In [17]:
```
%matplotlib inline
```

Vanilla Self Organizing Map (SOM)

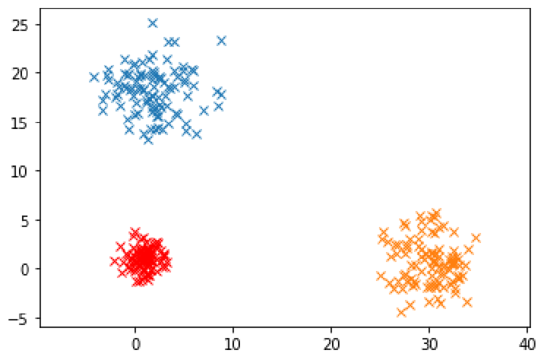
This Jupyter page is a place for you to play around with a BASIC SOM/SOFM.

Code assumptions:

- 8x8 map (that you can make bigger or smaller, no code support here for different topologies)
- Init is a uniformly spaced grid (between data min and max w.r.t. each dimension)
- I just randomly pick points
- I picked a fixed learning rate that slows the algorithm down (to "converge" over time)
- I only considered the 4 "directly" connected neighbors (no 2D Von Neumann diagonals)

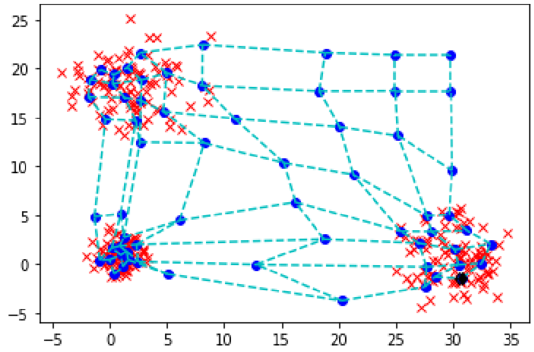
First, lets make our data set

In [18]:
```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
import time
import pylab as pl
from IPython import display
from tqdm import tqdm

NumPointsPerClass = 100

# class 1
c1_mean = [1,1]
c1_cov = [[1, 0], [0, 1]]
c1_x = np.random.multivariate_normal(c1_mean, c1_cov, NumPointsPerClass)

# class 2
c2_mean = [2, 18]
c2_cov = [[8, 0], [0, 6]]
c2_x = np.random.multivariate_normal(c2_mean, c2_cov, NumPointsPerClass)

# class 2
c3_mean = [30, 1]
c3_cov = [[6, 0], [0, 6]]
c3_x = np.random.multivariate_normal(c3_mean, c3_cov, NumPointsPerClass)

# plot it
plt.plot(c1_x[:,0], c1_x[:,1], 'rx')
plt.plot(c2_x[:,0], c2_x[:,1], 'x')
plt.plot(c3_x[:,0], c3_x[:,1], 'x')
plt.axis('equal')
plt.show()

# make data set
X = np.concatenate((c1_x, c2_x, c3_x), axis=0)
l1 = np.ones(c1_x.shape[0])
l2 = np.zeros(c2_x.shape[0])
l3 = np.zeros(c3_x.shape[0])
L = np.concatenate((l1, l2, l3), axis=0)
```

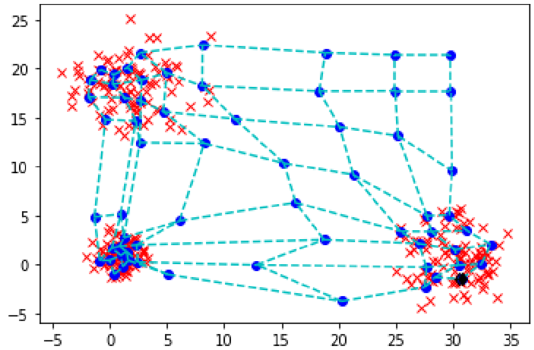


In [19]:
```
# size of the SOM
SomSize = 8

# make our weight map - pick some subset of our data
#W = np.zeros((SomSize,SomSize,2))
#for i in range(SomSize):
#    for j in range(SomSize):
#        whichind = np.random.randint(1,NumPointsPerClass*2)
#        W[i,j,0] = X[whichind,0]
#        W[i,j,1] = X[whichind,1]

# make our weight map - start with a grid over the space
W = np.zeros((SomSize,SomSize,2))
for i in range(SomSize):
    for j in range(SomSize):
        W[i,j,0] = (i/SomSize) * (np.max(X[:,0]) - np.min(X[:,0])) + np.min(X[:,0])
        W[i,j,1] = (j/SomSize) * (np.max(X[:,1]) - np.min(X[:,1])) + np.min(X[:,1])

# we will store our distances to things in this simple data structure
DVals = np.zeros((SomSize,SomSize))

# show plot/animation during algorithm? (if yes, keep NoIts low!!! (or goes on forever))
Show = 1

# how many epochs?
NoIts = 2000

# learning rate (driven below by iteration counter)
Lrate = 1.0

# We can randomly pick samples below (with replacement) or we can ...
#  randomly sort our data and sequentially walk through it
SampleWithReplacement = 0
SampleArray = np.random.permutation(NumPointsPerClass*3)

# the SOM
for k in tqdm(range(NoIts),'Main Loop'):

    # pick sample

    if( SampleWithReplacement == 1 ):
        RandomSampleIndex = np.random.randint(1,NumPointsPerClass*3)
    else:
        RandomSampleIndex = SampleArray[ k % (NumPointsPerClass*3) ]

    ###############################
    ###############################

    # find Euclidean distance of the selected point to everyone
    # yes, its slow, not teaching you how to efficiently code here! ;-)

    for i in range(SomSize):
        for j in range(SomSize):
            v = W[i,j,:] - X[RandomSampleIndex,:]
            v = np.multiply(v,v)
            v = np.sum(v)
            DVals[i,j] = np.sqrt(v) # yes, if your just comparing points, need the sqrt?

    # who is the closest to our sampled point? (the winner)

    MIndex = np.unravel_index(DVals.argmin(), DVals.shape)

    ###############################
    ###############################

    # lets slow this algorithm down over time

    Lrate = np.exp( (-1.0) * k / (NoIts * 0.5) )

    ###############################
    ###############################

    # draw? (and should we do only every say 50 iterations?)

    if( Show == 1 and (k % 50 == 0) ):

        # clear our plot
        pl.clf()
```

```
        # plot the data set
        plt.plot(X[:,0], X[:,1], 'rx')

        # plot our data point
        plt.plot(X[RandomSampleIndex,0], X[RandomSampleIndex,1], 'kd')

        # plot the SOM weight locations
        for i in range(SomSize):
            for j in range(SomSize):
                plt.plot(W[i,j,0],W[i,j,1], 'bo')

        # plot their edges
        for i in range(SomSize-1):
            for j in range(SomSize):
                plt.plot([W[i,j,0], W[i+1,j,0]], [W[i,j,1], W[i+1,j,1]], 'c--')
        for i in range(SomSize):
            for j in range(SomSize-1):
                plt.plot([W[i,j,0], W[i,j+1,0]], [W[i,j,1], W[i,j+1,1]], 'c--')

        # plot the winner
        plt.plot(W[MIndex[0],MIndex[1],0],W[MIndex[0],MIndex[1],1], 'kd')

        # animation, so pause it!
        display.clear_output(wait=True)
        display.display(pl.gcf())
        time.sleep(0.03)

    ################################
    ################################

    # now, update

    # look over direct neighbors
    BlendingFactor = 0.1 * Lrate
    # look above
    indxi = MIndex[0]
    indxj = MIndex[1] - 1
    if( indxj >= 0 ):
        W[indxi,indxj,:] = W[indxi,indxj,:] + BlendingFactor * ( X[RandomSampleIndex,:] - W[indxi,indxj,:] )
    # look left
    indxi = MIndex[0] - 1
    indxj = MIndex[1]
    if( indxi >= 0 ):
        W[indxi,indxj,:] = W[indxi,indxj,:] + BlendingFactor * ( X[RandomSampleIndex,:] - W[indxi,indxj,:] )
    # look right
    indxi = MIndex[0] + 1
    indxj = MIndex[1]
    if( indxi < SomSize ):
        W[indxi,indxj,:] = W[indxi,indxj,:] + BlendingFactor * ( X[RandomSampleIndex,:] - W[indxi,indxj,:] )
    # look below
    indxi = MIndex[0]
    indxj = MIndex[1] + 1
    if( indxj < SomSize ):
        W[indxi,indxj,:] = W[indxi,indxj,:] + BlendingFactor * ( X[RandomSampleIndex,:] - W[indxi,indxj,:] )

    # update our current point
    W[MIndex[0],MIndex[1],:] = W[MIndex[0],MIndex[1],:] + Lrate * ( X[RandomSampleIndex,:] - W[MIndex[0],MIndex[1
],:] )

##################################
##################################
# draw the final weight map

pl.clf()
plt.plot(X[:,0], X[:,1], 'rx')
for i in range(SomSize):
    for j in range(SomSize):
        plt.plot(W[i,j,0],W[i,j,1], 'bo')
```




```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-19-1290679a172f> in <module>
    103         display.clear_output(wait=True)
    104         display.display(pl.gcf())
--> 105         time.sleep(0.03)
    106 
    107     ###############################

KeyboardInterrupt: 
```



Things for you to ponder

- What is the *right* map size?
- What is the *right* map topological and neighborhood structure?
- Does the SOM "converge"?
- Are there always going to be weights "between" our clusters/classes?
- Any way to speed up what I did above?