Inteligenţă artificială
Laboratorul 13

Algoritmul de retro-propagare (*backpropagation*)

1. Obiectiv

Obiectivul acestui laborator este prezentarea algoritmului de retro-propagare (engl. "backpropagation"), utilizat pentru antrenarea rețelelor neuronale de tip perceptron multistrat.

2. Descrierea algoritmului

Rețelele neuronale au capacitatea de a învăța, însă modalitatea concretă în care se realizează acest proces este dată de algoritmul folosit pentru antrenare. O rețea se consideră antrenată dacă aplicarea unui vector de intrare conduce la obținerea unei ieşiri dorite, sau foarte apropiate de aceasta.

Antrenarea constă în aplicarea secvențială a diferiți vectori de intrare și ajustarea ponderilor din rețea în raport cu o procedură predeterminată. În acest timp, ponderile conexiunilor converg gradual spre anumite valori pentru care fiecare vector de intrare produce vectorul de ieşire dorit. După aplicarea unei intrări, se compară ieşirea calculată cu ieşirea dorită, după care diferența este folosită pentru modificarea ponderilor cu scopul minimizării erorii la un nivel acceptabil.

Algoritmul backpropagation este cel mai cunoscut și utilizat algoritm de antrenare pentru rețele neuronale de tip perceptron multistrat. Numit și *algoritmul delta generalizat*, el se bazează pe minimizarea diferenței dintre ieşirea dorită și ieşirea reală, prin metoda gradientului descendent. Se definește *gradientul* unei funcții $F(x,y,z,...)$ drept:

$$\overrightarrow{\nabla \mathrm{F}} = \frac{\partial \mathrm{F}}{\partial x}\vec{i} + \frac{\partial \mathrm{F}}{\partial y}\vec{j} + \frac{\partial \mathrm{F}}{\partial z}\vec{k} + ...$$

Gradientul ne spune cum variază funcția în diferite direcții. Ideea algoritmului este găsirea minimului funcției de eroare în raport cu ponderile conexiunilor. Eroarea este dată de diferența dintre ieşirea dorită și ieşirea efectivă a rețelei. Cea mai utilizată funcție de eroare este cea a celor mai mici pătrate. Dacă avem în mulțimea de antrenare K vectori iar rețeaua are N ieşiri, atunci:

$$E = \frac{1}{2K} \cdot \sum_k \left(\sum_n \left(y_{kn}^d - y_{kn} \right)^2 \right),$$

unde y_{kn}^d este valoarea dorită la ieşirea n a rețelei pentru vectorul k, iar y_{kn} este ieşirea efectivă a rețelei.

Algoritmul pentru un perceptron multistrat cu un singur strat ascuns este următorul:

Pasul 1: Iniţializarea

Toate ponderile şi pragurile reţelei sunt iniţializate cu valori aleatorii *nenule*, distribuite uniform într-un mic interval, de exemplu $(-0.1, 0.1)$.

Dacă valorile acestea sunt 0, gradienţii care vor fi calculaţi pe parcursul antrenării vor fi tot 0 (dacă nu există o legătură directă între intrare şi ieşire) şi reţeaua nu va învăţa. Este chiar indicată încercarea mai multor antrenări, cu ponderi iniţiale diferite, pentru găsirea celei mai bune valori pentru funcţia cost (minimul erorii). Dimpotrivă, dacă valorile iniţiale sunt mari, ele tind să satureze unităţile respective. În acest caz, derivata funcţiei sigmoide este foarte mică. Ea acţionează ca un factor de multiplicare în timpul învăţării şi deci unităţile saturate vor fi aproape blocate, ceea ce face învăţarea foarte lentă.

Pasul 2: O nouă epocă de antrenare

O epocă reprezintă prezentarea tuturor exemplelor din mulţimea de antrenare. În majoritatea cazurilor, antrenarea reţelei presupune mai multe epoci de antrenare.

Pentru a păstra rigoarea matematică, ponderile vor fi ajustate numai după ce *toţi* vectorii de test vor fi aplicaţi reţelei. În acest scop, gradienţii ponderilor trebuie memoraţi şi ajustaţi după fiecare vector din mulţimea de antrenare, iar la sfârşitul unei epoci de antrenare, se vor modifica ponderile *o singură dată*. Există şi varianta "on-line", mai simplă, în care ponderile sunt actualizate direct; în acest caz, poate conta ordinea în care sunt prezentaţi reţelei vectorii de test.

Se iniţializează corecţiile ponderilor şi eroarea curentă cu 0: $\Delta w_{ij} = 0$, $E = 0$.

Pasul 3: Propagarea semnalului înainte

3.1. La intrările reţelei se aplică un vector din mulţimea de antrenare.

3.2. Se calculează ieşirile neuronilor din stratul ascuns:

$$y_j = f\left(\sum_{i=1}^{n} x_i \cdot w_{ij} - \theta_j\right),$$

unde n este numărul de intrări ale neuronului j din stratul ascuns, iar f este funcţia de activare sigmoidă.

Pentru simplitate, pragul este considerat ca fiind ponderea unei conexiuni suplimentare a cărei intrare este întotdeauna egală cu 1. Expresia de mai sus devine acum:

$$y_j = f\left(\sum_{i=1}^{n+1} x_i \cdot w_{ij}\right).$$

3.3. Se calculează ieşirile reale ale reţelei:

$$y_k = f\left(\sum_{j=1}^{m+1} y_j \cdot w_{jk}\right),$$

unde *m* este numărul de intrări ale neuronului *k* din stratul de ieşire.

3.4. Se actualizează eroarea pe epocă:

$$E = \frac{1}{2K} \cdot \sum_k e_k^2 = \frac{1}{2K} \cdot \sum_k \left(\sum_n \left(y_{kn}^d - y_{kn}\right)^2\right).$$

Pasul 4: Propagarea erorilor înapoi şi ajustarea ponderilor

4.1. Se calculează gradienţii erorilor pentru neuronii din stratul de ieşire:

$$\delta_k = f' \cdot e_k,$$

unde f' este derivata funcţiei de activare iar eroarea $e_k = y_k^d - y_k$.

Dacă folosim sigmoida unipolară, derivata acesteia este:

$$f'(x) = \frac{e^{-x}}{\left(1+e^{-x}\right)^2} = f(x) \cdot \left(1 - f(x)\right).$$

Vom avea:

$$\delta_k = y_k \cdot \left(1 - y_k\right) \cdot e_k.$$

4.2. Se actualizează corecţiile ponderilor dintre stratul ascuns şi stratul de ieşire:

$$\Delta w_{jk} = \Delta w_{jk} + \alpha \cdot y_j \cdot \delta_k,$$

unde α este rata de învăţare.

4.3. Se calculează gradienţii erorilor pentru neuronii din stratul ascuns:

$$\delta_j = y_j \cdot \left(1 - y_j\right) \cdot \sum_{k=1}^{l} \delta_k \cdot w_{jk},$$

unde *l* este numărul de ieşiri ale reţelei.

4.4. Se actualizează corecțiile ponderilor dintre stratul de intrare și stratul ascuns:

$$\Delta w_{ij} = \Delta w_{ij} + \alpha \cdot x_i \cdot \delta_j .$$

Pasul 5. O nouă iterație

Dacă mai sunt vectori de test în epoca de antrenare curentă, se trece la pasul 3.
Dacă nu, se actualizează ponderile tuturor conexiunilor pe baza corecțiilor ponderilor:

$$w_{ij} = w_{ij} + \Delta w_{ij} .$$

Dacă s-a încheiat o epocă, se testează dacă s-a îndeplinit criteriul de terminare ($E < E_{max}$ sau atingerea unui număr maxim de epoci de antrenare). Dacă nu, se trece la pasul 2. Dacă da, algoritmul se termină.

3. Considerente practice

În implementarea algoritmului, apar o serie de probleme practice, legate în special de alegerea parametrilor antrenării și a configurației rețelei.

În primul rând, o rată de învățare mică determină o convergență lentă a algoritmului, pe când una prea mare poate cauza eșecul (algoritmul va „sări" peste soluție). Pentru probleme relativ simple, o rată de învățare $\alpha = 0.5$ este acceptabilă, însă în general se recomandă rate de învățare în jur de 0.1 – 0.2.

Este recomandată preprocesarea și postprocesarea datelor, astfel încât rețeaua să opereze cu valori scalate, de exemplu în intervalul [0.1, 0.9] pentru sigmoida unipolară.

O altă problemă caracteristică acestui mod de antrenare este dată de minimele locale. Într-un minim local, gradienții erorii devin 0 și învățarea nu mai continuă. O soluție este încercarea mai multor antrenări independente, cu ponderi inițializate diferit la început, ceea ce crește probabilitatea găsirii minimului global. Pentru probleme mari, acest lucru poate fi greu de realizat și atunci pot fi acceptate și minime locale, cu condiția ca erorile să fie suficient de mici. De asemenea, se pot încerca diferite configurații ale rețelei, cu un număr mai mare de neuroni în stratul ascuns sau cu mai multe straturi ascunse, care în general conduc la minime locale mai mici. Totuși, deși minimele locale sunt într-adevăr o problemă, în practică nu reprezintă dificultăți de nesoluționat.

O chestiune importantă este alegerea unei configurații cât mai bune pentru rețea din punct de vedere al numărului de neuroni în straturile ascunse. În multe situații, un singur strat ascuns este suficient. Nu există niște reguli precise de alegere a numărului de neuroni. În general, rețeaua poate fi văzută ca un sistem în care numărul vectorilor de test înmulțit cu numărul de ieșiri reprezintă numărul de ecuații iar numărul ponderilor reprezintă numărul necunoscutelor. Ecuațiile sistemului sunt în general neliniare și foarte complexe și deci este foarte dificilă rezolvarea lor exactă prin mijloace convenționale. Algoritmul de antrenare urmărește tocmai găsirea unor soluții aproximative care să minimizeze erorile.

O rețea neuronală trebuie să fie capabilă de generalizare. Dacă rețeaua aproximează bine mulțimea de antrenare, aceasta nu este o garanție că va găsi soluții la fel de bune și pentru datele din altă mulțime, cea de test. Generalizarea presupune existența în date a unor regularități, a unui model

care *poate fi* învățat. În analogie cu sistemele liniare clasice, aceasta ar însemna niște ecuații redundante. Astfel, dacă numărul de ponderi este mai mic decât numărul de vectori de test, pentru o aproximare corectă rețeaua trebuie să se bazeze pe modelele intrinseci din date, modele care se vor regăsi și în datele de test. O regulă euristică afirmă că numărul de ponderi trebuie să fie în jur sau sub o zecime din produsul dintre numărul de vectori de antrenare și numărul de ieșiri. În unele situații însă (de exemplu, dacă datele de antrenare sunt relativ puține), numărul de ponderi poate fi chiar jumătate din produs.

Pentru un perceptron multistrat se consideră că numărul de neuroni dintr-un strat trebuie să fie suficient de mare pentru ca acest strat să furnizeze trei sau mai multe laturi pentru fiecare regiune convexă identificată de stratul următor. Deci numărul de neuroni dintr-un strat ar trebui să fie aproximativ de trei ori mai mare decât cel din stratul următor.

După cum am menționat, un număr insuficient de ponderi conduce la „sub-potrivire" (engl. "underfitting"), în timp ce un număr prea mare de ponderi conduce la „supra-potrivire" (engl. "overfitting"), fenomene prezentate în figura 1. Același lucru apare dacă numărul de epoci de antrenare este prea mic sau prea mare.

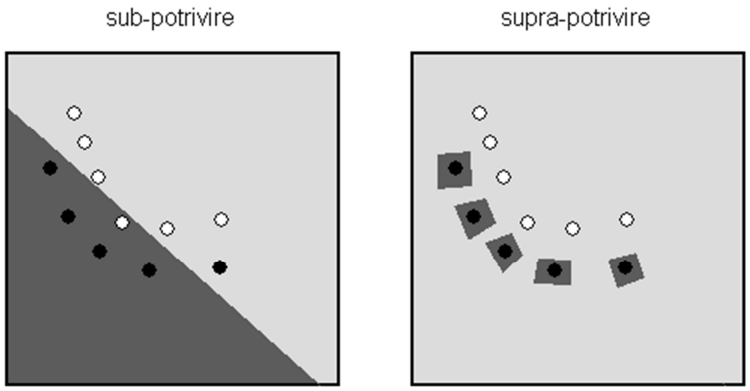


Figura 1. ***Capacitatea de generalizare***

O metodă de rezolvare a acestei probleme este oprirea antrenării în momentul în care se atinge cea mai bună generalizare. Pentru o rețea suficient de mare, eroarea de antrenare scade în mod continuu pe măsură ce crește numărul epocilor de antrenare. Totuși, pentru date diferite de cele din mulțimea de antrenare (mulțimea de test), se constată că eroarea scade la început până la un punct în care începe din nou să crească. De aceea, oprirea antrenării trebuie făcută în momentul când eroarea pentru mulțimea de *test* este minimă.

Cea mai simplă modalitate de a estima capacitatea de generalizare este împărțirea mulțimii de date disponibile în 2/3 pentru antrenare și 1/3 pentru testare. Antrenarea se oprește atunci când începe să crească eroarea pentru mulțimea de validare, moment numit „punct de maximă generalizare". În funcție de performanțele rețelei în acest moment, se pot încerca apoi diferite configurații, crescând sau micșorând numărul de neuroni din stratul (sau straturile) ascuns(e).

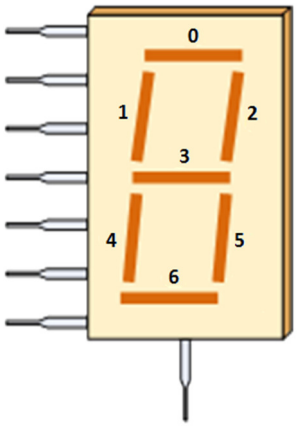
O metodă mai laborioasă, dar care este cea mai utilizată pentru evaluarea și compararea performanțelor modelelor de învățare, este *validarea încrucișată* (engl. "cross-validation"), în care datele sunt împărțite în n grupuri (de obicei 10), $n - 1$ grupuri se folosesc pentru antrenare, iar al n-lea pentru testare. Procesul se repetă de n ori, schimbând grupul de test. În final, se face media performanțelor obținute pentru cele n grupuri de test.

Pentru explicații suplimentare, consultați cursul 11.

4. Aplicație

Implementați algoritmul *backpropagation* pentru o rețea neuronală de tip perceptron multistrat, cu un singur strat ascuns și cu un număr variabil de unități în stratul ascuns.

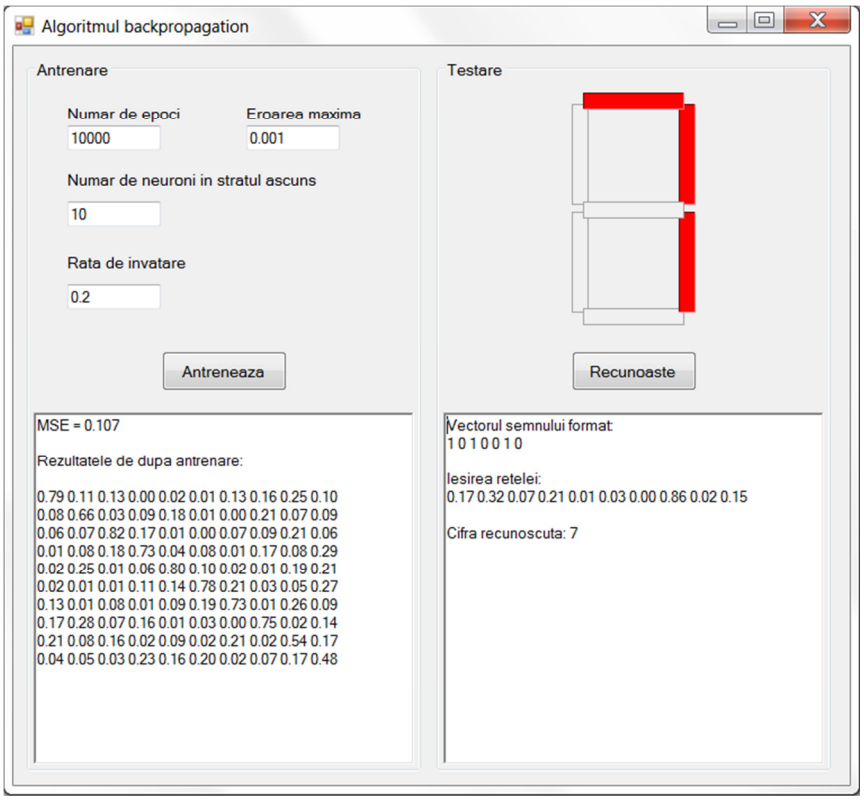
Se dă un prototip de aplicație care își propune recunoașterea cifrelor formate de utilizator pe un afișaj cu 7 led-uri.



Mulțimea de antrenare, citită din fișierul segments.data, conține 7 intrări (starea celor 7 led-uri), 10 ieșiri (1 pentru cifra formată, 0 în rest) și 10 vectori (cifrele 0-9):

```
7          10         10
1, 1, 1, 0, 1, 1, 1,     1, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 1, 0, 0, 1, 0,     0, 1, 0, 0, 0, 0, 0, 0, 0, 0
1, 0, 1, 1, 1, 0, 1,     0, 0, 1, 0, 0, 0, 0, 0, 0, 0
1, 0, 1, 1, 0, 1, 1,     0, 0, 0, 1, 0, 0, 0, 0, 0, 0
0, 1, 1, 1, 0, 1, 0,     0, 0, 0, 0, 1, 0, 0, 0, 0, 0
1, 1, 0, 1, 0, 1, 1,     0, 0, 0, 0, 0, 1, 0, 0, 0, 0
1, 1, 0, 1, 1, 1, 1,     0, 0, 0, 0, 0, 0, 1, 0, 0, 0
1, 0, 1, 0, 0, 1, 0,     0, 0, 0, 0, 0, 0, 0, 1, 0, 0
1, 1, 1, 1, 1, 1, 1,     0, 0, 0, 0, 0, 0, 0, 0, 1, 0
1, 1, 1, 1, 0, 1, 1,     0, 0, 0, 0, 0, 0, 0, 0, 0, 1
```


Florin Leon, Inteligenta artificiala - Laborator, http://florinleon.byethost24.com/lab_ia.htm



Trebuie implementate următoarele metode din clasa NeuralNetwork, corespunzătoare algoritmului *backpropagation*:

Metoda Train(int maxEpochs, double maxError)

Metoda de antrenare a rețelei. Antrenarea se realizează până la atingerea unui număr maxim de epoci (maxEpochs) sau până la atingerea unei erori acceptabile (maxError). Apelează metoda TrainOneEpoch, care implementează o epocă de antrenare pentru toți vectorii de antrenare și returnează eroarea medie pătratică obținută la finalul epocii.

Metoda ForwardPass(double[] vector)

Pasul de propagare înainte al algoritmului.

Metoda BackwardPass(double[] vector)

Pasul de propagare înapoi al algoritmului.

Metoda SigmoidDerivative(double f)

Expresia derivatei funcției sigmoide unipolare, unde f este valoarea funcției: $f' = f \cdot (1 - f)$.

Metoda UpdateWeights()

Actualizarea ponderilor w pe baza corecțiilor ponderilor Δw.