Laborator 3: Retele Convolutionale - Convolutional Neural Networks (CNNs)

1. Introducere

Laboratorul trecut s-a studiat problema *clasificarii* unei baze de date simple (MNIST) cu ajutorul unei retele de tip perceptron multistrat (Multilayer Perceptron - MLP). Pe parcursul acestui studiu de caz s-au abordat pasii elementari pentru rezolvarea unei astfel de probleme (alegerea arhitecturii, a functiei *loss*, a optimizatorului, a rate-ului de invatare, impartirea bazei de date in *batch-uri* precum si bucla de invatare.

In continuare, solutia usoara din laboratorul trecut va fi extinsa prin folosirea arhitecturilor mult mai puternice de tip retele convolutionale (CNN). De asemenea se vor discuta si alte aspecte utile in utilizarea retelelor neuronale.

2. Motivatie si aspecte generale

Reteaua neuronala folosita anterior a utilizat drept trasaturi de intrare chiar pixelii constituenti ai imaginilor. Desi rezultatele au fost satisfacatoare, acest deznodamant nu ar fi fost la fel de probabil pentru baze de date mai complicate. In general, valorile directe ale *pixelilor nu sunt considerate a fi trasaturi puternice*. Cu acest scop se utilizeaza *extractoare de trasaturi*, precum HoG si LBP, care surprind mai bine informatia utila din fiecare zona de interes sau din jurul fiecarui pixel.

Retelele convolutionale au adus o serie de imbunatatiri in ceea ce priveste algoritmii de *machine learning*. O parte dintre acestea vor fi discutate ulterior. Drept punct de plecare se va face referire la schema generala a unei retele convolutionale:

Schema generala a unei retele convolutionale. Figura preluata de pe kdnuggets

Se pot observa doua zone principale:

- Prima, formata din straturile de tip Convolution si Pooling
- A doua, formata din straturi *Fully Connected*

Prima zona s-a dovedit a avea rolul de extractor de trasaturi. Primele straturi convolutionale extrag informatii de tip contururi, iar acestea devin mai complexe odata cu parcurgerea retelei. S-a observat ca trasaturile lesite dupa prima zona a retelei sunt puternice (desi nu s-a descoperit ce inseamna), in general fiind mai reprezentative decat trasaturile obtinute prin aplicarea algoritmilor HoG sau LBP.

A doua zona, cu straturile *Fully Connected* este practic o retea MLP aplicata trasaturilor extrase de zona convolutionala a retelei.

2.1. Straturile convolutionale

Acest tip de strat este unitatea de baza din noile arhitecturi. Facand o analogie cu procedeele consacrate din prelucrarea imaginilor, un strat convolutional realizeaza o serie de operatii de filtrare liniara pe matricea de la intrarea in strat (pe rand). Aceasta matrice poate fi imaginea in sine, sau rezultatul altui strat, denumit *feature map* (harta de trasaturi). Un fapt relevant in acest caz este ca *nucleul de convolutie* are adancimea matricei de intrare (ex. 3 pentru o imagine RGB). De la modul in sine in care functioneaza stratul convolutional se pot observa niste diferente fata de modul de functionare al MLP, precum si proprietati relevante:

- *sparse interactions* (interactiuni "rare"): spre deosebire de MLP, unde*va fiecare neuron din stratul N interactiona cu toti neuronii din stratul $N + 1$, la un CNN doar o serie de neuroni din stratul curent vor participa la neuronul din stratul urmator (zona denumita campul receptiv, *receptive field*). Acest fapt implica stocarea a mai putini parametri, si a operatii mai putine;
- partajarea parametrilor: nucleul de convolutie este folosit pentru intreaga imagine, deci pondererile care conduc la un anume neuron din stratul urmator sunt mereu aceleasi. Aceasta trasatura este pusa in contrast cu arhitectura MLP, unde fiecare pondere era folosita o singura data, pentru o perechie anume de neuroni;
- echivarianta la translatie a reprezentarilor: se refera la proprietatea ca daca imaginea de intrare sufera o transformare de tip translatie, si harta de trasaturi de iesire va suferi aceeasi modificare.

Toate straturile convolutionale sunt urmate de o functie de activare (in general Rectified Linear Unit - ReLU). Pentru a creea un strat convolutional in PyTorch se foloseste sintaxa:

```
conv_x = nn.Conv2d(in_channels = nr_canale_input, out_channels = nr_canale_output, kernel_size = [linii_filtru,coloane_filtru], stride = [pas_orizontal,pas_vertical], padding = [bordare_linii,bordare_coloane])
```

Forma corecta pentru datele intrare este de tipul `[nr_imag,canale, linii,coloane]`.

!Atentie: cel mai frecvent, forma unei imagini obisnuite pentru alte biblioteci (scikit-image, OpenCV) este `[linii,coloane,canale]`.

Argumentul `stride` se refera la pasul pe care il face nucleul convolutional dupa ce opereaza asupra zonei curente. Un pas de `[1,1]` inseamna ca va parcurge toti neuronii **posibili**! Argumentul `padding` se leaga de capetele imaginii(bordare). Se decide care este numarul de linii/coloane cu valori de zero care trebuie adaugate la imagine inaintea operatiei de convolutie. Aceasta operatie este folosita pentru a influenta dimensiunea matricei de iesire din stratul de convolutie.

Comparatie intre rezultatele unei operatii de convolutie, cu pas orizontal si vertical 1.
Se vede ca in varianta cu padding [1,1] (jos), iesirea are aceeasi dimensiune ca intrarea

2.2. Straturile de *pooling*

Operatia de *pooling* este cel mai bine tradusa in limba romana ca o "grupare", o subsantionare. Pe scurt, aceasta operatie *inlocuieste* valoarea unei zone din imagine/harta de trasaturi cu o statistica a acelei zone. Functia de *max-pooling* este cea mai folosita in aplicatii si *inlocuieste* valoarea dintr-o zona bine definita cu maximul acelei zone, rezultand intr-o harta de trasaturi mai mica, dar care pastreaza cea mai relevanta statistica. In acest mod, pe langa reducerea dimensionalitatii, se obtine si o invarianta la translatii mici.

Modul de creare a unui strat de *max-pooling* care sa ia vecinatati 2x2:

```
pool_x =  nn.MaxPool2d(kernel_size = (liniii_vecinatate, coloane_vecinatate), stride = (pas_orizontal, pas_vertical))
```

Rezultatul operatiei de *Max-pooling* cu argumentul
kernel_size de valoare [2,2] si *stride* de valoare [2,2]

2.3 Straturile *fully connected*

Dupa cum a fost mentionat anterior, aceste straturi sunt cele **dens conectate**, obisnuite dintr-un MLP. Inainte de a le putea folosi, **hartile de trasaturi trebuie aplatizate (vectorizate)**, folosind fie o metoda dedicata de "aplatizare":

```
flat = torch.flatten(rezultat_strat_anterior, 1,3) # Se aplatizeaza dimensiunile 1-3 (adica se obtine ceva de dimensiunea canale x linii x coloane
```

fie generand o varianta aplatizata folosind metoda implicita "view":

```
flat = rezultat_strat_anterior.view(-1,nr_canale*nr_linii*nr_coloane)
```

!Atentie: ambele variante trebuie apelate direct 'in metoda `forward`, fiindca nu sunt straturi separate.

Pentru a crea un strat dens:

```
fc = nn.Linear(in_features = nr_neuroni_intrare, out_features = nr_neuroni_iesire)
```

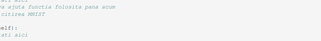
Nu uitati ca, in general, pe ultimul strat nu se aplica functie de activare (de tipul ReLU), iar `CrossEntropyLoss` aplica direct Softmax pe iesirea retelei.

3. Arhitecturi de baza

Exista o multitudine de arhitecturi *actuale*, unele care au adus imbunatatiri marginale, altele care sunt specializate pe o anumita sarcina, dar cateva arhitecturi sunt considerate a fi pietre de temelie pentru domeniu. In continuare vor fi prezentate cateva arhitecturi care au atras o atentie foarte mare la momentul aparitiei lor.

3.1. LeNet-5

Cea mai veche arhitectura convolutionala a fost prezentata in 1998 cu scopul de a recunoaste cifre scrise de mana in documente. A fost conceputa pentru imagini de rezolutie mica (32 x 32 pixeli) si, din cauza constrangerilor vremii in ceea ce priveste puterea de calcul, nu a prezentat o adancime mare (doar 2 straturi convolutionale cu nuclee de 5 x 5 pixeli). Schema arhitecturii:



Arhitectura LeNet-5. Figura preluata din [1]

3.2. AlexNet

Dupa mai bine de un deceniu (in 2012), arhitectura AlexNet a fost prima arhitectura neuronala care a castigat concursul ILSVRC cu o arhitectura avand 5 straturi convolutionale, imagini de intrare considerabil mai mari, nuclee de convolutie mai mari in straturile initiale (11 x 11) cu pasi mai mari de parcurgere a imaginii si a folosit activari de tip `ReLU`. Aceasta arhitectura mult mai puternica decat ce s-a vehiculat pana in acel moment a fost antrenata cu ajutorul a doua GPU-uri performante. Schema arhitecturii:

Arhitectura AlexNet. Figura preluata din [2]

3.3. VGG

Urmatorul pas important adus in domeniu a fost studiul *impactului* adancimii unei retele. In acest scop, retelele din familia VGG au demonstrat cresterea performantei odata cu adancimea. Reteaua VGG-19 (de la cele 19 straturi neuronale de orice tip) este printre cele mai mari retele utilizate in termeni de marimeti care trebuie invatati si cei tinuti minte in etapa de inferenta (propagare *inainte* a informatiei). In prezent, aceasta arhitectura este adesea folosita pentru trasaturile generale puternice extrase dupa ultimul strat convolutional, utile si in alte sarcini decat clasificarea. O alta observatie importanta este reprezentata de reducerea tuturor nucleelor de convolutie la dimensiunea 3 x 3 cu pas 1 la deplasare. Schema arhitecturii VGG-16:

Arhitectura VGG-16. Figura preluata din [3]

3.4. ResNet

Aparuta mai recent ca celelalte arhitecturi prezentate, importanta acestei arhitecturi a fost urmasa, rezolvand problema *vanishing gradient*. Aceasta reprezenta scaderea puternica a gradientilor odata cu avansarea in retea in etapa de propagare *inapoi*. Practic, reteleie cu un numar mare de straturi erau foarte greu de antrenat. Solutia acestui tip de arhitectura a fost introducerea blocului "rezidual", care presupunea ca la iesirea dintr-un bloc compus din mai multe convolutii se aduaga si intrarea in bloc. Aceste conexiuni de tip "scurtatura" (sau "scurtcircuit") au permis crearea unor retele mult mai adanci (inclusiv 1000 de straturi). Ca o consecinta a numarului crescut de straturi, s-a putut scadea numarul de filtre per strat, pastrand numarul de parametri care trebuiau invatati relativ redus. Arhitectura ResNet-34 este ilustrata alaturi de VGG-19 (care are un numar considerabil mai mare de parametri):

Arhitectura ResNet-34 in comparatie cu VGG-19 si o arhitectura cu 34 de straturi fara "scurtaturi". Figura preluata din [4]

Exercitiu. Incarcati baza de date MNIST si antrenati o retea convolutionala care are urmatoarea forma: convolutie + pooling + convolutie + pooling + 2 fully connected. Functiile modificate pentru citirea bazei de date se afla mai jos.

Atentie! Nu uitati de forma ceruta de straturile de convolutie si tineti cont de consumul crescut de resurse la o retea de dimensiuni mai mari.

```
In [ ]:
import numpy as np
# Nu trebuie tf pentru citirea datelor, dar trebuie pentru tot restul
import torch
def get_MNIST_train():

    mnist_train_data = np.zeros([60000,784])
    mnist_train_labels = np.zeros(60000)

    f = open('train-images.idx3-ubyte','r',encoding = 'latin-1')
    g = open('train-labels.idx1-ubyte','r',encoding = 'latin-1')

    byte = f.read(16) #f bytes magic number, 4 bytes nr imag, 4 bytes nr linii, 4 bytes nr coloane
    byte_label = g.read(8) #8bytes magic number, 4 bytes nr labels

    mnist_train_data = np.fromfile(f,dtype=np.uint8).reshape(60000,784)
    mnist_train_data = mnist_train_data.reshape(60000,1,28,28)
    mnist_train_labels = np.fromfile(g,dtype=np.uint8)

    # Conversii pentru a se potrivi cu procesul de antrenare
    mnist_train_data = mnist_train_data.astype(np.float32)
    mnist_train_labels = mnist_train_labels.astype(np.int64)

    return mnist_train_data, mnist_train_labels

def get_MNIST_test():

    mnist_test_data = np.zeros([10000,784])
    mnist_test_labels = np.zeros(10000)

    f = open('t10k-images.idx3-ubyte','r',encoding = 'latin-1')
    g = open('t10k-labels.idx1-ubyte','r',encoding = 'latin-1')

    byte = f.read(16) #f bytes magic number, 4 bytes nr imag, 4 bytes nr linii, 4 bytes nr coloane
    byte_label = g.read(8) #8bytes magic number, 4 bytes nr labels

    mnist_test_data = np.fromfile(f,dtype=np.uint8).reshape(10000,784)
    mnist_test_data = mnist_test_data.reshape(10000,1,28,28)
    mnist_test_labels = np.fromfile(g,dtype=np.uint8)

    # Conversii pentru a se potrivi cu procesul de antrenare
    mnist_test_data = mnist_test_data.astype(np.float32)
    mnist_test_labels = mnist_test_labels.astype(np.int64)

    return mnist_test_data, mnist_test_labels
```

4. Utilizarea seturilor de date complexe

Pana in acest moment, accentul a fost pus pe scrierea si rularea efectiva a retelelor. Totusi, o parte importanta a procesului de *antrenare* este reprezentat de modul in care este tratat setul de date. Nu toate sunt atat de simple si de dimensiuni asa reduse precum MNIST. In ajutorul programatorului, framework-urile precum PyTorch si Tensorflow ofera modalitati de a gestiona mai usor orice colectie de date dorita pentru antrenare. Inainte de a continua, sa urmarim cum a fost tratat setul de date pana acum:

- Exista 2 functii care citesc fisierele de date MNIST
- Acestе functii sunt apelate inainte de bucla de antrenare / testare
  `train_data, train_labels = get_MNIST_train()`
- Se calculeaza numarul de iteratii pe baza numarului de esantioane si al dimensiunii batch-ului
  `nr_iteratii = train_data.shape[0] // batch_size`
- In bucla de iteratii se aduna esantioane si etichete, pe baza valorii iteratiei
  `batch_data = train_data[it*batch_size : it*batch_size+batch_size, :]`
  `batch_labels = train_labels[it*batch_size : it*batch_size+batch_size]`
- Dupa o epoca, datele sunt amestecate (deci acest lucru se intampla doar la antrenare)
  `perm = np.random.permutation(train_data.shape[0])`
  `train_data = train_data[perm,:]`
  `train_labels = train_labels[perm]`

Pentru a simplifica lucrurile, ultimul batch este ignorat, daca acesta ar avea mai putine esantioane decat un batch obisnuit.

Urmatele pune la dispozitie de PyTorch presupun doua componente principale:
- O clasa care sa mosteneasca superclasa `Dataset`.
- Instantierea clasei `DataLoader`, care primeste ca argument un obiect de tipul clasei mentionate mai sus

!Atentie: Inainte de a continua, trebuie mentionat ca nu pot fi acoperite toate aspectele referitoare la acest capitol. Pentru mai multe detalii, este recomandat sa cautati in documentatia oficiala, precum si in tutorialele puse la dispozitie pe site-ul PyTorch.

4.1. Clase de tipul Dataset

Scopul unui obiect instantiat dintr-o clasa care mosteneste `Dataset` trebuie sa ofere urmatoarele functionalitati:

- Returnarea numarul de esantioane din setul de date, prin suprascrierea metodei `__len__`
- O modalitate de a returna un esantion, adica datele efective si etichetа/etichetele. Acest lucru se realizeaza prin suprascrierea metodei `__getitem__`. Obs: in Python, scrierea de tipul `obiect[index]` apeleaza `obiect.__getitem__(index)`. In constructorul clasei se fac operatiuni precum incarcarea intregului set de date, sau a unui fisier care contine cale catre fisiere si etichetele, sau alte operatii de procesare, inainte de citirea efectiva a datelor.

Exercitiu: Completati urmatorul cod, pentru a putea fi folosit in cazul setului MNIST folosit in laborator:

```
In [ ]:
class DatasetMNIST(Dataset):
    def __init__(self, cale_catre_date, cale_catre_etichete) :
        # Completati aici
        # Hint: va ajuta functia folosita pana acum
        # pentru citirea MNIST

    def __len__(self):
        # Completati aici

    def __getitem__(self, idx):
        # Ca sa stiti, daca aceasta clasa va fi folosita
        # de alte unelte PyTorch, Ide a-ar putea sa fie
        # un tensor, nu un intreg neo o lista. Trebuie
        # facuta o conversie in acest caz

        #if torch.is_tensor(idx):
            #idx = idx.tolist()

        # Completati aici.
        # Convertiа este sa returnati un dictionar care sa
        # contina separat datele si etichetele.
        # Ex: mnist_batch = {'date': <datele>, 'etichete': <etichetele>}
```

4.2. Clasa DataLoader

Un obiect din aceasta clasa se ocupa de amestecarea datelor, de crearea de batch-uri, si pune la dispozitie un iterator pentru a parcurge usor setul de date continut de obiectul de tip `Dataset`. Pentru exemplul curent, instantierea acestei clase se face in modul urmator:

```
mnistloader = DataLoader(obiectDatasetMNIST, batch_size=128, shuffle=True, num_workers=0)
```

O modificare care poate nu este evidenta este reprezentata de faptul ca acum a doua bucla `for` din procesul de antrenare se modifica, deoarece obiectul mnistloader este **iterabil**.

Exercitiu: Folosind clasa DatasetMNIST si o instanta de DataLoader, modificati codul original pentru a folosi aceasta abordare de folosire a setului de date.

Hints:

- Pentru a vedea unde trebuie umblat, uitati-va la inceputul sectiunii, unde exista o lista cu pasii folositi pana acum.
- Clasele Dataset si DataLoader se gasesc in pachetul `utils.data`, deci trebuie sa adaugati `from torch.utils.data import Dataset, DataLoader`
- In unele locuri, trebuia facuta conversia din matrice Numpy in tensor. DataLoader va returna tensori, insa.
- Nu uitati ca metodele din clasa voastra de tip Dataset trebuie sa dea si return la informatiile cerute de fiecare esantion in parte.

4.3. Ajutor din partea pachetul torchvision

Biblioteca `torchvision` este o alta componenta a proiectului PyTorch, care consta in pachete suplimentare care pot usura sarcina de a crea si antrenа retele neuronale. Pe langa o serie de arhitecturi consacrate deja antrenate, se pot gasi o serie de seturi de date standard pentru testarea de retele, dar si unelte pentru a incarca mai usor un set de date nou. Din ultima categorie, cea mai interesanta functionalitate este reprezentata de clasa `ImageFolder`.

Pentru a folosi clasa mentionata mai devreme, setul de date trebuie sa aiba urmatoarea structura:

```
root/clasa_1/nume.png
root/clasa_1/altnume.jpg
.....................
root/clasa_2/primul.png
root/clasa_2/aldoilea.png
.....................
```

Se poate observa ca numele fisierelor nu sunt constranse de o structura rigida (nici clasele, dar altfel), iar extensia poate fi `.jpg` sau `.png`. Folosind aceasta solutie, se rezolva etapa scrierii clase care sa mosteneasca `Dataset`. Pentru mai multe detalii, se poate consulta documentatia aferenta.

5. Bibliografie

1. LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.
2. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. Communications of the ACM, 60(6), 84-90.
3. Nash, W., Drummond, T., & Birbilis, N. (2018). A review of deep learning in the study of materials degradation. npj Materials Degradation, 2(1), 1-12.
4. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).