Below is a **simple example of training a house price model using a CSV dataset with PyTorch** and Pandas.

We will:

1️⃣ Load CSV data
2️⃣ Prepare features & labels
3️⃣ Train a neural network
4️⃣ Save the model
5️⃣ Use it for prediction

1️⃣ Install required libraries

```
pip install torch pandas scikit-learn
```

```
(env) F:\python demo>pip install torch pandas scikit-learn
```

Libraries used:

- PyTorch – AI model
- Pandas – CSV processing
- scikit-learn – scaling data

F:\>python --version

Python 3.13.3

F:\>python -m venv env

F:\>env\Scripts\activate

(env) F:\>cd python demo

(env) F:\python demo>pip install torch pandas scikit-learn

```
F:\>python --version
Python 3.13.3

F:\>python -m venv env

F:\>env\Scripts\activate

(env) F:\>cd python demo

(env) F:\python demo>pip install torch pandas scikit-learn
```

---

2 Example CSV file

Create **house_data.csv**

```
size,bedrooms,age,price
1000,2,20,200000
1500,3,15,300000
2000,4,10,400000
2500,4,5,500000
1800,3,12,350000
2200,4,8,420000
```

Columns:

```
size      -> sqft
bedrooms  -> number of bedrooms
age       -> house age
price     -> target value
```

3️⃣ Training Script (train.py)

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

# load CSV
data = pd.read_csv("house_data.csv")

# features and label
X = data[['size','bedrooms','age']].values
y = data[['price']].values

# normalize features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# convert to tensors
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32)

# neural network
class HouseModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3,16),
            nn.ReLU(),
            nn.Linear(16,8),
            nn.ReLU(),
            nn.Linear(8,1)
        )

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

model = HouseModel()

loss_fn = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# training loop
epochs = 1000

for epoch in range(epochs):

    pred = model(X)
    loss = loss_fn(pred,y)

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

    if epoch % 100 == 0:
        print("Epoch:",epoch,"Loss:",loss.item())

# save model
```

```
torch.save(model.state_dict(),"house_model.pth")

print("Model saved!")
```

---

4 Prediction Script (predict.py)

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

class HouseModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3,16),
            nn.ReLU(),
            nn.Linear(16,8),
            nn.ReLU(),
            nn.Linear(8,1)
        )

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

# Load trained model
model = HouseModel()
model.load_state_dict(torch.load("house_model.pth"))
model.eval()

# Get user input
size = float(input("Enter house size (sqft): "))
bedrooms = float(input("Enter number of bedrooms: "))
age = float(input("Enter house age (years): "))
```

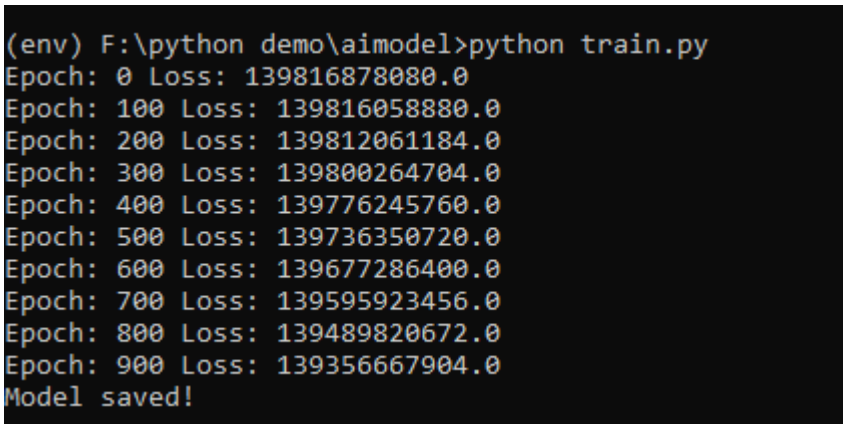
```
# Convert input to tensor

sample = torch.tensor([[size, bedrooms, age]], dtype=torch.float32)


# Predict

price = model(sample)


print("Predicted price:", price.item())
```

5️⃣ Run the training

```
python train.py
```

Output example:

```
Epoch: 0 Loss: 1.2e10
Epoch: 100 Loss: 5.4e8
Epoch: 900 Loss: 1.2e7
Model saved!
```



6️⃣ Run prediction

```
python predict.py
```

Example output:

```
(env) F:\python demo\aimodel>python predict.py
Enter house size (sqft): 2000
Enter number of bedrooms: 2
Enter house age (years): 10
Predicted price: 518503.6875
```

7️⃣ Project structure

```
house_ai/
│
├── house_data.csv
├── train.py
├── predict.py
└── house_model.pth
```

train.py in vs code:-

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

# load CSV
data = pd.read_csv("house_data.csv")

# features and label
X = data[['size','bedrooms','age']].values
y = data[['price']].values

# normalize features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# convert to tensors
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32)

# neural network
class HouseModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3,16),
            nn.ReLU(),
            nn.Linear(16,8),
            nn.ReLU(),
            nn.Linear(8,1)
        )

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

model = HouseModel()

loss_fn = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# training loop
epochs = 1000

for epoch in range(epochs):

    pred = model(X)
```

```
    loss = loss_fn(pred,y)

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

    if epoch % 100 == 0:
        print("Epoch:",epoch,"Loss:",loss.item())

# save model
torch.save(model.state_dict(),"house_model.pth")

print("Model saved!")
```

I'll explain **each line of your code in simple terms**.
Libraries used: PyTorch, Pandas, scikit-learn.

---

1 Import libraries

```
import torch
```

➡ Imports **PyTorch**, the main library used to build and train neural networks.

```
import torch.nn as nn
```

➡ Imports the **neural network module** from PyTorch.
`nn` contains layers like:

- Linear layers
- Activation functions
- Loss functions

```
import torch.optim as optim
```

➡ Imports **optimization algorithms** used to update model weights during training (like Adam, SGD).

```
import pandas as pd
```

➡ Imports **Pandas** for working with datasets and CSV files.

```
from sklearn.preprocessing import StandardScaler
```

➡ Imports **StandardScaler** from scikit-learn to **normalize data** (important for ML models).

---

2 Load CSV dataset

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

➡ Reads the CSV file into a **Pandas DataFrame**.

Example CSV:

```
size,bedrooms,age,price
1000,2,20,200000
1500,3,15,300000
```

Now `data` contains a table.

---

3 Separate features and label

```
X = data[['size','bedrooms','age']].values
```

➡ Selects **input features**.

These are the values the model will use to **predict price**.

```
size
bedrooms
age
```

`.values` converts them into a **NumPy array**.

Example:

```
[[1000,2,20],
 [1500,3,15]]
```

```
y = data[['price']].values
```

➡ Selects the **target value** (what we want to predict).

Example:

```
[[200000],
 [300000]]
```

4️⃣ Normalize the features

```
scaler = StandardScaler()
```

➡ Creates a **scaler object**.

StandardScaler transforms data so it has:

```
mean = 0
standard deviation = 1
```

This helps neural networks train better.

```
X = scaler.fit_transform(X)
```

Two things happen here:

1️⃣ **fit()** → calculates mean & std of dataset
2️⃣ **transform()** → scales the data

Example:

```
Original size: 1000
Scaled size: -1.24
```

5️⃣ Convert data to PyTorch tensors

Neural networks in PyTorch use **tensors instead of arrays**.

```
X = torch.tensor(X, dtype=torch.float32)
```

➡ Converts feature data into a **PyTorch tensor**.

`float32` is required for neural networks.

Example:

```
tensor([[-1.2, -0.8, 0.5]])
```

```
y = torch.tensor(y, dtype=torch.float32)
```

➡ Converts target values into tensors.

6️⃣ Define neural network

```
class HouseModel(nn.Module):
```

➡ Creates a **custom neural network class**.

It inherits from `nn.Module`, which is the base class for all PyTorch models.

```
def __init__(self):
```

➡ Constructor function that runs when the model is created.

```
super().__init__()
```

➡ Calls the parent class (`nn.Module`) constructor.

This initializes the neural network system.

```
self.net = nn.Sequential(
```

➡ Creates a **sequence of layers**.

Data flows through them one by one.

```
nn.Linear(3,16),
```

➡ First neural layer.

```
input features = 3
output neurons = 16
```

Meaning:

```
size
bedrooms
age
```

are converted into **16 hidden features**.

```
nn.ReLU(),
```

➡ Activation function.

ReLU introduces **non-linearity**, allowing the network to learn complex patterns.

Formula:

```
ReLU(x) = max(0,x)
```

```
nn.Linear(16,8),
```

➡ Second layer.

Transforms **16 neurons → 8 neurons**.

```
nn.ReLU(),
```

➡ Another activation layer.

```
nn.Linear(8,1)
```

➡ Final layer.

Produces **1 output value**:

```
Predicted house price
```

---

7 Forward pass

```
def forward(self,x):
```

➡ Defines how data flows through the network.

```
return self.net(x)
```

➡ Sends input through all layers.

8️⃣ Create model instance

```
model = HouseModel()
```

➡ Creates the neural network object.

9️⃣ Define loss function

```
loss_fn = nn.MSELoss()
```

➡ Loss function measures **prediction error**.

MSE = Mean Squared Error

Formula:

```
(predicted - actual)^2
```

Lower loss = better predictions.

🔟 Define optimizer

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

➡ Optimizer updates model weights to reduce loss.

Adam is a **popular optimizer for deep learning**.

```
lr = learning rate
```

Controls how fast the model learns.

1️⃣1️⃣ Training loop

```
epochs = 1000
```

➡ Model will train **1000 times over the dataset**.

```
for epoch in range(epochs):
```

➡ Loop for training.

Each loop = **one training step**.

```
pred = model(X)
```

➡ Model makes predictions using the input data.

```
loss = loss_fn(pred,y)
```

➡ Calculates error between:

```
predicted price
actual price
```

```
optimizer.zero_grad()
```

➡ Clears previous gradients before computing new ones.

```
loss.backward()
```

➡ Backpropagation step.

Calculates **gradients (how weights should change)**.

```
optimizer.step()
```

➡ Updates model weights to reduce loss.

---

1 2 Print progress

```
if epoch % 100 == 0:
```

➡ Every **100 epochs**, print training progress.

```
print("Epoch:",epoch,"Loss:",loss.item())
```

➡ Shows:

```
training step
error value
```

Example:

```
Epoch: 0 Loss: 2.1e10
Epoch: 100 Loss: 5.4e8
```

1️⃣3️⃣ Save trained model

```
torch.save(model.state_dict(),"house_model.pth")
```

➡ Saves learned weights to a file.

File created:

```
house_model.pth
```

This lets you **reuse the model later** without retraining.

```
print("Model saved!")
```

➡ Confirms the model was saved.

🧠 Overall Pipeline

Your code follows this **AI training pipeline**:

```
CSV dataset
     ↓
Load data (Pandas)
     ↓
Feature scaling
     ↓
Convert to tensors
     ↓
Neural network
     ↓
Training loop
     ↓
Loss calculation
     ↓
Weight updates
     ↓
Save trained model
```

predict.py in vs code:-

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

class HouseModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3,16),
            nn.ReLU(),
            nn.Linear(16,8),
            nn.ReLU(),
            nn.Linear(8,1)
        )

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

# Load trained model
model = HouseModel()
model.load_state_dict(torch.load("house_model.pth"))
model.eval()

# Get user input
size = float(input("Enter house size (sqft): "))
bedrooms = float(input("Enter number of bedrooms: "))
age = float(input("Enter house age (years): "))

# Convert input to tensor
sample = torch.tensor([[size, bedrooms, age]], dtype=torch.float32)

# Predict
price = model(sample)

print("Predicted price:", price.item())
```

I'll explain **every line of your code step-by-step** in simple terms.
Libraries used: PyTorch, Pandas, scikit-learn.

---

1 Import libraries

```
import torch
```

Meaning:
Imports the **PyTorch library**, which is used to create and run neural networks.

```
import torch.nn as nn
```

Meaning:
Imports the **neural network module** from PyTorch.

`nn` contains tools like:

- neural network layers
- activation functions
- loss functions

```
import pandas as pd
```

Meaning:
Imports **Pandas**, a library used for handling datasets like CSV files.

(In this prediction code it is not actually used, but it is often used in training.)

```
from sklearn.preprocessing import StandardScaler
```

Meaning:
Imports **StandardScaler** from scikit-learn.

This is used to **normalize data** (make features have similar scale).

Again, not used here but commonly used in training.

---

2 Create the neural network

```
class HouseModel(nn.Module):
```

Meaning:
Creates a **custom neural network class** named `HouseModel`.

It inherits from `nn.Module`, which is the **base class for all PyTorch models**.

```
def __init__(self):
```

Meaning:
Constructor function.
Runs automatically when the model is created.

Used to **define layers of the neural network**.

```
super().__init__()
```

Meaning:
Calls the constructor of the parent class (`nn.Module`).

This initializes PyTorch's neural network system.

```
self.net = nn.Sequential(
```

Meaning:
Creates a **sequence of layers** where data flows from top to bottom.

`Sequential` means:

```
Layer1 → Layer2 → Layer3 → Output
```

```
nn.Linear(3,16),
```

Meaning:
Creates a **fully connected layer**.

```
Input features = 3
Output neurons = 16
```

The inputs are:

```
size
bedrooms
age
```

These are transformed into **16 internal features**.

```
nn.ReLU(),
```

Meaning:
Adds an **activation function**.

ReLU formula:

```
ReLU(x) = max(0, x)
```

Purpose:

- introduces **non-linearity**
- helps neural networks learn complex patterns

```
nn.Linear(16,8),
```

Meaning:
Second neural layer.

```
16 neurons → 8 neurons
```

This reduces the internal representation.

```
nn.ReLU(),
```

Meaning:
Another activation function to help the network learn patterns.

```
nn.Linear(8,1)
```

Meaning:
Final layer.

```
8 neurons → 1 output
```

This output represents:

```
Predicted house price
```

3️⃣ Define forward pass

```
def forward(self,x):
```

Meaning:
Defines **how input data flows through the network**.

PyTorch automatically calls this during prediction.

```
return self.net(x)
```

Meaning:
Passes input `x` through the neural network layers and returns the output.

4️⃣ Load the trained model

```
model = HouseModel()
```

Meaning:
Creates an instance (object) of the neural network.

```
model.load_state_dict(torch.load("house_model.pth"))
```

Meaning:

1️⃣ `torch.load()` loads the saved weights from the file:

```
house_model.pth
```

2️⃣ `load_state_dict()` loads those weights into the model.

So now the model **remembers what it learned during training**.

```
model.eval()
```

Meaning:
Sets the model to **evaluation mode**.

Important because it:

- disables training features
- ensures correct prediction behavior

5️⃣ Get user input

```
size = float(input("Enter house size (sqft): "))
```

Meaning:

1️⃣ Asks the user to enter **house size**.
2️⃣ Converts input text to a **float number**.

Example input:

```
1800
```

```
bedrooms = float(input("Enter number of bedrooms: "))
```

Meaning:
Gets number of bedrooms from the user.

Example:

```
3
```

```
age = float(input("Enter house age (years): "))
```

Meaning:
Gets the **house age**.

Example:

```
12
```

6️⃣ Convert input to tensor

```
sample = torch.tensor([[size, bedrooms, age]], dtype=torch.float32)
```

Meaning:

Creates a **PyTorch tensor** from the user input.

The format must match the training data.

Example result:

```
tensor([[1800., 3., 12.]])
```

The double brackets mean:

```
1 row (one house)
3 features
```

---

7 Make prediction

```
price = model(sample)
```

Meaning:
Sends the input tensor into the neural network.

The model calculates the **predicted price**.

Example output tensor:

```
tensor([[356000.]])
```

---

8 Print prediction

```
print("Predicted price:", price.item())
```

Meaning:

`.item()` converts the tensor value into a normal Python number.

Example output:

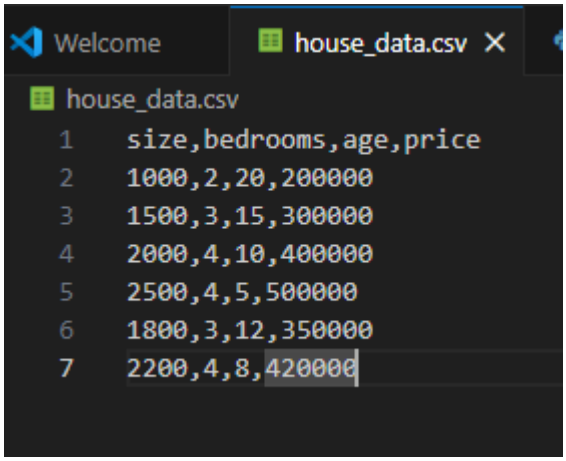
```
Predicted price: 356000
```

Full Flow of the Program

```
User input
   ↓
Convert to tensor
   ↓
Load trained model
   ↓
```

```
Send data into neural network
    ↓
Model predicts price
    ↓
Print predicted price
```

and house_data.csv file :-



Right now **`HouseModel` is written twice** in `train.py` and `predict.py`. That is not good practice. In real projects we **separate the model into its own file** and import it.

This reduces code duplication and makes the project cleaner.

Libraries used: PyTorch, Pandas, scikit-learn.

□ Better Project Structure

```
house_ai/
│
├── model.py
├── train.py
├── predict.py
├── house_data.csv
└── house_model.pth
```

---

1 □ Create `model.py`

This file will contain the **neural network definition only**.

```
import torch
import torch.nn as nn

class HouseModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3,16),
            nn.ReLU(),
            nn.Linear(16,8),
            nn.ReLU(),
            nn.Linear(8,1)
        )

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

Now the model exists **in one place only**.

---

2 Update train.py

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

from model import HouseModel   # import model

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

X = data[['size','bedrooms','age']].values
y = data[['price']].values

scaler = StandardScaler()
X = scaler.fit_transform(X)

X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32)

model = HouseModel()

loss_fn = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

epochs = 1000

for epoch in range(epochs):

    pred = model(X)
    loss = loss_fn(pred,y)

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

    if epoch % 100 == 0:
        print("Epoch:",epoch,"Loss:",loss.item())

torch.save(model.state_dict(),"house_model1.pth")
```

```
print("Model saved!")
```

---

3 Update `predict.py`

```
import torch
from model import HouseModel   # import model

model = HouseModel()
model.load_state_dict(torch.load("house_model1.pth"))
model.eval()

size = float(input("Enter house size (sqft): "))
bedrooms = float(input("Enter number of bedrooms: "))
age = float(input("Enter house age (years): "))

sample = torch.tensor([[size, bedrooms, age]], dtype=torch.float32)

price = model(sample)

print("Predicted price:", price.item())
```

Benefits of this approach

- Model code written **once only**
- Cleaner project structure
- Easier to maintain
- Used in **real machine learning projects**