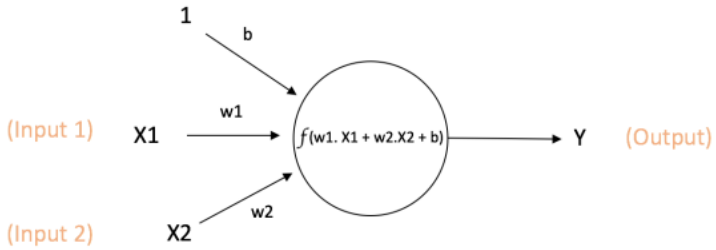
Clasificare folosind rețele neuronale simple

În cadrul unei metode de clasificare se urmărește încardrarea unor date cu anumiți parametri într-o categorie, numită și clasă, dintr-o mulțime finită și cunoscută de clase. Spre exemplu, pentru rezolvarea unei probleme de recunoaștere facială se dorește stabilirea corespondenței dintre mulțimea trăsăturilor faciale care pot fi extrase dintr-o imagine și o mulțime finită de persoane cunoscute. Algoritmii de clasificare realizează asocieri între spațiul parametrilor datelor ce se doresc clasificate și mulțimea claselor vizate. Cei mai eficienți algoritmi de clasificare sunt supervizați, ceea ce presupune o etapă de antrenare folosind date deja disponibile, algoritmii realizând asocieri între parametrii datelor de antrenare și clasele în care au fost încadrate acestea.

O rețea neuronală artificială (ANN) este un model computațional inspirat din modul în care rețelele neuronale biologice procesează informațiile în creierului uman. Rețelele neuronale artificiale se utilizează la scară largă pentru o multitudine de aplicații de cercetare și dezvoltare din domeniul învățării automate, datorită eficacității dovedite de acestea pentru rezolvarea unor probleme de recunoaștere a vorbirii, vedere artificială și procesarea textului.

Unitatea de bază a unei rețele neuronală este neuronul, care poate primi date de intrare de alți neuroni sau de la o sursă externă, pe baza cărora calculează o valoare la ieșire. Fiecare intrare are o pondere asociată (w), care este atribuită pe baza importanței sale relativ la alte intrări. Neuronul aplică o funcție de activare f pe suma ponderată a intrărilor sale:



Output of neuron = Y= $f(w1.\ X1 + w2.X2 + b)$

Rețeaua de mai sus are intrările X1 și X2 și ponderile w1 și w2 asociate acestor intrări. În plus, există o altă intrare 1 cu ponderea b (numită *bias*). Ieșirea Y a neuronului se calculează ca fiind suma intrărilor xi, ponderare de ponderile wi. Funcția f se numește funcție de activare și trebuie să fie neliniară. Așadar unul dintre scopurile funcției de activare este introducerea neliniarității în calculul valorii de ieșire a neuronului. Acest lucru este important deoarece majoritatea datelor din lumea reală sunt neliniare, prin urmare dorim ca neuronii să învețe

reprezentări neliniare a acestora. Fiecare funcție de activare (sau non-liniaritate) primește o singură valoare și execută o anumită operație matematică. În majoritatea cazurilor se folosesc următoarele funcții de activare:

Sigmoid: preia o valoare de intrare reală și o aduce în intervalul (0,1)

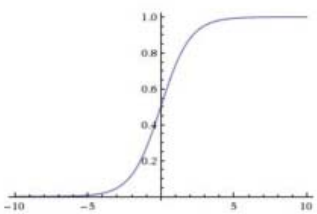
$$\sigma(x) = 1 / (1 + \exp(-x))$$

tanh: preia o valoare de intrare reală și o aduce în intervalul [-1, 1]

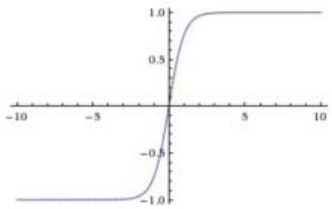
$$\tanh(x) = 2\sigma(2x) - 1$$

ReLU: Rectifier Linear Unit. Aplică o valoare prag 0 pe valoarea reală de intrare (înlocuiește valorile negative cu zero).

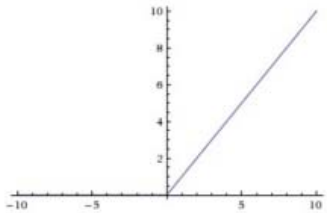
$$f(x) = \max(0, x)$$



Sigmoid

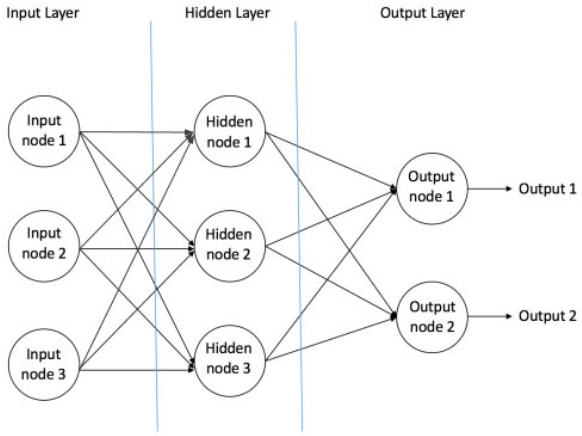


tanh



ReLU

Rețelele neuronale clasice sunt constituite din două sau mai multe straturi, fiecare cu cel puțin un neuron. Aceste rețele neuronale se mai numesc și complet conectate (fully-connected), deoarece există câte o conexiune de la fiecare neuron dintr-un strat către fiecare neuron din următorul strat. La nivelul fiecărui neuron au loc operațiile descrise anterior. Având în vedere direcția de efectuare a calculelor în cadrul unei astfel de rețele, ea mai poartă denumirea de rețea cu propagare înainte (feed forward). Într-o astfel de rețea datele de la intrare sunt supuse calculelor aferente începând cu primul strat, continuând cu straturile următoare în ordine și încheind cu determinarea valorilor stratului de ieșire.



Pentru rezolvarea problemelor de clasificare, rețelele neuronale sunt supuse unui proces de antrenare, în cadrul căruia ele învață corelații între datele de antrenare și clasele în care sunt încadrate acestea. Propagarea înapoi a erorilor (backpropagation, retropropagare) este una dintre metodele cel mai frecvent utilizate pentru antrenarea unei rețele neuronale. Este o metodă de învățare supervizată, ceea ce înseamnă învățarea se realizează pe baza unor date de antrenare deja clasificate (fiecare instanță a dateor de antrenare este etichetată ca aparținînd unei anumite clase). Algoritmul backpropagation se bazează pe învățarea din greșeli, el efectuând corecții asupra rețelei prin ajustarea ponderilor acesteia. Așadar scopul învățării este determinarea acelor valori ale ponderilor rețelei care să asigure realizarea unei corelații cât mai bune între datele de antrenare și etichetele acestora.

Inițial, toate ponderile iau valori aleatoare. Fiecare instanță din setul de date de antrenare se propagă înainte prin rețea, rezultînd un set de valori la ieșirea rețelei. Această ieșire este comparată cu ieșirea dorită, deja cunoscută (din eticheta datelor de antrenare corespunzătoare), iar eroarea este „propagată” înapoi la nivelul anterior. Ponderile rețelei se ajustează în sensul minimizării acestei erori. Procesul se repetă până când eroarea de ieșire se situează sub un prag prestabilit. Odată ce algoritmul se încheie, rețeaua neuronală este capabilă să proceseze date de intrare noi, pe care nu le-a mai întâlnit. Corectitudinea încadrării acestor date în clasa corespunzătoare depinde de numeroși factori, cum ar fi relevanța datelor folosite la antrenare, algoritmul de optimizare folosit, rata de învățare etc.

Aplicație:

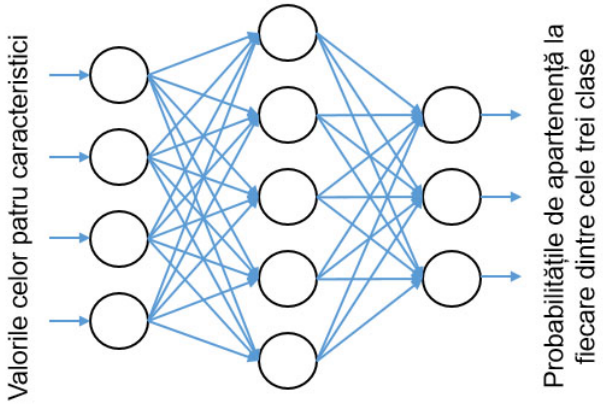
Dorim să proiectăm un algoritm de clasificare care să permită prelucrarea datelor din domeniul botanicii. Astfel, vom implementa o rețea neuronală cu ajutorul căreia vom determina speciile unor plante pe baza unor caracteristici biologice. Vom împărți plantele pe trei specii cunoscute, așadar urmărim asignarea plantelor în trei clase. Avem nevoie de o serie de date de antrenare (caracterictici ale plantelor și specia corespunzătoare fiecărui set de caracteristici), pe care le regăsim în fișierul **iris.csv:**

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
…				
7	3.2	4.7	1.4	versicolor
6.4	3.2	4.5	1.5	versicolor
6.9	3.1	4.9	1.5	versicolor
5.5	2.3	4	1.3	versicolor
…				
6.3	3.3	6	2.5	virginica
5.8	2.7	5.1	1.9	virginica
7.1	3	5.9	2.1	virginica
…				
6.3	2.9	5.6	1.8	virginica

Plantele vizate au 4 caracteristici specificate prin valori reale, și ele se pot încadra în trei specii (trei clase). Datele de antrenare sunt în număr de 150, pentru fiecare cunoscându-se valorile caracteristicilor, precum și clasa (specia) în care se încadrează. Scopul este de a construi o rețea neuronală și de a o antrena folosind datele cunoscute, în ideea de a putea identifica specia oricărei plante cu alte valori ale acelorași caracteristici.

Fiecărei clase i se asociază câte un label (valoare numerică = [0…nrClase-1]). Așadar, cele trei clase vor avea label-urile *setosa* = 0, *versicolor* = 1, *virginica* = 2 . Rețeaua neuronală va opera cu aceste label-uri în locul string-urilor aferente fiecărei clase.

Rețeaua pe care o vom implementa are următoarea structură:



Codul care implementează, antrenează și evaluează o astfel de rețea este ilustrat mai jos. Codul sursa este redactat în limbajul Python și se folosește biblioteca PyTorch. Prima dată listăm codul care implementează rețeaua neuronală, apoi îl vom parcurge pas cu pas:

```
class SimpleNN(nn.Module):
    def __init__(self, inputSize, hiddenSize, outputSize):
        super(SimpleNN, self).__init__()

        self.fc1 = torch.nn.Linear(inputSize, hiddenSize)
        self.fc2 = torch.nn.Linear(hiddenSize, outputSize)

    def forward(self, input):
        output = self.fc1(input)
        output = F.relu(output)
        output = self.fc2(output)

        return output

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

        for epoch in range(nrEpochs):
            accuracy = 0

            for input, target in zip(inputs, targets):
                optimizer.zero_grad()
                predicted = self.forward(input.unsqueeze(0))
                loss = lossFunc(predicted, target.unsqueeze(0))
                loss.backward()
                optimizer.step()

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

Semnificația liniilor de cod este următoarea:

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

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

```
def __init__(self, inputSize, hiddenSize, outputSize):
```

Constructorul primește ca parametri dimensiunile straturilor rețelei.

```
self.fc1 = torch.nn.Linear(inputSize, hiddenSize)
self.fc2 = torch.nn.Linear(hiddenSize, outputSize)
```

Clasa definită anterior constituie un modul care include două submodule, fc1 și fc2 (fc = "fully connected"). Acestea implementează operațiile liniare care au loc între două straturi succesive ale rețelei neuronale (suma ponderată a intrărilor). Ponderile rețelei sunt parametri predefiniți ai acestor două module.

În cazul modului fc1, se pornește cu datele din stratul de intrare (inputSize neuroni) și se realizează suma ponderată a acestora, rezultând hiddenSize valori în stratul ascuns al rețelei. Modulul fc2 preia valorile din stratul ascuns și realizează operațiile liniare aferente stratului de ieșire, rezultând outputSize valori.

```
    def forward(self, input):
        output = self.fc1(input)
        output = F.relu(output)
        output = self.fc2(output)
        return output
```

Metoda implementează propagarea înainte prin rețeaua neuronală. Pytorch prelucrează datele numerice folosind tensori. Acești tensori pot fi valori scalare, vectori sau matrici bi- și multi-dimensionale. Putem considera că tensorii sunt modalitatea Pytorch de gestiune a array-urilor de valori numerice.

Parametrul **input** este un tensor ce conține valorile de la intrarea rețelei neuronale (de exemplu, un set de patru caractersistici ale unei specii de plante). **input** este supus prelucrărilor care au loc la nivelul rețelei. Prima dată se realizează operațiile din primul strat al rețelei. Pentru neuronii din stratul ascuns, se determină sumele ponderate ale valorilor din stratul de intrare (fc1(input)), cărora li se aplică funcția de activare *relu*, descrisă anterior. Apoi, valorile ce rezultă în stratul ascuns sunt supuse procesărilor liniare de unde rezultă valorile de ieșire. Astfel, la ieșirea rețelei rezultă un tensor ce conține trei valori ce reprezintă scoruri de apartenență a datelor de intrare (cele 4 caracteristici ale plantei) la cele trei clase vizate (cele

trei specii de plante). Prin normalizarea acestor scoruri se pot obține probabilitățile de apartenență la cele trei clase, dar în exemplul furnizat nu este nevoie de această normalizare.

```
def train(self, inputs, targets):
```

Aceasta metodă realizează antrenarea rețelei. Algoritmul de antrenare folosește date de antrenare (inputs) pentru care se cunosc clasele în care se încadrează (targets).

inputs conține valorile caracteristicilor plantelor (așadar în exemplul furnizat este un tensor de dimensiune [150, 4] cu valori reale, 150 instanțe x 4 caracteristici)
targets conține câte un label pentru fiecare instanță – pentru fiecare set de 4 caracteristici din **inputs, targets** conține câte o valoare întreagă corespunzătoare uneia din cele trei clase disponibile. În exemplul furnizat, ***targets*** este un tensor de dimensiune [150, 1] ce conține valori întregi din mulțimea {0, 1, 2}, aceste valori corespunzând celor trei clase disponibile.

```
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.

```
for input, target in zip(inputs, targets):
```

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(input.unsqueeze(0))
```

predicted este rezultatul generat de rețeaua neuronală pentru valoarea dată ca parametru de intrare. În cazul de față **predicted** este un tensor ce conține trei 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. **Input** este un tensor de dimensiune [4], dar, întrucât funcția de calcul a erorii utilizează tensori bidimensionali, se folosește metoda **unsqueeze** pentru a adăuga o dimensiune suplimentară tensorului (dintr-un tensor 1D de dimensiune [4] devine un tensor 2D de dimensiune [1, 4] iar valorile nu se modifică).

```
loss = lossFunc(predicted, target.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 rețeua neuronală și se ajustează ponderile rețelei în sensul minimizării erorii, cu metoda implementată în optimizer.

Cerințe:

1. Determinați și afișați acuratețea rețelei la sfârșitul fiecărei epoci de antrenare

Acuratețea rețelei pentru epoca curentă se determină ca fiind probabilitatea ca datele de antrenare să fi fost clasificate corect în cadrul acelei epoci (numărul de instanțe din setul de date clasificate corect / numărul total al instanțelor).

Explicații:

Pentru un input de dimensiune 4, reprezentând valorile celor 4 caracteristici ale plantelor, rețeaua generează un output de dimensiune 3, plantele încadrându-se într-una cele trei clase disponibile.
De exemplu, presupunem că pentru setul de caracteristici [7, 3.2, 4.7, 1.4] rețeaua generează valorile [1.5, 7.7, 2.3]. Aceasta înseamnă că rețeaua consideră că planta cu cele patru caracteristici menționate aparține celei de-a doua clase, cea cu label-ul 1 (clasele au label-urile 0, 1, 2). Dacă acest lucru corespunde cu valoarea label-ului din setul de date de antrenare, înseamnă cu rețeaua a clasificat corect datele de intrare, prin urmare acest rezultat contribuie la acuratețea rețelei. După ce se parcurge setul de date de antrenare, se determină acuratețea procentuală a rețelei (nr datelor clasificare corect / numărul tututor datelor).

2. Realizați o evaluare a rețelei neuronale
Împărțiți setul de date în două părți: 66% din date vor constitui setul de antrenare, restul vor constitui setul de test. Antrenați rețeaua folosind setul de antrenare și determinați acuratețea acesteia folosind setul de test. Afișați acuratețea la sfârșitul fiecărei epoci (Similar cu cerința 1.)

3. Citiți de la tastatură un set de valori ale caracteristicilor plantelor și, folosind rețeaua antrenată, afișați probabilitățile ca planta cu setul de valori introdus de utilizator să aparțină celor trei specii. Probabilitățile de apartenență la clasele vizate se pot obține aplicând funcția *F.softmax()* pe valorile de ieșire ale rețelei (valorile de ieșire = cele care se obțin cu metoda *forward()*)

4. Determinați numărul optim de epoci de antrenare. Evaluați rețeaua pentru un număr de epoci din mulțimea {2, … , nr_max_epoci} și determinați numărul de epoci pentru care acuratețea este suficient de mare (de exemplu, câte epoci sunt necesare pentru ca acuratețea > 0.95)

5. Realizați o validare încrusișată (cross-validation) a rețelei. Folosiți 90% din datele disponibile pentru antrenarea rețelei, și evaluați rețeaua folosind restul de 10%. Apoi alegeți alte 90% instanțe ale datelor, antrenați din nou rețeaua și evaluați folosind restul datelor. Repetați până când întregul set de date s-a utilizat pentru antrenare și apoi afișați acuratețea medie ce rezultă în urma etapelor de antrenare-testare.