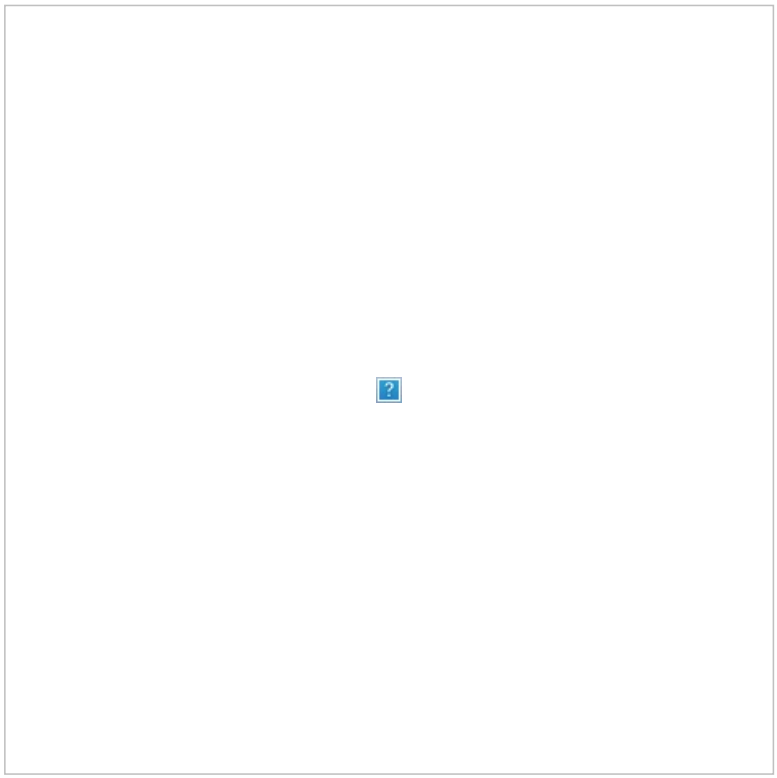
Laborator 4: Retele Convolutionale (continuare)

1. Evaluarea si imbunatatirea performantelor

Retelele convolutionale sunt sisteme puternice de clasificare cu un numar potential urias de parametri antrenabili. Una din cele mai mari probleme pentru sisteme de acest tip este ca pot ajunge sa invete foarte bine baza de date de antrenare, dar sa nu aiba capacitatea de a evalua corect datele din setul de test. Acest fenomen se numeste *overfitting* (sau memorizare), si presupune ca abilitatea de generalizare a sistemului este scazuta. Pe de alta parte, daca sistemul este prea slab, nu are capacitatea de a separa zonele de interes din spatiul trasaturilor, ducand la rezultate slabe (si la antrenare si la testare). In acest caz, se vorbeste despre *underfitting*. In figura de mai jos sunt prezentate aceste 2 probleme:



Lina galbena arata cum separa un clasificator prea slab baza de date (underfitting),
cea verde arata suprafata de separare in caz de overfitting (memorizare),
iar cea neagra prezinta o varianta mai buna de a separa cele doua clase (generalizare buna)
Imagine preluata de pe Wikipedia si modificata

Exista o serie de solutii pentru acest gen de problema:

- adaptarea numarului de parametri ai modelului astfel incat sa scada tendinta de memorizare (costisitor, necesita multiple antrenari ale sistemului cu diversi hiperparametri);
- set de antrenare cu un numar si mai mare de date, astfel incat liniile de separare in cazul de *overfiting* sa fie mai apropiate de cele ideale (in probleme reale este fie costisitor, fie aproape imposibil de alcatuit un set de date asa mare);
- imbunatatirea arhitecturii retelei.

Cea mai eficienta solutie pentru imbunatatirea performantelor unei retele neuronale este utilizarea diverselor tehnici de regularizare.

2. Tehnici de regularizare

2.1. Regularizare L2/L1

Aceste regularizari sunt frecvent intalnite si presupun adaugarea la functia loss a valorilor ponderilor invatate. In acest fel se penalizeaza aparitia unor ponderi foarte mari care sa le faca pe celelalte insignifiante, deci o uniformizare a intervalului de valori a ponderilor. Implementarea in PyTorch presupune un termen separat pentru loss. Pentru obtinerea tuturor ponderilor din retea, investigati modul de functionare al metodei `model.parameters()`.

2.2. Dropout

Aceasta tehnica s-a dovedit a fi foarte eficienta in evitarea fenomenului de overfitting la retele neuronale. Ea presupune ca in perioada antrenarii o parte din neuronii unui strat *fully connected* vor fi ignorati, simuland intr-un fel mai multe retele de dimensiuni reduse. Pentru setul de testare se vor pastra toti neuronii.

Diferenta intre o retea obisnuita si una care utilizeaza dropout
Imagine preluata din [1]

Pentru a crea un strat de dropout:

```
drop_x = nn.Dropout(p=0.5)
```

Pentru a folosi acest strat, trebuie mentionat pe care strat se aplica in metoda `forward` din clasa care descrie reteaua. Argumentul `p` indica probabilitatea ca o pondere sa fie facuta 0, deci instructiunea de mai sus introduce un strat dropout care renunta la 50% din ponderile existente in perioada antrenarii.

2.3. Batch normalization

Acest strat are rolul de a normaliza datele de la intrarea sa, accelerand invatarea si reducand tendinta de memorizare a sistemului. In perioada de antrenare, stratul invata media si varianta datelor de intrare. In etapa de test, stratul utilizeaza aceste statistici pentru a scala noile date. Pentru a introuce un astfel de strat in retea:

```
bn_x = nn.BatchNorm2d(num_features = <nr_canale_strat_intrare>)
```

Ca in toate exemplele anterioare, acest strat trebuie apelat in metoda `forward`, avand drept argument stratul anterior dorit.

2.4. Moduri ale retelei

In mod implicit, atunci cand o retea este instantiata, ea este pregatita pentru a fi antrenata. Totusi, anumite straturi au comportament diferit in antrenare fata de etapele de validare/testare. Atat straturile de Dropout cat si cele de Batch normalization se regasesc in aceasta categorie. Din acest motiv, de fiecare data cand se testeaza reteaua, trebuie apelata metoda care o trece in modul de validare:

```
model.eval()
```

Daca se doreste revenirea la modul de antrenare, se specifica:

```
model.train()
```

2.5. Aspecte suplimentare legate de Learning Rate

Setarea unei rate de invatare adecvate este un aspect important al antrenarii unei retele. Pana in acest moment, acest hiperparametru a fost setat odata cu initializarea optimizatorului si lasat constant. Totusi, in practica, cel mai des se recomanda setarea unei rate variabile de invatare. Acest lucru se realizeaza folosind un *scheduler*. O lista cu acestia se poate gasi in documentatia oficiala, dar in continuare se va discuta doar `optim.lr_scheduler.StepLR`. Aceasta functie modifica rata de invatare cu un factor constant, dupa un anumit numar de iteratii:

```
scheduler = torch.optim.lr_scheduler.StepLR(optimizer = optim, step_size = 20, gamma=0.1)
```

In linia de cod de mai sus a fost creat un *scheduler* care va modifica optimizatorul initializat anterior (`optim`). La fiecare 20 pasi, va face rata de invatare de 10 ori mai mica. Trecerea la urmatorul pas se face apeland `scheduler.step()` **la sfarsitul fiecarei epoci** (deci nu dupa fiecare iteratie din epoca).

Exercitiu: antrenati o retea formata din 3 straturi fully-connected (deci 2 ascunse). Setati primul strat sa contina 512 neuroni, iar al doilea 128. Setul de antrenare trebuie sa contina doar primele 200 de imagini din MNIST. Antrenarea trebuie sa dureze 40 de epoci. Evaluati rezultatele.

Sfat: utilizati learning rate 1e-4.

Exercitiu: introduceti learning rate variabil si adaugati un strat de Dropout (intre cele 2 FC ascunse) in reteaua anterioara. Inlocuiti stratul de dropout cu unul de Batch Normalization.

3. Utilizarea retelelor preantrenate

Mentionata si in laboratorul precedent, biblioteca `torchvision` usureaza diverse aspecte specifice rezolvarii unei probleme de Machine Learning. Pe langa `ImageFolder`, pachetul include si o serie de seturi de date cunoscute, transformari care pot fi aplicate imaginilor inainte de a fi trimise retelei, dar si o serie de arhitecturi cunoscute. Mai mult, acestea pot fi incarcate gata antrenate pe anumite seturi de date. In mod uzual, retelele folosite pentru clasificare se antreneaza pe setul `ImageNet`.

3.1. Motivatie

Termenul de *fine-tuning* (vedeti si *transfer learning*) se refera la antrena o arhitectura pe un set de date (se recomanda unul de dimensiuni mari, si cat se poate de general), iar aceasta retea sa fie apoi antrenata pe setul de date dorit. Practic, atunci cand incepe antrenarea pe al doilea set de date, reteaua nu va pleca de la ponderi aleatoare, ci de la unele invatate, relevante pentru problema dorita (clasificare,

segmentare, etc.). Aceasta tehnica isi arata cel mai clar avantajele atunci cand al doilea set (cel cerut de problema) este de dimensiuni reduse, sau este foarte complicat. In prezent, este o procedura destul de tipica, care asigura rezultate relevante destul de rapid.

3.2. Pasii necesari

In continuare, va fi explicat cum se poate folosi o retea AlexNet pentru cazul setului de date MNIST.

3.2.1. Instalarea torchvision

Biblioteca `torchvision` nu se instaleaza automat odata cu PyTorch, astfel ca trebuie instalata separat. Pentru a instala torchvision in Anaconda, se ruleaza urmatoarea comanda:

```
conda install -c pytorch torchvision
```

sau

```
conda install -c pytorch torchvision-cpu
```

daca nu se va folosi accelerarea grafica.

3.2.2. Incarcarea retelei

In modulul `models` al `torchvision` se gasesc toate arhitecturile puse la dispozitie de aceasta biblioteca. O lista cu acestea se gaseste aici, impreuna cu detalii despre retele si procesul de *fine-tuning*. Daca se doreste incarcarea unei retele AlexNet, impreuna cu ponderile invatate pe ImageNet:

```
from torchvision import models, transforms

cnn = models.alexnet(pretrained=True)
```

Ponderile vor fi descarcate intr-un director `.cache` din 'C:\Users\'.

!Atentie: Daca se incarca o retea in acest mod, clasa in care se definea reteaua anterior nu mai este necesara. Arhitectura descarcata este un obiect dintr-o clasa care mosteneste `nn.Module`, precum retelele din laboratoarele anterioare.

3.2.3. "Inghetarea" retelei

Ideea de *freezing* presupune blocarea anumitor ponderi din retea. In general, atunci cand se incarca o retea preantrenata, exista doua abordari principale:

- Se antreneaza toata reteaua, adaptand toate ponderile la noul set de date
- Se blocheaza ponderile tuturor straturilor din retea si se inlocuieste stratul de iesire. O alta varianta ar fi sa se blocheze doar partea convolutionala a retelei, lasand toate straturile *fully-connected* sa se antreneze.

Blocarea ponderilor este motivata de mai multi factori. In primul rand, daca setul de date este prea mic, se considera ca ponderile vechi sunt probabil mai bune la a descrie informatie generala. In al doilea rand, se accelereaza procesul de antrenare, ajungand la un rezultat satisfacator mult mai rapid. Pentru a bloca toate ponderile din retea, trebuie modificat atributul `requires_grad` sa fie fals, pentru toate straturile din retea:

```
for param in cnn.parameters():
    param.requires_grad = False
```

3.2.4. Modificarea retelei

In forma ei initiala, ultimul strat *fully-connected* din arhitectura descarcata va avea 1000 de neuroni (numarul de clase din ImageNet). Acest lucru se poate vedea si verificand structura retelei folosind `print(cnn)`. Rezultatul ar trebui sa fie:

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
```

```
      (0): Dropout(p=0.5)
      (1): Linear(in_features=9216, out_features=4096, bias=True)
      (2): ReLU(inplace)
      (3): Dropout(p=0.5)
      (4): Linear(in_features=4096, out_features=4096, bias=True)
      (5): ReLU(inplace)
      (6): Linear(in_features=4096, out_features=1000, bias=True)
    )
  )
```

Se observa ca reteaua este formata din 3 zone separate:

- (features), care contine partea convolutionala a retelei (`Conv2D/ReLU/MaxPool2d`).
- (avgpool), continand un singur strat de tipul `AdaptiveAvgPool2d` . Acest strat special functioneaza diferit fata de `MaxPool2d` , fiindca argumentul de intrare specifica **dimensiunea iesirii** din acest strat. Portiunea aceasta nu exista in reteaua AlexNet originala, insa este inclusa pentru a permite retelei sa functioneze cu imagini mai mari de 224 x 224.
- (classifier), zona cu straturi *fully-connected*, folosita pentru clasificare. Ultimul strat are argumentul `out_features` setat la valoarea 1000, corespunzatoare claselor din ImageNet.

Ultimul strat trebuie inlocuit, deoarece MNIST are doar 10 clase, iar ponderile invatate de acest strat sunt oricum utile doar pentru ImageNet:

```
cnn.classifier[6] = nn.Linear(in_features=4096, out_features=10)
```

Sintaxa poate fi legata rapid de rezultatul afisarii anterioare. Se specifica ca trebuie modificata reteaua `cnn` , in blocul `classifier` , pe a 6-a pozitie. Acest strat nou introdus are argumentul `requires_grad` setat ca adevarat, deci se va antrena (vezi punctul anterior).

3.2.5. Adaptarea imaginilor

Arhitectura AlexNet este vizibil diferita fata de cea definita in exercitiile anterioare. Printre altele, imaginile folosite la antrenare sunt de rezolutie diferita si normalizate. In articolul original, AlexNet foloseste imagini 227 x 227, dar acest lucru a fost modificat pentru a alinia aceasta arhitectura la altele din pachetul `models` care au rezolutia de intrare 224 x 224. Se pot folosi si rezolutii mai mari, dupa cum a fost explicat la punctul anterior (zona `avgpool`). De asemenea, imaginile din ImageNet au 3 plane de culoare, spre deosebire de cele MNIST. Pentru a trece din imagini cu un singur plan de culoare, in unele cu 3 plane, puteti folosi `np.tile` pentru a repeta de 3 ori unicul plan de culoare existent.

Din punct de vedere al normalizarii, imaginile au mediile (0.485, 0.456, 0.406) si abaterile standard (0.229, 0.224, 0.225) pentru planele R, G si B. Din fericire, modificarile se pot face usor folosind modulul `transforms` din `torchvision` , inlantuindu-le, folosind `Compose` . De asemenea, anumite transformari au cerinte specifice legate de datele de intrare. Metoda de rescalare `Resize` , necesita ca imaginea sa fie de tip `PIL` , care cere imagini de tipul `uint8` . Pastrarea imaginilor in format `uint8` se face modificand constructorul clasei de tip `Dataset` . La randul ei, metoda de normalizare `Normalize` cere ca datele sa fie tensori, nu imagini. Astfel transformarea completa va fi:

```
transf = transforms.Compose([
    transforms.ToPILImage(), transforms.Resize([224,224]),
    transforms.ToTensor(), transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])
```

Aceasta inlantuire de transformari poate fi declarata in constructorul clasei `Dataset` si apoi apelata in metoda de obtinere a esantioanelor.

Exercitiu: Folositi o retea AlexNet preantrenata pentru a antrena pe setul de date MNIST.

Bibliografie

1. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.