Machine Learning pentru Aplicatii Vizuale

Laborator 2: Feed Forward Networks (FFN) si baza de date MNIST

1. Introducere

Scopul algoritmilor de invatare automata (sau *machine learning* de acum incolo) este de a permite unor sisteme de calcul sa ia decizii sau sa faca anumite predictii, fara a fi programate in mod explicit pentru o sarcina anume. Pentru aceasta, exista o etapa de antrenare, care presupune introducerea in sistem a unui volum (preferabil mai mare) de date pentru ajustarea parametrilor algoritmului, astfel incat sarcina dorita sa fie indeplinita cat mai bine.

Pentru o intelege mai buna a tipului de probleme tratate in laborator, urmeaza o scurta clasificare a algoritmilor de machine learning. Dupa tipul de invatare, putem contura urmatoarele clase:

- Invatare **supervizata**: sistemului ii sunt prezentate o serie de date de intrare etichetate. Sarcinile de baza ale acestui tip de invatare sunt **clasificarea** si regresia (prezicerea unei valori continue pe baza unei intrari oarecare);
- Invatare nesupervizata: datele prezentate la intrare nu au etichete. Acesti algoritmi sunt folositi in probleme de clustering (gruparea datelor dupa trasaturi comune, gasirea unor tipare);
- Invatare semisupervizata: in acest caz, o parte din setul de date nu are etichete. In prezent este adesea vazuta ca o metoda de a augmenta algoritmi de invatare supervizata;
- Reinforcement learning: este radical diferit fata de celelalte tipuri de invatare, in care un agent este pus intr-un mediu dinamic care raspunde prin rasplati sau pedepse in functie de actiuni (un exemplu bun ar fi vehiculele autonome).

Sarcinile de clasificare sunt cele mai raspandite si vor fi tratate si in acest laborator. Exemplul de lucru din sedinta trecuta este un exemplu de problema simpla de clasificare. Baza de date *Moons* are doar doua trasaturi (cele doua coordonate din sistemul cartezian) si s-a observat distinctia clara intre cele doua clase, astfel ca antrenarea unui sistem cu acuratete mare nu este dificil. Problemele reale sunt considerabil mai dificile, atat pentru computer, cat si pentru oameni, in anumite cazuri.

2. Punerea problemei

Laboratorul de astazi propune clasificarea datelor din baza de date MNIST (cifre de la 0 la 9) folosind un clasificator MLP. Acesta este o retea neuronala formata dintr-un strat de intrare (care contine datele de intrare), (minim) un strat ascuns si un strat de iesire (care contine predictiile de apartenenta la clasa). Functionarea unui MLP presupune ca fiecare neuron din stratul curent sa participe la valoarea fiecarui neuron din stratul urmator. Practic, pentru a obtine valoarea unui neuron din stratul *N+1*, fiecare valoare din stratul *N* este inmultita cu o pondere (*weight*) **unica pentru fiecare pereche (neuron din stratul *N*, neuron din stratul *N+1*)** si se aduna toate aceste produse impreuna cu un deplasament suplimentar (*bias*). Rezultatul obtinut este trecut printr-o **functie de activare**, care decide daca informatia sa fie propagata, sau nu (ramane 0). Neuronul din stratul de iesire care are cea mai mare valoare este ales castigator, si determina clasa careia apartine esantionul testat.

Toate retelele neuronale se antreneaza iterativ:

- Un esantion este introdus la intrarea in retea;
- Informatia este propagata prin retea;
- Pe baza neuronului cu valoarea cea mai mare din stratul de iesire, se verifica daca predictia este buna sau nu;
- Daca predictia este gresita, atunci pe baza erorii se reajusteaza ponderile din retea (prin intermediul algoritmului de **backpropagation**). Daca predictia este corecta, nu se intampla nimic;

Acesti pasi se repeta pana la indeplinirea unui criteriu de stop (eroarea totala a esantioanelor este suficient de mica, procesul a fost repetat suficient, etc.). Formal, pentru rezolvarea unei probleme de clasificare cu retele neuronale trebuie parcursi o serie de pasi:

- Analizarea bazei de date (ce forma au datele de intrare, cat de mare este, cate clase exista, etc.);
- Alegerea arhitecturii (in cazul MLP situatia este mai simpla, fiind necesare doar alegerea numarului de straturi si numarului de neuroni din straturile ascunse, precum si o functie de activare);
- Pregatirea bazei de date, care presupune citirea integrala a datelor, daca este suficient de mic setul de date, rearanjarea datelor daca nu vin in formatul potrivit, eventuale prelucrari ale informatiei, impartire in batch-uri, impartirea in set de antrenare, set de validare si set de testare (daca nu este facuta deja impartirea);
- Alegerea unui optimizator. **Optimizatorul** este algoritmul care decide in ce directie si cat de puternic sa fie modificate ponderile din retea;
- Alegerea unei functii loss (cost). **Functia loss** este cea care ii spune sistemului "cat de gresita" a fost estimarea retelei;
- Alegerea parametrilor specifici pentru retele neuronale: **learning rate** (cat de mare poate sa fie pasul facut de o pondere intr-o directie), **dimensiunea unui batch** (esantioanele nu sunt prezentate secvential, ci in grupuri, *batch-uri*, fapt ce va fi discutat ulterior), **numarul de epoci/iteratii** (cat de multa vreme sa fie antrenata reteaua);
- Stabilirea unei metrici de performanta pentru utilizatorul uman (pentru clasificare, cea mai usoara si folosita metrica este **acuratetea clasificarii**, adica ce procent din baza de date a fost clasificat corect). **Metrica de performanta si functia loss pot fi doua functii diferite**, dupa cum se va vedea pe exemplul tratat in continuare;
- Antrenarea propriu-zisa a retelei;
- Testarea retelei si analizarea rezultatelor (de obicei, pe baza metricilor de performanta).

O parte din pasii enumerati mai sus sunt comuni indiferent de clasificatorul ales. Alta parte insa (alegerea optimizatorului, a functiei loss, learning rate-ul, numar epoci, etc.) nu se regaseste, acesti pasi sau parametri specifici fiind inlocuiti conform algoritmului de clasificare dorit.

!!! Tineti minte: desi ideea generala este aceeasi pentru toate problemele de clasificare sau regresie, fiecare pas se poate modifica intr-un fel sau altul (in functie de structura bazei de date, de cum sunt cerute datele de iesire pentru functia loss, de diverse limitari hardware, etc.)

3. Pregatirea bazei de date

Seturile de date folosite in machine learning pot contine orice fel de informatie: vizuala (imagini), audio (ex. in cazul algoritmilor care genereaza automat subtitrari, vedeti functia de auto-captioning pe YouTube), text (ex. algoritmi care traduc dintr-o limba in alta), etc.

In functie de ce tip de date vor intra in clasificator pregatirea datelor se poate schimba simtitor:

- Daca baza de date este suficient de mica (cum este cazul MNIST), toata informatia poate fi incarcata in memorie, fapt ce usureaza introducerea datelor in retea si creste viteza de rulare a algoritmului de antrenare;
- Daca baza de date contine un numar mare de imagini (eventual si ele de rezolutie mare), nu se poate incarca toata baza de date in memorie (RAM sau GPU). In acest caz, exista mai multe practici uzuale. Una este pastrarea listei de cai catre imagini impreuna cu etichetele lor de clasa intr-un fisier text. Cand trebuie creat un batch de imagini pentru a fi propagat in retea, se aleg imagini din fisierul text si **doar ele vor fi incarcate in memorie**. Similar, majoritatea bibliotecilor specializate pe lucrul cu retele neuronale ofera metode de gestiune a seturilor de date mari.

Actualizarea ponderilor se face dupa calculul functiei loss pe un batch intreg. Din acest motiv, nu este o idee buna ca imaginile sa fie ordonate in functie de clasa (mai intai doar imagini de clasa 0, apoi doar de clasa 1, etc.). **De asemenea, ajuta ca dupa fiecare epoca sa amestecam imaginile, astfel incat un batch sa nu contina mereu aceleasi imagini de la epoca anterioara**.

3.1. Baza de date MNIST

Probabil cea mai cunoscuta baza de date, MNIST este format din imagini cu cifre scrise de mana. Desi nu mai este de multa vreme dificil atingerea unei performante ridicate (eroare pe setul de test de <1%), ea ramane printre favorite pentru testarea diversilor algoritmi de machine learning.

Structura:

- 70.000 de poze (60.000 pentru antrenare si 10.000 pentru testare)
- Fiecare imagine este de dimensiunea 28 x 28 pixeli (grayscale, stocate ca un singur plan de culoare)

Tinand cont ca un MLP nu tine cont de forma trasaturilor, ci doar de numarul lor, imaginea introdusa in stratul de intrare este vectorizata.

Exemple de imagini din MNIST. Imagine preluata de pe www.wikipedia.org

Functiile cu care vom incarca baza de date MNIST in memorie sunt urmatoarele:

In [ ]:

```python
import struct
import numpy as np
# Nu trebuie torch 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) #4 bytes magic number, 4 bytes nr imag, 4 bytes nr linii, 4 bytes nr coloane
    byte_label = g.read(8) #4 bytes magic number, 4 bytes nr labels

    mnist_train_data = np.fromfile(f,dtype=np.uint8).reshape(60000,784)
    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) #4 bytes magic number, 4 bytes nr imag, 4 bytes nr linii, 4 bytes nr coloane
    byte_label = g.read(8) #4 bytes magic number, 4 bytes nr labels

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

    # Conversii pentru a se potrivi cu procesul de testare
    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. Arhitectura retelei

In acest moment exista o multitudine de arhitecturi neurale folosite in diverse sarcini de machine learning. Feed Forward Networks (FFNs) sunt o clasa de retele neuronale in care pentru obtinerea etichetei unui esantion, informatia trebuie propagata mereu inainte, catre straturile superioare din structura ierarhica a retelei. Dupa cum a fost mentionat anterior, pentru acest laborator introductiv, se va folosi o arhitectura de tip MLP. Tinand cont de modul in care se definesc retelele in Pytorch, se vor aplica urmatorii pasi:

- Definim variabilele care descriu ponderile si deplasamentele aferente din retea
- Definim operatiile care transmit datele de la stratul de intrare catre stratul ascuns, si apoi catre stratul de iesire

Dupa acest exemplu initial, o sa fie prezentata si modalitatea uzuala de descriere a retelei. Aceasta presupune sa nu se defineasca manual ponderile, ci sa se apeleaze functii care sa genereze direct straturile.

In [ ]:

```python
# Modulul nn contine o multitudine de elemente
# esentiale construirii unei retele neuronale
import torch.nn as nn

class Retea_MLP(nn.Module):

    def __init__(self, nr_neuroni_input, nr_neuroni_hidden, nr_clase):

        # Pentru a putea folosi mai departe reteaua, este recomandata mostenirea
        # clasei de baza nn.Module
        super(Retea_MLP,self).__init__()

        # Definirea ponderilor si a deplasamentelor din stratul ascuns
        self.w_h = torch.randn(nr_neuroni_input, nr_neuroni_hidden, dtype = torch.float, requires_grad=True)
        self.b_h = torch.randn(nr_neuroni_hidden, dtype = torch.float, requires_grad=True)

        # Definirea ponderilor si a deplasamentelor din stratul de iesire
        self.w_o = torch.randn(nr_neuroni_hidden, nr_clase, dtype = torch.float, requires_grad=True)
        self.b_o = torch.randn(nr_clase, dtype = torch.float, requires_grad=True)


    # Se aduna toate variabilele antrenabile intr-o lista, pentru a putea face referire rapida la ele
    def parameters(self):
        return [self.w_h, self.b_h, self.w_o, self.b_o]

    def forward(self,input_batch):
        # Intr-un MLP, intrarea este sub forma unui vector, deci un batch
        # este o matrice de dimensiunea nr_esantioane_batch x dimensiune esantion
        input_batch = torch.from_numpy(input_batch)
        self.hidden = torch.mm(input_batch, self.w_h) + self.b_h

        out = torch.mm(self.hidden, self.w_o) + self.b_o

        return out

# Instantiem reteaua
mlp = Retea_MLP(28*28,1000,10)
```

Desi arhitectura prezentata in codul de mai sus este valida, nu a fost luat in seama faptul ca anumiti neuroni pot sa nu fie "aprinsi". In acest scop se foloseste functia de activare care modifica valoarea unui neuron, in functie de un prag. Pentru a folosi functia de activare "Rectified Linear Unit" (ReLU) introduceti urmatorul cod in definirea retelei:

```python
self.relu = nn.ReLU()
```

, apoi apelati `self.relu` cu argumentul dorit in metoda `forward`.

5. Functii loss

Pentru a putea imbunatati arhitectura noastra trebuie sa stim cat de bine, sau mai bine zis, cat de rau, clasifica datele noastre. Pentru a exprima cantitativ conceptul de "cat de rau se descurca" se foloseste o functie de cost (denumita mai frecvent loss, in literatura de specialitate). O functie loss pentru o problema de clasificare primeste la intrare etichetele corecte ale esantioanelor si neuronii din stratul de iesire (toti neuronii, desi stim ca neuronul castigator este cel cu valoarea cea mai mare).

Cea mai folosita functie loss pentru clasificare este **Cross Entropy** care are valori cu atat mai mari atunci cu cat diferenta intre doua distributii de probabilitati este mai mare. Pentru aceasta, trebuie sa transformam atat etichetele, cat si neuronii de iesire in distributii de probabilitati. In cazul etichetelor corecte, acest procedeu este simplu: distributia care corespunda etichetei i are valoarea 1 pe pozitia i si 0 in rest. Valorile neuronilor de iesire sunt transformate folosind functia Softmax:

$$Y_i = \frac{e^{X_i}}{\sum_{i=0}^{nr\_etichete-1} e^{X_i}}$$

Astfel, suma neuronilor de iesire devine 1, iar toate valorile sunt >0, obtinand o distributie de probabilitati valida.

Majoritatea functiilor de cost sau eroare pot fi folosite, atat timp cat etichetele si neuronii sunt prelucrati adecvat: eroarea medie, eroarea patratica medie (des folosite in probleme de regresie), etc.

Pentru a specifica functia loss in cazul nostru:

In [ ]:

```python
# Specificarea functiei loss
loss_function = nn.CrossEntropyLoss(reduction='sum')
```

6. Optimizatori

Pentru a ajusta parametrii unei retele (ponderile si deplasamentele) trebuie specificat un **optimizator** care decide strategia de modificare in functie de gradientii obtinuti cu ajutorul functiei loss. Algoritmul de baza, in ceea ce priveste optimizatorii, este denumit **"Stochastic Gradient Descent" (SGD)** si a fost folosit multa vreme de majoritatea persoanelor care lucreaza in domeniu. Plecand de la acesta, au aparut mai multi algoritmi imbunatatiti care accelereaza procesul de minimizare a erorii. Dintre acesti algoritmi mai noi amintim:

- **Adam (cel mai folosit in acest moment)**
- Adadelta
- Adagard
- SGD with momentum
- RMSprop

Comparatie intre mai multi optimzatori. Animatii preluate din *CS231n Convolutional Neural Networks for Visual Recognition*

Optimizatorul necesita specificarea **dimensiunii pasului de optimizare, denumit si rata de invatare (*learning rate*)**, care in exemplul nostru va fi tinut constant 10^{-5}. De asemenea, trebuie mentionate si **variabilele care trebuie actualizate in timpul procesului de antrenare**:

In [ ]:

```python
# Specificarea optimizatorului
optim = torch.optim.SGD(mlp.parameters(), lr=1e-5)
```

7. Bucla de invatare

Avand definite diversele componente ale retelei, trebuie incarcate datele, scrisa metrica de acuratete, definiti hiperparametrii (precum dimensiunea unui batch, numarul de epoci, numarul de iteratii per epoca, si altii in alte cazuri). Dupa aceea, putem sa ne ocupam de antrenarea propriu-zisa a retelei.

In [ ]:

```python
train_data, train_labels = get_MNIST_train()
batch_size = 128 # Se poate si mai mult in cazul curent, dar este o valoare frecventa
nr_epoci = 15
nr_iteratii = train_data.shape[0] // batch_size # Din simplitate, vom ignora ultimul batch, daca este incomplet


for ep in range(nr_epoci):
    predictii = []
    etichete = []

    for it in range(nr_iteratii):
        # Luam urmatoarele <batch_size> esantioane si etichete
        batch_data = train_data[it*batch_size : it*batch_size+batch_size, :]
        batch_labels = train_labels[it*batch_size : it*batch_size+batch_size]
        # Se calculeaza predictia retelei pentru datele curente (forward pass/ propagare inainte)
        current_predict = mlp.forward(batch_data)

        # Se calculeaza valoarea momentana a functiei loss
        loss = loss_function(current_predict, torch.from_numpy(batch_labels))

        # Se memoreaza predictiile si etichetele aferente batch-ului actual (pentru calculul acuratetii)
        current_predict = np.argmax(current_predict.detach().numpy(), axis=1)
        predictii = np.concatenate((predictii,current_predict))
        etichete = np.concatenate((etichete,batch_labels))

        # Antrenarea propriu-zisa

            # 1. Se sterg toti gradientii calculati anteriori, pentru toate variabilele antrenabile
            # deoarece, metoda <backward> acumuleaza noile valori, in loc sa le inlocuiasca.
        optim.zero_grad()
            # 2. Calculul tuturor gradientilor. Backpropagation
        loss.backward()
            # 3. Actualizarea tuturor ponderilor, pe baza gradientilor.
        optim.step()



    # Calculam acuratetea
    acc = np.sum(predictii==etichete)/len(predictii)
    print( 'Acuratetea la epoca {} este {}%'.format(ep+1,acc*100) )

    # Se genereaza o permutare noua a tuturor esantioanelor si etichetelor corespunzatoare
    perm = np.random.permutation(train_data.shape[0])
    train_data = train_data[perm,:]
    train_labels = train_labels[perm]
```

8. Testarea retelei

Odata terminata antrenarea retelei, putem testa pe un set de date noi. Veti observa ca structura de la bucla de antrenare ne va ajuta in continuare:

In [ ]:

```python
test_data, test_labels = get_MNIST_test()
batch_size_test = 100 # pentru usurinta, ca sa testam toate etichetele alegem un divizor al numarului de date de test
nr_iteratii_test = test_data.shape[0] // batch_size_test

predictii = []
for it in range(nr_iteratii_test):
    batch_data = test_data[it*batch_size_test : it*batch_size_test+batch_size_test, :]
    batch_labels = test_labels[it*batch_size_test : it*batch_size_test+batch_size_test]

    current_predict = mlp.forward(batch_data)
    current_predict = np.argmax(current_predict.detach().numpy(),axis=1)
    predictii = np.concatenate((predictii,current_predict))

acc = np.sum(predictii==test_labels)/len(predictii)
print( 'Acuratetea la test este {}%'.format(acc*100) )
```

9. Simplificari

Exemplul de mai sus o a propus o abordare complicata pentru o problema simpla. Atunci cand a fost definita reteaua, s-au generat ponderi aleatoare si s-a descris interactiunea intre ele si datele de intrare. Setul de date a fost preprocesat manual si gestionat manual in bucla de antrenare. Specificarea ponderilor antrenabile optimizatorului a fost facuta folosind o lista construinta de programator. In practica, orice biblioteca de Machine Learning ofera mult mai multe unelte pentru construirea, antrenarea si utilizarea unei retele neuronale, permitand celui responsabil sa se concentreze pe probleme de nivel mult mai inalt.

Mai jos, se poate urmari construirea aceleiasi retele de tip MLP, dar folosind straturile definite in `torch.nn`:

In [ ]:

```python
# Modulul nn contine o multitudine de elemente
# esentiale construirii unei retele neuronale
import torch.nn as nn

class Retea_MLP(nn.Module):

    def __init__(self, nr_neuroni_input, nr_neuroni_hidden, nr_clase):

        # Pentru a putea folosi mai departe reteaua, este recomandata mostenirea
        # clasei de baza nn.Module
        super(Retea_MLP,self).__init__()

        self.hidden_layer = nn.Linear(nr_neuroni_input, nr_neuroni_hidden)
        self.out_layer = nn.Linear(nr_neuroni_hidden, nr_clase)

    def forward(self,input_batch):
        # Intr-un MLP, intrarea este sub forma unui vector, deci un batch
        # este o matrice de dimensiunea nr_esantioane_batch x dimensiune esantion
        input_batch = torch.from_numpy(input_batch)
        hidden = self.hidden_layer(input_batch)
        out = self.out_layer(hidden)

        return out

# Instantiem reteaua
mlp = Retea_MLP(28*28,1000,10)
```

!Atentie: Se poate observa ca a disparut, printre altele, metoda clasei de a aduna toate variabilele antrenabile. Aceasta este deja implementata in superclasa `nn.Module`. Astfel, instructiunea prin care este mentionat optimizatorul ramane neschimbata.

Anterior nu a fost folosita metoda mostenita, deoarece aceasta ia in considerare doar variabilele create de straturile din modulul `nn`, precum `nn.Linear`.

Daca folositi straturile definite in *nn* nu mai trebuie definita metoda *parameters*

10. Salvarea si incarcarea retelei

Pentru a putea fi folosita ulterior (sau reluata antrenarea, in cazul opririi acesteia), reteaua se poate salva. Toate variabilele corespunzatoare unei retele sunt stocate sub forma unui *dictionar de stare (state_dict)*, care se poate salva astfel:

In [ ]:

```python
torch.save(mlp.state_dict(), './mlp_mnist.pt')
```

Pentru a incarca reteaua antrenata anterior in alta sesiune de Python, trebuie instantiata clasa in care a fost descrisa reteaua, iar apoi incarcat dictionarul de stari:

In [ ]:

```python
# Descrierea arhitecturii
mlp_antrenat = Retea_MLP(28*28,1000,10)

# Se vor incarca ponderile calculate anterior.
mlp_antrenat.load_state_dict(torch.load('./mlp_mnist.pt'))

# In majoritatea cazurilor, trebuie mentionat faptul ca nu se mai antreneaza reteaua.
# Anumite straturi au comporamente diferite la antrenare, fata de testare. In cazul
# de fata, nu ar trebui sa fie necesara aceasta trecere in modul de inferenta.
mlp_antrenat.eval()
```

!Atentie: Dictionarul de stare este populat de variabilele generate de straturile precum `nn.Linear`. In varianta originala a retelei (in care s-au scris in mod explicit ponderile si operatiile), variabilele nu sunt trecute automat in dictionarul de stare. Din acest motiv, salvarea nu ar fi functionat.

In [ ]:

```python
import struct
import numpy as np
# Nu trebuie torch 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) #4 bytes magic number, 4 bytes nr imag, 4 bytes nr linii, 4 bytes nr coloane
    byte_label = g.read(8) #4 bytes magic number, 4 bytes nr labels

    mnist_train_data = np.fromfile(f,dtype=np.uint8).reshape(60000,784)
    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) #4 bytes magic number, 4 bytes nr imag, 4 bytes nr linii, 4 bytes nr coloane
    byte_label = g.read(8) #4 bytes magic number, 4 bytes nr labels

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

    # Conversii pentru a se potrivi cu procesul de testare
    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



# Modulul nn contine o multitudine de elemente
# esentiale construirii unei retele neuronale
import torch.nn as nn

class Retea_MLP(nn.Module):

    def __init__(self, nr_neuroni_input, nr_neuroni_hidden, nr_clase):

        # Pentru a putea folosi mai departe reteaua, este recomandata mostenirea
        # clasei de baza nn.Module
        super(Retea_MLP,self).__init__()

        self.hidden_layer = nn.Linear(nr_neuroni_input, nr_neuroni_hidden)
        self.relu = nn.ReLU()
        self.out_layer = nn.Linear(nr_neuroni_hidden, nr_clase)

    def forward(self,input_batch):
        # Intr-un MLP, intrarea este sub forma unui vector, deci un batch
        # este o matrice de dimensiunea nr_esantioane_batch x dimensiune esantion
        input_batch = torch.from_numpy(input_batch)
        hidden = self.relu(self.hidden_layer(input_batch))
        out = self.out_layer(hidden)

        return out

# Instantiem reteaua
mlp = Retea_MLP(28*28,1000,10)

# Specificarea functiei loss
loss_function = nn.CrossEntropyLoss(reduction='sum')

# Specificarea optimizatorului
optim = torch.optim.Adam(mlp.parameters(), lr=1e-3)

train_data, train_labels = get_MNIST_train()
batch_size = 128 # Se poate si mai mult in cazul curent, dar este o valoare frecventa
nr_epoci = 15
nr_iteratii = train_data.shape[0] // batch_size # Din simplitate, vom ignora ultimul batch, daca este incomplet


for ep in range(nr_epoci):
    predictii = []
    etichete = []

    for it in range(nr_iteratii):
        # Luam urmatoarele <batch_size> esantioane si etichete
        batch_data = train_data[it*batch_size : it*batch_size+batch_size, :]
        batch_labels = train_labels[it*batch_size : it*batch_size+batch_size]
        # Se calculeaza predictia retelei pentru datele curente (forward pass/ propagare inainte)
        current_predict = mlp.forward(batch_data)

        # Se calculeaza valoarea momentana a functiei loss
        loss = loss_function(current_predict, torch.from_numpy(batch_labels))

        # Se memoreaza predictiile si etichetele aferente batch-ului actual (pentru calculul acuratetii)
        current_predict = np.argmax(current_predict.detach().numpy(), axis=1)
        predictii = np.concatenate((predictii,current_predict))
        etichete = np.concatenate((etichete,batch_labels))

        # Antrenarea propriu-zisa

            # 1. Se sterg toti gradientii calculati anteriori, pentru toate variabilele antrenabile
            # deoarece, metoda <backward> acumuleaza noile valori, in loc sa le inlocuiasca.
        optim.zero_grad()
            # 2. Calculul tuturor gradientilor. Backpropagation
        loss.backward()
            # 3. Actualizarea tuturor ponderilor, pe baza gradientilor.
        optim.step()



    # Calculam acuratetea
    acc = np.sum(predictii==etichete)/len(predictii)
    print( 'Acuratetea la epoca {} este {}%'.format(ep+1,acc*100) )

    # Se genereaza o permutare noua a tuturor esantioanelor si etichetelor corespunzatoare
    perm = np.random.permutation(train_data.shape[0])
    train_data = train_data[perm,:]
    train_labels = train_labels[perm]



test_data, test_labels = get_MNIST_test()
batch_size_test = 100 # pentru usurinta, ca sa testam toate etichetele alegem un divizor al numarului de date de test
nr_iteratii_test = test_data.shape[0] // batch_size_test

predictii = []
for it in range(nr_iteratii_test):
    batch_data = test_data[it*batch_size_test : it*batch_size_test+batch_size_test, :]
    batch_labels = test_labels[it*batch_size_test : it*batch_size_test+batch_size_test]

    current_predict = mlp.forward(batch_data)
    current_predict = np.argmax(current_predict.detach().numpy(),axis=1)
    predictii = np.concatenate((predictii,current_predict))

acc = np.sum(predictii==test_labels)/len(predictii)
print( 'Acuratetea la test este {}%'.format(acc*100) )
```