# Exercise Sheet 3: Neural Networks 2

In [1]:
```
import numpy as np
import time
from matplotlib import pyplot as plt

def logistic(h,a=1):
    return 1/(1+np.exp(-a*h))
dlogistic=lambda x: logistic(x)-np.power(logistic(x),2)
signtrafunc=lambda x: (np.sign(x)+1)/2
```

In [2]:
```
import Exercise3helper36 as helper
```

## Task 1 Lateral Inhibition (4 points - programming)

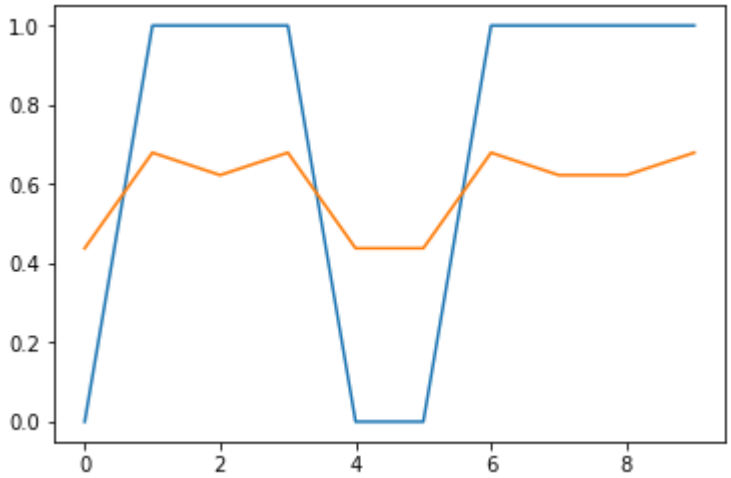
Build a network layer representing the simple lateral inhibition we have faced in lecture 2. This involves only a direct neighbour supression. It represents a not fully-connected MLP layer. The layer should be 10 neurons wide. Set the bias to $b = 0$ and the neighbour weights to $w = -0.25$. Later, repeat the simulation with $w = -0.5$.

As inputs, use a random vector of 0 and 1 (np.random.randi).

First solve the task with the logistic transfer function. Then use the sign-based transfer function and examine the results with plots.

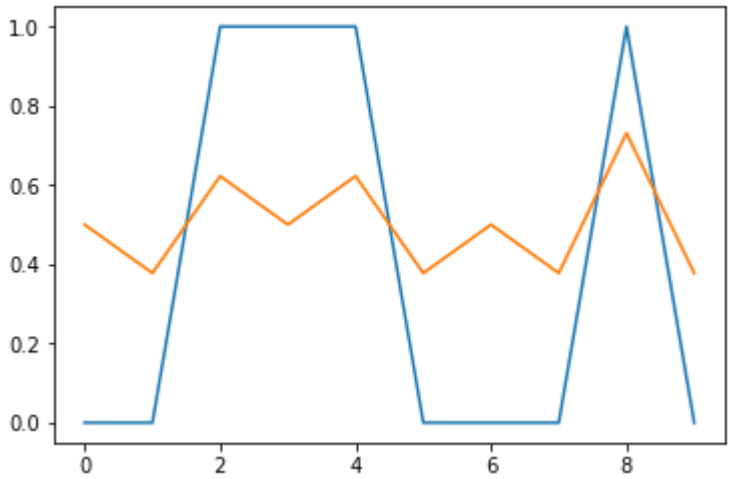
In [5]:
```
class neuron:

    def __init__(self,w,b=0,trafunc=logistic,dtrafunc=dlogi
stic):
        self.w=np.array(w)
        self.b=np.array(b)
        self.trafunc=trafunc
        self.dtrafunc=dtrafunc

    def out(self, x):
        return self.trafunc(np.dot(self.w,x)-self.b)

class LateralInhibitionLayer:

    def __init__(self, weight_inhibition, weightinit=1,b =
0,trafunc=logistic,dtrafunc=dlogistic):
        self.neurons = []
        self.neurons.append(neuron( [weightinit, weight_inh
ibition], b, trafunc=trafunc,dtrafunc= dtrafunc))
        for i in range(8):
             self.neurons.append(neuron( [weight_inhibition
, weightinit, weight_inhibition], b, trafunc=trafunc,dtrafu
nc= dtrafunc))
        self.neurons.append(neuron( [weight_inhibition, wei
ghtinit], b, trafunc=trafunc,dtrafunc= dtrafunc))

    def out(self,x):
        returnobj = np.zeros(len(self.neurons))
        for n in range(len(self.neurons)):

            #if first one\
            if n == 0:
                returnobj[n] = (self.neurons[n].out([x[n],
x[n+1]]))
            #if last one
            elif n == len(self.neurons)-1:
                returnobj[n] = (self.neurons[n].out([x[n-1]
, x[n]]))
            #if not, ask the neighbours len - 1 and len +1
            else:
                returnobj[n] = (self.neurons[n].out([x[n-1]
, x[n], x[n+1]]))

        return returnobj

#logistic function -0.25
layer_traf = LateralInhibitionLayer(-0.25)
x = np.random.randint(2,size=10)
plt.plot(x)
plt.plot(layer_traf.out(x))
```

Out[5]: [<matplotlib.lines.Line2D at 0x7f7b0c8d11d0>]



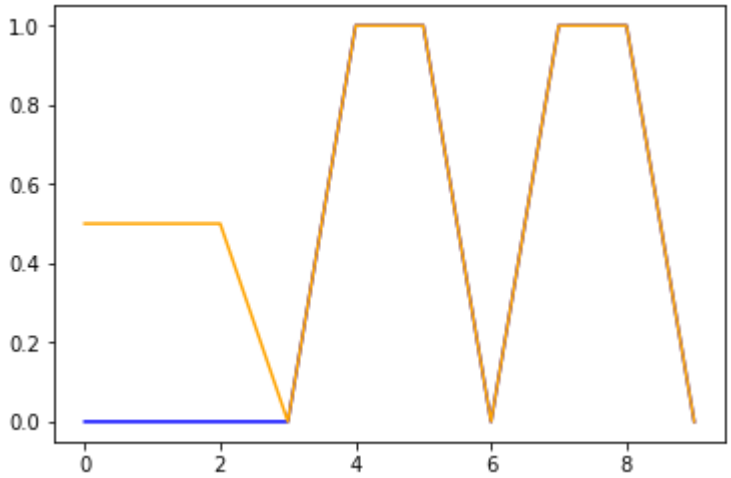

In [6]:
```
#logistic function -0.5
layer_traf = LateralInhibitionLayer(-0.5)
x = np.random.randint(2,size=10)
plt.plot(x)
plt.plot(layer_traf.out(x))
```

Out[6]: [<matplotlib.lines.Line2D at 0x7f7b0c8a3898>]



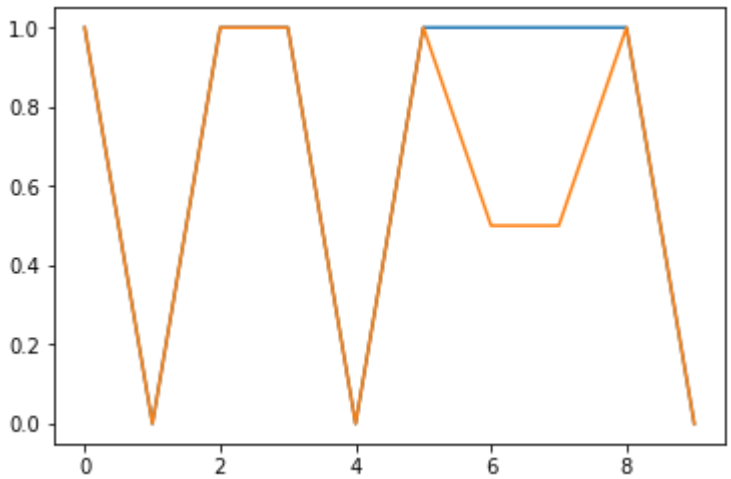
In [7]:
```
#Sign based function -0.25
layer_sign = LateralInhibitionLayer(-0.25, trafunc=signtraf
unc)
x = np.random.randint(2,size=10)
plt.plot(x,color = 'blue')
plt.plot(layer_sign.out(x),color = 'orange')
```

Out[7]: [<matplotlib.lines.Line2D at 0x7f7b0c934e80>]




In [8]:
```
#Sign based function -0.5
layer_sign = LateralInhibitionLayer(-0.5, trafunc=signtrafu
nc)
x = np.random.randint(2,size=10)
plt.plot(x)
plt.plot(layer_sign.out(x))
```

Out[8]: [<matplotlib.lines.Line2D at 0x7f7b0c7dd4e0>]



# Task 2 Lateral Inhibition: trained (2 points - programming)

Use the output of the lateral inhibition layer as a target output for an MLP with 1 layers of 10 neurons. Try out both tranfer functions for the lateral inhibition layer (target) but only the logistic for training the MLP. Use $w = -0.25$ for the inhibition.

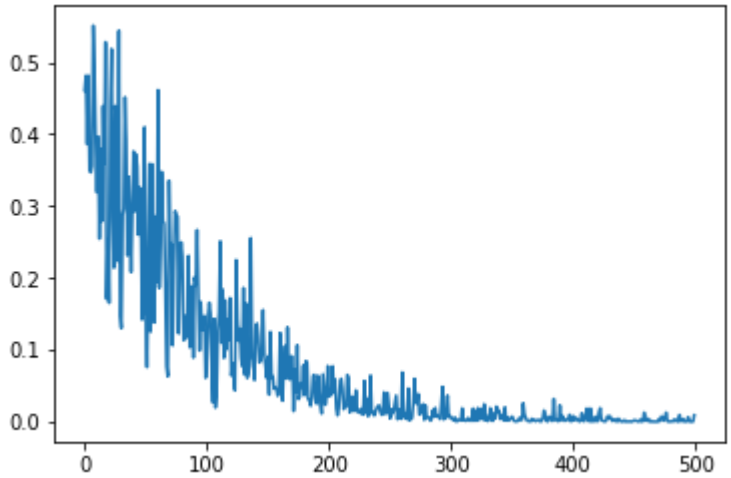
Determine a number of iterations that you decide to be enough for the network to have converged.

Measure the execution time by using the python module time and time.time() to get the current time. Also, investigate the weights after initialization and after the last iteration.

If you haven't solved task 1, you can use the class *LateralInhibitionLayer(NodeNo,Neighbourweight,bias=0,trafunc=logistic)* from the Exercise3Helper with the method LateralInhibitionLayer.out(x) as the target function.

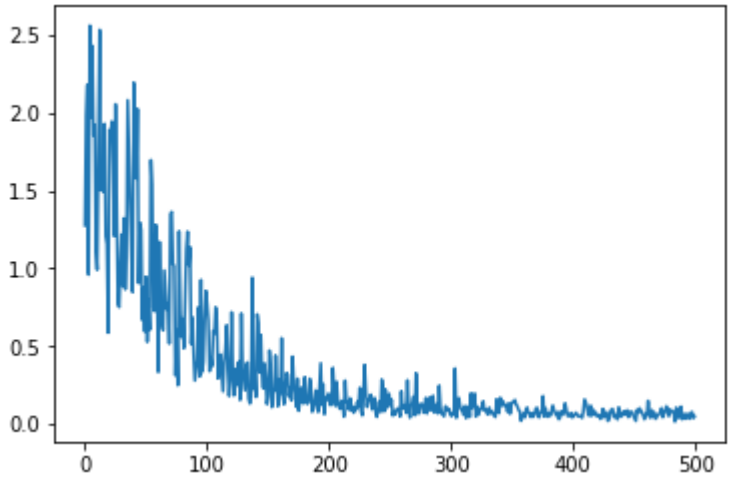
In [9]:
```
#with logistic
layer = LateralInhibitionLayer(-0.25)
x = np.random.randint(2,size=(500,10))
o = []
for n in range(500):
    o.append(layer.out(x[n]))

mlp = helper.MLP(10, 10)

#check weight before
print("Weights before")
for i in range(10):
    print(mlp.layers[0].nodes[i].w)

t1 = time.time()

plt.plot(mlp.train(NumIt=500,learnrate=1,x_train=x, o_train
=o))

t2 = time.time()

#check weight after
print("Weights after")
for i in range(10):
    print(mlp.layers[0].nodes[i].w)

print('time: ',t2 - t1)
```

```
Weights before
[-0.04469515 -0.7461625  -0.63102898  0.14991938  2.2967035
7 -1.26427792
  0.08810561  1.69859288  1.43930131 -1.55362495]
[ 0.20744136 -0.49150639 -0.56344617 -0.12081562  0.2894363
3 -0.43504143
  0.34479282  0.48071689  1.6822409  -0.26852216]
[ 1.17063032  0.49536925 -0.09924824  0.30728622  1.0340479
6 -0.89326946
  0.57776508 -1.53825453  0.31627395  0.12219656]
[-1.10583792 -0.81544753 -0.47927549  1.65630128 -0.5397639
3  0.21220166
  0.28163878  0.1211606  -0.69655894  0.14985224]
[-0.76085188 -0.50345275 -0.59755856  1.10763626  1.3404487
2 -0.16666512
 -1.05287155  0.46624355  1.8759696  -0.14807996]
[-1.25373942 -0.12005663  0.19648432  0.89173244  2.1106832
1  0.68176407
 -0.91222159  0.14342481 -0.41596975  1.15888968]
[ 0.19984924  0.32541681 -0.38899953  1.73271243  0.5328621
9 -0.07725437
  0.35175562  0.74051954 -0.40129671 -0.59573499]
[ 1.0470169  -0.02728557 -1.22061415 -2.30138742 -0.4080651
9  0.16169189
  0.90611072  1.06906688  0.63860585 -0.15429129]
[-0.18816185  1.39173597 -0.00258132 -0.30134433  0.2464092
5 -0.6220002
 -1.10004511  0.14751302  1.27646095  0.87144465]
[-0.10018277 -0.13596117 -0.77936723 -1.6038062  -0.7414740
2  1.01765444
  1.43579092 -1.21383747 -0.5001508   1.03522019]
Weights after
[ 1.08246635 -0.18786003  0.02696797  0.09076468  0.0587396
1  0.08979602
  0.07420845  0.11403704  0.08564657  0.02088697]
[-0.2090629   1.05719656 -0.22642205  0.05312506  0.0258764
8  0.07841308
  0.06287903  0.06818888  0.0683741   0.0389757 ]
[ 0.01727513 -0.22813301  1.00495574 -0.22937292  0.0110741
9  0.02711026
  0.0191006   0.02333014  0.02410071  0.01750984]
[-0.04327737 -0.05563402 -0.28214218  0.96828187 -0.2790603
-0.08264349
 -0.05839482 -0.0698517  -0.05862178 -0.03758895]
[-0.01123553 -0.02623775 -0.01252215 -0.2645928   1.0007063
7 -0.27844671
 -0.01892894 -0.02060263 -0.01142979 -0.01124939]
[ 3.28913465e-03  3.17178376e-03 -5.55015421e-03  1.7499932
8e-03
 -2.42040757e-01  1.00777182e+00 -2.51038299e-01  3.4616747
6e-03
 -5.00036548e-05  7.55994303e-03]
[-0.01131037 -0.01493936 -0.01134782 -0.01130705 -0.0016263
4 -0.26797683
  0.98745228 -0.26511493 -0.01772282 -0.01639833]
[-8.56337533e-03  1.49681747e-03  1.50532151e-03  8.5953499
```



In [10]:
```
layer = LateralInhibitionLayer(-0.25, trafunc=signtrafunc)
x = np.random.randint(2,size=(500,10))
o = []
for n in range(500):
    o.append(layer.out(x[n]))

mlp = helper.MLP(10, 10)
#print weight before
print("Weights before")
for i in range(10):
    print(mlp.layers[0].nodes[i].w)

t1 = time.time()
plt.plot(mlp.train(NumIt=500,learnrate=1,x_train=x, o_train
=o))
t2 = time.time()

#print weight after
print("Weights after")
for i in range(10):
    print(mlp.layers[0].nodes[i].w)

print('time: ',t2 - t1)
```

```
Weights before
[-0.47474971 -0.67582704 -1.32259917  0.80233973 -1.4847432
7  0.68074973
  1.19557104  0.34316238  0.99423074 -0.79903433]
[-0.19614966 -1.43128937  1.87107084 -1.0727831   1.5632175
5  0.28989575
 -1.02417627  0.98350123  0.47492775 -0.06300883]
[-2.67475703 -3.61834074  0.64930877 -0.6913922   0.2610165
3 -1.37157692
  0.25666989 -2.86689048 -0.89531355  0.39998348]
[-0.81062286  1.19736457  0.51910354 -0.02294064  0.0951366
8 -1.88966884
  1.62600982 -0.87092048 -0.16267302  0.37349009]
[ 0.80780189 -0.95809123  1.15888323  0.85658614 -0.0470166
3 -0.25087173
  0.63464692 -0.24349614 -1.80167505 -0.04489752]
[-0.94386606 -0.30752121 -1.33124612  2.55007818  0.4460881
6  0.03937739
 -0.25312311  1.33171732 -1.7930651   1.33160933]
[-0.33945803 -0.36607552  0.66272099  0.16389819 -1.0378252
2 -0.15476457
  0.62307657  0.60189895 -0.46563183 -0.57193028]
[ 0.30611658  0.29686119  0.47332026  0.66535971  1.0735942
1  0.56110788
 -0.19597704 -0.59400421 -0.63065819 -1.15321554]
[ 1.04606389 -0.52521523 -0.65728356 -0.84122046  0.4733848
3 -0.5318677
 -0.42327915  1.1928503  -2.12875233 -1.05960072]
[ 0.76065643  1.79592637 -0.33168467  0.92906956  0.9941262
7  0.82254939
 -0.89308444 -2.47011887  0.55052122 -1.26266495]
Weights after
[ 4.19569051 -1.90710366  0.00883261  0.0872395  -0.0617370
4  0.04367269
  0.12747095  0.03425456 -0.08145421 -0.00739037]
[-1.20874589e+00  4.67331747e+00 -9.78056278e-01  1.4607967
0e-01
  2.42758267e-01 -2.10675034e-01  1.01915307e-01  8.6222983
6e-02
  6.80589253e-02 -4.23554043e-03]
[-1.86024015e-01 -1.58265292e+00  5.60334468e+00 -1.5671162
5e+00
 -1.32433501e-03 -1.68737466e-01  2.88055561e-02 -3.5310905
9e-01
 -2.25091959e-01  3.15924862e-02]
[-0.03954886  0.10445039 -0.97764609  4.73390462 -1.0667914
4 -0.01544227
  0.12219956 -0.07913232 -0.0787268   0.01721775]
[-0.15978694 -0.07417722  0.05031461 -0.95850591  4.7694762
-0.89197278
 -0.08794438 -0.19922893 -0.03582464  0.04663194]
[-0.13419476 -0.05928641 -0.14372296  0.22251605 -0.9783124
4.93807957
 -1.01846091  0.22588258 -0.34492669  0.23411457]
[-0.21947551 -0.10321504 -0.09134435 -0.11683264 -0.0317261
```



# Task 3 1D-Convolutional NN layer (5 points - programming)

Build a 1-dimensional convolutional neural layer that can be trained with gradient descent and included as a layer into the MLP class from sheet 2.

To this extent, you combine the classes of single neurons and MLP layers into one that has one set of weights and a bias for the whole layer.

In the out function, it performs a convolution ( $*$ ) of the input with the weights (np.convolve(x,w,mode='same')), subtracts the bias and calculates the transferfunction of the result: $\mathbf{y} = f(\mathbf{w} * \mathbf{x} - b) = f(\mathbf{h})$.

The derivative of this is needed for the gradient descent training of the weights and the bias. For the single weights in an output layer, it becomes: $\frac{d\mathbf{y}}{dw_i} = \frac{df(\mathbf{h})}{d\mathbf{h}} \frac{d\mathbf{h}}{d\mathbf{x}} = f'(\mathbf{h}) \odot e_i * x$. Introducing $\mathbf{d}_i := \mathbf{e}_i * \mathbf{x}$, we can reach a squite simple formula for the weight updates. $\mathbf{e}_i$ is the unit vector, where entry $i$ is 1 and all others are 0.

This leads to the following weight and bias update rules, implemented in a *train(deltanext, weightsnext,...)* fuction similar to the perceptron:

$w_i(t+1) = w_i(t) - \eta \sum_k (y_k - o_k) f'(h_k) (\mathbf{d}_i)_{(k)}$

$b(t+1) = b(t) + \eta \sum_k (y_k - o_k) f'(h_k)$.

$(\mathbf{d}_i)_{(k)}$ is the $k$th element of $\mathbf{d}_i$.

If we use a local error gradient $\delta_l$ for the CNN layer l $\delta_l = f'(\mathbf{h}_l)\mathbf{W}_{l+1}\delta_{l+1}$ equivalent to that of a perceptron (lecture 2 slide 47), the columns i of the weightsnext matrix $\mathbf{W}_l$ for the layer before can be written as $[\mathbf{W}_l]_{(i)} = \mathbf{e}_i * \mathbf{w}$. To make it simpler, the weightsnext W returned by train are:

*WE=np.eye(len(self.lastin))*

*W=np.array([np.convolve(we,self.w,mode='same') for we in WE])*.

The weight update rule for any CNN layer l become:

$w_i(t+1) = w_i(t) - \eta \sum_k [\delta_{l+1}]_{(k)} [\mathbf{W}_{l+1}]_{(k)} f'(h_k) (\mathbf{d}_i)_{(k)}$

$b(t+1) = b(t) + \eta \sum_k [\delta_{l+1}]_{(k)} [\mathbf{W}_{l+1}]_{(k)} f'(h_k)$.

Note that $\mathbf{y}$,$\mathbf{h}_l$, $[\mathbf{W}_{l+1}]_{(k)}$, and $\delta_l$ have all the same dimensionality as the input $\mathbf{x}$.

In [11]:
```
class CNNlayer:

    def __init__(self,Filterwidth,weightinit=np.random.rand
n,biasinit=np.random.randn,trafunc=logistic,dtrafunc=dlogis
tic):
        self.Filterwidth = Filterwidth
        self.w = weightinit(Filterwidth)
        self.b = biasinit(1)
        self.trafunc = trafunc
        self.dtrafunc = dtrafunc


    def out(self,x):
        self.lastin = x
        self.lasth = np.convolve(self.w, x, mode='same') -
self.b
        self.lastout = self.trafunc(self.lasth)
        return self.lastout

    def delta(self, deltanext, weightsnext):
        self.lastdelta = self.dtrafunc(self.lasth) * np.dot
(deltanext, weightsnext)
        return self.lastdelta

    def train(self,deltanext,W,learnrate=0.1):
        self.lastdelta = self.delta(deltanext, W)
        for i in range(len(self.w)):
            ei = np.zeros(len(self.w)).astype(int)
            ei[i] = 1
            Di = np.convolve(ei , self.lastin, mode="same")
            errorgradient_di = sum(self.lastdelta * Di)
            self.w[i] = self.w[i] - learnrate*errorgradient
_di
        self.b = self.b + learnrate * sum(self.lastdelta)
        WE = np.eye(len(self.lastin))
        Wreturn = np.array([np.convolve(we, self.w, mode='s
ame') for we in WE])
        return self.lastdelta, Wreturn
```

# Task 4 Training a 1D-Convolutional NN on lateral inhibition (2 points - programming)

Use the output of the lateral inhibition layer as a target output for a CNN consisting of 1 layer with 10 neurons. You can simnply abuse the MLP class and assign a single layer using the CNNlayer class with 3 weights. Use $w = -0.25$ for the inhibition.

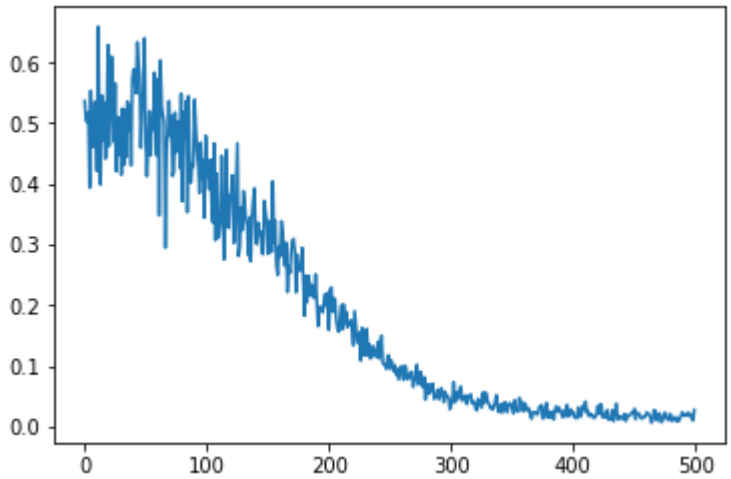
As inputs, use again a random vector of 0 and 1 (np.random.randint).

Measure and compare the execution time to the one of the MLP trained on the same problem (Task 2) by using the python module time and time.time() to get the current time. Use the same number of intializations. Also, investigate the weights after initialization and after the last iteration.

If you haven't solved task 1, you can use the class *LateralInhibitionLayer(NodeNo,Neighbourweight,bias=0,trafunc=logistic)* from the Exercise3Helper with the method LateralInhibitionLayer.out(x) as the target function. Similarly, if you haven't solved task 3, you can use the class *CNNlayer(Filterwidth, weightinit=np.random.randn, biasinit=np.random.randn, trafunc=logistic, dtrafunc=dlogistic)* of the helper for the layer.

In [19]:
```
iterations= 500
learnrate = 0.1

layer = LateralInhibitionLayer(-0.25)
x = np.random.randint(2,size=(iterations,10))
o = []
for n in range(iterations):
    o.append(layer.out(x[n]))

cnn = CNNlayer(10)
#cnn = helper.CNNlayer(10)

#print weight before
print('CNN weights before training')
print(cnn.w)

t1 = time.time()

#training of the layer
errors = np.zeros(iterations)

W = np.full(10, 1)

for it in range(iterations):
    desired_output = o[it]
    actual_output = cnn.out(x[it])
    deltanext_last = actual_output - desired_output
    errors[it] = 1/2*np.linalg.norm(np.array(deltanext_last
))
    deltanext, W = cnn.train(deltanext_last, W, learnrate=l
earnrate)


plt.plot(errors)
t2 = time.time()

#print weight after
print('CNN weights after training')
print(cnn.w)

print('time: ',t2 - t1)
```

```
CNN weights before training
[ 0.09082773  1.49874822  0.20842936  0.43111286 -1.1650763
4 -1.79609155
  0.38249429 -0.23229525 -0.70318759 -0.13119329]
CNN weights after training
[-0.03617328 -0.03786753 -0.02925368 -0.26952037  0.9944482
7 -0.27202301
 -0.02672137 -0.02836364 -0.0267795  -0.03312557]
time:  0.07714509963989258
```




## Task 5 Local Oscilations (2 points - programming)

Rewrite the neuron class below to receive a neuron with local feedback capable of oscillations. This will be a very simple example to demonstrate oscillations, biologically not very plausible as we e.g. don't include the refactory period.

To this extent, you need to save and feedback the last output in a certrain time interval. No training or delta functions are necessary. The output function becomes:
$y(t) = f(w \cdot x(t) + w_\tau y(t - \tau) - b)$

Initialize a regular perceptron and a local feedback neuron with the same weights and feed them the same random inputs for 200 iterations. The input should be of shape (2,), randomly distributed with an average of 0.5 and a standard deviation of 0.2. Repeat the 200 iterations each with every possible combination of the following parameters: $\tau = \{10, 5\}$ and $w_\tau = \{10, 5, 1, -1, -5, -10\}$.

In [20]:
```
#initialization of the model
w_taus=[10,5,1,-1,-5,-10]
taus=[10,5]
inputs = np.random.normal(0.5,0.2,(200,2))
iterations = 200
w = np.random.rand(2)
b = np.random.rand(1)
```

In [21]:
```
class neuron:

    def __init__(self,w,b,trafunc=logistic,dtrafunc=dlogist
ic):
        self.w=np.array(w)
        self.b=np.array(b)
        self.trafunc=trafunc
        self.dtrafunc=dtrafunc

    def out(self, x):
        return self.trafunc(np.dot(self.w,x)-self.b)

class fb_neuron:
    def __init__(self,w,b,trafunc=logistic,dtrafunc=dlogist
ic):
        self.w=np.array(w)
        self.b=np.array(b)
        self.trafunc=trafunc
        self.dtrafunc=dtrafunc

        self.y_ = np.zeros(200)

    def out(self, x, t,tau, w_t):
        if t-tau < 0:
            y_f = 0
        else:
            y_f = self.y_[t-tau]
        y = self.trafunc(np.dot(self.w,x)+w_t*y_f-self.b)
        self.y_[t] = y
        return y
```

In [22]:
```
def tau_iteration(tau,w_tau):
    n = neuron(w,b)
    n_fb = fb_neuron(w,b)
    y_n = np.zeros(iterations)
    y_n_fb = np.zeros(iterations)

    for t in range(iterations):
        y_n[t] = n.out(inputs[t])
        y_n_fb[t] = n_fb.out(inputs[t],t,tau,w_tau)

    plt.figure()
    plt.plot(y_n,color ='blue',label = 'perceptron')
    plt.plot(y_n_fb,color = 'red',label = 'feedback neuron'
)
    plt.title('tau,w_tau {}'.format((tau,w_tau)) )
    plt.legend()
```

In [23]:
```
for tau in taus:
    for w_tau in w_taus:
        tau_iteration(tau,w_tau)
```