**TÀI LIỆU TỔNG HỢP: PHÂN TÍCH EDA & XÂY DỰNG MÔ HÌNH LSTM, TRANSFORMER CHO DỰ BÁO GIÁ CỔ PHIẾU**

**Mục lục**

1. [Phân tích EDA đơn giản](#eda)
   * 1.1. Các bước tiền xử lý và EDA cơ bản
   * 1.2. Code minh họa EDA (có ma trận tương quan)
2. [Xây dựng mô hình LSTM & Bidirectional LSTM](#lstm)
   * 2.1. Định dạng đầu vào: (batch\_size, T, input\_dim)
   * 2.2. Code mẫu mô hình LSTM và Bidirectional LSTM
3. [Xây dựng mô hình Transformer chuẩn (StockCloseTransformerV0)](#transformer-v0)
   * 3.1. Kiến trúc cơ bản của Transformer chuẩn
   * 3.2. Code mẫu mô hình StockCloseTransformerV0
4. [Hàm dự đoán kết hợp fine-tune (optional)](#predict)
   * 4.1. Ý tưởng và quy trình fine-tune
   * 4.2. Code mẫu hàm predict
5. [Huấn luyện mô hình và tính lỗi trên tập xác thực/kiểm tra](#training-validation)
   * 5.1. Hàm huấn luyện (train)
   * 5.2. Hàm tính lỗi (validation)
6. [Các cách biến đổi các thành phần trong Transformer](#modification)
   * 6.1. Positional Encoding (Learnable, v.v.)
   * 6.2. Thay đổi Attention (Gaussian, Linear, …)
   * 6.3. Thay đổi FFN, Dropout
   * 6.4. Tạo ra các phiên bản V1, V2, V3
7. [Chiến lược huấn luyện cho dữ liệu nhiều ticker & lựa chọn đặc trưng](#multi-ticker)
   * 7.1. Mô hình tổng quát trên 5 đặc trưng
   * 7.2. Fine-tuning cho ticker mới
8. [Kết luận và hướng phát triển](#conclusion)

<a name="eda"></a>

**1. PHÂN TÍCH EDA ĐƠN GIẢN**

**1.1. Các bước tiền xử lý và EDA cơ bản**

* **Thu thập dữ liệu:** Lấy dữ liệu lịch sử từ yfinance cho 50 cổ phiếu với các trường Open, High, Low, Close, Volume.
* **Sắp xếp và tiền xử lý:**
  + Sắp xếp theo Ticker và Date (theo thứ tự thời gian).
  + Xử lý các ngày không giao dịch (thường thiếu T7, CN) – điều này là tự nhiên trong dữ liệu cổ phiếu.
  + Chuẩn hóa dữ liệu (min-max hoặc z-score).
* **EDA cơ bản:**
  + Thống kê mô tả (mean, std, min, max, skew, kurtosis).
  + Ma trận tương quan để xem mối liên hệ giữa Open, High, Low, Close, Volume.
  + Biểu đồ line (giá theo thời gian), boxplot, heatmap.

**1.2. Code minh họa EDA (có ma trận tương quan)**

python

Sao chépChỉnh sửa

import yfinance as yf

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Danh sách tickers (ví dụ 50 cổ phiếu)

tickers = [

"AAPL", "MSFT", "GOOGL", "AMZN", "META", "TSLA", "BRK-B", "JNJ", "V", "WMT",

"PG", "JPM", "UNH", "NVDA", "HD", "DIS", "MA", "KO", "PFE", "INTC",

"SAP", "SIEGY", "HSBA.L", "BP.L", "RDS-A", "ASML", "VOD", "BAS.DE", "SAN", "TOT",

"OR", "DAI.DE", "MC.PA", "AIR.PA", "7203.T", "6758.T", "9984.T", "TCEHY", "BABA", "TSM",

"NVO", "RNO.PA", "RIO", "BHP", "ITUB", "RELIANCE.NS", "HDFCBANK.NS", "RY.TO", "BMO.TO", "WES.AX"

]

# Tải dữ liệu

data = yf.download(tickers, start="2018-01-01", end="2023-01-01", group\_by='ticker')

# Chuyển sang dạng phẳng

data\_flat\_list = []

for ticker in tickers:

df\_ticker = data[ticker].reset\_index() # DataFrame: Date, Open, High, Low, Close, Adj Close, Volume

df\_ticker['Ticker'] = ticker

data\_flat\_list.append(df\_ticker)

df = pd.concat(data\_flat\_list, ignore\_index=True)

df = df[['Date', 'Ticker', 'Open', 'High', 'Low', 'Close', 'Volume']]

df['Date'] = pd.to\_datetime(df['Date'])

df = df.sort\_values(by=['Ticker', 'Date'])

# Thống kê mô tả cho AAPL

df\_aapl = df[df['Ticker'] == 'AAPL']

desc = df\_aapl[['Open', 'High', 'Low', 'Close', 'Volume']].describe()

print("Thống kê mô tả AAPL:\n", desc)

# Ma trận tương quan

corr = df\_aapl[['Open', 'High', 'Low', 'Close', 'Volume']].corr()

plt.figure(figsize=(8,6))

sns.heatmap(corr, annot=True, cmap='coolwarm')

plt.title("Ma trận tương quan các đặc trưng AAPL")

plt.show()

<a name="lstm"></a>

**2. XÂY DỰNG MÔ HÌNH LSTM & BIDIRECTIONAL LSTM**

**2.1. Định dạng đầu vào: (batch\_size, T, input\_dim)**

* **T**: số bước thời gian (ví dụ 50 ngày).
* **input\_dim**: số đặc trưng (1, 2, 3, 4, hoặc 5).
* Mỗi mẫu huấn luyện là một “khối” dữ liệu (T,input\_dim)(T, input\\_dim)(T,input\_dim).

**2.2. Code mẫu**

**LSTM cơ bản**

python

Sao chépChỉnh sửa

import torch

import torch.nn as nn

class LSTMModel(nn.Module):

def \_\_init\_\_(self, input\_dim, hidden\_dim, num\_layers, output\_dim):

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

self.lstm = nn.LSTM(input\_dim, hidden\_dim, num\_layers, batch\_first=True)

self.fc = nn.Linear(hidden\_dim, output\_dim)

def forward(self, x):

out, \_ = self.lstm(x) # out: (batch\_size, T, hidden\_dim)

out = out[:, -1, :] # lấy hidden state bước cuối

return self.fc(out)

# Ví dụ

model\_lstm = LSTMModel(input\_dim=1, hidden\_dim=64, num\_layers=2, output\_dim=1)

print("Mô hình LSTM cơ bản:", model\_lstm)

**Bidirectional LSTM**

python

Sao chépChỉnh sửa

class BidirectionalLSTMModel(nn.Module):

def \_\_init\_\_(self, input\_dim, hidden\_dim, num\_layers, output\_dim, dropout=0.2):

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

self.lstm = nn.LSTM(input\_dim, hidden\_dim, num\_layers, batch\_first=True,

bidirectional=True, dropout=dropout)

self.fc = nn.Linear(hidden\_dim \* 2, output\_dim)

def forward(self, x):

out, \_ = self.lstm(x) # (batch\_size, T, hidden\_dim\*2)

out = out[:, -1, :] # lấy hidden state bước cuối

return self.fc(out)

model\_bilstm = BidirectionalLSTMModel(input\_dim=1, hidden\_dim=64, num\_layers=2, output\_dim=1)

print("Mô hình Bi-LSTM:", model\_bilstm)

<a name="transformer-v0"></a>

**3. XÂY DỰNG MÔ HÌNH TRANSFORMER CHUẨN (StockCloseTransformerV0)**

**3.1. Kiến trúc cơ bản**

* **Input Projection**: Chiếu 5 đặc trưng lên d\_model.
* **Positional Encoding**: Sử dụng sinusoidal encoding.
* **Transformer Encoder**: Sử dụng nn.TransformerEncoderLayer của PyTorch.
* **Pooling + Output**: Lấy trung bình theo chiều thời gian và đưa qua fully connected.

**3.2. Code mẫu**

python

Sao chépChỉnh sửa

import math

import torch

import torch.nn as nn

class PositionalEncoding(nn.Module):

def \_\_init\_\_(self, d\_model, dropout=0.1, max\_len=5000):

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

self.dropout = nn.Dropout(p=dropout)

pe = torch.zeros(max\_len, d\_model)

position = torch.arange(0, max\_len, dtype=torch.float).unsqueeze(1)

div\_term = torch.exp(torch.arange(0, d\_model, 2).float() \*

(-math.log(10000.0) / d\_model))

pe[:, 0::2] = torch.sin(position \* div\_term)

pe[:, 1::2] = torch.cos(position \* div\_term)

pe = pe.unsqueeze(0)

self.register\_buffer('pe', pe)

def forward(self, x):

x = x + self.pe[:, :x.size(1), :]

return self.dropout(x)

class StockCloseTransformerV0(nn.Module):

def \_\_init\_\_(self, input\_dim, d\_model, nhead, num\_layers, ff\_dim, output\_dim, dropout=0.1):

super(StockCloseTransformerV0, self).\_\_init\_\_()

self.input\_proj = nn.Linear(input\_dim, d\_model)

self.pos\_encoder = PositionalEncoding(d\_model, dropout)

encoder\_layer = nn.TransformerEncoderLayer(d\_model=d\_model, nhead=nhead,

dim\_feedforward=ff\_dim, dropout=dropout)

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

self.fc = nn.Linear(d\_model, output\_dim)

def forward(self, x):

# x: (batch\_size, seq\_len, input\_dim)

x = self.input\_proj(x) # (batch\_size, seq\_len, d\_model)

x = self.pos\_encoder(x) # (batch\_size, seq\_len, d\_model)

x = x.permute(1, 0, 2) # (seq\_len, batch\_size, d\_model)

x = self.transformer\_encoder(x) # (seq\_len, batch\_size, d\_model)

x = x.mean(dim=0) # (batch\_size, d\_model)

return self.fc(x)

# Ví dụ

model\_transformer\_v0 = StockCloseTransformerV0(

input\_dim=5, d\_model=64, nhead=8, num\_layers=6, ff\_dim=256, output\_dim=1

)

print("StockCloseTransformerV0:", model\_transformer\_v0)

<a name="predict"></a>

**4. HÀM DỰ ĐOÁN KẾT HỢP FINE-TUNE (OPTIONAL)**

**4.1. Ý tưởng & Quy trình**

* Hàm predict nhận input\_data để dự báo.
* Tùy chọn fine\_tune\_data nếu muốn huấn luyện nhanh trên dữ liệu mới (ticker mới).

**4.2. Code mẫu hàm predict**

python

Sao chépChỉnh sửa

import torch

import torch.nn as nn

import torch.optim as optim

def predict(model, input\_data, fine\_tune\_data=None, num\_epochs\_ft=5, lr\_ft=1e-4):

"""

Hàm dự báo với tùy chọn fine-tune.

- model: mô hình đã huấn luyện.

- input\_data: (batch\_size, seq\_len, input\_dim).

- fine\_tune\_data: (X\_ft, y\_ft) nếu muốn fine-tune.

- num\_epochs\_ft: số epoch fine-tune.

- lr\_ft: learning rate fine-tune.

"""

if fine\_tune\_data is not None:

X\_ft, y\_ft = fine\_tune\_data

optimizer = optim.Adam(model.parameters(), lr=lr\_ft)

criterion = nn.MSELoss()

model.train()

for epoch in range(num\_epochs\_ft):

optimizer.zero\_grad()

outputs\_ft = model(X\_ft)

loss = criterion(outputs\_ft, y\_ft)

loss.backward()

optimizer.step()

print(f"Fine-tune Epoch {epoch+1}/{num\_epochs\_ft}, Loss: {loss.item():.4f}")

model.eval()

with torch.no\_grad():

predictions = model(input\_data)

return predictions

<a name="training-validation"></a>

**5. HUẤN LUYỆN MÔ HÌNH VÀ TÍNH LỖI TRÊN TẬP XÁC THỰC/KIỂM TRA**

**5.1. Hàm huấn luyện (train)**

Dưới đây là hàm huấn luyện tổng quát, có thể áp dụng cho LSTM hoặc Transformer:

python

Sao chépChỉnh sửa

def train\_model(model, train\_loader, val\_loader, num\_epochs, lr=1e-3):

"""

Huấn luyện mô hình trên train\_loader, đánh giá trên val\_loader.

- model: mô hình (LSTM hoặc Transformer).

- train\_loader, val\_loader: DataLoader cho train và val.

- num\_epochs: số epoch huấn luyện.

- lr: learning rate.

"""

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

criterion = nn.MSELoss()

for epoch in range(num\_epochs):

model.train()

train\_loss = 0.0

for X\_batch, y\_batch in train\_loader:

optimizer.zero\_grad()

outputs = model(X\_batch)

loss = criterion(outputs, y\_batch)

loss.backward()

optimizer.step()

train\_loss += loss.item() \* X\_batch.size(0)

train\_loss /= len(train\_loader.dataset)

# Đánh giá trên tập val

val\_loss = evaluate\_model(model, val\_loader)

print(f"Epoch {epoch+1}/{num\_epochs}, Train Loss: {train\_loss:.4f}, Val Loss: {val\_loss:.4f}")

def evaluate\_model(model, data\_loader):

"""

Tính lỗi (MSE) trên data\_loader (có thể là tập validation hoặc test).

"""

model.eval()

criterion = nn.MSELoss()

running\_loss = 0.0

with torch.no\_grad():

for X\_batch, y\_batch in data\_loader:

outputs = model(X\_batch)

loss = criterion(outputs, y\_batch)

running\_loss += loss.item() \* X\_batch.size(0)

return running\_loss / len(data\_loader.dataset)

**5.2. Hàm tính lỗi (validation)**

* Ở trên, hàm evaluate\_model đã thực hiện tính MSE.
* Bạn có thể thay thế bằng các chỉ số khác như MAE, RMSE, R² nếu cần.

<a name="modification"></a>

**6. CÁC CÁCH BIẾN ĐỔI CÁC THÀNH PHẦN TRONG TRANSFORMER**

**6.1. Positional Encoding**

* **Learnable Positional Encoding:** Cho phép mô hình tự học cách mã hóa vị trí.
* **Time2Vec**: Một biến thể phù hợp với dữ liệu time-series.

**6.2. Thay đổi Attention (Gaussian, Kernel, …)**

* **Gaussian Attention:** Sử dụng hàm Gaussian để tính độ tương đồng thay vì scaled dot-product.
* **Linear/Kernal Attention:** Giảm độ phức tạp tính toán từ O(T²) xuống O(T).

**6.3. Thay đổi FFN, Dropout**

* Thay **ReLU** bằng **GELU**.
* Điều chỉnh **dropout** (0.1 - 0.3) để giảm overfitting.

**6.4. Tạo các phiên bản V1, V2, V3**

* **V1:** Giữ nguyên sinusoidal PE, scaled dot-product attention.
* **V2:** Sử dụng Learnable PE, số layer encoder ít hơn (2-4) để phù hợp time-series.
* **V3:** Sử dụng Gaussian Attention, hoặc kernel attention.

<a name="multi-ticker"></a>

**7. CHIẾN LƯỢC HUẤN LUYỆN CHO DỮ LIỆU NHIỀU TICKER & LỰA CHỌN ĐẶC TRƯNG**

**7.1. Mô hình tổng quát trên 5 đặc trưng**

* Gom dữ liệu của 50 ticker, mỗi mẫu có 5 đặc trưng (Open, High, Low, Close, Volume).
* Huấn luyện mô hình tổng quát (LSTM hoặc Transformer).
* Không đưa ticker vào mô hình → tăng khả năng tổng quát cho ticker mới.

**7.2. Fine-tuning cho ticker mới**

* Khi có ticker mới, bạn có thể fine-tune mô hình (chỉ lớp cuối hoặc toàn bộ) trên dữ liệu của ticker mới nếu có.
* Nếu không có nhiều dữ liệu, có thể chỉ fine-tune lớp cuối để tránh overfitting.

<a name="conclusion"></a>

**8. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN**

1. **Tổng hợp:**
   * Bạn đã có EDA đơn giản để hiểu dữ liệu (thống kê mô tả, ma trận tương quan).
   * Mô hình LSTM, Bidirectional LSTM và Transformer cơ bản (StockCloseTransformerV0) cho dự báo giá cổ phiếu với đầu vào (batch\_size,T,input\_dim)(\text{batch\\_size}, T, \text{input\\_dim})(batch\_size,T,input\_dim).
   * Hàm predict có tùy chọn fine-tune để điều chỉnh mô hình khi có dữ liệu mới.
   * Có thể tùy chỉnh các thành phần của Transformer (positional encoding, attention, FFN, dropout) để tạo các phiên bản V1, V2, V3.
2. **Hướng phát triển:**
   * **Multi-Task Learning**: Thêm output head riêng cho từng nhóm ticker.
   * **Embedding Ticker**: Nếu cần phân biệt ticker cũ, bạn có thể sử dụng embedding để mô hình học đặc trưng riêng của mỗi ticker.
   * **Feature Engineering**: Bổ sung đặc trưng như SMA, EMA, RSI, Bollinger Bands, … để cải thiện hiệu suất dự báo.
   * **Chiến lược Fine-tuning**: Chỉ fine-tune lớp cuối hay layer-wise learning rate để tránh overfitting.
   * **So sánh với Baseline**: Persistence Model, Moving Average, ARIMA để đánh giá hiệu suất mô hình học sâu.

Với nội dung trên, bạn có một cái nhìn toàn diện về cách tổ chức dữ liệu, xây dựng và huấn luyện mô hình LSTM/Transformer cho bài toán dự báo giá cổ phiếu, kèm theo các chiến lược tùy chỉnh và fine-tuning. Hy vọng tài liệu này giúp ích cho quá trình nghiên cứu và triển khai dự án của bạn.

Dưới đây là một ví dụ về cách tự xây dựng một lớp Transformer Encoder tiêu chuẩn sử dụng các module có sẵn trong PyTorch (trong torch.nn), mà không sử dụng trực tiếp nn.TransformerEncoderLayer. Trong ví dụ này, chúng ta sẽ tách riêng hai thành phần chính:

1. **Attention Block:**
   * **Multi-Head Self-Attention:** Sử dụng nn.MultiheadAttention để tính toán scaled dot-product attention.
   * Trong quá trình này, mỗi token (hoặc bước thời gian) sẽ "chú ý" đến tất cả các token khác trong chuỗi bằng cách tính toán ma trận dot-product giữa các vector query, key và value, sau đó chia cho căn bậc hai của kích thước của mỗi head (d\_k) và áp dụng softmax để lấy trọng số attention.
2. **Feed-Forward Network (FFN):**
   * Một mạng nơ-ron đơn giản gồm hai lớp Linear với một hàm kích hoạt phi tuyến (ReLU hoặc GELU) ở giữa.
   * FFN được áp dụng riêng rẽ cho từng token sau khi attention, nhằm tăng khả năng biểu diễn phi tuyến và tạo ra các đặc trưng phức tạp hơn.

Ngoài ra, theo kiến trúc Transformer, chúng ta cần áp dụng các **Residual Connections** (nối lại đầu vào với đầu ra của mỗi khối) và **Layer Normalization** sau mỗi sub-layer (attention và FFN) để làm ổn định quá trình huấn luyện.

Dưới đây là mã nguồn minh họa:

python

Sao chépChỉnh sửa

import math

import torch

import torch.nn as nn

import torch.nn.functional as F

class CustomTransformerEncoderLayer(nn.Module):

def \_\_init\_\_(self, d\_model, nhead, dim\_feedforward, dropout=0.1):

"""

d\_model: Kích thước của embedding (số chiều của vector biểu diễn)

nhead: Số lượng head trong multi-head attention

dim\_feedforward: Số chiều của tầng ẩn trong FFN (thường là 2-4 lần d\_model)

dropout: Tỷ lệ dropout áp dụng sau mỗi sub-layer

"""

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

# Multi-Head Self-Attention (Scaled Dot-Product Attention)

self.self\_attn = nn.MultiheadAttention(d\_model, nhead, dropout=dropout)

# Feed-Forward Network (FFN)

self.linear1 = nn.Linear(d\_model, dim\_feedforward)

self.activation = nn.ReLU() # Hoặc bạn có thể dùng GELU()

self.linear2 = nn.Linear(dim\_feedforward, d\_model)

# Dropout sau mỗi sub-layer

self.dropout1 = nn.Dropout(dropout)

self.dropout2 = nn.Dropout(dropout)

# Layer Normalization cho mỗi sub-layer (trước khi cộng residual)

self.norm1 = nn.LayerNorm(d\_model)

self.norm2 = nn.LayerNorm(d\_model)

def forward(self, src, src\_mask=None, src\_key\_padding\_mask=None):

"""

src: Tensor có shape (seq\_len, batch\_size, d\_model)

src\_mask: (Optional) mask cho self-attention, nếu cần

src\_key\_padding\_mask: (Optional) mask cho các phần tử padding

"""

# --- Self-Attention Block ---

# Tính toán Multi-Head Self-Attention: Q = K = V = src

attn\_output, \_ = self.self\_attn(src, src, src,

attn\_mask=src\_mask,

key\_padding\_mask=src\_key\_padding\_mask)

# Áp dụng dropout và residual connection, sau đó LayerNorm

src2 = self.dropout1(attn\_output)

src = self.norm1(src + src2)

# --- Feed-Forward Network (FFN) Block ---

ffn\_output = self.linear2(self.dropout2(self.activation(self.linear1(src))))

# Residual connection và LayerNorm

src = self.norm2(src + ffn\_output)

return src

# Ví dụ sử dụng lớp CustomTransformerEncoderLayer:

if \_\_name\_\_ == "\_\_main\_\_":

batch\_size = 32

seq\_len = 50

d\_model = 64

nhead = 8

dim\_feedforward = 256

dropout = 0.1

# Giả lập đầu vào cho encoder: (seq\_len, batch\_size, d\_model)

dummy\_input = torch.randn(seq\_len, batch\_size, d\_model)

encoder\_layer = CustomTransformerEncoderLayer(d\_model, nhead, dim\_feedforward, dropout)

output = encoder\_layer(dummy\_input)

print("Kích thước đầu ra của CustomTransformerEncoderLayer:", output.shape)

# Kết quả mong đợi: (seq\_len, batch\_size, d\_model)

**Giải thích chi tiết:**

* **Attention Block:**
  + nn.MultiheadAttention được sử dụng để tính toán scaled dot-product attention.
  + Các vector query, key và value được chiếu từ đầu vào (src), sau đó tính toán điểm số tương tự bằng cách nhân dot-product và chia cho √(d\_k).
  + Softmax được áp dụng để lấy trọng số attention, sau đó nhân với V để có đầu ra context.
* **Residual Connection & LayerNorm:**
  + Sau khi tính attention, đầu ra được cộng với đầu vào gốc (residual) và sau đó được chuẩn hóa bằng LayerNorm (self.norm1).
* **Feed-Forward Network (FFN):**
  + Hai lớp Linear với hàm kích hoạt ReLU giữa chúng chuyển đổi dữ liệu từ kích thước d\_model sang dim\_feedforward và quay lại d\_model.
  + Sau đó, dropout được áp dụng, và kết quả được cộng với đầu vào (residual) và chuẩn hóa bằng LayerNorm (self.norm2).
* **Lưu ý về định dạng:**
  + Trong PyTorch, module nn.MultiheadAttention yêu cầu đầu vào có định dạng (seq\_len, batch\_size, d\_model), vì vậy khi sử dụng trong một pipeline Transformer Encoder, bạn cần chuyển đổi chiều của dữ liệu (thường nếu dữ liệu ban đầu có dạng (batch\_size, seq\_len, d\_model)).

Với cách xây dựng như trên, bạn đã tự cài đặt một lớp Transformer Encoder tiêu chuẩn, làm rõ các thành phần Self-Attention và FFN, thay vì sử dụng trực tiếp nn.TransformerEncoderLayer. Điều này giúp bạn hiểu sâu hơn về cách thức hoạt động của từng thành phần cũng như cho phép bạn tùy chỉnh khi cần thiết cho dữ liệu dự báo giá cổ phiếu.

Nếu bạn có câu hỏi thêm hoặc cần điều chỉnh phần nào, hãy cho tôi biết!