

Clasificarea imaginilor folosind rețele neuronale convoluționale

Rețelele neuronale convoluționale (Convolutional Neural Networks, prescurtat CNN) sunt clasificatori special concepuți pentru lucrul cu imagini. Structura acestora facilitează clasificarea datelor cu un număr mare de parametri, deoarece aceste tipuri de rețele permit prelucrarea eficientă a acestor categorii de date.

CNN nu au o arhitectură sau o structură prestabilită, aceasta se adaptează la tipul de imagine care se prelucrează sau la natura obiectelor care trebuie clasificate. Fig 1. ilustrează un exemplu de astfel de arhitectură.

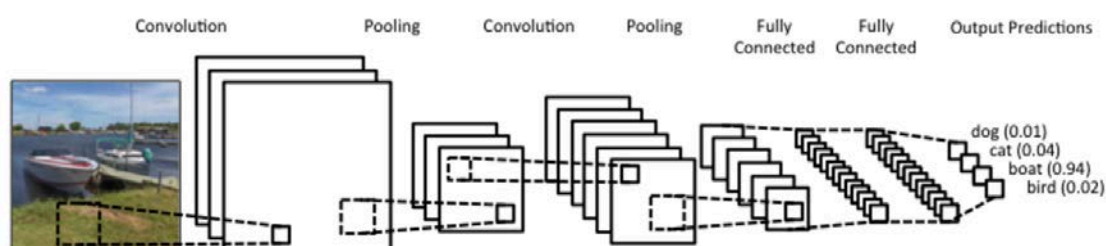


Fig. 1. Exemplu de CNN. La intrare se trimite o imagine, care este supusă operațiilor corespunzătoare straturilor de convoluție (*convolution*) și agregare (*pooling*). Rezultatul se propagă prin straturi complet conectate (*fully connected*). Ieșirea rețelei constă într-un vector de probabilități ce încadrează conținutul imaginii într-o mulțime de clase preexistente.

În general, CNN au două componente:

- I. O componentă de extragere a trăsăturilor din imagini. Aceasta se compune din mai multe straturi, care pot fi:
 - Straturi de convoluție – în cadrul acestora se realizează convoluția dintre imaginea de intrare și un filtru (matrice $n \times n$ de dimensiuni mici). Spre deosebire de straturile din rețelele neuronale clasice, în straturile de convoluție numărul de conexiuni se reduce la dimensiunea filtrelor utilizate, și nu la cea a imaginii de la intrare. Straturile de convoluție produc una sau mai multe *feature maps*, imagini care conțin anumite trăsături sau caracteristici ale imaginii de la intrare (Fig 2). Valorile filtrelor cu care se face convoluția în cadrul unui anumit strat constituie ponderile conexiunilor din acel strat. La antrenarea rețelei, aceste ponderi se modifică după o logică similară cu cea a rețelelor clasice, ideea fiind de a minimiza eroarea dintre rezultatul de la ieșire dorit și cel obținut efectiv.

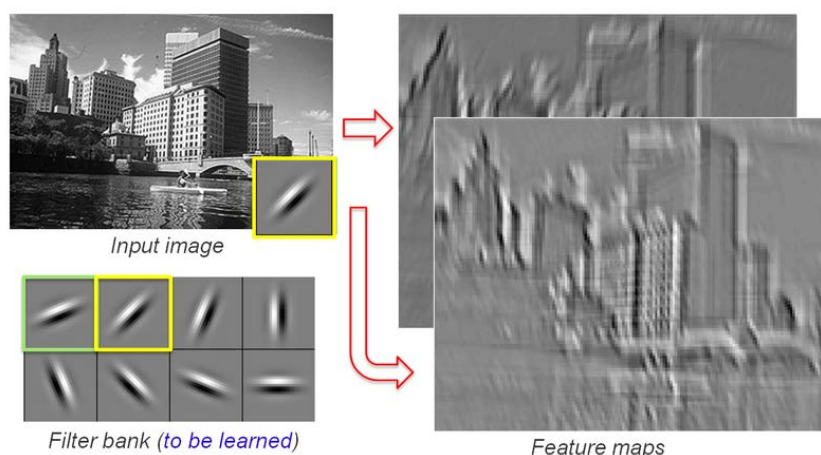


Fig. 2. Imaginea de la intrare și feature map-urile care rezultă în urma aplicării convoluției cu anumite filtre

- Straturi de agregare (*engl: pooling*) – în cadrul acestor straturi se realizează o reducere a dimensiunii feature map-urilor de la intrare. Astfel, operația de agregare 2x2 aplicată pe un feature map de dimensiune 64x64 generează un feature map de dimensiune 32x32. Reducerea dimensiunii pentru fiecare grup 2x2 de pixeli din feature map-ul de la intrare se poate realiza prin mai multe metode: de ex. Se determină maximul dintre cei 2x2 pixeli (*max pooling*), se poate face suma lor (*sum pooling*) sau se poate determina valoarea lor medie (*average pooling*). Fig 3. ilustrează rezultatul operațiilor *max pooling* și *sum pooling*.

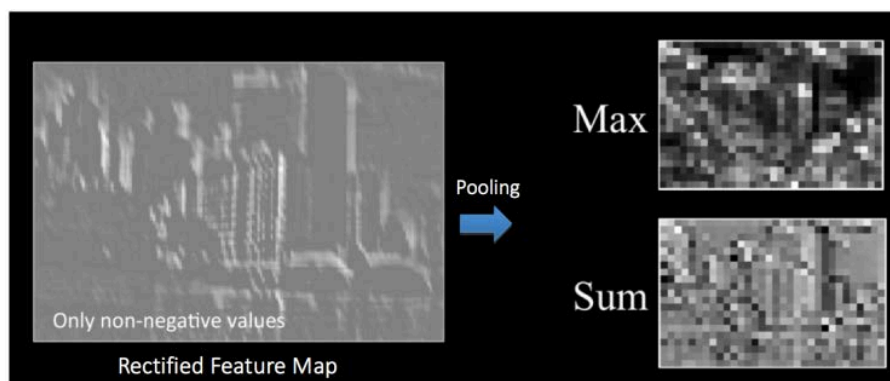


Fig. 3. Rezultatul operației de agregare, care constă în reducerea dimensiunii unui feature map.

Straturile de convoluție și agregare sunt supuse unei funcții de activare cu rolul de a asigura comportamentul neliniar al rețelei. Ca funcție de activare, de cele mai multe ori se folosește ReLU (Rectified Linear Unit). Fig. 4 ilustrează rezultatul aplicării funcției ReLU pe un feature map.

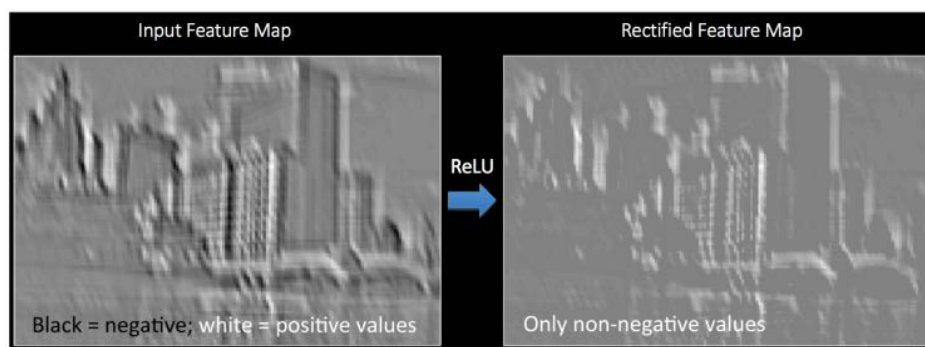


Fig. 4. Rezultatul operației de agregare, care constă în reducerea dimensiunii unui feature map.

- II. O componentă complet conectată (*fully connected*), în cadrul căreia se realizează clasificarea propriu-zisă. Această componentă este o rețea neuronală clasică, la intrarea căreia se furnizează feature map-urile, și la ieșirea căreia se aplică funcția de activare *softmax*. Ieșirea acestei componente este un vector de probabilități cu un număr de componente egal cu numărul de clase. Fiecare componentă a vectorului reprezintă probabilitatea ca imaginea de la intrare să se încadreze în clasa corespunzătoare.

Exemplificăm aspectele menționate până acum în Fig. 5.

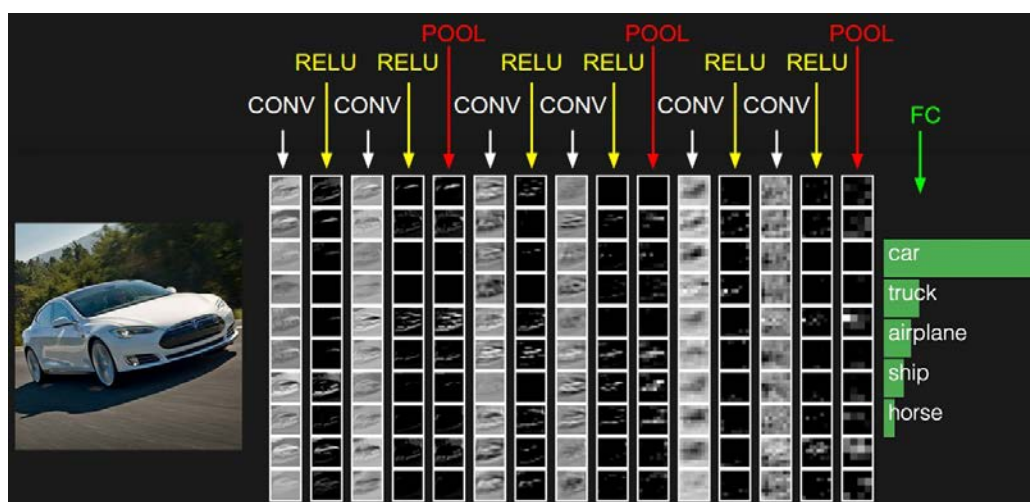


Fig. 5. Rezultatele generate de o rețea neuronală de convoluție. Se observă multiple straturi de convoluție și agregare, precum și etapele în care se aplică funcția de activare ReLU. Imaginile intermediare sunt feature map-urile generate în cadrul fiecărui pas.

Imaginea de la intrare conține un obiect care trebuie încadrat într-una din cele 5 clase preexistente (categoriile vizibile în extremitatea dreaptă a imaginii din Fig. 5). Imaginea este supusă mai multor operații de convoluție (CONV), agregare (POOL) și rectificare (RELU), de fiecare dată generându-se mai multe feature map-uri. Acestea sunt imagii care conțin trăsături semnificative ale obiectelor din imaginea inițială. Trăsăturile sunt apoi supuse unui proces de

clasificare prin intermediul unei rețele neuronale complet conectate (FC), rezultând probabilitățile ca imaginea să aparțină fiecărei categorii.

Aplicație:

Următoarea secvență de cod implementează o CNN cu un strat de convoluție, unul de agregare (*pooling*) și unul fully-connected. Scopul este de a proiecta o rețea care să recunoască cifre în imagini grayscale, folosind fișierele din setul de date din aplicația care însoțește documentația pentru a antrena rețeaua. Listăm codul care implementează rețeaua în întregime, apoi îl vom analiza pe porțiuni.

```
class SimpleCNN(nn.Module):
    def __init__(self, imgWidth, imgHeight):
        super(SimpleCNN, self).__init__()

        inputWidth = imgWidth
        inputHeight = imgHeight
        nrConvFilters = 3
        convFilterSize = 5
        poolSize = 2
        outputSize = 10

        self.convLayer = nn.Conv2d(1, nrConvFilters, convFilterSize)
        self.poolLayer = nn.MaxPool2d(poolSize)
        fcInputSize = (inputWidth - 2*(convFilterSize // 2)) * (inputWidth -
2*(convFilterSize // 2)) * nrConvFilters // (2 * poolSize)
        self.fcLayer = nn.Linear(fcInputSize, outputSize)

    def forward(self, input):
        output = self.convLayer(input)
        output = self.poolLayer(output)
        output = F.relu(output)
        output = output.view([1, -1])
        output = self.fcLayer(output)
        return output

    def train(self, images, labels):
        lossFunc = nn.CrossEntropyLoss()
        nrEpochs = 10
        learnRate = 0.01
        optimizer = torch.optim.SGD(self.parameters(), learnRate)













        for epoch in range(nrEpochs):

            for image, label in zip(images, labels):

                optimizer.zero_grad()
                predicted = self.forward(image.unsqueeze(0))
                loss = lossFunc(predicted, label.unsqueeze(0))
                loss.backward()
                optimizer.step()

            print('Epoch', epoch, 'loss', loss.item())
```

Imaginile de antrenare constau în 1000 imagini 28x28 grayscale, câte 100 imagini pentru fiecare cifră. Clasa în care se încadrează fiecare imagine este reprezentată printr-un întreg din mulțimea $\{0, \text{nrClase}-1\}$. Pentru fiecare imagine, acest întreg se numește etichetă (*label*). În cazul de față, sunt 10 clase (cifrele '0', '1', '2', ..., '9') reprezentate prin label-urile 0, 1, 2, ..., 9. Este o coincidență faptul ca label-urile și denumirile claselor coincid. Aceeași metodă de codificare s-ar aplica pentru oricare alte clase, indiferent de semnificația lor (nu doar pentru cele care reprezintă cifre). Așadar, datele de antrenare sunt perechi formate din imagini și label-urile care indică cifra conținută:

Images	Labels
	0
	0
	0
...	...
	1
	1
	1
...	...
	2
	2
	2
...	...
...	...
...	...
	9
	9
	9
...	...

Semnificația liniilor de cod este următoarea:

```
class SimpleCNN(nn.Module):
```

Rețeaua neuronală este organizată sub forma unui modul. Un modul în Pytorch implementează o serie de operații predefinite și poate la rândul său conține mai multe submodule.

```
def __init__(self, imgWidth, imgHeight):
```

Constructorul primește ca parametri dimensiunile unei imagini.

```
inputWidth = imgWidth  
inputHeight = imgHeight
```

Dimensiunile datelor de intrare. Este nevoie de acestea pentru a defini dimensiunea stratului fully-connected, cel convoluțional va ști să deducă dimensiunea din tensorii pe care îi primește ca input.

```
nrConvFilters = 3  
convFilterSize = 5  
poolSize = 2
```

Trei parametri care servesc la definirea straturilor speciale ale rețelelor convoluționale.

```
outputSize = 10
```

Dimensiunea stratului de ieșire al rețelei = numărul de clase

```
self.convLayer = nn.Conv2d(1, nrConvFilters, convFilterSize)
```

Stratul convoluțional: primul parametru se referă la numărul de valori ale pixelilor imaginilor de intrare (i.e. numărul de valori de culoare, *color channels*). Întrucât lucrăm cu imagini grayscale, acest parametru are valoarea 1. Stratul convoluțional generează 3 feature maps aplicând 3 filtre convoluționale de dimensiune 5x5 pe imaginile de intrare.

```
self.poolLayer = nn.MaxPool2d(poolSize)
```

Stratul de agregare: reduce dimensiunea feature map-urilor de intrare de 2 ori. **Max** se referă la faptul că reducerea se face alegând valoarea maximă din fiecare grup de pixeli ai feature map-urilor de intrare.

```
fcInputSize = (inputWidth - 2*(convFilterSize // 2)) * (inputHeight - 2*(convFilterSize // 2)) * nrConvFilters // (2 * poolSize)
```

```
self.fcLayer = nn.Linear(fcInputSize, outputSize)
```

Stratul fully connected: dimensiunea de intrare a acestui strat se determină pe baza rezultatelor operațiilor de convoluție și agregare anterioare, astfel:

Stratul convoluțional generează 3 feature map-uri folosind filtre 5x5. În urma aplicării filtrelor pe imaginea de la intrare de dimensiune 28x28, rezultă 3 feature maps de dimensiune 24x24. Se pierde câte 2 pixeli din fiecare extremitate a imaginii inițiale deoarece se realizează convoluția cu un filtru 5x5, fără padding sau alte operații suplimentare. Pe aceste filtre se aplică o operație de agregare 2x2, care reduce dimensiunea feature map-urilor de 2x pe fiecare dimensiune. Așadar, dimensiunea stratului fully connected va fi $12 \times 12 \times 3 = 432$. Dimensiunea datelor de la ieșire este egală cu nr. de clase (în cazul nostru, 10).

```
def forward(self, input):
    output = self.convLayer(input)
    output = self.poolLayer(output)
    output = F.relu(output)
    output = output.view([1, -1])
    output = self.fcLayer(output)
    return output
```

Metoda care realizează propagarea înainte. În ordine, se realizează următoarele operații:

- se propagă imaginea prin stratul de convoluție și se generează feature map-uri la ieșire.
- pe aceste feature map-uri se aplică o agregare 2x2. Fiecare grup de 2x2 valori din feature map-ul inițial se reduce la o singură valoare, egală cu maximumul celor patru.
- pe rezultatul anterior se aplică funcția de activare ReLU.
- se "aplatizează" rezultatul anterior, care dintr-un tensor de dimensiune [12,12,3] devine un tensor de dimensiune [1, 432].
- se propagă rezultatul anterior prin stratul fully connected
- valorile de ieșire constau într-o mulțime de 10 valori numerice (câte una pentru fiecare clasă) care, dacă ar fi normalizate, ar indica probabilitățile ca imaginea de la intrarea metodei **forward** să se încadreze în fiecare din cele 10 clase.

```
def train(self, images, labels):
```

Această metodă realizează antrenarea rețelei. Algoritmul de antrenare folosește imaginile de antrenare (images) pentru care se cunosc clasele în care se încadrează (labels).

```
lossFunc = nn.CrossEntropyLoss()
```

Funcția cu care se evaluează eroarea dintre valorile furnizate de rețeaua neuronală și labelurile din setul de date de antrenare.

```
nrEpochs = 10
```

Numărul de epoci – o epocă este o etapă din faza de antrenare a rețelei în cadrul căreia întregul set de date de antrenare a fost folosit pentru ajustarea parametrilor rețelei. Așadar, în faza de antrenare rețeaua "vede" datele de antrenare de **nrEpochs** ori.

```
learnRate = 0.01
```

Rata de învățare = parametru care controlează măsura în care se modifică ponderile rețelei în faza de antrenare. Cu cât `learnRate` este mai mare, cu atât parametrii rețelei variază mai multe de la o faza de antrenare la alta.

```
optimizer = torch.optim.SGD(self.parameters(), learnRate)
```

Metoda de optimizare este algoritmul de ajustare al ponderilor rețelei în urma apariției unei erori între valoarea furnizată de rețea și cea din setul de date. În cazul nostru, metoda folosită se numește Stochastic Gradient Descent (SGD).

```
for epoch in range(nrEpochs):
```

Antrenarea se realizează pe parcursul mai multor epoci.

```
or image, label in zip(images, labels):
```

La fiecare epocă, întregul set de date de antrenare va fi supus procesărilor din faza de antrenare a rețelei.

```
optimizer.zero_grad()
```

Optimizarea presupune căutarea valorilor ponderilor rețelei care minimizează eroarea generată de rețea. Pentru aceasta, metoda de căutare se folosește de variațiile erorii în raport cu ponderile rețelei, exprimate prin componentele gradientului erorii în raport cu ponderile. Aceste componente ale gradientului trebuie resetate la începutul procesului de re-ajustare a ponderilor.

```
predicted = self.forward(image.unsqueeze(0))
```

predicted este rezultatul generat de rețeaua neuronală pentru imaginea dată ca parametru de intrare. În cazul de față **predicted** este un tensor ce conține 10 valori, câte una pentru fiecare clasă. Cu cât valoarea aferentă unei clase este mai mare, cu atât se consideră ca rețeaua asociază acea clasă cu datele de intrare într-o măsură mai mare. Metoda **unsqueeze** adaugă o dimensiune suplimentară tensorului (necesară pentru aplicarea corectă a funcției de calcul al erorii)

```
loss = lossFunc(predicted, label.unsqueeze(0))
```

Se determină diferența dintre valoarea generată de rețea și label-ul din setul de date, folosind funcția definită anterior.


```
loss.backward()
```

```
optimizer.step()
```

Se realizează propagarea înapoi a erorii prin rețeaua neuronală și se ajustează ponderile rețelei în sensul minimizării erorii, cu metoda implementată în optimizer.

Sarcini:

1. Pentru fiecare epocă, determinați și afișați eroarea de clasificare a rețelei (nr de imagini clasificate greșit / nr de imagini din acea epocă)
2. Ajustați parametrii rețelei, prin încercări, astfel încât să obțineți o eroare cât mai mică, pentru un număr cât mai mic de epoci.
3. Realizați o evaluare a rețelei:
 - Împărțiți setul de imagini și labels în două părți. Folosiți prima parte pentru a antrena rețeaua, cealaltă va fi setul de date de test.
 - Determinați și afișați eroarea de clasificare în cele două situații: când aceasta se determină folosind datele de antrenare și când se determină folosind datele de test.
4. Implementați o rețea clasică (non-convoluțională, formată doar din straturi nn.Linear) și antrenați-o cu aceleași date (imaginile de această dată vor fi furnizate sub forma de vectori 1D, așadar primul strat din rețea va avea dimensiunea de input 28x28). Comparați convergența acestei rețele cu cea a rețelei convoluționale (i.e. câte epoci sunt necesare pentru ca eroarea să scadă sub o valoare prag)
5. Creați câteva imagini cu cifre și clasificați-le folosind rețeaua proiectată. Atenție, imaginile pe care le primește rețeaua trebuie să fie de aceleași dimensiuni ca și cele ale imaginilor folosite la antrenare (28x28 grayscale).