```
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
from sklearn.metrics import r2_score, mean_squared_error
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LassoCV
import pywt
import os
import random
from itertools import product

# ========== 1. Utility Functions ==========
def set_seed(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

def wavelet_denoise_rowwise(row, wavelet='db4', level=1):
    coeff = pywt.wavedec(row, wavelet, level=level)
    coeff[1:] = [pywt.threshold(i, value=0.1 * max(i), mode='soft') for i in coeff[1:]]
    return pywt.waverec(coeff, wavelet)[:len(row)]

def spxy(x, y, test_size=0.3):
    y_backup = y.flatten()
    M = x.shape[0]
    N = round((1 - test_size) * M)
    samples = np.arange(M)
    y_norm = (y - np.mean(y)) / np.std(y)
    D, Dy = np.zeros((M, M)), np.zeros((M, M))
    for i in range(M - 1):
        for j in range(i + 1, M):
            D[i, j] = np.linalg.norm(x[i] - x[j])
            Dy[i, j] = np.linalg.norm(y_norm[i] - y_norm[j])
    D = D / np.max(D) + Dy / np.max(Dy)
    maxD = D.max(axis=0)
    index_row = D.argmax(axis=0)
    index_column = maxD.argmax()
    m = np.zeros(N, dtype=int)
```

```
    m[0], m[1] = index_row[index_column], index_column
    for i in range(2, N):
        pool = np.delete(samples, m[:i])
        dmin = np.array([min(D[min(p, m[k]), max(p, m[k])] for k in range(i)) for p in pool])
        m[i] = pool[np.argmax(dmin)]
    m_complement = np.delete(np.arange(M), m)
    return x[m], x[m_complement], y_backup[m], y_backup[m_complement]

def lasso_selection(X, y):
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)
    lasso = LassoCV(cv=5, max_iter=500000, random_state=42).fit(X_scaled, y.flatten())
    selected_indices = np.where(lasso.coef_ != 0)[0]
    if len(selected_indices) == 0:
        selected_indices = np.arange(min(10, X.shape[1]))
    return X[:, selected_indices], selected_indices

# ========== 2. CNN Model ==========
class ImprovedCNNModel(nn.Module):
    def __init__(self, num_outputs, input_length):
        super().__init__()
        self.pool_layers = self._determine_pool_layers(input_length)
        layers = []
        in_channels = 1
        channels = [16, 32, 64]

        for i in range(3):
            out_channels = channels[i]
            layers.append(nn.Conv1d(in_channels, out_channels, kernel_size=3, padding=1))
            layers.append(nn.BatchNorm1d(out_channels))
            layers.append(nn.ReLU())
            if i < self.pool_layers:
                layers.append(nn.MaxPool1d(kernel_size=2))
            in_channels = out_channels

        self.conv_layers = nn.Sequential(*layers)
        self.fc_input_size = self._get_fc_input_size(input_length)
        self.fc_layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(self.fc_input_size, 256),
            nn.ReLU(),
            nn.Dropout(0.6),
            nn.Linear(256, 128),
            nn.ReLU(),
```

```
                nn.Dropout(0.4),
                nn.Linear(128, num_outputs)
            )

        def forward(self, x):
            x = x.unsqueeze(1)
            x = self.conv_layers(x)
            return self.fc_layers(x)

        def _get_fc_input_size(self, input_length):
            with torch.no_grad():
                self.eval()
                dummy_input = torch.randn(1, 1, input_length)
                out = self.conv_layers(dummy_input)
                return out.view(1, -1).size(1)

        def _determine_pool_layers(self, input_length):
            max_pools = 0
            length = input_length
            while length >= 2 and max_pools < 3:
                length //= 2
                max_pools += 1
            return max_pools

# ========== 3. Main Program ==========
# Modify the paths below to your actual file locations
input_file = r'path/to/your/input_data.xlsx'
output_dir = r'path/to/your/output_folder'
os.makedirs(output_dir, exist_ok=True)

# Data loading and preprocessing
data = pd.read_excel(input_file, header=0)
target = data.iloc[:, 1].values.reshape(-1, 1)
reflectivity = data.iloc[:, 2:].values
reflectivity_denoised = np.apply_along_axis(wavelet_denoise_rowwise, axis=1, arr=reflectivity)
corrected = reflectivity_denoised - np.min(reflectivity_denoised, axis=1, keepdims=True)
reflectivity_scaled = StandardScaler().fit_transform(corrected)

# Grid search parameters
epochs_list = [3000, 4000, 5000]
batch_size_list = [16, 24, 32]
param_combinations = list(product(epochs_list, batch_size_list))

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
global_best = {'score': -1}

for epochs, batch_size in param_combinations:
    print(f"\n{'='*50}")
    print(f'Grid Search: epochs={epochs}, batch_size={batch_size}')
    print('='*50)

    for run in range(10):
        set_seed(42 + run)
        X_train, X_test, y_train, y_test = spxy(reflectivity_scaled, target)
        X_train, selected_features = lasso_selection(X_train, y_train)
        X_test = X_test[:, selected_features]

        X_train_tensor = torch.tensor(X_train, dtype=torch.float32).to(device)
        X_test_tensor = torch.tensor(X_test, dtype=torch.float32).to(device)
        y_train_tensor = torch.tensor(y_train, dtype=torch.float32).to(device).unsqueeze(1)
        y_test_tensor = torch.tensor(y_test, dtype=torch.float32).to(device).unsqueeze(1)

        net = ImprovedCNNModel(1, X_train.shape[1]).to(device)
        loss_fn = nn.MSELoss()
        optimizer = optim.Adam(net.parameters(), lr=0.003, weight_decay=0.001)
        scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5,
patience=10)

        best_loss, no_improve_epochs = float('inf'), 0
        patience = 200
        best_weights = net.state_dict()
        train_loader = DataLoader(TensorDataset(X_train_tensor, y_train_tensor),
batch_size=batch_size, shuffle=True)

        for epoch in range(epochs):
            net.train()
            for Xb, yb in train_loader:
                if Xb.size(0) == 1: continue
                optimizer.zero_grad()
                loss = loss_fn(net(Xb), yb)
                loss.backward()
                optimizer.step()

            net.eval()
            with torch.no_grad():
                train_loss = loss_fn(net(X_train_tensor), y_train_tensor)
                if train_loss < best_loss:
                    best_loss = train_loss
```

```
                    best_weights = net.state_dict()
                    no_improve_epochs = 0
                else:
                    no_improve_epochs += 1
                    if no_improve_epochs >= patience:
                        break
            scheduler.step(train_loss)

        net.load_state_dict(best_weights)
        net.eval()
        with torch.no_grad():
            y_pred_train = net(X_train_tensor).cpu().numpy()
            y_pred_test = net(X_test_tensor).cpu().numpy()
            y_true_train = y_train_tensor.cpu().numpy()
            y_true_test = y_test_tensor.cpu().numpy()

            r2_train = r2_score(y_true_train, y_pred_train)
            r2_test = r2_score(y_true_test, y_pred_test)
            r_train = np.sqrt(r2_train) if r2_train >= 0 else -np.sqrt(-r2_train)
            r_test = np.sqrt(r2_test) if r2_test >= 0 else -np.sqrt(-r2_test)
            rmse_train = np.sqrt(mean_squared_error(y_true_train, y_pred_train))
            rmse_test = np.sqrt(mean_squared_error(y_true_test, y_pred_test))

        # Update global best
        if r_test > global_best['score']:
            global_best = {
                'score': r_test,
                'epochs': epochs,
                'batch_size': batch_size,
                'run': run + 1,
                'train': pd.DataFrame({'True': y_true_train.flatten(), 'Pred': y_pred_train.flatten()}),
                'test': pd.DataFrame({'True': y_true_test.flatten(), 'Pred': y_pred_test.flatten()}),
                'summary': [r_train, r_test, rmse_train, rmse_test]
            }

# Save global best results
with pd.ExcelWriter(os.path.join(output_dir, 'best_predictions.xlsx')) as writer:
    pd.DataFrame([{
        'Best epochs': global_best['epochs'],
        'Best batch_size': global_best['batch_size'],
        'Best run': global_best['run'],
        'Train R': global_best['summary'][0],
        'Test R': global_best['summary'][1],
        'Train RMSE': global_best['summary'][2],
```

```
            'Test RMSE': global_best['summary'][3]
        }]).to_excel(writer, sheet_name='Summary', index=False)
        global_best["train"].to_excel(writer, sheet_name='Train Set', index=False)
        global_best["test"].to_excel(writer, sheet_name='Test Set', index=False)

print("\n" + "="*50)
print("Grid search completed!")
print(f"Best parameters: epochs={global_best['epochs']}, batch_size={global_best['batch_size']},
run={global_best['run']}")
print(f"Best Test R: {global_best['score']:.4f}")
print("="*50)
```