KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Politechnika Wrocławska

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Poznawanie własności sieci neuronowych w środowisku MATLAB

Piotr Ciskowski

„Wzrost liczby absolwentów w Politechnice Wrocławskiej na kierunkach o kluczowym znaczeniu dla gospodarki opartej na wiedzy"

Projekt współfinansowany ze środków Unii Europejskiej
w ramach Europejskiego Funduszu Społecznego

Recenzent:
dr inż. Anna Czemplik

OFICYNA WYDAWNICZA POLITECHNIKI WROCŁAWSKIEJ
Wybrzeże Wyspiańskiego 27, 50 – 370 Wrocław

ISBN – 978-83-7493-712-2

Piotr Ciskowski

Poznawanie własności sieci neuronowych w środowisku MATLAB

Spis treści

Rozdział 1

Wstęp

Sztuczne sieci neuronowe to rodzaj maszyn uczących inspirowanych sposobem działania ludzkiego mózgu. Początków ich teorii można doszukiwać się w badaniach pierwszych neurobiologów z początku ubiegłego wieku, a nawet w słynnych eksperymentach Pawłowa. Za bardziej konkretną datę narodzin nowej dziedziny uznaje się opublikowanie pierwszego modelu sztucznego neuronu przez McCullocha i Pittsa w 1943r. Dziś sieci neuronowe to powszechnie stosowane instrumenty statystyczne i narzędzia do przetwarzania sygnałów. Jest to obecnie zaawansowana już dziedzina nauki, jednak wciąż rozwijająca się i znajdująca nowe zastosowania.

W wielu pozycjach literatury, przytoczonych na końcu tej książki, w początkowych rozdziałach można znaleźć wyczerpujący opis zarówno historii rozwoju teorii sieci neuronowych, jak również inspiracji biologicznych, a także porównanie sposobu działania tradycyjnych komputerów i neurokomputerów. Szczególnie polecamy książki [4, 5]. Czytelników zainteresowanych biologicznymi podstawami przewodnictwa sygnałów w ludzkim systemie nerwowym zachęcamy do zapoznania się z początkowymi rozdziałami [3]. Z kolei [1] jest ciekawą rozprawą o poszukiwaniu metod budowy myślących maszyn, lecz przede wszystkim o dążeniu do poznania sposobu funkcjonowania ludzkiego mózgu. Dyskusja ta rysuje również tło dla teorii sztucznych sieci neuronowych umiejscawiając je pośród innych kierunków nauki związanych z szeroko rozumianą „sztuczną inteligencją". W tej książce zakładamy, że Czytelnik zna ogólnie inspiracje biologiczne teorii sztucznych sieci neuronowych, więc bez przypominania biologicznego pierwowzoru przedstawimy od razu model sztucznego neuronu.

Niniejszy skrypt ma być z założenia pomostem między dwiema książkami od lat obecnymi na polskim rynku, stanowiącymi w przekonaniu autora obowiązkowe pozycje w bibliotece każdego badacza sieci neuronowych. Pierwsza z nich to książka prof. Tadeusiewicza ([5]), ciekawie, inspirująco i bardzo przystępnie wprowadzająca w świat sieci neuronowych, opisująca ich działanie, możliwości oraz procesy uczenia bez użycia ani jednego równania, za to przy pomocy wielu programów komputerowych ilustrujących kolejne zagadnienia i umożliwiających

przeprowadzanie własnych eksperymentów. Książka prof. Osowskiego ([4]) to z kolei swego rodzaju biblia opisująca (przy użyciu wielu równań, ale nadal bardzo przystępnie) większość znanych rodzajów, architektur, metod uczenia sieci, a także przykłady ich zastosowania.

Program MATLAB, przeznaczony do wydajnego prowadzenia obliczeń macierzowych i łatwej wizualizacji ich wyników, jest idealnym środowiskiem dla sztucznych sieci neuronowych, które – jak później zobaczymy – są w zasadzie zbiorem macierzy. Również w macierzach zapisane są zwykle przykłady zadań, których sieć ma się nauczyć. Łatwość programowania i wizualizacji wyników obliczeń oraz powszechność w środowiskach akademickich to również niezwykle ważne zalety tego pakietu. Dlatego też warto wykorzystać to środowisko do przystępnego przedstawienia zasad działania sieci neuronowych i zjawisk zachodzących podczas ich uczenia. Jest ono również idealnym narzędziem do implementacji sieci neuronowych w praktycznych zastosowaniach, szczególnie w dziedzinach technicznych.

Podstawy teorii sieci neuronowych zostaną przedstawione poprzez szereg zadań, wprowadzających kolejne modele i metody. Rozwiązaniami zadań są funkcje i skrypty, z początku bardzo proste, niekiedy składające się zaledwie z kilku linii kodu (co po części jest zasługą zwięzłego zapisu wyrażeń macierzowych w MATLABie, ukrywającego szczegóły implementacyjne i pozwalającego skupić się na istocie rzeczy). W głównym nurcie książki Czytelnik znajdzie zadania oraz opis teorii towarzyszącej danemu zadaniu, często również „szkielet" funkcji do uzupełnienia i wskazówki programistyczne.

Najważniejsze programy (funkcje i skrypty) – wszystkie dotyczące sieci jednowarstwowych i podstawowe dla sieci dwuwarstwowych – załączone są jako rozwiązania zadań na końcu książki. Podstawowe programy dostępne są również na stronie autora: `staff.iiar.pwr.wroc.pl/piotr.ciskowski` - jako kody źródłowe i jako skompilowane programy. Autor zakłada znajomość środowiska MATLAB w podstawowym zakresie.

Rozdział 2

Sieci jednowarstwowe

Choć będzie to wielkim uproszczeniem, ludzki mózg możemy postrzegać jako układ przetwarzający pewne wejścia (bodźce, sygnały z receptorów) i wytwarzający na ich podstawie pewne wyjścia. Nie będziemy teraz wnikać, czy wyjściami tymi będą myśli, skojarzenia, czy sygnały sterujące innymi częściami naszego ciała, np. ruchem kończyn. Sztuczne sieci neuronowe, jako uproszczone modele oparte na zasadach działania mózgu, będziemy traktować jako układy typu wejście–wyjście.

Sieci są zbudowane z pojedynczych neuronów – pojedynczych bardzo prostych procesorów, ułożonych w warstwy. Sygnały są przesyłane do kolejnych warstw, w których zadanie do rozwiązania jest kolejno upraszczane. Dzięki równoległemu przetwarzaniu danych w warstwach, nieskomplikowane z pozoru neurony tworzą razem bardzo wydajną maszynę uczącą. Wyczerpujące porównanie działania tradycyjnych komputerów osobistych z neurokomputerami można znaleźć w [5].

W tym rozdziale poznamy sieci złożone z jednej warstwy neuronów. Niejako przy okazji poznamy własności pojedynczego neuronu oraz jego możliwości jako klasyfikatora.

W celu poznania budowy, zasad działania oraz uczenia jednowarstwowych sieci jednokierunkowych (ang. MLP – *multilayer perceptron*), napiszemy trzy funkcje:

- `init1`
- `dzialaj1`
- `ucz1`

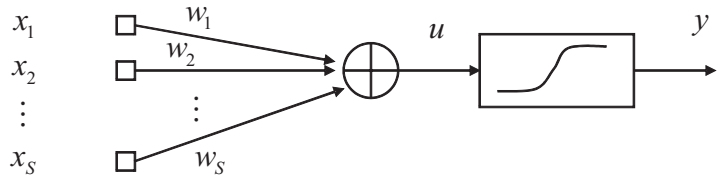
Pierwsza z nich utworzy sieć, druga będzie symulować jej działanie, trzecia umożliwi przeprowadzenie procesu uczenia na zadanym ciągu uczącym. Wszystkie te trzy funkcje zostaną wykorzystane w skrypcie `test1`, który posłuży do zdefiniowania prostego zadania rozpoznawania zwierząt, utworzenia odpowiedniej sieci i nauczenia jej rozwiązywania postawionego problemu.

2.1. Inicjalizacja sieci

Zadanie 1. *Napisz funkcję, która utworzy macierz wag jednowarstwowej sieci o S wejściach i K wyjściach i wypełni ją liczbami losowymi z przedziału od -0.1 do 0.1. Nagłówek funkcji jest następujący:*

```
function [ W ] = init1 ( S , K )
```

Model pojedynczego neuronu o S wejściach przedstawia rys. 2.1. Do wejść oznaczonych jako x_1 do x_S prowadzą wagi od w_1 do w_S. W modelu tym nie uwzględniono wejścia progowego o indeksie 0. Do rozwiązania zadania, jakie postawimy przed siecią w tym rozdziale (rozpoznawanie zwierząt) nie jest to konieczne. Znaczenie progu przy określaniu granic decyzyjnych neuronu zostanie wyjaśnione na końcu rozdziału. Podczas badania własności sieci i przy rozwiązywaniu bardziej skomplikowanych zadań przez sieci dwuwarstwowe, to dodatkowe wejście będzie już zawsze uwzględnione.



Rysunek 2.1. Model pojedynczego neuronu bez wejścia progowego (tzw. *biasu*)

Sygnały podawane na wejścia neuronu można zapisać w postaci wektora $\mathbf{X}$ o wymiarach $S \times 1$

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_S \end{bmatrix}_{S\times 1} \tag{2.1}$$

Podobnie wagi neuronu, prowadzące do odpowiednich wejść, możemy zebrać w wektorze o takich samych rozmiarach

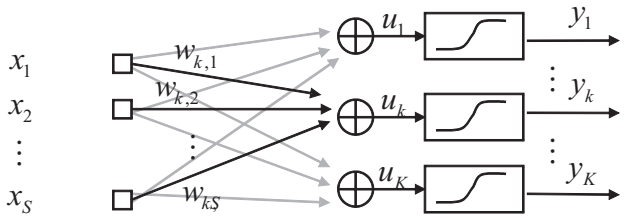
$$\mathbf{W} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_S \end{bmatrix}_{S\times 1} \tag{2.2}$$

Wagi neuronu definiują znaczenie poszczególnych wejść - im waga większa, tym wejście „ważniejsze” przy obliczaniu odpowiedzi neuronu. Wagi mogą mieć

wartości zarówno dodatnie, jak i ujemne, przez co bywają określane jako odpowiednio pobudzające i hamujące. W zadaniach klasyfikacji od wag zależą granice decyzyjne, jakie potrafi wyznaczyć pojedynczy neuron, a także cała sieć.

To właśnie w wagach zawarta jest cała „wiedza" neuronu. „Zdolności" pojedynczego neuronu nie są zbyt wyrafinowane – potrafi on wykonywać proste zadania klasyfikacyjne, choć jak pokażemy później w tych zastosowaniach może on pełnić jedynie rolę dyskryminatora liniowego. Jednak to właśnie z pojedynczych neuronów można budować skomplikowane struktury sieci wielowarstwowych, których zdolności przetwarzania sygnałów bywają zadziwiające.

Oprócz wag i sygnałów wejściowych, na rys. 2.1 zaznaczono również sygnał wewnętrzny neuronu u, będący wynikiem iloczynu sygnałów podanych na wejścia sieci z opowiadającymi im wagami, funkcję aktywacji oraz sygnał wyjściowy y. Wielkościami tymi zajmiemy się później podczas symulacji działania neuronu.



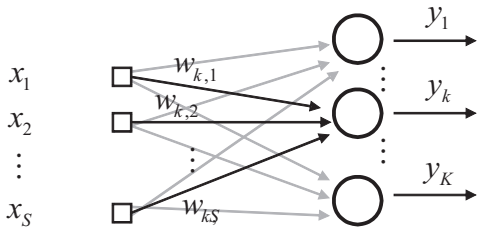
Rysunek 2.2. Sieć jednowarstwowa

Neuron jest podstawową jednostką obliczeniową sieci. Sieć jednowarstwowa to nic innego, jak zespół kilku neuronów, przetwarzających sygnały z tych samych wejść. Sieć taką, złożoną z K neuronów, przedstawia rys. 2.2.

Zaznaczono na nim neuron o indeksie k, aby pokazać powszechnie stosowane oznaczenia. Pobudzenie k-tego neuronu w warstwie będziemy oznaczać przez u_k, zaś jego wyjście przez y_k. Natomiast wagi tego neuronu będziemy indeksować podając najpierw numer neuronu, do którego dana waga należy, a następnie numer wejścia, do którego ona prowadzi. Tak więc np. $w_{3,2}$ oznacza wagę neuronu 3 do wejścia 2.

Tę samą sieć w nieco bardziej zwięzłej postaci przedstawia rys. 2.3. Ponieważ w tej chwili wewnętrzna budowa neuronu nie ma dla nas znaczenia, możemy ją pominąć i oznaczyć każdy neuron pojedynczym symbolem okręgu, zachowując wagi połączeń do niego prowadzących oraz sygnał wyjściowy. Czasem w okręgu zaznacza się również przybliżony kształt funkcji aktywacji danego neuronu.

Wagi poszczególnych neuronów można umieścić w wektorach o odpowiednich indeksach $\mathbf{W}_k$, gdzie $k = 1, 2, ..., K$. Są to wektory kolumnowe o wymiarach $S \times 1$. Wektory wag kolejnych neuronów umieszcza się z kolei obok siebie w wierszu, przez



Rysunek 2.3. Sieć jednowarstwowa (bez wewnętrznej budowy neuronów)

co powstaje macierz wag sieci jednowarstwowej o wymiarach $S \times K$

$$\mathbf{W} = \left[\begin{array}{cccc} \mathbf{W}_1 & \mathbf{W}_2 & \cdots & \mathbf{W}_K \end{array} \right] = \left[\begin{array}{cccc} w_{1,1} & w_{2,1} & \cdots & w_{K,1} \\ w_{1,2} & w_{2,2} & \cdots & w_{K,2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,S} & w_{2,S} & \cdots & w_{K,S} \end{array} \right]_{S \times K} \tag{2.3}$$

Jak już wspomnieliśmy, wiedza sieci neuronowych zawarta jest w ich wagach. Jednocześnie wiemy, że sieci nie rozwiązują stawianych przed nimi problemów w sposób algorytmiczny, czyli nie sposób ich zaprogramować do rozwiązywania jakiegoś zadania. Sieci neuronowe zdobywają umiejętność rozwiązywania zadań na podstawie przykładów pokazywanych im w trakcie uczenia. Przykładami tymi są zestawy par (wejście-wyjście), zawierające prawidłowe odpowiedzi sieci na pobudzenia. Uczenie sieci jest procesem dochodzenia wag do wartości optymalnych – zapewniających odpowiednie reakcje sieci na pokazywane jej sygnały wejściowe, czyli prawidłowe rozwiązanie zadań.

Wagi sieci jednowarstwowej zostały przez nas ułożone w macierzy o wymiarach $S \times K$. Jak zobaczymy później, taka organizacja wag znacznie uprości obliczenia służące symulacji działania oraz uczeniu sieci. Z punktu widzenia uczenia sieci jako procesu optymalizacji, wagi sieci jednowarstwowej są punktem w $S \cdot K$-wymiarowej przestrzeni parametrów, zaś samo uczenie polega na znalezieniu optymalnego punktu w tej przestrzeni, dla którego funkcja celu, zwykle określana jako średnia kwadratów błędów na wyjściach sieci osiągnie najmniejszą wartość.

Im bliżej tego punktu zaczniemy nasze poszukiwania, czyli im „lepsze" będą wagi początkowe sieci, tym efektywniejszy będzie proces ich uczenia. Przez analogię do procesów uczenia się ludzi, możemy powiedzieć obrazowo, że początkowe wagi sieci to jej „wrodzone" zdolności. Opracowano wiele metod określania początkowych wag sieci, z których kilka przedstawiono w [4]. My zastosujemy najprostszą z nich - losowe ustalanie wag. Ich zakres przyjmiemy arbitralnie jako ± 0.1.

Funkcja inicjująca sieć jednowarstwową o S wejściach i K neuronach jest więc niezwykle prosta - jej treść zawiera zaledwie jedną linię kodu. Szablon funkcji wraz z komentarzami przedstawiono poniżej.

```
function [ W ] = init1 ( S , K )
% funkcja tworzy sieć jednowarstwową
% i wypełnia jej macierz wag wartościami losowymi
% z zakresu od -0.1 do 0.1
% parametry: S - liczba wejść do sieci
%            K - liczba neuronów w warstwie
% wynik:     W - macierz wag sieci

W = ...
```

Działanie funkcji można sprawdzić wpisując z linii poleceń:

```
>> W = init(5,3)
```

Jako wynik MATLAB powinien zwrócić macierz o wymiarach 5×3 wypełnioną liczbami losowymi z zakresu od -0.1 do 0.1.

W uproszczeniu można powiedzieć, że sztuczna sieć neuronowa jest po prostu zestawem macierzy. Sieć jednowarstwowa to zaledwie jedna macierz wag, a sieci wielowarstwowe to zbiór kilku macierzy. Jeśli znamy lub ustaliliśmy sposób konstrukcji tych macierzy, to macierze te definiują dokładnie strukturę sieci – liczbę wejść i wyjść (neuronów), oraz wiedzę sieci – wartości elementów macierzy. Jedyne, czego nie możemy się z niej domyślić, to funkcje aktywacji neuronów.

Zarówno symulacja działania sieci, jak i proces uczenia, to operacje na macierzach. Dlatego też idealnym środowiskiem dla sieci neuronowych jest pakiet MATLAB – pod warunkiem, że zadbamy o dostosowaną do tego środowiska implementację: jak największą liczbę wyrażeń macierzowych, a niewielką liczbę pętli `for`. Zobaczymy również, że łatwość sporządzania różnego rodzaju wykresów znacząco uprości nam wizualizację procesów uczenia sieci oraz ilustrację sposobu ich działania.

2.2. Symulacja działania sieci

Zadanie 2. *Napisz funkcję, która obliczy wyjście* $\mathbf{Y}$ *jednowarstwowej sieci o macierzy wag* $\mathbf{W}$ *jako odpowiedź na wektor sygnałów wejściowych* $\mathbf{X}$ *podany na jej wejście. Neurony sieci mają sigmoidalną funkcję aktywacji. Nagłówek funkcji dany jest jako:*

```
function [ Y ] = dzialaj1 ( W , X )
```

Z modelu neuronu przedstawionego na rys. 2.1 można wywnioskować, że obliczenie odpowiedzi neuronu y na zadany sygnał wejściowy $\mathbf{X}$ następuje w dwóch etapach:

1. obliczenie pobudzenia neuronu u, czyli sumy ważonej sygnałów wejściowych z wagami neuronu,
2. obliczenie wyjścia y jako wartości funkcji aktywacji w punkcie u.

Wagi przypisane poszczególnym wejściom neuronu odpowiadają przewodności poszczególnych połączeń między neuronami biologicznymi i pokazują (zgodnie ze swoją nazwą), jak ważny jest dany sygnał dla odpowiedzi neuronu, czy ma na nią wpływ pobudzający, czy hamujący. Pobudzenie neuronu jest sumą ważoną wszystkich sygnałów wejściowych, które można obliczyć ze wzoru

$$\begin{aligned} u &= \sum_{s=1}^{S} w_s \cdot x_s \\ &= w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_S \cdot x_S = \mathbf{W}^T \cdot \mathbf{X} \end{aligned} \tag{2.4}$$

jako iloczyn transponowanego wektora wag neuronu z wektorem jego wejść.

Jeśli nasze rozważania dotyczyłyby k-tego neuronu w warstwie sieci, to we wzorze (2.5) użylibyśmy oznaczeń u_k, w_k, s, gdzie $s = 1, 2, ..., S$, oraz $\mathbf{W_k}$.

W przypadku warstwy sieci składającej sie z K neuronów, pobudzenia poszczególnych neuronów zbiera się w jednym wektorze o wymiarach $K \times 1$

$$\mathbf{U} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_K \end{bmatrix}_{K \times 1} \tag{2.5}$$

Łatwo można sprawdzić, że wartość wektora pobudzeń całej warstwy sieci można obliczyć przy pomocy jednego iloczynu macierzy wag $\mathbf{W}$ z wektorem wejść $\mathbf{X}$, pamiętając o transpozycji jednej z nich.

Kolejnym etapem obliczania wyjścia sieci jest przejście sygnału pobudzenia neuronu przez jego funkcję aktywacji. Przywołując w dużym uproszczeniu pierwowzór biologiczny, każdy naturalny neuron wysyła swój sygnał wyjściowy dalej (do następnych neuronów) tylko wtedy, gdy jego pobudzenie przekroczy pewien określony próg.

W podstawowym modelu sztucznego neuronu - perceptronie McCullocha-Pittsa, jest to równoznaczne z przejściem sygnału pobudzenia (czyli ważonej sumy sygnałów wejściowych) przez progową funkcję aktywacji. W naszym przypadku rezygnacja z wejścia progowego oznacza przyjęcie progu równego 0, przez co funkcja progowa przyjmuje postać funkcji signum.

Użycie funkcji progowej jako analogii do biologicznego pierwowzoru neuronu jest dużym uproszczeniem. Przewodnictwo elektryczno-chemiczne sygnałów w neuronach biologicznych oparte jest na wielu skomplikowanych zjawiskach, w tym np. o charakterze częstotliwościowym, czy czasowo-przestrzennym, o których można przeczytać np. w [3].

Jako funkcji aktywacji można użyć wielu funkcji jednej zmiennej. Wspomniana skokowa funkcja aktywacji jest używana nadzwyczaj rzadko, przede wszystkim ze względu na swoją nieciągłość. Najczęściej używa się funkcji sigmoidalnej unipolarnej lub sigmoidalnej bipolarnej, danych odpowiednio wzorami

$$y = f_u(u) = \frac{1}{1+e^{-\beta u}} \tag{2.6}$$

$$y = f_b(u) = 2 \cdot f_u(u) - 1 = tgh(\beta u) \tag{2.7}$$

W niniejszym modelu sieci użyjemy funkcji unipolarnej. Aby obejrzeć jej kształt dla różnych wartości parametru β, proponujemy wykonanie następującego prostego ćwiczenia z linii poleceń programu MATLAB:

```
>> beta = 1
>> u = % od -5 do 5 co 0.01 ;
>> y = ...
>> plot (u,y)
>> % zamroź wykres
>> beta =  2, y = ... , plot (u,y)
>> beta =  5, y = ... , plot (u,y)
>> beta = 10, y = ... , plot (u,y)
```

Część poleceń wymaga uzupełnienia przez Czytelnika, aby miał okazję powtórzyć kilka zasad związanych z zapisem wyrażeń w środowisku MATLAB.

Jedną z głównych zalet funkcji sigmoidalnych jest łatwość obliczenia pochodnej w punkcie u, nawet bez znajomości wartości u. Pochodne funkcji aktywacji obu funkcji sigmoidalnych dadzą się wyrazić wzorami (odpowiednio)

$$f'_u(u) = \ldots = \beta \cdot y \cdot (1-y) \tag{2.8}$$

$$f_b'(u) \;=\; \ldots \;= \beta \cdot \left(1 - y^2\right) \tag{2.9}$$

Sprawdzenie kształtu pochodnej pozostawiamy Czytelnikowi.

Obliczenie wyjścia warstwy sieci neuronowej polega na obliczeniu wyjść wszystkich jej neuronów. Zakładamy, że wszystkie neurony w warstwie posiadają taką samą funkcję aktywacji – sigmoidalną unipolarną. Podobnie, jak w przypadku pobudzeń, wyjścia wszystkich neuronów danej warstwy sieci zostaną zebrane w jednym wektorze

$$\mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_K \end{bmatrix}_{K\times 1} = \begin{bmatrix} f(u_1) \\ f(u_2) \\ \vdots \\ f(u_K) \end{bmatrix} \tag{2.10}$$

Ostatecznie funkcja obliczająca wyjście całej warstwy sieci będzie zawierała zaledwie trzy linie kodu.

```
function [ Y ] = dzialaj1 ( W , X )
% funkcja symuluje działanie sieci jednowarstwowej
% parametry: W - macierz wag sieci
%            X - wektor wejść do sieci
%                sygnał podany na wejście
% wynik:     Y - wektor wyjść sieci
%                sygnał na wyjściu sieci

beta = 5 ;
U = ...  ;
Y = ...  ;
```

Przyjęto współczynnik nachylenia funkcji aktywacji za równy 5, czego rezultatem będzie średnia stromość zbocza tej funkcji. Pozostawiamy Czytelnikowi uzupełnienie poniższego szablonu funkcji w taki sposób, aby mnożenia macierzy zapisać możliwie zwięźle i wykorzystać fakt, że wbudowane funkcje numeryczne MATLABa operują na całych macierzach i wektorach. Odpowiedni zapis operacji mnożenia macierzy – unikanie zbędnych pętli `for` – ma wpływ nie tylko na elegancję zapisu, lecz również na wydajność. Funkcja `dzialaj` będą wykonywana wielokrotnie w pętlach podczas uczenia sieci. Wpływ wydajnej implementacji na szybkość procesu uczenia będzie szczególnie widoczny podczas uczenia sieci dwuwarstwowych o większych rozmiarach.

2.3. Uczenie sieci

Zanim przejdziemy do samego uczenia sieci oraz funkcji implementującej ten proces, przygotujemy skrypt, w którym:

· zdefiniujemy ciąg uczący dla sieci,
· zainicjujemy sieć losowymi wagami,
· przeprowadzimy uczenie sieci,
· porównamy działanie sieci przed i po uczeniu.

Skrypt o nazwie `test1` ma postać:

```
% ciąg uczący:
% przykłady: 1------2-----3         % wejścia sieci:
P = [        4      2    -1     ;   % we 1 - ile ma nóg
             0.01  -1     3.5    ;   % we 2 - czy żyje w wodzie
             0.01   2     0.01   ;   % we 3 - czy umie latać
            -1      2.5  -2     ;   % we 4 - czy ma pióra
            -1.5    2     1.5   ]   % we 5 - czy jest jajorodne

% przykłady: 1------2-----3         % żądane wyjścia sieci:
T = [        1      0     0     ;   % ssak
             0      1     0     ;   % ptak
             0      0     1     ]   % ryba

Wprzed = init1    ( 5 , 3 )
Yprzed = dzialaj1 ( Wprzed , P )
Wpo    = ucz1     ( Wprzed , P , T , 100 )
Ypo    = dzialaj1 ( Wpo    , P )
```

Ciąg uczący, będący przykładem problemu klasyfikacyjnego, jakie mogą rozwiązywać sieci neuronowe, został zaczerpnięty bezpośrednio z [5] (rozdz. 4.6). Dotyczy on rozpoznawania zwierząt. Na podstawie pięciu informacji:

· ile zwierzę ma nóg,
· czy żyje w wodzie,
· czy umie latać,
· czy ma pióra,
· czy jest jajorodne,

sieć rozpoznaje, czy jest to ssak, ptak, czy ryba.

Informacje te podane zostaną na pięć wejść jednowarstwowej sieci neuronowej złożonej z trzech neuronów. Każdy z nich będzie „odpowiedzialny" za jedną gromadę zwierząt, tzn. pierwszy ma się nauczyć rozpoznawać ssaki, drugi – ptaki, a trzeci – ryby.

Ciąg uczący zawarty jest w dwóch macierzach: **P** – która zawiera zestawy wejść dla kolejnych przykładów uczących, oraz **T** – odpowiadające tym przykładom wartości wyjść, jakimi powinna odpowiedzieć sieć. Jako jeden przykład należy więc traktować parę kolumn o tych samych indeksach z macierzy **P** i **T**.

W rzeczywistych zastosowaniach na ciąg uczący składa się wiele setek, a nawet tysięcy przykładów uczących. Na temat zdolności generalizacyjnych sieci neuronowych oraz rozmiarów zbiorów uczących niezbędnych do uzyskania dobrej generalizacji napisano wiele publikacji, w szczególności związanych z teorią wymiaru Vapnika-Chervonenkisa. Zwięzłe podsumowanie tych prac można znaleźć w [4]. W przypadku klasyfikacji zwierząt na trzy grupy, należałoby zebrać odpowiednio wiele (np. po 100) przykładów ssaków, ptaków i ryb. Na ich podstawie każdy z trzech neuronów naszej prostej sieci określiłby w procesie nauki swego rodzaju uśredniony prototyp kategorii, którą ma rozpoznawać. W naszym przykładzie – z racji jego prostoty – zostały zdefiniowane jedynie 3 przykłady, po jednym na każdą kategorię, jaką ma rozpoznawać nasza sieć. Można je traktować jako z góry ustalone uśrednione „prototypy". Dobór poszczególnych cech wzorcowych przedstawicieli każdej klasy jest w oryginale ([5]) argumentowany w następujący sposób (porównaj z wartościami wektora **P**):

- ssak – ma 4 nogi; czasem żyje w wodzie, ale nie jest to dla niego typowe (foka, delfin), czasem umie latać (nietoperz), ale również nie jest to dla niego typowe; nie ma piór; jest żyworodny i jest to dość ważne;
- ptak – ma 2 nogi; nie żyje w wodzie (co najwyżej na wodzie); zwykle umie latać i jest to bardzo ważna jego cecha; jak również ta, że ma pióra; oraz że jest jajorodny;
- ryba – nie ma nóg (płetwy się nie liczą); żyje w wodzie i jest to jej najważniejsza cecha; nie umie latać (choć niektóre próbują); nie jest pokryta piórami; jest zwykle jajorodna, ale nie jest to aż tak ważne, jak u ptaków.

Można zauważyć, że jedynie pierwsza cecha ma charakter ilościowy. Nie jest ona tu w żaden sposób skalowana, ani normalizowana – nie jest to konieczne przy tak prostym zadaniu, jednak w praktyce bardzo często stosowane. O wiele łatwiejszym zadaniem dla sieci jest bowiem analiza wpływu poszczególnych wejść na żądany wynik, gdy wszystkie wejścia przyjmują wartości z podobnego zakresu, niż gdy jedno wejście ma zakres np. od 0 do 1, a drugie od 1000 do 5000. Kolejne cztery wejścia neuronu są typu logicznego. Również i one zostały w oryginale potraktowane dosyć swobodnie. Wartości logiczne koduje się zwykle albo jako 0 i 1, albo -1 i 1 (odpowiednio: fałsz i prawda). Tu autor zdecydował się na podkreślenie znaczenia niektórych cech przez wyjście poza ten zakres.

W prezentowanym przykładzie wykorzystano trzy neurony wyjściowe do klasyfikacji sygnałów wejściowych na trzy klasy. Wystarczy spojrzeć na wektor żą-

danych sygnałów wyjściowych **T**, aby dostrzec, że każdy neuron wyjściowy odpowiada za jedną klasę, tzn. gromadę zwierząt – przez rozpoznanie danej klasy rozumiemy wystąpienie jedynki na wyjściu przypisanego do niej neuronu i zer na wyjściach pozostałych neuronów. W praktyce spodziewamy się oczywiście wartości tylko zbliżonych do powyższych.

Przedstawiony sposób kodowania klas nazywany jest metodą „jeden z N" i jest jednym z kilku możliwych. Do oznaczenia trzech klas wystarczyłyby bowiem dwa wyjścia. Moglibyśmy próbować nauczyć dwa neurony reagowania następującymi wartościami wyjść: 01 dla ssaków, 10 dla ptaków, 11 dla ryb, zaś 00 oznaczałoby dodatkową klasę – „brak decyzji". Zaproponowany sposób kodowania „jeden z N" jest jednak łatwiejszy do nauczenia przez sieć. Jest również łatwiejszy w interpretacji – wielkości poszczególnych wyjść można bowiem odczytywać jako stopień „przekonania" sieci do danej klasy, czyli prawdopodobieństwo rozpoznania danej klasy. Za rozpoznaną klasę można uznać tę przypisaną do neuronu, który najsilniej zareagował na sygnał wejściowy.

Gdy wartości na wyjściach żadnego neuronu nie przekraczają założonego progu, rezultatem działania sieci może być „brak decyzji". Podobny wynik możemy przyjąć w przypadku, gdy kandydatów do zwycięstwa jest dwóch lub więcej (wysokie poziomy wyjść na kilku neuronach i mała różnica między nimi). Postępowanie takie jest często stosowane w zadaniach klasyfikacji, szczególnie w takich zastosowaniach (np. medycznych), w których złe rozpoznanie jest gorsze w skutkach niż wstrzymanie się od decyzji.

Przedstawiony wyżej skrypt można uruchomić, aby przekonać się, jak działa sieć dopiero co zainicjowana wartościami losowymi, a także po to, by sprawdzić poprawność dotychczas napisanych funkcji. Należy oczywiście dwie ostatnie linie kodu potraktować jako komentarz.

Zauważmy jeszcze, że w zaproponowanym skrypcie sprawdzenie wszystkich przykładów z ciągu uczącego odbywa się „za jednym zamachem" – przez podanie na wejścia sieci od razu całej macierzy **P**. Może się wydawać zaskakujące, że wynikiem działania funkcji `dzialaj1` jest macierz (a nie wektor, jak należało by się spodziewać), której kolumny zawierają odpowiedzi sieci na wszystkie trzy przykłady. Dzięki temu, łatwo można porównać tę macierz z macierzą żądanych odpowiedzi **T**. Takie postępowanie jest możliwe tylko dzięki prostocie sieci jednowarstwowej oraz dzięki pominięciu w modelu sieci wejścia progowego (biasu). W zadaniach dotyczących sieci dwuwarstwowych, będziemy już musieli po kolei podawać przykłady na wejścia sieci i odczytywać jej odpowiedzi.

Zadanie 3. *Napisz funkcję, która będzie uczyć sieć jednowarstwową o danej macierzy wag* $\mathbf{W}$ *przez zadaną liczbę epok n na ciągu uczącym podanym w postaci macierzy przykładów* $\mathbf{P}$ *(wejścia) i* $\mathbf{T}$ *(żądane wyjścia). Sieć zbudowana jest z neuronów o sigmoidalnej funkcji aktywacji. Wynikiem działania funkcji powinna być macierz wag nauczonej sieci. Nagłówek funkcji określony jest następująco:*

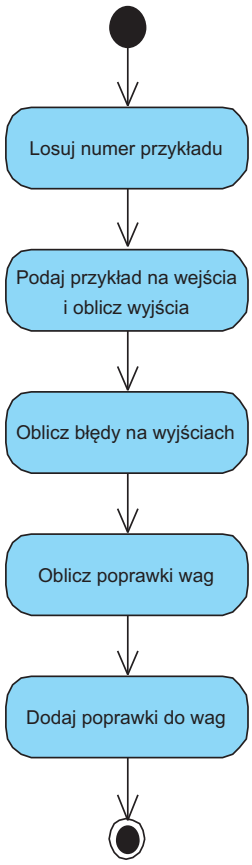
```
function [ Wpo ] = ucz1 ( Wprzed , P , T , n )
```

Uczenie sieci polega na wielokrotnym „pokazywaniu" jej kolejnych przykładów z ciągu uczącego. Podczas każdego takiego pokazu wyznaczana jest odpowiedź sieci, po czym jest ona porównywana z żądaną odpowiedzią dla danego przykładu, zawartą w ciągu uczącym. Na podstawie obliczonej różnicy, czyli błędu popełnionego przez sieć, ustala się następnie poprawki wszystkich wag sieci, zgodnie z przyjętym algorytmem uczenia. Poprawki te dodaje się do wag po to, aby w następnych krokach, gdy sieci zostanie przedstawiony ten sam przykład, jej odpowiedź była bliższa żądanej odpowiedzi z ciągu uczącego.

Schemat pojedynczego kroku uczenia sieci jednowarstwowej można przedstawić na diagramie czynności języka UML, przedstawionym na rys. 2.4.

W tym ćwiczeniu nie będziemy zastanawiać się nad strategiami dotyczącymi wybierania przykładów uczących, ani liczby przykładów pokazywanych w pojedynczym kroku uczenia sieci. W dalszej części książki przedyskutujemy pojęcia kroku i epoki uczenia oraz kwestię równomiernego pokazywania przykładów z ciągu uczącego w każdej epoce. Pojedynczy krok uczenia, przedstawiony na rys. 2.4, obejmuje:

- wylosowanie numeru przykładu do pokazania w danym kroku ,
- podanie przykładu na wejścia sieci i obliczenie jej wyjść – czyli wybranie odpowiedniej kolumny z macierzy $\mathbf{P}$ zbioru uczącego i obliczenie wartości wyjść sieci,
- porównanie wyjść sieci z żądanymi wyjściami dla danego przykładu – czyli z odpowiednią kolumną macierzy $\mathbf{T}$,
- obliczenie macierzy poprawek wag, zgodnie z wybranym algorytmem uczenia,
- dodanie macierzy poprawek do macierzy wag sieci.



Rysunek 2.4. Pojedynczy krok uczenia sieci jednowarstwowej

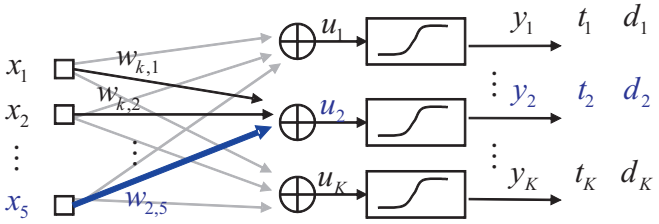
Szkielet funkcji można zapisać następująco:

```
function [ Wpo ] = ucz1 ( Wprzed , P , T , n )
% funkcja uczy sieć jednowarstwową
% na podanym ciągu uczącym (P,T)
% przez zadaną liczbę epok (n)
% parametry: Wprzed - macierz wag sieci przed uczeniem
%            P      - ciąg uczący - przykłady - wejścia
%            T      - ciąg uczący - żądane wyjścia
%                                   dla poszczególnych przykładów
%            n      - liczba epok
% wynik:     Wpo    - macierz wag sieci po uczeniu

liczbaPrzykladow = size ( P , 2 ) ;
W = Wprzed ;

for i = 1 : n ,

   % losuj numer przykładu
   nrPrzykladu = ...
   % podaj przykład na wejścia i oblicz wyjścia
   X = ...
   Y = ...
   % oblicz błędy na wyjściach
   D = ...
   % oblicz poprawki wag
   dW = ...
   % dodaj  poprawki do wag
   W = ...

end % i to wszystko n razy

Wpo = W ;
```

Uzupełnienie powyższego schematu nie powinno sprawić Czytelnikowi większego problemu. Jedynym dotychczas nie wyjaśnionym zagadnieniem jest obliczanie poprawek wag neuronów. Wyczerpujący przegląd algorytmów uczenia sieci można znaleźć w [4, 6]. W niniejszym ćwiczeniu użyjemy niezbyt ścisłej, ale intuicyjnej metody obliczania poprawek wag.

Rys. 2.5 przedstawia model sieci jednowarstwowej o 5 wejściach i K wyjściach, wraz z sygnałami znanymi podczas jednego kroku uczenia. Na wejścia sieci podany został losowo wybrany przykład z ciągu uczącego w postaci wektora $\mathbf{X} = [\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_5}]^{\mathbf{T}}$. Obliczone zostały pobudzenia oraz wyjścia wszystkich neuronów – odpowiednio wektory $\mathbf{U} = [\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_K}]^{\mathbf{T}}$. Podczas porównania

z odpowiednim wektorem żądanych wyjść $\mathbf{T} = [\mathbf{t_1}, \mathbf{t_2}, \ldots, \mathbf{t_K}]^\mathbf{T}$ obliczony został wektor błędów wszystkich neuronów $\mathbf{D} = [\mathbf{d_1}, \mathbf{d_2}, \ldots, \mathbf{d_K}]^\mathbf{T}$. Zastanówmy się teraz, od których wielkości powinna zależeć poprawka wagi np. neuronu drugiego prowadząca do wejścia piątego, czyli $w_{2,5}$.



Rysunek 2.5. Uczenie neuronu - obliczanie poprawek wag

Po chwili namysłu dojdziemy do wniosku, że poprawka tej wagi powinna zależeć od błędu na wyjściu neuronu, czyli od d_2. Jednak od tego błędu zależeć będą poprawki wszystkich wag neuronu drugiego. Zatem poprawka wagi $w_{2,5}$ powinna zależeć przynajmniej od jeszcze jednej wielkości – takiej, która pozwoli zróżnicować poprawki poszczególnych wag danego neuronu. Tą wielkością jest sygnał wejściowy, do którego prowadzi dana waga, w naszym przypadku x_5.

Intuicyjne i logiczne wyjaśnienie powyższej reguły można znaleźć w [5].

- Wagi zmieniane są tym silniej, im większy błąd został popełniony na danym neuronie. Reguła ta nie wymaga dalszego uzasadniania. Dzięki niej wagi tych neuronów, które nie popełniły błędów na danym przykładzie uczącym, nie są poprawiane – wszak działają one prawidłowo. Ponadto, sieć w praktyce sama przerywa uczenie, gdy jest już dobrze wytrenowana i drobne błędy skutkują wtedy jedynie „kosmetycznymi" zmianami wag.
- Im większy sygnał pojawił się na wejściu, do którego prowadzi dana waga, tym bardziej jest ona poprawiana. Również i tu logiczne uzasadnienie nie budzi wątpliwości. Jeśli sygnał wyjściowy był poprawny, to i poprawka danej wagi będzie niewielka (dzięki poprzedniej zasadzie). Jeśli zaś był błędny, to ten sygnał wejściowy, który był większy od innych, miał większy wpływ na powstanie tego błędu. Stąd waga prowadząca do wejścia, na którym ten sygnał się pojawił, powinna być bardziej skorygowana.

W oparciu o powyższe dwie reguły, można już sformułować wzór na poprawkę wagi $w_{k,s}$ jako

$$dw_{k,s} = \eta \cdot d_k \cdot x_s \tag{2.11}$$

Widoczny w nim współczynnik uczenia η przyjmuje zwykle wartości mniejsze od 1. Można więc powiedzieć, że służy do „złagodzenia" decyzji hipotetycznego na-

uczyciela sieci. W naszym przykładzie przyjmiemy wartość współczynnika równą 0.1. W dalszej części skryptu będziemy badać jego wpływ na efektywność procesu uczenia.

Poprawki wszystkich wag sieci należy obliczyć zgodnie ze wzorem (2.11). Macierz poprawek zawiera poprawki odpowiednich wag sieci i ma wymiar taki sam, jak macierz wag, czyli $S \times K$. Obliczenie całej macierzy poprawek może zostać dokonane za pomocą jednego mnożenia – pamiętajmy, że w środowisku MATLAB unikamy używania pętli `for`. Pozostawiamy Czytelnikowi sprawdzenie, w jakiej kolejności należy pomnożyć odpowiednie macierze.

Po uzupełnieniu najważniejszej linii kodu funkcji `ucz1` – tej, w której obliczane są poprawki wag, sieć powinna być w stanie nauczyć się rozpoznawania zwierząt. Efektem wywołania skryptu `test1` powinny być odpowiedzi programu MATLAB zbliżone do poniższych:

```
P =    4.0000    2.0000   -1.0000
       0.0100   -1.0000    3.5000
       0.0100    2.0000    0.0100
      -1.0000    2.5000   -2.0000
      -1.5000    2.0000    1.5000

T =    1     0     0
       0     1     0
       0     0     1

Wprzed =    0.0629   -0.0805   -0.0685
            0.0812   -0.0443    0.0941
           -0.0746    0.0094    0.0914
            0.0827    0.0915   -0.0029
            0.0265    0.0930    0.0601

Yprzed =    0.6564    0.6848    0.6162
            0.0592    0.8298    0.3566
            0.1423    0.5800    0.9223

Wpo =    0.1411   -0.1499   -0.3363
        -0.1368   -0.2648    0.2041
        -0.2945    0.0681   -0.0240
        -0.0551    0.3208   -0.1426
        -0.4102    0.1400    0.0191

Ypo =    0.9979    0.0035    0.0035
         0.0035    0.9973    0.0024
         0.0021    0.0020    0.9989
```

Dowodem na to, że sieć nauczyła się tego prostego zadania jest widoczna wyżej macierz `Ypo`, zbliżona do macierzy żądanej `T`. O wartości rzeczywistych sieci używanych w „poważnych" zastosowaniach, świadczą jednak nie jej wyniki na przykładach uczących, lecz zdolności uogólniania zdobytej wiedzy na nowe przypadki danych wejściowych, nie widziane przez sieć podczas uczenia. Szersza dyskusja tego problemu będzie zamieszczona w dalszej części książki.

Obecnie sprawdzimy zdolności generalizacyjne naszej prostej sieci na kilku przykładach. W tym celu wystarczy zdefiniować wektor wejściowy sieci opisujący cechy jakiegoś zwierzęcia zgodnie z założonym opisem wejść sieci, podać go na wejścia sieci, obliczyć wyjście i sprawdzić poprawność odpowiedzi. Pamiętajmy, że wartości na wyjściach poszczególnych neuronów możemy interpretować jako prawdopodobieństwo rozpoznania przez sieć danej klasy. Autor niniejszej pracy został na przykład dość zdecydowanie rozpoznany jako ssak:

```
>> ja = [ 2 ;    % mam dwie nogi
          0 ;    % słabo pływam
          0 ;    % rzadko latam
          0 ;    % nie obrosłem w piórka
          0 ] ;  % nie jestem jajorodny
>> odp = dzialaj1 ( Wpo , ja )
   odp =    0.8039
            0.1825
            0.0335
```

Uwzględnienie w wektorze wejściowym „statystycznego" człowieka nadprzyrodzonych zdolności niektórych osób zmniejszyło przekonanie sieci do właściwej klasy:

```
>> czlowiek = [ 2   ;    % ile ma nóg
                0.2 ;    % czy żyje w wodzie - M. Phelps
                0.2 ;    % czy umie latać    - A. Małysz
                0.1 ;    % czy ma pióra      - Winnetou
                0   ] ;  % czy jest jajorodne
>> odp = dzialaj1 ( Wpo , czlowiek )
   odp =    0.7215
            0.1772
            0.0372
```

Wyzwanie, jakim dla sieci było rozpoznanie nietypowych ssaków, skończyło się niestety albo niepewnością (co prawda w przypadku nietoperza kategoria *ssak* wygrała, ale poziom wyjścia nie przekroczył 0.5 i mało się różnił od poziomu wyjścia neuronu rozpoznającego ptaki), albo pomyłką (delfin został uznany za rybę).

```
>> nietoperz = [ 2 ; 0 ;  1   ;  0 ;  0 ] ;  % ssak, lata
>> delfin    = [ 0 ; 1 ;  0.1 ;  0 ;  0 ] ;  % ssak, żyje w wodzie
>> odp = dzialaj1 ( Wpo , [ nietoperz , delfin ] )
   odp =    0.4845    0.3034
            0.2389    0.2158
            0.0298    0.7327
```

Sprawdzenie innych przykładów zwierząt (zarówno typowych, jak i nietypowych, np. węża, czy strusia) pozostawiamy Czytelnikom.

Celem ćwiczeń zawartych w tym rozdziale było przystępne przedstawienie budowy i podstawowych zasad działania oraz uczenia sieci jednowarstwowych. Pominięto w nich wiele szczegółów, jak choćby wcześniejsze zakończenie procesu uczenia po osiągnięciu zakładanego poziomu błędu, sporządzenie wykresu błędu średniokwadratowego w kolejnych epokach, pokazywanie kilku przykładów uczących w jednej epoce, czy analizę granic decyzyjnych neuronów. Do tych zagadnień wrócimy później po umówieniu podstaw działania i uczenia sieci dwuwarstwowych

Rozdział 3

Sieci dwuwarstwowe MLP

Podobnie, jak w przypadku sieci jednowarstwowych, napiszemy trzy funkcje, służące do inicjalizacji, symulacji oraz uczenia sieci o dwóch warstwach:

- `init2`
- `dzialaj2`
- `ucz2`

Podczas ich implementacji skorzystamy z doświadczeń zdobytych podczas implementacji sieci jednowarstwowych. W funkcji służącej do uczenia sieci skupimy się już nie na samych poprawkach wag, lecz na podstawowym, najczęściej stosowanym algorytmie uczenia sieci wielowarstwowych - metodzie propagacji wstecznej błędu (ang. *backpropagation*).

Definiowanie ciągu uczącego, sprawdzenie sieci przed i po uczeniu oraz samo uczenie zaimplementujemy w skrypcie o nazwie `test2`. W tym rozdziale wykorzystamy sieć dwuwarstwową do rozwiązania jednego z podstawowych zadań klasyfikacyjnych - problemu XOR.

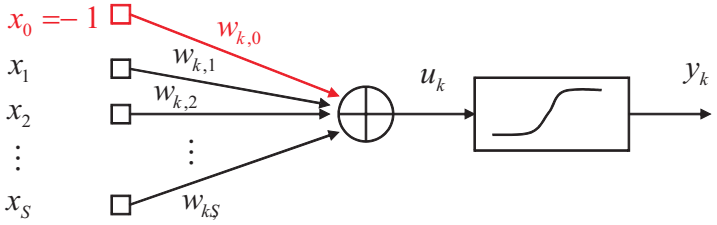
3.1. Inicjalizacja sieci

Zadanie 4. *Napisz funkcję, która utworzy macierz wag dwuwarstwowej sieci o S wejściach, K_1 neuronach w warstwie pierwszej i K_2 neuronach w warstwie drugiej, oraz wypełni ją liczbami losowymi z przedziału od -0.1 do 0.1. Nagłówek funkcji jest następujący:*

```
function [ W1 , W2 ] = init2 ( S , K1 , K2 )
```

W poprzednim rozdziale, podczas implementacji sieci jednowarstwowych korzystaliśmy z uproszczonego modelu neuronu — pozbawionego wejścia progowego, zwanego *biasem*. Począwszy od tego rozdziału, będziemy używać pełnego modelu neuronu, pokazanego na rys. 3.1.

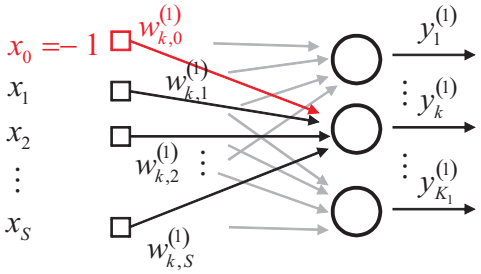
Przedstawia on budowę k-tego neuronu w warstwie sieci. Jedyną różnica w stosunku do neuronu z rys. 2.1 jest istnienie wejścia o indeksie 0, przyjmującego



Rysunek 3.1. Pełny model pojedynczego neuronu – z wejściem progowym (tzw. *biasem*)

zawsze wartość -1, do którego prowadzi waga neuronu o indeksie 0. Waga ta jest odpowiednikiem progu aktywacji neuronu biologicznego – po przekroczeniu przez ważone sygnały wejściowe tej granicy, neuron wysyła sygnał do dalszych neuronów, w przeciwnym razie pozostaje w spoczynku. W modelu matematycznym znaczenia progu można się dopatrywać albo w przesunięciu funkcji aktywacji wzdłuż osi x, albo w możliwości przesuwania granic decyzyjnych neuronu, co zostanie pokazane później.

Wszystkie neurony w obu warstwach sieci posiadają wejścia progowe. Istnienie *biasu* jest „wewnętrzną sprawą" każdego neuronu – wejścia o indeksach zerowych są niewidoczne dla użytkownika sieci. Funkcje symulujące działanie sieci będą same musiały zadbać o rozszerzenie wektora wejściowego o dodatkowe wejście, zaś macierze wag będą musiały zostać uzupełnione o dodatkowy wiersz. Próg neuronu, podobnie jak wagi, podlega uczeniu. Włączenie go do wektora wag jest zabiegiem czysto matematycznym, w naszym przypadku ułatwiającym operacje macierzowe. Nie zawsze jednak jest on stosowany. Na przykład sieci neuronowe zaimplementowane w pakiecie MATLAB Neural Network Toolbox korzystają z osobnych macierzy wag i wektorów wejść progowych (*biasów*).



Rysunek 3.2. Pierwsza warstwa sieci dwuwarstwowej

Pierwszą warstwę sieci przedstawia rys. 3.2. Wszystkie wielkości odnoszące się do neuronów warstwy pierwszej będziemy teraz oznaczać indeksem górnym (1). Wektor sygnałów wejściowych dla całej sieci, zawierający wektor cech opisują-

cych badany obiekt lub zjawisko, będziemy oznaczać przez $\mathbf{X}$. Po rozszerzeniu o wejście zerowe $x_0 = -1$ stanie się on wektorem wejściowym pierwszej warstwy $\mathbf{X^{(1)}} = [\mathbf{x_0}, \mathbf{x_1}, \ldots, \mathbf{x_S}]^{\mathbf{T}}$, o wymiarze $S + 1 \times 1$. Macierz wag pierwszej warstwy sieci jest zbudowana, podobnie, jak w przypadku sieci jednowarstwowej, z wektorów kolumnowych wag kolejnych neuronów. Tym razem jednak, pierwszy jej wiersz, indeksowany przez nas jako zerowy (drugi dolny indeks), zawiera wagi poszczególnych neuronów do wejścia progowego, czyli krótko ich *biasy*.

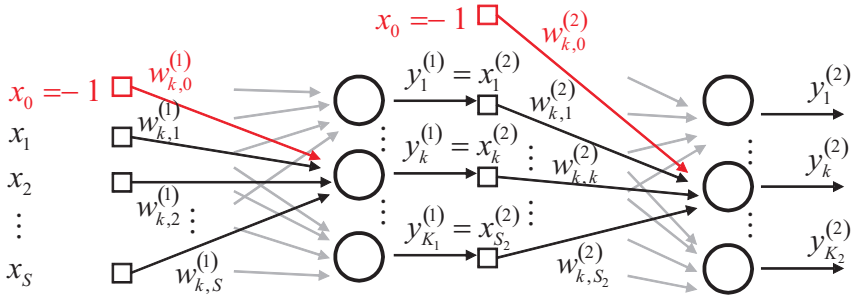
$$\mathbf{W}^{(1)} = \left[\begin{array}{cccc} \overline{\mathbf{W}}_1^{(1)} & \overline{\mathbf{W}}_2^{(1)} & \ldots & \overline{\mathbf{W}}_{K_1}^{1} \end{array} \right] = \left[\begin{array}{cccc} w_{1,0}^{(1)} & w_{2,0}^{(1)} & \cdots & w_{K_1,0}^{(1)} \\ w_{1,1}^{(1)} & w_{2,1}^{(1)} & \cdots & w_{K_1,1}^{(1)} \\ w_{1,2}^{(1)} & w_{2,2}^{(1)} & \cdots & w_{K_1,2}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,S}^{(1)} & w_{2,S}^{(1)} & \cdots & w_{K_1,S}^{(1)} \end{array} \right]_{S+1 \times K_1} \tag{3.1}$$

Przy implementacji następnych funkcji trzeba będzie pamiętać o tym, że program MATLAB indeksuje wiersze macierzy od 1, a nie on 0, jak np. język C++.

Sieć dwuwarstwowa jest kaskadowym połączeniem warstwy pierwszej i drugiej. Pomimo tego, że jedną z głównych i podkreślanych zalet sieci neuronowych jest równoległe przetwarzanie danych, to jednak architekturę rozważanych właśnie wielowarstwowych sieci jednokierunkowych (*Multi Layer Perceptron* – MLP) możemy określić jako równoległo-szeregową. W jednej warstwie rzeczywiście sygnały przetwarzane są równolegle przez wszystkie neurony, jednak z jednej warstwy do kolejnej przesyłane są szeregowo. Zobaczymy później, że kolejne warstwy po kolei upraszczają zadanie, które ma do wykonania sieć. Wiele badań dowodzi, że jest to sposób działania zbliżony do procesów, zachodzących w naszym mózgu ([1]), szczególnie tych związanych z percepcją. W mózgu występuje jednak dodatkowo bardzo wiele połączeń zwrotnych. Sprawiają one, ze sieci sieci biologiczne są układami dynamicznymi o wielkich możliwościach przetwarzania i kojarzenia sygnałów. Sieci rekurencyjne wykraczają jednak daleko poza zakres niniejszej książki.

Pełny model sieci dwuwarstwowej przedstawia rys. 3.3. Wektor wyjść warstwy pierwszej, po rozszerzeniu o wejście progowe, staje się wektorem wejść do warstwy drugiej. Macierz wag warstwy drugiej, konstruowana w analogiczny sposób, jak macierz wag warstwy pierwszej, ma wymiar $K_1 + 1 \times K_2$. Wszystkie wielkości odnoszące się do neuronów drugiej warstwy sieci oznaczamy indeksem górnym (2).

Funkcja inicjująca sieć dwuwarstwową o S wejściach, K_1 neuronach w pierwszej i K_2 neuronach w drugiej warstwie zawiera zaledwie dwie linie kodu, w których



Rysunek 3.3. Obie warstwy sieci dwuwarstwowej

tworzone są macierze losowych wag o odpowiednich wymiarach.

```
function [ W1 , W2 ] = init2 ( S , K1 , K2 )
% funkcja tworzy sieć dwuwarstwową
% i wypełnia jej macierze wag wartościami losowymi
% z zakresu od -0.1 do 0.1
% parametry: S  - liczba wejść do sieci (wejść warstwy 1)
%            K1 - liczba neuronów w warstwie 1
%            K2 - liczba neuronów w warstwie 2  (wyjść sieci)
% wynik:     W1 - macierz wag warstwy 1 sieci
%            W2 - macierz wag warstwy 2 sieci

W1 = ...
W2 = ...
```

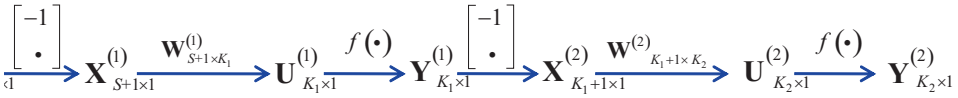
3.2. Symulacja działania sieci

Zadanie 5. *Napisz funkcję, która obliczy wyjście* $\mathbf{Y}$ *dwuwarstwowej sieci o macierzach wag* $\mathbf{W}_1$ *i* $\mathbf{W}_2$ *jako odpowiedź na wektor sygnałów wejściowych* $\mathbf{X}$ *podany na jej wejście. Neurony obu warstw sieci mają sigmoidalną unipolarną funkcję aktywacji o współczynniku nachylenia równym 5. Nagłówek funkcji dany jest jako:*

```
function [ Y1 , Y2 ] = dzialaj2 ( W1 , W2 , X )
```

Jak widać, wynikiem funkcji są wektory wyjściowe obu warstw sieci. W zasadzie funkcja powinna podawać jedynie wyjścia drugiej warstwy, jako odpowiedź całej sieci. Jednak, jak zobaczymy później, wektor wyjść warstwy pierwszej, będzie nam potrzebny podczas uczenia sieci.

Obliczanie wyjścia sieci dwuwarstwowej przedstawione jest schematycznie na rys. 3.4. Dla ułatwienia przy macierzach podano ich rozmiary. Po zaimplemento-



Rysunek 3.4. Obliczanie wyjścia sieci dwuwarstwowej

waniu funkcji `dzialaj1` dla sieci jednowarstwowej, uzupełnienie poniższego szkieletu funkcji nie powinno stanowić dla Czytelnika większego problemu.

```
function [ Y1 , Y2 ] = dzialaj2 ( W1 , W2 , X )
% funkcja symuluje działanie sieci dwuwarstwowej
% parametry: W1 - macierz wag pierwszej warstwy sieci
%            W2 - macierz wag  drugiej  warstwy sieci
%            X  - wektor wejść do sieci
%                 sygnał podany na wejście ( sieci / warstwy 1 )
% wynik:     Y1 - wektor wyjść warstwy 1   ( przyda się podczas uczenia )
%            Y2 - wektor wyjść warstwy 2   / sieci
%                 sygnał na wyjściu sieci

beta = 5 ;
X1 = ...
U1 = ...
Y1 = ...
X2 = ...
U2 = ...
Y2 = ...
```

3.3. Uczenie sieci

Podobnie, jak w przypadku uczenia sieci jednowarstwowej, zanim napiszemy funkcję implementującą proces uczenia sieci, przygotujemy skrypt, w którym zdefiniujemy ciąg uczący, przeprowadzimy uczenie sieci oraz sprawdzimy jej działanie przed i po uczeniu. Tym razem problemem, którego rozwiązywania będzie musiała się nauczyć sieć, będzie zadanie klasyfikacji czterech punktów na płaszczyźnie przy pomocy funkcji logicznej XOR. Definicję ciągu uczącego dla tego zadania, inicjalizację, uczenie oraz sprawdzenie sieci przed i po uczeniu zawiera skrypt o nazwie `test2`, którego treść jest następująca:

```
                                % zadanie XOR
P = [    0    0    1    1  ;  % wejścia sieci
         0    1    0    1  ]
T = [    0    1    1    0  ]  % żądane wyjście sieci
```

```
[ W1przed , W2przed ] = init2 ( 2 , 2 , 1 )

               % sprawdzenie działania sieci przed uczeniem
[ Y1 , Y2a ] = dzialaj2 ( W1przed , W2przed , P (:,1) ) ;
[ Y1 , Y2b ] = dzialaj2 ( W1przed , W2przed , P (:,2) ) ;
[ Y1 , Y2c ] = dzialaj2 ( W1przed , W2przed , P (:,3) ) ;
[ Y1 , Y2d ] = dzialaj2 ( W1przed , W2przed , P (:,4) ) ;
Yprzed = [ Y2a , Y2b , Y2c , Y2d ]

[ W1po , W2po ] = ucz2 ( W1przed , W2przed , P , T , 2000 )

               % sprawdzenie działania sieci po uczeniu
[ Y1 , Y2a ] = dzialaj2 ( W1po , W2po , P (:,1) ) ;
[ Y1 , Y2b ] = dzialaj2 ( W1po , W2po , P (:,2) ) ;
[ Y1 , Y2c ] = dzialaj2 ( W1po , W2po , P (:,3) ) ;
[ Y1 , Y2d ] = dzialaj2 ( W1po , W2po , P (:,4) ) ;

Ypo = [ Y2a , Y2b , Y2c , Y2d ]
```

Ciąg uczący składa się teraz z czterech przykładów, zawartych w kolejnych parach kolumn macierzy $\mathbf{P}$ i $\mathbf{T}$. W ćwiczeniu użyjemy najmniejszej możliwej sieci, która jest w stanie rozwiązać problem XOR. Jest to sieć o dwóch wejściach, dwóch neuronach w warstwie ukrytej i jednym neuronie wyjściowym. W dalszej części rozdziału przeanalizujemy możliwości pojedynczych neuronów oraz sieci wielowarstwowych w zakresie kształtowania granic decyzyjnych, a wtedy dowiemy się, dlaczego właśnie taka architektura jest konieczna do rozwiązania tego problemu. Oczywiście w powyższym skrypcie wystarczy zmienić tylko jeden parametr, aby móc przebadać zachowanie sieci o innej liczbie neuronów w pierwszej warstwie.

Jak wspomniano wcześniej, w sieciach dwuwarstwowych nie można już przy pomocy jednego polecenia sprawdzić działania sieci na całym zbiorze uczącym, czy testującym. Jak widać, trzeba po kolei podawać przykłady na wejście sieci, a otrzymane wyjścia (tylko warstwy wyjściowej) zebrać w wektor odpowiedzi i porównać z wektorem wartości żądanych $\mathbf{T}$.

Zadanie 6. *Napisz funkcję, która będzie uczyć sieć dwuwarstwową o danych macierzach wag* $\mathbf{W_1}$ *i* $\mathbf{W_2}$ *przez zadaną liczbę epok n na ciągu uczącym podanym w postaci macierzy przykładów* $\mathbf{P}$ *(wejścia) i* $\mathbf{T}$ *(żądane wyjścia). Sieć zbudowana jest z neuronów o sigmoidalnej funkcji aktywacji. Wynikiem działania funkcji powinny być macierze wag nauczonej sieci. Nagłówek funkcji określony jest następująco:*

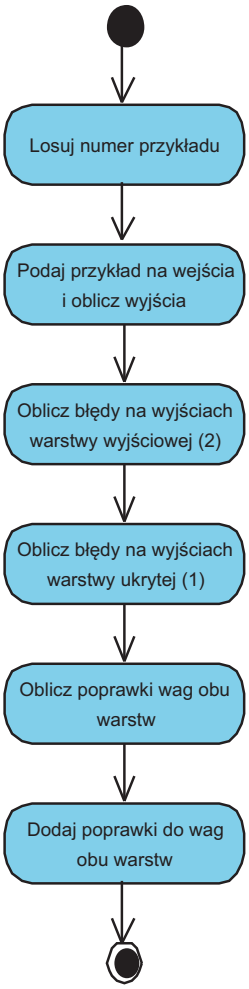
```
function [ W1po , W2po ] = ucz1 ( W1przed , W2przed , P , T , n )
```

Uczenie sieci dwuwarstwowej przebiega w podobny sposób, jak sieci jednowarstwowej. Pojedynczy krok uczenia również składa się z dwóch etapów – możemy je nazwać fazą obliczania wyjścia oraz fazą obliczania poprawek wag, lub stosując angielskie nazewnictwo sugerujące kierunek przepływu sygnałów – fazą *feedforward* i fazą *backpropagation*. Diagram aktywności pojedynczego kroku uczenia sieci dwuwarstwowej przedstawia rys. 3.5.

Z kolei rys. 3.6 akcentuje kierunek przepływu sygnałów w poszczególnych fazach i pokazuje kolejność obliczania kolejnych wielkości podczas kroku uczenia. Dla wygody podano na nim również wymiary poszczególnych macierzy i wektorów. Górna część schematu ilustruje fazę obliczania wyjścia sieci dla wylosowanego przykładu. Odpowiada ona dokładnie zaprezentowanemu wcześniej schematowi dla funkcji `dzialaj2` (na wywołaniu tej właśnie funkcji polega faza *feedforward*). Zauważmy, że w tej fazie sygnały w sieci są przesyłane od wejść do wyjść.

Dolna część rysunku przedstawia fazę obliczania poprawek wag. Jak widać, można to robić w ściśle określonej kolejności — od ostatniej warstwy sieci (wyjściowej) w kierunku wejścia sieci. Wynika to z faktu, że jedynie dla warstwy wyjściowej jesteśmy w stanie obliczyć dokładne wartości błędów na wyjściach neuronów, czyli wektor $\mathbf{D^{(2)}}$ (są to jednocześnie błędy całej sieci). W tym celu wystarczy porównać wartości wyjść neuronów (wektor $\mathbf{Y^{(2)}}$) z żądanymi wartościami wyjść sieci (wektor $\mathbf{T}$).

Znając wartości błędów dla warstwy wyjściowej, jesteśmy w stanie obliczyć poprawki wag dla tej warstwy – dokładnie według tej samej zasady, jaką stosowaliśmy poprzednio dla sieci jednowarstwowej. Problemem staje się jednak wyznaczenie błędów na wyjściach neuronów warstwy pierwszej (ukrytej). W zbiorze uczącym nie znajdziemy wartości żądanych dla tych neuronów – są one podane jedynie dla całej sieci, tzn. dla warstwy drugiej (wyjściowej). Użytkownika sieci nie interesuje bowiem architektura sieci, czyli np. z ilu warstw się ona składa, albo ile zawiera neuronów w poszczególnych warstwach. Dla niego sieć jest czarną skrzynką o odpowiedniej liczbie wejść i wyjść, a prawidłowe przykłady jej działania zawarte są w ciągu uczącym. Warstwy sieci (w naszym przypadku jedna), do których użytkownik nie ma dostępu, nazywamy właśnie warstwami *ukrytymi*

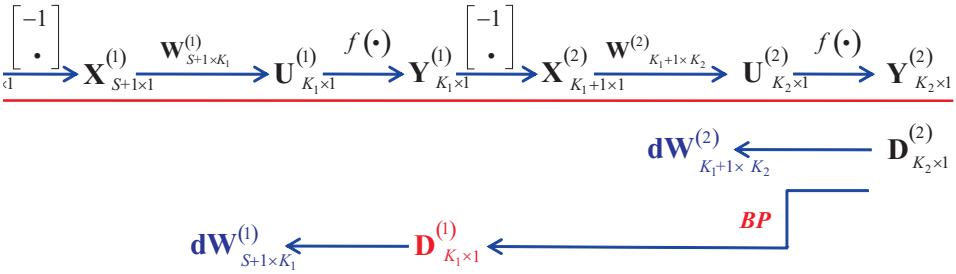


Rysunek 3.5. Pojedynczy krok uczenia sieci dwuwarstwowej - diagram czynności

– również dlatego, że nie są dla nich znane żądane wartości wyjść, więc nie jest też możliwe dokładne obliczenie błędów na tych neuronach. Domniemane błędy tych neuronów możemy jedynie wydedukować na podstawie błędów popełnianych przez całą sieć – czyli przez neurony warstw następnych. Stąd jedynym możliwym kierunkiem szacowania błędów warstw ukrytych sieci jest kierunek od wyjścia do wejścia, a metoda ich obliczania nazwana została *metodą propagacji wstecznej błędów* (ang. *backpropagation of error*).

W naszym ćwiczeniu w pierwszej kolejności wyprowadzimy w sposób intuicyjny przybliżone wzory na obliczanie błędów warstw ukrytych na podstawie błędów warstw następnych – skupimy się jedynie na istocie metody *backpropagation*. Następnie zaś pokażemy, że algorytm ten jest wynika bezpośrednio ze wzorów dotyczących minimalizacji funkcji celu. Wtedy też zastosujemy pełne, ścisłe pod względem matematycznym wzory uwzględniające wszystkie potrzebne pochodne funkcji aktywacji neuronów.

Gdy oszacujemy wartości błędów dla pierwszej warstwy sieci $\mathbf{D}^{(1)}$, obliczenie poprawek wag dla neuronów tej warstwy nie będzie się różniło niczym od obliczania poprawek dla warstwy wyjściowej (jak i każdej innej warstwy). Widać więc, że istotą i głównym celem metody *backpropagation* jest obliczenie błędów na wyjściach neuronów warstw ukrytych.



Rysunek 3.6. Pojedynczy krok uczenia sieci dwuwarstwowej - schemat uproszczony

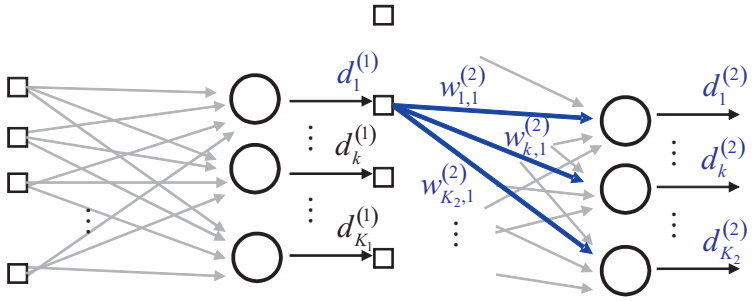
Szkielet funkcji uczącej sieć dwuwarstwową można zapisać następująco:

```
function [ W1po , W2po ] = ucz2 ( W1przed , W2przed , P , T , n )
% funkcja uczy sieć dwuwarstwową na podanym ciągu uczącym (P,T)
%                               przez zadaną liczbę epok (n)
% wersja uproszczona
% parametry: W1przed - macierz wag warstwy 1 przed uczeniem
%            P       - ciąg uczący - przykłady - wejścia
%            T       - ciąg uczący - żądane wyjścia
%            n       - liczba epok
% wynik:     W1po    - macierz wag warstwy 1 po uczeniu
%            W2po    - macierz wag warstwy 2 po uczeniu
```

```
liczbaPrzykladow = ...
wierW2           = ...              % liczba wierszy macierzy W2
W1 = W1przed ;
W2 = W2przed ;
wspUcz = 0.1 ;

for i = 1 : n ,

  nrPrzykladu = ...

  % podaj przykład na wejścia...
    X = ...                                  % wejścia sieci
  % ...i oblicz wyjścia
  [ Y1 , Y2 ] = ...

    X1 = ...                                 % wejścia warstwy 1
    X2 = ...                                 % wejścia warstwy 2
    D2 = ...                          % oblicz błędy dla warstwy 2
    D1 = ...                          % oblicz błędy dla warstwy 1
   dW1 = ...                       % oblicz poprawki wag warstwy 1
   dW2 = ...                       % oblicz poprawki wag warstwy 2
    W1 = ...                    % dodaj  poprawki do wag warstwy 1
    W2 = ...                    % dodaj  poprawki do wag warstwy 2

end  % i to wszystko n razy

W1po = W1 ;
W2po = W2 ;
```

Czytelnik, który poznał zasadę uczenia sieci jednowarstwowych w poprzednim rozdziale, powinien również poradzić sobie z uzupełnieniem powyższej funkcji. Losowanie przykładu uczącego odbywa się w ten sam sposób, co poprzednio. Należy pamiętać o tym, że na wejście sieci podajemy sygnały wejściowe wprost z ciągu uczącego. Sieć sama będzie sobie rozszerzać wektory wejściowe poszczególnych warstw o wejścia progowe. Dla użytkownika sieci jest to proces całkowicie ukryty. Również w naszej funkcji nie trzeba byłoby obliczać faktycznych wektorów wejściowych „widzianych" przez obie warstwy (**X1** i **X2**), gdyby nie instrukcje obliczające poprawki wag – w nich musimy użyć pełnych wektorów wejściowych, zawierających wejścia progowe. Jak wspomniano wcześniej, błędy obliczane są najpierw dla warstwy 2, a potem dla warstwy 1. Po ich obliczeniu kolejność obliczania poprawek wag, a następnie kolejność ich dodawania do wag nie ma oczywiście znaczenia.

Jedynym nie wyjaśnionym do tej pory krokiem jest obliczanie błędu warstwy pierwszej (linia zaczynająca się od `D1 = ...`). Problemem tym zajmiemy się za chwilę. W celu sprawdzenia poprawności pozostałej części funkcji, można tę linię tymczasowo uzupełnić przyjmując zerowe błędy warstwy pierwszej: `D1=Y1.*0;` pamiętając, że wektor błędów musi być tych samych rozmiarów, co wektor wyjść pierwszej warstwy sieci. Proponujemy teraz uruchomić skrypt `test2`, aby sprawdzić poprawność wszystkich pozostałych linii kodu. Jeśli skrypt i wywoływana w nim funkcja `ucz2` będzie działać bez komunikatów o błędach (choć oczywiście sieć nie będzie się uczyć), będziemy mogli się skupić już tylko na obliczaniu błędów warstwy pierwszej, czyli na metodzie *backpropagation*.



Rysunek 3.7. Sieć dwuwarstwowa - obliczanie błędu neuronu warstwy pierwszej

Zastanówmy się teraz, w jaki sposób można obliczyć, a raczej oszacować, błąd jednego z neuronów pierwszej warstwy sieci. Rys. 3.7 przedstawia sieć dwuwarstwową z zaznaczonym rozważanym neuronem, neuronami warstwy wyjściowej oraz łączącymi je wagami. Wszystkie nieistotne wielkości zostały ukryte.

Po przyjrzeniu się rysunkowi sieci, nasuwającym się wprost sposobem szacowania błędu pierwszego neuronu warstwy ukrytej jest obliczenie sumy ważonej błędów neuronów warstwy następnej, do których wysyła on swój sygnał.

Podejście takie można dosyć intuicyjnie wyjaśnić w następujący sposób:

- Podczas fazy obliczania sygnału wyjściowego, pierwszy neuron warstwy ukrytej wysyła swój sygnał do wszystkich neuronów drugiej warstwy.
- Sygnały te są przez neurony drugiej warstwy mnożone przez swoje wagi, tak więc to wagi neuronów drugiej warstwy decydują o tym, jaki udział ma sygnał danego neuronu pierwszej warstwy w powstaniu poprawnych lub błędnych sygnałów wyjściowych neuronów drugiej warstwy.
- Kierując się „zasadą wzajemności" można odwrócić to rozumowanie i z taką wagą propagować wstecz błąd danego neuronu warstwy drugiej, z jaką sygnał wyjściowy neuronu warstwy pierwszej miał udział w jego powstaniu.

· Pod koniec uczenia, gdy sieć jako całość nie popełnia już dużych błędów, małe błędy neuronów warstwy wyjściowej są w znikomym stopniu propagowane na błędy pierwszej warstwy, więc wagi neuronów obu warstw nie są niepotrzebnie poprawiane.

Wzór na domniemany błąd k-tego neuronu w pierwszej warstwie $d_k^{(1)}$ można więc zapisać jako

$$d_k^{(1)} = \sum_{k_2=1}^{K_2} w_{k_2,k}^{(2)} \cdot d_{k_2}^{(2)} \tag{3.2}$$

Zauważmy, że do jego oszacowania używamy wyłącznie wielkości związanych z drugą warstwą sieci - macierzy wag i wektora błędów. W celu obliczenia wszystkich błędów w warstwie pierwszej, należałoby przyjąć $k = 1, 2, \ldots, K_1$. Widać więc, że niektóre wagi z macierzy $\mathbf{W^{(2)}}$ nie będą w ogóle używane. Pozostawiamy Czytelnikowi przeanalizowanie elementów i rozmiarów poszczególnych macierzy oraz wymyślenie takiego sposobu mnożenia macierzy wag i błędów warstwy drugiej, aby obliczyć wektor błędów warstwy pierwszej za pomocą jednego wyrażenia (znów bez pętli `for`!).

Po poprawnym uzupełnieniu funkcji `ucz2` sieć powinna być w stanie nauczyć się zadanego jej problemu XOR. Wynik wywołania skryptu `test2` może być następujący:

```
P =     0     0     1     1
        0     1     0     1
T =     0     1     1     0

W1przed =   -0.0830    0.0525
            -0.0678    0.0316
            -0.0934   -0.0876
W2przed =   -0.0765
             0.0579
            -0.0153
Yprzed =     0.6280    0.6221    0.6217    0.6157

W1po =   -0.5245   -1.3398
         -1.3059   -0.9069
         -1.3158   -0.9067
W2po =    0.8621
         -1.8507
          1.8213
Ypo =     0.0210    0.9755    0.9753    0.0284
```

Przy wielokrotnym wywoływaniu skryptu, zapewne po kilku próbach Czytelnik trafi na sieć, która nie zdoła nauczyć się rozwiązywać problemu XOR podczas

założonych tu 2000 epok. Czasem nie pomoże nawet zwiększenie tej liczby, czyli wydłużenie uczenia. Spowodowane jest to faktem, że nasza sieć rozpoczyna uczenie od wag losowych. Trzy neurony sieci mają łącznie 9 wag, stąd przestrzeń parametrów sieci jest 9-wymiarowa. Nam, przyzwyczajonym do życia w trzech wymiarach, trudno jest sobie wyobrażać w takiej przestrzeni punkty, hiperpłaszczyzny, czy inne obiekty geometryczne. Jednak podczas uczenia sieć neuronowa musi, poruszając się w tej przestrzeni po hiperpowierzchni, znaleźć punkt, w którym wartość błędu na zadanym ciągu uczącym będzie najmniejsza.

Wspomnianą hiperpowierzchnię, czyli wykres błędu sieci, zależnego od 9 parametrów, musielibyśmy narysować w przestrzeni 10-wymiarowej. Moglibyśmy w niej znaleźć wiele miejsc, w których wartość błędu jest mniejsza niż dookoła – miejsca takie nazywamy *minimami lokalnymi*. Sieć neuronowa tak naprawdę nie potrzebuje znaleźć minimum globalnego tej funkcji, czyli zestawu wag zapewniającego najmniejszy błąd ze wszystkich. Wystarczy jej takie minimum, które zapewni poprawne rozwiązanie problemu zdefiniowanego w ciągu uczącymi jednocześnie zdolność do uogólniania zdobytej wiedzy na przypadki nie widziane podczas uczenia. Aby odróżnić to poszukiwane minimum od innych lokalnych minimów, nie zapewniających poprawnego rozwiązania zadania, będziemy jednak nazywać je minimum globalnym.

Większość metod uczenia sieci, których wyczerpujący przegląd można znaleźć w [4], dąży do znalezienia najlepszego minimum funkcji celu w jak najkrótszym czasie (w najmniejszej liczbie kroków). Zadanie uczenia sieci neuronowej na zadanym ciągu uczącym jest więc zadaniem optymalizacyjnym. Stosowana przez nas obecnie metoda jest przy tym najmniej wydajnym algorytmem poszukiwania tego minimum (ale za to intuicyjnym, najłatwiejszym do zrozumienia i najprostszym do zaprogramowania). W ćwiczeniach pod koniec książki poznamy najprostsze metody zwiększenia efektywności tej metody, a także wspomnimy o innych, wydajniejszych algorytmach uczenia sieci. Funkcję uczącą wzbogacimy zaś o wykresy obrazujące przebieg uczenia.

Najczęściej funkcją celu minimalizowaną w procesie uczenia jest miara błędu na wyjściu – błąd średniokwadratowy. Dla jednego przykładu uczącego jest on dany wzorem

$$E_{MSE} = \frac{1}{2}\sum_{k=1}^{K_2}\left(d_k^{(2)}\right)^2 = \frac{1}{2}\sum_{k=1}^{K_2}\left(t_k - y_k^{(2)}\right)^2 \tag{3.3}$$

gdzie K_2 jest liczbą wyjść drugiej warstwy sieci, $d_k^{(2)}$ jest błędem na wyjściu k-tego neuronu drugiej warstwy sieci, $y_k^{(2)}$ jest wartością tego wyjścia, a t_k jest jego żądaną wartością podaną w ciągu uczącym dla danego przykładu.

Proponujemy Czytelnikowi aby rozpisał powyższy wzór, przechodząc do coraz dokładniejszych szczegółów, aby w końcu otrzymać wzór ilustrujący bezpośrednio zależność funkcji celu od wszystkich wag sieci

$$E_{MSE} = \frac{1}{2} \sum_{k_2=1}^{K_2} \left[t_k - f \left(\sum_{k_1=0}^{K_1} w_{k_2,k_1}^{(2)} \cdot f \left(\sum_{s=0}^{S} w_{k_1,s}^{(2)} \cdot x_s^{(1)} \right) \right) \right]^2 \tag{3.4}$$

Oznaczenia użyte w (3.4) są zgodne z dotychczas używanymi: K_1 i K_2 to liczby neuronów w obu warstwach sieci, S jest liczbą wejść sieci (czyli warstwy pierwszej), górne indeksy przy wagach oznaczają numer warstwy, zaś $f(\cdot)$ jest funkcją aktywacji, wspólną dla wszystkich neuronów.

Najprostszą metodą minimalizacji tak zdefiniowanej funkcji celu jest metoda najszybszego spadku gradientu. W każdym kroku uczenia wektor parametrów sieci znajduje się w pewnym punkcie przestrzeni parametrów (9-wymiarowej przestrzeni wag). W punkcie tym funkcja celu, czyli błąd średniokwadratowy przyjmuje pewną wartość. Naszym zadaniem jest znalezienie takiego kierunku w tej przestrzeni, w którym funkcja celu najszybciej maleje.

W przestrzeni trójwymiarowej moglibyśmy sobie wyobrazić ten proces jako schodzenie z gór w gęstej mgle, gdy widzimy tylko najbliższe otoczenie miejsca, w którym się znajdujemy, a każdy następny krok czynimy w tym kierunku, w którym czujemy największy spadek terenu. Ten „górski" przykład naprowadza nas na fakt, że opisana metoda pozwala jedynie na znalezienie minimum lokalnego najbliższego punktowi startowemu. Bowiem, gdy dojdziemy do punktu, w którym wszystkie kierunki powodują zwiększenie błędu, uznamy, że znaleźliśmy minimum – nie będziemy jednak mieli pewności, że za lekkim wzniesieniem nie znajdziemy jeszcze niższego minimum. Wiele zależy w tej metodzie od doboru kroku, a nie tylko kierunku poszukiwań. W następnym ćwiczeniach poznamy metody pozwalające usprawnić proces minimalizacji.

Kierunek najszybszego spadku funkcji celu można wyznaczyć obliczając jej gradient, czyli pochodne względem wszystkich parametrów. Wszystkie wagi sieci możemy zapisać w jednym wektorze kolumnowym, w naszym przypadku będzie dany jako

$$\mathbf{W} = \left[w_{1,0}^{(1)}, w_{1,1}^{(1)}, w_{1,2}^{(1)}, w_{2,0}^{(1)}, w_{2,1}^{(1)}, w_{2,2}^{(1)}, w_{1,0}^{(2)}, w_{1,1}^{(2)}, w_{1,2}^{(2)} \right]^T \tag{3.5}$$

Wtedy gradient funkcji celu będzie zdefiniowany przez

$$\nabla E_{MSE} = \left[\frac{\partial E_{MSE}}{\partial w_{1,0}^{(1)}}, \ldots, \frac{\partial E_{MSE}}{\partial w_{1,2}^{(2)}} \right]^T \tag{3.6}$$

Jako poprawki wag dodawane do nich w każdym kroku uczenia należy przyjąć ujemny kierunek gradientu funkcji celu, pomnożony przez współczynnik uczenia. Poprawki wszystkich wag sieci, zebrane w jednym wektorze mają więc postać

$$\mathbf{dW} = -\eta \cdot \nabla E_{MSE} \tag{3.7}$$

Można oczywiście obliczyć poprawki dla poszczególnych wag, z podziałem na macierze wag i macierze poprawek dla obu warstw sieci – tak, jak czynimy to w naszych funkcjach uczących.

Zadanie 7. *Oblicz pochodne funkcji celu danej wzorem (3.4) względem:*

- *dowolnej wagi z drugiej warstwy sieci* $\frac{\partial E_{MSE}}{\partial w_{k,s}^{(2)}}$
- *dowolnej wagi z pierwszej warstwy sieci* $\frac{\partial E_{MSE}}{\partial w_{k,s}^{(1)}}$

Pierwsze z powyższych zadań pozwoli nam skorygować wzór na poprawkę dowolnej wagi w sieci przy założeniu, że znamy błąd na wyjściu neuronu, do którego należy waga, oraz wartość wejścia, do którego ona prowadzi. Wzór ten, zapisany przez nas poprzednio w uproszczonej postaci jako (2.11), możemy teraz zapisać dokładnie

$$dw_{k,s}^{(l)} = \eta \cdot d_k^{(l)} \cdot f'\left(u_k^{(l)}\right) \cdot x_s^{(l)} \tag{3.8}$$

gdzie $w_{k,s}^{(l)}$ jest wagą k-tego neuronu w warstwie l-tej do wejścia s-tego, na którym podany jest sygnał $x_s^{(l)}$, η jest współczynnikiem uczenia, zwykle z zakresu od 0 do 1, zaś $f'\left(u_k^{(l)}\right)$ jest pochodną funkcji aktywacji tego neuronu w punkcie $u_k^{(l)}$, stanowiącym wartość pobudzenia neuronu. W naszym przypadku, dla sigmoidalnej funkcji aktywacji pochodna upraszcza się do

$$f'\left(u_k^{(l)}\right) = y_k^{(l)} \cdot \left(1 - y_k^{(l)}\right) \tag{3.9}$$

Dla ułatwienia kolejnych rozważań błąd k-tego neuronu w l-tej warstwie sieci pomnożony przez pochodną funkcji aktywacji tego neuronu nazwiemy „wewnętrznym" błędem neuronu i oznaczymy przez $e_k^{(1)}$:

$$e_k^{(l)} = d_k^{(l)} \cdot f'\left(u_k^{(l)}\right) \tag{3.10}$$

Druga część zadania 7 miała uzasadnić matematycznie metodę wstecznej propagacji błędu i pokazać, że rzeczywiście błąd neuronu warstwy ukrytej należy szacować jako sumę ważoną błędów neuronów, do których wysyła on swój sygnał. Porównując strukturę ostatecznego wzoru na pochodną $\frac{\partial E_{MSE}}{\partial w_{k,s}^{(1)}}$ ze wzorem (3.9)

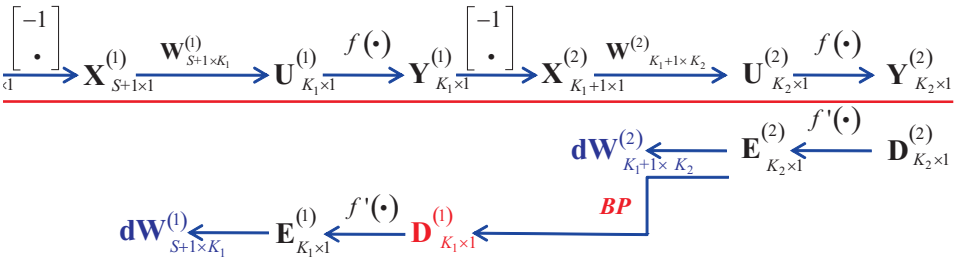
na poprawkę dowolnego neuronu sieci, Czytelnik powinien zauważyć w nim część odpowiadającą błędowi tego neuronu.

Oszacowanie błędu k-tego neuronu w pierwszej warstwie sieci $d_k^{(1)}$, zapisane wcześniej przybliżonym wzorem (3.2), można teraz uściślić uzupełniając odpowiednimi pochodnymi funkcji celu:

$$\begin{aligned} d_k^{(1)} &= \sum_{k_2=1}^{K_2} w_{k_2,k}^{(2)} \cdot d_{k_2}^{(2)} \cdot f'\left(u_{k_2}^{(2)}\right) = \\ &= \sum_{k_2=1}^{K_2} w_{k_2,k}^{(2)} \cdot e_{k_2}^{(2)} \end{aligned} \tag{3.11}$$

Powyższy przykład pokazał prostotę i piękno teorii uczenia sieci neuronowych. Wyprowadzone z początku intuicyjnie wzory – zarówno wzór na poprawkę wag dowolnego neuronu, jak i wzór na oszacowanie błędu neuronu warstwy ukrytej – znalazły potwierdzenie w skomplikowanej teorii optymalizacji leżącej u podstaw uczenia sieci.

Schematyczny zapis fazy *backpropagation* pojedynczego kroku uczenia możemy teraz przedstawić jak na rys. 3.8.



Rysunek 3.8. Pojedynczy krok uczenia sieci dwuwarstwowej - schemat dokładny

Szkielet funkcji uczącej sieć dwuwarstwową w pełnej wersji można zaś uzupełnić do następującej postaci:

```
function [ W1po , W2po ] = ucz2 ( W1przed , W2przed , P , T , n )
% funkcja uczy sieć dwuwarstwową na podanym ciągu uczącym (P,T)
%                                 przez zadaną liczbę epok (n)
% wersja dokładna
% ...

liczbaPrzykladow = ...
wierW2           = ...              % liczba wierszy macierzy W2
W1 = W1przed ;
```

```
W2 = W2przed ;
wspUcz = 0.1 ;

for i = 1 : n ,

  nrPrzykladu = ...

  % podaj przykład na wejścia...
    X = ...                                  % wejścia sieci
  % ...i oblicz wyjścia
  [ Y1 , Y2 ] = ...

    X1 = ...                                 % wejścia warstwy 1
    X2 = ...                                 % wejścia warstwy 2
    D2 = ...                          % oblicz błędy dla warstwy 2
    E2 = ...                          % błąd'wewnętrzny' warstwy 2
    D1 = ...                          % oblicz błędy dla warstwy 1
                                      %     (użyj E2 zamiast D2)
    E2 = ...                          % błąd'wewnętrzny' warstwy 1
   dW1 = ...                       % oblicz poprawki wag warstwy 1
                                   %        (użyj E1 zamiast D1)
   dW2 = ...                       % oblicz poprawki wag warstwy 2
                                   %        (użyj E2 zamiast D2)
    W1 = ...                    % dodaj  poprawki do wag warstwy 1
    W2 = ...                    % dodaj  poprawki do wag warstwy 2

end  % i to wszystko n razy

W1po = W1 ;
W2po = W2 ;
```

Zarówno na schemacie, jak i w szablonie funkcji widać, że „wewnętrzne" błędy neuronów obu warstw można zebrać w odpowiednich wektorach (w naszym przykładzie **E1** i **E2**), zaś dzięki prostej postaci pochodnej funkcji sigmoidalnej, ich obliczenie nie jest skomplikowane (i nie wymaga użycia pętli `for`). Zarówno we wzorach na poprawki wag, jak i we wzorze na domniemane błędy warstwy pierwszej, należy teraz użyć „wewnętrznych" błędów neuronów.

Uzupełnienie funkcji `ucz` do postaci ścisłej matematycznie powinno skutkować przyspieszeniem uczenia, zarówno w przypadku sieci jedno-, jak i dwuwarstwowych.

Rozdział 4

Badanie procesów uczenia jednowarstwowych sieci neuronowych

W niniejszym rozdziale wzbogacimy funkcje uczące sieci o szereg wykresów ilustrujących proces uczenia sieci - wykres zmian błędu średniokwadratowego w kolejnych epokach, wykres błędu klasyfikacji, wykres przebiegu wag, czy wreszcie granic decyzyjnych poszczególnych neuronów oraz całej sieci. Będzie to dla nas okazja do przeanalizowania ich możliwości, sposobu w jaki rozwiązują problemy, a także zjawisk zachodzących podczas uczenia. Poznamy również kilka podstawowych zasad, do których należy się stosować, aby prawidłowo oceniać jakość działania sieci i jej zdolności do uogólniania wiedzy. Uzupełnimy także podstawowy algorytm uczenia sieci wielowarstwowej o metody pozwalające na usprawnienie procesu minimalizacji błędu.

Na początek jednak zadbajmy o przyjazność pisanych przez nas programów dla użytkowników (czyli dla nas). Musimy przyznać, że w skryptach `test1` i `test2` komunikacja ta była wyjątkowo lakoniczna – ograniczała się w zasadzie do wyprowadzania wartości macierzy. Projektowanie interfejsu użytkownika jest w dużym stopniu procesem subiektywnym, podporządkowanym potrzebom pokazania niezbędnych informacji w jak najbardziej przystępny i obrazowy sposób, ale zależnym również od indywidualnego poczucia estetyki. Dlatego ostateczne decyzje dotyczące sposobu wyświetlania komunikatów, czy postaci wykresów pozostawiamy Czytelnikowi.

4.1. Komunikacja z użytkownikiem

W pierwszym zadaniu przedstawimy propozycję wyświetlania komunikatów przez podstawową wersję skryptu uczącego sieci jednowarstwowe. W następnych zadaniach będziemy dodawać kolejne usprawnienia do naszych funkcji, koncentrując się na stronie algorytmicznej. Przedstawiamy własne propozycje komunikatów i wykresów, które Czytelnik może prześledzić uruchamiając programy umieszczone na stronie internetowej książki, a następnie zaimplementować podobne, lub zupełnie inne – wedle własnego uznania.

Zadanie 8. *Uzupełnij skrypt* `test1` *tak, aby w zwięzły i przystępny sposób:*

- *informował użytkownika o zadaniu wykonywanym przez sieć,*
- *podawał znaczenie wejść i wyjść sieci,*
- *przedstawiał ciąg uczący (macierze* $\mathbf{P}$ *i* $\mathbf{T}$*),*
- *wyświetlał wagi sieci i odpowiedzi na przykłady z ciągu uczącego – przed uczeniem,*
- *wyświetlał wagi sieci oraz wynik jej działania – po uczeniu.*

Dialog z użytkownikiem może przypominać wydruk poniżej.

```
************************************************
Sieć jednowarstwowa uczy się rozpoznawać zwierzęta
- wersja podstawowa
************************************************
wejścia sieci:  1 - ile ma nóg
                2 - czy żyje w wodzie
                3 - czy umie latać
                4 - czy ma pióra
                5 - czy jest jajorodne
 wyjścia sieci: 1 - ssak
                2 - ptak
                3 - ryba
************************************************
Ciąg uczący:
P - wejścia sieci
T - żądane wyjścia
kolejne przykłady w kolumnach
P =    4.0000    2.0000   -1.0000
       0.0100   -1.0000    3.5000
       0.0100    2.0000    0.0100
      -1.0000    2.5000   -2.0000
      -1.5000    2.0000    1.5000
T =    1     0     0
       0     1     0
       0     0     1

---------------------------------------
Sieć przed uczeniem:
Wprzed =    0.0978    0.0567   -0.0724
           -0.0866    0.0068   -0.0564
            0.0879    0.0771   -0.0636
           -0.0964    0.0798   -0.0916
            0.0368    0.0252   -0.0786

---------------------------------------
Tak działa:
powinnoByc =     1     0     0
                 0     1     0
                 0     0     1
```

```
jest =    0.8969    0.8105    0.3184
          0.6344    0.9278    0.3166
          0.3997    0.0470    0.4251
----------------------------------------
Naciśnij ENTER, aby rozpocząć uczenie...

Uczę sieć...

----------------------------------------
Sieć po uczeniu:
Wpo =    0.1136   -0.1832   -0.3287
        -0.2712   -0.2355    0.1666
        -0.1307    0.1218   -0.0767
        -0.2384    0.3495   -0.1869
        -0.3635    0.1085    0.0689
----------------------------------------
Tak działa:
powinnoByc =     1     0     0
                 0     1     0
                 0     0     1
jest =    0.9979    0.0044    0.0035
          0.0020    0.9976    0.0028
          0.0021    0.0014    0.9990
Naciśnij ENTER...
```

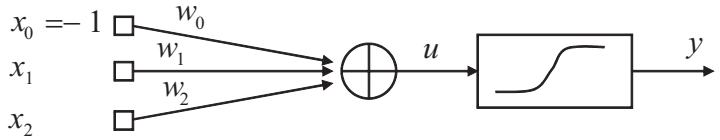
Proponujemy zmienić nazwy plików z funkcjami dotyczącymi sieci (oraz odpowiednio nagłówki samych funkcji oraz ich wywołania) na: `init1a`, `dzialaj1a` i `ucz1a`, a samego skryptu na: `test1a`. Takie właśnie nazwy plików będą używane w dalszej części książki oraz na stronie internetowej autora w celu odróżnienia kolejnych zmian wprowadzanych do naszych programów. Już w następnej wersji dodamy bowiem do sieci jednowarstwowej wejście progowe, a następnie możliwość pokazania sieci kilku przykładów uczących w jednym kroku oraz obliczanie, sprawdzanie i rysowanie błędu średniokwadratowego.

4.2. Dokładny model neuronu

Jak wspomniano w rozdziale 2, do rozwiązania prostego zadania klasyfikacji zwierząt nie była potrzebna obecność wejścia progowego w wektorze wejść neuronu. Obecnie uzupełnimy jednak jego model do pełnej postaci, aby właściwie przeanalizować jego zdolności rozdzielcze.

Zadanie 9. *Uzupełnij funkcje* `init1a` *oraz* `dzialaj1a` *o obsługę wejścia progowego. Dostosuj odpowiednio funkcję uczącą* `ucz1a` *oraz skrypt* `test1a`*. Nowym funkcjom nadaj nazwy, odpowiednio:* `init1b`*,* `dzialaj1b` *oraz* `ucz1b`*, a skryptowi —* `test1b`*.*

Wprowadzenie zmian proponowanych w ćwiczeniu nie powinno stanowić dla Czytelnika większego problemu. Podobnie, jak w przypadku sieci dwuwarstwowych, należy teraz pamiętać o tym, że macierz wag musi posiadać jeden dodatkowy wiersz, zawierający wartości wag prowadzących do wejść progowych, zaś przed mnożeniem macierzy wag z wektorem sygnałów wejściowych funkcja `działaj1b` musi rozszerzyć ten ostatni o wejście progowe, zawsze równe -1. Funkcja ucząca sieć powinna korzystać z funkcji `dzialaj1b` podczas obliczania odpowiedzi sieci, podając jako jeden z parametrów wektor wejść sieci nie zawierający progu. Do obliczania poprawek wag należy zaś użyć wektora wejść, który „widzi" warstwa sieci, a więc tego rozszerzonego o wejście progowe. W skrypcie `test1b` nie będzie juz można jednym poleceniem sprawdzić działania sieci przed i po uczeniu, lecz należy to zrobić przykład po przykładzie – podobnie, jak w przypadku sieci dwuwarstwowych.



Rysunek 4.1. Pojedynczy neuron o dwóch wejściach

Rys. 4.1 przedstawia model pojedynczego neuronu z dwoma wejściami. Sygnały podawane na jego wejścia można więc przestawić jako punkty na płaszczyźnie (x_1, x_2). Pojedynczy neuron może zostać wykorzystany do rozwiązywania prostego zadania klasyfikacji punktów na płaszczyźnie. Zakładając unipolarną sigmoidalna funkcję aktywacji, możemy przyjąć, że dla punktów należących do jednej klasy odpowiedź neuronu ma wynosić 1, zaś dla punktów drugiej klasy 0. Dla bipolarnej funkcji aktywacji odpowiedzi sieci wynosiłyby odpowiednio 1 i -1.

Ponieważ oba rodzaje funkcji sigmoidalnej przyjmują wartości ciągłe, musimy przyjąć pewną wartość graniczną decydującą o podziale na klasy. Naturalnym wyborem jest wartość 0.5 dla funkcji unipolarnej i 0 dla funkcji bipolarnej. Punkty, dla których odpowiedź jest większa od wartości granicznej, będą uznane za należące do pierwszej klasy, a punkty o odpowiedzi poniżej tej wartości – do klasy drugiej. Łatwo zauważyć, że obie funkcje aktywacji przyjmują wartość graniczną przy pobudzeniu równym 0, czyli $u = 0$, co możemy rozpisać jako

$$u = w_0 x_0 + w_1 x_1 + w_2 x_2 = 0 \tag{4.1}$$

Uwzględniając zawsze jednakową wartość *biasu*

$$-w_0 + w_1 x_1 + w_2 x_2 = 0 \tag{4.2}$$

Ostatecznie otrzymaliśmy równanie prostej

$$x_2 = -\frac{w_1}{w_2} x_1 + \frac{w_0}{w_2} \tag{4.3}$$

która jest granicą decyzyjną neuronu, a której parametry (nachylenie i przesuniecie) zależą od wag neuronu (a dokładniej: od ich wzajemnych proporcji).

Pojedynczy neuron jest więc *dyskryminatorem liniowym*. Oznacza to, że na płaszczyźnie jest w stanie rozdzielić punkty do różnych klas linią prostą. Jak łatwo zauważyć, pozbawienie neuronu wejścia progowego ograniczało jego zdolności do linii przechodzących przez środek układu współrzędnych, których nachylenie można regulować zmieniając wagi neuronu, bez możliwości regulacji ich przesunięcia. Jak pokazały nam dotychczasowe ćwiczenia, w zupełności wystarczało to do klasyfikacji zwierząt jako: ssaki, ptaki i ryby, już nie na płaszczyźnie, lecz w pięciowymiarowej przestrzeni sygnałów wejściowych. Rozważane tu zdolności klasyfikacyjne neuronu można więc uogólnić na przestrzenie wejściowe o większej liczbie wymiarów.

Neuron o S wejściach, wyposażony w wejście progowe (bias) ma $S + 1$ parametrów, które można dowolnie „dostroić" w procesie uczenia. Ponieważ jest on dyskryminatorem liniowym, to w S-wymiarowej przestrzeni sygnałów wejściowych jest w stanie rozdzielać punkty na dwie klasy przy pomocy hiperpłaszczyzn.

Czytelnik może sprawdzić, że neuron o dwóch wejściach może na płaszczyźnie ustawić swoje wagi tak, aby rozdzielić 3 punkty na wszystkie możliwe sposoby (kombinacje klas dla poszczególnych punktów). W ogólności potrafi on w S-wymiarowej przestrzeni rozdzielić $S + 1$ punktów na wszystkie możliwe sposoby – i tyle właśnie wynosi jego wymiar Vapnika-Chervonenkisa. Wielkość ta jest ściśle związana z teorią rozpoznawania i pozwala określić zdolności klasyfikacyjne maszyn uczących (w tym sieci neuronowych). Przydaje się także m.in. do szacowania liczby przykładów w zbiorze uczącym potrzebnych do prawidłowego nauczenia sieci, lub do określenia, jaki błąd sieć jest w stanie osiągnąć przy zbiorze uczącym o ustalonej liczbie przykładów.

Jedno z następnych ćwiczeń pokaże nam, że w istocie jedyną granicą decyzyjną, jaką jest w stanie wytworzyć pojedynczy neuron, jest właśnie linia prosta. Dalsze ćwiczenia pokażą nam jednak, że z takich pojedynczych liniowych granic decyzyjnych, sieci dwuwarstwowe są w stanie tworzyć bardzo skomplikowane granice nieliniowe.

4.3. Miara jakości działania sieci - błąd średniokwadratowy

Zadanie 10. *Uzupełnij funkcję* `ucz1b` *o obliczanie błędu średniokwadratowego. Sprawdzaj ten błąd po każdym kroku uczenia, kończ uczenie po osiągnięciu założonego poziomu błędu lub po upływie maksymalnej założonej liczby kroków. Rysuj wykres błędu średniokwadratowego podczas uczenia. Nazwij nową funkcję uczącą* `ucz1c`*, a jej nagłówek zdefiniuj następująco:*

```
function [ Wpo ] = ucz1c ( Wprzed , P , T , maxLiczbaKrokow  , ...
                                             docelowyBladMSE  )
```

Błąd średniokwadratowy dla sieci dwuwarstwowej został już zdefiniowany wzorem (3.3). Jest to po prostu suma kwadratów odchyłek wszystkich neuronów od wartości żądanych dla danego przykładu. Aby miara ta bardziej odzwierciedlała błąd na pojedynczym neuronie, należy jeszcze tę sumę podzielić przez liczbę neuronów w warstwie. Wzór na błąd MSE dla sieci jednowarstwowej o K neuronach jest więc dany jako

$$D_{MSE} = \frac{1}{2}\sum_{k=1}^{K} d_k^2 = \frac{1}{2}\sum_{k=1}^{K} (t_k - y_k)^2 \tag{4.4}$$

W naszej funkcji uczącej można go obliczyć bezpośrednio na podstawie wektora błędów `D` stosując odpowiednie mnożenie macierzowe (bez użycia pętli `for`). Uzupełnienie funkcji o porównywanie błędu sieci w danym kroku z założonym błędem docelowym nie powinno stanowić dla Czytelnika większego problemu, podobnie jak zapisanie kolejnych wartości błędu w macierzy i wyświetlenie wykresu błędu w kolejnych krokach.

Zadanie 11. *Napisz funkcję* `sprawdz1`*, która sprawdza sieć jednowarstwową, zdefiniowaną macierzą wag* $\mathbf{W}$*, na ciągu danych podanym w postaci pary macierzy* $\mathbf{P}$ *i* $\mathbf{T}$*. Funkcja powinna podać również liczbę błędnych klasyfikacji (jako procent wszystkich przykładów), zakładając unipolarną sigmoidalną funkcję aktywacji neuronów (czyli żądane wartości wyjść 0-1). Oprócz obu tych błędów funkcja powinna zwracać również macierz odpowiedzi sieci na kolejne przykłady z ciągu danych oraz odpowiadającą im macierz błędów. Nagłówek funkcji dany jest następująco:*

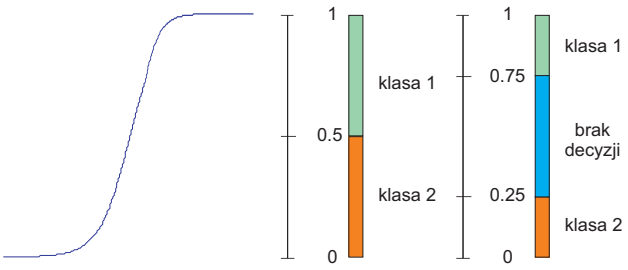
```
function [ Y , D , MSE , CE ] = sprawdz1 ( W , P , T )
```

Zadanie 12. *Na podstawie poprzednich skryptów napisz* `test1c`*, który oprócz użycia funkcji uczącej* `ucz1c` *będzie sprawdzał działanie sieci przed i po uczeniu na całym ciągu uczącym, podając wartość błędu średniokwadratowego oraz liczbę błędnych klasyfikacji. Skrypt powinien pytać się użytkownika o maksymalną dopuszczalną liczbę kroków uczenia oraz zakładany do osiągnięcia błąd średniokwadratowy i przekazywać te parametry do funkcji* `ucz1c`*.*

Funkcja sprawdzająca odpowiedzi sieci na zadanym ciągu uczącym pozwoli nam właściwie ocenić jakość jej działania. W dalszych zadaniach poznamy zasady prawidłowego podziału danych na ciągi: uczący, sprawdzający i testujący. Na razie będziemy sprawdzać działanie sieci na całym zbiorze uczącym.

Oprócz błędu średniokwadratowego sieci na zadanym ciągu, funkcja ma zwracać kilka wartości pomocniczych. Odpowiedzi sieci $\mathbf{Y}$ na kolejne przykłady z zadanego ciągu oraz błędy $\mathbf{D}$ możemy wykorzystać do sprawdzenia, których przykładów sieć nie jest w stanie się nauczyć, aby potem przeanalizować czy wynika to np. z błędów pomiarowych lub grubych w danych, błędnego zapisu wejść, czy błędnego kodowania wyjść.

Błąd średniokwadratowy mierzy odchylenia wyjść sieci od wartości żądanych – w zadaniach klasyfikacji przy użyciu unipolarnej funkcji aktywacji neuronów są to wartości 0 i 1. Wyjścia sieci nie muszą jednak osiągać wartości bardzo bliskich tym idealnym, aby poprawnie klasyfikować obiekty opisane na wejściach. W przykładowym zadaniu rozpoznawania zwierząt wystarczy, że na wyjściu neuronu przypisanego do odpowiedniej klasy pojawi się wartość większa od 0.5, a na pozostałych neuronach – mniejsza od 0.5. W ten prosty sposób liczy błędne klasyfikacje funkcja `sprawdz1` zaimplementowana prze autora. Czytelnik może pokusić się o bardziej zaawansowaną interpretację wyjść sieci, np. wprowadzenie dodatkowej klasy *brak decyzji*, czy sprawdzanie poziomu aktywacji zwycięzcy i odległości od następnego przy kodowaniu klas „jeden z N”. Przykłady możliwych interpretacji wyjścia dla pojedynczego neuronu dzielącego obiekty na dwie klasy przedstawia rys. 4.2.

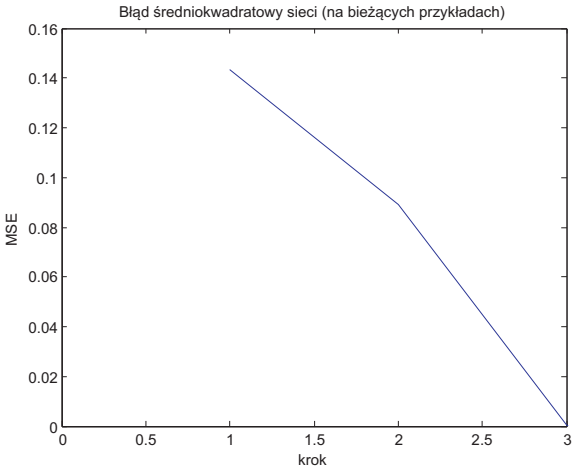


Rysunek 4.2. Interpretacje wyjścia pojedynczego neuronu rozpoznającego dwie klasy obiektów

Po wprowadzeniu odpowiednich zmian o funkcji uczącej i skryptu ją wywołującego, uczenie sieci może przebiegać podobnie do poniższego (fragmenty znane już z poprzednich wydruków zostały pominięte):

```
************************************************
Sieć jednowarstwowa uczy się rozpoznawać zwierzęta
- wersja z wykresem błędu MSE
************************************************
(...)
----------------------------------------
Podaj maksymalną liczbę kroków uczenia: 100
Podaj docelowy błąd średniokwadratowy MSE (na bieżących przykładach): 0.01
----------------------------------------
Sieć przed uczeniem:
Wprzed =   -0.0405   -0.0196   -0.0690
           -0.0207   -0.0410   -0.0999
           -0.0158   -0.0387   -0.0433
           -0.0377   -0.0789    0.0102
            0.0388    0.0188    0.0742
           -0.0816   -0.0435   -0.0915
----------------------------------------
Tak działa:
powinnoByc =     1     0     0
                 0     1     0
                 0     0     1
jest =    0.5508    0.3466    0.2743
          0.3787    0.2484    0.2907
          0.2078    0.4197    0.2073
Błąd średniokwadratowy        MSE =    0.1131979
Procent błędnych klasyfikacji  CE =   66.7
----------------------------------------
Naciśnij ENTER, aby rozpocząć uczenie...
Uczę sieć...
```

```
Krok:       1 /    100 - błąd MSE:  0.1435238 /  0.0100000
Krok:       2 /    100 - błąd MSE:  0.0892425 /  0.0100000
Krok:       3 /    100 - błąd MSE:  0.0001102 /  0.0100000
---------------------------------------
Sieć po uczeniu:
Wpo =    0.0483   -0.0888   -0.0771
        -0.0358    0.1153   -0.2339
        -0.1709   -0.1346    0.1742
        -0.1076    0.0714   -0.0733
         0.0605    0.2185   -0.1310
        -0.2323    0.0980   -0.1003
---------------------------------------
Tak działa:
powinnoByc =     1     0     0
                 0     1     0
                 0     0     1
jest =    0.6147    0.0842    0.0045
          0.7148    0.9988    0.0192
          0.0531    0.0020    0.9943
Błąd średniokwadratowy        MSE =    0.0372100
Procent błędnych klasyfikacji  CE =   33.3
Naciśnij ENTER...
```



Rysunek 4.3. Przebieg uczenia neuronu – test1c

Wykres błędu średniokwadratowego odpowiadający powyższemu przebiegowi uczenia przedstawia rys. 4.3. Widać na nim, podobnie, jak na wydruku działania programu, że sieć zakończyła uczenie za wcześnie, a odpowiedzi „nauczonej" sieci nie są zadowalające. Powodem tego jest fakt, że błąd średniokwadratowy, na pod-

stawie którego sieć podejmuje decyzję o zakończeniu uczenia, jest obliczany tylko na jednym przykładzie pokazywanym w danym kroku. Wystarczy, że sieć nauczy się dobrze działać dla jednego przykładu, a już uzna, że nauczyła się wszystkich. Oczywiste jest więc, że sieć powinna w każdym kroku sprawdzać swoje działanie na całym ciągu uczącym przy użyciu funkcji `sprawdz1` (a tak naprawdę — na niezależnym od niego ciągu sprawdzającym). Błąd ten poprawimy w następnej wersji funkcji `ucz`, w której ponadto wprowadzimy możliwość pokazania sieci kilku przykładów w jednym kroku uczenia.

4.4. Prezentacja kilku przykładów w w jednym kroku

Zadanie 13. *Zmodyfikuj funkcję* `ucz1c` *tak, aby w każdym kroku obliczała błąd średniokwadratowy sieci nie tylko dla przykładu pokazanego w danym kroku, ale również na całym ciągu uczącym — decyzję o zakończeniu uczenia przed osiągnięciem maksymalnej liczby kroków powinna podejmować na podstawie tego drugiego błędu. Na wykresie błędu średniokwadratowego pokaż oba błędy. Nową funkcję nazwij* `ucz1d`.

Proponowane zmiany wymagają jedynie uzupełnienia funkcji `test1c` o sprawdzenie sieci na całym ciągu uczącym (przy pomocy funkcji `sprawdz1`) przed samym końcem pętli obsługującej kolejne kroki uczenia. Dla odróżnienia obu błędów proponujemy nazwać obie zmienne, odpowiednio:

- `bladMSEkrok` - wartość błędu dla przykładu pokazanego w danym kroku,
- `bladMSEciag` - wartość błędu dla całego ciągu uczącego.

Obie zmienne są wektorami o wymiarze równym maksymalnej liczbie kroków, aby można było je prezentować na wykresie. Należy również uwzględnić nowy błąd (dla całego ciągu) w instrukcji warunkowej przerywającej pętlę po wszystkich krokach.

Zadanie 14. *Uzupełnij funkcję* `ucz1d` *o możliwość pokazywania sieci kilku przykładów w jednym kroku uczenia. W każdym kroku uczenia funkcja ucząca powinna:*

- *losować przykłady do pokazania sieci w danym kroku,*
- *po kolei pokazać sieci wylosowane przykłady,*
- *dla każdego z nich obliczyć poprawki wag, ale nie dodawać ich do wag, a jedynie kumulować,*
- *po pokazaniu wszystkich przykładów w danym kroku - znormalizować skumulowane poprawki i dodać do wag,*
- *sprawdzić działanie sieci i ewentualnie zakończyć uczenie.*

Odpowiednio zmień sposób obliczania błędu MSE dla przykładów pokazanych w danym kroku (podobnie, jak poprawki wag, ten błąd również należy kumulować i na koniec kroku normalizować).

Proponowane ulepszenie procesu uczenia sieci o możliwość pokazywania kilku przykładów w jednym kroku jest bardzo ważne z punktu widzenia jego efektywności, o czym przekonamy się szczególnie w dalszych ćwiczeniach, gdy będziemy rozwiązywać trudniejsze zadania.

W teorii uczenia sieci neuronowych wyróżniamy dwa skrajne podejścia dotyczące strategii prezentowania sieci danych uczących i uaktualniania wag. Pierwsza z nich – metoda *on-line*, adaptacyjna – zakłada adaptację wag po każdej prezentacji wzorca uczącego. Druga – metoda *batch*, wsadowa – zakłada uaktualnienie wag po prezentacji wszystkich wzorców ze zbioru uczącego, czyli po całej epoce uczenia. My zastosujemy podejście pośrednie, które pozwoli nam również korzystać z obu wymienionych metod.

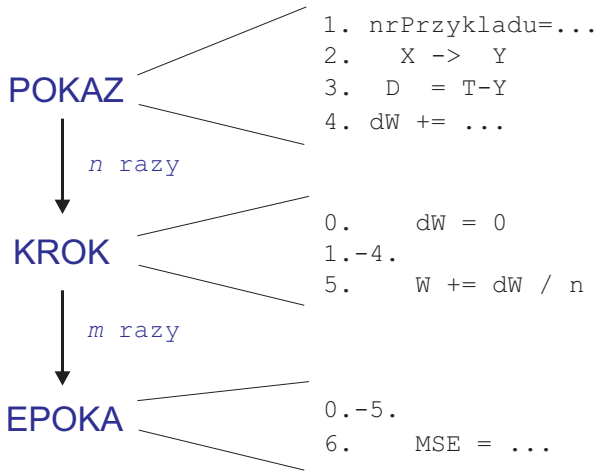
Aby uporządkować terminologię, wprowadzimy następujące pojęcia:

- *pokaz* - prezentacja pojedynczego przykładu z ciągu uczącego, obliczenie odpowiedzi sieci, błędu oraz poprawek wag,
- *krok* - wybrana liczba *pokazów*, kumulacja poprawek wag obliczonych w każdym pokazie, na koniec kroku następuje adaptacja wag przy użyciu skumulowanych i znormalizowanych poprawek,
- *epoka* - taka liczba *kroków*, aby zaprezentować sieci wszystkie dane z ciągu uczącego, na zakończenie epoki mierzy się błąd na ciągu sprawdzającym i podejmuje decyzję o kontynuacji lub zakończeniu uczenia.

Oczywiście, aby pojęcie epoki miało sens, przykłady prezentowane podczas wszystkich kroków uczenia w danej epoce nie mogą się powtarzać. Przy jednej prezentacji w kroku uczenia, otrzymamy metodę *on-line*, jeśli wybierzemy rozmiar kroku równy liczbie przykładów – metodę *batch*. Możemy również wybrać „złoty środek" i np. 1000-elementowy zbiór danych uczących pokazywać w 100 krokach po 10 przykładów.

W niniejszym ćwiczeniu umożliwimy prezentację kilku przykładów w jednym kroku, a pojęcie epoki wprowadzimy dopiero przy sieciach dwuwarstwowych. Docelowo pamiętajmy jednak, że zwykle długość uczenia sieci (szczególnie przy bardziej zaawansowanych algorytmach) mierzy się w epokach. Przy skomplikowanych zadaniach, których ma się nauczyć sieć, proces uczenia może trwać wiele tysięcy epok, przy czym podczas każdej prezentowanych może być dziesiątki tysięcy przykładów. W dalszych ćwiczeniach (szczególnie przy sieciach dwuwarstwowych) Czytelnik będzie mógł sprawdzić, jaki wpływ ma rozmiar kroku uczenia na

jego efektywność. Uczenie sieci z uwzględnieniem podziału na pokaz-krok-epokę przedstawia schematycznie przy użyciu pseudokodu rys. 4.4.



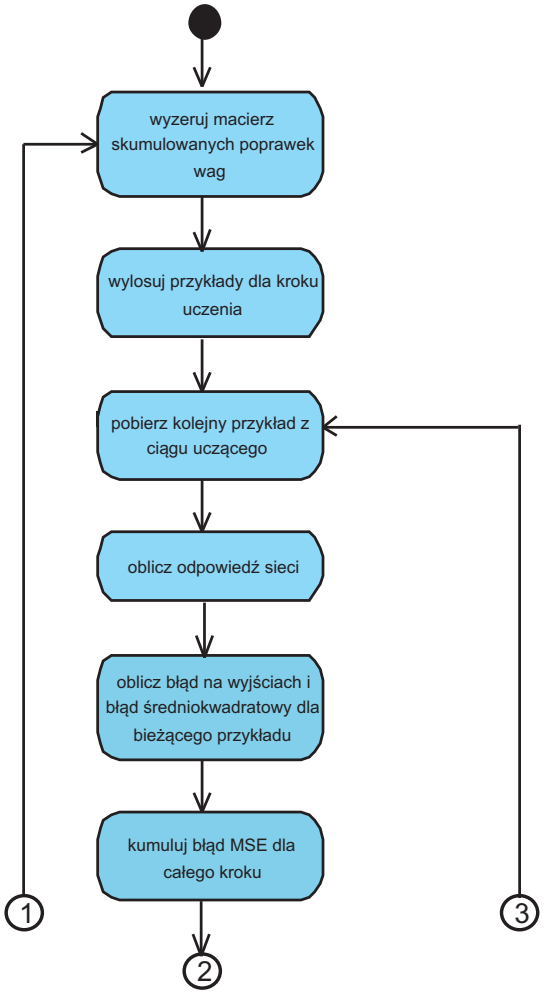
Rysunek 4.4. Schemat uczenia sieci z podziałem na kroki i epoki

Szkielet funkcji `ucz1d`, ograniczony do najistotniejszych elementów algorytmicznych przedstawia się następująco:

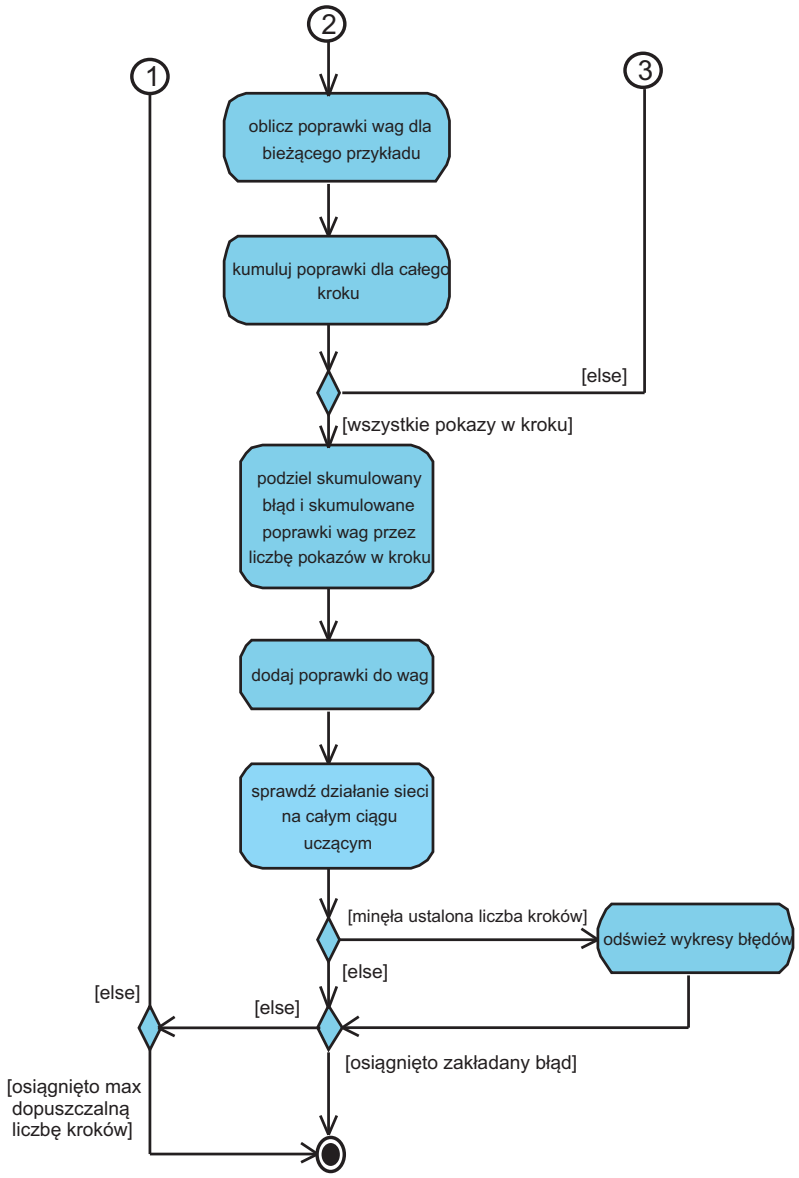
```
function [ Wpo ] = ucz1d ( Wprzed , P , T , maxLiczbaKrokow  , ...
                                             liczbaPokazow    , ...
                                             czestoscWykresow , ...
                                             docelowyBladMSE  )
% ------------------------------------------------------------------
% ustalenie różnych wielkości (...)
% przygotowanie macierzy pod wykres (...)
% ------------------------------------------------------------------
% UCZENIE SIECI
% ------------------------------------------------------------------
for krok = 1 : maxLiczbaKrokow ,

   dWkrok = ...           %    na początku każdego kroku
                          %    wyzerowanie macierzy poprawek
   los = ...              % 1) losuj numery przykładów dla danego kroku
   nryPrzykladow = ...    %    aby się powtarzały

   for pokaz = 1 : liczbaPokazow ,
       nrPrzykladu = ...
       X = ...            % 2) podaj przykład na wejścia...
       Y = ...            % 3) ...i oblicz wyjścia
       D = ...            % 4) oblicz błędy na wyjściach
       bladMSEpokaz       = ...  % oblicz błąd średniokwadratowy
                                 % - na bieżącym przykładzie
```

```
      bladMSEkrok(krok) = ...  % i skumuluj
      X = ...           %    macierz wejść z biasem
      dWpokaz = ...     % 5) oblicz poprawki wag dla danego pokazu
      dWkrok  = ...     %    - i skumuluj dla całego kroku
   end  % for pokaz

   bladMSEkrok(krok) = ...  % znormalizuj skumulowany błąd
   W = ...                  % 6) dodaj poprawki do wag
                            %  - skumulowane i znormalizowane
   % oblicz błąd średniokwadratowy na całym ciągu uczącym
   %   oraz procent błędnych klasyfikacji
   [ Yciag , Dciag , bladMSEciag(krok) , bladCEciag(krok) ] = ...
   % ------------------------------------------------------------
   % co czestoscWykresow kroków...
   if  mod(krok,czestoscWykresow) == 0 ,
       %  ...komunikat o postępach
       %  ...wykresy błędów
   end  % wykres ------------------------------------------------
   % jeśli osiągnięto zadany błąd,
   % to kończymy uczenie nie zważając na max liczbę kroków
   if  ( bladMSEciag(krok) < docelowyBladMSE ) ,
       break
   end
end  % krok
% ustawienie granic wykresów (...)
% ------------------------------------------------------------
Wpo = W ;
```

Ideę prezentowania kilku przykładów w jednym kroku uczenia oraz kumulacji poprawek wag przedstawiają diagramy czynności na rys. 4.5 i 4.6.



Rysunek 4.5. Uczenie sieci z prezentacją kilku przykładów w jednym kroku - diagram czynności, część 1



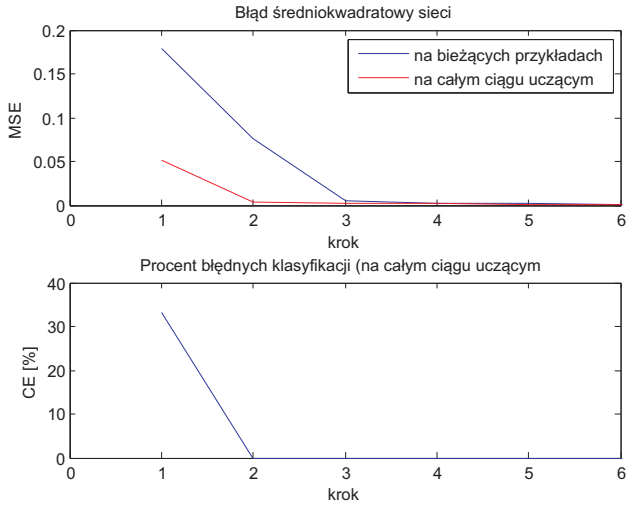
Rysunek 4.6. Uczenie sieci z prezentacją kilku przykładów w jednym kroku - diagram czynności, część 2

Zadanie 15. *Uzupełnij skrypt* `test1c` *tak, aby pytał się użytkownika o liczbę przykładów do pokazania w jednym kroku uczenia oraz o szybkość odświeżania wykresów. Do uczenia sieci skrypt powinien używać funkcji* `ucz1d`. *Nazwij skrypt* `test1d`.

Efektem działania poprawionego skryptu może być następujący przebieg uczenia sieci (pominięto w nim znane już szczegóły).

```
************************************************
Sieć jednowarstwowa uczy się rozpoznawać zwierzęta
- wersja z wykresem błędu MSE
         i wykresem błędu CE
************************************************
(...)
---------------------------------------
Podaj maksymalną liczbę kroków uczenia: 100
Podaj liczbę przykładów do pokazania w jednym kroku uczenia: 2
Co ile epok odświeżać wykres? 1
Podaj docelowy błąd średniokwadratowy MSE (na całym ciągu uczącym): 0.001
---------------------------------------
Sieć przed uczeniem:
Wprzed = (...)
---------------------------------------
Tak działa:
powinnoByc = (...)
jest =       (...)
Błąd średniokwadratowy        MSE =    0.1678431
Procent błędnych klasyfikacji  CE =  100.0
---------------------------------------
Naciśnij ENTER, aby rozpocząć uczenie...
Uczę sieć...
Epoka: 1 / 100 - błąd MSE:  0.0513717 /  0.0010000 - błąd CE:  33.3
Epoka: 2 / 100 - błąd MSE:  0.0041746 /  0.0010000 - błąd CE:   0.0
Epoka: 3 / 100 - błąd MSE:  0.0022572 /  0.0010000 - błąd CE:   0.0
Epoka: 4 / 100 - błąd MSE:  0.0018230 /  0.0010000 - błąd CE:   0.0
Epoka: 5 / 100 - błąd MSE:  0.0010998 /  0.0010000 - błąd CE:   0.0
Epoka: 6 / 100 - błąd MSE:  0.0009091 /  0.0010000 - błąd CE:   0.0
---------------------------------------
Sieć po uczeniu:
Wpo = (...)
---------------------------------------
Tak działa:
powinnoByc = (...)
jest =       (...)
Błąd średniokwadratowy        MSE =    0.0009091
Procent błędnych klasyfikacji  CE =    0.0
Naciśnij ENTER...
```

Wykres towarzyszący przykładowemu przebiegowi uczenia przedstawia rys. 4.7.



Rysunek 4.7. Przebieg uczenia neuronu – test1d

Zadanie 16. *Do funkcji* `ucz1d` *dodaj rysowanie wykresu zmian wag neuronów sieci podczas kolejnych kroków uczenia. Nową funkcję nazwij* `ucz1e`*, zaś skrypt ją wywołujący –* `test1e`*.*

Rozwiązanie tego prostego zadania pozostawimy Czytelnikowi. Wykresy wag utworzone przy pomocy nowej funkcji przedstawimy w następnych ćwiczeniach.

4.5. Jakie zadania klasyfikacyjne potrafi rozwiązać jeden neuron

Uzupełniliśmy funkcję uczącą o możliwości rysowania szeregu wykresów, których obserwacja pozwoli nam śledzić wydajność uczenia sieci. Zadanie, które obecnie stawiamy przed siecią (rozpoznawanie zwierząt), jest na tyle proste, że nasze sieci jednowarstwowe złożone z trzech neuronów uczą się je rozwiązywać „w mgnieniu oka". Wektory opisujące poszczególne rodzaje zwierząt są bowiem liniowo separowalne w 5-wymiarowej przestrzeni wejść. Czytelnik może przeanalizować wagi nauczonej sieci, aby stwierdzić, jak poszczególne neurony dostosowały swoje wagi, aby odpowiadać najsilniejszym sygnałem na przypisane im zwierzęta.

Ponieważ trudno jest nam wyobrażać sobie wielowymiarowe przestrzenie sygnałów wejściowych sieci, na zakończenie tego rozdziału przeanalizujemy zdolności klasyfikacyjne pojedynczego neuronu o dwóch wejściach. Przestrzeń sygnałów wejściowych jest w takim wypadku płaszczyzną, łatwo więc będzie zaznaczyć na niej punkty należące do rozpoznanych przez neuron klas. Zastosujemy takie same

neurony, jakich używaliśmy do tej pory – o unipolarnej sigmoidalnej funkcji aktywacji, a więc o wartościach na wyjściu z zakresu 0 - 1, i przyjmiemy, że neuron rozpoznaje dwie klasy obiektów, reagując na nie wartościami odpowiednio 0 i 1. Obie klasy punktów będziemy zaznaczać różnymi kolorami.

Zadanie 17. *Uzupełnij funkcję* `ucz1e` *o rysowanie wykresów:*

- *parametrów granicy decyzyjnej neuronu – jej nachylenia i przesunięcia, wg (4.3),*
- *uproszczonej granicy decyzyjnej, pokazującej dwoma kolorami klasy, do jakich neuron przypisuje punkty z ciągu uczącego.*

Nową funkcję nazwij `ucz1f`

Stworzenie wykresów współczynników nachylenia i przesunięcia granicy decyzyjnej neuronu (ich zmian w kolejnych krokach uczenia) na podstawie wzoru (4.3) nie jest trudne. Dużo więcej pracy wymaga dodanie do funkcji wykresu uproszczonych granic decyzyjnych. Kod źródłowy funkcji `ucz1f` (wraz z wyczerpującym komentarzem) można znaleźć na stronie internetowej autora.

Zaproponowana funkcja odświeża wszystkie wykresy co ustaloną liczbę kroków. Sprawdza wtedy na wszystkich punktach z ciągu uczącego odpowiedź neuronu, dzieli punkty na dwie macierze oznaczające rozpoznane klasy (granicą podziału jest wartość 0.5 na wyjściu neuronu), po czym rysuje wykres obu macierzy w dwóch kolorach przy użyciu funkcji `plot3`.

Aby takie uproszczone wykresy przedstawiały w zadowalający sposób granice decyzyjne sieci, punktów w zbiorze uczącym musi być dużo (np. 2000). Ustaleniem takich parametrów, jak: liczba przykładów w zbiorze uczącym, liczba przykładów prezentowanych sieci podczas jednego kroku uczenia, czy liczba kroków między odświeżeniami wykresu, zajmuje się skrypt `test1f`. Pozwala on również użytkownikowi wybrać problem klasyfikacyjny, jakiego ma się nauczyć pojedynczy neuron.

Program proponuje trzy przykłady zadań:

- problem liniowo separowalny – losowe punkty z kwadratu (0,0) - (1,1) rozdzielone są linią prostą o parametrach definiowanych przez użytkownika,
- problem prawie liniowo separowalny – losowe punkty z kwadratu (0,0) - (1,1) leżące w środku koła o parametrach definiowanych przez użytkownika należą do jednej klasy, a punkty leżące na zewnątrz – do drugiej,
- problem liniowo nieseparowalny – punkty są losowane z kwadratu (0,0) - (1,1), a użytkownik może zdefiniować parametry dwóch kół, wewnątrz których punkty należą do jednej klasy, a na zewnątrz – do drugiej.

W pierwszym zadaniu granica decyzyjna, jakiej ma się nauczyć neuron jest w pełni zgodna z granicą, jaką jest on w stanie wytworzyć. Zdolność neuronu do

rozwiązania dwóch ostatnich zadań zależy od parametrów kół wybranych przez użytkownika. Jeśli będą one takie, że w kwadracie (0,0) - (1,1) żądana granica decyzyjna będzie prawie liniowa, neuron będzie w stanie się jej nauczyć z dość dużą dokładnością, jeśli jednak będzie ona mocno zakrzywiona, lub klasy będą rozłączne, neuron nigdy nie będzie w stanie się jej nauczyć.

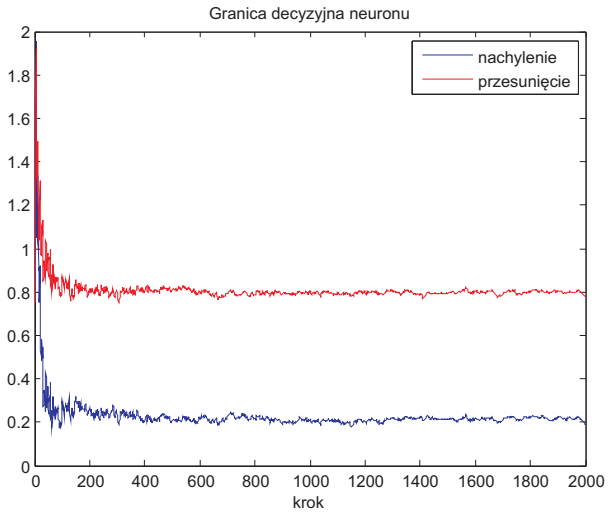
Przy pomocy programu `test1f` Czytelnik może obserwować próby pojedynczego neuronu do dopasowania swojej liniowej granicy decyzyjnej do liniowych, prawie liniowych, bądź nieliniowych granic decyzyjnych zadanych mu do nauczenia. Skrypt `test1f` można również w łatwy sposób uzupełnić o generowanie innych ciągów uczących, przedstawiających znacznie bardziej skomplikowane granice decyzyjne.

Poniżej przedstawiamy przykładowy wydruk działania programu `test1f` dla pierwszej klasy zadań.

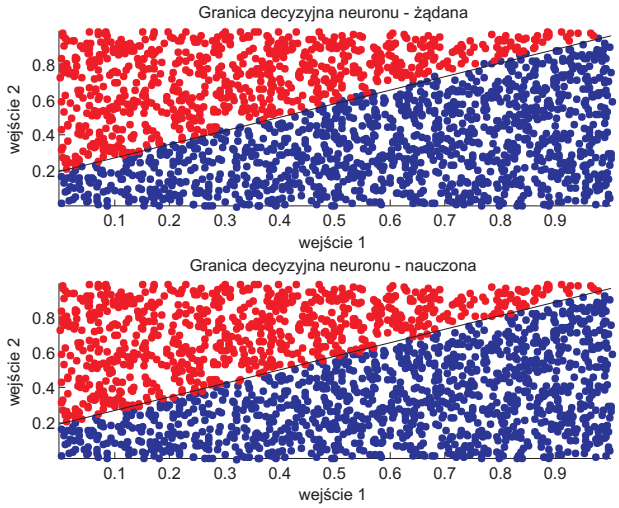
```
******************************************************
Sieć jednowarstwowa uczy się różnych zadań klasyfikacji
- wersja z wykresem błędu MSE
           wykresem błędu CE
           wykresem wag sieci
         i wykresem granic decyzyjnych neuronów
******************************************************
Wybierz zestaw danych do nauczenia się przez sieć:
  1 - linia prosta - problem liniowo separowalny
  2 - jedno koło - problem prawie liniowo separowalny
  3 - dwa koła - problem liniowo nieseparowalny
Co wybierasz? 1
Wybrałeś   1 - linia prosta - problem liniowo separowalny
Podaj liczbę przykładów uczących: 2000
Podaj  nachylenie  prostej: 0.8
Podaj przesunięcie prostej: 0.2
----------------------------------------
Ciąg uczący:
2000 punktów nad i pod prostą: y = 0.8*x+0.2
----------------------------------------
Podaj maksymalną liczbę kroków uczenia: 2000
Podaj liczbę przykładów do pokazania w jednym kroku uczenia: 10
Co ile kroków odświeżać wykres? 100
Podaj docelowy błąd średniokwadratowy MSE (na całym ciągu uczącym): 0.001
----------------------------------------
Sieć przed uczeniem:
----------------------------------------
Tak działa:
Błąd średniokwadratowy        MSE =    0.1106913
Procent błędnych klasyfikacji  CE =   37.9
----------------------------------------
Naciśnij ENTER, aby rozpocząć uczenie...
```

```
Uczę sieć...
Ustaw sobie okienka i naciśnij ENTER...
Epoka:  100 / 2000 - błąd MSE:  0.0509804 /  0.001 - błąd CE: 7.1
Epoka:  200 / 2000 - błąd MSE:  0.0373927 /  0.001 - błąd CE: 5.6
(...)
Epoka: 1900 / 2000 - błąd MSE:  0.0147360 /  0.001 - błąd CE: 2.1
Epoka: 2000 / 2000 - błąd MSE:  0.0147459 /  0.001 - błąd CE: 1.8
---------------------------------------
Sieć po uczeniu:
Wpo =    0.4989
        -2.0519
         2.6410
---------------------------------------
Tak działa:
Błąd średniokwadratowy        MSE =    0.0147459
Procent błędnych klasyfikacji  CE =    1.8
Naciśnij ENTER...
```

Nie będziemy w tym miejscu zamieszczać wykresu przebiegu błędu średniokwadratowego, ani wag neuronu podczas uczenia, gdyż znamy te rodzaje wykresów z poprzednich ćwiczeń. Warto natomiast zwrócić uwagę na rys. 4.8, który pokazuje, jak szybko wagi neuronu osiągnęły wartości zapewniające poprawne parametry granicy decyzyjnej (nachylenie i przesunięcie). Samą granice decyzyjną, jakiej nauczył się neuron, przedstawia zaś rys. 4.9.

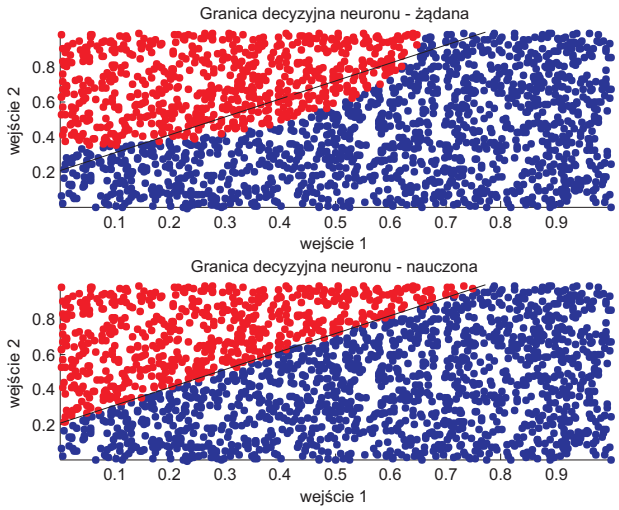


Rysunek 4.8. Pojedynczy neuron rozwiązuje zadanie liniowo separowalne - wykres zmian parametrów granic decyzyjnych

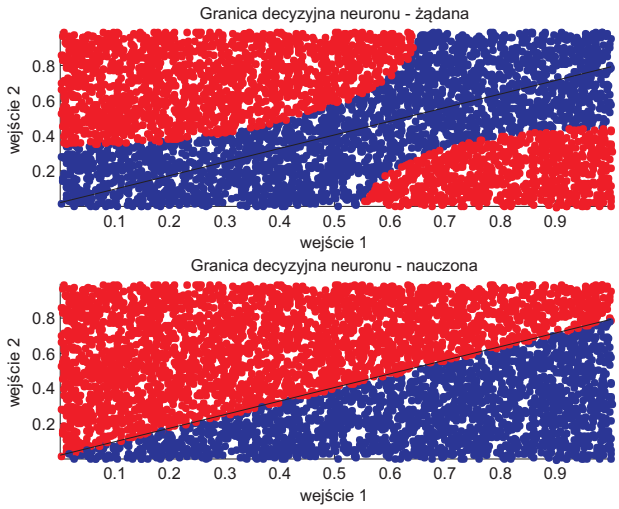


Rysunek 4.9. Pojedynczy neuron rozwiązuje zadanie liniowo separowalne - wykres granic decyzyjnych

Sprawdzenie przebiegu uczenia neuronu dla pozostałych dwóch zadań pozostawimy Czytelnikowi. Przedstawiamy jedynie przykładowe wykresy granic decyzyjnych neuronu, gdy granica decyzyjna jest wycinkiem okręgu o środku w punkcie (0.0,1.0) i promieniu 0.7 (rys. 4.10, końcowy błąd CE: 7.1%), oraz gdy granica decyzyjna jest rozłączna, wyznaczona przez dwa koła o środkach w punktach (0.0,1.0) i (1.0,0.0), a promieniach odpowiednio 0.7 i 0.5 (rys. 4.11, końcowy błąd CE: 42.7%).



Rysunek 4.10. Pojedynczy neuron rozwiązuje zadanie prawie liniowo separowalne - wykres granic decyzyjnych



Rysunek 4.11. Pojedynczy neuron rozwiązuje zadanie liniowo nieseparowalne - wykres granic decyzyjnych

Rozdział 5

Badanie procesów uczenia dwuwarstwowych sieci neuronowych

W niniejszym rozdziale zastosujemy kolejne ulepszenia funkcji uczącej sieci dwuwarstwowe, m.in. wprowadzimy pojęcie epoki. Przede wszystkim jednak będziemy obserwować, jak sieć tworzy nieliniowe granice decyzyjne z liniowych granic pojedynczych neuronów warstwy pierwszej. Niejako „przy okazji" zapoznamy się ze zjawiskami towarzyszącymi procesom uczenia oraz zasadami, do których należy się stosować, aby efektywnie uczyć sieci.

5.1. Komunikacja z użytkownikiem, podstawowe wykresy

Zadanie 18. *Na podstawie ćwiczeń dotyczących sieci jednowarstwowych zmodyfikuj funkcję uczącą oraz skrypt testujący sieć dwuwarstwową:*

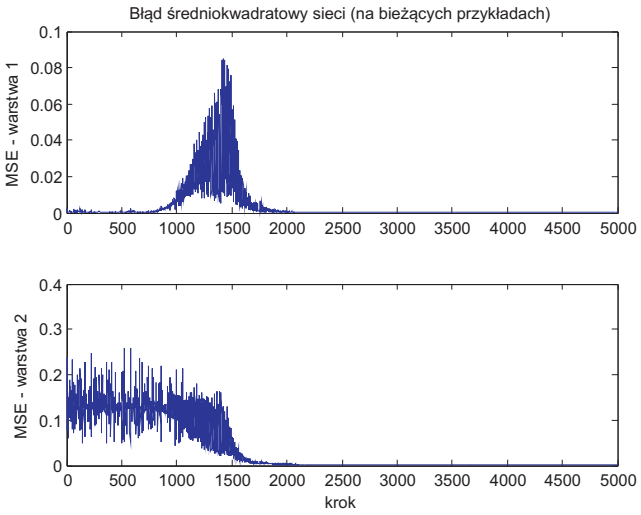
- *Uzupełnij skrypt* `test2` *o komunikację z użytkownikiem - przedstawianie zadania rozwiązywanego przez sieć, wag sieci oraz wyników jej działania przed i po uczeniu.*
- *Uzupełnij funkcję* `ucz2` *o wykreślanie błędu średniokwadratowego MSE dla obu warstw sieci.*
- *Dodaj do skryptu* `test2` *możliwość określenia liczby neuronów w warstwie ukrytej sieci i zmień odpowiednio instrukcję konstruującą sieć.*
- *Napisz funkcję* `sprawdz2`*, podającą błąd średniokwadratowy oraz liczbę błędnych klasyfikacji na dowolnym ciągu danych uczących. Wykorzystaj ją w skrypcie* `test2` *do sprawdzania działania sieci przed i po uczeniu na całym ciągu uczącym.*
- *Wprowadź do funkcji* `ucz2` *możliwość wcześniejszego zakończenia uczenia po osiągnięciu zadanego poziomu błędu średniokwadratowego na wyjściu lub po upływie maksymalnej liczby kroków uczenia. W skrypcie* `test2` *pytaj użytkownika o te wielkości i przesyłaj je do funkcji* `ucz2` *jako parametry.*

Po wykonaniu ćwiczeń z poprzedniego rozdziału, dotyczących sieci jednowarstwowych, wprowadzenie proponowanych zmian do funkcji i skryptu obsługujących sieci dwuwarstwowe nie powinno stanowić dla Czytelnika większego problemu, a może okazać się ciekawym zadaniem programistycznym.

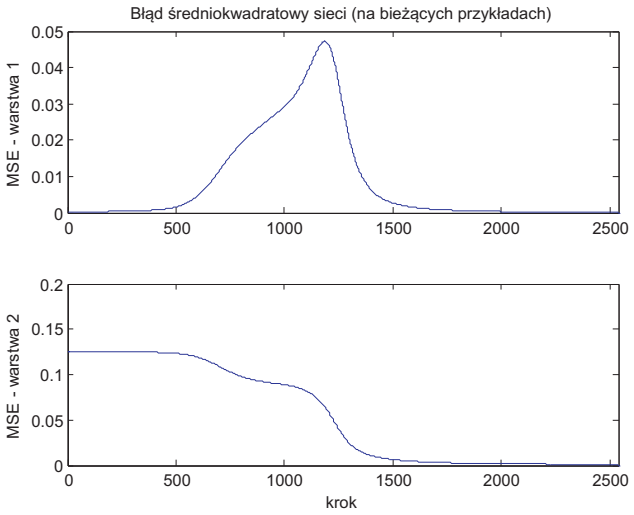
Kody źródłowe oraz skompilowany skrypt można znaleźć na stronie autora. Wersja programu wzbogacona o wyświetlanie komunikatów dla użytkownika nazwana została `test2a`, zaś pozostałe założenia wdrożone zostały w funkcji `ucz2b` i `test2b`. Już tak proste udogodnienia pozwolą nam przeprowadzić pierwsze badania i obserwować proces uczenia sieci.

Większość sieci, zainicjowanych wagami losowymi, zaczyna od zaklasyfikowania wszystkich punktów do jednej klasy. Błąd klasyfikacji wynosi wtedy 50% (bo przecież połowa punktów jest zupełnie przypadkowo przypisana do właściwej klasy) i na pierwszy rzut oka nie wydaje się być duży. Zauważmy jednak, że taki sam efekt możemy uzyskać rzucając monetą, więc aby jakikolwiek algorytm decyzyjny był użyteczny, błąd klasyfikacji musi spaść znacznie poniżej tej wartości, a najlepiej gdyby zbliżył się do 0%. W naszym zadaniu, gdy mamy do zaklasyfikowania 4 punkty, błąd klasyfikacji może się zmieniać skokowo co 25%.

Mimo, że w tym ćwiczeniu funkcja `ucz2b` nie rysuje jeszcze błędu klasyfikacji, to na przykładowych wykresach błędu średniokwadratowego (rys. 5.1,5.2), przedstawiających możliwe przebiegi uczenia sieci, widać wyraźnie chwile, w których sieć nauczyła się poprawnie klasyfikować kolejne dwa brakujące przykłady. Dzięki temu, że na wykresie uwzględniliśmy również domniemany błąd pierwszej warstwy sieci, można na nim zauważyć charakterystyczne „przesilenie" w momencie opanowania przez sieć ostatniego przykładu.



Rysunek 5.1. Przebieg uczenia sieci (`ucz2b`) – błąd średniokwadratowy (zadanie XOR, sieć 2-2-1, 2 pokazy w kroku uczenia)



Rysunek 5.2. Przebieg uczenia sieci (`ucz2b`) – błąd średniokwadratowy (zadanie XOR, sieć 2-2-1, 4 pokazy w kroku)

Jak już zauważono w rozdziale 3, problem klasyfikacji XOR jest zadaniem liniowo nieseparowalnym, a to, czy (i jak szybko) sieć znajdzie punkt w wielowymiarowej przestrzeni wag zapewniający prawidłowe rozwiązanie, zależy od wag początkowych i od przebiegu uczenia. Czytelnik może sam zaobserwować, wielokrotnie uruchamiając uczenie sieci, że nawet przy tak prostym zadaniu, nie każda sieć jest w stanie znaleźć punkt w przestrzeni wag zapewniający minimum funkcji błędu i nauczenie się zadania.

Funkcja `ucz2b` ma znaczącą wadę — podobnie jak jedna z pierwszych funkcji uczących sieci jednowarstwowe, oblicza błąd MSE jedynie na przykładach pokazanych w danym kroku i na jego podstawie podejmuje decyzję o zakończeniu uczenia. Może to prowadzić do zbyt wczesnego zakończenia uczenia, gdy sieć nie zdąży się jeszcze nauczyć rozwiązywać całego zadania, ale w danym kroku uczenia trafi na takie przykłady, dla których błąd średniokwadratowy osiągnie zakładany poziom. Dlatego też w następnej kolejności zajmiemy się właściwym wykorzystaniem danych zawartych w ciągu uczącym. Tymczasowym sposobem rozwiązania tego problemu jest pokazywanie sieci wszystkich przykładów z ciągu uczącego w jednym kroku uczenia (wspomniana już metoda *batch*) oraz podanie bardzo małego docelowego błędu uczenia. Zachęcamy Czytelnika do przeanalizowania wykresów dla różnych wartości następujących parametrów: liczby przykładów pokazywanych w kroku oraz liczby neuronów w ukrytej warstwie sieci.

5.2. Właściwe użycie danych uczących

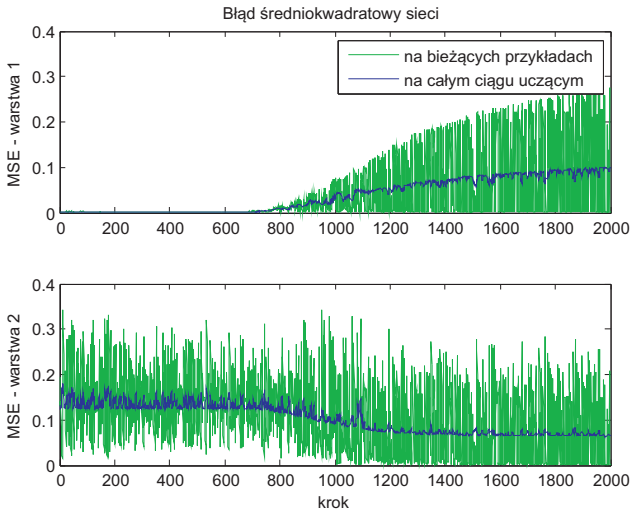
Zadanie 19. *Wprowadź kolejne ulepszenia do funkcji uczącej sieć, aby na koniec właściwie używać danych uczących:*

- *Uzupełnij Funkcję* `ucz2` *o możliwość prezentacji sieci kilku przykładów w jednym kroku uczenia. Na wykresie błędu MSE pokaż błąd uśredniony względem wszystkich przykładów pokazanych w danym kroku uczenia., ale również błąd obliczony na zakończenie każdego kroku na całym ciągu uczącym (użyj do tego funkcji* `sprawdz2`*).*
- *Wprowadź podział na: pokaz – krok – epoka podczas wybierania przykładów uczących. Zadbaj o to, aby przykłady prezentowane sieci nie powtarzały się w kolejnych krokach aż do pokazania jej wszystkich przykładów z ciągu uczącego. Dopiero po pokazaniu wszystkich przykładów uczących (jedna epoka uczenia) sprawdzaj błąd średniokwadratowy i podejmuj decyzję o ewentualnym zakończeniu uczenia.*
- *Wprowadź podział danych uczących na ciągi: uczący – sprawdzający – testujący. W funkcji* `ucz2` *używaj ciągów uczącego (do obliczania poprawek wag) i sprawdzającego (do rysowania wykresów i podejmowania decyzji o zakończeniu uczenia). W skrypcie* `test2` *użyj ciągu testującego do sprawdzenia działania sieci po uczeniu (a dla zaspokojenia ciekawości – również przed uczeniem).*

Uzupełnienie funkcji o możliwość prezentacji kilku przykładów uczących w jednym kroku należy wykonać podobnie, jak dla sieci jednowarstwowych. W tym przypadku kumulowane będą po prostu poprawki wag obu warstw sieci. Do pokazania na wykresie błędu średniokwadratowego zarówno błędów obliczanych na przykładach pokazanych w danym kroku, jak i na całym ciągu uczącym, będziemy musieli na zakończenie każdego kroku uczenia sprawdzić działanie sieci przy pomocy funkcji `sprawdz`. Jeśli chcemy na wykresie błędu pokazać błędy obu warstw sieci (obliczane na całym ciągu uczącym), trzeba będzie poprawić tę funkcję tak, aby zwracała wyjścia błędy oraz błąd średniokwadratowy obu warstw (autor użył w tym przypadku drugiej funkcji o nazwie `sprawdz2bis`).

Funkcja `ucz2bb` implementuje pierwsze zadanie z ćwiczenia 19. Oprócz wykresu błędu średniokwadratowego na całym ciągu uczącym, poprawiono w niej również podejmowanie decyzji o zakończeniu uczenia — teraz decyduje o tym osiągnięcie założonej wartości na całym ciągu uczącym. Jest to rozwiązanie o wiele lepsze niż poprzednio. Sieć nie powinna już zbyt szybko kończyć uczenia, lecz dopiero po nauczeniu się wszystkich przykładów z ciągu uczącego. Jednak w pełni prawidłowe rozwiązanie dopiero przed nami: sprawdzanie sieci co epokę na niezależnym od uczącego ciągu sprawdzającym. Wprowadzimy je w następnym kroku.

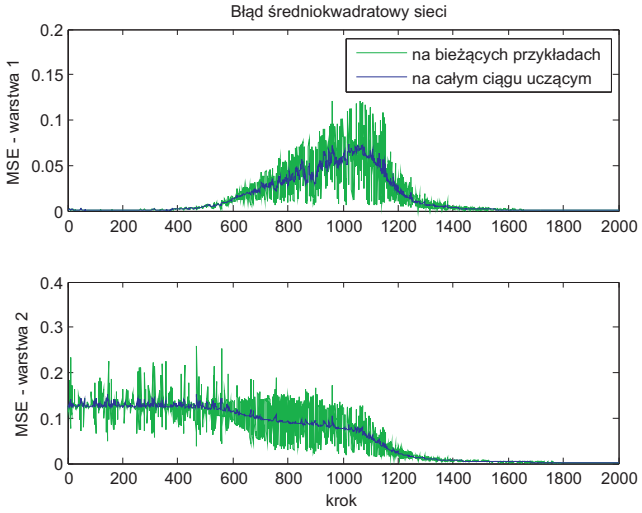
Rys. 5.3 pokazuje przykład sieci, która nie jest w stanie nauczyć się problemu XOR. Na obecnym wykresie błędu widać, jak błąd obliczany tylko na jednym przykładzie pokazywanym w danym kroku różni się od błędu obliczanego na całym ciągu uczącym. Na rys. 5.4 możemy z kolei zauważyć, że zwiększając liczbę przykładów pokazywanych w jednym kroku uczenia możemy wygładzić przebieg błędu obliczanego na całym ciągu uczącym (oceniającego postępy uczenia sieci), jak również zmniejszyć wahania bieżącego błędu na przykładach pokazywanych sieci. Sprawdzenie kształtu wykresu błędu przy czterech pokazach w kroku uczenia (czyli przy uczeniu wsadowym).



Rysunek 5.3. Przebieg uczenia sieci (`ucz2bb`) – błąd średniokwadratowy (zadanie XOR, sieć 2-2-1, 1 pokaz w kroku uczenia)

Organizacja prezentacji sieci przykładów uczących z podziałem na pokaz – krok – epokę uczenia została już wyjaśniona w rozdziale 4.4 (rys. 4.4). Przy sieciach jednowarstwowych nie wprowadziliśmy jednak do funkcji pojęcia epoki. Obecnie uczynimy to w funkcji `ucz2bbb`. Wymaga to jedynie niewielkiego przeorganizowania pętli po wszystkich przykładach uczących. Zostawimy to zadanie do wykonania Czytelnikowi. Zajmiemy się teraz innym problemem – prawidłowym podziałem danych uczących na trzy ciągi: uczący, sprawdzający i testujący.

We wszystkich dotychczasowych programach funkcje uczące podejmowały decyzję o zakończeniu uczenia na podstawie błędu, jaki sieć popełnia na przykładach z ciągu uczącego, czyli na danych użytych uprzednio do obliczania poprawek wag. Można powiedzieć, że postępowanie takie przypomina trochę „bycie sędzią w swo-



Rysunek 5.4. Przebieg uczenia sieci (`ucz2bb`) – błąd średniokwadratowy (zadanie XOR, sieć 2-2-1, 2 pokazy w kroku uczenia)

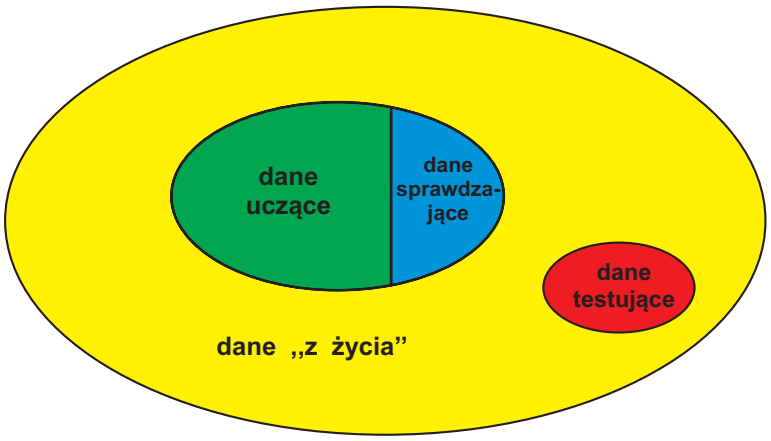
jej własnej sprawie" (lub np. zadawanie studentom na egzaminie dokładnie takich samych zadań, jakie były przerabiane na ćwiczeniach).

Aby obiektywnie stwierdzić, czy sieć nauczyła się rozwiązywać zadany jej problem, należy sprawdzić ją na ciągu danych, którego sieć nie „widziała" podczas uczenia. Dopiero po sprawdzeniu działania sieci na nowych danych będziemy w stanie ocenić jej zdolność do generalizacji wiedzy, czyli do jej uogólniania na nowe przypadki (dokładnie na tej samej zasadzie na egzaminie chcemy sprawdzić zdolność studenta do logicznego myślenia i rozwiązywania problemów z jakiejś dziedziny, a nie do „wkuwania" na pamięć).

Głównym celem uczenia sieci neuronowych jest wydobycie z danych uczących wiedzy dotyczącej jakiegoś problemu przy jak największej zdolności do generalizacji tej wiedzy na nowe, nie widziane jeszcze przypadki. Dopiero sieć o takich własnościach sprawdzi się po zakończeniu uczenia w realnych sytuacjach, na danych „z życia wziętych".

Sprawdzanie działania sieci podczas uczenia przy użyciu danych sprawdzających, niezależnych od danych użytych bezpośrednio do obliczania poprawek wag jest działaniem poprawnym, ale niewystarczającym do otrzymania w pełni obiektywnej oceny jakości sieci. Dane te są bowiem również dadzą nam obciążone oszacowanie. Mimo, że nie są one używane bezpośrednio do adaptacji wag, to jednak przecież właśnie one decydują o zakończeniu procesu uczenia — czyli również na niego wpływają.

Dlatego też, do ostatecznej, w pełni niezależnej oceny jakości sieci i jej zdolności do uogólniania wiedzy na nowe przypadki, powinniśmy po zakończeniu uczenia użyć trzeciego zbioru danych — niezależnego od dwóch poprzednich. Oczywiście należy zadbać o to, aby we wszystkich trzech ciągach znalazły się dane reprezentatywne dla problemu, jakiego ma się nauczyć sieć. Oznacza to, że np. w zadaniach klasyfikacji wszystkie klasy rozpoznawanych obiektów powinny znaleźć się w równych proporcjach w każdym z tych trzech ciągów. Mówiąc obrazowo: nie powinniśmy sieci uczyć rozpoznawać pacjentów zdrowych, a testować ją na przypadkach pacjentów chorych. W zadaniach predykcji giełdowej nie powinniśmy uczyć sieci na okresie hossy, a testować w czasie bessy, itd. Zalecany podział danych uczących ilustruje rys. 5.5.



Rysunek 5.5. Schematyczny podział danych uczących na trzy podzbiory

W nazewnictwie angielskim najczęściej spotkamy następujące odpowiedniki polskich określeń:

· ciąg uczący – *training set*,
· ciąg sprawdzający – *validating set*,
· ciąg testujący – *testing set*.

Zarówno pakiet MATLAB, jak i STATISTICA stosują podczas uczenia sieci przedstawiony wyżej sposób podziału danych. Zwykle są one dzielone w proporcjach zbliżonych do: 70%-15%-15%.

Jak już wspominano, do uczenia sieci potrzeba wielu, często wielu tysięcy przykładów uczących. Konieczność ich podziału na trzy podzbiory dodatkowo zwiększa ten wymóg. W przypadku nie dysponowania odpowiednią liczbą danych, stosuje się różne techniki „sztucznego” zwiększenia ich liczebności, np. *cross-validation*, których opis można znaleźć w podanej literaturze.

W naszym prostym przykładzie dysponujemy zaledwie czterema punktami definiującymi problem XOR i w następnych ćwiczeniach nadal będziemy używać tych czterech punktów zarówno jako ciągu uczącego, jak i sprawdzającego oraz testującego. Funkcja `ucz2bbb` jest jednak przystosowana do użycia różnych danych uczących i sprawdzających, zaś dane testujące definiuje się w skrypcie `test2bbb`. Odtąd wszystkie funkcje uczące będą już w pełni poprawnie używały danych uczących.

Podział danych uczących na podzbiory i porównanie działania sieci na danych uczących i sprawdzających przydaje się podczas uczenia do wykrywania zjawiska przeuczenia (ang. *overfitting*). Dyskusję tego problemu zostawimy jednak na dalsze rozdziały.

5.3. Dodatkowe wykresy

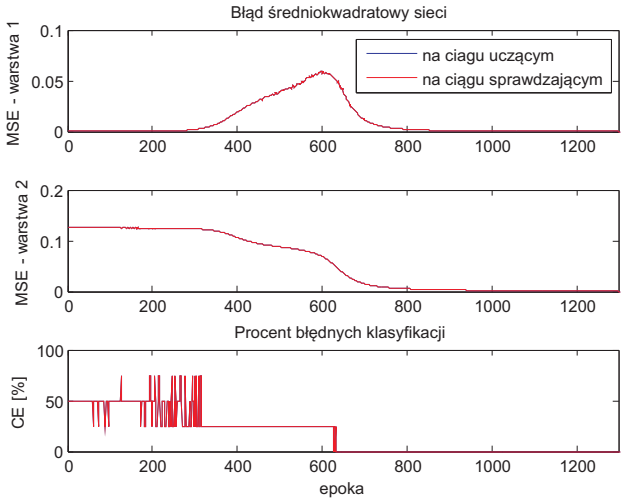
Zadanie 20. *Dodaj do funkcji* `ucz2` *możliwość rysowania:*

- *wykresu zmian błędu klasyfikacji podczas uczenia,*
- *wykresu zmian wszystkich wag sieci podczas uczenia,*
- *wykresu zmian wag poszczególnych neuronów i wag prowadzących do poszczególnych wejść sieci.*

Uzupełnienie funkcji o rysowanie błędu klasyfikacji nie jest trudne, zważywszy na fakt, że dodaliśmy już taki wykres do funkcji uczącej sieć jednowarstwową. Rys. 5.6, przedstawiający przykładowy przebieg uczenia sieci, potwierdza wcześniejsze obserwacje, że kolejne „przesilenia błędów obu warstw sieci" odpowiadają momentom, w których sieć opanowuje poprawne klasyfikacje kolejnych punktów z ciągu uczącego.

Dodanie do funkcji `ucz2` rysowania wszystkich wag w sieci na jednym wykresie jest zabiegiem równie prostym, jak w przypadku sieci jednowarstwowych. Wykres taki pozwala nam obserwować np., czy wagi neuronów nie osiągają zbyt dużych wartości, albo czy nie podlegają zbyt dużym oscylacjom (co znacznie utrudnia znalezienie optymalnych wartości).

Przydatna może się okazać również analiza wag poszczególnych neuronów, a także wag prowadzących do poszczególnych wejść sieci. Informacje te można wykorzystać do kształtowania architektury sieci. W przypadku zadań bardziej skomplikowanych od naszego problemu XOR, często nie wiadomo, jak duża powinna być sieć, aby była w stanie nauczyć się jego rozwiązywania. Najczęściej stosuje się proste architektury sieci z jedną warstwą ukrytą. Według twierdzenia znanego pod nazwą *Universal Approximation Theorem*, dla każdego problemu można znaleźć wartości wag sieci z jedną warstwą ukrytą zapewniające rozwiąza-



Rysunek 5.6. Przebieg uczenia sieci (`ucz2c`) – błąd średniokwadratowy i błąd klasyfikacji (zadanie XOR, sieć 2-2-1, 2 pokazy w kroku uczenia)

nie problemu z założoną dokładnością – oczywiście pod warunkiem dysponowania odpowiednią liczbą przykładów uczących oraz neuronów w warstwie ukrytej. Nasuwa się jednak pytanie: ile ta warstwa ukryta powinna zawierać neuronów, aby jej zdolności (można również powiedzieć: jej pojemność) były wystarczające do nauczenia się danego problemu.

Stosuje się dwa podejścia do kształtowania architektury sieci – wstępujące i zstępujące. W uproszczeniu można powiedzieć, że pierwsze z nich zakładają rozrost sieci począwszy od małych rozmiarów aż do uzyskania zadowalających rezultatów uczenia. Druga grupa zakłada start uczenia na sieci o wstępnie dobranej, większej liczbie neuronów, a następnie stosowanie różnych współczynników wrażliwości w celu usunięcia tych neuronów i wag, które w najmniejszym stopniu wpływają na jej działanie. Dwa przykłady takich metod noszą dość znaczące nazwy *Optimal Brain Damage* i *Optimal Brain Surgeon*.

Kolejna grupa algorytmów uczenia służących poprawie zdolności sieci do generalizacji wiedzy opiera się na uzupełnieniu optymalizowanej funkcji celu o dodatkowe składniki i na tej podstawie wyprowadzanie wzorów na poprawki wag. Przy pomocy tych metod można uzyskać dodatkowe efekty, np. niewielkie wartości wag sieci, czy odporność na błędy grube.

Często stosowanymi ogólnymi zasadami jest przyjęcie „piramidowej” architektury sieci (wiele wejść, mniej neuronów w warstwie ukrytej i mało wyjść) oraz stosowanie początkowej liczby neuronów w warstwie ukrytej równej średniej geo-

metrycznej z liczby wejść i wyjść. Przegląd wszystkich wspomnianych wyżej metod optymalizacji architektury sieci wraz z odpowiednimi wzorami i odnośnikami do literatury można znaleźć w [4].

Zaproponowana w ćwiczeniu metoda obserwacji wag na wykresie została zaimplementowana przez autora w funkcjach `ucz2d` i `ucz2dd` (odpowiednio: wszystkie wagi sieci oraz wagi poszczególnych neuronów i do poszczególnych wejść). Stosowne programy wykorzystujące te funkcje można znaleźć na stronie internetowej, mogą one posłużyć Czytelnikowi do przeprowadzenia własnych obserwacji, a następnie implementacji bardziej zaawansowanych metod. Należy przy tym pamiętać, że małe wagi o małych wartościach wcale nie muszą być wagami o małym znaczeniu dla funkcjonowania sieci.

Użycie sieci o rozmiarze odpowiadającym skali trudności danego zadania powinno przynieść sukces w procesie jej uczenia. Jak już wspomniano wcześniej przy wyborze sieci o odpowiednim rozmiarze, a także przy oszacowaniu minimalnego rozmiaru zbioru uczącego może pomóc teoria związana z wymiarem Vapnika-Chervonenkisa. W przypadku niedopasowania sieci do rozwiązywanego problemu, zaobserwować możemy dwa zjawiska – niedouczenie (ang. *underfitting*) i przeuczenie sieci (ang. *oerfitting*).

Czytelnik może z łatwością przetestować zachowanie sieci o różnych rozmiarach nawet przy użyciu programu rysującego tylko wykres błędu. Sieć o zbyt małej liczbie neuronów nie będzie w stanie nauczyć się wymaganego zadania. Sieć o rozmiarze zbliżonym do optymalnego nauczy się go w rozsądnej liczbie epok. W przypadku problemu XOR jest to sieć o dwóch neuronach w warstwie ukrytej. Dobre rezultaty można uzyskać również dla nieco większych sieci, np. 3, 5, a nawet 10 neuronów. Można tę nadmiarowość architektury tłumaczyć w taki sposób, że nawet jeśli dwa neurony nie będą w stanie osiągnąć optymalnych wag, to uda się je znaleźć pozostałym nadmiarowym neuronom, przy czym te pierwsze „niezbyt zdolne" neurony nie będą im przeszkadzać. Przy większej liczbie neuronów będzie można zauważyć niestabilność uczenia. Sieci o skrajnie zbyt dużym rozmiarze nie będą w stanie nauczyć się zadania w ogóle, podobnie jaj sieci zbyt małe.

Problem dopasowania sieci do rozwiązywanego zadania najlepiej ilustruje problem aproksymacji funkcji jednej zmiennej, będący treścią zadania 23. Warto poobserwować, jak zadaną funkcję przybliża sieć o zbyt małej, odpowiedniej oraz zbyt dużej liczbie neuronów.

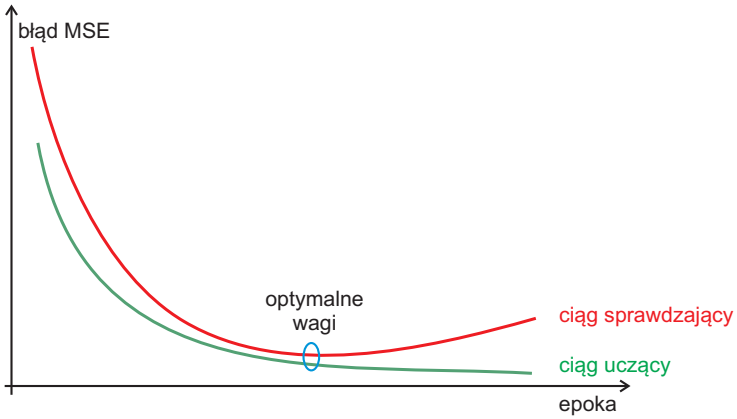
Dopasowanie rozmiaru sieci do rozwiązywanego zadania to jeden z kluczy do osiągnięcia przez nią umiejętności uogólniania danych na nowe przypadki. Podział danych uczących na trzy podzbiory: uczące, sprawdzające i testujące oraz analiza błędów sieci na poszczególnych ciągach pozwala przeanalizować zdolność generalizacji wiedzy. Schemat oceny błędów w poszczególnych przypadkach przed-

błąd	sieć za mała (*underfitting*)	sieć w sam raz	sieć za duża (*overfitting*)
dane uczące	duży	mały	b. mały
dane sprawdzające	duży	mały	duży
dane testujące	duży	mały	duży

Tablica 5.1. Błędy osiągane na podzbiorach danych uczących przez sieci o różnych rozmiarach

stawia tab. 5.1. Sieć o zbyt małej liczbie neuronów nie będzie w stanie nauczyć się stawianego jej zadania, osiągając duży błąd na wszystkich ciągach danych. Sieć dobrze dopasowana do rozwiązywanego problemu osiągnie niewielki błąd na danych uczących, a dzięki zdolności sieci do generalizacji wiedzy na nowe przypadki, błędy na ciągu sprawdzający oraz testującym również będą małe, niewiele większe od błędu na danych uczących. Sieć o zbyt dużym rozmiarze nauczy się przykładów z ciągu uczącego „na pamięć" osiągając na tych danych bardzo mały błąd, jednak nie będzie potrafiła uogólnić tej wiedzy na przykłady spoza ciągu uczącego, więc jej błąd na pozostałych dwóch podzbiorach danych będzie znacznie większy.

Problem przeuczenia sieci (*overfitting*) może wynikać jednak nie tylko z niedopasowania jej architektury do rozwiązywanego zadania, lecz również ze zbyt długiego procesu uczenia. Zjawisko to można wykryć obserwując i porównując przebieg błędu średniokwadratowego sieci podczas uczenia na ciągu uczącym z błędem na ciągu sprawdzającym. Rys. 5.7 ilustruje schematycznie, że zwykle podczas zbyt długiego uczenia sieci błąd obliczany na danych uczących stale maleje (sieć uczy się tych przykładów coraz dokładniej), natomiast błąd obliczany na danych sprawdzających w pewnym momencie zaczyna rosnąć. To jest właśnie moment, w którym należy przerwać uczenie i przywrócić wagi do wartości optymalnych sprzed kilku epok. Przed uczeniem należy więc określić maksymalną dozwoloną liczbę epok, podczas której błąd na ciągu sprawdzającym może chwilowo rosnąć, i pamiętać wagi sieci z poprzednich epok. Pakiety programistyczne MATLAB Neural Network Toolbox oraz STATISTICA Neural Networks mają zaimplementowaną taką właśnie metodę obserwacji błędu podczas uczenia.



Rysunek 5.7. Schematyczna ilustracja procesu wykrywania zjawiska przeuczenia podczas uczenia sieci

5.4. Jak sieć dwuwarstwowa rozwiązuje zadanie klasyfikacji XOR

Zadanie 21. *Uzupełnij funkcję* `ucz2` *o możliwość rysowania granic decyzyjnych podczas uczenia. Funkcja powinna rysować granicę całej sieci względem dwóch pierwszych wejść sieci (jeśli jest ich więcej niż 2), na podstawie wyjścia pierwszego neuronu ostatniej warstwy sieci (jeśli wyjść jest więcej niż 1).*

Granice sieci dwuwarstwowej mogą być granicami nieliniowymi, dlatego też najprostszym sposobem ich prawidłowego wykreślenia jest obliczenie odpowiedzi sieci na pewnej podprzestrzeni sygnałów wejściowych, w naszym wypadku dla obu wejść z zakresu od 0 do 1. W zaimplementowanej przez autora i dostępnej na stronie internetowej funkcji, rysowanie granic decyzyjnych odbywa się w sposób uproszczony przy użyciu funkcji `plot3` na punktach ze zbioru sprawdzającego (program `test2e`, rys. 5.8) oraz w sposób dokładny dzięki użyciu funkcji `surf` na całej analizowanej podprzestrzeni sygnałów wejściowych (program `test2ee`, rys. 5.9).

W obu tych programach zdefiniowano również przykładowe zbiory uczące dla różnych klas zadań klasyfikacyjnych.

```
Wybierz zestaw danych do nauczenia się przez sieć:
  1 - linia prosta - problem liniowo separowalny
  2 - jedno koło - problem prawie liniowo separowalny
  3 - dwa koła - problem liniowo nieseparowalny
  4 - trzy koła - problem liniowo nieseparowalny
  5 - XOR (4 punkty) - problem liniowo nieseparowalny
  6 - XOR (wiele punktów) - problem liniowo nieseparowalny
Co wybierasz?
```

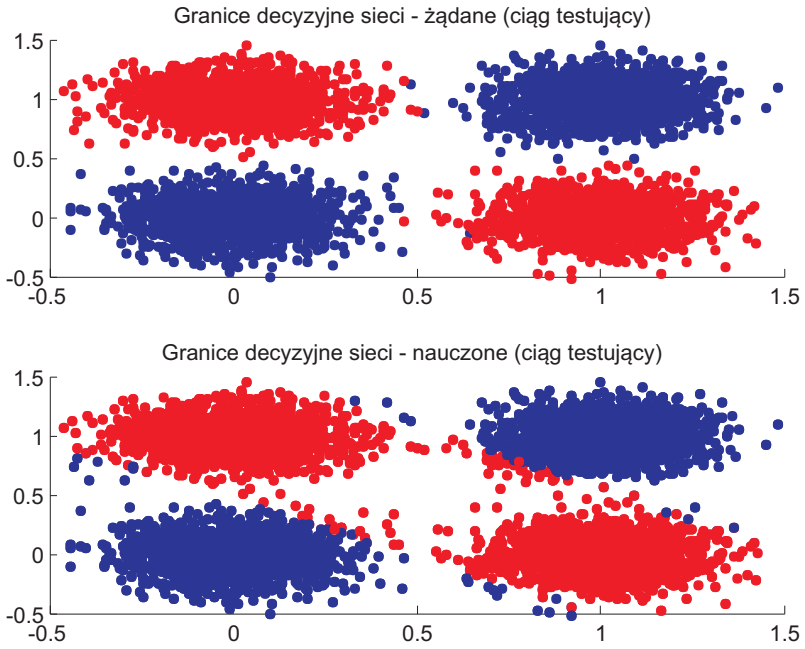
Pierwsze trzy zadania znamy już ze skryptu `test1f` dla sieci jednowarstwowych, zadanie 4 daje możliwość swobodniejszego kształtowania granicy decyzyjnej, której ma się nauczyć sieć, przez podanie 3 okręgów, wewnątrz których punkty będą zaliczane do tej samej klasy. Można więc zdefiniować obszar dla jednej klasy jako np. trzy rozłączne koła lub trzy koła tworzące bardzo nieliniową granicę, a następnie sprawdzić, ile neuronów w warstwie ukrytej potrzeba do nauczenia się takiej granicy przez sieć. W piątym zadaniu sieć uczy się klasycznego problemu XOR, zaś w zadaniu szóstym - problemu XOR zdefiniowanego w postaci czterech „chmur" punktów wylosowanych wokół punktów z klasycznego zadania.

Zachęcamy Czytelnika do przeprowadzenia własnych badań przebiegu uczenia sieci o różnej liczbie neuronów w warstwie ukrytej. Można do tego celu wykorzystać zadania zaproponowane we wspomnianym programie, albo tworzyć swoje zbiory danych uczących. Nietrudne jest również napisanie skryptu, który pozwoli użytkownikowi zdefiniować swoje własne zadanie przez podanie np. parametrów kół, czy linii dla punktów należących do poszczególnych klas. Można wreszcie wykorzystać szereg danych dostępnych w internecie (np. [7, 8]), dotyczących zadań klasyfikacyjnych, i zbadać granice decyzyjne sieci na rzeczywistych danych.

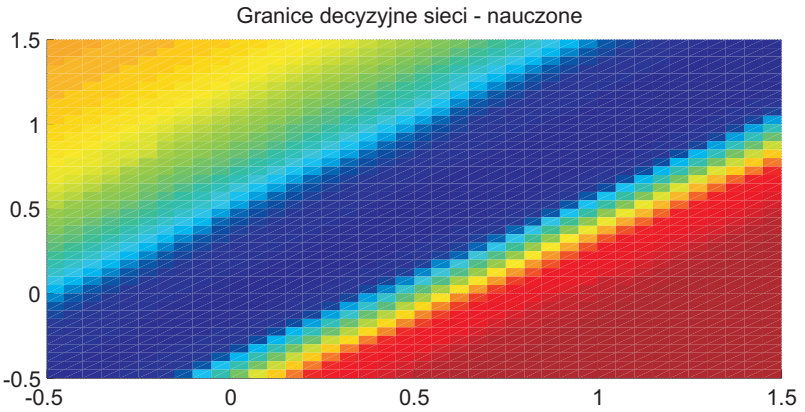
Zadanie 22. *Naucz sieć o dwóch neuronach w warstwie ukrytej rozwiązywać zadanie XOR. Przeanalizuj wagi nauczonej sieci. Korzystając z równania (4.3), oblicz parametry granic decyzyjnych wszystkich trzech neuronów sieci. Narysuj granice neuronów warstwy pierwszej wraz z czterema punktami definiującymi problem XOR. Narysuj granicę neuronu wyjściowego wraz z czterema punktami problemu XOR po przetworzeniu ich przez warstwę pierwszą. Czy dla drugiej warstwy sieci problem nadal jest liniowo nieseparowalny?*

Podczas rozwiązywania tego zadania Czytelnik może skonstruować następującą tabelę, aby przeanalizować przetwarzanie poszczególnych punktów problemu XOR przez kolejne warstwy sieci:

x_1	x_2	$u_1^{(1)}$	$u_2^{(1)}$	$y_1^{(1)}$	$y_2^{(1)}$	$u^{(2)}$	$y^{(2)}$
0	0						
0	1						
1	0						
1	1						



Rysunek 5.8. Przebieg uczenia sieci (`ucz2e`) – uproszczone granice decyzyjne sieci (zadanie XOR „w chmurze”, sieć 2-2-1, 4 pokazy w kroku uczenia)



Rysunek 5.9. Przebieg uczenia sieci (`ucz2ee`) – granice decyzyjne sieci (zadanie XOR „w chmurze”, sieć 2-2-1, 40 pokazów w kroku uczenia)

5.5. Inne rodzaje zadań, jakie może rozwiązywać sieć

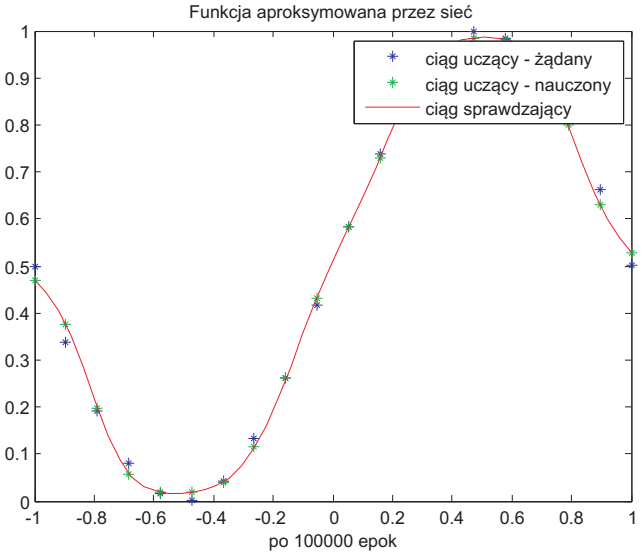
Zadanie 23. *Zmień funkcję* `ucz2` *oraz skrypt* `test2` *tak, aby można było przy ich pomocy śledzić postępy sieci o jednym wejściu i jednym wyjściu w nauce aproksymacji wybranej funkcji jednej zmiennej. Co ustaloną liczbę epok funkcja* `ucz` *powinna rysować wykres odwzorowania wejście – wyjście, jakiego sieć powinna się nauczyć oraz odwzorowania, jakiego nauczyła się po zadanej liczbie epok.*

Zaproponowane zadanie stanowi bardzo dobrą ilustrację dla omawianego wcześniej problemu dopasowania rozmiaru sieci do stopnia skomplikowania rozwiązywanego przez nią problemu. Czytelnik może zdefiniować zadania o różnym stopniu trudności (funkcje kwadratowe, wyższych rzędów, trygonometryczne i inne), a następnie obserwować, jak dokładnie będą w stanie nauczyć się tego zadania sieci o różnej liczbie neuronów w warstwie ukrytej. Proponujemy rysować również wykres błędu średniokwadratowego oraz wag sieci i obserwować ich zmiany w kolejnych epokach, wraz z coraz dokładniejszym przybliżaniem funkcji przez sieć.

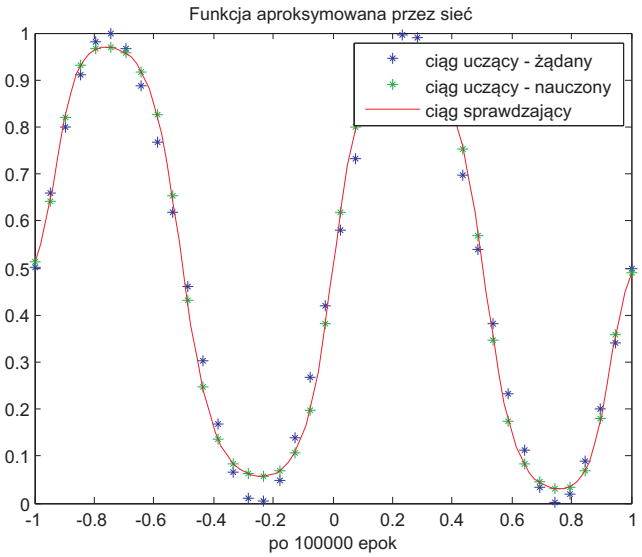
Przykłady działania sieci o 5 neuronach w warstwie ukrytej aproksymujących coraz bardziej skomplikowane funkcje *sin* przedstawiają rys. od 5.10 do 5.14. W przypadku ostatniej funkcji widać wyraźnie po przebiegu błędów obu warstw, a także po zmianach wag w poszczególnych epokach, jak skomplikowanym zadaniem optymalizacyjnym jest znalezienie odpowiednich wartości wag przy stosunkowo niewielkiej sieci. Podobnie, jak w przypadku problemu XOR bezpośrednia obserwacja „na żywo" zmian działania sieci podczas uczenia pokazuje, że na początkowym etapie uczenia sieć dość długo przynosi kiepskie rezultaty aproksymacji, w ogóle nie przypominającej zakładane funkcji, po czym kilka dość gwałtownych zmian wag przynosi szybką poprawę działania. Czytelnik może obserwować proces uczenia się przez sieci różnych funkcji jednej zmiennej, m.in. w różnym stopniu „zagęszczonych" wersji funkcji sinus uruchamiając program `test2fun` na stronie internetowej książki. Możliwe jest również zdefiniowanie swojej własnej funkcji, której ma się nauczyć sieć.

Szczególnym przypadkiem problemu aproksymacji funkcji jest predykcja szeregów czasowych. Dostosowanie naszych programów do tego zastosowania wymaga jedynie zmiany skryptu `test2` – odpowiedniego przygotowania danych uczących, i ewentualnie dostosowania wykresów prezentowanych w trakcie uczenia przez funkcję `ucz2`.

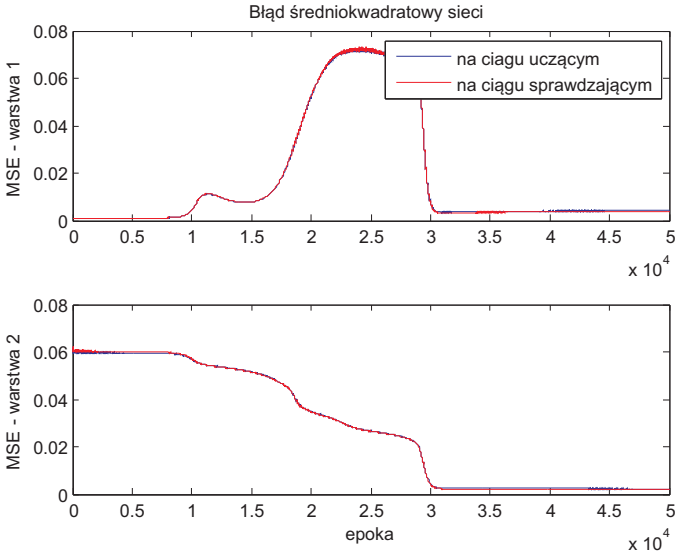
W zadaniu 23 zakładamy zależność funkcyjną wyjścia sieci od jej wejścia. W przypadku sieci o wielu wejściach i/lub wyjściach, czyli zależności wielowymiarowej, program powinien dać użytkownikowi wybór, zależność którego wyjścia od którego wejścia ma ilustrować. Niezależnie jednak od liczby wejść i wyjść sieci, to tę właśnie zależność (lub jej fragment) przedstawiamy na wykresie



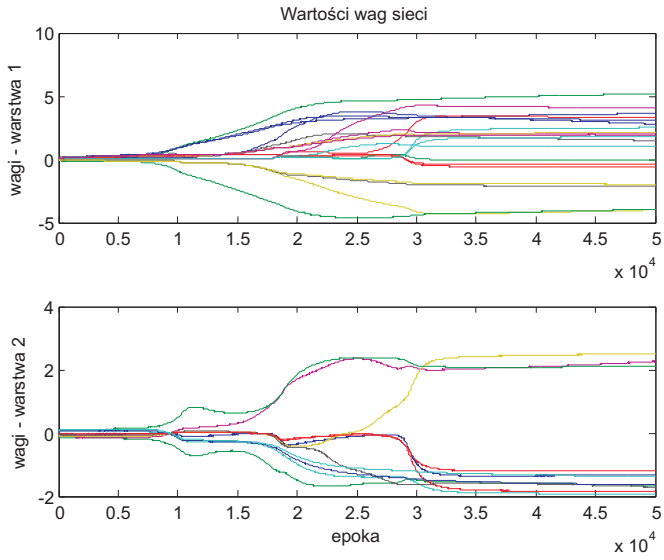
Rysunek 5.10. Funkcja $y = sin0.5x$ aproksymowana przez sieć o 5 neuronach ukrytych



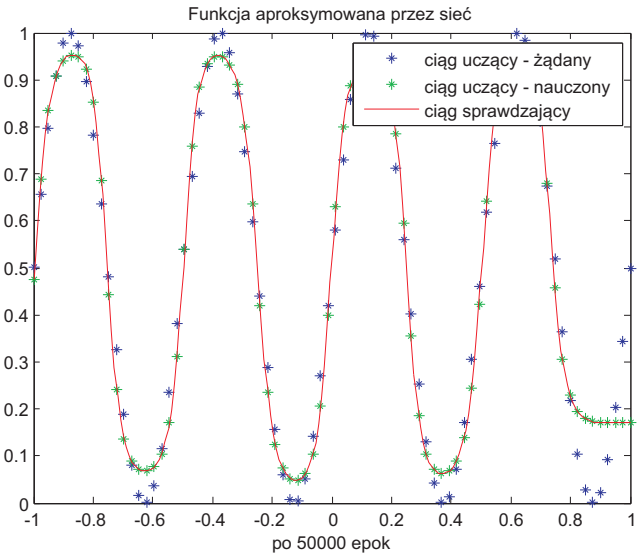
Rysunek 5.11. Funkcja $y = sinx$ aproksymowana przez sieć o 5 neuronach ukrytych



Rysunek 5.12. Funkcja $y = sin2x$ aproksymowana przez sieć o 10 neuronach ukrytych - błąd MSE podczas uczenia



Rysunek 5.13. Funkcja $y = sin2x$ aproksymowana przez sieć o 10 neuronach ukrytych - zmiana wag podczas uczenia



Rysunek 5.14. Funkcja $y = sin2x$ aproksymowana przez sieć o 10 neuronach ukrytych - funkcja

W przypadku predykcji nadal oczywiście istnieje zależność wyjść sieci od jej wejść – gdyby nie istniała, sieć nie miałaby się czego uczyć. Teraz jednak na wykresie ilustrującym działanie sieci nie będziemy pokazywać tej zależności. Umieścimy na nim jedynie kolejne odpowiedzi sieci dla kolejnych przykładów. W tej klasie zadań żądanym wyjściem sieci jest wartość przewidywanego szeregu czasowego w założonym horyzoncie. Może on być wyrażony w różnych jednostkach w zależności od problemu, jednak jego sensem pozostają dyskretne momenty w czasie przyszłym.

W najprostszym podejściu na wejściach sieci podaje się wartości tego samego szeregu z przeszłości. W przypadku notowań giełdowych przyszły (np. jutrzejszy) kurs jakiegoś instrumentu – akcji, waluty, kontraktu itp., traktowalibyśmy jako żądane wyjście, a jego wartości przeszłe (np. z dnia dzisiejszego, z wczoraj i przedwczoraj) podalibyśmy na wejścia sieci. Takie naiwne potraktowanie sieci neuronowej jako czarnej skrzynki przewidującej przyszłość (czy raczej „szklanej kuli") jest często zbyt entuzjastycznie stosowane przez początkujących badaczy sieci neuronowych, a przecież łatwo zauważyć, że sprowadza ją do prostego filtru MA (*moving average*). Dlatego też sugerujemy bardziej wyrafinowane podejście do finansowych zastosowań sieci neuronowych, uwzględniające np. staranny dobór wejść sieci (poprzedzony analizą statystyczną zależności zmiennych), prepro-

cessing tych wejść, a wreszcie zmianę rodzaju zadania z dokładnej predykcji na przybliżone określanie trendu lub na rozpoznawanie wzorców.

Przewiduje się oczywiście nie tylko notowania giełdowe. Predykcja zapotrzebowania na energię elektryczną jest bardzo ważnym zagadnieniem w rozległych systemach energetycznych, do którego również chętnie stosuje się modele oparte na sieciach neuronowych. Dodanie do funkcji `ucz` wykresu szeregu czasowego oraz odpowiednią zmianę organizacji danych uczących i sprawdzających (wykres powinien przedstawiać szereg czasowy „w jednym kawałku" lub chociaż w odcinkach) pozostawiamy Czytelnikowi.

5.6. Wydajniejsze uczenie sieci - adaptacyjny współczynnik uczenia

Stosowany przez nas do tej pory w procesie uczenia sieci algorytm najszybszego spadku gradientu jest najłatwiejszy do zrozumienia, najprostszy do zaprogramowania, ale i najmniej wydajny spośród znanych metod – podobne miejsce zajmuje np. algorytm bąbelkowy wśród metod sortowania. Algorytm najszybszego spadku gradientu jest doskonałym wstępem do bardziej zaawansowanych metod, o których wspomnimy później. Jednak nawet tę podstawową metodę uczenia można w prosty sposób uzupełnić o metody mające na celu podniesienie jej wydajności. W tym podrozdziale zajmiemy się zmiennym współczynnikiem uczenia.

Zadanie 24. *Zmodyfikuj funkcję* `ucz2` *tak, aby użytkownik miał możliwość wyboru następujących strategii doboru współczynnika uczenia:*

- *stały współczynnik uczenia o określonej wartości,*
- *stały współczynnik zależny od liczby wyjść,*
- *współczynnik adaptacyjny.*

Funkcja `ucz2` *powinna rysować przynajmniej wykres błędu średniokwadratowego MSE podczas uczenia. W przypadku użycia współczynnika adaptacyjnego, dodaj wykres zmian współczynnika uczenia w kolejnych epokach. Zbadaj, jak użycie różnych współczynników uczenia wpływa na szybkość uczenia sieci.*

Podobnie jak np. liczba przykładów pokazywanych sieci w pojedynczym kroku uczenia, dobór współczynnika uczenia również może mieć wpływ na efektywność tego procesu, choć oczywiście nie tak dużą, jak wybór algorytmu uczenia, poprawne zebranie danych uczących, czy użycie odpowiedniego rodzaju i rozmiaru sieci. Powszechnie stosuje się kilka algorytmów zmian współczynnika uczenia w kolejnych epokach.

Proponujemy uzupełnić nagłówek funkcji o parametr dotyczący współczynnika uczenia:

```
function [ W1po , W2po ] = ucz2 ( W1przed , W2przed , Pucz , Tucz , ...
                                                      Pspr , Tspr , ...
                                  maxLiczbaEpok    , ...
                                  liczbaPokazow    , ...
                                  czestoscWykresow , ...
                                  docelowyBladMSE  , ...
                                  wspUcz           )
```

Współczynnik uczenia nie może być ujemny ani równy 0. Można więc wykorzystać następującą metodę „kodowania", aby uniknąć konieczności definiowania algorytmu zmian współczynnika uczenia w osobnej zmiennej:

- jeśli `wspUcz > 0`, to przyjmujemy stały współczynnik uczenia równy zadanej wartości (najlepiej nie przekraczającej 1),
- dla `wspUcz = 0` przyjmujemy różne wartości współczynnika uczenia w każdej warstwie sieci, równe $1/K_l$ dla l-tej warstwy, gdzie K_l jest liczbą wyjść (neuronów) danej warstwy,
- dla ujemnych wartości – współczynnik adaptacyjny, startujący od podanej wartości (bezwzględnej).

Czytelnik może zaimplementować także inne sposoby zmian współczynnika uczenia, jak np. współczynnik malejący wraz z postępami uczenia. Należałoby w tym przypadku przyjąć jeszcze inną metodę kodowania wyboru algorytmu, wykorzystując np. fakt, ze współczynnik uczenia nie powinien być większy od 1.

Algorytm adaptacyjnego współczynnika uczenia dostosowuje jego wartość do bieżących postępów sieci w kolejnych epokach. Jest to kolejna oparta na intuicyjnych założeniach metoda stosowana podczas uczenia sieci neuronowych. Analogie do niej można znaleźć w postępowaniu ludzi – w tym wypadku nauczycieli, którzy przyspieszają proces uczenia, gdy mają zdolnych uczniów, szybko „chwytających" wiedzę, a łagodnieją gdy natrafią na bardziej „oporną" klasę lub skomplikowane zagadnienie, przy którym nawet zdolni uczniowie mają problemy ze zrozumieniem.

W przypadku sieci neuronowych taka adaptacja szybkości uczenia może następować na podstawie bieżącej analizy błędu średniokwadratowego na ciągu sprawdzającym (co i tak już czynimy w celu podjęcia decyzji o kontynuowaniu lub zakończeniu uczenia, jak również do rysowania wykresu błędu MSE) oraz modyfikacji współczynnika uczenia np. według wzoru

$$\eta_{n+1} = \begin{cases} \rho_d \cdot \eta_n & \text{gdy} \varepsilon_n > k_w \cdot \varepsilon_{n-1} \\ \rho_i \cdot \eta_n & \text{gdy} \varepsilon_n \leqslant k_w \cdot \varepsilon_{n-1} \end{cases} \tag{5.1}$$

gdzie η_n jest wartością współczynnika uczenia w n-tej epoce, ε_n jest wartością błędu średniokwadratowego w n-tej epoce, ρ_i jest współczynnikiem zwiększenia (ang. *increase*, zwykle przyjmowanym jako 1.05), a ρ_d współczynnikiem zmniejszenia (ang.*decrease*, zwykle 0.7). Dopuszcza się chwilowy wzrost wartości funkcji błędu i przyjmuje się wartość współczynnika k_w za 1.05.

Powyższe wartości współczynników zostały dobrane heurystycznie przez autorów biblioteki MATLAB Neural Network Toolbox i przyjęły się w praktycznym stosowaniu. Analizując ich wartości, można powiedzieć, że nauczyciel dopuszcza chwilowy spadek jakości uczenia oraz że dość ostrożnie manipuluje tempem uczenia – jest bardziej skłonny do spowolnienia procesu uczenia w przypadku niepowodzenia, niż do jego przyspieszenia w razie sukcesu.

Pozostawiamy Czytelnikowi implementację przedstawionego algorytmu adaptacji współczynnika uczenia, dodanie do funkcji `ucz2` wykresu wartości tego współczynnika w kolejnych epokach, a także zbadanie jego wpływu na szybkość uczenia sieci, również dla innych wartości ρ_d i ρ_i niż te zaproponowane wyżej.

5.7. Wydajniejsze uczenie sieci - momentum

Drugim uzupełnieniem algorytmu najszybszego spadku gradientu, mającym na celu jego przyspieszenie, będzie wprowadzenie składnika momentum do poprawek wag,

Zadanie 25. *Zmodyfikuj funkcję* `ucz2` *tak, aby jako algorytmu adaptacji wag można było użyć:*

- *metody najszybszego spadku gradientu*
- *metody najszybszego spadku gradientu z momentum*

Funkcja powinna rysować przynajmniej wykres błędu średniokwadratowego MSE w kolejnych epokach. Zbadaj, jak użycie składnika momentum wpływa na szybkość uczenia sieci.

Proponujemy zastosować następujący nagłówek funkcji:

```
function [ W1po , W2po ] = ucz2 ( W1przed , W2przed , Pucz , Tucz , ...
                                                      Pspr , Tspr , ...
                                  maxLiczbaEpok    , ...
                                  liczbaPokazow    , ...
                                  czestoscWykresow , ...
                                  docelowyBladMSE  , ...
                                  wspUcz,  wspMom  )
```

W dwóch ostatnich parametrach można podać współczynniki, jakie mają być stosowane podczas uczenia. Drugi z nich nazywany jest współczynnikiem momentum. Metoda momentum wprowadza do wzoru na poprawkę wag dodatkowy składnik – poprawkę z kroku poprzedniego. Ogólnie wzór na zmianę dowolnej wagi sieci w kroku n-tym można zapisać jako

$$\begin{aligned} w_{n+1} = w_n + \quad & \Delta w_n \\ & \Delta w_n \quad = \eta_n p_n + \alpha \left(w_n - w_{n-1} \right) \end{aligned} \tag{5.2}$$

gdzie p_n jest poprawką wynikającą z metody najszybszego spadku gradientu, daną wzorem (3.8). Dzięki składnikowi momentum, we wzorze na poprawkę wagi w danym kroku uwzględniona jest historia zmian tej wagi z poprzednich kroków uczenia. Wprowadza to pewną bezwładność do procesu uczenia, stabilizuje go, ale również pomaga uniknąć lokalnych minimów funkcji błędu.

Współczynnik momentum, oznaczony w powyższym wzorze przez α decyduje o tym, jak dużo historii będzie uwzględnione w bieżącej poprawce wag. Ważne są tu wzajemne proporcje współczynnika uczenia η i współczynnika momentum. Oba powinny przyjmować wartości z przedziału od 0 do 1, jednak ich optymalne wielkości niełatwo jest dobrać – zależą one bowiem od specyfiki problemu. Często przyjmuje się następujące rozwiązania:

- oba współczynniki stałe: $\eta = 0.1$, $\alpha = 0.9$,
- $\alpha_n = 1 - \eta_n$, współczynnik uczenia η stały lub adaptacyjny.

Jak już wielokrotnie mówiliśmy, uczenie sieci to proces znajdowania minimum funkcji błędu w przestrzeni wag. Jak pokazano w [4], składnik momentum potrafi przyspieszyć proces uczenia na płaskich odcinkach tej funkcji, wtedy gdy tradycyjne uczenie postępowałoby bardzo wolno. Przy wspomnianych wyżej stałych wartościach współczynnika uczenia 0.1 oraz momentum 0.9, wzrost efektywnego tempa uczenia mógłby być nawet 10-krotny.

Z kolei w sytuacji skrajnie odmiennej, w pobliżu minimum lokalnego funkcji celu, składnik momentum jest niezwiązany z bieżącą wartością gradientu funkcji celu, lecz ze wszystkimi poprzednimi. Dzięki temu jest on w stanie tak wpłynąć na bieżący kierunek zmian wag, aby kosztem chwilowego wzrostu funkcji opuścić strefę przyciągania tego minimum.

Zdominowanie procesu uczenia przez składnik momentum mogłoby prowadzić do jego niestabilności. Dlatego też w każdym kroku należy kontrolować wartość funkcji celu i dopuszczać jedynie niewielki chwilowy jej wzrost, rzędu kilku procent. Jeśli wzrost błędu miałby przekroczyć tę wartość, należy we wzorze (5.3) przyjąć $(w_n - w_{n-1}) = 0$, co sprowadza poprawki wag w bieżącym kroku do metody najszybszego spadku.

Odpowiednie uzupełnienie funkcji `ucz2` nie powinno stanowić dla Czytelnika większego problemu. W każdym kroku uczenia sieci należy jedynie pamiętać poprzednie poprawki wag (autor użył do tego celu zmiennych `dWpoprz`), a wzór aktualizujący wagi uzupełnić o składnik momentum. W każdym kroku uczenia po aktualizacji wag należy sprawdzić sieć na ciągu sprawdzającym. Jeśli wzrost błędu przekroczy dozwoloną wartość, należy odjąć składnik momentum od wag, w przeciwnym razie zostawić nowe wagi.

5.8. Wydajniejsze uczenie sieci - inne metody

Przedstawione wyżej propozycje stanowią łatwe do implementacji modyfikacje podstawowego algorytmu uczenia sieci wielowarstwowych – metody najszybszego spadku gradientu. Obecny stopień rozwoju teorii sieci neuronowych daje nam szeroki wybór wśród metod gradientowych: algorytmu gradientów sprzężonych, zmiennej metryki, czy Levenberga-Marquardta; metod optymalizacji globalnej, a także heurystycznych. Opis większości z nich można znaleźć w literaturze (np. [4]). Ich zaprogramowanie jest już znacznie bardziej skomplikowanym zadaniem niż w odniesieniu do metod przedstawionych w niniejszym skrypcie. Bardzo dużą wartość poznawczą miałoby zaimplementowanie odpowiednich wykresów trójwymiarowych, umożliwiających obserwacje i porównanie przebiegu procesu minimalizacji funkcji celu w przestrzeni wag przy użyciu wyżej wymienionych metod.

Z powodu stopnia skomplikowania tych algorytmów, o wiele lepszym rozwiązaniem od samodzielnej implementacji wydaje się być umiejętne skorzystanie z funkcji pakietu MATLAB Neural Network Toolbox, zawierającego wszystkie wyżej wymienione, a także wiele innych algorytmów uczenia, czy rodzajów sieci. Jest on wyposażony nawet w kompleksowe rozwiązania dopasowane do konkretnych problemów: klasyfikacji, aproksymacji, predykcji, wykrywania skupisk danych. Opis tego pakietu wykracza jednak poza zakres niniejszego skryptu.

Celem tej książki miało być zainteresowanie Czytelnika własnościami i możliwościami sieci neuronowych, „bezbolesne" wprowadzenie do ich teorii, przedstawienie podstawowych zjawisk zachodzących podczas uczenia oraz zasad, do jakich należy się stosować projektując, ucząc i używając sieci. Zaprezentowane tu proste rozwiązania miały zachęcić Czytelnika do dalszych prac programistycznych i badawczych. Metody przedstawione w niniejszej pracy to zaledwie wstęp do rozległej teorii uczenia sieci neuronowych. Pominęliśmy choćby zagadnienie szczegółowej obserwacji procesów minimalizacji funkcji celu w sieciach dwuwarstwowych.

Po serii prostych eksperymentów przedstawionych w tej książce, Czytelnik może zająć się bardziej zaawansowanymi tematami, takimi jak bardziej skomplikowane i wydajne algorytmy uczenia, użycie nowych zestawów danych, przede

wszystkim „z życia wziętych", badanie różnych architektur i rodzajów sieci. Mam nadzieję, że własne obserwacje procesów uczenia przyniosą Czytelnikowi nie tylko wiele satysfakcji poznawczej, ale również dadzą możliwość znalezienia nowych praktycznych zastosowań dla sieci neuronowych.

Rozdział 6

Rozwiązania zadań

Zadanie 1. Inicjalizacja wag sieci jednowarstwowej

```
function [ W ] = init1 ( S , K )
% funkcja tworzy sieć jednowarstwową
% i wypełnia jej macierz wag wartościami losowymi
% z zakresu od -0.1 do 0.1
% parametry: S - liczba wejść do sieci
%            K - liczba neuronów w warstwie
% wynik:     W - macierz wag sieci

W = rand ( S , K ) * 0.2 - 0.1 ;
```

Zadanie 2. Symulacja działania sieci jednowarstwowej

```
function [ Y ] = dzialaj1 ( W , X )
% funkcja symuluje działanie sieci jednowarstwowej
% parametry: W - macierz wag sieci
%            X - wektor wejść do sieci
%                sygnał podany na wejście
% wynik:     Y - wektor wyjść sieci
%                sygnał na wyjściu sieci

beta = 5 ;
U = W' * X ;
Y = 1 ./ ( 1 + exp ( -beta * U ) ) ;
```

Zadanie 3. Uczenie sieci jednowarstwowej – rozwiązanie intuicyjne

```
function [ Wpo ] = ucz1 ( Wprzed , P , T , n )
% funkcja uczy sieć jednowarstwową
% na podanym ciągu uczącym (P,T)
% przez zadaną liczbę epok (n)
% parametry: Wprzed - macierz wag sieci przed uczeniem
%            P      - ciąg uczący - przykłady - wejścia
%            T      - ciąg uczący - żądane wyjścia
%                                   dla poszczególnych przykładów
%            n      - liczba epok
% wynik:     Wpo    - macierz wag sieci po uczeniu

liczbaPrzykladow = size ( P , 2 ) ;
W = Wprzed ;
wspUcz = 0.1 ;

for i = 1 : n ,
   % losuj numer przykładu
   nrPrzykladu = ceil ( rand(1) * liczbaPrzykladow ) ;
   % lub: los = randperm ( liczbaPrzykladow ) ;
   %      nrPrzykladu = los ( 1 ) ;
   % podaj przykład na wejścia i oblicz wyjścia
   X = P ( : , nrPrzykladu ) ;
   Y = dzialaj1 ( W , X ) ;
   % oblicz błędy na wyjściach
   D = T ( : , nrPrzykladu ) - Y ;
   % oblicz poprawki wag
   dW = wspUcz * X * D' ;
   % dodaj  poprawki do wag
   W = W + dW ;
end  % i to wszystko n razy

Wpo = W ;
```

Zadanie 3. Uczenie sieci jednowarstwowej – rozwiązanie dokładne

```
function [ Wpo ] = ucz1 ( Wprzed , P , T , n )
% funkcja uczy sieć jednowarstwową
% na podanym ciągu uczącym (P,T)
% przez zadaną liczbę epok (n)
% parametry: Wprzed - macierz wag sieci przed uczeniem
%            P      - ciąg uczący - przykłady - wejścia
%            T      - ciąg uczący - żądane wyjścia
%                                   dla poszczególnych przykładów
%            n      - liczba epok
% wynik:     Wpo    - macierz wag sieci po uczeniu

liczbaPrzykladow = size ( P , 2 ) ;
W = Wprzed ;
wspUcz = 0.1 ;
beta = 5 ;

for i = 1 : n ,
   % losuj numer przykładu
   nrPrzykladu = ceil ( rand(1) * liczbaPrzykladow ) ;
   % lub: los = randperm ( liczbaPrzykladow ) ;
   %      nrPrzykladu = los ( 1 ) ;
   % podaj przykład na wejścia i oblicz wyjścia
   X = P ( : , nrPrzykladu ) ;
   Y = dzialaj1 ( W , X ) ;
   % oblicz błędy na wyjściach
   D = T ( : , nrPrzykladu ) - Y ;
   E = D .* beta*Y.*(1-Y) ;  % razy pochodna funkcji aktywacji
   % oblicz poprawki wag
   dW = wspUcz * X * E' ;
   % dodaj  poprawki do wag
   W = W + dW ;
end  % i to wszystko n razy

Wpo = W ;
```

Zadanie 4. Inicjalizacja sieci dwuwarstwowej

```
function [ W1 , W2 ] = init2 ( S , K1 , K2 )
% funkcja tworzy sieć dwuwarstwową
% i wypełnia jej macierze wag wartościami losowymi
% z zakresu od -0.1 do 0.1
% parametry: S  - liczba wejść do sieci (wejść warstwy 1)
%            K1 - liczba neuronów w warstwie 1
%            K2 - liczba neuronów w warstwie 2  (wyjść sieci)
% wynik:     W1 - macierz wag warstwy 1 sieci
%            W2 - macierz wag warstwy 2 sieci

W1 = rand ( S +1 , K1 ) * 0.2 - 0.1 ;
W2 = rand ( K1+1 , K2 ) * 0.2 - 0.1 ;
```

Zadanie 5. Symulacja działania sieci dwuwarstwowej

```
function [ Y1 , Y2 ] = dzialaj2 ( W1 , W2 , X )
% funkcja symuluje działanie sieci dwuwarstwowej
% parametry: W1 - macierz wag pierwszej warstwy sieci
%            W2 - macierz wag  drugiej  warstwy sieci
%            X  - wektor wejść do sieci
%                 sygnał podany na wejście ( sieci / warstwy 1 )
% wynik:     Y1 - wektor wyjść warstwy 1   ( przyda się podczas uczenia )
%            Y2 - wektor wyjść warstwy 2   / sieci
%                 sygnał na wyjściu sieci

beta = 5 ;
X1 = [ -1 ; X ] ;
U1 = W1' * X1 ;
Y1 = 1 ./ ( 1 + exp ( -beta * U1 ) ) ;
X2 = [ -1 ; Y1 ] ;
U2 = W2' * X2 ;
Y2 = 1 ./ ( 1 + exp ( -beta * U2 ) ) ;
```

Zadanie 6. Uczenie sieci dwuwarstwowej – rozwiązanie intuicyjne

```
function [ W1po , W2po ] = ucz2 ( W1przed , W2przed , P , T , n )
% funkcja uczy sieć dwuwarstwową na podanym ciągu uczącym (P,T)
%                                 przez zadaną liczbę epok (n)
% parametry: W1przed - macierz wag warstwy 1 przed uczeniem
%            P       - ciąg uczący - przykłady - wejścia
%            T       - ciąg uczący - żądane wyjścia
%            n       - liczba epok
% wynik:     W1po    - macierz wag warstwy 1 po uczeniu
%            W2po    - macierz wag warstwy 2 po uczeniu

liczbaPrzykladow = size(P ,2) ;
wierW2           = size(W2,1) ;      % liczba wierszy macierzy W2
W1 = W1przed ;
W2 = W2przed ;
wspUcz = 0.1 ;

for i = 1 : n ,

    nrPrzykladu = ceil(rand(1)*liczbaPrzykladow);

  % podaj przykład na wejścia...
    X  = P(:,nrPrzykladu) ;                  % wejścia sieci
  % ...i oblicz wyjścia
  [ Y1 , Y2 ] = dzialaj2 ( W1 , W2 , X ) ;
    X1 = [ -1 ; X  ] ;                       % wejścia warstwy 1
    X2 = [ -1 ; Y1 ] ;                       % wejścia warstwy 2
    D2 = T(:,nrPrzykladu) - Y2 ;     % oblicz błędy dla warstwy 2
    D1 = W2(2:wierW2,:)   * D2 ;     % oblicz błędy dla warstwy 1
   dW1 = wspUcz * X1 * D1' ;      % oblicz poprawki wag warstwy 1
   dW2 = wspUcz * X2 * D2' ;      % oblicz poprawki wag warstwy 2
    W1 = W1 + dW1 ;            % dodaj  poprawki do wag warstwy 1
    W2 = W2 + dW2 ;            % dodaj  poprawki do wag warstwy 2

end  % i to wszystko n razy

W1po = W1 ;
W2po = W2 ;
```

Zadanie 6. Uczenie sieci dwuwarstwowej – rozwiązanie dokładne

```
function [ W1po , W2po ] = ucz2 ( W1przed , W2przed , P , T , n )
% funkcja uczy sieć dwuwarstwową na podanym ciągu uczącym (P,T)
%                                 przez zadaną liczbę epok (n)
% ...

liczbaPrzykladow = size(P ,2) ;
wierW2           = size(W2,1) ;      % liczba wierszy macierzy W2
W1 = W1przed ;
W2 = W2przed ;
wspUcz = 0.1 ;
  beta = 5   ;

for i = 1 : n ,

  los = randperm(liczbaPrzykladow) ;  nrPrzykladu = los(1) ;

  % podaj przykład na wejścia...
    X  = P(:,nrPrzykladu) ;                   % wejścia sieci
  % ...i oblicz wyjścia
  [ Y1 , Y2 ] = dzialaj2 ( W1 , W2 , X ) ;
    X1 = [ -1 ; X  ] ;                        % wejścia warstwy 1
    X2 = [ -1 ; Y1 ] ;                        % wejścia warstwy 2
    D2 = T(:,nrPrzykladu)-Y2 ;       % oblicz błędy dla warstwy 2
    E2 = beta*D2.*Y2.*(1-Y2) ;       % błąd,,wewnętrzny'' warstwy 2
    D1 = W2(2:wierW2,:) * E2 ;       % oblicz błędy dla warstwy 1
    E1 = beta*D1.*Y1.*(1-Y1) ;       % błąd,,wewnętrzny'' warstwy 1
   dW1 = wspUcz * X1 * E1' ;      % oblicz poprawki wag warstwy 1
   dW2 = wspUcz * X2 * E2' ;      % oblicz poprawki wag warstwy 2
    W1 = W1 + dW1 ;            % dodaj  poprawki do wag warstwy 1
    W2 = W2 + dW2 ;            % dodaj  poprawki do wag warstwy 2

end  % i to wszystko n razy

W1po = W1 ;
W2po = W2 ;
```

Zadanie 9. Inicjalizacja sieci jednowarstwowej (z wejściem progowym)

```
function [ W ] = init1b ( S , K )
% ------------------------------------------------------
% funkcja tworzy jednowarstwową sieć neuronową
% i wypełnia jej macierz wag wartościami losowymi
% z zakresu od -0.1 do 0.1
% uwaga: funkcja już nie "zapomina" o biasie
%                tak jak "zapominała" init1a
% ...
% ------------------------------------------------------
% parametry: S - liczba wejść do sieci
%            K - liczba neuronów w warstwie
% wynik:     W - macierz wag sieci
% ------------------------------------------------------

    W = rand ( S + 1 , K ) * 0.2 - 0.1 ;
```

Zadanie 9. Symulacja działania sieci jednowarstwowej (z wejściem progowym)

```
function [ Y ] = dzialaj1b ( W , X )
% ------------------------------------------------------
% funkcja symuluje działanie sieci jednowarstwowej
%
% uwaga: funkcja nie "zapomina" o biasie
%        tak jak "zapominała" dzialaj1a
% ...
% ------------------------------------------------------
% parametry: W - macierz wag sieci
%            X - wektor wejść do sieci
%                sygnał podany na wejście
% wynik:     Y - wektor wyjść sieci
%                sygnał na wyjściu sieci
% ------------------------------------------------------
%
% współczynnik nachylenia funkcji aktywacji:
beta = 5 ;

X = [ -1 ; X ] ;
U =   W' * X ;
Y = 1 ./ ( 1 + exp ( -beta * U ) ) ;
```

Zadanie 9. Uczenie sieci jednowarstwowej (z wejściem progowym)

```
function [ Wpo ] = ucz1b ( Wprzed , P , T , n )
% ------------------------------------------------------------------------
% funkcja uczy sieć jednowarstwową przez zadaną liczbę kroków
%                                    na zadanym ciągu uczącym
% * w jednym kroku uczenia funkcja pokazuje sieci jeden przykład
%   po czym oblicza poprawki i uaktualnia wagi
% * funkcja nie zna pojęcia epoki (krok = pokaz - tylko uczenie on-line),
%   ani nie sprawdza błędu średniokwadratowego aby wcześniej zakończyć uczenie
% uwaga: funkcja już nie "zapomina" o biasie (używa   funkcji dzialaj1b)
%                     jak "zapominała" ucz1a  (używała funkcji dzialaj1a)
% ...
% ------------------------------------------------------------------------
% parametry: Wprzed - macierz wag sieci przed uczeniem
%            P      - ciąg uczący - przykłady - wejścia
%            T      - ciąg uczący - przykłady - żądane wyjścia
%            n      - liczba krokóww uczenia
% wynik: Wpo - macierz wag sieci po uczeniu
% ------------------------------------------------------------------------
liczbaPrzykladow = size(P,2) ;  % kolejne przykłady w parze macierzy P-T
                                % są ułożone w kolumnach, więc:
                                % * liczba  kolumn P = liczba przykładów
                                % * liczba  kolumn T = liczba przykładów
                                % * liczba wierszy P = liczba wejść do sieci
                                % * liczba wierszy T = liczba wyjść    sieci
% tu można by sprawdzić, czy liczba wejść i wyjść sieci
%                         zgadza się z liczbą wierszy i kolumn w ciągu uczącym
% ------------------------------------------------------------------------
% UCZENIE SIECI
% ------------------------------------------------------------------------
W = Wprzed ;
wspUcz = 0.1 ;
% ------------------------------------------------------------

for krok = 1 : n ,

   % losuj numer przykładu
   los = randperm(liczbaPrzykladow) ;
   nrPrzykladu = los(1) ;

   % podaj przykład na wejścia i oblicz wyjścia
   X = P ( : , nrPrzykladu ) ;
   Y = dzialaj1b ( W , X ) ;

   % oblicz błędy na wyjściach
   D = T ( : , nrPrzykladu ) - Y ;

   X = [ -1 ; X ] ;
```

```
      % macierz wejść, którą tak naprawdę widzi warstwa sieci
      % (uzupełniona o bias)

   % oblicz poprawki wag
   dW = wspUcz * X * D' ;

   % dodaj  poprawki do wag
   W = W + dW ;

end  % i to wszystko n razy
% ------------------------------------------------------------
Wpo = W ;
```

Zadanie 9. Rozpoznawanie zwierząt (z wejściem progowym)

```
clc
clear all
close all
% ------------------------------------------------------------
disp ( '************************************************' ) ;
disp ( 'Sieć jednowarstwowa uczy się rozpoznawać zwierzęta' ) ;
disp ( '- wersja podstawowa z biasem                    ' ) ;
disp ( '************************************************' ) ;
disp ( 'wejścia sieci:  1 - ile ma nóg                  ' ) ;
disp ( '                2 - czy żyje w wodzie           ' ) ;
disp ( '                3 - czy umie latać              ' ) ;
disp ( '                4 - czy ma pióra                ' ) ;
disp ( '                5 - czy jest jajorodne          ' ) ;
disp ( ' wyjścia sieci: 1 - ssak                        ' ) ;
disp ( '                2 - ptak                        ' ) ;
disp ( '                3 - ryba                        ' ) ;
disp ( '************************************************' ) ;
% ------------------------------------------------------------
disp ( ' ' ) ;
disp ( '_______________________________________'    ) ;
disp ( 'Ciąg uczący:'       ) ;
disp ( ' ' ) ;
disp ( 'P - wejścia sieci'  ) ;
disp ( 'T - żądane wyjścia' ) ;
disp ( 'kolejne przykłady w kolumnach' ) ;
disp ( ' ' ) ;
                              % wejścia sieci:
P = [  4       2     -1     ;  % we 1 - ile ma nóg
       0.01   -1      3.5   ;  % we 2 - czy żyje w wodzie
       0.01    2      0.01  ;  % we 3 - czy umie latać
      -1       2.5   -2     ;  % we 4 - czy ma pióra
      -1.5     2      1.5   ]  % we 5 - czy jest jajorodny
                              % żądane wyjścia sieci:
```

```
T = [   1       0      0    ;  % ssak
        0       1      0    ;  % ptak
        0       0      1    ]  % ryba
% ------------------------------------------------------------
disp ( ' ' ) ;
disp ( '_________________________________________' ) ;
disp ( 'Sieć przed uczeniem: ' ) ;
  Wprzed = init1b    ( 5 , 3 )
% ------------------------------------------------------------
  Ya     = dzialaj1b ( Wprzed , P(:,1) ) ;
  Yb     = dzialaj1b ( Wprzed , P(:,2) ) ;
  Yc     = dzialaj1b ( Wprzed , P(:,3) ) ;
  Yprzed = [ Ya , Yb , Yc ] ;
disp ( ' ' ) ;
disp ( '_________________________________________' ) ;
disp ( 'Tak działa: ' ) ;
powinnoByc = T
      jest = Yprzed
% ------------------------------------------------------------
disp ( ' ' ) ;
disp ( '_________________________________________' ) ;
disp ( 'Naciśnij ENTER, aby rozpocząć uczenie...' ) ;
pause
disp ( ' ' ) ;
disp ( 'Uczę sieć...' ) ;
  Wpo    = ucz1b     ( Wprzed , P , T , 100 ) ;
disp ( ' ' ) ;
disp ( '_________________________________________' ) ;
disp ( 'Sieć po uczeniu: ' ) ;
  Wpo
% ------------------------------------------------------------
  Ya  = dzialaj1b ( Wpo , P(:,1) ) ;
  Yb  = dzialaj1b ( Wpo , P(:,2) ) ;
  Yc  = dzialaj1b ( Wpo , P(:,3) ) ;
  Ypo = [ Ya , Yb , Yc ] ;
disp ( ' ' ) ;
disp ( '_________________________________________' ) ;
disp ( 'Tak działa: ' ) ;
powinnoByc = T
      jest = Ypo
% ------------------------------------------------------------
disp ( 'Naciśnij ENTER...' ) ;
pause
```

Zadanie 12. Sprawdzanie działania sieci na zadanym ciągu danych

```
function [ Y , D , MSE , CE ] = sprawdz1 ( W , P , T )
% ------------------------------------------------------------------
%        [ Y , D , MSE , CE ] = sprawdz1 ( W , P , T )
% ------------------------------------------------------------------
% funkcja testuje sieć jednowarstwową na zadanym ciągu
%               (uczącym / sprawdzającym / testującym)
% uwaga: funkcja już nie "zapomina" o biasie (używa funkcji dzialaj1b)
% ...
% ------------------------------------------------------------------
% parametry: W - macierz wag sieci
%            P - ciąg - przykłady - wejścia
%            T - ciąg - przykłady - żądane wyjścia
% wynik:   Y - macierz odpowiedzi sieci na   kolejne  przykłady  z P
%          D - macierz   błędów   sieci dla kolejnych przykładów z P
%        MSE -  mean  square  error - błąd średniokwadratowy sieci
%         CE - classification error - procent błędnych klasyfikacji
%              - jeśli sieć nie rozwiązuje zadania klasyfikacji,
%                to można to wyjście pominąć
%              - funkcja zakłada unipolarną funkcję aktywacji,
%                czyli żądane wyjścia to 0 - 1
% ------------------------------------------------------------------
liczbaPrzykladow = size ( P , 2 ) ;  % kolejne przykłady w parze macierzy P-T
                                     % są ułożone w kolumnach, więc:
                                     % * liczba  kolumn P = liczba przykładów
                                     % * liczba  kolumn T = liczba przykładów
                                     % * liczba wierszy P = liczba wejść do sieci
liczbaWyjscSieci = size ( T , 1 ) ;  % * liczba wierszy T = liczba wyjść    sieci
% ------------------------------------------------------------
  Y = zeros ( liczbaWyjscSieci , liczbaPrzykladow ) ;
  D = zeros ( liczbaWyjscSieci , liczbaPrzykladow ) ;
MSE = 0 ;  CE = 0 ;
% ------------------------------------------------------------
for i = 1 : liczbaPrzykladow ,
    Xi = P(:,i) ;                  % wybranie kolejnego przykładu z ciągu
    Yi = dzialaj1b ( W , Xi ) ;    % obliczenie wyjscia sieci
    Di = T(:,i) - Yi  ;            % obliczenie błędu na wyjściach
    Y(:,i) = Yi ;                  % wpisanie bieżącego wyjścia sieci do Y
    D(:,i) = Di ;                  % wpisanie bieżącego  błędu  sieci do D
    MSE = MSE + Di'*Di ;           % kumulacja błędu średniokwadratowego
    if  ( any ( abs(Di)>0.5 )  ) ,  % kumulacja liczby błędnych klasyfikacji
        CE = CE + 1 ;
    end
end
% ------------------------------------------------------------
% normalizacja błędów
MSE = 0.5 * MSE / liczbaPrzykladow / liczbaWyjscSieci ;
 CE =        CE / liczbaPrzykladow * 100
```

Zadanie 14. Uczenie sieci jednowarstwowej - kilka pokazów w kroku uczenia

```
function [ Wpo ] = ucz1d ( Wprzed , P , T , maxLiczbaKrokow  , ...
                                             liczbaPokazow    , ...
                                             czestoscWykresow , ...
                                             docelowyBladMSE  )
% ------------------------------------------------------------------
% (...)
% ------------------------------------------------------------------
% funkcja uczy sieć jednowarstwową przez zadaną liczbę kroków
%                       lub do osiągnięcia zadanego błędu MSE
%                                 na zadanym ciągu uczącym
% i rysuje wykres błędu średniokwadratowego
%          w poszczególnych krokach uczenia
%          (błąd na przykładach pokazywanych w danym kroku
%           oraz na całym ciągu uczącym)
%        i wykres liczby (procentu) błędnych klasyfikacji
% (...)
% ------------------------------------------------------------------
liczbaPrzykladow = size ( P , 2 ) ;
liczbaWejscSieci = size ( P , 1 ) ;
liczbaWyjscSieci = size ( T , 1 ) ;
liczbaWejscWarstwy = size ( Wprzed , 1 ) ;
liczbaNeuronow     = size ( Wprzed , 2 ) ;
liczbaPokazow = min ( liczbaPokazow , liczbaPrzykladow ) ;
% ------------------------------------------------------------------
              % przygotowanie macierzy pod wykres:
              % * przebieg błędu średniokwadratowego podczas uczenia
              %   - tylko na przykładach pokazywanych w danym kroku
bladMSEkrok = zeros ( 1 , maxLiczbaKrokow ) ;
              %   - na całym ciągu uczącym
bladMSEciag = zeros ( 1 , maxLiczbaKrokow ) ;
              % * przebieg liczby błędnych klasyfikacji
bladCEciag  = zeros ( 1 , maxLiczbaKrokow ) ;
              % utworzenie okienka z wykresami
wykresBledu = ...
              % * przebieg błędu MSE          - górny wykres (...)
              % * przebieg błędu klasyfikacji - dolny wykres (...)
% ------------------------------------------------------------------
% UCZENIE SIECI
% ------------------------------------------------------------------
W = Wprzed ;
wspUcz = 0.1 ;
% ------------------------------------------------------------------
for krok = 1 : maxLiczbaKrokow ,
   % na początku każdego kroku wyzerowanie macierzy poprawek
   dWkrok = W * 0 ;
   % 1) losuj numery przykładów dla danego kroku
   %     aby przykłady nie powtarzały się
```

```
   los = randperm ( liczbaPrzykladow ) ;
   nryPrzykladow = los ( 1 : liczbaPokazow ) ;
   for pokaz = 1 : liczbaPokazow ,
       nrPrzykladu = nryPrzykladow(pokaz) ;
       % 2) podaj przykład na wejścia...
       X = P ( : , nrPrzykladu ) ;
       % 3) ...i oblicz wyjścia
       Y = dzialaj1b ( W , X ) ;
       % 4) oblicz błędy na wyjściach
       D = T ( : , nrPrzykladu ) - Y ;
       % oblicz błąd średniokwadratowy - na bieżącym przykładzie
       bladMSEpokaz  = 0.5 * D' * D / liczbaWyjscSieci ;
         % i skumuluj
       bladMSEkrok(krok) = bladMSEkrok(krok) + bladMSEpokaz ;
       X = [ -1 ; X ] ;  %    wektor wejść z biasem
       % 5) oblicz poprawki wag dla danego pokazu
       dWpokaz = wspUcz * X * D'  ;
       %   - i skumuluj dla całego kroku
       dWkrok  = dWkrok + dWpokaz ;
   end  % for pokaz
   % znormalizuj skumulowany błąd
   bladMSEkrok(krok) = bladMSEkrok(krok) / liczbaPokazow ;
   % 6) dodaj poprawki do wag
   %    - skumulowane i znormalizowane
   W = W + dWkrok / liczbaPokazow ;
   % oblicz błąd średniokwadratowy na całym ciągu uczącym
   %   oraz procent błędnych klasyfikacji
   [ Yciag , Dciag , bladMSEciag(krok) , ...
                     bladCEciag(krok) ] = sprawdz1 ( W , P , T ) ;
   % ------------------------------------------------------------
   % co czestoscWykresow kroków...
   if  mod(krok,czestoscWykresow) == 0 ,
       % ...komunikat o postępach
       % ...wykresy błędów
   end  % wykres ------------------------------------------------
   % jeśli osiągnięto zadany błąd,
   % to kończymy uczenie nie zważając na max liczbę kroków
   if  ( bladMSEciag(krok) < docelowyBladMSE ) ,
       break
   end
end  % krok
% ustawienie granic wykresów
% (...)
% ------------------------------------------------------------------
Wpo = W ;
```

Bibliografia

[1] Hawkins J., Blakeslee S., *Istota inteligencji*, Wydawnictwo Helion, 2006
[2] Korbicz J., Obuchowicz A., Uciński D., *Sztuczne sieci neuronowe. Podstawy i zastosowania*, Akademicka Oficyna Wydawnicza PLJ, Warszawa 1994
[3] Matthews G., *Neurobiologia. Od cząsteczek i komórek do układów*, Wydawnictwo Lekarskie PZWL, Warszawa 2000
[4] Osowski S., *Sieci neuronowe do przetwarzania informacji*, Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 2006
[5] Tadeusiewicz R., Gąciarz T., Borowik B., Leper B., *Odkrywanie właściwości sieci neuronowych przy użyciu programów w języku C#*, Polska Akademia Umiejętności, Kraków 2007
[6] Żurada J., Barski M., Jędruch W., *Sztuczne sieci neuronowe: podstawy teorii i zastosowania*, Wydawnictwo Naukowe PWN, Warszawa 1996
[7] UCI Machine Learning Repository. `http://archive.ics.uci.edu/ml/`
[8] KDnugets - Datasets. `http://www.kdnuggets.com/datasets/`