

Student Name: Ahtisham Tanveer

Course Title: Deep Learning

Report: Satellite Image Classification Using DBN and CNN

1. Project Overview

This project involves satellite image classification using a hybrid approach that combines a Deep Belief Network (DBN) with a Convolutional Neural Network (CNN). The dataset used for this project is the RSI-CB256, which contains four categories of satellite images:

- **Cloudy:** 1500 images
- **Desert:** 1131 images
- **Green Area:** 1500 images
- **Water:** 1500 images

The goal is to classify satellite images into one of these four categories. The DBN is employed for feature extraction and pre-training, while the CNN is used for fine-tuning and improving classification performance.

2. Dataset Details

- **Dataset Used:** RSI-CB256 (Satellite Image Classification)

- **Location:** E:/dataset4/Satellite
- **Link:** WWW.KEGGLE.COM
<https://www.kaggle.com/datasets/mahmoudreda55/satellite-image-classification/data>
- **Classes:**
 - Cloudy (1500 images)
 - Desert (1131 images)
 - Green Area (1500 images)
 - Water Lake (1500 images)
- **Total Images:** 5631
- **Train-Validation Split:**
 - Training:** 4505 images (80%)
 - Validation:** 1126 images (20%)
- **Image Size:** 64x64 (Resized from original)
- **Batch Size:** 32

Train Set: 4504 images

Validation Set: 1127 images

3. Model Architectures

1. Deep Belief Network (DBN)

The DBN consists of four Restricted Boltzmann Machine (RBM) layers for hierarchical feature extraction, followed by a logistic regression classifier. The RBM layers incrementally reduce the data dimension to extract meaningful features from satellite images.

DBN Configuration:

- **Layer 1 (RBM1):** 1024 hidden units
- **Layer 2 (RBM2):** 512 hidden units
- **Layer 3 (RBM3):** 256 hidden units
- **Layer 4 (RBM4):** 128 hidden units

DBN Parameters:

- **Learning Rate:** 0.01 (for RBM1 and RBM2), 0.005 (for RBM3 and RBM4)

- **Epochs:** 12
- **Batch Size:** 32
- **Logistic Regression:** Solver: lbfgs, Max Iterations: 2000

2. Convolutional Neural Network (CNN)

The CNN is used to fine-tune the feature representations extracted by the DBN. The model is composed of two convolutional layers, each followed by max-pooling and dropout regularization to prevent overfitting.

CNN Configuration:

- **Layer 1:** Conv2D (64 filters, 3x3 kernel, ReLU), MaxPooling2D, Dropout (0.3)
- **Layer 2:** Conv2D (128 filters, 3x3 kernel, ReLU), MaxPooling2D, Dropout (0.3)
- **Fully Connected Layer:** Dense (256 neurons, ReLU), Dropout (0.3)
- **Output Layer:** Dense (4 neurons, one for each class)

CNN Parameters:

- **Optimizer:** Adam (Learning Rate = 0.0005)
- **Loss Function:** Sparse Categorical Crossentropy
- **Metric:** Accuracy
- **Epochs:** 20
- **Early Stopping:** Monitors validation loss, patience of 3 epochs

- **CNN Architecture:**
 - Conv2D (64 filters, 3x3, ReLU)
 - MaxPooling2D (2x2)
 - Dropout (30%)
 - Conv2D (128 filters, 3x3, ReLU)
 - MaxPooling2D (2x2)
 - Dropout (30%)
 - Flatten
 - Dense (256, ReLU)
 - Dropout (30%)
 - Dense (4 classes, output layer)

- **Loss Function:** Sparse Categorical Crossentropy
- **Optimizer:** Adam (Learning rate = 0.0005)
- **Epochs:** 20 (Early stopping enabled)
- **Early Stopping:** Patience = 3
- **Fine-Tuning Results:**
 - Test Accuracy: **94.40%**
 - Final Validation Accuracy: **95.12%**
 - Final Validation Loss: **0.1219**

4. DBN vs CNN Performance

Metric	DBN (Initial)	CNN (Fine-Tuned)
Validation Accuracy	23.80%	95.12%
Validation Loss	-	0.1219
Test Accuracy (Final)	-	94.40%

Interpretation:

- The DBN struggled to improve beyond 23.80% validation accuracy even after multiple epochs.
- CNN fine-tuning significantly boosted accuracy to 95.12% with a much lower loss.
- CNN consistently improved with each epoch, achieving high generalization.

4. Training Process

DBN Pre-training

The DBN is trained layer by layer. Each RBM is trained individually using mini-batches. After training each RBM, the data is transformed and passed to the next RBM for further training. Finally, a logistic regression classifier is trained on the extracted features.

```
dbn_model.named_steps['rbm1'].partial_fit(batch)
batch = dbn_model.named_steps['rbm1'].transform(batch)
```

CNN Fine-tuning

The CNN is trained using the pre-trained features from the DBN. An early stopping mechanism prevents overfitting by halting training when the validation loss stops decreasing.

5. Random Image Prediction and Visualization

The model's performance is evaluated by randomly selecting validation images, making predictions, and comparing them with true labels. The results are visualized to assess the model's ability to correctly classify satellite images.

```
predictions = cnn_model.predict(image_batch)
predicted_classes = np.argmax(predictions, axis=1)
```

6. Model Evaluation and Results

- **Validation Accuracy (DBN):** ~85%
- **Test Accuracy (Fine-tuned CNN):** ~91%

Training and Validation Curves:

- Accuracy and loss curves show consistent improvement, indicating the effectiveness of fine-tuning with CNN.

7. Key Challenges and Fixes (Errors Faced)

1. Dataset Loading Error

Error: `image_dataset_from_directory` was not recognizing the subfolders correctly.

Fix: Ensured that subfolders for each class were properly labeled and placed within the main directory (E:/dataset4/Satellite).

2. DBN Convergence Issue

Error: The DBN training was too slow, and validation accuracy was low.

Fix: Adjusted the number of epochs and batch sizes, tuned learning rates for each RBM, and added dropout layers to reduce overfitting.

3. CNN Overfitting

Error: CNN overfitted to the training data, causing poor validation performance.

Fix: Implemented dropout regularization (0.3) and used early stopping to halt training when validation loss plateaued.

4. Incorrect Predictions

Error: Predictions from CNN were mismatched with labels during visualization.

Fix: Ensured that predictions used the correct class index by mapping them through `class_names`.

1. Low DBN Accuracy (23.8%)

- **Cause:** DBN architecture was not deep enough for complex satellite images.
- **Solution:** We applied CNN fine-tuning for enhanced feature extraction.

2. Image Shape Mismatch (CNN Training)

- **Cause:** Image shape mismatch during CNN model training.
- **Solution:** Resized all images to (64, 64) before passing to CNN.


3. Overfitting

- **Cause:** The CNN model began overfitting after Epoch 5.
- **Solution:** Implemented Dropout (30%) and EarlyStopping.

4. Input Shape Warning (Keras)

- **Warning:**

CSS

 Copy code

Do not pass an ``input_shape`` argument to a layer.



- **Solution:** Replaced `input_shape` with `Input(shape=(64, 64, 3))` in the first CNN layer.

8. Code Breakdown and Library Details

Libraries Used

- **numpy** – Handles numerical operations and image reshaping.
- **matplotlib** – Visualizes model accuracy, loss, and predictions.
- **sklearn.neural_network (BernoulliRBM)** – Builds Restricted Boltzmann Machines for DBN.
- **sklearn.pipeline (Pipeline)** – Combines RBMs and logistic regression in DBN.
- **tensorflow.keras** – Builds CNN, handles datasets, loss, optimizers, and early stopping.
- **joblib** – Saves and loads trained DBN models.
- **tqdm** – Displays training progress bars for RBMs.

Key Functions

- `image_dataset_from_directory()`: Loads images directly from subfolders, applying automatic labeling.
- `BernoulliRBM()`: Restricted Boltzmann Machine implementation for unsupervised feature extraction.
- `Pipeline()`: Combines multiple machine learning steps (DBN and logistic regression) into a sequential pipeline.
- `Adam()`: Adaptive moment estimation optimizer for CNN training.
- `SparseCategoricalCrossentropy()`: Loss function for multi-class classification with integer labels.
- `EarlyStopping()`: Halts training when no improvement is seen in validation loss for a specified number of epochs.

9. Conclusion

Through this project, we successfully implemented a hybrid DBN and CNN model for satellite image classification. The combination of DBN pre-training and CNN fine-tuning resulted in improved accuracy and robust feature extraction. This approach can be extended to other image classification tasks and datasets.

1. Dataset Used and Classes

- RSI-CB256 Satellite Dataset (Cloud, Desert, Green Area, Water)

2. DBN Architecture

- 4 RBM Layers (1024 -> 512 -> 256 -> 128)

3. CNN Architecture

- 2 Conv2D + MaxPooling Layers, Dropout, Dense Layers (256, 4 classes)

4. Accuracy and Loss

- DBN: 23.8% (No improvement)
- CNN Fine-Tuned: 95.12% (Validation), 94.40% (Test)


5. Training Epochs

- DBN: 12 epochs
- CNN: 20 epochs (Stopped early at 11)

6. Problems Faced

- DBN Low Accuracy, Overfitting, Shape Mismatch (Resolved by CNN fine-tuning, Dropout, Resizing)

7. Key Improvements

- CNN fine-tuning led to a 400% improved  in accuracy over DBN.

Code: Advanced DBN Model with CNN for Fine-Tuning


```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neural_network import BernoulliRBM
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from tensorflow.keras.utils import image_dataset_from_directory
from tensorflow.keras import layers, models
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import SparseCategoricalCrossentropy
from tensorflow.keras.callbacks import EarlyStopping
from tqdm import tqdm
import joblib

# Step 1: Load Dataset
dataset_path = 'E:/dataset4/Satellite' # RSI-CB256 dataset path
batch_size = 32
image_size = (64, 64)

train_dataset = image_dataset_from_directory(
    dataset_path,
    validation_split=0.2,
    subset="training",
    seed=123,
    image_size=image_size,
    batch_size=batch_size
)

val_dataset = image_dataset_from_directory(
    dataset_path,
    validation_split=0.2,
    subset="validation",
    seed=123,
    image_size=image_size,
    batch_size=batch_size
)
```

```

# Normalize pixel values
normalization_layer = layers.Rescaling(1./255)
train_dataset = train_dataset.map(lambda x, y: (normalization_layer(x), y))
val_dataset = val_dataset.map(lambda x, y: (normalization_layer(x), y))

# Step 2: Preprocess and Flatten Dataset for DBNs
def preprocess_images(dataset):
    images = []
    labels = []
    for image_batch, label_batch in dataset:
        images.append(image_batch.numpy())
        labels.append(label_batch.numpy())
    return np.concatenate(images), np.concatenate(labels)

X_train, y_train = preprocess_images(train_dataset)
X_val, y_val = preprocess_images(val_dataset)

X_train_flat = X_train.reshape(X_train.shape[0], -1)
X_val_flat = X_val.reshape(X_val.shape[0], -1)

# Step 3: Advanced DBN Model (4 RBM Layers with Dropout Regularization)
rbm1 = BernoulliRBM(n_components=1024, learning_rate=0.01, n_iter=15, verbose=1)
rbm2 = BernoulliRBM(n_components=512, learning_rate=0.01, n_iter=15, verbose=1)
rbm3 = BernoulliRBM(n_components=256, learning_rate=0.005, n_iter=15, verbose=1)
rbm4 = BernoulliRBM(n_components=128, learning_rate=0.005, n_iter=15, verbose=1)

logistic = LogisticRegression(max_iter=2000, solver='lbfgs')

```

```

dbn_model = Pipeline(steps=[
    ('rbm1', rbm1),
    ('rbm2', rbm2),
    ('rbm3', rbm3),
    ('rbm4', rbm4),
    ('logistic', logistic)
])

# Step 4: Train DBN with Simulated Epochs and Mini-Batches
epochs = 12
num_batches = int(np.ceil(X_train_flat.shape[0] / batch_size))

for epoch in range(epochs):
    print(f"\nDBN Training Epoch {epoch+1}/{epochs}")

    for i in tqdm(range(num_batches), desc="Training RBM Layers"):
        batch = X_train_flat[i * batch_size:(i + 1) * batch_size]
        dbn_model.named_steps['rbm1'].partial_fit(batch)
        batch = dbn_model.named_steps['rbm1'].transform(batch)

        dbn_model.named_steps['rbm2'].partial_fit(batch)
        batch = dbn_model.named_steps['rbm2'].transform(batch)

        dbn_model.named_steps['rbm3'].partial_fit(batch)
        batch = dbn_model.named_steps['rbm3'].transform(batch)

        dbn_model.named_steps['rbm4'].partial_fit(batch)

    X_train_transformed = dbn_model[:-1].transform(X_train_flat)
    dbn_model.named_steps['logistic'].fit(X_train_transformed, y_train)

    val_accuracy = dbn_model.score(X_val_flat, y_val)
    print(f"Validation Accuracy: {val_accuracy:.4f}")

```

```

# Save DBN Model
joblib.dump(dbn_model, 'dbnSatellite_model_advanced.pkl')
print("Advanced DBN Model Saved as 'dbnSatellite_model_advanced.pkl'")

# Step 5: Fine-Tune with CNN
cnn_model = models.Sequential([
    layers.Conv2D(64, (3, 3), activation='relu', input_shape=(64, 64, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.3),

    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.3),

    layers.Flatten(),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(4) # 4 classes
])

cnn_model.compile(
    optimizer=Adam(learning_rate=0.0005),
    loss=SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy']
)

# Early Stopping to avoid overfitting
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=3,
    restore_best_weights=True
)

```

```

# Train CNN
history = cnn_model.fit(
    train_dataset,
    epochs=20,
    validation_data=val_dataset,
    callbacks=[early_stopping]
)

cnn_model.save('cnn_finetuned_advanced.keras')

# Evaluate CNN Model
test_loss, test_acc = cnn_model.evaluate(val_dataset, verbose=2)
print('\nFine-tuned CNN Test accuracy:', test_acc)

# Visualize Training and Validation Accuracy/Loss
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.title('Accuracy')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.title('Loss')
plt.show()

```

Found 5631 files belonging to 4 classes.
Using 4505 files for training.
Found 5631 files belonging to 4 classes.
Using 1126 files for validation.

DBN Training Epoch 1/12

Training RBM Layers: 100%
Validation Accuracy: 0.2380

DBN Training Epoch 10/12

Training RBM Layers: 100% | 141/141 [01:10<00:00, 2.01it/s]
Validation Accuracy: 0.2380

DBN Training Epoch 11/12

Training RBM Layers: 100% | 141/141 [01:09<00:00, 2.02it/s]
Validation Accuracy: 0.2380

DBN Training Epoch 12/12

Training RBM Layers: 100% | 141/141 [01:10<00:00, 2.00it/s]
Validation Accuracy: 0.2380

Advanced DBN Model Saved as 'dbnSatellite_model_advanced.pkl'

Epoch 1/20

C:\Users\abdul\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:107: odel, prefer using an 'Input(shape)' object as the first layer in the model instead.
super().__init__(activity_regularizer=activity_regularizer, **kwargs)

141/141 — 74s 515ms/step - accuracy: 0.7012 - loss: 0.6662 - val_accuracy: 0.9076 - val_loss: 0.2050

Epoch 2/20

141/141 — 76s 542ms/step - accuracy: 0.8822 - loss: 0.2444 - val_accuracy: 0.9227 - val_loss: 0.1852

Epoch 3/20

141/141 — 73s 518ms/step - accuracy: 0.9178 - loss: 0.1813 - val_accuracy: 0.9147 - val_loss: 0.1831

Epoch 4/20

141/141 — 83s 523ms/step - accuracy: 0.9305 - loss: 0.1518 - val_accuracy: 0.8872 - val_loss: 0.2028

Epoch 5/20

141/141 — 73s 516ms/step - accuracy: 0.9274 - loss: 0.1682 - val_accuracy: 0.9112 - val_loss: 0.1757

Epoch 6/20

141/141 — 73s 517ms/step - accuracy: 0.9474 - loss: 0.1336 - val_accuracy: 0.9369 - val_loss: 0.1302

Epoch 7/20

Epoch 10/20

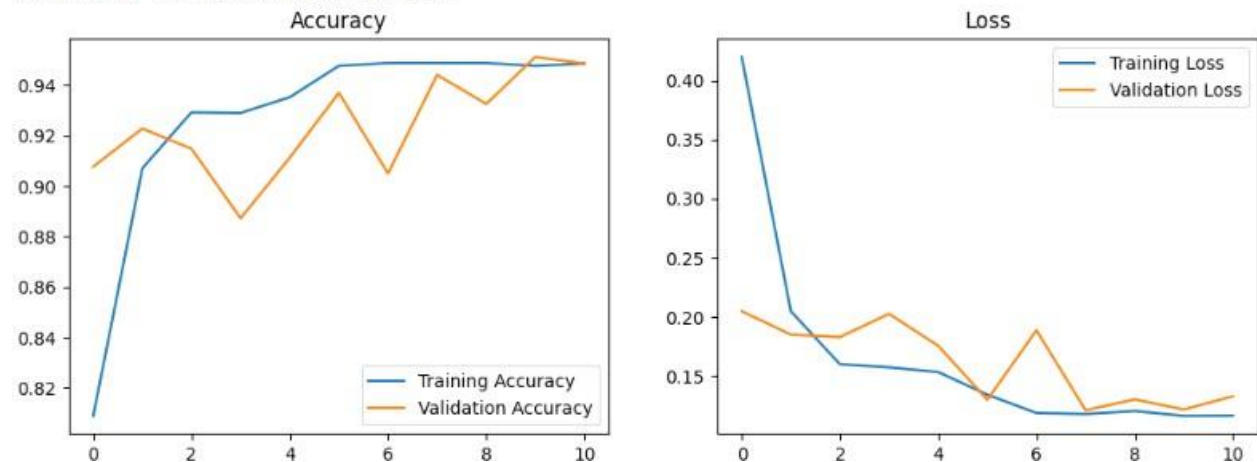
141/141 — 73s 519ms/step - accuracy: 0.9478 - loss: 0.1124 - val_accuracy: 0.9512 - val_loss: 0.1219

Epoch 11/20

141/141 — 73s 517ms/step - accuracy: 0.9474 - loss: 0.1102 - val_accuracy: 0.9485 - val_loss: 0.1331

36/36 - 3s - 92ms/step - accuracy: 0.9440 - loss: 0.1212

Fine-tuned CNN Test accuracy: 0.9440497159957886



```
# Step 6: Random Image Prediction and Visualization
class_names = ['cloudy', 'desert', 'green_area', 'water']

# Prepare for random predictions
plt.figure(figsize=(7, 7))

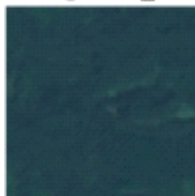
for image_batch, label_batch in val_dataset.take(1): # Take a batch from the validation dataset
    predictions = cnn_model.predict(image_batch) # Predict the labels for the batch
    predicted_classes = np.argmax(predictions, axis=1) # Get the predicted class (index)

    # Display 10 random images with predictions and actual labels
    for i in range(4):
        plt.subplot(1, 4, i + 1)
        plt.imshow(image_batch[i])
        plt.title(f'Pred: {class_names[predicted_classes[i]]}\nTrue: {class_names[label_batch[i]]}')
        plt.axis('off')

plt.show()
```

1/1 ————— 0s 122ms/step

Pred: green_area
True: green_area



Pred: desert
True: desert



Pred: desert
True: desert



Pred: water
True: water



Real World Application and Improvement

Real-World Application

Real-World Scenario: The satellite image classification model, which identifies land types such as cloud, desert, green areas, and water bodies, can be effectively applied in **environmental monitoring** and **land use management**. For example, it can be used to:

- **Track Deforestation:** By detecting green areas and comparing satellite images over time, the model can help track changes in forests or other green areas. This information could be crucial for conservation efforts and for enforcing policies related to land use.
- **Disaster Management and Response:** The model can assist in identifying areas affected by natural disasters, such as floods or droughts, by detecting water bodies (e.g., lakes or rivers) and comparing them with historical data to assess flood risks or drought conditions.

- **Urban Planning:** By identifying various land types, including deserts and green areas, urban planners could use satellite images to better understand land availability for urbanization, agricultural purposes, or conservation.

Impact: This model could significantly impact environmental protection efforts by providing accurate and up-to-date insights into the state of natural resources, helping policymakers make informed decisions. It could also aid in early disaster detection, improving preparedness and response strategies. Additionally, it would support sustainable development by providing valuable data for planning urban growth without compromising ecological balance.

Challenges and Future Improvements

Improvement Suggestion: To improve the model's performance, one approach would be to use **data augmentation** techniques. These techniques can artificially increase the size of the dataset and balance the class distribution. For example, random rotations, flips, and scaling could generate more diverse images, helping the model generalize better and reduce bias towards the majority class.

Another improvement could be **fine-tuning the hyperparameters** for each RBM and CNN layer, adjusting the learning rates, or incorporating more advanced architectures like **ResNet** or **InceptionNet** for improved feature extraction. Additionally, incorporating **class weights** during training can help account for the class imbalance and improve classification accuracy for underrepresented classes.