

Colegiul Național "I.L. Caragiale" București  
Clasa a-XII-a  
Profil Mate-Info, intensiv Informatică

# LUCRARE DE ATESTAT

## Clasificarea unei poze folosind o rețea neurală convolutională

Elev  
Ramadan Omar

Coordonator  
Prof. dr. ing. Dumitrescu Iuliana

București, 2021

# Cuprins

<b>Listă de figuri</b>	<b>iii</b>
<b>1. Introducere</b>	<b>1</b>
1.1. Abstract . . . . .	1
1.2. Motivul Temei . . . . .	1
1.3. Tehnologii folosite . . . . .	1
<b>2. Noțiuni de Teorie</b>	<b>3</b>
2.1. Ce sunt Rețelele Neuronale? . . . . .	3
2.2. Perceptroni . . . . .	3
2.3. Rețea simplă de neuroni cu mai multe straturi . . . . .	4
2.4. Neuroni de tip Sigmoid . . . . .	5
2.5. Arhitectura unei rețele neuronale . . . . .	6
2.6. Feed-Forward Neuronal Network . . . . .	7
2.7. Rețele Neuronale Convoluționale . . . . .	7
2.7.1. Câmpuri receptive locale . . . . .	7
2.7.2. Greutăți și bias-uri comune . . . . .	8
2.7.3. Strat de pooling . . . . .	8
<b>3. Implementarea Algoritmului</b>	<b>9</b>
3.1. Implementarea Algoritmului in python . . . . .	9
3.1.1. Importarea Algoritmilor . . . . .	9
3.1.2. Împărțirea setului de date . . . . .	9
3.1.3. Preprocesarea Datelor . . . . .	9
3.1.4. Pregătirea pentru non-liniaritate . . . . .	10
3.1.5. Rețeaua Neuronală Convoluțională . . . . .	10
3.1.6. Evaluarea Modelului . . . . .	11
<b>Anexe</b>	<b>13</b>
<b>A. Fișiere sursă</b>	<b>13</b>

## Listă de figuri

2.1. Reprezentarea grafica a unui perceptron . . . . .	3
2.2. Reprezentarea completă a unui Perceptron . . . . .	4
2.3. Reprezentarea unei rețele simple . . . . .	5
2.4. Reprezentarea funcției sigmoid . . . . .	6
2.5. Reprezentarea unei rețele neuronale . . . . .	7
2.6. Neuroni input într-o rețea convoluțională . . . . .	8

# 1. Introducere

## 1.1. Abstract

Oamenii au început să devină din ce în ce mai conștienți de mediu luând tot felul de măsuri pentru a reduce impactul asupra acestuia. Una din măsurile adaptate de multe corporații și instituții de stat este folosirea exclusivă a documentelor digitale. Multe dintre ele au arhive cu sute de mii de documente, ceea ce face transcrierea manuală o opțiune imposibilă. Datorită acestui fapt, în piață s-a dezvoltat o cerere foarte mare de a converti documente scrise de mână în formele lor digitale. Pentru a automatiza acest proces, detectarea cifrelor dintr-o poză este inevitabilă și extrem de folositoare.

Una din cele mai noi și performante metode din domeniul detectării obiectelor este folosirea rețelelor neuroale convoluționale (CNN). Rezultate noastre arată faptul că putem atinge rezultate deosebite (peste 97.00% acuratețe) pe setul de date MNIST.

## 1.2. Motivul Temei

Sistemul vizual al omului este una din minunile lumii. Ia în considerare următoarea secvență de cifre:

A handwritten sequence of six digits, '504192', written in black ink on a white background. The digits are slightly slanted and have a casual, human-like appearance.

Majoritatea oamenilor pot recunoaște fără nicio problemă cifrele ca și când ar fi: 504192. Însă, acest lucru poate fi înșelător de simplu: În fiecare hemisferă a creierului nostru, avem un vortex vizual primar, cunoscut și drept  $V_1$ , care conține peste 140 de milioane de neuroni, cu zeci de miliarde de conexiuni între ele. Cu toate acestea, viziunea umană nu se limitează doar la un  $V_1$ , dar la o serie întreagă de cortexuri vizuale:  $V_2, V_3, V_4$  și  $V_5$  - care fac tot felul de procesări și analize al imaginilor. Ceea ce avem noi în cap este un supercomputer, antrenat și tunat de evoluție pe parcursul a sute de milioane de ani.

Recunoașterea cifrelor nu este un job simplu, care devine aparent dacă te gândești la cum ai putea scrie un program care să recunoască astfel de caractere. Ceea ce pare chiar trivial la început, va ajunge să fie ceva extrem de complicat. Intuiții simple despre cum recunoaștem o formă, un 9 are o buclă sus, și un arc jos, ajung să fie foarte greu de exprimat algoritmic. Chiar dacă ai reușit să scrii un astfel de algoritm, el nu ia în considerare faptul că fiecare om scrie diferit fiecare cifră.

## 1.3. Tehnologii folosite

Pentru realizarea acestui proiect, am folosit limbajul de programare Python 3 (versiunea 3.8.8). Python este un limbaj de programare general high-level, care are ca filozofie lăgibilitatea codului

cu un accent mare pus pe indentare. Limbajul permite atât programarea funcțională cât și cea orientată pe obiecte, dispunând de un garbage-collector, asemănător cu cel regăsit în C++.

Una din caracteristicile importante ale limbajului Python este aceea de a opera într-un mediu de dezvoltare (virtual environment) izolat de restul mașinii. În acest mediu de dezvoltare, vom adăuga unele librării Open-Source pentru a ușura procesul de dezvoltare.

Librăriile de care ne vom folosi sunt:

1. Numpy: O librărie care adaugă suport pentru vectori și matrici multidimensionale, scrisă în C care ne permite să ne bucurăm de flexibilitatea unui limbaj High-Level precum Python păstrând în același timp performanța unui cod compilat.
2. Keras: este un API proiectat pentru a ușura implementarea straturilor neuronale optimizatorii, funcțiile de cost, schemele de inițializare/regulazire și funcțiile de activate, permitându-ne să ne folosim de plăcile video pentru a face calculele și a micșora timpul de antrenare al modelului.
3. Tensorflow: este o librărie creată de echipa celor de la Google Brain, care se bazează pe API-ul Keras, care ușurează procesul de achiziții de date, modelelor de instruire, servirea predicțiilor și rafinarea rezultatelor viitoare. Tensorflow folosește un cod sursă scris în C++, cu un API în Python care, la fel ca Numpy, ne permite să păstrăm performanța unui limbaj low-level compilat.
4. Matplotlib este o bibliotecă multi-platformă, de vizualizare a datelor și grafică grafică pentru Python și extensia sa numerică NumPy.

Pentru a dezvolta algoritmul și a antrena modelele, a fost folosit un calculator cu sistemul de operare Ubuntu 20.04, și o placă video nVidia 1660Ti. Dacă nu aveți o placă video, puteți închiria una de pe un Cloud Hosting Provider cum ar fi: Amazon Web Services, Google Cloud, Microsoft Azure.

## 2. Noțiuni de Teorie

### 2.1. Ce sunt Rețelele Neuronale?

Rețelele Neuronale Artificiale (ANN), des întâlnite sub denumirea de Rețele Neuronale (NN), sunt sisteme de calculare inspirate din biologia rețelor de neuroni care constituie creierul de animale.

O rețea neuronală este formată dintr-o colecție de noduri numite neuroni, care seamănă în structură cu neuronii din creierul nostru. Fiecare conexiune, la fel ca sinapsele din creierul nostru, pot transmite un semnal către alt neuron. Un neuron poate primi un semnal, pe care îl procesează și-l transmite mai departe neuronilor conectați la el. În acest caz, semnalul este un număr real, iar output-ul fiecărui neuron este calculat de o funcție non-lineară al sumei intrărilor (input-urilor). Conexiunile sunt numite edges. Neuronii și edge-urile au un parametru numit greutate (weight) care ajustează procesul de învățare. Acest parametru crește și scade în funcție de importanța neuronului în rețea. Semnalul se propaghează de la stratul de început (Input Layer) până la stratul de ieșire (Output layer).

Rețelele neuronale sunt folosite datorită trăsăturii unice pe care o au, aceea de a învăța pe baza de exemple, folosindu-se de experiență acumulată anterior pentru a obține rezultate mai bune. Acest proces este numit “Antrenarea” unei rețele.

### 2.2. Perceptroni

Pentru a înțelege mai bine rețelele neuronale, voi explica cel mai simplu tip de neuron artificial, perceptronul. Perceptronul a fost dezvoltat în anii 1950 și 1960 de omul de știință Frank Rosenblatt, inspirat la rândul lui de munciile lui Warren McCulloch și Walter Pitts. În zilele noastre, este mai comun să folosim alți tipuri de neuroni artificiali, cel mai comun fiind *neuronul sigmoid* (*Sigmoid Neuron*). Vom ajunge la neuronii sigmoid în curând, dar ca să înțelegem cum funcționează, trebuie mai întâi să înțelegem cum funcționează un neuron de tip perceptron.

Perceptronul funcționează prin a lua mai multe input-uri  $x_1, x_2, \dots$ , și a produce un singur rezultat (output) binar:

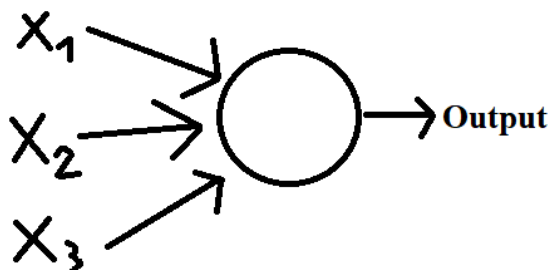


Figura 2.1.: Reprezentarea grafică a unui perceptron

În exemplu arătat mai sus, perceptronul are 3 input-uri  $x_1, x_2$  și  $x_3$ . În general, el poate avea mai multe sau mai puține input-uri. Rosenblatt a introdus o regulă simplă pentru a calcula rezultat-ul. A introdus greutatea  $w_1, w_2, \dots$ , unde  $w_1, w_2, \dots, \mathbb{R}$ . Rezultatul neuronului este calculat prin următoarea ecuație:  $\sum_j w_j x_j \geq \text{threshold}$ , unde  $\text{threshold}$  este o valoare aleasă de noi. În termeni algebrici, acesta va arăta ca:

$$\text{rezultat} = \begin{cases} 0 & \text{dacă } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{dacă } \sum_j w_j x_j > \text{threshold} \end{cases}$$

Reprezentarea completă grafică a unui perceptron va arăta ca atare:

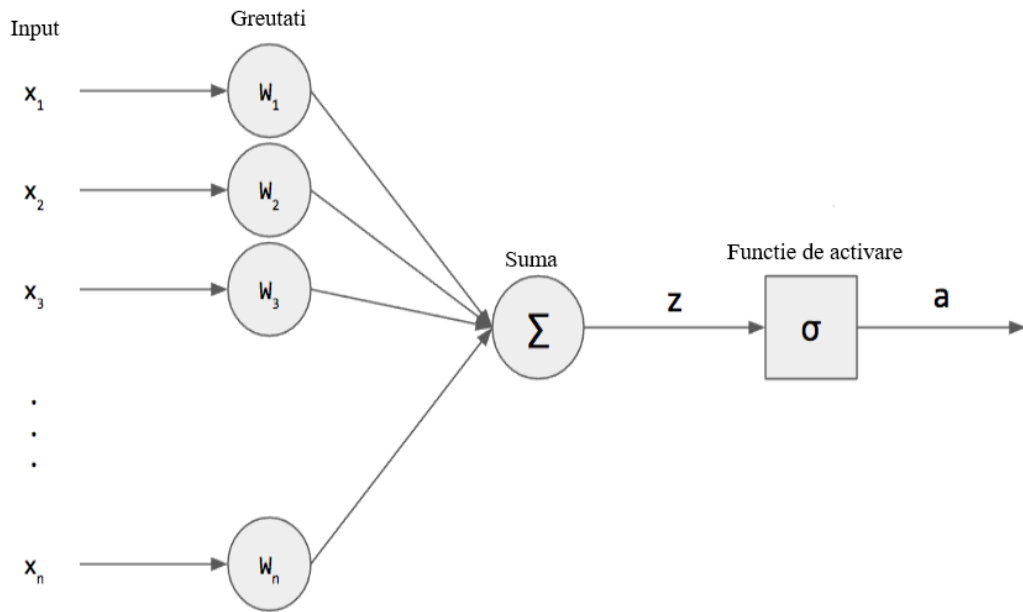


Figura 2.2.: Reprezentarea completă a unui Perceptron

Este evident că un singur neuron nu poate replica modelul de gândire și de analizare al unui om, dar ne putem folosi de mai mulți neuroni pentru a forma o rețea neuronală care poate să facă chiar și cele mai subtile decizii.

### 2.3. Rețea simplă de neuroni cu mai multe straturi

Hai să considerăm un o rețea mai complexă de neuroni:

În această rețea, primul strat de neuroni - pe care îl vom numi primul strat de perceptroni, ia 3 decizii simple în funcție de evidența primită de la input-uri. Al doilea strat de perceptroni ia decizii în funcție de rezultatul deciziilor primilor neuroni. În așa fel, al doilea strat poate lua decizii mult mai complexe decât primul strat de neuroni. Acest lucru permite Rețelei să ia decizii mult mai complexe și la un nivel mult mai abstract. La fel, neuronul de pe ultimul strat ia o decizie mai complexă decât neuronii de pe stratul anterior. Acest lucru ne permite să luăm decizii foarte complexe îmbinând mai multe mini-decizii.

Poate vă întrebați de ce apare de parcă fiecare perceptron ar avea mai multe output-uri deși în definiția perceptronului este că are un singur rezultat. Cu adevărat, perceptron-ul are

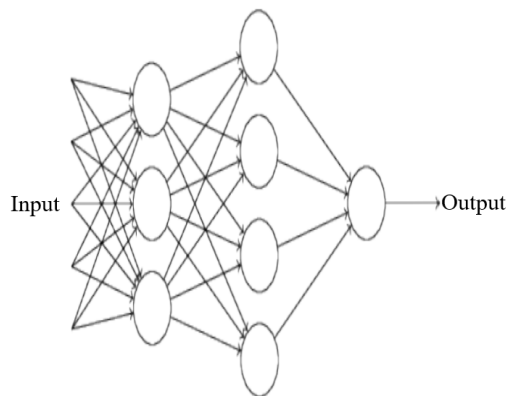


Figura 2.3.: Reprezentarea unei rețele simple

un singur rezultat, dar trasăm mai multe linii ca să indicăm faptul că rezultatul perceptronului este folosit ca input pentru următoarele neuroane.

Pentru a face treabă mai ușoară, hai să simplificăm modul în care descriem perceptronele. Condiția  $\sum_j w_j x_j > \text{threshold}$  este cam urâtă, dar putem face două schimbări în notații pentru a arăta mai bine. Prima ar fi să scriem  $\sum_j w_j x_j$  ca și când ar fi produsul scalar  $w \cdot x = \sum_j w_j x_j$  unde  $w$  și  $x$  sunt doi vectori al căror componente sunt greutatea și input-urile. A doua schimbare ar fi să mutăm threshold-ul în cealaltă parte a inegalității și să o numim *bias-ul perceptronului* (pe care îl notăm cu litera  $b$ ), unde  $b = -\text{threshold}$ . Folosind bias-ul în loc de threshold, regula perceptronului poate fi scrisă ca:

$$\text{rezultat} = \begin{cases} 0 & \text{dacă } w \cdot x + b \leq 0 \\ 1 & \text{dacă } w \cdot x + b > 0 \end{cases}$$

## 2.4. Neuroni de tip Sigmoid

Să programezi un algoritm care învață singur sună cumplit de greu. Dar cum am putea crea un astfel de algoritm pentru rețeaua noastră neuronală? Să supunem că avem o rețea de neuroni de tip perceptron pe care am vrea să o învățăm să rezolve o anumită problemă.

De exemplu: input-urile către rețea ar putea fi pixelii unei imagini scanate cu o cifră scrisă de mână. Noi vrem ca rețeaua să învețe greutățile și bias-urile corecte astfel încât rezultatul să clasifice corect o cifră. Ca să vedem cum funcționează procesul de învățare (des numit și antrenare), vom face o mică schimbare în greutatea/bias-ului unui neuron. Ce am vrea noi este ca această mică schimbare să aibe doar un mic impact asupra rezultatului.

Hai să considerăm faptul că rețeaua noastră a clasificat din greșeală o imagine ca și când ar fi "8" când ea trebuia să fie în realitate "9". Trebuie să găsim o modalitate care ne permite să facem o mică schimbare în greutățile și bias-ul sistemului astfel încât să clasifice bine imaginea și rețeaua să tot învețe. Problema cu neuronii de tip perceptron este că rezultatul lor este binar, adică ori 0/1. Schimbarea aceasta ar cauza tot sistemul să se comporte diferit. Deși acum imaginea este clasificată corect, cel mai probabil s-a schimbat radical comportamentul rețelei. Avem nevoie de o metodă mai "controlată" pentru a face astfel de schimbări

Rezolvarea problemei noastre este introducerea unui nou tip de neuron, acela fiind neuronul sigmoid. Neuronii de tip sigmoid sunt foarte asemănători cu perceptronii, singura diferență fiind faptul că pot avea ca rezultat un număr real (Output  $\in \mathbb{R}$ )



La fel ca un perceptron, neuronul sigmoid are input-uri  $x_1, x_2, \dots$ . În loc ca acestea să fie 0 sau 1, acestea pot avea ca valoare orice număr între 0 și 1. De exemplu: 0.638... este un input valid pentru un neuron sigmoid. Încă ceva asemănător cu perceptronul, este că în afara de input-uri  $w_1, w_2, \dots$ , el mai are și un bias,  $b$ . Diferența fiind că rezultat-ul nu este 0 sau 1, în schimb el este  $\sigma(wx + b)$ ,  $\sigma$  este funcția sigmoidă, care este definită de:

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

Că să fim mai expliciti, rezultatul unui neuron de tip sigmoid cu inputurile  $x_1, x_2, \dots$ , greutatea  $w_1, w_2, \dots$  și bias-ul  $b$  este:

$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}$$

Un lucru de reținut este că  $\sigma$  este des numit și funcția logistică (logistic function), și acest tip de neuron sunt numiți neuroni logistici. Dar pentru acest atestat, îi vom numi exclusiv neuroni sigmoizi.

Pentru a înțelege similaritatea cu modelul perceptron, hai să supunem că  $z \equiv w \dots x + b$  este un număr pozitiv mare. Atunci  $e^{-z} \approx 0$ . Atunci, când  $z = w \dots x + b$  este foarte negativ, funcția sigmoidă aproximează foarte aproape de modelul perceptron.

Deși forma algebrică este foarte diferită, tot ce contează este cum arată graficul trasat:

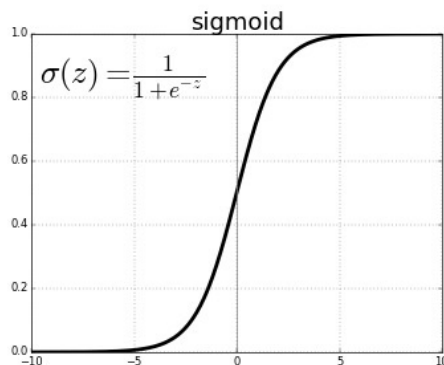


Figura 2.4.: Reprezentarea funcției sigmoid

## 2.5. Arhitectura unei rețele neuronale

Ca să descriem rețeaua neuronală care recunoaște cifre scrise de mână, trebuie să vorbim despre diferitele părți ale unei rețele. Hai să supunem că avem o rețea:

Cum am menționat mai devreme, primul strat este numit strat-ul input. Ultimul strat este numit strat-ul output (cel care ne dă rezultatul). Straturile din mijloc sunt numite straturi ascunse, deoarece neuronii din straturile acestea nu sunt nici input-uri și nici output-uri. Rețeaua din figura 2.5 are două straturi ascunse. O chestie de notat este că astfel de rețele sunt des numite și MultiLayer Perceptron (MLP-uri), deși neuronii din care sunt alcătuite sunt de tip sigmoid nu perceptron.

Arhitectura strat-urilor input și output sunt dese ori foarte clare. De exemplu, vrem să determinăm dacă cifra dintr-o imagine este 9 sau nu. Cel mai simplu mod de a face asta este de a encoda fiecare pixel al imaginii cu un neuron input. Dacă imaginea este alb/negru de dimensiunea 28x28 (ca în cazul MNIST), vom avea  $28 * 28 = 784$  neuroni de intrare. Strat-ul de output va conține un singur neuron al cărui rezultat va fi între 0 și 1. Orice valoare peste 0.5 indică faptul că cifra este 9.

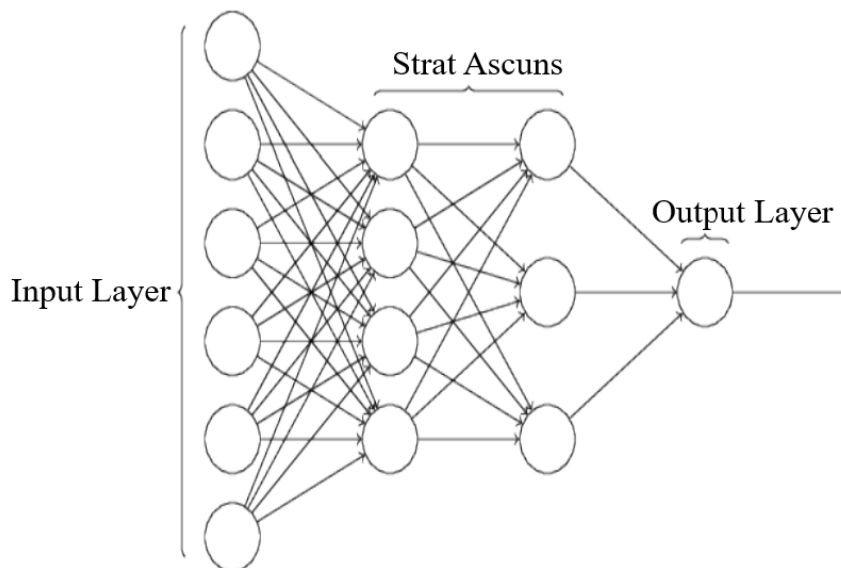


Figura 2.5.: Reprezentarea unei rețele neuronale

## 2.6. Feed-Forward Neuronal Network

Strat-urile ascunse nu sunt atât de clare precum cele input/output. Există foarte multe tipuri de rețele neuronale dezvoltate de către cercetători pe parcursul timpului, care schimbă comportamentul strat-urilor ascunse pentru a obține rezultatul dorit. Până acum, am discutat de rețeaua neuronală care ia output-ul unui strat și-l folosește ca input în următorul strat. Astfel de rețele sunt numite feedforward neural networks. Asta înseamnă că nu există nicio buclă (loop) în rețea. Informația este mereu propagată în față și niciodată în spate. Dacă am fi avut bucle într-o rețea de acest fel, am ajunge cu rezultate în care funcția  $\sigma$  va fi dependentă de input.

## 2.7. Rețele Neuronale Convoluționale

Rețele neuronale convoluționale folosesc o arhitectură specială care este perfect adaptată pentru clasificarea imaginilor. Folosind această arhitectură, rețelele neuronale convoluționale sunt foarte rapide de antrenat, ceea ce ne permite să antrenăm modele foarte adânci (cu foarte multe straturi ascunse), care sunt foarte bune la clasificat imagini. În zilele noastre, CNN-urile sau o variantă apropiată sunt folosite pentru clasificarea imaginilor.

Rețele neuronale convoluționale folosesc 3 idei pe care le vom dezvolta: câmpuri receptive locale (local receptive fields), greutatea comună și pooling.

### 2.7.1. Câmpuri receptive locale

Intr-un strat conectat complet, input-urile erau desenate ca și când ar fi o linie verticală. Într-o rețea convoluțională, este mai ușor să ne gândim la input-uri (acei 28x28 pixeli) ca un patrat de neuroni, al căror valoare corespunde cu intensitatea pixelilor.

Ca de obicei, vom conecta pixelii de intrare la un strat de neuroni ascunși. Dar nu vom face conexiuni de la fiecare pixel de intrare către fiecare neuron ascuns. În schimb, vom face conexiuni numai în regiuni mici ale imaginii de intrare. Ca să fim mai precisi, vom conecta

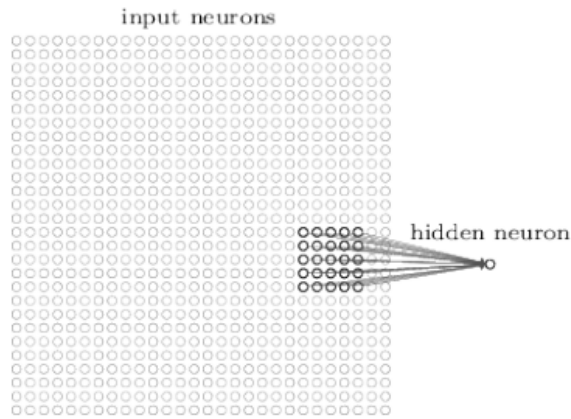


Figura 2.6.: Neuroni input într-o rețea convoluțională

fiecare neuron al primului strat ascuns cu , sa zicem, o regiune de 5x5 pixeli. Deci, pentru un neuron ascuns, vom avea o conexiune care arata ca cea din figura 2.6:

Regiunea este numită câmpul receptiv local. Fiecare conexiune învață o greutate. Neuronul ascuns învață și un bias. Un mod de a privi acest lucru este că neuronul ascuns încearcă să învețe cum să analizeze cel mai bine câmpul lui local.

### 2.7.2. Greutăți și bias-uri comune

Am menționat mai devreme că fiecare neuron ascuns are un bias și 5x5 greutăți conectate pentru câmpul lui local. Ce nu am menționat este că vom folosi aceeași greutate pentru toți 24x24 neuroni. În alte cuvinte, pentru  $j$ , și al  $k$ -lea neuron, rezultatul este

$$\sigma(b + \sum_{l=0}^4 \sum_{m=0}^4 w_{l,m} a_{j+l,k+m})$$

Aici,  $\sigma$  este funcția de activare a neuronului - precum funcția sigmoidă pe care am descris-o mai sus.  $b$  este valoarea comună pentru bias.  $w_{l,m}$  este o matrice de 5x5 greutăți, iar  $a_{x,y}$  este folosită pentru a denota activarea la poziția  $x,y$

### 2.7.3. Strat de pooling

Rețelele neuronale convoluționale mai conțin și un strat de tip pooling. Strat-urile pooling sunt folosite în general fix după ce avem un strat convoluțional. Strat-ul pooling simplifică informația primită de la rezultatul strat-ului convoluțional.

## 3. Implementarea Algoritmului

### 3.1. Implementarea Algoritmului in python

În această secțiune, vom implementa algoritmul folosind limbajul python și IDE-ul Spyder3 într-un mediu virtual bazat pe ultima versiune de Anaconda.

#### 3.1.1. Importarea Algoritmilor

```
1 #Importarea librariilor
2 import tensorflow as tf
3 import keras
4 from keras.datasets import mnist
5 from keras.models import Sequential
6 from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
7 from keras import backend as K
8 import numpy as np
```

În secțiunea aceasta am importat librările pe care le vom folosi: Tensorflow, keras, setul de date MNIST și numpy.

#### 3.1.2. Împărțirea setului de date

Pentru a ne antrena modelul, vom împărți setul de date într-un grup pentru antrenare și unul pentru test. Setul de antrenare va conține 60000 de poze cu cifre scrise de mână, iar cel de test/verificare va conține 10000 de poze.

```
1 (x_train, y_train), (x_test, y_test) = mnist.load_data()
```

#### 3.1.3. Preprocesarea Datelor


x\_train și x\_test sunt de forma 28x28. Vă trebui să adaugăm încă un rând pentru rețeaua noastră neuronală.

```
1 x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
2 x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
3 input_shape = (28, 28, 1)
```

y\_train și y\_test conțin doar un număr, care corespunde cu ce cifra este în imaginea respectivă. Ar trebui să convertim vectorul în unul de categorii. Pentru a face acest lucru, vom folosi o metoda numită One Hot Encoding. Figura de mai jos reprezintă grafic ceea ce face procesul de One-Hot Encoding.

Poza fiind alb și negru ( numită și grayscale ), pixeli pot avea o valoare între 0 și 255. După cum știm, neuronii Sigmoid nu pot accepta că input decât o valoare între 0 și 1. De aceea, vom împărți valorile pixelilor cu 255 pentru a normaliza valoarea. Pentru a face acest lucru

id	color			
1	red			
2	blue			
3	green			
4	blue			



id	color_red	color_blue	color_green
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0

posibil, trebuie să-i menționăm programului că valorile sunt de tip float32. Altfel, am primi un rezultat de 0 pentru toți pixelii care nu sunt 255 (împărțirea unui număr întreg va da doar partea întreagă a numărului)

```

1 x_train = x_train.astype('float32')
2 x_test = x_test.astype('float32')
3 x_train /= 255
4 x_test /= 255

```

### 3.1.4. Pregatirea pentru non-liniaritate

Vom declara batch size-ul si numerele de epoci pe care le va parcurge algoritmul. Cele mai bune rezultate le-am avut folosind valorile de 128 pentru batch size si 10 pentru numarul de epoci.

```

1 batch_size = 128
2 epochs = 10

```

### 3.1.5. Rețeaua Neuronală Convoluțională

Vom folosi o rețea neuronală convoluțională sequentiala. Pentru acest lucru, ne vom folosi de librăria Keras. Vom declara obiectul model ca fiind Sequential.

```

1 model = Sequential()

```

După aceea, vom adăuga straturile de neuroni. Primul strat va fi un strat convolutional, care folosește ca funcție de activare ReLU. Primul strat de neuroni ascunși va fi de 64 de neuroni, folosind ca funcție de activare tot ReLU.

```

1 model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape =
    input_shape))
2 model.add(Conv2D(64, (3, 3), activation= 'relu'))

```

După straturile convoluționale, vom adăuga un strat de tip pooling, mai precis MaxPooling.

```

1 model.add(MaxPooling2D(pool_size=(2, 2)))

```

Pentru a evita overfitting-ul, vom folosi o rată de dropout de 25%. Acest lucru înseamnă că la fiecare iterație, greutatea fiecărui neuron are o șansă de 25% de a fi inițializată cu 0. După acestea, vom nivela rețeaua.

```

1 model.add(MaxPooling2D(pool_size=(2, 2)))
2 model.add(Dropout(0.25))
3 model.add(Flatten())

```

Următorul pas va fi de a adaugă niște strat-uri dense (toți neuronii sunt conectați între ei). Pentru a reduce timpul de antrenament, am decis să aduagăm doar 2 strat-uri alcătuite din 256 de neuroni. Încă o dată, pentru a evita overfitting-ul, am selectat o șansă de dropout de 50%.

```
1 model.add(Dense(256, activation='relu'))
2 model.add(Dense(256, activation='relu'))
3 model.add(Dropout(0.5))
```

În final, vom avea un strat output cu 10 neuroni, fiecare neuron corespunzând cu câte o cifră. De exemplu, dacă primul neuron are valoarea de 0.9, înseamnă că modelul crede că imaginea este 0 (cu o siguranță de 90%).

```
1 model.add(Dense(10, activation = 'softmax'))
```

După toate acestea, vom compila modelul, folosind ca funcție de pierdere (Loss Function) categorical Cross-Entropy-ul. Vom antrena modelul folosind algoritmul Stochastic Gradient Descend, și ne vom concentra pe acuratețea modelului. După ce am compilat modelul, îl vom antrena cu setul nostru de date.

```
1 model.compile(loss=keras.losses.categorical_crossentropy, optimizer='SGD', metrics=['
    accuracy'])
2 hist = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, verbose=1,
    validation_data=(x_test, y_test))
```

### 3.1.6. Evaluarea Modelului

Pentru a evalua modelul, îl vom testa asupra setului nostru test.

```
1 score = model.evaluate(x_test, y_test, verbose=0)
2 print('Test loss:', score[0])
3 print('Test accuracy:', score[1])
```

# Anexe

## A. Fișiere sursă

```
1 #Codul Sursa
2 #Scris de Ramadan Omar, elev al colegiului National "I.L. Caragiale" Bucuresti
3
4 #Importarea librariilor
5 import tensorflow as tf
6 import keras
7 from keras.datasets import mnist
8 from keras.models import Sequential
9 from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
10 from keras import backend as K
11 import numpy as np
12
13 #Impartirea setului de date intr-un set de antrenament si un set de test
14 (x_train, y_train), (x_test, y_test) = mnist.load_data()
15
16
17 #Preprocesarea Datelor
18
19 x_train = x_train.reshape(x_train.shape[0], 28, 28, 1) #Am adaugat inca un rand
20 x_test = x_test.reshape(x_test.shape[0], 28, 28, 1) #Pentru CNN
21 input_shape = (28, 28, 1)
22
23
24 #Din Vector in Matrice Binara
25 #One Hot Encoding
26 y_train = tf.keras.utils.to_categorical(y_train, 10) # 10 Cifre
27 y_test = tf.keras.utils.to_categorical(y_test, 10) # 10 Cifre
28
29 #Scalarea pixelilor in [0,1]
30 x_train = x_train.astype('float32')
31 x_test = x_test.astype('float32')
32 x_train /= 255
33 x_test /= 255
34
35 print('x_train shape:', x_train.shape)
36 print(x_train.shape[0], 'train samples')
37 print(x_test.shape[0], 'test samples')
38
39 #Pregatirea pentru NonLiniaritate
40
41 batch_size = 128
42 epochs = 10
43
44 #Reteaua Neuronala Convolutionala
45 model = Sequential()
46 model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape =
    input_shape))
47 model.add(Conv2D(64, (3, 3), activation= 'relu'))
48
49 model.add(MaxPooling2D(pool_size=(2, 2)))
50 model.add(Dropout(0.25))
51 model.add(Flatten())
52
53 model.add(Dense(256, activation='relu'))
54 model.add(Dense(256, activation='relu'))
55 model.add(Dropout(0.5))
```



```
56
57 model.add(Dense(10, activation = 'softmax'))
58
59 model.compile(loss=keras.losses.categorical_crossentropy,optimizer='SGD',metrics=['
    accuracy'])
60
61 #Antrenarea Modelului
62 hist = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, verbose=1,
    validation_data=(x_test, y_test))
63 print("Modelul a fost antrenat cu succes")
64
65 model.save('cnn1.h5')
66 print("Modelul a fost salvat ca cnn.h5")
67
68
69 score = model.evaluate(x_test, y_test, verbose=0)
70 print('Test loss:', score[0])
71 print('Test accuracy:', score[1])
```

Listarea A.1: Cod Python – fișier complet