Learn an XOR Neural Network using gradient-based optimization

Steps

1. Setting the scene
2. Model 1: linear, one layer, no activation function
3. Model 2: almost linear, one layer, sigmoid activation function
4. Model 3: one hidden layer with linear and one output layer with sigmoid activation
5. Model 4: one hidden layer with relu and one output layer with sigmoid activation
6. Comparison with Model 4 learned with Tensorflow

Setting the scene

All models m_i follow the standard Neural Network architecture:

The models vary in the number of layers, the numbers of neurons per layer and in the activation function.

During learning, we optimize the mean squared error MSE of the models m for the model parameters $\mathbf{w}, \mathbf{b}$:

$$MSE(\mathbf{w}, \mathbf{b}, m, X, Y) = \frac{1}{N} \sum_{i=1}^{N} (y_i - m(\mathbf{w}, \mathbf{b}, x_i))^2$$

In [ ]:
```
def mse(ws, m, X, Y):
    N = len(X)
    err = 0
    for i in range(N):
        y_hat = m(ws, X[i])
        err += (Y[i] - y_hat)**2
    return err / N
```

In other words, we find $\arg\min_{\mathbf{w},\mathbf{b}} MSE(\mathbf{w}, \mathbf{b}, m, X, Y)$. Since we implement a binary, logic function (two arguments, two values per argument), $N = 2^2 = 4$ and X, Y define the XOR function:

In [ ]:
```
import numpy as np

X = np.array([[0,0],[1,0],[0,1],[1,1]])
Y = np.array([0,1,1,0])
```

Model 1: linear, one layer, no activation function

The first model is a (too) simple linear Neural Network model. It consists of one neuron connected to the input x_1 and x_2 and an identity, i.e., no effective, activation function. The neuron and the whole model m_1 implements $m_1(\mathbf{w}, b, x) = w_1 x_1 + w_2 x_2 + b$.

In [ ]:
```
m1 = lambda ws, x: ws[0] * x[0] + ws[1] * x[1] + ws[2]
mse1 = lambda ws: mse(ws, m1, X, Y)
```

As the Tensorflow default, we implement the Glorot uniform initializer for setting the initial weights $\mathbf{w}_0$. It draws samples from a uniform random distribution within $[-limit, limit]$, where $limit = \sqrt{\frac{6}{in+out}}$, and where in and out is the number of input and output units, resp. The initial bias b_0 is set to 0.

In [ ]:
```
import math
import random

input = 2
output = 1
limit = math.sqrt(6/(input+output))
random.seed(1) # for reproducibility
ws0 = [random.uniform(-limit, limit), random.uniform(-limit, limit), 0]
print(f'ws0 = {ws0}')
```

```
ws0 = [-1.0341740897295608, 0.9826910056052014, 0]
```

We are ready to assess MSE for m_1 for this initial weights setting.

In [ ]:
```
mse1(ws0)
```

Out[ ]:
```
1.0352035841412506
```

The gradient of $MSE(\mathbf{w}, b)$ for any $\mathbf{w}, b$ is defiend as:

$$\begin{aligned}
\nabla MSE(\mathbf{w}, b) &= \left[\frac{\partial MSE(\mathbf{w}, b)}{\partial w_1}, \frac{\partial MSE(\mathbf{w}, b)}{\partial w_2}, \frac{\partial MSE(\mathbf{w}, b)}{\partial b}\right]^T \\
&= \frac{1}{4}\left[\frac{\partial \sum_{i=1}^4 (y_i - m_1(\mathbf{w}, b, x_i))^2}{\partial w_1}, \frac{\partial \sum_{i=1}^4 (y_i - m_1(\mathbf{w}, b, x_i))^2}{\partial w_2}, \frac{\partial \sum_{i=1}^4 (y_i}{} \right. \\
&= \frac{1}{4}\left[\sum_{i=1}^4 2(y_i - m_1(\mathbf{w}, b, x_i))\frac{-\partial m_1(\mathbf{w}, b, x_i)}{\partial w_1}, \sum_{i=1}^4 2(y_i - m_1(\mathbf{w}, b, x_i))\frac{-\partial}{} \right. \\
&= -\frac{1}{2}\left[\sum_{i=1}^4 (y_i - m_1(\mathbf{w}, b, x_i))\frac{\partial m_1(\mathbf{w}, b, x_i)}{\partial w_1}, \sum_{i=1}^4 (y_i - m_1(\mathbf{w}, b, x_i))\frac{\partial m_1(\mathbf{v}}{\partial} \right.
\end{aligned}$$

We can plug in the function m_1 and the first derivative of m_1 wrt. w_1, w_2, and b resp.

$$\frac{\partial m_1(\mathbf{w}, b, x_i)}{\partial w_j} = \frac{\partial(\sum_{j=1}^{2} w_j x_{ij}) + b}{\partial w_j} = x_{ij}$$
$$\frac{\partial m_1(\mathbf{w}, b, x_i)}{\partial b} = \frac{\partial(\sum_{j=1}^{2} w_j x_{ij}) + b}{\partial b} = 1$$

We have already defined the function m_1. Let's also define the functions of the first derivative of m_1 wrt. w_1, w_2, and b resp. For the sake of generality, we keep the parameters $\mathbf{w}, b$ that are actually not needed for derivatives of this concrete model m_1.

In [ ]:
```
g1 = lambda ws,x: x[0]
g2 = lambda ws,x: x[1]
g3 = lambda ws,x: 1
gradients1 = [g1, g2, g3]
```

The gradient of MSE can now be defined using m_1 and the first derivative of m_1 wrt. w_1, w_2, and b, resp., as parameters.

In [ ]:
```
def grad_mse(ws, m, grads, X, Y):
    N = len(X)
    M = len(ws)
    grad_ws = np.zeros((M))
    for i in range(N):
        xi = X[i,:]
        yi = Y[i]
        tmp = yi - m(ws, xi)
        for j in range(M):
            grad_ws[j] = grad_ws[j] + tmp*grads[j](ws, xi)
    grad_ws = grad_ws / -2
    return grad_ws
```

Finally, we are ready to define the gradient descent function optimizing $\mathbf{w}, b$ by iterating over:

$$(\mathbf{w}_{k+1}, b_{k+1}) = (\mathbf{w}_k, b_k) - \varepsilon \nabla MSE(\mathbf{w}_k, b_k)$$

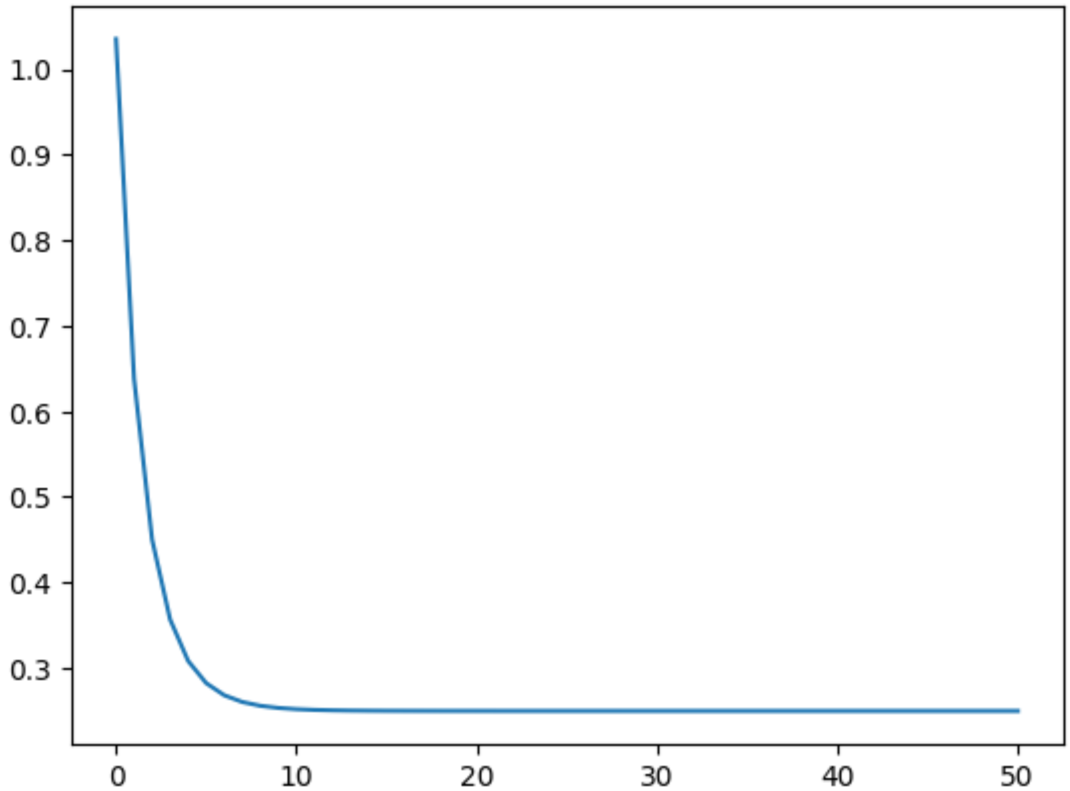
staring with $(\mathbf{w}_0, b_0)$.

In [ ]:
```
import matplotlib.pyplot as plt

def grad_desc_mse(K, ws, learning_eps, loss, grad_loss, verbose):
    hist = np.zeros((K+1))
    hist[0] = loss(ws)
    for k in range(K):
        grad_ws = grad_loss(ws)
        old_ws = ws
        ws = old_ws - learning_eps * grad_ws
        if verbose:
            plt.plot([old_ws[0], ws[0]], [old_ws[1], ws[1]])

        hist[k+1] = loss(ws)
    return ws, hist
```

In [ ]:
```
grad_loss = lambda ws: grad_mse(ws, m1, gradients1, X, Y)
K = 50
learning_eps = 0.5

ws, hist = grad_desc_mse(K, ws0, learning_eps, mse1, grad_loss, False)
print(f'ws = {ws}')
print(f'history = {hist}')
```

```
ws = [3.06534067e-05 3.17956011e-05 4.99962963e-01]
history = [1.03520358 0.63820097 0.45027301 0.35680191 0.3083812  0.28250154
 0.26834683 0.26047332 0.25603787 0.25351378 0.25206467 0.25122569
 0.25073574 0.25044695 0.25027501 0.25017149 0.25010841 0.25006948
 0.25004512 0.25002967 0.25001974 0.25001327 0.25000901 0.25000616
 0.25000425 0.25000294 0.25000205 0.25000143 0.250001   0.25000071
 0.2500005  0.25000035 0.25000025 0.25000018 0.25000012 0.25000009
 0.25000006 0.25000004 0.25000003 0.25000002 0.25000002 0.25000001
 0.25000001 0.25000001 0.25       0.25       0.25       0.25
 0.25       0.25       0.25      ]
```

In [ ]:
```
plt.plot(range(len(hist)), hist.tolist())
plt.show()
```



In [ ]:
```python
my_xor1 = lambda x1, x2: round(ws[0] * x1 + ws[1] * x2 + ws[2])
for i in range(len(X)):
    print(f'my_xor1({X[i,0]},{X[i,1]})={my_xor1(X[i,0], X[i,1])}')
```

```
my_xor1(0,0)=0
my_xor1(1,0)=0
my_xor1(0,1)=0
my_xor1(1,1)=1
```

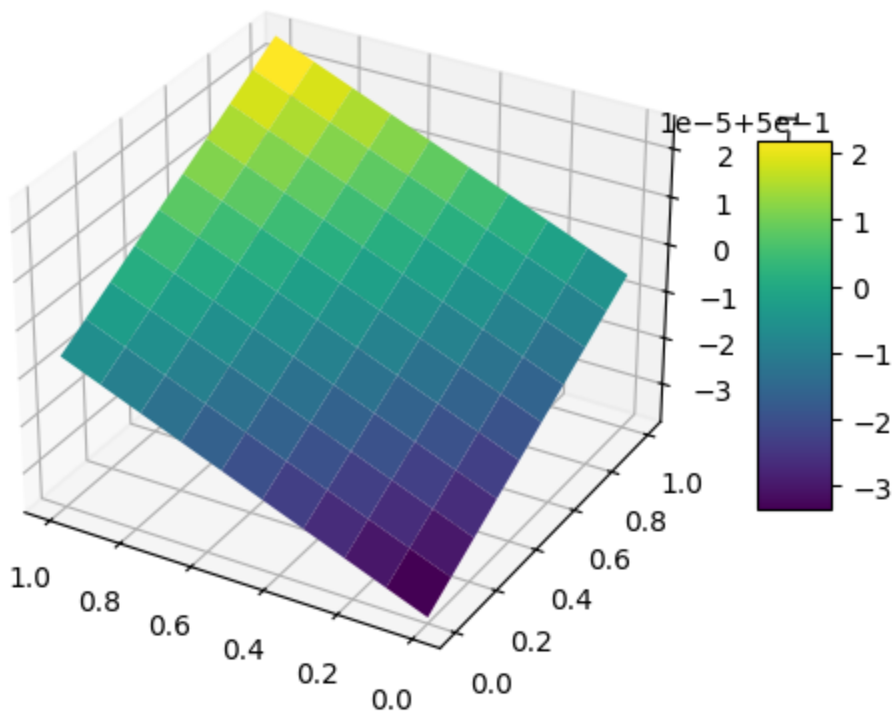
In [ ]:
```python
def acc(X,Y,f):
    correct = 0
    N = len(X)
    for i in range(N):
        y_hat = f(X[i,0], X[i,1])
        if y_hat == Y[i]:
            correct += 1
        else:
            print(f'({X[i,0]},{X[i,1]})={y_hat}, but should be {Y[i]}')
    return correct/N
```

In [ ]:
```python
print(f'accuracy = {acc(X,Y,my_xor1)}')
```

```
(1,0)=0, but should be 1
(0,1)=0, but should be 1
(1,1)=1, but should be 0
accuracy = 0.25
```

In [ ]:
```
def plot3d(f,A,B,real3d):
    vectorized_func = np.vectorize(f)
    Z = vectorized_func(A,B)
    if real3d:
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        surf = ax.plot_surface(A, B, Z, cmap='viridis')
        fig.colorbar(surf, shrink=0.5, aspect=5)
        ax.invert_xaxis()
    else:
        fig = plt.figure()
        ax = fig.add_subplot(111)
        surf = ax.contour(A, B, Z, cmap='viridis')
        fig.colorbar(surf, shrink=0.5, aspect=5)
    return fig, ax
```

In [ ]:
```
f = lambda x1, x2: ws[0] * x1 + ws[1] * x2 + ws[2]
A, B = np.meshgrid(np.linspace(0,1,10), np.linspace(0,1,10))

plot3d(f, A, B, True)
plt.show()
```



Model 2: linear, one layer, sigmoid activation function

The second model is still a (too) simple Neural Network model. As model m_1, it consists of one neuron connected to the input x_1 and x_2 but it uses a sigmoid (logistic) activation function $s(x) = \frac{1}{1+e^{-x}}$. The cell body still implements $m_1(\mathbf{w}, b, x) = w_1x_1 + w_2x_2 + b$, the whole model m_2 implements:

$$m_2(\mathbf{w}, b, x) = \frac{1}{1 + e^{-m_1(\mathbf{w},b,x)}}$$

Note that we just plug in m_1 into m_2 and m_2 into the loss function MSE to obtain the new composite model and loss functions.

In [ ]:
```
sigmoid = lambda x: 1.0 / (1.0 + math.exp(-x))
m2 = lambda ws, x: sigmoid(m1(ws, x))
mse2 = lambda ws: mse(ws, m2, X, Y)
```

As we use the same initialization, we can assess MSE for m_2 and this initial weights setting.

In [ ]:
```
mse2(ws0)
```

Out[ ]:
```
0.27642845152006257
```

For model m_2, the gradient of $MSE(\mathbf{w}, b)$ for any $\mathbf{w}, b$ is almost identical as before. We just plug in m_2 instead of m_1 and m_2' instead of m_1':

$$\nabla MSE(\vec{w}, b) = -\frac{1}{2}\left[\sum_{i=1}^{4}(y_i - m_2(\mathbf{w}, b, x_i))\frac{\partial m_2(\mathbf{w}, b, x_i)}{\partial w_1}, \sum_{i=1}^{4}(y_i - m_2(\mathbf{w}, b, x_i))\frac{\partial m_2(\mathbf{w}}{\partial}\right.$$

Let $s(x) = \frac{1}{1+e^{-x}}$. Then $\frac{\partial s(x)}{\partial x} = s(x)(1 - s(x))$, see the details here.

$$\begin{aligned}\frac{\partial m_2(\mathbf{w}, b, x_i)}{\partial \mathbf{w}, b} &= \frac{\partial s(m_1(\mathbf{w}, b, x_i))}{\partial \mathbf{w}, b} \\ &= s(m_1(\mathbf{w}, b, x_i))(1 - s(m_1(\mathbf{w}, b, x_i))\frac{\partial m_1(\mathbf{w}, b, x_i)}{\partial \mathbf{w}, b}\end{aligned}$$

We can plug in the function m_1 and the first derivative of m_1 wrt. w_1, w_2, and b, resp., as computed before.

$$\begin{aligned}\frac{\partial m_1(\mathbf{w}, b, x_i)}{\partial w_j} &= \frac{\partial(\sum_{j=1}^{2} w_j x_{ij}) + b}{\partial w_j} = x_{ij} \\ \frac{\partial m_1(\mathbf{w}, b, x_i)}{\partial b} &= \frac{\partial(\sum_{j=1}^{2} w_j x_{ij}) + b}{\partial b} = 1\end{aligned}$$

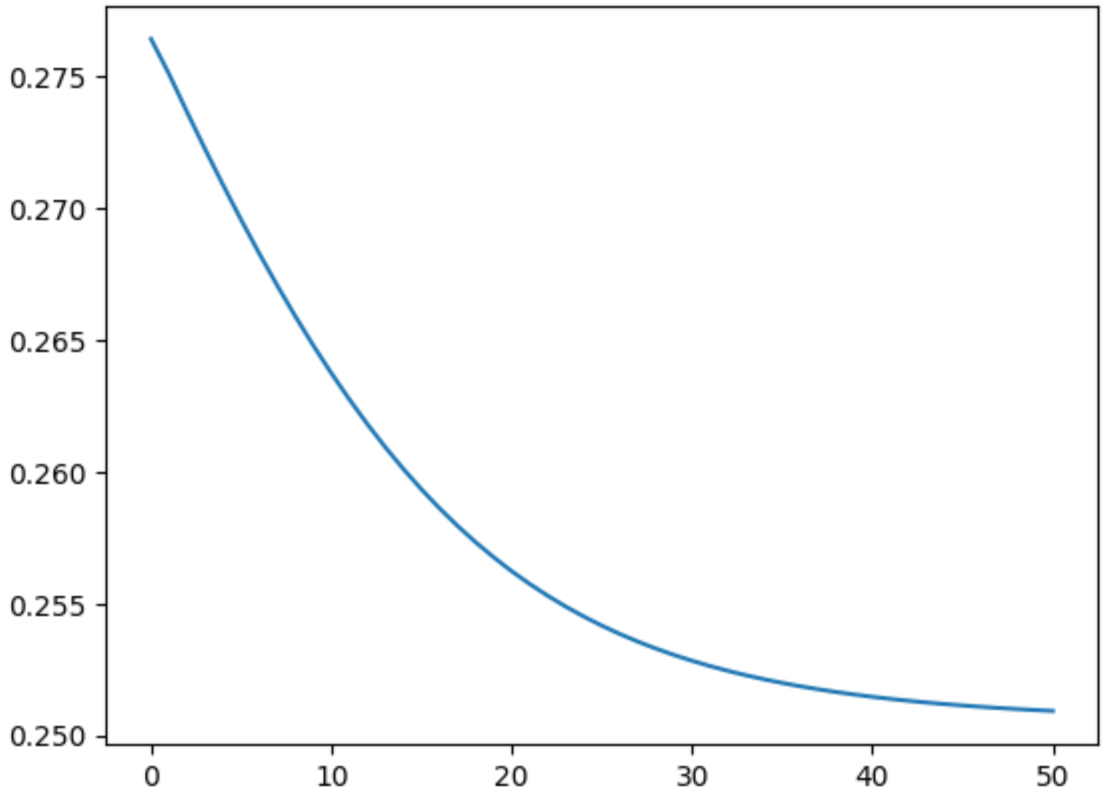
We have already defined the function m_1, the functions of the first derivative of m_1 wrt. w_1, w_2, and b, resp, and the sigmoid activation function f. The gradient calculation depends of these functions and the gradient descent method does not change either. We just need to pulg together the new neural network:

In [ ]:
```
g1 = lambda ws, x: sigmoid(m1(ws,x)) * (1-sigmoid(m1(ws, x)))*gradients1[0](ws,x)
g2 = lambda ws, x: sigmoid(m1(ws,x)) * (1-sigmoid(m1(ws, x)))*gradients1[1](ws,x)
g2 = lambda ws, x: sigmoid(m1(ws,x)) * (1-sigmoid(m1(ws, x)))*gradients1[2](ws,x)

gradients2 = [g1, g2, g3]
```

In [ ]:
```
grad_loss = lambda ws: grad_mse(ws, m2, gradients2, X, Y)
K = 50
learning_eps = 2

ws, hist = grad_desc_mse(K, ws0, learning_eps, mse2, grad_loss, False)
print(f'ws = {ws}')
print(f'history = {hist}')
```

```
ws = [-0.10627717  0.21898256 -0.05440505]
history = [0.27642845 0.27509438 0.27364767 0.27224037 0.2708757  0.26955672
 0.26828644 0.26706759 0.26590246 0.26479284 0.26373994 0.26274438
 0.26180622 0.26092497 0.26009971 0.25932905 0.2586113  0.25794448
 0.25732638 0.25675465 0.25622684 0.25574043 0.2552929  0.25488176
 0.25450454 0.25415886 0.25384243 0.25355306 0.25328865 0.25304723
 0.25282695 0.25262608 0.25244299 0.25227618 0.25212424 0.2519859
 0.25185995 0.25174531 0.25164096 0.25154599 0.25145955 0.25138086
 0.25130922 0.251244   0.25118459 0.25113048 0.25108116 0.25103622
 0.25099523 0.25095784 0.25092371]
```

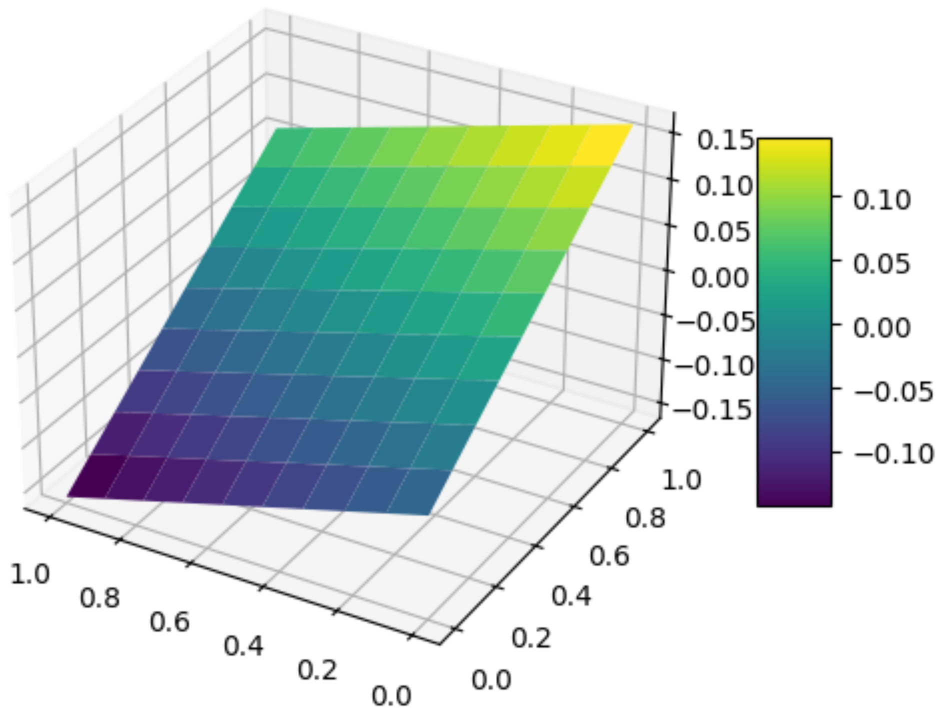
In [ ]:
```
plt.plot(range(len(hist)), hist.tolist())
plt.show()
```



In [ ]:
```
my_xor2 = lambda x1, x2: round(ws[0] * x1 + ws[1] * x2 + ws[2])
for i in range(len(X)):
    print(f'my_xor2({X[i,0]},{X[i,1]})={my_xor2(X[i,0], X[i,1])}')
print(f'acc = {acc(X, Y, my_xor2)}')
```

```
my_xor2(0,0)=0
my_xor2(1,0)=0
my_xor2(0,1)=0
my_xor2(1,1)=0
(1,0)=0, but should be 1
(0,1)=0, but should be 1
acc = 0.5
```

In [ ]:
```
f = lambda x1, x2: ws[0] * x1 + ws[1] * x2 + ws[2]
A, B = np.meshgrid(np.linspace(0,1,10), np.linspace(0,1,10))

plot3d(f, A, B, True)
plt.show()
```



Not much changes as the co-domain of m_1 is just in the domain of the sigmoid activation function s where it is almost linear.

Model 3: one hidden layer with linear and one output layer with sigmoid activation

The third model adds a hidden layer of 32 neurons with linear activation function. Each of these neurons uses the three parameters w_1, w_2, b. As before, all parameters are captured in the parameter array `ws` . The parameters for each neuron are at different positions in this array. Let's define the model m_3 with hidden and output layers and the corresponding loss function MSE.

In [ ]:
```
def hidden_layer(ws, x, n, act_fuct=lambda x:x):
    vectorized_act_func = np.vectorize(act_fuct, otypes=[float])
    # The upper line in crucial. It took a lot of time to figure out that the vecto
    y = np.zeros((n))
    for i in range(n):
        i1 = i*3
        i2 = i*3+1
        i3 = i*3+2
        y[i] = ws[i1] * x[0] + ws[i2] * x[1] + ws[i3]
    return vectorized_act_func(y)
```

In [ ]:
```
n=32
m_out = lambda ws, x: np.dot(hidden_layer(ws, x, n), ws[n*3:n*3+n])+ws[-1]
m3 = lambda ws,x: sigmoid(m_out(ws,x))
mse3 = lambda ws: mse(ws, m3, X, Y)
```

We initialize the weight vector:

In the original notebook, the `output` *variable was set to 1 each time. This is not meant by the Golrot initialization.* $W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}\right]$ *(see https://www.overleaf.com/learn/latex/Brackets_and_Parentheses) defines the value of the variable* `output` *as the number of neurons of the following layer and not always with the number of output neurons.*

In [ ]:
```
ws0 = np.zeros((n*3+n+1))

input = 2
output = n # here I set it to n instead of 1 (see comment above)
random.seed(1) # for reproducibility
limit = math.sqrt(6 / (input + output))

for i in range(n):
    i1 = i*3
    i2 = i*3+1
    i3 = i*3+2
    ws0[i1] = random.uniform(-limit, limit)
    ws0[i2] = random.uniform(-limit, limit)
    ws0[i3] = 0

input = n
output = 1
limit = math.sqrt(6 / (input + output))

for i in range(n*3, len(ws0)-1):
    ws0[i] = random.uniform(-limit, limit)

ws0
```

```
Out[ ]: array([-0.30719548,  0.29190273,  0.        ,  0.22161501, -0.20578318,
         0.        , -0.00383529, -0.04243599,  0.        ,  0.12736357,
         0.24257614,  0.        , -0.3412262 , -0.39626738,  0.        ,
         0.28209911, -0.05648696,  0.        ,  0.22035935, -0.41831459,
         0.        , -0.04588393,  0.18613086,  0.        , -0.22788532,
         0.37410221,  0.        ,  0.33726652, -0.3943833 ,  0.        ,
        -0.39870523,  0.03479344,  0.        ,  0.3689591 , -0.0998084 ,
         0.        , -0.23810413, -0.06543516,  0.        , -0.39568488,
        -0.23382577,  0.        , -0.05218486, -0.00351842,  0.        ,
        -0.22425392, -0.22611733,  0.        , -0.23627119, -0.03393988,
         0.        , -0.17661877, -0.40202906,  0.        ,  0.28362223,
         0.04743112,  0.        ,  0.11955118, -0.26389152,  0.        ,
         0.41381924,  0.30241557,  0.        , -0.31851614, -0.14056416,
         0.        ,  0.18608412,  0.17743658,  0.        ,  0.36668344,
        -0.06544321,  0.        ,  0.27728544,  0.1430853 ,  0.        ,
        -0.16520349,  0.07358243,  0.        ,  0.32134664,  0.29086401,
         0.        ,  0.0044393 ,  0.07477685,  0.        , -0.39107653,
        -0.21614165,  0.        ,  0.24986955, -0.07199064,  0.        ,
        -0.27472873,  0.04099916,  0.        ,  0.17058836,  0.14659742,
         0.        , -0.10685362, -0.0520537 ,  0.00718613,  0.23745666,
         0.01785634, -0.09103236, -0.0087894 , -0.40117982, -0.38931535,
         0.17344483,  0.41206387,  0.07946735, -0.09073849, -0.28112715,
         0.00190905,  0.41111634,  0.23070291,  0.03378587,  0.30725616,
        -0.22840097,  0.01174451,  0.38586549,  0.06634364, -0.03485258,
        -0.19675912,  0.04093139,  0.38983007, -0.42153267,  0.241902  ,
         0.2733113 ,  0.32933505,  0.205102  ,  0.        ])
```

We are ready to compute the initial loss:

```
In [ ]: mse3(ws0)
```

```
Out[ ]: 0.251957493796363
```

Even for model the m_3 the gradient of $MSE(\mathbf{ws}_o, \mathbf{ws}_h)$ for any weight $\mathbf{ws}_o, \mathbf{ws}_h$ of the output and hidden layers, resp., is almost identical as before. We just plug in m_3 instead of m_2 and m_3' instead of m_2':

$$\nabla MSE(\mathbf{ws}_o, \mathbf{ws}_h) = -\frac{1}{2}\left[\sum_{i=1}^{4}(y_i - m_3(\mathbf{ws}_o, \mathbf{ws}_h)\frac{\partial m_3(\mathbf{ws}_o, \mathbf{ws}_h, x_i)}{\partial \mathbf{ws}_o, \mathbf{ws}_h}\right]^T$$

$$\frac{\partial m_3(\mathbf{ws}_o, \mathbf{ws}_h, x_i)}{\partial(\mathbf{ws}_o, \mathbf{ws}_h)} = \frac{\partial s(m_o(\vec{ws}_o, \vec{ws}_h, x_i))}{\partial(\mathbf{ws}_o, \mathbf{ws}_h)}$$

$$= s(m_o(\mathbf{ws}_o, \mathbf{ws}_h, x_i))(1 - s(m_o(\mathbf{ws}_o, \mathbf{ws}_h, x_i))\frac{\partial m_o(\mathbf{ws}_o, \mathbf{ws}_h, x_i)}{\partial(\mathbf{ws}_o, \mathbf{ws}_h)}$$

$$\frac{\partial m_o(\mathbf{ws}_o, \mathbf{ws}_h, x_i)}{\partial(\mathbf{ws}_o, \mathbf{ws}_h)} = \frac{\partial \sum_{k=1}^{n}(m_1(\mathbf{ws}_{h,3(k-1)+1:2}, \mathbf{ws}_{h,3(k-1)+3}, x_i)\mathbf{ws}_{o,k}) + \mathbf{ws}_{o,n+1}}{\partial(\mathbf{ws}_o, \mathbf{ws}_h)}$$

The term $\mathbf{ws}_{o,k}$ selects the k weights of the output layer and the term $\mathbf{ws}_{o,k+1}$ selects the bias b from the vector $\mathbf{ws}_o$. The term $\mathbf{ws}_{h,3(k-1)+1:2}$ selects the two weights $\mathbf{w}_{k,1}$ and $\mathbf{w}_{k,2}$ of the k-th neuron in the hidden layer, and the term $\mathbf{ws}_{h,3(k-1)+3}$ selects the k-th bias $\mathbf{b}_k$ from the vector $\mathbf{ws}_h$.

The gradients wrt. the parameters of the hidden layer are

$$\frac{\partial m_o(\mathbf{ws}_o, \mathbf{ws}_h, x_i)}{\partial \mathbf{ws}_{h,3(k-1)+j}} = \begin{cases} \frac{\partial m_1(\mathbf{w}_k, b_k, x_i)}{\partial \mathbf{w}_{k,j}} \mathbf{ws}_{o,k}, & \text{if } j = 1, 2 \\ \frac{\partial m_1(\mathbf{w}_k, b_k, x_i)}{\partial b_k} \mathbf{ws}_{o,k}, & \text{if } j = 3 \end{cases}$$

Since the cell function of the hidden layer is m_1, we can plug in the first derivative of m_1 wrt. w_1, w_2, and b, resp., as computed before.

$$\frac{\partial m_1(\mathbf{w}, b, x_i)}{\partial w_j} = \frac{\partial(\sum_{j=1}^{2} w_j x_{ij}) + b}{\partial w_j} = x_{ij}$$
$$\frac{\partial m_1(\mathbf{w}, b, x_i)}{\partial b} = \frac{\partial(\sum_{j=1}^{2} w_j x_{ij}) + b}{\partial b} = 1$$

The gradients wrt. the weights and the bias, resp., of the output layer are

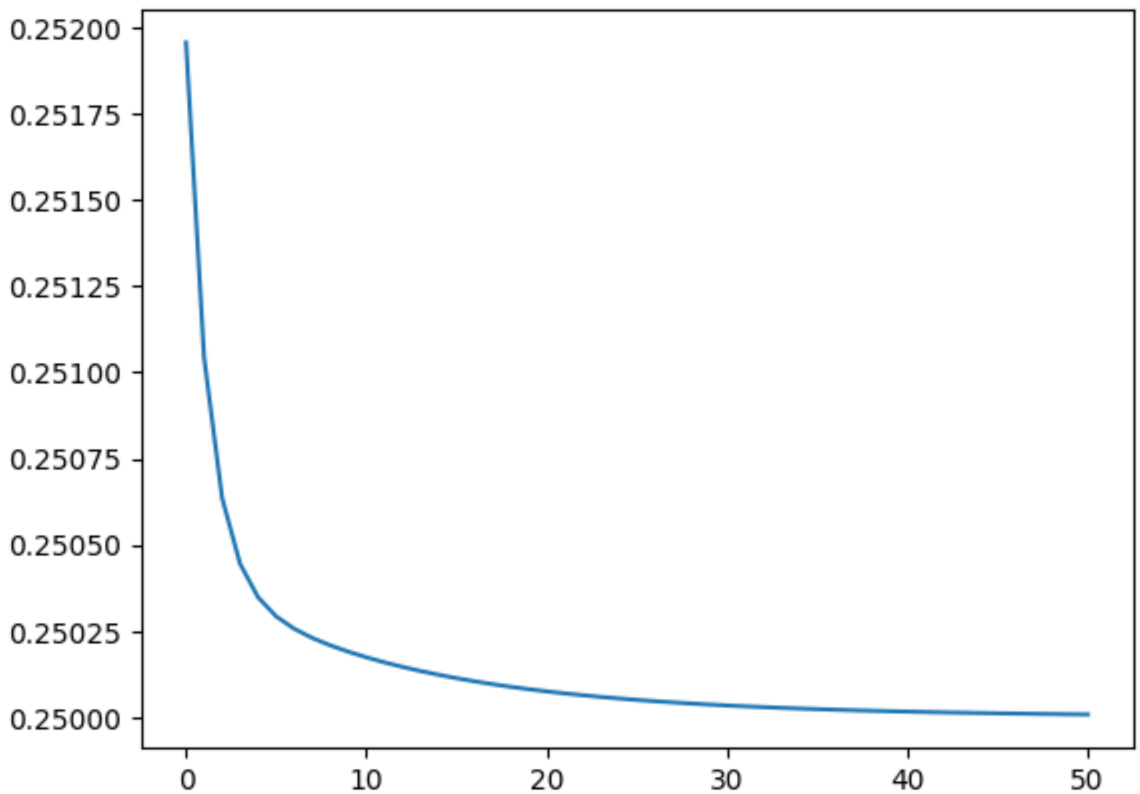
$$\frac{\partial m_o(\mathbf{ws}_o, \mathbf{ws}_h, x_i)}{\partial \mathbf{ws}_{o,k}} = m_1(\vec{w}_k, b_k, x_i)$$
$$\frac{\partial m_o(\mathbf{ws}_o, \mathbf{ws}_h, x_i)}{\partial \mathbf{ws}_{o,n+1}} = 1$$

In [ ]:
```
# Here I need to define some classes to store the iteration variable for the lazy e

class HiddenWeightGradient():

    def __init__(self, n, k, x_idx):
        self.n = n
        self.k = k
        self.x_idx = x_idx

    def apply(self, ws, x):
        return sigmoid(m_out(ws, x))*(1-sigmoid(m_out(ws, x)))*x[self.x_idx]*ws[3*s

class HiddenBiasGradient():

    def __init__(self, n, k):
        self.n = n
        self.k = k

    def apply(self, ws, x):
        return sigmoid(m_out(ws, x))*(1-sigmoid(m_out(ws, x)))*ws[3*self.n+self.k]

class Model1Gradient():

    def __init__(self, w1_idx, w2_idx, bias_idx) -> None:
        self.w1 = w1_idx
        self.w2 = w2_idx
        self.bias = bias_idx

    def apply(self, ws, x):
        return ws[self.w1]*x[0] + ws[self.w2] * x[1] + ws[self.bias]

class OutputWeightGradient():

    def __init__(self, mod1_grad):
        self.mod1_grad = mod1_grad

    def apply(self, ws, x):
        return sigmoid(m_out(ws, x))*(1-sigmoid(m_out(ws, x)))*self.mod1_grad(ws,x)



gradients3 = []
model1 = []

for k in range(n):
    gradients3.append(HiddenWeightGradient(n,k,0).apply)
    gradients3.append(HiddenWeightGradient(n,k,1).apply)
    gradients3.append(HiddenBiasGradient(n,k).apply)

    i1 = k*3
    i2 = k*3+1
    i3 = k*3+2
    model1.append(Model1Gradient(i1, i2, i3).apply)

assert len(model1) == n
```

```python
k=0
for i in range(3*n, len(ws0)-1):
    mod = model1[k]
    gradients3.append(OutputWeightGradient(mod).apply)
    k += 1

gradients3.append(lambda ws,x : sigmoid(m_out(ws, x))*(1-sigmoid(m_out(ws, x)))) #

assert len(gradients3) == 3*n+n+1
```

In [ ]:
```python
grad_loss = lambda ws: grad_mse(ws, m3, gradients3, X, Y)
K = 50
learning_eps = 0.5
ws, hist = grad_desc_mse(K, ws0, learning_eps, mse3, grad_loss, False)
```

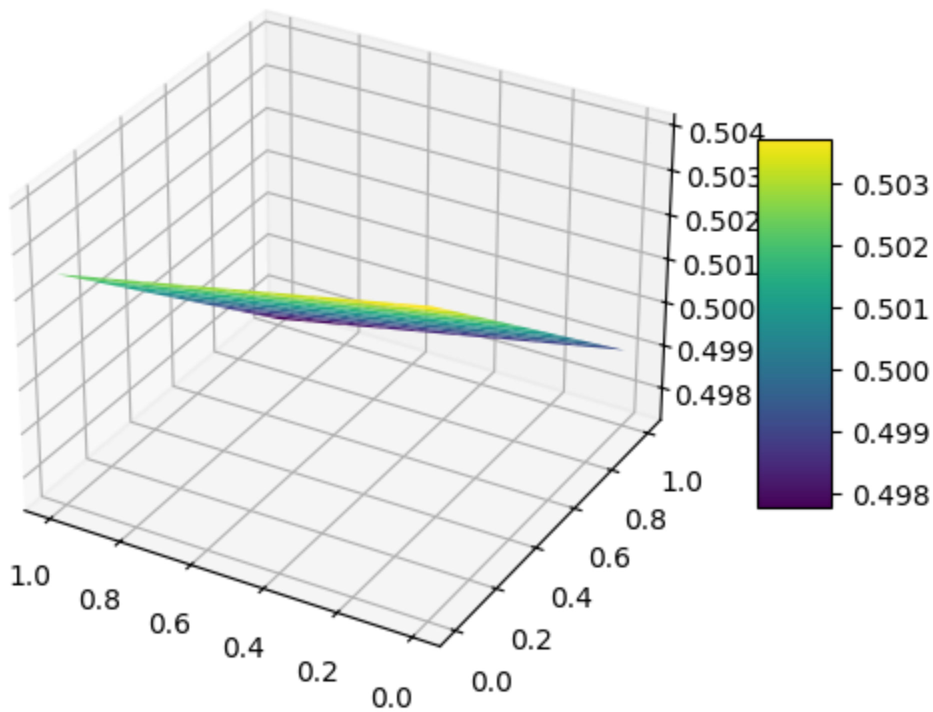
In [ ]:
```python
plt.plot(range(len(hist)), hist.tolist())
plt.show()
```



In [ ]:
```python
my_xor3 = lambda x1, x2: round(m3(ws, [x1, x2]))
for i in range(len(X)):
    print(f'my_xor3({X[i,0]},{X[i,1]})={my_xor3(X[i,0], X[i,1])}')
print(f'acc = {acc(X, Y, my_xor3)}')
```

```
my_xor3(0,0)=1
my_xor3(1,0)=1
my_xor3(0,1)=0
my_xor3(1,1)=0
(0,0)=1, but should be 0
(0,1)=0, but should be 1
acc = 0.5
```

In [ ]:
```
f = lambda x1, x2: m3(ws, [x1, x2])
A, B = np.meshgrid(np.linspace(0,1,10), np.linspace(0,1,10))

plot3d(f, A, B, True)
plt.show()
```



Model 4: one hidden layer with relu and one output layer with sigmoid activation

The final model uses the Rectified Linear Unit (ReLU) activation function $r(x) = \max(0, x)$ in the hidden layer. Hence, the hidden layer neurons compute $\max(0, w_1x_1 + w_2x_2 + b) = \max(0, m_1(\vec{w}, b, x))$. We can easily change the hidden layer function accordingly.

In [ ]:
```
hidden_layer_4 = lambda ws, x, n: hidden_layer(ws, x, n, lambda x:max(0,x))

m_out4 = lambda ws, x: np.dot(hidden_layer_4(ws, x, n), ws[n*3:n*3+n]) + ws[-1]
m4 = lambda ws, x: sigmoid(m_out4(ws, x))
mse4 = lambda ws: mse(ws, m4, X, Y)
```

The network structure does not change, hence, the initialization remains the same as well. We can directly compute the initial loss:

In [ ]:
```
mse4(ws0)
```

Out[ ]:
```
0.2575479161356454
```

No surprise it is the same as for model m_3 as all weights and all input is positive.

As for the model m_4 itself, we need to subsitute $m_1(\mathbf{w}, b, x)$ with $r(\mathbf{w}, b, x) = \max(0, m_1(\mathbf{w}, b, x))$ and $m_1'(\mathbf{w}, b, x)$ with $r'(m_1(\mathbf{w}, b, x))m_1'(\mathbf{w}, b, x)$.

$$r'(z) = \begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{if } z > 0 \end{cases}$$

Note that $r'(z)$ is undefined in $z = 0$, since its left and right derivative are not equal for $z = 0$. In practice , we pick $r'(0) = 0$ as Tensorflow does. Consequently:

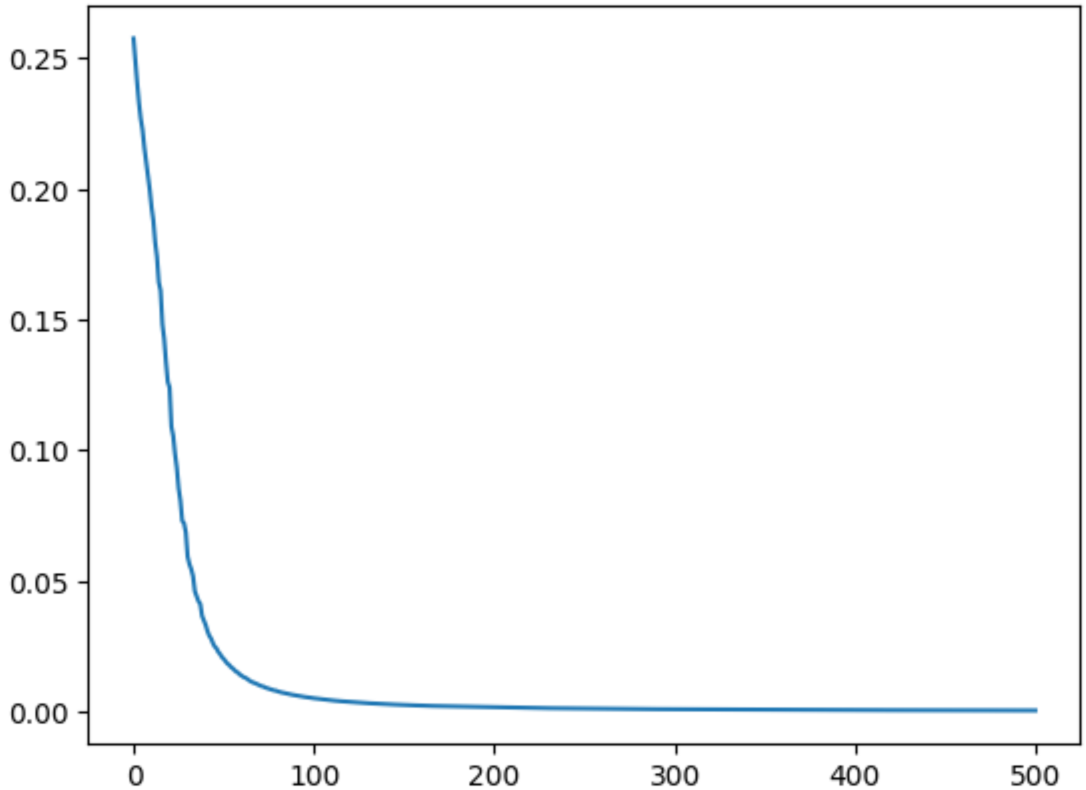
$$r'(m_1(\mathbf{w}, b, x))m_1'(\mathbf{w}, b, x) = \begin{cases} 0, & \text{if } m_1(\mathbf{w}, b, x) \leq 0 \\ m_1'(\mathbf{w}, b, x), & \text{if } m_1(\mathbf{w}, b, x) > 0 \end{cases}$$

In [ ]:
```
def ite(cond, tc, fc):
    return tc if cond else fc
```

In [ ]:
```
class HiddenGradient4():

    def __init__(self, n, k, x_idx, model1_grad):
        self.n = n
        self.k = k
        self.x_idx = x_idx
        self.mod1_grad = model1_grad

    def apply(self, ws, x):
        return sigmoid(m_out4(ws, x))*(1-sigmoid(m_out4(ws, x)))*ite(self.mod1_grad

class Model1Gradient4():

    def __init__(self, w1_idx, w2_idx, bias_idx) -> None:
        self.w1 = w1_idx
        self.w2 = w2_idx
        self.bias = bias_idx

    def apply(self, ws, x):
        return max(0,ws[self.w1]*x[0] + ws[self.w2] * x[1] + ws[self.bias])

class OutputWeightGradient4():

    def __init__(self, mod1_grad):
        self.mod1_grad = mod1_grad

    def apply(self, ws, x):
        return sigmoid(m_out4(ws, x))*(1-sigmoid(m_out4(ws, x)))*max(0,self.mod1_gr



gradients4 = []
model1 = []

for k in range(n):
    i1 = k*3
    i2 = k*3+1
    i3 = k*3+2
    mod1_grad = Model1Gradient4(i1, i2, i3)
    model1.append(mod1_grad.apply)
    gradients4.append(HiddenGradient4(n,k,0, mod1_grad.apply).apply)
    gradients4.append(HiddenGradient4(n,k,1, mod1_grad.apply).apply)
    gradients4.append(HiddenGradient4(n,k,2, mod1_grad.apply).apply)


assert len(model1) == n

k=0
for i in range(3*n, len(ws0)-1):
    mod = model1[k]
    gradients4.append(OutputWeightGradient4(mod).apply)
    k += 1

gradients4.append(lambda ws,x : sigmoid(m_out4(ws, x))*(1-sigmoid(m_out4(ws, x))))

assert len(gradients4) == 3*n+n+1
```

In [ ]:
```
grad_loss = lambda ws:grad_mse(ws, m4, gradients4, X, Y)
K = 500
learning_eps = 1.5
ws, hist = grad_desc_mse(K, ws0, learning_eps, mse4, grad_loss, False)
```

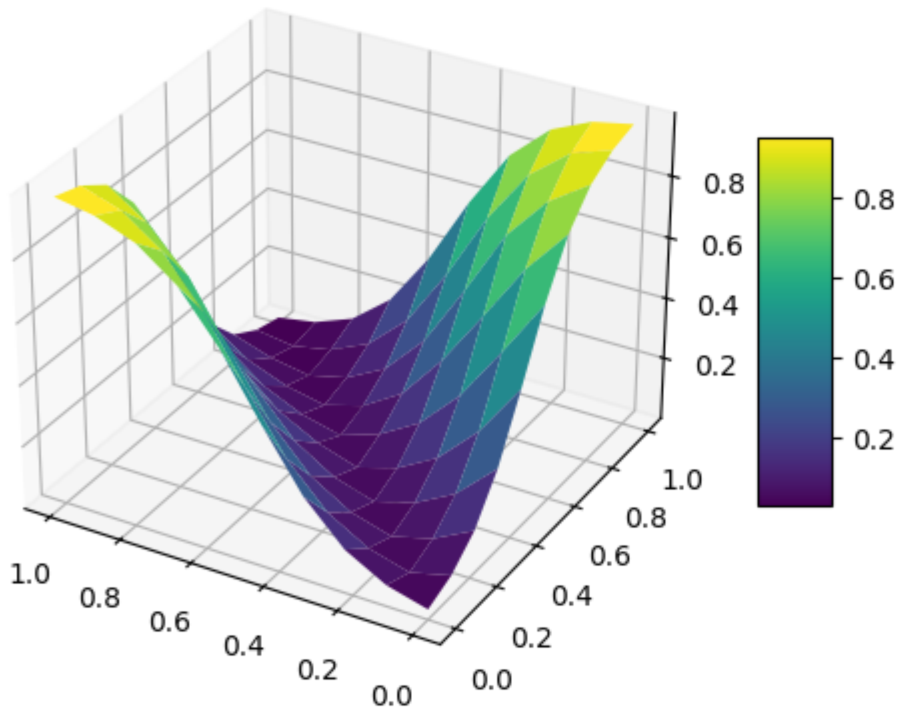
In [ ]:
```
plt.plot(range(len(hist)), hist.tolist())
plt.show()
```



In [ ]:
```
my_xor4 = lambda x1, x2: round(m4(ws, [x1, x2]))
for i in range(len(X)):
    print(f'my_xor4({X[i,0]},{X[i,1]})={my_xor4(X[i,0], X[i,1])}')
print(f'acc = {acc(X, Y, my_xor4)}')
```

```
my_xor4(0,0)=0
my_xor4(1,0)=1
my_xor4(0,1)=1
my_xor4(1,1)=0
acc = 1.0
```

In [ ]:
```
f = lambda x1, x2: m4(ws, [x1, x2])
A, B = np.meshgrid(np.linspace(0,1,10), np.linspace(0,1,10))
plot3d(f, A, B, True)
plt.show()
```



Comparison with Model 4 learned with Tensorflow

Weights of the hidden layer (manually copied):

In [ ]:
```
w11 = np.reshape([ 0.41652033,  0.7749157 ,  0.83778775,  0.64122725,  0.3972386 ,
         -0.13516521, -0.60432655, -0.20290181, -0.7232665 , -0.35317   ,
         -0.8889785 , -0.2863993 ,  0.54030126,  0.37175888, -0.82367843,
         -0.33497408, -0.8567182 ,  0.5161414 , -0.25462216,  0.59564906,
          0.80922335,  0.00252599,  0.3720131 , -0.1701207 ,  0.34805146,
         -0.38333094, -0.17244412,  0.7132528 , -0.14648864, -0.17876345,
         -0.6636416, -0.39717668], (32,1))

w12 = np.reshape([0.41662493, -0.7749062 , -0.8376726 , -0.64101034,  0.40976927,
         -0.09223783,  0.60428804, -0.09886253,  0.7417394 , -0.08979583,
          1.1194593 ,  0.13293397,  0.5400226 ,  0.37171456,  0.8246372 ,
         -0.43565178,  0.85677105,  0.51622313, -0.41895077, -0.5953253 ,
         -0.8091849 , -0.3766787 ,  0.28819063, -0.38122708,  0.34833792,
          0.15245543, -0.25060123, -0.7132655 , -0.30406952, -0.3443008,
          0.66343206, -0.03848529], (32,1))

b1 = np.reshape([-4.16518986e-01, -4.11640503e-05, -1.45169579e-05, -3.96516771e-06
        -1.09622735e-04,  0.00000000e+00,  2.90098578e-05,  0.00000000e+00,
        -1.63983065e-03,  0.00000000e+00, -6.36966957e-04, -1.47912502e-01,
        -5.39916575e-01, -3.71752203e-01, -5.50682598e-04,  1.15736163e+00,
        -6.27825721e-05, -5.16140461e-01,  0.00000000e+00, -1.62737619e-04,
        -8.02413124e-05, -2.25486048e-02,  1.79730232e-05,  0.00000000e+00,
        -3.48306417e-01, -1.66664287e-01,  0.00000000e+00, -3.03934885e-05,
         0.00000000e+00,  0.00000000e+00,  6.44910688e-05,  0.00000000e+00], (32,1)

ws1 = np.hstack((w11,w12,b1))
ws1 = ws1.reshape(-1)
```

Weights of the output layer manually copied:

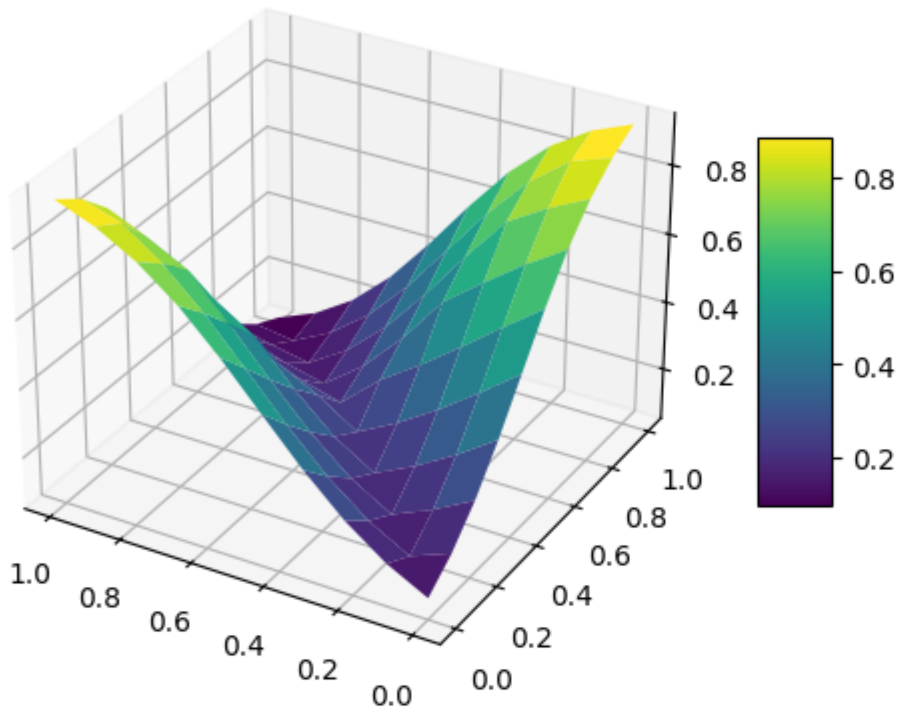
In [ ]:
```
ws2 = [-0.8178053 ,
1.0259871 ,
0.7672013 ,
0.9983522 ,
0.3300904 ,
0.1318875 ,
0.9221373 ,
0.20387506,
1.0057546 ,
0.2067241 ,
0.7479725 ,
-0.25578788,
-0.83140665,
-0.952559  ,
0.946334  ,
-1.0112743 ,
0.5946371 ,
-1.0230583 ,
-0.04098088,
0.9588608 ,
0.7448052 ,
-0.10435582,
0.25728533,
0.30141222,
-0.8235109 ,
-0.21171094,
-0.28320622,
1.0405817 ,
0.40911728,
0.37331975,
0.87603277,
0.19098908,
-0.8885394]
wsc = np.array(ws1.tolist() + ws2)
```

Replace the learned weights of model m_4 with the copied weights and assess the function:

In [ ]:
```
my_xor5 = lambda x1, x2: round(m4(wsc, [x1, x2]))
for i in range(len(X)):
    print(f'my_xor5({X[i,0]},{X[i,1]})={my_xor5(X[i,0], X[i,1])}')
print(f'acc = {acc(X, Y, my_xor5)}')
```

```
my_xor5(0,0)=0
my_xor5(1,0)=1
my_xor5(0,1)=1
my_xor5(1,1)=0
acc = 1.0
```

In [ ]:
```
f = lambda x1, x2: m4(wsc, [x1, x2])
A, B = np.meshgrid(np.linspace(0,1,10), np.linspace(0,1,10))
plot3d(f, A, B, True)
plt.show()
```


0.2
0.4
0.6
0.8
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
0.2
0.4
0.6
0.8