```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

# Load dataset
df = pd.read_csv('house-prices.csv')

# Data preprocessing
# Assuming 'Price' is the target column and we convert it to categories
bins = [0, 100000, 300000, 500000, np.inf]
labels = ['Low', 'Medium', 'High', 'Very High']
df['Price Category'] = pd.cut(df['Price'], bins=bins, labels=labels)

# Encoding categorical target variable
label_encoder = LabelEncoder()
df['Price Category'] = label_encoder.fit_transform(df['Price Category'])

# Splitting features and target
X = df.drop(columns=['Price', 'Price Category'])  # Drop target column and original price
y = df['Price Category']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Convert categorical features to numerical using one-hot encoding
X_train = pd.get_dummies(X_train, drop_first=True)
X_test = pd.get_dummies(X_test, drop_first=True)

# Align columns in training and testing sets
# This ensures both sets have the same columns after one-hot encoding
X_train, X_test = X_train.align(X_test, join='outer', axis=1, fill_value=0)

# Impute missing values after one-hot encoding
X_train.fillna(X_train.mean(), inplace=True)
X_test.fillna(X_test.mean(), inplace=True)

# Standardize features
scaler = StandardScaler() # Initialize scaler here
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Define MLP model
def create_mlp(input_dim):
    model = Sequential([
        Dense(128, activation='relu', input_shape=(input_dim,)),
        Dropout(0.3),
        Dense(64, activation='relu'),
        Dropout(0.2),
        Dense(32, activation='relu'),
        Dense(4, activation='softmax')  # 4 output classes
    ])
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model
    # Create and train the model
mlp_model = create_mlp(X_train.shape[1])
history = mlp_model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=50, batch_size=16)

# Fine-tuning with additional training
def fine_tune_model(model, X_train, y_train, X_test, y_test, epochs=20):
    model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=epochs, batch_size=16)
    return model

# Fine-tune the model
mlp_model = fine_tune_model(mlp_model, X_train, y_train, X_test, y_test)

# Evaluate the model
test_loss, test_acc = mlp_model.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_acc:.2f}")
```

```
0
━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.9062 - loss: 0.1982 - val_accuracy: 0.8846 - val_loss: 0.3176
0
━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.9046 - loss: 0.1821 - val_accuracy: 0.8846 - val_loss: 0.3279
0
━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.9078 - loss: 0.1813 - val_accuracy: 0.8846 - val_loss: 0.3426
0
━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.8736 - loss: 0.2117 - val_accuracy: 0.8846 - val_loss: 0.3491
0
━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.8897 - loss: 0.2004 - val_accuracy: 0.8846 - val_loss: 0.3546
0
━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.9018 - loss: 0.1800 - val_accuracy: 0.8846 - val_loss: 0.3652
0
━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.9264 - loss: 0.1618 - val_accuracy: 0.8846 - val_loss: 0.3729
0
━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.9360 - loss: 0.1373 - val_accuracy: 0.8846 - val_loss: 0.3790
0
━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9193 - loss: 0.1282 - val_accuracy: 0.8846 - val_loss: 0.3631
0
━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - accuracy: 0.9398 - loss: 0.1560 - val_accuracy: 0.8846 - val_loss: 0.3785
0
━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.9100 - loss: 0.1438 - val_accuracy: 0.8846 - val_loss: 0.4240
0
━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.9001 - loss: 0.1490 - val_accuracy: 0.8846 - val_loss: 0.4245
0
━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.9185 - loss: 0.1371 - val_accuracy: 0.8846 - val_loss: 0.4431
0
━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.8798 - loss: 0.1507 - val_accuracy: 0.8846 - val_loss: 0.4589
0
━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9211 - loss: 0.1591 - val_accuracy: 0.8846 - val_loss: 0.4434
0
━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9340 - loss: 0.1162 - val_accuracy: 0.8846 - val_loss: 0.4672
0
━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9512 - loss: 0.0990 - val_accuracy: 0.8846 - val_loss: 0.4853
0
━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9161 - loss: 0.1368 - val_accuracy: 0.8846 - val_loss: 0.4721
0
━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9573 - loss: 0.1121 - val_accuracy: 0.8846 - val_loss: 0.4825
0
━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.9064 - loss: 0.1684 - val_accuracy: 0.8846 - val_loss: 0.4990
0
━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9518 - loss: 0.1016 - val_accuracy: 0.8846 - val_loss: 0.5412
0
━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9641 - loss: 0.0661 - val_accuracy: 0.8846 - val_loss: 0.5829
0
━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9623 - loss: 0.0742 - val_accuracy: 0.8846 - val_loss: 0.5504
0
━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9332 - loss: 0.1123 - val_accuracy: 0.8846 - val_loss: 0.5039
```

```
df
```