Order Volume Prediction with Cross-Validation

This notebook implements a neural network model for predicting order volumes, with a focus on robust validation through k-fold cross-validation. The implementation includes:

- A deep neural network with batch normalization and dropout for regularization
- Cyclical encoding of temporal features to capture weekly patterns
- K-fold cross-validation to ensure model stability
- Advanced training techniques including learning rate scheduling and early stopping
- Comprehensive model evaluation across multiple metrics

The goal is to create a reliable and robust model that can accurately predict order volumes while avoiding overfitting through careful validation procedures.

Setup

Let's start by importing the necessary libraries. Each library serves a specific purpose in our implementation:

- PyTorch (`torch`): The main deep learning framework
- NumPy and Pandas: For data manipulation and numerical operations
- Scikit-learn: For data preprocessing and model validation tools

In [12]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, SubsetRandomSampler

import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split, KFold
from sklearn.preprocessing import OneHotEncoder, StandardScaler

import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
```

Custom Dataset Implementation

A crucial component of any PyTorch-based machine learning pipeline is the Dataset class. This custom implementation:

1. Inherits from PyTorch's Dataset class

2. Handles conversion of features and targets to PyTorch tensors
3. Provides required methods for data loading

In [2]:
```
class OrderVolumeDataset(Dataset):
    """
    Custom dataset class for handling order volume data.
    Converts features and targets into PyTorch tensors and provides
    necessary methods for DataLoader compatibility.
    """
    def __init__(self, features, targets):
        self.features = torch.FloatTensor(features)
        self.targets = torch.FloatTensor(targets)

    def __len__(self):
        return len(self.features)

    def __getitem__(self, idx):
        return self.features[idx], self.targets[idx]
```

Neural Network Architecture

Our `OrderVolumePredictor` implements a deep neural network with several key features:

1. Multiple hidden layers with increasing then decreasing dimensionality
2. Batch normalization after each linear layer to stabilize training
3. ReLU activation functions for non-linearity
4. Dropout layers for regularization
5. A final output layer for prediction

This architecture is designed to capture complex patterns while preventing overfitting.

In [3]:
```
class OrderVolumePredictor(nn.Module):
    def __init__(self, input_dim, hidden_dims=[64, 128, 256, 128, 64]):
        super(OrderVolumePredictor, self).__init__()

        layers = []
        prev_dim = input_dim

        for hidden_dim in hidden_dims:
            layers.extend([
                nn.Linear(prev_dim, hidden_dim),
                nn.BatchNorm1d(hidden_dim),
                nn.ReLU(),
                nn.Dropout(0.2),  # Increased dropout
            ])
            prev_dim = hidden_dim

        # Add a final layer before output
        layers.extend([
            nn.Linear(prev_dim, 32),
```

```
            nn.BatchNorm1d(32),
            nn.ReLU(),
            nn.Dropout(0.1)
        ])

        layers.append(nn.Linear(32, 1))

        self.model = nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x)
```

Data Preprocessing

The data preparation pipeline includes several sophisticated steps:

1. Loading and cleaning the raw data
2. Converting temporal features (ORDERWEEK) into cyclical representations
3. Encoding categorical variables using one-hot encoding
4. Scaling numerical features and target variables
5. Splitting data into training and test sets

The cyclical encoding of week numbers is particularly important as it helps the model understand the periodic nature of order patterns throughout the year.

In [4]:
```
def prepare_data(csv_path, test_size=0.2, random_state=42):
    """
    Prepare the data for training by reading CSV and preprocessing features
    """
    # Read the CSV file
    df = pd.read_csv(csv_path, encoding='utf-8', quotechar='"', encoding_err

    # Convert ORDERWEEK to a proper date format and extract week number
    df['ORDERWEEK'] = df['ORDERWEEK'].apply(lambda x: int(x.split()[1]))  #

    # Create cyclical features for the week number
    # This helps the model understand the cyclical nature of weeks in a year
    df['WEEK_SIN'] = np.sin(2 * np.pi * df['ORDERWEEK']/53)  # Using 53 week
    df['WEEK_COS'] = np.cos(2 * np.pi * df['ORDERWEEK']/53)

    # Separate features and target
    target = df['ORDERVOLUME'].values

    # Select features for encoding
    categorical_columns = ['CUSTOMER_NAME', 'ORDERTYPE', 'WAREHOUSE', 'CITY'
    numerical_columns = ['WEEK_SIN', 'WEEK_COS']

    # Process categorical features
    categorical_data = df[categorical_columns]
    encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
    categorical_features = encoder.fit_transform(categorical_data)
```

```
    # Process numerical features
    numerical_data = df[numerical_columns]
    scaler_features = StandardScaler()
    numerical_features = scaler_features.fit_transform(numerical_data)

    # Combine all features
    features = np.hstack([categorical_features, numerical_features])

    # Split the data
    X_train, X_test, y_train, y_test = train_test_split(
        features, target, test_size=test_size, random_state=random_state
    )

    # Scale the target variable
    scaler_target = StandardScaler()
    y_train_scaled = scaler_target.fit_transform(y_train.reshape(-1, 1)).rav
    y_test_scaled = scaler_target.transform(y_test.reshape(-1, 1)).ravel()

    return X_train, X_test, y_train_scaled, y_test_scaled, scaler_target
```

In [13]:
```
# Let's load and prepare our data
X_train, X_test, y_train, y_test, scaler = prepare_data('2024_OrderVolume_As

# Display some information about our processed data
print("Data shapes after preprocessing:")
print(f"Training features: {X_train.shape}")
print(f"Test features: {X_test.shape}")
```

```
Data shapes after preprocessing:
Training features: (11172, 1167)
Test features: (2794, 1167)
```

Model Training Implementation

The training procedure incorporates several advanced techniques:

1. AdamW optimizer with weight decay for regularization
2. Learning rate scheduling with ReduceLROnPlateau
3. Huber Loss for robustness against outliers
4. Early stopping to prevent overfitting
5. Model checkpointing to save the best version

These components work together to ensure stable and efficient training.

In [6]:
```
def train_model(model, train_loader, val_loader, num_epochs=150,  # Increase
                learning_rate=0.001,
                patience=15,  # Increased patience
                device='cuda' if torch.cuda.is_available() else 'cpu'):
    """
    Train the neural network with early stopping
    """
    model = model.to(device)
```

```
    # Use AdamW instead of Adam and add weight decay
    optimizer = optim.AdamW(model.parameters(),
                           lr=learning_rate,
                           weight_decay=0.01)  # L2 regularization

    # Modified learning rate scheduler
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(
        optimizer,
        mode='min',
        factor=0.5,
        patience=5,
        min_lr=1e-6,
        verbose=True
    )

    criterion = nn.HuberLoss(delta=1.0)  # Use Huber loss instead of MSE

    print(f"Training on device: {device}")
    model = model.to(device)

    # Early stopping variables
    best_val_loss = float('inf')
    early_stopping_counter = 0

    # Training history
    train_losses = []
    val_losses = []

    for epoch in range(num_epochs):
        # Training phase
        model.train()
        train_loss = 0.0
        for features, targets in train_loader:
            features, targets = features.to(device), targets.to(device)

            optimizer.zero_grad()
            outputs = model(features).squeeze()
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        train_loss /= len(train_loader)
        train_losses.append(train_loss)

        # Validation phase
        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for features, targets in val_loader:
                features, targets = features.to(device), targets.to(device)
                outputs = model(features).squeeze()
                val_loss += criterion(outputs, targets).item()

        val_loss /= len(val_loader)
```

```
        val_losses.append(val_loss)

        # Learning rate scheduling
        scheduler.step(val_loss)

        # Print progress
        if (epoch + 1) % 10 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss:

        # Early stopping check
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            early_stopping_counter = 0
            # Save best model
            torch.save(model.state_dict(), 'best_model.pth')
        else:
            early_stopping_counter += 1
            if early_stopping_counter >= patience:
                print(f'Early stopping triggered after {epoch+1} epochs')
                break

    return train_losses, val_losses
```

K-Fold Cross-Validation

Cross-validation is crucial for:

- Assessing model stability
- Detecting overfitting
- Getting reliable performance estimates

Our implementation uses 5-fold cross-validation, meaning:

1. The data is split into 5 parts
2. The model is trained 5 times, each time using a different fold as validation
3. Performance metrics are averaged across all folds

In [7]:
```
def k_fold_cross_validation(dataset, k_folds, input_dim, batch_size=32, num_
    """
    Perform k-fold cross validation on our OrderVolumePredictor model.

    Args:
        dataset: The full dataset to perform cross-validation on
        k_folds: Number of folds for cross-validation
        input_dim: Input dimension for the model
        batch_size: Batch size for training
        num_epochs: Number of epochs to train each fold

    Returns:
        Dictionary containing training history and performance metrics for e
    """
    # Initialize k-fold cross validation
```

```
    kfold = KFold(n_splits=k_folds, shuffle=True, random_state=42)

    # Store results for each fold
    fold_results = {
        'train_losses': [],
        'val_losses': [],
        'mse_scores': [],
        'rmse_scores': [],
        'mae_scores': []
    }

    # Device configuration
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(f"Using device: {device}")

    # Perform k-fold cross validation
    for fold, (train_ids, val_ids) in enumerate(kfold.split(dataset)):
        print(f'\nFold {fold + 1}/{k_folds}')

        # Create data samplers for obtaining train/validation batches
        train_sampler = SubsetRandomSampler(train_ids)
        val_sampler = SubsetRandomSampler(val_ids)

        # Create data loaders for this fold
        train_loader = DataLoader(
            dataset,
            batch_size=batch_size,
            sampler=train_sampler,
        )
        val_loader = DataLoader(
            dataset,
            batch_size=batch_size,
            sampler=val_sampler,
        )

        # Initialize a fresh model for this fold
        model = OrderVolumePredictor(input_dim).to(device)

        # Train the model on this fold
        train_loss_history, val_loss_history = train_model(
            model=model,
            train_loader=train_loader,
            val_loader=val_loader,
            num_epochs=num_epochs,
            device=device
        )

        # Evaluate model on validation set
        model.eval()
        val_predictions = []
        val_actuals = []

        with torch.no_grad():
            for features, targets in val_loader:
                features, targets = features.to(device), targets.to(device)
                outputs = model(features).squeeze()
```

```
                val_predictions.extend(outputs.cpu().numpy())
                val_actuals.extend(targets.cpu().numpy())

        # Calculate metrics for this fold
        mse = np.mean((np.array(val_predictions) - np.array(val_actuals)) **
        rmse = np.sqrt(mse)
        mae = np.mean(np.abs(np.array(val_predictions) - np.array(val_actual

        # Store results for this fold
        fold_results['train_losses'].append(train_loss_history)
        fold_results['val_losses'].append(val_loss_history)
        fold_results['mse_scores'].append(mse)
        fold_results['rmse_scores'].append(rmse)
        fold_results['mae_scores'].append(mae)

        print(f'Fold {fold + 1} - MSE: {mse:.4f}, RMSE: {rmse:.4f}, MAE: {ma

    # Calculate and print average metrics across all folds
    avg_mse = np.mean(fold_results['mse_scores'])
    avg_rmse = np.mean(fold_results['rmse_scores'])
    avg_mae = np.mean(fold_results['mae_scores'])

    print('\nAverage metrics across all folds:')
    print(f'MSE: {avg_mse:.4f} ± {np.std(fold_results["mse_scores"]):.4f}')
    print(f'RMSE: {avg_rmse:.4f} ± {np.std(fold_results["rmse_scores"]):.4f}
    print(f'MAE: {avg_mae:.4f} ± {np.std(fold_results["mae_scores"]):.4f}')

    return fold_results
```

In [8]:

```
# Create our dataset
full_dataset = OrderVolumeDataset(X_train, y_train)

# Perform cross-validation
input_dim = X_train.shape[1]
fold_results = k_fold_cross_validation(
    dataset=full_dataset,
    k_folds=5,
    input_dim=input_dim,
    batch_size=32,
    num_epochs=100
)
```

```
Using device: cpu

Fold 1/5
/Users/sathwiktoduru/anaconda3/envs/tod-env/lib/python3.11/site-packages/tor
ch/optim/lr_scheduler.py:62: UserWarning: The verbose parameter is deprecate
d. Please use get_last_lr() to access the learning rate.
  warnings.warn(
```

```
Training on device: cpu
Epoch [10/100], Train Loss: 0.1237, Val Loss: 0.1121
Epoch [20/100], Train Loss: 0.1049, Val Loss: 0.1010
Epoch [30/100], Train Loss: 0.0936, Val Loss: 0.1009
Epoch [40/100], Train Loss: 0.0853, Val Loss: 0.1002
Early stopping triggered after 47 epochs
Fold 1 - MSE: 0.3446, RMSE: 0.5870, MAE: 0.2428

Fold 2/5
Training on device: cpu
Epoch [10/100], Train Loss: 0.1294, Val Loss: 0.1131
Epoch [20/100], Train Loss: 0.1063, Val Loss: 0.1089
Epoch [30/100], Train Loss: 0.0903, Val Loss: 0.1023
Epoch [40/100], Train Loss: 0.0846, Val Loss: 0.1043
Epoch [50/100], Train Loss: 0.0804, Val Loss: 0.1071
Early stopping triggered after 54 epochs
Fold 2 - MSE: 0.3667, RMSE: 0.6056, MAE: 0.2461

Fold 3/5
Training on device: cpu
Epoch [10/100], Train Loss: 0.1221, Val Loss: 0.1080
Epoch [20/100], Train Loss: 0.1077, Val Loss: 0.1040
Epoch [30/100], Train Loss: 0.0917, Val Loss: 0.1006
Epoch [40/100], Train Loss: 0.0840, Val Loss: 0.0984
Epoch [50/100], Train Loss: 0.0827, Val Loss: 0.0989
Early stopping triggered after 50 epochs
Fold 3 - MSE: 0.3976, RMSE: 0.6305, MAE: 0.2332

Fold 4/5
Training on device: cpu
Epoch [10/100], Train Loss: 0.1307, Val Loss: 0.1013
Epoch [20/100], Train Loss: 0.1098, Val Loss: 0.0948
Epoch [30/100], Train Loss: 0.0920, Val Loss: 0.0899
Epoch [40/100], Train Loss: 0.0876, Val Loss: 0.0921
Epoch [50/100], Train Loss: 0.0856, Val Loss: 0.0916
Early stopping triggered after 51 epochs
Fold 4 - MSE: 0.2583, RMSE: 0.5082, MAE: 0.2391

Fold 5/5
Training on device: cpu
Epoch [10/100], Train Loss: 0.1310, Val Loss: 0.0931
Epoch [20/100], Train Loss: 0.1061, Val Loss: 0.0921
Epoch [30/100], Train Loss: 0.0973, Val Loss: 0.0845
Epoch [40/100], Train Loss: 0.0898, Val Loss: 0.0873
Early stopping triggered after 46 epochs
Fold 5 - MSE: 0.2910, RMSE: 0.5395, MAE: 0.2208

Average metrics across all folds:
MSE: 0.3316 ± 0.0506
RMSE: 0.5742 ± 0.0445
MAE: 0.2364 ± 0.0089
```

Final Model Evaluation

After cross-validation, we train a final model on the entire training dataset and evaluate it on the held-out test set. This gives us our final performance metrics that we can expect in production.

In [9]:
```
def evaluate_model(model, test_loader, scaler):
    """
    Evaluate the model on the test set and print metrics
    """
    model.eval()
    device = next(model.parameters()).device
    test_predictions = []
    actual_values = []

    with torch.no_grad():
        for features, targets in test_loader:
            features, targets = features.to(device), targets.to(device)
            outputs = model(features).squeeze()
            # Convert predictions back to original scale
            predictions = scaler.inverse_transform(outputs.cpu().numpy().res
            actual = scaler.inverse_transform(targets.cpu().numpy().reshape(
            test_predictions.extend(predictions)
            actual_values.extend(actual)

    # Calculate and print metrics
    mse = np.mean((np.array(test_predictions) - np.array(actual_values)) **
    rmse = np.sqrt(mse)
    mae = np.mean(np.abs(np.array(test_predictions) - np.array(actual_values

    print("\nFinal Test Set Metrics:")
    print(f"MSE: {mse:.4f}")
    print(f"RMSE: {rmse:.4f}")
    print(f"MAE: {mae:.4f}")
```

In [10]:
```
# Train and evaluate final model
final_model = OrderVolumePredictor(input_dim)
final_train_loader = DataLoader(full_dataset, batch_size=32, shuffle=True)
test_dataset = OrderVolumeDataset(X_test, y_test)
test_loader = DataLoader(test_dataset, batch_size=32)

# Train final model
train_model(final_model, final_train_loader, test_loader)

# Evaluate on test set
evaluate_model(final_model, test_loader, scaler)
```

```
Training on device: cpu
Epoch [10/150], Train Loss: 0.1199, Val Loss: 0.1074
Epoch [20/150], Train Loss: 0.0992, Val Loss: 0.1001
Epoch [30/150], Train Loss: 0.0934, Val Loss: 0.1064
Epoch [40/150], Train Loss: 0.0892, Val Loss: 0.0994
Epoch [50/150], Train Loss: 0.0828, Val Loss: 0.1053
Early stopping triggered after 50 epochs

Final Test Set Metrics:
MSE: 4.3037
RMSE: 2.0745
MAE: 0.8740
```

Results Analysis

Let's analyze our model's performance across different metrics and visualize the results:

- Cross-validation performance
- Final test set performance
- Model behavior analysis

In [11]:
```python
# Plot cross-validation results
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))
for fold in range(len(fold_results['train_losses'])):
    plt.plot(fold_results['train_losses'][fold], label=f'Fold {fold+1} Trair
    plt.plot(fold_results['val_losses'][fold], label=f'Fold {fold+1} Val')
plt.title('Training and Validation Losses Across Folds')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```