House price prediction Ai model using pytroch :-

Step 1:- create csv data file house_prices.csv:-

size_sqft,bedrooms,bathrooms,age_years,price

1500,3,2,10,300000

2000,4,3,5,450000

850,2,1,20,150000

1200,3,2,15,250000

1800,4,3,7,400000

950,2,1,30,130000

2200,5,4,2,500000

1100,3,2,18,240000

1750,4,3,12,380000

1400,3,2,8,320000

Step 2 write down code for python file torch-house.py :-

```
import torch

import torch.nn as nn

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler


# 1. Load dataset

data = pd.read_csv("house_prices.csv")


# 2. Features and target

feature_cols = ["size_sqft", "bedrooms", "bathrooms", "age_years"]
```

```
target_col = "price"

X = data[feature_cols].values
y = data[[target_col]].values

# 3. Scale the features and target
scaler_X = StandardScaler()
scaler_y = StandardScaler()

X_scaled = scaler_X.fit_transform(X)
y_scaled = scaler_y.fit_transform(y)

# 4. Convert to PyTorch tensors
X_tensor = torch.tensor(X_scaled, dtype=torch.float32)
y_tensor = torch.tensor(y_scaled, dtype=torch.float32)

# 5. Train/test split
X_train, X_test, y_train, y_test = train_test_split(X_tensor, y_tensor, test_size=0.2, random_state=42)

# 6. Create DataLoader
train_dataset = torch.utils.data.TensorDataset(X_train, y_train)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=2, shuffle=True)

# 7. Define model
class HousePriceModel(nn.Module):
```

```
    def __init__(self, input_dim):
        super().__init__()
        self.linear = nn.Linear(input_dim, 1)

    def forward(self, x):
        return self.linear(x)

model = HousePriceModel(input_dim=X.shape[1])

# 8. Loss and optimizer
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# 9. Training loop
epochs = 300
for epoch in range(epochs):
    for batch_X, batch_y in train_loader:
        predictions = model(batch_X)
        loss = criterion(predictions, batch_y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if (epoch + 1) % 50 == 0:
```

```
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")


# 10. Prediction from user input
print("\n=== House Price Prediction ===")


user_input = []
for col in feature_cols:
    val = float(input(f"Enter {col}: "))
    user_input.append(val)


# 11. Convert user input
user_input_scaled = scaler_X.transform([user_input])
user_tensor = torch.tensor(user_input_scaled, dtype=torch.float32)


# 12. Predict
model.eval()
with torch.no_grad():
    pred_scaled = model(user_tensor)
    pred_price = scaler_y.inverse_transform(pred_scaled.numpy())


print(f"\n Estimated House Price: ${pred_price[0][0]:,.2f}")
```

output:-

```
D:\>cd practice

D:\practice>python   torch-house.py
Epoch [50/300], Loss: 0.0020
Epoch [100/300], Loss: 0.0012
Epoch [150/300], Loss: 0.0027
Epoch [200/300], Loss: 0.0024
Epoch [250/300], Loss: 0.0011
Epoch [300/300], Loss: 0.0016

=== House Price Prediction ===
Enter size_sqft: 1200
Enter bedrooms: 3
Enter bathrooms: 2
Enter age_years: 10

  Estimated House Price: $277,847.09
```

Let's go through the code **line by line** and explain everything in detail, including its **purpose**, **function**, and **importance** in the overall program.

□ Imports

```
import torch
import torch.nn as nn
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

- `torch`: PyTorch's main package — used for creating tensors, models, etc.
- `torch.nn`: Subpackage of PyTorch for building neural network layers.
- `pandas as pd`: Used to read and manipulate tabular data.
- `train_test_split`: Splits dataset into training and testing sets.
- `StandardScaler`: Standardizes features (mean=0, std=1), which helps training.

□ Step 1: Load Dataset

```
data = pd.read_csv("house_prices.csv")
```

- Loads a CSV file named `house_prices.csv` into a DataFrame called `data`.
- Assumes the CSV file has columns like: `size_sqft`, `bedrooms`, `bathrooms`, `age_years`, and `price`.

□ Step 2: Features and Target

```
feature_cols = ["size_sqft", "bedrooms", "bathrooms", "age_years"]
target_col = "price"

X = data[feature_cols].values
y = data[[target_col]].values
```

- `feature_cols`: List of column names used as **input features**.
- `target_col`: Column to predict (`price`).
- `X`: 2D NumPy array of input features.
- `y`: 2D NumPy array (even for single target column).

□ Step 3: Scale Features and Target

```
scaler_X = StandardScaler()
scaler_y = StandardScaler()

X_scaled = scaler_X.fit_transform(X)
y_scaled = scaler_y.fit_transform(y)
```

- `StandardScaler`: Scales data to have mean=0, std=1 (for better training stability).
- `.fit_transform(X)`:
  - `.fit()` calculates mean and std.
  - `.transform()` applies the scaling.
- `X_scaled`, `y_scaled`: Standardized versions of `X` and `y`.

□ Step 4: Convert to PyTorch Tensors

```
X_tensor = torch.tensor(X_scaled, dtype=torch.float32)
y_tensor = torch.tensor(y_scaled, dtype=torch.float32)
```

- Converts NumPy arrays to PyTorch tensors.
- Specifies `dtype=torch.float32` because neural networks work with 32-bit floats.

□ Step 5: Split into Train/Test Sets

```
X_train, X_test, y_train, y_test = train_test_split(X_tensor, y_tensor,
test_size=0.2, random_state=42)
```

- Splits data: 80% for training, 20% for testing.
- `random_state=42` ensures reproducibility (same split every run).

□ Step 6: DataLoader for Batching

```
train_dataset = torch.utils.data.TensorDataset(X_train, y_train)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=2,
shuffle=True)
```

- `TensorDataset`: Combines `X` and `y` into one dataset.
- `DataLoader`: Allows training in small **batches** (`batch_size=2`), which is memory efficient and improves gradient estimation.
- `shuffle=True`: Shuffles data every epoch to prevent overfitting to sequence.

□ Step 7: Define Neural Network Model

```
class HousePriceModel(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.linear = nn.Linear(input_dim, 1)

    def forward(self, x):
        return self.linear(x)

model = HousePriceModel(input_dim=X.shape[1])
```

- Defines a simple **linear regression model** using PyTorch.
- `input_dim=X.shape[1]`: Number of features (e.g., 4).

- `nn.Linear(input_dim, 1)`: Fully connected layer (one output).
- `forward`: Method that runs the model when you call `model(x)`.

□ Step 8: Loss Function and Optimizer

```
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

- `MSELoss`: Mean Squared Error — standard loss for regression problems.
- `Adam`: Adaptive learning rate optimizer, usually performs better than basic SGD.
- `lr=0.01`: Learning rate controls how fast weights update.

□ Step 9: Training Loop

```
epochs = 300
for epoch in range(epochs):
    for batch_X, batch_y in train_loader:
        predictions = model(batch_X)
        loss = criterion(predictions, batch_y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if (epoch + 1) % 50 == 0:
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
```

- Loops over `300` training epochs.
- For each mini-batch:
    - `model(batch_X)`: Makes predictions.
    - `loss = criterion(...)`: Computes how bad predictions are.
    - `optimizer.zero_grad()`: Clears gradients before backpropagation.
    - `loss.backward()`: Computes gradients.
    - `optimizer.step()`: Updates model weights.
- Every 50 epochs, it prints the current loss.

□ Step 10: User Input for Prediction

```
print("\n=== House Price Prediction ===")

user_input = []
for col in feature_cols:
    val = float(input(f"Enter {col}: "))
    user_input.append(val)
```

- Prompts user to enter values for each input feature.
- `float(input(...))`: Converts user input to float.
- Builds a list of inputs in the same order as `feature_cols`.

□ Step 11: Scale and Convert User Input

```
user_input_scaled = scaler_X.transform([user_input])
user_tensor = torch.tensor(user_input_scaled, dtype=torch.float32)
```

- Scales the input using the **same scaler** used during training.
- Converts scaled input into a PyTorch tensor for model prediction.

□ Step 12: Predict and Show Result

```
model.eval()
with torch.no_grad():
    pred_scaled = model(user_tensor)
    pred_price = scaler_y.inverse_transform(pred_scaled.numpy())

print(f"\n□ Estimated House Price: ${pred_price[0][0]:,.2f}")
```

- `model.eval()`: Switches model to evaluation mode (e.g., disables dropout).
- `torch.no_grad()`: Disables gradient tracking (faster and memory-efficient).
- `model(user_tensor)`: Predicts **scaled** price.
- `scaler_y.inverse_transform(...)`: Converts prediction back to **original price** scale.
- `print(...)`: Displays estimated house price nicely formatted.

□ Summary

This code:

- Loads housing data.
- Preprocesses it with scaling and splitting.
- Defines and trains a simple **linear regression model** using PyTorch.
- Allows user input for features.
- Predicts and prints the estimated house price.