The information provided is quite comprehensive for generating a Python pipeline code for the described research problem. Below is a structured Python code that follows the proposed pipeline for automating and improving crop yield predictions using a deep learning approach.

### Python Code for Crop Yield Prediction Pipeline

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt

# Step 1: Data Preprocessing
def load_and_preprocess_data(file_path):
    # Load the dataset
    data = pd.read_csv(file_path)

    # Handle missing values (imputation)
    data.fillna(data.mean(), inplace=True)

    # Remove duplicates
```

```
    data.drop_duplicates(inplace=True)

    # Identify and treat outliers (using Z-score method)
    from scipy import stats
    z_scores = np.abs(stats.zscore(data.select_dtypes(include=[np.number])))
    data = data[(z_scores < 3).all(axis=1)]

    # Normalize numerical features
    scaler = StandardScaler()
    numerical_features = data.select_dtypes(include=[np.number])
    data[numerical_features.columns] = scaler.fit_transform(numerical_features)

    return data

# Step 2: Model Selection and Architecture
def build_model(input_shape):
    model = keras.Sequential()
    model.add(layers.Input(shape=input_shape))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(1))  # Output layer for yield prediction
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model

# Step 3: Main function to execute the pipeline
def main():
```

```
# Load and preprocess data
file_path = 'path/to/your/crop_yield_data.csv'  # Update with your file path
data = load_and_preprocess_data(file_path)

# Split data into features and target variable
X = data.drop('yield', axis=1)  # Assuming 'yield' is the target variable
y = data['yield']

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Build the model
model = build_model(input_shape=(X_train.shape[1],))

# Step 4: Model Training
history = model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.2)

# Step 5: Evaluation
y_pred = model.predict(X_test)
r2 = r2_score(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print(f'R^2: {r2:.4f}')
print(f'Mean Absolute Error: {mae:.4f}')
print(f'Root Mean Square Error: {rmse:.4f}')
```

```
    # Plot training history
    plt.plot(history.history['loss'], label='train')
    plt.plot(history.history['val_loss'], label='validation')
    plt.title('Model Loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend()
    plt.show()

# Step 6: Deployment (Placeholder for deployment code)
def deploy_model(model):
    # Code for deployment goes here (e.g., saving the model, creating an API, etc.)
    model.save('crop_yield_model.h5')

if __name__ == "__main__":
    main()
```

### Notes:

1. **File Path:** Update the `file_path` variable with the actual path to your CSV file.

2. **Outlier Treatment:** The outlier treatment method used here is based on the Z-score. You may want to adjust this based on your specific dataset and requirements.

3. **Model Training:** The number of epochs and batch size can be adjusted based on the performance and available resources.

4. **Deployment:** The deployment function is a placeholder. You can implement it based on your deployment strategy (e.g., saving the model, creating a REST API, etc.).

This code provides a complete pipeline from data preprocessing to model evaluation, following the steps outlined in your proposed pipeline. If you need any further modifications or additional features, feel free to ask!