Program 6: Implementation of MLP and RBF Network

Objective:

1. Build a BPN (Backpropagation Network) classifier
2. Build an RBF (Radial Basis Function) classifier
3. Compare the results

Dataset: Paper Reviews from UCI Machine Learning Repository

Step 1: Import Required Libraries

In [31]:
```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, classification_report, confus
from scipy.spatial.distance import cdist
import warnings
warnings.filterwarnings('ignore')

print("Libraries imported successfully!")
```

```
Libraries imported successfully!
```

Step 2: Load and Explore the Dataset

The Paper Reviews dataset contains reviews of conference papers with various features.

In [32]:
```
# Load the local reviews.json dataset
import json

print("Loading reviews.json dataset...")
with open('reviews.json', 'r') as f:
    data = json.load(f)

print(f"Dataset loaded successfully!")
print(f"Total number of papers: {len(data['paper'])}")

# Explore the structure
print("\nDataset structure:")
sample_paper = data['paper'][0]
print(f"Sample paper keys: {sample_paper.keys()}")
print(f"Sample review keys: {sample_paper['review'][0].keys()}")
print(f"\nFirst paper ID: {sample_paper['id']}")
print(f"Preliminary decision: {sample_paper['preliminary_decision']}")
print(f"Number of reviews: {len(sample_paper['review'])}")
```

```
Loading reviews.json dataset...
Dataset loaded successfully!
Total number of papers: 172

Dataset structure:
Sample paper keys: dict_keys(['id', 'preliminary_decision', 'review'])
Sample review keys: dict_keys(['confidence', 'evaluation', 'id', 'lan', 'o
rientation', 'remarks', 'text', 'timespan'])

First paper ID: 1
Preliminary decision: accept
Number of reviews: 3
```

In [33]:

```
# Process the JSON data and create a structured dataset
# We'll extract features from reviews and use preliminary_decision as tar

print("Processing JSON data into structured format...")
processed_data = []

for paper in data['paper']:
    paper_id = paper['id']
    decision = paper['preliminary_decision']

    # Aggregate review features for each paper
    reviews = paper['review']

    # Calculate aggregate features from all reviews (handling None values
    confidences = [int(r['confidence']) for r in reviews if r['confidence
    evaluations = [int(r['evaluation']) for r in reviews if r['evaluation
    orientations = [int(r['orientation']) for r in reviews if r['orientat

    # Skip papers with no valid reviews
    if not confidences or not evaluations or not orientations:
        continue

    # Aggregate statistics
    row = {
        'paper_id': paper_id,
        'num_reviews': len(reviews),
        'avg_confidence': np.mean(confidences),
        'max_confidence': np.max(confidences),
        'min_confidence': np.min(confidences),
        'std_confidence': np.std(confidences) if len(confidences) > 1 els
        'avg_evaluation': np.mean(evaluations),
        'max_evaluation': np.max(evaluations),
        'min_evaluation': np.min(evaluations),
        'std_evaluation': np.std(evaluations) if len(evaluations) > 1 els
        'avg_orientation': np.mean(orientations),
        'positive_orientations': sum(1 for o in orientations if o == 1),
        'negative_orientations': sum(1 for o in orientations if o == 0),
        'decision': decision
    }
    processed_data.append(row)

# Create DataFrame
df = pd.DataFrame(processed_data)
```

```
print(f"Processed dataset shape: {df.shape}")
print("\nFirst few rows:")
print(df.head())
print("\nDataset Info:")
print(df.info())
print("\nStatistical Summary:")
print(df.describe())
```

```
Processing JSON data into structured format...
Processed dataset shape: (169, 14)

First few rows:
   paper_id  num_reviews  avg_confidence  max_confidence  min_confidence  
\
0         1            3        4.333333               5               4   
1         2            3        4.000000               4               4   
2         3            3        3.333333               4               3   
3         4            2        3.000000               4               2   
4         5            3        4.333333               5               4   

   std_confidence  avg_evaluation  max_evaluation  min_evaluation  \
0        0.471405        1.000000               1               1   
1        0.000000        2.000000               2               2   
2        0.471405        1.333333               2               0   
3        1.000000        0.000000               2              -2   
4        0.471405        2.000000               2               2   

   std_evaluation  avg_orientation  positive_orientations  \
0        0.000000         0.666667                      2   
1        0.000000         0.333333                      1   
2        0.942809         0.333333                      2   
3        2.000000         0.500000                      0   
4        0.000000         0.666667                      2   

   negative_orientations decision  
0                      1   accept  
1                      2   accept  
2                      0   accept  
3                      0   accept  
4                      1   accept  

Dataset Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 169 entries, 0 to 168
Data columns (total 14 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   paper_id               169 non-null    int64  
 1   num_reviews            169 non-null    int64  
 2   avg_confidence         169 non-null    float64
 3   max_confidence         169 non-null    int64  
 4   min_confidence         169 non-null    int64  
 5   std_confidence         169 non-null    float64
 6   avg_evaluation         169 non-null    float64
 7   max_evaluation         169 non-null    int64  
 8   min_evaluation         169 non-null    int64  
 9   std_evaluation         169 non-null    float64
 10  avg_orientation        169 non-null    float64
 11  positive_orientations  169 non-null    int64  
 12  negative_orientations  169 non-null    int64  
 13  decision               169 non-null    object 
dtypes: float64(5), int64(8), object(1)
memory usage: 18.6+ KB
None

Statistical Summary:
         paper_id  num_reviews  avg_confidence  max_confidence  \
count  169.000000   169.000000      169.000000      169.000000   
```

```
mean    86.751479     2.396450        3.603550        4.000000
std     50.104626     0.780864        0.690816        0.731925
min      1.000000     1.000000        1.000000        1.000000
25%     43.000000     2.000000        3.000000        4.000000
50%     87.000000     2.000000        3.666667        4.000000
75%    130.000000     3.000000        4.000000        4.000000
max    172.000000     4.000000        5.000000        5.000000

       min_confidence  std_confidence  avg_evaluation  max_evaluation  \
count      169.000000      169.000000      169.000000      169.000000
mean         3.189349        0.373741        0.282051        1.017751
std          0.886207        0.363167        1.217093        1.207483
min          1.000000        0.000000       -2.000000       -2.000000
25%          3.000000        0.000000       -0.666667        0.000000
50%          3.000000        0.471405        0.500000        1.000000
75%          4.000000        0.500000        1.333333        2.000000
max          5.000000        1.500000        2.000000        2.000000

       min_evaluation  std_evaluation  avg_orientation  positive_orientati
ons  \
count      169.000000      169.000000       169.000000              169.000
000
mean        -0.420118        0.667995        -0.150888                0.568
047
std          1.462165        0.593200         0.844223                0.704
862
min         -2.000000        0.000000        -2.000000                0.000
000
25%         -2.000000        0.000000        -0.666667                0.000
000
50%         -1.000000        0.500000         0.000000                0.000
000
75%          1.000000        1.000000         0.500000                1.000
000
max          2.000000        2.000000         2.000000                3.000
000

       negative_orientations
count             169.000000
mean                0.692308
std                 0.707107
min                 0.000000
25%                 0.000000
50%                 1.000000
75%                 1.000000
max                 3.000000
```

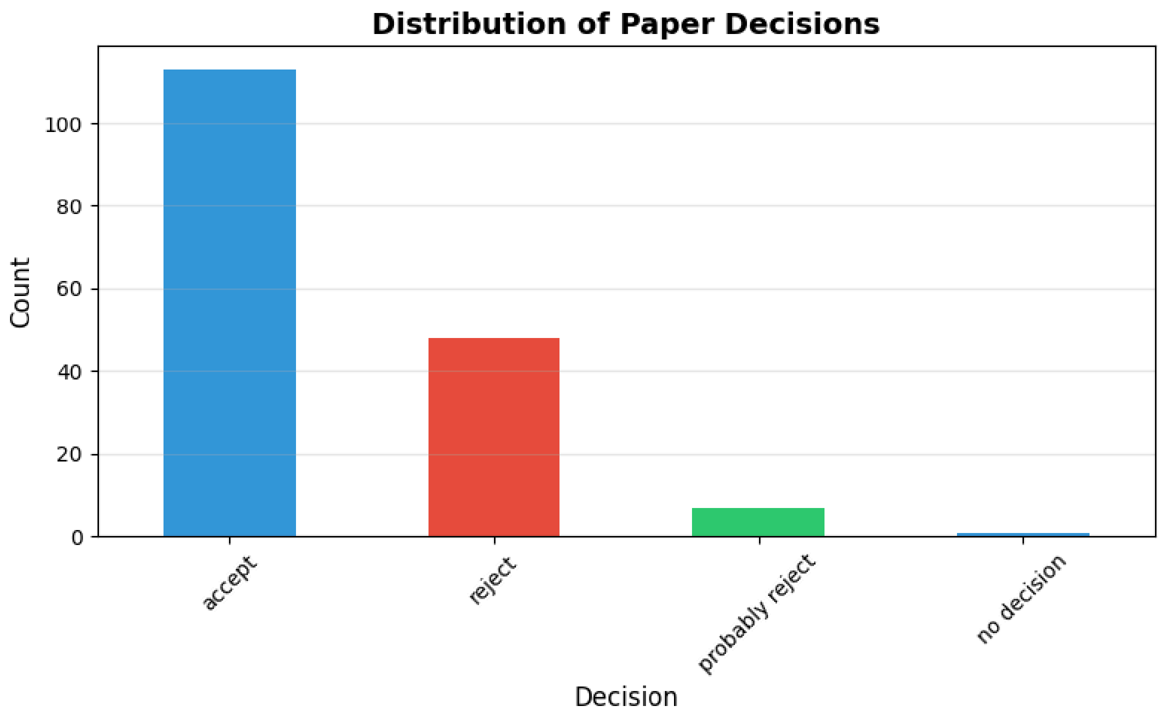
Step 3: Data Preprocessing

We'll check the data quality and prepare it for classification.

In [34]:
```
# Check for missing values
print("Missing Values:")
print(df.isnull().sum())

# Check class distribution (target variable)
print("\nClass Distribution (Preliminary Decision):")
print(df['decision'].value_counts())
```

```
# Visualize class distribution
plt.figure(figsize=(8, 5))
df['decision'].value_counts().plot(kind='bar', color=['#3498db', '#e74c3c
plt.title('Distribution of Paper Decisions', fontsize=14, fontweight='bol
plt.xlabel('Decision', fontsize=12)
plt.ylabel('Count', fontsize=12)
plt.xticks(rotation=45)
plt.grid(axis='y', alpha=0.3)
plt.tight_layout()
plt.show()

print("\nNo missing values - data is clean!")
```

```
Missing Values:
paper_id                 0
num_reviews              0
avg_confidence           0
max_confidence           0
min_confidence           0
std_confidence           0
avg_evaluation           0
max_evaluation           0
min_evaluation           0
std_evaluation           0
avg_orientation          0
positive_orientations    0
negative_orientations    0
decision                 0
dtype: int64

Class Distribution (Preliminary Decision):
decision
accept             113
reject              48
probably reject      7
no decision          1
Name: count, dtype: int64
```



```
No missing values - data is clean!
```

In [35]:
```python
# Separate features and target
# Target: decision (accept, reject, etc.)
# Features: all other columns except paper_id and decision

X = df.drop(columns=['paper_id', 'decision'])
y = df['decision']

print(f"Features shape: {X.shape}")
print(f"Target shape: {y.shape}")
print(f"\nFeature columns:")
print(X.columns.tolist())
print(f"\nTarget classes: {y.unique()}")
```

```
Features shape: (169, 12)
Target shape: (169,)

Feature columns:
['num_reviews', 'avg_confidence', 'max_confidence', 'min_confidence', 'std
_confidence', 'avg_evaluation', 'max_evaluation', 'min_evaluation', 'std_e
valuation', 'avg_orientation', 'positive_orientations', 'negative_orientat
ions']

Target classes: ['accept' 'probably reject' 'reject' 'no decision']
```

In [36]:
```python
# Encode target variable (decision)
print("Encoding target variable...")
le_target = LabelEncoder()
y_encoded = le_target.fit_transform(y)

print(f"Target classes encoded:")
for i, cls in enumerate(le_target.classes_):
    print(f"  {cls} -> {i}")

print(f"\nEncoded target shape: {y_encoded.shape}")
print(f"Class distribution after encoding:")
unique, counts = np.unique(y_encoded, return_counts=True)
for cls, count in zip(unique, counts):
    print(f"  Class {cls}: {count} samples")
```

```
Encoding target variable...
Target classes encoded:
  accept -> 0
  no decision -> 1
  probably reject -> 2
  reject -> 3

Encoded target shape: (169,)
Class distribution after encoding:
  Class 0: 113 samples
  Class 1: 1 samples
  Class 2: 7 samples
  Class 3: 48 samples
```

In [37]:
```python
# Split the data into training and testing sets
# Note: Cannot use stratify due to class with only 1 sample
print("Note: Some classes have very few samples, so stratification is not

X_train, X_test, y_train, y_test = train_test_split(
    X, y_encoded, test_size=0.3, random_state=42
)

print(f"Training set size: {X_train.shape[0]} samples")
print(f"Testing set size: {X_test.shape[0]} samples")
print(f"\nClass distribution in training set:")
unique, counts = np.unique(y_train, return_counts=True)
for cls, count in zip(unique, counts):
    print(f"  Class {cls} ({le_target.classes_[cls]}): {count} samples")
print(f"\nClass distribution in testing set:")
unique, counts = np.unique(y_test, return_counts=True)
for cls, count in zip(unique, counts):
    print(f"  Class {cls} ({le_target.classes_[cls]}): {count} samples")
```

```
Note: Some classes have very few samples, so stratification is not used.
Training set size: 118 samples
Testing set size: 51 samples

Class distribution in training set:
  Class 0 (accept): 77 samples
  Class 1 (no decision): 1 samples
  Class 2 (probably reject): 5 samples
  Class 3 (reject): 35 samples

Class distribution in testing set:
  Class 0 (accept): 36 samples
  Class 2 (probably reject): 2 samples
  Class 3 (reject): 13 samples
```

In [38]:
```
# Normalize the features using StandardScaler
print("Normalizing features...")
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print("Feature normalization completed!")
print(f"\nScaled training data shape: {X_train_scaled.shape}")
print(f"Scaled testing data shape: {X_test_scaled.shape}")
```

```
Normalizing features...
Feature normalization completed!

Scaled training data shape: (118, 12)
Scaled testing data shape: (51, 12)
```

Step 4: Build BPN (Backpropagation Network) Classifier

Multi-Layer Perceptron with single hidden layer using backpropagation algorithm.

In [39]:
```
# Calculate optimal number of hidden neurons
# Rule of thumb: (input_features + output_classes) / 2
n_features = X_train_scaled.shape[1]
n_classes = len(np.unique(y_train))
n_hidden = (n_features + n_classes) // 2

print(f"Number of input features: {n_features}")
print(f"Number of output classes: {n_classes}")
print(f"Number of hidden neurons: {n_hidden}")
```

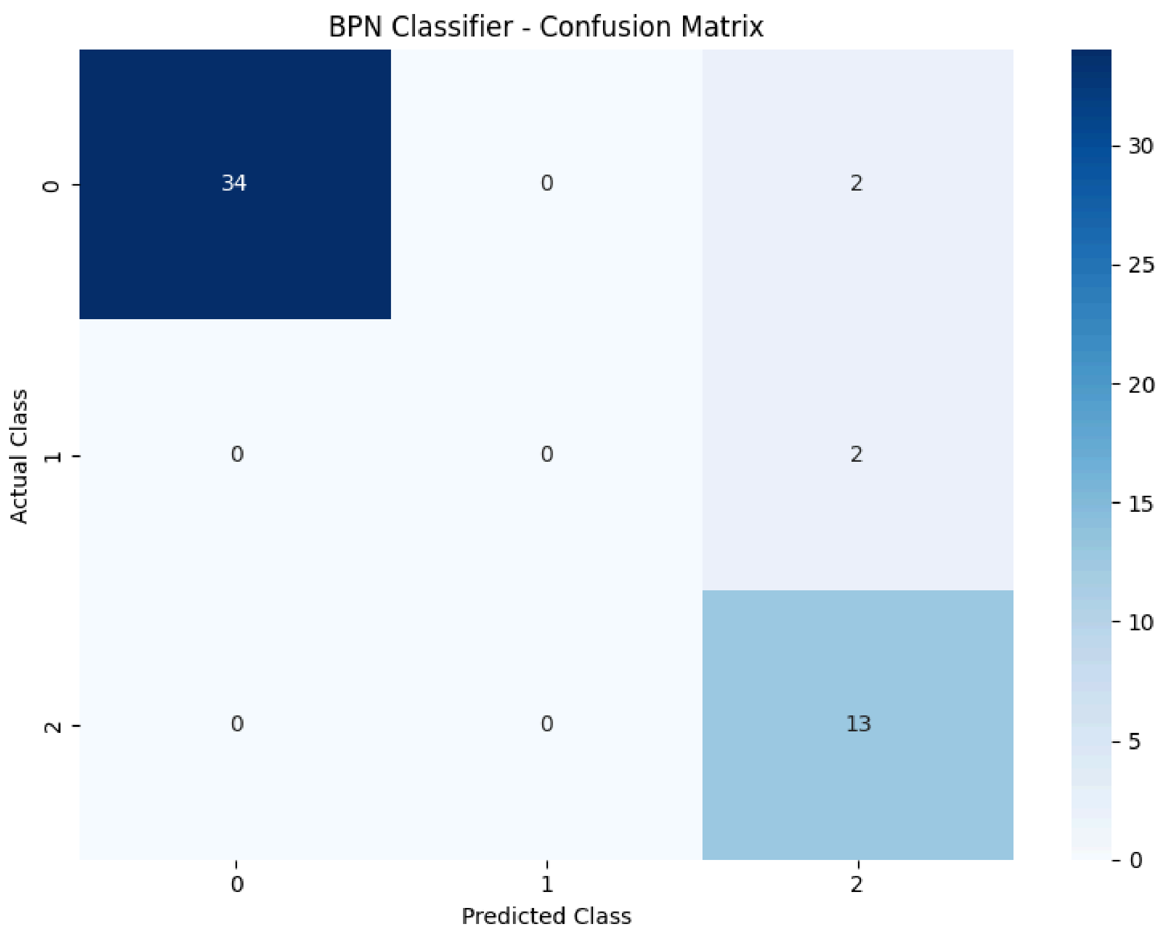
```
# Create MLP classifier with single hidden layer
print("\nBuilding BPN classifier...")
mlp_classifier = MLPClassifier(
    hidden_layer_sizes=(n_hidden,),  # Single hidden layer
    activation='relu',                # Activation function
    solver='adam',                    # Optimizer
    max_iter=500,                     # Maximum iterations
    random_state=42,
    learning_rate_init=0.001,         # Initial learning rate
    verbose=True                      # Show training progress
)

# Train the model
print("\nTraining BPN classifier...")
mlp_classifier.fit(X_train_scaled, y_train)
print("\nBPN training completed!")
```

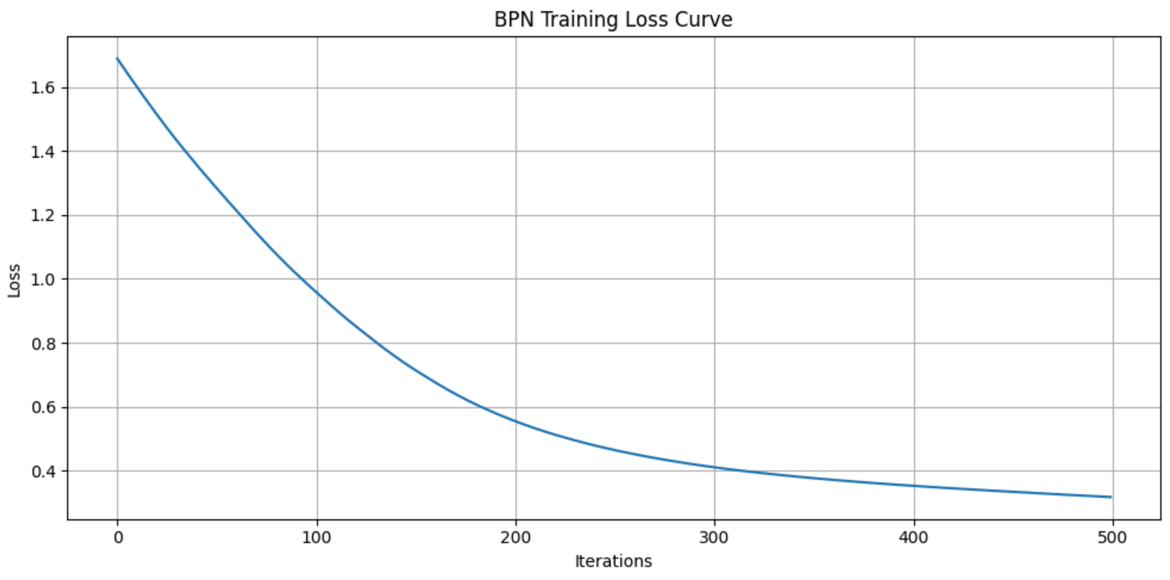
```
Number of input features: 12
Number of output classes: 4
Number of hidden neurons: 8

Building BPN classifier...

Training BPN classifier...
Iteration 1, loss = 1.68926941
Iteration 2, loss = 1.68014081
Iteration 3, loss = 1.67104823
Iteration 4, loss = 1.66198984
Iteration 5, loss = 1.65296993
Iteration 6, loss = 1.64400302
Iteration 7, loss = 1.63508398
Iteration 8, loss = 1.62619638
Iteration 9, loss = 1.61735634
Iteration 10, loss = 1.60855104
Iteration 11, loss = 1.59974674
Iteration 12, loss = 1.59096013
Iteration 13, loss = 1.58216122
Iteration 14, loss = 1.57337766
Iteration 15, loss = 1.56462759
Iteration 16, loss = 1.55593939
Iteration 17, loss = 1.54729866
Iteration 18, loss = 1.53870124
Iteration 19, loss = 1.53014785
Iteration 20, loss = 1.52164234
Iteration 21, loss = 1.51321857
Iteration 22, loss = 1.50486654
Iteration 23, loss = 1.49656072
Iteration 24, loss = 1.48830142
Iteration 25, loss = 1.48009102
Iteration 26, loss = 1.47192865
Iteration 27, loss = 1.46381369
Iteration 28, loss = 1.45574499
Iteration 29, loss = 1.44772199
Iteration 30, loss = 1.43976201
Iteration 31, loss = 1.43191498
Iteration 32, loss = 1.42413880
Iteration 33, loss = 1.41639583
Iteration 34, loss = 1.40870794
Iteration 35, loss = 1.40107836
Iteration 36, loss = 1.39349546
Iteration 37, loss = 1.38592983
Iteration 38, loss = 1.37838040
Iteration 39, loss = 1.37085548
Iteration 40, loss = 1.36334533
Iteration 41, loss = 1.35587342
Iteration 42, loss = 1.34844100
Iteration 43, loss = 1.34106126
Iteration 44, loss = 1.33374173
Iteration 45, loss = 1.32646606
Iteration 46, loss = 1.31923574
Iteration 47, loss = 1.31199984
Iteration 48, loss = 1.30477671
Iteration 49, loss = 1.29756689
Iteration 50, loss = 1.29037170
Iteration 51, loss = 1.28320707
Iteration 52, loss = 1.27608574
Iteration 53, loss = 1.26895395
Iteration 54, loss = 1.26188181
Iteration 55, loss = 1.25483711
Iteration 56, loss = 1.24782547
Iteration 57, loss = 1.24080592
Iteration 58, loss = 1.23377870
Iteration 59, loss = 1.22677856
Iteration 60, loss = 1.21981067
Iteration 61, loss = 1.21286948
Iteration 62, loss = 1.20594125
Iteration 63, loss = 1.19895491
Iteration 64, loss = 1.19193371
Iteration 65, loss = 1.18493101
Iteration 66, loss = 1.17795084
Iteration 67, loss = 1.17096233
Iteration 68, loss = 1.16399506
Iteration 69, loss = 1.15704873
Iteration 70, loss = 1.15012854
Iteration 71, loss = 1.14326265
Iteration 72, loss = 1.13642519
Iteration 73, loss = 1.12962861
Iteration 74, loss = 1.12287206
Iteration 75, loss = 1.11617105
Iteration 76, loss = 1.10952998
Iteration 77, loss = 1.10293052
Iteration 78, loss = 1.09637523
Iteration 79, loss = 1.08989050
Iteration 80, loss = 1.08345652
Iteration 81, loss = 1.07706871
Iteration 82, loss = 1.07072951
Iteration 83, loss = 1.06443686
Iteration 84, loss = 1.05818395
Iteration 85, loss = 1.05197401
Iteration 86, loss = 1.04580980
Iteration 87, loss = 1.03969134
Iteration 88, loss = 1.03360366
Iteration 89, loss = 1.02756017
Iteration 90, loss = 1.02159007
Iteration 91, loss = 1.01566090
Iteration 92, loss = 1.00976339
Iteration 93, loss = 1.00390048
Iteration 94, loss = 0.99807867
Iteration 95, loss = 0.99229721
Iteration 96, loss = 0.98655522
Iteration 97, loss = 0.98085451
Iteration 98, loss = 0.97519882
Iteration 99, loss = 0.96958972
Iteration 100, loss = 0.96402878
Iteration 101, loss = 0.95851381
Iteration 102, loss = 0.95295037
Iteration 103, loss = 0.94729589
Iteration 104, loss = 0.94164121
Iteration 105, loss = 0.93600804
Iteration 106, loss = 0.93040070
Iteration 107, loss = 0.92481371
Iteration 108, loss = 0.91924527
Iteration 109, loss = 0.91370920
Iteration 110, loss = 0.90820670
Iteration 111, loss = 0.90274076
Iteration 112, loss = 0.89732220
Iteration 113, loss = 0.89198434
Iteration 114, loss = 0.88668929
Iteration 115, loss = 0.88144489
Iteration 116, loss = 0.87624859
Iteration 117, loss = 0.87109955
Iteration 118, loss = 0.86600863
Iteration 119, loss = 0.86096395
Iteration 120, loss = 0.85595567
Iteration 121, loss = 0.85097524
Iteration 122, loss = 0.84602899
Iteration 123, loss = 0.84111145
Iteration 124, loss = 0.83617227
Iteration 125, loss = 0.83120047
Iteration 126, loss = 0.82622060
Iteration 127, loss = 0.82127325
Iteration 128, loss = 0.81636002
Iteration 129, loss = 0.81143929
Iteration 130, loss = 0.80655301
Iteration 131, loss = 0.80170932
Iteration 132, loss = 0.79691022
Iteration 133, loss = 0.79215716
Iteration 134, loss = 0.78744713
Iteration 135, loss = 0.78277774
Iteration 136, loss = 0.77814046
Iteration 137, loss = 0.77354258
Iteration 138, loss = 0.76898326
Iteration 139, loss = 0.76445305
Iteration 140, loss = 0.75997382
Iteration 141, loss = 0.75553810
Iteration 142, loss = 0.75114637
Iteration 143, loss = 0.74679902
Iteration 144, loss = 0.74248662
Iteration 145, loss = 0.73821213
Iteration 146, loss = 0.73398386
Iteration 147, loss = 0.72981579
Iteration 148, loss = 0.72569379
Iteration 149, loss = 0.72161770
Iteration 150, loss = 0.71758547
Iteration 151, loss = 0.71358953
Iteration 152, loss = 0.70963649
Iteration 153, loss = 0.70572736
Iteration 154, loss = 0.70185501
Iteration 155, loss = 0.69799205
Iteration 156, loss = 0.69413677
Iteration 157, loss = 0.69027338
Iteration 158, loss = 0.68642455
Iteration 159, loss = 0.68259425
Iteration 160, loss = 0.67879886
Iteration 161, loss = 0.67503657
Iteration 162, loss = 0.67130980
Iteration 163, loss = 0.66762315
Iteration 164, loss = 0.66397483
Iteration 165, loss = 0.66036481
Iteration 166, loss = 0.65680310
Iteration 167, loss = 0.65327295
Iteration 168, loss = 0.64977526
Iteration 169, loss = 0.64631553
Iteration 170, loss = 0.64289164
Iteration 171, loss = 0.63950398
Iteration 172, loss = 0.63615119
Iteration 173, loss = 0.63283190
Iteration 174, loss = 0.62956032
Iteration 175, loss = 0.62632629
Iteration 176, loss = 0.62313070
Iteration 177, loss = 0.61998128
Iteration 178, loss = 0.61687170
Iteration 179, loss = 0.61379963
Iteration 180, loss = 0.61076847
Iteration 181, loss = 0.60777521
Iteration 182, loss = 0.60481798
Iteration 183, loss = 0.60188714
Iteration 184, loss = 0.59899141
Iteration 185, loss = 0.59613137
Iteration 186, loss = 0.59330834
Iteration 187, loss = 0.59052262
Iteration 188, loss = 0.58777169
Iteration 189, loss = 0.58505612
Iteration 190, loss = 0.58236078
Iteration 191, loss = 0.57969504
Iteration 192, loss = 0.57705893
Iteration 193, loss = 0.57445239
Iteration 194, loss = 0.57188265
Iteration 195, loss = 0.56934027
Iteration 196, loss = 0.56681794
Iteration 197, loss = 0.56431652
Iteration 198, loss = 0.56184351
Iteration 199, loss = 0.55939831
Iteration 200, loss = 0.55698094
Iteration 201, loss = 0.55459135
Iteration 202, loss = 0.55222946
Iteration 203, loss = 0.54989515
Iteration 204, loss = 0.54759013
Iteration 205, loss = 0.54531523
Iteration 206, loss = 0.54306871
Iteration 207, loss = 0.54085150
Iteration 208, loss = 0.53865981
Iteration 209, loss = 0.53649742
Iteration 210, loss = 0.53435774
Iteration 211, loss = 0.53224280
Iteration 212, loss = 0.53015239
Iteration 213, loss = 0.52808629
Iteration 214, loss = 0.52604614
Iteration 215, loss = 0.52403500
Iteration 216, loss = 0.52204707
Iteration 217, loss = 0.52007895
Iteration 218, loss = 0.51813269
Iteration 219, loss = 0.51620880
Iteration 220, loss = 0.51431159
Iteration 221, loss = 0.51243023
Iteration 222, loss = 0.51056881
Iteration 223, loss = 0.50872806
Iteration 224, loss = 0.50690766
Iteration 225, loss = 0.50510837
Iteration 226, loss = 0.50332888
Iteration 227, loss = 0.50156059
Iteration 228, loss = 0.49981047
Iteration 229, loss = 0.49807904
Iteration 230, loss = 0.49636626
Iteration 231, loss = 0.49467207
Iteration 232, loss = 0.49299637
Iteration 233, loss = 0.49133469
Iteration 234, loss = 0.48968335
Iteration 235, loss = 0.48805569
Iteration 236, loss = 0.48644771
Iteration 237, loss = 0.48485765
Iteration 238, loss = 0.48328568
Iteration 239, loss = 0.48173007
Iteration 240, loss = 0.48019103
Iteration 241, loss = 0.47866898
Iteration 242, loss = 0.47716117
Iteration 243, loss = 0.47567226
Iteration 244, loss = 0.47419996
Iteration 245, loss = 0.47274435
Iteration 246, loss = 0.47129897
Iteration 247, loss = 0.46987083
Iteration 248, loss = 0.46845821
Iteration 249, loss = 0.46706183
Iteration 250, loss = 0.46567817
Iteration 251, loss = 0.46430760
Iteration 252, loss = 0.46295091
Iteration 253, loss = 0.46160814
Iteration 254, loss = 0.46027919
Iteration 255, loss = 0.45896657
Iteration 256, loss = 0.45766483
Iteration 257, loss = 0.45637746
Iteration 258, loss = 0.45510388
Iteration 259, loss = 0.45384331
Iteration 260, loss = 0.45259557
Iteration 261, loss = 0.45135683
Iteration 262, loss = 0.45013040
Iteration 263, loss = 0.44891619
Iteration 264, loss = 0.44771476
Iteration 265, loss = 0.44652667
Iteration 266, loss = 0.44535180
Iteration 267, loss = 0.44418789
Iteration 268, loss = 0.44303478
Iteration 269, loss = 0.44189287
Iteration 270, loss = 0.44076156
Iteration 271, loss = 0.43964123
Iteration 272, loss = 0.43853278
Iteration 273, loss = 0.43743495
Iteration 274, loss = 0.43634653
Iteration 275, loss = 0.43527036
Iteration 276, loss = 0.43420403
Iteration 277, loss = 0.43314947
Iteration 278, loss = 0.43210410
Iteration 279, loss = 0.43106245
Iteration 280, loss = 0.43003363
Iteration 281, loss = 0.42901692
Iteration 282, loss = 0.42801109
Iteration 283, loss = 0.42701295
Iteration 284, loss = 0.42602503
Iteration 285, loss = 0.42504665
Iteration 286, loss = 0.42407654
Iteration 287, loss = 0.42311469
Iteration 288, loss = 0.42216396
Iteration 289, loss = 0.42122121
Iteration 290, loss = 0.42028240
Iteration 291, loss = 0.41935550
Iteration 292, loss = 0.41843701
Iteration 293, loss = 0.41752808
Iteration 294, loss = 0.41662643
Iteration 295, loss = 0.41573228
Iteration 296, loss = 0.41484641
Iteration 297, loss = 0.41397475
Iteration 298, loss = 0.41310861
Iteration 299, loss = 0.41224785
Iteration 300, loss = 0.41139296
Iteration 301, loss = 0.41054869
Iteration 302, loss = 0.40971389
Iteration 303, loss = 0.40888562
Iteration 304, loss = 0.40806397
Iteration 305, loss = 0.40724881
Iteration 306, loss = 0.40644013
Iteration 307, loss = 0.40563863
Iteration 308, loss = 0.40484684
Iteration 309, loss = 0.40405949
Iteration 310, loss = 0.40328220
Iteration 311, loss = 0.40251385
Iteration 312, loss = 0.40175176
Iteration 313, loss = 0.40099548
Iteration 314, loss = 0.40024500
Iteration 315, loss = 0.39950298
Iteration 316, loss = 0.39876756
Iteration 317, loss = 0.39803580
Iteration 318, loss = 0.39731311
Iteration 319, loss = 0.39659683
Iteration 320, loss = 0.39588551
Iteration 321, loss = 0.39517912
Iteration 322, loss = 0.39447768
Iteration 323, loss = 0.39378227
Iteration 324, loss = 0.39309448
Iteration 325, loss = 0.39241110
Iteration 326, loss = 0.39173484
Iteration 327, loss = 0.39106521
Iteration 328, loss = 0.39040322
Iteration 329, loss = 0.38974417
Iteration 330, loss = 0.38909107
Iteration 331, loss = 0.38844539
Iteration 332, loss = 0.38780400
Iteration 333, loss = 0.38716722
Iteration 334, loss = 0.38653752
Iteration 335, loss = 0.38591122
Iteration 336, loss = 0.38528678
Iteration 337, loss = 0.38466880
Iteration 338, loss = 0.38405451
Iteration 339, loss = 0.38344426
Iteration 340, loss = 0.38283947
Iteration 341, loss = 0.38223851
Iteration 342, loss = 0.38164328
Iteration 343, loss = 0.38104995
Iteration 344, loss = 0.38046385
Iteration 345, loss = 0.37988123
Iteration 346, loss = 0.37930186
Iteration 347, loss = 0.37872577
Iteration 348, loss = 0.37815300
Iteration 349, loss = 0.37758486
Iteration 350, loss = 0.37702156
Iteration 351, loss = 0.37646137
Iteration 352, loss = 0.37590601
Iteration 353, loss = 0.37535428
Iteration 354, loss = 0.37480568
Iteration 355, loss = 0.37426226
Iteration 356, loss = 0.37372177
Iteration 357, loss = 0.37318555
Iteration 358, loss = 0.37265176
Iteration 359, loss = 0.37212205
Iteration 360, loss = 0.37159668
Iteration 361, loss = 0.37107316
Iteration 362, loss = 0.37055379
Iteration 363, loss = 0.37003785
Iteration 364, loss = 0.36952516
Iteration 365, loss = 0.36901580
Iteration 366, loss = 0.36850892
Iteration 367, loss = 0.36800761
Iteration 368, loss = 0.36750654
Iteration 369, loss = 0.36700998
Iteration 370, loss = 0.36651710
Iteration 371, loss = 0.36602642
Iteration 372, loss = 0.36553801
Iteration 373, loss = 0.36505291
Iteration 374, loss = 0.36457124
Iteration 375, loss = 0.36409234
Iteration 376, loss = 0.36361707
Iteration 377, loss = 0.36314484
Iteration 378, loss = 0.36267460
Iteration 379, loss = 0.36220637
Iteration 380, loss = 0.36174019
Iteration 381, loss = 0.36128124
Iteration 382, loss = 0.36082163
Iteration 383, loss = 0.36036085
Iteration 384, loss = 0.35990704
Iteration 385, loss = 0.35945502
Iteration 386, loss = 0.35900496
Iteration 387, loss = 0.35856072
Iteration 388, loss = 0.35811507
Iteration 389, loss = 0.35767267
Iteration 390, loss = 0.35723409
Iteration 391, loss = 0.35679708
Iteration 392, loss = 0.35636169
Iteration 393, loss = 0.35592807
Iteration 394, loss = 0.35549621
Iteration 395, loss = 0.35507195
Iteration 396, loss = 0.35464634
Iteration 397, loss = 0.35421885
Iteration 398, loss = 0.35379871
Iteration 399, loss = 0.35338031
Iteration 400, loss = 0.35296343
Iteration 401, loss = 0.35254821
Iteration 402, loss = 0.35213459
Iteration 403, loss = 0.35172263
Iteration 404, loss = 0.35131227
Iteration 405, loss = 0.35090367
Iteration 406, loss = 0.35049676
Iteration 407, loss = 0.35009154
Iteration 408, loss = 0.34968803
Iteration 409, loss = 0.34929383
Iteration 410, loss = 0.34889644
Iteration 411, loss = 0.34849576
Iteration 412, loss = 0.34810075
Iteration 413, loss = 0.34771001
Iteration 414, loss = 0.34732053
Iteration 415, loss = 0.34693237
Iteration 416, loss = 0.34654559
Iteration 417, loss = 0.34616017
Iteration 418, loss = 0.34577616
Iteration 419, loss = 0.34539356
Iteration 420, loss = 0.34501239
Iteration 421, loss = 0.34463266
Iteration 422, loss = 0.34425439
Iteration 423, loss = 0.34387756
Iteration 424, loss = 0.34351021
Iteration 425, loss = 0.34313886
Iteration 426, loss = 0.34276352
Iteration 427, loss = 0.34239398
Iteration 428, loss = 0.34202840
Iteration 429, loss = 0.34166391
Iteration 430, loss = 0.34130056
Iteration 431, loss = 0.34093832
Iteration 432, loss = 0.34057722
Iteration 433, loss = 0.34021728
Iteration 434, loss = 0.33985852
Iteration 435, loss = 0.33950095
Iteration 436, loss = 0.33914460
Iteration 437, loss = 0.33878946
Iteration 438, loss = 0.33843406
Iteration 439, loss = 0.33807445
Iteration 440, loss = 0.33771920
Iteration 441, loss = 0.33735478
Iteration 442, loss = 0.33699462
Iteration 443, loss = 0.33663480
Iteration 444, loss = 0.33627542
Iteration 445, loss = 0.33591658
Iteration 446, loss = 0.33555833
Iteration 447, loss = 0.33520076
Iteration 448, loss = 0.33484392
Iteration 449, loss = 0.33448785
Iteration 450, loss = 0.33413262
Iteration 451, loss = 0.33377825
Iteration 452, loss = 0.33342478
Iteration 453, loss = 0.33307225
Iteration 454, loss = 0.33272092
Iteration 455, loss = 0.33237062
Iteration 456, loss = 0.33202135
Iteration 457, loss = 0.33167313
Iteration 458, loss = 0.33132597
Iteration 459, loss = 0.33097987
Iteration 460, loss = 0.33063486
Iteration 461, loss = 0.33029092
Iteration 462, loss = 0.32994807
Iteration 463, loss = 0.32960631
Iteration 464, loss = 0.32926564
Iteration 465, loss = 0.32892606
Iteration 466, loss = 0.32858757
Iteration 467, loss = 0.32825016
Iteration 468, loss = 0.32791384
Iteration 469, loss = 0.32757860
Iteration 470, loss = 0.32724449
Iteration 471, loss = 0.32691437
Iteration 472, loss = 0.32658544
Iteration 473, loss = 0.32625763
Iteration 474, loss = 0.32593094
Iteration 475, loss = 0.32560537
Iteration 476, loss = 0.32528097
Iteration 477, loss = 0.32495774
Iteration 478, loss = 0.32463561
Iteration 479, loss = 0.32431457
Iteration 480, loss = 0.32399460
Iteration 481, loss = 0.32367571
Iteration 482, loss = 0.32335788
Iteration 483, loss = 0.32304109
Iteration 484, loss = 0.32272535
Iteration 485, loss = 0.32240712
Iteration 486, loss = 0.32208817
Iteration 487, loss = 0.32176890
Iteration 488, loss = 0.32144943
Iteration 489, loss = 0.32112989
Iteration 490, loss = 0.32081040
Iteration 491, loss = 0.32049138
Iteration 492, loss = 0.32017282
Iteration 493, loss = 0.31985466
Iteration 494, loss = 0.31953694
Iteration 495, loss = 0.31921972
Iteration 496, loss = 0.31890304
Iteration 497, loss = 0.31858693
Iteration 498, loss = 0.31827141
Iteration 499, loss = 0.31795674
Iteration 500, loss = 0.31764363

BPN training completed!
```

In [40]:
```python
# Make predictions on test set
print("Making predictions on test set...")
y_pred_mlp = mlp_classifier.predict(X_test_scaled)

# Calculate accuracy
mlp_accuracy = accuracy_score(y_test, y_pred_mlp)
print(f"\nBPN Classifier Accuracy: {mlp_accuracy * 100:.2f}%")

# Display classification report
print("\nBPN Classification Report:")
print(classification_report(y_test, y_pred_mlp))

# Confusion Matrix
mlp_cm = confusion_matrix(y_test, y_pred_mlp)
print("\nBPN Confusion Matrix:")
print(mlp_cm)
```

```
Making predictions on test set...

BPN Classifier Accuracy: 92.16%

BPN Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.94      0.97        36
           2       0.00      0.00      0.00         2
           3       0.76      1.00      0.87        13

    accuracy                           0.92        51
   macro avg       0.59      0.65      0.61        51
weighted avg       0.90      0.92      0.91        51


BPN Confusion Matrix:
[[34  0  2]
 [ 0  0  2]
 [ 0  0 13]]
```

In [41]:
```python
# Visualize BPN Confusion Matrix
plt.figure(figsize=(8, 6))
sns.heatmap(mlp_cm, annot=True, fmt='d', cmap='Blues', cbar=True)
plt.title('BPN Classifier - Confusion Matrix')
plt.ylabel('Actual Class')
plt.xlabel('Predicted Class')
plt.tight_layout()
plt.show()

# Plot training loss curve
plt.figure(figsize=(10, 5))
plt.plot(mlp_classifier.loss_curve_)
plt.title('BPN Training Loss Curve')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.grid(True)
plt.tight_layout()
plt.show()
```





Step 5: Build RBF (Radial Basis Function) Network Classifier

RBF Network with Gaussian activation functions in the hidden layer.

In [42]:
```python
# Implement RBF Network from scratch
class RBFNetwork:
    """
    Radial Basis Function Network with single hidden layer
    """
    def __init__(self, n_centers=10, gamma=1.0):
        """
        Initialize RBF Network
        n_centers: number of RBF centers (hidden neurons)
        gamma: spread parameter for RBF (controls width of Gaussian)
        """
        self.n_centers = n_centers
        self.gamma = gamma
        self.centers = None
        self.weights = None

    def _gaussian_rbf(self, X, centers):
        """
        Compute Gaussian RBF activations
        """
        # Calculate Euclidean distances from samples to centers
        distances = cdist(X, centers, metric='euclidean')
        # Apply Gaussian function
        return np.exp(-self.gamma * distances**2)

    def fit(self, X, y):
        """
        Train the RBF network
        """
        print(f"Training RBF Network with {self.n_centers} centers...")

        # Step 1: Select RBF centers using random sampling from training
        np.random.seed(42)
        random_indices = np.random.choice(X.shape[0], size=self.n_centers
        self.centers = X[random_indices]

        # Step 2: Compute hidden layer activations
        H = self._gaussian_rbf(X, self.centers)

        # Add bias term
        H = np.column_stack([np.ones(H.shape[0]), H])

        # Step 3: Solve for output weights using pseudo-inverse (least sq
        # Convert y to one-hot encoding if needed
        y_unique = np.unique(y)
        n_classes = len(y_unique)

        if n_classes > 2:
            # Multi-class: one-hot encoding
            y_encoded = np.zeros((y.shape[0], n_classes))
            for i, label in enumerate(y):
                y_encoded[i, label] = 1
        else:
            # Binary: use single output
            y_encoded = y.reshape(-1, 1)

        # Compute weights: W = (H^T H)^(-1) H^T Y
        self.weights = np.linalg.pinv(H) @ y_encoded

        print("RBF Network training completed!")

        return self

    def predict(self, X):
        """
        Make predictions
        """
        # Compute hidden layer activations
        H = self._gaussian_rbf(X, self.centers)

        # Add bias term
        H = np.column_stack([np.ones(H.shape[0]), H])

        # Compute output
        output = H @ self.weights

        # For multi-class, take argmax
        if output.shape[1] > 1:
            predictions = np.argmax(output, axis=1)
        else:
            predictions = (output > 0.5).astype(int).flatten()

        return predictions

print("RBF Network class defined successfully!")
```

```
RBF Network class defined successfully!
```

In [43]:

```
# Create and train RBF classifier
# Number of centers: similar to hidden neurons in MLP
n_rbf_centers = n_hidden

print(f"Creating RBF Network with {n_rbf_centers} centers...")
rbf_classifier = RBFNetwork(n_centers=n_rbf_centers, gamma=0.5)

# Train the RBF network
rbf_classifier.fit(X_train_scaled, y_train)
```

```
Creating RBF Network with 8 centers...
Training RBF Network with 8 centers...
RBF Network training completed!
```

Out[43]:

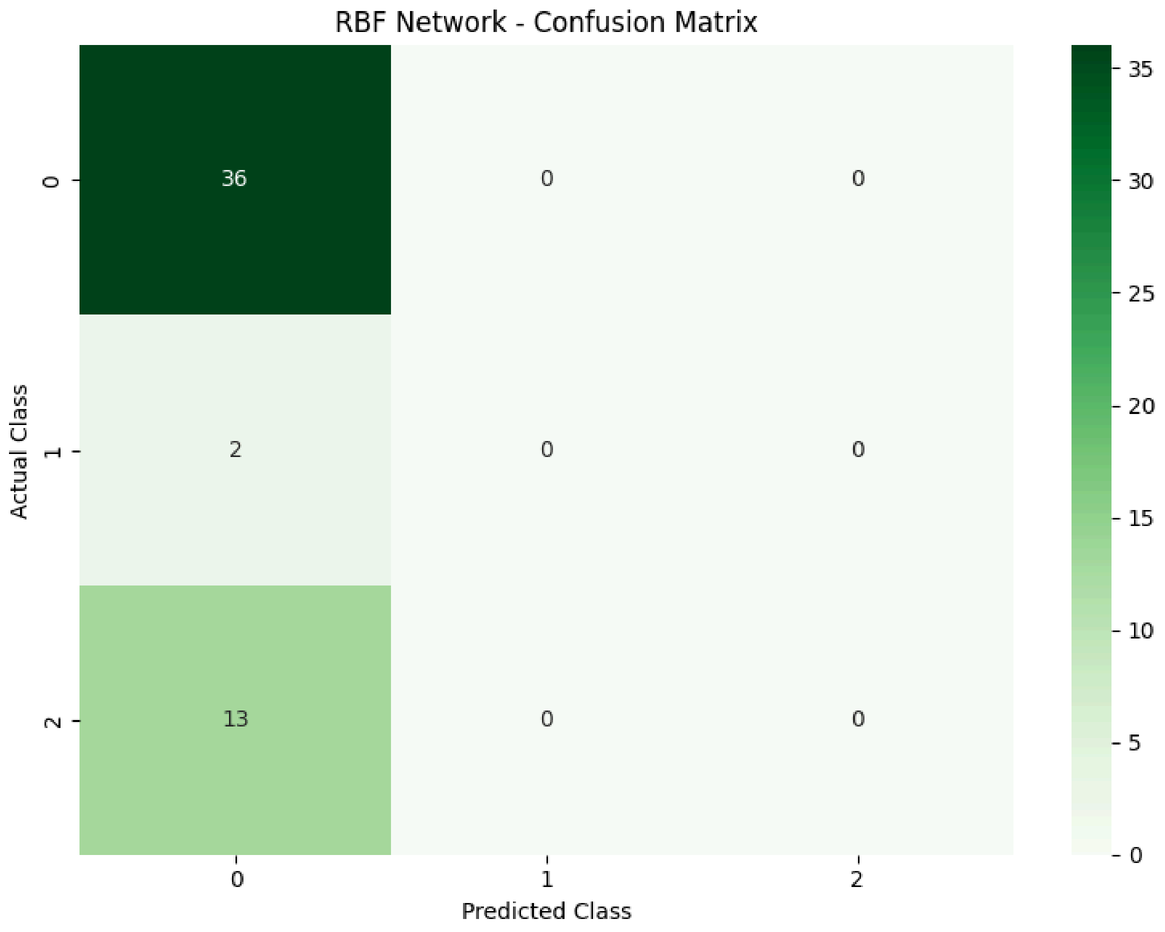
```
<__main__.RBFNetwork at 0x128cf6430>
```

In [44]:

```
# Make predictions on test set
print("Making predictions on test set...")
y_pred_rbf = rbf_classifier.predict(X_test_scaled)

# Calculate accuracy
rbf_accuracy = accuracy_score(y_test, y_pred_rbf)
print(f"\nRBF Classifier Accuracy: {rbf_accuracy * 100:.2f}%")

# Display classification report
print("\nRBF Classification Report:")
print(classification_report(y_test, y_pred_rbf))

# Confusion Matrix
rbf_cm = confusion_matrix(y_test, y_pred_rbf)
print("\nRBF Confusion Matrix:")
print(rbf_cm)
```

```
Making predictions on test set...

RBF Classifier Accuracy: 70.59%

RBF Classification Report:
              precision    recall  f1-score   support

           0       0.71      1.00      0.83        36
           2       0.00      0.00      0.00         2
           3       0.00      0.00      0.00        13

    accuracy                           0.71        51
   macro avg       0.24      0.33      0.28        51
weighted avg       0.50      0.71      0.58        51


RBF Confusion Matrix:
[[36  0  0]
 [ 2  0  0]
 [13  0  0]]
```

In [45]:
```python
# Visualize RBF Confusion Matrix
plt.figure(figsize=(8, 6))
sns.heatmap(rbf_cm, annot=True, fmt='d', cmap='Greens', cbar=True)
plt.title('RBF Network - Confusion Matrix')
plt.ylabel('Actual Class')
plt.xlabel('Predicted Class')
plt.tight_layout()
plt.show()
```



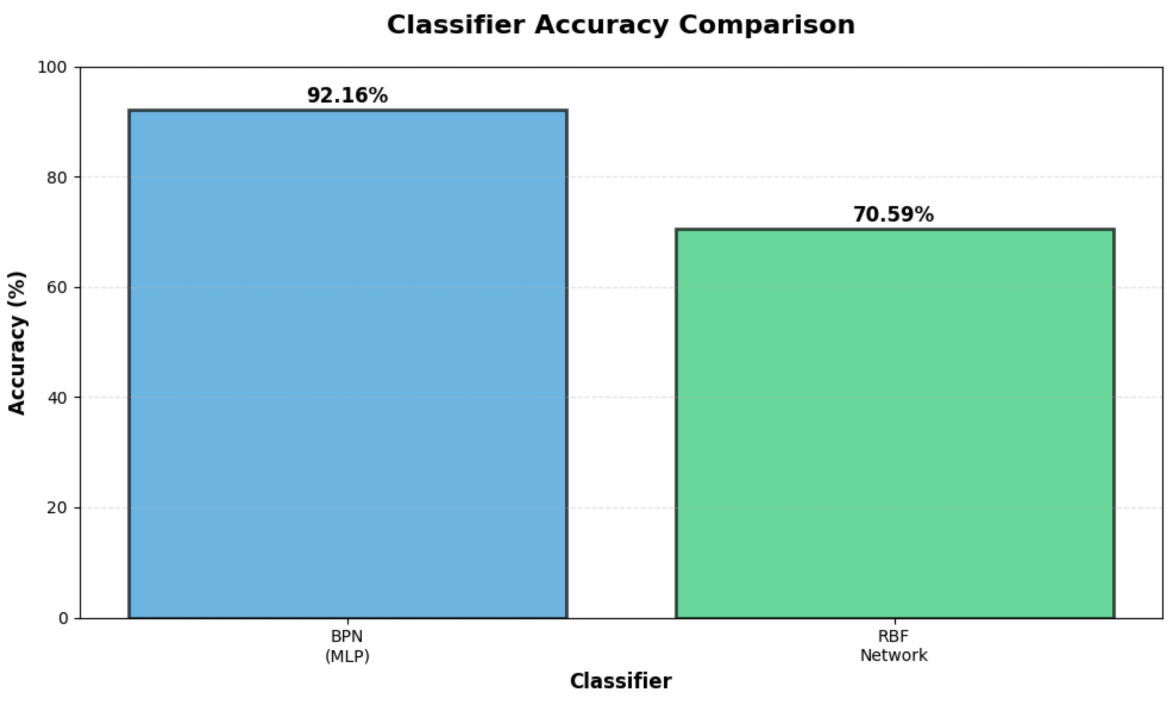
Step 6: Compare BPN and RBF Classifiers

Let's compare the performance of both classifiers.

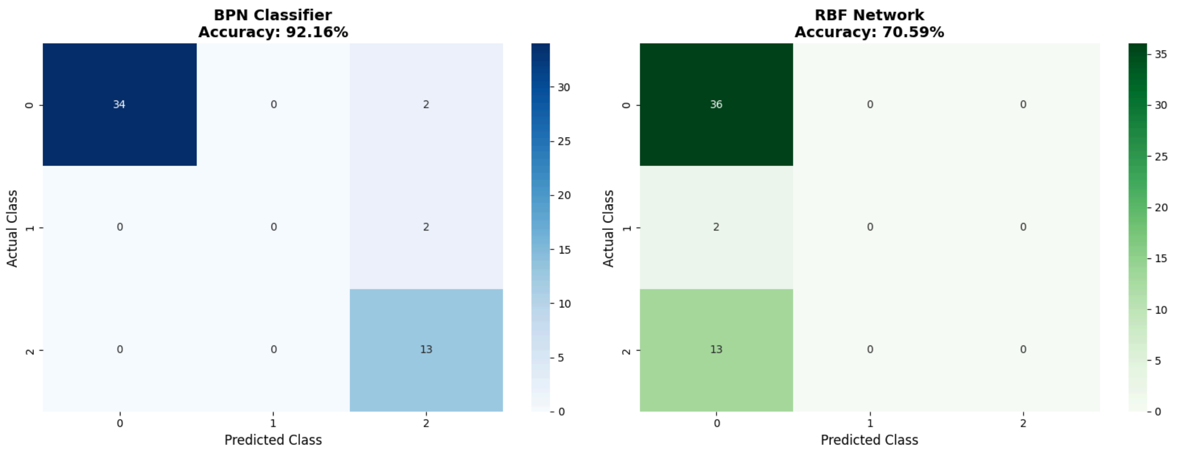
In [46]:
```
# Create comparison dataframe
comparison_df = pd.DataFrame({
    'Classifier': ['BPN (MLP)', 'RBF Network'],
    'Accuracy (%)': [mlp_accuracy * 100, rbf_accuracy * 100],
    'Hidden Units/Centers': [n_hidden, n_rbf_centers]
})

print("=" * 60)
print("COMPARISON OF BPN AND RBF CLASSIFIERS")
print("=" * 60)
print(comparison_df.to_string(index=False))
print("=" * 60)

# Determine which performed better
if mlp_accuracy > rbf_accuracy:
    print(f"\n✓ BPN classifier performed better by {(mlp_accuracy - rbf_a
elif rbf_accuracy > mlp_accuracy:
    print(f"\n✓ RBF classifier performed better by {(rbf_accuracy - mlp_a
else:
    print(f"\n✓ Both classifiers achieved the same accuracy!")
```

```
============================================================
COMPARISON OF BPN AND RBF CLASSIFIERS
============================================================
 Classifier  Accuracy (%)  Hidden Units/Centers
  BPN (MLP)     92.156863                     8
RBF Network     70.588235                     8
============================================================

✓ BPN classifier performed better by 21.57%
```

In [47]:
```
# Visualize accuracy comparison
plt.figure(figsize=(10, 6))
classifiers = ['BPN\n(MLP)', 'RBF\nNetwork']
accuracies = [mlp_accuracy * 100, rbf_accuracy * 100]
colors = ['#3498db', '#2ecc71']

bars = plt.bar(classifiers, accuracies, color=colors, alpha=0.7, edgecolo

# Add value labels on bars
for i, (bar, acc) in enumerate(zip(bars, accuracies)):
    plt.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.5,
             f'{acc:.2f}%', ha='center', va='bottom', fontsize=12, fontwe

plt.title('Classifier Accuracy Comparison', fontsize=16, fontweight='bold
plt.ylabel('Accuracy (%)', fontsize=12, fontweight='bold')
plt.xlabel('Classifier', fontsize=12, fontweight='bold')
plt.ylim([0, 100])
plt.grid(axis='y', alpha=0.3, linestyle='--')
plt.tight_layout()
plt.show()
```



In [48]:
```
# Side-by-side confusion matrix comparison
fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# BPN Confusion Matrix
sns.heatmap(mlp_cm, annot=True, fmt='d', cmap='Blues', cbar=True, ax=axes
axes[0].set_title(f'BPN Classifier\nAccuracy: {mlp_accuracy * 100:.2f}%',
axes[0].set_ylabel('Actual Class', fontsize=12)
axes[0].set_xlabel('Predicted Class', fontsize=12)

# RBF Confusion Matrix
sns.heatmap(rbf_cm, annot=True, fmt='d', cmap='Greens', cbar=True, ax=axe
axes[1].set_title(f'RBF Network\nAccuracy: {rbf_accuracy * 100:.2f}%', fo
axes[1].set_ylabel('Actual Class', fontsize=12)
axes[1].set_xlabel('Predicted Class', fontsize=12)

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


BPN Classifier
Accuracy: 92.16%
Actual Class
Predicted Class
RBF Network
Accuracy: 70.59%
Actual Class
Predicted Class