In [1]:

```python
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Spectral library for handling ENVI files
from spectral import open_image

# Data preprocessing and PCA
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix

# Deep learning libraries
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, Dense, Flatten, Dropout, BatchNormalization, MaxPooling1D, InputLayer
from tensorflow.keras.utils import to_categorical

# For reproducibility
np.random.seed(42)
tf.random.set_seed(42)
```

In [2]:

```python
# Update the paths according to your local environment
image_hdr_path = r"C:\Users\Shewak Heera\Desktop\dataset\image.hdr"
gt_hdr_path    = r"C:\Users\Shewak Heera\Desktop\dataset\gt.hdr"

# Load hyperspectral image (ENVI format)
img = open_image(image_hdr_path)
data = img.load().astype(np.float32)  # shape: (rows, cols, bands)
print("Hyperspectral image shape:", data.shape)

# Load ground truth image (assuming ENVI format for gt)
gt = open_image(gt_hdr_path).load().astype(np.int32)
print("Ground truth image shape:", gt.shape)
```

```
Hyperspectral image shape: (340, 650, 308)
Ground truth image shape: (340, 650, 1)
```

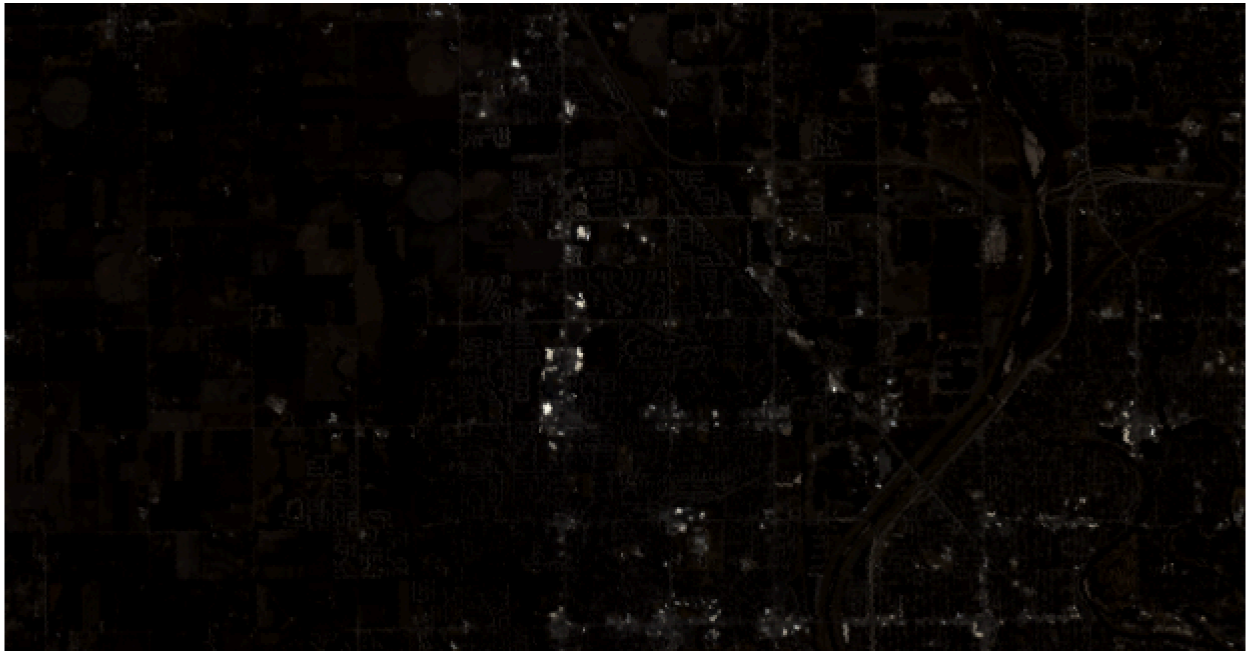
In [3]:
```python
# Display some statistics for a few bands
n_bands = data.shape[2]
print("Total number of bands:", n_bands)

# Example: print statistics for the first 3 bands
for i in range(3):
    band = data[:, :, i]
    print(f"Band {i+1}: min={np.min(band):.2f}, max={np.max(band):.2f}, mean={np.mean(band):.2f}, std={np.std(band):.2f}")
```

```
Total number of bands: 308
Band 1: min=-1662.00, max=6184.00, mean=-264.08, std=452.46
Band 2: min=-1483.00, max=5956.00, mean=-160.96, std=413.82
Band 3: min=-722.00, max=5911.00, mean=71.88, std=351.82
```

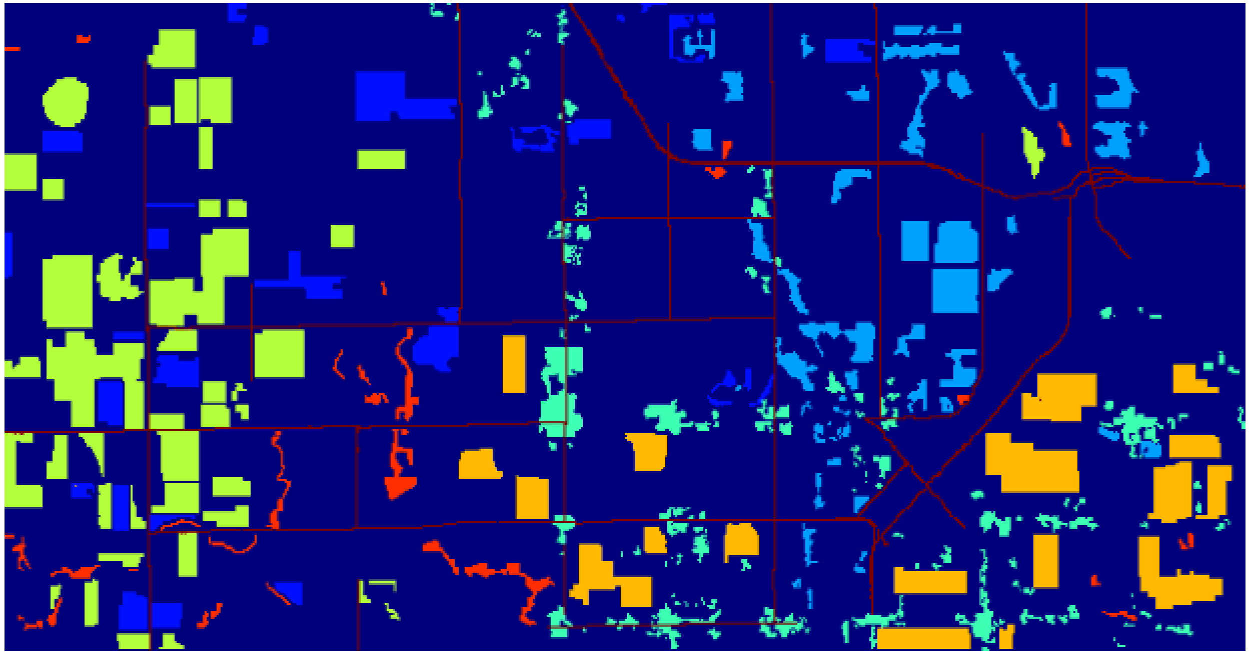
In [4]:
```python
# Plot a composite of three selected bands (for example, bands 29, 19, 9)
# Adjust these band indices based on the dataset specifics.
band_r = data[:, :, 28]  # using 0-indexing: band 29
band_g = data[:, :, 18]  # band 19
band_b = data[:, :, 8]   # band 9

composite = np.dstack((band_r, band_g, band_b))
plt.figure(figsize=(8, 6))
plt.imshow(composite / np.max(composite))
plt.title("Pseudo-RGB Composite")
plt.axis('off')
plt.show()

# Plot the ground truth image (provided as a pseudo-color image)
plt.figure(figsize=(8, 6))
plt.imshow(gt, cmap='jet')
plt.title("Ground Truth Classification")
plt.axis('off')
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). Got range [-0.062
01719..1.0].
```

Pseudo-RGB Composite



Ground Truth Classification



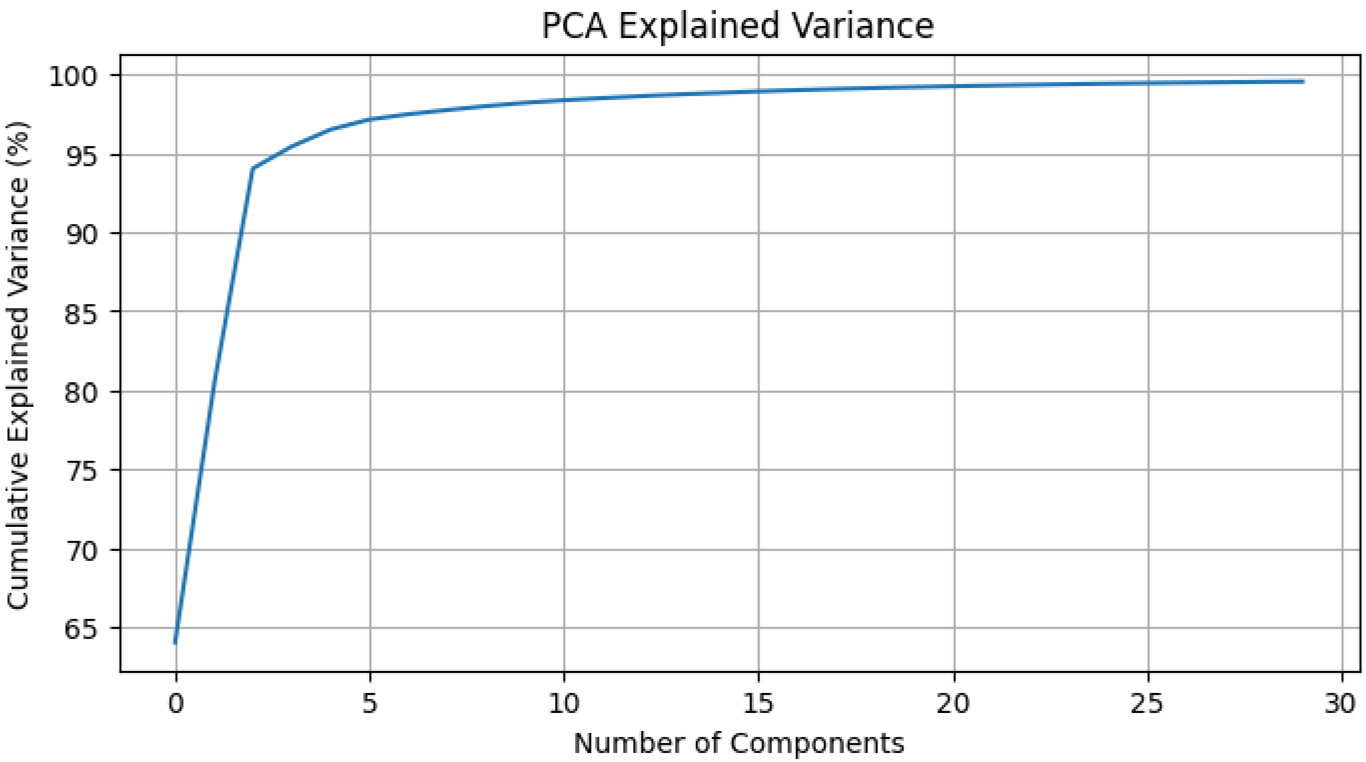
In [5]:
```
# Get the spatial dimensions and number of bands
n_rows, n_cols, n_bands = data.shape

# Reshape data to (n_pixels, n_bands)
data_reshaped = data.reshape(-1, n_bands)

# Normalize the data
scaler = StandardScaler()
data_norm = scaler.fit_transform(data_reshaped)
print("Data normalized. Shape:", data_norm.shape)
```

```
Data normalized. Shape: (221000, 308)
```

In [6]:
```
# Set number of PCA components (e.g., 30 for a good trade-off)
n_components = 30
pca = PCA(n_components=n_components)
data_pca = pca.fit_transform(data_norm)
print("PCA output shape:", data_pca.shape)
```

```python
# Explained variance plot
plt.figure(figsize=(8,4))
plt.plot(np.cumsum(pca.explained_variance_ratio_)*100)
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance (%)')
plt.title('PCA Explained Variance')
plt.grid(True)
plt.show()

# Reshape PCA output back to image dimensions
data_pca_image = data_pca.reshape(n_rows, n_cols, n_components)
```

```
PCA output shape: (221000, 30)
```



In [7]:

```python
# Create a mask: assume ground truth labels > 0 indicate labeled pixels
mask = gt > 0
```

```python
# Extract training samples (only from labeled pixels)
X = data_pca[mask.flatten()]   # shape: (n_samples, n_components)
y = gt[mask]                   # labels

# If ground truth labels are 1-indexed, shift them to start at 0
y = y - 1
print("Training samples:", X.shape, "Labels shape:", y.shape)

# Define number of classes (from the PDF: 7 classes)
num_classes = 7

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)
print("Train samples:", X_train.shape, "Test samples:", X_test.shape)
```

```
Training samples: (41953, 30) Labels shape: (41953,)
Train samples: (29367, 30) Test samples: (12586, 30)
```

In [8]:

```python
# Reshape input data for CNN: (samples, n_components, 1)
input_dim = X_train.shape[1]
X_train_cnn = X_train.reshape(-1, input_dim, 1)
X_test_cnn  = X_test.reshape(-1, input_dim, 1)

# One-hot encode labels
y_train_cat = to_categorical(y_train, num_classes=num_classes)
y_test_cat  = to_categorical(y_test, num_classes=num_classes)

# Build the CNN model
model = Sequential([
    InputLayer(input_shape=(input_dim, 1)),
    Conv1D(64, kernel_size=3, activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),
    Dropout(0.3),
    Conv1D(128, kernel_size=3, activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),
    Dropout(0.3),
    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
```

```
    Dense(num_classes, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

```
C:\Users\Shewak Heera\AppData\Local\Programs\Python\Python310\lib\site-packages\keras\src\layers\core\input_layer.py:27: UserWarning: Argument `input_shape` is deprecated. Use `shape` instead.
  warnings.warn(
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 30, 64)	256
batch_normalization (BatchNormalization)	(None, 30, 64)	256
max_pooling1d (MaxPooling1D)	(None, 15, 64)	0
dropout (Dropout)	(None, 15, 64)	0
conv1d_1 (Conv1D)	(None, 15, 128)	24,704
batch_normalization_1 (BatchNormalization)	(None, 15, 128)	512
max_pooling1d_1 (MaxPooling1D)	(None, 7, 128)	0
dropout_1 (Dropout)	(None, 7, 128)	0
flatten (Flatten)	(None, 896)	0
dense (Dense)	(None, 256)	229,632
dropout_2 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 7)	1,799

Total params: 257,159 (1004.53 KB)

Trainable params: 256,775 (1003.03 KB)

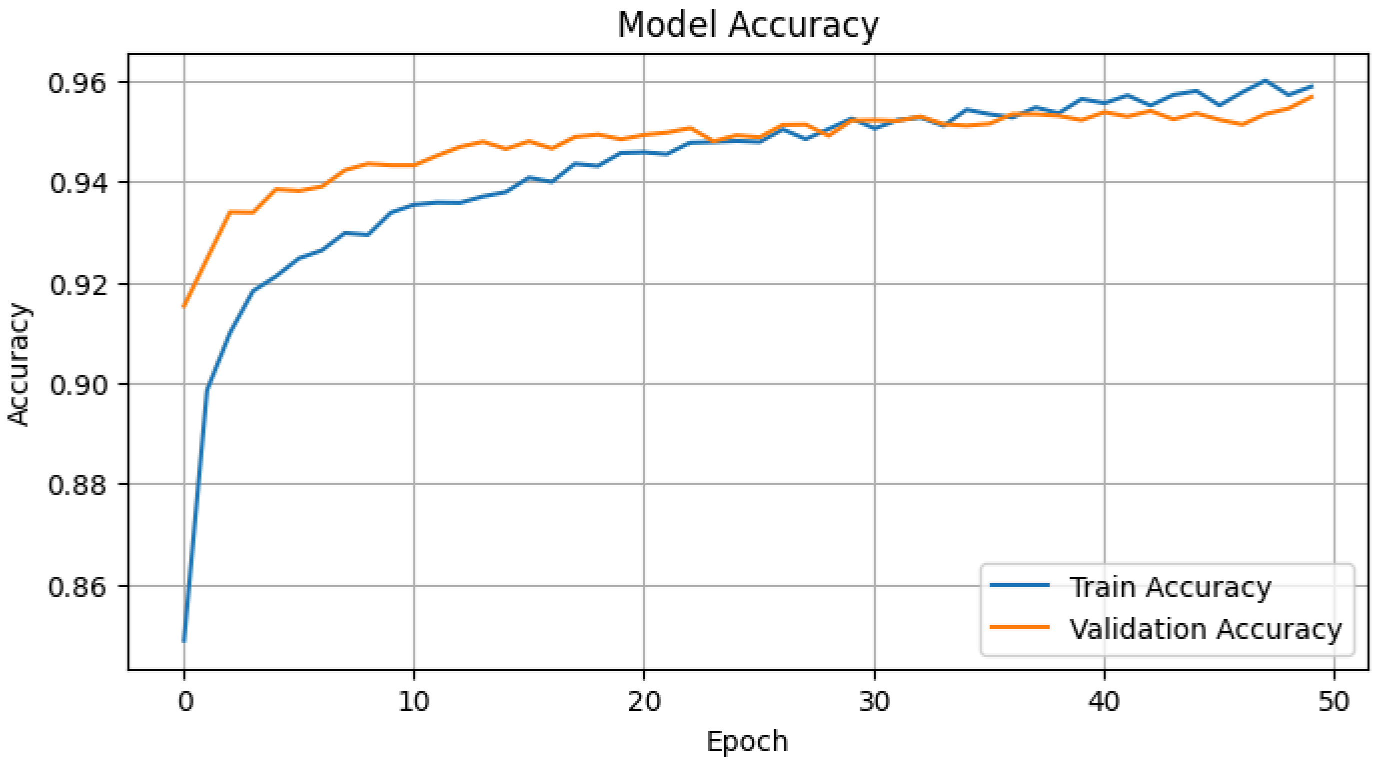
Non-trainable params: 384 (1.50 KB)

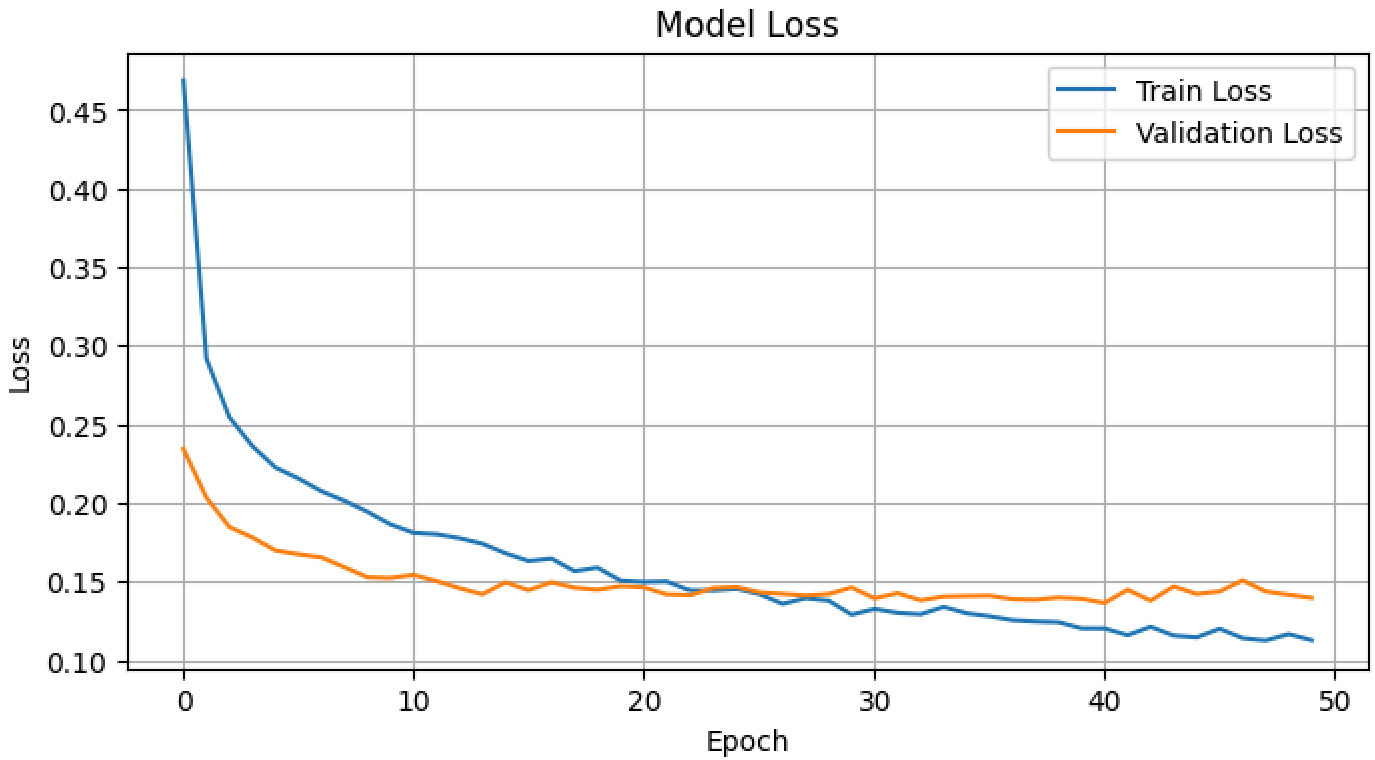
In [9]:
```
# Train the model (increase epochs if needed for better convergence)
history = model.fit(X_train_cnn, y_train_cat,
                    validation_data=(X_test_cnn, y_test_cat),
                    epochs=50, batch_size=64, verbose=1)

# Plot training & validation accuracy values
plt.figure(figsize=(8, 4))
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()

# Plot training & validation loss values
plt.figure(figsize=(8, 4))
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

```
Epoch 1/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 9s 13ms/step - accuracy: 0.7787 - loss: 0.7394 - val_accuracy: 0.9153 - val_loss: 0.2343
Epoch 2/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.8924 - loss: 0.3095 - val_accuracy: 0.9247 - val_loss: 0.2034
Epoch 3/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9054 - loss: 0.2663 - val_accuracy: 0.9340 - val_loss: 0.1847
Epoch 4/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9147 - loss: 0.2461 - val_accuracy: 0.9339 - val_loss: 0.1781
Epoch 5/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9178 - loss: 0.2295 - val_accuracy: 0.9385 - val_loss: 0.1698
Epoch 6/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 11s 15ms/step - accuracy: 0.9225 - loss: 0.2209 - val_accuracy: 0.9382 - val_loss: 0.1674
Epoch 7/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9241 - loss: 0.2139 - val_accuracy: 0.9391 - val_loss: 0.1654
Epoch 8/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 11s 14ms/step - accuracy: 0.9281 - loss: 0.2060 - val_accuracy: 0.9423 - val_loss: 0.1592
Epoch 9/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9294 - loss: 0.1973 - val_accuracy: 0.9436 - val_loss: 0.1528
Epoch 10/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9323 - loss: 0.1903 - val_accuracy: 0.9433 - val_loss: 0.1524
Epoch 11/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9334 - loss: 0.1852 - val_accuracy: 0.9433 - val_loss: 0.1543
Epoch 12/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9348 - loss: 0.1880 - val_accuracy: 0.9452 - val_loss: 0.1504
Epoch 13/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9355 - loss: 0.1841 - val_accuracy: 0.9469 - val_loss: 0.1459
Epoch 14/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9345 - loss: 0.1804 - val_accuracy: 0.9480 - val_loss: 0.1420
Epoch 15/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9367 - loss: 0.1718 - val_accuracy: 0.9465 - val_loss: 0.1496
Epoch 16/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9396 - loss: 0.1657 - val_accuracy: 0.9480 - val_loss: 0.1446
Epoch 17/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 11s 14ms/step - accuracy: 0.9364 - loss: 0.1700 - val_accuracy: 0.9466 - val_loss: 0.1496
Epoch 18/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9416 - loss: 0.1629 - val_accuracy: 0.9489 - val_loss: 0.1462
Epoch 19/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9416 - loss: 0.1664 - val_accuracy: 0.9494 - val_loss: 0.1449
Epoch 20/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 11s 14ms/step - accuracy: 0.9438 - loss: 0.1544 - val_accuracy: 0.9484 - val_loss: 0.1472
Epoch 21/50
```

```
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9426 - loss: 0.1582 - val_accuracy: 0.9493 - val_loss: 0.1465
Epoch 22/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9430 - loss: 0.1560 - val_accuracy: 0.9498 - val_loss: 0.1419
Epoch 23/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9456 - loss: 0.1514 - val_accuracy: 0.9507 - val_loss: 0.1415
Epoch 24/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9478 - loss: 0.1446 - val_accuracy: 0.9480 - val_loss: 0.1460
Epoch 25/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9473 - loss: 0.1477 - val_accuracy: 0.9492 - val_loss: 0.1466
Epoch 26/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9464 - loss: 0.1452 - val_accuracy: 0.9488 - val_loss: 0.1432
Epoch 27/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 11s 15ms/step - accuracy: 0.9485 - loss: 0.1405 - val_accuracy: 0.9513 - val_loss: 0.1423
Epoch 28/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9468 - loss: 0.1469 - val_accuracy: 0.9514 - val_loss: 0.1412
Epoch 29/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9505 - loss: 0.1390 - val_accuracy: 0.9491 - val_loss: 0.1421
Epoch 30/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9513 - loss: 0.1354 - val_accuracy: 0.9522 - val_loss: 0.1463
Epoch 31/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9498 - loss: 0.1377 - val_accuracy: 0.9522 - val_loss: 0.1394
Epoch 32/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9515 - loss: 0.1335 - val_accuracy: 0.9521 - val_loss: 0.1427
Epoch 33/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9523 - loss: 0.1328 - val_accuracy: 0.9530 - val_loss: 0.1382
Epoch 34/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9507 - loss: 0.1346 - val_accuracy: 0.9515 - val_loss: 0.1406
Epoch 35/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9521 - loss: 0.1361 - val_accuracy: 0.9511 - val_loss: 0.1409
Epoch 36/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9525 - loss: 0.1327 - val_accuracy: 0.9515 - val_loss: 0.1411
Epoch 37/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9508 - loss: 0.1305 - val_accuracy: 0.9534 - val_loss: 0.1389
Epoch 38/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9533 - loss: 0.1304 - val_accuracy: 0.9534 - val_loss: 0.1386
Epoch 39/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9523 - loss: 0.1248 - val_accuracy: 0.9531 - val_loss: 0.1399
Epoch 40/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9553 - loss: 0.1232 - val_accuracy: 0.9522 - val_loss: 0.1391
Epoch 41/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9546 - loss: 0.1218 - val_accuracy: 0.9538 - val_loss: 0.1365
```

```
Epoch 42/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9560 - loss: 0.1183 - val_accuracy: 0.9530 - val_loss: 0.1449
Epoch 43/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9542 - loss: 0.1271 - val_accuracy: 0.9541 - val_loss: 0.1379
Epoch 44/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9562 - loss: 0.1192 - val_accuracy: 0.9524 - val_loss: 0.1470
Epoch 45/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9567 - loss: 0.1180 - val_accuracy: 0.9536 - val_loss: 0.1422
Epoch 46/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 11s 15ms/step - accuracy: 0.9555 - loss: 0.1208 - val_accuracy: 0.9523 - val_loss: 0.1438
Epoch 47/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9581 - loss: 0.1147 - val_accuracy: 0.9514 - val_loss: 0.1508
Epoch 48/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9588 - loss: 0.1177 - val_accuracy: 0.9534 - val_loss: 0.1438
Epoch 49/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9571 - loss: 0.1180 - val_accuracy: 0.9546 - val_loss: 0.1417
Epoch 50/50
459/459 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9580 - loss: 0.1155 - val_accuracy: 0.9569 - val_loss: 0.1397
```


Model Accuracy
0.96
0.94
0.92
0.90
0.88
0.86
Accuracy
0
10
20
30
40
50
Epoch
Train Accuracy
Validation Accuracy



In [10]:

```
# Evaluate on test data
score = model.evaluate(X_test_cnn, y_test_cat, verbose=0)
print(f"Test Loss: {score[0]:.4f}  Test Accuracy: {score[1]:.4f}")

# Generate classification report
y_pred = model.predict(X_test_cnn)
y_pred_classes = np.argmax(y_pred, axis=1)
print(classification_report(y_test, y_pred_classes, target_names=[f"C{i+1}" for i in range(num_classes)]))
```

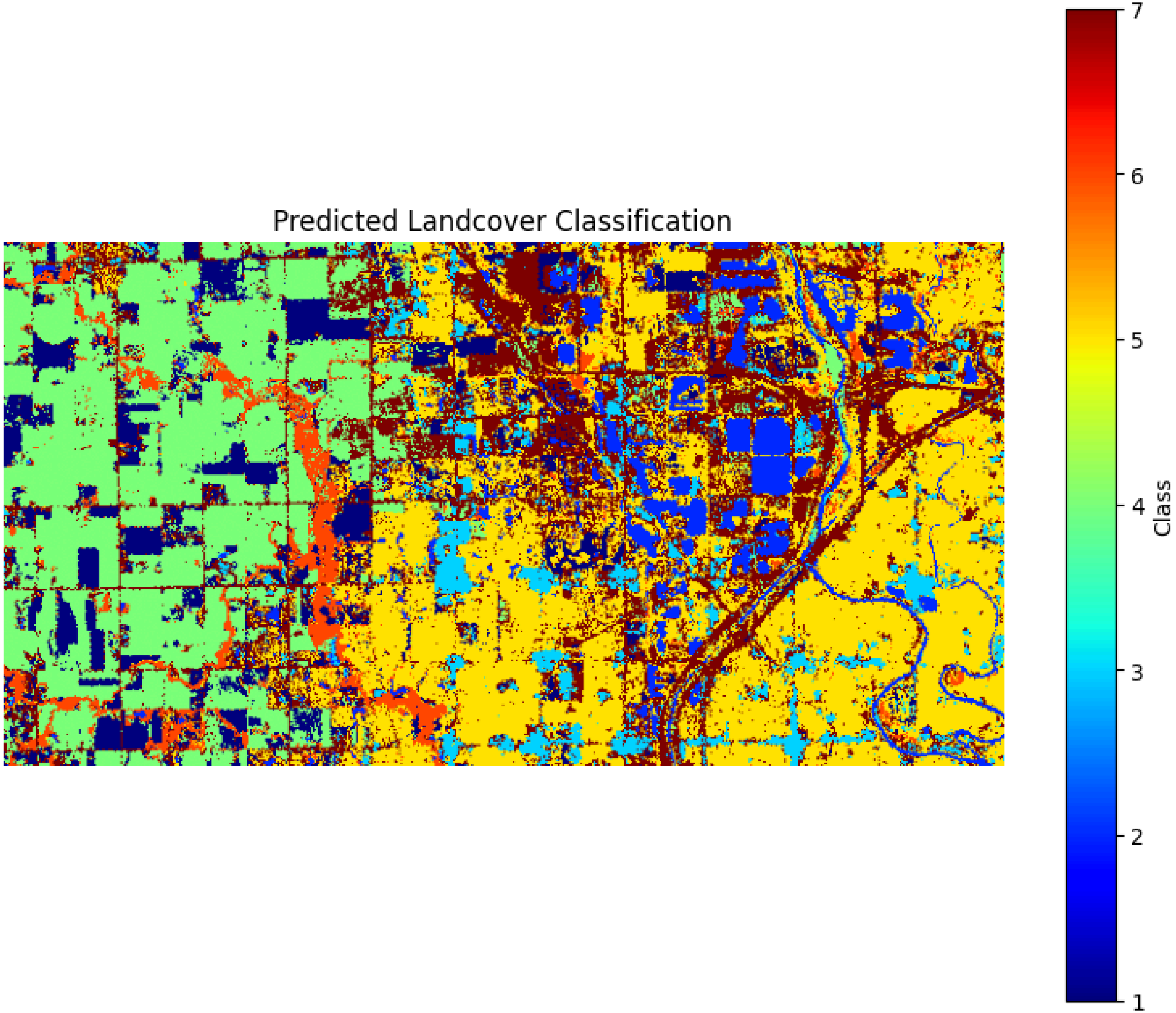
```
Test Loss: 0.1397  Test Accuracy: 0.9569
394/394 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
              precision    recall  f1-score   support

          C1       0.98      0.99      0.99      1561
          C2       1.00      1.00      1.00      1640
          C3       0.91      0.94      0.92      1409
          C4       0.98      0.99      0.99      3626
          C5       0.93      0.98      0.96      2471
          C6       0.96      0.96      0.96       505
          C7       0.88      0.75      0.81      1374

    accuracy                           0.96     12586
   macro avg       0.95      0.94      0.95     12586
weighted avg       0.96      0.96      0.96     12586
```

In [11]:

```python
# Predict for the complete image:
# Use the PCA-transformed data (data_pca of shape (n_pixels, n_components))
data_pca_cnn = data_pca.reshape(-1, input_dim)  # already normalized & PCA-transformed
data_pca_cnn = data_pca_cnn.reshape(-1, input_dim, 1)

# Predict the classes for all pixels
predictions = model.predict(data_pca_cnn)
predicted_labels = np.argmax(predictions, axis=1)

# Reshape back to original image spatial dimensions (rows x cols)
classified_map = predicted_labels.reshape(n_rows, n_cols)

# To display results, add 1 to shift back to original label numbering if needed.
classified_map_disp = classified_map + 1

plt.figure(figsize=(10, 8))
plt.imshow(classified_map_disp, cmap='jet')
plt.title("Predicted Landcover Classification")
plt.axis('off')
plt.colorbar(label="Class")
plt.show()
```

```
6907/6907 ━━━━━━━━━━━━━━━━━━━━ 22s 3ms/step
```

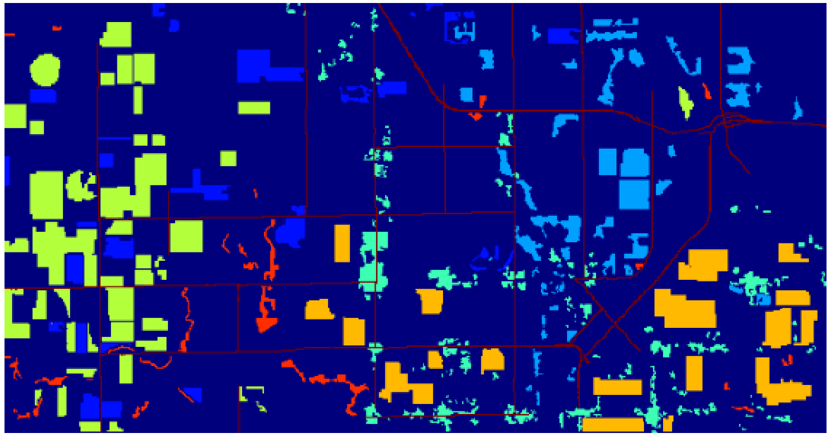
Predicted Landcover Classification



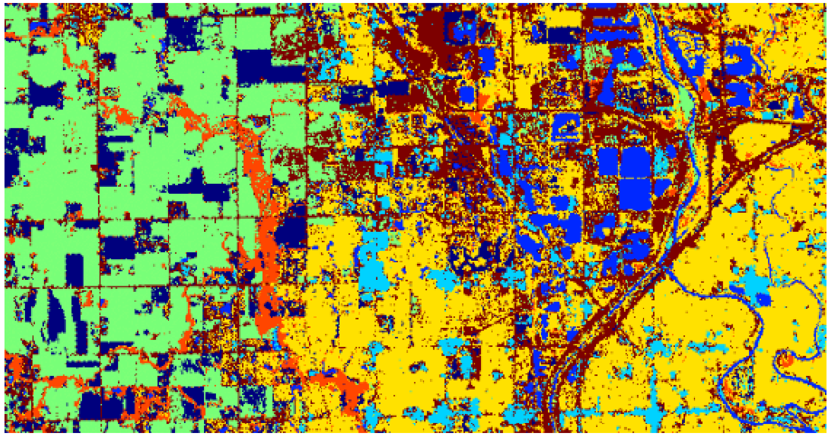
In [12]:

```python
# Compare ground truth and predicted maps side by side
plt.figure(figsize=(16, 8))
plt.subplot(1, 2, 1)
plt.imshow(gt, cmap='jet')
plt.title("Ground Truth")
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(classified_map_disp, cmap='jet')
plt.title("Predicted Classification")
plt.axis('off')
plt.show()

# Plot an example spectral signature from a random labeled pixel
import random
indices = np.argwhere(mask)
random_idx = random.choice(indices)
row, col = random_idx
spectrum = data[row, col, :]
plt.figure(figsize=(8,4))
plt.plot(spectrum)
plt.title(f"Spectral Signature at Pixel ({row}, {col})")
plt.xlabel("Band Index")
plt.ylabel("Reflectance")
plt.grid(True)
plt.show()
```

Ground Truth



Predicted Classification



```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[12], line 18
     16 indices = np.argwhere(mask)
     17 random_idx = random.choice(indices)
---> 18 row, col = random_idx
     19 spectrum = data[row, col, :]
     20 plt.figure(figsize=(8,4))

ValueError: too many values to unpack (expected 2)
```

In [13]:

```python
# Define patch size (should be an odd number)
patch_size = 5
half_patch = patch_size // 2

# Initialize lists for storing patches and labels
patches = []
labels = []

# Loop over all labeled pixels while avoiding image boundaries
for i in range(half_patch, n_rows - half_patch):
    for j in range(half_patch, n_cols - half_patch):
        if gt[i, j] > 0:  # Only use pixels with a label
            patch = data_pca_image[i - half_patch: i + half_patch + 1,
                                   j - half_patch: j + half_patch + 1, :]
            patches.append(patch)
```

```python
            # Adjust label: if ground truth labels are 1-indexed, subtract 1 for zero-indexing
            labels.append(gt[i, j] - 1)

patches = np.array(patches)
labels = np.array(labels)
print("Extracted patches shape:", patches.shape)
print("Labels shape:", labels.shape)
```

```
Extracted patches shape: (41578, 5, 5, 30)
Labels shape: (41578, 1)
```

In [14]:

```python
# Define number of classes (from the provided dataset info: 7 classes)
num_classes = 7

# One-hot encode labels
labels_cat = to_categorical(labels, num_classes=num_classes)

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(patches, labels_cat,
                                                    test_size=0.3, random_state=42,
                                                    stratify=labels)
print("Training samples:", X_train.shape, "Testing samples:", X_test.shape)
```

```
Training samples: (29104, 5, 5, 30) Testing samples: (12474, 5, 5, 30)
```

In [17]:

```python
from tensorflow.keras.layers import Conv2D, Dense, Flatten, Dropout, BatchNormalization, MaxPooling2D, InputLayer
input_shape = (patch_size, patch_size, n_components)

model = Sequential([
    InputLayer(input_shape=input_shape),
    Conv2D(32, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.3),

    Conv2D(64, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.3),

    Flatten(),
```

```python
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 5, 5, 32)	8,672
batch_normalization_2 (BatchNormalization)	(None, 5, 5, 32)	128
max_pooling2d (MaxPooling2D)	(None, 2, 2, 32)	0
dropout_3 (Dropout)	(None, 2, 2, 32)	0
conv2d_1 (Conv2D)	(None, 2, 2, 64)	18,496
batch_normalization_3 (BatchNormalization)	(None, 2, 2, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 1, 1, 64)	0
dropout_4 (Dropout)	(None, 1, 1, 64)	0
flatten_1 (Flatten)	(None, 64)	0
dense_2 (Dense)	(None, 128)	8,320
dropout_5 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 7)	903

Total params: 36,775 (143.65 KB)

Trainable params: 36,583 (142.90 KB)

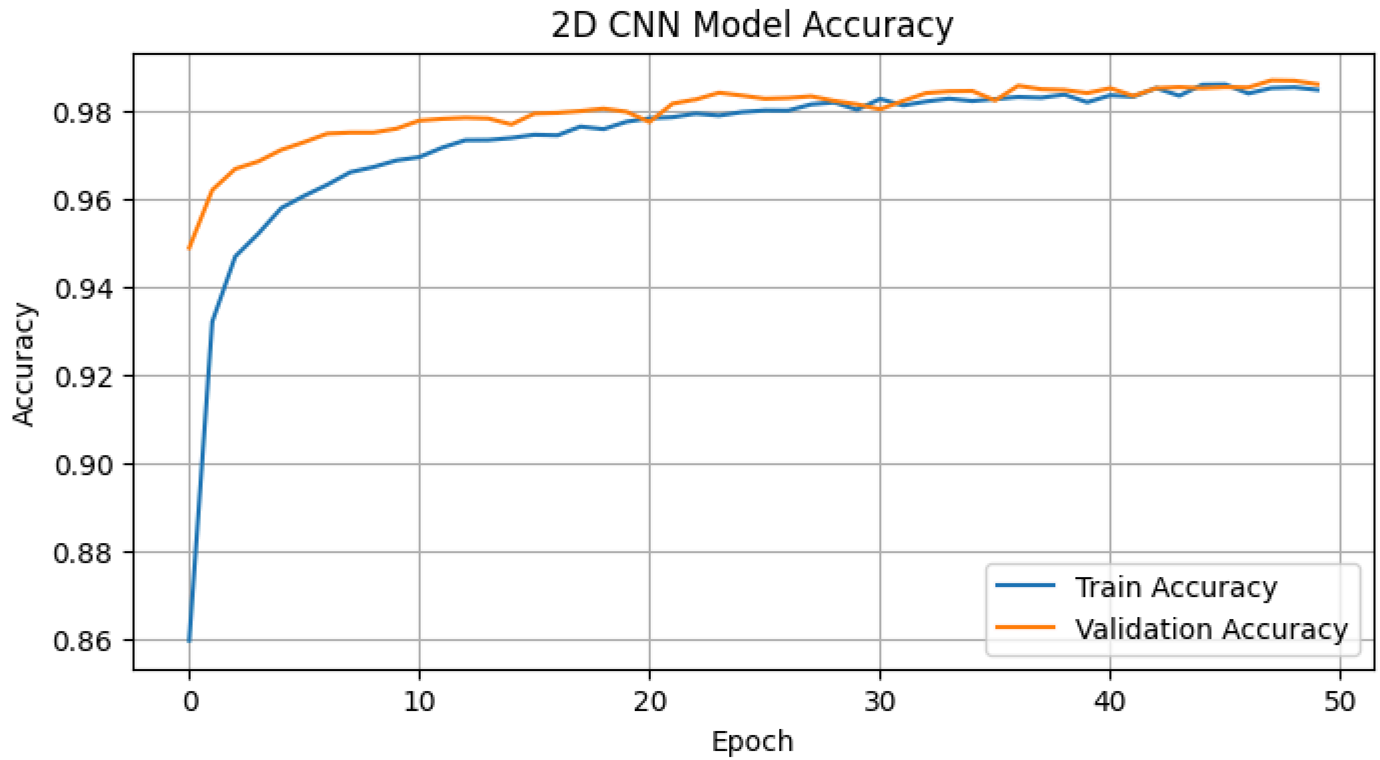
Non-trainable params: 192 (768.00 B)

In [18]:
```
# Train the model (adjust epochs/batch size as needed)
history = model.fit(X_train, y_train,
                    validation_data=(X_test, y_test),
                    epochs=50, batch_size=64, verbose=1)

# Plot training & validation accuracy
plt.figure(figsize=(8, 4))
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('2D CNN Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()

# Plot training & validation loss
plt.figure(figsize=(8, 4))
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('2D CNN Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

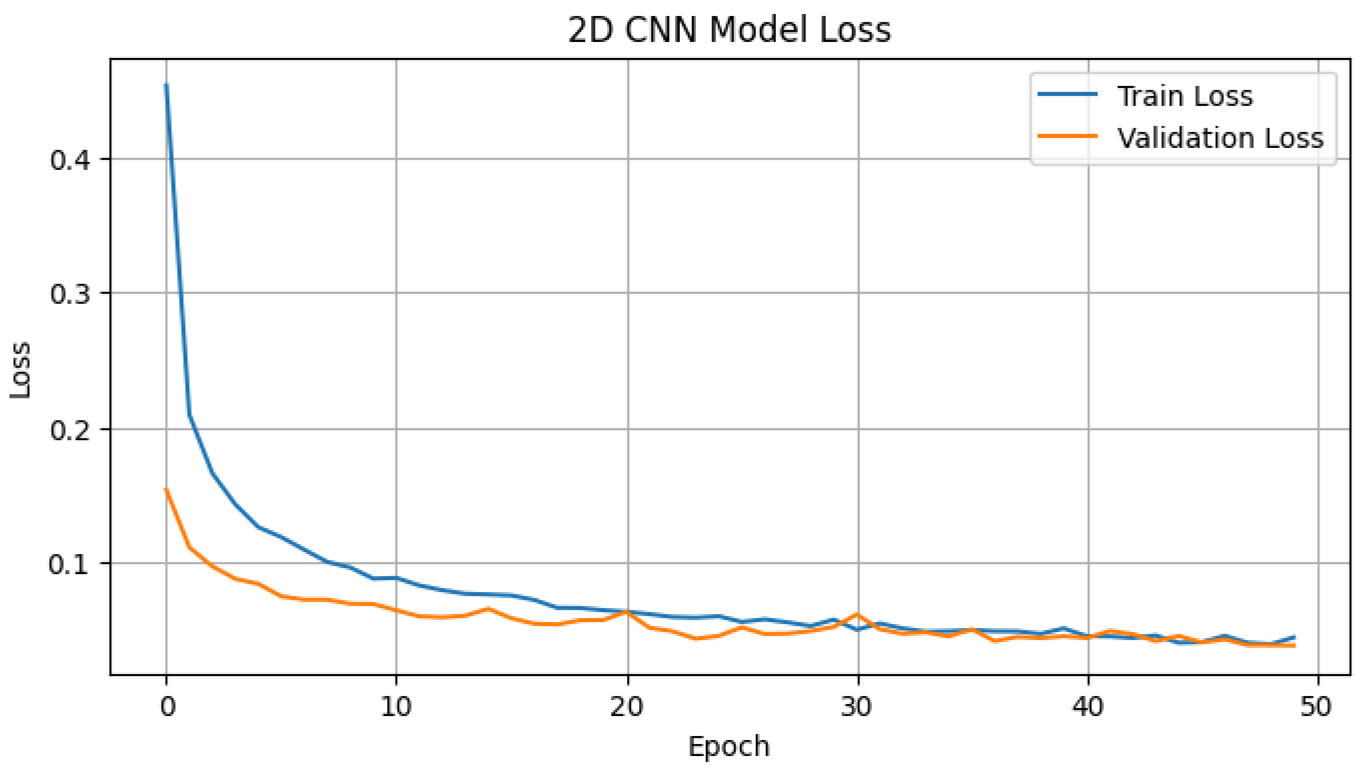
```
Epoch 1/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 7s 9ms/step - accuracy: 0.7600 - loss: 0.7593 - val_accuracy: 0.9489 - val_loss: 0.1536
Epoch 2/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9243 - loss: 0.2297 - val_accuracy: 0.9620 - val_loss: 0.1110
Epoch 3/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9439 - loss: 0.1791 - val_accuracy: 0.9668 - val_loss: 0.0969
Epoch 4/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9489 - loss: 0.1516 - val_accuracy: 0.9685 - val_loss: 0.0877
Epoch 5/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9544 - loss: 0.1321 - val_accuracy: 0.9711 - val_loss: 0.0839
Epoch 6/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9572 - loss: 0.1251 - val_accuracy: 0.9729 - val_loss: 0.0747
Epoch 7/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9618 - loss: 0.1112 - val_accuracy: 0.9748 - val_loss: 0.0722
Epoch 8/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9640 - loss: 0.1059 - val_accuracy: 0.9751 - val_loss: 0.0722
Epoch 9/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9649 - loss: 0.0982 - val_accuracy: 0.9751 - val_loss: 0.0692
Epoch 10/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9664 - loss: 0.0939 - val_accuracy: 0.9760 - val_loss: 0.0690
Epoch 11/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9690 - loss: 0.0892 - val_accuracy: 0.9778 - val_loss: 0.0643
Epoch 12/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9692 - loss: 0.0885 - val_accuracy: 0.9782 - val_loss: 0.0599
Epoch 13/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9725 - loss: 0.0814 - val_accuracy: 0.9784 - val_loss: 0.0591
Epoch 14/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9731 - loss: 0.0777 - val_accuracy: 0.9783 - val_loss: 0.0603
Epoch 15/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9732 - loss: 0.0754 - val_accuracy: 0.9769 - val_loss: 0.0654
Epoch 16/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9727 - loss: 0.0808 - val_accuracy: 0.9794 - val_loss: 0.0584
Epoch 17/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9727 - loss: 0.0753 - val_accuracy: 0.9796 - val_loss: 0.0544
Epoch 18/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9760 - loss: 0.0663 - val_accuracy: 0.9800 - val_loss: 0.0538
Epoch 19/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9757 - loss: 0.0658 - val_accuracy: 0.9805 - val_loss: 0.0569
Epoch 20/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 5s 10ms/step - accuracy: 0.9761 - loss: 0.0652 - val_accuracy: 0.9798 - val_loss: 0.0571
Epoch 21/50
```

```
455/455 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9780 - loss: 0.0634 - val_accuracy: 0.9775 - val_loss: 0.0632
Epoch 22/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9763 - loss: 0.0642 - val_accuracy: 0.9816 - val_loss: 0.0514
Epoch 23/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9780 - loss: 0.0602 - val_accuracy: 0.9826 - val_loss: 0.0487
Epoch 24/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9772 - loss: 0.0622 - val_accuracy: 0.9841 - val_loss: 0.0432
Epoch 25/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9784 - loss: 0.0610 - val_accuracy: 0.9835 - val_loss: 0.0453
Epoch 26/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9792 - loss: 0.0563 - val_accuracy: 0.9828 - val_loss: 0.0517
Epoch 27/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9785 - loss: 0.0594 - val_accuracy: 0.9829 - val_loss: 0.0468
Epoch 28/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9794 - loss: 0.0610 - val_accuracy: 0.9833 - val_loss: 0.0470
Epoch 29/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9828 - loss: 0.0497 - val_accuracy: 0.9823 - val_loss: 0.0489
Epoch 30/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9797 - loss: 0.0584 - val_accuracy: 0.9815 - val_loss: 0.0520
Epoch 31/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - accuracy: 0.9820 - loss: 0.0511 - val_accuracy: 0.9804 - val_loss: 0.0612
Epoch 32/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9801 - loss: 0.0555 - val_accuracy: 0.9823 - val_loss: 0.0503
Epoch 33/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9806 - loss: 0.0556 - val_accuracy: 0.9840 - val_loss: 0.0469
Epoch 34/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9823 - loss: 0.0474 - val_accuracy: 0.9844 - val_loss: 0.0480
Epoch 35/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - accuracy: 0.9821 - loss: 0.0502 - val_accuracy: 0.9845 - val_loss: 0.0450
Epoch 36/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9816 - loss: 0.0495 - val_accuracy: 0.9824 - val_loss: 0.0503
Epoch 37/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9833 - loss: 0.0470 - val_accuracy: 0.9857 - val_loss: 0.0415
Epoch 38/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9824 - loss: 0.0502 - val_accuracy: 0.9849 - val_loss: 0.0446
Epoch 39/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9833 - loss: 0.0483 - val_accuracy: 0.9848 - val_loss: 0.0437
Epoch 40/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9810 - loss: 0.0536 - val_accuracy: 0.9840 - val_loss: 0.0451
Epoch 41/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9826 - loss: 0.0454 - val_accuracy: 0.9852 - val_loss: 0.0437
```

```
Epoch 42/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9839 - loss: 0.0427 - val_accuracy: 0.9834 - val_loss: 0.0490
Epoch 43/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9845 - loss: 0.0448 - val_accuracy: 0.9852 - val_loss: 0.0466
Epoch 44/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9834 - loss: 0.0449 - val_accuracy: 0.9854 - val_loss: 0.0415
Epoch 45/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9857 - loss: 0.0404 - val_accuracy: 0.9852 - val_loss: 0.0453
Epoch 46/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9856 - loss: 0.0416 - val_accuracy: 0.9854 - val_loss: 0.0406
Epoch 47/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9837 - loss: 0.0451 - val_accuracy: 0.9853 - val_loss: 0.0427
Epoch 48/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 6s 9ms/step - accuracy: 0.9838 - loss: 0.0418 - val_accuracy: 0.9869 - val_loss: 0.0385
Epoch 49/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9844 - loss: 0.0406 - val_accuracy: 0.9869 - val_loss: 0.0385
Epoch 50/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9840 - loss: 0.0446 - val_accuracy: 0.9861 - val_loss: 0.0382
```



2D CNN Model Accuracy
Accuracy
Epoch
0.86
0.88
0.90
0.92
0.94
0.96
0.98
0
10
20
30
40
50
Train Accuracy
Validation Accuracy



In [19]:

```
# Evaluate the model on the test set
score = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Loss: {score[0]:.4f}  Test Accuracy: {score[1]:.4f}")

# Generate predictions and classification report
y_test_labels = np.argmax(y_test, axis=1)
y_pred = model.predict(X_test)
y_pred_labels = np.argmax(y_pred, axis=1)
print(classification_report(y_test_labels, y_pred_labels, target_names=[f"C{i+1}" for i in range(num_classes)]))
```

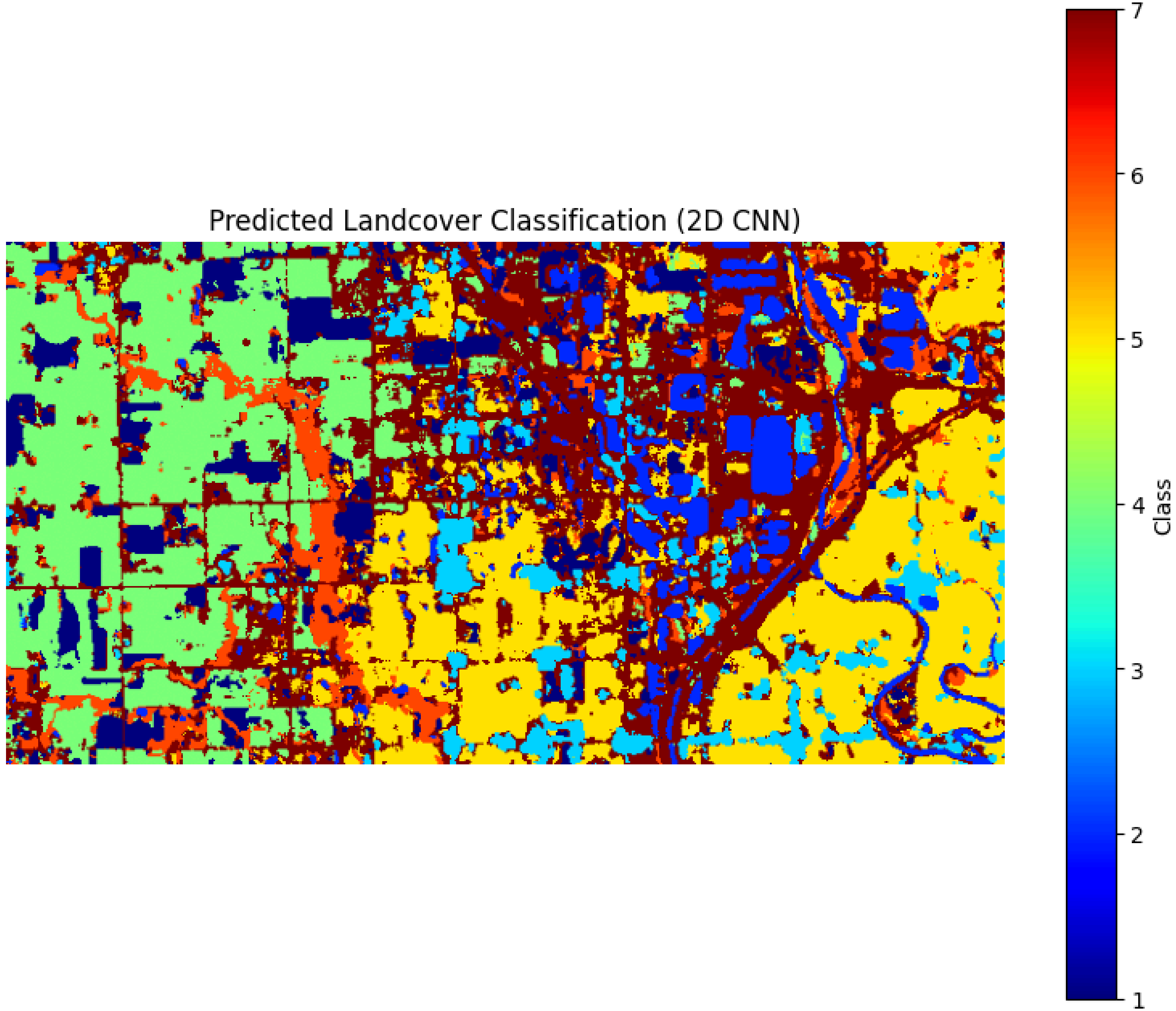
```
Test Loss: 0.0382  Test Accuracy: 0.9861
390/390 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
              precision    recall  f1-score   support

          C1       0.99      1.00      1.00      1532
          C2       1.00      1.00      1.00      1640
          C3       0.95      0.98      0.97      1399
          C4       1.00      1.00      1.00      3578
          C5       0.98      1.00      0.99      2454
          C6       0.99      0.99      0.99       503
          C7       0.97      0.91      0.94      1368

    accuracy                           0.99     12474
   macro avg       0.98      0.98      0.98     12474
weighted avg       0.99      0.99      0.99     12474
```

In [20]:

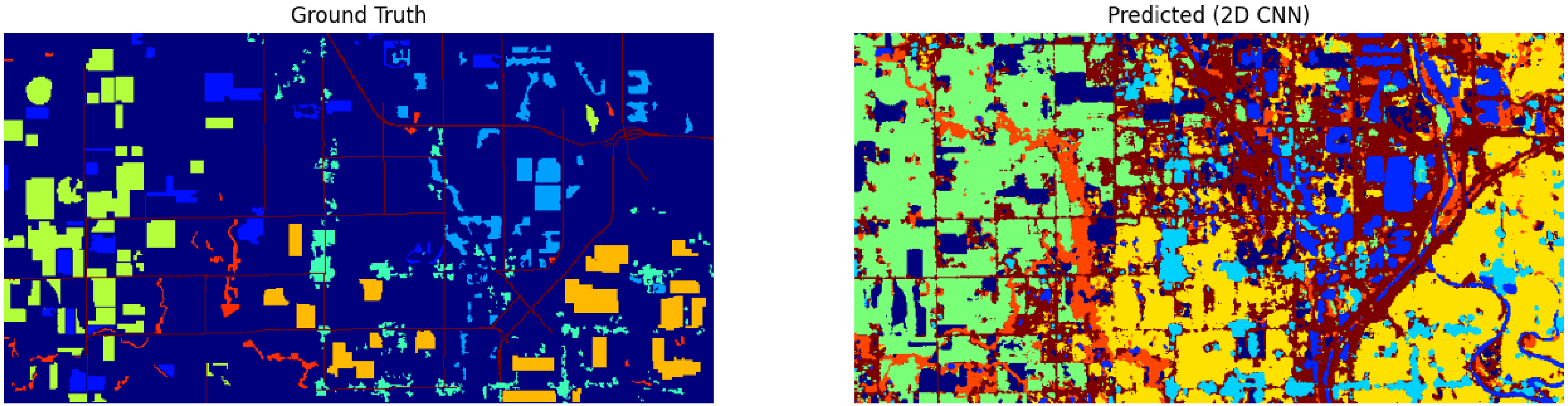
```
# Define a function to predict the class for every pixel using sliding window patch extraction
def classify_full_image(data_img, model, patch_size):
    half_patch = patch_size // 2
    padded = np.pad(data_img, ((half_patch, half_patch), (half_patch, half_patch), (0, 0)), mode='reflect')
    # Convert to tensor and add batch dimension: shape (1, padded_rows, padded_cols, n_components)
    padded_tensor = tf.convert_to_tensor(padded[np.newaxis, ...], dtype=tf.float32)

    # Use tf.image.extract_patches to extract patches for every pixel in the original image
    patches_tf = tf.image.extract_patches(images=padded_tensor,
                                          sizes=[1, patch_size, patch_size, 1],
                                          strides=[1, 1, 1, 1],
                                          rates=[1, 1, 1, 1],
                                          padding='VALID')
    # The patches are flattened. Reshape them to (n_pixels, patch_size, patch_size, n_components)
    patches_shape = patches_tf.shape
    patches_reshaped = tf.reshape(patches_tf, (-1, patch_size, patch_size, data_img.shape[2]))

    # Predict classes for all patches
    preds = model.predict(patches_reshaped, batch_size=256)
    pred_labels = np.argmax(preds, axis=1)

    # Reshape predictions back to the image shape
    pred_map = pred_labels.reshape(data_img.shape[0], data_img.shape[1])
    return pred_map
```

```python
# Run full image classification using the PCA-transformed image
classified_map = classify_full_image(data_pca_image, model, patch_size)
# Shift labels back to original (if training labels were zero-indexed, add 1)
classified_map_disp = classified_map + 1

plt.figure(figsize=(10, 8))
plt.imshow(classified_map_disp, cmap='jet')
plt.title("Predicted Landcover Classification (2D CNN)")
plt.axis('off')
plt.colorbar(label="Class")
plt.show()
```

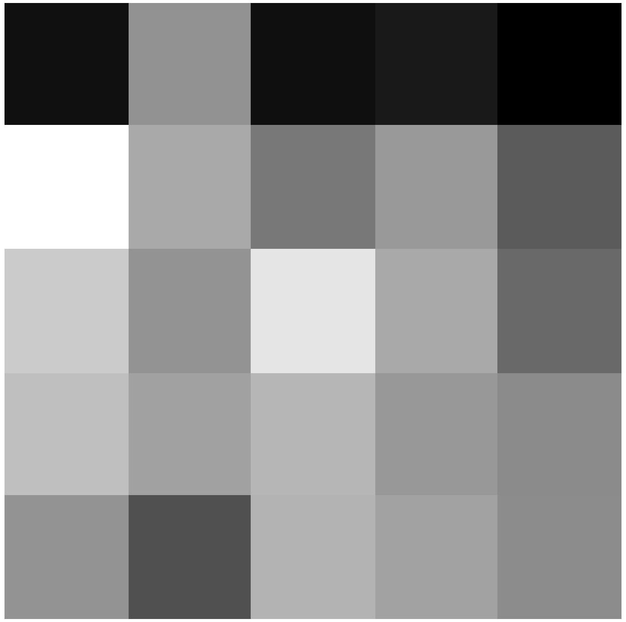
```
864/864 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step
```


Predicted Landcover Classification (2D CNN)
Class
7
6
5
4
3
2
1

```
In [21]: # Side-by-side comparison of ground truth and predicted classification maps
plt.figure(figsize=(16, 8))
plt.subplot(1, 2, 1)
plt.imshow(gt, cmap='jet')
plt.title("Ground Truth")
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(classified_map_disp, cmap='jet')
plt.title("Predicted (2D CNN)")
plt.axis('off')
plt.show()

# Display an example patch and its predicted class
sample_idx = np.random.randint(0, X_test.shape[0])
plt.figure(figsize=(4, 4))
plt.imshow(X_test[sample_idx].reshape(patch_size, patch_size, n_components)[:, :, 0], cmap='gray')
plt.title(f"Example Patch - True Class: {np.argmax(y_test[sample_idx])+1}")
plt.axis('off')
plt.show()
```



Example Patch - True Class: 1



In [22]:
```
# Define patch size (use an odd number, e.g., 5)
patch_size = 5
half_patch = patch_size // 2

patches_3d = []
labels_3d = []

# Loop over valid pixel locations (avoiding boundaries)
for i in range(half_patch, n_rows - half_patch):
    for j in range(half_patch, n_cols - half_patch):
        if gt[i, j] > 0:  # Only consider labeled pixels
            patch = data_pca_image[i - half_patch: i + half_patch + 1,
                                   j - half_patch: j + half_patch + 1, :]
            patches_3d.append(patch)
            # Adjust label to be zero-indexed
            labels_3d.append(gt[i, j] - 1)

patches_3d = np.array(patches_3d)
labels_3d = np.array(labels_3d)
```

```python
print("Extracted 3D patches shape:", patches_3d.shape)
print("Labels shape:", labels_3d.shape)

# Add a channel dimension for 3D CNN: new shape becomes (samples, patch_size, patch_size, n_components, 1)
patches_3d = patches_3d[..., np.newaxis]
```

```
Extracted 3D patches shape: (41578, 5, 5, 30)
Labels shape: (41578, 1)
```

In [23]:

```python
from tensorflow.keras.layers import Conv3D, MaxPooling3D, Dense, Flatten, Dropout, BatchNormalization, InputLayer
# Assume the dataset has 7 classes as per the provided information
num_classes = 7

# One-hot encode the labels
labels_3d_cat = to_categorical(labels_3d, num_classes=num_classes)

# Split the patches into training and testing sets
X3_train, X3_test, y3_train, y3_test = train_test_split(patches_3d, labels_3d_cat,
                                                        test_size=0.3, random_state=42,
                                                        stratify=labels_3d)
print("Training samples (3D):", X3_train.shape, "Testing samples (3D):", X3_test.shape)
```

```
Training samples (3D): (29104, 5, 5, 30, 1) Testing samples (3D): (12474, 5, 5, 30, 1)
```

In [24]:

```python
# Define the input shape for the 3D CNN: (patch_size, patch_size, n_components, 1)
input_shape_3d = (patch_size, patch_size, n_components, 1)

model_3d = Sequential([
    InputLayer(input_shape=input_shape_3d),
    Conv3D(32, kernel_size=(3,3,3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling3D(pool_size=(2,2,2)),
    Dropout(0.3),

    Conv3D(64, kernel_size=(3,3,3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling3D(pool_size=(2,2,2)),
    Dropout(0.3),

    Flatten(),
    Dense(128, activation='relu'),
```

```
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
])
model_3d.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model_3d.summary()
```

```
C:\Users\Shewak Heera\AppData\Local\Programs\Python\Python310\lib\site-packages\keras\src\layers\core\input_layer.py:27: UserWa
rning: Argument `input_shape` is deprecated. Use `shape` instead.
  warnings.warn(
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv3d (Conv3D)	(None, 5, 5, 30, 32)	896
batch_normalization_4 (BatchNormalization)	(None, 5, 5, 30, 32)	128
max_pooling3d (MaxPooling3D)	(None, 2, 2, 15, 32)	0
dropout_6 (Dropout)	(None, 2, 2, 15, 32)	0
conv3d_1 (Conv3D)	(None, 2, 2, 15, 64)	55,360
batch_normalization_5 (BatchNormalization)	(None, 2, 2, 15, 64)	256
max_pooling3d_1 (MaxPooling3D)	(None, 1, 1, 7, 64)	0
dropout_7 (Dropout)	(None, 1, 1, 7, 64)	0
flatten_2 (Flatten)	(None, 448)	0
dense_4 (Dense)	(None, 128)	57,472
dropout_8 (Dropout)	(None, 128)	0
dense_5 (Dense)	(None, 7)	903

Total params: 115,015 (449.28 KB)

Trainable params: 114,823 (448.53 KB)

Non-trainable params: 192 (768.00 B)

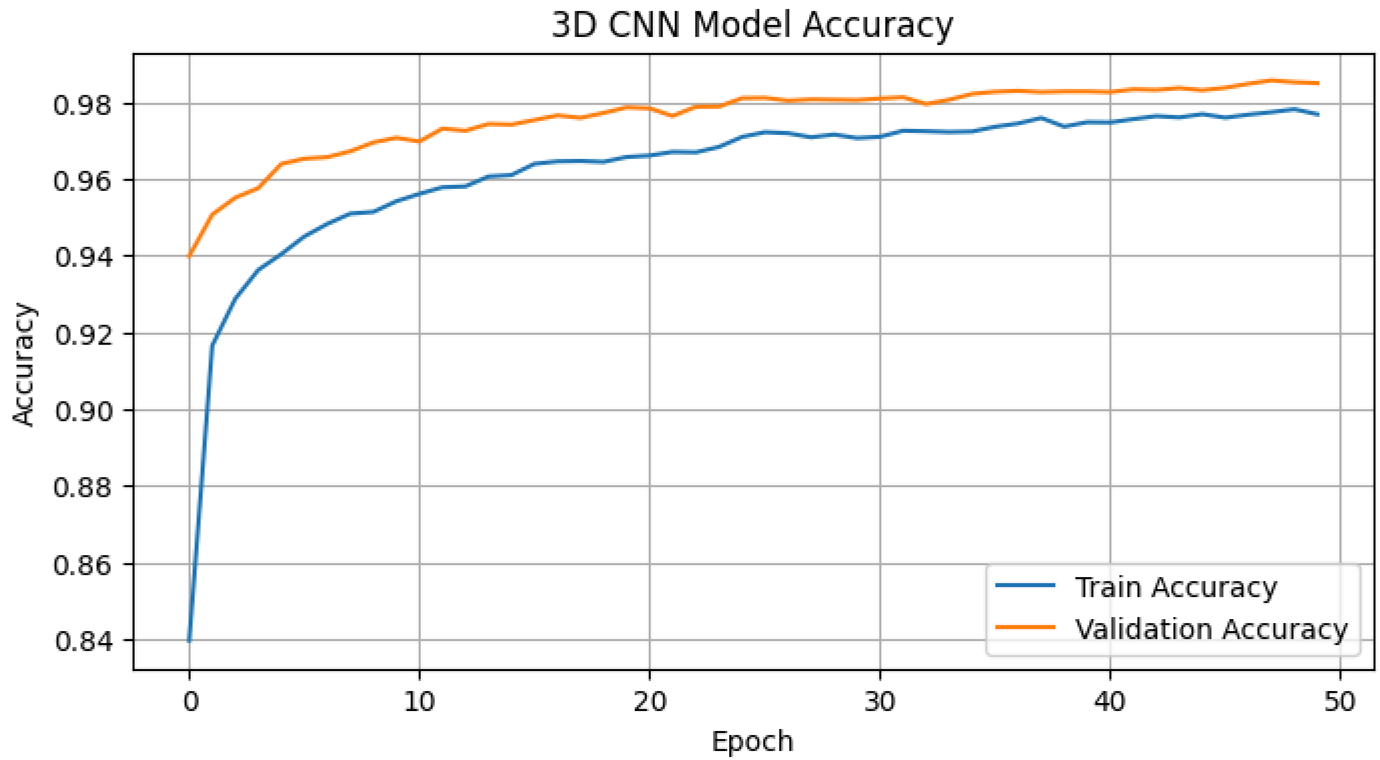
In [25]:

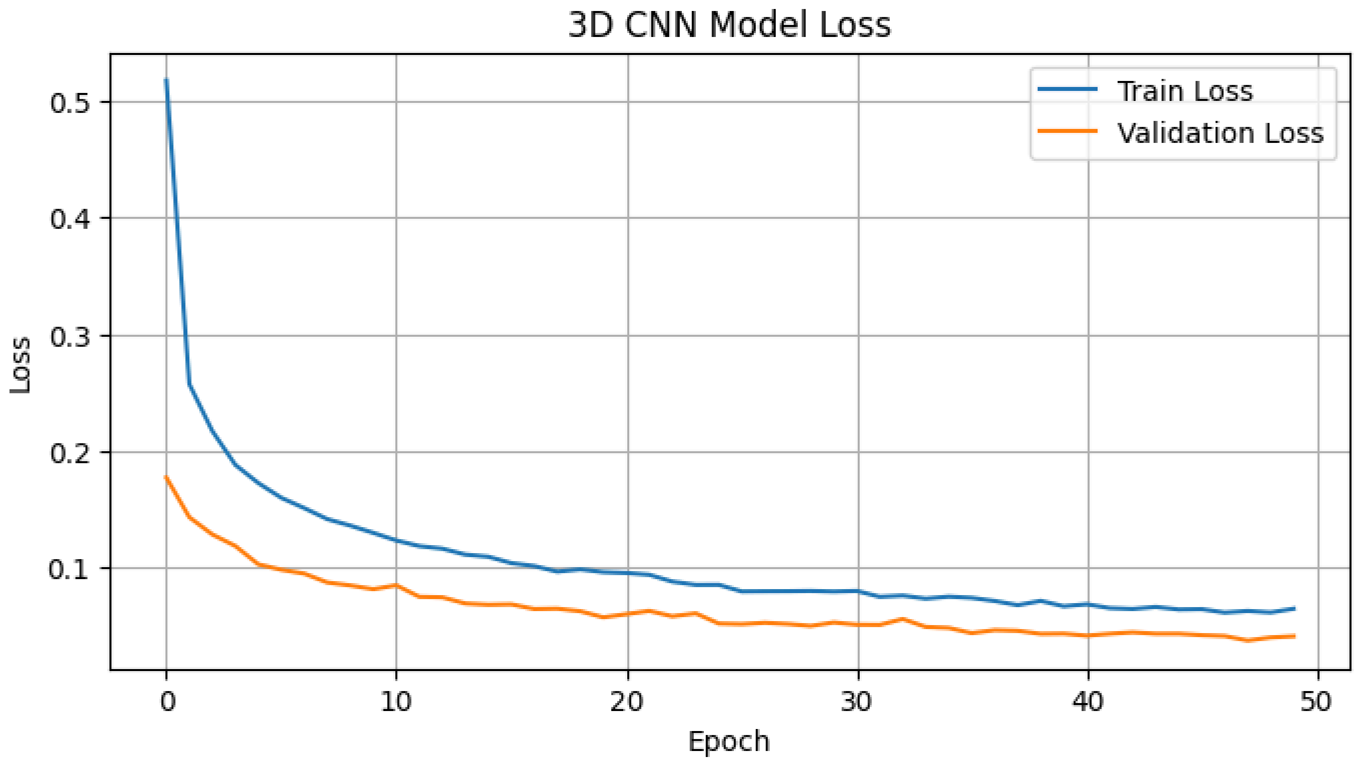
```python
# Train the 3D CNN model (adjust epochs and batch size as needed)
history_3d = model_3d.fit(X3_train, y3_train,
                          validation_data=(X3_test, y3_test),
                          epochs=50, batch_size=64, verbose=1)

# Plot training and validation accuracy
plt.figure(figsize=(8,4))
plt.plot(history_3d.history['accuracy'], label='Train Accuracy')
plt.plot(history_3d.history['val_accuracy'], label='Validation Accuracy')
plt.title('3D CNN Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()

# Plot training and validation loss
plt.figure(figsize=(8,4))
plt.plot(history_3d.history['loss'], label='Train Loss')
plt.plot(history_3d.history['val_loss'], label='Validation Loss')
plt.title('3D CNN Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

```
Epoch 1/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 17s 30ms/step - accuracy: 0.7407 - loss: 0.8847 - val_accuracy: 0.9398 - val_loss: 0.1772
Epoch 2/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 12s 27ms/step - accuracy: 0.9095 - loss: 0.2827 - val_accuracy: 0.9506 - val_loss: 0.1432
Epoch 3/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 27ms/step - accuracy: 0.9267 - loss: 0.2246 - val_accuracy: 0.9550 - val_loss: 0.1285
Epoch 4/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 28ms/step - accuracy: 0.9311 - loss: 0.1969 - val_accuracy: 0.9575 - val_loss: 0.1186
Epoch 5/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 14s 31ms/step - accuracy: 0.9354 - loss: 0.1867 - val_accuracy: 0.9638 - val_loss: 0.1029
Epoch 6/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 14s 31ms/step - accuracy: 0.9436 - loss: 0.1633 - val_accuracy: 0.9652 - val_loss: 0.0982
Epoch 7/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 14s 31ms/step - accuracy: 0.9463 - loss: 0.1531 - val_accuracy: 0.9656 - val_loss: 0.0949
Epoch 8/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 28ms/step - accuracy: 0.9499 - loss: 0.1418 - val_accuracy: 0.9671 - val_loss: 0.0873
Epoch 9/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 28ms/step - accuracy: 0.9497 - loss: 0.1411 - val_accuracy: 0.9695 - val_loss: 0.0848
Epoch 10/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9517 - loss: 0.1316 - val_accuracy: 0.9707 - val_loss: 0.0816
Epoch 11/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 28ms/step - accuracy: 0.9556 - loss: 0.1230 - val_accuracy: 0.9697 - val_loss: 0.0849
Epoch 12/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9555 - loss: 0.1221 - val_accuracy: 0.9731 - val_loss: 0.0751
Epoch 13/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 28ms/step - accuracy: 0.9570 - loss: 0.1163 - val_accuracy: 0.9725 - val_loss: 0.0746
Epoch 14/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9577 - loss: 0.1170 - val_accuracy: 0.9743 - val_loss: 0.0693
Epoch 15/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 14s 30ms/step - accuracy: 0.9601 - loss: 0.1093 - val_accuracy: 0.9741 - val_loss: 0.0683
Epoch 16/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9632 - loss: 0.1054 - val_accuracy: 0.9753 - val_loss: 0.0686
Epoch 17/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 28ms/step - accuracy: 0.9616 - loss: 0.1074 - val_accuracy: 0.9765 - val_loss: 0.0646
Epoch 18/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9637 - loss: 0.0986 - val_accuracy: 0.9759 - val_loss: 0.0649
Epoch 19/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 14s 31ms/step - accuracy: 0.9632 - loss: 0.0990 - val_accuracy: 0.9772 - val_loss: 0.0627
Epoch 20/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 14s 31ms/step - accuracy: 0.9646 - loss: 0.0973 - val_accuracy: 0.9786 - val_loss: 0.0575
Epoch 21/50
```

```
455/455 ━━━━━━━━━━━━━━━━━━━━ 14s 30ms/step - accuracy: 0.9645 - loss: 0.0963 - val_accuracy: 0.9784 - val_loss: 0.0602
Epoch 22/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9659 - loss: 0.0954 - val_accuracy: 0.9764 - val_loss: 0.0630
Epoch 23/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9662 - loss: 0.0883 - val_accuracy: 0.9788 - val_loss: 0.0584
Epoch 24/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9673 - loss: 0.0905 - val_accuracy: 0.9788 - val_loss: 0.0609
Epoch 25/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9705 - loss: 0.0845 - val_accuracy: 0.9810 - val_loss: 0.0521
Epoch 26/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9717 - loss: 0.0801 - val_accuracy: 0.9811 - val_loss: 0.0516
Epoch 27/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 14s 30ms/step - accuracy: 0.9700 - loss: 0.0829 - val_accuracy: 0.9804 - val_loss: 0.0528
Epoch 28/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9696 - loss: 0.0813 - val_accuracy: 0.9807 - val_loss: 0.0518
Epoch 29/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9704 - loss: 0.0790 - val_accuracy: 0.9806 - val_loss: 0.0501
Epoch 30/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 14s 30ms/step - accuracy: 0.9702 - loss: 0.0808 - val_accuracy: 0.9805 - val_loss: 0.0531
Epoch 31/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9702 - loss: 0.0834 - val_accuracy: 0.9809 - val_loss: 0.0512
Epoch 32/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9724 - loss: 0.0762 - val_accuracy: 0.9812 - val_loss: 0.0510
Epoch 33/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9719 - loss: 0.0766 - val_accuracy: 0.9795 - val_loss: 0.0560
Epoch 34/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9706 - loss: 0.0778 - val_accuracy: 0.9806 - val_loss: 0.0492
Epoch 35/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 16s 35ms/step - accuracy: 0.9719 - loss: 0.0753 - val_accuracy: 0.9821 - val_loss: 0.0484
Epoch 36/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9730 - loss: 0.0761 - val_accuracy: 0.9827 - val_loss: 0.0439
Epoch 37/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 15s 32ms/step - accuracy: 0.9731 - loss: 0.0732 - val_accuracy: 0.9829 - val_loss: 0.0465
Epoch 38/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 14s 31ms/step - accuracy: 0.9753 - loss: 0.0673 - val_accuracy: 0.9826 - val_loss: 0.0459
Epoch 39/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 14s 32ms/step - accuracy: 0.9724 - loss: 0.0721 - val_accuracy: 0.9828 - val_loss: 0.0433
Epoch 40/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 17s 38ms/step - accuracy: 0.9753 - loss: 0.0658 - val_accuracy: 0.9828 - val_loss: 0.0435
Epoch 41/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 15s 32ms/step - accuracy: 0.9733 - loss: 0.0696 - val_accuracy: 0.9826 - val_loss: 0.0417
```

```
Epoch 42/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 15s 33ms/step - accuracy: 0.9748 - loss: 0.0653 - val_accuracy: 0.9833 - val_loss: 0.0434
Epoch 43/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 14s 30ms/step - accuracy: 0.9757 - loss: 0.0651 - val_accuracy: 0.9832 - val_loss: 0.0446
Epoch 44/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 14s 30ms/step - accuracy: 0.9758 - loss: 0.0655 - val_accuracy: 0.9836 - val_loss: 0.0434
Epoch 45/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 30ms/step - accuracy: 0.9762 - loss: 0.0632 - val_accuracy: 0.9831 - val_loss: 0.0434
Epoch 46/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9761 - loss: 0.0628 - val_accuracy: 0.9837 - val_loss: 0.0421
Epoch 47/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 30ms/step - accuracy: 0.9771 - loss: 0.0615 - val_accuracy: 0.9848 - val_loss: 0.0414
Epoch 48/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 13s 29ms/step - accuracy: 0.9765 - loss: 0.0616 - val_accuracy: 0.9857 - val_loss: 0.0376
Epoch 49/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 14s 30ms/step - accuracy: 0.9766 - loss: 0.0628 - val_accuracy: 0.9852 - val_loss: 0.0402
Epoch 50/50
455/455 ━━━━━━━━━━━━━━━━━━━━ 14s 31ms/step - accuracy: 0.9766 - loss: 0.0670 - val_accuracy: 0.9849 - val_loss: 0.0412
```


3D CNN Model Accuracy
Accuracy
0.98
0.96
0.94
0.92
0.90
0.88
0.86
0.84
0
10
20
30
40
50
Epoch
Train Accuracy
Validation Accuracy



In [27]:

```
score_3d = model_3d.evaluate(X3_test, y3_test, verbose=0)
print(f"3D CNN Test Loss: {score_3d[0]:.4f}  Test Accuracy: {score_3d[1]:.4f}")

y3_test_labels = np.argmax(y3_test, axis=1)
y3_pred = model_3d.predict(X3_test)
y3_pred_labels = np.argmax(y3_pred, axis=1)
print(classification_report(y3_test_labels, y3_pred_labels, target_names=[f"C{i+1}" for i in range(num_classes)]))
```

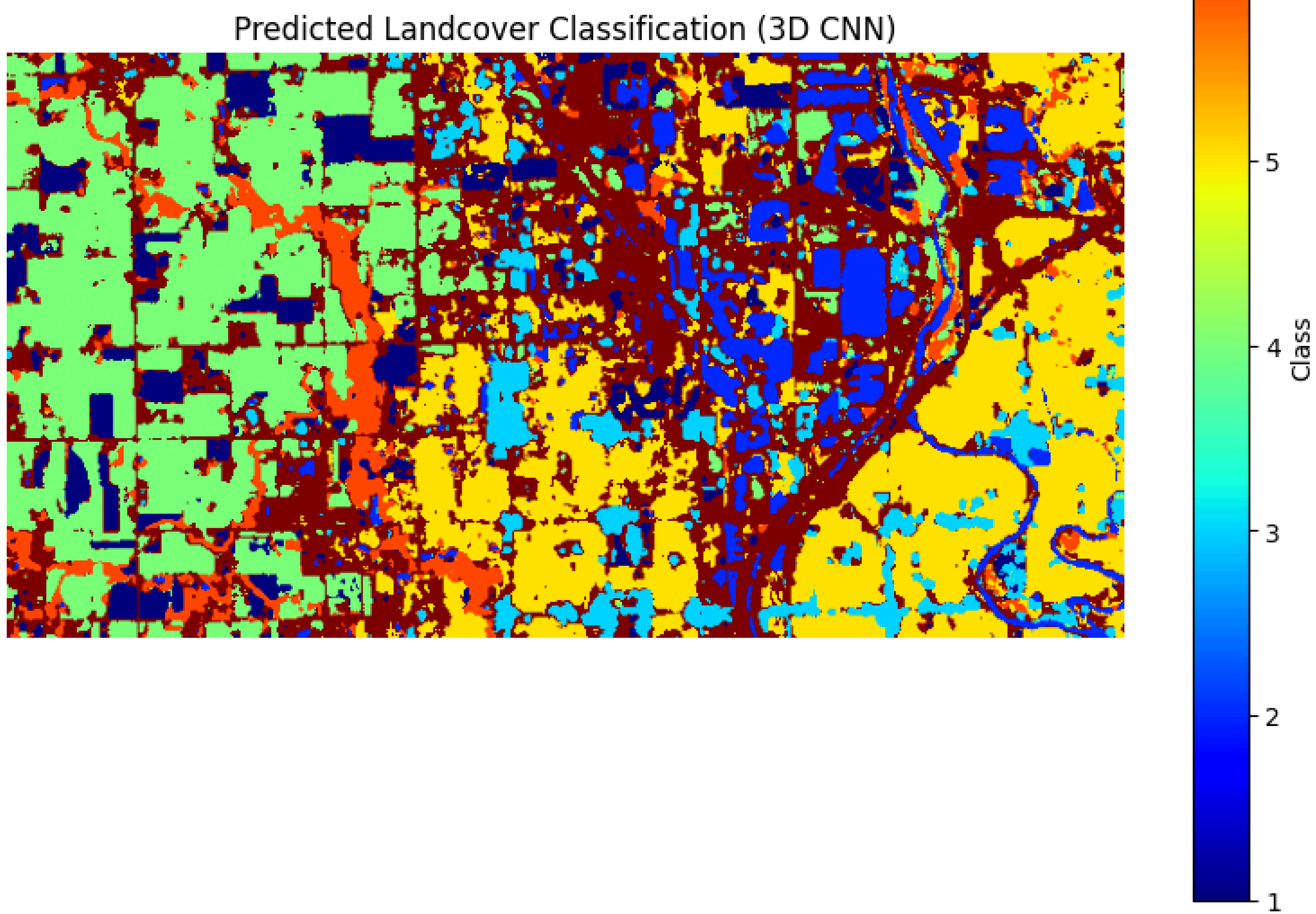
```
3D CNN Test Loss: 0.0412  Test Accuracy: 0.9849
390/390 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
              precision    recall  f1-score   support

          C1       1.00      1.00      1.00      1532
          C2       1.00      1.00      1.00      1640
          C3       0.94      0.98      0.96      1399
          C4       0.99      1.00      1.00      3578
          C5       0.99      1.00      0.99      2454
          C6       0.99      0.99      0.99       503
          C7       0.97      0.90      0.93      1368

    accuracy                           0.98     12474
   macro avg       0.98      0.98      0.98     12474
weighted avg       0.99      0.98      0.98     12474
```

In [26]:

```python
def classify_full_image_3d(data_img, model, patch_size):
    half_patch = patch_size // 2
    # Pad the image to allow patch extraction for boundary pixels (reflect padding)
    padded = np.pad(data_img, ((half_patch, half_patch), (half_patch, half_patch), (0, 0)), mode='reflect')
    # Add a batch dimension for TensorFlow processing
    padded_tensor = tf.convert_to_tensor(padded[np.newaxis, ...], dtype=tf.float32)

    # Use tf.image.extract_patches to extract patches for every pixel
    patches_tf = tf.image.extract_patches(
        images=padded_tensor,
        sizes=[1, patch_size, patch_size, 1],
        strides=[1, 1, 1, 1],
        rates=[1, 1, 1, 1],
        padding='VALID'
    )
    # Reshape patches: originally flattened, reshape to (n_pixels, patch_size, patch_size, n_components)
    patches_shape = patches_tf.shape
    patches_reshaped = tf.reshape(patches_tf, (-1, patch_size, patch_size, data_img.shape[2]))

    # For 3D CNN, add a channel dimension: shape becomes (n_pixels, patch_size, patch_size, n_components, 1)
    patches_reshaped = tf.expand_dims(patches_reshaped, axis=-1)

    # Predict classes for all patches
    preds = model.predict(patches_reshaped, batch_size=256)
```

```
    pred_labels = np.argmax(preds, axis=1)

    # Reshape predictions to the original image spatial dimensions
    pred_map = pred_labels.reshape(data_img.shape[0], data_img.shape[1])
    return pred_map

# Run full image classification using the 3D CNN
classified_map_3d = classify_full_image_3d(data_pca_image, model_3d, patch_size)
# If training labels were zero-indexed, shift by 1 for display
classified_map_3d_disp = classified_map_3d + 1

plt.figure(figsize=(10, 8))
plt.imshow(classified_map_3d_disp, cmap='jet')
plt.title("Predicted Landcover Classification (3D CNN)")
plt.axis('off')
plt.colorbar(label="Class")
plt.show()
```

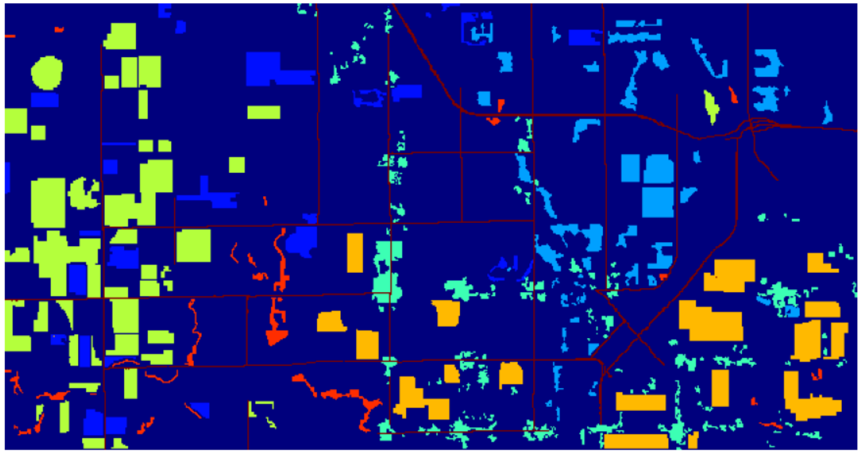
```
864/864 ━━━━━━━━━━━━━━━━━━━━ 14s 16ms/step
```



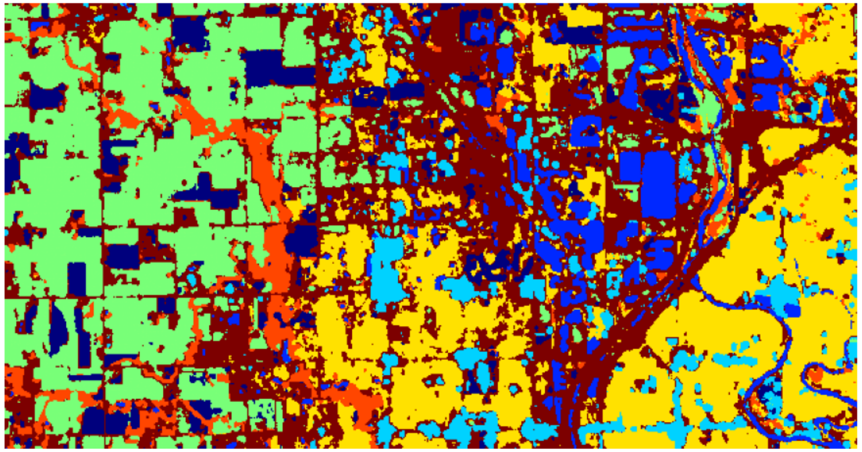
Predicted Landcover Classification (3D CNN)
Class
7
6
5
4
3
2
1

In [29]:
```python
# For comparison, assume:
# classified_map_1d_disp: output from the 1D CNN (with labels shifted by +1)
# classified_map_2d_disp: output from the 2D CNN (with labels shifted by +1)
# classified_map_3d_disp: output from the 3D CNN (computed above)
# gt: ground truth image (with original labeling)

# Example composite visualization
plt.figure(figsize=(20, 10))

plt.subplot(2, 2, 1)
plt.imshow(gt, cmap='jet')
plt.title("Ground Truth")
plt.axis('off')

'''plt.subplot(2, 2, 2)
plt.imshow(classified_map_1d_disp, cmap='jet')
plt.title("Predicted - 1D CNN")
plt.axis('off')

plt.subplot(2, 2, 3)
plt.imshow(classified_map_2d_disp, cmap='jet')
plt.title("Predicted - 2D CNN")
plt.axis('off')'''

plt.subplot(2, 2, 4)
plt.imshow(classified_map_3d_disp, cmap='jet')
plt.title("Predicted - 3D CNN")
plt.axis('off')

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

Ground Truth



Predicted - 3D CNN



```
In [ ]:
```