CmpE597 Homework 1 Part 2

March 11, 2018

Mine Melodi Çalkan - 2015705009

Multilayer Perceptron Implementation

In this simulation cross entropy error is used for the cost function. Training is done using stochastic gradient descent with momentum.

The cross entropy error

$E = -\sum_{i=1}^{n}(t_i log(y_i) + (1-t_i)log(1-y_i)) \quad (1)$

where t is the target vector, y is the output vector.

The output prediction is always between zero and one. Training corresponds to minimizing the negative log-likelihood of the data.

Outputs are computed by applying the sigmoid function to the weighted sums of the hidden layer activations.

$y_i = \frac{1}{1+e^{-z_i}}$ where $z_i = \sum_{j=1} h_j w_{ji}$

The derivative of the error with respect to each weight connecting the hidden units to the output units using the chain rule:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial z_i}\frac{\partial z_i}{\partial w_{ji}}$$

- $\frac{\partial E}{\partial y_i} = \frac{-t_i}{y_i} + \frac{1-t_i}{1-y_i} = \frac{y_i - t_i}{y_i(1-y_i)}$
- $\frac{\partial y_i}{\partial z_i} = y_i(1-y_i)$
- $\frac{\partial z_i}{\partial w_{ji}} = h_j$

The gradients of the error with respect to the output weights:

$$\frac{\partial E}{\partial w_{ji}} = (y_i - t_i)h_j$$

And $\frac{\partial E}{\partial z_i} = \frac{y_i - t_i}{y_i(1-y_i)} y_i(1-y_i) = (y_i - t_i)$

Backpropagation algorithm for gradients with respect to the hidden layer weights

$\frac{\partial E}{\partial w_{kj}^l} = \frac{\partial E}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{kj}^l}$

z_j^1 is the weighted input sum at hidden unit j, and $h_j = \frac{1}{1+e^{-z_j^1}}$ is the activation at unit j.

- $\frac{\partial E}{\partial z_j^l} = \sum_{i=1} \frac{\partial E}{\partial z_i}\frac{\partial z_i}{\partial h_j}\frac{\partial h_j}{\partial z_j^l} = \sum_{i=1}(y_i - t_i)(w_{ji})(h_j(1-h_j))$

$\frac{\partial E}{\partial h_j} = \sum_{i=1} \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial z_i}\frac{\partial z_i}{\partial x_j} = \sum_{i=1} \frac{\partial E}{\partial y_i} y_i(1-y_i)w_{ji}$
Then \$ w^1_{kj}\$
connecting input unit k to hidden unit j has gradient
$\frac{\partial E}{\partial w^l_{kj}} = \frac{\partial E}{\partial z^l_j}\frac{\partial z^l_j}{\partial w^l_{kj}} = \sum_{i=1}(y_i - t_i)w_{ji}(h_j(1-h_j))x_k$

Momentum

SGD with momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations.It does this by adding a fraction γ of the update vector of the past time step to the current update vector:

$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$
$\theta = \theta - v_t$

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
X = np.array([ [0,0],[0,1],[1,0],[1,1]])
y = np.array([[0,1,1,0]]).T
def initialize_network_m(number_hidden_units=2):
    np.random.seed(0)
    w_hidden = 2*np.random.rand(2,number_hidden_units) - 1
    w_output = 2*np.random.rand(number_hidden_units,1) - 1
    velocity_hidden = np.zeros((2,number_hidden_units))
    velocity_output = np.zeros((number_hidden_units,1))
    bias_1=np.zeros(number_hidden_units)
    bias_2=np.zeros(1)
    return w_hidden,w_output,velocity_hidden,velocity_output,bias_1,bias_2

def f(x):
    res = 1 / (1+np.exp(-x))
    return res

#stochastic gradient descent with momentum
def backprop_sgd_m(epochs,number_hidden_units,lr_init=0.1):
    weight_hidden,weight_output,velocity_hidden,velocity_output,b1,b2=initialize_netwo
    X = np.array([ [0,0],[0,1],[1,0],[1,1]])
    y = np.array([[0,1,1,0]]).T
    alpha=lr_init
    w_hidden_list=[]
    b1_list=[]
    for j in range(epochs):

        b11=b1.copy()
        w11=weight_hidden.copy()
        for i in range(4):
            hidden_layer =f(np.dot(X[i], weight_hidden)+b1)
            output_layer = f(np.dot(hidden_layer, weight_output)+b2)
            output_layer_delta = (output_layer - y[i][0])

            deriv_hidden =(hidden_layer * (1-hidden_layer))
```

```
                hidden_layer_delta = output_layer_delta.dot(weight_output.T) *deriv_hidden
                gradient_output = [[alpha*x*output_layer_delta[0]] for x in hidden_layer]
                gradient_b2 = np.sum(output_layer_delta, axis=0)
                #adjust dimensions and shapes
                dim_adjusted_x=np.array([[x] for x in X[i]])
                dim_adjusted_hidden=np.array([[x] for x in hidden_layer_delta]).T
                gradient_hidden =  alpha*dim_adjusted_x.dot(dim_adjusted_hidden)
                gradient_b1 = np.sum(hidden_layer_delta, axis=0)

                velocity_output = 0.9 * velocity_output + gradient_output
                velocity_hidden = 0.9 * velocity_hidden + gradient_hidden

                weight_output -= velocity_output
                weight_hidden -= velocity_hidden

                b1 -= alpha * gradient_b1
                b2 -= alpha * gradient_b2
            if j%100==0:
                w_hidden_list.append(w11)
                b1_list.append(b11)

        return weight_hidden,weight_output,b1,w_hidden_list,b1_list
```

In [2]:
```
def test_m(test_data,number_hidden_units,epochs):
    hiddenWeight,outputWeight,b,hlist,blist=backprop_sgd_m(epochs,number_hidden_units)
    _hiddenLayer=f(np.dot(test_data,hiddenWeight))
    _outputLayer=f(np.dot(_hiddenLayer,outputWeight))
    ##threshold for output probabilities
    results = [1 if i>=0.5 else 0 for i in _outputLayer]
    return results,_outputLayer,hiddenWeight,b,hlist,blist
```

In [3]:
```
test_data=np.array([[0,0],[0,1],[1,0],[1,1]])
```

In [4]:
```
result, outputlayer,hidden,bias,hl,bl=test_m(test_data,2,2000)
```

In [5]:
```
outputlayer
```

Out[5]:
```
array([[ 0.34211239],
       [ 0.55236437],
       [ 0.55233821],
       [ 0.50073344]])
```

In [6]:
```
hidden
```

Out[6]:
```
array([[-9.64301217, -4.29052066],
       [-9.64263244, -4.29001113]])
```

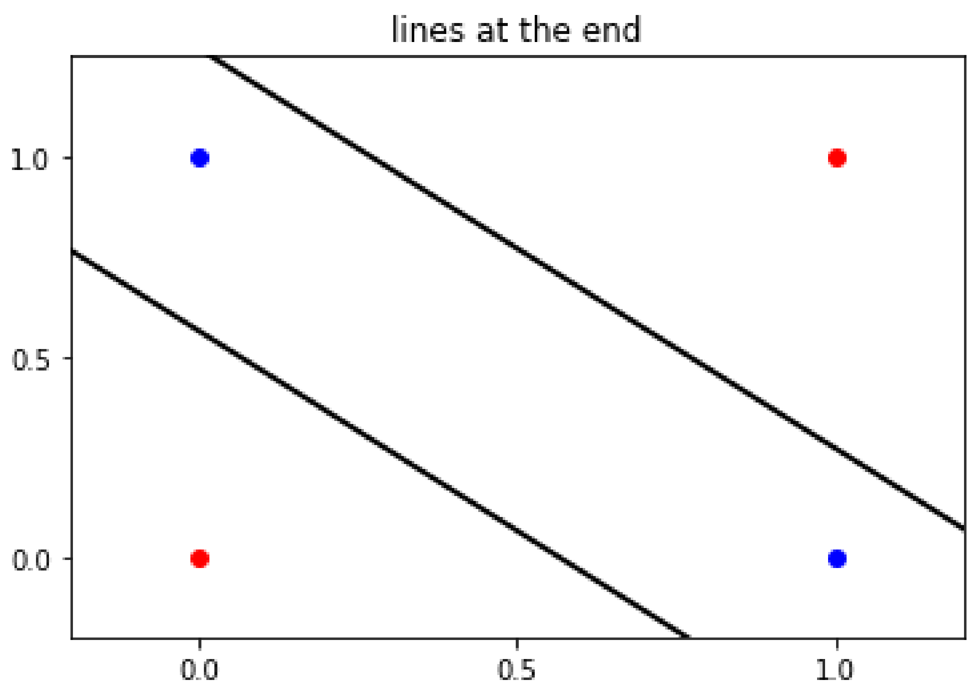
In [7]:
```
bias
```

Out[7]:
```
array([ 5.44861768,  5.44861768])
```

In [18]:
```
X=np.array([[0,0],[0,1],[1,0],[1,1]])
y=np.array([[0],[1],[1],[0]])
```

In [8]:
```
ww=hidden
bb=bias
```

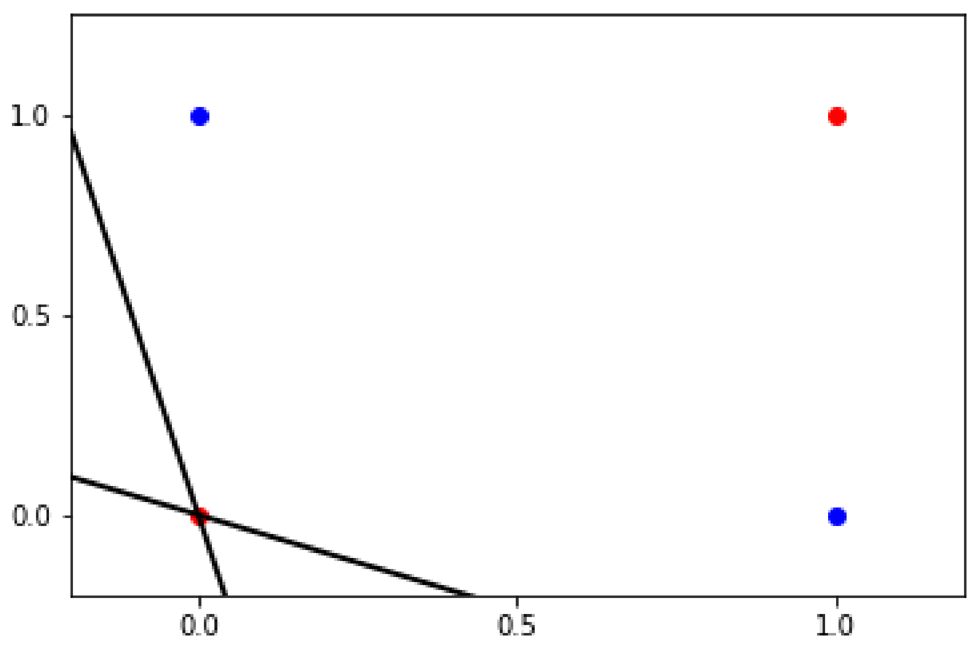
In [9]:
```
plot_x = np.array([np.min(X[:, 0] - 0.2), np.max(X[:, 1]+0.2)])
plot_y =  -1 / ww[1, 0] * (ww[0, 0] * plot_x + bb[0])
plot_y = np.reshape(plot_y, [2, -1])
plot_y = np.squeeze(plot_y)

plot_y2 = -1 / ww[1, 1] * (ww[0, 1] * plot_x + bb[1])
plot_y2 = np.reshape(plot_y2, [2, -1])
plot_y2 = np.squeeze(plot_y2)
yy=[0,1,1,0]
for i in range(len(yy)):
    if yy[i] == 1:
        plt.scatter(X[i, 0], X[i, 1],c='b')
    elif yy[i] == 0:
        plt.scatter(X[i, 0], X[i, 1], c='r')
plt.plot(plot_x, plot_y, color='k', linewidth=2)
plt.plot(plot_x, plot_y2, color='k', linewidth=2)
plt.xlim([-0.2, 1.2]); plt.ylim([-0.2, 1.25]);
plt.xticks([0.0, 0.5, 1.0]); plt.yticks([0.0, 0.5, 1.0])
plt.title('lines at the end')
plt.show()
```

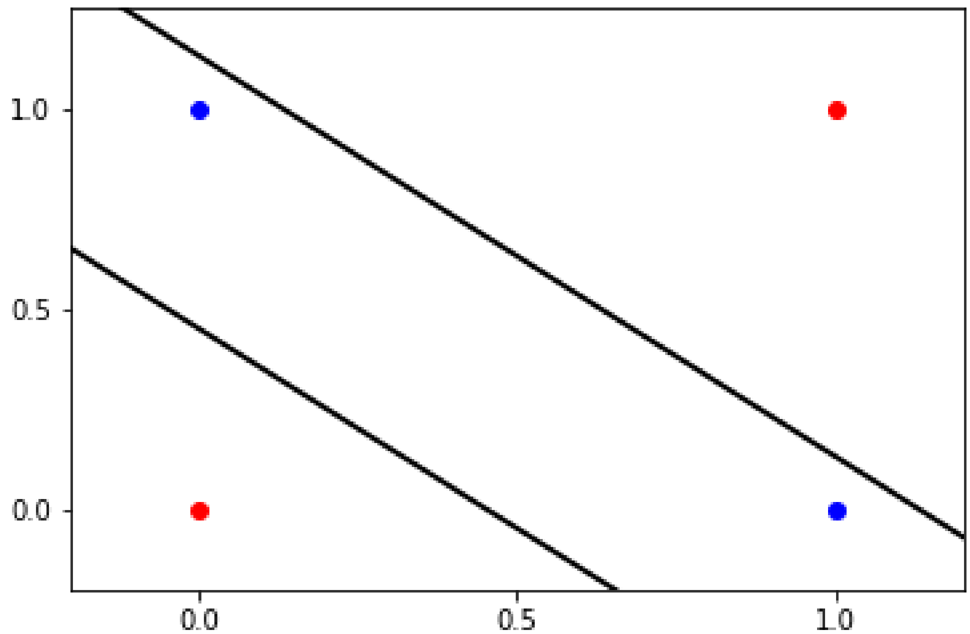


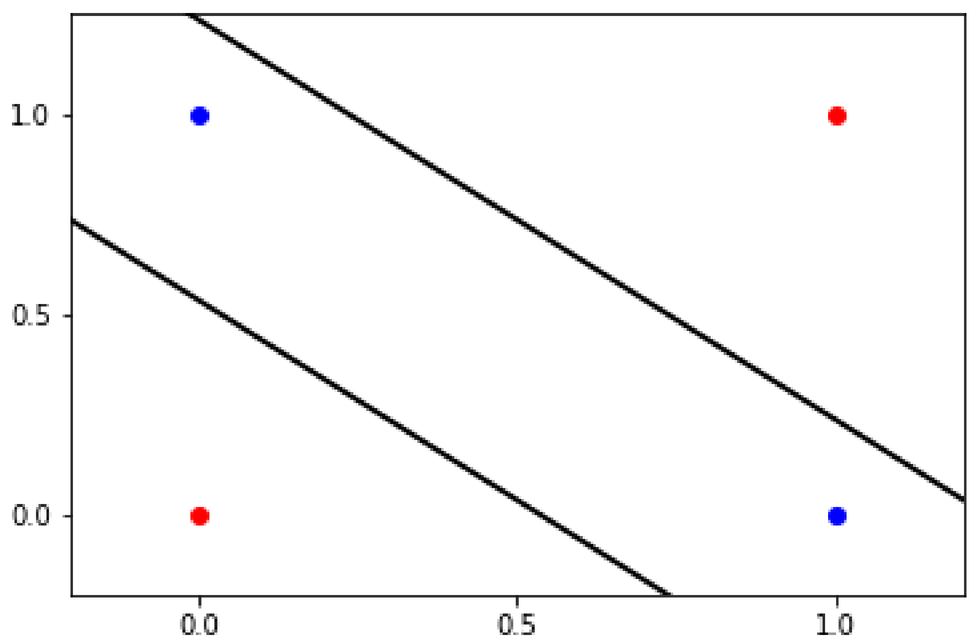
In [10]:
```python
indices=[0,5,10,15,19]
```

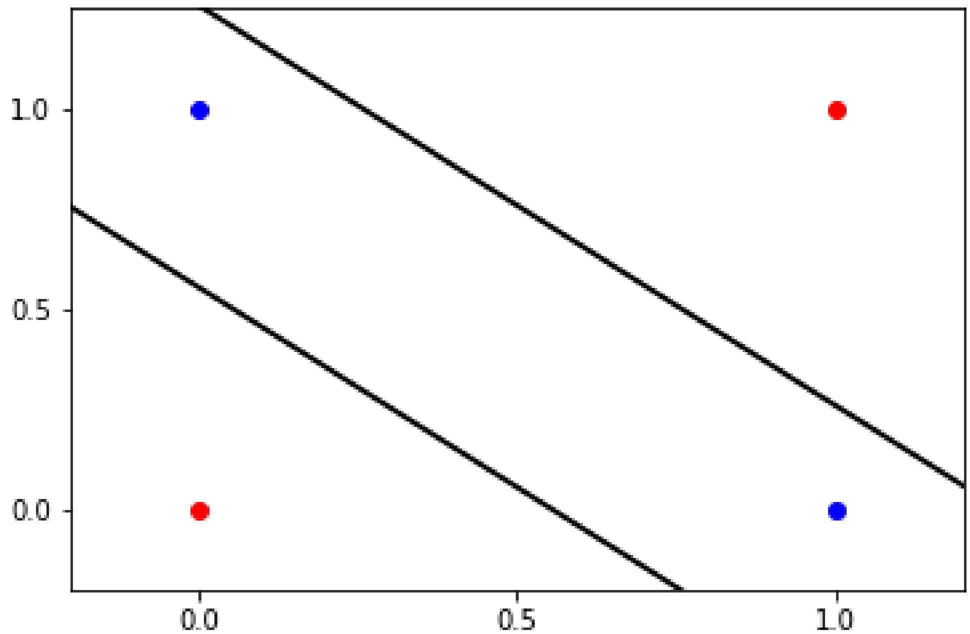
In [11]:
```python
X= np.array([[0,0],[0,1],[1,0],[1,1]])
w=hl
b=bl
```

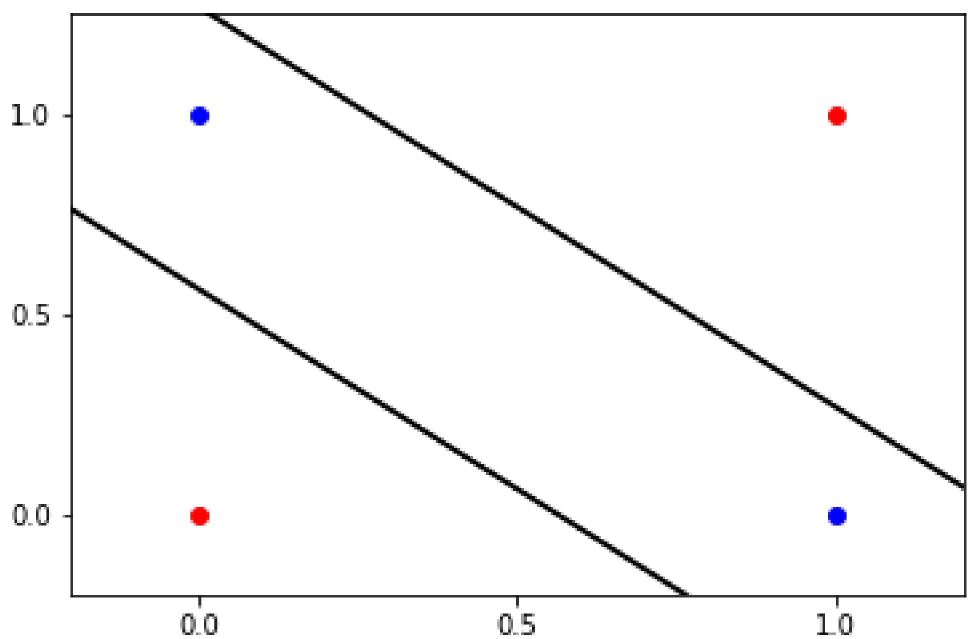
In [12]:
```python
for i in indices:
    plot_x = np.array([np.min(X[:, 0] - 0.2), np.max(X[:, 1]+0.2)])
    plot_y =  -1 / w[i][1, 0] * (w[i][0, 0] * plot_x + b[i][0])
    plot_y = np.reshape(plot_y, [2, -1])
    plot_y = np.squeeze(plot_y)

    plot_y2 = -1 / w[i][1, 1] * (w[i][0, 1] * plot_x + b[i][1])
    plot_y2 = np.reshape(plot_y2, [2, -1])
    plot_y2 = np.squeeze(plot_y2)
    yy=[0,1,1,0]
    for i in range(len(yy)):
        if yy[i] == 1:
            plt.scatter(X[i, 0], X[i, 1],c='b')
        elif yy[i] == 0:
            plt.scatter(X[i, 0], X[i, 1], c='r')
    plt.plot(plot_x, plot_y, color='k', linewidth=2)
    plt.plot(plot_x, plot_y2, color='k', linewidth=2)
    plt.xlim([-0.2, 1.2]); plt.ylim([-0.2, 1.25]);
    plt.xticks([0.0, 0.5, 1.0]); plt.yticks([0.0, 0.5, 1.0])
    plt.show()
```




1.0
0.5
0.0
0.0
0.5
1.0


1.0
0.5
0.0
0.0
0.5
1.0


1.0
0.5
0.0
0.0
0.5
1.0


1.0
0.5
0.0
0.0
0.5
1.0