Rozdział 2

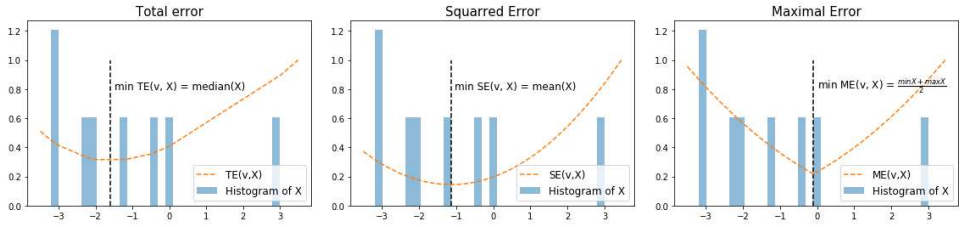
Wprowadzenie do uczenia maszynowego

Definicja modelu uczenia maszynowego rozpoczyna się od zdefiniowania funkcji kosztu, czyli kryterium jakości znalezionego rozwiązania. Rozwiązania poszukuje się poprzez minimalizację tej funkcji. Dla prostych funkcji rozwiązanie możemy znaleźć analitycznie, co jest najbardziej efektywne. Jednak dla typowych przykładów, konieczne jest zastosowanie numerycznych schematów minimalizacji.
W poniższym rozdziale pokażemy jak definicja funkcji kosztu wpływa na uzyskane rozwiązanie. Następnie przedstawimy dwa typowe schematy minimalizacji – iteracyjną oraz gradientową.

2.1 Analiza modelu i funkcja kosztu

Aby przedstawić jak definicja funkcji kosztu wpływa na jakość rozwiązania, posłużymy się prostym przykładem w sytuacji skalarnej (jednowymiarowej). Przypuśćmy, że chcemy opisać (dokonać reprezentacji) zbioru danych $X \subset \mathbb{R}$ za pomocą jednego punktu w. Możemy traktować to jako problem kompresji. Aby rozwiązać problem konieczne jest doprecyzowanie jak będziemy oceniać jakość rozwiązania (przybliżenia). Funkcję oceny danego wyniku (propozycji rozwiązania) nazywamy funkcją kosztu.

Funkcja kosztu Chcemy wybrać taki punkt $w \in \mathbb{R}$, żeby błąd zastąpienia wszystkich elementów X przez w był najmniejszy. Pokażemy, że wynik zależy od tego, jak zdefiniujemy błąd. Rozpatrzymy trzy możliwości, które odpowiadają różnym sposobom



Rysunek 2.1: Porównanie trzech funkcji kosztu: total error, squarred-error, maximal error dla danych zobrazowanych histogramem.

mierzenie błędu:

$$\begin{aligned} \mathrm{TE}(X;w) &= \sum_i |x_i - w| \\ &\quad \text{total error - sumaryczny błąd } l_1, \\ \mathrm{SE}(X;w) &= \sum_i |x_i - w|^2 \\ &\quad \text{squarred-error - błąd kwadratowy } l_2, \\ \mathrm{ME}(X;w) &= \max_i |x_i - w| \\ &\quad \text{maximal error - największy błąd } l_\infty. \end{aligned}$$

Powyższe błędy różnią się rodzajem użytej normy: l_1, l_2, l_∞. Podkreślmy, że żadna z powyższych miar błędu nie jest lepsza ani gorsza od pozostałych, choć każda funkcja ma inne minimum.

Punkt, który minimalizuje zadane kryterium jest rozwiązaniem problemu optymalizacyjnego. Funkcja kosztu jest podstawą definicji modelu uczenia maszynowego i oznacza funkcję której minimalizacja prowadzi do rozwiązania interesującego nas zadania. Pokażemy poniżej, że każdy z powyższych błędów prowadzi do innego rozwiązania.

Sumaryczny błąd Minimum funkcji TE jest mediana zbioru danych. Aby to pokazać, załóżmy, że $(x_i)_{i=1}^n \subset \mathbb{R}$ jest posortowany, to znaczy $x_1 \leq x_2 \leq \cdots \leq x_{n-1} \leq x_n$, i policzmy pochodną $\mathrm{TE}(w) = \sum_{i=1}^n |x_i - w|$. Dla $w \in (x_i, x_{i+1})$, mamy:

$$\begin{aligned} \mathrm{TE}'(w) &= \mathrm{card}\{i : x_i < w\} - \mathrm{card}\{i : x_i > w\} \\ &= \mathrm{card}\{i : x_i < w\} - (n - \mathrm{card}\{i : x_i < w\}) = 2i - n, \end{aligned}$$

gdzie card oznacza liczność zbioru.

W związku z powyższym funkcja TE ma następujące własności:

- jeśli n jest parzyste, to TE silnie maleje na przedziale $(-\infty, x_{n/2}]$; jest stała na przedziale $[x_{n/2}, x_{n/2+1}]$; silnie rośnie na przedziale $[x_{n/2+1}, \infty)$.
- jeśli n jest nieparzyste, to TE silnie maleje na przedziale $(-\infty, x_{(n+1)/2}]$; silnie rośnie na przedziale $[x_{(n+1)/2}, \infty)$.

W konsekwencji TE osiąga minimum w punkcie

- $x_{(n+1)/2}$, jeżeli n jest nieparzyste,

- dowolnym $w \in [x_{n/2}, x_{n/2+1}]$, jeżeli n jest parzyste.

Powyższa wartość jest medianą ciągu $x_1, \ldots, x_n$, czyli taką liczbą, że co najwyżej połowa populacji jest mniejsza od rozważanej wartości i co najwyżej połowa jest większa (patrz rysunek 2.1). Istotną wadą błędu TE jest brak możliwości łatwego wyliczenia mediany, czyli minimum.

Błąd średniokwadratowy Przez $\mathrm{mean}(X)$ rozumiemy średnią zbioru X, czyli $\mathrm{mean}(X) = \frac{1}{n}\sum_{i=1}^{n} x_i$. Wówczas błąd kwadratowy można przedstawić jako:

$$\begin{aligned}\sum_i |x_i - w|^2 = \sum_i (x_i - w)^2 &= \sum_i x_i^2 - 2nw\mathrm{mean}(X) + nw^2 \\ &= n|w - \mathrm{mean}(X)|^2 + \sum_i x_i^2 - n\mathrm{mean}(X)^2.\end{aligned} \tag{2.1}$$

Z powyższej równości wynika, że najmniejsza wartość SE jest realizowana przez średnią z danych (patrz rysunek 2.1). Z jednej strony średnia jest łatwa do wyliczenia, ale z drugiej posiada pewne wady:

- jest bardzo czuła na błędy (pojawienie się outliersów, czyli potencjalnie błędnych danych spoza rozkładu),
- zwraca zazwyczaj wynik, który może nie być reprezentowany w zbiorze danych, ponieważ $\mathrm{mean}(X)$ może nie należeć do X.

Błąd maksymalny W ostatnim przypadku szukamy takiego elementu, który minimalizuje błąd maksymalny:

$$\mathrm{ME}(X; w) = \max_i |x_i - w| = \max(\max x_i - w, w - \min_i x_i).$$

Jak widać optymalne rozwiązanie dane jest przez

$$w = \frac{\min X + \max X}{2},$$

które jest po prostu środkiem przedziału (zakresu), w którym mieszczą się dane (patrz rysunek 2.1).

Podsumowanie Chcielibyśmy podkreślić, że pomimo tego, że wszystkie rozważane funkcje kosztu stanowią poprawne doprecyzowanie postawionego problemu, prowadzą one do istotnie różnych rozwiązań. Co więcej żadnego z tych rozwiązań nie można określić jako lepszego od pozostałych – w konkretnych typach zadań używa się każdej z wymienionych funkcji (średnia, mediana, środek zakresu). W konsekwencji dobór najlepszej funkcji kosztu może zależeć od tego, w jakim kontekście będziemy chcieli używać naszego rozwiązania. Istotne znaczenie wyboru funkcji kosztu ma również umiejętność łatwego znajdywania minimum.

2.2 Minimalizacja funkcji kosztu

W poprzednim rozdziale rozważyliśmy najprostszy przypadek funkcji kosztu, dla których możemy wyliczyć jawny wzór na minimum. Dla bardziej skomplikowanych funkcji zazwyczaj nie istnieje jawny wzór. Zaobserwujmy to dla przykładu na uogólnieniu problemu z poprzedniej sekcji na przypadek wyżej-wymiarowy. Wtedy dla zbioru danych $X \subset \mathbb{R}^D$ szukamy w, który minimalizuje:

$$\begin{aligned} \mathrm{TE}(X;w) &= \sum_i \|x_i - w\| \\ &\quad \text{total error - sumaryczny błąd } l_1, \\ \mathrm{SE}(X;w) &= \sum_i \|x_i - w\|^2 \\ &\quad \text{squarred-error - błąd kwadratowy } l_2, \\ \mathrm{ME}(X;w) &= \max_i \|x_i - w\| \\ &\quad \text{maximal error - największy błąd } l_\infty. \end{aligned}$$

Jak łatwo sprawdzić jedynie rozumowanie dotyczące błędu kwadratowego da się powtórzyć w sytuacji wyżej-wymiarowej, i wtedy rozwiązaniem też jest średnia ze zbioru danych:

$$\operatorname*{argmin}_w \mathrm{SE}(X;w) = \mathrm{mean}X.$$

Wynika to z wektorowego odpowiednika wzoru (2.1)

$$\mathrm{SE}(X;w) = n\|w - \mathrm{mean}(X)\|^2 + \sum_i \|x_i\|^2 - n\|\mathrm{mean}X\|^2.$$

Zarówno minimalizacja sumarycznego błędu jak i największego (który jest równoważny znalezieniu kuli o najmniejszym promieniu zawierającej zbiór danych) nie mają jawnych analitycznych wzorów na rozwiązanie. W związku z tym w nauczaniu maszynowym został położony większy nacisk na minimalizację bardziej ogólnych klas funkcji niż te, dla których potrafimy znaleźć jawne rozwiązanie.

Omówimy dwa najczęściej spotykane podejścia minimalizacji stosowane w uczeniu maszynowym. Jeśli funkcja nie jest wypukła, to nie gwarantują one jednak osiągnięcia minimum globalnego, a jedynie lokalne.

Minimalizacja iteracyjna To podejście znajduje zwykle dobre minimum przy niewielkiej ilości wykonanych iteracji, choć nie dla każdej funkcji kosztu możemy je stosować. W tym modelu zakładamy, że mając dany punkt v_i potrafimy za pomocą wzoru znaleźć taki punkt v_{i+1}, że

$$f(v_{i+1}) \leq f(v_i),$$

gdzie $f : \mathbb{R}^D \to \mathbb{R}$ jest funkcją kosztu. Zazwyczaj chcemy też by $f(v_{i+1}) < f(v_i)$ o ile v_i nie jest lokalnym minimum funkcji f. Wtedy ogólny algorytm minimalizacyjny wygląda następująco: wybieramy losowo punkt v_0 i powtarzamy procedurę iteracyjną dopóki nie ustabilizuje się wartość $f(v_i)$.

Pokażemy teraz jedną z często stosowanych wersji podejścia iteracyjnego (będzie ona później użyta do optymalizacji funkcji kosztu k-means). Zakładamy, że

$f = f(x, y)$ jest dwuparametryczną funkcją kosztu i potrafimy znaleźć analitycznym wzorem minimum funkcji f względem każdej współrzędnej z osobna, czyli

$$\phi(x) = \operatorname*{argmin}_y f(x, y) \text{ oraz } \psi(y) = \operatorname*{argmin}_x f(x, y).$$

Startując od ustalonego punktu $v_i = (x_i, y_i)$ chcemy zbudować v_{i+1}. W tym celu dokonujemy minimalizacji najpierw po współrzędnej y (dostając punkt (x_i, y_{i+1})), a następnie względem współrzędnej x:

$$v_i = (x_i, y_i) \to (x_i, \phi(x_i)) \to (\psi(\phi(x_i)), \phi(x_i)) = v_{i+1}.$$

Finalnie:

$$v_{i+1} = (\psi(\phi(x_i)), \phi(x_i)) \text{ dla } v_i = (x_i, y_i).$$

Poniżej przedstawiamy przykład zastosowania naprzemiennej minimalizacji iteracyjnej.

Przykład 2.1. Przypuśćmy, że chcemy zminimalizować funkcję:

$$f(x, y) = |x - y| + |x - 1| + |y|.$$

Zauważmy, że dla dowolnego a, b mamy:

$$\operatorname*{argmin}_r \{|r - a| + |r - b|\} = \frac{a + b}{2},$$

gdyż powyższa funkcja osiąga minimum na odcinku $[a, b]$. W takim razie otrzymujemy:

$$\phi(x) = x/2 \text{ oraz } \psi(y) = (y + 1)/2,$$

Możemy zatem łatwo i efektywnie wykonać procedurę iteracyjną opisaną powyżej. Efekt minimalizacji jest zobrazowany na rysunku 2.2 (lewa strona).

Minimalizacja gradientowa (GD=gradient descent) w przypadku skalarnym Najbardziej rozpowszechnionym sposobem optymalizacji jest minimalizacja gradientowa. W celu zilustrowania tej procedury skupimy się najpierw na przypadku skalarnym (funkcja jednej zmiennej), gdzie $f : \mathbb{R} \to \mathbb{R}$. Dodatkowo zakładamy, że funkcja f jest dana na tyle skomplikowanym wzorem, że nie potrafimy stworzyć iteracyjnego schematu minimalizującego ani wyliczyć jawnego wzoru na rozwiązania.

Zauważmy, że lokalnie funkcję możemy przybliżyć funkcją liniową zadaną przez pochodną:

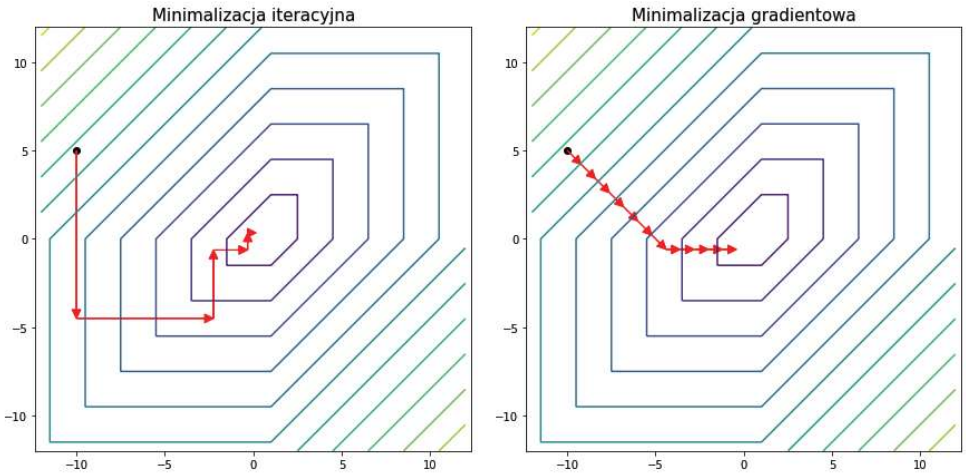
$$f(x + \Delta x) \approx f(x) + f'(x)\Delta x.$$

Jak wiemy, funkcja f lokalnie rośnie w otoczeniu punktu x jeżeli $f'(x) > 0$, zaś maleje, jeżeli $f'(x) < 0$. Oznacza to, że jeżeli chcemy minimalizować wartość funkcji f, powinniśmy się poruszać w kierunku przeciwnym do pochodnej.

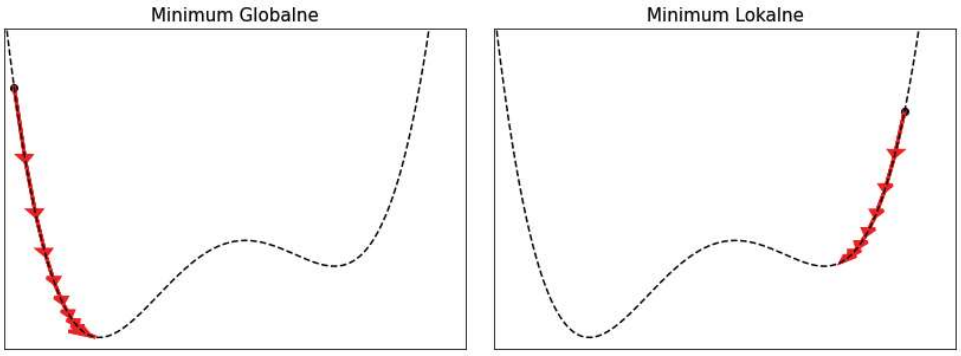
Aby to zrealizować, ustalamy wielkość kroku $h > 0$, startujemy w losowo wybranym punkcie v_0 i poruszamy się zgodnie z regułą:

$$v_{n+1} = v_n - hf(v_n).$$

Zauważmy, że tak zdefiniowany algorytm GD ma dwa hiperparametry:



Rysunek 2.2: Porównanie minimalizacji iteracyjnej i gradientowej.



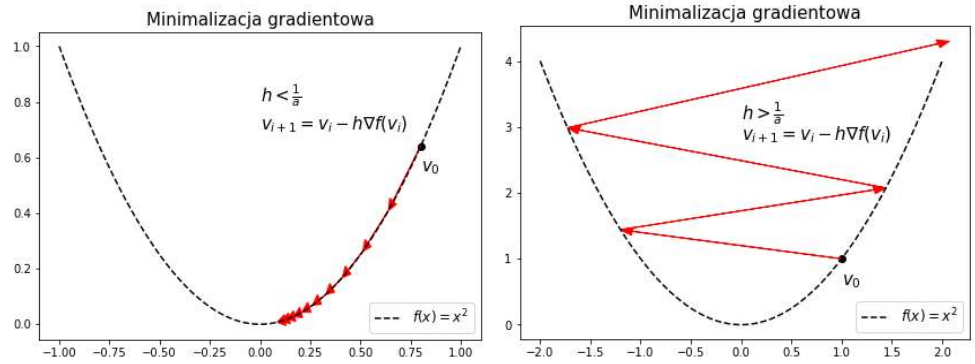
Rysunek 2.3: Wpływ punktu startowego na znalezione minimum dla niewypukłej funkcji (wielomianu) w przypadku minimalizacji gradientowej.

- początkowy punkt v_0,
- wielkość kroku $h > 0$.

Pokażemy, że wybór tych parametrów ma istotne znaczenie dla działania algorytmu.

Po pierwsze metoda gradientowa ma tendencję do wpadania w lokalne minima (rysunek 2.3) i nie mamy zazwyczaj gwarancji, że udało nam się trafić w minimum globalne. W związku z tym w praktyce zazwyczaj poleca się wystartowanie wielokrotnie z wylosowanych punktów startowych i wybranie najlepszego wyniku (względem funkcji kosztu). W przypadku funkcji zdefiniowanych za pomocą sieci neuronowych określa się też często rozkłady z których należy losować, gdyż zły wybór punktu startowego może spowodować, że trafimy w region w którym procedura będzie rozbieżna (patrz rozdział 8).

Drugim istotnym parametrem minimalizacji gradientowej jest wielkość kroku h.



Rysunek 2.4: Wizualizacja minimalizacji gradientowej w zależności od wielkości kroku uczącego h.

Jeżeli jest za mały, to zbieżność może być zbyt wolna, bądź wartość znalezionego minimum lokalnego może być daleka od wartości minimum globalnego. Jeżeli jest za duży, to minimalizacja gradientowa może nie być zbieżna.

Przykład 2.2. Aby zaobserwować wpływ wielkości kroku, rozpatrzmy

$$f(x) = ax^2, \text{ gdzie } a > 0 \text{ jest ustalonym parametrem.}$$

Naszym celem jest znalezienie najmniejszej wartości f (patrz rysunek 2.4). Mamy oczywiście:

$$\nabla f(x) = 2ax.$$

W konsekwencji, startując z v_0 po wykonaniu k-kroków algorytmu GD z wielkością kroku h dostajemy punkt:

$$v_n = (1 - 2ah)^k v_0.$$

Widzimy, że algorytm zbiega do zera (czyli minimum funkcji f) o ile $|1 - 2ah| < 1$, czyli gdy $h < 1/a$, w przeciwnym razie jest rozbieżny. W konsekwencji im a jest większe, czyli funkcja kosztu szybciej rośnie, tym mniejsza musi być wielkość kroku, by zagwarantować zbieżność algorytmu GD.

Minimalizacja gradientowa funkcji wielu zmiennych Bazowy schemat służący do minimalizacji funkcji $f : \mathbb{R}^D \to \mathbb{R}$ wielu zmiennych jest dokładnym odpowiednikiem dla funkcji jednej zmiennej, a mianowicie dokujemy minimalizacji względem każdej ze współrzędnych. Wprowadźmy następujące oznaczenie. Przez $\nabla f(x)$ oznaczamy gradient funkcji f, czyli zestawienie jej wszystkich pochodnych cząstkowych

$$\nabla f(x) = (\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_D}(x)) \in \mathbb{R}^D.$$

Ogólny schemat minimalizacji gradientowej można przedstawić w kilku krokach:

1. Wybierz (losowo) punkt startowy $v_0 \in \mathbb{R}^D$, $i = 0$.
2. Ustal wielkość kroku (learning rate) $h > 0$ i warunek stopu (domyślnie jest to ilość planowanych kroków k lub to, że funkcja kosztu już nie spada).

Rozdział 7

Wprowadzenie do sieci neuronowych

Idea metod kernelowych polega na transformacji danych do takiej przestrzeni, w której problem można rozwiązać za pomocą metod liniowych. O ile metody kernelowe są bardzo przydatne na niewielkich zbiorach danych, o tyle są kosztowne obliczeniowo dla większych danych, a kernel musi zostać zdefiniowany ręcznie i pozostaje stały w czasie całego procesu uczenia.
W tym rozdziale wprowadzimy sieci neuronowe, które w sposób automatyczny tworzą nową reprezentację danych, odpowiednią dla rozwiązania zadanego problemu. Przedstawimy podstawowe pojęcia dotyczące sieci neuronowych oraz zademonstrujemy ich działanie na dwóch przykładach. Rozdział zakończymy rozważaniami dotyczącymi teoretycznych aspektów związanych z sieciami neuronowymi.

7.1 Budowa sieci neuronowych

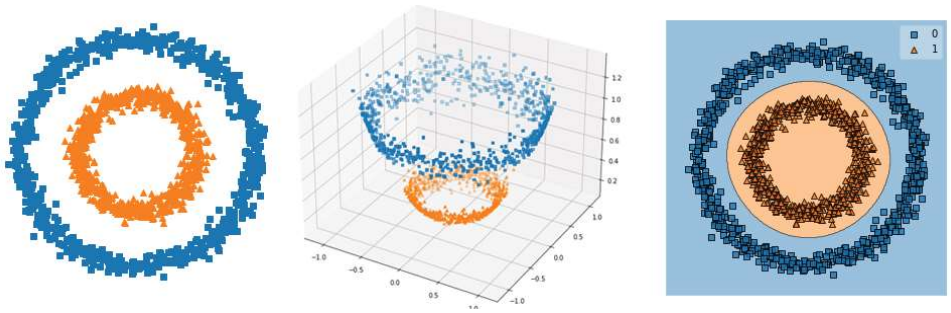
W metodach kernelowych docelowa reprezentacja jest ustalana w oparciu o wybór kernela. W konsekwencji zanurzenie (kernel) pozostaje stałe w trakcie treningu, a uczony jest tylko model liniowy w docelowej przestrzeni.

Główna idea stojąca za sieciami neuronowymi polega na pójściu o krok dalej niż metody kernelowe i stworzeniu takiego modelu, w którym transformacja danych do nowej przestrzeni również będzie podlegała procesowi uczenia. W uproszczeniu można powiedzieć, że sieć neuronowa buduje transformację:

$$\Phi : \mathbb{R}^D \to \mathbb{R}^N,$$

która zadaje nam reprezentację $\hat{x} = \Phi(x)$ i stosuje model liniowy w docelowej przestrzeni, żeby rozwiązać postawiony problem. Zarówno reprezentacja jak i model liniowy są uczone jednocześnie.

Przykład 7.1.1. Aby zobrazować ideę działania sieci neuronowych, załóżmy, że mamy problem klasyfikacji przedstawiony na rysunku 7.1 po lewej stronie, w którym chcemy oddzielić czerwone punkty od niebieskich w dwuwymiarowej przestrzeni (po lewej). Można łatwo zauważyć, że model liniowy nie zdoła rozwiązać tego zadania – granica decyzyjna w tym przypadku powinna być okręgiem. Jeżeli natomiast stworzymy nową reprezentację naszych punktów za pomocą $\hat{x} = \Phi(x) = (x_1, x_2, x_1^2 + x_2^2)$ (po prawej), to przy użyciu takiej przestrzeni jesteśmy w stanie znaleźć płaszczyznę decyzyjną i tym samym możemy wykorzystać model liniowy. O ile tę transformację możemy również zadać korzystając z metod kernelowych, o tyle sieć neuronowa będzie mogła zdefiniować ją w miarę automatycznie i uczyć klasyfikator jednocześnie z reprezentacją danych.

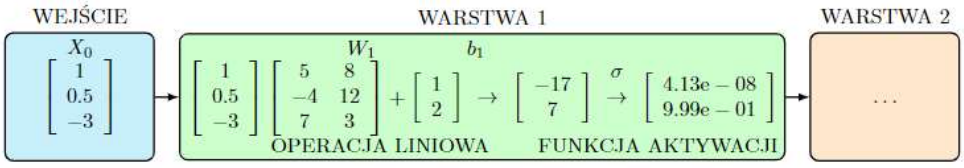


Rysunek 7.1: Transformacja nieliniowo separowanych 2-wymiarowych danych do 3-wymiarowej przestrzeni za pomocą przekształcenia $\hat{x} = \Phi(x) = (x_1, x_2, x_1^2 + x_2^2)$, które pozwala oddzielić klasy za pomocą hiperpłaszczyzny.

Warstwa sieci neuronowej Rozpoczniemy od konstrukcji odwzorowania Φ zadającego sieć neuronową. Sieć neuronowa stanowi nałożenie na siebie wielu prostych odwzorowań, które mogą być szybko ewaluowane. Naturalnym wyborem rodzaju odwzorowania są funkcje liniowe. Niestety złożenie odwzorowań liniowych pozostaje liniowe, co oznacza, że cała sieć pozostałaby liniowa. Aby wzbogacić reprezentację i uzyskać efekt nieliniowości, po każdej warstwie liniowej aplikujemy ustaloną nieliniową funkcję, nazywaną funkcją aktywacji. Funkcja aktywacji jest nakładana na każdą współrzędną wyjściowego wektora z osobna. Chociaż istnieje wiele strategii wyboru funkcji aktywacji, na razie będziemy zakładać że funkcją aktywacji jest sigmoid: $\sigma(x) = \frac{\exp(x)}{1+\exp(x)} \in (0, 1)$. Finalnie sieć neuronowa składa się z sekwencji aplikowanych na przemian operacji liniowych oraz funkcji aktywacji, tak jak pokazano na rysunku 7.2.

Formalnie, oznaczmy przez $x_0 = x$ punkt wejściowy. Wówczas sieć neuronowa to złożenie funkcji $\phi = f_k \circ \ldots \circ f_1$ postaci:

$$f_i(x_{i-1}) = \sigma(W_i^T x_{i-1} + b_i),$$



Rysunek 7.2: Uproszczony schemat sieci neuronowej (wielowarstwowego perceptronu).

gdzie $\sigma : \mathbb{R} \to \mathbb{R}$ to nieliniowa funkcja aktywacji, W_i to macierz wag, a b_i to wektor wyrazów wolnych (tak zwany bias). Wymiar W_i oraz b_i zależy od tego, jaki chcemy uzyskać wymiar reprezentacji na wyjściu z i-tej warstwy. Funkcja f_i jest nazywana warstwą ukrytą (hidden layer), jako że jej wynik jest potrzebny tylko wewnątrz sieci i nie będziemy go wykorzystywać poza nią. Na koniec definiujemy ostatnią warstwę wyjściową:

$$\hat{y} = W_o^T \phi(x) + b_o,$$

która daje odpowiedź sieci. Jeśli rozważany problem klasyfikacji $\hat{y}$ jest funkcją scorującą, dla regresji może to być pojedyncza liczba rzeczywista. Taki model sieci neuronowej nazywamy wielowarstwowym perceptronem i oznaczmy MLP (multilayer perceptron). O sieci MLP (jak również o jej warstwach) mówi się też sieć fully connected, gdyż wszystkie neurony sąsiednich warstw są ze sobą połączone. W kolejnych rozdziałach poznamy sieci neuronowe, których warstwy mają inną postać.

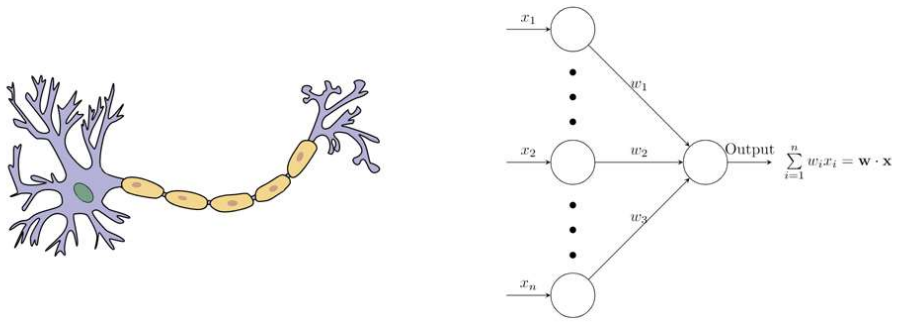
Pojęcie neuronu Sztuczne sieci neuronowe mają swój początek w biologii. Zauważmy, że wynik pojedynczej warstwy $f(x) = W^T x + b$ to wektor, którego j-ty element zadany jest przez:

$$(f(x))_j = \sum_k W_{kj} x_k + b_j.$$

O pojedynczej kolumnie $W_{\cdot j}$ i skalarze b_j można myśleć jako o pojedynczej jednostce, która niezależenie od reszty warstwy produkuje jedną liczbę. Tę jednostkę nazywamy neuronem, jako że w bardzo uproszczony sposób przybliża działanie biologicznych neuronów. Porównanie sztucznego i biologicznego neuronu znajduje się na rysunku 7.3. Formalnie neuron to złożenie funkcji aktywacji ze skalarną funkcją liniową.

Każda warstwa składa się więc z szeregu neuronów – i od tego pochodzi właśnie nazwa „sieci neuronowe". Liczba neuronów w danej warstwie jest równoznaczna z liczbą kolumn w macierzy W, nazywa się ją także szerokością warstwy.

Parametryzacja i trenowanie sieci neuronowych W konstrukcji sieci neuronowych nie ma żadnych ograniczeń dotyczących zmiany wymiarowości przez kolejne warstwy f_i. Innymi słowy wymiar $f_i(x_{i-1})$ może być inny niż wymiar x_{i-1}. Dostosowanie wymiarów poszczególnych warstw oraz ich liczby, czyli wybór architektury sieci, ma bardzo istotny wpływ na wynikową reprezentację. W praktyce im więcej warstw i im więcej neuronów w każdej warstwie, tym reprezentacja będzie bogatsza, ale również bardziej kosztowana obliczeniowo i podatna na overfitting. Z drugiej strony, używanie mniejszej liczby warstw z niższą wymiarowością pozwala wydobyć



Rysunek 7.3: Porównanie neuronów biologicznego i sztucznego.

kluczowe cechy danych, kosztem mniejszej ekspresji. Finalnie istotne jest uzyskanie końcowej reprezentacji $\Phi(x)$, w której jesteśmy w stanie rozwiązać problem metodami liniowymi.

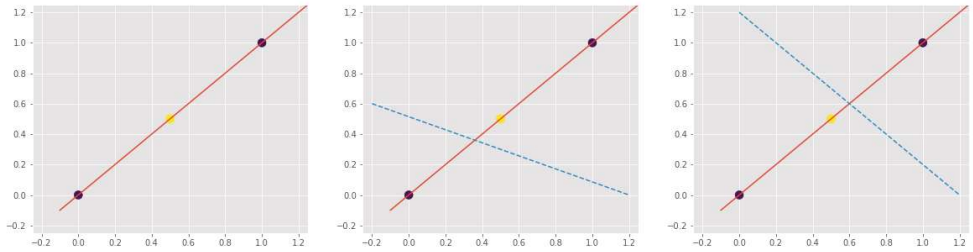
Uczenie sieci neuronowej polega na znalezieniu optymalnych parametrów modelu – w tym przypadku wag W_i oraz biasów b_i. Aby mówić o uczeniu konieczne jest zdefiniowanie funkcji kosztu podobnie jak to miało miejsce w modelach płytkich. W problemie regresji dla modeli linowych przedstawionym w rozdziale 4 minimalizowaliśmy błąd średniokwadratowy (MSE), a w problemie klasyfikacji z rozdziału 5 minimializowaliśmy entropię krzyżową (negatywny log-likelihood). Te same funkcje kosztu będziemy minimalizować w przypadku sieci neuronowej. Zmienia się tylko nasz model – zamiast jednej warstwy, dostajemy kilka oraz dodajemy funkcję aktywacji. Uczenie sieci neuronowej przebiega w sposób gradientowy, a dokładniejszy opis znajduje się w kolejnym rozdziale.

7.2 Klasyfikacja nieliniowa: spojrzenie geometryczne

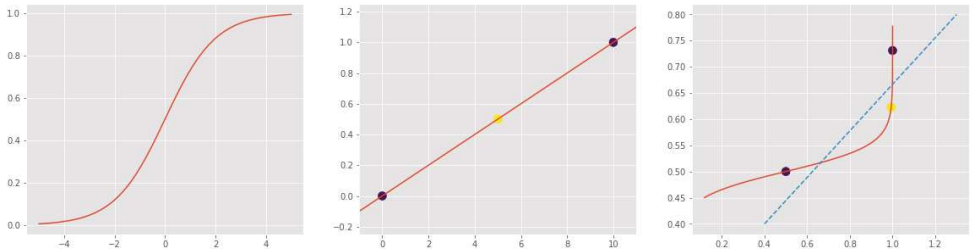
Żeby lepiej zrozumieć, w jaki sposób sieci neuronowe rozwiązują problem nieliniowy, rozważmy prosty problem klasyfikacji na płaszczyźnie. Na rysunku 7.5 są pokazane trzy punkty, z których $x_1 = (0, 0), x_2 = (1, 1)$ należą do klasy 0, natomiast $x_3 = (0.5, 0.5)$ należy do klasy 1. Chcemy zbudować model, który będzie w stanie poprawnie sklasyfikować te punkty, co jest geometrycznie równoznaczne z oddzieleniem klasy 0 od klasy 1 granicą decyzyjną.

Aby problem klasyfikacyjny był możliwy do rozwiązania przez model liniowy, granica decyzyjna musi być wyrażona za pomocą hiperpłaszczyzny (w tym przypadku linii). Niestety, w naszej sytuacji wszystkie trzy punkty leżą na tej samej linii. Granica decyzyjna musiałaby więc przeciąć tę linię w dwóch punktach: pomiędzy punktem x_1 a x_3 oraz pomiędzy x_3 a x_2. W konsekwencji nie jesteśmy w stanie rozwiązać tego zadania za pomocą modelu liniowego.

W celu rozwiązania postawionego problemu zastosujemy sieć neuronową i pokażemy, jak na jej wyniki wpłynie dobór funkcji aktywacji. Rozważmy bardzo



Rysunek 7.4: Zbiór 3 współliniowych punktów do klasyfikacji binarnej (kolory obrazują klasy), której nie da się rozwiązać za pomocą modelu liniowego, jak widać na drugim i trzecim rysunku.



Rysunek 7.5: Wykres funkcji sigmoid (lewa strona) oraz jej zastosowanie do problemu klasyfikacji danych przedstawionych na środkowym rysunku. Po prawej stronie są widoczne dane po transformacji liniowej z dodaniem aktywacji sigmoidalnej, które można rozdzielić prawidłowo za pomocą linii.

prostą sieć neuronową z jedną warstwą ukrytą postaci $f_1(x) = g(W_1^T x + b_1)$, gdzie g to funkcja aktywacji i $W \in \mathbb{R}^{2\times 2}, b \in \mathbb{R}^2$ oraz warstwą wyjściową daną przez $f_o(x) = \sigma(W_o^T x + b_o)$, gdzie σ to aktywacja sigmoidalna a $W \in \mathbb{R}^{2\times 1}, b \in \mathbb{R}$. Zauważmy, że przy takim doborze wymiarów wynik po pierwszej warstwie ukrytej dalej będzie wektorem dwuwymiarowym, co ułatwi nam zwizualizowanie problemu na płaszczyźnie.

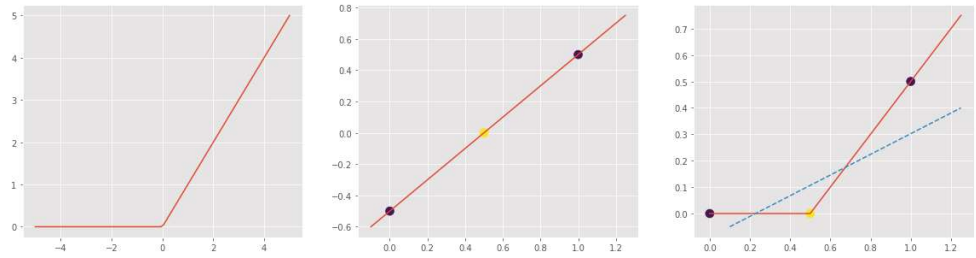
Rozpatrzmy kilka możliwości doboru funkcji g.

Aktywacja liniowa Załóżmy że g jest identycznością: $g(x) = x$. Mamy wtedy złożenie funkcji liniowych, które jest również funkcją liniową. Taki model nie jest w stanie rozwiązać problemu.

Model z sigmoidalną nieliniowością. Kluczem do rozwiązania problemu jest zastosowanie funkcji nieliniowej. Rozważmy funkcję sigmoidalną:

$$g(x) = \sigma(x) = \frac{\exp(x)}{1 + \exp(x)}.$$

Patrząc na wykres sigmoidy (rysunek 7.5, lewy) możemy zauważyć, że działa podobnie jak funkcja liniowa w pobliżu zera, ale wartości bardzo małe oraz bardzo duże będą ściągane blisko siebie, odpowiednio do 0 oraz do 1. Dla przykładu $\sigma(100) - \sigma(10) \approx 5 \cdot 10^{-5}$.



Rysunek 7.6: Wykres funkcji ReLU oraz jej zastosowanie do klasyfikacji nieliniowo separowanych danych przedstawionych na środkowym rysunku. Po prawej stronie widoczne są dane po transformacji liniowej z dodaniem aktywacji sigmoidalnej, które można rozdzielić prawidłowo za pomocą linii.

Weźmy teraz macierz W_1, która powiększy wartości na osi x a oś y pozostawi niezmienioną:

$$W_1 = \begin{bmatrix} 10 & 0 \\ 0 & 1 \end{bmatrix}$$

Wektor $b = [0, 0]^T$ niech będzie zerowy. Jeśli na tak przekształconych danych wywołamy funkcję sigmoid, to linia, na której leżały punkty zostanie wygięta za pomocą nieliniowej aktywacji, co zostało zobrazowane na rysunku 7.5. W efekcie nasz model zyskał wystarczająco ekspresji, żeby rozwiązać nasz problem nieliniowy.

Model z nieliniowością ReLU W tym przykładzie w miejsce funkcji sigmoid można zastosować wiele innych nieliniowych funkcji, aby uzyskać separowalność w wynikowej przestrzeni. W szczególności zamiast efektu nieliniowego rozciągania przestrzeni, przydatne przy rozwiązywaniu postawionego problemu może być również przycinanie przestrzeni za pomocą funkcji ReLU:

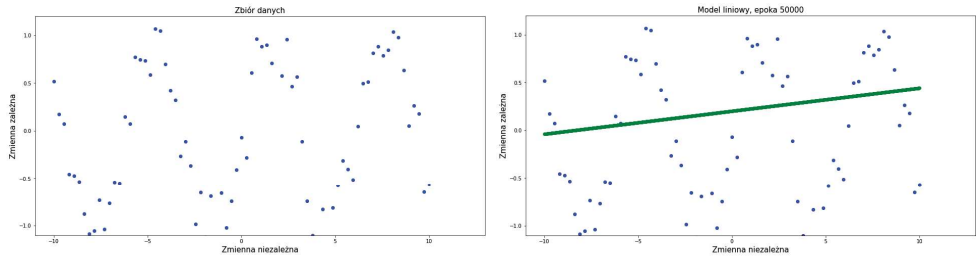
$$\text{ReLU}(x) = \begin{cases} 0, \text{jeżeli } x < 0 \\ x, \text{jeżeli } x \geq 0, \end{cases} .$$

która została zobrazowana na rysunku 7.6 (lewy).

Do reprezentacji warstwy ukrytej $\phi(x) = \text{ReLU}(W_1^T x + b_1)$ użyjmy identyczności owej macierzy W_1, natomiast niech wektor b_1 przesuwa punkty w dół na osi y:

$$W = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$
$$b = \begin{bmatrix} 0 \\ -0.5 \end{bmatrix}$$

Po zastosowaniu funkcji ReLU wszystkie wartości ujemne zostaną przycięte do zera. Tym samym nasza prosta zostanie w tym przypadku złamana w momencie przejścia przez wartość $y = 0$. Jako że punkty nie leżą teraz na prostej, potrafimy znaleźć liniową granicę decyzyjną.



Rysunek 7.7: Zbiór danych dla problemu regresji - jak widać, model liniowy nie jest w stanie dobrze dopasować się do danych.

Podsumowanie Powyższe przykłady pokazują, że dzięki nielinowym funkcjom aktywacji sieci neuronowe są w stanie zakrzywiać przestrzeń, w której znajdują się dane. Celem jest przeniesienie danych do takiej przestrzeni, żeby ostatnia warstwa była w stanie rozwiązać problem w sposób liniowy. Możemy zadać pytanie, która funkcja jest najlepsza. Empiryczne badania pokazują, że w większych sieciach od aktywacji sigmoidalnych znacznie lepiej działają aktywacje ReLU – związane jest to z problemami optymalizacyjnymi przy liczeniu gradientu, o których opowiemy więcej w rozdziale 12. Natomiast nie oznacza to, że ReLU jest zawsze idealnym wyborem i warto przetestować wiele różnych funkcji aktywacji, żeby znaleźć tę, która będzie działała najlepiej dla konkretnego problemu.

7.3 Uczenie sieci neuronowej na przykładzie regresji

W poprzednich sekcjach opisaliśmy budowę sieci neuronowych, następnie pokazaliśmy, że dzięki zastosowaniu nieliniowych funkcji aktywacji, sieć może rozwiązywać problemy nieliniowe. Teraz pokażemy jak automatycznie nauczyć sieć neuronową na danych i wykorzystać ją do rozwiązania konkretnego problemu.

Rozważmy problem regresji ze zbiorem danych składającym się z 1000 elementów generowanych przez poniższą formułę:

$$f(x) := \sin(x) + 0.3 \cdot \mathcal{N}(0,1) \quad \text{dla } x \in [-10, 10],$$

których nie da się dobrze przybliżyć modelem liniowym (rysunek 7.7). Gdybyśmy znali odpowiednie zanurzenie (na przykład wielomianowe o odpowiednim stopniu), to bylibyśmy w stanie na jego podstawie znaleźć dobre rozwiązanie modelem liniowym. Niestety znalezienie odpowiedniego zanurzenia będzie wymagało dobrej intuicji na temat problemu. W tym wypadku moglibyśmy sobie wyobrazić, że wielomian szóstego stopnia powinien zadziałać dobrze, ale w przypadku wyżej wymiarowych problemów już nie będziemy w stanie tego tak łatwo określić. Jak zobaczymy poniżej, sieć neuronowa samodzielnie znajdzie odpowiednią reprezentację. Wciąż będziemy musieli podjąć szereg decyzji dotyczących architektury, funkcji kosztu czy optymalizacji, żeby nauczyć sieć neuronową, ale dokonywanie tych wyborów będzie w przypadku

skomplikowanych problemów znacznie prostsze niż ręczne wyszukiwanie dobrego zanurzenia.

Architektura Pierwszym krokiem do rozwiązania powyższego problemu jest ustalenie architektury naszej sieci neuronowej. Musimy zadecydować, ile będzie zawierać warstw (głębokość) oraz jaka będzie wymiarowość poszczególnych warstw (szerokość). Im większa architektura, tym większa ekspresja sieci, co oznacza, że sieć jest w stanie przybliżać więcej funkcji. Jednocześnie zbyt duża ekspresja sieci może prowadzić do overfittingu, czyli przesadnie dokładnego dopasowania do szumu obecnego w zbiorze treningowym, co pogarsza wyniki na zbiorze testowym (pojęcie to omawialiśmy wcześniej w rozdziale 4). Natomiast im mniejsza sieć, tym mniejsza ekspresywność. W skrajnym przypadku jednej warstwy otrzymamy model liniowy.

Na potrzeby podanego problemu przetestujemy kilka różnych sieci neuronowych o różnych architekturach. W każdym przypadku mamy jeden neuron wejściowy i jeden wyjściowy:

(a) Jedna warstwa ukryta z 2 neuronami: $1 \to 2 \to 1$.

(b) Dwie warstwy ukryte z 16 neuronami: $1 \to 16 \to 16 \to 1$.

(c) Trzy warstwy ukryte z 32 neuronami: $1 \to 32 \to 32 \to 32 \to 1$.

(d) Trzy warstwy ukryte z 1024 neuronami: $1 \to 1024 \to 1024 \to 1024 \to 1$.

(e) Sześć warstw ukrytych z 1024 neuronami: $1 \to 1024 \to 1024 \to 1024 \to 1024 \to 1024 \to 1024 \to 1$.

Wybór architektury zadaje nam zbiór parametrów θ, które będziemy następnie optymalizować. W ogólności chcemy myśleć o θ jako o bardzo wysoko wymiarowym wektorze zawierającym wszystkie parametry naszego modelu. Jeżeli na przykład mielibyśmy sieć neuronową z dwoma warstwami ukrytymi, to nasz θ powstałby na podstawie wektoryzacji i złączenia trzech macierzy wag oraz trzech wektorów biasu:

$$\theta = [\mathrm{vec}(W_1), b_1, \mathrm{vec}(W_2), b_2, \mathrm{vec}(W_o), b_o],$$

gdzie przez W_o, b_o oznaczamy parametry ostatniej warstwy.

Funkcja kosztu Wybranie architektury zadaje nam zbiór parametrów θ. Uczenie sieci neuronowej polega na znalezieniu takich wartości tych parametrów, które dobrze nadają się do rozwiązania problemu. W przypadku regresji jako funkcję kosztu weźmiemy błąd kwadratowy:

$$L(x, y) = \mathrm{SE}(X, Y, f) = \|f(x) - y\|_2^2.$$

W praktyce schemat uczenia sieci neuronowych jest bardzo podobny do iteracyjnego uczenia modeli liniowych. Skorzystamy tutaj z metody spadku gradientu. Zatem musimy dobrać parametry (wagi) sieci tak, aby minimalizowały wartość funkcji kosztu na zbiorze treningowym. Jest to ponownie analogiczne do sytuacji z modeli płytkich, gdzie wykorzystywaliśmy metodę spadku gradientu, w której poprawiamy wartości parametrów w kierunku najszybszego spadku funkcji kosztu zadanego przez gradient.

Liczenie gradientu W przypadku modeli liniowych policzenie gradientu można było wykonać własnoręcznie. W przypadku sieci neuronowych, zwłaszcza takich z wieloma warstwami, policzenie gradientu jest znacznie bardziej skomplikowane. W szczególności postać gradientu będzie się istotnie zmieniać w zależności od liczby i szerokości warstw. W konsekwencji musielibyśmy wyprowadzać gradient od nowa przy każdej zmianie architektury. Na szczęście operację liczenia gradientu da się łatwo zautomatyzować za pomocą metody nazywanej propagacją wsteczną (wrócimy do niej w kolejnym rozdziale). Na razie zakładamy, że jesteśmy w stanie łatwo policzyć gradient $\nabla_\theta L(x, y; \theta)$ dla zadanej pary przykładu x i etykiety y oraz aktualnych parametrów sieci θ.

Interesuje nas zminimalizowanie średniego błędu na całym zbiorze treningowym $\frac{1}{N}\sum_{i=0}^{N} L(x_i, y_i; \theta)$. W sieciach neuronowych zazwyczaj liczymy wartości funkcji kosztu jednocześnie, dzięki wykorzystaniu kart graficznych, które są w stanie zrównoleglić obliczenia. Chociaż wydaje się to detalem technicznym, możliwość zrównoleglenia obliczeń jest istotną częścią sukcesu sieci neuronowych i głębokiego uczenia w ogóle. Gdybyśmy w trakcie optymalizacji musieli liczyć gradienty przykładów sekwencyjnie, wydłużyłoby to czas uczenia do wielkości, która jest całkowicie niepraktyczna. Liczba przykładów, jaka może się zmieścić w pamięci karty graficznej jest w praktyce ograniczona, ale na ten moment przyjmijmy że potrafimy policzyć funkcję kosztu dla całego zbioru danych.

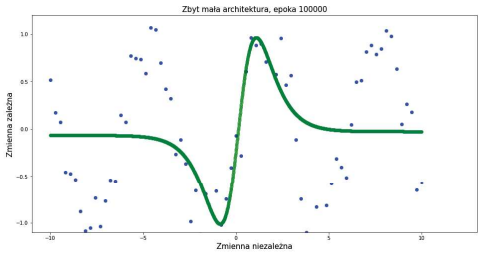
Zadanie 7.1. Porównaj czas treningu modelu z tej sekcji z użyciem CPU oraz GPU.

Metoda spadku gradientu dla sieci neuronowych Poniższy schemat jest przypomnieniem metody spadku gradientu opisanej dla metod liniowych w rozdziale 2:

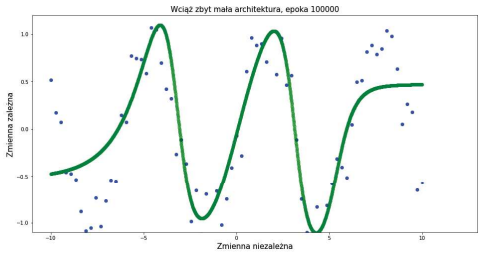
1. Ustal hiperparametry: η (krok uczenia), n (liczba iteracji).
2. Ustaw przypadkowe wartości parametrów θ_0.
3. Dla $k = 1, \ldots, n$:
   - Policz wartość $L(x, y; \theta_k)$ (forward pass);
   - Policz gradient $\nabla_{\theta_k} L(x, y; \theta_k)$ za pomocą mechanizmu propagacji wstecznej (backward pass);
   - Wykonaj krok gradientowy: $\theta_{k+1} = \theta_k - \nabla_{\theta_k} L(x, y; \theta_k)$.

Warto podkreślić, że w podanym algorytmie jest kluczowe, że gradient w nowym punkcie jest zawsze liczony od nowa. Gradient policzony dla θ_k nie mówi nam nic o gradiencie dla θ_{k+1}, więc nie możemy go ponownie użyć.

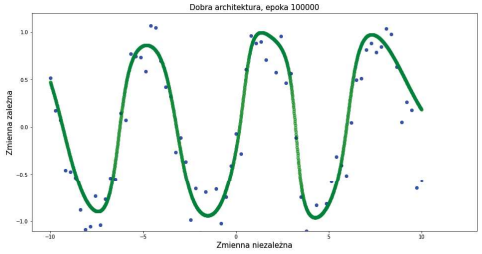
W praktyce dobór hiperparametrów, takich jak krok uczenia, liczba iteracji czy sposób inicjalizacji parametrów jest nietrywialnym problemem, który często wymaga długich empirycznych eksperymentów. Natomiast na tę chwilę przyjmiemy że jesteśmy w stanie dobrać te wartości poprawnie.



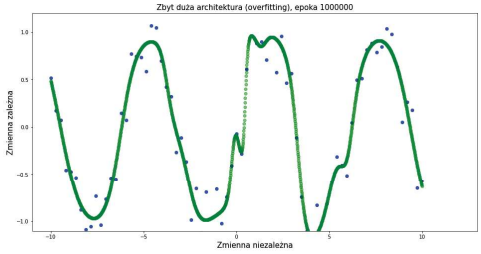
(a) Zbyt mała architektura.



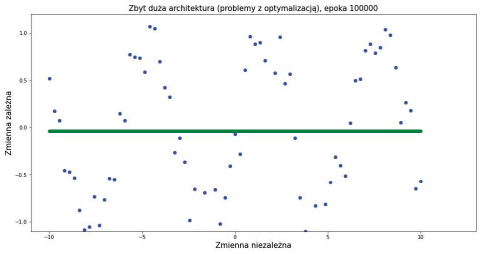
(b) Wciąż zbyt mała architektura.



(c) Dobrze dopasowana architektura.



(d) Zbyt duża architektura (overfitting).



(e) Zbyt duża architektura (problemy z optymalizacją).

Rysunek 7.8: Wynik regresji dla (a) zbyt małej architektury, (b) nieco zbyt małej architektury, (c) dobrej architektury, (d) zbyt dużej architektury, która powoduje overfitting, i (e) jeszcze większej architektury, która powoduje problemy z optymalizacją.

Wyniki Wyniki działania poszczególnych modeli zilustrowane na rysunku 7.8 pokazują, że trzecia architektura dobrze pasuje do zadanego problemu. Sieć poprawnie dopasowuje się do kształtu sinusoidy, bez przesadnych odchyleń.

Jednocześnie obserwujemy, że dla innych architektur problem ma znacznie gorsze rozwiązania. Architektura (a) nie ma wystarczająco ekspresji, żeby rozwiązać ten prosty problem i jest w stanie dopasować się właściwie tylko do linii prostej dzielącej minimum w punkcie $-\pi$ z maksimum w punkcie π. Chociaż architektura (b) radzi sobie nieco lepiej, dalej nie jest w stanie dopasować się do krzywych na brzegu widzianego przedziału. Te dwie architektury obrazują problem underfittingu – sieć neuronowa nie ma wystarczająco ekspresji żeby dobrze dopasować się do danych.

Przy zastosowaniu zbyt dużej architektury, sieć zaczyna dopasowywać się do szumu. W przypadku sieci (d) sinusoida przestaje być gładka, zaczyna falować i do-

pasowywać się do szumu. Jest to przykład overfittingu. Co więcej przy zastosowaniu jeszcze większej architektury, sieć neuronowa nie jest nawet w stanie dopasować się do punktów, co jest spowodowane problemami optymalizacyjnymi (e). Chociaż tak duża sieć ma wystarczająco ekspresji, żeby dopasować się do punktów, znalezienie odpowiednich wartości parametrów za pomocą metody spadku gradientu staje się bardzo trudne.

Wnioski Powyższe przykłady obrazują dwa kluczowe problemy, które sieć neuronowa powinna rozwiązywać:

1. Optymalizacja: znalezienie modelu, który dobrze dopasowuje się do danych treningowych. Problemu tego nie rozwiązują sieci (a) i (b), jako że są zbyt płytkie oraz sieć (e), w przypadku której metoda spadku gradientu nie zdołała znaleźć dobrego minimum. Sieci (c) i (d) dobrze dopasowują się do punktów, więc rozwiązują problem optymalizacyjny.

2. Generalizacja: znalezienie modelu, który dobrze dopasowuje się do prawdziwego rozkładu. Sieć (c) rozwiązuje ten problem, ucząc się funkcji bardzo podobnej do sinusoidy, natomiast sieć (d) odstaje kształtem od sinusoidy i tym samym nie generalizuje poprawnie. W samej generalizacji sieć (b) wydaje się lepiej dopasowana niż sieć (d).

Jakość optymalizacji mierzymy na zbiorze treningowym, sprawdzając czy sieć neuronowa poprawnie zapamiętała przykłady. Natomiast jakość generalizacji sprawdzimy na zbiorze testowym, weryfikując czy sieć jest w stanie ekstrapolować do wcześniej nie widzianych danych.

7.4 Teoria a praktyka w sieciach neuronowych

Patrząc na przykłady z poprzednich sekcji możemy zauważyć, że sieci neuronowe są bardziej ekspresywne niż modele liniowe. Można się w takim razie zastanowić czy istnieją takie funkcje, których sieć neuronowa nie potrafi aproksymować? Twierdzenie o uniwersalnej aproksymowalności sieci neuronowej mówi nam, że przynajmniej w teorii takiej sieci nie ma:

Twierdzenie 7.1. (Cybenko 1989) Niech $g : A \to \mathbb{R}^N$ będzie funkcją ciągłą zdefiniowaną na zbiorze zwartym $A \subset \mathbb{R}^D$. Wówczas dla każdego błędu $\varepsilon > 0$ istnieje sieć neuronowa $f : \mathbb{R}^D \to \mathbb{R}^N$ z jedną warstwą ukrytą, która dla każdego $x \in A$ spełnia:

$$\|f(x) - g(x)\| < \varepsilon.$$

Powyższe twierdzenie pokazuje, że każdą funkcję ciągłą możemy dowolnie dobrze opisać za pomocą sieci neuronowej z jedną warstwą ukrytą. Chociaż ten wynik jest interesujący teoretycznie, okazuje się że nie pomaga w praktyce w budowaniu sieci neuronowych. W szczególności twierdzenie to sugeruje, że sieć neuronowa z jedną warstwą ukrytą o wystarczającej liczbie neuronów to wszystko, co jest potrzebne przy budowaniu sieci neuronowych. Okazuje się, że w zastosowaniach takie płytkie sieci