# PROBLEM - 1

Given -

$$J(n) = \frac{1}{2}\left[ y_d(n) - \sum_{k=1}^{N} w_k(n) \exp\left(-\frac{\|x(n) - c_k(n)\|^2}{2\sigma_k^2(n)}\right)\right]^2$$

(1) $$w(n+1) = w(n) - \mu_w \frac{\partial}{\partial w} J(n)\Big|_{w = w(n)}$$

(2) $$c_k(n+1) = c_k(n) - \mu_c \frac{\partial}{\partial c_k} J(n)\Big|_{c_k = c_k(n)}$$

(3) $$\sigma_k(n+1) = \sigma_k(n) - \mu_\sigma \frac{\partial}{\partial \sigma_k} J(n)\Big|_{\sigma_k = \sigma_k(n)}$$

* For equation (1) we need to show

$$w(n+1) = w(n) + \mu_w e(n) \psi(n), \text{ where } \psi(n) = [\phi\{x(n), c_1, \sigma_1\} \ldots \phi\{(x(n), c_n, \sigma_n\}]$$

$$w(n+1) = w(n) - \mu_w \frac{\partial}{\partial w} J(n)\Big|_{w = w(n)}$$

On replacing the given value of $J(n)$

$$w(n+1) = w(n) - \mu_w \left(\frac{\partial}{\partial w} \frac{1}{2}\left[ y_d(n) - \sum_{k=1}^{N} w_k(n) \exp\left(\frac{-\|x(n) - c_k(n)\|^2}{2\sigma_k^2(n)}\right)\right]^2\right)$$

On replacing the given value of J(n)

$$W(n+1) = W(n) - \mu_w \frac{\partial}{\partial w}\left[\frac{1}{2}\left(y_d(n) - \sum_{K=1}^{N} w_K(n).\exp\left(\frac{-\|x(n)-c_K(n)\|^2}{2\sigma_K^2(n)}\right)\right)^2\right]$$

we can also write the above as

$$W(n+1) = W(n) - \mu_w \frac{\partial}{\partial w}\left[\frac{1}{2}\left(y_d(n) - w^T(n).\phi\right)^2\right], \text{ where } w^T(n)$$

denotes weight for n-th iteration and $\phi$ denotes radial function outputs for n-th iteration

Now, calculating the partial derivative with respect to w-vector

$$= \frac{\partial}{\partial w}\left[\frac{1}{2}\left(y_d(n) - w^T(n).\phi\right)^2\right]$$

we know if $f(x) = (g(x))^n$
$\Rightarrow f'(x) = n \times (g(x))^{n-1} \times g'(x)$

$$= \frac{1}{2} \times 2 \times (y_d(n) - w^T(n).\phi)^{2-1} \times \frac{\partial}{\partial w}(-w^T(n).\phi)$$

We know $w^T(n).\phi$ is a scalar value
Hence we can use scalar by vector identity

$$\frac{\partial}{\partial x} x^T a = a^T$$

$$= (y_d(n) - w^T(n).\phi) \times (-\phi^T)$$

We can replace $w^T(n).\phi = y(n)$, since it is the network's output for nth iteration

$$= -(y_d(n) - y(n)) \times [\phi^T$$

expanding $\phi$

$$= -(y_d(n) - y(n)) \times [\phi\{x(n), c_1, \sigma_1\}, \phi\{x(n), c_2, \sigma_2\} \ldots \ldots \phi\{x(n), c_n, \sigma_n\}]^T$$

We are given

$$y_d(n) - y(n) = e(n)$$

$$[\phi\{x(n), c_1, \sigma_1\} \ldots\ldots \phi\{x(n), c_n, \sigma_n\}]^T = \psi(n)$$

On replacing we get

$$= -e(n).\psi(n)$$

On putting it back to the starting equation

$$w(n+1) = w(n) + \mu_w e(n) \psi(n)$$

* For equation (2) we need to show

$$c_k(n+1) = c_k(n) + \frac{\mu_c\, e(n)\, w_k(n)}{\sigma_k^2(n)}\, \phi\{x(n), c_k(n), \sigma_k\}\,[x(n) - c_k(n)]$$

given $\Rightarrow$ $c_k(n+1) = c_k(n) - \mu_c \dfrac{\partial}{\partial c_k} J(n)\Big|_{c_k = c_k(n)}$

On calculating partial derivative of $J(n)$ with respect to $c_k$

$$\frac{\partial}{\partial c_k} J(n) = \frac{\partial}{\partial c_k}\left[\frac{1}{2}\left(y_d(n) - \sum_{k=1}^{N} w_k(n) \exp\left(-\frac{\|x(n) - c_k(n)\|^2}{2\sigma_k^2(n)}\right)\right)^2\right]$$

we know, if $f(x) = (g(x))^n$

then $f'(x) = n \times (g(x))^{n-1} \times g'(x)$

$$= \frac{1}{2} \times 2 \times \left(y_d(n) - \sum_{k=1}^{N} w_k(n) \exp\left(-\frac{\|x(n) - c_k(n)\|^2}{2\sigma_k^2(n)}\right)\right)^{2-1} \times$$

$$\left(-w_k(n) \times \frac{(-\|x(n) - c_k(n)\|^{2-1} \times (-1) \times 2 \times}{2\sigma_k^2(n)} \exp\left(\frac{-\|x(n) - c_k(n)\|^2}{2\sigma_k^2(n)}\right)\right)$$

All derivates except for the [$c_k$-th] ( k-th ) radial function would be zero since they are constants.

Also we know, $\sum_{k=1}^{N} w_k(n) \exp\left(\frac{-\|x(n) - c_k(n)\|^2}{2\sigma_k^2(n)}\right)$

is $y(n)$, the output of network for iteration $n$.

Also, for k-th radial function we can write

$$\exp\left(\frac{-\|x(n)-c_k(n)\|^2}{2\sigma_k^2(n)}\right) = \phi\{x(n), c_k(n), \sigma_k\}$$

On using the above two we get

$$\frac{\partial J(n)}{\partial c_k} = -\mu_c \frac{(y_d(n)-y(n)) \times w_k(n) \times [x(n)-c_k(n)] \times \phi\{x(n), c_k(n), \sigma_k\}}{\sigma_k^2(n)}$$

On substituting $y_d(n) - y(n) = e(n)$ and everything the value of partial derivate back to the update equation

$$c_k(n+1) = c_k(n) + \frac{\mu_c e(n) w_k(n) (x(n)-c_k(n)) \phi\{x(n), c_k(n), \sigma_k\}}{\sigma_k^2(n)}$$

* For equation (3) we need to show

$$\sigma_k(n+1) = \sigma_k(n) + \mu_\sigma \frac{e(n) w_k(n)}{\sigma_k^3(n)} \phi\{x(n), c_k(n), \sigma_k\} \| x(n) - c_k(n) \|^2$$

given $\Rightarrow$ $\sigma_k(n+1) = \sigma_k(n) - \mu_\sigma \frac{\partial}{\partial \sigma_k} J(n) \Big|_{\sigma_k = \sigma_k(n)}$

On calculating partial derivative of $J(n)$ with respect to $\sigma_k$

$$\frac{\partial J(n)}{\partial \sigma_k} = \frac{\partial}{\partial \sigma_k} \left[ \frac{1}{2} \left( y_d(n) - \sum_{k=1}^{N} w_k(n) \exp\left( \frac{-\| x(n) - c_k(n) \|^2}{2 \sigma_k^2(n)} \right) \right)^2 \right]$$

We know, $f(x) = (g(x))^n$

$f'(x) = n \times (g(x))^{n-1} \times g'(x)$

$$= \frac{1}{2} \left[ 2 \times \left( y_d(n) - \sum_{k=1}^{N} w_k(n) \exp^{\frac{-\| x(n) - c_k(n) \|^2}{2 \sigma_k^2(n)}} \right) \times \left( - w_k(n) \times -\| x(n) - c_k(n) \|^2 \times (-2) \times \frac{1}{2 \sigma_k^3(n)} \times \exp^{\frac{-\| x(n) - c_k(n) \|^2}{2 \sigma_k^2(n)}} \right) \right]$$

We can replace -

$$\sum_{k=1}^{N} w_k(n) \exp^{\frac{-\| x(n) - c_k(n) \|^2}{2 \sigma_k^2(n)}} = y(n) \quad , \text{ and}$$

$$\exp\left( \frac{-\| x(n) - c_k(n) \|^2}{2 \sigma_k^2(n)} \right) = \phi\{x(n), c_k(n), \sigma_k\}$$

for k-th radial function

On simplifying we get

$$= -\frac{(y_d(n) - y(n)) \times w_k(n) \times \|x(n) - c_k(n)\|^2 \times \phi\{x(n), c_k(n), \sigma_k\}}{\sigma_k^3(n)}$$

Replacing the above in the given equation

$$\sigma_k(n+1) = \sigma_k(n) + \frac{\mu_\sigma e(n) w_k(n) \phi\{x(n), c_k(n), \sigma_k\} \|x(n) - c_k(n)\|^2}{\sigma_k^3(n)}$$

# A2Question2 - RBFN

July 4, 2023

[7]:
```
import numpy as np
import pandas as pd
import random
import matplotlib.pyplot as plt
import time
from sklearn.cluster import KMeans
import warnings
warnings.filterwarnings("ignore")
```

[8]:
```
# initialize dataset
N = 441
k = 150
x = []
y = []
np.random.seed(1)

for j in range(0,21):
    for k in range(0,21):
        x.append([-2+0.2*j,-2+0.2*k])

#x[i][0] represents the x-coordinate of the i-th data point,
#x[i][1] represents the y-coordinate of the i-th data point
#calculate  squared Euclidean distance from origin 0,0
for i in range(0,N):
    f = (x[i][0] - 0 **2 +x[i][1]- 0 **2 )

    if f <= 1:
        x[i].append(1)
    else:
        x[i].append(-1)

x = np.array(x).reshape(N,3)
# the first two columns represents for the sample data points, and the last␣
 ↪column is their label
```

[9]:
```
#training network
# to ensure that the model does not learn any ordering or bias from the␣
 ↪original arrangement of the data point
```

```
np.random.shuffle(x)
train_size = int(N * 0.8)

train_x  = x[:train_size, :-1]
test_x   = x[train_size:, :-1]
train_y  = x[:train_size, -1:]
test_y   = x[train_size:, -1:]
```

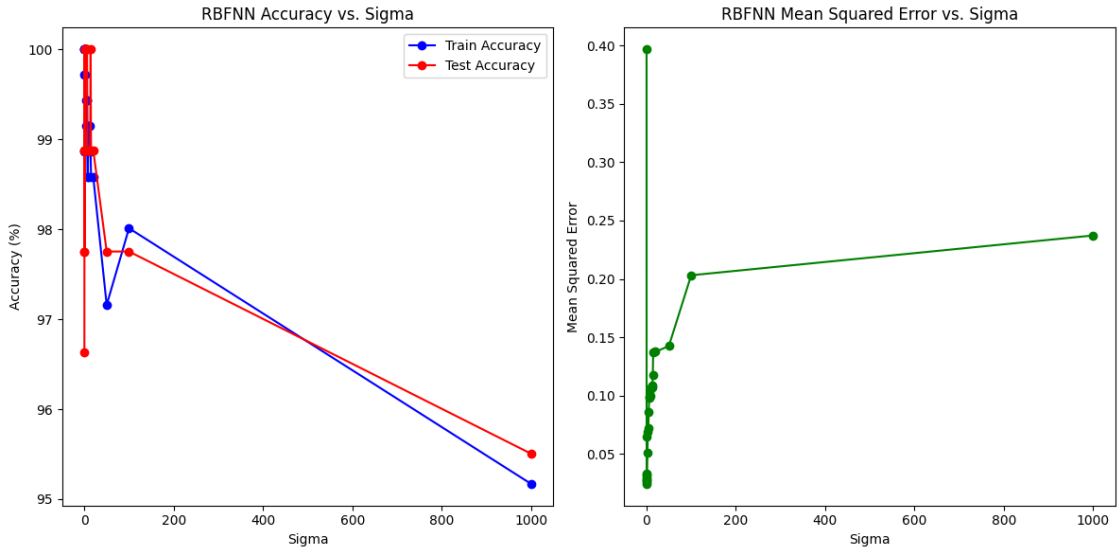
[10]:
```
# define a class of RBF Neural Network
class RBFNN(object):
    def __init__(self,k,sigma):
        random.seed(1)
        self.k = k
        self.weights = None
        self.sigma = sigma
        self.weights = np.random.randn(k)
        self.centers = None

    def kernel_function(self, x, center):
        return np.exp(-1 / (2 * self.sigma ** 2) * np.linalg.norm(x - center)␣
 ↪** 2)

    def interpolation_matrix(self, x):
        n = x.shape[0]
        G = np.empty((n, self.k))
        for i in range(0, n):
            for j in range(0, self.k):
                G[i, j] = self.kernel_function(x[i], self.centers[j])
        return G

    def random_centers(self, x):
        indices = np.random.choice(x.shape[0], self.k)
        centers = x[indices]
        return centers

    def kmeans_centers(self, x):
        kmeans = KMeans(n_clusters=self.k)
        kmeans.fit(x)
        centers = kmeans.cluster_centers_
        return centers

    def full_centers(self, x):
        centers = x
        return centers

    def fit_full(self, x, y):
        self.centers = self.full_centers(x)
```

```
        G = self.interpolation_matrix(x)
        self.weights = np.linalg.pinv(G) @ y
        interpolated_values = G @ self.weights
        error = interpolated_values - y
        squared_error = np.square(error)
        mean_squared_error = squared_error.mean()
        return mean_squared_error

    def fit_randomly(self, x, y):
        self.centers = self.random_centers(x)
        G = self.interpolation_matrix(x)
        self.weights = np.linalg.pinv(G) @ y
        interpolated_values = G @ self.weights
        error = interpolated_values - y
        squared_error = np.square(error)
        mean_squared_error = squared_error.mean()
        return mean_squared_error

    def fit_kmeans(self, x, y):
        self.centers = self.kmeans_centers(x)
        G = self.interpolation_matrix(x)
        self.weights = np.linalg.pinv(G) @ y
        interpolated_values = G @ self.weights
        error = interpolated_values - y
        squared_error = np.square(error)
        mean_squared_error = squared_error.mean()
        return mean_squared_error

    def predict(self, x, y):
        G = self.interpolation_matrix(x)
        pred = np.dot(G, self.weights)
        pred = np.sign(pred)
        accuracy = (pred == y)
        accuracy_mean = (accuracy).mean() * 100
        return accuracy_mean

    def __del__(self):
        self.weights = None
        self.centers = None
```

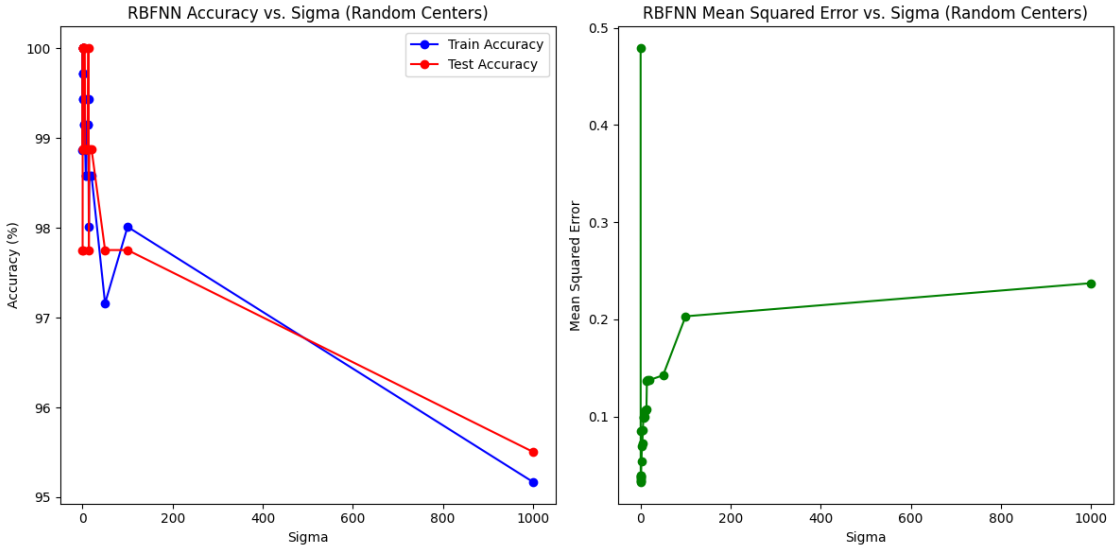
[11]:
```
sigmas = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.
↪9,1,2,3,4,5,6,7,8,9,11,12,13,14,15,20,50,100,1000]
```

[12]:
```
train_accuracy_list_full = []
accuracy_list_full = []
cost_list_full = []
```

```
  # using all training datapoints
for sigma in sigmas:
    rbf_full = RBFNN(k=150, sigma=sigma)

    cost = rbf_full.fit_full(train_x, train_y)
    train_full = rbf_full.predict(train_x, train_y)
    acc_full = rbf_full.predict(test_x, test_y)

    train_accuracy_list_full.append(train_full)
    accuracy_list_full.append(acc_full)
    cost_list_full.append(cost)
    del rbf_full

# Plotting the results
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(sigmas, train_accuracy_list_full, 'bo-', label='Train Accuracy')
plt.plot(sigmas, accuracy_list_full, 'ro-', label='Test Accuracy')
plt.xlabel('Sigma')
plt.ylabel('Accuracy (%)')
plt.title('RBFNN Accuracy vs. Sigma')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(sigmas, cost_list_full, 'go-')
plt.xlabel('Sigma')
plt.ylabel('Mean Squared Error')
plt.title('RBFNN Mean Squared Error vs. Sigma')

plt.tight_layout()
plt.show()
```

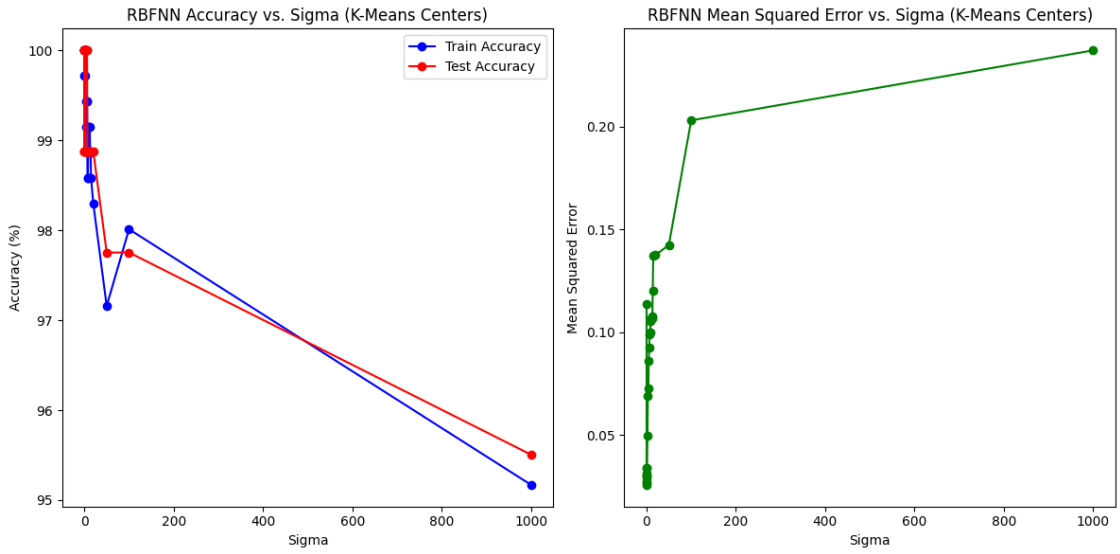


[13]:
```
train_accuracy_list_random = []
accuracy_list_random = []
cost_list_random = []
cost_rand = []
# using only 150 centers for random selected
for sigma in sigmas:
    rbf_random = RBFNN(k=150, sigma=sigma)

    cost_rand = rbf_random.fit_randomly(train_x, train_y)
    train_rand = rbf_random.predict(train_x, train_y)
    acc_rand = rbf_random.predict(test_x, test_y)

    train_accuracy_list_random.append(train_rand)
    accuracy_list_random.append(acc_rand)
    cost_list_random.append(cost_rand)
    del rbf_random

# Plotting the results
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(sigmas, train_accuracy_list_random, 'bo-', label='Train Accuracy')
plt.plot(sigmas, accuracy_list_random, 'ro-', label='Test Accuracy')
plt.xlabel('Sigma')
plt.ylabel('Accuracy (%)')
plt.title('RBFNN Accuracy vs. Sigma (Random Centers)')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(sigmas, cost_list_random, 'go-')
plt.xlabel('Sigma')
plt.ylabel('Mean Squared Error')
plt.title('RBFNN Mean Squared Error vs. Sigma (Random Centers)')

plt.tight_layout()
plt.show()
```



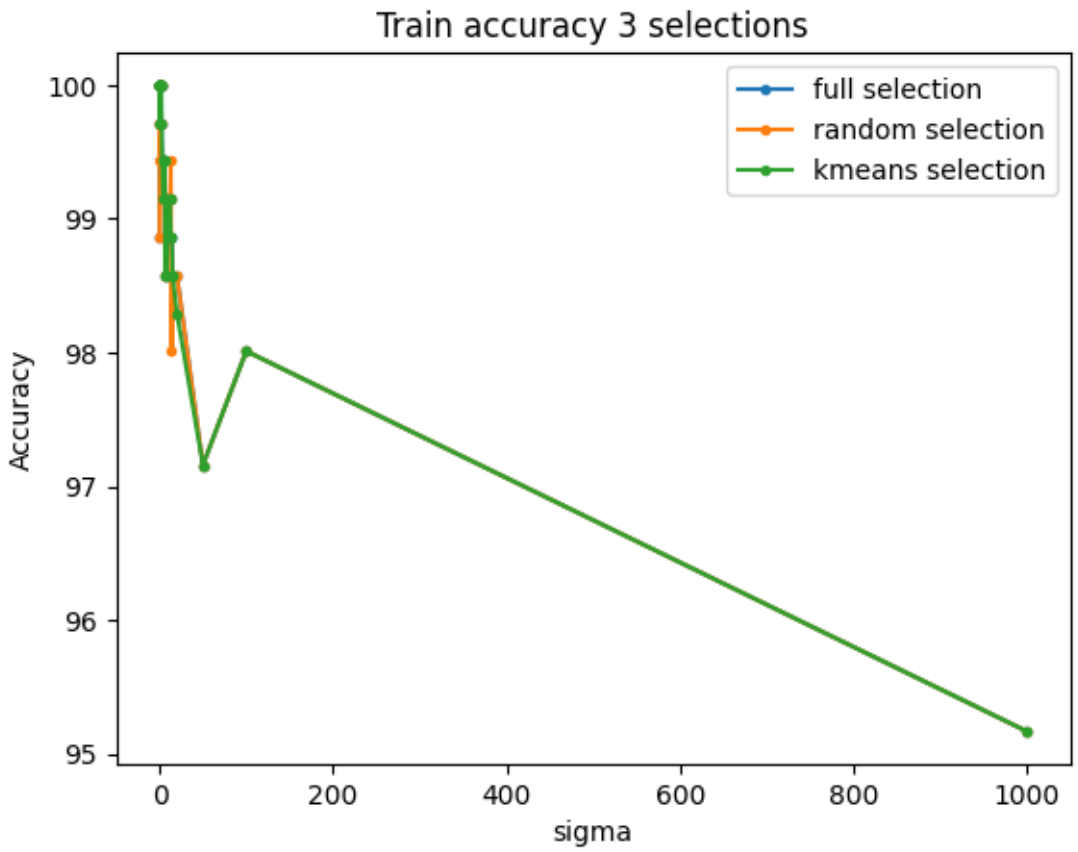

[14]:
```
train_accuracy_list_kmeans = []
accuracy_list_kmeans = []
cost_list_kmeans = []

# using only 150 centers for random selected
for sigma in sigmas:
    rbf_kmeans = RBFNN(k=150, sigma=sigma)

    cost = rbf_kmeans.fit_kmeans(train_x, train_y)
    train_kmeans = rbf_kmeans.predict(train_x, train_y)
    acc_kmeans = rbf_kmeans.predict(test_x, test_y)

    train_accuracy_list_kmeans.append(train_kmeans)
    accuracy_list_kmeans.append(acc_kmeans)
    cost_list_kmeans.append(cost)
    del rbf_kmeans

# Plotting the results
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(sigmas, train_accuracy_list_kmeans, 'bo-', label='Train Accuracy')
plt.plot(sigmas, accuracy_list_kmeans, 'ro-', label='Test Accuracy')
plt.xlabel('Sigma')
plt.ylabel('Accuracy (%)')
plt.title('RBFNN Accuracy vs. Sigma (K-Means Centers)')
plt.legend()

plt.subplot(1, 2, 2)
```

```
plt.plot(sigmas, cost_list_kmeans, 'go-')
plt.xlabel('Sigma')
plt.ylabel('Mean Squared Error')
plt.title('RBFNN Mean Squared Error vs. Sigma (K-Means Centers)')

plt.tight_layout()
plt.show()
```



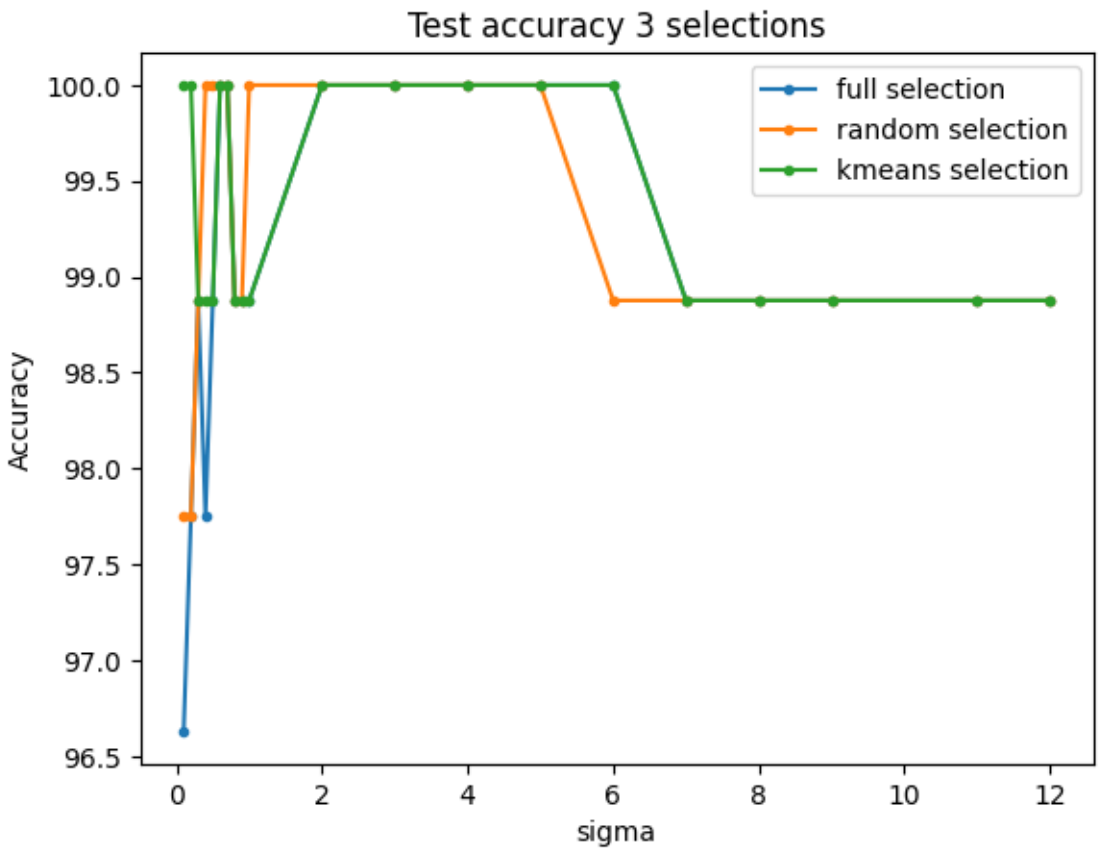

[15]:
```
#plot
plt.plot(sigmas,train_accuracy_list_full,marker='.',label="full selection")
plt.plot(sigmas,train_accuracy_list_random,marker='.',label="random selection")
plt.plot(sigmas,train_accuracy_list_kmeans,marker='.',label="kmeans selection")

plt.xlabel('sigma')
plt.ylabel('Accuracy')
plt.title('Train accuracy 3 selections')
plt.legend()
plt.show
```

[15]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```

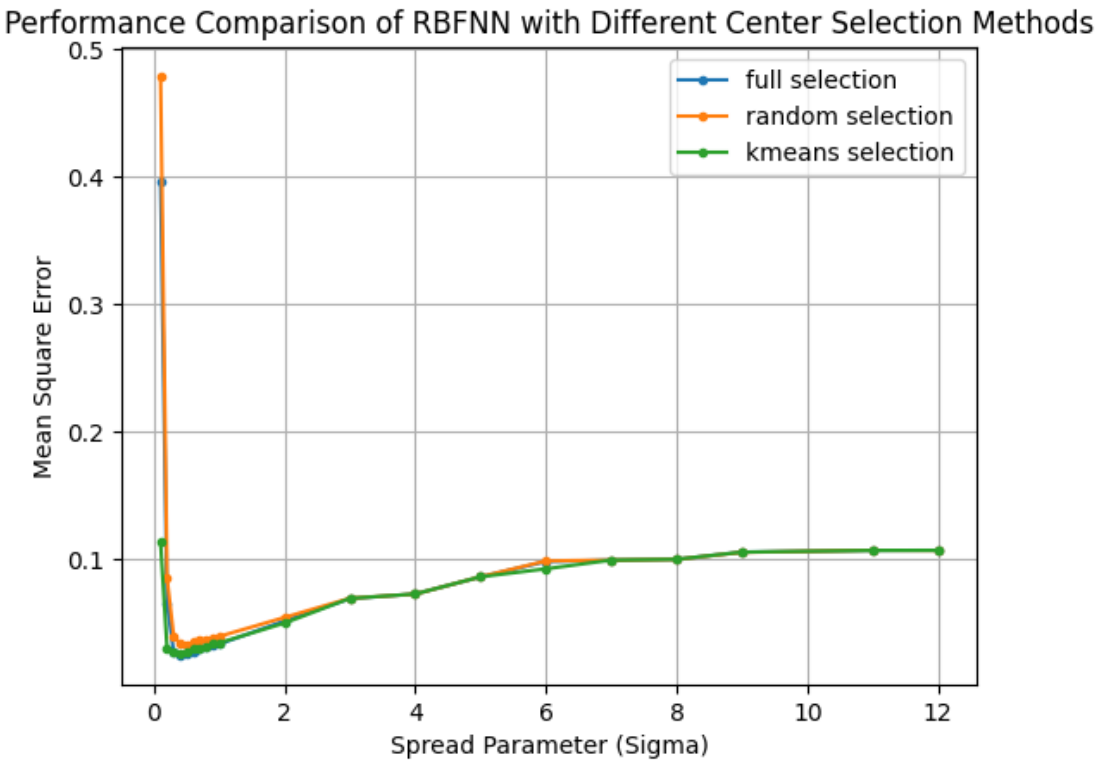



[16]:
```
#plot
plt.plot(sigmas[:20],accuracy_list_full[:20],marker='.',label="full selection")
plt.plot(sigmas[:20],accuracy_list_random[:20],marker='.',label="random␣
 ↪selection")
plt.plot(sigmas[:20],accuracy_list_kmeans[:20],marker='.',label="kmeans␣
 ↪selection")

plt.xlabel('sigma')
plt.ylabel('Accuracy')
plt.title('Test accuracy 3 selections')
plt.legend()
plt.show
```

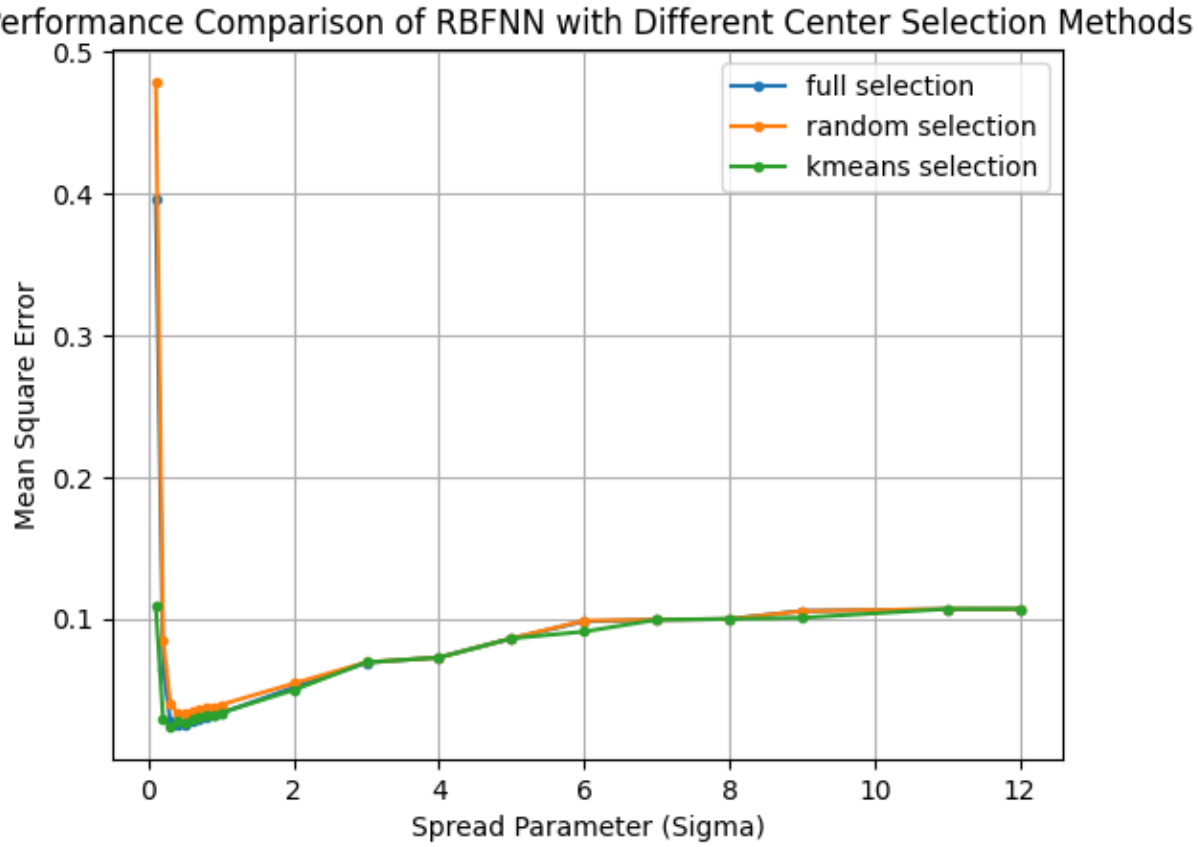
[16]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```




[17]:
```
# Plotting
plt.plot(sigmas[:20],cost_list_full[:20],marker='.',label="full selection")
plt.plot(sigmas[:20],cost_list_random[:20],marker='.',label="random selection")
plt.plot(sigmas[:20],cost_list_kmeans[:20],marker='.',label="kmeans selection")

plt.xlabel('Spread Parameter (Sigma)')
plt.ylabel('Mean Square Error')
plt.title('Performance Comparison of RBFNN with Different Center Selection␣
 ↪Methods')
plt.legend()
plt.grid(True)
plt.show()
```


Performance Comparison of RBFNN with Different Center Selection Methods
full selection
random selection
kmeans selection
Mean Square Error
Spread Parameter (Sigma)
0.5
0.4
0.3
0.2
0.1
0
2
4
6
8
10
12

**Performance**




The order of mean square error (MSE) being highest for random selection, followed by full centers, and then lower for K-Means, for small sigma values, and then becoming similar for larger sigma values, can be attributed to the following reasons:

1. Random Selection:
   - When randomly selecting centers, there is no guarantee that the selected centers will be representative or evenly distributed across the data.
   - Random selection can result in centers that are not well-aligned with the underlying data patterns, leading to less accurate predictions and a higher MSE.
   - The random selection approach introduces more variability and inconsistency in the performance, especially when the spread parameter is small. Thus, the RBF function struggles to capture the data patterns effectively.

2. Full Centers:
- Using all points as centers creates a high-dimensional space in which the RBF functions are defined.
- This approach tends to have higher model complexity, which leads to overfitting when the spread parameter is small.
- Overfitting occurs as the model becomes too specialized for the training data, resulting in low training error but high testing error. Thus leading to a higher MSE.

3. K-Means:
- The K-Means algorithm aims to find representative cluster centers that minimize the within-cluster sum of squares.
- K-Means can help identify clusters that are more aligned with the underlying data patterns, resulting in better generalization and a lower MSE compared to random selection.
- For smaller sigma values, the K-Means centers provide a more structured and effective representation of the data, resulting in a lower MSE.
- However, as the spread parameter becomes larger, the impact of the center selection method diminishes, and the performance of K-Means and full center approaches becomes similar.

In summary, the differences in MSE for different center selection methods can be attributed to the quality of the selected centers and the resulting model complexity.
Random selection may lead to less representative centers and a higher MSE. Full centers can result in overfitting, especially with small sigma values, leading to high MSE. K-Means tends to provide better generalization and lower MSE, particularly for smaller sigma values, by identifying representative cluster centers. However, as sigma increases, the performance differences between K-Means and full centers diminish.

# A2Question3 - Kohonen Self Organizing Map

July 4, 2023

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
import math

'''Create dataset of colors (24 data points with each having 3 dimensions)'''
colors = np.array([[0,255,255], [205,92,92], [255,255,0], [0,0,205],␣
 ↪[250,250,210], [0,100,0], [255,69,0], [255,20,147], [65,105,225],␣
 ↪[46,139,87], [250,128,114], [178,34,34], [0,206,209], [50,205,50],␣
 ↪[25,25,112], [255,0,255], [255,248,220], [0,0,128], [30,144,255], [139,0,0],␣
 ↪[107,142,35], [70,130,180], [165,42,42], [95,158,160]])

names_of_colors = ['aqua', 'indian red', 'yellow', 'medium blue', 'light golden␣
 ↪rod yellow', 'dark green', 'orange red', 'deep pink', 'royal blue', 'sea␣
 ↪green', 'salmon', 'firebrick', 'dark turquoise', 'lime green', 'midnight␣
 ↪blue', 'magenta', 'corn silk', 'navy', 'dodger blue', 'dark red', 'olive␣
 ↪drab', 'steel blue', 'brown', 'cadet blue']

'''Calibrate the color values to be between 0 & 1'''
colors = colors/255

'''Create a 100x100 output grid with coordinates to every node'''
op_grid = np.array([(i, j) for i in range(100) for j in range(100)])
```

[2]:
```
'''Sigma and learning rate decay'''
def change_learning_rate(lr, k, T):
    return lr * np.exp(-k/T)

def change_sigma(sig, k, T):
    return sig * np.exp(-k/T)
```

[3]:
```
def calc_distance(inp_x, old_weights):
    '''Calculate distance between KSOM nodes and the input'''
    norm = np.linalg.norm(old_weights - inp_x, axis=2)
    return norm
```

[4]:
```
def get_winner_node(inp_x, old_weights):
    dist = calc_distance(inp_x, old_weights)
    node = np.argmin(dist)
    '''Find the index of the node closest to the input'''
    min_distance_node_indices = np.unravel_index(node, dist.shape)
    return min_distance_node_indices
```

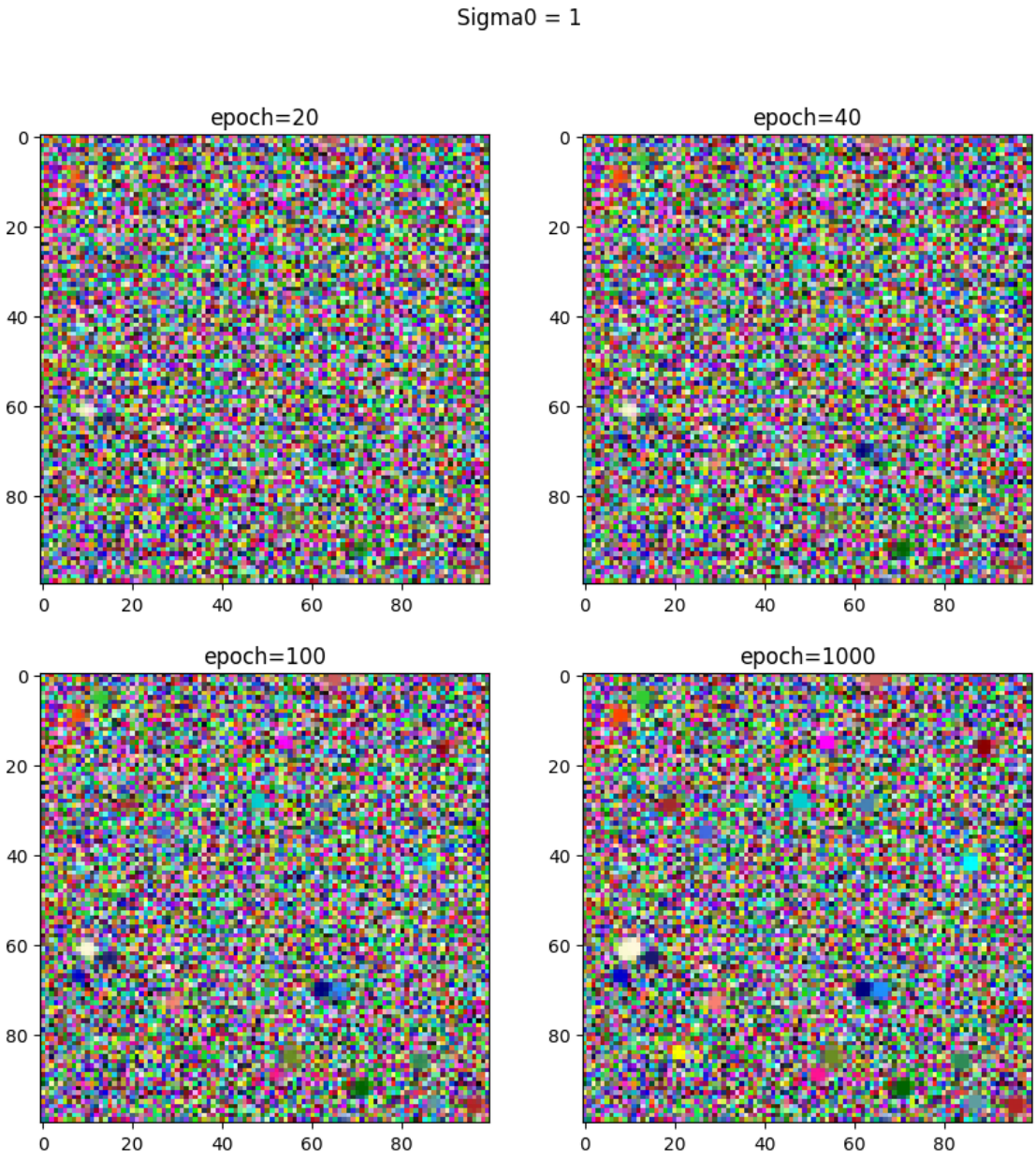
[5]:
```
def change_weights(lr, inp_x, old_weights,sig):
    new_weights = np.copy(old_weights)
    winner_node = get_winner_node(inp_x, old_weights)
    for i in range(100):
        for j in range(100):
            d = math.sqrt(sum([(x - y) ** 2 for x, y in zip(winner_node, [i,␣
 ↪j])]))
            h = np.exp(-1*(d**2)/(2 * (sig**2)))
            '''Apply weight updates to neighbours'''
            new_weights[i][j] = old_weights[i][j] + (lr*h*(inp_x -␣
 ↪old_weights[i][j]))
    return new_weights
```

[6]:
```
def run(sigma0):
    np.random.seed(1);
    weights = np.empty([100,100,3])

    for i in range(0, 100):
        for j in range(0, 100):
            weights[i][j][0] = random.randint(0,255)/255
            weights[i][j][1] = random.randint(0,255)/255
            weights[i][j][2] = random.randint(0,255)/255

    T = 1001
    lr = 0.8
    sigma = sigma0

    epoch_weights = []

    for epoch in range(1, T):
        '''Select a random data point'''
        ds = random.randint(0, 23)
        '''Feed the point to the map'''
        weights = change_weights(lr, colors[ds], weights, sigma)

        '''Update learning rate and sigma over epochs'''
        lr = change_learning_rate(0.8, epoch, T)
        sigma = change_sigma(sigma0, epoch, T)

        '''Copying weights to graph later'''
```

```
        if epoch == 20:
            epoch_weights.append(weights.copy())

        if epoch == 40:
            epoch_weights.append(weights.copy())

        if epoch == 100:
            epoch_weights.append(weights.copy())

        if epoch == 1000:
            epoch_weights.append(weights.copy())


    return epoch_weights
```

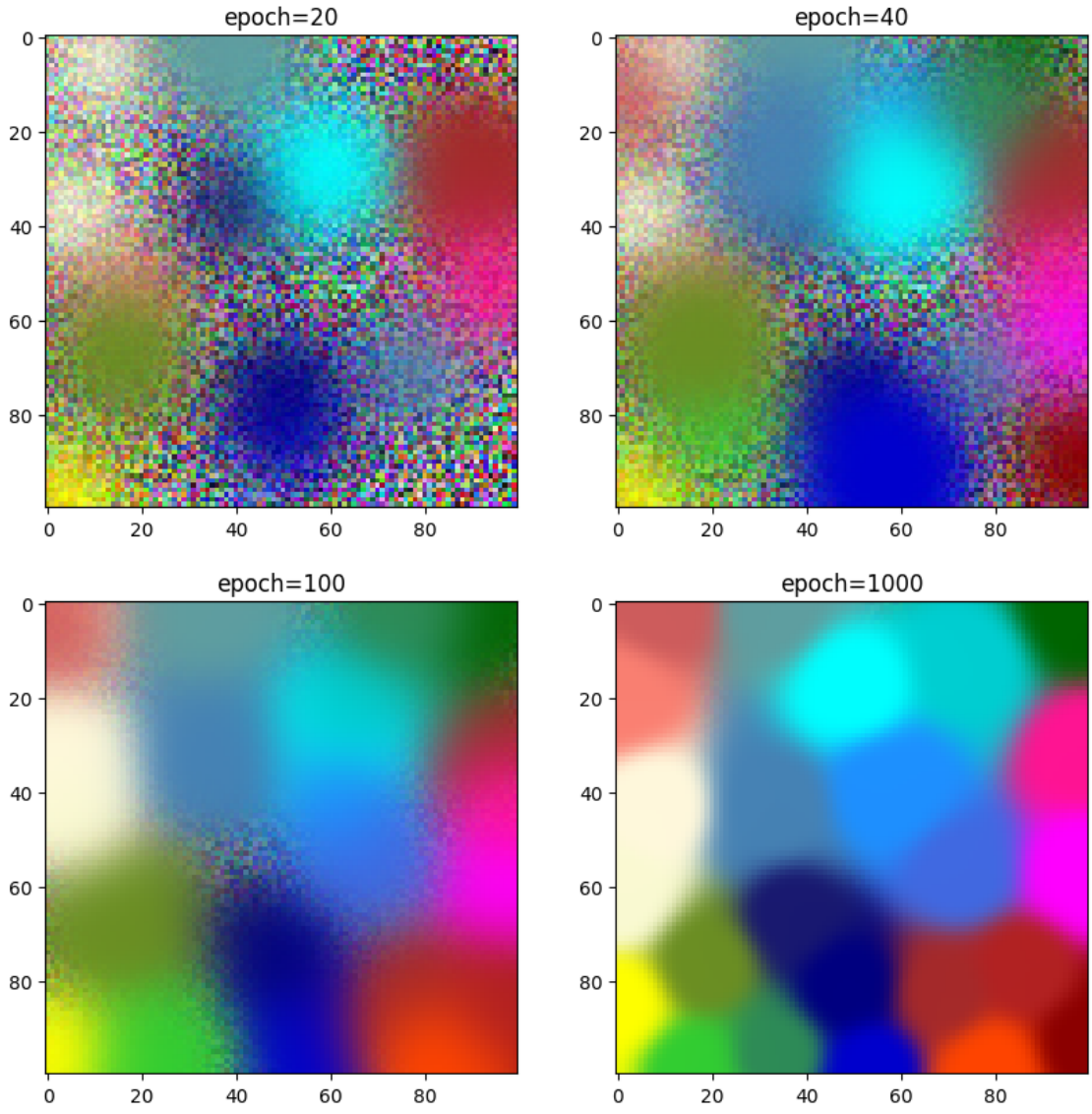
[7]:
```
def plot_graphs(epoch_weights, sigma):
    w_20 = epoch_weights[0]
    w_40 = epoch_weights[1]
    w_100 = epoch_weights[2]
    w_1000 = epoch_weights[3]

    plt.figure(figsize=(10, 10))
    plt.subplot(221)
    plt.title('epoch=20')
    plt.imshow(w_20)
    plt.subplot(222)
    plt.title('epoch=40')
    plt.imshow(w_40)
    plt.subplot(223)
    plt.title('epoch=100')
    plt.imshow(w_100)
    plt.subplot(224)
    plt.title('epoch=1000')
    plt.imshow(w_1000)
    plt.suptitle('Sigma0 = '+str(sigma))
    plt.show()  # Display the plot
```

[8]:
```
'''For sigma0 = 1'''
epoch_weights = run(1)
plot_graphs(epoch_weights, 1)
```

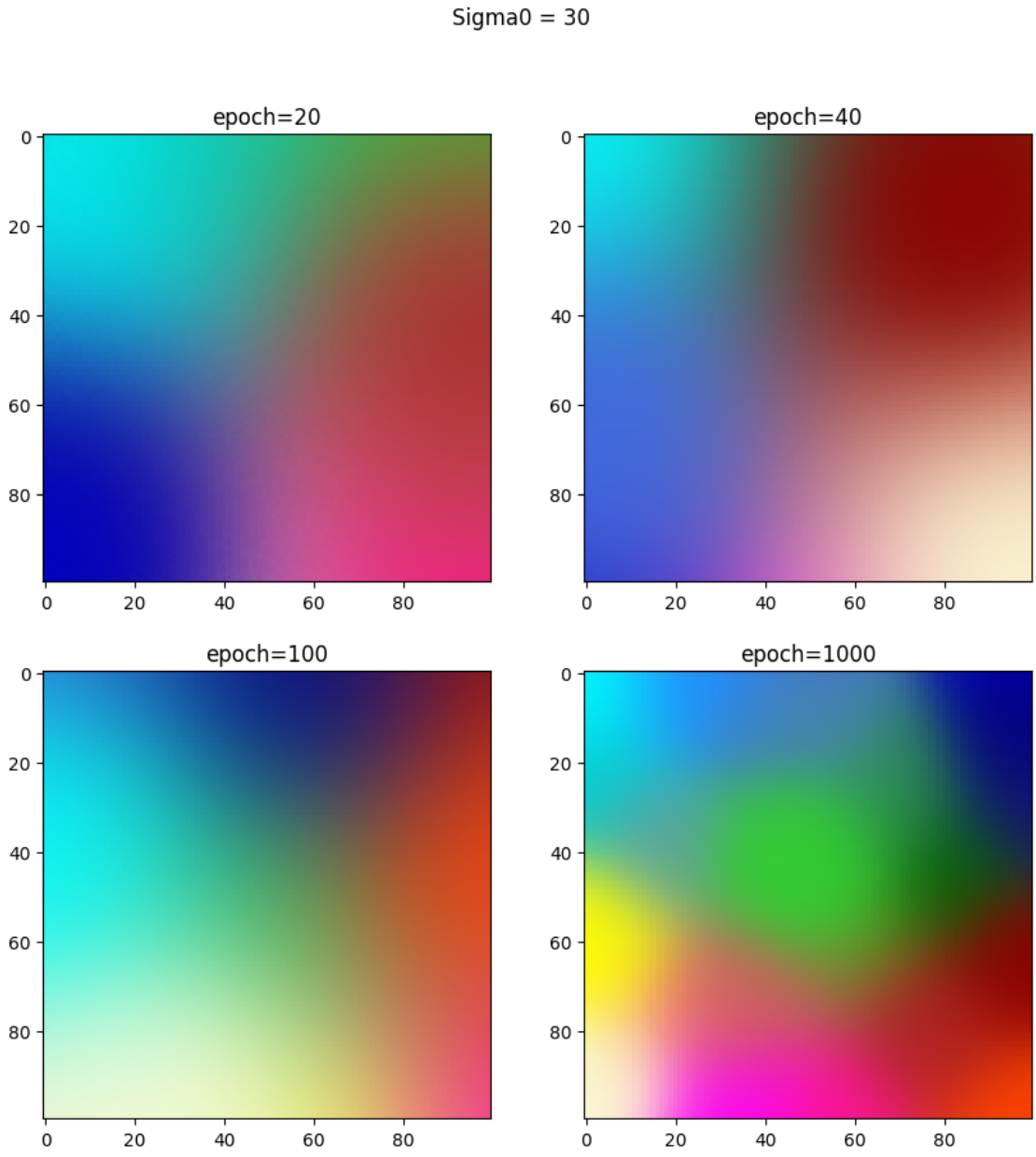



[9]:
```
'''For sigma0 = 10'''
epoch_weights = run(10)
plot_graphs(epoch_weights, 10)
```

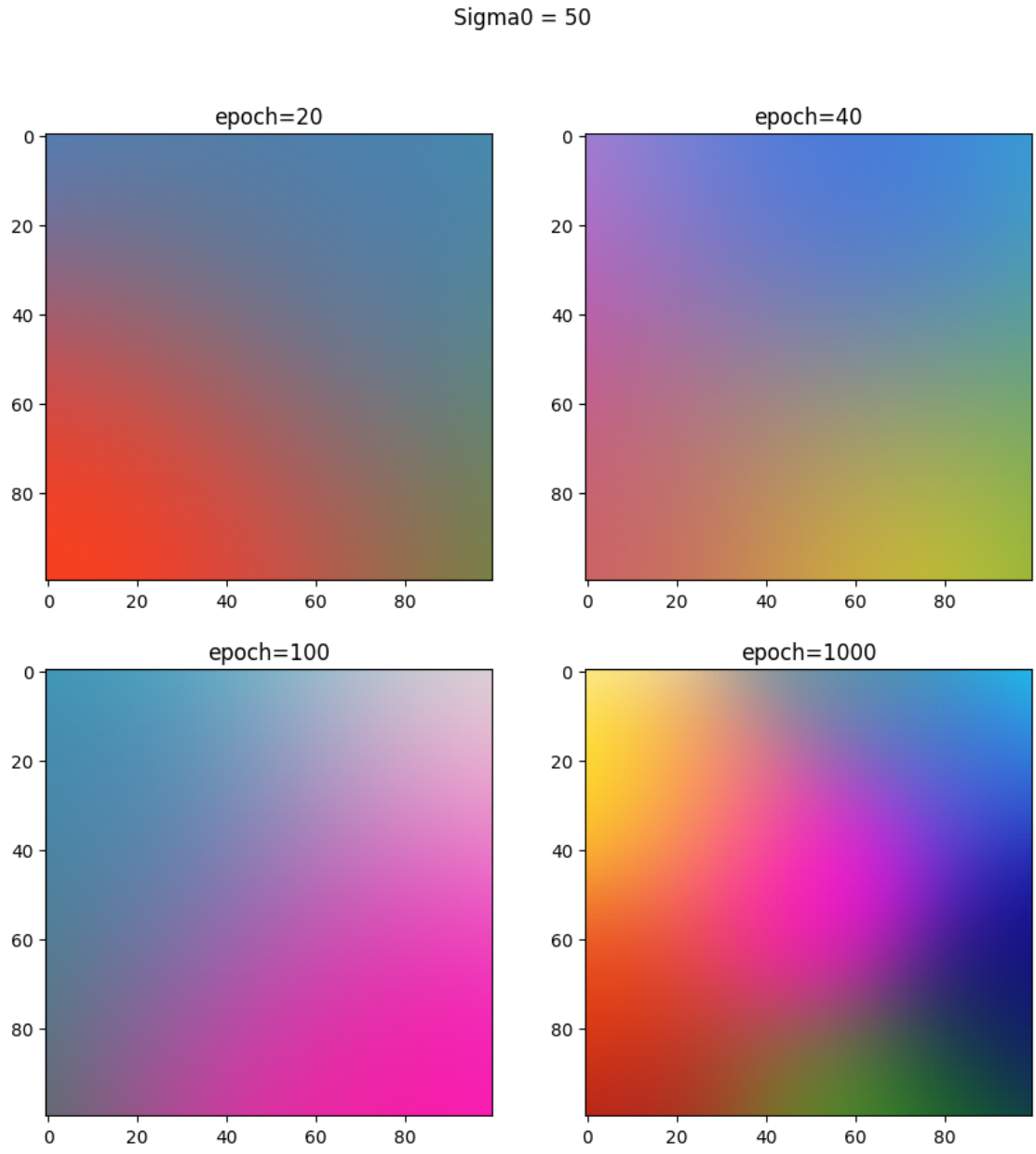
Sigma0 = 10



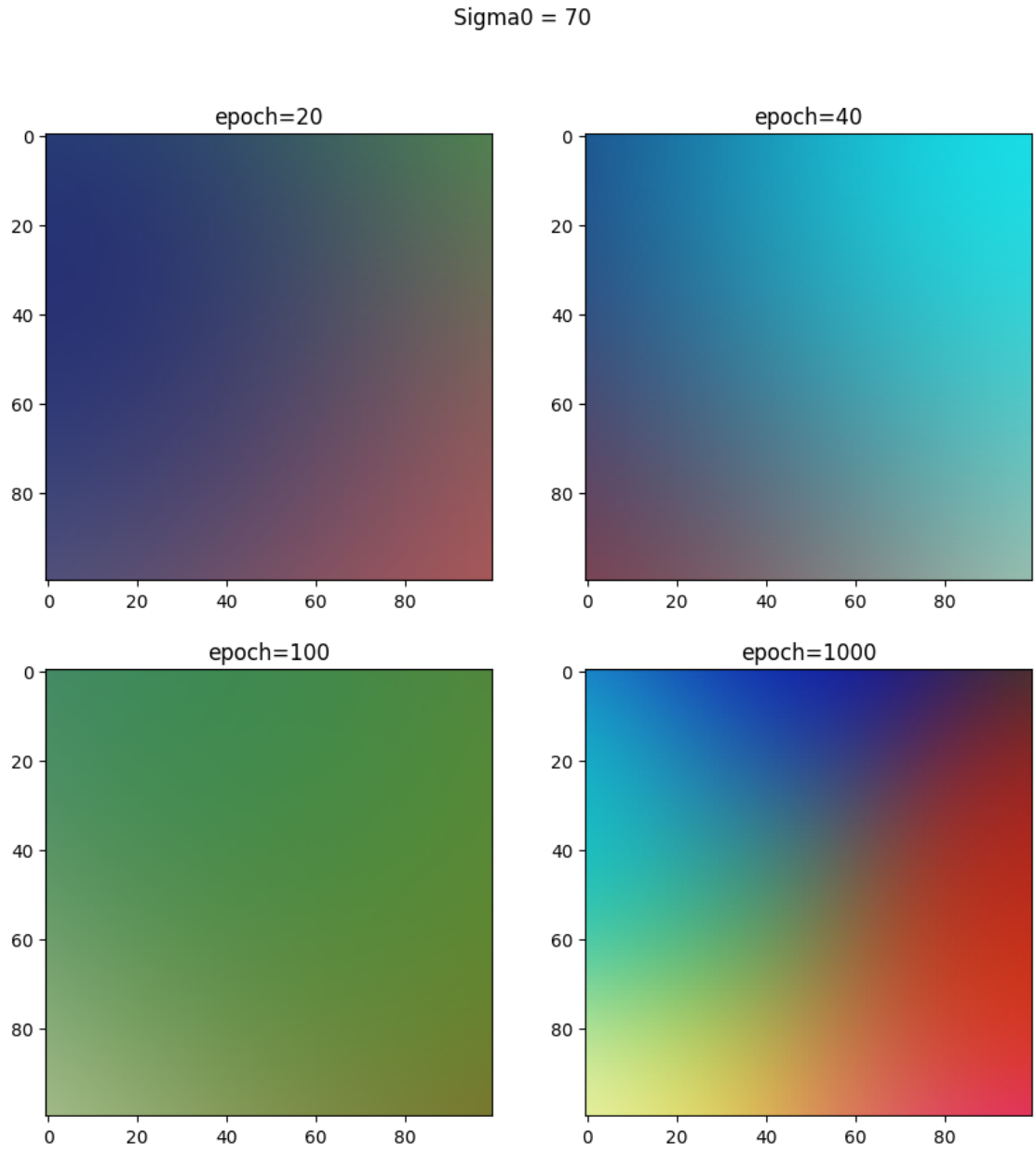

[10]:
```
'''For sigma0 = 30'''
epoch_weights = run(30)
plot_graphs(epoch_weights, 30)
```




[11]:
```
'''For sigma0 = 50'''
epoch_weights = run(50)
plot_graphs(epoch_weights, 50)
```




[12]:
```
'''For sigma0 = 70'''
epoch_weights = run(70)
plot_graphs(epoch_weights, 70)
```




# 1 Conclusion -

**Varying sigma0 from smaller to larger values:**

1. Sigma value is directly proportional to the influence of winning node over it's neighbours.
2. With smaller sigma, neighboring neurons have less influence by the winning neuron(winner takes all type of update) during the learning process. This leads to a sharper distinction between different colors in the map, resulting in clearer boundaries. While in case of larger sigmas the disctinctions is less sharp with fading boundaries.

3. Each neuron will represent a specific color more precisely, and there will be less blending or overlap of colors across neighboring neurons. When sigma increases overlap is increases and neighbouring colours blend more.

**Sigma decay over epochs:**

1. Over 1000 epochs value of sigma decrease to approximately 35% of the inital value. As sigma decreases, the influence radius narrows, and the updates become more localized to the immediate neighbors of the winning node.
2. It results in creating sharper boundaries and more distinct clusters within the KSOM and contributes towards convergence of the map.