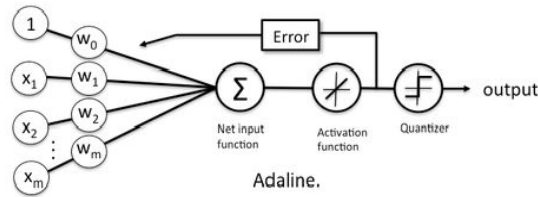


Perceptronul și rețele de perceptroni



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

1. Perceptronul

Perceptronul este un clasificator liniar. Predicția clasificatorului pentru exemplul

$X = \{x_1, x_2, \dots, x_n\}$ este $y_{hat} = f(\sum_{i=1}^{i=n} x_i * w_i + b)$, unde $W = \{w_1, w_2, \dots, w_n\}$ și $b = w_0$

sunt ponderile, respectiv bias-ul perceptronului, iar f este funcția de transfer (numită și funcție de activare). Putem înlocui suma din calcularea lui y_{hat} cu produsul dintre vectorul datelor de intrare X și matricea ponderilor W , rezultând $y_{hat} = f(X \cdot W + b)$.

2. Algoritmul Widrow-Hoff.

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

Regula de actualizare folosește derivata parțială a funcției de pierdere, în funcție de ponderi și bias. În continuare vom calcula detaliat derivatele parțiale ale funcției de pierdere. Funcția de activare a perceptronului din algoritmul Widrow-Hoff este *identitatea* ($f(x) = x$).

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

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

$$\frac{\partial \text{loss}}{\partial W} = (x \cdot W + b - y) \cdot x$$

$$\frac{\partial \text{loss}}{\partial W} = (y_{\text{hat}} - y) \cdot x$$

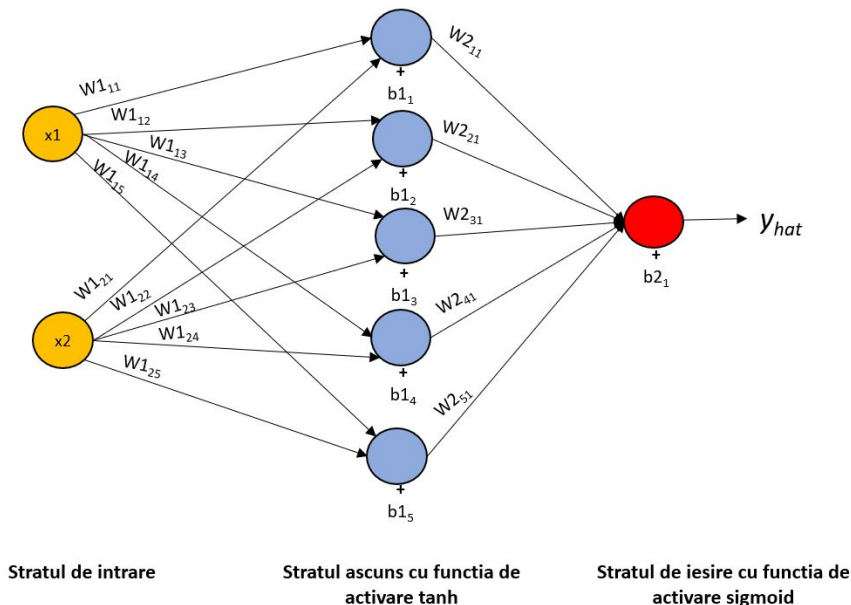
$$\frac{\partial \text{loss}}{\partial b} = (x \cdot W + b - y) \cdot 1$$

$$\frac{\partial \text{loss}}{\partial b} = (y_{\text{hat}} - y)$$

Algoritmul Widrow-Hoff.

1. $X = \{x_0, x_1, \dots, x_{T-1}\}$, $X \in R^{T \times N}$ – datele de intrare, $Y = \{y_0, y_1, \dots, y_{T-1}\}$ – etichetele
2. $W = \{w_0, w_1, \dots, w_{N-1}\} = 0$; $b = 0$ // initializeaza ponderile cu un vector de 0-uri
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. $y_{\text{hat}} = x_t \cdot W + b$ // calculeaza predictia
 - ii. $\text{loss} = \frac{(y_{\text{hat}} - y_t)^2}{2}$ // calculeaza eroarea pentru exemplul x_t
 - iii. $W = W - \eta(y_{\text{hat}} - y_t)x_t$ // actualizeaza ponderile folosind $\frac{\partial \text{loss}}{\partial W}$
 - iv. $b = b - \eta(y_{\text{hat}} - y_t)$ // actualizeaza bias-ul folosind $\frac{\partial \text{loss}}{\partial b}$

3. Retele feedforward de perceptroni

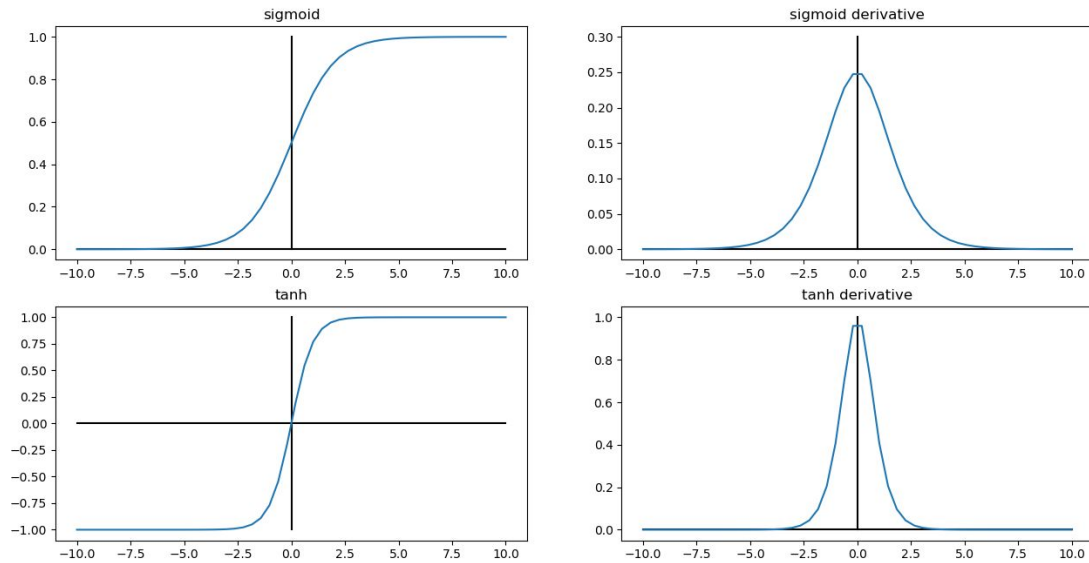


O retea neuronală cu 5 perceptroni pe stratul ascuns și un perceptron pe stratul de ieșire.

Retelele neurale feedforward sunt rețele de perceptroni grupate pe straturi, în care propagarea informației se realizează numai dinspre intrare spre ieșire (de la stânga la dreapta). Retelele feedforward sunt multistrat, conținând 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 rețea se numește *strat de ieșire (output layer)*.

În cadrul laboratorului vom antrena o rețea cu un strat ascuns cu **num_hidden_neurons** neuroni și funcția de activare **tanh** și un neuron pe stratul de ieșire cu funcție de activare **logistic** (sigmoid) pentru rezolvarea problemei **XOR**. Predicția rețelei pentru un exemplu X este $y_{\hat{a}} = \text{sigmoid}(\text{tanh}(X \cdot W_1 + b_1) \cdot W_2 + b_2)$.

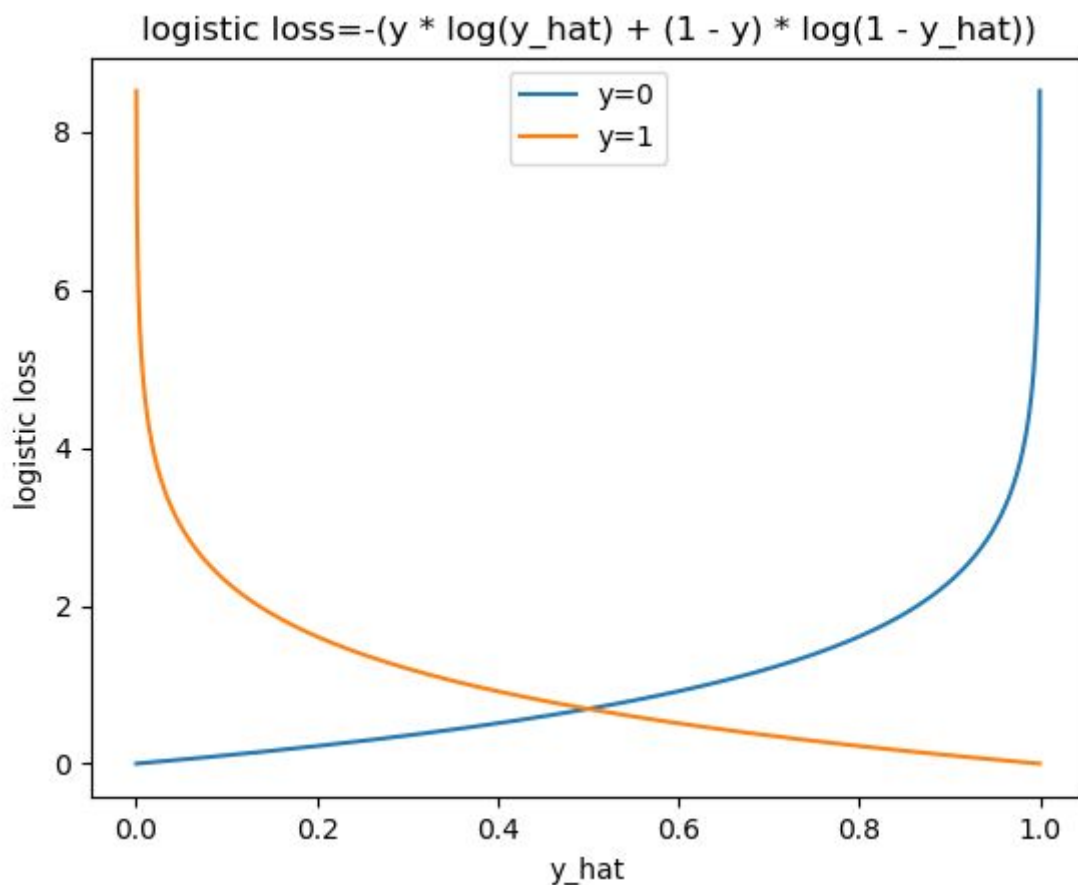


Stanga -sus: graficul funcției sigmoid; *Dreapta-sus:* graficul funcției sigmoid derivat.
Stanga-jos: graficul funcției tanh; *Dreapta-jos:* graficul funcției tanh derivat.

Funcția de pierdere pe care o vom folosi pentru antrenarea rețelei este:

$$\text{logistic_loss}(y_{\hat{a}}, y) = -(y * \log(y_{\hat{a}}) + (1 - y) * \log(1 - y_{\hat{a}})), \text{ unde}$$

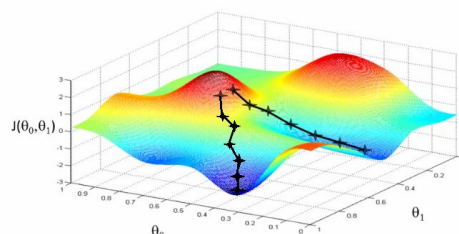
$y_{\hat{a}}$ este predicția rețelei pentru exemplul X , iar y este eticheta binară (0 sau 1) a lui X



Linia portocalie: valoarea funcției *logistic loss*, când $y=1$, iar y_hat variază între (0,1). Observăm că cu cât ne apropiem de 1 (pe axa Ox) valoarea funcției scade. Se observă că dacă $y=1$, valoarea funcției este dată doar de produsul din partea stângă (partea dreaptă înmulțindu-se cu 0).

Linia albastră: valoarea funcției *logistic loss*, când $y=0$, iar y_hat variază între (0,1). Observăm că cu cât ne îndepărtăm de 0 (pe axa Ox) valoarea funcției crește. Se observă că dacă $y=0$, valoarea funcției este dată doar de produsul din partea dreaptă (partea stângă înmulțindu-se cu 0).

4. Antrenarea rețelelor de percepștroni cu algoritmul coborării pe gradient



Observăm că în funcție de inițializarea ponderilor putem ajunge în minime locale diferite.

Algoritmul coborării pe gradient se bazează pe derivata de ordinul 1, pentru a găsi minimul funcției de pierdere. Pentru a găsi un minim local al funcției de pierdere, vom actualiza ponderile rețelei proporțional cu negativul gradientului funcției la pasul curent.

În continuare vom detalia implementarea (pseudo-cod) algoritmului de coborare pe gradient pentru rețeaua descrisă anterior.

Pasii algoritmului sunt:

- 1) **Initializare ponderilor** - ponderile și bias-ul rețelei se initializează 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 calculează predicția rețelei folosind ponderile actuale și datele de intrare date ca parametri, apoi vom calcula pentru fiecare strat valoarea lui z (z = înmulțirea datelor de intrare cu ponderile și adunarea bias-ului) și valoarea lui a (a = aplicarea funcției 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
rețelei
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) **Calculăm valoarea funcției de eroare (logistic loss) și 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)) / num_samples
accuracy = (round(a_2) == y) / num_samples
```

<i>functia</i>	<i>derivata</i>	<i>Derivata functiei compuse</i>
$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$	$\text{sigmoid}(x) * (1 - \text{sigmoid}(x))$	$\text{sigmoid}(u(x)) * (1 - \text{sigmoid}(u(x))) * x'$
$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$	$1 - \tanh(x)^2$	$(1 - \tanh(u(x))^2) * x'$
x	1	—
$c * x$	c	—
$\ln x$	$\frac{1}{x}$	$\frac{x'}{u(x)}$
x^n	$n * x^{n-1}$	$n * u(x)^{n-1} * x'$
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.

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

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

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

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

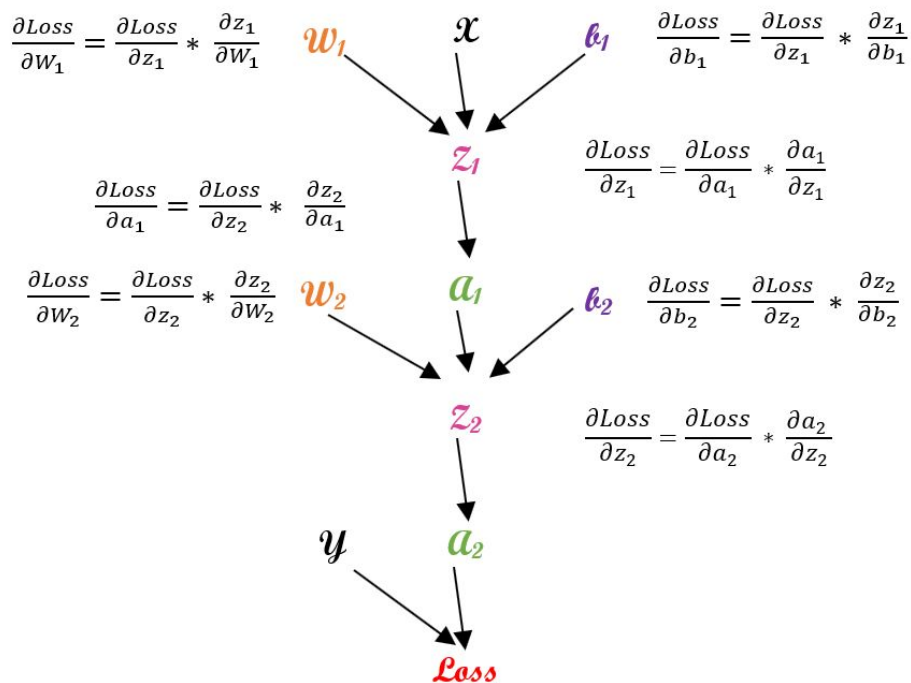
$$\frac{\partial \text{loss}}{\partial a_2} = \left(\frac{-y}{a_2} + \frac{1-y}{1-a_2} \right)$$

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

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

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

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



Calcularea derivatele parțiale pe direcțiile ponderilor și a fiecărui bias folosind regula de înlănțuire.

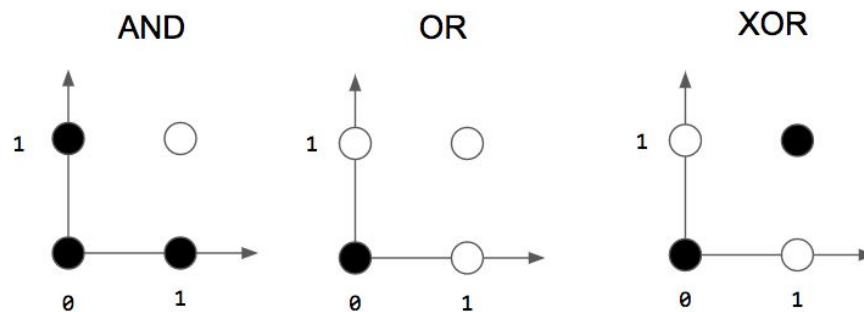
```
backward(a_1, a_2, z_1, w_2, X, Y, num_samples)
dz_2 = a_2 - y # derivata functiei de pierdere (Logistic Loss) in
funcție de z
dw_2 = (a_1.T * dz_2) / num_samples
# der(L/w_2) = der(L/z_2) * der(dz_2/w_2) = dz_2 * der((a_1 * w_2
+ b_2)/ w_2)
db_2 = sum(dz_2) / num_samples
# der(L/b_2) = der(L/z_2) * 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/z_1) = 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 neuronală cu ajutorul algoritmului coborării pe gradient trebuie să:
- Stabilim numărul de epoci
 - Stabilim rata de învățare
 - Să inițiazăm ponderile (pasul 1)
 - Să amestecăm datele la fiecare epocă
 - Să luăm un subset din mulțimea (sau toată mulțimea) de antrenare și să executăm pașii 2, 3, 4, 5 până la convergență.

Exercitii



- Se dau următoarele mulțimi de antrenare $X = [[0, 0], [0, 1], [1, 0], [1, 1]]$, $y = [-1, 1, 1, 1]$. Să se găsească o dreaptă care separă perfect mulțimea de antrenare.
- Antrenați un Perceptron cu algoritmul Widrow-Hoff pe mulțimea de antrenare de la exercitiul anterior timp de 70 epoci cu rata de învățare 0.1. Care este acuratețea pe mulțimea de antrenare? Apelați funcția `plot_decision_boundary` la fiecare pas al algoritmului pentru a afișa 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 $X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$, $y = [-1, 1, 1, -1]$. 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 neuronală pentru rezolvarea problemei XOR cu arhitectura rețelei descrise în 3, și algoritmul coborării pe gradient descris în 4, folosind 70 epoci, rata de învățare 0.5, media și deviația standard pentru initializarea ponderilor 0, respectiv 1, și 5 neuroni pe stratul ascuns. Afisati valoarea erorii și 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_pass(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)
```