Ce a înțeles Gareth din Rețelele Neuronale?

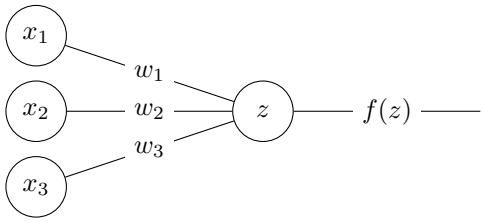
Iulian Oleniuc

3 noiembrie 2022

1 Ce este un perceptron?

Perceptronul este cel mai simplu clasificator binar, adică un algoritm care știe să distingă între două clase de obiecte. Obiectele sunt văzute ca niște vectori n-dimensionali, unde fiecare dimensiune reprezintă un atribut al obiectului dat ca input.

Acești vectori sunt definiți pe $\mathbb{R}^n$, iar output-ul pe mulțimea $\{0, 1\}$. Vom vedea că este mult mai eficient, prin prisma calculelor, să etichetăm clasele de obiecte cu 0 și 1 decât cu `false` și `true`. O altă convenție care ne va ușura munca este să restricționăm mulțimea vectorilor de intrare la $[0, 1]^n$.



1.1 Interpretare algebrică

Un perceptron este definit de un vector w de lungime n (numit *weights*) și o valoare b (numită *bias*). Pentru un input x de lungime n, perceptronul calculează valoarea $z \triangleq w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$. Output-ul algoritmului va fi obținut prin trecerea lui z printr-o *funcție de activare*. Momentan, această funcție nu are rost să fie alta decât

$$f(z) = \begin{cases} 0 & \text{dacă } z < 0 \\ 1 & \text{dacă } z \geq 0 \end{cases}.$$

Funcția poartă acest nume deoarece ne indică dacă, în urma calculelor, neuronul va fi activat sau nu. Mai târziu vom vedea că funcția f poate fi și continuă, spunându-ne deci *cât* de activat va fi neuronul.

1.2 Interpretare geometrică

Exemplele din viața reală sunt plictisitoare, așa că vom trece direct la perspectiva geometrică. Vom considera cazul bidimensional, pentru că este ușor de vizualizat. Așadar, avem o mulțime de puncte în plan, fiecare punct putând fi fie roșu (adică din clasa 0), fie verde (adică din clasa 1).

Ce reprezintă w și b?

În urma antrenării perceptronului nostru pe un set random de puncte, vrem să obținem un vector w de lungime 2 și un bias b care, împreună, să ne spună cumva ce culoare are punctul dat ca input. Păi, ce reprezintă de fapt acești w și b, mai ales în contextul în care $z = w_1x_1 + w_2x_2 + b$? Ecuația unei drepte!

În continuare, vom schimba notațiile în $w_1 \leftarrow a$, $w_2 \leftarrow b$ și $b \leftarrow c$. Așadar, dreapta despre care vorbim are ecuația $ax + by + c = 0$. De asemenea, ar fi util să redenumim și input-ul din (x_1, x_2) în (x_0, y_0).

Am observat că un lucru care ne obstrucționează percepția asta geometrică e faptul că bias-ul este adesea tratat separat, ca un lucru care nu are nicio legătură cu w, plus că poartă un nume oarecum ciudat. Mie cel puțin, la început mi se părea inutil – de ce să îl adaug la z când pot schimba funcția f în

$$f(z) = \begin{cases} 0 & \text{dacă } z < -b \\ 1 & \text{dacă } z \geq -b \end{cases}?$$

Ei bine, de fapt, bias-ul nu este altceva decât o componentă a ecuației dreptei.

Ce reprezintă z și $f(z)$?

Conform noilor notații, avem că $z = ax_0 + by_0 + c$. Asta poate vă aduce aminte de formula distanței de la punctul (x_0, y_0) la dreapta de ecuație $ax + by + c = 0$, mai precis

$$d = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}.$$

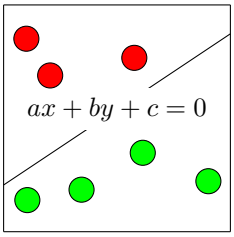
Dacă eliminăm modulul de la numărător, obținem distanța *cu semn* de la punct la dreaptă. Ei bine, pe noi asta ne și interesează de fapt – semnul distanței! El ne indică de care parte a dreptei se află punctul (x_0, y_0).

Din acest motiv, numitorul din formulă este ignorat total, căci el este mereu pozitiv, neinfluențând deci semnul distanței. Ce-i rămâne de făcut funcției f este să identifice semnul lui z, comparându-l cu 0, pentru ca în final să returneze valoarea binară corespunzătoare.

Care este scopul training-ului?

Ca să rezumăm, prin antrenarea perceptronului cu un număr cât mai mare de teste, testele fiind perechi de puncte și etichete (culori) corespunzătoare, dorim să obținem o dreaptă care să separe cât mai bine punctele roșii de cele verzi.

Ce înseamnă *cât mai bine*? Depinde de natura celor două clase de puncte. Dacă aceste clase se întâmplă să fie *liniar-separabile*, adică dacă există o dreaptă care să le separe complet, atunci putem găsi cu certitudine o astfel de dreaptă, deci modelul clasic de perceptron este suficient.



Altfel, pentru obținerea unei acurateți cât mai bune la testare, este clar că avem nevoie de ceva mai multă variație în luarea deciziei – dreapta va trebui transformată într-un fel de *curbă*. Altfel spus, output-ul va trebui să nu mai fie o *combinație liniară* a input-urilor.

Am animat aici antrenamentul unui perceptron pentru cazul în care cele două mulțimi de puncte sunt liniar-separabile.

Încă nu am spus nimic despre algoritmul de training în sine, pentru că metoda nu este decât un caz particular al algoritmului de *backpropagation*, și chiar dacă nu implică cine-știe-ce derivate, nu are sens să-l învățăm înainte să-i înțelegem cu adevărat varianta generală.

Asta cred că e una dintre greșelile cursului de RN – faptul că învățăm să facem training pe rețele neuronale cu doar două layere, respectiv pe perceptroni simpli, înainte să învățăm despre metoda *gradient descent*.

Generalizarea la n dimensiuni

Despre generalizare nu sunt multe de spus. Pur și simplu dreapta dată de w și b devine un *hiperplan* în n dimensiuni. Formula pentru distanța dintre punct și hiperplan se generalizează și ea convenabil. Nu are sens să ne batem capul cu vizualizarea a mai mult de 3 dimensiuni. Einstein abia a reușit cu 4.

1.3 Notația vectorială

Să revenim la formula lui z, și anume

$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b.$$

Suma asta arată aproape ca produsul scalar (dot product) dintre doi vectori. Deocamdată, putem rescrie ecuația vectorial ca

$$z = w \cdot x + b,$$

însă ne încurcă b-ul ăla singuratic.

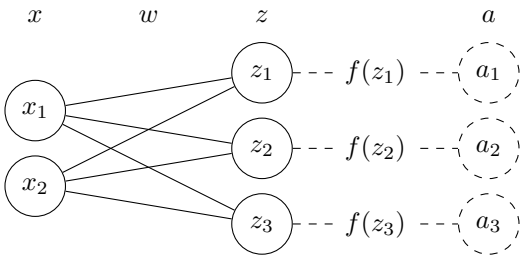
Pentru a nu mai trata separat bias-ul mereu, putem considera doi vectori noi $w' \triangleq w \,\|[b]$ și $x' \triangleq x \,\|[1]$, unde $\|$ desemnează operația de concatenare a vectorilor. Obținem așadar

$$z = w' \cdot x'.$$

Cu alte cuvinte, am arătat cum putem evita tratarea separată (mai ales în cod) a bias-ului. Prin urmare, în cele ce urmează vom face abstracție totală de bias-uri.

2 Rețele neuronale

Ce facem dacă vrem să distingem între mai mult de două clase de obiecte? Un perceptron nu mai este de ajuns. Însă putem folosi mai multe! Dacă avem m clase de obiecte, putem folosi m perceptroane – fiecare perceptron i va decide dacă obiectul dat face parte din clasa i sau nu. De remarcat că toți perceptronii vor fi conectați la aceleași noduri de input.



Acum avem de a face cu o întreagă *rețea neuronală*! Aceasta este compusă din două *layere* – unul de input și unul de output. Să introducem noile notații:

- x este vectorul de input-uri.
- z este vectorul obținut din valorile z_i calculate de fiecare perceptron i, folosind metoda din capitolul precedent.
- a este vectorul de output-uri. Numele vine de la *activation* și este motivat de faptul că $a_i \triangleq f(z_i)$.
- w este matricea de weight-uri. Vectorul w_i reprezintă weight-urile perceptronului i.

Valorile z_i și a_i trebuie privite ca făcând parte din același neuron.

2.1 Notația matriceală

Folosind notația vectorială din capitolul precedent, avem că $z_i = w_i \cdot x$. Ei bine, ordinea pe care am ales-o pentru indicii matricei w ne permite să scriem o singură ecuație, matriceală, pentru întregul vector z:

$$z = w \cdot x.$$

În mod similar, putem restrânge ecuația pentru vectorul a, dacă prin aplicarea unei funcții anume pe un vector înțelegem aplicarea acesteia pe fiecare dintre elementele vectorului:

$$a = f(z).$$

2.2 Distingerea între clase

Să spunem că rețeaua noastră neuronală primește ca input o imagine alb-negru de $28 \times 28 = 784$ de pixeli și vrea să distingă cifra scrisă în imagine. Layer-ul de intrare va conține 784 de neuroni, câte unul pentru fiecare pixel. Aceștia vor fi dați ca numere din intervalul $[0, 1]$, reprezentând nuanțele lor de gri. Layer-ul de ieșire va conține 10 perceptroni, câte unul pentru fiecare cifră posibilă.

Dacă cifra din imaginea dată este i, cum trebuie să arate vectorul a? Păi, elementul i trebuie să fie cât mai aproape de 1, pentru că perceptronul i trebuie să recunoască cifra i ca fiind într-adevăr i, pe când celelalte elemente trebuie să fie cât mai apropiate de 0.

Am folosit expresia *cât mai aproape* pentru că acum valorile a_i nu mai pot fi discrete. Cel puțin în timpul training-ului, ne interesează *cât de aproape* sunt predicțiile perceptronilor de răspunsurile corecte, pentru a putea alege cifra corespunzătoare perceptronului care ia *cea mai corectă* decizie.

În acest sens, trebuie să schimbăm funcția de activare f. Acum ne interesează distanța dintre punctul 784-dimensional dat ca input și hiperplanul dat de vectorul w_i, nu doar semnul ei. Cu cât punctul are distanța (cu semn) față de hiperplan mai mare, cu atât suntem mai siguri că perceptronul i l-a clasificat bine. Așadar, vom folosi $f(z) = z$ și răspunsul final va fi dat de perceptronul cu a-ul cel mai mare.

Putem folosi mai puțini perceptroni?

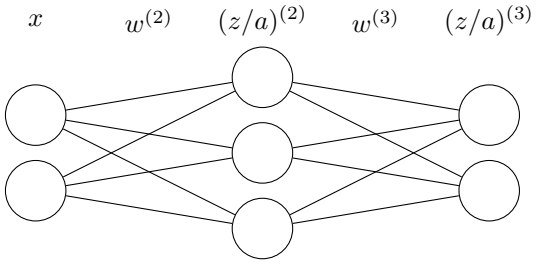
Da, conform unui rezultat din *teoria informației*, care ne spune că, pentru a reprezenta o variabilă care poate lua n valori, sunt suficienți $\lceil \log_2 n \rceil$ biți. Astfel, putem folosi doar $\lceil \log_2 10 \rceil = 4$ perceptroni, fiecare corespunzând unui bit din reprezentarea cifrei căutate în baza 2. Este o optimizare drăguță, dar inutilă.

3 Rețele neuronale cu mai multe layere

Deocamdată, weight-urile fiecărui perceptron de pe layer-ul de output formează un hiperplan. După cum spuneam mai devreme, în practică datele de intrare nu sunt niciodată liniar-separabile, așa că valorile a_i trebuie să nu mai fie combinații liniare ale input-urilor.

3.1 Nevoia de mai multe layere

Primul pas în această direcție este să mai adăugăm un layer *ascuns* în rețeaua neuronală. Fiecare neuron din acesta va comunica cu fiecare input și cu fiecare output.



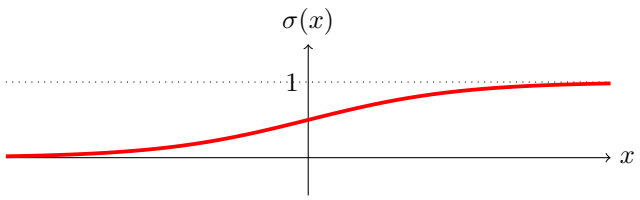
Problema este că, dacă toate funcțiile de activare din rețea vor rămâne $f(z) = z$, atunci output-urile vor fi tot combinații liniare ale input-urilor. Motivul este că o combinație liniară de combinații liniare este tot o CoMBiNAȚie LiNiARă – exercițiu lăsat pentru cititor.

3.2 Nevoia de o nouă funcție de activare

Așadar, f-urile ar trebui să fie funcții non-liniare. În plus, ar fi util să alegem o funcție cu codomeniul $[0, 1]$, deoarece output-urile dorite vor fi întotdeauna 1 (neuron activat) sau 0 (neuron neactivat), iar o valoare între 0 și 1 ne-ar spune cât de aproape este neuronul să se activeze. Nu în ultimul rând, funcția trebuie să fie (strict) monotonă, sau chiar crescătoare, conform nevoii precedente. Din aceste considerente, vom alege funcția *sigmoid*:

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

Putem observa din graficul de mai jos că σ mapează uniform fiecare număr din $\mathbb{R}$ la o valoare cuprinsă între 0 și 1:



3.3 Algoritmul Feed-Forward

Algoritmul Feed-Forward ne arată pur și simplu cum sunt calculate valorile de pe fiecare layer. Ecuațiile rămân aceleași ca și în cazul cu două layere, doar că acum avem mai mulți vectori z și a și mai multre matrice w. Pentru a face distincția între ele, vom folosi un superscript (l), unde $l \in \{2, 3, \ldots, k\}$ reprezintă indexul layer-ului curent.

- $z^{(l)} = w^{(l)} \cdot x$, pentru $l = 2$.
- $z^{(l)} = w^{(l)} \cdot a^{(l-1)}$, pentru $l > 2$.
- $a^{(l)} = f(z^{(l)})$, pentru $l \geq 2$.

4 Antrenarea rețelei neuronale

Pentru început, pornim de la ipoteza că avem un input x fixat și vrem să îmbunătățim rezultatul oferit de rețeaua neuronală pe *acest* test.

4.1 Funcția de cost

Primul pas este să găsim o modalitate de a cuantifica cât de bine seamănă output-ul $a^{(k)} \in [0, 1]^m$ cu vectorul pe care dorim să-l obținem – target-ul $t \in \{0, 1\}^m$, care este și el fixat.

În acest sens, vom defini o *funcție de cost*, să-i zicem c, care va primi ca parametri doi vectori m-dimensionali și ne va spune, pe o scară de la 0 la ∞, cât de *diferiți* sunt. O astfel de funcție este *mean squared error*:

$$c(v, w) = \frac{1}{2} \sum_{i=1}^{m} (v_i - w_i)^2.$$

Mean squared error

Problema cu formula aceasta e că de obicei nu ni se spune de unde vine, ci doar faptul că funcționează: Dacă v_i și w_i sunt foarte apropiate, atunci poziția i nu va contribui cu aproape nimic la rezultat, în timp ce, dacă cele două valori diferă puternic, rezultatul va crește cu aproape o unitate.

Dar explicația asta dă naștere la o întrebare arzătoare – cel puțin pentru mine – și anume, de ce diferențele sunt ridicate la pătrat, când este suficient să le scriem în modul? Pentru a găsi un răspuns satisfăcător, trebuie să ne întoarcem la adevărata origine a formulei:

$$c(v, w) = \frac{1}{2} \|v - w\|^2,$$

unde

$$\|x\| \triangleq \sqrt{x_1^2 + x_2^2 + \cdots + x_m^2}$$

reprezintă *norma* vectorului x, adică lungimea sa în spațiu.

Acum funcția de cost parcă are ceva mai mult sens. Întâi, efectuăm diferența (pe componente) dintre vectorii v și w. Apoi, calculăm lungimea (la pătrat) a vectorului-diferență, adică distanța sa (la pătrat) față de vectorul nul, care ne indică o diferență zero. Putem ignora radicalul din formula normei deoarece, fiind o funcție crescătoare, acesta nu afectează rezultatul comparării a două valori $c(v_1, w)$ și $c(v_2, w)$. Din același motiv, putem adăuga factorul $1/2$. Acesta va fi util mai târziu la derivarea lui c.

Un alt motiv pentru care formula se bazează pe $(v_i - w_i)^2$ și nu pe $|v_i - w_i|$, de care m-am prins în timp ce scriam aceste rânduri – v-am zis că înțelegi chestiile mult mai profund când încerci să le explici –, este faptul că modulul e dificil de derivat, iar noi vom avea nevoie să putem deriva funcția c cu ușurință.

Redefinirea funcției pentru rețeaua neuronală

Până acum, funcția c a fost definită pentru doi vectori. În cazul nostru, aceștia sunt $a^{(k)}$ și respectiv t. După cum spuneam, t este fixat, așa că îl putem scoate din signatura lui c.

Cât despre vectorul $a^{(k)}$, el este funcție de variabilele $w_{ij}^{(l)}$, adică depinde de fiecare weight din rețea în parte. În consecință, cea mai avantajoasă definiție a funcției c implică lista $w \triangleq \langle w^{(2)}, w^{(3)}, \ldots, w^{(k)} \rangle$ și nimic altceva:

$$c(w) = \frac{1}{2} \|a^{(k)} - t\|.$$

Nevoia de minimizare a funcției

Ceea ce ne dorim de la antrenarea rețelei este să găsim niște weight-uri care să conducă la un $c(w)$ cât mai apropiat de 0. Cu alte cuvinte, vrem să *minimizăm* funcția c.

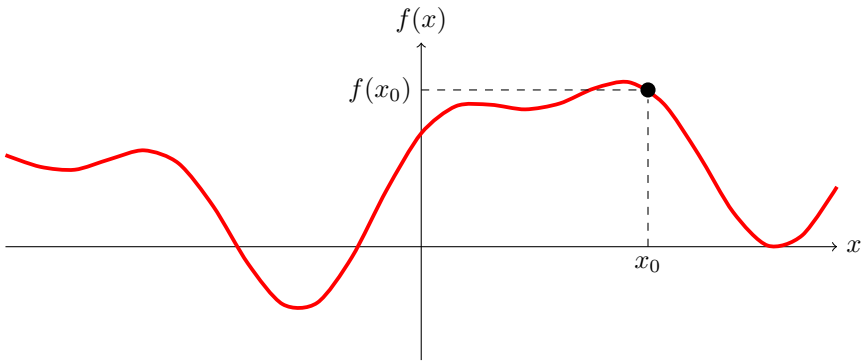
Faza este că funcția asta primește o grămadă de parametri. De exemplu, $c(w)$ pentru rețeaua de dimensiuni $\langle 784, 100, 10 \rangle$ din tema de la RN depinde de $784 \cdot 100 + 100 \cdot 10 = 79\,400$ variabile! Așadar, avem nevoie de o metodă eficientă de a minimiza o funcție cu număr arbitrar de parametri.

4.2 Gradient descent

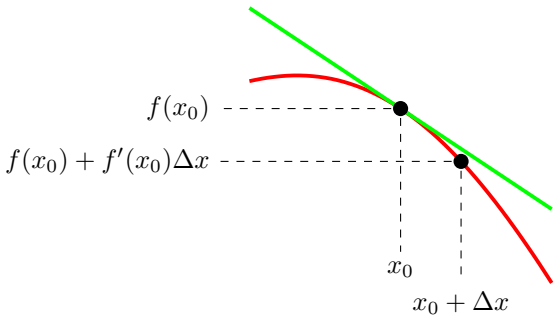
În această secțiune voi prezenta intuiția metodei *gradient descent* de minimizare a unei funcții de n variabile. Desigur, vom analiza doar cazul $n = 1$, pentru a putea vizualiza graficul funcției. Generalizarea va părea naturală pentru cei cărora le-a plăcut matematica din anul 1, semestrul 1.

Cazul unidimensional

Să luăm o funcție non-liniară, dar ușor derivabilă, cum ar fi $f(x) = \sigma(x) + \sin(x) + \cos^2(x)$. Fie un punct inițial $(x_0, f(x_0))$. Metoda *gradient descent* presupune ca la fiecare pas să-l creștem pe x_0 cu o valoare Δx (eventual negativă) care să ne ducă la un $f(x_0)$ mai mic decât cel precedent.



Valoarea lui f în noul punct este $f(x + \Delta x)$. Ideea este că, pentru un Δx suficient de mic, putem aproxima Δf prin $f'(x_0)\Delta x$. Acest lucru este justificat chiar de definiția derivatei lui f într-un punct x_0 – rata de schimbare a lui f în raport cu x pe măsură ce Δx se apropie de 0.



Pe noi ne interesează să alegem Δx în așa fel încât să obținem un Δf negativ, pentru a fi siguri că ne îndreptăm spre un minim (local) al funcției. Aici intervine ideea de bază a metodei *gradient descent*, și anume să alegem $\Delta x = -f'(x_0)$. Astfel, obținem $\Delta f \approx -(f'(x_0))^2$ – valoare care știm sigur că este negativă!

Iterând acest proces de un număr finit de ori, avem garanția că vom atinge un minim local al lui f. Aveți aici o vizualizare a algoritmului aplicat pe un set random de puncte inițiale pentru funcția de mai sus.

În practică, funcția $c(w)$ este suficient de random încât minimele locale să nu difere foarte mult de cele globale, astfel încât ne putem mulțumi cu ele. Asta e o vrăjeală pe care am inventat-o pe moment, nu cred că e adevărată.

Mai trebuie menționat faptul că $|\Delta x|$ poate fi prea mare, caz în care aproximarea pentru Δf va fi prea slabă, iar algoritmul va da greș. Ăsta e motivul pentru care trebuie să introducem *rata de învățare* $\eta > 0$. Redefinim Δx drept $-\eta f'(x_0)$ și, dacă alegem la început un η suficient de mic (depinde de la rețea la rețea, trebuie pur și simplu să încercăm diverse valori), $f(x_0)$ va converge într-adevăr către un minim local.

Generalizarea la n dimensiuni

Ce facem dacă funcția f depinde de n variabile? Exact același lucru, dar pentru fiecare variabilă (și deci derivată parțială) în parte:

$$x_i \leftarrow x_i - \eta \frac{\partial f}{\partial x_i}(x_i).$$

Vectorul de funcții $\partial f / \partial x_i$ poartă numere de *gradient* al lui f (de unde și numele algoritmului) și se notează cu ∇f. În consecință, putem rescrie vectorial atribuirea de mai sus pentru întregul vector x:

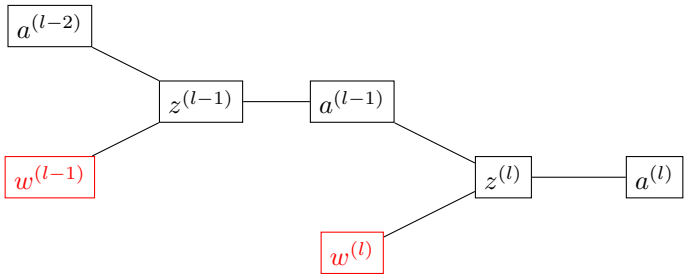
$$x \leftarrow x - \eta \nabla f(x).$$

Vă puteți gândi că ar trebui ales câte un learning rate diferit pentru fiecare dimensiune în parte, însă este evident că un η suficient de mic va fi ok pentru toate.

4.3 Algoritmul Back-Propagate

Tot ce ne mai rămâne de făcut este să aplicăm metoda *gradient descent* pe funcția c! Asta implică să calculăm câte o derivată parțială a lui c pentru fiecare weight $w_{ij}^{(l)}$. Chestia e că $c(w)$ nu depinde *în mod direct* de aceste variabile.

Valoarea lui $c(w)$ depinde în mod direct doar de $a^{(k)}$. Acest $a^{(k)}$ depinde la rândul lui de $z^{(k)}$. Abia $z^{(k)}$ depinde în mod direct de niște weight-uri, și anume de $w^{(l)}$. Totodată, $z^{(k)}$ depinde și de $a^{(l-1)}$, iar de aici dependințele continuă recursiv până la layer-ul 2.



Cheia este deci să privim vectorii $z^{(l)}$ și $a^{(l)}$ ca pe niște funcții de w și să calculăm derivatele parțiale pe rând, mergând mereu pe dependințe directe. De exemplu, putem calcula $\partial c / \partial w^{(l-1)}$ pornind de la $\partial c / \partial a^{(l)}$ în patru pași astfel:

$$\frac{\partial c}{\partial w^{(l-1)}} = \frac{\partial z^{(l-1)}}{\partial w^{(l-1)}} \cdot \frac{\partial a^{(l-1)}}{\partial z^{(l-1)}} \cdot \frac{\partial z^{(l)}}{\partial a^{(l-1)}} \cdot \frac{\partial a^{(l)}}{\partial z^{(l)}} \cdot \frac{\partial c}{\partial a^{(l)}}.$$

Regula aplicată se numește *chain rule* și este destul de intuitivă: În particular, dacă vrem să calculăm cu cât variază $f(g(x))$ când îl modificăm pe x, calculăm mai întâi cu cât variază $g(x)$ în raport cu x, iar apoi înmulțim această valoare cu variația lui $f(g(x))$ în raport cu $g(x)$.

Eroarea de pe layer-ul l

Înainte de a trece la calculele propriu-zise, vom introduce notația următoare, deoarece această derivată apare frecvent în calcule:

$$\delta_i^{(l)} \triangleq \frac{\partial c}{\partial z_i^{(l)}}.$$

Aceasta se numește *eroarea* de pe layer-ul l – denumire foarte confusing, pentru că noi de fapt nu propagăm erori, ci derivate.

Însă, aceste valori *pot* fi văzute drept erori: Cu cât o derivată parțială este mai mică, cu atât modificarea componentei respective va influența mai puțin rezultatul final, deci eroarea provocată de acel neuron este de asemenea foarte mică.

Derivata după $a_i^{(l)}$ pentru layer-ul $l = k$

Valoarea lui c depinde în mod direct de $a_i^{(k)}$, așa că înlocuim funcția c cu definiția ei și ajungem destul de ușor la rezultat:

$$\frac{\partial c}{\partial a_i^{(k)}} = \frac{\partial(\frac{1}{2}(a_i^{(k)} - t_i))^2}{\partial a_i^{(k)}} = a_i^{(k)} - t_i.$$

Derivata după $a_i^{(l)}$ pentru layer-ul $l < k$

Valoarea lui c depinde de $a_i^{(l)}$ prin mai mulți intermediari, și anume $z_j^{(l+1)}$. Așadar, vom exprima derivata după $a_i^{(l)}$ ca o sumă după j:

$$\frac{\partial c}{\partial a_i^{(l)}} = \sum_j \frac{\partial c}{\partial z_j^{(l+1)}} \cdot \frac{\partial z_j^{(l+1)}}{\partial a_i^{(l)}} = \sum_j \delta_j^{(l+1)} \cdot \frac{\partial z_j^{(l+1)}}{\partial a_i^{(l)}}.$$

În continuare, îl scriem pe $z_j^{(l+1)}$ ca o sumă de termeni de forma $w_{ji'}^{(l+1)} a_{i'}^{(l)}$, dintre care cei cu $i' \neq i$ vor dispărea atunci când derivăm după $a_i^{(l)}$:

$$\begin{aligned}\frac{\partial c}{\partial a_i^{(l)}} &= \sum_j \delta_j^{(l+1)} \cdot \frac{\partial(w_{j1}^{(l+1)} a_1^{(l)} + w_{j2}^{(l+1)} a_2^{(l)} + \cdots + w_{ji}^{(l+1)} a_i^{(l)} + \cdots)}{\partial a_i^{(l)}} \\ &= \sum_j \delta_j^{(l+1)} w_{ji}^{(l+1)}.\end{aligned}$$

Derivata după $z_i^{(l)}$ pentru layer-ul l

Valoarea lui c depinde de $z_i^{(l)}$ prin $a_i^{(l)}$. Vom avea nevoie să derivăm funcția de activare σ, care se întâmplă să aibă o derivată foarte elegantă (exercițiu pentru cititor), și anume $\sigma'(x) = \sigma(x)(1 - \sigma(x))$:

$$\begin{aligned}\delta_i^{(l)} = \frac{\partial c}{\partial z_i^{(l)}} &= \frac{\partial c}{\partial a_i^{(l)}} \cdot \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} \\ &= \frac{\partial c}{\partial a_i^{(l)}} \cdot \frac{\partial \sigma(z_i^{(l)})}{\partial z_i^{(l)}} \\ &= \frac{\partial c}{\partial a_i^{(l)}} \sigma'(z_i^{(l)}).\end{aligned}$$

Derivata după $w_{ij}^{(l)}$ pentru layer-ul l

Valoarea lui c depinde de $w_{ij}^{(l)}$ doar prin $z_i^{(l)}$. Cel din urmă se scrie ca sumă din $w_{ij'}^{(l)} a_{j'}^{(l-1)}$. Din nou, termenii cu $j' \neq j$ vor dispărea la derivare:

$$\begin{aligned}\frac{\partial c}{\partial w_{ij}^{(l)}} &= \frac{\partial c}{\partial z_i^{(l)}} \cdot \frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}} \\ &= \delta_i^{(l)} \cdot \frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}} \\ &= \delta_i^{(l)} \cdot \frac{\partial (w_{i1}^{(l)} a_1^{(l-1)} + w_{i2}^{(l)} a_2^{(l-1)} + \cdots + w_{ij}^{(l)} a_j^{(l-1)} + \cdots)}{\partial w_{ij}^{(l)}} \\ &= \delta_i^{(l)} a_j^{(l-1)}.\end{aligned}$$

Rezumat

Iată că am obținut singurele trei formule de care avem nevoie:

$$\begin{aligned}\delta_i^{(k)} &= \sigma'(z_i^{(k)})(a_i^{(k)} - t_i), \\ \delta_i^{(l)} &= \sigma'(z_i^{(l)}) \sum_j \delta_j^{(l+1)} w_{ji}^{(l+1)}, \\ \frac{\partial c}{\partial w_{ij}^{(l)}} &= \delta_i^{(l)} a_j^{(l-1)}.\end{aligned}$$

Notația vectorială

Putem scăpa de indici transformând operațiile pe componente în operații pe vectori:

$$\delta^{(k)} = \sigma'(z^{(k)}) \odot (a^{(k)} - t),$$
$$\delta^{(l)} = \sigma'(z^{(l)}) \odot (\delta^{(l+1)} w^{(l+1)}),$$
$$\frac{\partial c}{\partial w^{(l)}} = (\delta^{(l)})^T a^{(l-1)},$$

unde transpusa unui vector îl transformă pe acesta din vector-linie în vector-coloană, iar operația $\odot$ se numește *produsul Hadamard* și semnifică înmulțirea a doi vectori element cu element.

4.4 Mini-batch training

Acum avem tot ce ne trebuie pentru *backpropagation*! Singura problemă este că până acum am antrenat rețeaua pe un singur input, fixat. Noi vrem ca ea să se comporte bine pe cât mai multe input-uri. O idee în acest sens ar fi să rulăm rețeaua pe mai multe input-uri, α să zicem, iar apoi să aplicăm *gradient descent* pe media tuturor erorilor obținute.

Dacă α este egal cu numărul total de input-uri de antrenament, atunci progresul rețelei va fi lent, pentru că va trebui să satisfacă prea multe teste simultan. Dacă $\alpha = 1$, atunci progresul va fi din nou lent, deoarece nu prea va exista corelare între ajustările făcute pentru un test și cele făcute pentru altul. Un $\alpha \approx 10$ ar fi destul de ok, plus că, de exemplu, pe un training-set de mărime 50 000, ar accelera propagarea erorilor (nu și *feedforward*-ul, care se efectuează separat) cu un factor de 5 000. Am animat aici training-ul unei rețele cu șapte layere pe funcția de la capitolul *gradient descent*!!

4.5 Inițializarea weight-urilor

Sunt convins că secțiunea precedentă v-a făcut atât de entuziasmați încât m-ați rugat să vă țin berea și v-ați dus repede să codați rețeaua în Python. Ei bine, probabil ați observat că acuratețea ei nu se schimbă deloc. Asta se întâmplă dacă inițializăm weight-urile cu valori random între 0 și 1.

Motivul implică prea multă statistică, de care n-am chef acum, mai ales că trebuie să mă apuc de TPMP. Dar, pe scurt, ideea este că neuronii se *saturează*. Mai precis, probabilitatea ca z-urile să fie în afara intervalului $[-6, 6]$ este prea mare, ceea ce implică faptul că a-urile, care sunt de fapt $\sigma(z)$-uri, vor fi practic doar 0 sau 1, deci jumătate dintre neuroni vor funcționa din start foarte prost. Micile schimbări în weight-uri îi vor influența extrem de puțin, când ei de fapt trebuie să se schimbe puternic.

O soluție mai bună pentru inițializarea weight-urilor pentru fiecare layer în parte este să folosim *distribuția gaussiană normală*. Chiar mai bine ar fi ca deviația sa standard să fie $1/\sqrt{in}$, unde in este numărul de neuroni de pe layer-ul precedent conectați la un neuron de pe cel curent.

5 Bibliografie

- Neural Networks and Deep Learning, Michael Nielsen
- Neural Networks, 3Blue1Brown

6 Mulțumiri

- Lizuca
- Andreea
- Mihai