Project: Neural Network to Predict Boiling Points of Alkanes (Using PyTorch)

Goal: Predict alkane boiling points using molecular features, now with PyTorch instead of TensorFlow.

Step 1: Setup in PyCharm

1. Ensure you’ve installed the necessary libraries:

bash

pip install numpy pandas matplotlib torch scikit-learn

1. Create a new Python file (e.g., boiling\_point\_nn\_pytorch.py).

Step 2: Code Example

python

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

import torch

import torch.nn as nn

import torch.optim as optim

# Step 1: Create the dataset (same as before)

data = {

'n\_carbons': [1, 2, 3, 4, 5, 6, 7, 8],

'mol\_weight': [16.04, 30.07, 44.10, 58.12, 72.15, 86.18, 100.20, 114.23],

'boiling\_point': [111.7, 184.6, 231.1, 272.7, 309.2, 341.9, 371.6, 398.8]

}

df = pd.DataFrame(data)

# Step 2: Prepare data

X = df[['n\_carbons', 'mol\_weight']].values # Features

y = df['boiling\_point'].values # Target

# Split into train/test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Standardize features

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# Convert to PyTorch tensors (like MATLAB matrices, but for PyTorch)

X\_train\_tensor = torch.FloatTensor(X\_train\_scaled)

y\_train\_tensor = torch.FloatTensor(y\_train).view(-1, 1) # Reshape to column vector

X\_test\_tensor = torch.FloatTensor(X\_test\_scaled)

y\_test\_tensor = torch.FloatTensor(y\_test).view(-1, 1)

# Step 3: Define the neural network

class BoilingPointNN(nn.Module):

def \_\_init\_\_(self):

super(BoilingPointNN, self).\_\_init\_\_()

self.layer1 = nn.Linear(2, 16) # Input: 2 features, Output: 16 neurons

self.layer2 = nn.Linear(16, 8) # Hidden layer: 16 -> 8 neurons

self.layer3 = nn.Linear(8, 1) # Output layer: 8 -> 1 (boiling point)

self.relu = nn.ReLU() # ReLU activation function

def forward(self, x):

x = self.relu(self.layer1(x)) # First layer + activation

x = self.relu(self.layer2(x)) # Second layer + activation

x = self.layer3(x) # Output layer (no activation for regression)

return x

# Instantiate the model

model = BoilingPointNN()

# Step 4: Define loss function and optimizer

criterion = nn.MSELoss() # Mean squared error loss

optimizer = optim.Adam(model.parameters(), lr=0.01) # Adam optimizer

# Step 5: Train the model

epochs = 100

train\_losses = []

for epoch in range(epochs):

model.train() # Set model to training mode

optimizer.zero\_grad() # Clear previous gradients

outputs = model(X\_train\_tensor) # Forward pass

loss = criterion(outputs, y\_train\_tensor) # Compute loss

loss.backward() # Backward pass (compute gradients)

optimizer.step() # Update weights

train\_losses.append(loss.item()) # Store loss for plotting

if (epoch + 1) % 20 == 0:

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

# Step 6: Evaluate the model

model.eval() # Set model to evaluation mode

with torch.no\_grad(): # Disable gradient computation

y\_pred\_tensor = model(X\_test\_tensor)

test\_loss = criterion(y\_pred\_tensor, y\_test\_tensor)

print(f'Test Loss (MSE): {test\_loss.item():.2f}')

# Convert predictions to numpy for plotting

y\_pred = y\_pred\_tensor.numpy()

y\_test = y\_test\_tensor.numpy()

# Step 7: Plot results

plt.scatter(y\_test, y\_pred, color='blue', label='Predicted vs Actual')

plt.plot([min(y\_test), max(y\_test)], [min(y\_test), max(y\_test)], 'r--', label='Perfect Fit')

plt.xlabel('Actual Boiling Point (K)')

plt.ylabel('Predicted Boiling Point (K)')

plt.title('Neural Network Prediction of Alkane Boiling Points (PyTorch)')

plt.legend()

plt.show()

# Plot training loss

plt.plot(train\_losses, label='Training Loss')

plt.xlabel('Epoch')

plt.ylabel('Loss (MSE)')

plt.title('Training Loss Over Time')

plt.legend()

plt.show()

Key Differences from TensorFlow

1. PyTorch Tensors:
   * Replace NumPy arrays with torch.FloatTensor for model compatibility (similar to MATLAB’s matrix-to-GPU transition).
   * .view(-1, 1) reshapes tensors, akin to MATLAB’s reshape.
2. Model Definition:
   * PyTorch uses a class-based approach (nn.Module) instead of Keras’s sequential stacking.
   * You explicitly define the forward method, giving more control (MATLAB-like explicitness).
3. Training Loop:
   * PyTorch requires a manual training loop (for epoch in range(epochs)), unlike Keras’s model.fit.
   * optimizer.zero\_grad(), loss.backward(), and optimizer.step() mirror MATLAB’s gradient descent steps but are more transparent.
4. Evaluation:
   * model.eval() and torch.no\_grad() disable training-specific operations, a bit more hands-on than Keras.

Transition Notes

* MATLAB Similarity: PyTorch’s explicit control over layers and training loops feels closer to MATLAB’s manual matrix operations than Keras’s high-level abstraction.
* Flexibility: PyTorch is great for research (e.g., custom loss functions for chemistry problems), which suits your Ph.D. mindset.
* Debugging: PyTorch’s dynamic nature makes it easier to inspect intermediate results, like MATLAB’s interactive workspace.

Next Steps

1. Run it: Test the code and adjust epochs, lr (learning rate), or layer sizes (e.g., nn.Linear(2, 32)).
2. Experiment: Add a feature (e.g., carbon-hydrogen ratio) or try a different optimizer (e.g., optim.SGD).
3. Challenge: Modify the loss function to include a physics-based term (e.g., penalize predictions violating thermodynamic trends).

Let me know how this works for you or if you’d like another twist (e.g., using scikit-learn’s simpler neural nets or a physics-based example)!