Tomasz Zapiórkowski

**Podstawy Sztucznej Inteligencji – Sprawozdanie**

Scenariusz 2

**Opis wykonywanego ćwiczenia:**

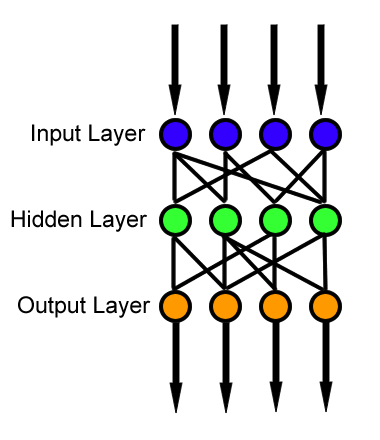
Celem ćwiczenia było poznanie i działanie sieci neuronowych oraz uczenie rozpoznawania wielkości liter.

**Wykonane zadania:**

1. Wygenerowano dane uczące i testujące, zawierające 10 wielkich i 10 małych liter alfabetu łacińskiego w postaci dwuwymiarowej tablicy 7x5, reprezentowanej w kodzie jako jednowymiarowa tablica 35 elementowa
2. Wykorzystano narzędzie PyBrain do stworzenia 2 sieci neuronowych (FeedForward i Recurrent) wraz z modyfikacją warstwy uczącej (Sigmoidalna i Tanh).
3. Uczono sieci modyfikując współczynnik uczenia.
4. Testowano poprawność działania sieci.

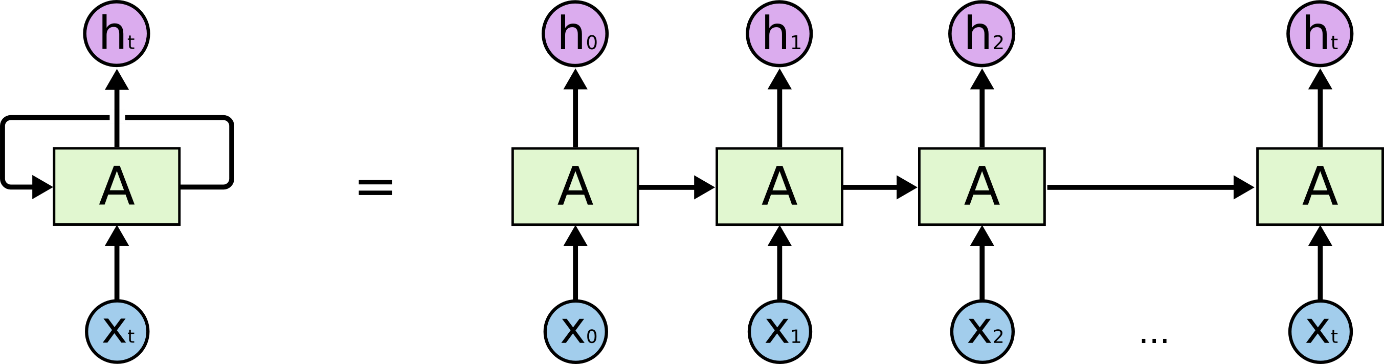
**Specyfikacja:**

Sieć neuronowa typu FeedForward opiera się na założeniu, że połączenia między neuronami nie tworzą cyklu, czyli informacja cały czas posuwa się w jednym kierunku, naprzód, zaczynając od warstwy wejścia, przez warstwę ukrytą(jeżeli istnieje) aż do warstwy wyjścia. Nie ma cyklów ani pętli w takiej sieci.



*Schemat sieci typu FeedForward*

Sieć neuronowa typu Recurrent to sieć, w której neurony połączone są za pomocą kierunkowych cykli. W sieciach rekurencyjnych istnieją sprzężenia zwrotne między wejściem, a wyjściem. Zmiana stanu jednego neuronu przenosi się przez masowe sprzężenie zwrotne na całą sieć.



*Schemat sieci typu Reccurent*

Sygnał wyjściowy neuronu w takiej sieci, gdzie f to aktywacja neuronu, a u to sygnałem aktywacji, to:

http://iisi.pcz.pl/nn/wz/80.gif

Neuron sigmoidalny jest budową zbliżony do perceptronu, różni się on przede wszystkim funkcją aktywacji i zwracaną wartością. Wykorzystuje on ciągłą funkcję aktywacji.

W neuronie sigmoidalnym do obliczenia nowej wagi wykorzystywana jest pochodna funkcji aktywacji: △*wij*(*k* + 1) = − *ηδixj* + *α*△*wij*(*k*)

gdzie:

i - numer neutronu

j - numer wejścia

w - waga

𝛈 - learning rate

𝛅 - pochodna funkcji aktywacji

Neuron tanh jest to modyfikacja neurona liniowego, jednak funkcja aktywacji nie jest liniowa a hyperboliczna. Konkretnie jest to tangens hiperboliczny.

Algorytm wstecznej propagacji błędu: metoda uczenia sieci wielowarstwowej, w której błąd ostatniej warstwy jest przesyłany wstecz i wykorzystywany do zmiany wartości wag w poprzednich warstwach.

Ogólny schemat procesu trenowania sieci wygląda następująco:

1. Ustalamy topologię sieci, tzn. liczbę warstw, liczbę neuronów w warstwach.
2. Inicjujemy wagi losowo (na małe wartości).
3. Dla danego wektora uczącego obliczamy odpowiedź sieci (warstwa po warstwie).
4. Każdy neuron wyjściowy oblicza swój błąd, oparty na różnicy pomiędzy obliczoną odpowiedzią *y* oraz poprawną odpowiedzią *t*.
5. Błędy propagowane są do wcześniejszych warstw.
6. Każdy neuron (również w warstwach ukrytych) modyfikuje wagi na podstawie wartości błędu i wielkości przetwarzanych w tym kroku sygnałów.
7. Powtarzamy od punktu 3. dla kolejnych wektorów uczących. Gdy wszystkie wektory zostaną użyte, losowo zmieniamy ich kolejność i zaczynamy wykorzystywać powtórnie.
8. Zatrzymujemy się, gdy średni błąd na danych treningowych przestanie maleć. Możemy też co jakiś czas testować sieć na specjalnej puli nieużywanych do treningu próbek testowych i kończyć trenowanie, gdy błąd przestanie maleć.

**Wykonanie zadania:**

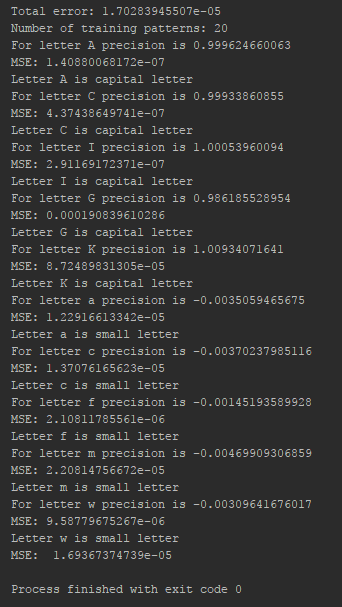
Do trenowania użyto wykonanego zestawu danych z małymi i dużymi literami. W obu sieciach do rozpoznawania wielkości liter użyto po 5 neuronów.

Zastosowano algorytm propagacji wstecznej.

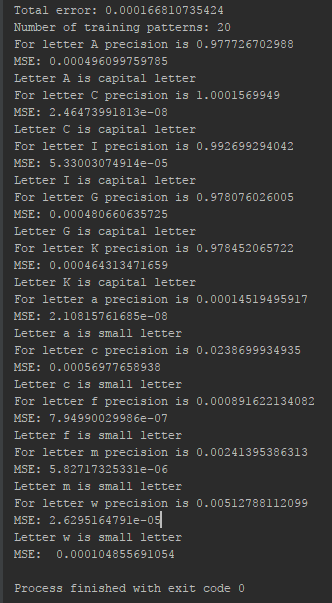
Wyniki:

Przykładowe zrzuty ekranu z procesu testowania:

lrate=0.1, epochs=1000



lrate=0.1, epochs=100



Wyniki zostały zamieszczone w formie wykresów w załączonym pliku.  
  
**Analiza:**

Proces uczenia odbywał się przez ograniczoną ilość epok, początkowo 10000 a następnie 2000, gdyż te 2000 było wystarczające w testowanych przypadkach. Lrate zmieniał się w zakresie 0.01-0.5.

Dla sieci FeedForward z neuronami sigmoidalnymi przy lrate=0.01 odpowiednio niski błąd(<0.001) został osiągany przy około 1000 epokach (zależnie od wartości początkowych). Dodatkowo na wykresie widać również wyraźny skok przy około 500 epokach. Oprócz tego skoku wykres jest dosyć ciągły bez wyraźnych skoków.  
Bardzo podobnie wyglądają wyniki dla lrate=0.05, z tym że odpowiednio niski błąd zostaje osiągnięty już po około 250 epokach, a skok na wykresie jest przy około 150 epokach.

Idąc dalej przy lrate=0.1 mamy analogiczną sytuację, jeszcze szybszy spadek błędu, już po 150 epokach błąd jest odpowiednio niski, a wyraźny skok widać po około 70 epokach.

Przy lrate=0.5 nie widać już tego skoku charakterystycznego dla wcześniejszych prób, jednak próg błędu zostaje osiągnięty po 40 próbach.

Kolejno przeszedłem do testowania sieci FeedForward z neuronami typu tanh. Dla lrate=0.01 próg błędu zostaje osiągnięty po 750 epokach, nie widać również wyraźnego skoku, wykres jest ciągły.

Dla lrate=0.5 próg błędu zostaje osiągnięty po 70 epokach, brak wyraźnego skoku.

**Dla wszystkich powyższych konfiguracji testowana sieć poprawnie rozpoznawała wielkość liter.**

Kolejno przeszedłem do testowania rekurencyjnej sieci z neuronami sigmoidalnymi. Lrate=0.01 spowodował, że odpowiednio niski błąd został osiągnięty dopiero po 1300 epokach. W dodatku sieć miała problemy z rozpoznawaniem wielkości litery ‘A’ gdzie zamiast zwracać wagę w okolicach 1, to wyniki wahały się od 0.5 do 2.

Przy lrate=0.5 sieć po około 100 epokach była odpowiednio nauczona, lecz znów powtarzał się problem z literą ‘A’.

Kolejno zmieniono konfigurację: sieć rekurencyjna z neuronami typu tanh. Dla lrate=0.01 widać wyraźną poprawę wyników. Próg błędu zostaje osiągnięty po 700 epokach, jednak nadal problem z literą ‘A’ występuje.

Ostatnią testowaną konfiguracją była powyższa z lrate=0.1, gdyż przy lrate=0.5 sieć była niestabilna i błąd rósł zamiast maleć. Co więcej nawet przy lrate=0.1 próg błędu został osiągnięty dopiero po 1100 epokach co było niewydajne, dodatkowo samo rozwiązanie było niestabilne gdyż występowały skoki na wykresie błędów.

**Wnioski**

Na podstawie otrzymanych wyników można stwierdzić, że do tego typu zastosowań sieć rekurencyjna nie jest odpowiednia. Po pierwsze działała tak samo albo wolniej od sieci FeedForward a w dodatku miała problem z rozpoznawaniem wielkości litery ‘A’.

Dodatkowo ewidentnie można zauważyć że neurony sigmoidalne lepiej zachowywały się przy dużym lrate. Niezależnie od typu sieci przy dużym lrate szybciej osiągały zamierzony próg błędu od neuronów typu tanh.

Neurony typu tanh za to znacznie lepiej radziły sobie przy małych lrate, przy dużych za to rozwiązania oparte na tych neuronach stawały się niestabilne.

Jednakże dla tak trywialnego zadania bardziej optymalne było wybranie wysokiego lrate co znacząco skróciło czas nauki sieci (z 1000 epok do 40 – poprawa 25 krotna).

**Listing:**

inputLetters.py

#!/usr/bin/python  
  
from pybrain3.datasets import SupervisedDataSet  
  
inputLettersDataSet = SupervisedDataSet(35, 1)  
  
#A  
inputLettersDataSet.addSample((  
 -1, 1, 1, 1, -1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 1, 1, 1, 1, 1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1  
 ),  
 1)  
  
#B  
inputLettersDataSet.addSample((  
 1, 1, 1, 1, -1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 1, 1, 1, 1, -1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 1, 1, 1, 1, -1  
 ),  
 1)  
  
#C  
inputLettersDataSet.addSample((  
 -1, 1, 1, 1, -1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, -1,  
 1, -1, -1, -1, -1,  
 1, -1, -1, -1, -1,  
 1, -1, -1, -1, 1,  
 -1, 1, 1, 1, -1  
 ),  
 1)  
  
#D  
inputLettersDataSet.addSample((  
 1, 1, 1, 1, -1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 1, 1, 1, 1, -1  
 ),  
 1)  
  
#F  
inputLettersDataSet.addSample((  
 1, 1, 1, 1, 1,  
 1, -1, -1, -1, -1,  
 1, -1, -1, -1, -1,  
 1, 1, 1, 1, -1,  
 1, -1, -1, -1, -1,  
 1, -1, -1, -1, -1,  
 1, -1, -1, -1, -1  
 ),  
 1)  
  
#G  
inputLettersDataSet.addSample((  
 -1, 1, 1, 1, -1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, -1,  
 1, -1, 1, 1, 1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 -1, 1, 1, 1, -1  
 ),  
 1)  
  
#H  
inputLettersDataSet.addSample((  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 1, 1, 1, 1, 1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1  
 ),  
 1)  
  
#I  
inputLettersDataSet.addSample((  
 -1, -1, 1, -1, -1,  
 -1, -1, 1, -1, -1,  
 -1, -1, 1, -1, -1,  
 -1, -1, 1, -1, -1,  
 -1, -1, 1, -1, -1,  
 -1, -1, 1, -1, -1,  
 -1, -1, 1, -1, -1  
 ),  
 1)  
  
#K  
inputLettersDataSet.addSample((  
 1, -1, -1, -1, 1,  
 1, -1, -1, 1, -1,  
 1, -1, 1, -1, -1,  
 1, 1, -1, -1, -1,  
 1, -1, 1, -1, -1,  
 1, -1, -1, 1, -1,  
 1, -1, -1, -1, 1  
 ),  
 1)  
  
#U  
inputLettersDataSet.addSample((  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 -1, 1, 1, 1, -1  
 ),  
 1)  
  
#a  
inputLettersDataSet.addSample((  
 -1, -1, -1, -1, -1,  
 -1, -1, -1, -1, -1,  
 -1, -1, -1, -1, -1,  
 -1, 1, 1, 1, -1,  
 -1, 1, -1, 1, -1,  
 -1, 1, -1, 1, -1,  
 -1, 1, 1, 1, 1  
 ),  
 0)  
  
#b  
inputLettersDataSet.addSample((  
 1, -1, -1, -1, -1,  
 1, -1, -1, -1, -1,  
 1, -1, -1, -1, -1,  
 1, 1, 1, 1, -1,  
 1, -1, -1, 1, -1,  
 1, -1, -1, 1, -1,  
 1, 1, 1, 1, -1  
 ),  
 0)  
  
#c  
inputLettersDataSet.addSample((  
 -1, -1, -1, -1, -1,  
 -1, -1, -1, -1, -1,  
 -1, -1, -1, -1, -1,  
 -1, 1, 1, 1, -1,  
 -1, 1, -1, -1, -1,  
 -1, 1, -1, -1, -1,  
 -1, 1, 1, 1, -1  
 ),  
 0)  
  
#d  
inputLettersDataSet.addSample((  
 -1, -1, -1, -1, 1,  
 -1, -1, -1, -1, 1,  
 -1, -1, -1, -1, 1,  
 -1, -1, 1, 1, 1,  
 -1, 1, -1, -1, 1,  
 -1, 1, -1, -1, 1,  
 -1, 1, 1, 1, 1  
 ),  
 0)  
  
#f  
inputLettersDataSet.addSample((  
 -1, -1, 1, 1, -1,  
 -1, -1, 1, -1, -1,  
 -1, -1, 1, -1, -1,  
 -1, 1, 1, 1, -1,  
 -1, -1, 1, -1, -1,  
 -1, -1, 1, -1, -1,  
 -1, -1, 1, -1, -1  
 ),  
 0)  
  
#h  
inputLettersDataSet.addSample((  
 1, -1, -1, -1, -1,  
 1, -1, -1, -1, -1,  
 1, -1, -1, -1, -1,  
 1, 1, 1, -1, -1,  
 1, -1, 1, -1, -1,  
 1, -1, 1, -1, -1,  
 1, -1, 1, -1, -1  
 ),  
 0)  
  
#m  
inputLettersDataSet.addSample((  
 -1, -1, -1, -1, -1,  
 -1, -1, -1, -1, -1,  
 -1, -1, -1, -1, -1,  
 1, 1, 1, 1, 1,  
 1, -1, 1, -1, 1,  
 1, -1, 1, -1, 1,  
 1, -1, 1, -1, 1  
 ),  
 0)  
  
#o  
inputLettersDataSet.addSample((  
 -1, -1, -1, -1, -1,  
 -1, -1, -1, -1, -1,  
 -1, -1, -1, -1, -1,  
 -1, 1, 1, 1, -1,  
 -1, 1, -1, 1, -1,  
 -1, 1, -1, 1, -1,  
 -1, 1, 1, 1, -1  
 ),  
 0)  
  
#w  
inputLettersDataSet.addSample((  
 -1, -1, -1, -1, -1,  
 -1, -1, -1, -1, -1,  
 -1, -1, -1, -1, -1,  
 1, -1, -1, -1, 1,  
 1, -1, -1, -1, 1,  
 1, -1, 1, -1, 1,  
 -1, 1, -1, 1, -1  
 ),  
 0)  
  
#z  
inputLettersDataSet.addSample((  
 -1, -1, -1, -1, -1,  
 -1, -1, -1, -1, -1,  
 -1, -1, -1, -1, -1,  
 -1, 1, 1, 1, 1,  
 -1, -1, -1, 1, -1,  
 -1, -1, 1, -1, -1,  
 -1, 1, 1, 1, 1  
 ),  
 0)

network.py

#!/usr/bin/python  
  
from pybrain3.structure import TanhLayer  
  
from pybrain3.structure import FeedForwardNetwork, LinearLayer, SigmoidLayer, FullConnection, RecurrentNetwork  
from pybrain3.structure.modules.biasunit import BiasUnit  
  
# network = FeedForwardNetwork()  
network = RecurrentNetwork()  
  
inLayer = LinearLayer(35)  
# hiddenLayer = SigmoidLayer(5)  
hiddenLayer = TanhLayer(5)  
outLayer = LinearLayer(1)  
bias = BiasUnit()  
  
network.addInputModule(inLayer)  
network.addModule(bias)  
network.addModule(hiddenLayer)  
network.addOutputModule(outLayer)  
  
in\_to\_hidden = FullConnection(inLayer, hiddenLayer)  
bias\_to\_hidden = FullConnection(bias, hiddenLayer)  
hidden\_to\_out = FullConnection(hiddenLayer, outLayer)  
  
network.addConnection(in\_to\_hidden)  
network.addConnection(bias\_to\_hidden)  
network.addConnection(hidden\_to\_out)  
  
network.sortModules()

trainer.py

#!/usr/bin/python  
  
from pybrain3.supervised.trainers import BackpropTrainer  
from pybrain3.tools.validation import Validator  
  
import inputLetters  
import network  
  
trainer = BackpropTrainer(network.network, inputLetters.inputLettersDataSet, learningrate=0.1, verbose=True)  
  
trainer.trainEpochs(2000)  
  
testInput = inputLetters.inputLettersDataSet['input']  
testTarget = inputLetters.inputLettersDataSet['target']  
errorComparator = 0.900  
  
print("Number of training patterns:", len(inputLetters.inputLettersDataSet))  
  
letters = ["A", "B", "C", "D", "I", "F", "G", "H", "K", "U", "a", "b", "c", "d", "f", "h", "m", "o", "w", "z"]  
  
MSE = 0  
  
for i in range(20):  
 temp = network.network.activate(testInput[i])  
 print("For letter", letters[i], "precision is", temp[0])  
 print("MSE:", Validator.MSE(network.network.activate(testInput[i]), testTarget[i]))  
 MSE += Validator.MSE(network.network.activate(testInput[i]), testTarget[i])  
 if errorComparator < temp:  
 print("Letter", letters[i], "is capital letter")  
 else:  
 print("Letter", letters[i], "is small letter")  
  
print("MSE: ", (MSE/20))