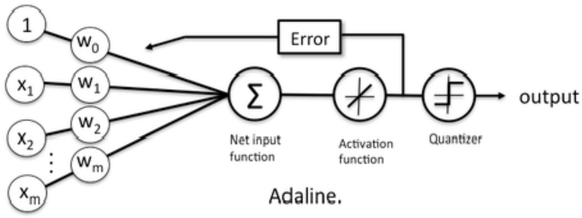
Perceptronul și rețele de perceptroni



Structura unui perceptron cu *m* ponderi. Functia de predictie a perceptronului este $y_{hat} = sign(\sum_{i=0}^{i=m} x_i * w_i)$.

1. Perceptronul

Perceptronul este un clasificator liniar. Predictia clasificatorului pentru exemplul $X = \{x_1, x_2, ..., x_n\}$ este $y_{hat} = f(\sum_{i=1}^{i=n} x_i * w_i + b)$, unde $W = \{w_1, w_2, ..., w_n\}$ si $b = w_0$ sunt ponderile, respectiv bias-ul perceptronului, iar f este functia de transfer (numita si functie de activare). Putem inlocui suma din calcularea lui y_{hat} cu produsului dintre vectorul datelor de intrare X si matricea ponderilor W, rezultatand $y_{hat} = f(X \cdot W + b)$.

2. Algoritmul Widrow-Hoff.

Algoritmul Widrow-Hoff, numit si *metoda celor mai mici patrate* (*Least mean squares*), este un algoritm de optimizare a erorii perceptronului pe baza metodei coborarii pe gradient tinand cont doar de eroare de la exemplul curent.

Regula de actualizare foloseste derivata partiala a functiei de pierdere, in functie de ponderi si bias. In continuare vom calcula detaliat derivatele partiale ale functiei de pierdere. Functia de activare a perceptronului din algoritmul Widrow-Hoff este *identitatea* ($f(x) = x$).

$$loss = \frac{(y_{hat} - y)^2}{2}, \ unde\ y_{hat} = X \cdot W + b,\ iar\ y\ este\ eticheta\ lui\ X$$

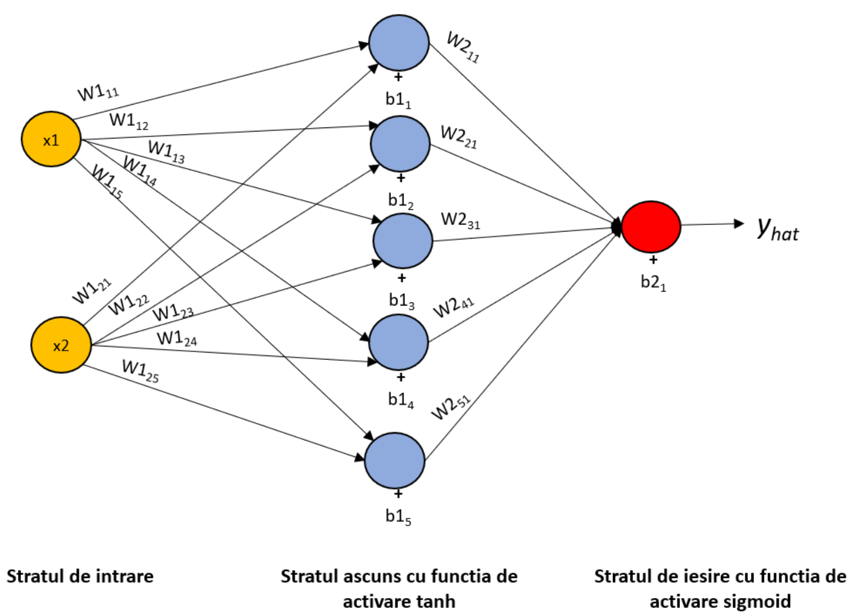
∂W	∂b
$\frac{\partial loss}{\partial W} = \frac{\partial \frac{(y_{hat} - y)^2}{2}}{\partial W}$ $\frac{\partial loss}{\partial W} = \frac{\partial \frac{(x \cdot W + b - y)^2}{2}}{\partial W}$	$\frac{\partial loss}{\partial b} = \frac{\partial \frac{(y_{hat} - y)^2}{2}}{\partial b}$ $\frac{\partial loss}{\partial b} = \frac{\partial \frac{(x \cdot W + b - y)^2}{2}}{\partial b}$

$\frac{\partial loss}{\partial W} = \frac{2 \cdot (x \cdot W + b - y) \cdot \frac{\partial(x \cdot W + b - y)}{\partial W}}{2}$	$\frac{\partial loss}{\partial b} = \frac{2 \cdot (x \cdot W + b - y) \cdot \frac{\partial(x \cdot W + b - y)}{\partial b}}{2}$
$\frac{\partial loss}{\partial W} = (x \cdot W + b - y) \cdot x$	$\frac{\partial loss}{\partial b} = (x \cdot W + b - y) \cdot 1$
$\frac{\partial loss}{\partial W} = (y_{hat} - y) \cdot x$	$\frac{\partial loss}{\partial b} = (y_{hat} - y)$

Algoritmul Widrow-Hoff.

1. $X = \{x_0, x_1, ..., x_{T-1}\}, X \in R^{TxN}$ $-$ *datele de intrare*, $Y = \{y_0, y_1, ..., y_{T-1}\}$ $-$ *etichet*
2. $W = \{w_0, w_1, ..., w_{N-1}\} = 0$; $b = 0$ // *initializeaza ponderile cu un vector de* 0 –
3. *pentru* $e = 0: E - 1$ *executa*: // *pentru fiecare epoca*
   a. *amesteca datele de antrenare*
   b. *pentru* $t = 0: T - 1$ *executa*: // *pentru fiecare exemplu* x_t *din* X
      i. $y_{hat} = x_t \cdot W + b$ // *calculeaza predictia*
      ii. $loss = \frac{(y_{hat} - y_t)^2}{2}$
      // *calculeaza eroarea pentru exemplul* x_t
      iii. $W = W - \eta(y_{hat} - y_t)x_t$ // *actualizeaza ponderile folosind* $\frac{\partial loss}{\partial W}$
      iv. $b = b - \eta(y_{hat} - y_t)$ // *actualizeaza bias – ul folosind* $\frac{\partial loss}{\partial b}$

3. Retele feedforward de perceptroni

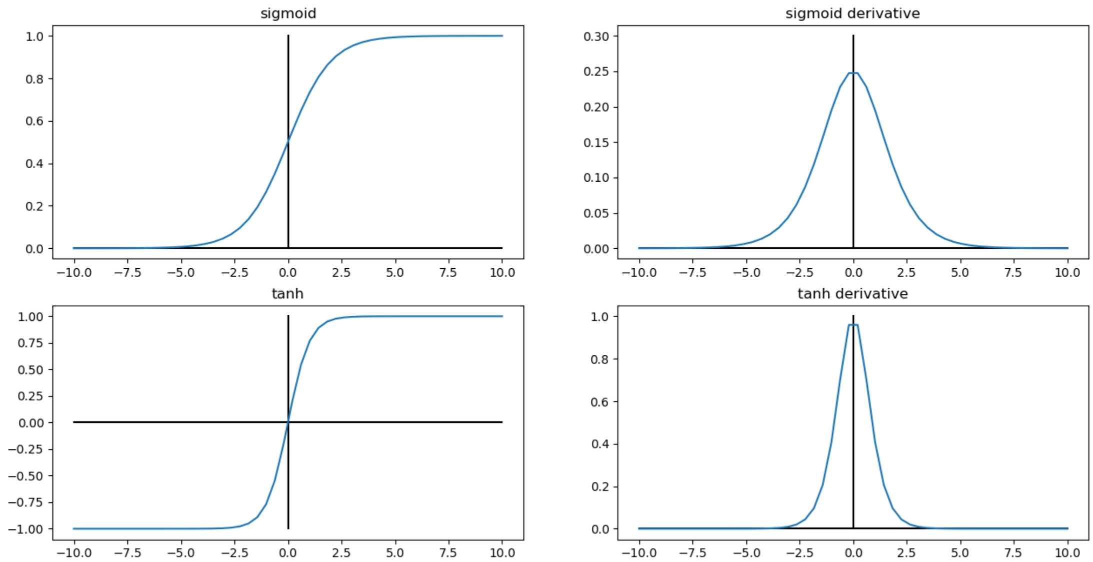


O retea neuronala cu 5 perceptronii pe stratul ascuns si un perceptron pe stratul de iesire.

Retelele neurale feedforward sunt retele de perceptroni grupati pe straturi, in care propagarea informatiei se realizeaza numai dinspre intrare spre iesire (de la stanga la dreapta). Retelele feedforward sunt multistrat, continand mai multe straturi de perceptroni. Perceptronii de pe primul strat sunt singurii care primesc date de intrare din exterior. Perceptronii de pe celelalte straturi (numite *straturi ascunse (hidden layers)*), primesc ca date de intrare rezultatul stratului anterior. Ultimul strat din retea se numeste *strat de iesire (output layer)*.

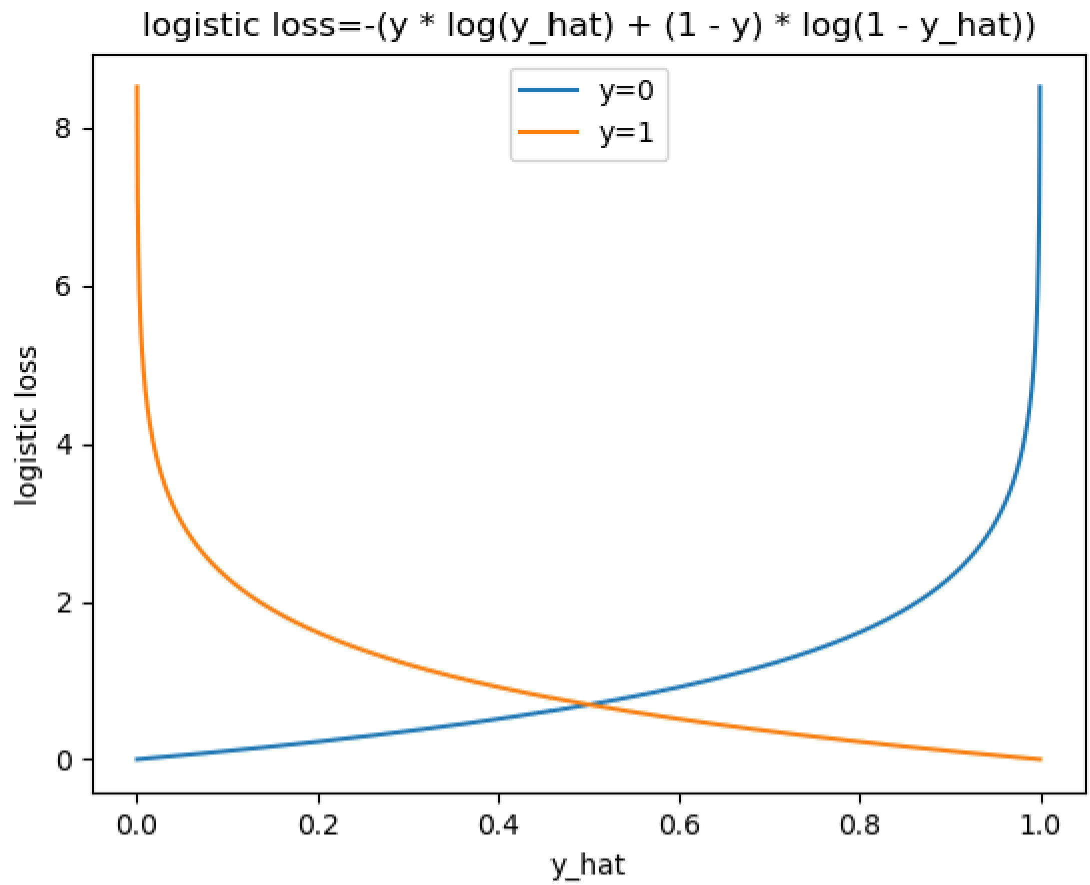
In cadrul laboratorului vom antrena o retea cu un strat ascuns cu ***num_hidden_neurons*** neuroni si functia de activare ***tanh*** si un neuron pe stratul de iesire cu functie de activare ***logistic*** (sigmoid) pentru rezolvarea problemei **XOR**. Predictia retelei pentru un exemplu X este

$y_{hat} = sigmoid(tanh(X \cdot W_1 + b_1) \cdot W_2 + b_2)$.



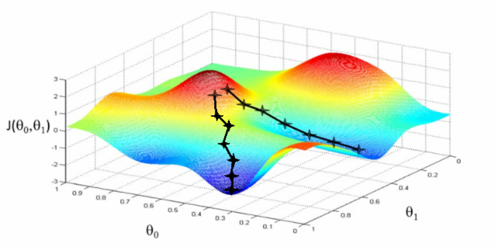
Stânga -sus: graficul functiei sigmoid; *Dreapta-jos*: graficul functiei sigmoid derivat. *Stânga Jos*:graficul funcției tanh; *Dreapta-jos*: graficul functiei tanh derivat.

Functia de pierdere pe care o vom folosi pentru antrenarea retelei este: $logistic\_loss(y_{hat}, y) = -(y * log(y_{hat}) + (1 - y) * log(1 - y_{hat}))$, unde y_{hat} este predicția rețelei pentru exemplul X, iar y este eticheta binara (0 sau 1) a lui X



Linia portocalie: valoarea functiei *logistic loss*, cand y=1, iar *y_hat* variaza intre (0,1). Observam ca cu cat ne apropiem de 1 (pe axa Ox) valoarea functiei scade. Se observa ca daca y=1, valoarea functiei este data doar de produsul din partea stanga (partea dreapta inmultindu-se cu 0).
Linia albastra: valoarea functiei *logistic loss*, cand y=0, iar *y_hat* variaza intre (0,1). Observam ca cu cat ne indepartam de 0 (pe axa Ox) valoarea functiei creste. Se observa ca daca y=0, valoarea functiei este data doar de produsul din partea dreapta (partea stanga inmultindu-se cu 0).

4. Antrenarea retelelor de perceptroni cu algoritmul coborarii pe gradient



Observam ca in functie de initializarea ponderilor putem ajunge in minime locale diferite.

Algoritmul coborarii pe gradient se bazeaza pe derivata de ordinul 1, pentru a gasi minimul functiei de pierdere. Pentru a gasi un minim local al functiei de pierdere, vom actualiza ponderile retelei proportional cu negativul gradientului functiei la pasul curent.

In continuare vom detalia implementarea (pseudo-cod) algoritmului de coborare pe gradient pentru reteaua descrisa anterior.

Pasii algoritmului sunt:

1) Initializare ponderilor - ponderile si bias-ul retelei se initializeaza aleator cu valori mici aproape de 0 sau cu valoare 0.

```
W_1 = random((2, num_hidden_neurons), miu, sigma)
# generam aleator matricea ponderilor stratului ascuns (2 - dimensiunea datelor de intrare, num_hidden_neurons - numarul neuronilor de pe stratul ascuns), cu media miu  si deviatia standard sigma.
b_1 = zeros(num_hidden_neurons) # initializam bias-ul cu 0
W_2 = random((num_hidden_neurons, 1), miu, sigma)
# generam aleator matricea ponderilor stratului de iesire (num_hidden_neurons - numarul neuronilor de pe stratul ascuns, 1 - un neuron pe stratul de iesire), cu media miu  si deviatia standard sigma.
b_2 = zeros(1) # initializam bias-ul cu 0
```

2) Pasul **forward -** Vom defini o metoda forward care calculeaza predictia retelei folosind ponderile actuale si datele de intrare date ca parametri, apoi vom calcula pentru fiecare strat valoarea lui ***z*** (***z*** = inmultirea datelor de intrare cu ponderile si adunarea bias-ului) si valoarea lui ***a*** (***a*** = aplicarea functiei de activare lui ***z***, ($a = f(z)$)).

```
forward(X, W_1, b_1, W_2, b_2)
# X - datele de intrare,  W_1, b_1, W_2 si b_2 sunt ponderile retelei
z_1 =  X * W_1 + b_1
a_1 = tanh(z_1)
z_2 = a_1 * W_2 + b_2
a_2 = sigmoid(z_2)
return z_1, a_1, z_2, a_2 # vom returna toate elementele calculate
```

3) Calculam valoarea functiei de eroare (logistic loss) si acuratetea.

```
z_1, a_1, z_2, a_2 = forward(X, W_1, b_1, W_2, b_2)
loss = (-y .* log(a_2) - (1 - y) .* log(1 - a_2)).mean()
accuracy = (round(a_2) == y).mean()
```

functia	***derivata***	***Derivata functiei compuse***
$sigmoid(x)\ =\ \frac{}{1-}$	$sigmoid(x)\ *\ (1\ -\ sigmoi$	$sigmoid(u(x))\ *\ (1\ -\ sigmoid(u(x)))\ *$
$tanh(x)\ =\ \frac{e^{2x}-}{e^{2x}+}$	$1\ -\ tanh(x)^2$	$(1\ -\ tanh(u(x))^2)\ *\ u(x)'$
x	1	$-$
$c\ *\ x$	c	$-$
$ln\ x$	$\frac{1}{x}$	$\frac{u(x)'}{u(x)}$
x^n	$n\ *\ x^{n-1}$	$n\ *\ u(x)^{n-1}\ *\ u(x)'$

Derivatele functiilor folosite in laborator.

4) Pasul **backward** - vom defini o metoda backward care calculeaza derivata functiei de eroare pe directiile ponderilor, respectiv a fiecarui bias. Vom incepe calculul cu derivata functiei de pierdere pe directia ***z_2*** folosind regula de inlantuire (chain-rule) a derivatelor.

$$\frac{\partial loss}{\partial z_2} = \frac{\partial loss}{\partial a_2} * \frac{\partial a_2}{\partial z_2}\ |\ aplicam\ regula\ de\ inlantuire$$

$$Stim\ ca\ \ a_2 = sigmoid(z_2)\ ,\ folosind\ derivata\ functiei\ sigmoid\ \ rezulta:$$

$$\frac{\partial a_2}{\partial z_2} = \frac{sigmoid(z_2)}{\partial z_2} = sigmoid(z_2)\ *\ (1\ -\ sigmoid(z_2))\ =\ a_2 * (1 - a_2)$$

$$\frac{\partial loss}{\partial a_2} = \frac{\partial(\ -y * log(a_2)\ -\ (1 - y) * log(1 - a_2))}{\partial a_2}$$

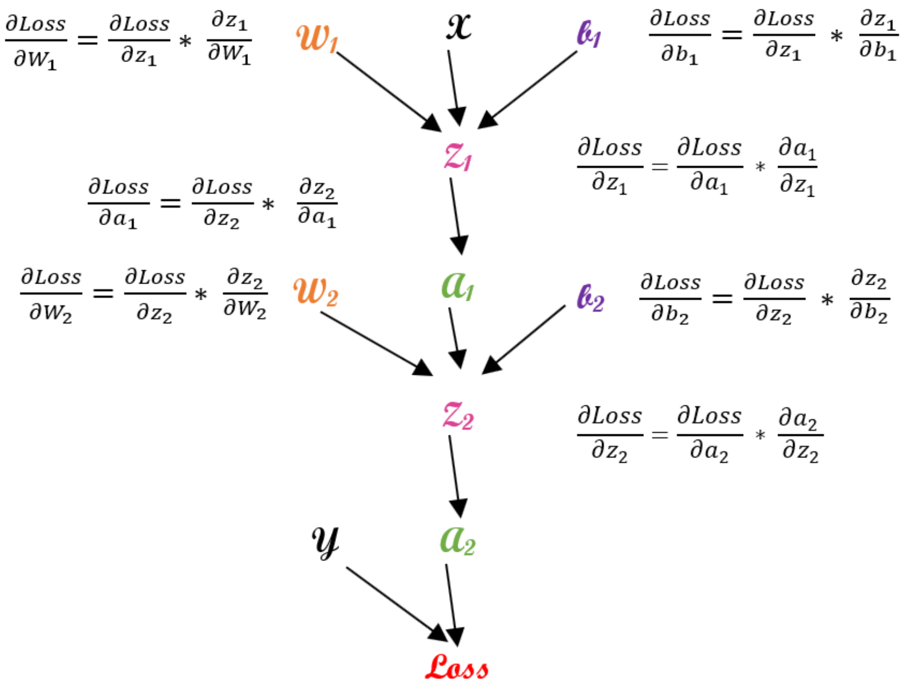
$$\frac{\partial loss}{\partial a_2} = (\frac{-y}{a_2} + \frac{1 - y}{1 - a_2})$$

$$\frac{\partial loss}{\partial a_2} = \frac{-y + a_2 * y\ + a_2 - a_2 * y}{a_2 * (1 - a_2)}$$

$$\frac{\partial loss}{\partial z_2} = \frac{\partial loss}{\partial a_2} * \frac{\partial a_2}{\partial z_2}$$

$$\frac{\partial loss}{\partial z_2} = \frac{-y + a_2 * y + a_2 - a_2 * y}{a_2 * (1 - a_2)} * a_2 * (1 - a_2)$$

$$\frac{\partial loss}{\partial z_2} = a_2 - y$$



Calcularea derivatele partiale pe directiile ponderilor si a fiecarui bias folosind regula de inlantuire.

```
backward(a_1, a_2, z_1, W_2, X, Y, num_samples)
dz_2 = a_2 - y # derivata functiei de pierdere (logistic loss) in
functie de z
dw_2 = (a_1.T * d_z2) / num_samples
# der(L/w_2) = der(L/z_2) * der(dz_2/w_2) = dz_2 * der((a_1 * W_2
+ b_2)/ W_2)
db_2 = sum(dz_2) / num_samples
# der(L/b_2) = der(L/z_2) * der(z_2/b_2) = dz_2 * der((a_1 * W_2 +
b_2)/ b_2)
# primul strat
da_1 = dz_2 * W_2.T
# der(L/a_1) = der(L/z_2) * der(z_2/a_1) = dz_2 * der((a_1 * W_2 +
b_2)/ a_1)
dz_1 = da_1 .* tanh_derivative(z_1)
# der(L/z_1) = der(L/a_1) * der(a_1/z1) = da_1 .* der((tanh(z_1))/
z_1)
```

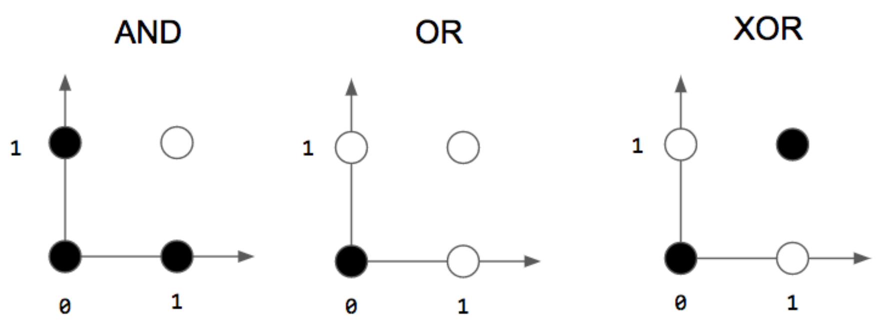
```
dw_1 = X.T * dz_1 / num_samples
# der(L/w_1) = der(L/z_1) * der(z_1/w_1) = dz_1 * der((X * W_1 +
b_1)/ W_1)
db_1 = sum(dz_1) / num_samples
# der(L/b_1) = der(L/z_1) * der(z_1/b_1) = dz_1 * der((X * W_1 +
b_1)/ b_1)
return dw_1, db_1, dw_2, db_2
```

5) Actualizarea ponderilor - ponderile se actualizeaza proportional cu negativul mediei derivatelor din batch (mini-batch).

```
W_1 -= lr * dw_1 # lr - rata de invatare (learning rate)
b_1 -= lr * db_1
W_2 -= lr * dw_2
b_2 -= lr * db_2
```

6) Pentru a antrena o retea neuronala cu ajutorul algoritmului coborarii pe gradient trebuie sa:
   a) Stabilim numarul de epoci
   b) Stabilim rata de invatare
   c) Sa initiliazam ponderile (pasul 1)
   d) Sa amestecam datele la fiecare epoca
   e) Sa luam un subset din multimea (sau toata multimea) de antrenare si sa executam pasii 2, 3, 4, 5 pana la convergenta.

Exercitii



1. Se da urmatoare multime de antrenare X =[[0, 0], [0, 1], [1, 0], [1, 1]], y = [-1, 1, 1, 1]. Sa se gaseasca o dreapta care separa perfect multimea de antrenare.
2. Antrenati un Perceptron cu algoritmul Widrow-Hoff pe multimea de antrenare de la exercitiul anterior timp de 70 epoci cu rata de invatare 0.1. Care este acuratetea pe multimea de antrenare? Apelati functia *plot_decision_boundary* la fiecare pas al algoritmului pentru a afisa dreapta de decizie.

```
import matplotlib.pyplot as plt

def compute_y(x, W, bias):
    # dreapta de decizie
    # [x, y] * [W[0], W[1]] + b = 0
    return (-x * W[0] - bias) / (W[1] + 1e-10)

def plot_decision_boundary(X, y , W, b, current_x, current_y):
    x1 = -0.5
    y1 = compute_y(x1, W, b)
    x2 = 0.5
    y2 = compute_y(x2, W, b)
    # sterge continutul ferestrei
    plt.clf()
    # ploteaza multimea de antrenare
    color = 'r'
    if(current_y == -1):
        color = 'b'
    plt.ylim((-1, 2))
    plt.xlim((-1, 2))
    plt.plot(X[y == -1, 0], X[y == -1, 1], 'b+')
    plt.plot(X[y == 1, 0], X[y == 1, 1], 'r+')
    # ploteaza exemplul curent
    plt.plot(current_x[0], current_x[1], color+'s')
    # afisarea dreptei de decizie
    plt.plot([x1, x2] ,[y1, y2], 'black')
    plt.show(block=False)
    plt.pause(0.3)
```

3. Antrenati un Perceptron cu algoritmul Widrow-Hoff pe multimea de antrenare X =[[0, 0], [0, 1], [1, 0], [1, 1]], y = [-1, 1, 1, -1]. Care este acuratetea pe multimea de antrenare? Apelati functia *plot_decision_boundary* la fiecare pas al algoritmului pentru a afisa dreapta de decizie.
4. Antrenati o retea neuronala pentru rezolvarea problemei XOR cu arhitectura retelei descrise in **3**, si algoritmul coborarii pe gradient descris in **4**, folosind 70 epoci, rata de invatare 0.5, media si deviatia standard pentru initializarea ponderilor 0, respectiv 1, si 5 neuroni pe stratul ascuns. Afisati valoarea erorii si a acuratetii la fiecare epoca. Apelati functia *plot_decision* la fiecare pas al algoritmului pentru a afisa functia de decizie.

```
def compute_y(x, W, bias):
    # dreapta de decizie
    # [x, y] * [W[0], W[1]] + b = 0
    return (-x*W[0] - bias) / (W[1] + 1e-10)

def plot_decision(X_, W_1, W_2, b_1, b_2):
```

```
# sterge continutul ferestrei
plt.clf()
# ploteaza multimea de antrenare
plt.ylim((-0.5, 1.5))
plt.xlim((-0.5, 1.5))
xx = np.random.normal(0, 1, (100000))
yy = np.random.normal(0, 1, (100000))
X = np.array([xx, yy]).transpose()
X = np.concatenate((X, X_))
_, _, _, output = forward(X, W_1, b_1, W_2, b_2)
y = np.squeeze(np.round(output))
plt.plot(X[y == 0, 0], X[y == 0, 1], 'b+')
plt.plot(X[y == 1, 0], X[y == 1, 1], 'r+')
plt.show(block=False)
plt.pause(0.1)
```