3 MNIST 101 *(40p)*

3.1 Enunț

Trecând prin periplul său prin algoritmi numerici și predicție cu ajutorul regresiei liniare, Mihai face un ultim pas pentru o introducere completă în învățarea supervizată. Foarte captivat de regresia liniară, el se întreabă cum ar putea adapta algoritmul și metodele de optimizare deja cunoscute pentru regresia liniară pentru a le putea folosi și la alt gen de probleme, cum ar fi problemele de **clasificare**.

Astfel, task-ul pe care și-l propune este să clasifice poze conținând cifre zecimale scrise de mână (de la 0 la 9) folosind un model de clasificare potrivit. Pentru că este vorba despre o problemă de clasificare în mai multe clase, clasificatorul ales de Mihai este o mică rețea neurală care are un strat de input cu 400 de unități neuronale (valorile pixelilor unei poze de dimensiune 20×20), un strat de output cu 10 unități (câte una pentru fiecare clasă) și un strat ascuns, cu un număr intermediar de unități neuronale (25 de unități), folosit pentru a crește complexitatea și deci și performanța modelului de clasificare.

Figura 10: Câteva exemple din dataset-ul MNIST

3.2 Referințe teoretice

3.2.1 Adaptare a regresiei liniare: regresia logistică

Principiul de bază prin care se realizau predicții cu ajutorul regresiei liniare era faptul că rezultatul dorit reprezenta o combinație liniară a unui set de parametri dați (*features* - ați întâlnit deja câteva exemple de features în cadrul celei de-a doua părți a temei). Astfel, un model similar poate fi folosit și pentru clasificarea datelor primite într-un număr finit de clase.

Să luăm mai întâi o problemă foarte simplă de clasificare în două clase. O problemă de clasificare cu două clase are drept date de intrare un vector de parametri (la fel ca la regresia liniară), împreună cu un rezultat, reprezentat de un label (*o etichetă*). În cazul clasificării binare, acest label poate avea valorile $y \in \{0, 1\}$. Încă din acest pas observăm ineficiența aplicării regresiei liniare pentru o problemă de clasificare: regresia liniară poate da drept rezultat (valoare prezisă) orice număr real (pozitiv sau negativ), un rezultat nepotrivit pentru predicția noastră în doar 2 clase.

Din acest motiv, avem nevoie de o metodă (o *neliniaritate*) prin care să mapăm rezultatul obținut în urma combinației liniare în intervalul $[0, 1]$. Acesta este motivul pentru care, în loc de binecunoscuta ipoteză

$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{\theta}^T \boldsymbol{x}$ să utilizăm o ipoteză modificată, de forma

$$h_{\boldsymbol{\theta}} = \sigma\left(\boldsymbol{\theta}^T \boldsymbol{x}\right)$$

unde $\sigma : \mathbb{R} \to [0, 1]$ este neliniaritatea amintită anterior. De obicei, se alege funcția *sigmoid* pentru maparea dorită, adică funcția:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

În figura de mai jos, aveți graficul funcției sigmoid, în care se evidențiază rolul acesteia de a mapa orice rezultat real în intervalul $[0, 1]$:

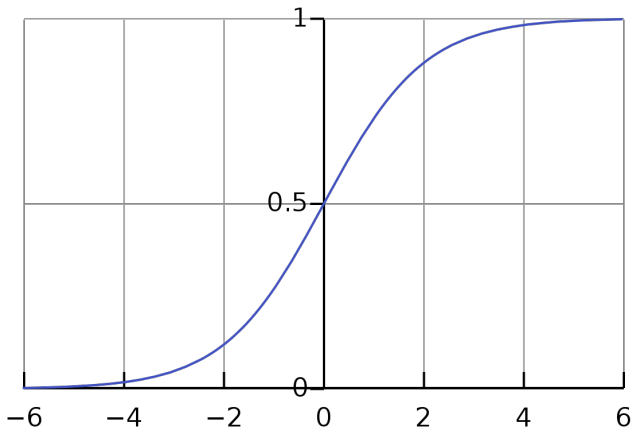

Figura 11: Graficul funcției sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$

Având acum o nouă formă a ipotezei noastre, trebuie să redefinim și funcția de cost (*loss function*) care va trebui să fie optimizată, întrucât eroarea pătratică (în termeni de normă 2 – cele mai mici pătrate) este insuficientă. Ne dorim ca un model neantrenat să aibă o eroare mare dacă valoarea ipotezei de regresie diferă semnificativ față de valoarea efectivă a clasei în care un exemplu este încadrat. De aceea, s-a introdus o nouă funcție de cost, al cărei rol este să evidențieze acest caz extrem, numită *cross-entropy*:

$$\text{cost}_i = -y^{(i)} \cdot \log\left[h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)\right] - (1 - y^{(i)}) \cdot \log\left[1 - h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)\right]$$

Funcția de cost (pe toate exemplele de training) devine:

$$J(\boldsymbol{\theta}) = \frac{1}{m} \cdot \sum_{i=1}^{m} \text{cost}_i = \frac{1}{m} \cdot \sum_{i=1}^{m} \left\{-y^{(i)} \cdot \log\left[h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)\right] - (1 - y^{(i)}) \cdot \log\left[1 - h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)\right]\right\}$$

Pentru acest caz, putem aplica tehnicile de optimizare a funcției de cost cunoscute deja din secțiunea anterioară a temei (**Gradient Descent** și o formă modificată de Gradient Conjugat), obținând un model cu performanțe foarte bune pentru task-uri simple de clasificare.

3.2.2 Neajunsurile regresiei logistice

Regresia logistică este o tehnică de învățare supervizată foarte bună atunci când avem de-a face cu probleme simple de clasificare (numărul de features este mic). Cu toate acestea, are unele neajunsuri, dintre care merită menționate următoarele:

- Regresia logistică clasică nu se poate extinde ușor la mai mult de 2 clase. Extinderea problemei de clasificare necesită câte un model particular pentru fiecare clasă introdusă (*one vs all classification*);
- Regresia logistică nu scalează la probleme de clasificare mai complexe, cum ar fi probleme specifice din zona de Computer Vision (identificarea obiectelor și procesarea imaginilor). Pentru astfel de probleme, este necesară utilizarea unor clasificatori mai complecși.

3.2.3 Extinderea de la regresia logistică la o rețea neurală. Perceptronul

Regresia logistică poate fi privită ca o rețea, așa cum este prezentat în figura de mai jos. În rețeaua dată, componentele noastre sunt reprezentate de:

- **Nodurile rețelei**, numite și **neuroni**;
- **Legăturile** între nodurile rețelei. Acestea semnifică contribuția (cu o anumită pondere[13]) a respectivului perceptron pentru calcularea valorii unui neuron din următorul strat;
- **Funcția de activare**, care reprezintă o neliniaritate. Cele mai uzuale funcții de activare sunt sigmoid (prezentat anterior), Rectified Linear Unit (ReLU) și tangenta hiperbolică. În cazul nostru, vom analiza strict cazul în care funcția de activare este sigmoid;
- Pentru modelul de mai jos, avem un anumit număr de unități neuronale de intrare și o singură unitate de ieșire, corespunzătoare clasei din care va face parte.

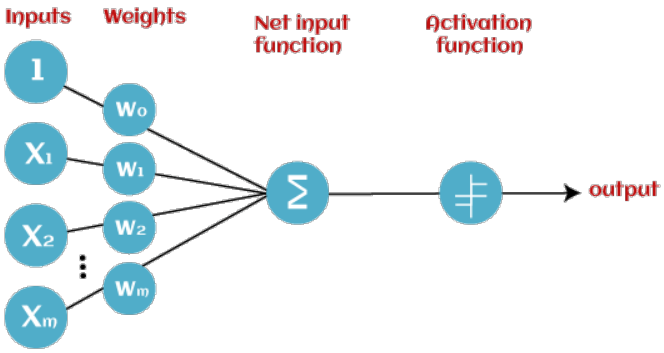

Figura 12: Perceptronul

Acest design poate fi extins prin includerea unor unități neuronale intermediare, care să formeze un **strat ascuns**[14] și prin mărirea numărului de unități neuronale de ieșire, corespunzător numărului de clase ale clasificatorului nostru. Astfel, am obținut o rețea neuronală conectată[15].

O rețea neuronală poate avea oricâte straturi ascunse, însă în cadrul acestui task noi vom folosi o arhitectură care are un singur strat ascuns, ca în figura următoare.

[13] en. *weight*
[14] en. *hidden layer*
[15] en. *fully-connected neural network*

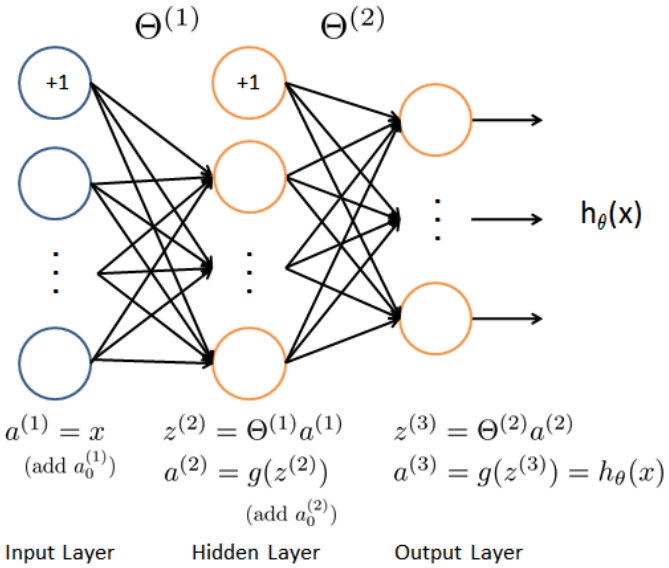

Figura 13: Arhitectura rețelei neurale folosite

Mărimile ce apar într-o rețea neurală descrisă de arhitectura de mai sus sunt:

- **Cele trei straturi existente** (denumite în literatura de specialitate *input layer*, *hidden layer* și *output layer*) au dimensiunile $s_1, s_2, s_3 \in \mathbb{N}$. Dimensiunea unui layer este reprezentată de numărul de neuroni din acel strat;
- **Numărul de clase finale** (care este egal cu numărul de neuroni din stratul de output) se notează cu $K \in \mathbb{N}$;
- Fiecare neuron dintr-un strat este caracterizat de o mărime, numită **activare**. Activările pentru layer-ul de input sunt chiar datele de intrare în rețeaua neurală (în cazul nostru, vor fi 400 de pixeli ai unor poze 20×20). Pentru layer-ul de output, activările sunt chiar predicțiile noastre (layer-ul de output va avea 10 unități neuronale). Pentru layer-ul intermediar (hidden) și pentru cel de output, activările vor fi determinate în funcție de toate activările neuronilor din layer-ul anterior (de aici și noțiunea de *fully-connected*);
- Pentru trecerea de la un layer la altul, vom utiliza o serie de parametri care alcătuiesc două matrice, $\Theta^{(1)} \in \mathbb{R}^{s_2 \times (s_1+1)}$ și $\Theta^{(2)} \in \mathbb{R}^{s_3 \times (s_2+1)}$.

3.2.4 Predicție. Forward propagation

Predicția clasei din care face parte un anumit exemplu este un procedeu efectuat atât în etapa de antrenare a modelului, cât și în etapa de testare (după antrenare). În cazul rețelei neurale, procedeul prin care se realizează determinarea activărilor neuronilor din layer-ul intermediar și determinarea predicțiilor finale se numește *forward propagation*. Acest procedeu are următorii pași:

- Fie $\left(\boldsymbol{x}^{(i)}, y(i)\right)$ un exemplu din dataset-ul de antrenare, unde $\boldsymbol{x}^{(i)}$ reprezintă datele de intrare în rețeaua neurală și $y^{(i)}$ reprezintă clasa din care face parte exemplul dat;
- Se construiește vectorul activărilor neuronilor din layer-ul de input din datele de intrare, la care se adaugă o unitate (*bias*):

$$\boldsymbol{a}^{(1)} = \begin{bmatrix} 1 \\ \boldsymbol{x}^{(i)} \end{bmatrix}$$

- Se aplică prima transformare liniară, dată de matricea $\Theta^{(1)}$, și se obține un vector $\boldsymbol{z}^{(2)}$ al rezultatelor intermediare (pre-activări).
$$\boldsymbol{z}^{(2)} = \Theta^{(1)} \cdot \boldsymbol{a}^{(1)}$$
- Se aplică funcția de activare (în cazul nostru, funcția sigmoid – ec. 3). Funcția de activare va fi implementată vectorizat, rezultatul aplicării acesteia pe un tablou fiind aplicarea funcției *sigmoid* pe fiecare element din acel tablou.
$$\boldsymbol{a}^{(2)} = \sigma\left(\boldsymbol{z}^{(2)}\right)$$
- Se adaugă o unitate la începutul vectorului activărilor (pentru *bias*):
$$\boldsymbol{a}^{(2)} = \begin{bmatrix} 1 \\ \boldsymbol{a}^{(2)} \end{bmatrix}$$
- Se aplică și cea de-a doua transformare liniară, astfel:
$$\boldsymbol{z}^{(3)} = \Theta^{(2)} \cdot \boldsymbol{a}^{(2)}$$
- Se aplică, din nou, funcția de activare (sigmoid):
$$\boldsymbol{a}^{(3)} = \sigma\left(\boldsymbol{z}^{(3)}\right)$$
- Aflându-ne în contextul ultimului layer, nu mai adăugăm unitatea pentru *bias*, iar activările obținute vor reprezenta predicțiile noastre pentru clasele propuse.

În cazul concret al task-ului nostru, vom obține un vector de 10 predicții, fiecare element reprezentând o predicție (similaritate) a exemplului dat cu una dintre cele 10 clase disponibile.

3.2.5 Determinarea gradienților. Backpropagation

La fel cum am observat la regresia liniară, orice model de învățare are nevoie de o modalitate prin care să își optimizeze (în acest caz, minimizeze) funcția de cost prin ajustarea parametrilor săi.

Pentru început, să scriem funcția de cost pentru o rețea neurală. Această funcție de cost reprezintă o generalizare a funcției de cost pentru regresie logistică și se bazează, de asemenea, pe **cross entropy**.

Înainte de a vă furniza formula, clarificăm că:

- $\Theta^{(1)}$ este o **matrice**, $\Theta^{(1)} \in \mathbb{R}^{s_2 \times (s_1+1)}$;
- $\Theta^{(2)}$ este o **matrice**, $\Theta^{(2)} \in \mathbb{R}^{s_3 \times (s_2+1}$;
- $\boldsymbol{\theta}$ este un **vector**, $\boldsymbol{\theta} \in \mathbb{R}^{s_2 \cdot (s_1+1)+s_3 \cdot (s_2+1)}$, și reprezintă vectorul care conține toate elementele din cele 2 matrice, în mod desfășurat.

$$J(\theta) = \frac{1}{m} \cdot \sum_{i=1}^{m} \text{cost}_i = \frac{1}{m} \cdot \sum_{i=1}^{m} \sum_{k=1}^{K} \left\{ -y_k^{(i)} \cdot \log\left[h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)_k\right] - \left(1 - y_k^{(i)}\right) \cdot \log\left[1 - h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)_k\right] \right\}$$
$$+ \frac{\lambda}{2m} \left[\sum_{j=2}^{s_1+1} \sum_{k=1}^{s_2} \left(\Theta_{k,j}^{(1)}\right)^2 + \sum_{j=2}^{s_2+1} \sum_{k=1}^{s_3} \left(\Theta_{k,j}^{(2)}\right)^2 \right]$$

A se observa că, la fel ca la regresie liniară, nu am regularizat și ponderile corespunzătoare activărilor constante (*biases*).

Dacă la regresie liniară ajustarea parametrilor se putea realiza cu ajutorul gradienților determinați analitic (sub forma derivatelor parțiale ale funcției de cost), în cazul rețelelor neurale acest lucru nu mai este posibil direct, întrucât o expresie analitică a gradienților este foarte dificil de obținut.

Cu toate acestea, putem folosi un algoritm cu ajutorul căruia să determinăm gradienții pentru fiecare parametru al modelului. Acest algoritm se numește **backpropagation**.

Algoritmul de backpropagation se bazează pe soluționarea gradienților prin intermediul determinării **erorilor de activare**. Concret, să presupunem că tocmai am realizat *forward propagation* pentru a determina predicțiile exemplului de antrenament curent pentru fiecare clasă. Atunci, putem defini eroarea care a apărut în activarea din layer-ul l, în neuronul de indice k, notată cu $\delta_k^{(l)}$. De asemenea, vom mai păstra matricele $\Delta^{(1)}$ și $\Delta^{(2)}$, de aceleași dimensiuni cu matricele $\Theta^{(1)}$ și $\Theta^{(2)}$, în care vom acumula gradienții parametrilor rețelei.

Pe scurt, pașii pentru algoritmul de *backpropagation* sunt:

- Determinăm eroarea în layer-ul de output:
$$\delta^{(3)} = a^{(3)} - y^{(i)}$$
- Putem acumula gradienții pentru parametrii care fac trecerea de la layer-ul intermediar la layer-ul de output, folosind formula:
$$\Delta^{(2)} = \Delta^{(2)} + \delta^{(3)} \cdot (a^{(2)})^T$$
- Pentru determinarea erorii în layer-ul intermediar, folosim formula:
$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* \sigma'(z^{(2)})$$
unde σ' este derivata funcției de activare *sigmoid*, avem adevărată relația $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$. De asemenea, operatorul $.*$ reprezintă produsul Hadamard a două tablouri (produsul *elementwise*). Hint: ca notă de implementare, puteți folosi direct activarea calculată anterior, iar din primul termen al operației $.*$ puteți elimina prima componentă (contribuția la eroarea unității de bias). În caz contrar, puteți obține o eroare de tip *dimension mismatch*.
- Eliminăm prima componentă din $\delta^{(2)}$ (aceasta este componenta pentru bias, acolo unde nu are sens să calculăm o eroare în valoarea activării).
- Acumulăm gradienții și pentru parametrii care fac trecerea de la layer-ul de input la cel intermediar:
$$\Delta^{(1)} = \Delta^{(1)} + \delta^{(2)} \cdot (a^{(1)})^T$$
- După ce am realizat acumularea gradienților pentru toate exemplele de antrenament, putem împărți la numărul exemplelor de antrenament:
$$\frac{\partial J(\Theta)}{\partial \theta_{ij}^{(l)}} = \frac{1}{m} \cdot \Delta_{ij}^{(l)}$$
- La final, putem adăuga și termenul corespunzător regularizării (numai pentru $j > 1$, pentru $j = 1$ formula anterioară rămâne valabilă):
$$\frac{\partial J(\Theta)}{\partial \theta_{ij}^{(l)}} = \frac{1}{m} \cdot \Delta_{ij}^{(l)} + \frac{\lambda}{m} \cdot \Theta_{ij}^{(l)}$$

Având la dispoziție acum gradienții și valoarea funcției de cost, putem realiza optimizarea funcției de cost prin Gradient Descent sau Gradient Conjugat.

3.2.6 Inițializarea parametrilor

În cazul rețelelor neurale, inițializarea parametrilor (elementelor matricelor) cu valori nule nu este posibilă (avem simetrie și vom obține o simetrie în ceea ce privește clasificarea exemplului nostru în diversele clase disponibile). De asemenea, inițializarea cu zero a parametrilor duce la anularea gradienților (rețeaua neurală este incapabilă să învețe), o problemă care în Deep Learning se numește Vanishing Gradient Problem. Puteți citi mai multe despre această problemă aici.
Soluția este inițializarea parametrilor cu valori aleatoare, din intervalul $(-\epsilon, \epsilon)$. Empiric, s-a constatat că o valoare potrivită pentru ϵ este dată de următoarea formulă:

$$\epsilon_0 = \frac{\sqrt{6}}{\sqrt{L_{prev} + L_{next}}}$$

3.3 Cerințe

Având în vedere referințele teoretice, aveți de implementat următoarele funcții:

- `function [X, y] = load_dataset(path)`
  Funcția *load_dataset* primește o cale relativă la un fișier .mat și încarcă în memorie acel fișier, returnând matricea care conține exemplele folosite pentru training și pentru test. Liniile matricei X vor reprezenta exemplele de date. Pentru fiecare linie, se va adăuga la început o coloană care să reprezinte termenul de *bias* (o coloană numai cu valori de 1).

- `function [x_train, y_train, X_test, y_test] = split_dataset(X, y, percent)`
  Funcția *split_dataset* primește un dataset, așa cum a fost el returnat de funcția anterioară (training examples, împreună cu labels) și împarte setul de date în 2 seturi: un set de training și un set de test, ambele reprezentate printr-o matrice de features și un vector de clase. Împărțirea pe cele 2 seturi se va face astfel: se amestecă exemplele, iar apoi o fracțiune egală cu parametrul *percent* din exemplele date în dataset va fi adăugată în setul de training (valorile de retur X_train și y_train), iar restul exemplelor vor fi plasate în setul de test.

- `function [matrix] = initialize_weights(L_prev, L_next)`
  Funcția *initialize_weights* primește dimensiunile (numărul de neuroni) celor 2 straturi între care se aplică transformarea liniară și întoarce o matrice cu elemente aleatoare din intervalul $(-\epsilon, \epsilon)$, conform precizărilor din referințe.

- `function [grad, J] = cost_function(params, X, y, lambda, input_layer_size, hidden_layer_size, output_layer_size)`
  Funcția *cost_function* primește următorii parametri:
  - *params* reprezintă un vector coloană care conține toate valorile ponderilor (*weights*) din matricele $\Theta^{(1)}$ și $\Theta^{(2)}$. Cu alte cuvinte, folosind elementele din acest vector și dimensiunile straturilor putem construi matricele pentru transformările liniare. Hint: **reshape**.
  - X reprezintă mulțimea exemplelor de training, fără labels asociate (*feature matrix*).
  - y reprezintă label-urile asociate exemplelor de mai sus.
  - *input_layer_size* reprezintă dimensiunea stratului de input.
  - *hidden_layer_size* reprezintă dimensiunea stratului intermediar/ascuns.
  - *output_layer_size* reprezintă dimensiunea stratului final (care este egal cu numărul de clase).

  Funcția returnează un vector de aceeași dimensiune cu parametrul *params*, obținut prin desfășurarea (*unrolling*) matricelor în care calculăm gradienții după aplicarea algoritmului de *backpropagation*, și J, care reprezintă funcția de cost pentru valoarea momentană a parametrilor.

- `function [classes] = predict_classes(X, weights, input_layer_size, hidden_layer_size, output_layer_size)`
  Funcția primește un set de exemple de test și vectorul pentru weights, precum și dimensiunile layerelor rețelei și întoarce un vector cu toate predicțiile pentru exemplele date din setul de test.
  Hint: **forward propagation**

3.3.1 Restricții și precizări

- În dataset-ul folosit, pentru clasa corespunzătoare cifrei 0 s-a folosit label-ul 10, tocmai pentru ca label-urile să fie conforme cu indexarea din GNU Octave.