Rețele neuronale.
Concepte despre modele de învățare deep.

Prof. Dr. Radu Ionescu
raducu.ionescu@gmail.com
Facultatea de Matematică și Informatică
Universitatea din București

Din cursul trecut:

- O mulțime de perechi (x,y)
- O funcție de atribuire a scorului: $s = f(x; W) = Wx$ e.g.
- O funcție de pierdere:

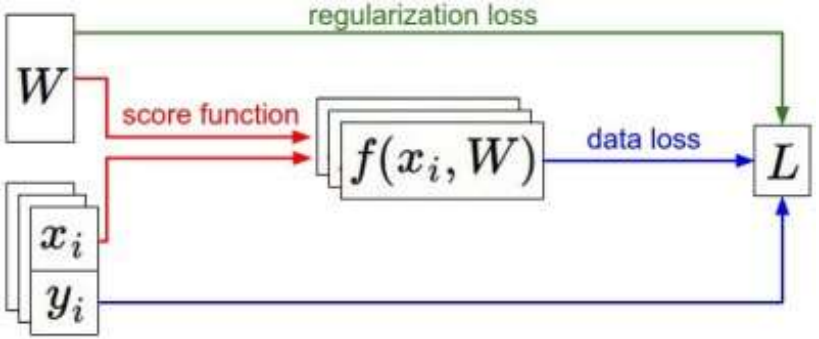
Softmax

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

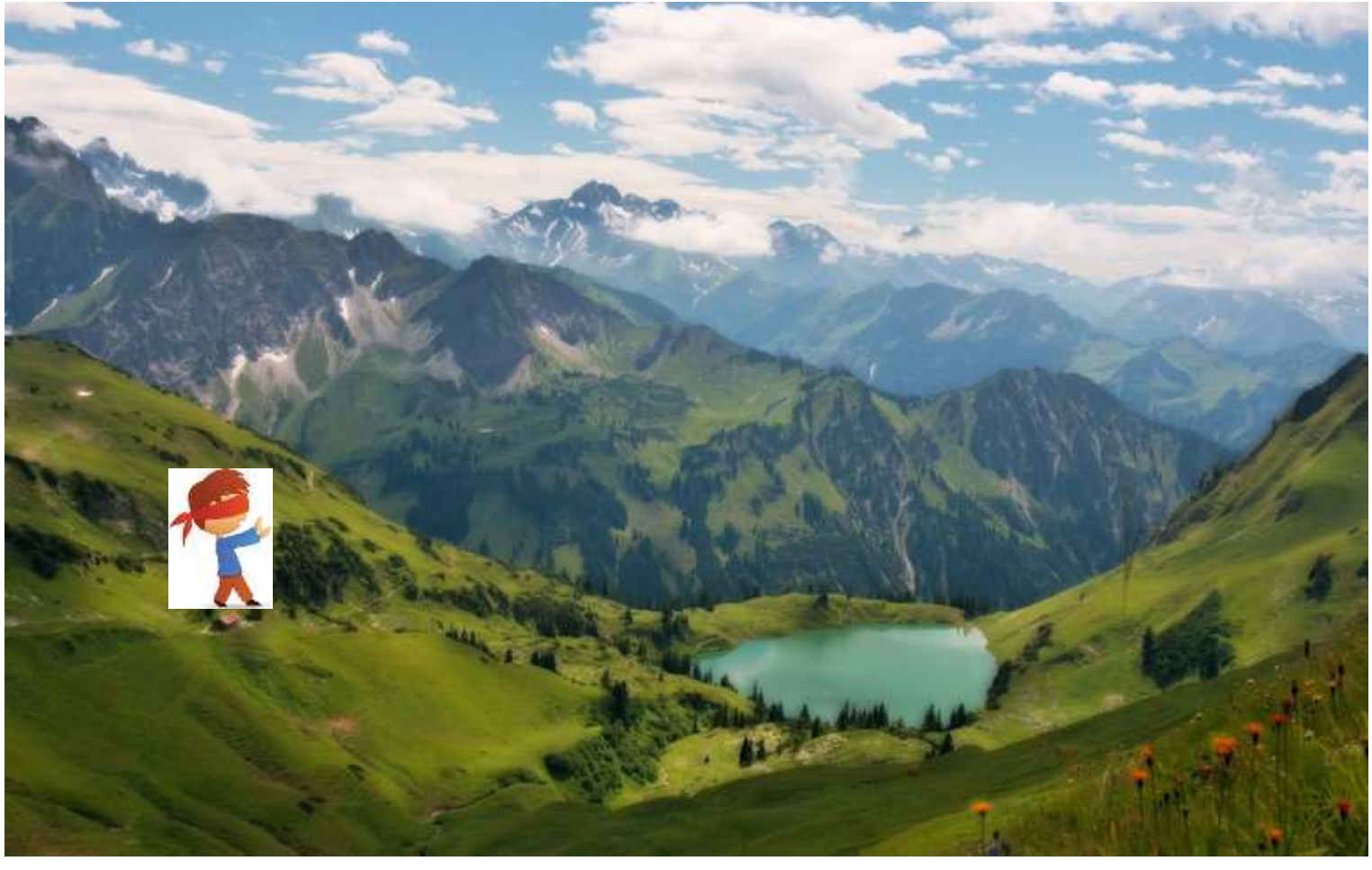
SVM

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + R(W)$$ Cu regularizare



Algoritm: Coborârea pe gradient



Algorimtul coborârii pe gradient (Python)

```
def GD(W0, X, goal, learningRate):
    perfGoalNotMet = true
    W = W0

    while perfGoalNotMet:
        gradient = eval_gradient(X, W)
        W_old = W
        W = W - learningRate * gradient
        perfGoalNotMet = sum(abs(W - W_old)) > goal
```

De la extragere “manuală” către învățare

vector ce descrie statistici despre imagine, e.g. bag-of-words

[32x32x3]

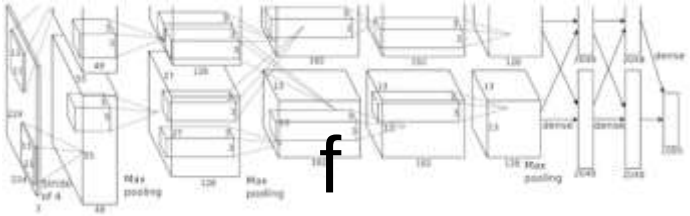
Extragere de trăsături

f

N numere ce indică scorurile pentru fiecare clasă

învățare

[32x32x3]

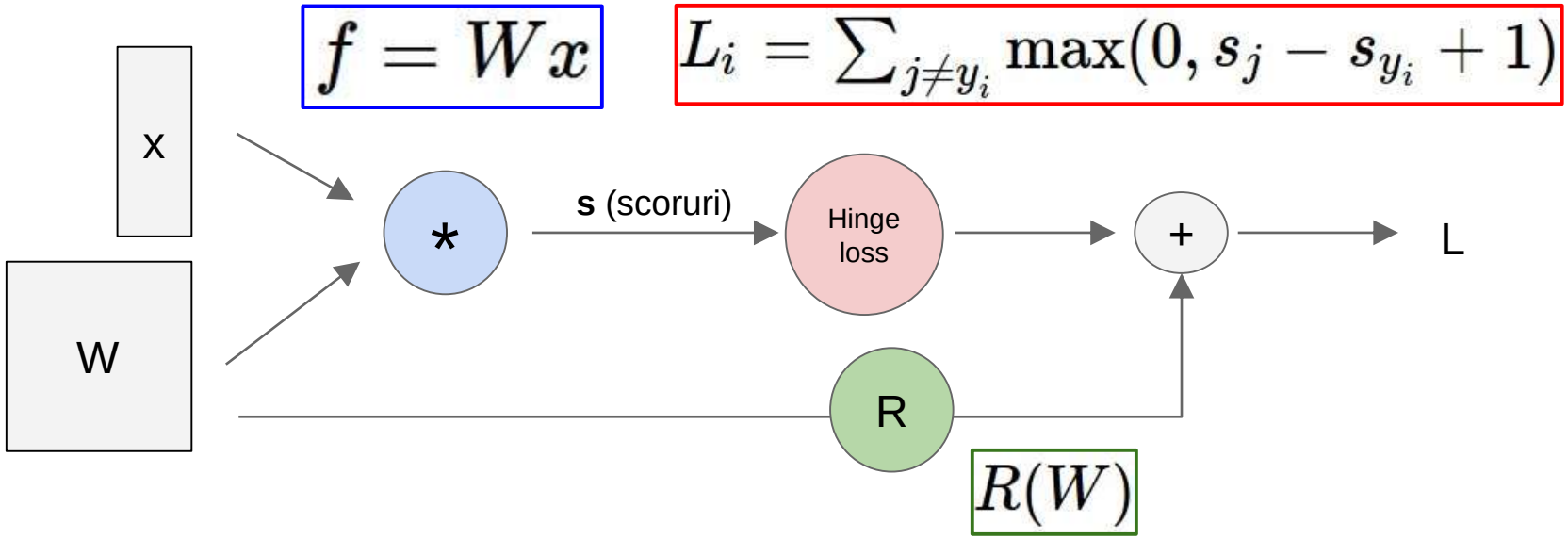


f

N numere ce indică scorurile pentru fiecare clasă

învățare “end-to-end”

Privim algoritmul ca un graf computațional



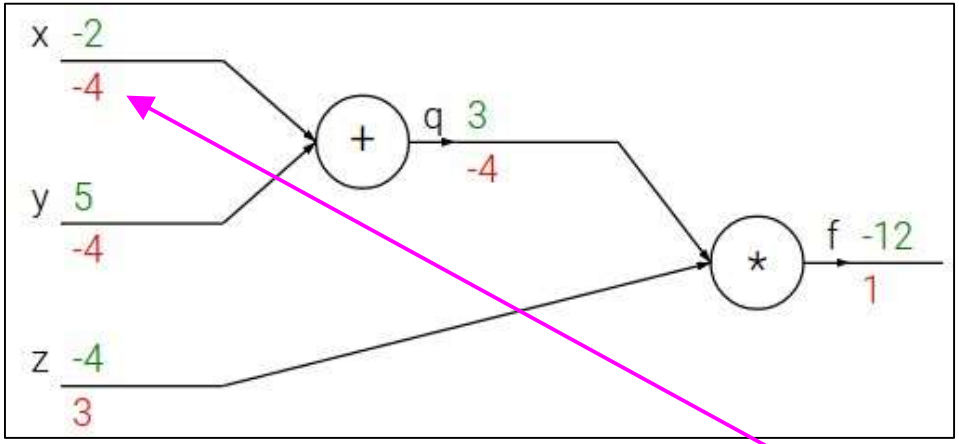
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

vrem: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

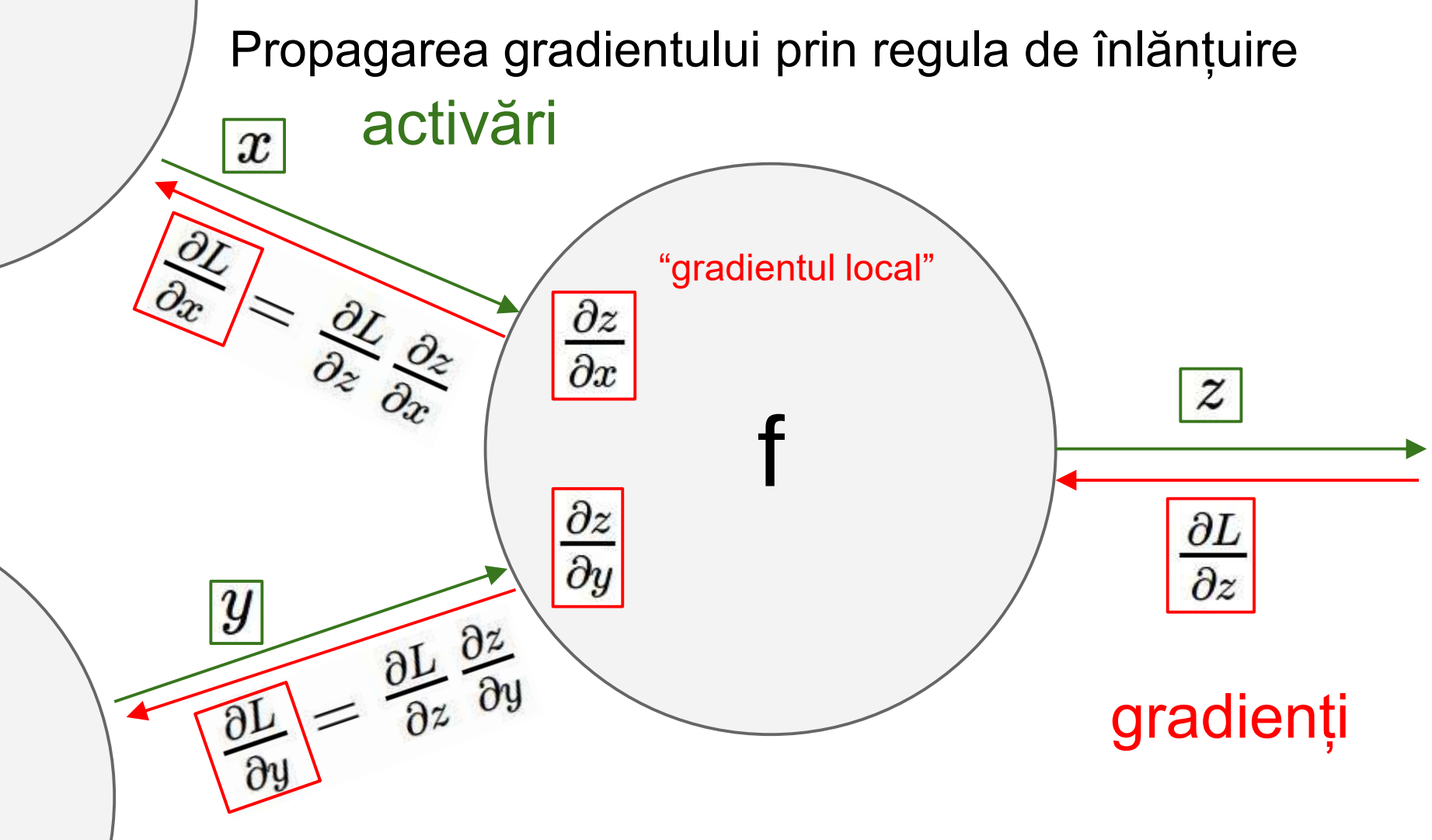


$\frac{\partial f}{\partial x}$

Regula de înlănțuire:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

Propagarea gradientului prin regula de înlănțuire



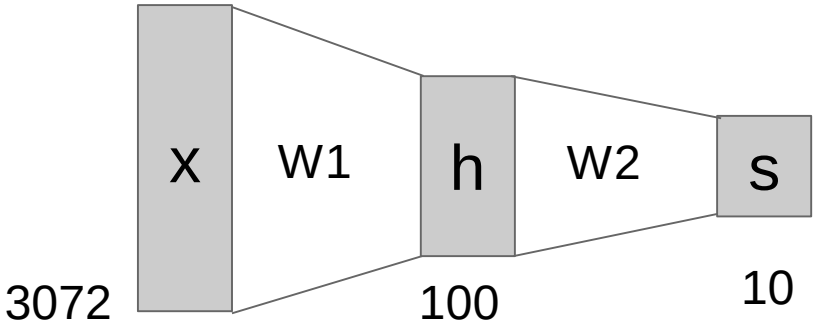
Din cursul trecut…

- Rețelele neuronale pot fi foarte mari: nici o speranță să scriem formula de mână pentru toți parameterii (folosim gradientul analitic)
- **Backpropagare** = aplicarea recursivă a regulii de înlănțuire (chain rule) de-a lungul unui graf computațional pentru calcularea gradienților parametrilor / intrărilor
- Implementările mențin o structură de graf în care nodurile implementează funcțiile **forward**() / **backward**()
- **forward**: calculează rezultatul unei operații și salvează în memorie intrările / rezultatele intermediare necesare la calcularea gradientului
- **backward**: aplicarea regulii de înlănțuire pentru calcularea gradientului funcției de pierdere în raport cu intrările

Rețele neuronale: fără paralela cu neurologia

(**Înainte**) Funcție liniară de scoring: $f = Wx$

(**Acum**) Rețea neuronală cu 2 nivele: $f = W_2 \max(0, W_1 x)$



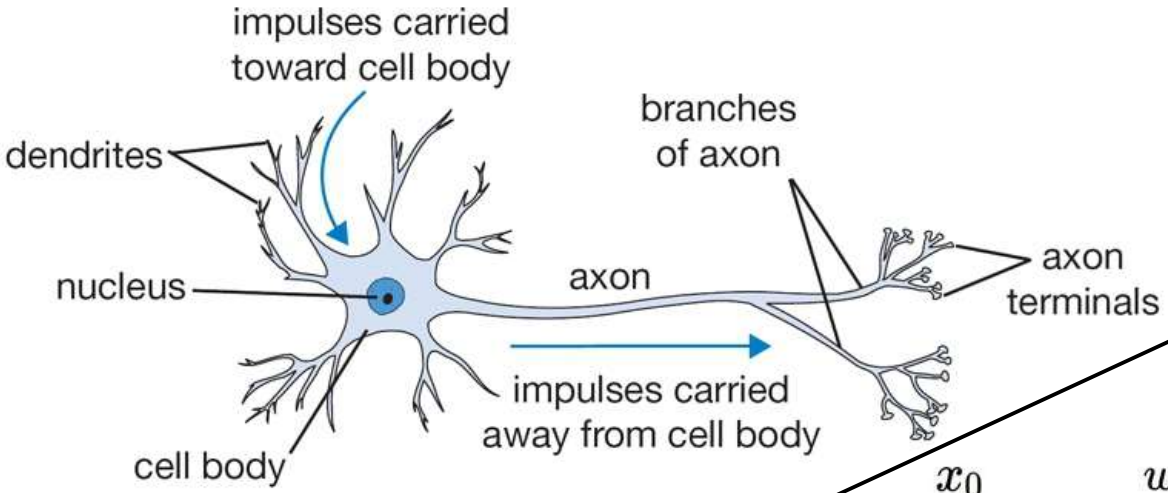
Rețele neuronale: fără paralela cu neurologia

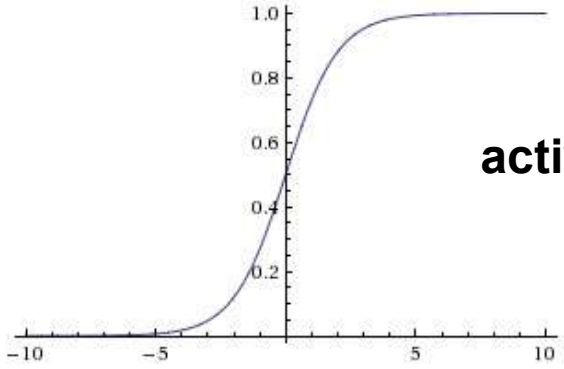
(**Înainte**) Funcție liniară de scoring: $f = Wx$

(**Acum**) Rețea neuronală cu 2 nivele: $f = W_2 \max(0, W_1 x)$

sau cu 3 nivele:

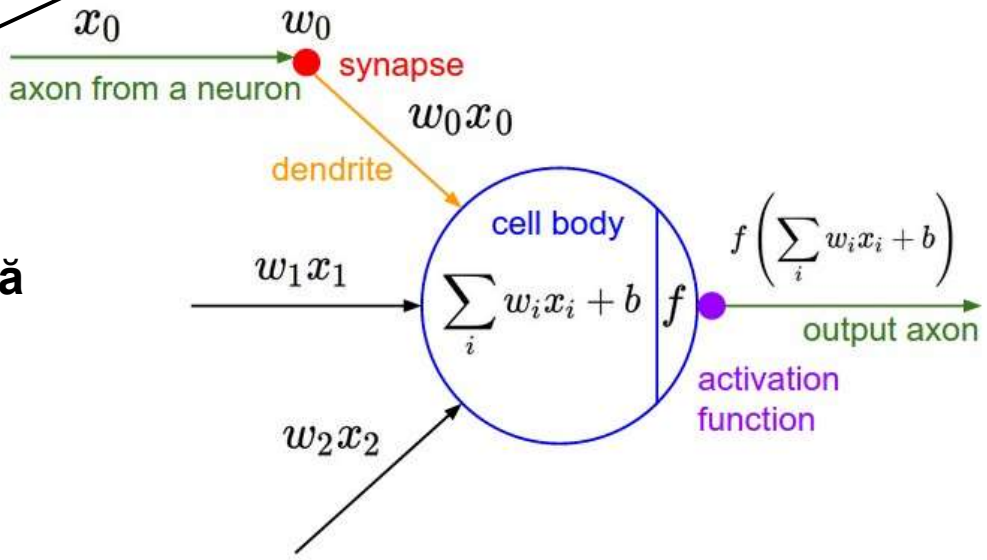
$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$





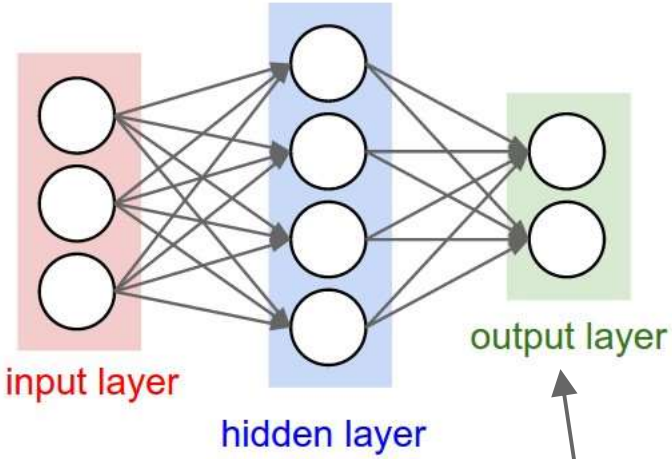
funcția de activare sigmoidă

$$\frac{1}{1+e^{-x}}$$

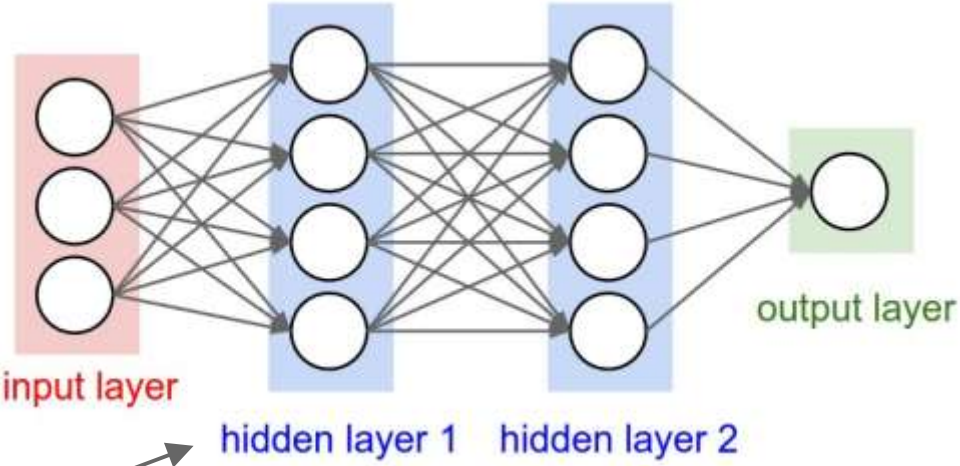


Arhitecturi de rețele neuronale

Rețea neuronală cu două straturi
(cu un singur strat ascuns)



Rețea neuronală cu trei straturi
(cu două straturi ascunse)

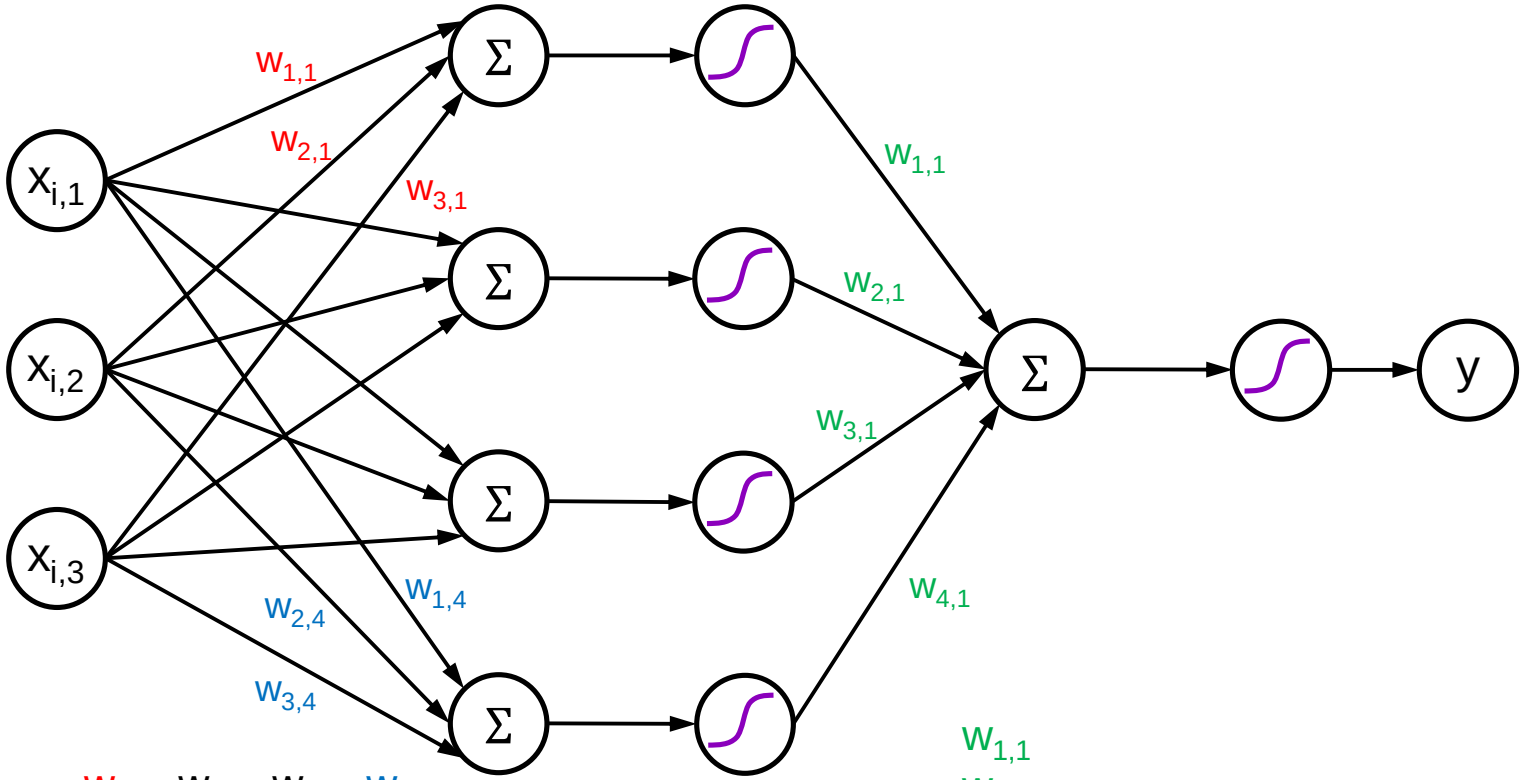


Straturi “fully-connected”

Antrenarea unei rețele cu două niveluri necesită ~11 linii de cod (Python)

```
X = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]])
Y = np.array([[0,1,1,0]]).T

W0 = 2 * np.random.random((3,4)) - 1
W1 = 2 * np.random.random((4,1)) - 1

for i in range(5000):

    # forward pass
    l1 = 1 / (1 + np.exp(-np.matmul(X, W0)))
    l2 = 1 / (1 + np.exp(-np.matmul(l1, W1)))

    # backward pass
    delta_l2 = (Y - l2) * (l2 * (1 - l2))
    delta_l1 = np.matmul(delta_l2, W1.T) * (l1 * (1 - l1))

    # gradient descent
    W1 = W1 + np.matmul(l1.T, delta_l2)
    W0 = W0 + np.matmul(X.T, delta_l1)
```

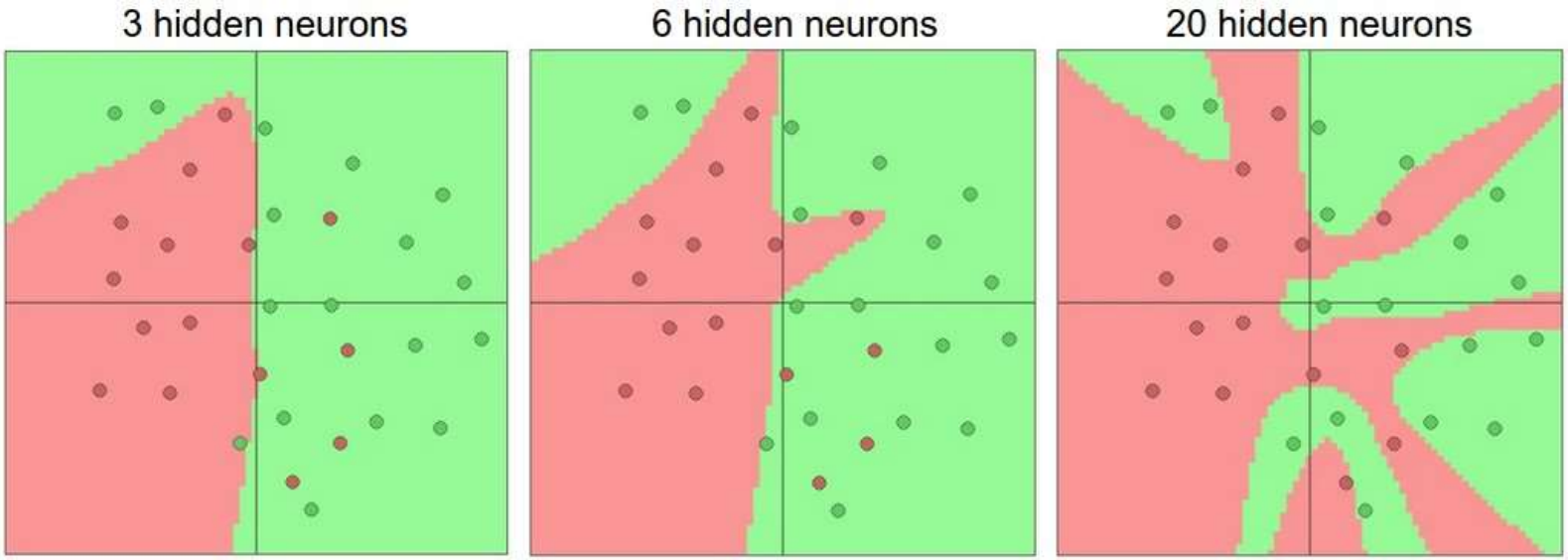
Arhitectura rețelei cu două niveluri implementată anterior



$$W_0 = \begin{matrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} \end{matrix}$$

$$W_1 = \begin{matrix} w_{1,1} \\ w_{2,1} \\ w_{3,1} \\ w_{4,1} \end{matrix}$$

Alegerea numărului de straturi și a numărului de neuroni



mai mulți neuroni = mai multă capacitate

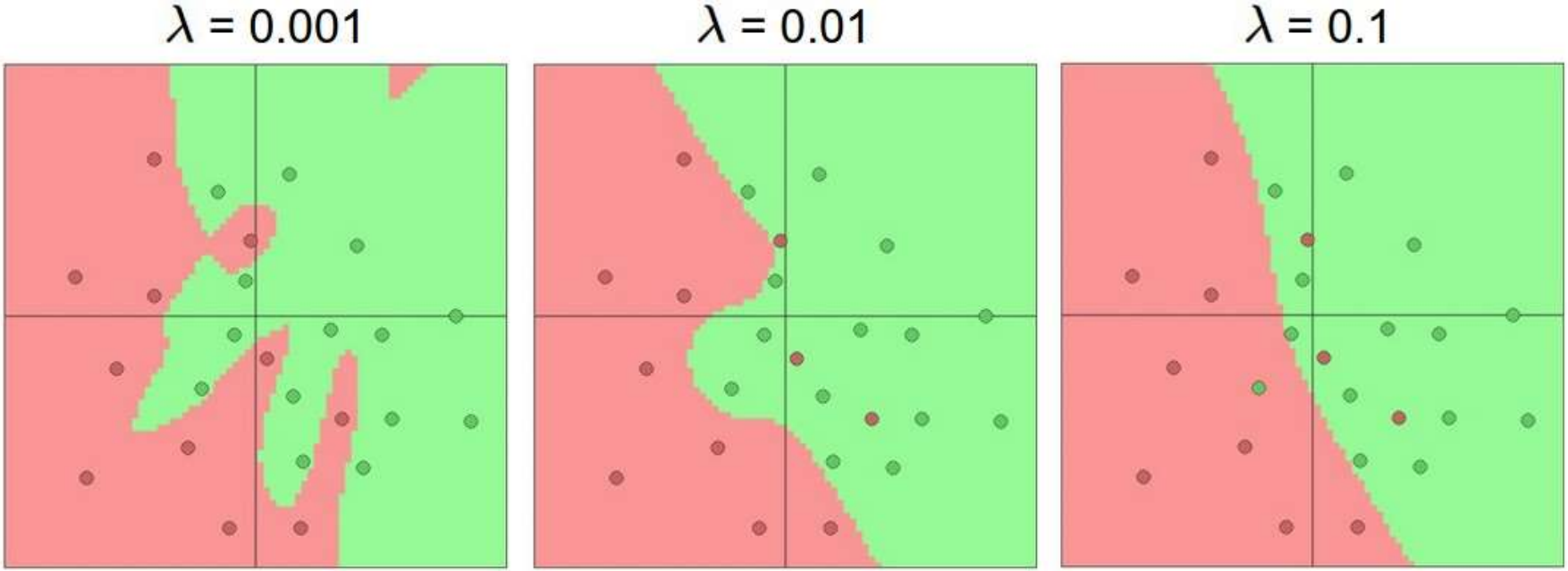
Rețele neuronale sunt funcții universale de aproximare

- **Teorema Aproximării Universale:**

O rețea neuronală de tip feed-forward cu un strat ascuns având un număr finit de neuroni poate aproxima orice funcție continuă definită pe un subset compact din $\mathbb{R}^n$.

- Deși rețelele neuronale cu două straturi (un strat ascuns) sunt funcții universale de aproximare, lățimea (numărul de perceptroni) acestor rețele poate fi exponențial de mare.
- În practică, preferăm rețele mai adânci (cu mai multe straturi)

Alegerea parametrului de regularizare



Sfat practic: În general este mai bine să folosim regularizare mai puternică în loc să reducem capacitatea modelului

Alegerea arhitecturii potrivite

- Aranjăm neuronii în straturi fully-connected
- La nivel de implementare, abstractizarea unui strat ne permite să utilizăm cod vectorial (e.g. înmulțirea matricilor)
- Performanța crește cu cât arhitectura rețelei este mai adâncă (deep), i.e. are mai multe straturi (dar trebuie să folosim o regularizare mai puternică)

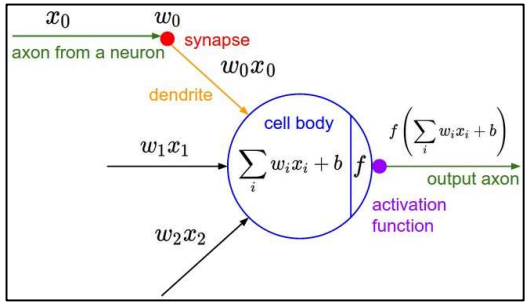
Antrenarea rețelelor neuronale

Scurt istoric

Mașina **Mark I Perceptron** a fost prima implementare a algoritmului perceptronului.

Mașina era conectată la o camera cu 20×20 fotocelule de sulfat de cadmiu pentru a produce o imagine cu 400 de pixeli.

Folosită pentru a recunoaște litere din alfabet.

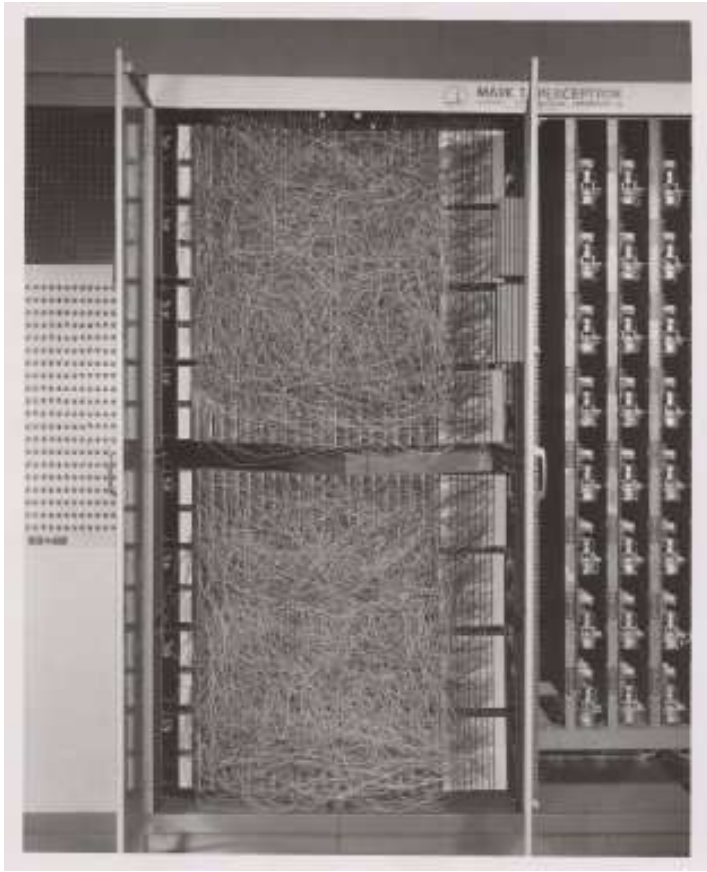


$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

Regula de actualizare

$$w_i(t+1) = w_i(t) + \alpha(d_j - y_j(t))x_{j,i},$$

Frank Rosenblatt, ~1957: Perceptron



Scurt istoric

Mașina **ADALINE** folosea rezistoare cu memorie capabile să execute operații logice și să stocheze informații.

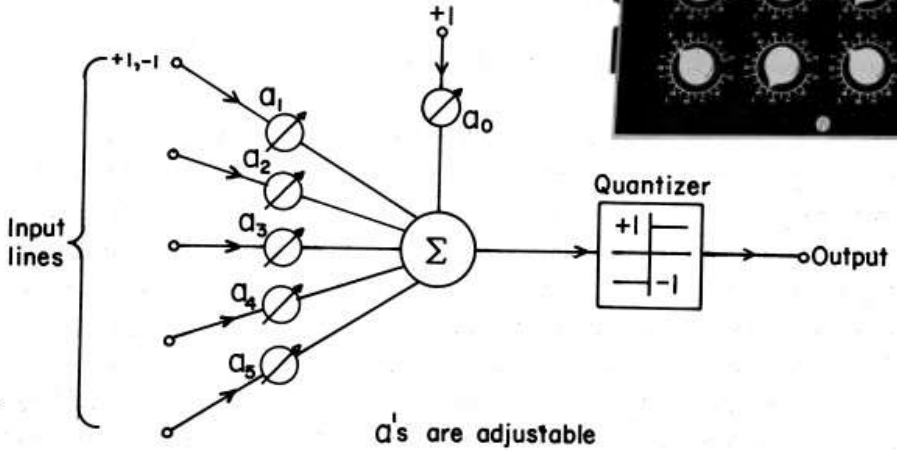
Funcția de pierdere (suma pătratelor erorilor)

$$\frac{1}{2}\sum_i (d^i - y^i)^2, \text{ unde } y^i = (x^i)^T w + b$$

Regula de actualizare

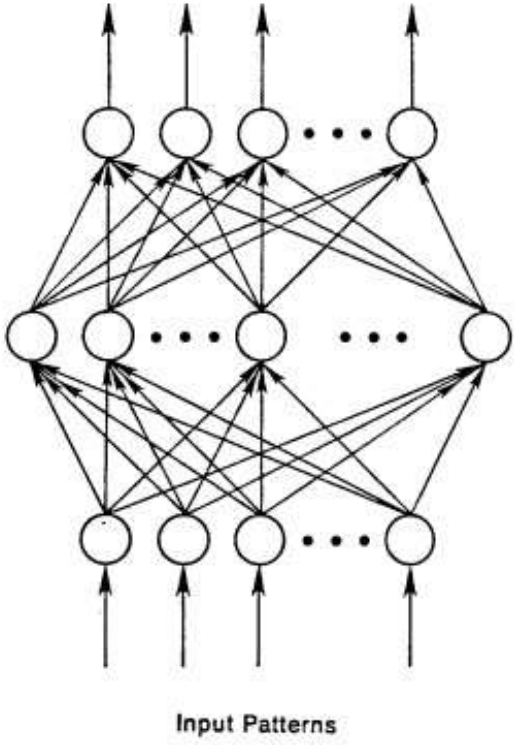
$$w^{k+1} = w^k + \mu \sum_{i=1}^{m} (d^i - y^i) x^i$$

$$b^{k+1} = b^k + \mu \sum_{i=1}^{m} (d^i - y^i)$$



Widrow and Hoff, ~1960: Adaline

Scurt istoric



Formule matematice
mai ușor de înțeles

To be more specific, then, let

$$E_p = \frac{1}{2}\sum_j (t_{pj} - o_{pj})^2 \qquad (2)$$

be our measure of the error on input/output pattern p and let $E = \sum E_p$ *be our overall measure of the error. We wish to show that the delta rule implements a gradient descent in E when the units are linear. We will proceed by simply showing that*

$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} i_{pi},$$

which is proportional to $\Delta_p w_{ji}$ *as prescribed by the delta rule. When there are no hidden units it is straightforward to compute the relevant derivative. For this purpose we use the chain rule to write the derivative as the product of two parts: the derivative of the error with respect to the output of the unit times the derivative of the output with respect to the weight.*

$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial w_{ji}}. \qquad (3)$$

The first part tells how the error changes with the output of the j*th unit and the second part tells how much changing* w_{ji} *changes that output. Now, the derivatives are easy to compute. First, from Equation 2*

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj}) = -\delta_{pj}. \qquad (4)$$

Not surprisingly, the contribution of unit u_j *to the error is simply proportional to* δ_{pj}*. Moreover, since we have linear units,*

$$o_{pj} = \sum_i w_{ji} i_{pi}, \qquad (5)$$

from which we conclude that

$$\frac{\partial o_{pj}}{\partial w_{ji}} = i_{pi}.$$

Thus, substituting back into Equation 3, we see that

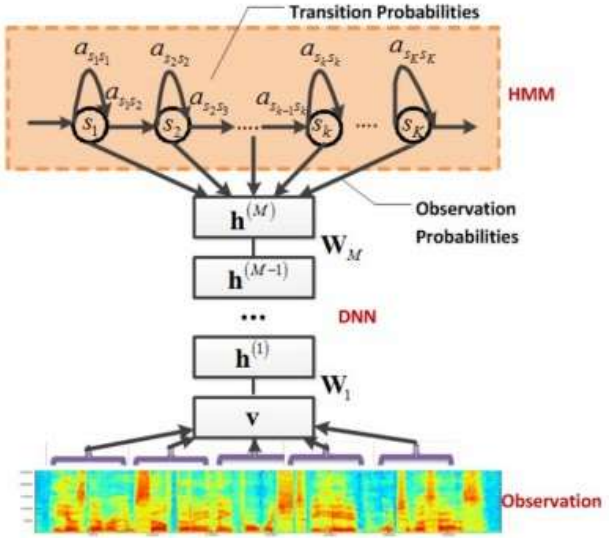
$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} i_i \qquad (6)$$

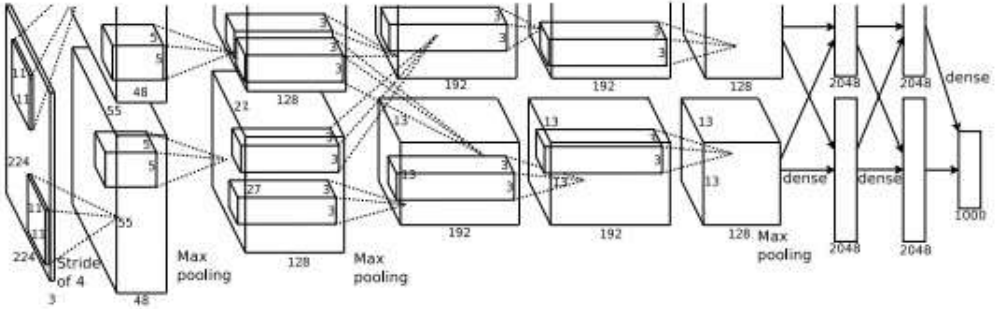
Hinton et al. 1986: Algoritmul de propagare înapoi a erorii (backpropagation)

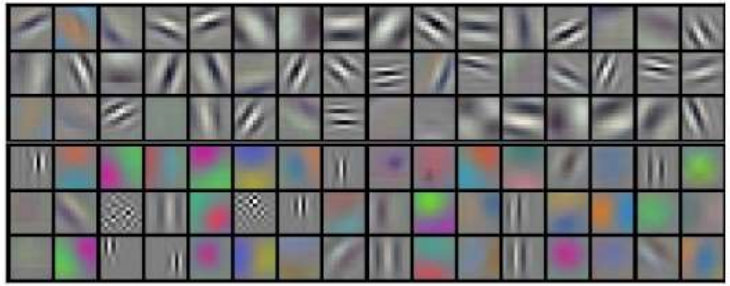
Primele rezultate semnificative bazate pe învățare cu modele deep

Context-Dependent Pre-trained Deep Neural Networks for Large Vocabulary Speech Recognition
George Dahl, Dong Yu, Li Deng, Alex Acero, 2010

ImageNet classification with deep convolutional neural networks
Alex Krizhevsky, Ilya Sutskever, Geoffrey E Hinton, 2012







Antrenarea rețelelor neuronale: privire de ansamblu

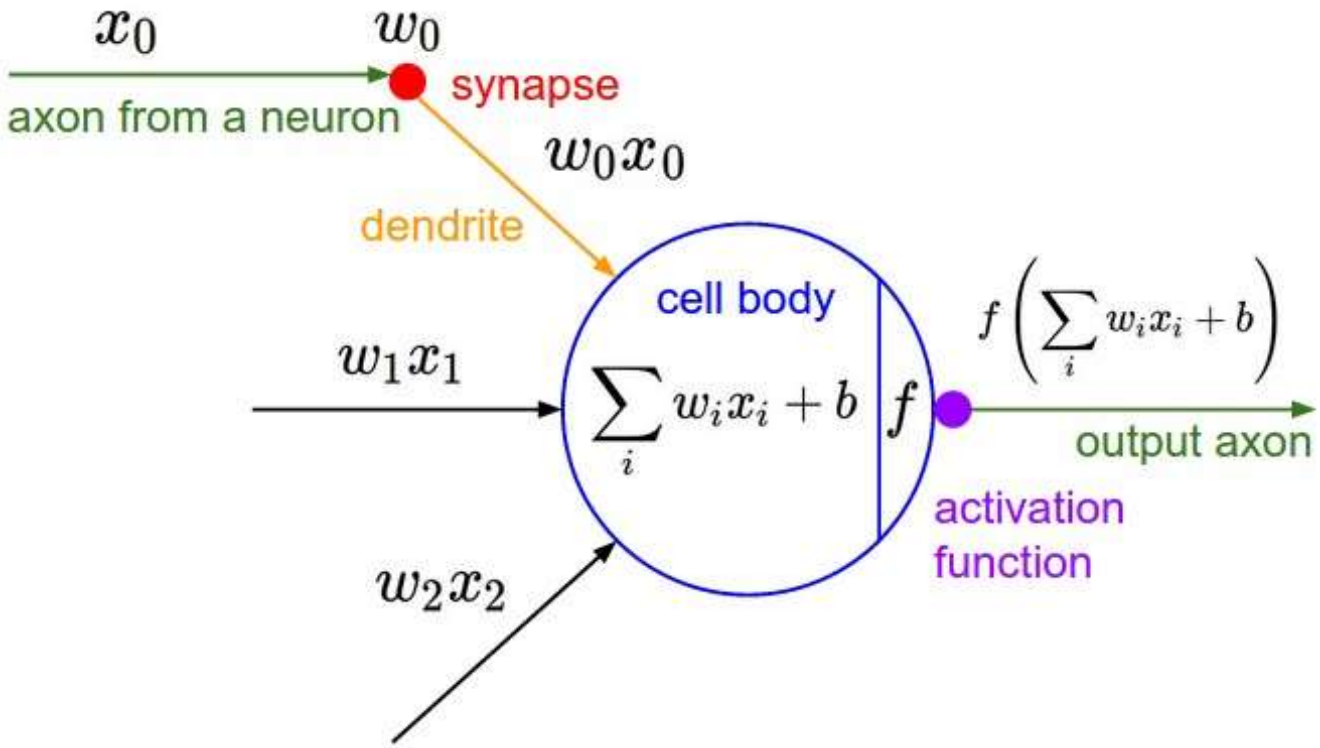
1. Ce trebuie să stabilim la început (o dată)
Funcțiile de activare, preprocesarea, inițializarea ponderilor, regularizarea, verificarea gradientului

2. Ce ține de dinamica antrenării
Asistarea procesului de învățare, actualizarea parametrilor, optimizarea hiperparametrilor

3. Evaluare
Ansamble de modele

Funcții de activare

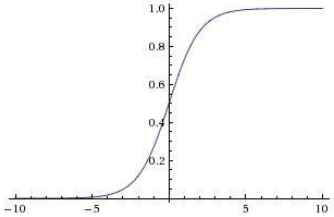
Funcții de activare



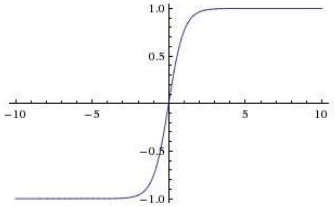
Funcții de activare

sigmoidă

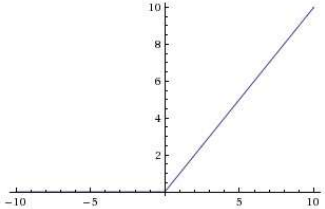
$$\sigma(x) = 1/(1 + e^{-x})$$



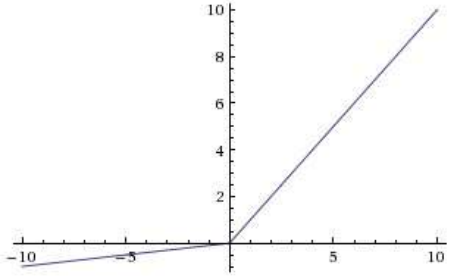
tanh tanh(x)



ReLU max(0,x)

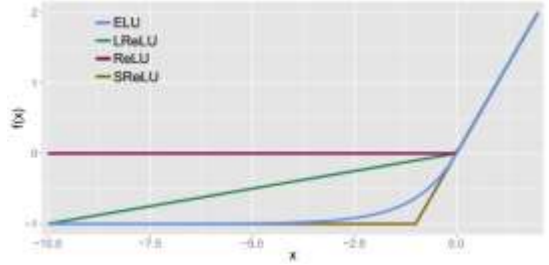


Leaky ReLU
max(0.1x, x)

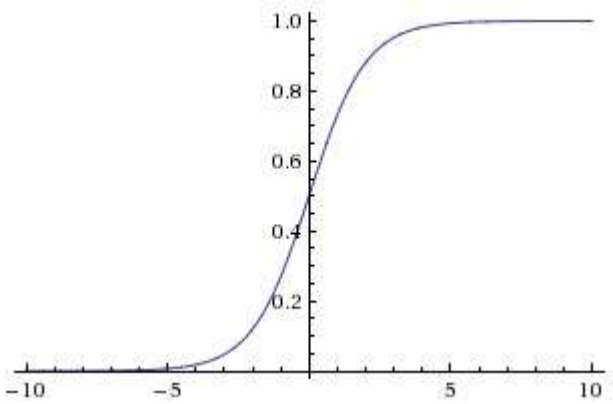


Maxout $\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU $$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$



Funcții de activare



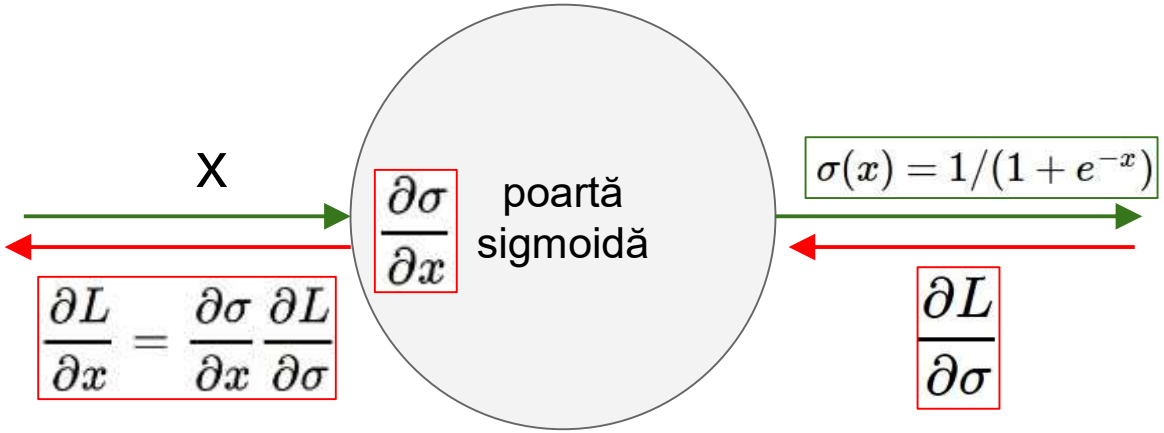
sigmoidă

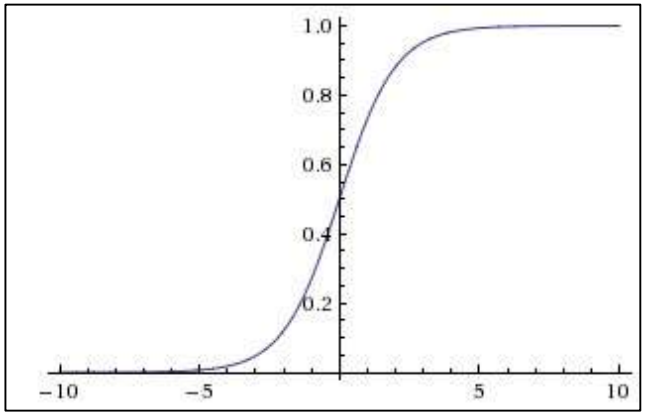
$$\sigma(x) = 1/(1 + e^{-x})$$

- Aduce numerele în intervalul [0,1]
- Populară din punct de vedere istoric deorece are interpretarea biologică a saturării ratei de activare a unui neuron

3 probleme:

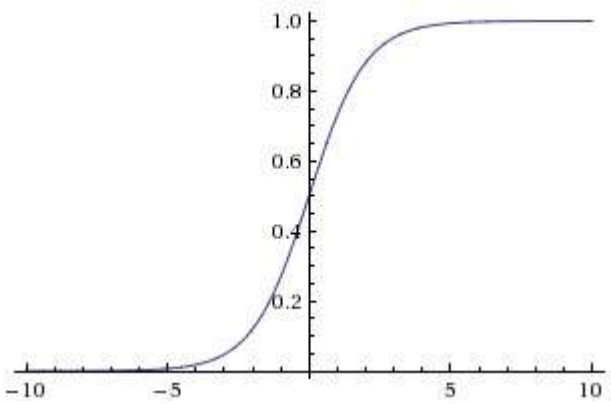
1. Neuronii saturați “omoară” gradienții





Ce se întâmplă când x = -10?
Ce se întâmplă când x = 0?
Ce se întâmplă când x = 10?

Funcții de activare



sigmoidă

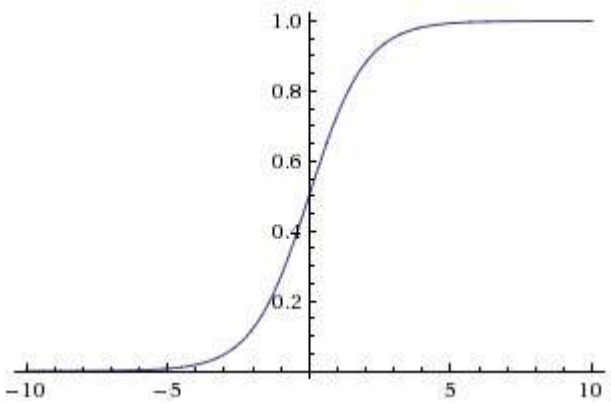
$$\sigma(x) = 1/(1 + e^{-x})$$

- Aduce numerele în intervalul [0,1]
- Populară din punct de vedere istoric deorece are interpretarea biologică a saturării ratei de activare a unui neuron

3 probleme:

1. Neuronii saturați “omoară” gradienții
2. Output-ul funcției sigmoide nu este centrat în zero

Funcții de activare



sigmoidă

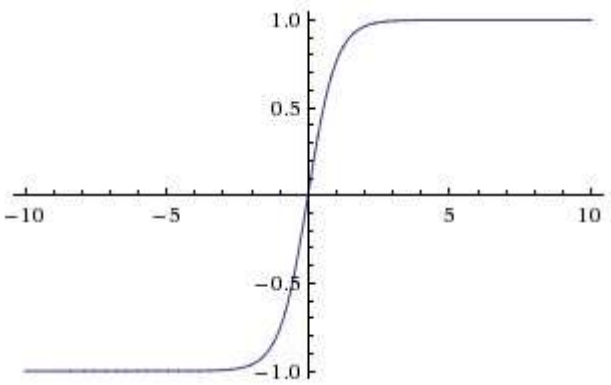
$$\sigma(x) = 1/(1 + e^{-x})$$

- Aduce numerele în intervalul [0,1]
- Populară din punct de vedere istoric deorece are interpretarea biologică a saturării ratei de activare a unui neuron

3 probleme:

1. Neuronii saturați “omoară” gradienții
2. Output-ul funcției sigmoide nu este centrat în zero
3. exp() are un cost computațional ridicat

Funcții de activare



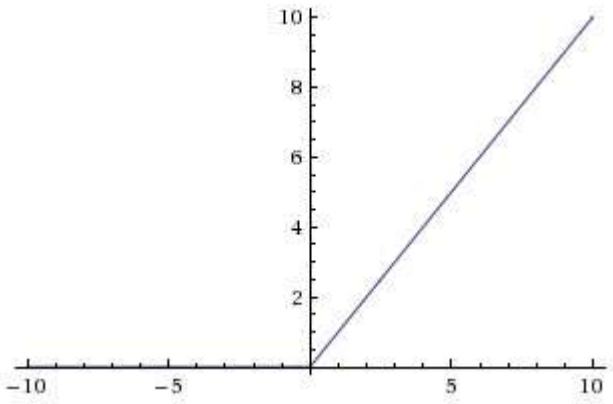
tanh(x)

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

- Aduce numerele în intervalul [-1,1]
- De medie zero (bine)
- Încă omoară gradienții atunci când se saturează :(

[LeCun et al., 1991]

Funcții de activare



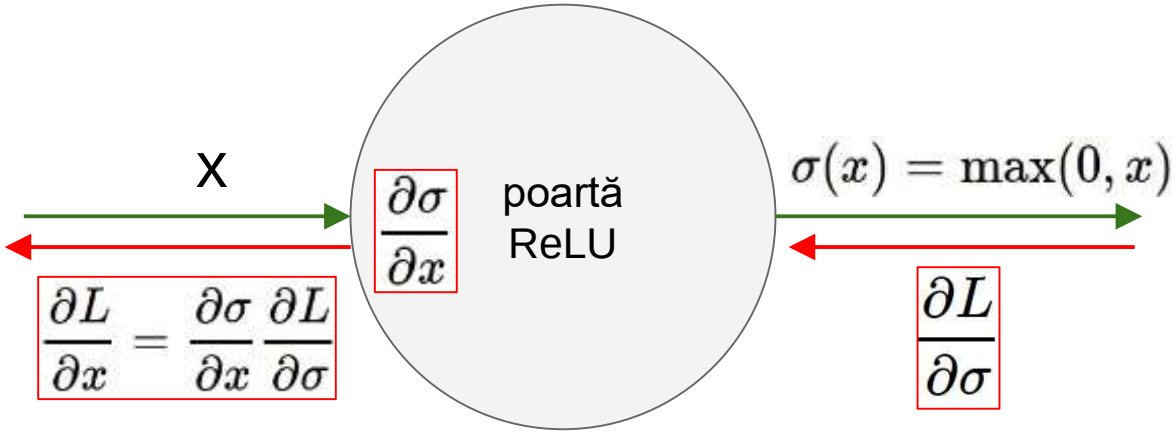
ReLU
(Rectified Linear Unit)

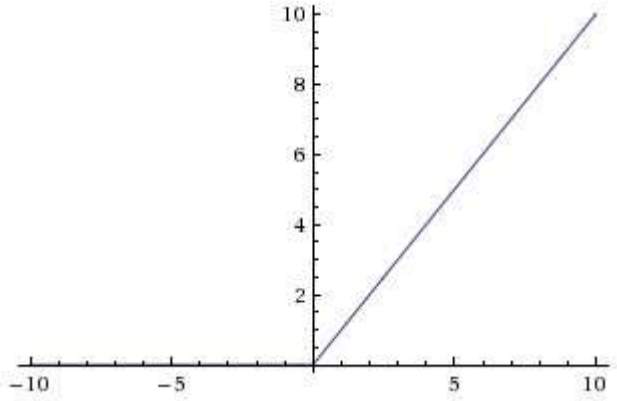
f(x) = max(0,x)

- Nu se saturează (în partea pozitivă)
- Foarte eficient computațional
- În practică, converge mult mai rapid decât sigmoida/tanh (e.g. 6x)

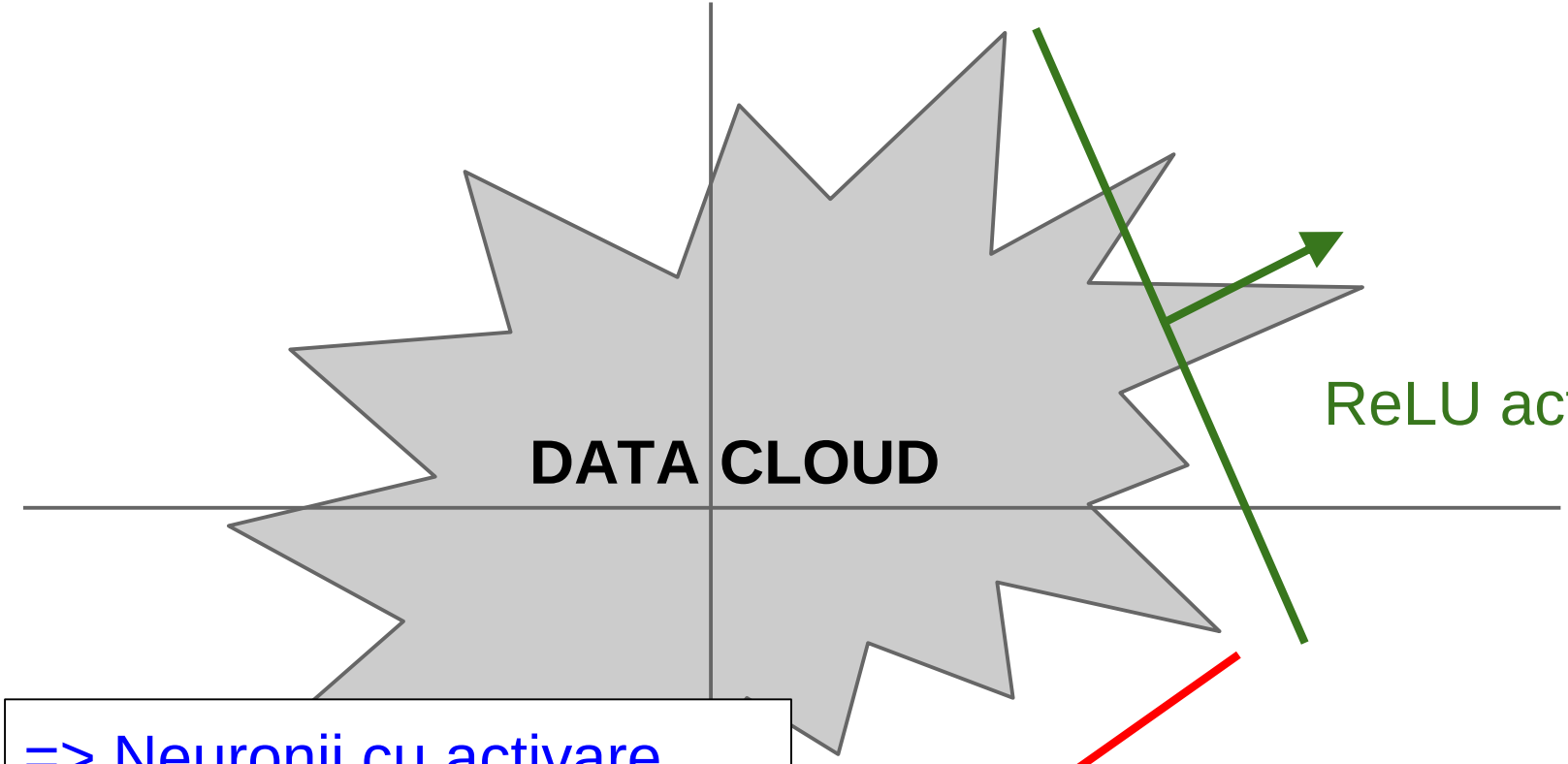
- Output-ul nu are media zero
- O situație neplăcută (atunci când x < 0, gradientul este 0)

[Krizhevsky et al., 2012]





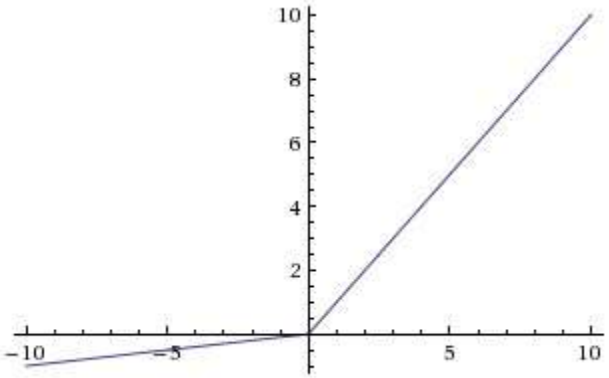
Ce se întâmplă când x = -10?
Ce se întâmplă când x = 0?
Ce se întâmplă când x = 10?



=> Neuronii cu activare ReLU se inițializează cu bias ușor pozitiv (e.g. 0.01)

ReLU inactiv => nu se actualizează niciodată

Funcții de activare



- Foarte eficient computațional
- În practică, converge mult mai rapid decât sigmoida/tanh (e.g. 6x)
- **Nu se saturează**

Leaky ReLU

$$f(x) = \max(0.01x, x)$$

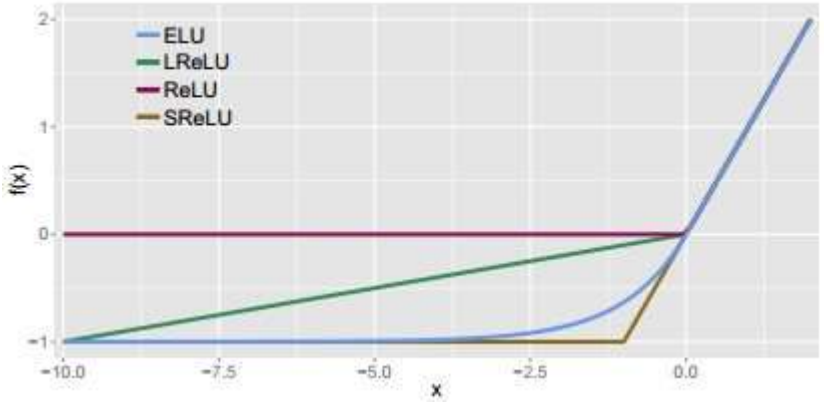
Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

[Mass et al., 2013]
[He et al., 2015]

Funcții de activare

Exponential Linear Units (ELU)



- Toate beneficiile ReLU
- Nu se saturează
- Output aproape de medie zero

- Implică calculul exp()

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\left(\exp(x) - 1\right) & \text{if } x \leq 0 \end{cases}$$

[Clevert et al., 2015]

Neuronul cu funcție de activare Maxout

- Nu are forma generală a produsului scalar => non-liniaritate
- Generalizează ReLU și Leaky ReLU
- Liniar pe intervale! Nu se saturează!

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Problemă: se dublează numărul de parametrii/neuroni :(

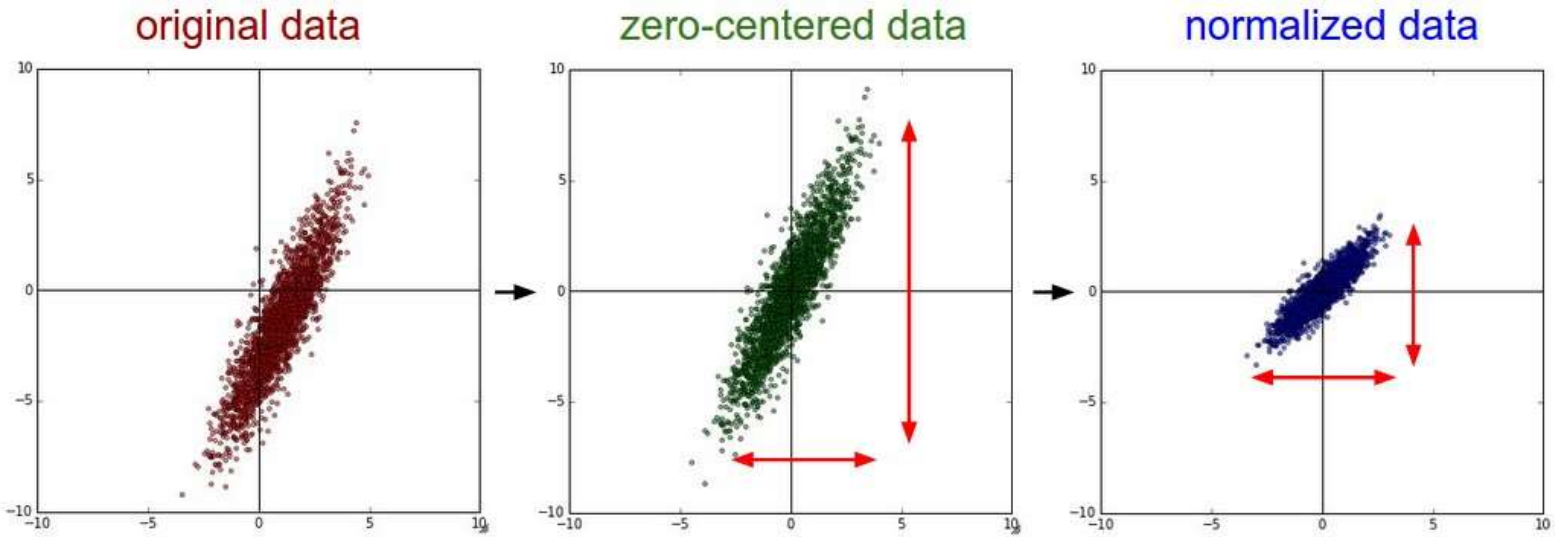
[Goodfellow et al., 2013]

Ce funcții de activare în practică?

- Utilizăm ReLU. Trebuie să avem grijă cu rata de învățare
- Putem încerca Leaky ReLU / Maxout / ELU
- Putem încerca tanh (fără așteptări prea mari)
- Evităm pe cât posibil sigmoida

Preprocesarea datelor

Preprocesarea datelor




```
X = X - np.mean(X, axis=0, keepdims=True)
```

```
X = X / np.std(X, axis=0, keepdims=True))
```

(X este o matrice [NxD], câte un exemplu pe linie)

Pentru imagini este suficient să centrăm datele

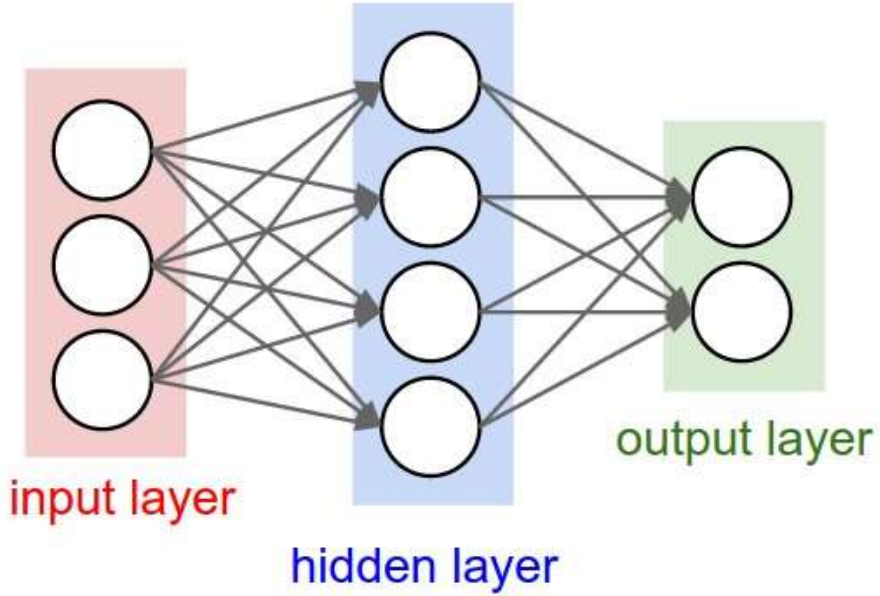
Considerând setul de date CIFAR-10 cu imagini de [32,32,3]

- Scădem imaginea medie (e.g. AlexNet)
  (imaginea medie = matrice [32,32,3])
- Scădem media pe fiecare canal (e.g. VGGNet)
  (media pe fiecare canal = 3 numere)

Nu este o practică obligatorie să normalizăm imaginile

Inițializarea ponderilor

Ce se întâmplă dacă inițializăm W=0?



O primă idee: Inițializăm cu numere aleatorii aproape de zero

```
W = np.random.normal(0, 0.01, (N,D))
```

(distribuție normală de medie zero și dispersie 0.01)

Funcționează ~bine pentru rețele mici, dar poate conduce la distribuții neomogene ale funcțiilor de activare din straturile unei rețele.
Aproape toți neuronii se saturează complet, fie spre -1 fie spre 1. Gradienții vor fi zero.

A doua abordare: Inițializare Xavier

Probleme cu alegerea ponderilor inițiale:
- Dacă sunt prea mici, semnalul care se propragă în rețea se diminuează cu fiecare nivel și devine prea mic pentru a fi util
- Dacă sunt prea mari, semnalul care se propragă în rețea crește cu fiecare nivel până când devine prea mare pentru a fi util

Inițializarea Xavier ne asigură că ponderile au magnitudinea potrivită, păstrând semnalul într-un interval rezonabil.
Ponderile inițiale provin dintr-o distribuție normală de medie 0 și o dispresie dată de numărul de perceptroni de pe stratul anterior / posterior:

$$\mathrm{Var}(W) = \frac{2}{n_{\mathrm{in}} + n_{\mathrm{out}}}$$

[Glorot and Bengio, 2010]

Normalizarea Batch

Vrem activări normale de medie 0 și dispersie 1?
Le transformăm a.î. să devină așa.

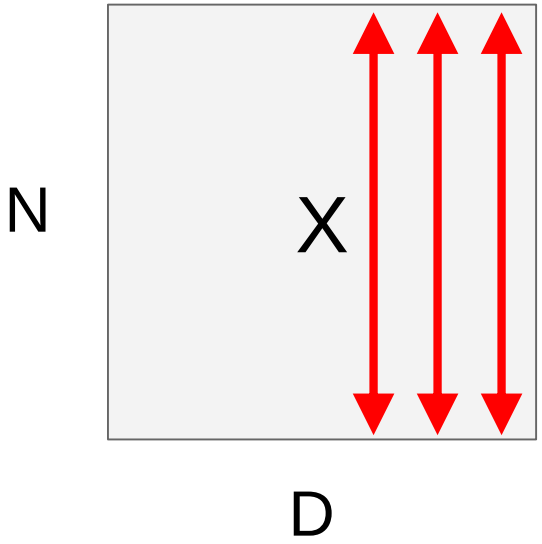
Considerăm activările pe un anumit strat pentru un mini-batch. Pentru a transforma fiecare dimensiune aplicăm:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

[Ioffe and Szegedy, 2015]

Normalizarea Batch

Vrem activări normale de medie 0 și dispersie 1?
Le transformăm a.î. să devină așa.



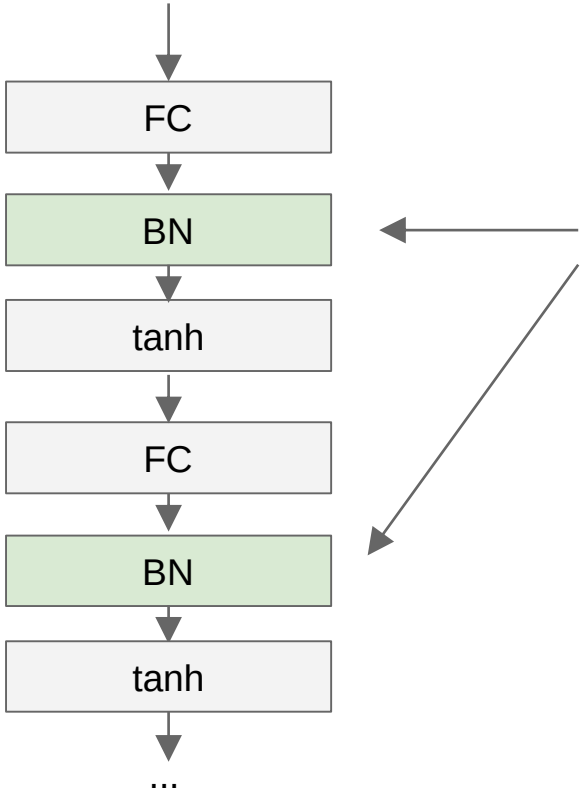
1. Calculăm media empirică și dispersia pentru fiecare dimensiune (independent)

2. Normalizăm

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

[Ioffe and Szegedy, 2015]

Normalizarea Batch



Se inserează de obicei după straturile “fully connected” sau după cele convoluționale, înainte de non-liniarități.

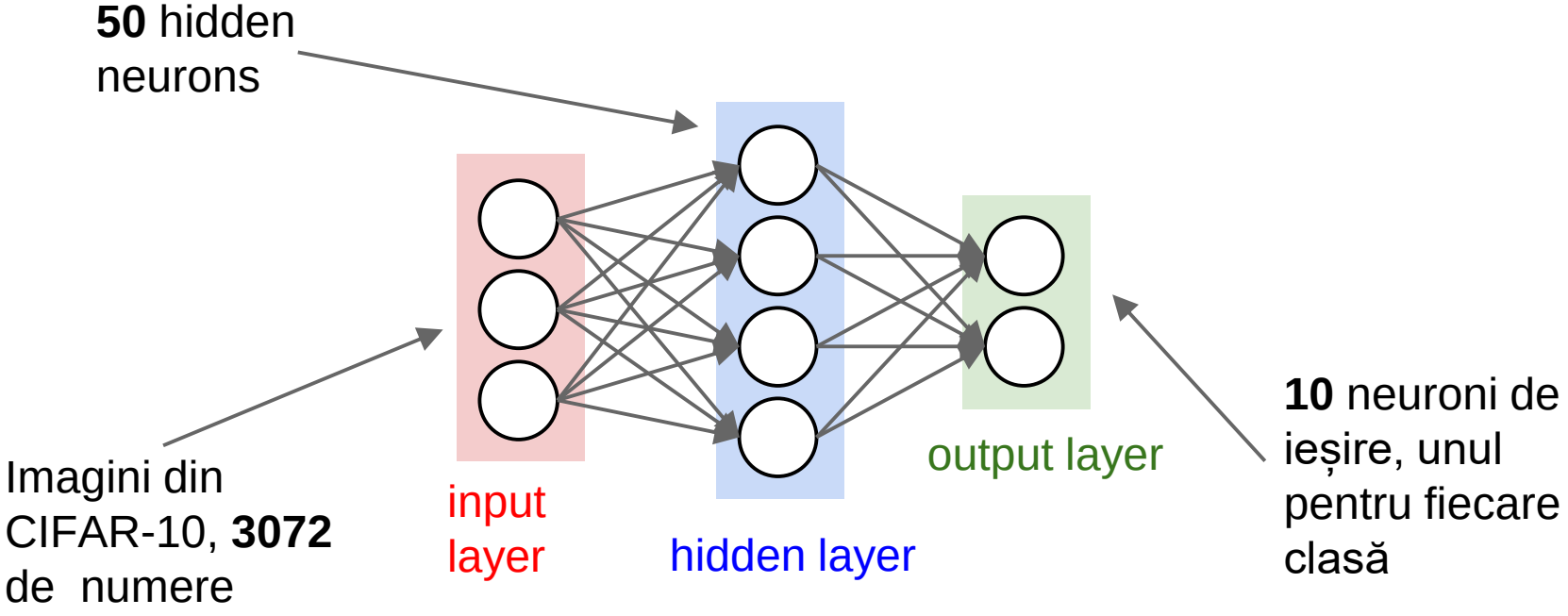
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

[Ioffe and Szegedy, 2015]

Asistarea procesului de învățare

Alegerea arhitecturii potrivite

Începem cu un strat ascuns de 50 de neuroni, apoi mărim gradual capacitatea rețelei

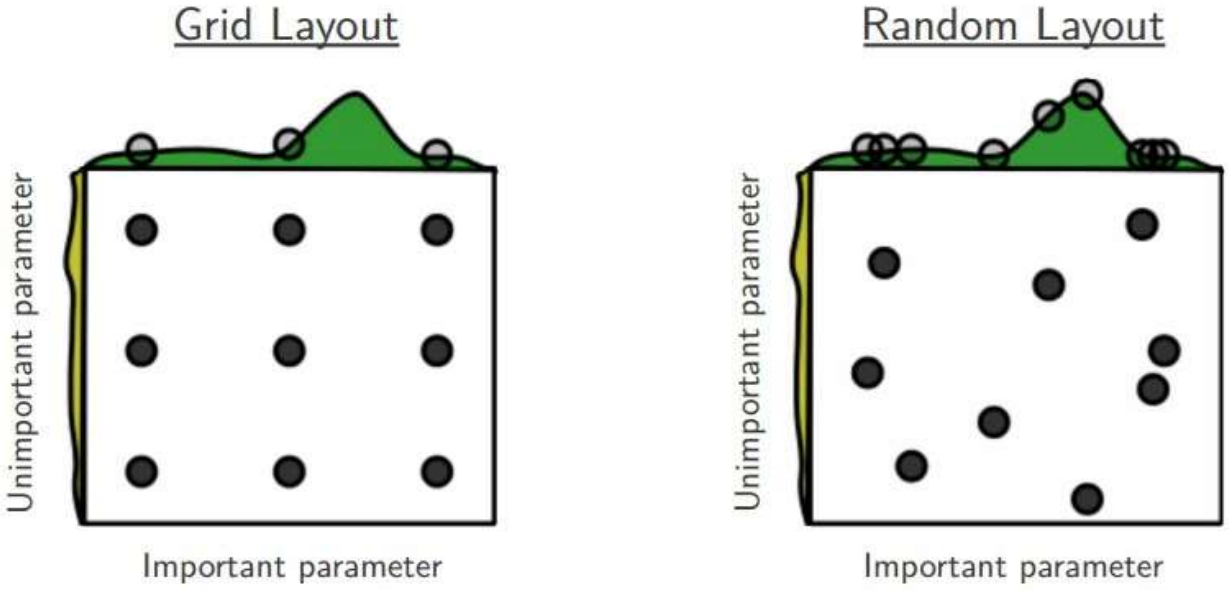


Sfaturi practice

1. Dezactivăm regularizarea și verificăm dacă valoarea funcției de pierdere este rezonabilă (~2.5 pentru 10 clase este ok)
2. Când adăugăm regularizare, valoarea funcției de pierdere ar trebui să crească, e.g. 3.2
3. Ne asigurăm că putem face overfitting pe o parte mică din setul de antrenare (e.g. 20 de exemple)

Optimizarea hiperparametrilor

Strategii de căutare: aleator versus grid



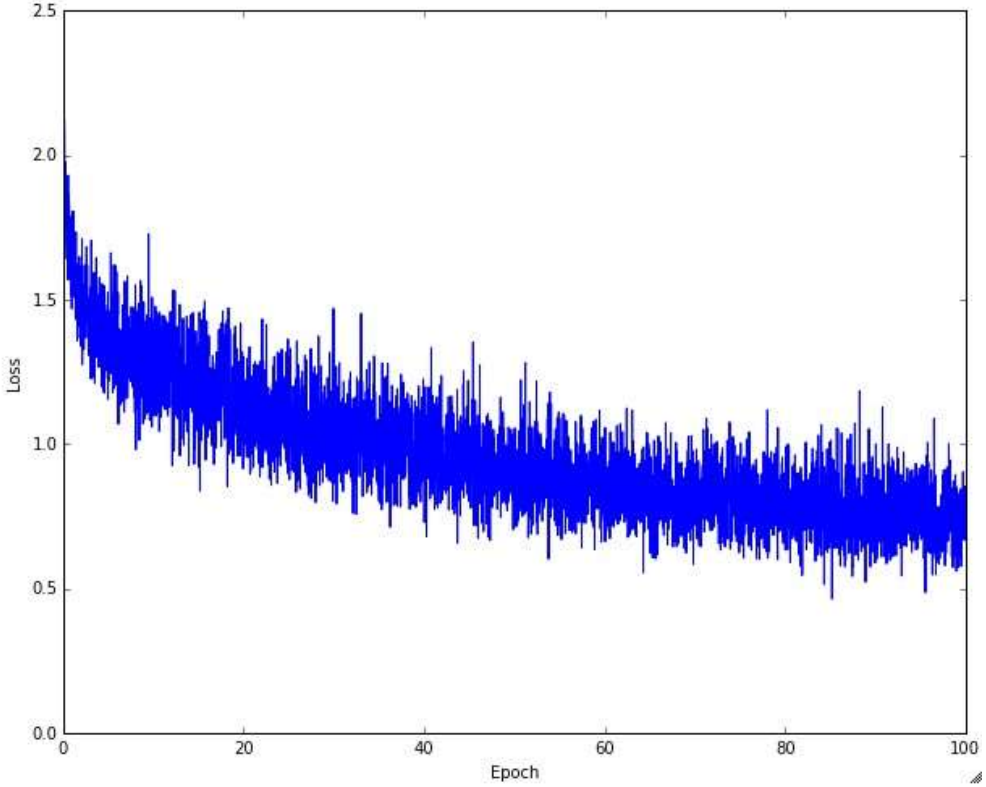
Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

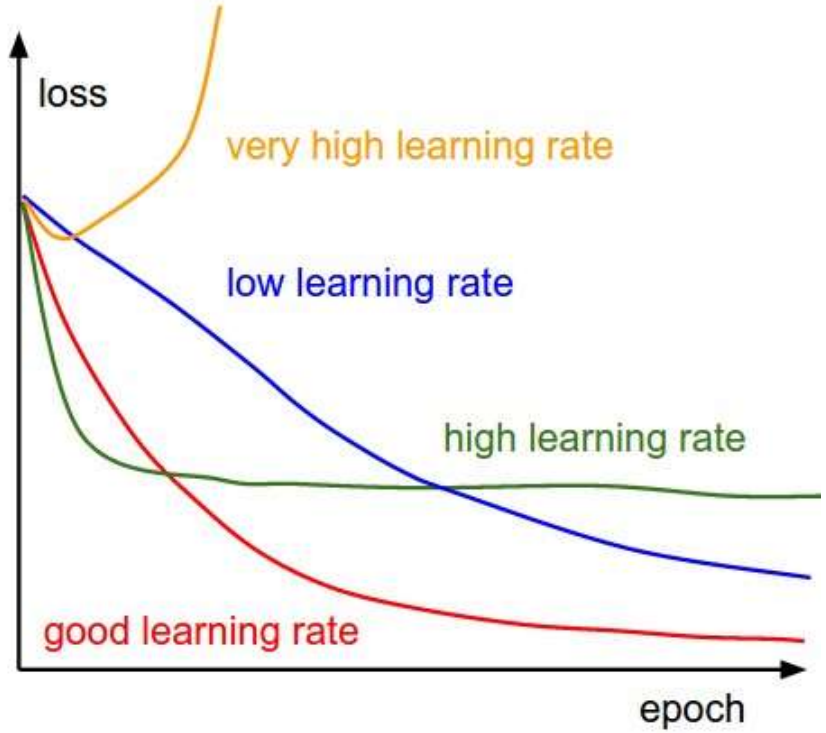
Hiperparametrii care pot fi optimizați

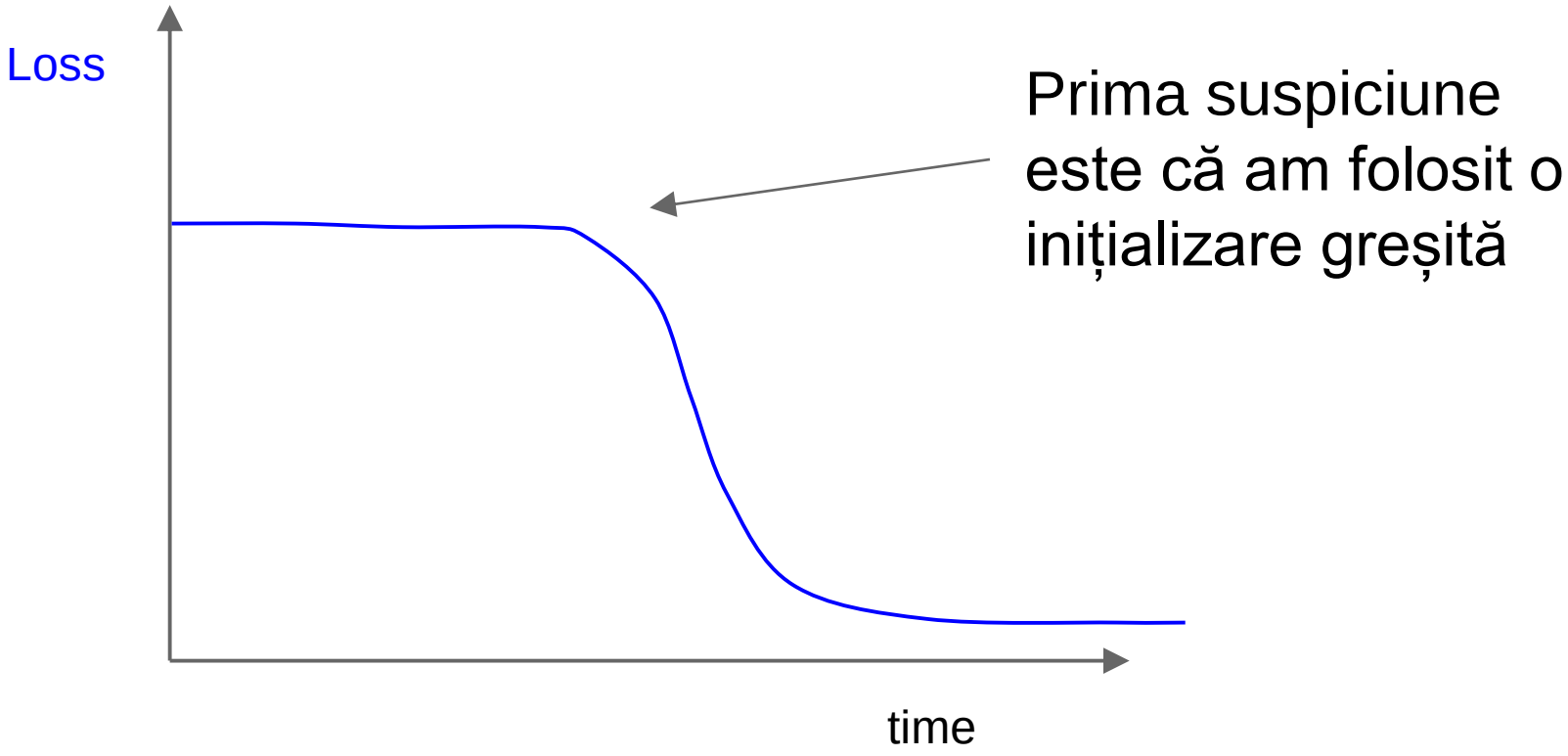
- Arhitectura rețelei
- Rata de învățare, cum se degradează rata (decay)
- Algortimul de învățare: SGD, SGD cu moment, etc.
- Regularizarea (L2 / Dropout)

Lucrul cu rețele neuronale
(muzica = funcția de pierdere)

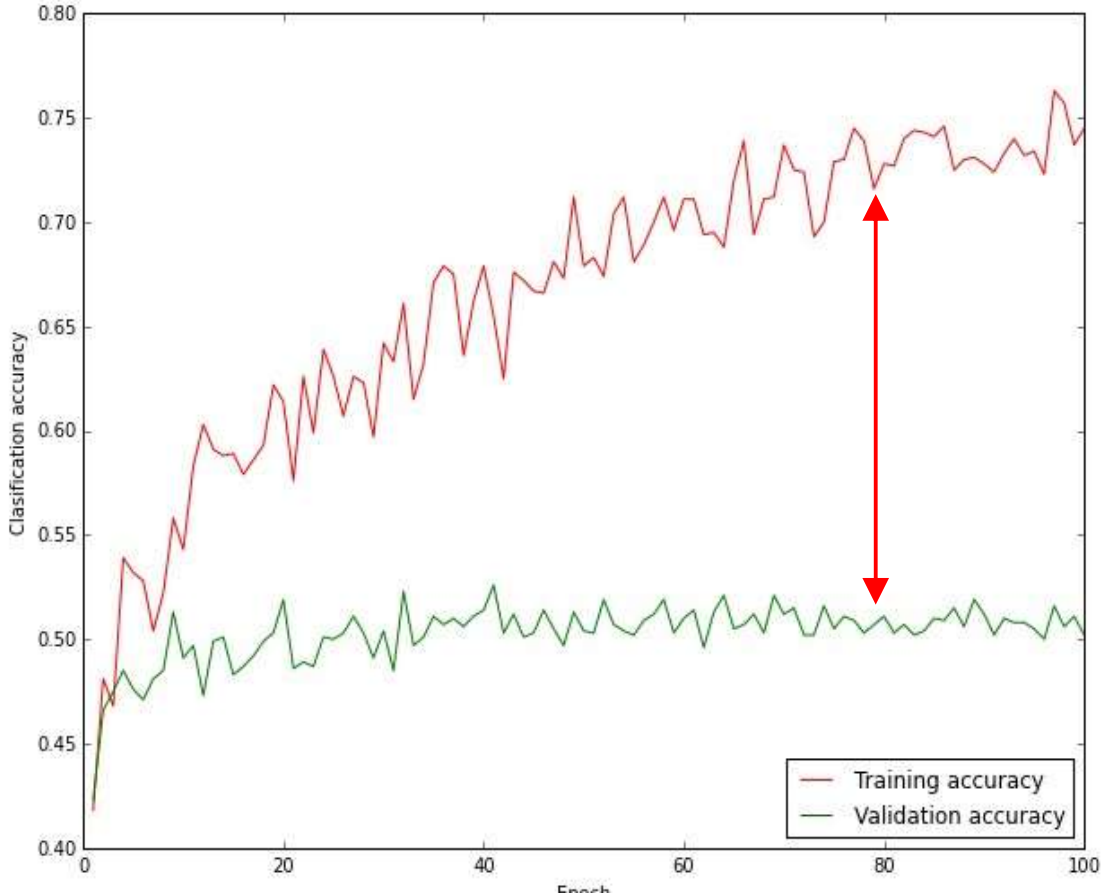
Monitorizăm evoluția funcției de pierdere






Loss
Prima suspiciune
este că am folosit o
inițializare greșită
time

Monitorizăm evoluția acurateții



distanță mare = overfitting

=> Creștem regularizarea?

distanță foarte mică

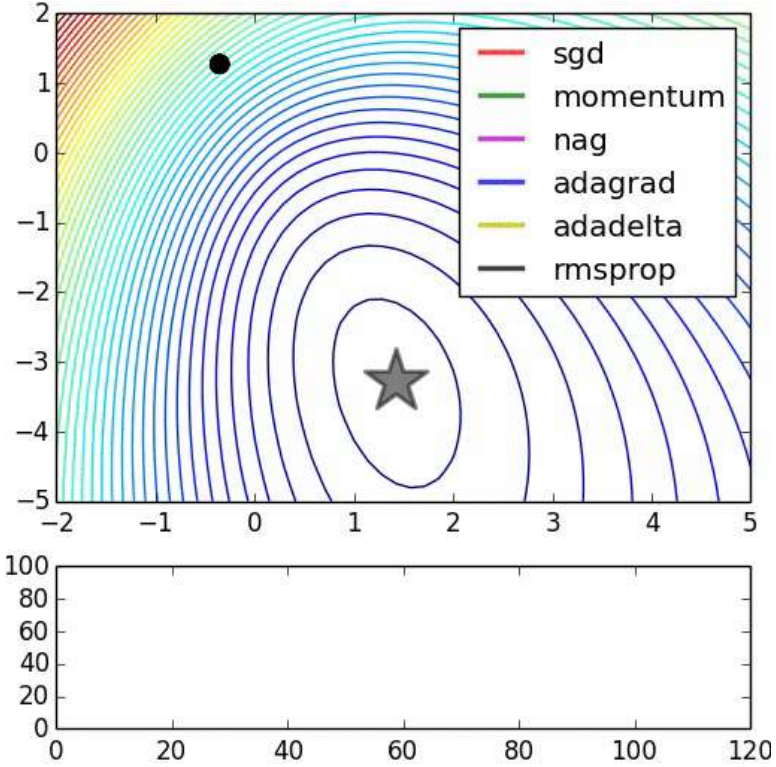
=> Creștem capacitatea modelului?

Sfaturi practice (până acum)

- Funcții de activare (folosim ReLU)
- Preprocesarea datelor (imagini: scădem media)
- Inițializarea ponderilor (folosim Xavier)
- Batch Normalization (folosim)
- Asistarea procesului de învățare
- Optimizarea hiperparametrilor (încercări aleatoare)

Algoritmul de optimizare

Există diverse variante ale algoritmului de antrenare

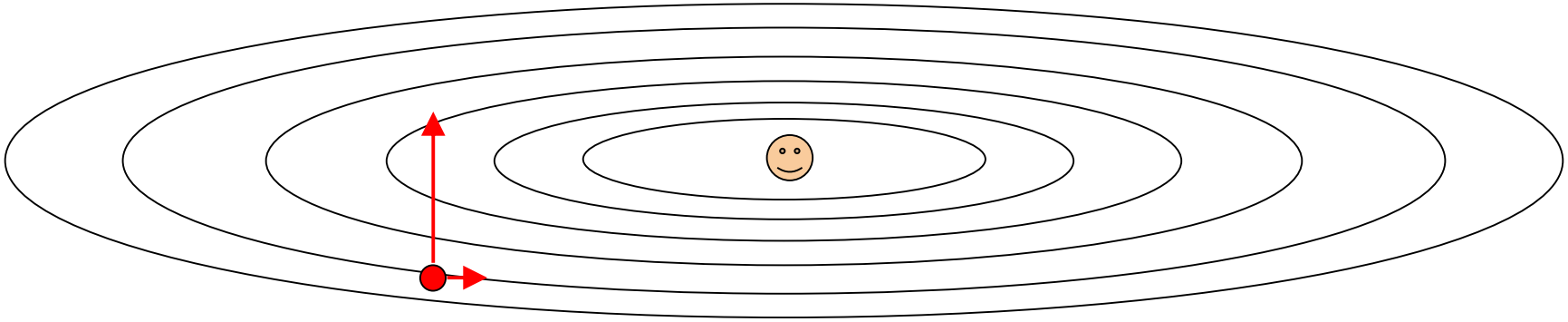


Imagine de Alec Radford

Algorimtul coborârii pe gradient (Python)

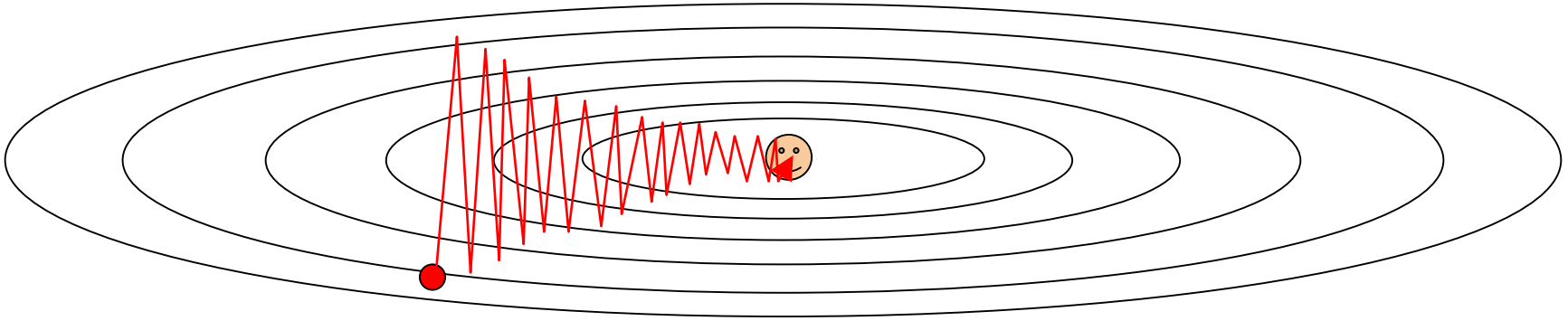
```
def GD(W0, X, goal, learningRate):
    perfGoalNotMet = true
    W = W0

    while perfGoalNotMet:
        gradient = eval_gradient(X, W)
        W_old = W
        W = W - learningRate * gradient
        perfGoalNotMet = sum(abs(W - W_old)) > goal
```

Dacă funcție este abruptă pe verticală, dar lină pe orizontală:



Q: Care este traiectoria de-a lungul căreia algoritmul SGD converge către minim?

Dacă funcție este abruptă pe verticală, dar lină pe orizontală:



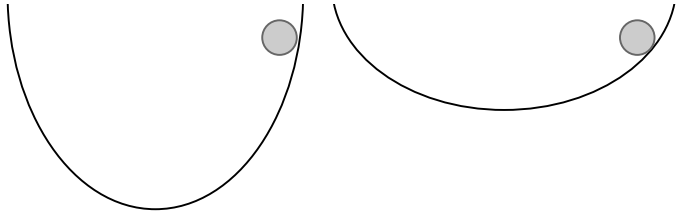
Q: Care este traiectoria de-a lungul căreia algoritmul SGD converge către minim? Progres încet pe direcția cu pantă lină, zig zag pe direcția abruptă

Algorimtul SGD cu moment (Python)

```
def GD(W0, X, goal, learningRate, mu):
    perfGoalNotMet = true
    W = W0
    V = 0
    while perfGoalNotMet:
        gradient = eval_gradient(X, W)
        W_old = W
        V = mu * V - learningRate * gradient
        W = W + V

        perfGoalNotMet = sum(abs(W - W_old)) > goal
```

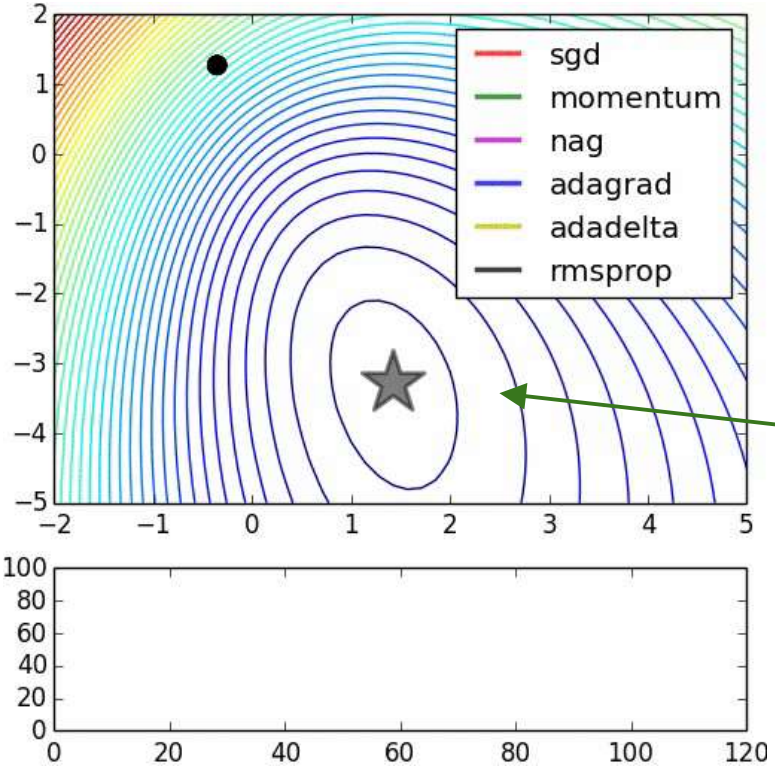
Algorimtul SGD cu moment

- Interpretarea fizică a unei mingi care se rostogolește pe funcția de pierdere
- Forța de frecare este dată de coeficientul mu
- mu = deobicei în jur de ~0.9, 0.95 sau 0.99 (câteodată se modifică în timp, e.g. de la 0.5 către 0.99)



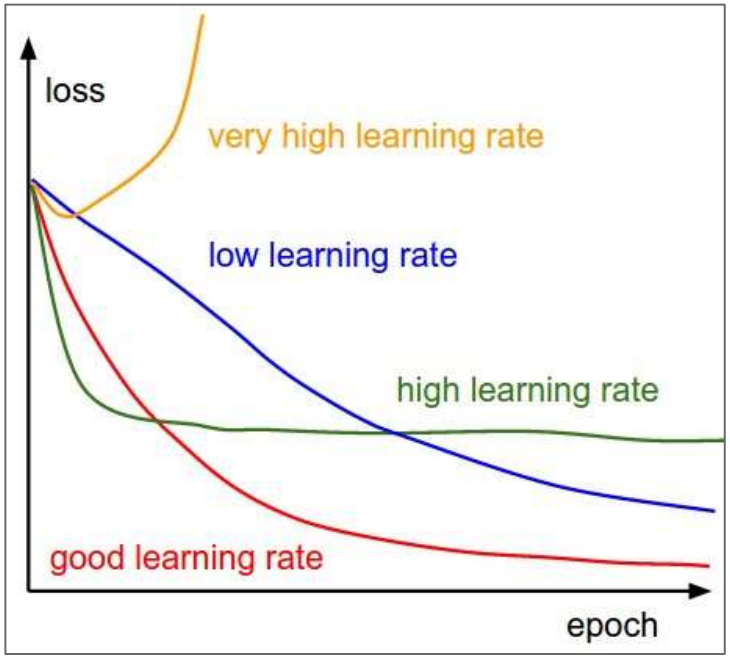
- Permite acumularea vitezei de-a lungul direcțiilor cu pantă lină
- Viteza se amortizează de-a lungul direcției abrupte din cauza schimbării dese a semnului / direcției de coborâre

SGD vs SGD cu moment



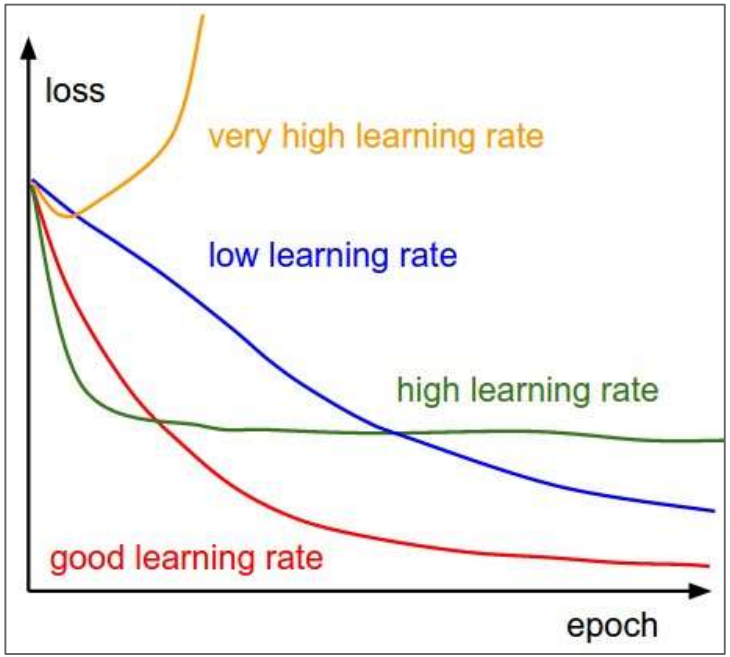
Observăm cum SGD cu moment depășește ținta, dar per total ajunge la minimul local mult mai rapid

Rata de învățare este un hiperparametru al SGD / SGD cu moment



Q: Care din aceste rate de învățare este mai potrivită?

Rata de învățare este un hiperparametru al SGD / SGD cu moment



=> Declinul ratei de învățare în timp

step decay:
e.g. rata de învățare se înjumătățește după fiecare câteva epoci

exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

Evaluare:
Ansamble de modele

Ansamble de modele

1. Antrenăm independent mai multe modele
2. La testare, calculăm media predicțiilor

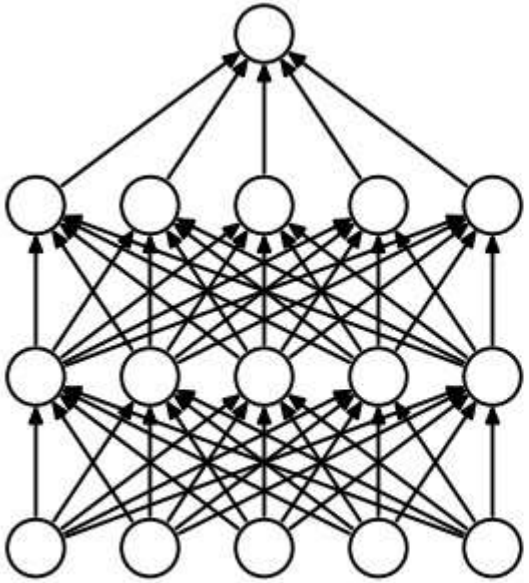
De obicei, acuratețea crește cu ~2%

Sfat practic: o mică îmbunătățire se poate obține și prin calcularea mediei predicțiilor date de un singur model, salvat la momente de timp diferite în timpul antrenării

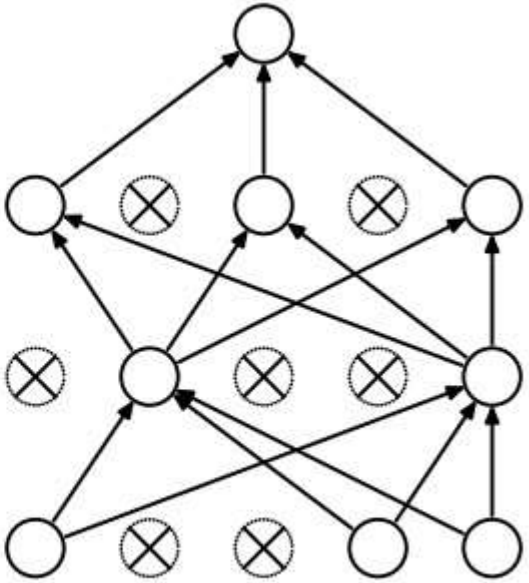
Regularizare folosind **Dropout**

Regularizarea folosind **Dropout**

Atribuim în mod aleator ponderi egale cu zero pentru o parte din neuroni (echivalent cu a deconecta o parte din neuroni)



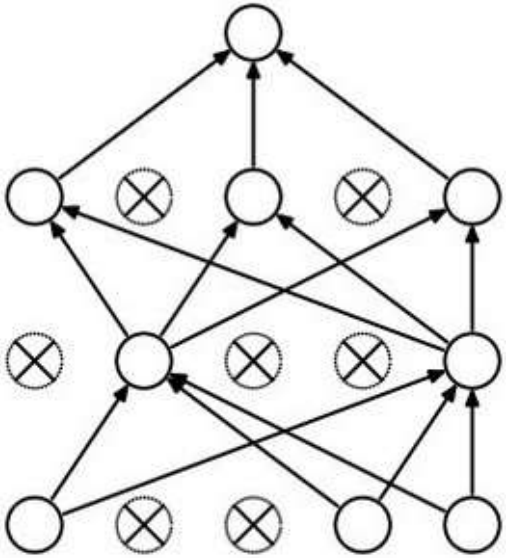
(a) Standard Neural Net



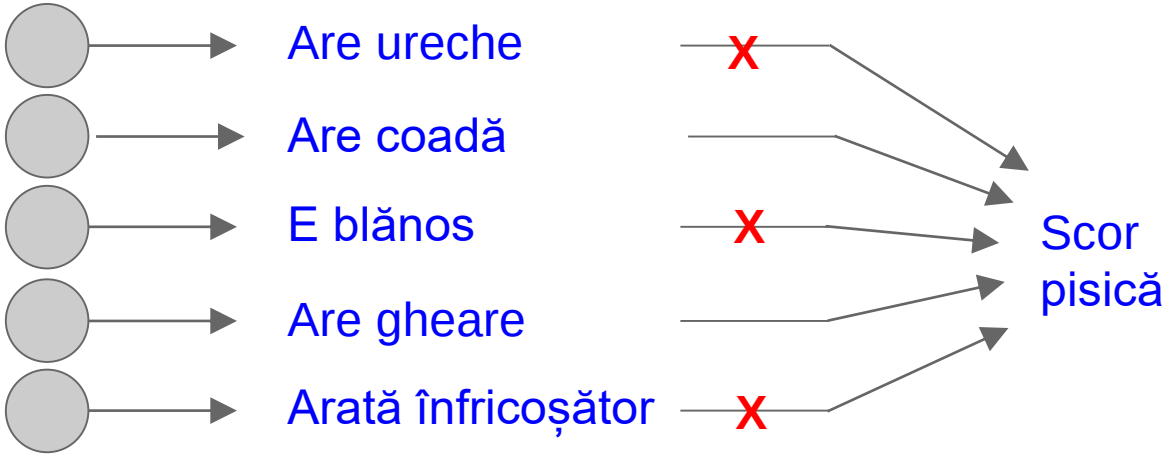
(b) After applying dropout.

[Srivastava et al., 2014]

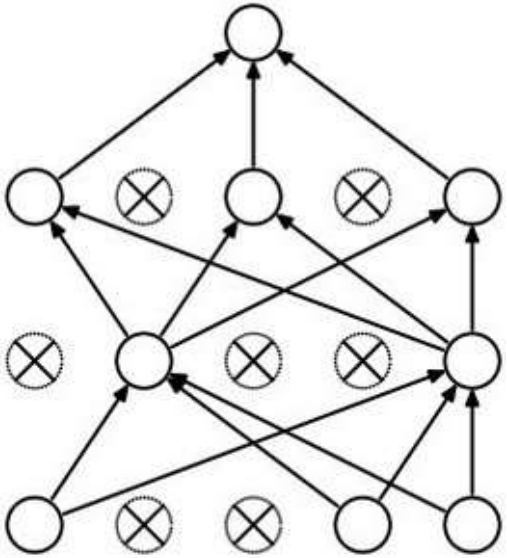
Cum ar putea fi asta o idee bună?



Forțează rețeaua să producă o reprezentare redundantă



Cum ar putea fi asta o idee bună?

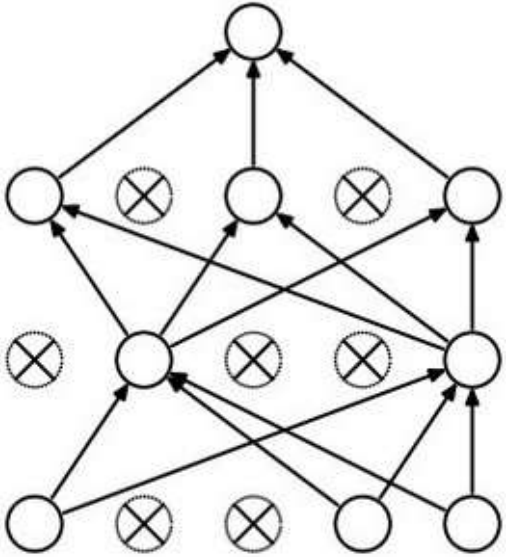


O altă interpretare:

Dropout este echivalent cu antrenarea unui ansamblu de multe modele (care au în comun parametrii)

Fiecare mască binară produce un model care se antrenează pe un exemplu / mini-batch

La testare…

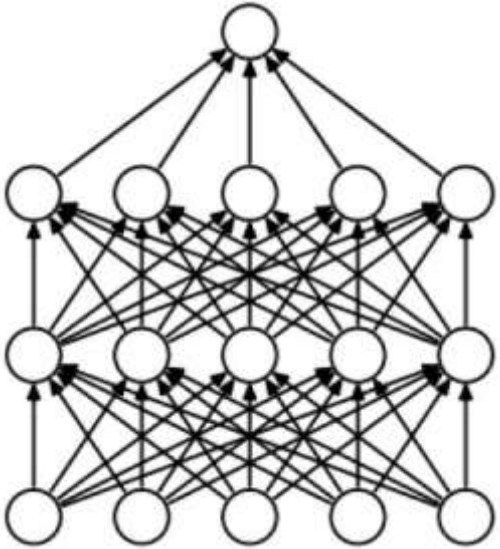


Ideal:
Vrem să eliminăm tot zgomotul

Aproximare Monte Carlo:
Facem mai multe treceri prin rețea folosind diverse măști de dropout, calculând apoi media predicțiilor

La testare…

Putem face totul printr-o singură trecere! (aproximativ)



Activăm toți neuronii (fără dropout)

(acestă variantă poate fi interpretată ca o aproximare a întregului ansamblu)