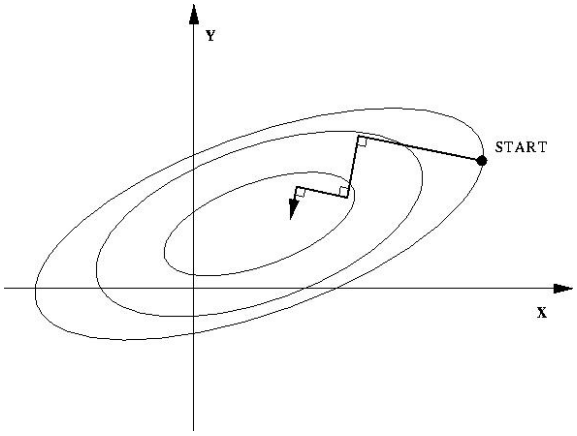
[4] Ian Goodfellow, Yoshua Bengio, i Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[5] Genevieve B. Orr i Klaus-Robert Mueller, editors. *Neural Networks : Tricks of the Trade*, volume 1524 of *Lecture Notes in Computer Science*. Springer, 1998.

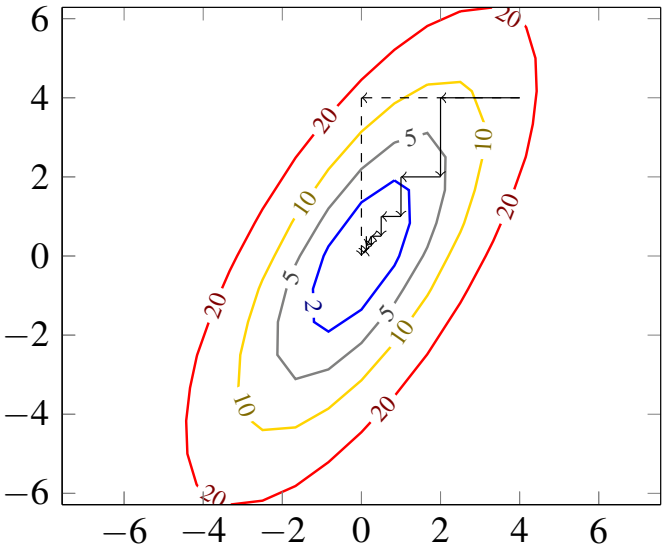
[6] Shai Shalev-Shwartz i Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, New York, NY, USA, 2014. ISBN 1107057132, 9781107057135.

7. Metody kierunków sprzężonych

Na poprzednich zajęciach poznaliśmy metodę Newtona, która w stosunkowo małej liczbie iteracji potrafiła znaleźć minimum funkcji (o ile była dobrze zainicjalizowana). W szczególności potrafiła ona znaleźć minimum funkcji kwadratowej w jednej iteracji. Metoda idealna? Niezupełnie. Jedna iteracja metody Newtona wymaga od nas stworzenia (i odwrócenia) hesjanu czyli macierzy o wymiarach $n \times n$, co przy problemach o dużej liczbie wymiarów n uniemożliwia jej wykorzystanie (zbyt duże wymagania pamięciowe) lub sprawia, że metoda działa bardzo wolno. Z drugiej strony, jedna iteracja metody najszybszego spadku była znacznie szybsza i nie wymagała składowania w pamięci hesjanu, jednakże, w przypadku problemów o wysokim współczynniku uwarunkowania κ pojawiał się silny efekt tzw. zygzakowania, który uniemożliwiał szybką zbieżność. Intuicyjnie polegał on na wykonywaniu bardzo małych kroków w powtarzających się po sobie, prostopadłych kierunkach. Możemy to zaobserwować na rysunku po prawej stronie: kierunki w pierwszej i trzeciej iteracji (oraz w drugiej i czwartej) są dokładnie takie same. Co zaskakujące, pomimo tego że metoda najszybszego spadku wybiera kierunek *najszybszego* lokalnego spadku (ujemny gradient) i porusza się w tym kierunku tak długo jak funkcja spada (optymalizacja η) to wydaje się, że kierunki wybierane przez tę metodę są w pewien sposób nieoptymalne lub przynajmniej niedostatecznie dobrze wykorzystane – w kolejnych iteracjach możemy optymalizować w powtarzających



Rysunek 7.1: Działanie algorytmu najszybszego spadku [?].



Rysunek 7.2: Działanie algorytmu najszybszego spadku oraz *nieistniejący* algorytm idealny (strzałki przerywane) który efektywnie wykorzystywałby kierunki przeszukiwania.

się po sobie kierunkach. Na tych zajęciach będziemy rozważać metody analogiczne do najszybszego spadku, ale które jednak nie zawsze wybierają kierunki najszybszego lokalnego spadku – kierunki te będziemy nazwać kierunkami sprzężonymi (ang. *conjugate*).

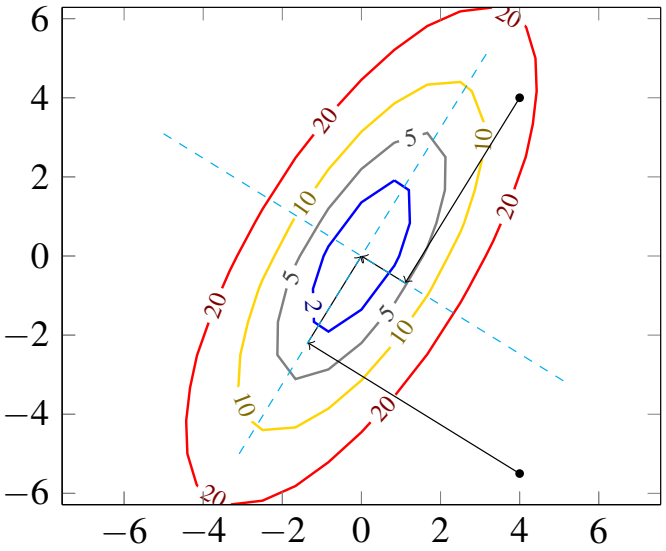
7.1 Optymalizacja w kierunkach sprzężonych

! Rozważania w tym rozdziale są ograniczone do wypukłych wielowymiarowych funkcji kwadratowych, dopiero na końcu zostaną omówione modyfikacje zaprezentowanej metody dla ogólnych funkcji.

$$f(\mathsf{x}) = \frac{1}{2}\mathsf{x}^\mathsf{T}\mathsf{A}\mathsf{x} - \mathsf{b}^\mathsf{T}\mathsf{x} + c$$

$$\nabla f(\mathsf{x}) = \mathsf{A}\mathsf{x} - \mathsf{b} \quad \Rightarrow \quad \mathsf{x}^* = \mathsf{A}^{-1}\mathsf{b}$$

Jak już chwilę temu zauważyliśmy metoda najszybszego spadku nie wykorzystuje dostatecznie dobrze wyznaczonego kierunku optymalizacji tj. jest możliwa sytuacja (problem zygzakowania) w której algorytm kolejno optymalizuje w powtarzających się po sobie kierunkach np. patrząc na rysunek 7.1 można by połączyć pierwszą i trzecią iterację i przesunąć się w żądanym kierunku raz a dobrze. Problem polega jednak na tym, że długości takiego łączonego kroku nie potrafimy z góry oszacować. Patrząc na pierwszą iterację na rysunku 7.2 (czy na rysunku 7.1) widzimy, że gdyby dłużej poruszać się w wyznaczonym kierunku to funkcja zaczęłaby rosnąć. Co ważne, krok o (a posteriori znanej) idealnej długości wykonałby aktualizację do punktu, który ma taką samą (albo nawet gorszą) wartość funkcji celu! Wydaje się więc, że byłoby niezwykle trudno zaprojektować metodę, która rzeczywiście wykonałaby krok w danym kierunku o idealnej długości (o ile oczywiście nie mamy jakiejś dodatkowej wiedzy jak np. o hesjanie – metoda Newton'a).



Rysunek 7.3: Optymalizacja w kierunkach własnych dla dwóch wybranych punktów. Kierunki własne zaznaczono przerywanym krzyżem.

7.1.1 Optymalizacja w kierunkach wyznaczonych przez wektory własne

Zaproponujmy więc ogólny schemat algorytmu wykonujący krok w pewnych kierunkach d_k:

$$\mathsf{x}_{k+1} = \mathsf{x}_k + \eta_k \mathsf{d}_k$$

gdzie η_k jest wyznaczane poprzez jednowymiarową optymalizację. Zwróć uwagę, że gdyby za kierunki optymalizacji d_k uznać kolejne gradienty w punktach x_k to dostalibyśmy dokładnie metodę najszybszego spadku (z drobnostką notacyjną: η_k byłoby zawsze ujemne zamiast dodatnie).

Sam pomysł algorytmu najszybszego spadku, aby wybierać najlepszą długość kroku η poprzez jednowymiarową optymalizację funkcji celu w danym kierunku wydaje się być bardzo dobrym. Nie wiem czy miałbyś pomysł na wskazanie lepszej procedury wyboru długości kroku niż „poruszaj się w danym kierunku tak długo jak wartość funkcji spada”. Tego elementu metody raczej nie uda nam się poprawić: czy zatem istnieją może lepsze kierunki do wykonywania optymalizacji? Czyli takie, dla których wykonanie jednowymiarowej optymalizacji w ich kierunku oznacza zoptymalizowanie funkcji celu w tym kierunku „raz a dobrze”, a algorytm w całym swoim przyszłym działaniu nie będzie miał potrzeby ponownego optymalizowania w tym kierunku?

Ćwiczenie 7.1 Spójrz na wykres 7.2 i zastanów się czy widzisz takie kierunki? ■

Oczywiście, takie działanie metody byłoby możliwe, o ile poruszalibyśmy się po kierunkach osi głównych elips wyznaczonych przez warstwice funkcji! Minimum funkcji kwadratowej leży dokładnie w centrum elipsy-warstwicy funkcji. Idąc więc w kierunku wybranej osi głównej elipsy funkcja spada aż na tej wybranej osi elipsa osiągnie swoje centrum (minumum). Nie jest więc konieczne optymalizowanie w takim kierunku ponownie – w zrozumieniu tego faktu może być pomocny rysunek 7.3. Kierunki wyznaczające osie główne elips to oczywiście kierunki wyznaczone przez wektory własne macierzy.

WYKŁAD

Zaproponujmy więc algorytm, który najpierw wyznacza wektory własne q_i macierzy, a potem wykonuje w ich kierunkach iterację:

$$\mathsf{x}_{k+1} = \mathsf{x}_k + \eta_k \mathsf{q}_k$$

gdzie η_k jest wyznaczane poprzez jednowymiarową optymalizację. Zwróć uwagę, że właśnie stworzyłeś algorytm który zoptymalizuje dowolną formę kwadratową zawsze w n iteracjach (gdzie n to liczba wymiarów)! Po wykonaniu iteracji w danym kierunku $\mathsf{d}_k = \mathsf{q}_k$ nigdy nie będzie potrzeby ponownej optymalizacji w tym kierunku, ponieważ (dla tego kierunku) znaleźliśmy się w centrum elipsy (minimum). Zwróć uwagę, że ta gwarancja optymalizacji w n iteracjach jest zupełnie niezależna od współczynnika uwarunkowania problemu – tak dzieje się zawsze! I, co istotne, w żadnym miejscu naszego algorytmu nie potrzeba policzyć hesjanu, który dla problemów o dużej wymiarowości mógłby się nawet nie zmieścić w pamięci, o jego odwracaniu i implementacji nie wspominając.

Współczynnik η_k wyznaczamy w takim algorytmie np. poprzez metodę złotego podziału. Jednakże dzięki założeniu, że funkcja jest funkcją kwadratową wzór na η_k można prosto wyznaczyć. Skoro η_k jest wybierana w taki sposób, że funkcji w danym kierunku nie można już optymalizować (jest w swoim jednowymiarowym minimum) to gradient w kolejnym punkcie algorytmu musi być prostopadły do tego kierunku (tak samo jak w metodzie najszybszego spadku) czyli $\mathsf{d}_k^T \nabla f(\mathsf{x}_{k+1}) = 0$. Przekształcając otrzymujemy:

$$\begin{aligned}
\mathsf{d}_k^T \nabla f(\mathsf{x}_{k+1}) &= \mathsf{d}_k^T (\mathsf{A}\mathsf{x}_{\mathsf{k}+1} - \mathsf{b}) \quad \text{/ze wzoru na gradient/} \\
&= \mathsf{d}_k^T (\mathsf{A}(\mathsf{x}_\mathsf{k} + \eta_\mathsf{k} \mathsf{d}_\mathsf{k}) - \mathsf{b}) \quad \text{/z iteracji algorytmu/} \\
&= \mathsf{d}_k^T (\mathsf{A}\mathsf{x}_k + \eta_k \mathsf{A}\mathsf{d}_k - \mathsf{b}) \quad \text{/przestawiając b/} \\
&= \mathsf{d}_k^T (\underbrace{\mathsf{A}\mathsf{x}_k - \mathsf{b}}_{\nabla f(\mathsf{x}_k)} + \eta_k \mathsf{A}\mathsf{d}_k) = \mathsf{d}_k^T \nabla f(\mathsf{x}_k) + \eta_k \mathsf{d}_k^T \mathsf{A}\mathsf{d}_k
\end{aligned} \tag{7.1}$$

A skoro wektory te są prostopadłe:

$$\mathsf{d}_k^T \nabla f(\mathsf{x}_k) + \eta_k \mathsf{d}_k^T \mathsf{A}\mathsf{d}_k = 0 \quad \Rightarrow \quad \eta_k \mathsf{d}_k^T \mathsf{A}\mathsf{d}_k = -\mathsf{d}_k^T \nabla f(\mathsf{x}_k)$$

$$\eta_k = -\frac{\mathsf{d}_k^T \nabla f(\mathsf{x}_k)}{\mathsf{d}_k^T \mathsf{A}\mathsf{d}_k}$$

Powyższy wzór działa zawsze dla ogólnej postaci algorytmu i funkcji kwadratowej, ponieważ nigdzie nie wykorzystaliśmy faktu, że kierunki są wektorami własnymi!

ADVANCED

Zakładając, że rzeczywiście optymalizujemy w kierunkach wektorów własnych o długości jednostkowej $\sqrt{\mathsf{q}_\mathsf{i}^\mathsf{T} \mathsf{q}_\mathsf{i}} = 1$ otrzymujemy po podstawieniu $\mathsf{d}_\mathsf{k} = \mathsf{q}_\mathsf{i}$:

$$\eta_k = -\frac{\mathsf{q}_i^T \nabla f(\mathsf{x}_k)}{\mathsf{q}_i^T \mathsf{A}\mathsf{q}_i} = -\frac{\mathsf{q}_i^T \nabla f(\mathsf{x}_k)}{\lambda_i \underbrace{\mathsf{q}_i^T \mathsf{q}_i}_{=1}} = -\frac{1}{\lambda_i} \mathsf{q}_i^T \nabla f(\mathsf{x}_k)$$

Możemy również udowodnić, że rzeczywiście po n iteracjach funkcja kwadratowa zostanie zminimalizowana przez nasz prototypowy algorytm. Pokażemy to poprzez pokazanie, że po wykonaniu iteracji wektor błędu e_{k+1} jest prostopadły do właśnie wyko-

rzystanego wektora[1] q_k czyli cała część odpowiedzialna za błąd w kierunku q_k została wyeliminowana.

$$
\begin{aligned}
q_k^T e_{k+1} &= q_k^T (x_{k+1} - x^*) = q_k^T (x_k + \eta_k q_k - \underbrace{A^{-1}b}_{=x^*}) = q_k^T x_k + \eta_k \underbrace{q_k^T q_k}_{=1} - q_k^T A^{-1} b \\
&= q_k^T x_k - \frac{1}{\lambda_k} q_k^T \nabla f(x_k) - q_k^T A^{-1} b \quad \text{/wartość własna macierzy odwrotnej to } \lambda_i^{-1} \text{ /} \\
&= q_k^T x_k - \frac{1}{\lambda_k} q_k^T (Ax_k - b) - \frac{1}{\lambda_k} q_k^T b \\
&= q_k^T x_k - \frac{1}{\lambda_k} \lambda_k q_k^T x_k + \frac{1}{\lambda_k} q_k^T b - \frac{1}{\lambda_k} q_k^T b = 0
\end{aligned}
\tag{7.2}
$$

Dlaczego ten algorytm nie ma jednak swojej własnej nazwy, a obecne zajęcia noszą tytuł „kierunki sprzężone” zamiast „kierunki własne”? Ponieważ wyznaczenie kierunków własnych formy kwadratowej jest tak samo złożone obliczeniowo jak znalezienie jej minimum w sposób analityczny (poprzez odwrócenie macierzy)! Przedstawiony algorytm oprócz ciekawej idei jest w zasadzie bezwartościowy.

7.1.2 Wektory sprzężone

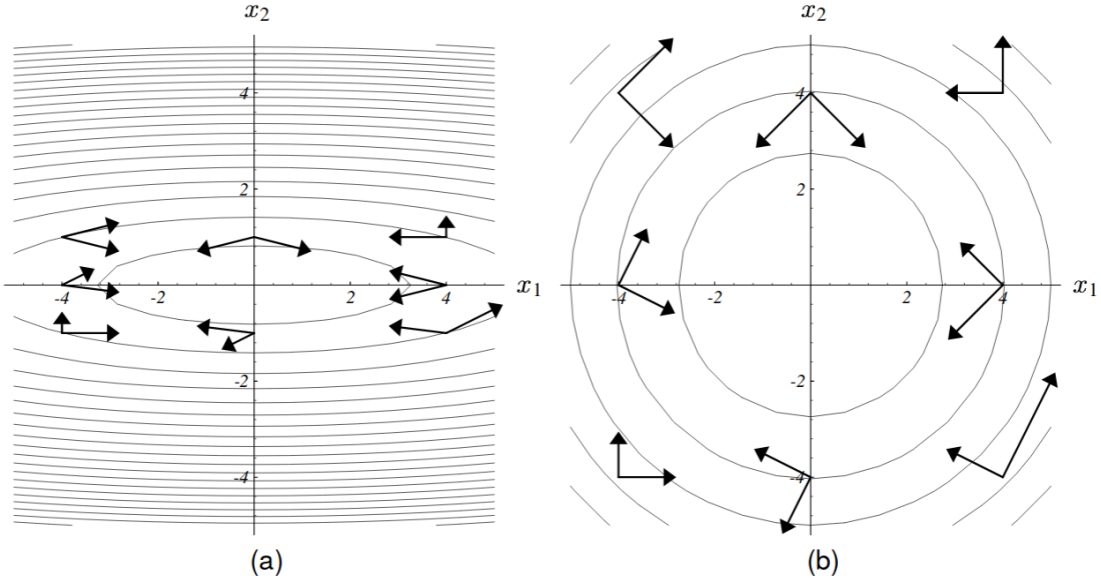
Czy istnieją może zatem jakieś kierunki podobne do wektorów własnych w przedstawionym kontekście optymalizacyjnym? Nasze zadanie zostało ciekawie zdefiniowane: chcielibyśmy znaleźć kierunki, które miałyby wszystkie pożądane właściwości wektorów własnych (optymalizacja funkcji kwadratowej w n iteracjach!), ale jednocześnie byłyby możliwe do wyznaczenia w szybki sposób. Okazuje się, że jest całe mnóstwo takich zestawów wektorów, a wektory te wcale nie muszą być prostopadłe – takie wektory to wektory sprzężone.

Przypomnijmy, wektory własne macierzy symetrycznej optymalizują funkcję kwadratową w n iteracjach, stanowią bazę oraz są prostopadłe. Kierunki sprzężone również optymalizują funkcję kwadratową w n iteracjach, stanowią bazę oraz... są prostopadłe *w zetknięciu się z macierzą* A. Co to oznacza? Wektory które są prostopadłe spełniają równość $d_i^T d_j = 0$. Poprzez prostopadłość w kontekście macierzy A rozumiem to, że wektor po „przekręceniu” przez tę macierz będzie prostopadły do drugiego tj. że Ad_j będzie prostopadłe do d_i i na odwrót. Czyli

$$
(Ad_j)^T d_i = d_j^T \underbrace{A^T}_{\text{symetryczna}} d_i = d_j^T A d_i = 0 \quad \text{oraz} \quad (Ad_i)^T d_j = d_i^T A^T d_j = d_i^T A d_j = 0
$$

Tę idee przedstawia rysunek 7.4. Zwróć uwagę, że po „rozciągnięciu” elipis wynikających z formy kwadratowej zbudowanej na macierzy A wektory są prostopadłe i dodatkowo w tej rozciągniętej przestrzeni każdy z nich jest kierunkiem osi głównej elipsy (która jest w tej przestrzeni po prostu kołem, więc każdy wektor ma taką własność). Widać tu

[1] Dodatkowo wiemy, że w kolejnych iteracjach do wektora błędu na pewno nie zostanie dodany wcześniej wykorzystany q_k, ponieważ algorytm modyfikuje x poprzez dodawanie kolejnych wektorów własnych, które są prostopadłe.



Rysunek 7.4: Pary wektorów na rysunku (a) są sprzężone (prostopadłe w kontekście w macierzy A), ponieważ po „rozprostowaniu” elipsy są one prostopadłe. [**?**]

pewną analogię do metody najszybszego spadku: metoda ta tworzyła kolejne prostopadłe kierunki w przestrzeni oryginalnej, podczas gdy metody oparte o kierunki sprzężone będą tworzyły kierunki (zwykle) nieprostopadłe w przestrzeni oryginalnej, ale za to prostopadłe w kontekście macierzy A.

Definicja 7.1 — Wektory sprzężone. Niech A będzie dodatnio określoną macierzą symetryczną o wymiarach $n \times n$. Zbiór n niezerowych wektorów nazywamy sprzężonymi o ile zachodzi:

$$\mathsf{d}_i^T \mathsf{A} \mathsf{d}_j = 0$$

dla każdego[a] $i \neq j$.

[a] Nota bene: gdyby warunek zachodził dla $i = j$ to macierz nie mogła by być dodatnio określona.

Wektory własne są wektorami sprzężonymi

Analizując rysunek 7.4 być może zauważyłeś wektory, które są prostopadłe zarówno w kontekście macierzy (sprzężone) jak i prostopadłe w zwykły sposób. Jakie to wektory? To oczywiście wektory własne ułożone wzdłuż osi głównej elipsy-warstwicy! Wektory własne i sprzężone mają podobne własności (w naszym optymalizacyjnym kontekście) ponieważ w rzeczywistości wektory własne *są* wektorami sprzężonymi! Zgodnie z definicją wektorów sprzężonych względem symetrycznej macierzy A:

$$\mathsf{q}_i^T \mathsf{A} \mathsf{q}_j = \lambda_j \underbrace{\mathsf{q}_i^T \mathsf{q}_j}_{\text{prostopadłe}} = 0$$

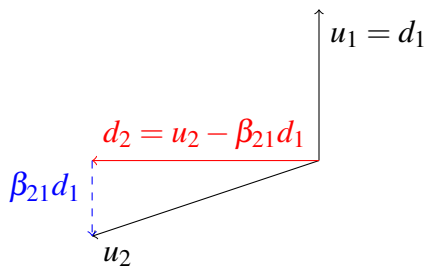
Wracając więc pamięcią do poprzedniej sekcji 7.1.1 przedstawiony algorytm był algorytmem w kierunkach sprzężonych, ale kierunki te niestety można było wyznaczyć w bardzo kosztowny obliczeniowo sposób.

Ćwiczenie 7.2 Metoda Gaussa-Seidela dla problemów ze współczynnikiem uwarunkowania $\kappa = 1$ działa jak metoda w kierunkach sprzężonych. Dlaczego? ■

7.1.3 Jak z wektorów liniowo niezależnych stworzyć wektory sprzężone?

Zatem czy możemy znaleźć wektory sprzężone w łatwy obliczeniowo sposób? Koncepcja naszego rozwiązania będzie następująca: rozpoczniemy z jakimiś wektorami u_i co do których jedynym wymaganiem będzie, że są liniowo niezależne[2], a następnie *przerobimy* je na wektory sprzężone. Istnieją przecież gotowe algorytmy do przerabiania wektorów na wektory prostopadłe (ortogonalizacja Grama-Schmidta) więc powinniśmy byli być w stanie przerabiać wektory na wektory prostopadłe po zetknięciu się z macierzą A.

Rozpocznijmy od opisu procesu zwykłej ortogonalizacji dla pary wektorów u_1 i u_2, które będziemy przerabiać na prostopadłe do siebie kierunki d_1 i d_2. Pierwszy wektor u_1 zachowujemy jak jest i podstawiamy $\mathsf{d}_1 = \mathsf{u}_1$. Kolejne wektory będziemy po prostu tak dopasowywać, aby były prostopadłe do niego. Rozważamy kolejny wektor u_2, który należy zortogonalizować do wektora d_1. Zrobimy to poprzez odjęcie od wektora u_2 części odpowiedzialnej za kierunek współdzielony z wektorem d_1



$$\mathsf{d}_2 = \mathsf{u}_2 - \beta_{21}\mathsf{d}_1$$

gdzie β_{21} jest pewną stałą którą trzeba by w jakiś sposób wyznaczyć. Dalej, gdybyśmy mieli jakiś kolejny wektor u_3 to należałoby go zortogonalizować względem zarówno d_1 jak i d_2 poprzez odjęcie od niego części związanych z tymi wektorami.

$$\mathsf{d}_3 = \mathsf{u}_3 - \beta_{31}\mathsf{d}_1 - \beta_{32}\mathsf{d}_2$$

Zwróć uwagę na podwójną indeksację współczynników β_{ij} – dla każdego wektora musimy dobierać inną stałą „ile chcemy odjąć kierunku pierwszego”, „ile drugiego” itd. Czyli pierwszy wektor nie wymaga od nas doboru stałej, drugi wymaga doboru jednej, trzeci dwóch i n-ty wymaga znalezienia $n-1$ współczynników.

$$\mathsf{d}_i = \mathsf{u}_i - \sum_{j=1}^{i-1} \beta_{ij}\mathsf{d}_j$$

Taką samą procedurę będziemy stosować dla wektorów strzeżonych z tą różnicą, że będziemy inaczej wybierać β tj. będziemy odejmować dany kierunek tak długo, aż w końcu u_i stanie się strzeżonym wektorem do reszty. Jak tylko znaleźć takie β_{ik}? Aby otrzymać wzór na β_{ik} pomnóżmy wyżej otrzymaną równość obustronnie przez $\mathsf{d}_k^T\mathsf{A}$. Zwróć uwagę, że żeby w ogóle β_{ik} było obecne w przestawionym wzorze na d_i (było elementem sumy) to potrzeba aby $k < i$. W związku z tym $k \neq i$ i (z definicji wektorów sprzężonych)

[2] czysto teoretycznie nawet jak byśmy wygenerowali po prostu losowe wektory to z dużą szansą będą one liniowo niezależne

lewa strona równości wynosi zero ($\mathsf{d}_k^T \mathsf{A} \mathsf{d}_i = 0$).

$$
\begin{aligned}
0 &= \mathsf{d}_k^T \mathsf{A} \mathsf{u}_i - \sum_{j=1}^{i-1} \beta_{ij} \mathsf{d}_k^T \mathsf{A} \mathsf{d}_j \\
&= \mathsf{d}_k^T \mathsf{A} \mathsf{u}_i - \beta_{i1} \underbrace{\mathsf{d}_k^T \mathsf{A} \mathsf{d}_1}_{=0} - \beta_{i2} \underbrace{\mathsf{d}_k^T \mathsf{A} \mathsf{d}_2}_{=0} - \beta_{i3} \underbrace{\mathsf{d}_k^T \mathsf{A} \mathsf{d}_3}_{=0} - \cdots - \beta_{ik} \mathsf{d}_k^T \mathsf{A} \mathsf{d}_k \ldots \\
&= \mathsf{d}_k^T \mathsf{A} \mathsf{u}_i - \beta_{ik} \mathsf{d}_k^T \mathsf{A} \mathsf{d}_k
\end{aligned}
\tag{7.3}
$$

$$
\mathsf{d}_k^T \mathsf{A} \mathsf{u}_i = \beta_{ik} \mathsf{d}_k^T \mathsf{A} \mathsf{d}_k \quad \Rightarrow \quad \beta_{ik} = \frac{\mathsf{d}_k^T \mathsf{A} \mathsf{u}_i}{\mathsf{d}_k^T \mathsf{A} \mathsf{d}_k}
$$

Mamy więc gotowy algorytm przerabiania dowolnych wektorów liniowo niezależnych na wektory sprzężone! Dość nieciekawa jest informacja, że nie jest on taki tani obliczeniowo: dla każdego kolejnego wektora wymaga on wyznaczania całego zestawu współczynników β, których obliczenie wymaga od nas mnożenia macierzy razy wektor.

7.1.4 Wybór „dobrych” wektorów do przerabiania na sprzężone

Poświęćmy chwilę na podsumowanie dotychczasowych rozdziałów. Po pierwsze wiemy, że optymalizacja wypukłej funkcji kwadratowej w kierunkach sprzężonych daje nam gwarancję osiągnięcia minimum po n iteracjach bez konieczności obliczania Hesjanu i jego odwracania. Po drugie, znamy metodę przerobienia dowolnych niezależnych wektorów w kierunki sprzężone. Możemy więc (podobnie jak w metodzie Gaussa-Seidela) rozpocząć z wektorami powiązanymi z kolejnymi wymiarami problemu np. dla problemu dwu-wymiarowego $\mathsf{i} = [1,0]^T$ oraz $\mathsf{j} = [0,1]^T$, a następnie przerobić te kierunki naszym algorytmem na kierunki sprzężone. To jak najbardziej zadziała, ale trochę problematyczne jest to, że algorytm przerabiania wektorów na wektory sprzężone jest dość kosztowny obliczeniowo – wyznaczanie mnóstwa kosztownych współczynników β. Czy można zatem wybrać lepsze wektory do przerabiania na wektory sprzężone? Tj. takie dla których pokazalibyśmy że np. niektóre z β są na pewno równe 0 i w ogóle nie musimy ich wyznaczać, albo wzór na nie się jakoś upraszcza i możemy je szybciej policzyć? Okazuje się, że tak – są to gradienty funkcji.

! Jeśli śledziłeś uważnie materiały dodatkowe to powinieneś być zaznajomiony z algorytmem Powell'a – jest to *bezgradientowy* algorytm w kierunkach sprzężonych! Co więcej, nie wymagał on liczenia żadnych współczynników β, a kierunki sprzężone pojawiały się w nim „automatycznie”. Dlaczego kierunki wybierane przez ten algorytm są sprzężone?

Wybór ujemnego gradientu w punkcie startowym jako pierwszego kierunku w którym chcemy dokonać optymalizacji wydaje się być dobrym pomysłem. Skoro zaczynam optymalizacje i jeszcze nic nie wiem o charakterystyce funkcji celu to rozsądnie jest wybrać kierunek najszybszego lokalnego spadku. Podstawiamy więc: $\mathsf{u}_1 = -\nabla f(\mathsf{x}_1)$. Jak pamiętasz metoda przerabiania wektorów na wektory sprzężone nie modyfikuje pierwszego wektora czyli

$$
\mathsf{u}_1 = \mathsf{d}_1 = -\nabla f(\mathsf{x}_1)
$$

Wykonujemy krok algorytmu, optymalizując funkcję w kierunku d_1 tak długo jak się da. W następnym kroku wybieramy $u_2 = -\nabla f(x_2)$ – natomiast wektor ten nie jest raczej automatycznie sprzężony do poprzedniego kierunku d_1, więc musimy go zmodyfikować/sprzężyć:

$$d_2 = u_2 - \beta_{21} d_1 = -\nabla f(x_2) - \beta_{21}(-\nabla f(x_1))$$

W kolejnej iteracji postępujemy analogicznie wybierając za $u_3 = -\nabla f(x_3)$ i sprzęgając go używając β_{31} i β_{32} itd. W tym miejscu warto sobie przypomnieć, że wektory które „sprzęgamy" powinny być liniowo niezależne – okazuje się że mamy taką gwarancję dla kolejnych gradientów (są one prostopadłe) – ale o tym za chwilę.

Mówiliśmy, że chcielibyśmy uzyskać wektory/kierunki przeszukiwań, które pozwoliłyby nam znacznie przyśpieszyć mechanizm sprzęgania kierunków np. poprzez pewność, że niektóre z β_{ij} byłyby zawsze równe 0 lub uproszczenie wzoru. Wyrażenie $\beta_{ij} = \frac{u_i^T A d_j}{d_j^T A d_j}$ jest równe zero o ile jego licznik czyli $u_i^T A d_j$ jest równe 0. Czy tak się rzeczywiście stanie dla niektórych β_{ij} jeśli za u_i podstawimy gradienty kolejnych punktów?

Zauważmy, że gradient funkcji kwadratowej możemy zapisać jako

$$\nabla f(x_{i+1}) = Ax_{i+1} - b = A \underbrace{(x_i + \eta_i d_i)}_{\text{wzór na iteracje}} - b = Ax_i - b + \eta_i A d_i = \nabla f(x_i) + \eta_i A d_i$$

czyli po przestawieniu[3] otrzymujemy $Ad_i = \eta_i^{-1}(\nabla f(x_{i+1}) - \nabla f(x_i))$ i możemy podstawić:

$$\begin{aligned}
\beta_{ij} &= \frac{u_i^T A d_j}{d_j^T A d_j} = \frac{u_i^T \eta_i^{-1}(\nabla f(x_{j+1}) - \nabla f(x_j))}{d_j^T \eta_i^{-1}(\nabla f(x_{j+1}) - \nabla f(x_j))} \quad \text{/skracamy } \eta_i \text{ /} \\
&= \frac{u_i^T(\nabla f(x_{j+1}) - \nabla f(x_j))}{d_j^T(\nabla f(x_{j+1}) - \nabla f(x_j))} \quad \text{/ponieważ za } u_i \text{ podstawiamy negacje gradientów/} \\
&= \frac{-\nabla f(x_i)^T(\nabla f(x_{j+1}) - \nabla f(x_j))}{d_j^T(\nabla f(x_{j+1}) - \nabla f(x_j))}
\end{aligned}$$

Wzór na pozór trochę się skomplikował, jednak w rzeczywistości możemy go obliczyć znacznie szybciej niż poprzednio, ponieważ zastąpiliśmy kosztowne mnożenie macierzy razy wektor (złożoność kwadratowa) poprzez tańsze mnożenie wektorowe (złożoność liniowa)! Ale będzie jeszcze lepiej! Używając takiego wzoru, łatwo teraz zbadać problem zerowania się β_{ij} – wystarczy zbadać iloczyny pomiędzy różnymi gradientami, a w szczególności czy takie iloczyny gradientów w pewnych sytuacjach nie zerują się.

Prześledźmy jeszcze raz proces sprzęgania się gradientów. Pierwszy gradient zachowujemy jak jest, w drugim potrzebujemy obliczyć $\beta_{21} = \frac{-\nabla f(x_2)^T(\nabla f(x_2) - \nabla f(x_1))}{d_1^T(\nabla f(x_2) - \nabla f(x_1))}$. Analizując licznik tego wyrażenia: iloczyn $\nabla f(x_2)^T \nabla f(x_2)$ nie równa się 0, o ile nie znaleźliśmy się w punkcie stacjonarnymi (po prostu $\nabla f(x_2) = 0$). Natomiast drugi iloczyn licznika: $\nabla f(x_2)^T \nabla f(x_1) = \nabla f(x_2)^T d_1 = 0$. Wynika to z faktu, że kierunek d_1 został zoptymalizowany i dalsze poruszanie się w tym kierunku na pewno nie powoduje spadku funkcji (analogiczne rozumowanie jak przy metodzie najszybszego spadku). Ostatecznie otrzymujemy więc $\beta_{21} = \frac{-\nabla f(x_2)^T \nabla f(x_2)}{\nabla f(x_1)^T \nabla f(x_1)}$.

[3] Zakładamy, że $\eta_i \neq 0$ bo gdyby $\eta_i = 0$ to algorytm w ogóle nie wyznaczałby kolejnego gradientu – utknąłby w poprzednim punkcie (po prostu zakończyłby swoje działanie).

W kolejnej iteracji algorytmu sprzęgamy kierunek $\mathsf{u}_3 = -\nabla f(\mathsf{x}_3)$ i musimy wyznaczyć aż dwa współczynniki β_{31} oraz β_{32}. Znów, zastanówmy się jak wyglądają iloczyny wektorowe pomiędzy kolejnymi gradientami?

$$\begin{aligned}\nabla f(\mathsf{x}_1)^T \nabla f(\mathsf{x}_3) &= \nabla f(\mathsf{x}_1)^T (\nabla f(\mathsf{x}_2) + \eta_2 \mathsf{A}\mathsf{d}_2) \quad \text{/ze wzoru na kolejny gradient/} \\ &= \underbrace{\nabla f(\mathsf{x}_1)^T \nabla f(\mathsf{x}_2)}_{=0} + \eta_2 \underbrace{\nabla f(\mathsf{x}_1)^T}_{=\mathsf{d}_1} \mathsf{A}\mathsf{d}_2 \\ &= \eta_2 \mathsf{d}_1{}^T \mathsf{A}\mathsf{d}_2 = 0 \quad \text{/bo sprzężone/}\end{aligned}$$

Z drugiej strony, ponieważ kierunek d_2 jest w tej iteracji optymalizowany wiemy, że $\mathsf{d}_2{}^T \nabla f(\mathsf{x}_3) = 0$. Podstawiając wzór na d_2 otrzymujemy:

$$0 = (\nabla f(\mathsf{x}_2) + \beta_{21} \nabla f(\mathsf{x}_1))^T \nabla f(\mathsf{x}_3) = \nabla f(\mathsf{x}_2)^T \nabla f(\mathsf{x}_3) + \beta_{21} \underbrace{\nabla f(\mathsf{x}_1)^T \nabla f(\mathsf{x}_3)}_{=0} = \nabla f(\mathsf{x}_2)^T \nabla f(\mathsf{x}_3)$$

Zarówno w tej, jak i poprzedniej iteracji wszystkie gradienty były do siebie prostopadłe! I rzeczywiście tak jest dla *każdej kolejnej iteracji* (prześledź koniecznie dowód na wykładzie!). To z kolei pozwala nam na dalsze uproszczenie wzoru na $\beta_{ij} = \frac{-\nabla f(\mathsf{x}_i)^\mathsf{T}(\nabla f(\mathsf{x}_{j+1}) - \nabla f(\mathsf{x}_j))}{\mathsf{d}_j^\mathsf{T}(\nabla f(\mathsf{x}_{j+1}) - \nabla f(\mathsf{x}_j))}$. Zauważ, że w liczniku mamy (po rozwinięciu nawiasu) dwa iloczyny wektorowe gradientów. Ponieważ zawsze $i < j$ (żeby sprężyć j-ty wektor usuwamy od niego poprzednie) wiemy, że drugi iloczyn na pewno wyniesie 0, ponieważ *różne* gradienty są do siebie prostopadłe. Z kolei pierwszy element iloczynu również wyniesie 0 o ile $i \neq j+1$ (mnożymy różne gradienty). Podsumowując: mamy dwie sytuacje, dla współczynnika ostatniego kierunku ($j = i-1$) otrzymujemy:

$$\beta_{i,j} = \frac{-\nabla f(\mathsf{x}_i)^\mathsf{T}(\nabla f(\mathsf{x}_i) - \nabla f(\mathsf{x}_{i-1}))}{\mathsf{d}_{i-1}^\mathsf{T}(\nabla f(\mathsf{x}_i) - \nabla f(\mathsf{x}_{i-1}))} = \frac{-\nabla f(\mathsf{x}_i)^\mathsf{T} \nabla f(\mathsf{x}_i)}{\mathsf{d}_{i-1}^\mathsf{T}(\nabla f(\mathsf{x}_i) - \nabla f(\mathsf{x}_{i-1}))} \tag{7.4}$$

A dla pozostałych współczynników ($j < i-1$) otrzymamy (tutaj na przykładzie $i = 5, j = 1$):

$$\beta_{5,1} = \frac{-\nabla f(\mathsf{x}_5)^\mathsf{T}(\nabla f(\mathsf{x}_2) - \nabla f(\mathsf{x}_1))}{\mathsf{d}_1^\mathsf{T}(\nabla f(\mathsf{x}_2) - \nabla f(\mathsf{x}_1))} = \frac{\underbrace{-\nabla f(\mathsf{x}_5)^\mathsf{T} \nabla f(\mathsf{x}_2)}_{=0} + \underbrace{\nabla f(\mathsf{x}_5)^\mathsf{T} \nabla f(\mathsf{x}_1)}_{=0}}{\mathsf{d}_1^\mathsf{T}(\nabla f(\mathsf{x}_2) - \nabla f(\mathsf{x}_1))} = 0$$

Okazuje się, że do sprzężenia danego gradientu potrzebujemy obliczyć tylko jeden, ostatni $\beta_{i,i-1}$ bo pozostałe wynoszą po prostu 0! Oznacza to też, że nie musimy pamiętać wszystkich starych kierunków, żeby je odejmować – wystarczy znać tylko poprzedni kierunek! To jest prawdziwa magia gradientów sprzężonych.

7.1.5 Podsumowanie: metoda gradientów sprzężonych

Ostatecznie, gdy spojrzymy na finalny kod algorytmu to metoda gradientów sprzężonych różni się od metody najszybszego spadku w zasadzie jedną linijką kodu: sprzężania gradientów zamiast ich bezpośredniego użycia. Całe to sprzężanie wymaga od nas jedynie odjęcia poprzedniego kierunku optymalizacji z wagą $\beta_t = \frac{-\nabla f(\mathsf{x}_{t+1})^\mathsf{T} \nabla f(\mathsf{x}_{t+1})}{\mathsf{d}_t^\mathsf{T}(\nabla f(\mathsf{x}_{t+1}) - \nabla f(\mathsf{x}_t))}$ która wymaga od nas dodatkowo pamiętania poprzedniego gradientu.

Algorytm 7.1 — Metoda gradientów sprzężonych.

$\mathsf{x}_1 \leftarrow$ INICJALIZUJ
$\mathsf{d}_1 \leftarrow \nabla f(\mathsf{x}_1)$
while warunek stopu nie jest spełniony[a] **do**

$$\eta_t = arg\min_{\eta} f(\mathsf{x}_t + \eta \mathsf{d}_t)$$

$$\mathsf{x}_{t+1} = \mathsf{x}_t + \eta_t \mathsf{d}_t$$

$$\mathsf{d}_{t+1} = -\nabla f(\mathsf{x}_t) - \beta_t \mathsf{d}_t$$

end while

[a] dla funkcji kwadratowej maksymalnie n iteracji!

Okazuje się, że podany wzór na współczynnik β_t można dalej odrobinę uprościć (jest to matematyczny ekwiwalent) do $\beta_t = \frac{-\nabla f(\mathsf{x}_{t+1})^{\mathsf{T}} \nabla f(\mathsf{x}_{t+1})}{\nabla f(\mathsf{x}_t)^T \nabla f(\mathsf{x}_t)}$ (postać Fletchera-Reevesa). Jednak w praktyce, podczas optymalizacji funkcji niekwadratowych zwykle preferujemy wersję Polaka-Ribiére'a:

$$\beta_t = \frac{-\nabla f(\mathsf{x}_{t+1})^{\mathsf{T}} (\nabla f(\mathsf{x}_{t+1}) - \nabla f(\mathsf{x}_t))}{\nabla f(\mathsf{x}_t)^T \nabla f(\mathsf{x}_t)}$$

Jeśli porównasz licznik tego wzoru do naszych wcześniejszych wyprowadzeń (równanie 7.4) to odkryjesz, że ta wersja wzoru po prostu nie zakłada, że $\nabla f(\mathsf{x}_{t+1})^T \nabla f(\mathsf{x}_t) = 0$. Nasze wyprowadzenie tego faktu zakładało, że optymalizowana funkcja jest funkcją kwadratową, jeśli więc optymalizujesz ogólną funkcję wypukłą – to stwierdzenie jest błędne i postać Polaka-Ribiére'a uwzględnia ten fakt.

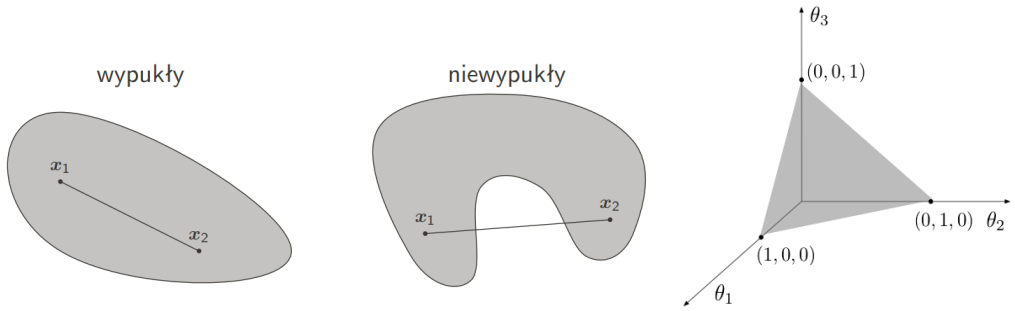
Nie będziemy tutaj analizować zbieżności tej metody dla funkcji niekwadratowej i poprzestańmy jedynie na dwie uwagi. Pierwsza: jeśli nasz algorytm dojdzie do miejsca w którym aproksymacja optymalizowanej funkcji funkcją kwadratową będzie poprawna to zbiegnie do minimum w maksymalnie n kolejnych iteracjach. Druga: na poprzednim laboratorium bardzo krótko wspominaliśmy o rozszerzeniach metody Newtona, które redukują złożoność obliczeniową (metody quasi-newtonowskie) okazuje się, że metoda gradientów sprzężonych może być interpretowana jako szczególny przypadek jednej z takich metod (BFGS).

! W praktyce, z powodu błędów numerycznych, algorytm sprzężonych gradientów może nie zoptymalizować funkcji kwadratowej w n iteracjach i kolejne iteracje będą potrzebne.

7.2 Funkcje wypukłe

7.2.1 Czemu funkcje wypukłe?

Pytanie brzmi czemu właściwie interesują nas funkcje wypukłe? Otóż odpowiedź jest prosta: *minimum lokalne* funkcji wypukłej jest zarazem *minimum globalnym*. W przypadku funkcji ściśle wypukłej jest także gwarancja, że takie minimum jest tylko jedno. Czyni to funkcje wypukłe prostymi do optymalizacji - prawdziwym wyzwaniem staje się w tym momencie transformacja rzeczywistych problemów do problemów wypukłych.



Rysunek 7.5: Przykład zbioru wypukłego i niewypukłego (lewa) oraz simplexa standardowego (prawa)

7.2.2 Kombinacje wypukłe

W świecie optymalizacji popularną rodziną funkcji są *funkcje wypukłe*. Zanim jednak przejdziemy do funkcji wypukłych omówmy kilka innych pomocnych bytów tak jak kombinacja wypukła:

Definicja 7.2 — Kombinacja wypukła. Kombinacją wypukłą punktów $x_i \in \mathbb{R}^d, d \in \mathbb{N}$ nazywamy sumę:

$$\sum_{i=1}^{n} \alpha_i x_i; \alpha_i \in \mathbb{R}+; \sum_{i=1}^{n} \alpha_i = 1$$

Kombinacja liniowa jest zatem sumą ważoną składowych punktów, gdzie wagi sumują się do 1 i są nieujemne (tworzą zatem rozkład prawdopodobieństwa - rozkład kategoryczny). Wszystkie kombinacje liniowe wybranych punktów tworzą razem zbiór wypukły.

Definicja 7.3 — Zbiór wypukły. Zbiór dowolnych punktów jest wypukły jeśli dowolna kombinacja wypukła tych punktów także znajduje się w tym zbiorze.

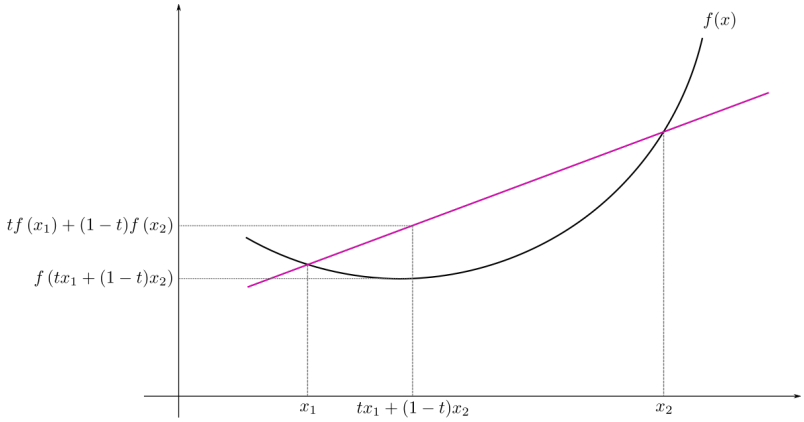
Zbiór wypukły możemy zatem zinterpretować geometrycznie jako figurę zawartą pomiędzy tymi punktami. W przypadku dwóch punktów dostajemy odcinek (λ decyduje o konkretnym położeniu na tym odcinku), dla 3 punktów mamy trójkąt, dla 4 czworokąt i tak dalej (aż do nieskończoności).

Jedną z bardziej znanych figur wypukłych jest simplex. W przestrzeni (k+1) - wymiarowej, simpleks definiowany jest przez (k+1) niezależnych liniowo punktów. Dla k=1 otrzymujemy 2 punkty - simpleks jest odcinkiem. Dla k=2 otrzymujemy trójkąt równoboczny, dla k=3 otrzymujemy czworościan (tetrahedron - nie jest to piramida!). Szczególnym przypadkiem simplexa jest simplex standardowy (probability/standard simplex):

Definicja 7.4 — Simpleks standardowy. k-wymiarowym simpleksem standardowym nazywamy taki zbiór punktów:

$$\{x \in \mathbb{R}^{k+1} : x_0 + \cdots + x_k = 1, x_i >= 0, i = 0, \ldots k\}$$

7.2.3 Funkcja wypukła



Rysunek 7.6: Wizualizacja definicji funkcji wypukłej.

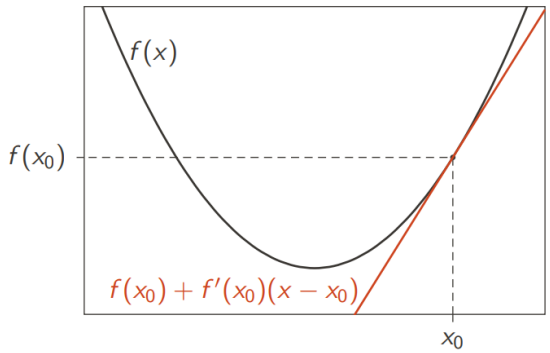
Definicja 7.5 — Funkcja wypukła. Funkcja f(x) jest wypukła, jeśli dla dowolnych dwóch punktów $\mathbf{x_1}$, $\mathbf{x_2}$ i dowolnego $\lambda \in [0,1]$ zachodzi:

$$f(\lambda x_1(1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda) f(x_2)$$

Jeśli nierówność jest ostra (<), to funkcja jest ściśle wypukła.
Zapis ten możemy zinterpretować geometrycznie: odcinek łączący dwa dowolne punkty na wykresie (cięciwa /odcinek siecznej) leży w całości powyżej lub na wykresie funkcji. Warto wspomnieć, że punkty leżące ponad wykresem funkcji wypukłej tworzą *zbiór wypukły*.

Jeśli funkcja f(x) jest różniczkowalna, to jest wypukła wtedy i tylko wtedy gdy dla dowolnych $\mathbf{x_0}$, $\mathbf{x_1}$ zachodzi:

$$f(x) \geq f(x_0) + \nabla f(x_0)^T (x - x_0)$$



Czyli styczna do wykresu leży zawsze pod wykresem.

Operacje zachowujące wypukłość (wybrane):

- złożenie funkcji wypukłej z niemalejącą funkcją wypukłą
- max
- kombinacja wypukła

Operacje niekoniecznie zachowujące wypukłość

- mnożenie, dzielenie
- przeciwny znak
- kombinacja liniowa/afiniczna
- złożenie funkcji

Ważne funkcje wypukłe:

- funkcja kwadratowa
- norma euklidesowa (i dowolna inna)
- funkcja wykładnicza
- -logarytm
- błąd zawiasowy (hinge loss)
- entropia krzyżowa (cross entropy, logloss)
- abs

7.2.4 Normy, metryki i dywergencje

Normy służą nam do oceny wielkości wektorów (cokolwiek). Jednak nie każda funkcja może być użyta jako norma.

Norma

Definicja 7.6 — Norma. Odwzorowanie $||.|| : X \to [0,\infty]$ jest normą jeśli spełnia następujące warunki:

1. $||x|| = 0 \Rightarrow x = 0$
2. $||\alpha x|| = |\alpha|||x||; \alpha \in \mathbb{R}$
3. $||x+y|| \leq ||x|| + ||y||$ (nierówność trójkąta)

Do najbardziej znanych norm należą:

- norma l_p: $||x||_p = \sqrt[p]{\sum_{i=1}^{n} |x_i|^p}$
  w szczególności:
- $l_1(x) = \sum_{i=1}^{n} |x_i|$ (norma Manhaattańska/taksówkowa),
- $l_2(x) = ||x||$ (norma Euklidesowa)
- $l_\infty(x) = \max_i(|x_i|)$ (norma Czebyszewa)

Metryka

Metryki służą do reprezentacji dystansu między dwoma wektorami.

Definicja 7.7 — Metryka. Odwzorowanie $d : XxX \to [0,\infty]$ nazywamy metryką jeśli dla każdego $x,y,z \in X$ spełnione są następujące warunki:

1. $d(x,y) \geq 0$
2. $d(x,y) = 0 \Leftrightarrow x = y$
3. $d(x,y) = d(y,x)$ (symetria)
4. $d(x,z) \leq d(x,y) + d(y,z)$ (nierówność trójkąta)

Na pierwszy rzut oka normy i metryki wydają się podobne. Nie jest to przypadek, gdyż każda norma indukuje metrykę (w drugą stronę to nie zachodzi).

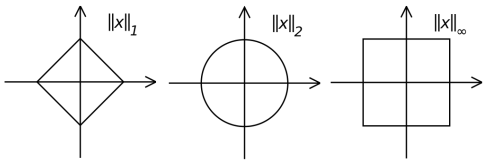
Definicja 7.8 — Metryka indukowana przez normę. Metryka d jest indukowana przez normę $||.||$ jeśli:

$$d(x,y) = ||x-y||$$

Najbardziej znane metryki są indukowane przez najbardziej znane normy, mianowicie metryka euklidesowa, dystans manhattański (odległość taksówkowa) czy dystans Czebyszewa.

Okrąg jednostkowy

Okrąg jednostkowy jest to zbiór wektorów, których norma jest równa jeden - alternatywnie: odległość od środka układu współrzędnych to 1. To jak będzie wyglądał okrąg jednostkowy zależy od normy jakiej użyjemy:



Dywergencja

Nie wchodząc w dalsze szczegóły chcieliśmy tu wspomnieć o dywergencjach, którą służą jako bardziej uogólniona definicja dystansu w stosunku do normy. Dywergencje nie muszą być symetryczne i nie muszą spełniać nierówności trójkąta. Dywergencja Kullbacka-Leiblera (KL divergence) jest najbardziej powszechnie używaną dywergencją i służy do oceny rozbieżności dwóch rozkładów prawdopodobieństwa. Jest to kluczowe w przypadku klasyfikacji - próbujemy doprawodzić do sytuacji gdzie rozkład prawdopodobieństwa naszych predykcji i danych mają jak najmniej rozbieżności (mała dywergencja). By dowiedzieć się więcej zachęcamy do zapoznania z [**?**].

7.2.5 Problemy wypukłe nieróżniczkowalne

ADVANCED

Niektóre algorytmy np. algorytm (stochastycznego) spadku wzdłuż gradientu wymaga, aby funkcja była różniczkowalna tj. aby było możliwe policzenie kierunku spadku $\nabla f(x)$. Okazuje się jednak, że algorytm ten można zastosować także do nieróżniczkowalnych funkcji wypukłych!

Przypomnijmy, że dla funkcji wypukłych styczna do wykresu leży zawsze całkowicie pod nim.

$$f(\mathsf{x}) \geq f(\mathsf{x}_0) + \nabla f(\mathsf{x}_0)^T(\mathsf{x} - \mathsf{x}_0)$$

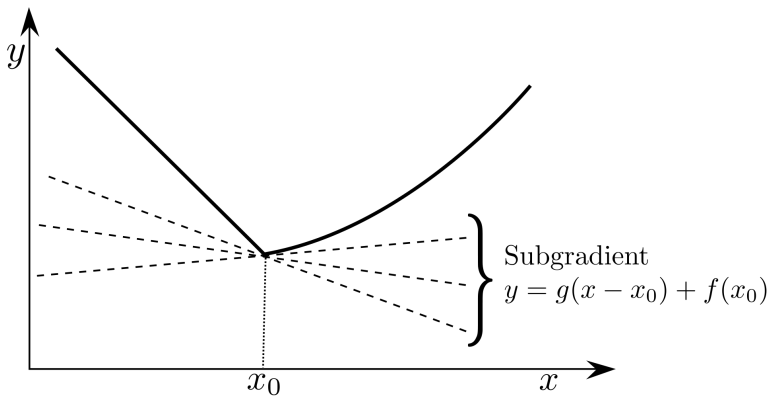
Funkcja wartości bezwzględnej jest nieróżniczkowalna w punkcie $x = 0$, ale... bez problemu znajdziemy wiele linii przechodzących przez ten punkt, aby spełniały powyższą równość! Ta obserwacja prowadzi nas do uogólnienia definicji pochodnej i zdefiniowania pod pochodnej.

Definicja 7.9 — Podpochodna. Podpochodną funkcji wypukłej nazywamy takie g, które spełnia warunek:

$$f(\mathsf{x}) \geq f(\mathsf{x}_0) + \mathsf{g}^T(\mathsf{x} - \mathsf{x}_0)$$

Zwróć uwagę, że jeżeli pochodna istnieje w danym punkcie funkcji wypukłej to automatycznie jest ona jej podpochodną. W punktach nieróżniczkowalnych może istnieć wiele wektorów (liczb w 1D) spełniających to równanie (jak w przykładzie z wartością bezwzględną lub na rysunku 7.7).

Używając podpochodnej modyfikacja algorytmu jest trywialna. W każdej z iteracji zamiast pochodnej wstawiamy jedną z podpochodnych (przypominam: tam gdzie funkcja



Rysunek 7.7: Wizualizacja definicji podpochodnej w punkcie nieróżniczkowalnym.

jest różniczkowalna istnieje tylko jedna podpochodna równa pochodnej, więc w takich punktach algorytm pozostaje bez zmian) i to tyle... działa :)

Literatura

Literatura powtórkowa

Metoda gradientów sprzężonych nie jest wcale łatwa w zrozumieniu – zachęcam zapoznanie się z bardzo eleganckim wyprowadzeniem na wykładzie [**?**]. Dla naprawdę opornych można przeczytać [**?**].

Literatura dla chętnych

W tym tygodniu zachęcamy do zapoznania się z algorytmami do automatycznego liczenia pochodnych. Wśród takich technik popularne jest pojęcie grafu obliczeń (ang. *computational graph*) i liczenie pochodnych na takim grafie. Opis grafów obliczeń wraz z algorytmem wstecznej propagacji można znaleźć np. na stronie `http://colah.github.io/posts/2015-08-Backprop/`