# Matematyczne podstway sieci neuronowych

Piotr Dyszewski

2026-06-08

# Spis treści

# Słowo wstępne

Notatki te powstają na potrzeby kursu o tym samym tytule, zaproponowanego w semestrze letnim roku akademickiego 2026/2027 w Instytucie Matematycznym Uniwersytetu Wrocławskiego.

Ich głównym celem jest zaznajomienie autora (a siłą rzeczy także czytelnika) z podstawowymi zasadami działania sieci neuronowych, które stanowią jeden z fundamentów dużych modeli językowych oraz modeli generatywnych. Nie chcemy traktować poszczególnych warstw sieci neuronowych jak czarnych skrzynek. Zamiast tego, poprzez analizę i implementację prostych, niskowymiarowych przykładów, będziemy starali się zrozumieć mechanizmy ich działania, a także ich mocne i słabe strony.

Lista źródeł, z których będę korzystał podczas przygotowywania tych notatek, ma charakter rozwojowy. Obecnie głównymi pozycjami są:

- *Mathematics of Neural Networks*, Bart Smets, https://arxiv.org/pdf/2403.04807
- *Alice's Adventures in a Differentiable Wonderland*, Simone Scardapane, https://arxiv.org/pdf/2404.17625
- *Neural Networks and Deep Learning*, Michael Nielsen, http://neuralnetworksanddeeplearning.com/

Notatki mają charakter roboczy i (jak wszystko we wstępnych fazach rozwoju) mogą zawierać błędy. Będę wdzięczny za zgłoszenie każdej usterki znalezionej w tekście.

Notatki są przygotowywane w Quarto, otwartym systemie do tworzenia publikacji naukowych i technicznych. Więcej informacji o Quarto można znaleźć na stronie https://quarto.org/.

# 1 Wstęp

W ciągu ostatnich kilkunastu lat uczenie maszynowe przeszło ogromny rozwój. Szczególną rolę odegrało w tym głębokie uczenie, oparte na sieciach neuronowych. Dzięki połączeniu prostych idei algorytmicznych, dużych zbiorów danych oraz rosnącej mocy obliczeniowej możliwe stało się rozwiązywanie zadań, które wcześniej były bardzo trudne, takich jak rozpoznawanie obrazów, przetwarzanie języka naturalnego, rozpoznawanie mowy czy analiza złożonych danych.

Uczenie maszynowe to nauka programowania komputerów w taki sposób, aby mogły uczyć się na podstawie danych. Oto nieco bardziej ogólna definicja:

> Uczenie maszynowe to dziedzina badań, która daje komputerom zdolność uczenia się bez bycia jawnie zaprogramowanymi.
>
> — Samuel (1959)

Sieci neuronowe są dziś jednym z podstawowych narzędzi uczenia maszynowego. Wykorzystuje się je zarówno w widocznych zastosowaniach, takich jak duże modele językowe i generatory obrazów, jak i w systemach działających w tle: wyszukiwarkach, systemach rekomendacyjnych, analizie obrazów medycznych, projektowaniu leków czy technologiach autonomicznych.

Uczenie maszynowe polega na budowaniu modeli, które poprawiają swoje działanie na podstawie danych. Zamiast ręcznie projektować algorytm rozwiązujący dane zadanie, wybieramy ogólny model i uczymy go na przykładach. Gdy modelem tym jest sieć neuronowa z wieloma warstwami, mówimy o głębokim uczeniu.

Mimo imponujących sukcesów praktycznych nadal nie rozumiemy w pełni, dlaczego głębokie sieci neuronowe działają tak dobrze. Szczególnie interesujące jest to, że duże modele potrafią nie tylko zapamiętywać dane treningowe, ale także generalizować, czyli poprawnie działać na nowych przykładach.

Celem tych notatek jest wprowadzenie w podstawowe idee sieci neuronowych i głębokiego uczenia. Będziemy analizować zarówno intuicję, jak i formalny opis matematyczny modeli, aby zrozumieć, jak proste komponenty - warstwy, funkcje aktywacji, parametry, funkcje straty i algorytmy optymalizacji - pozwalają budować systemy zdolne do rozwiązywania złożonych problemów. Aby w pełni zrozumieć zasady działania sieci neuronowych skupimy się na niskowymiarowych przykładach.

# 2 Uczenie nadzorowane

## 2.1 Uczenie maszynowe

Istnieje kilka sposobów poznawania otaczającego nas świata. Paradygmaty kilku z nich widać w metodach uczenia maszynowego.

Uczyć się możemy od nauczyciela, który pokazuje nam błędy w wypracowaniach czy rachunkach. Może być to też rodzic z małym dzieckiem, który opowiada mu otaczający go świat (na widok psa tłumaczy dziecku "to jest pies"). Mówimy wówczas o **uczeniu nadzorowanym**.

Możemy uczyć się sami. Po wysłuchaniu kilku utworów muzycznych lub obejrzeniu kilku filmów sami zaczynamy dostrzegać pewne powtarzające się motywy. Dziecko bawiące się samodzielnie samo zauważa, że okrągłe przedmioty się toczą. Są to przykłady **uczenia nienadzorowanego**. Jest to pewnego rodzaju porządkowanie doświadczenia. Umysł nie tylko biernie odbiera dane, ale sam szuka regularności i tworzy kategorie.

Możemy też uczyć się przez konsekwencje. Dziecko może raz dotknąć gorącego kubka, po czym więcej tak nie zrobi. Ogólniej człowiek uczy się ostrożności po bolesnym błędzie. Podczas tresury psy są wynagradzane przez swoich właścicieli przysmakami. Są to przykłady **uczenia przez wzmacnianie**.

W tym momencie można zapytać jeszcze co co znaczy nauczyć się czegoś. Oznacza to opanowanie pewnej umiejętności na tyle, by móc ją powtarzać w zmieniających się warunkach. Aby móc doprecyzować to stwierdzenie, będziemy skupiać się od tej pory na **uczeniu nadzorowanym**.

**Przykład 2.1.** Student na kursie algebry liniowej uczy się liczyć wyznaczniki macierzy $3 \times 3$. Jego celem jest wykazanie się tą umiejętnością na kolokwium. Oznacza to, że musi być w stanie policzyć wyznacznik dowolnej macierzy $3 \times 3$ nawet jeżeli nie widział jej wcześniej. Wyznacznik macierzy det to funkcja ze zbioru macierzy $M_{3\times 3}$ w zbiór liczb rzeczywistych $\mathbb{R}$. Matematycznie możemy powiedzieć, że w trakcie nauki student oblicza wartości wyznacznika $\det(m)$ dla macierzy $m$ pochodzących z pewnego podzbioru $A \subseteq M_{3\times 3}$ macierzy na których się uczy. W trakcie kolokwium student ma za zadanie wykazać się umiejętnością obliczenia wyznacznika macierzy spoza zbioru $A$.

W podobnym duchu będziemy rozumieli nauczenie się czegoś przez maszynę. Chcemy aby program komputerowy był w stanie obliczyć wartość funkcji $f\colon \mathcal{X} \to \mathcal{Y}$. Oczywiście w przypadku niewielkich dziedzin $X$ i łatwych odwzorowań $f$ możemy podać komputerowi jawny wzór funkcji

$f$. Wyzwanie pojawie się w momencie, w którym przestrzeń $X$ staje się duża, a interesująca nas funkcja $f$ zbyt złożona aby móc podać ją bezpośrednim wzorem.

**Przykład 2.2.** W trakcie nauki czytania pisma odręcznego dziecko uczy się rozpoznawać litery. Dziecko rozpoznaje literę alfabetu na podstawie obrazka o wymiarach $h \times w$. Matematycznie jest to funkcja z podzbioru czarno białych obrazków $X \subseteq \{0,1\}^{h \times w}$ (wykluczając pewne patologiczne przykłady) w alfabet $Y = \{a, b, \ldots, z\}$. Jeżeli $X$ składa się z czytelnych liter, to $f$ jest dobrze określoną funkcją, ponieważ dla każdego czytelnego obrazka $x \in X$ potrafimy jednoznacznie odczytać zapisaną na nim literę $f(x)$. Napisanie jednak $f$ wzorem ogólnym jest praktycznie niemożliwe.

Zauważmy, że Przykład 2.1 oraz Przykład 2.2 są fundamentalnie różne. W przypadku pierwszego stanieje jawna procedura postępowania. W drugim przypadku nie jesteśmy w stanie efektywnie wyznaczyć procedury. Okazuje się jednak, że jesteśmy w stanie

## 2.2 Problem uczenia nadzorowanego

Uczenie nadzorowane to paradygmat uczenia maszynowego, w którym dostępne dane składają się z par wejść oraz znanych, poprawnych wyjść. Nieznane jest natomiast to, jak dokładnie działa odwzorowanie między tymi wejściami i wyjściami. Celem jest wywnioskowanie, na podstawie dostępnych danych, ogólnej struktury tego odwzorowania, w nadziei, że będzie ono uogólniać się na niewidziane wcześniej sytuacje. Formalnie problem ten możemy sformułować następująco. Szukana funkcja $f\colon \mathcal{X} \to \mathcal{Y}$ między przestrzeniami $\mathcal{X}$ oraz $\mathcal{Y}$. Znamy wartości funkcji $f$ na pewnym podzbiorze $\mathcal{A} \subseteq \mathcal{X}$. Matematycznie należy więc znaleźć dobre rozszerzenie funkcji $f$ z $\mathcal{A}$ do całej przestrzeni $\mathcal{X}$.

W praktyce dany mamy zbiór danych złożony z $N$ próbek:

$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N},$$

przy czym

$$f(x_i) = y_i,$$

dla $i \leq N$.

Przestrzeń $\mathcal{X}$ nazywana jest również **przestrzenią cech**, a jej elementy określa się mianem wektorów cech. Przestrzeń $\mathcal{Y}$ nazywana jest również **przestrzenią etykiet**, a jej elementy określa się mianem etykiet. Okazuje się, że w kontekście nauczania maszynowego zamiast uczyć komputer zwracać konkretną wartość ze zbioru etykiet $\mathcal{Y}$ wygodniej jest nauczyć go zwracać rozkład prawdopodobieństwa na $\mathcal{Y}$. W ten sposób otrzymujemy więcej informacji, mamy kontrolę nad pewnością wyboru oraz możemy kontrolować przypadki brzegowe.

**Przykład 2.3.** Niech $\mathcal{X}$ będzie przestrzenią wszystkich obrazów kotów i psów. Obraz możemy zapisać jako wektor:

$$\mathcal{X} = [0,1]^{3\times H\times W},$$

gdzie $H$ to wysokość obrazu, $W$ to szerokość, a 3 bierze się z trzech kanałów kolorów: RGB. Klasyfikator $f$ ma przypisać obraz do jednej z dwóch klas:

$$\{\text{pies}, \text{kot}\}.$$

Zamiast jednak zwracać od razu etykietę pies albo kot, model często zwraca wektor prawdopodobieństw:

$$f(x) = (p_{\text{pies}}, p_{\text{kot}}).$$

Na przykład $f(x) = (0.87, 0.13)$ oznacza, że model ocenia obraz jako psa z prawdopodobieństwem 0.87, a jako kota z prawdopodobieństwem 0.13. Decyzję końcową otrzymujemy przez wybór największego prawdopodobieństwa:

$$y = \arg\max f(x).$$

W naszym przykładzie $y =$ pies. Wektor prawdopodobieństw zawiera informację o pewności modelu. Przykładowo oba wyniki prowadzą do tej samej etykiety:

$$(0.51, 0.49) \mapsto \text{pies},$$

oraz

$$(0.99, 0.01) \mapsto \text{pies}.$$

Pierwszy przypadek oznacza, że model jest prawie niezdecydowany, a drugi, że jest bardzo pewny swojej decyzji.

Ta informacja jest przydatna, ponieważ można ustalić próg akceptacji decyzji. Na przykład jeżeli

$$\max\{p_{\text{pies}}, p_{\text{kot}}\} < 0.8$$

to model nie klasyfikuje automatycznie. Wtedy wynik $(0.51, 0.49)$ można potraktować jako „nie wiem”, a wynik: $(0.99, 0.01)$ jako pewną klasyfikację.

Ogólne podejście do rozwiązania tego problemu składa się z trzech głównych kroków.

1. Wybieramy model

   $$F : \mathcal{X} \times \mathcal{W} \to \mathcal{Y}$$

   parametryzowany przez parametr z przestrzeni $\mathcal{W}$. Litera $\mathcal{W}$ pochodzi od angielskiego słowa *weights*, ponieważ tak często określa się parametry w sieciach neuronowych.

2. Musimy określić ilościowo jakość wyjścia modelu, dlatego wybieramy **funkcję straty**

$$\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R},$$

gdzie $\ell(y_1, y_2)$ określa, jak różne są $y_1$ oraz $y_2$.

3. Na podstawie zbioru danych $\mathcal{D}$ oraz funkcji straty wybieramy $w \in W$ tak, aby

$$F(\cdot, w)$$

było możliwie najlepszym przybliżeniem funkcji docelowej $f$.

To wysokopoziomowe podejście pozostawia jednak kilka otwartych pytań.

- Jak wybrać model?
- Jak wybrać funkcję straty?
- Jak przeprowadzić optymalizację?

Na żadne z tych pytań nie ma kanonicznych odpowiedzi. W dużej mierze rozwiązuje się je metodą prób i błędów oraz za pomocą heurystyk. Niezależnie od tego wybory te mają duży wpływ na końcowy wynik, mimo że nie są bezpośrednio uzasadnione przez dostępne dane ani przez jakieś podstawowe zasady. Zbiorczy zestaw założeń, które przyjmujemy, aby móc zająć się problemem, nazywa się w uczeniu maszynowym **biasem indukcyjnym** (*inductive bias*).

Na potrzeby tego kursu jako modele wybierzemy oczywiście sieci neuronowe; omówimy je dokładniej później. Drugą rzeczą, którą musimy zrobić, jest wybór funkcji straty. Funkcja straty działa podobnie do metryki, ale jest mniej restrykcyjna.

- Zwykle (choć nie zawsze) mamy $\ell(y) \geq 0$ dla każdego $y \in \mathcal{Y}$.
- Ponieważ chcemy minimalizować stratę, minimum powinno istnieć, tak aby wyrażenie

$$\min_{y \in Y} \ell(y_0, y)$$

było dobrze określone dla każdego $y_0 \in \mathcal{Y}$.

- Funkcja powinna być różniczkowalna, ponieważ chcemy stosować metody gradientowe.
- Własności metryki, takie jak symetria czy nierówność trójkąta, nie muszą być spełnione.

Po wybraniu modelu i funkcji straty przechodzimy do optymalizacji, czyli pytania: jaki jest „najlepszy” wybór parametru $w \in \mathcal{W}$? Najprostsza, choć niekoniecznie preferowana, odpowiedź brzmi: należy zminimalizować stratę na zbiorze danych $\mathcal{D} = \{(x_k, y_k)\}_{k \leq N}$, tzn. znaleźć

$$w^* = \arg\min_{w \in W} \sum_{i=1}^{N} \ell(F(x_i; w), y_i).$$

**Przykład 2.4.** Niech $w = (w_1, \dots, w_n)$. Rozważmy model

$$F(x; w) = \sum_{j=1}^{n} w_j \varphi_j(x)$$

dla pewnych funkcji bazowych $\{\varphi_j\}_{j \le n}$. Przy funkcji straty

$$\ell(y_1, y_2) = |y_1 - y_2|^2$$

optymalizacja sprowadza się do metody najmniejszych kwadratów.

**Przykład 2.5.** Załóżmy, że mierzymy wysokość piłki w kolejnych chwilach czasu. Otrzymujemy dane pomiarowe:

$$\begin{aligned}\mathcal{D} =&\{(0, 1.0), (1, 3.1), (2, 4.0),\\ &(3, 3.2), (4, 1.1)\}.\end{aligned}$$

Pierwsza współrzędna oznacza czas $t$, a druga wysokość piłki $y$. Wiemy, że torem lotu piłki jest parabola (to jest nasz inductive bias!).

Jest to przykład polynomial fitting, czyli dopasowania wielomianu do danych. Możemy zapisać ten model w postaci:

$$\begin{aligned}F(t; w) &= w_0 + w_1 t + w_2 t^2\\ &= \sum_{j=0}^{2} w_j \varphi_j(t),\end{aligned}$$

gdzie funkcje bazowe to:

$$\varphi_0(t) = 1, \qquad \varphi_1(t) = t,$$
$$\varphi_2(t) = t^2.$$

Parametry $w_0, w_1, w_2$ nie są znane z góry. Model uczy się ich z danych, dobierając je tak, aby przewidywana wysokość $F(t_i; w)$ była możliwie blisko zmierzonej wysokości $y_i$.

W naszym przykładzie optymalizacja względem $w = (w_1, w_2, w_3)$ sprowadza się do minimalizacji

$$L(w) = \sum_{i=1}^{m} \left(w_0 + w_1 t_i + w_2 t_i^2 - y_i\right)^2. \tag{2.1}$$

Im mniejsza wartość $L$, tym lepiej $F(\cdot, w)$ opisuje tor lotu piłki. Zauważmy, że matematycznie $L$ jest wielomianem kwadratowym trzech zmiennych. Dokładna optymalizacja sprowadza się do algebry liniowej.

Dane zapisujemy w macierzy cech:

$$\Phi = \begin{bmatrix} 1 & t_1 & t_1^2 \\ 1 & t_2 & t_2^2 \\ \vdots & \vdots & \vdots \\ 1 & t_m & t_m^2 \end{bmatrix}. \tag{2.2}$$

Wektor wag to:

$$w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}, \tag{2.3}$$

a wektor obserwacji to:

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}. \tag{2.4}$$

Predykcje danych treningowych to

$$\Phi w = \begin{bmatrix} F(t_1, w) \\ F(t_2, w) \\ \vdots \\ F(t_m, w) \end{bmatrix}. \tag{2.5}$$

Mamy

$$L(w) = \|\Phi w - y\|^2. \tag{2.6}$$

Problem polega na tym, że zwykle układ:

$$\Phi w = y \tag{2.7}$$

nie ma dokładnego rozwiązania. Oznacza to, że nie istnieje taka parabola, która przechodzi idealnie przez wszystkie punkty. Szukamy więc najlepszego przybliżenia.

Równania normalne mają postać:

$$\Phi^T \Phi w = \Phi^T y. \tag{2.8}$$

Ich rozwiązanie daje takie wagi $w$, dla których błąd średniokwadratowy jest najmniejszy.

### 2.2.1 Skąd się one biorą?

Niech reszta, czyli błąd modelu, będzie dana przez:

$$r = y - \Phi w. \tag{2.9}$$

Najlepsze dopasowanie oznacza, że reszta $r$ nie powinna już zawierać informacji, którą da się wyjaśnić za pomocą kolumn macierzy $\Phi$. Innymi słowy, $r$ ma być prostopadła do każdej kolumny macierzy $\Phi$.

Zapisujemy to jako:

$$\Phi^T r = 0. \tag{2.10}$$

Podstawiając $r = y - \Phi w$, dostajemy:

$$\Phi^T (y - \Phi w) = 0. \tag{2.11}$$

Po przekształceniu:

$$\Phi^T y - \Phi^T \Phi w = 0. \tag{2.12}$$

Stąd:

$$\Phi^T \Phi w = \Phi^T y. \tag{2.13}$$

To są właśnie równania normalne.

### 2.2.2 Interpretacja geometryczna

Wektor $y$ reprezentuje prawdziwe pomiary. Wektor $\Phi w$ reprezentuje predykcje modelu. Wszystkie możliwe predykcje $\Phi w$ tworzą pewną podprzestrzeń.

Szukamy punktu $\Phi w$ w tej podprzestrzeni, który leży najbliżej $y$. Jest to rzut prostopadły wektora $y$ na przestrzeń generowaną przez kolumny macierzy $\Phi$.

Dlatego błąd:

$$r = y - \Phi w \tag{2.14}$$

musi być prostopadły do tej przestrzeni.

### 2.2.3 W Pythonie

```
import numpy as np

t = np.array([0, 1, 2, 3, 4], dtype=float)
y = np.array([1.0, 3.1, 4.0, 3.2, 1.1], dtype=float)

Phi = np.column_stack([
    np.ones_like(t),
    t,
    t**2
])

A = Phi.T @ Phi
b = Phi.T @ y

w = np.linalg.solve(A, b)

print(w)
```

```
[ 0.97714286  2.91571429 -0.72142857]
```

W tym kodzie rozwiązujemy układ:

$$Aw = b, \tag{2.15}$$

gdzie:

$$A = \Phi^T \Phi, \qquad b = \Phi^T y. \tag{2.16}$$

Czyli zamiast iteracyjnie poprawiać wagi, dostajemy je bezpośrednio przez rozwiązanie jednego układu równań liniowych.

konieiiii

```
import numpy as np
import matplotlib.pyplot as plt

# Dane pomiarowe
t = np.array([0, 1, 2, 3, 4], dtype=float)
```

```
y = np.array([1.0, 3.1, 4.0, 3.2, 1.1], dtype=float)

def model(t, w0, w1, w2):
    return w0 + w1 * t + w2 * t**2

def loss(w0, w1, w2):
    y_pred = model(t, w0, w1, w2)
    return np.mean((y_pred - y)**2)
```

Najpierw możemy sprawdzić kilka przykładowych parabol.

```
candidates = [
    (1.0, 1.0, 0.0),
    (1.0, 2.0, -0.3),
    (1.0, 3.0, -0.7),
    (0.5, 3.0, -0.7)
]

for w0, w1, w2 in candidates:
    print(
        f"w0={w0:4.1f}, w1={w1:4.1f}, w2={w2:5.1f}, "
        f"L={loss(w0, w1, w2):.4f}"
    )
```

```
w0= 1.0, w1= 1.0, w2=  0.0, L=3.6120
w0= 1.0, w1= 2.0, w2= -0.3, L=2.2040
w0= 1.0, w1= 3.0, w2= -0.7, L=0.1640
w0= 0.5, w1= 3.0, w2= -0.7, L=0.0940
```

Teraz zamiast zgadywać jedną parabolę, przeszukamy wiele możliwych wartości wag. To jest bardzo prosty sposób minimalizacji: sprawdzamy dużo kombinacji i wybieramy najlepszą.

```
w0_values = np.linspace(0, 2, 41)
w1_values = np.linspace(1, 4, 61)
w2_values = np.linspace(-1.5, 0, 61)

best_loss = np.inf
best_weights = None

for w0 in w0_values:
    for w1 in w1_values:
```

```
        for w2 in w2_values:
            current_loss = loss(w0, w1, w2)

            if current_loss < best_loss:
                best_loss = current_loss
                best_weights = (w0, w1, w2)

w0_best, w1_best, w2_best = best_weights

print(f"Najlepsze znalezione wagi:")
print(f"w0 = {w0_best:.4f}")
print(f"w1 = {w1_best:.4f}")
print(f"w2 = {w2_best:.4f}")
print(f"L  = {best_loss:.6f}")
```

```
Najlepsze znalezione wagi:
w0 = 0.9500
w1 = 2.9500
w2 = -0.7250
L  = 0.003750
```

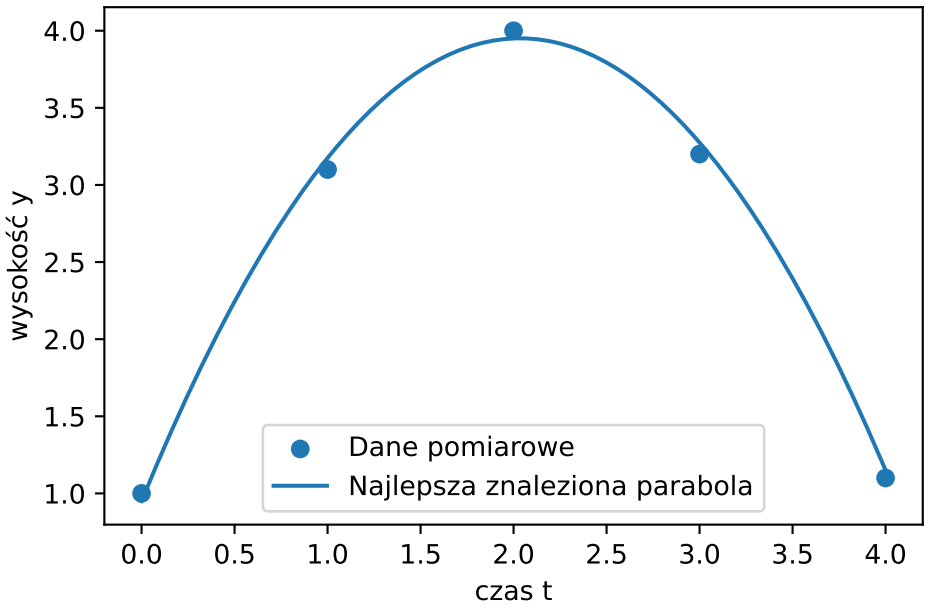
Otrzymaliśmy więc przybliżony model:

$$F(t; w) = w_0 + w_1 t + w_2 t^2. \tag{2.17}$$

Możemy go teraz narysować razem z danymi.

```
t_grid = np.linspace(0, 4, 200)
y_grid = model(t_grid, w0_best, w1_best, w2_best)

plt.scatter(t, y, label="Dane pomiarowe")
plt.plot(t_grid, y_grid, label="Najlepsza znaleziona parabola")
plt.xlabel("czas t")
plt.ylabel("wysokość y")
plt.legend()
plt.show()
```




W tym podejściu nie używamy jeszcze gradientu. Minimalizacja polega po prostu na tym, że traktujemy $w_0, w_1, w_2$ jak trzy pokrętła. Dla wielu ustawień tych pokręteł liczymy błąd $L$, a potem wybieramy ustawienie z najmniejszym błędem.

Ten sposób nie jest wydajny dla dużych modeli, ale dobrze pokazuje podstawową ideę uczenia: szukamy takich parametrów $w$, dla których model najlepiej pasuje do danych.

Po dopasowaniu otrzymujemy parabolę, która możliwie dobrze opisuje tor lotu piłki. Ważne jest to, że model nie musi przechodzić dokładnie przez każdy punkt pomiarowy, ponieważ dane mogą być zaszumione.

W tym przykładzie:

- $t$ oznacza czas,
- $y$ oznacza zmierzoną wysokość piłki,
- $F(t; w)$ oznacza przewidywaną wysokość,
- $w_0, w_1, w_2$ są parametrami modelu,
- funkcje $1, t, t^2$ tworzą bazę wielomianową.

„Najlepszy” parametr $w \in W$ niekoniecznie jest tym, który minimalizuje stratę na zbiorze danych. W każdym problemie optymalizacyjnym częstym problemem jest **przeuczenie** (*overfitting*). Prowadzi nas to do **regularyzacji**.

Techniki regularyzacji dla sieci neuronowych omówimy szerzej później. Na razie wspomnimy tylko, że **regularyzacja parametrów** jest powszechną techniką znaną z regresji, często stosowaną również w sieciach neuronowych. Ten typ regularyzacji polega na dodaniu do straty

na danych składnika karzącego, który zniechęca parametry do przyjmowania niepożądanych wartości, na przykład zbyt dużych. Zmodyfikowany problem optymalizacyjny ma postać

$$w^* = \arg\min_{w \in W} \sum_{i=1}^{N} \ell(F(x_i; w), y_i) + \lambda C(w),$$

gdzie $\lambda > 0$ oraz

$$C : W \to \mathbb{R}^+$$

jest funkcją karzącą w pewien sposób złożoność modelu.

**Przykład 1.3** *(regularyzacja Tichonowa).*
Niech $W = \mathbb{R}^n$ oraz

$$C(w) = \|w\|^2.$$

W kontekście sieci neuronowych ten typ regularyzacji parametrów nazywany jest również **zanikiem wag** (*weight decay*).

## 2.3 Regresja i klasyfikacja

Uczenie nadzorowane powinno już brzmieć znajomo. Rzeczywiście, jeśli

$$X = \mathbb{R}^n \quad \text{oraz} \quad Y = \mathbb{R}^m,$$

to mamy do czynienia z **regresją**.

Regresja stanowi dużą część uczenia nadzorowanego, ale pojęcie uczenia nadzorowanego formułuje się ogólniej, tak aby obejmowało również inne zadania, takie jak **klasyfikacja**.

Klasyfikacja to w uczeniu maszynowym określenie na logistyczną regresję wieloklasową, w której

$$X = \mathbb{R}^n$$

i próbujemy przypisać każdy punkt $x \in X$ do jednej z $m$ klas. Numeryczne wyjście dla tego typu problemu jest zwykle dyskretnym rozkładem prawdopodobieństwa na $m$ klasach:

$$Y = \left\{ (y_1, \dots, y_m) \in [0,1]^m \;\middle|\; \sum_{i=1}^{m} y_i = 1 \right\}.$$

Przykład 1.1 jest właśnie tego typu.

W teorii uczenia statystycznego zakłada się, że próbki danych $(x_i, y_i)$ są losowane niezależnie i z tego samego rozkładu, czyli i.i.d., z pewnego rozkładu prawdopodobieństwa $\mu$ na $X \times Y$. *Zastanów się, dlaczego jest to dość duże założenie.*

Interesuje nas wtedy minimalizacja **ryzyka populacyjnego**:

$$R(w) := \mathbb{E}_{(x,y)\sim\mu} \left[ \ell(F(x;w), y) \right] := \int_{X\times Y} \ell(F(x;w), y)\, d\mu(x,y).$$

Celem byłoby znalezienie zestawu parametrów, który minimalizuje to ryzyko populacyjne:

$$w^* = \arg\min_{w\in W} R(w).$$

W rzeczywistości jednak nie znamy $\mu$, więc nie możemy nawet obliczyć ryzyka populacyjnego, nie mówiąc już o jego minimalizacji. Robimy zatem najlepszą dostępną rzecz: minimalizujemy **ryzyko empiryczne**:

$$\widehat{R}(w) := \frac{1}{N} \sum_{i=1}^{N} \ell(F(x_i;w), y_i).$$

Zestaw parametrów minimalizujący ryzyko empiryczne nazywa się **minimalizatorem empirycznym**:

$$\widehat{w} = \arg\min_{w\in W} \widehat{R}(w),$$

co w kontekście uczenia nadzorowanego jest tym samym, co minimalizacja straty na zbiorze danych.

Gdy dodamy składnik regularyzacyjny, tak jak wcześniej, otrzymujemy tak zwaną **minimalizację ryzyka strukturalnego**:

$$\widehat{w} = \arg\min_{w\in W} \widehat{R}(w) + \lambda C(w).$$

Teoria uczenia statystycznego zajmuje się następnie badaniem takich zagadnień jak oszacowania różnicy

$$\widehat{R}(\widehat{w}) - R(w^*).$$

# 3 Model neuronu i funkcja aktywacji

# 4 Płytkie uczenie

# 5 Stochastic Gradient Descent

# 6 Training

# 7 Głębokie sieci neuronowe

# 8 Initialization

# 9 Convolutional Neural Networks

# 10 Automatic Differentiation and Backpropagation

# 11 Adaptive Learning Rate Algorithms

# 12 Redukcja Wymiaru

PCA Krótko: po co redukujemy wymiar? SNE / t-SNE jako metoda wizualizacji UMAP jako metoda wizualizacji

# 13 Autoenkodery i encodery

# 14 Diffusion Models

# 15 Summary

In summary, this book has no content whatsoever.

# References

Samuel, Arthur L. 1959. „Some studies in machine learning using the game of checkers". *IBM Journal of research and development* 3 (3): 210–29.