Crop Yield Prediction: PyTorch 1D-CNN with Optuna (Part 5)

Overview

This notebook trains a **1D Convolutional Neural Network (1D-CNN)** model using PyTorch to predict crop yields. While 1D-CNNs are often used for time-series sequences, they can also extract local feature interactions in tabular data. You can configure the specific target crop in the data loading section.

Methodology

1. **Crop Selection:** Choose the specific crop to predict.
2. **Feature Analysis:** Review the input variables.
3. **Time-Series Split:** Divide data by year to ensure we don't predict the past using the future:
   - **Train:** Learn patterns.
   - **Validation:** Tune settings.
   - **Test:** Final evaluation.
4. **Data Scaling:** Normalize features for Neural Network stability.
5. **Baseline:** Compare against a simple guess (Last Year's Yield).
6. **Initial Model:** Train a default 1D-CNN model and check learning curves for errors.
7. **Optimization:** Use **Optuna** to automatically find the best network architecture (filters, kernel sizes) and hyperparameters.
8. **Final Evaluation:** Compare accuracy (RMSE) across all stages.

In [1]:
```
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import optuna
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.preprocessing import StandardScaler
from sklearn.inspection import permutation_importance

# Optuna Visualization Tools
from optuna.visualization import plot_optimization_history
from optuna.visualization import plot_parallel_coordinate
from optuna.visualization import plot_slice
from optuna.visualization import plot_param_importances

# Set plotting style
sns.set_style("whitegrid")
plt.rcParams['figure.figsize'] = (12, 6)
```

```
# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
```

```
C:\Users\PavinP\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.12_qbz5n
2kfra8p0\LocalCache\local-packages\Python312\site-packages\tqdm\auto.py:21: TqdmW
arning: IProgress not found. Please update jupyter and ipywidgets. See https://ip
ywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
Using device: cpu
```

1. Data Preparation and Crop Choice

We load the main dataset and identify the available crops. For this analysis, we focus specifically on **Rice**. We clean the data by removing columns related to other crops and deleting any rows where the target yield information is missing.

In [2]:

```
# Load dataset
df = pd.read_parquet('Parquet/XY_v2.parquet')

# --- LIST AVAILABLE CROPS ---
# Assumes targets start with 'Y_'
target_columns = [col for col in df.columns if col.startswith('Y_')]
available_crops = [col.replace('Y_', '') for col in target_columns]

print("--- Available Crops found in Dataset ---")
print(available_crops)
print("-" * 40)

# --- CONFIGURATION: SET CROP HERE ---
CHOSEN_CROP = 'rice'   # <--- CHANGE THIS to 'lettuce', 'pepper', etc. based on l
# ------------------------------------

# Define Target and Dynamic Lag Features
TARGET_COL = f'Y_{CHOSEN_CROP}'
LAG_1_FEATURE = f'avg_yield_{CHOSEN_CROP}_1y'

if TARGET_COL not in df.columns:
    raise ValueError(f"Target {TARGET_COL} not found in dataset. Check spelling.

print(f"Predicting Target: {TARGET_COL}")
print(f"Using Lag 1 Feature: {LAG_1_FEATURE}")

# Clean Missing Targets for the chosen crop
df_model = df.dropna(subset=[TARGET_COL])

print(f"Data Loaded. Rows with valid target: {len(df_model)}")
```

```
--- Available Crops found in Dataset ---
['bananas', 'barley', 'cassava_fresh', 'cucumbers_and_gherkins', 'maize_corn', 'o
il_palm_fruit', 'other_vegetables_fresh_nec', 'potatoes', 'rice', 'soya_beans',
'sugar_beet', 'sugar_cane', 'tomatoes', 'watermelons', 'wheat']
----------------------------------------
Predicting Target: Y_rice
Using Lag 1 Feature: avg_yield_rice_1y
Data Loaded. Rows with valid target: 4729
```

2. Selecting Features, Splitting, and Scaling Data

We identify the input variables. We split data by year to avoid data leakage. **Crucially**, for Neural Networks, we must scale the data (StandardScaler) so that all features have a mean of 0 and variance of 1, preventing gradient instability.

1D-CNN Note: While we are treating tabular data, we will reshape the tensors later to be `[Batch_Size, Channels, Length]`. Here, we treat each feature as a "time-step" or sequence element with 1 channel, or more commonly for tabular CNNs, treat the row as `[Batch, 1, Features]`.

In [3]:
```
# --- IMPORTS (Add these if not already present) ---
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
import torch
import pandas as pd

# --- DROP UNWANTED COLUMNS ---
# Drop all columns that start with "avg_yield_" but do NOT match the chosen crop
cols_to_drop = [c for c in df_model.columns
                if c.startswith("avg_yield_") and CHOSEN_CROP not in c]

df_model = df_model.drop(columns=cols_to_drop)

# --- FEATURE SELECTION ---
# Select independent variables (exclude 'Y_' columns and metadata)
feature_cols = [c for c in df_model.columns
                if not c.startswith('Y_') and c not in ['area']]

# --- DISPLAY FEATURES TABLE ---
print(f"Total Features Used: {len(feature_cols)}")
print("-" * 30)
feature_preview = pd.DataFrame(feature_cols, columns=['Feature Name']).T
display(feature_preview)

# --- TIME-SERIES SPLIT ---
TRAIN_END_YEAR = 2014
VAL_END_YEAR = 2019

# 1. Training Set (< 2014)
mask_train = df_model['year'] < TRAIN_END_YEAR
X_train_raw = df_model[mask_train][feature_cols]
y_train = df_model[mask_train][TARGET_COL]

# 2. Validation Set (>= 2014 and < 2019)
mask_val = (df_model['year'] >= TRAIN_END_YEAR) & (df_model['year'] < VAL_END_YE
X_val_raw = df_model[mask_val][feature_cols]
y_val = df_model[mask_val][TARGET_COL]

# 3. Test Set (>= 2019)
mask_test = df_model['year'] >= VAL_END_YEAR
X_test_raw = df_model[mask_test][feature_cols]
y_test = df_model[mask_test][TARGET_COL]

# --- IMPUTATION (Handle NaNs before scaling) ---
imputer = SimpleImputer(strategy='mean')  # Or 'median' if data is skewed
```

```
X_train_imputed = pd.DataFrame(imputer.fit_transform(X_train_raw), columns=featu
X_val_imputed = pd.DataFrame(imputer.transform(X_val_raw), columns=feature_cols)
X_test_imputed = pd.DataFrame(imputer.transform(X_test_raw), columns=feature_col

# Optional: Print NaN counts to verify (should be 0 after imputation)
print("NaNs in X_train_imputed:", X_train_imputed.isnull().sum().sum())
print("NaNs in X_val_imputed:", X_val_imputed.isnull().sum().sum())
print("NaNs in X_test_imputed:", X_test_imputed.isnull().sum().sum())

# --- SCALING (Required for Neural Networks) ---
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train_imputed)
X_val = scaler.transform(X_val_imputed)
X_test = scaler.transform(X_test_imputed)

# --- RESHAPE FOR 1D CNN ---
# PyTorch Conv1d expects input shape: (Batch Size, Channels, Sequence Length)
# For tabular data, we usually treat 'Channels' as 1 and 'Sequence Length' as nu
X_train_cnn = X_train.reshape(X_train.shape[0], 1, X_train.shape[1])
X_val_cnn = X_val.reshape(X_val.shape[0], 1, X_val.shape[1])
X_test_cnn = X_test.reshape(X_test.shape[0], 1, X_test.shape[1])

# Convert to PyTorch Tensors
X_train_tensor = torch.tensor(X_train_cnn, dtype=torch.float32).to(device)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.float32).view(-1, 1).t

X_val_tensor = torch.tensor(X_val_cnn, dtype=torch.float32).to(device)
y_val_tensor = torch.tensor(y_val.values, dtype=torch.float32).view(-1, 1).to(de
X_test_tensor = torch.tensor(X_test_cnn, dtype=torch.float32).to(device)

print(f"\nTraining Samples   (<{TRAIN_END_YEAR})      : {len(X_train)}")
print(f"Validation Samples ({TRAIN_END_YEAR}-{VAL_END_YEAR - 1}): {len(X_val)}")
print(f"Testing Samples    (>={VAL_END_YEAR})     : {len(X_test)}")
print(f"CNN Input Shape (Batch, Channels, Features): {X_train_tensor.shape}")
```

```
Total Features Used: 23
------------------------------
```

	0	1	2	3	4
Feature Name	year	avg_yield_rice_1y	avg_yield_rice_3y	avg_yield_rice_5y	sum_rain_winter	sum_rair

1 rows × 23 columns

```
NaNs in X_train_imputed: 0
NaNs in X_val_imputed: 0
NaNs in X_test_imputed: 0

Training Samples   (<2014)      : 3579
Validation Samples (2014-2018): 575
Testing Samples    (>=2019)     : 575
CNN Input Shape (Batch, Channels, Features): torch.Size([3579, 1, 23])
```

3. Setting a Baseline

Before using complex AI, we create a simple baseline to measure success. We assume that the yield this year will be exactly the same as last year. We calculate the error (RMSE) of this simple guess to establish a score we must beat.

In [4]:
```
# Baseline: yield(t) = yield(t-1)
# Note: We use the raw dataframe for baseline lag feature access
y_pred_baseline = df_model[mask_test][LAG_1_FEATURE]

# Clean NaNs for metric calculation
mask_valid = ~y_pred_baseline.isna() & ~y_test.isna()
y_test_clean = y_test[mask_valid]
y_pred_clean = y_pred_baseline[mask_valid]

rmse_baseline = np.sqrt(mean_squared_error(y_test_clean, y_pred_clean))
r2_baseline = r2_score(y_test_clean, y_pred_clean)

print(f"Baseline RMSE: {rmse_baseline:.2f}")
```

```
Baseline RMSE: 533.44
```

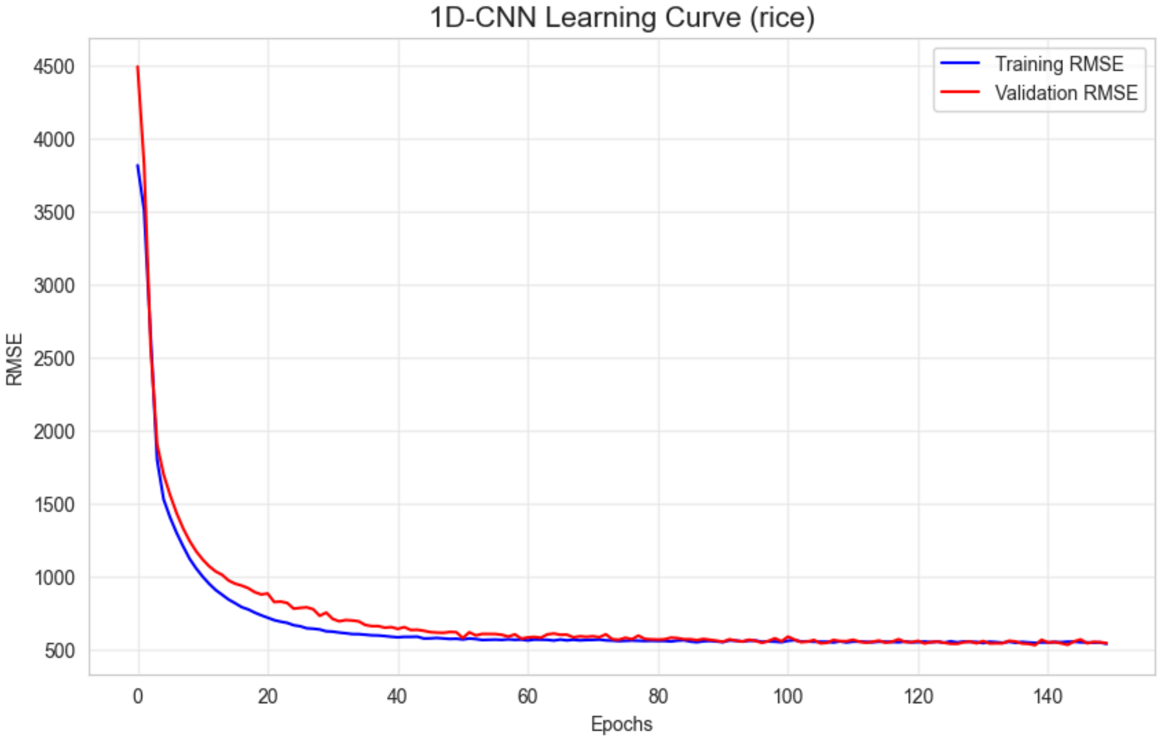
4. Initial Model Testing

We train a basic **1D-CNN** model using standard settings. We plot the training vs validation loss to check for overfitting or underfitting. The architecture involves a convolutional layer, pooling, flattening, and dense layers.

In [5]:
```
# --- DEFINE 1D-CNN STRUCTURE ---
class SimpleCNN1D(nn.Module):
    def __init__(self, input_len):
        super(SimpleCNN1D, self).__init__()
        # Conv1d: in_channels=1 (tabular row), out_channels=32 filters
        self.conv1 = nn.Conv1d(in_channels=1, out_channels=32, kernel_size=3, pa
        self.pool = nn.MaxPool1d(kernel_size=2)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.1)

        # Calculate size after pooling for the linear layer
        # Length after pool = input_len / 2 (floor)
        conv_out_size = input_len // 2 * 32

        self.fc1 = nn.Linear(conv_out_size, 32)
        self.fc2 = nn.Linear(32, 1)

    def forward(self, x):
        # x shape: [batch, 1, features]
        x = self.conv1(x)
        x = self.relu(x)
        x = self.pool(x)
        x = x.view(x.size(0), -1)  # Flatten
        x = self.dropout(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# --- TRAINING HELPER FUNCTION ---
def train_model(model, X_t, y_t, X_v, y_v, lr=0.001, epochs=150, batch_size=32,
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)

    train_loader = DataLoader(TensorDataset(X_t, y_t), batch_size=batch_size, sh
```

```
    train_losses = []
    val_losses = []

    for epoch in range(epochs):
        model.train()
        epoch_loss = 0
        for batch_X, batch_y in train_loader:
            optimizer.zero_grad()
            outputs = model(batch_X)
            loss = criterion(outputs, batch_y)
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item() * batch_X.size(0)

        # Calculate average losses (RMSE representation)
        train_mse = epoch_loss / len(X_t)
        train_rmse = np.sqrt(train_mse)

        model.eval()
        with torch.no_grad():
            val_outputs = model(X_v)
            val_loss = criterion(val_outputs, y_v)
            val_rmse = np.sqrt(val_loss.item())

        train_losses.append(train_rmse)
        val_losses.append(val_rmse)

        if verbose and (epoch % 20 == 0 or epoch == epochs-1):
            print(f"Epoch {epoch}/{epochs} | Train RMSE: {train_rmse:.2f} | Val

    return train_losses, val_losses

# --- INITIAL MODEL TRAINING ---
input_features = X_train_tensor.shape[2] # Sequence length / Num features
model_init = SimpleCNN1D(input_features).to(device)

train_hist, val_hist = train_model(model_init, X_train_tensor, y_train_tensor, X

# --- PLOT LEARNING CURVE ---
plt.figure(figsize=(10, 6))
plt.plot(train_hist, label='Training RMSE', color='blue')
plt.plot(val_hist, label='Validation RMSE', color='red')
plt.title(f'1D-CNN Learning Curve ({CHOSEN_CROP})', fontsize=15)
plt.xlabel('Epochs')
plt.ylabel('RMSE')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

# Evaluate on TEST Set
model_init.eval()
with torch.no_grad():
    y_pred_init_test = model_init(X_test_tensor).cpu().numpy().flatten()

rmse_init_test = np.sqrt(mean_squared_error(y_test, y_pred_init_test))
r2_init_test = r2_score(y_test, y_pred_init_test)

print(f"Initial Model Test RMSE: {rmse_init_test:.2f}")
```

```
Epoch 0/150 | Train RMSE: 3814.55 | Val RMSE: 4491.56
Epoch 20/150 | Train RMSE: 714.66 | Val RMSE: 880.42
Epoch 40/150 | Train RMSE: 580.40 | Val RMSE: 638.03
Epoch 60/150 | Train RMSE: 558.64 | Val RMSE: 579.45
Epoch 80/150 | Train RMSE: 555.16 | Val RMSE: 564.75
Epoch 100/150 | Train RMSE: 556.12 | Val RMSE: 584.91
Epoch 120/150 | Train RMSE: 547.37 | Val RMSE: 555.82
Epoch 140/150 | Train RMSE: 544.54 | Val RMSE: 544.49
Epoch 149/150 | Train RMSE: 535.87 | Val RMSE: 540.17
```




```
Initial Model Test RMSE: 519.88
```

5. Tuning the Model (Optuna)

To improve performance, we use **Optuna** to find the best CNN architecture. We run trials adjusting the number of filters, kernel sizes, dropout rate, learning rate, and batch size.

In [6]:

```
# --- DYNAMIC 1D-CNN BUILDER ---
class DynamicCNN1D(nn.Module):
    def __init__(self, input_len, n_filters, kernel_size, dropout, activation_na
        super(DynamicCNN1D, self).__init__()

        # Choose Activation Function
        if activation_name == "ReLU":
            self.activation = nn.ReLU()
        elif activation_name == "LeakyReLU":
            self.activation = nn.LeakyReLU()
        else:
            self.activation = nn.Tanh()

        # Convolutional Block
        # padding = kernel_size // 2 maintains roughly same size before pooling
        self.conv1 = nn.Conv1d(in_channels=1, out_channels=n_filters,
                               kernel_size=kernel_size, padding=kernel_size//2)
        self.pool = nn.MaxPool1d(kernel_size=2)
        self.dropout_layer = nn.Dropout(dropout)
```

```
        # Calculate Flatten Size
        # Output length after MaxPool1d(2) is floor(input_len / 2)
        # We must handle cases where input_len < 2 properly, though tabular usua
        feature_map_len = input_len // 2
        flat_dim = feature_map_len * n_filters

        # Dense Layers
        self.fc1 = nn.Linear(flat_dim, n_filters)
        self.fc2 = nn.Linear(n_filters, 1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.activation(x)
        x = self.pool(x)
        x = x.view(x.size(0), -1)  # Flatten
        x = self.dropout_layer(x)
        x = self.fc1(x)
        x = self.activation(x)
        x = self.fc2(x)
        return x

# --- OPTUNA OBJECTIVE FUNCTION ---
def objective(trial):
    # 1. Suggest Hyperparameters
    n_filters = trial.suggest_int("n_filters", 16, 128, step=16)
    kernel_size = trial.suggest_categorical("kernel_size", [3, 5, 7])
    dropout = trial.suggest_float("dropout", 0.1, 0.5)
    lr = trial.suggest_float("lr", 1e-4, 1e-2, log=True)
    batch_size = trial.suggest_categorical("batch_size", [16, 32, 64])
    activation = trial.suggest_categorical("activation", ["ReLU", "LeakyReLU", "
    optimizer_name = trial.suggest_categorical("optimizer", ["Adam", "SGD"])

    # 2. Build Model
    # input_features taken from X_train_tensor.shape[2]
    model = DynamicCNN1D(input_features, n_filters, kernel_size, dropout, activa

    # 3. Setup Training
    criterion = nn.MSELoss()
    if optimizer_name == "Adam":
        optimizer = optim.Adam(model.parameters(), lr=lr)
    else:
        optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9)

    train_loader = DataLoader(TensorDataset(X_train_tensor, y_train_tensor),
                              batch_size=batch_size, shuffle=True)

    # 4. Training Loop with Pruning
    epochs = 50  # Reduced slightly for speed during tuning
    for epoch in range(epochs):
        model.train()
        for batch_X, batch_y in train_loader:
            optimizer.zero_grad()
            outputs = model(batch_X)
            loss = criterion(outputs, batch_y)
            loss.backward()
            optimizer.step()

        # Evaluate on Validation
        model.eval()
        with torch.no_grad():
```

```
            val_pred = model(X_val_tensor)
            val_mse = criterion(val_pred, y_val_tensor).item()
            val_rmse = np.sqrt(val_mse)

        # Pruning check
        trial.report(val_rmse, epoch)
        if trial.should_prune():
            raise optuna.exceptions.TrialPruned()

    return val_rmse

# --- RUN OPTIMIZATION ---
study_name = f'{CHOSEN_CROP.capitalize()}_Yield_CNN'
study = optuna.create_study(direction='minimize', study_name=study_name)
study.optimize(objective, n_trials=20)

print("\nBest Parameters found:")
print(study.best_params)
```

```
[I 2025-11-29 18:34:33,791] A new study created in memory with name: Rice_Yield_C
NN
[I 2025-11-29 18:34:49,935] Trial 0 finished with value: 536.6732595350732 and pa
rameters: {'n_filters': 112, 'kernel_size': 7, 'dropout': 0.1421212062282532, 'l
r': 0.0020511032204876224, 'batch_size': 64, 'activation': 'LeakyReLU', 'optimize
r': 'Adam'}. Best is trial 0 with value: 536.6732595350732.
[I 2025-11-29 18:35:04,798] Trial 1 finished with value: 678.2366843514143 and pa
rameters: {'n_filters': 16, 'kernel_size': 5, 'dropout': 0.3853725420394044, 'l
r': 0.002515260728576184, 'batch_size': 32, 'activation': 'LeakyReLU', 'optimize
r': 'Adam'}. Best is trial 0 with value: 536.6732595350732.
[I 2025-11-29 18:35:23,971] Trial 2 finished with value: 2212.1725746424036 and p
arameters: {'n_filters': 80, 'kernel_size': 5, 'dropout': 0.3492427825106077, 'l
r': 0.0071558706264793145, 'batch_size': 32, 'activation': 'Tanh', 'optimizer':
'SGD'}. Best is trial 0 with value: 536.6732595350732.
[W 2025-11-29 18:35:42,067] Trial 3 failed with parameters: {'n_filters': 48, 'ke
rnel_size': 3, 'dropout': 0.3095983982589121, 'lr': 0.0002136195754278501, 'batch
_size': 64, 'activation': 'LeakyReLU', 'optimizer': 'SGD'} because of the followi
ng error: The value nan is not acceptable.
[W 2025-11-29 18:35:42,068] Trial 3 failed with value np.float64(nan).
[I 2025-11-29 18:36:03,582] Trial 4 finished with value: 709.9820860416128 and pa
rameters: {'n_filters': 112, 'kernel_size': 7, 'dropout': 0.42260606592204975, 'l
r': 0.00045458313269396844, 'batch_size': 64, 'activation': 'ReLU', 'optimizer':
'Adam'}. Best is trial 0 with value: 536.6732595350732.
[I 2025-11-29 18:36:19,875] Trial 5 finished with value: 526.5065882778676 and pa
rameters: {'n_filters': 128, 'kernel_size': 7, 'dropout': 0.3736274762664984, 'l
r': 0.0031232313487845526, 'batch_size': 64, 'activation': 'LeakyReLU', 'optimize
r': 'Adam'}. Best is trial 5 with value: 526.5065882778676.
[I 2025-11-29 18:36:20,057] Trial 6 pruned.
[I 2025-11-29 18:36:21,230] Trial 7 pruned.
[I 2025-11-29 18:37:09,457] Trial 8 finished with value: 551.669143373091 and par
ameters: {'n_filters': 112, 'kernel_size': 3, 'dropout': 0.3016947733455931, 'l
r': 0.0035615379396113827, 'batch_size': 16, 'activation': 'ReLU', 'optimizer':
'Adam'}. Best is trial 5 with value: 526.5065882778676.
[I 2025-11-29 18:37:12,192] Trial 9 pruned.
[I 2025-11-29 18:37:12,506] Trial 10 pruned.
C:\Users\PavinP\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.12_qbz5n
2kfra8p0\LocalCache\local-packages\Python312\site-packages\optuna\pruners\_percen
tile.py:21: RuntimeWarning: All-NaN slice encountered
  return np.nanmin(values)
[I 2025-11-29 18:37:13,148] Trial 11 pruned.
[I 2025-11-29 18:37:13,419] Trial 12 pruned.
[I 2025-11-29 18:37:13,650] Trial 13 pruned.
[I 2025-11-29 18:37:24,272] Trial 14 finished with value: 601.4782570883174 and p
arameters: {'n_filters': 96, 'kernel_size': 7, 'dropout': 0.19838977913247122, 'l
r': 0.00909395341677632, 'batch_size': 64, 'activation': 'LeakyReLU', 'optimize
r': 'Adam'}. Best is trial 5 with value: 526.5065882778676.
[I 2025-11-29 18:37:24,456] Trial 15 pruned.
[I 2025-11-29 18:37:53,596] Trial 16 finished with value: 529.7786094209542 and p
arameters: {'n_filters': 96, 'kernel_size': 3, 'dropout': 0.2317765924842522, 'l
r': 0.0015851748113847566, 'batch_size': 16, 'activation': 'LeakyReLU', 'optimize
r': 'Adam'}. Best is trial 5 with value: 526.5065882778676.
[I 2025-11-29 18:37:53,984] Trial 17 pruned.
[I 2025-11-29 18:38:21,546] Trial 18 finished with value: 632.0578494093717 and p
arameters: {'n_filters': 96, 'kernel_size': 3, 'dropout': 0.3397147285785614, 'l
r': 0.003456324496507389, 'batch_size': 16, 'activation': 'LeakyReLU', 'optimize
r': 'Adam'}. Best is trial 5 with value: 526.5065882778676.
[I 2025-11-29 18:38:21,830] Trial 19 pruned.
```

```
Best Parameters found:
{'n_filters': 128, 'kernel_size': 7, 'dropout': 0.3736274762664984, 'lr': 0.00312
32313487845526, 'batch_size': 64, 'activation': 'LeakyReLU', 'optimizer': 'Adam'}
```

6. Visualizing Optimization

We generate charts to understand the tuning process. These visual tools show us which specific settings (like kernel size or number of filters) had the biggest impact on reducing the model's error.

In [7]:
```
# --- OPTUNA VISUALIZATIONS ---
name = f"{CHOSEN_CROP.capitalize()}_Yield_CNN"

# 1. Optimization History
fig = plot_optimization_history(study)
fig.update_layout(title=f'{name} Optimization History', width=900, height=500)
fig.show()

# 2. Parallel Coordinate (Hyperparameter Relationships)
fig = plot_parallel_coordinate(study)
fig.update_layout(title=f'{name} Parallel Coordinate Plot', width=900, height=50
fig.show()

# 3. Slice Plot (Individual Parameter impact)
fig = plot_slice(study)
fig.update_layout(title=f'{name} Slice Plot', width=900, height=500)
fig.show()

# 4. Parameter Importance
try:
    fig = plot_param_importances(study)
    fig.update_layout(title=f'{name} Hyperparameter Importance', width=900, heig
    fig.show()
except (ValueError, RuntimeError) as e:
    print(f'Could not plot parameter importance: {e}')
```

7. Final Model Training

Using the best settings found during the tuning phase, we build the final 1D-CNN. We train this model on both the Training and Validation data combined to maximize learning.

In [8]:
```
# 1. Combine Train + Validation for Final Training
# Combine raw arrays and reshape for CNN [Total Samples, 1, Features]
X_train_full = np.vstack((X_train, X_val))
X_train_full_cnn = X_train_full.reshape(X_train_full.shape[0], 1, X_train_full.s
y_train_full = np.concatenate((y_train, y_val))

# Convert full set to tensor
X_train_full_tensor = torch.tensor(X_train_full_cnn, dtype=torch.float32).to(dev
y_train_full_tensor = torch.tensor(y_train_full, dtype=torch.float32).view(-1, 1

# 2. Retrieve Best Params
bp = study.best_params

# 3. Initialize Best Model
```

```python
final_model = DynamicCNN1D(
    input_features,
    bp['n_filters'],
    bp['kernel_size'],
    bp['dropout'],
    bp['activation']
).to(device)

# 4. Train on Full History
# We train for more epochs since we are using more data
optimizer_name = bp['optimizer']
if optimizer_name == "Adam":
    optimizer = optim.Adam(final_model.parameters(), lr=bp['lr'])
else:
    optimizer = optim.SGD(final_model.parameters(), lr=bp['lr'], momentum=0.9)

criterion = nn.MSELoss()
train_loader = DataLoader(TensorDataset(X_train_full_tensor, y_train_full_tensor
                          batch_size=bp['batch_size'], shuffle=True)

print("Training Final Model...")
final_model.train()
for epoch in range(150):
    for batch_X, batch_y in train_loader:
        optimizer.zero_grad()
        outputs = final_model(batch_X)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()

# 5. Final Prediction on TEST Data
final_model.eval()
with torch.no_grad():
    y_pred_final_test = final_model(X_test_tensor).cpu().numpy().flatten()

rmse_final_test = np.sqrt(mean_squared_error(y_test, y_pred_final_test))
r2_final_test = r2_score(y_test, y_pred_final_test)
```

```
Training Final Model...
```

8. Results and Analysis

We evaluate the final performance on the Test data (2019–2023).

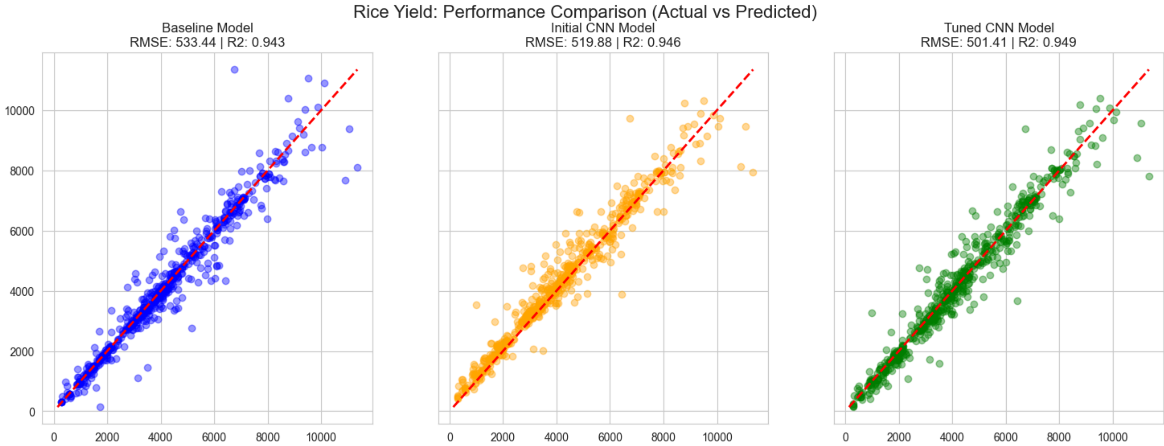
- **Comparison:** We check if the Tuned 1D-CNN beats the Baseline and the Initial Model.
- **Trend Analysis:** We plot the predicted yields against actual yields over time.

In [9]:

```python
# Calculate Improvement %
imp_final = (rmse_baseline - rmse_final_test) / rmse_baseline * 100

print("--- Final Performance Report (Test Set) ---")
print(f"Baseline Model: RMSE={rmse_baseline:.2f}, R2={r2_baseline:.4f}")
print(f"Initial Model:  RMSE={rmse_init_test:.2f}, R2={r2_init_test:.4f}")
print(f"Tuned CNN Model: RMSE={rmse_final_test:.2f}, R2={r2_final_test:.4f} (RMS

# --- PLOTTING RESULTS ---
fig, axes = plt.subplots(1, 3, figsize=(18, 6), sharey=True)
```

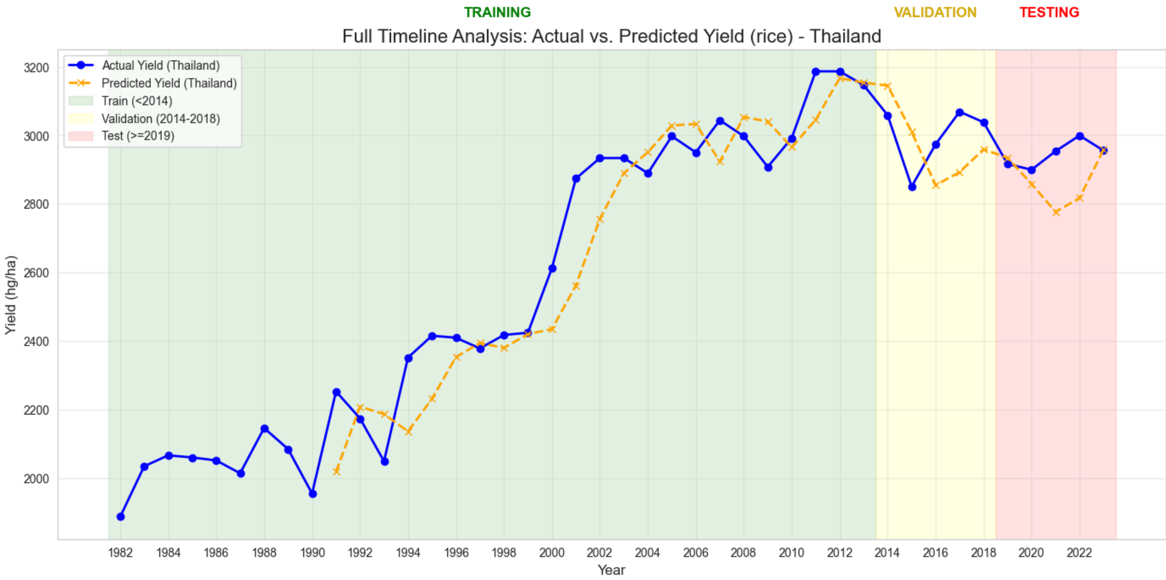
```python
# Axis Limits
all_preds = np.concatenate([y_pred_clean, y_pred_init_test, y_pred_final_test])
all_true = np.concatenate([y_test_clean, y_test, y_test])
min_val, max_val = min(min(all_preds), min(all_true)), max(max(all_preds), max(a

# 1. Baseline Plot
axes[0].scatter(y_test_clean, y_pred_clean, alpha=0.4, color='blue')
axes[0].plot([min_val, max_val], [min_val, max_val], 'r--', linewidth=2)
axes[0].set_title(f'Baseline Model\nRMSE: {rmse_baseline:.2f} | R2: {r2_baseline

# 2. Initial Model Plot
axes[1].scatter(y_test, y_pred_init_test, alpha=0.4, color='orange')
axes[1].plot([min_val, max_val], [min_val, max_val], 'r--', linewidth=2)
axes[1].set_title(f'Initial CNN Model\nRMSE: {rmse_init_test:.2f} | R2: {r2_init

# 3. Tuned Model Plot
axes[2].scatter(y_test, y_pred_final_test, alpha=0.4, color='green')
axes[2].plot([min_val, max_val], [min_val, max_val], 'r--', linewidth=2)
axes[2].set_title(f'Tuned CNN Model\nRMSE: {rmse_final_test:.2f} | R2: {r2_final

plt.suptitle(f'{CHOSEN_CROP.capitalize()} Yield: Performance Comparison (Actual
plt.show()
```

```
--- Final Performance Report (Test Set) ---
Baseline Model: RMSE=533.44, R2=0.9427
Initial Model:  RMSE=519.88, R2=0.9456
Tuned CNN Model: RMSE=501.41, R2=0.9494 (RMSE Improved 6.00%)
```



In [10]:
```python
# --- FULL TIMELINE PLOT (THAILAND ONLY) ---
import matplotlib.pyplot as plt

# 1. Generate Predictions for all data
# Using the scaler and model already fit/loaded
X_all_scaled = scaler.transform(df_model[feature_cols])
# Reshape for CNN [Samples, 1, Features]
X_all_cnn = X_all_scaled.reshape(X_all_scaled.shape[0], 1, X_all_scaled.shape[1]
X_all_tensor = torch.tensor(X_all_cnn, dtype=torch.float32).to(device)

final_model.eval()
with torch.no_grad():
    all_predictions = final_model(X_all_tensor).cpu().numpy().flatten()

# 2. Create DataFrame with 'Area' included
df_full_trend = pd.DataFrame({
    'Year': df_model['year'],
    'Area': df_model['area'], # Added Area column for filtering
```

```python
    'Actual': df_model[TARGET_COL],
    'Predicted': all_predictions
})

# 3. Filter for Thailand Only
TARGET_COUNTRY = 'Thailand'
country_trend = df_full_trend[df_full_trend['Area'] == TARGET_COUNTRY].sort_valu

# Check if data exists for the country
if country_trend.empty:
    print(f"No data found for {TARGET_COUNTRY}. Please check the spelling or cho
else:
    # 4. Plotting
    plt.figure(figsize=(14, 7))

    # Plot Lines for specific country
    plt.plot(country_trend['Year'], country_trend['Actual'],
             marker='o', label=f'Actual Yield ({TARGET_COUNTRY})', linewidth=2,
    plt.plot(country_trend['Year'], country_trend['Predicted'],
             marker='x', linestyle='--', label=f'Predicted Yield ({TARGET_COUNTR

    # Define Split Boundaries based on the global config
    MIN_YEAR = country_trend['Year'].min()
    MAX_YEAR = country_trend['Year'].max()
    train_boundary = TRAIN_END_YEAR - 0.5
    val_boundary = VAL_END_YEAR - 0.5

    # --- Highlight Periods ---\
    # We use a try/except or safe bounds in case the country doesn't have data i
    try:
        plt.axvspan(MIN_YEAR - 0.5, train_boundary, color='green', alpha=0.1, la
        plt.axvspan(train_boundary, val_boundary, color='yellow', alpha=0.1, lab
        plt.axvspan(val_boundary, MAX_YEAR + 0.5, color='red', alpha=0.1, label=

        # Add Text Labels
        y_max = country_trend['Actual'].max()
        text_y = y_max * 1.05

        plt.text((MIN_YEAR + train_boundary)/2, text_y, 'TRAINING', ha='center',
        plt.text((train_boundary + val_boundary)/2, text_y, 'VALIDATION', ha='ce
        plt.text((val_boundary + MAX_YEAR)/2, text_y, 'TESTING', ha='center', fc
    except:
        pass # Skip highlighting if year ranges don't align perfectly with this

    # Final Formatting
    plt.title(f'Full Timeline Analysis: Actual vs. Predicted Yield ({CHOSEN_CROP
    plt.xlabel('Year', fontsize=12)
    plt.ylabel('Yield (hg/ha)', fontsize=12)
    plt.legend(loc='upper left')
    plt.grid(True, alpha=0.3)

    # Ensure integer ticks for years
    plt.xticks(np.arange(int(MIN_YEAR), int(MAX_YEAR) + 1, 2))

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



- **Geographic Error:** We map the error rates by country to see where the model performs best and where it struggles.

In [11]:

```
# --- RE-CREATE COMPARISON DF WITH FEATURE JOINED ---
# We need the original 'area' column from the test set for joining
mask_test = df_model['year'] >= VAL_END_YEAR
test_set_context = df_model[mask_test][['area', 'year']]

comparison_df = pd.DataFrame({
    'Actual_Value': y_test,
    'Predicted_Value': y_pred_final_test
})

# Join works automatically because y_test retained index from original df_model
comparison_df = comparison_df.join(test_set_context)
comparison_df = comparison_df[['year', 'area', 'Actual_Value', 'Predicted_Value'

print("--- Actual vs. Predicted Test Set Results ---")
print(comparison_df.head())
```

```
--- Actual vs. Predicted Test Set Results ---
    year         area  Actual_Value  Predicted_Value
37  2019  Afghanistan        4476.6      4687.828125
38  2020  Afghanistan        4441.7      4757.332031
39  2021  Afghanistan        4406.5      4643.744141
40  2022  Afghanistan        4625.0      4671.758789
41  2023  Afghanistan        4627.9      4611.639648
```

In [12]:

```
import plotly.express as px

# Name Cleaning for Map Plotting
comparison_df['area'] = comparison_df['area'].replace({
    'United_States_of_America': 'United States',
    'United_Kingdom_of_Great_Britain_and_Northern_Ireland': 'United Kingdom',
    'Russian_Federation': 'Russia',
    'Viet_Nam': 'Vietnam',
    'Türkiye': 'Turkey',
    'Bolivia_(Plurinational_State_of)': 'Bolivia',
    'Iran_(Islamic_Republic_of)': 'Iran',
    "Lao_People's_Democratic_Republic": 'Laos',
```

```
    'China,_mainland': 'China',
    'China,_Taiwan_Province_of': 'Taiwan',
    "Democratic_People's_Republic_of_Korea": 'North Korea',
    'Republic_of_Korea': 'South Korea',
    'Côte_d\'Ivoire': "Cote d'Ivoire",
    'United_Republic_of_Tanzania': 'Tanzania',
    'Micronesia_(Federated_States_of)': 'Micronesia',
    'Venezuela_(Bolivarian_Republic_of)': 'Venezuela'
})

def plot_geographic_error(comparison_df):
    # Squared Error (for RMSE)
    comparison_df['Squared_Error'] = (comparison_df['Actual_Value'] - comparison
    # Squared Percentage Error (for RMSPE)
    epsilon = 1e-6
    comparison_df['Squared_Percentage_Error'] = (
        (comparison_df['Actual_Value'] - comparison_df['Predicted_Value']) /
        (comparison_df['Actual_Value'] + epsilon)
    ) ** 2

    # Aggregate Errors by Country
    rmse_df = (
        comparison_df.groupby('area')['Squared_Error']
        .mean().apply(np.sqrt).reset_index()
        .rename(columns={'area': 'Country', 'Squared_Error': 'RMSE'})
    )
    rmspe_df = (
        comparison_df.groupby('area')['Squared_Percentage_Error']
        .mean().apply(np.sqrt).multiply(100).reset_index()
        .rename(columns={'area': 'Country', 'Squared_Percentage_Error': 'RMSPE'}
    )
    ap_df = comparison_df.groupby('area')[['Actual_Value', 'Predicted_Value']].m
    ap_df = ap_df.rename(columns={'area': 'Country'})

    # Merge stats
    error_stats = rmspe_df.merge(rmse_df, on='Country', how='left')
    error_stats = error_stats.merge(ap_df, on='Country', how='left')

    # Plot
    fig = px.choropleth(
        error_stats,
        locations='Country',
        color='RMSPE',
        locationmode='country names',
        color_continuous_scale=['green', 'red'],
        range_color=[0, 50],
        title='Geographic Distribution of Prediction Error (RMSPE) - 1D CNN',
        labels={'RMSPE': 'RMSPE (%)'},
        hover_name='Country',
        hover_data={'RMSPE': ':.2f', 'RMSE': ':.2f', 'Actual_Value': ':.2f', 'Pr
        projection='natural earth'
    )
    fig.update_layout(
        title_font_size=18,
        coloraxis_colorbar=dict(title='RMSPE (%)', orientation='h', len=0.5, yan
        geo=dict(showframe=False, showcoastlines=True, showcountries=True, count
    )
    fig.show()

plot_geographic_error(comparison_df)
```

```
C:\Users\PavinP\AppData\Local\Temp\ipykernel_13772\527999851.py:52: DeprecationWa
rning:

The library used by the *country names* `locationmode` option is changing in an u
pcoming version. Country names in existing plots may not work in the new version.
To ensure consistent behavior, consider setting `locationmode` to *ISO-3*.
```

9. Key Factors (Feature Importance)

1D-CNNs, like basic Neural Networks, don't have built-in feature importance. We calculate **Permutation Importance** by shuffling each feature column one at a time and measuring the error increase. Since the CNN requires 3D input, we create a wrapper to handle the reshaping automatically during the permutation process.

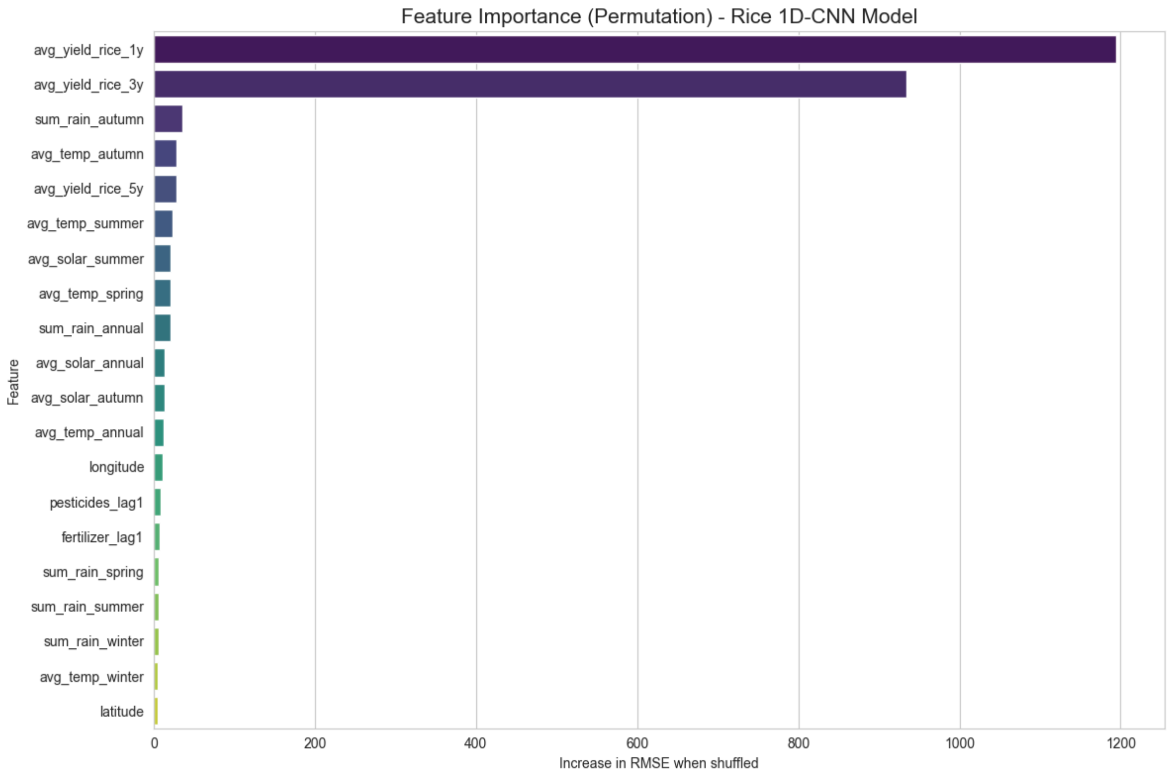
In [13]:

```
# --- WRAPPER FOR SKLEARN COMPATIBILITY (Reshaping logic added) ---
class PyTorchEstimator:
    """Wrapper to make PyTorch model behave like a Sklearn estimator for permuta
    def __init__(self, model, device):
        self.model = model
        self.device = device

    def fit(self, X, y):
        pass # Model is already trained

    def predict(self, X):
        self.model.eval()
        with torch.no_grad():
            # Reshape X from [Batch, Features] to [Batch, 1, Features] for CNN
            X_reshaped = X.reshape(X.shape[0], 1, X.shape[1])
            X_tensor = torch.tensor(X_reshaped, dtype=torch.float32).to(self.dev
            preds = self.model(X_tensor).cpu().numpy().flatten()
        return preds

# --- CALCULATE PERMUTATION IMPORTANCE ---
# Using validation set to gauge importance generalization
wrapped_model = PyTorchEstimator(final_model, device)
# X_val is the 2D array, the wrapper handles reshaping to 3D
results = permutation_importance(wrapped_model, X_val, y_val, scoring='neg_root_

# --- PROCESS RESULTS ---
importance_means = np.abs(results.importances_mean)
feature_names = np.array(feature_cols)

# Create DataFrame
fi_df = pd.DataFrame({
    'Feature': feature_names,
    'Importance': importance_means
}).sort_values(by='Importance', ascending=False).reset_index(drop=True)

# Print Top 20
print("\n--- Top 20 Most Important Features (Permutation Importance) ---")
print(fi_df.head(20))

# --- PLOT ---
plt.figure(figsize=(12, 8))
sns.barplot(x='Importance', y='Feature', data=fi_df.head(20), palette='viridis')
plt.title(f'Feature Importance (Permutation) - {CHOSEN_CROP.capitalize()} 1D-CNN
```

```python
plt.xlabel('Increase in RMSE when shuffled')
plt.ylabel('Feature')
plt.tight_layout()
plt.show()
```

```
--- Top 20 Most Important Features (Permutation Importance) ---
              Feature   Importance
0    avg_yield_rice_1y  1194.937645
1    avg_yield_rice_3y   934.788699
2      sum_rain_autumn    35.348964
3      avg_temp_autumn    28.481342
4    avg_yield_rice_5y    27.978241
5      avg_temp_summer    23.288354
6     avg_solar_summer    21.355493
7      avg_temp_spring    21.320265
8      sum_rain_annual    20.383939
9     avg_solar_annual    13.863126
10    avg_solar_autumn    13.427896
11     avg_temp_annual    12.666036
12           longitude    11.296990
13     pesticides_lag1     9.003609
14     fertilizer_lag1     7.838474
15     sum_rain_spring     6.222551
16     sum_rain_summer     6.221428
17     sum_rain_winter     5.990742
18     avg_temp_winter     4.992779
19            latitude     4.454972
```

```
C:\Users\PavinP\AppData\Local\Temp\ipykernel_13772\3682473412.py:42: FutureWarnin
g:



Passing `palette` without assigning `hue` is deprecated and will be removed in v
0.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same effe
ct.
```


Feature Importance (Permutation) - Rice 1D-CNN Model
avg_yield_rice_1y
avg_yield_rice_3y
sum_rain_autumn
avg_temp_autumn
avg_yield_rice_5y
avg_temp_summer
avg_solar_summer
avg_temp_spring
sum_rain_annual
avg_solar_annual
avg_solar_autumn
avg_temp_annual
longitude
pesticides_lag1
fertilizer_lag1
sum_rain_spring
sum_rain_summer
sum_rain_winter
avg_temp_winter
latitude
Feature
0
200
400
600
800
1000
1200
Increase in RMSE when shuffled