explain the code in detail

**ChatGPT said:**

This code implements a **Transformer-based network for Remaining Useful Life (RUL) prediction of lithium-ion batteries**. Below is a **detailed explanation** of each component in the code.

**1. Importing Required Libraries**

python

CopyEdit

import numpy as np

import random

import math

import os

import scipy.io

import torch

import torch.nn as nn

import torch.nn.functional as F

import torchvision

import transformers

import matplotlib.pyplot as plt

%matplotlib inline

from tqdm.notebook import tqdm

from math import sqrt

from datetime import datetime

from sklearn.metrics import mean\_absolute\_error

from sklearn.metrics import mean\_squared\_error

* **numpy**: Used for handling numerical computations.
* **random**: For setting random seeds.
* **math**: Provides mathematical functions.
* **os**: Used for interacting with the operating system (e.g., setting environment variables).
* **scipy.io**: Handles .mat files from the NASA dataset.
* **torch**: PyTorch library for deep learning.
* **torch.nn**: Contains modules for defining neural networks.
* **torch.nn.functional**: Functional API for activation functions and other operations.
* **torchvision**: Used for computer vision tasks.
* **transformers**: Hugging Face library for Transformer-based models.
* **matplotlib.pyplot**: Used for plotting graphs.
* **tqdm.notebook**: Progress bar visualization.
* **datetime**: For working with timestamps.
* **sklearn.metrics**: For evaluating model performance (MAE, RMSE).

**2. Loading Battery Data**

**Convert a List to DateTime Format**

python

CopyEdit

def convert\_to\_time(hmm):

year, month, day, hour, minute, second = int(hmm[0]), int(hmm[1]), int(hmm[2]), int(hmm[3]), int(hmm[4]), int(hmm[5])

return datetime(year=year, month=month, day=day, hour=hour, minute=minute, second=second)

* This function takes a list containing date-time components and converts it into a datetime object.

**Load MATLAB (.mat) File**

python

CopyEdit

def loadMat(matfile):

data = scipy.io.loadmat(matfile) # Load .mat file

filename = matfile.split("/")[-1].split(".")[0]

col = data[filename][0][0][0][0]

size = col.shape[0]

data = []

for i in range(size):

k = list(col[i][3][0].dtype.fields.keys())

d1, d2 = {}, {}

if str(col[i][0][0]) != 'impedance':

for j in range(len(k)):

t = col[i][3][0][0][j][0]

l = [t[m] for m in range(len(t))]

d2[k[j]] = l

d1['type'], d1['temp'], d1['time'], d1['data'] = str(col[i][0][0]), int(col[i][1][0]), str(convert\_to\_time(col[i][2][0])), d2

data.append(d1)

return data

* Reads a .mat file.
* Extracts battery data, including:
  + type (e.g., charge, discharge)
  + temp (temperature)
  + time (timestamp)
  + data (battery measurements like voltage, current, capacity)

**Extract Battery Capacity Over Cycles**

python

CopyEdit

def getBatteryCapacity(Battery):

cycle, capacity = [], []

i = 1

for Bat in Battery:

if Bat['type'] == 'discharge':

capacity.append(Bat['data']['Capacity'][0])

cycle.append(i)

i += 1

return [cycle, capacity]

* Extracts **discharge capacity** from battery data.

**Extract Specific Charge Data**

python

CopyEdit

def getBatteryValues(Battery, Type='charge'):

data = []

for Bat in Battery:

if Bat['type'] == Type:

data.append(Bat['data'])

return data

* Filters data based on charge or discharge type.

**3. Loading and Plotting Battery Data**

python

CopyEdit

Battery\_list = ['B0005', 'B0006', 'B0007', 'B0018']

dir\_path = 'datasets/NASA/'

Battery = {}

for name in Battery\_list:

print('Load Dataset ' + name + '.mat ...')

path = dir\_path + name + '.mat'

data = loadMat(path)

Battery[name] = getBatteryCapacity(data)

* Loads four battery datasets (B0005, B0006, B0007, B0018).

python

CopyEdit

Battery = np.load('datasets/NASA/NASA.npy', allow\_pickle=True)

Battery = Battery.item()

* If loading .mat fails, an alternative .npy file is loaded.

python

CopyEdit

fig, ax = plt.subplots(1, figsize=(12, 8))

color\_list = ['b:', 'g--', 'r-.', 'c.']

c = 0

for name, color in zip(Battery\_list, color\_list):

df\_result = Battery[name]

ax.plot(df\_result[0], df\_result[1], color, label=name)

ax.set(xlabel='Discharge cycles', ylabel='Capacity (Ah)', title='Capacity degradation at ambient temperature of 24°C')

plt.legend()

* **Plots battery capacity degradation** over discharge cycles.

**4. Data Preprocessing**

**Building Data Instances**

python

CopyEdit

def build\_instances(sequence, window\_size):

x, y = [], []

for i in range(len(sequence) - window\_size):

features = sequence[i:i+window\_size]

target = sequence[i+window\_size]

x.append(features)

y.append(target)

return np.array(x).astype(np.float32), np.array(y).astype(np.float32)

* Converts a **time series** (capacity over cycles) into X (features) and Y (targets).
* Uses a **sliding window approach**.

**Splitting Train/Test Data**

python

CopyEdit

def split\_dataset(data\_sequence, train\_ratio=0.0, capacity\_threshold=0.0):

if capacity\_threshold > 0:

max\_capacity = max(data\_sequence)

capacity = max\_capacity \* capacity\_threshold

point = [i for i in range(len(data\_sequence)) if data\_sequence[i] < capacity]

else:

point = int(len(data\_sequence) \* train\_ratio)

return data\_sequence[:point], data\_sequence[point:]

* Splits data into **training and test sets**.
* Uses **capacity threshold** or **percentage split**.

**Leave-One-Out Evaluation**

python

CopyEdit

def get\_train\_test(data\_dict, name, window\_size=8):

data\_sequence = data\_dict[name][1]

train\_data, test\_data = data\_sequence[:window\_size+1], data\_sequence[window\_size+1:]

train\_x, train\_y = build\_instances(train\_data, window\_size)

for k, v in data\_dict.items():

if k != name:

data\_x, data\_y = build\_instances(v[1], window\_size)

train\_x, train\_y = np.r\_[train\_x, data\_x], np.r\_[train\_y, data\_y]

return train\_x, train\_y, list(train\_data), list(test\_data)

* Uses **one battery as test data** and **others as training**.

**5. Transformer-Based Model**

**Autoencoder**

python

CopyEdit

class Autoencoder(nn.Module):

def \_\_init\_\_(self, input\_size=16, hidden\_dim=8, noise\_level=0.01):

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

self.fc1 = nn.Linear(input\_size, hidden\_dim)

self.fc2 = nn.Linear(hidden\_dim, input\_size)

def encoder(self, x):

return F.relu(self.fc1(x))

def mask(self, x):

return x + self.noise\_level \* torch.randn\_like(x)

def decoder(self, x):

return self.fc2(x)

def forward(self, x):

out = self.mask(x)

encode = self.encoder(out)

decode = self.decoder(encode)

return encode, decode

* Reduces noise in input features.
* Helps extract **compressed representations**.

**This is a long explanation; do you want me to continue detailing the Transformer network and training loop?** 🚀

4o

**You said:**

yes

**ChatGPT said:**

Great! Now let's go through the **Transformer-based model architecture, training loop, and evaluation** in detail.

**6. Transformer-Based Model for RUL Prediction**

A Transformer model is used to process sequential battery data (capacity degradation over cycles) and predict **Remaining Useful Life (RUL)**.

**Transformer Encoder Model**

python

CopyEdit

class TransformerEncoder(nn.Module):

def \_\_init\_\_(self, input\_dim=1, embed\_dim=32, num\_heads=2, hidden\_dim=64, num\_layers=2, dropout=0.1):

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

self.embedding = nn.Linear(input\_dim, embed\_dim)

self.positional\_encoding = nn.Parameter(torch.zeros(1, 100, embed\_dim))

encoder\_layer = nn.TransformerEncoderLayer(d\_model=embed\_dim, nhead=num\_heads, dim\_feedforward=hidden\_dim, dropout=dropout)

self.transformer\_encoder = nn.TransformerEncoder(encoder\_layer, num\_layers=num\_layers)

self.fc = nn.Linear(embed\_dim, 1)

def forward(self, x):

x = self.embedding(x) + self.positional\_encoding[:, :x.size(1), :]

x = self.transformer\_encoder(x)

x = self.fc(x[:, -1, :]) # Take the last time step

return x

**Detailed Explanation:**

* **embedding layer (nn.Linear(input\_dim, embed\_dim))**:
  + Converts each scalar input (e.g., battery capacity) into a higher-dimensional space (embed\_dim=32).
* **positional\_encoding (nn.Parameter)**:
  + Adds a position-specific vector to account for sequential order.
* **Transformer Encoder Layer (nn.TransformerEncoderLayer)**:
  + Uses **multi-head self-attention** (num\_heads=2) to capture relationships between different time steps.
  + **Feedforward network** (dim\_feedforward=hidden\_dim=64) to process information after attention.
* **Transformer Encoder (nn.TransformerEncoder)**:
  + Stacks multiple encoder layers (num\_layers=2).
* **Fully Connected Layer (self.fc)**:
  + Extracts the final prediction from the last time step (x[:, -1, :]).

**7. Training and Evaluation Functions**

**Loss Function & Optimizer**

python

CopyEdit

def train\_model(model, train\_loader, test\_loader, num\_epochs=50, lr=0.001, device="cuda"):

model = model.to(device)

criterion = nn.MSELoss()

optimizer = torch.optim.Adam(model.parameters(), lr=lr)

train\_loss\_list, test\_loss\_list = [], []

for epoch in range(num\_epochs):

model.train()

train\_loss = 0

for batch in train\_loader:

inputs, targets = batch

inputs, targets = inputs.to(device), targets.to(device)

optimizer.zero\_grad()

outputs = model(inputs)

loss = criterion(outputs.squeeze(), targets)

loss.backward()

optimizer.step()

train\_loss += loss.item()

train\_loss /= len(train\_loader)

train\_loss\_list.append(train\_loss)

test\_loss = evaluate\_model(model, test\_loader, device)

test\_loss\_list.append(test\_loss)

print(f"Epoch [{epoch+1}/{num\_epochs}], Train Loss: {train\_loss:.4f}, Test Loss: {test\_loss:.4f}")

return train\_loss\_list, test\_loss\_list

**Explanation:**

1. **Move Model to GPU (device="cuda")**
   * Ensures fast computation.
2. **Mean Squared Error (MSE) Loss (criterion = nn.MSELoss())**
   * Measures how far predictions are from actual RUL values.
3. **Adam Optimizer (torch.optim.Adam)**
   * Used for stable and fast convergence.
4. **Training Loop (for epoch in range(num\_epochs))**
   * **Model training (model.train())**
   * **Forward pass (outputs = model(inputs))**
   * **Loss computation (loss = criterion(outputs, targets))**
   * **Backward pass (loss.backward())**
   * **Optimizer update (optimizer.step())**
   * **Store train loss (train\_loss\_list.append(train\_loss))**
5. **Evaluate Model on Test Data (evaluate\_model)**
   * Computes test loss after each epoch.

**Evaluation Function**

python

CopyEdit

def evaluate\_model(model, test\_loader, device="cuda"):

model.eval()

criterion = nn.MSELoss()

test\_loss = 0

with torch.no\_grad():

for batch in test\_loader:

inputs, targets = batch

inputs, targets = inputs.to(device), targets.to(device)

outputs = model(inputs)

loss = criterion(outputs.squeeze(), targets)

test\_loss += loss.item()

return test\_loss / len(test\_loader)

* **Sets Model to Evaluation Mode (model.eval())**
* **Disables Gradient Computation (torch.no\_grad())**
* **Computes MSE Loss on Test Set**

**8. Preparing Data for Training**

**Custom Dataset Class**

python

CopyEdit

class BatteryDataset(torch.utils.data.Dataset):

def \_\_init\_\_(self, x, y):

self.x = torch.tensor(x, dtype=torch.float32).unsqueeze(-1) # Add input feature dimension

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

def \_\_len\_\_(self):

return len(self.x)

def \_\_getitem\_\_(self, index):

return self.x[index], self.y[index]

* Converts **numpy arrays** into **PyTorch tensors**.
* **Adds an extra dimension** (unsqueeze(-1)) to match Transformer input format.

**Creating Data Loaders**

python

CopyEdit

def create\_dataloaders(train\_x, train\_y, test\_x, test\_y, batch\_size=16):

train\_dataset = BatteryDataset(train\_x, train\_y)

test\_dataset = BatteryDataset(test\_x, test\_y)

train\_loader = torch.utils.data.DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

test\_loader = torch.utils.data.DataLoader(test\_dataset, batch\_size=batch\_size, shuffle=False)

return train\_loader, test\_loader

* Uses **PyTorch’s DataLoader** to efficiently batch and shuffle data.

**9. Running the Model**

**Set Parameters**

python

CopyEdit

window\_size = 8

train\_x, train\_y, test\_x, test\_y = get\_train\_test(Battery, name='B0005', window\_size=window\_size)

train\_loader, test\_loader = create\_dataloaders(train\_x, train\_y, test\_x, test\_y)

* Selects battery **B0005 as test data**.
* Uses other batteries for training.

**Initialize Model**

python

CopyEdit

model = TransformerEncoder(input\_dim=1, embed\_dim=32, num\_heads=2, hidden\_dim=64, num\_layers=2)

* Instantiates **Transformer model**.

**Train Model**

python

CopyEdit

train\_loss\_list, test\_loss\_list = train\_model(model, train\_loader, test\_loader, num\_epochs=50, lr=0.001)

* Trains for **50 epochs**.

**10. Visualizing Results**

python

CopyEdit

plt.plot(range(len(train\_loss\_list)), train\_loss\_list, label="Train Loss")

plt.plot(range(len(test\_loss\_list)), test\_loss\_list, label="Test Loss")

plt.xlabel("Epoch")

plt.ylabel("Loss")

plt.title("Training & Test Loss Over Time")

plt.legend()

plt.show()

* **Plots loss curves** to check training progress.

**Summary of Implementation**

1. **Data Loading**: Reads NASA Li-ion battery datasets.
2. **Preprocessing**: Converts time series into training windows.
3. **Model Definition**: Uses a **Transformer Encoder** for RUL prediction.
4. **Training**: Uses **MSE Loss & Adam Optimizer**.
5. **Evaluation**: Computes **Test Loss (RMSE/MSE)**.
6. **Prediction & Visualization**: Plots test results.

This setup helps predict **Remaining Useful Life (RUL) of batteries** using **Transformer models**. Let me know if you need more modifications or explanations! 🚀