

Pretele Neuronale

Cours 5 :

→ Metrii de clasificare:

$$\hookrightarrow \text{Accuracy} = ACC = \frac{TP + TN}{TP + FP + TN + FN}$$

(utilizat cînd datasetul e balanced)

$$\hookrightarrow \text{False Positive Rate} = FPR = \frac{FP}{FP + TN}$$

$$\hookrightarrow \text{False Negative Rate} = FNR = \frac{FN}{FN + TP}$$

(utilizate cînd datasetul e balanced)

$$\hookrightarrow \text{Precision} = \frac{TP}{TP + FP}$$

(utilizat cînd impactul unei predicții greșite e mare)

$$\hookrightarrow \text{Recall} = \frac{TP}{TP + FN}$$

(utilizat cînd lipsă unui sample pozitiv este foarte rău)

$$\hookrightarrow \text{Specificity} = \frac{TN}{TN + FP}$$

→ Tipuri de neuroni

\hookrightarrow Lineari = outputul este o sumă ponderată a inputurilor + un bias

$$y = \sum_i w_i x_i + b$$

\hookrightarrow Binary Threshold = calculatează suma ponderată a inputurilor; dacă este mai mare decât un threshold (θ), afisează 1; altfel afisează 0

$$y = \begin{cases} 1 & \text{dacă } \sum_i w_i x_i \geq \theta \\ 0 & \text{altfel} \end{cases}$$

$$\text{Dacă } \theta = -b \text{ atunci } y = \begin{cases} 1 & \text{dacă } \sum_i w_i x_i + b \geq 0 \\ 0 & \text{altfel} \end{cases}$$

\hookrightarrow Rectified Linear Unit = combină un neuron linear și un binary threshold unit

$$z = \sum_i w_i x_i + b$$

$$y = \begin{cases} z & \text{dacă } z \geq 0 \\ 0 & \text{altfel} \end{cases}$$

\hookrightarrow Logistic = $y = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-\sum_i w_i x_i + b}}$

(cel mai utilizat într-o rețea neuronală)

\hookrightarrow Stochastic linear = consideră valoarea ca pe o probabilitate de a genera valoarea 1 : $z = \sum_i w_i x_i + b$

$$P = \frac{1}{1+e^{-z}}$$

$$y = \begin{cases} 1 & \text{cu probabilitate } p \\ 0 & \text{cu probabilitate } 1-p \end{cases}$$

Curs 2

Algoritmul Perceptron + clasificator liniar definit de o ecuație liniară

Algoritmul Perceptron + clasificator liniar definit de o ecuație liniară

Algoritmul Perceptron + clasificator liniar definit de o ecuație liniară

↳ În dețină "căutând" eroranta:

output = $\begin{cases} 1 & \text{daca } \sum_{i=0}^n w_i \cdot x_i + b > 0 \\ 0 & \text{daca } \sum_{i=0}^n w_i \cdot x_i + b \leq 0 \end{cases}$ $b = \text{bias} = -\text{threshold}$

↳ Încărcarea este performată variind weighturile și thresholdul

↳ Util pentru clasificarea datele ce sunt liniar-separabile.

↳ Teorema de convergență a perceptronului spune că dacă un dataset este liniar separabil atunci algoritmul perceptron va găsi întotdeauna un hiperplan de separare într-un număr finit de pași

ONLINE TRAINING

↳ Testăm dacă fiecare element este clasificat corect; dacă nu este clasificat corect, ajustăm hiperplanul în direcția dorită

↳ Repetăm procedeul până când toate datele sunt clasificate corect sau până când s-a depășit un număr de iterări

BATCH TRAINING

↳ Se lezează pe aceeași idee cu cea de la ONLINE (ajustarea hiperplanului în caz de greșală), dar de această dată, se updatează un termen intermediar (Δ pentru weights, β pentru bias) care va fi folosit la final pentru a ajusta hiperplanul

↳ Δ conține toate ajustările necesare pentru toate elementele din batch

BATCH vs. ONLINE:

↳ În ONLINE învățarea se face cu un singur element adăugat, în BATCH se face considerând mai multe elemente pentru o singură ajustare a weighturilor.

↳ Ordinea MJ este importantă în BATCH, în ONLINE datasetul trebuie amestecat.

↳ BATCH trainingul poate fi paralelizat.

↳ În BATCH training ajustările weighturilor se fac mai rare și sunt mai orientate pe cele care minimizează eroarea tuturor exemplușilor.

MINIBATCH TRAINING:

- ↳ Recomandă că în BATCH training, doar că de acasă datele ajustarea se face la finalul fiecărui batch

MINIBATCH vs. BATCH vs. ONLINE:

- ↳ MiniBatch este la mijlocul BATCH și ONLINE în termenul utilizării.
- ↳ Ordinea elementelor în MiniBatch training afectează rezultatul, necesitând amestecarea datasetului.
- ↳ Parallelizarea poate fi utilizată.
- ↳ MiniBatch face updateuri mai des decât Batch Training; poate obține rezultate similare mult mai rapid.

Perceptronul ADALINE

- ↳ Perceptronul standard nu se mai adaptează dacă toate elementele sunt corect clasificate; hiperplanul recalculator nu va generaliza bine pentru elemente neterminate.
- ↳ Dacă elementele nu sunt liniar separabile, perceptronul nu oprește ajustarea weighturilor și nu se stabilizează la o limită de decizie liniară.
- ↳ Labelurile samplelor (target) va conține valori de -1, 0, 1 sau de 0, 1; în cazul perceptronului, labelurile reprezintă mai mult o convenție.
- ↳ Se ia în calcul și căt de său este clasificata gresit o întrebare.

Curs 3

- ↳ Arhitectura unei rețele Feed-Forward.
- ↳ Compusă din minimum 3 straturi (Input - Hidden - Output)
- ↳ Fiecare layer conține un număr de neuroni / unități.
- ↳ Fiecare unitate este conectată simetric cu toate unitățile din layerul următor.
- ↳ Neuronii din stratul ascuns și din cel de output sunt nonliniarii.

Gradient Descent

↳ ajustăm weighturile și biasurile în aşa fel încât să minimizăm o funcție de cost (axigată o valoare pentru că de rău este clasificată o instanță)

↳ Funcție Mean-Square-Error: $C(w, b) = \frac{1}{2m} \sum_{i=1}^m (t_i - y_i)^2$

w - toate weighturile din rețea

b - toate biasurile din rețea

t - vectorul-target pentru inputul x

y - outputul pentru x ($g(x)$)

BACKPROPAGATION (formulele sunt valabile pentru mean-square-error)

↳ Pas 0: Calculăm eroarea de pe ultimul strat:

$$\delta_i^L = y_i^L (1 - y_i^L) (y_i^L - t_i)$$

L - last layer w_{ij}^e - weightul neuronului i de pe stratul $l-1$ ce merge la neuronul j de pe stratul l

y_i^L - activarea neuronului i de pe stratul L

b_i^L - biasul neuronului i de pe stratul L

z_i^L - inputul neuronului i din layerul l $\leftarrow \sum_j w_{ji}^e \cdot y_j^{L-1} + b_i^L$

↳ Procesăm stratul de jos

↳ Pas 1: Calculăm eroarea stratului anterior:

$$\delta_i^L = y_i^L (1 - y_i^L) \cdot \sum_k \delta_k^{L+1} \cdot w_{ik}^{L+1}$$

↳ Pas 2: Calculăm gradientul weighturilor din stratul curent:

$$\frac{\partial C}{\partial w_{ij}} = \delta_j^L \cdot y_i^{L-1}$$

↳ Pas 3: Calculăm gradientul biasurilor din stratul curent:

$$\frac{\partial C}{\partial b_i} = \delta_i^L$$

↳ Pas 4: Ajustăm weighturile neuronilor din layerul anterior și repetăm pași anteriori:

$$w_{ij}^L = w_{ij}^L - \frac{\partial C}{\partial w_{ij}} \cdot \eta$$

$$b_i^L = b_i^L - \frac{\partial C}{\partial b_i} \cdot \eta$$

Curs 4:

- Problema costului patratic**: - Înredareea este onoarea când costul este mare
- ↳ Dacă eroarea este mare, dorim să facem ajustări mari pentru a scădea costul.
 - ↳ Dacă eroarea este mică, dorim să facem ajustări mici pentru a nu depăși limita.

→ Utilizând o altă funcție de cost, problema este rezolvată

$$\text{Cross-Entropy} = -\frac{1}{m} \sum_{i=1}^m (t_i \ln y_i + (1-t_i) \ln(1-y_i))$$

$$\frac{\partial C}{\partial w_i} = -\frac{1}{m} \sum_{i=1}^m (t_i - y_i) \frac{\partial}{\partial w_i} \left(\frac{e^{z_i^T w}}{\sum_k e^{z_k^T w}} \right)$$

→ Softmax ($y_j^T = \frac{e^{z_j^T w}}{\sum_k e^{z_k^T w}}$) este o funcție de activare ce funcționează bine cu funcția de cost cross-entropy când datele ce trebuie clasificate fac parte din multimi disjuncte.

↳ Transformă valoriile de output în probabilități.

Problema saturării unui neuron: - Atunci când inputul net al unui neuron este mare, rezultatul activării se apropiă de

limitele asymptotice de reprezentare în memorie.

↳ Astfel, soluția este de a inițializa weighturile cu valori în astfel încât să nu saturize neuronul.

! Toate valoriile sunt inițializate cu valori aleatorie urmând o distribuție normală cu media 0 și deviația standard $\frac{1}{\sqrt{m_{in}}}$

m_{in} = numărul total de conexiuni ce reîn către un neuron

↳ Utilizând weighturi mici, rețeaua ajunge mai repede la acuratețea maximă și răstește în acest mod faza de călătorie care nu utilizează distribuția menționată.

Ajustarea hiperparametrilor (learning-rate, marime mini-batch, număr epoci, număr rețele ascunse)

↳ Alegerea numărului rețele ascunse este importantă; se alege dimensiunea cea mai bună (care obține acuratețea la validare cea mai mare în funcție de temp)

↳ La început, rețeaua este antrenată pentru un număr mare de iterații; apoi se recurge la strategia de early-stopping (după fiecare iterație rețeaua este testată pe setul de validare și dacă în ultimele X iterări nu se descreză niciodată în lumenătate - ne oprim. X poate fi 10)

↪ ajustarea ratei de învățare se face prin magnitudine (η) până observăm că asciindat cîntul; acesta este thresholdul, iar valoarea cea mai bună pentru learning rate ar trebui să fie un factor sau doar sub threshold (se obține crescînd η cu o valoare mică până când se apropiie de threshold)

Burs 5

↪ Overfitting = fenomen în care acuratețea modelului la realizare este semnificativ mai mică decât acuratețea modelului pe setul de antrenare

↪ Soluții:

(1) Regularizare

↪ Scopul regularizării este de a controla căt de mult bias (măsură căt de departate sunt valoile de tîntă) vs. variantă (căt de împrăștiat sunt elementele estimate de media lor) dorim.

↪ Ideea este de a obține un echilibru între bias și variantă (modelele cu bias mare produc underfitting iar cele cu varianta mare produc overfitting)

↪ Regularizare L2 (WEIGHT DECAY) - modificăm funcția de cost $C = C_{initial} + \frac{\lambda}{2m} \sum_w w^2$ (λ - parametru de regularizare)

La Backpropagation, weighturile se modifîcă astfel:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{m} w ; w = w - \eta \frac{\partial C_0}{\partial w} - \eta \frac{\lambda}{m} w$$

↪ Regularizare L1 - $C = C_{initial}(C_0) + \frac{\lambda}{m} \sum_w |w|$

La Backpropagation, weighturile se modifîcă astfel:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{m} \text{sgn}(w) \quad w = w - \lambda \frac{2}{m} \text{sgn}(w) - m \frac{\partial C_0}{\partial w}$$

L1 vs. L2

↪ L1 modifica weighturile cu o valoare constantă (dacă sunt foarte mari, nu vor fi reduse foarte mult; dacă sunt mici, vor fi reduse și)

↪ L2 modifica weighturile cu o valoare ce depinde de weight în sine (pentru weighturi mari se obțin valori mari de scădere; pentru valori mici se obțin valori mici de creștere)

↪ L1 tende să creze modelul pe anumite weighturi pe când L2 ia în calcul mai multe weighturi dar cu valori mai mici.

(2) Dropout:

- ↪ La antrenare, rețeaua folosește mai puține weighturi
- ↪ Pentru fiecare minibatch de antrenare, selectăm aleator un procent din neuroni ascunși și îi facem inactivi; acești neuroni nu vor face parte din antrenament deoarece weighturile ce vor să pleacă și din acesti neuroni nu sunt luate în considerare
- ↪ După procesul de backpropagation, weighturile inactivelor sunt restaurate și se aleg alți neuroni ce vor fi inactivi.

↪ Alte avantaje:

- ↪ Face rețeaua rezistentă la pierderea oricărui neuron individual
- ↪ Crește eficiența deoarece mai mulți neuroni participă în procesul de învățare

(3) Maxnorm:

- ↪ Limităază weighturile matricei ale fiecărui neuron în mod individual

$$\hookrightarrow \text{Dacă } \|w\|_2 = \sqrt{\sum_i w_i^2} > c \text{ atunci } W = W * \frac{c}{\|w\|_2 + \epsilon}$$
$$\epsilon = 10^{-8}$$

- ↪ Permite utilizarea unei rate de învățare mare

(4) Augmentarea setului de date:

- ↪ Se poate face pe 2 căi:

(I) Adăugând date noi setului

(II) Adăugând date generate pe baza datelor deja existente (fake-data)

↪ Modificând o imagine (flip, crop, mutare la stânga/dreapta sus/jos etc. cu un anumit număr de pixeli) obținem instanțe noi

Curs 6

↪ Problemele Stochastic Gradient Descentului

(1) Rata de învățare: dacă este prea mare, creșterea va oscila de-a lungul graficului; dacă este prea mică, nu va exista îmbunătățiri notabile

(2) Nu întotdeauna, la fiecare pas, minimizăm eroarea.

Optimizatori

↪ **Momentum** → Trăbește pasul de învățare; acest lucru este obținut prin varierea ratei de învățare (când gradientul este orientat în aceeași direcție, acumulează velocitate)

$$v_{ij} = \mu v_{ij} - \eta \frac{\partial C}{\partial w_{ij}} \quad w_{ij} = w_{ij} + v_{ij}$$

μ - valoare de frecare / coefficient momentum $\in [0, 1]$

↪ Crește acuratețea dacă este utilizat cu software.

↪ **Nesterov Accelerated Gradient NAG** → Momentum modificat; la fiecare moment de timp se evaluă și se approximează unde ne va duce momentumul; pe baza acestei approximări, putem ajusta mai bine direcția

$$v_{ij}^t = \mu v_{ij}^{t-1} - \eta \frac{\partial C}{\partial w_{ij}^{t-1}} \quad w_{ij}^t = w_{ij}^{t-1} - \mu v_{ij}^{t-1} + \mu v_{ij}^t + v_{ij}^t$$

↪ Superior momentului standard deoarece featureal de look-ahead îi permite să-și corecteze direcția mai rapid.

↪ **Resilient Propagation Rprop** → În calcul semnul gradientului un parametru în plus ($\Delta\theta_i$) care stocă mărimea de deplasare în direcția datea de gradient; reține semnul gradientului din iterată precedență și îl compara cu semnul gradientului curent

↪ Dacă semnul este același, $\Delta\theta_i = \Delta\theta_i + 1.2$ (crește) (maxim 50)

↪ Altfel, $\Delta\theta_i = \Delta\theta_{i-1} * 0.5$ (scade) (minimum 10^{-6})

↪ Valorile inițiale pentru $\Delta\theta_i > 0$ (de obicei 0.01)

Avantaje → Deoarece updateurile sunt independente de mărimea gradientului, poate evita plateauurile foarte rapid (la fel ca Momentum)

→ Nu necesită ajustarea unui factor momentum sau a unei learning rate

→ Unul dintre cei mai rapizi algoritmi disponibili

Deavantaje → Nu funcționează pe minibatching / dataseturi mari

→ **Adaptive Gradient** → Adăptația fiecarei rute de învățare la marimea gradientului (gradientă mare vor primi rate de învățare mici, cei mici rate de învățare mari)

→ Dividă fiecare gradient cu o normă L2 a gradientilor din trecut $m_i^t = \frac{m}{\sqrt{\sum_{k=1}^t (\nabla \theta_i^k)^2}}$ $\theta_i^t = \theta_i^{t-1} - \eta_i^t \cdot \nabla \theta_i^t$

θ_i^t - valoarea unui parametru (weight / bias) la momentul t
 $\nabla \theta_i^t$ - gradientul parametrului i la momentul t

η_i^t - rata de învățare a parametrului i la momentul t

Avantaje - Adăptația feature-urilor rare (cele care apar rar în dataset vor fi ajustate cu rate de învățare mai mari)

- Cele mai des întâlnite feature-uri, cu gradienți mari, vor conduce gradientul la 0.

Desavantaje - Se blochează în platouri (norma L2 poate să crească)

- Nu crește acuratețea dacă se utilizează activare sigmoid
- Punctul de plecare este important (valoarea inițială a learning rate-ului); definește cât de mult va continua Adagrad să învețe.

→ **Root Mean Square Propagation** → Construit pe AdaGrad (ajută la separarea ratele de învățare pentru fiecare weight utilizând gradienți din trecut) deoarece retine o medie exponentială de rulare, împărțind rata de învățare la fiecare iterație cu această valoare

$$E[(\nabla \theta_i^t)^2] = \gamma E[(\nabla \theta_i^{t-1})^2] + (1-\gamma)(\nabla \theta_i^t)^2$$

$$m_i^t = M / \sqrt{E[(\nabla \theta_i^t)^2] + 10^{-8}}$$

$$\theta_i^t = \theta_i^{t-1} - \eta_i^t \nabla \theta_i^t$$

↳ Una dintre cele mai utilizate metode în rețelele neuronale obține rezultate leibile dacă este utilizată cu funcții de activare ReLU (crește acuratețea)

→ Adaptive Moment Estimation → Construit pe RMSProp și Momentum
 Adam → Folosește rate de învățare pentru fiecare weight împărțind la norma gradientului anterior și folosește un factor similar cu momentum

$$M_t = \beta_1 M_{t-1} + (1-\beta_1) \nabla \theta_i^t - \text{estimarea momentumului}$$

$$R_t = \beta_2 R_{t-1} + (1-\beta_2) (\nabla \theta_i^t)^2 - \text{ratea de decay exponentială medie a gradientelor patrate}$$

$$\beta_1, \beta_2 \in [0.9, 0.999]$$

→ Deoarece valurile M_0, R_0 sunt 0 la început, M_t și R_t sunt estimate la 0 în timpul primilor pasi de execuție; de aceea, M_t și R_t se împart cu un factor de corecție:

$$\hat{M}_t = \frac{M_t}{1-\beta_1^t} \quad \hat{R}_t = \frac{R_t}{1-\beta_2^t}$$

→ Update similar cu RMSProp și Adagrad: $\theta_i^t = \theta_i^{t-1} - \eta \cdot \frac{\hat{M}_t}{\sqrt{\hat{R}_t + \epsilon}}$

→ AdaGrad → Apărut simultan cu RMSProp și are același rezolvarea ratelor de învățare ce scad radical la AdaGrad

→ În loc de a utiliza un learning rate (ca RMSProp), folosește o medie exponentială de rulare a factorilor delta precedenți

→ Intuiția vine de la metoda lui Newton de a approxima gradientă cu serii Taylor.

$$E[(\nabla \theta_i^t)^2] = \gamma E[(\nabla \theta_i^{t-1})^2] + (1-\gamma)(\nabla \theta_i^t)^2$$

$$\Delta \theta_i^t = \frac{\sqrt{E[(\nabla \theta_i^{t-1})^2]}}{\sqrt{E[(\nabla \theta_i^t)^2]} + \epsilon} \cdot \nabla \theta_i^t$$

$$\theta_i^t = \theta_i^{t-1} - \Delta \theta_i^t$$

Curs 7

Problema funcțiilor de activare logistice

↳ output întotdeauna pozitiv ⇒ dacă un neuron trebuie să schimbe direcția, este nevoie să facă mai mulți pasi.

↳ Media activărilor pe un layer trebuie să fie 0 (concluzie pe layerul de output)

↳ Outputul poate fi 0 când neuroni sunt saturati ⇒ gradientă devenind 0

↪ Sigmoide: funcția $\tanh = \text{tangenta hiperbolică}$

$$\frac{e^x - e^{-x}}{e^x + e^{-x}}$$

↪ valoare medie = 0

↪ saturarea pentru valoriile -1 și 1, deci reweighturile vor fi mult ajustate.

↪ gradientul \tanh sunt mai mari decât cei sigmoidi; eroarea este minimizată mai rapid

↪ Vanishing Gradient = eroarea trebuie să fie mai mică pe măsură ce back propagation ajunge la primele straturi (toate valoările din gradient sunt subunitare), adaptându-se pe leașa valoilor mari; problema apare când folosim reweighturi mici

↪ Exploding Gradient = valoările calculate pentru gradient sunt foarte mari (atunci când utilizăm reweighturi mari)

→ ReLU - Rectified Linear Unit

$$\text{relu}(x) = \max(0, x) = \begin{cases} 0 & \text{dacă } x < 0 \\ x & \text{dacă } x \geq 0 \end{cases} \quad \text{relu}'(x) = \begin{cases} 0 & \text{dacă } x > 0 \\ 1 & \text{dacă } x \leq 0 \end{cases}$$

Avantaje

↪ Resolvă problema vanishing gradient

↪ Ieftină din punct de vedere computațional

↪ 50% din neuroni au valoarea 0 după initializarea weighturilor; aceasta variație poate crea regulatizarea

↪ Gradientul cînd user pe căile neuronilor active.

↪ Gradientul cînd user pe căile neuronilor inactice.
(Funcțiile calculate de fiecare neuron sunt liniare în părți)

Deavantaje

↪ Nu este centralizat în 0 - se poate rezolva cu batch normalization

↪ Nu are limite și suferă de gradient exploding - se poate rezolva limitând valoarea gradientului

↪ Dying ReLU (pentru valori negative, gradientul sunt 0 și nu propagă eroarea mai departe, producând neuroni morți) - se poate rezolva utilizând variante diferite de ReLU

- ↳ Leaky ReLU; în loc de 0 absolut, returnează $0.01 * \text{input}$
- ↳ parametric ReLU; în loc de 0, returnează $\alpha * \text{input}$ unde α este un număr foarte mic
- ↳ exponential ReLU; pentru valori negative, returnează

$$y = \alpha(e^x - 1)$$

→ Elimină problema Dying ReLU; ultima funcție saturată pentru valori mari

Retele Radial Basis Function

- ↳ Contine 3 straturi - Input, Hidden, Output
- ↳ În fiecare neuron ascuns descrie un set de elemente din spațiul de input - mai este denumit prototip; poate fi chiar un element din spațiul de input (cu căt este mai reprezentativ, cu atât mai bine)
- ↳ În fiecare vector de input va fi comparat pe leza unei distanțe cu fiecare neuron ascuns.
- ↳ În fiecare neuron ascuns va returna căt de similar este inputul de antrenare cu neuronalul ascuns.
- ↳ Bazat pe activarea fiecărui neuron ascuns, neuroni de output vor calcula o sumă ponderată și vor alegea o clasă inputului.
- Antrenament
 - ↳ (1) Găsirea prototipurilor corecte (neșterzisat)
 - ↳ (2) Îmrețarea modului în care sunt combinate outputurile dat de fiecare prototip (șterzisat)
- Funcția de activare
 - ↳ $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ $\sigma = 2\sigma^2$ - controlată dimensiunea similarității considerate de prototip
 - ↳ $\mu = \frac{1}{\sigma\sqrt{2\pi}}$ - controlată multimea graficului
 - ↳ f măsoară distanța dintre instanță (x) și prototip (μ)
 - ↳ Funcția de activare va fi: $e^{-\rho(x-\mu)^2}$
- Locația fiecărui prototip: orice algoritm de clusterizare poate fi folosit (kMeans, kMeans++)

- Valorile pentru β :

$$(1) \beta = \frac{1}{2\sigma^2} - \text{din ecuația Guisarnei}$$

$$(2) \sigma = \sqrt{\frac{1}{N} \sum_i^N \|x_i - \mu\|^2}$$

Guisarnea

μ - centrul clusterului
 x_i - un element din cluster
 N - totalul numărului de elemente din cluster

- Output - fiecare neuron de output este un clasificator liniar antrenat separat

↳ (1) Un element / batch va fi setat ca input

(2) Outputul fiecărui cluster va fi considerat parte din inputul perceptronului

(3) Weighturile se vor ajusta la fel ca în casul unui perceptor

↳ În cele mai multe cazuri, neuronul specific unei clase va încerca să dea weighturi pozitive tuturor probabilităților din această clasă și weighturi negative celorlalte.