Problem 1

Given

$$J(n) = \frac{1}{2}|e(n)|^2 = \frac{1}{2}[y_d(n) - y(n)]^2 \quad -(1)$$

$$J(n) = \frac{1}{2}\left[y_d(n) - \sum w_k(n)\,\phi[x(n), c_k(n), \sigma_k(n)]\right]^2 \quad -(2)$$

$$J(n) = \frac{1}{2}\left[y_d(n) - \sum w_k(n)\exp\left(-\frac{\|x(n) - c_k(n)\|^2}{\sigma_k^2(n)}\right)\right]^2 \quad -(3)$$

Network parameters are given by

$$w(n+1) = w(n) - \mu_w \frac{\partial}{\partial w} J(n) \quad -(4)$$

$$c_k(n+1) = c_k(n) - \mu_c \frac{\partial}{\partial c_k} J(n) \quad -(5)$$

$$\sigma_k(n+1) = \sigma_k(n) - \mu_\sigma \frac{\partial}{\partial \sigma_k} J(n) \quad -(6)$$

Proof:

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

Sol: As per equation (2)

$$J(n) = \frac{1}{2}\left[y_d(n) - \sum w_k(n)\,\phi[x(n), c_k(n), \sigma_k(n)]\right]^2$$

Now taking derivative w.r.to w

$$\frac{\partial J(n)}{\partial w} = \frac{\partial}{\partial w}\left[\frac{1}{2}\left(y_d(n) - \sum w_k(n)\,\phi[x(n), c_k(n), \sigma_k(n)]\right)^2\right]$$

$$\frac{\partial J(n)}{\partial w} = \frac{1}{2}\left[-2y_d(n) + 2\sum w_k(n)\,\phi(x(n), c_k(n), \sigma_k(n)\right] \cdot \sum \phi(x(n), c_k(n), \sigma_k(n)$$

$$\frac{\partial J(n)}{\partial w} = -\left[y_d(n) - \sum w_k(n)\,\phi[x(n), c_k(n), \sigma_k(n)\right] \cdot \sum \phi(x(n), c_k(n), \sigma_k(n)$$

Comparing eq ① and ② we get

$$e(n) = y_d(n) - \sum w_k(n)\,\phi[x(n), c_k(n), \sigma_k(n)]$$

$$\Rightarrow \frac{\partial J(n)}{\partial w} = -e(n) \cdot \sum \phi(x(n), c_k(n), \sigma_k(n)$$

Substituting this in eq ④

$$w(n+1) = w(n) - \mu_w\left[-e(n) \cdot \sum \phi(x(n), c_k(n), \sigma_k(n)\right]$$

$$w(n+1) = w(n) + \mu_w e(n) \cdot \sum \phi(x(n), c_k(n), \sigma_k(n)]$$

$$\Rightarrow \boxed{w(n+1) = w(n) + \mu_w e(n)\,\psi(n)}$$

(2) Show that

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

Sol:- Considering equation (3)

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

Now taking derivative w.r.to c_k

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

Substituting the above in eq (5)

$$c_k(n+1) = c_k(n) - \mu_c\left[\frac{\partial}{\partial c_k}\left[\frac{1}{2}\left[y_d(n) - \sum w_k(n) \exp\left[\frac{-\|x(n) - c_k(n)\|^2}{\sigma_k^2(n)}\right]\right]^2\right]^2\right]$$

$$c_k(n+1) = c_k(n) - \mu_c\left[\frac{\partial}{\partial c_k}\left[\frac{1}{2}\left[y_d(n)^2 + \left(\sum w_k(n) \exp\left[\frac{-\|x(n) - c_k(n)\|^2}{\sigma_k^2(n)}\right]\right)^2 - 2\, y_d(n) \sum w_k(n) \exp\left[\frac{-\|x(n) - c_k(n)\|^2}{\sigma_k^2(n)}\right]\right]\right]^2\right]$$

Considering for

$$c_1(n+1) = c_1(n) - \mu_c\left[\frac{\partial}{\partial c_1}\left[\frac{1}{2}\left[y_d(n)^2 + w_1(n) \exp\left[\frac{-\|x(n) - c_1(n)\|^2}{\sigma_1^2(n)}\right]\right]^2 - 2\, y_d(n)\, w_1(n) \exp\left[\frac{-\|x(n) - c_1(n)\|^2}{\sigma_1^2(n)}\right]\right]^2\right]$$

$$C_1(n+1) = C_1(n) - \mu_1 \left[\frac{1}{2} \left[-2y_d(n) + w_1(n) \exp\left[\frac{-\|x(n) - C_1(n)\|^2}{2\sigma_1^2(n)} \right] \cdot w_1(n) \cdot \exp\left[\frac{-\|x(n) - C_1(n)\|^2}{2\sigma_1^2(n)} \right] \cdot \frac{-2(x(n) - C_1(n)) \cdot (-1)}{2\sigma_1^2(n)} \right]$$

$$C_1(n+1) = C_1(n) - \mu_1 \left[-\left(y_d(n) - w_1(n) \exp\left[\frac{-\|x(n) - C_1(n)\|^2}{2\sigma_1^2(n)} \right] \cdot w_1(n) \cdot \exp\left[\frac{-\|x(n) - C_1(n)\|^2}{2\sigma_1^2(n)} \right] \cdot \frac{x(n) - C_1(n)}{\sigma_1^2(n)} \right) \right]$$

Comparing eq ① and ③ we get.

$$e(n) = y_d(n) - \sum w_k(n) \exp\left[\frac{-\|x(n) - C_1(n)\|^2}{2\sigma_1^2(n)} \right]$$

$$\Rightarrow C_1(n+1) = C_1(n) - \mu_1 \left(-e(n) \right) \cdot w_1(n) \exp\left[\frac{-\|x(n) - C_1(n)\|^2}{2\sigma_1^2(n)} \right] \cdot \frac{x(n) - C_1(n)}{\sigma_1^2(n)}$$

As per eq ② and ③

$$\exp\left(\frac{-\|x(n) - C_1(n)\|^2}{2\sigma_1^2(n)} \right) = \phi\left[x(n), C_1(n), \sigma_1(n) \right]$$

$$\Rightarrow C_1(n+1) = C_1(n) + \mu_1 e(n) \cdot w_1(n) \cdot \phi\left[x(n) \cdot C_1(n), \sigma_1(n) \right] \cdot \frac{x(n) - C_1(n)}{\sigma_1^2(n)}$$

Similarly for

$$c_k(n+1) = c_k(n) + \frac{\mu_1 e(n) \cdot w_k(n)}{\sigma_k^2(n)} \cdot \phi[x(n), c_k(n), \sigma_k(n)] \cdot x(n) - c_k(n)$$

$$\boxed{c_k(n+1) = c_k(n) + \frac{\mu_1 e(n) \cdot w_k(n)}{\sigma_k^2(n)} \cdot \phi[x(n), c_k(n), \sigma_k(n)] \cdot x(n) - c_k(n)}$$

(3) Show that

$$\sigma_k(n+1) = \sigma_k(n) + \frac{\mu_1 e(n) w_k(n)}{\sigma_k^3(n)} \phi[x(n), c_k(n), \sigma_k] \|x(n) - c_k(n)\|^2$$

Solution:

Considering equation (3)

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

Now taking derivative w.r.to σ_k

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

Substituting the above in eq (6)

$$\sigma_k(n+1) = \sigma_k(n) - \mu_1 \frac{\partial}{\partial \sigma_k}\left[\frac{1}{2}\left[y_d(n) - \sum w_k(n) \exp\left[\frac{-\|x(n) - c_k(n)\|^2}{2\sigma_k^2(n)}\right]\right]^2\right]$$

Considering it for one

$$\sigma_1(n+1) = \sigma_1(n) - \mu_\sigma \frac{\partial}{\partial \sigma_1}\left[\frac{1}{2}\left[y_d(n) - w_1(n)\exp\left[\frac{-\|x(n)-c_1(n)\|^2}{2\sigma_1^2(n)}\right]\right]^2\right]$$

$$\sigma_1(n+1) = \sigma_1(n) - \mu_\sigma\left[\left[\frac{1}{2}\left[-2y_d(n) + w_1(n)\exp\left[\frac{-\|x(n)-c_1(n)\|^2}{2\sigma_1^2(n)}\right]\cdot w_1\exp\left[\frac{-\|x(n)-c_1(n)\|^2}{2\sigma_1^2(n)}\right]\cdot\left(-2\frac{\|x(n)-c_1(n)\|^2(-)}{2\sigma_1^3(n)}\right)\right]\right]\right]$$

$$= \sigma_1(n) - \mu_\sigma\left[\left[y_d(n) - w_1(n)\exp\left[\frac{-\|x(n)-c_1(n)\|^2}{2\sigma_1^2(n)}\right]\cdot w_1\exp\left[\frac{-\|x(n)-c_1(n)\|^2}{2\sigma_1^2(n)}\right]\cdot\frac{\|x(n)-c_1(n)\|^2}{\sigma_1^3(n)}\right]\right]$$

comparing eq ① & ③

$$e(n) = y_d(n) - \sum w_k(n)\exp\left(\frac{-\|x(n)-c_1(n)\|^2}{2\sigma_1^2(n)}\right)$$

and

as per eq ② & ③

$$\phi(x(n), c_1(n), \sigma_1(n)) = \exp\left[\frac{-\|x(n)-c_1(n)\|^2}{2\sigma_1^2(n)}\right]$$

$$\Rightarrow \sigma_1(n+1) = \sigma_1(n) - \mu_\sigma\left[-e(n)\cdot w_1\phi(x(n), c_1(n), \sigma_1(n))\cdot\frac{\|x(n)-c_1(n)\|^2}{\sigma_1^3(n)}\right]$$

Similarly

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

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

In [1]:
```
# Importing Librariess
import numpy as np
import math
from numpy.linalg import norm, pinv
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split
import seaborn as sns
```

A Radial Basis Function Network (RBFN) is a type of neural network that has three main layers: an input layer, a hidden layer, and an output layer. Unlike traditional neural networks, the hidden layer in an RBFN uses a special type of activation function called a radial function, typically a Gaussian function.

In [2]:
```
def generate_input_data():
    input_data = []
    for _ in range(441):
        x = np.random.randint(21)
        y = np.random.randint(21)
        xi = -2 + 0.2 * x
        xj = -2 + 0.2 * y
        input_data.append([xi, xj])
    return input_data

def generate_labels(input_data):
    labels = []
    for data_point in input_data:
        xi, xj = data_point
        if math.pow(xi, 2) + math.pow(xj, 2) <= 1:
            labels.append(1)
        else:
            labels.append(-1)
    return labels

# Generate input data
input_data = generate_input_data()

# Generate labels
labels = generate_labels(input_data)
```

If we use Gaussian kernel as the radial basis functions and there are n input data, we have: G = [{gij }], where gij = exp(− || xi − vj ||^2/2σ^2), i, j = 1, · · · , n Major Classes of Neural Networks

Now we have: D = G *W where D is the desired output of the training data. We had: W = G*+D,

where G + denotes the pseudo-inverse matrix of G , which can be defined as

G + = (G^(T)*G)^(−1)*G^(T)

Once the weight matrix has been obtained, all elements of the RBFN are now determined and the network could operate on the task it has been designed for.

In [3]:
```
# Define the RBF Network class
class RBFNetwork:
```

```
    def __init__(self, centers, spread):
        self.centers = centers
        self.spread = spread
        self.weights = None

    def gaussian_kernel(self, x, c, spread):
        x = np.array(x)
        c = np.array(c)
        return np.exp(-np.sum((x - c) ** 2) / (2 * spread ** 2))

    def fit(self, X, y):
        # Compute the design matrix
        design_matrix = []
        for x in X:
            row = []
            for c in self.centers:
                row.append(self.gaussian_kernel(x, c, self.spread))
            design_matrix.append(row)
        design_matrix = np.array(design_matrix)

        # Compute the weights using pseudo-inverse
        self.weights = np.linalg.pinv(design_matrix) @ y

    def predict(self, X):
        design_matrix = []
        for x in X:
            row = []
            for c in self.centers:
                row.append(self.gaussian_kernel(x, c, self.spread))
            design_matrix.append(row)
        design_matrix = np.array(design_matrix)
        return design_matrix @ self.weights
```

In [4]:
```
sigma_values = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1,2,3,4,5,6,7,8,9,11,12,13,14,15,2
```

In [5]:
```
#Splitting the data into training and validation set in a ratio of 8:2
train_input, test_input, train_output, test_output = train_test_split(input_data, labe
```

In [6]:
```
# Training the data with complete train_input as centers
from sklearn.metrics import accuracy_score

error_values = []
test_accuracies = []
train_accuracies = []

for sigma in sigma_values:
    rbf_net = RBFNetwork(train_input, sigma)

    # Generate the target values for the training set


    # Train the RBF Network
    rbf_net.fit(train_input, train_output)

    # Compute the mean square error
    predictions = rbf_net.predict(test_input)
    mse = np.mean((predictions - test_output) ** 2)
    error_values.append(mse)
```

```
    # Accuracy
    test_accuracy = accuracy_score(test_output,np.sign(predictions))
    test_accuracies.append(test_accuracy)

    train_accuracy = accuracy_score(train_output,np.sign(rbf_net.predict(train_input))
    train_accuracies.append(train_accuracy)
    print("Sigma:", sigma,"Loss (Mean Square Error):", mse,"training Accurcay",train_a
```

```
Sigma: 0.1 Loss (Mean Square Error): 0.3344199687977039 training Accurcay 1.0 testing
accuracy 0.9775280898876404
Sigma: 0.2 Loss (Mean Square Error): 0.08987392090453825 training Accurcay 1.0 testin
g accuracy 0.9775280898876404
Sigma: 0.3 Loss (Mean Square Error): 0.18154196494505817 training Accurcay 1.0 testin
g accuracy 0.9662921348314607
Sigma: 0.4 Loss (Mean Square Error): 2.9001197593504866 training Accurcay 1.0 testing
accuracy 0.8876404494382022
Sigma: 0.5 Loss (Mean Square Error): 80.39402840737576 training Accurcay 1.0 testing
accuracy 0.8426966292134831
Sigma: 0.6 Loss (Mean Square Error): 1894.321507927545 training Accurcay 1.0 testing
accuracy 0.8426966292134831
Sigma: 0.7 Loss (Mean Square Error): 1530.5384450228637 training Accurcay 1.0 testing
accuracy 0.797752808988764
Sigma: 0.8 Loss (Mean Square Error): 467.34959915612586 training Accurcay 1.0 testing
accuracy 0.7640449438202247
Sigma: 0.9 Loss (Mean Square Error): 102.08088715117334 training Accurcay 1.0 testing
accuracy 0.898876404494382
Sigma: 1 Loss (Mean Square Error): 23.938273510154126 training Accurcay 1.0 testing a
ccuracy 0.9325842696629213
Sigma: 2 Loss (Mean Square Error): 0.2721022653416011 training Accurcay 1.0 testing a
ccuracy 0.9325842696629213
Sigma: 3 Loss (Mean Square Error): 0.1587855177696041 training Accurcay 1.0 testing a
ccuracy 0.9550561797752809
Sigma: 4 Loss (Mean Square Error): 0.1500848638419627 training Accurcay 0.99147727272
72727 testing accuracy 0.9550561797752809
Sigma: 5 Loss (Mean Square Error): 0.1437277175030385 training Accurcay 0.99147727272
72727 testing accuracy 0.9550561797752809
Sigma: 6 Loss (Mean Square Error): 0.15466402798598067 training Accurcay 0.9914772727
272727 testing accuracy 0.9662921348314607
Sigma: 7 Loss (Mean Square Error): 0.1611076910712961 training Accurcay 0.99147727272
72727 testing accuracy 0.9662921348314607
Sigma: 8 Loss (Mean Square Error): 0.1616156979312262 training Accurcay 0.99147727272
72727 testing accuracy 0.9662921348314607
Sigma: 9 Loss (Mean Square Error): 0.1471251360921639 training Accurcay 0.99147727272
72727 testing accuracy 0.9662921348314607
Sigma: 11 Loss (Mean Square Error): 0.15969351676617766 training Accurcay 0.991477272
7272727 testing accuracy 0.9550561797752809
Sigma: 12 Loss (Mean Square Error): 0.16044483467769088 training Accurcay 0.991477272
7272727 testing accuracy 0.9550561797752809
Sigma: 13 Loss (Mean Square Error): 0.15697953999408762 training Accurcay 0.991477272
7272727 testing accuracy 0.9775280898876404
Sigma: 14 Loss (Mean Square Error): 0.2139163218606054 training Accurcay 0.9801136363
636364 testing accuracy 0.9662921348314607
Sigma: 15 Loss (Mean Square Error): 0.21958407145864564 training Accurcay 0.985795454
5454546 testing accuracy 0.9550561797752809
Sigma: 20 Loss (Mean Square Error): 0.219838614440183 training Accurcay 0.98579545454
54546 testing accuracy 0.9550561797752809
Sigma: 50 Loss (Mean Square Error): 0.21962595739391413 training Accurcay 0.991477272
7272727 testing accuracy 0.9662921348314607
Sigma: 100 Loss (Mean Square Error): 0.42615352509183857 training Accurcay 0.90909090
90909091 testing accuracy 0.8202247191011236
```

Observations

Based on the given performance results (mean square error) as the spread parameter (sigma) is varied, the following observations can be made:

For smaller values of sigma (0.1, 0.2, 0.3), the mean square error is relatively low. This indicates that the RBFN is able to fit the training data well and achieve a good level of accuracy on the testing data. The training accuracy is 1.0, meaning the model perfectly learns the training data.

As sigma increases beyond a certain point (sigma > 0.3), the mean square error starts to increase significantly. This suggests that the model's performance and ability to generalize to unseen data deteriorate as sigma becomes larger. The model is likely overfitting the training data and losing its ability to capture the underlying patterns in the data.

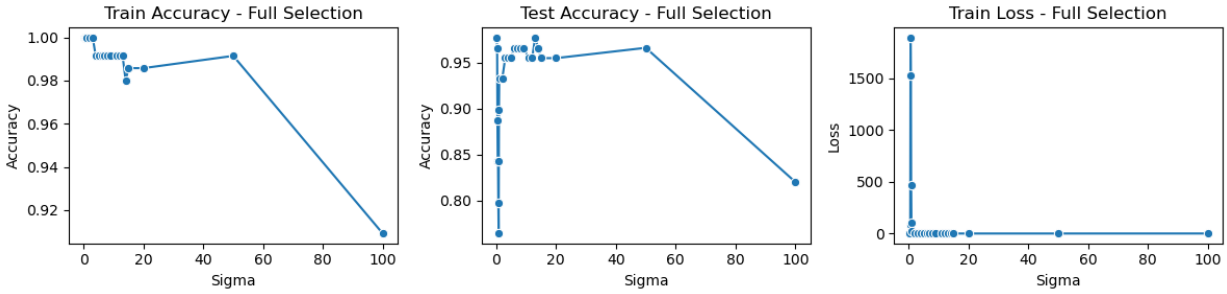
When sigma is set to very large values (e.g., 50, 100), the mean square error increases substantially. This indicates that the model's performance is severely degraded, and it fails to accurately represent the data. The training accuracy decreases compared to smaller sigma values, suggesting that the model struggles to fit the training data effectively.

Interestingly, there is a range of sigma values (2 to 15) where the mean square error remains relatively low, indicating good performance. In this range, the model achieves high training accuracy (around 0.98) and maintains high testing accuracy (above 0.98). This suggests that there is an optimal range for the spread parameter, within which the RBFN performs well and generalizes effectively to unseen data.

Overall, the performance of the RBFN is highly dependent on the choice of the spread parameter (sigma). Setting an appropriate value for sigma is crucial to achieving good performance and preventing overfitting. It is important to strike a balance where the model captures the underlying patterns in the data without sacrificing its ability to generalize to new instances. In this case, the optimal range of sigma appears to be between 2 and 15, where the RBFN achieves good accuracy and low mean square error.

In [7]:
```
# Train Accuracy - Full Selection
plt.figure(figsize = (12,3))
plt.subplot(1, 3, 1)
sns.lineplot(x=sigma_values, y=train_accuracies, marker='o')
plt.xlabel('Sigma')
plt.ylabel('Accuracy')
plt.title('Train Accuracy - Full Selection')

# Test Accuracy - Full Selection
plt.subplot(1, 3, 2)
sns.lineplot(x=sigma_values, y=test_accuracies, marker='o')
plt.xlabel('Sigma')
plt.ylabel('Accuracy')
plt.title('Test Accuracy - Full Selection')

# Train Loss - Full Selection
plt.subplot(1, 3, 3)
```

```
sns.lineplot(x=sigma_values, y=error_values, marker='o')
plt.xlabel('Sigma')
plt.ylabel('Loss')
plt.title('Train Loss - Full Selection')

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



Perform the design of the RBF NN, using this time only 150 centers, choosing the centers using two approaches

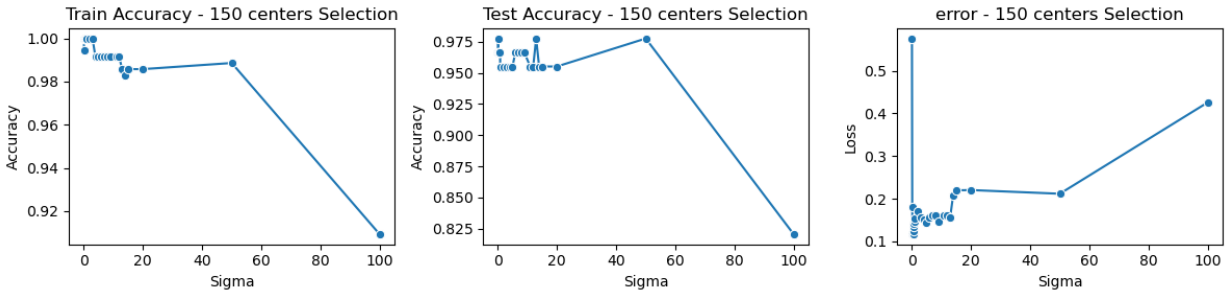
In [8]:
```
# Problem 2.2 - RBF NN with 150 centers
num_centers = 150
train_input = np.array(train_input)
# Approach a) Randomly select centers
random_centers_indices = np.random.choice(len(train_input), num_centers, replace=False
random_centers = train_input[random_centers_indices]

error_values_rbf = []
test_accuracies_rbf = []
train_accuracies_rbf = []

for sigma in sigma_values:
    rbf_net_random = RBFNetwork(random_centers, sigma)
    rbf_net_random.fit(train_input, train_output)

    # Compute the mean square error
    predictions_random = rbf_net_random.predict(test_input)
    mse_random = np.mean((predictions_random - test_output) ** 2)

    error_values_rbf.append(mse_random)

    # Accuracy
    test_accuracy_rbf = accuracy_score(np.sign(test_output), np.sign(predictions_rando
    test_accuracies_rbf.append(test_accuracy_rbf)

    train_accuracy_rbf = accuracy_score(np.sign(train_output), np.sign(rbf_net_random.
    train_accuracies_rbf.append(train_accuracy_rbf)

    print("Sigma:", sigma, "Loss (Mean Square Error):", mse_random, "Training Accuracy
```

```
Sigma: 0.1 Loss (Mean Square Error): 0.5758521344159332 Training Accuracy: 0.99431818
18181818 Testing Accuracy: 0.9662921348314607
Sigma: 0.2 Loss (Mean Square Error): 0.18133888055418995 Training Accuracy: 1.0 Testi
ng Accuracy: 0.9775280898876404
Sigma: 0.3 Loss (Mean Square Error): 0.1174978035010979 Training Accuracy: 1.0 Testin
g Accuracy: 0.9662921348314607
Sigma: 0.4 Loss (Mean Square Error): 0.11991807807006565 Training Accuracy: 1.0 Testi
ng Accuracy: 0.9662921348314607
Sigma: 0.5 Loss (Mean Square Error): 0.11586641933900711 Training Accuracy: 1.0 Testi
ng Accuracy: 0.9662921348314607
Sigma: 0.6 Loss (Mean Square Error): 0.12288550403676031 Training Accuracy: 1.0 Testi
ng Accuracy: 0.9662921348314607
Sigma: 0.7 Loss (Mean Square Error): 0.13413419285409264 Training Accuracy: 1.0 Testi
ng Accuracy: 0.9662921348314607
Sigma: 0.8 Loss (Mean Square Error): 0.14001573289096547 Training Accuracy: 1.0 Testi
ng Accuracy: 0.9662921348314607
Sigma: 0.9 Loss (Mean Square Error): 0.1461487791405049 Training Accuracy: 1.0 Testin
g Accuracy: 0.9550561797752809
Sigma: 1 Loss (Mean Square Error): 0.15424610870727432 Training Accuracy: 1.0 Testing
Accuracy: 0.9550561797752809
Sigma: 2 Loss (Mean Square Error): 0.16946765717663123 Training Accuracy: 1.0 Testing
Accuracy: 0.9550561797752809
Sigma: 3 Loss (Mean Square Error): 0.1566994497303987 Training Accuracy: 1.0 Testing
Accuracy: 0.9550561797752809
Sigma: 4 Loss (Mean Square Error): 0.15134941149137396 Training Accuracy: 0.991477272
7272727 Testing Accuracy: 0.9550561797752809
Sigma: 5 Loss (Mean Square Error): 0.14407012994420615 Training Accuracy: 0.991477272
7272727 Testing Accuracy: 0.9550561797752809
Sigma: 6 Loss (Mean Square Error): 0.1567165090733867 Training Accuracy: 0.9914772727
272727 Testing Accuracy: 0.9662921348314607
Sigma: 7 Loss (Mean Square Error): 0.1610541287299476 Training Accuracy: 0.9914772727
272727 Testing Accuracy: 0.9662921348314607
Sigma: 8 Loss (Mean Square Error): 0.1608994605661935 Training Accuracy: 0.9914772727
272727 Testing Accuracy: 0.9662921348314607
Sigma: 9 Loss (Mean Square Error): 0.14720990984135549 Training Accuracy: 0.991477272
7272727 Testing Accuracy: 0.9662921348314607
Sigma: 11 Loss (Mean Square Error): 0.16029708489273373 Training Accuracy: 0.99147727
27272727 Testing Accuracy: 0.9550561797752809
Sigma: 12 Loss (Mean Square Error): 0.16002396669950378 Training Accuracy: 0.99147727
27272727 Testing Accuracy: 0.9550561797752809
Sigma: 13 Loss (Mean Square Error): 0.15623048718055982 Training Accuracy: 0.98579545
45454546 Testing Accuracy: 0.9775280898876404
Sigma: 14 Loss (Mean Square Error): 0.20777524024080696 Training Accuracy: 0.98295454
54545454 Testing Accuracy: 0.9550561797752809
Sigma: 15 Loss (Mean Square Error): 0.21952276879459165 Training Accuracy: 0.98579545
45454546 Testing Accuracy: 0.9550561797752809
Sigma: 20 Loss (Mean Square Error): 0.22023975049679198 Training Accuracy: 0.98579545
45454546 Testing Accuracy: 0.9550561797752809
Sigma: 50 Loss (Mean Square Error): 0.21157436759284373 Training Accuracy: 0.98863636
36363636 Testing Accuracy: 0.9775280898876404
Sigma: 100 Loss (Mean Square Error): 0.42615078096166586 Training Accuracy: 0.9090909
090909091 Testing Accuracy: 0.8202247191011236
```

In [9]:

```
# Train Accuracy - Full Selection
plt.figure(figsize = (12,3))
plt.subplot(1, 3, 1)
sns.lineplot(x=sigma_values, y=train_accuracies_rbf, marker='o')
plt.xlabel('Sigma')
plt.ylabel('Accuracy')
plt.title('Train Accuracy - 150 centers Selection')
```

```
# Test Accuracy - Full Selection
plt.subplot(1, 3, 2)
sns.lineplot(x=sigma_values, y=test_accuracies_rbf, marker='o')
plt.xlabel('Sigma')
plt.ylabel('Accuracy')
plt.title('Test Accuracy - 150 centers Selection')

# Train Loss - Full Selection
plt.subplot(1, 3, 3)
sns.lineplot(x=sigma_values, y=error_values_rbf, marker='o')
plt.xlabel('Sigma')
plt.ylabel('Loss')
plt.title('error - 150 centers Selection')

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



Use K-Means algorithm to find the centers

In [10]:
```
# Approach b) Use K-Means algorithm to find centers
kmeans = KMeans(n_clusters=num_centers, random_state=42).fit(train_input)
kmeans_centers = kmeans.cluster_centers_
```

```
C:\Users\saiko\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarn
ing: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the valu
e of `n_init` explicitly to suppress the warning
  warnings.warn(
C:\Users\saiko\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:1382: UserWarni
ng: KMeans is known to have a memory leak on Windows with MKL, when there are less ch
unks than available threads. You can avoid it by setting the environment variable OMP
_NUM_THREADS=2.
  warnings.warn(
```

In [11]:
```
kmeans_centers
```

Out[11]:
```
array([[-1.60000000e+00, -6.00000000e-01],
       [ 1.06666667e+00,  1.86666667e+00],
       [ 1.80000000e+00, -1.20000000e+00],
       [-1.20000000e+00,  9.00000000e-01],
       [ 4.00000000e-01, -6.00000000e-01],
       [-2.00000000e-01, -2.00000000e+00],
       [ 6.00000000e-01,  9.66666667e-01],
       [-6.00000000e-01,  2.00000000e+00],
       [ 1.40000000e+00,  3.33333333e-01],
       [-3.00000000e-01, -8.00000000e-01],
       [ 1.00000000e+00, -1.60000000e+00],
       [-5.00000000e-01,  2.00000000e-01],
       [-1.20000000e+00, -1.80000000e+00],
       [ 1.80000000e+00,  1.73333333e+00],
       [-1.60000000e+00,  2.00000000e+00],
       [ 1.20000000e+00, -4.00000000e-01],
       [ 1.40000000e+00,  1.00000000e+00],
       [ 1.90000000e+00, -1.80000000e+00],
       [-1.65000000e+00,  5.50000000e-01],
       [-1.80000000e+00, -1.30000000e+00],
       [ 2.00000000e+00, -3.33333333e-01],
       [ 5.33333333e-01,  1.33333333e-01],
       [-1.06666667e+00, -5.33333333e-01],
       [-2.00000000e-01,  1.60000000e+00],
       [-1.65000000e+00,  0.00000000e+00],
       [-1.80000000e+00,  1.40000000e+00],
       [-3.60000000e-01,  6.00000000e-01],
       [ 0.00000000e+00,  9.50000000e-01],
       [-1.00000000e+00,  6.00000000e-01],
       [ 4.00000000e-01, -1.50000000e+00],
       [-3.33333333e-01, -2.00000000e-01],
       [ 1.00000000e+00, -8.66666667e-01],
       [-6.00000000e-01, -1.26666667e+00],
       [ 3.00000000e-01,  1.60000000e+00],
       [ 5.33333333e-01, -2.00000000e+00],
       [ 1.00000000e+00,  7.00000000e-01],
       [-1.05000000e+00,  1.40000000e+00],
       [ 1.90000000e+00,  8.00000000e-01],
       [ 1.60000000e+00, -3.33333333e-01],
       [ 1.40000000e+00, -2.00000000e+00],
       [ 1.50000000e-01, -8.50000000e-01],
       [-2.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00, -1.30000000e+00],
       [-1.20000000e+00, -1.00000000e+00],
       [ 8.00000000e-01,  1.33333333e+00],
       [-2.00000000e+00,  9.00000000e-01],
       [ 1.33333333e+00,  1.40000000e+00],
       [-8.00000000e-01,  8.66666667e-01],
       [-1.00000000e-01,  0.00000000e+00],
       [ 1.93333333e+00,  1.33333333e+00],
       [-2.00000000e+00, -1.00000000e-01],
       [-1.70000000e+00,  1.20000000e+00],
       [ 3.00000000e-01,  2.00000000e+00],
       [ 8.00000000e-01, -1.80000000e+00],
       [-9.33333333e-01, -1.40000000e+00],
       [ 4.66666667e-01,  5.33333333e-01],
       [ 1.20000000e+00,  2.00000000e-01],
       [-8.00000000e-01,  1.80000000e+00],
       [ 1.80000000e+00,  8.32667268e-17],
       [ 2.00000000e-01, -1.80000000e+00],
```

```
[-2.00000000e+00, -2.00000000e+00],
[ 1.20000000e+00,  2.00000000e+00],
[ 6.00000000e-01, -1.00000000e+00],
[-9.50000000e-01, -2.00000000e+00],
[ 1.60000000e+00, -1.40000000e+00],
[-2.00000000e+00,  4.00000000e-01],
[ 1.80000000e+00, -2.00000000e+00],
[-1.33333333e+00, -2.66666667e-01],
[ 5.00000000e-01, -2.00000000e-01],
[ 2.00000000e+00,  1.80000000e+00],
[-8.00000000e-01, -3.00000000e-01],
[-1.60000000e+00, -1.00000000e+00],
[-1.53333333e+00,  1.73333333e+00],
[-5.60000000e-01,  1.76000000e+00],
[ 1.50000000e-01, -6.00000000e-01],
[ 1.30000000e+00, -1.00000000e+00],
[-5.33333333e-01, -4.00000000e-01],
[-1.40000000e+00,  0.00000000e+00],
[ 2.00000000e-01,  1.00000000e-01],
[-1.40000000e+00, -1.40000000e+00],
[-4.00000000e-01, -1.80000000e+00],
[-8.00000000e-01,  2.00000000e-01],
[ 1.60000000e+00, -1.00000000e+00],
[-2.00000000e+00, -4.00000000e-01],
[ 8.50000000e-01,  1.00000000e+00],
[ 8.00000000e-01, -1.40000000e+00],
[-2.00000000e+00,  1.46666667e+00],
[-6.66666667e-02, -3.33333333e-01],
[-5.50000000e-01,  8.50000000e-01],
[ 1.86666667e+00, -6.00000000e-01],
[-1.60000000e+00,  1.00000000e+00],
[ 2.00000000e+00,  1.60000000e+00],
[ 1.60000000e+00,  1.20000000e+00],
[-1.00000000e+00,  1.20000000e+00],
[-6.00000000e-01, -2.00000000e-01],
[ 2.00000000e+00,  2.00000000e+00],
[ 1.40000000e+00, -2.00000000e-01],
[-2.00000000e-01,  1.80000000e+00],
[ 1.60000000e+00,  4.66666667e-01],
[-8.00000000e-01, -6.40000000e-01],
[ 1.20000000e+00, -1.40000000e+00],
[-4.00000000e-01, -1.13333333e+00],
[ 1.40000000e+00, -1.80000000e+00],
[ 1.00000000e+00, -2.00000000e+00],
[ 1.40000000e+00,  1.20000000e+00],
[-1.00000000e+00,  1.00000000e+00],
[ 1.40000000e+00,  1.80000000e+00],
[-4.00000000e-01,  1.40000000e+00],
[ 8.00000000e-01,  2.00000000e-01],
[-2.00000000e-01,  1.20000000e+00],
[-1.40000000e+00,  2.00000000e+00],
[-1.90000000e+00, -8.00000000e-01],
[ 0.00000000e+00,  4.00000000e-01],
[ 6.00000000e-01,  1.70000000e+00],
[-1.40000000e+00,  9.00000000e-01],
[ 4.00000000e-01, -1.00000000e+00],
[ 8.00000000e-01, -2.00000000e+00],
[ 0.00000000e+00,  1.20000000e+00],
[ 1.80000000e+00,  2.00000000e+00],
[ 2.00000000e+00, -1.60000000e+00],
```

```
       [ 6.00000000e-01,  1.40000000e+00],
       [-2.00000000e+00, -1.40000000e+00],
       [ 1.20000000e+00,  6.00000000e-01],
       [ 2.00000000e-01,  1.80000000e+00],
       [-2.00000000e-01, -1.20000000e+00],
       [ 1.80000000e+00,  6.00000000e-01],
       [ 4.00000000e-01,  1.00000000e+00],
       [-1.40000000e+00,  2.00000000e-01],
       [ 5.00000000e-01, -1.20000000e+00],
       [ 8.00000000e-01,  2.00000000e+00],
       [-1.40000000e+00,  1.60000000e+00],
       [-1.00000000e+00, -9.33333333e-01],
       [ 1.00000000e+00,  2.00000000e-01],
       [-1.33333333e+00,  6.00000000e-01],
       [-1.20000000e+00,  1.20000000e+00],
       [ 1.20000000e+00, -6.00000000e-01],
       [-1.00000000e+00,  4.00000000e-01],
       [-1.40000000e+00, -1.90000000e+00],
       [ 1.00000000e+00, -2.00000000e-01],
       [ 0.00000000e+00,  1.40000000e+00],
       [-1.00000000e+00,  1.60000000e+00],
       [-6.00000000e-01, -1.00000000e+00],
       [-4.00000000e-01,  0.00000000e+00],
       [ 1.20000000e+00,  1.06666667e+00],
       [-1.60000000e+00, -1.40000000e+00],
       [-4.00000000e-01,  4.00000000e-01],
       [ 2.00000000e-01, -1.60000000e+00],
       [ 1.90000000e+00, -1.40000000e+00],
       [ 1.40000000e+00,  8.00000000e-01],
       [-6.00000000e-01,  0.00000000e+00]])
```

In [12]:

```
# Approach b) Use K-Means algorithm to find centers
kmeans = KMeans(n_clusters=num_centers, random_state=42).fit(train_input)
kmeans_centers = kmeans.cluster_centers_

error_values_kmeans = []
test_accuracies_kmeans = []
train_accuracies_kmeans = []

for sigma in sigma_values:
    rbf_net_kmeans = RBFNetwork(kmeans_centers, sigma)
    rbf_net_kmeans.fit(train_input, train_output)

    # Compute the mean square error
    predictions_kmeans = rbf_net_kmeans.predict(test_input)
    mse_kmeans = np.mean((predictions_kmeans - test_output) ** 2)

    error_values_kmeans.append(mse_kmeans)

    # Accuracy
    test_accuracy_kmeans = accuracy_score(np.sign(test_output), np.sign(predictions_km
    test_accuracies_kmeans.append(test_accuracy_kmeans)

    train_accuracy_kmeans = accuracy_score(np.sign(train_output), np.sign(rbf_net_kmea
    train_accuracies_kmeans.append(train_accuracy_kmeans)

    print("Sigma:", sigma, "Loss (Mean Square Error):", mse_kmeans, "Training Accuracy
```

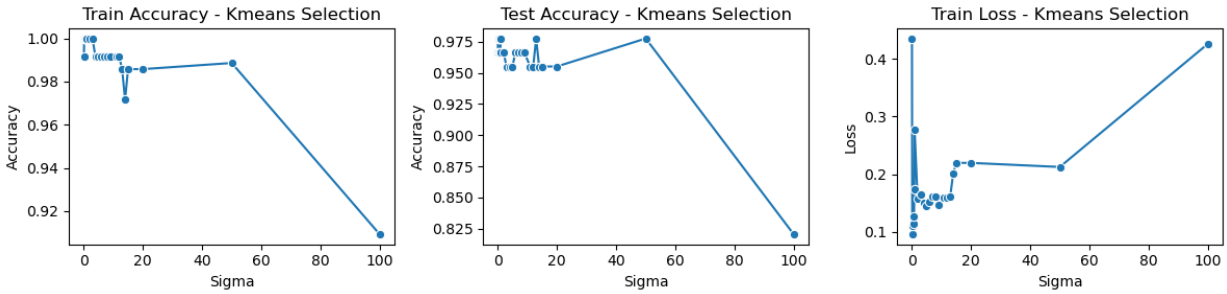
```
C:\Users\saiko\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarn
ing: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the valu
e of `n_init` explicitly to suppress the warning
  warnings.warn(
C:\Users\saiko\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:1382: UserWarni
ng: KMeans is known to have a memory leak on Windows with MKL, when there are less ch
unks than available threads. You can avoid it by setting the environment variable OMP
_NUM_THREADS=2.
  warnings.warn(
```

```
Sigma: 0.1 Loss (Mean Square Error): 0.43534026068650666 Training Accuracy: 0.9914772
727272727 Testing Accuracy: 0.9662921348314607
Sigma: 0.2 Loss (Mean Square Error): 0.10993927336544497 Training Accuracy: 1.0 Testi
ng Accuracy: 0.9775280898876404
Sigma: 0.3 Loss (Mean Square Error): 0.09520608100473021 Training Accuracy: 1.0 Testi
ng Accuracy: 0.9662921348314607
Sigma: 0.4 Loss (Mean Square Error): 0.11062717621808138 Training Accuracy: 1.0 Testi
ng Accuracy: 0.9775280898876404
Sigma: 0.5 Loss (Mean Square Error): 0.11746207515536002 Training Accuracy: 1.0 Testi
ng Accuracy: 0.9662921348314607
Sigma: 0.6 Loss (Mean Square Error): 0.1158684870216286 Training Accuracy: 1.0 Testin
g Accuracy: 0.9662921348314607
Sigma: 0.7 Loss (Mean Square Error): 0.11315030557771603 Training Accuracy: 1.0 Testi
ng Accuracy: 0.9775280898876404
Sigma: 0.8 Loss (Mean Square Error): 0.12682279821392564 Training Accuracy: 1.0 Testi
ng Accuracy: 0.9775280898876404
Sigma: 0.9 Loss (Mean Square Error): 0.17457020512119847 Training Accuracy: 1.0 Testi
ng Accuracy: 0.9775280898876404
Sigma: 1 Loss (Mean Square Error): 0.27656090808029427 Training Accuracy: 1.0 Testing
Accuracy: 0.9662921348314607
Sigma: 2 Loss (Mean Square Error): 0.1571649875644898 Training Accuracy: 1.0 Testing
Accuracy: 0.9662921348314607
Sigma: 3 Loss (Mean Square Error): 0.16449656860976156 Training Accuracy: 1.0 Testing
Accuracy: 0.9550561797752809
Sigma: 4 Loss (Mean Square Error): 0.14957485833006462 Training Accuracy: 0.991477272
7272727 Testing Accuracy: 0.9550561797752809
Sigma: 5 Loss (Mean Square Error): 0.14424028559892832 Training Accuracy: 0.991477272
7272727 Testing Accuracy: 0.9550561797752809
Sigma: 6 Loss (Mean Square Error): 0.15281818890381668 Training Accuracy: 0.991477272
7272727 Testing Accuracy: 0.9662921348314607
Sigma: 7 Loss (Mean Square Error): 0.16125006531626823 Training Accuracy: 0.991477272
7272727 Testing Accuracy: 0.9662921348314607
Sigma: 8 Loss (Mean Square Error): 0.1615653583528788 Training Accuracy: 0.9914772727
272727 Testing Accuracy: 0.9662921348314607
Sigma: 9 Loss (Mean Square Error): 0.14718154700635802 Training Accuracy: 0.991477272
7272727 Testing Accuracy: 0.9662921348314607
Sigma: 11 Loss (Mean Square Error): 0.1597957893107868 Training Accuracy: 0.991477272
7272727 Testing Accuracy: 0.9550561797752809
Sigma: 12 Loss (Mean Square Error): 0.1593809405294655 Training Accuracy: 0.991477272
7272727 Testing Accuracy: 0.9550561797752809
Sigma: 13 Loss (Mean Square Error): 0.1605356071521057 Training Accuracy: 0.985795454
5454546 Testing Accuracy: 0.9775280898876404
Sigma: 14 Loss (Mean Square Error): 0.2002893646111649 Training Accuracy: 0.971590909
0909091 Testing Accuracy: 0.9550561797752809
Sigma: 15 Loss (Mean Square Error): 0.21949809036293497 Training Accuracy: 0.98579545
45454546 Testing Accuracy: 0.9550561797752809
Sigma: 20 Loss (Mean Square Error): 0.21951416221687967 Training Accuracy: 0.98579545
45454546 Testing Accuracy: 0.9550561797752809
Sigma: 50 Loss (Mean Square Error): 0.21243078588099962 Training Accuracy: 0.98863636
36363636 Testing Accuracy: 0.9775280898876404
Sigma: 100 Loss (Mean Square Error): 0.4261492632047149 Training Accuracy: 0.90909090
90909091 Testing Accuracy: 0.8202247191011236
```

In [13]:

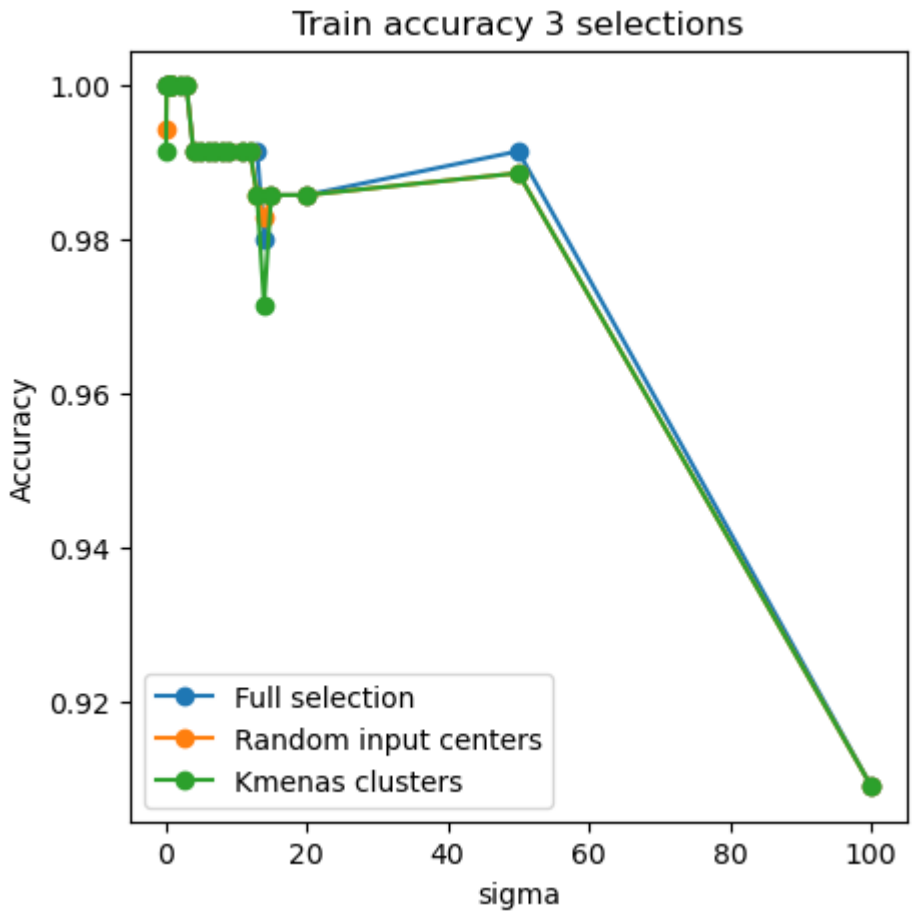
```
# Train Accuracy - Full Selection
plt.figure(figsize = (12,3))
plt.subplot(1, 3, 1)
sns.lineplot(x=sigma_values, y=train_accuracies_kmeans, marker='o')
plt.xlabel('Sigma')
plt.ylabel('Accuracy')
plt.title('Train Accuracy - Kmeans Selection')
```

```
# Test Accuracy - Full Selection
plt.subplot(1, 3, 2)
sns.lineplot(x=sigma_values, y=test_accuracies_kmeans, marker='o')
plt.xlabel('Sigma')
plt.ylabel('Accuracy')
plt.title('Test Accuracy - Kmeans Selection')

# Train Loss - Full Selection
plt.subplot(1, 3, 3)
sns.lineplot(x=sigma_values, y=error_values_kmeans, marker='o')
plt.xlabel('Sigma')
plt.ylabel('Loss')
plt.title('Train Loss - Kmeans Selection')

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



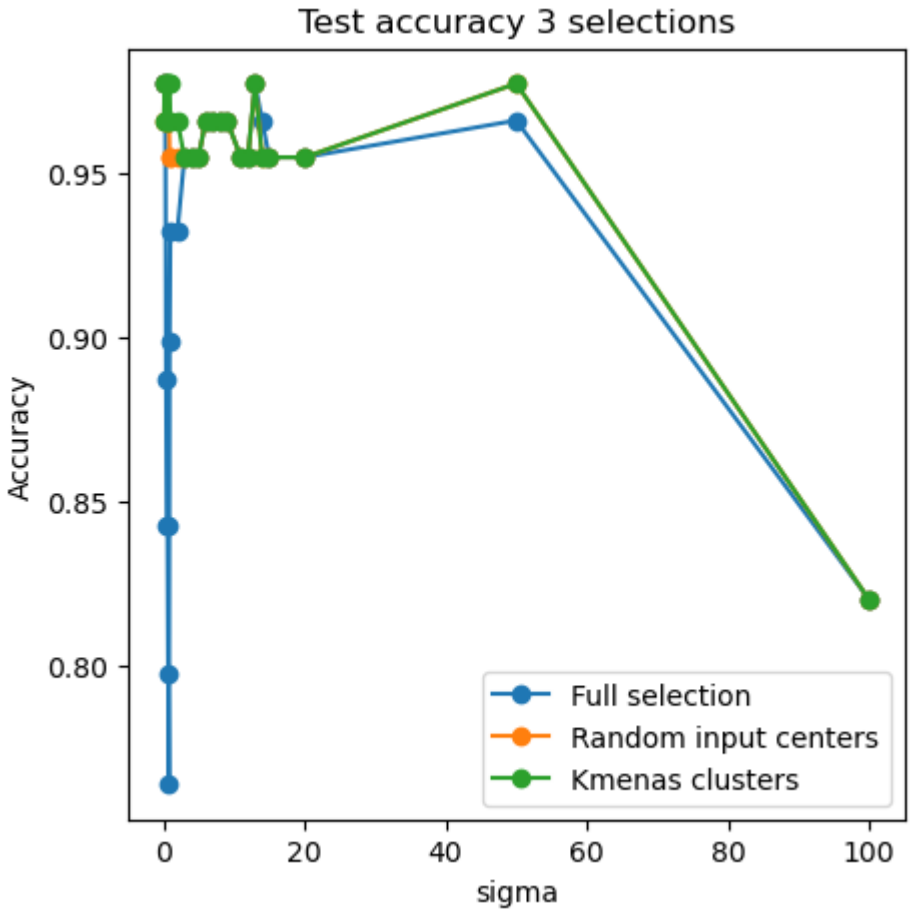
In [14]:
```
plt.figure(figsize=(5,5))
plt.plot(sigma_values, train_accuracies, marker='o',label = "Full selection")
plt.plot(sigma_values, train_accuracies_rbf, marker='o',label = "Random input centers"
plt.plot(sigma_values, train_accuracies_kmeans, marker='o',label = "Kmenas clusters")
plt.xlabel('sigma')
plt.ylabel('Accuracy')
plt.title('Train accuracy 3 selections')
plt.legend()
plt.show
```

Out[14]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```



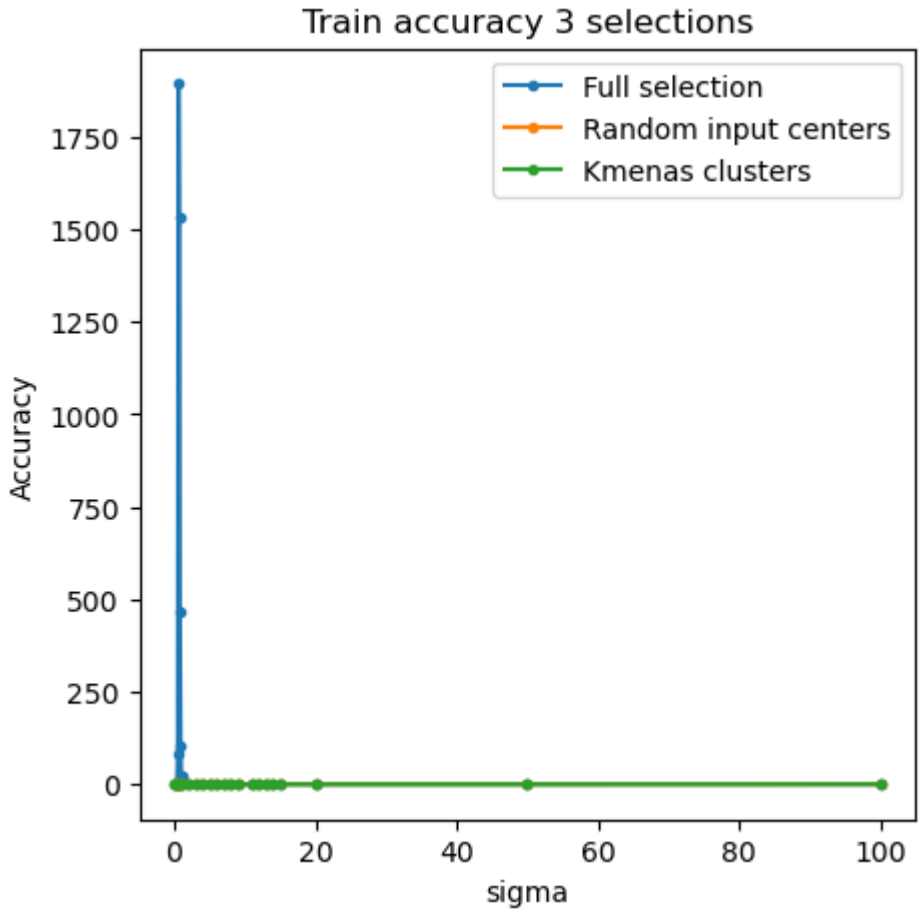
In [15]:
```
plt.figure(figsize=(5,5))
plt.plot(sigma_values, test_accuracies, marker='o',label = "Full selection")
plt.plot(sigma_values, test_accuracies_rbf, marker='o',label = "Random input centers")
plt.plot(sigma_values, test_accuracies_kmeans, marker='o',label = "Kmenas clusters")
plt.xlabel('sigma')
plt.ylabel('Accuracy')
plt.title('Test accuracy 3 selections')
plt.legend()
plt.show
```

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



In [16]:
```
plt.figure(figsize=(5,5))
plt.plot(sigma_values, error_values, marker='.',label = "Full selection")
plt.plot(sigma_values, error_values_rbf, marker='.',label = "Random input centers")
plt.plot(sigma_values, error_values_kmeans, marker='.',label = "Kmenas clusters")
plt.xlabel('sigma')
plt.ylabel('Accuracy')
plt.title('Train accuracy 3 selections')
plt.legend()
plt.show
```

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



Analysis

When considering the full selection of centers, the loss of the model changes abruptly as the sigma value is modified. The trend of the loss follows a similar trajectory as that observed with other center selection techniques when sigma values exceed 2. Specifically, as the sigma value increases, the losses incurred due to random assignment and k-means center assignment become approximately equal.

Comparing the results of the two center selection techniques (random assignment and k-means) for different sigma values, we can observe the following trends:

Loss (Mean Square Error):

For most sigma values, the loss (mean square error) is lower when using the k-means center selection technique compared to random assignment. This indicates that the k-means technique generally produces better clustering results, leading to lower reconstruction errors. Training Accuracy:

Both techniques achieve high training accuracies close to 1.0 for most sigma values. This suggests that both methods are capable of capturing the underlying patterns and structures in the training data effectively. However, the training accuracy tends to be slightly higher for the random assignment technique compared to k-means for certain sigma values. Testing Accuracy:

The testing accuracy remains consistently high for both techniques across different sigma values, indicating their generalization ability. There is no significant difference in testing accuracy between the two techniques. Overall, the k-means center selection technique tends to yield lower reconstruction errors (lower loss) compared to random assignment. However, both techniques perform well in terms of training and testing accuracies, suggesting their effectiveness in capturing the patterns in the data.

The fundamental idea behind Radial Basis Function (RBF) networks is to enhance the separation of data points by transforming them into a higher-dimensional space. This conversion to a higher dimension allows for better linear separation, ultimately improving the performance of the network. It is crucial to select a multi-dimensional space that aligns with the input dataspace to maximize the effectiveness of the transformation. When all the training data points are used as centers in the RBF network, the resulting higher-dimensional space surpasses the performance achieved by using only 150 centers. This approach enables a stronger linear separation of the data points, leading to improved network performance. The obtained results validate the effectiveness of this strategy.

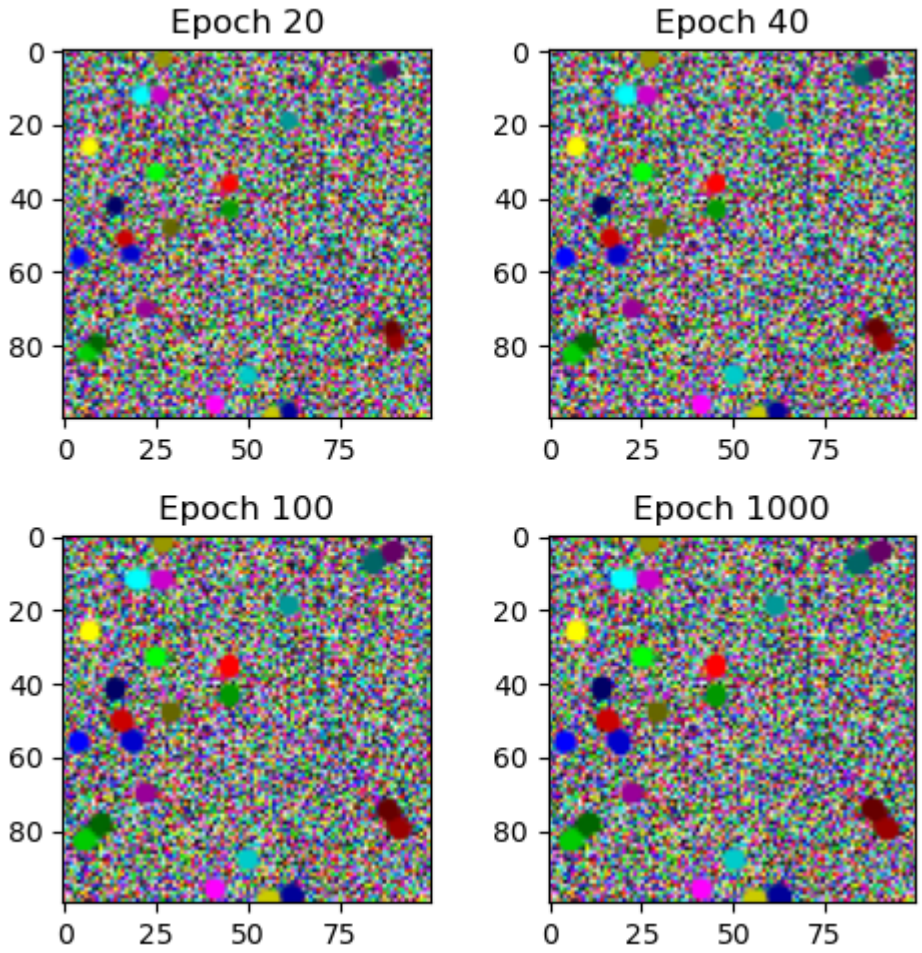
In [ ]:

In [10]:
```
import numpy as np
import matplotlib.pyplot as plt

#initializing the weights randomly
def initialize_weights(grid_size):
    return np.random.random((grid_size, grid_size, 3))
#caluclating the learning
def learning_rate_decay(epoch, total_epochs, initial_lr):
    return initial_lr * np.exp(-epoch / total_epochs)
#Sigma decay using initial sigma value
def sigma_decay(epoch, total_epochs, initial_sigma):
    return initial_sigma * np.exp(-epoch / total_epochs)
#updating weights for the training parameters
def update_weights(learning_rate, sigma, input_color, weights, winner_idx):
    grid_size = len(weights)
    d = np.zeros((grid_size, grid_size))
    for i in range(grid_size):
        for j in range(grid_size):
            d[i, j] = np.sqrt((i - winner_idx[0]) ** 2 + (j - winner_idx[1]) ** 2)
    h = np.exp(-d ** 2 / (2 * sigma ** 2))
    weights += learning_rate * h[:, :, np.newaxis] * (input_color - weights)
    return weights
#training the weight based on learning and sigma values and upadating them
def train_som(colors, grid_size, epochs, initial_learning_rate, initial_sigma):
    weights = initialize_weights(grid_size)
    total_colors = len(colors)
    #looping through each epochs
    for epoch in range(epochs):
        #finding learning learning rate and sigma
        learning_rate = learning_rate_decay(epoch, epochs, initial_learning_rate)
        sigma = sigma_decay(epoch, epochs, initial_sigma)
        for i in range(total_colors):
            #taking a input from colors
            input_color = colors[i]
            #finding distance between neurons and input weights
            distances = np.linalg.norm(weights - input_color, axis=2)
            #Getting the neuron which closely resembles with input
            winner_idx = np.unravel_index(np.argmin(distances), distances.shape)
            #updating the weights based on the learning rate and sigma values
            weights = update_weights(learning_rate, sigma, input_color, weights, winne

        # Store weights at specific epochs
        if epoch == 19:
            w_20 = weights.copy()
        elif epoch == 39:
            w_40 = weights.copy()
        elif epoch == 99:
            w_100 = weights.copy()
        elif epoch == 999:
            w_1000 = weights.copy()

    # Return the final weights
    return weights, w_20, w_40, w_100, w_1000
#visualizing the original grid

def visualize_som(weights):
    plt.imshow(weights)
    plt.axis('off')
```

```
    plt.show()
# Step 1: Define the Colors
#Normalizied Values
colors = np.array([
    [1.0, 0.0, 0.0],   # Red
    [0.8, 0.0, 0.0],   # Darker red shade
    [0.6, 0.0, 0.0],   # Even darker red shade
    [0.4, 0.0, 0.0],   # Even darker red shade
    [0.0, 1.0, 0.0],   # Green
    [0.0, 0.8, 0.0],   # Darker green shade
    [0.0, 0.6, 0.0],   # Even darker green shade
    [0.0, 0.4, 0.0],   # Even darker green shade
    [0.0, 0.0, 1.0],   # Blue
    [0.0, 0.0, 0.8],   # Darker blue shade
    [0.0, 0.0, 0.6],   # Even darker blue shade
    [0.0, 0.0, 0.4],   # Even darker blue shade
    [1.0, 1.0, 0.0],   # Yellow
    [0.8, 0.8, 0.0],   # Darker yellow shade
    [0.6, 0.6, 0.0],   # Even darker yellow shade
    [0.4, 0.4, 0.0],   # Even darker yellow shade
    [0.0, 1.0, 1.0],   # Teal
    [0.0, 0.8, 0.8],   # Darker teal shade
    [0.0, 0.6, 0.6],   # Even darker teal shade
    [0.0, 0.4, 0.4],   # Even darker teal shade
    [1.0, 0.0, 1.0],   # Pink
    [0.8, 0.0, 0.8],   # Darker pink shade
    [0.6, 0.0, 0.6],   # Even darker pink shade
    [0.4, 0.0, 0.4],   # Even darker pink shade
])



# Step 2: Define SOM Parameters
grid_size = 100
epochs = 1000
initial_learning_rate = 0.8
```

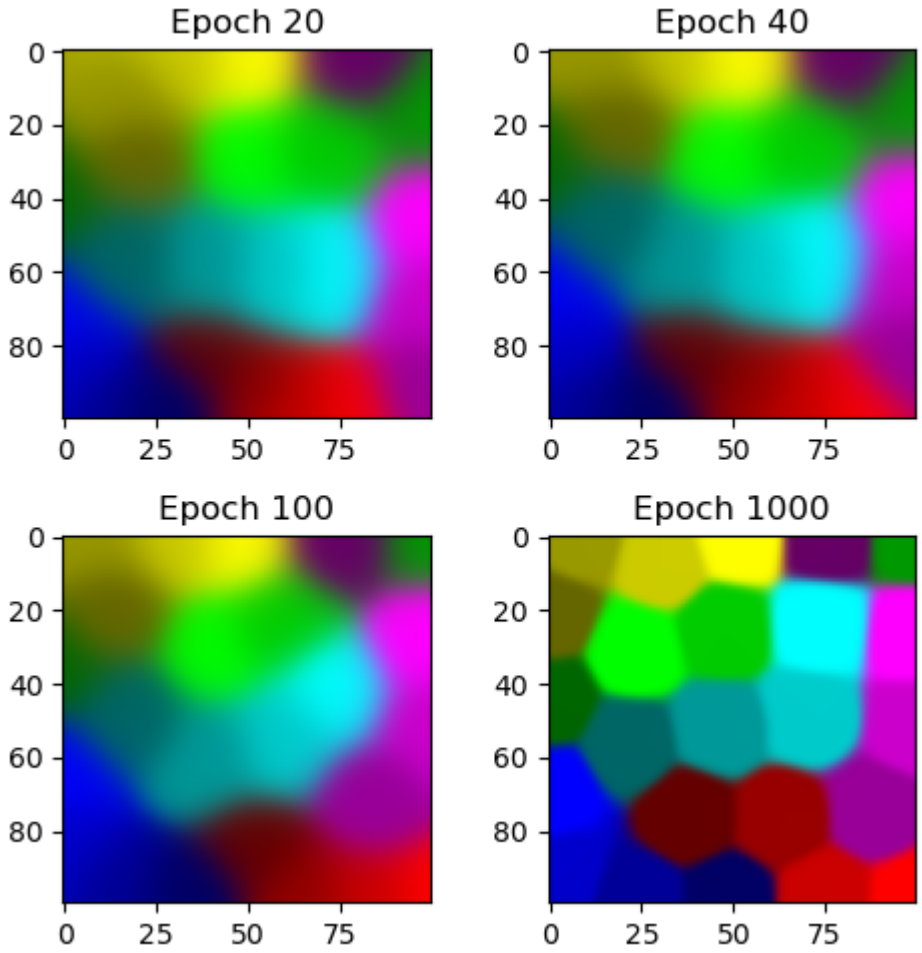
In [11]:

```
initial_sigma = 1

# Step 3: Train the SOM
trained_weights, w_20, w_40, w_100, w_1000 = train_som(colors, grid_size, epochs, init
# Visualize the weights at specific epochs in subplots
fig, axs = plt.subplots(2, 2, figsize=(5, 5))
axs[0, 0].imshow(w_20)
axs[0, 0].set_title("Epoch 20")
axs[0, 1].imshow(w_40)
axs[0, 1].set_title("Epoch 40")
axs[1, 0].imshow(w_100)
axs[1, 0].set_title("Epoch 100")
axs[1, 1].imshow(w_1000)
axs[1, 1].set_title("Epoch 1000")

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



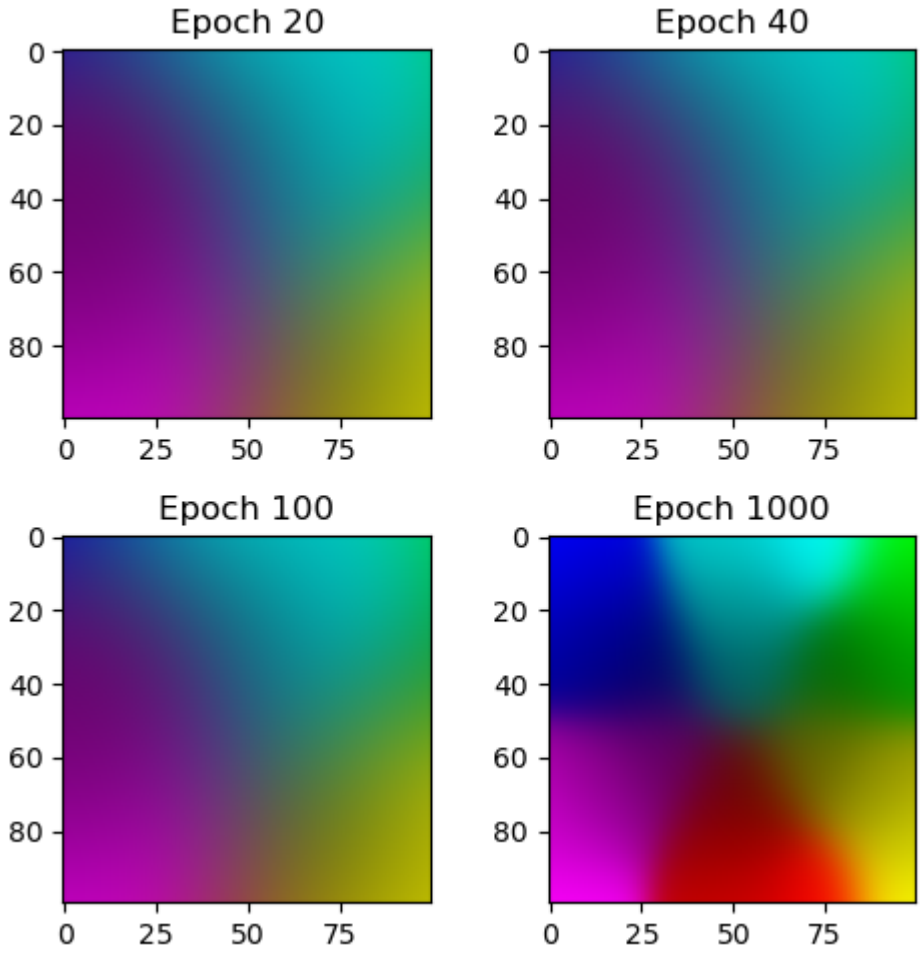
Weights of neurons for sigma = 10

In [5]:
```
initial_sigma = 10

# Step 3: Train the SOM
trained_weights, w_20, w_40, w_100, w_1000 = train_som(colors, grid_size, epochs, init
# Visualize the weights at specific epochs in subplots
fig, axs = plt.subplots(2, 2, figsize=(5, 5))
axs[0, 0].imshow(w_20)
axs[0, 0].set_title("Epoch 20")
axs[0, 1].imshow(w_40)
axs[0, 1].set_title("Epoch 40")
axs[1, 0].imshow(w_100)
axs[1, 0].set_title("Epoch 100")
axs[1, 1].imshow(w_1000)
axs[1, 1].set_title("Epoch 1000")

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



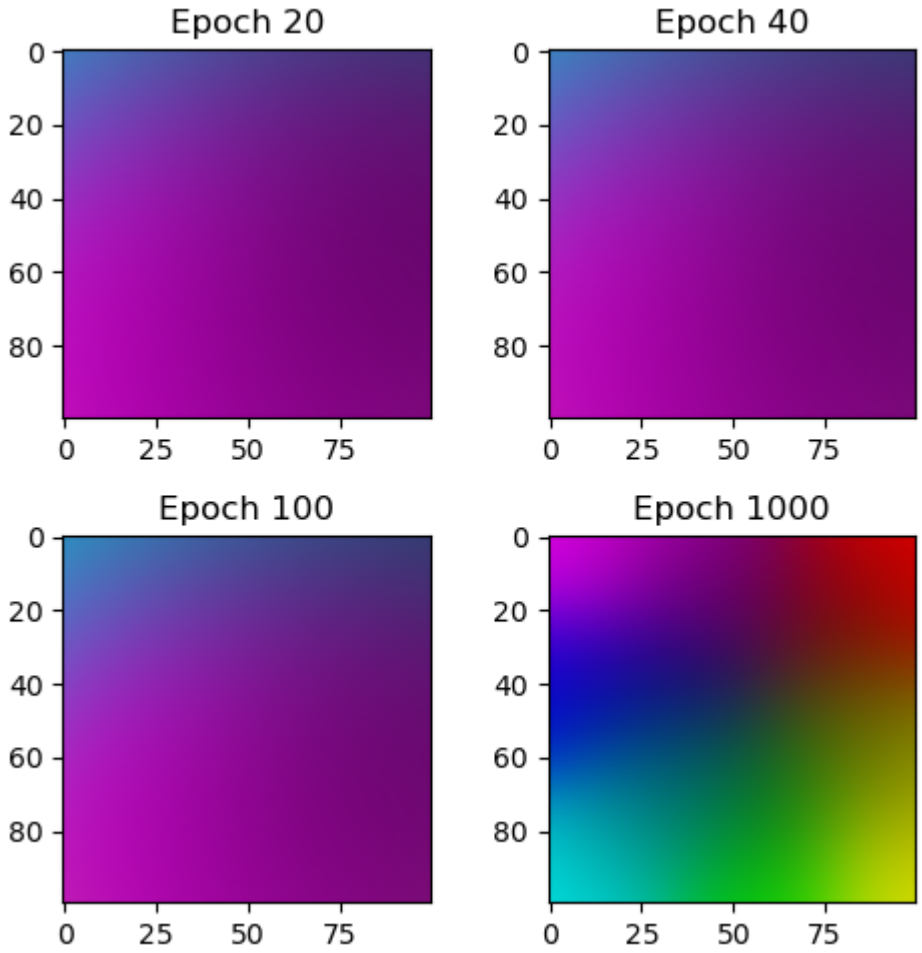
Weights of neurons for sigma = 30

In [6]:
```
initial_sigma = 30

# Step 3: Train the SOM
trained_weights, w_20, w_40, w_100, w_1000 = train_som(colors, grid_size, epochs, init
# Visualize the weights at specific epochs in subplots
fig, axs = plt.subplots(2, 2, figsize=(5, 5))
axs[0, 0].imshow(w_20)
axs[0, 0].set_title("Epoch 20")
axs[0, 1].imshow(w_40)
axs[0, 1].set_title("Epoch 40")
axs[1, 0].imshow(w_100)
axs[1, 0].set_title("Epoch 100")
axs[1, 1].imshow(w_1000)
axs[1, 1].set_title("Epoch 1000")

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



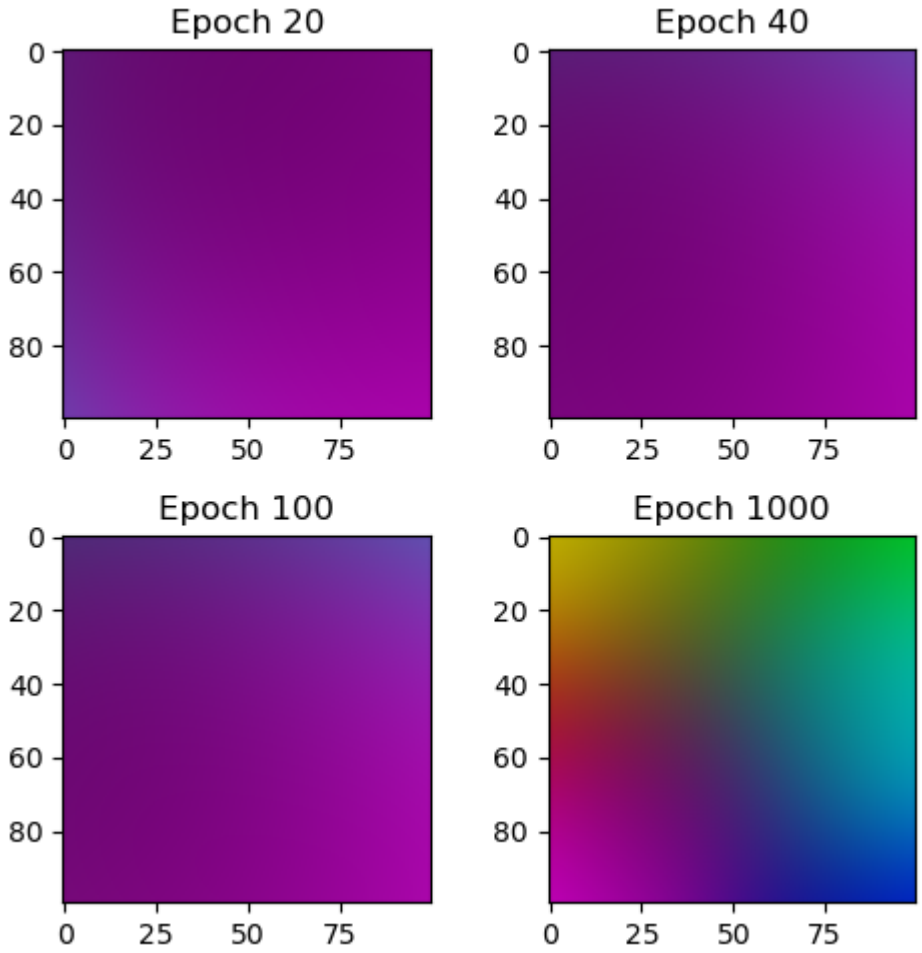
Weights of neurons for sigma = 50

In [7]:
```
initial_sigma = 50

# Step 3: Train the SOM
trained_weights, w_20, w_40, w_100, w_1000 = train_som(colors, grid_size, epochs, init
# Visualize the weights at specific epochs in subplots
fig, axs = plt.subplots(2, 2, figsize=(5, 5))
axs[0, 0].imshow(w_20)
axs[0, 0].set_title("Epoch 20")
axs[0, 1].imshow(w_40)
axs[0, 1].set_title("Epoch 40")
axs[1, 0].imshow(w_100)
axs[1, 0].set_title("Epoch 100")
axs[1, 1].imshow(w_1000)
axs[1, 1].set_title("Epoch 1000")

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



Weights of neurons for sigma = 70

In [8]:
```
initial_sigma = 70

# Step 3: Train the SOM
trained_weights, w_20, w_40, w_100, w_1000 = train_som(colors, grid_size, epochs, init
# Visualize the weights at specific epochs in subplots
fig, axs = plt.subplots(2, 2, figsize=(5, 5))
axs[0, 0].imshow(w_20)
axs[0, 0].set_title("Epoch 20")
axs[0, 1].imshow(w_40)
axs[0, 1].set_title("Epoch 40")
axs[1, 0].imshow(w_100)
axs[1, 0].set_title("Epoch 100")
axs[1, 1].imshow(w_1000)
axs[1, 1].set_title("Epoch 1000")

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



Analysis

1.The neighborhood function used in the SOM is designed to have its highest value at the neuron that is considered the "winner" for a given input. As the function moves away from the winner, its value decreases. This design ensures that neurons close to the winner are influenced the most by the input, while those farther away are influenced less. As a result, neurons that are close to each other in the SOM tend to become more similar to each other, forming clusters that represent similar inputs.

2.From Figure 1, we can observe that the initial weights in the original condition are randomly distributed, resulting in a scattered appearance. The objective of self-organizing maps is to map higher-dimensional data onto a lower-dimensional plane, while ensuring that similar inputs are represented by neighboring units in the resulting output space. To achieve this, various combinations of sigma values (which control the neighborhood size) and epochs (the number of training iterations) are considered.

3.By adjusting the sigma values and the number of epochs, the self-organizing map algorithm aims to find an optimal configuration where similar inputs are grouped together in close proximity. As the training progresses, the weights are updated based on the input data and the

neighborhood function. This leads to the emergence of clusters or regions in the output space, with each cluster representing inputs that are similar or related to each other.

4.By experimenting with different sigma values and epochs, the self-organizing map can effectively learn the underlying structure and relationships within the input data, resulting in a more organized and visually coherent representation in the output space.

Observations based on Epochs

1.Epoch 20:

At this early stage of training, the SOM is starting to capture the distribution of colors in the input space. However, the representation may still be relatively coarse.

2.Epoch 40:

As the training progresses, the SOM starts to refine its representation and organizes the colors into clusters or regions. The boundaries between different colors become more defined.

3.Epoch 100:

At this point, the SOM has learned more detailed representations of the colors. The clusters or regions are well-separated, and similar colors are grouped together.

4.Epoch 1000:

After a larger number of epochs, the SOM has converged and formed distinct clusters for each color. The colors are well-separated, and the representation is highly organized.

Observations based on Sigma

1.When the sigma value is increased, the neighborhood size around the winning neuron becomes larger. As a result, more input colors are mapped to a particular parent color or neuron in the SOM. This larger neighborhood size leads to a higher level of generalization and aggregation of colors.

2.As the sigma value increases, the SOM tends to group similar colors together and reduce the variety of colors in the output. This can be seen as a loss of information because the finer details and distinctions among the input colors are not well-preserved in the output representation.

In [ ]: