Wydział Inżynierii Mechanicznej i Robotyki

Katedra Robotyki i Mechatroniki

Podstawy Sztucznej Inteligencji I Uczenia Głębokiego

Inżynieria Mechatroniczna

Instrukcja 4:

Sztuczne Sieci Neuronowe

Nauczysz się: jak korzystać z toolboxa MATLAB-a do projektowania i wykorzystywania sieci neuronowych oraz jak oceniać ich skuteczność na podstawie danych symulowanych. Nauczymy się rozpoznawać i radzić sobie z przeuczeniem, a także zaprojektujemy własne eksperymenty, które pomogą nam lepiej zrozumieć działanie sieci neuronowych.

Dodatkowe materiały:

- Wykłady 1 - 4

Ta instrukcja została przetłumaczona przez ChatGPT na podstawie angielskiego oryginału – materiałów z przedmiotu *Basic of Artificial Intelligence and Deep Learning*

Koordynator przedmiotu:
Krzysztof Holak, holak@agh.edu.pl

Autor instrukcji:
Ziemowit Dworakowski, zdw@agh.edu.pl

Porównanie możliwości sieci neuronowych z klasyfikatorami i regresorami używanymi do tej pory

Projektowaliśmy wielomianowe regresory oraz klasyfikatory oparte na ręcznie implementowanych sieciach neuronowych (i trenowaliśmy je przy użyciu naszych własnych implementacji algorytmów optymalizacyjnych). Choć pozwoliło to lepiej zrozumieć działanie tych metod „od środka", to jednak niekoniecznie jest to podejście najbardziej efektywne w rozwiązywaniu praktycznych problemów.

Porównajmy teraz implementacje opracowane w laboratoriach 2 i 3 z tymi, które są dostępne w toolboxach MATLAB-a. Zacznijmy od problemu regresji. Podobnie jak wcześniej, do konfiguracji parametrów modelu użyjemy *RegressionTrainingData*, a do końcowej oceny – *RegressionTestingData*. Jako model regresyjny użyjemy wielowarstwowej sieci neuronowej (MLP).

Aby wytrenować sieć, musimy podzielić dane na próbki treningowe oraz odpowiadające im wartości docelowe – które do tej pory były przechowywane jedynie w trzeciej kolumnie macierzy danych. Teraz musimy przekazać je do sieci osobno:

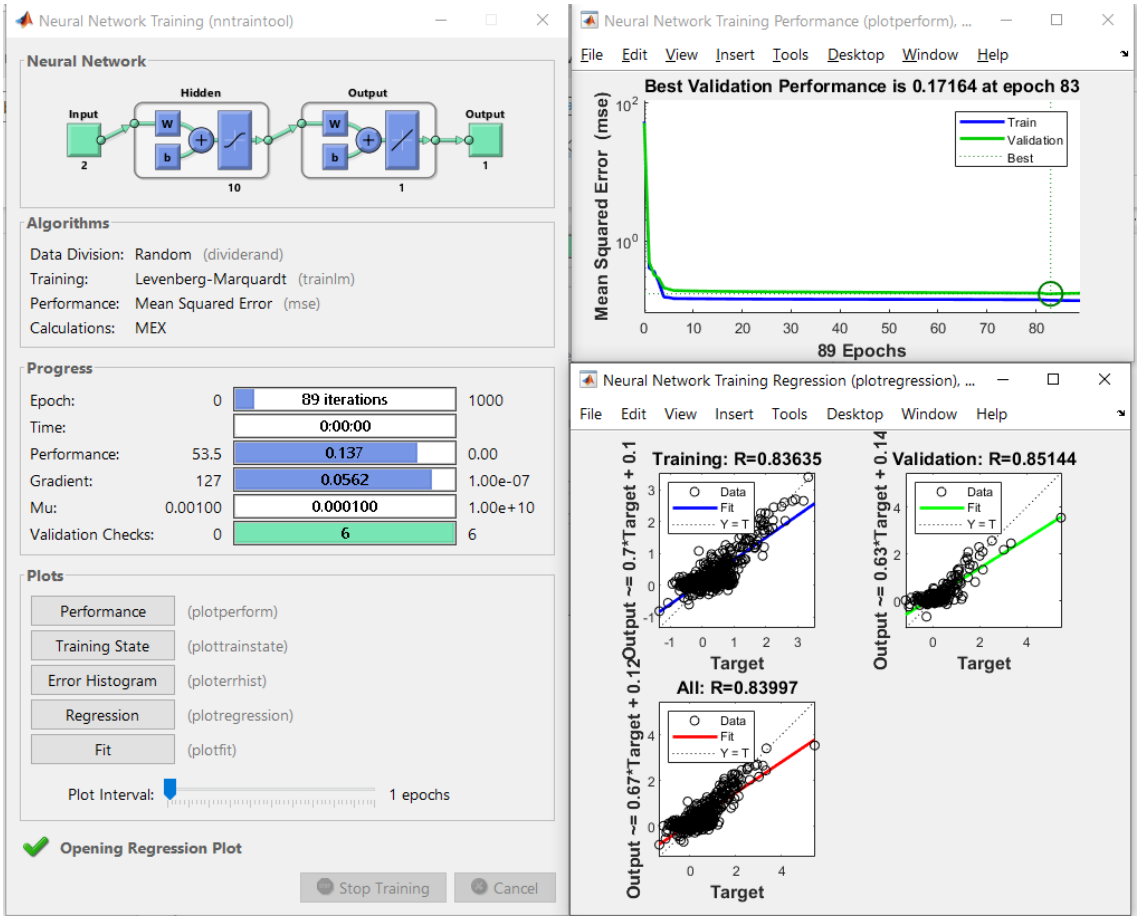
```
load('RegressionData1.mat');  % Make sure the .mat file is in your path

% Inputs (first two columns) and targets (third column)
X = RegressionTrainingData(:, 1:2)';
T = RegressionTrainingData(:, 3)';
```

Teraz musimy zaprojektować naszą sieć, a następnie ją wytrenować, co możemy zrobić za pomocą poniższego kodu. Na początek użyjemy prostej sieci, o jednej warstwie ukrytej zawierającej 10 neuronów:

```
%% Create a feedforward neural network
hiddenLayerSize = 10;
net = fitnet(hiddenLayerSize);  % Regression network

% Set training/validation/test split
net.divideParam.trainRatio = 0.7;
net.divideParam.valRatio   = 0.3;
net.divideParam.testRatio  = 0.0;  % We will not use the internal test set

net.trainFcn = 'trainlm';          % Define training function
net.trainParam.showWindow = true;  % Show training window

[net, tr] = train(net, X, T);      % Train the network
```

Zwróć uwagę, że w kodzie możemy automatycznie podzielić zbiór danych na podzbiory (dane przekazane do sieci zostaną podzielone losowo podczas procesu treningu). Ponieważ zbiór testowy mamy już przygotowany, opcję tę wykorzystujemy tylko do utworzenia podzbiorów treningowego i walidacyjnego.

Przyjrzyjmy się teraz sieci, którą stworzyliśmy. Wynik treningu powinien wyglądać mniej więcej tak, jak przedstawiono na rys. 1 (zwróć uwagę, że wygląd grafiki może się różnić w zależności od wersji MATLAB-a i toolboxa, ale powinna ona zawierać podobne informacje).



Rysunek 1 – Wyniki treningu w graficznym interfejsie MATLAB-a: nntraintool dostarcza informacji o strukturze sieci (2 wejścia, 10 neuronów w warstwie ukrytej), innych metaparametrach oraz przebiegu treningu, wskazując wykorzystane kryterium zatrzymania). Można również otworzyć dodatkowe wykresy, z przebiegiem treningu („Performance") lub rozrzut wyników względem idealnej linii.

Sprawdźmy teraz wyniki, które uzyskaliśmy. Możemy pobrać predykcję z wytrenowanej sieci i porównać ją z tą, którą zwracał model wielomianowy z laboratorium 2.

```
PredictedValues = net(X);  % Here we can use our trained net to predict new data
MSE = mse(T,PredictedValues)
fprintf('MSE on all training+validation data: %.4f\n', MSE);

%% Plot predicted vs actual
figure;
plot(T, PredictedValues, '.')
xlabel('Actual Target')
ylabel('Predicted Output')
title('Neural Network Regression: Actual vs Predicted')
```

Zwróć jednak uwagę, że to jest wartość MSE dla zbioru treningowego. Aby obliczyć ją dla zbioru testowego, musimy wczytać i przepuścić przez sieć część danych przeznaczoną do testowania. Oczywiście nie chcemy uwzględniać tego zbioru w poleceniu *train* – wystarczy po prostu użyć polecenia *net* z powyższego kodu (oznaczonego na żółto) na nowych danych.

Zadanie 4.1: Korzystając ze swojego **indywidualnego*** zbioru danych do regresji, wytrenuj sieć neuronową, używając kodu przedstawionego powyżej. Następnie przetestuj wytrenowaną sieć na zbiorze testowym swojego zestawu danych – i porównaj jej wynik z regresorem wielomianowym z laboratorium 2. Wyniki porównania przedstaw za pomocą wykresu słupkowego.

Teraz wykonajmy ten sam proces dla problemu klasyfikacji, który rozwiązywaliśmy w ramach Laboratorium 3. Podobnie jak wcześniej, musimy przygotować dane do użycia przez sieć neuronową. Możemy to zrobić za pomocą następującego kodu:

```
% Load training and validation data
load('TR_DATA.mat');
load('VA_DATA.mat');

% Extract features and labels from training data
X_train = TR_DATA(2:3, :);    % Features (2×N)
T_train = TR_DATA(1, :);      % Class labels (1×N)

% Extract features and labels from validation data
X_val = VA_DATA(2:3, :);
T_val = VA_DATA(1, :);
```

Projektowanie i konfiguracja sieci przebiega podobnie jak wcześniej – jedyną różnicą jest to, że zamiast *fitnet* używamy *patternnet*, co informuje Matlaba, że nasza sieć ma rozpoznawać klasy w warstwie wyjściowej. Rezygnujemy także z automatycznego podziału danych treningowych i walidacyjnych. Zwróć również uwagę, że zmieniliśmy algorytm treningu w naszym zadaniu rozpoznawania wzorców:

```
% Create a MLP network:
hiddenLayerSize = 10;
net = patternnet(hiddenLayerSize);

% Training metaparameters:
net.divideFcn = 'dividetrain';  % For manual control of validation subset
net.trainFcn = 'trainscg';      % Scaled conjugate gradient training method
net.trainParam.showWindow = true;

% Train the network
[net, tr] = train(net, X_train, T_train);
```

I wreszcie jesteśmy w stanie obliczyć skuteczność działania sieci na dowolnym zbiorze danych. Zacznijmy od zbioru treningowego:

```
% Predict and compute training accuracy
Y_train = net(X_train);              % Network output: probabilities
Y_predtrain = round(Y_train);        % Convert to binary class predictions
trainaccuracy(iteration) = sum(Y_predtrain == T_train) / length(T_train);
fprintf('Training Accuracy: %.2f%%\n', trainaccuracy(iteration) * 100);
```

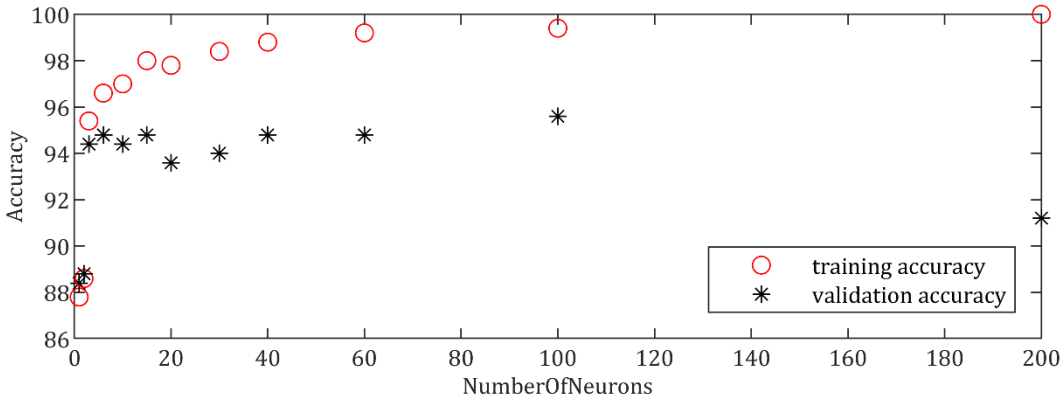
Oczywiście chcemy znać nie tylko dokładność na zbiorze treningowym, ale również wynik walidacji (na podstawie zbioru *VA_DATA*, który wczytaliśmy wcześniej, ale jeszcze go nie wykorzystaliśmy).

Zadanie 4.2: Korzystając ze swojego **indywidualnego** zbioru danych do klasyfikacji z laboratorium 3, wytrenuj sieć neuronową, używając podzbioru TR_DATA. Następnie sprawdź jej skuteczność, korzystając z podzbiorów TR_DATA oraz VA_DATA. Porównaj uzyskane rozwiązanie z rozwiązaniami wieloliniowymi opartymi na algorytmie 1+1 oraz algorytmie gradientowym, opracowanymi w ramach laboratorium 3. Wyniki tego porównania przedstaw za pomocą wykresu słupkowego.

Teraz pojawia się pytanie: Czy rzeczywiście mamy najlepszą możliwą sieć dla tego zadania? W tym miejscu zaczynamy myśleć o metaparametrach – algorytmach treningowych, strukturze sieci, warunkach zakończenia optymalizacji itd. Zacznijmy prosto: co się stanie, jeśli zmienimy liczbę neuronów w naszym zadaniu klasyfikacyjnym?

Zróbmy to systematycznie: zwróć uwagę, że przygotowałem wektor *hlsConfigurations* z predefiniowanymi konfiguracjami, które chcę przetestować (oznaczone na niebiesko). Powód jest prosty: mogą dziać się ciekawe rzeczy przy zmianie liczby neuronów z 2 na 3 (więc nie chcę tego przegapić), ale nie spodziewam się, żeby miało znaczenie, czy mamy 43 czy 44 neurony. Wyniki, które otrzymałem, możesz zobaczyć na Rys 2.

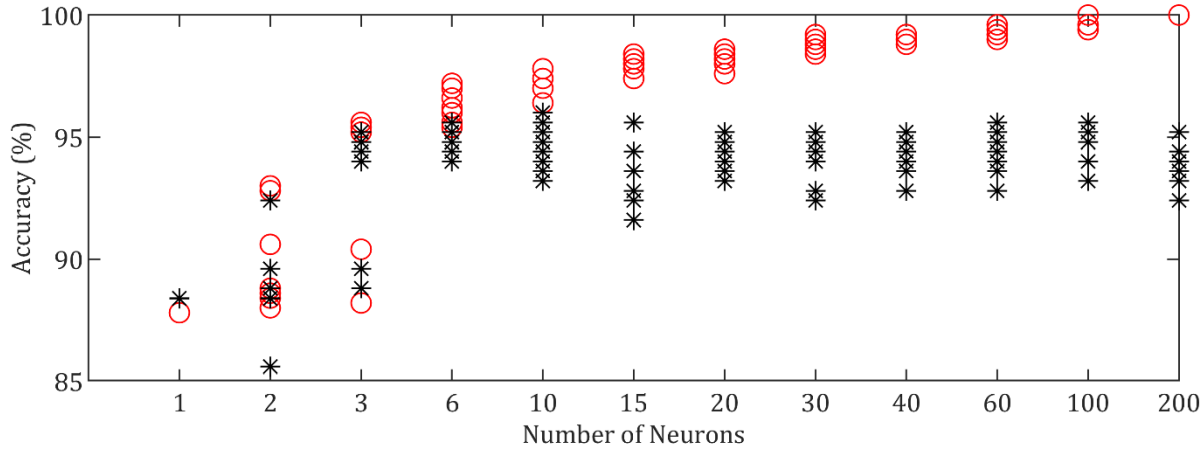
```
hlsConfigurations = [1,2,3,6,10,15,20,30,40,60,100,200];

for ConfigurationNumber = 1:length(hlsConfigurations)
        hiddenLayerSize = hlsConfigurations(ConfigurationNumber);
        % Create a MLP network:
        net = patternnet(hiddenLayerSize);

% ... - And here the rest of the code for testing the network
```



Rys 2 – Skuteczność treningowa i walidacyjna ze względu na ilość neuronów w warstwie ukrytej.

Okazuje się, że dokładność walidacji znacząco spada dla większych sieci, mimo że dokładność treningowa wciąż rośnie. Widzimy również, że zależność ta wydaje się zaszumiona – w przypadku sieci z 20 i 30 neuronami pojawia się spadek dokładności walidacji, który może, ale nie musi wynikać z właściwości samej sieci. Powtórzmy ten test statystycznie, aby mieć pewność. Zdecydowałem się uruchomić test 10 razy dla każdej wielkości sieci, a następnie zapisać wyniki w nowej strukturze *Statistics*.
Numer przypisany do wektora *Statistics* odpowiada numerowi testowanej konfiguracji. Częściowe wyniki są przechowywane w wektorach *TrainResults* i *ValResults*.
Na końcu mogę wyświetlić wyniki za pomocą poniższego kodu, którego rezultatem jest rys. 3. Zwróć uwagę na zielony fragment kodu – odpowiada on za ręczne ustawienie znaczników osi X (odpowiadających rzeczywistej liczbie testowanych neuronów).
Jeśli chcesz wygenerować podobne wizualizacje, musisz oczywiście dostosować ten kod do swojej struktury danych.

```
figure;
for configurationNumber = 1:length(Statistics)
    plot(configurationNumber, 100 * Statistics(configurationNumber).TrainResults, 'or'); hold on
    plot(configurationNumber, 100 * Statistics(configurationNumber).ValResults, '*k'); hold on
end

xlabel('Number of Neurons')
ylabel('Accuracy (%)')
xticks(1:length(Statistics))
xticklabels(string([Statistics(:).NumberOfNeurons]))
```



Rys 3 – Skuteczność treningowa i walidacyjna ze względu na ilość neuronów w warstwie ukrytej – zestawienie wyników 10 powtórzeń eksperymentu

Zadanie 4.3: Korzystając ze swojego **indywidualnego** zbioru danych do klasyfikacji z laboratorium 3, przeprowadź statystyczną ocenę skuteczności treningu i walidacji dla rosnącej liczby neuronów – zaczynając od 1 aż do 200. Przedstaw wyniki w formie graficznej. Następnie powtórz to samo zadanie dla problemu regresji. Zapisz wszystkie wyniki do swojej struktury danych.

Jeśli zadanie 4.3 zostało wykonane poprawnie, prawdopodobnie możemy zauważyć coś interesującego: nasza sieć klasyfikacyjna ma silną tendencję do przeuczenia, podczas gdy sieć regresyjna wydaje się znacznie bardziej odporna i lepiej uogólnia. Dzieje się tak, ponieważ tak naprawdę nie trenowaliśmy klasyfikatora we właściwy sposób. Ręcznie projektując zbiory treningowe i walidacyjne, wymusiliśmy na algorytmie, aby nie korzystał z walidacji jako kryterium zatrzymania treningu. Możemy rozwiązać ten problem na dwa sposoby.

Pierwszy sposób: przygotujemy osobny zbiór walidacyjny, odcinając część danych treningowych (zmniejszy to liczbę próbek dostępnych do uczenia, ale jest proste do zaimplementowania):

```
% Set custom division
net.divideFcn = 'dividerand';
net.divideParam.trainRatio = 0.7;
net.divideParam.valRatio   = 0.3;
net.divideParam.testRatio  = 0.0;
```

Albo lepiej: możemy połączyć treningowy I walidacyjny podzbiór a następnie zmusić siec do wykorzystania ich tak jak tego oczekujemy, na podstawie przypisanych do próbek indeksów:

```
% Combine data
X_all = [X_train, X_val];
T_all = [T_train, T_val];

% We will not be using the following division function now:
% net.divideFcn = 'dividetrain';

% Define indices (1 = train, 2 = validation, 3 = test)
trainInd = 1:size(X_train, 2);
valInd   = size(X_train, 2)+1 : size(X_train, 2)+size(X_val, 2);
testInd  = [];  % We again have no test data defined here

% Set custom division
net.divideFcn = 'divideind';
net.divideParam.trainInd = trainInd;
net.divideParam.valInd   = valInd;
net.divideParam.testInd  = testInd;
```

Zwróć uwagę, że aby drugi sposób zadziałał, będziemy musieli również przekazać nowy zbiór danych do sieci podczas treningu::

```
[net, tr] = train(net, X_all, T_all);
```

Zadanie 4.4: Zaktualizuj kod służący do oceny sieci klasyfikacyjnej w taki sposób, aby sieć mogła używać zbioru walidacyjnego jako kryterium zatrzymania treningu. Sprawdź, czy taka modyfikacja wpływa na statystyczną wydajność sieci – tzn. czy przeuczenie nadal występuje, czy może zostało ograniczone.

Zadanie 4.5: Przeprowadź statystyczną ocenę wydajności sieci klasyfikacyjnej dla dwóch różnych algorytmów treningowych: *trainscg* oraz *trainlm*, w zależności od rozmiaru sieci. Pozwól, aby rozmiar sieci zmieniał się w szerokim zakresie – tak, aby testowane były sieci o wielkości od 10 do co najmniej 300 neuronów w warstwie ukrytej. Sprawdź dokładność walidacji oraz czas treningu dla obu algorytmów, a następnie przedstaw wyniki w formie graficznej.

*Wskazówka: czas treningu można pobrać za pomocą **tr.time(end)** – jest to czas ostatniej iteracji treningu.*

Zadanie 4.6: Oceń znaczenie głębokości sieci neuronowej: zaprojektuj cel badawczy, a następnie eksperyment, który pozwoli określić, jakie są konsekwencje stosowania sieci z większą liczbą warstw ukrytych. Skonsultuj swój wybór eksperymentu z prowadzącym przed rozpoczęciem testów.

Wskazówka: Na tym etapie istnieje zbyt wiele zmiennych do sprawdzenia, więc musisz coś wybrać. Zastanów się, czy chcesz sprawdzić, czy wpływ liczby warstw jest podobny w problemach regresji i klasyfikacji? A może czy liczba warstw wpływa na czas treningu? A może czy pozwala osiągnąć lepszą dokładność, czy raczej utrudnia trening? Czy wpływ głębokości sieci jest podobny, czy różny dla różnych algorytmów treningowych? Celem jest sformułowanie pytania badawczego, które Cię interesuje, a następnie zaprojektowanie procedury, która pozwoli Ci uzyskać na nie odpowiedź.

Zadania dodatkowe

Zadanie 4.7: Zajrzyj do dokumentacji MATLAB-a w poszukiwaniu innych algorytmów uczenia sieci neuronowych, które można zastosować w zadaniach klasyfikacyjnych. Następnie znajdź informacje o tych algorytmach w internecie (możesz w tym celu skorzystać z ChatGPT), aby porównać je z *trainlm* i *trainscg*. Wybierz dwóch kandydatów, którzy wyraźnie różnią się od algorytmów, które już testowaliśmy. Następnie zaprojektuj eksperymenty, które pozwolą potwierdzić lub obalić właściwości, którymi różnią się od algorytmów które stosowaliśmy do tej pory.

Zadanie 4.8: Utwórz podzbiory swojego zbioru treningowego, losowo wybierając 10%, 30%, 60% i 100% oryginalnych próbek. Następnie wykonaj to samo dla zbioru walidacyjnego. Wytrenuj tę samą sieć neuronową dla każdej kombinacji podzbiorów treningowych i walidacyjnych. Powtórz eksperymenty tak, aby możliwa była statystyczna ocena wyników. Odpowiedz na pytanie, co jest ważniejsze – liczba próbek w zbiorze treningowym czy w walidacyjnym? Co zaobserwowałeś w kontekście: zachowania procesu uczenia, przeuczenia, stabilności działania sieci w warunkach ograniczonej liczby danych?

Zadanie 4.9: Spróbuj wielokrotnie powtórzyć trening tej samej sieci (ta sama architektura, te same dane, ten sam algorytm). Czy wyniki różnią się pomiędzy uruchomieniami? Dlaczego? Następnie jawnie ustaw początkowe wagi sieci, używając np. *net.initFcn* oraz/lub *net.IW{},* wybierając wartości startowe z różnych zakresów, i powtórz eksperymenty. Jak zakres losowej inicjalizacji danych wpływa na proces uczenia i końcowy rezultat? Pokaż wyniki w formie graficznej i sformułuj wnioski na temat znaczenia inicjalizacji sieci w zadaniach klasyfikacyjnych.

Zadanie 4.10 Używając sieci wytrenowanej do klasyfikacji dwuwymiarowej, oceń wyjście sieci dla gęstej siatki punktów pokrywających przestrzeń wejściową. Dla każdego punktu siatki narysuj przewidywaną przez sieć klasę, aby zwizualizować granicę decyzyjną. Następnie powtórz to samo dla sieci o różnych rozmiarach (z różną liczbą neuronów w warstwach ukrytych). Zbadaj: jak złożoność modelu wpływa na ostrość i kształt granicy decyzyjnej.

Zadanie 4.11 Wyłącz domyślną normalizację danych wejściowych, ustawiając przed treningiem: *net.inputs{1}.processFcns = {}*

Następnie wytrenuj sieć i zapisz jej wydajność. W kolejnym kroku samodzielnie przeskaluj dane wejściowe, np. standaryzując je do średniej zero i wariancji jeden. Następnie porównaj szybkość uczenia oraz dokładność walidacyjną z wcześniejszym przypadkiem.
Jakie wnioski możesz wyciągnąć na temat znaczenia wstępnego przetwarzania danych wejściowych?