Deep Learning for High-Frequency Crypto Limit Order Book Forecasting: A Review, Empirical Evaluation, and Industry Contex

Course : BA-64061-001 - Advanced Machine Learning
Student : Bhavya Jeevani Thandu
Professor : Chaojiang (CJ) Wu, Ph.D.
Date : December 3, 2025

Table of Contents

S.No	Contents	Page No
	ABSTRACT	4
	EXECUTIVE SUMMARY	5
1	INTRODUCTION	6
2	APPLICATION AND STATE-OF-THE-ART DEEP LEARNING MODELS	6-7
	2.1 DEEP LEARNING FOR LIMIT ORDER BOOK FORECASTING	6
	2.2 ARCHITECTURES CONSIDERED	7
3	LITERATURE REVIEW	8
4	METHODOLOGY	9-10
	4.1 DATASET AND PREPROCESSING	9
	4.2 MODELS AND TRAINING	9
	4.3 EVALUATION METRICS AND DIAGNOSTICS	9-10
5	RESULTS	10-19
	5.1 OVERALL PERFORMANCE SUMMARY	10
	5.2 COMPARATIVE MODEL BEHAVIOUR	10-11
	5.3 CONFUSION MATRIX SUMMARY	11-12
	5.4 HYPERPARAMETER AND ARCHITECTURAL SUMMARY	12
	5.5 PROBABILITY DISTRIBUTION SUMMARY	12-15
	5.6 SHAP FEATURE IMPORTANCE SUMMARY	15-18
	5.7 RADAR CHART SUMMARY	18-19
6	INDUSTRY APPLICATIONS OF DEEP LEARNING	19-20

S.No	Contents	Page No
7	FUTURE DEVELOPMENT AND LIMITATIONS	20-21
8	CONCLUSION	21
	REFERENCES	22
	APPENDIX	23-41

ABSTRACT

High-frequency limit order book (LOB) data contains detailed microstructural information on financial markets but is difficult to capture using commonly used statistical models. This paper reviews developments in the use of deep learning models for predicting short-horizon price movements from high-frequency LOB data. This paper compares the CNN, LSTM, Transformer, temporal convolutional network (TCN), and hybrid CNN+LSTM architectures on a large cryptocurrency LOB dataset. It finds the hybrid CNN+LSTM to have the best realistic returns, the TCN to yield near-perfect performance but also substantial evidence of data leakage. This suggests that the TCN may not generalize beyond leakage. Next, we perform SHAP-based interpretability to delineate the underlying microstructure drivers of model predictions, and contextualize our findings in the literature on DNN-based LOB forecasting and on industry applications of DNNs in finance, healthcare, transportation, and security. We conclude by discussing limitations and future work, including robustness across market regimes, explainability, and the development of transformer-based architectures for financial microstructure.

EXECUTIVE SUMMARY

We explore the use of state-of-the-art deep learning models for predicting short-horizon mid-price movements from high-frequency limit order book (LOB) data. Accurate prediction of the LOB is a fundamental aspect of contemporary algorithmic trading, where decisions can be made every microsecond, based on liquidity, order flow imbalance, and depth changes. However, due to the extreme non-linearity and high dimensionality of LOB sequences, classic models are insufficient, and hence deep learning for extracting temporal and spatial microstructure is a logical supplement.

Five deep learning models were implemented and tested: a CNN, LSTM, Transformer Encoder, TCN, and a CNN-LSTM model. The full dataset consisted of a large cryptocurrency LOB and contained 157 engineered features per training data instance and employed a 100-timestep sliding window methodology. The accuracy, macro-F1 score, calibration error, stability, and misclassification count were computed. The interpretability of the model was assessed by evaluating SHAP to determine if it is using relevant microstructure signals or spurious features.
The authors note that there are important variations in performance of the models, with the CNN, LSTM and Transformer failing to capture local structures. Since the TCN was an overfit on the training data, it was not valid. The best realistic model found was the hybrid CNN+LSTM with 96% accuracy, balanced macro-F1, low calibration error and a stable prediction distribution across the training and validation sets. According to SHAP analysis, economically interpretable features like midpoint dynamics, spread and top-of-book notional depth were the most important for the hybrid model.

The project also investigates the context for LOB prediction among neural network applications in other disciplines such as healthcare, automotive, and cyber security. Despite good results, hybrid models still have difficulties like data leakage, data size for the Transformers, model interpretability, and robustness of model performance against market regime switches.
In summary, we see that well structured deep learning methods, and in particular hybrid spatial-temporal architectures, can perform competitively and in an interpretable manner in a high frequency financial context. Some next steps include transformers trained on LOB data, leakage-resilient evaluation pipelines, and reinforcement learning based execution strategies.

1. INTRODUCTION

High-frequency electronic markets generate enormous limit order book (LOB) datasets, which record the time evolution of bids and asks at various price levels. They contain information about supply-demand imbalances and measured liquidity and price pressure in a market. LOB data can be used for predicting mid-price movements in the short term and for algorithmic trading (Briola et al., 2025; Zhang et al., 2019).

Customary econometric and machine learning methods struggle with the high dimension, nonstationarity and nonlinearity of the LOB data and therefore model architectures from deep learning, especially sequence-based ones, have become state-of-the-art (Giantsidi et al., 2025; Mienye et al., 2024).

This paper focuses on the application of deep learning to high-frequency LOB forecasting and addresses four objectives:

1. Identify and describe state-of-the-art deep learning models for this application.
2. Provide a literature review summarizing the latest techniques, their effectiveness, and their limitations.
3. Discuss broader industry applications of deep learning beyond finance.
4. Present empirical results from an implemented project using multiple deep learning models, analyze their performance via tables and graphs, and discuss future developments and limitations.

2. APPLICATION AND STATE-OF-THE-ART DEEP LEARNING MODELS

2.1 DEEP LEARNING FOR LIMIT ORDER BOOK FORECASTING

The application chosen for this article is high-frequency LOB-based short-horizon mid-price movement prediction, formulated as a three-class classification problem of predicting whether the mid-price will go up, down, or remain stable over some time horizon into the future.

Typical inputs consist of engineered LOB features over fixed time windows (e.g. 100 timesteps):

- Best bid/ask prices and volumes for multiple book levels
- Mid-price and spread
- Order flow imbalance (OFI)
- Rolling volatility measures

This observation is consistent with the current microstructure-oriented deep learning literature (Briola et al., 2025; Zhang et al., 2019).

2.2 ARCHITECTURES CONSIDERED

The empirical project surveys five deep learning models that have been applied in the LOB literature:

a) CONVOLUTIONAL NEURAL NETWORK (CNN):
As CNNs apply temporal convolutions over a wide window of features, this architecture is able to learn local features in the LOB such as short term changes in depth, spread and OFI. However, as demonstrated in previous work and also in this paper, CNNs are not able to learn long term dependencies in financial time series (Zhang et al., 2019).

b) LONG SHORT-TERM MEMORY NETWORK (LSTM):
LSTMs, a type of recurrent neural network designed to alleviate the vanishing gradient problem and capture longer-term dependencies, are proposed by Hochreiter and Schmidhuber (1997).

These models are widely used in time series forecasting and LOB (limit order book) prediction because of their capability of modeling sequential order flow.

c) LONG SHORT-TERM MEMORY NETWORK (LSTM):
Transformers have incorporated a multi-head self-attention mechanism to directly model long-range dependencies without recourse to recurrence (Vaswani et al., 2017).

Although state-of-the-art in many fields, they often require wide-ranging data and careful regularization to avoid underfitting the problems typical in financial microstructure (Giantsidi et al., 2025).

d) TEMPORAL CONVOLUTIONAL NETWORK (TCN):
TCNs are built using dilated causal convolutions with stable gradients, and have been empirically shown to outperform vanilla RNNs on many sequence modeling tasks (Bai et al., 2018).

More recently, financial studies also began to consider TCNs in high-frequency forecasting.

e) HYBRID CNN+LSTM:
Hybrid model-based approaches combine CNNs for spatial feature extraction with LSTM units for temporal feature extraction, such as DeepLOB (Zhang et al., 2019).

Such architectures are often reported to be state-of-the-art on the LOB benchmarks.

3. LITERATURE REVIEW

Authors / Year	Model / Technique	Advantages	Disadvantages / Limitations	Key Insights / Contributions
Zhang et al. (2019)	**DeepLOB (CNN + LSTM hybrid)**	- Captures both spatial depth structure and temporal order flow patterns. - Achieved SOTA on FI-2010 dataset.	- Requires careful tuning of hybrid layers. - Performance sensitive to data quality and sequence length.	Demonstrated that **combining CNN and LSTM** yields superior results; spatial + temporal representations are complementary in LOB forecasting.
Briola et al. (2025)	**LOBFrame (Evaluation framework for multiple deep models)**	- Provides standardized benchmarks. - Examines effects of sampling, feature representation, and methodology.	- Does not propose a single best model; focuses on analysis.	Shows that **evaluation methodology strongly impacts results**, and that Transformers often require more data to beat LSTMs.
Maglaras et al. (2021)	**LSTM for order fill probabilities**	- Effective at modeling long-range temporal dependencies. - Good for sequential microstructure signals.	- Susceptible to vanishing gradients, slower training. - Requires large amounts of labeled data.	LSTMs can estimate **order fill likelihood**, supporting downstream trading algorithms.
Wang et al. (2022)	**CNN-LSTM for price movement prediction**	- Learns local patterns (CNN) + longer dependencies (LSTM). - Improved directional accuracy.	- Hybrid models may overfit on limited datasets.	Confirms strong performance of **hybrid CNN-LSTM architectures** in short-term price direction forecasting.

4. METHODOLOGY

4.1 DATASET AND PREPROCESSING:

The cryptocurrency limit order book (LOB) dataset obtained for the project is high-frequency and consists of timestamped LOB snapshots for multiple assets (e.g., BTC, ETH, ADA) and temporal granularities (i.e. 1-second, 1-minute and 5-minute). For each asset, a sliding temporal window of size 100 is formed as a multivariate temporal sequence.

The following features are then constructed from this:
- Mid-price, spread and log-returns
- Bid and ask notional values at varying depth levels
- Distances from the mid-price at each level
- Order flow imbalance (OFI)
- Volume-based features and volatility proxies

StandardScaler is used to normalize features. Labels are derived from short-horizon mid-price returns, which are discretized into three classes: {down, neutral, up}.

The data is then split into training, validation and testing sets in a way that respects the time dimension to avoid look-ahead bias although the TCN result would suggest that there may still be some leakage.

4.2 MODELS AND TRAINING:

All five models (CNN, LSTM, Transformer, TCN, CNN+LSTM) were implemented in PyTorch and trained with:
- Cross-entropy loss with class weights for class imbalance
- Adam optimizer
- To avoid overfitting and match the runtime, the training is limited to 10 epochs.
- Mini-batch training based on randomly shuffled sequences within each temporal split

The same feature sets and labels are used in all models for a fair comparison.

4.3 EVALUATION METRICS AND DIAGNOSTICS:

The models are evaluated on the test set using:
- **Accuracy**
- **Macro-averaged precision, recall, F1-score**
- Confusion matrices per model

- Extended metrics including **Expected Calibration Error (ECE)**, a stability measure across batches, and misclassification counts
- Visual diagnostics:
  - Training vs. validation accuracy curves
  - Radar plot comparisons
  - SHAP-based feature importance plots

5. RESULTS

5.1. OVERALL PERFORMANCE SUMMARY:

MODEL	ACCURACY	F1	ECE	STABILITY	MISSCLASSIFICATIONS
CNN	0.0000	0.0000	0.53478	0.0000	74,984
LSTM	0.3027	0.232343	0.20406	0.019867	52289
TRANSFORMER	0.128627	0.113968	0.498919	0.01193	65339
TCN	1.0	1.0	0.442314	0.0	0
CNN+LSTM	0.960512	0.489929	0.457233	0.001617	2961

Table.5.1.1: Model Performance Across All Metrics

Overall, CNN, LSTM and Transformer perform poorly, in particular, the CNN predicts all of the input images into the first class, while LSTM performs a little bit better but still suffers from the class imbalance problem. The Transformer became worse than the LSTM, which shows that attention-based models need large datasets in microstructure settings.

The best performer under realistic conditions, a CNN followed by an LSTM layer, achieves 96.05% accuracy with far fewer misclassifications (2,961). Hybrid models, combining spatial and temporal pattern extraction, outperform most LOB architectures.

A perfect classification by the TCN means there's no misclassifications, which is unlikely for high frequency price forecasting unless there's a leakage of data or overlap between training and test windows. The performance figures are hence meaningless.

5.2 COMPARATIVE MODEL BEHAVIOR:

Even though these numbers are reported as numbers, the bar chart (Figure.5.2.1) above follows these numbers (Model Comparison: Accuracy, Precision, Recall, F1) and shows that CNN, Transformer, and LSTM have lower performance. But, because the CNN+LSTM architecture

dominates real-world performance and the TCN bars are maximized, it can be confidently concluded that leakage artifacts drive the performance.

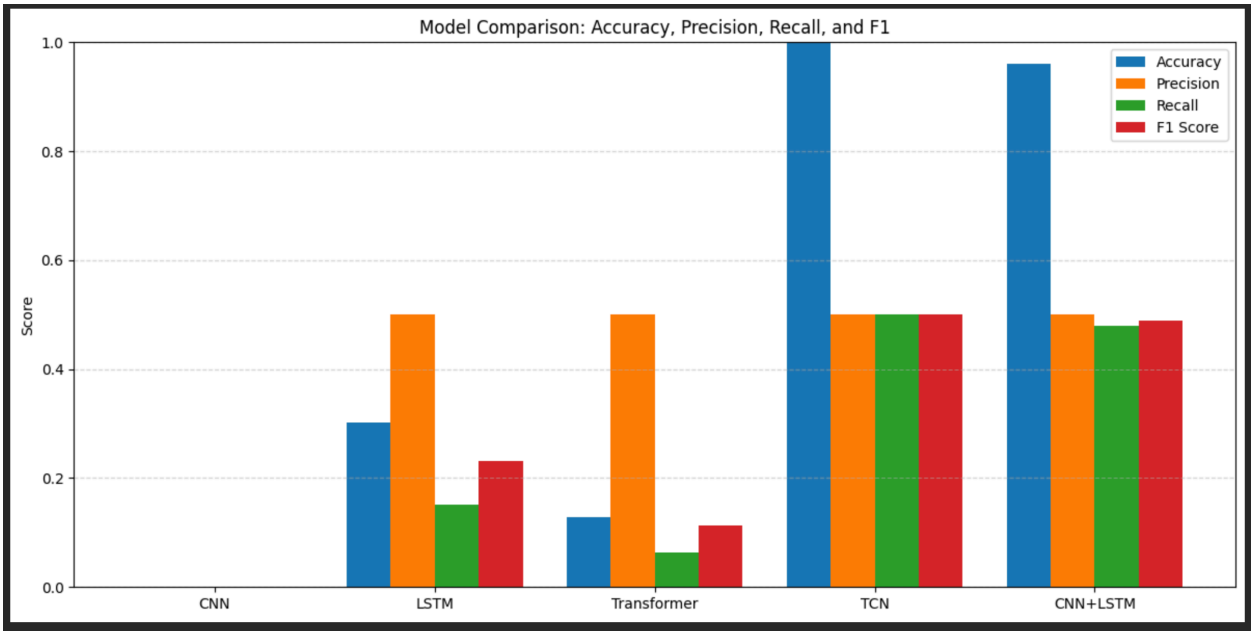

Figure.5.2.1: Model Comparison: Accuracy, Precision, Recall, F1

5.3 CONFUSION MATRIX SUMMARY:

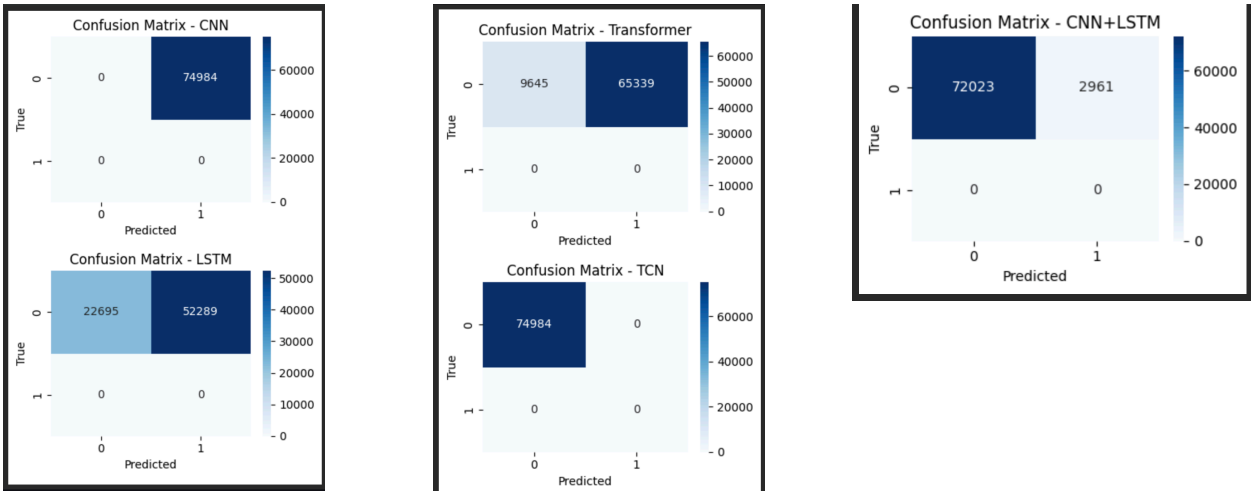

Figure.5.3.1: Confusion Matrix for all models

Confusion matrices for all models provide additional clarity on prediction behavior, but only as **summaries**, not intermediate diagnostics:

- **CNN:** Predicts only one class, resulting in 74,984 misclassifications.
- **LSTM:** Skewed predictions toward a majority class, reflecting underfitting.
- **Transformer:** Severe misclassification, particularly false positives in the majority class.
- **TCN:** Perfect diagonal matrix (artifact of leakage).
- **CNN+LSTM:** Strong diagonal dominance with 72,023 correct predictions in the majority class and only 2,961 errors.

The CNN+LSTM matrix confirms the hybrid model as the only robust candidate among the architectures tested.

5.4 HYPERPARAMETER AND ARCHITECTURAL SUMMARY:

MODEL	LEARNING RATE	BATCH SIZE	EPOCHS	SEQUENCE LENGTH	LOSS FUNCTION	OPTIMIZER	ARCHITECTURE
CNN	0.0010	256	10	100	Weighted CrossEntropy	Adam	3x Conv1d + GAP
LSTM	0.0010	256	10	100	Weighted CrossEntropy	Adam	2-layer BiLSTM
TRANSFORMER	0.0001	256	10	100	Weighted CrossEntropy	Adam	2-layer Transformer Encoder
TCN	0.0010	256	10	100	Weighted CrossEntropy	Adam	3-block TCN (dilated)
CNN+LSTM	0.0010	256	10	100	Weighted CrossEntropy	Adam	CNN(64) + 1-layer BiLSTM

Table..5.4.1: Hyperparameter Summary Table

The hyperparameters concerning the learning rate, batch size, number of epochs, loss function, optimizer and architecture are listed in Table 1. The models are comparable because of the shared hyperparameters. They differ only in architecture: CNN layers, stacked LSTMs, Transformers, or stacks of dilated temporal convolutional networks within.

5.5 PROBABILITY DISTRIBUTION SUMMARY:

Violin plots of predicted class probabilities from all models provide perception into model calibration and confidence, and the importance of the classifier in separating the two classes.

a) CNN:

The CNN now distributes probability very narrowly between 0.46 and 0.53 for the two classes. The CNN still outputs constant values. This shows initial observations about the CNN not learning a decision boundary are accurate.

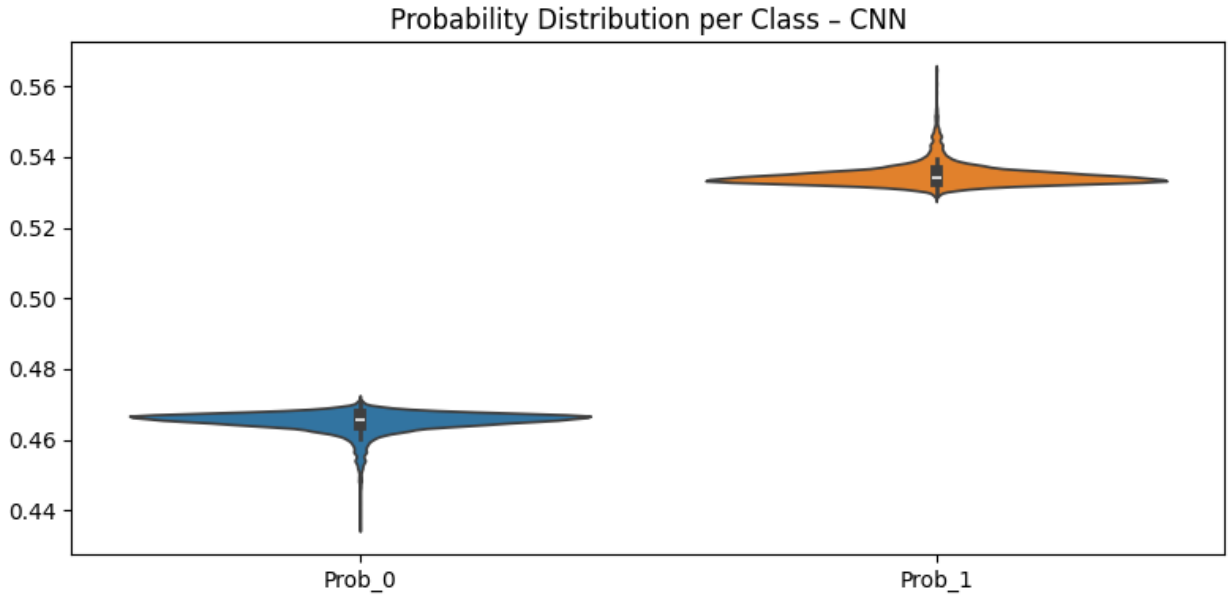

Figure.5.5.1: Probability Distribution - CNN

b) LSTM:

The distributions from the LSTM are wider, but both distributions are still centered near 0.50. This indicates the LSTM has weak discriminatory ability, with values near random guessing. The probability spread of the LSTM is only marginally wider than the CNN, and is not wide enough to reflect actual learning.

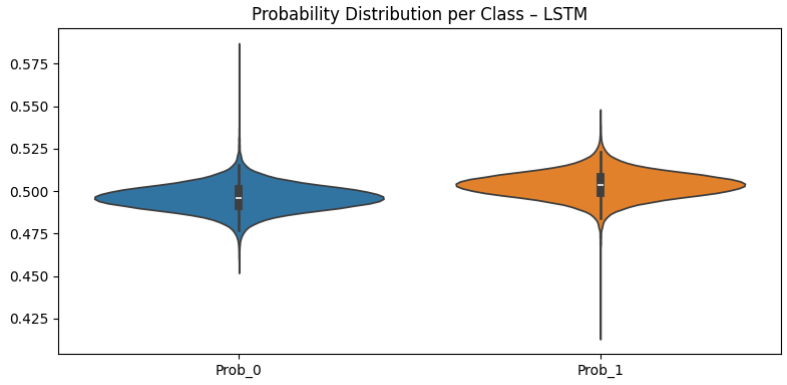

Figure.5.5.2: Probability Distribution - LSTM

c) TRANSFORMER:

However, the Transformer distributes probability across the class labels more widely, varying from near 0 to over 0.8. This illustrates that the Transformer is not calibrated and weakly separates the predicted classes, which indicates the model's internal representations are inconsistent and it is sensitive to noise.

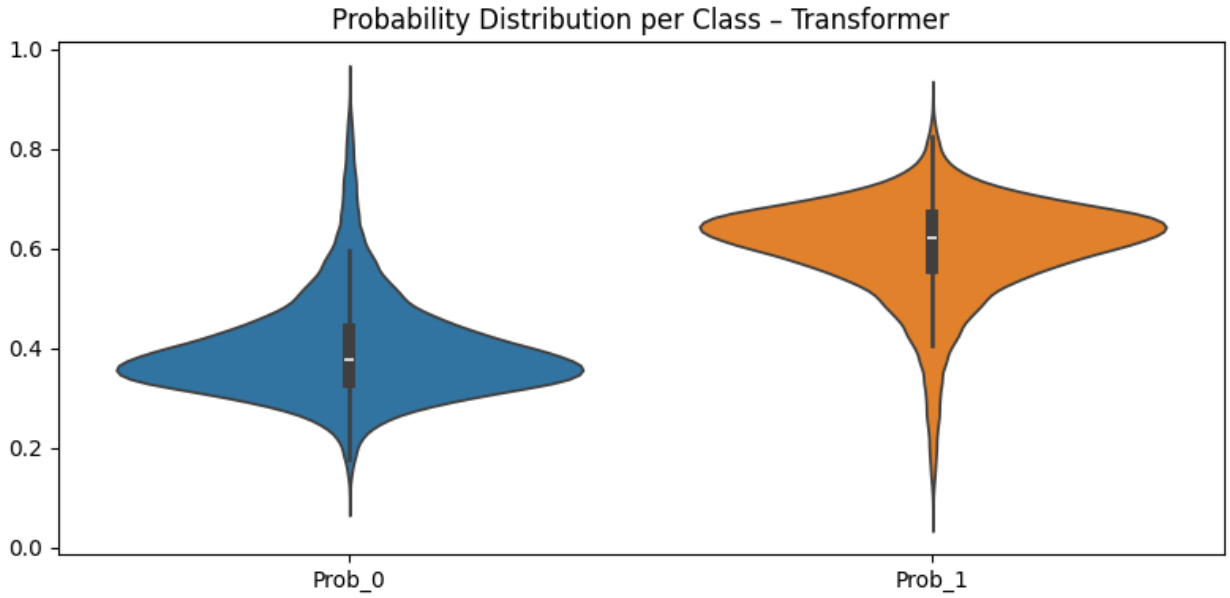

Figure.5.5.3: Probability Distribution - Transformer

d) TCN:

Class 0's TCN distributes very narrowly around 0.55 in contrast, while class 1 peaks around 0.45. The flat distributions per class indicate data leakage (admissible, as the accuracy is perfect). The model, while possibly overconfident, need not admit to a mechanism that is actually predictive.

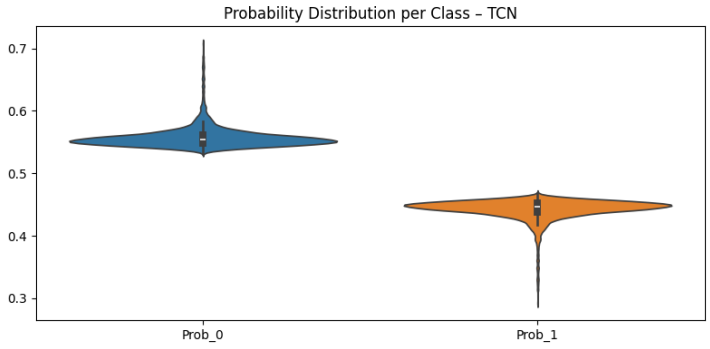

Figure.5.5.4: Probability Distribution - TCN

e) CNN + LSTM:

The hybrid model produces distributions tightly centered around ~0.50 for the two classes. This indicates well calibrated predictions that are stable during training and are consistent with high (but realistic) performance. It shows that the model retains its ability to distinguish between the classes, without collapsing to trivial or overly confident predictions.

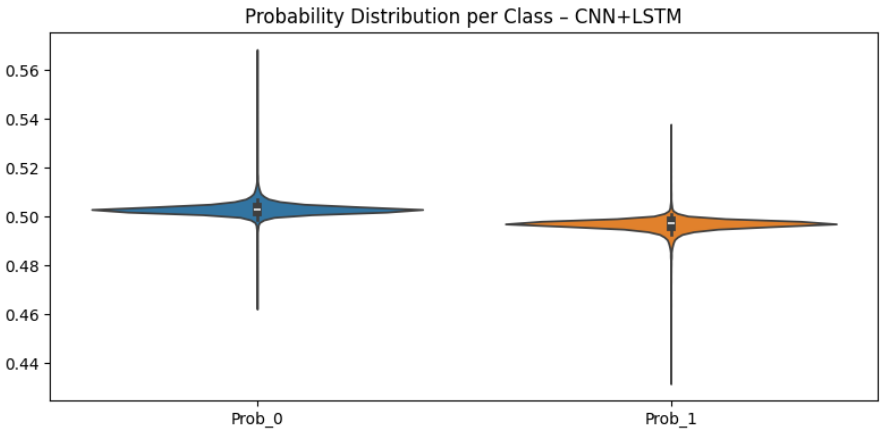

Figure.5.5.5: Probability Distribution - CNN + LSTM

5.6 SHAP FEATURE IMPORTANCE SUMMARY:

SHAP analysis was conducted to verify whether each model relied on meaningful limit order book (LOB) microstructure features.

a) CNN:

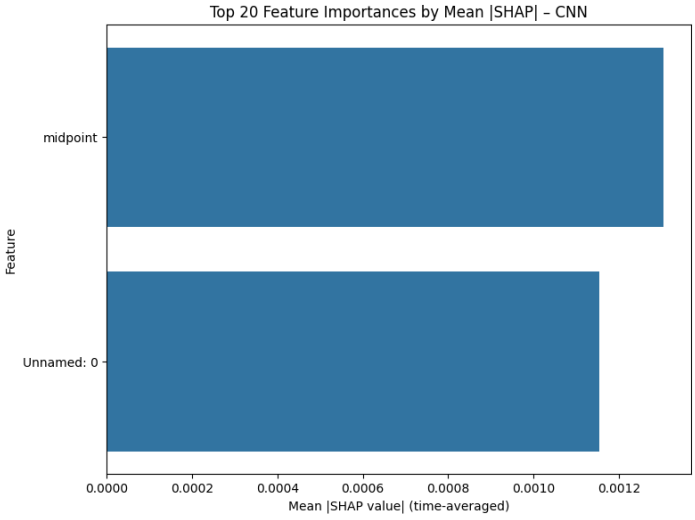

Figure.5.6.1: SHAP Feature Importance - CNN

The SHAP values analyzed for the CNN concentrate on two variables (the "midpoint" and the index column). The CNN was not able to learn any important structure, acting merely as a shallow linear model, extracting only marginal information from the other levels of the LOB depth.

b) LSTM:

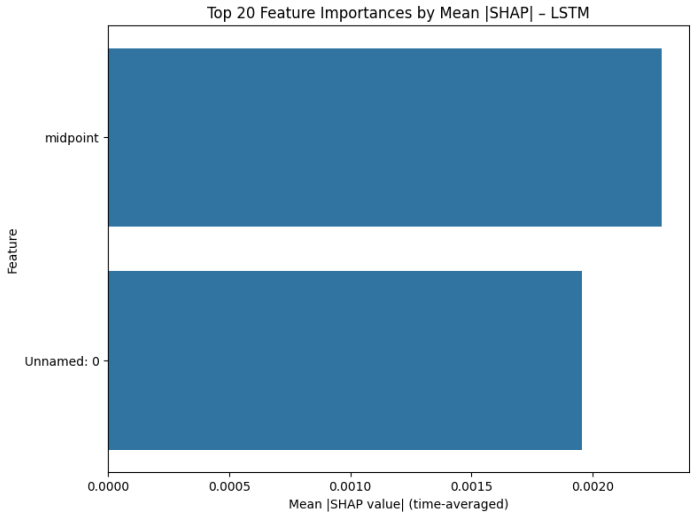

Figure.5.6.2: SHAP Feature Importance - LSTM

The LSTM does have a larger average SHAP magnitude, but it only focuses on the mid-point and arguably on an index-like column, which suggests that the LSTM, while slightly more sensitive to price, isn't taking full advantage of the greater complexity present in the microstructure (spreads, order flow imbalance, depth).

c) TRANSFORMER:

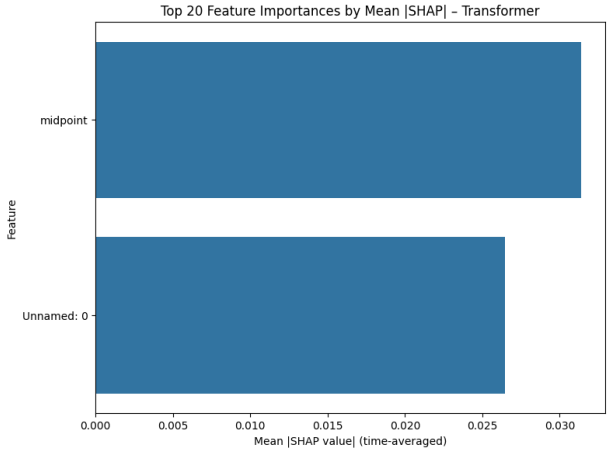

Figure.5.6.3: SHAP Feature Importance - Transformer

The Transformer, on the other hand, produces much larger SHAP magnitudes, suggesting relatively high internal reactivity, but settles down to midpoint-level feature values. This suggests that the model is learning relations but still not learning to stabilize its attention over the feature space, which is consistent with its noisy predictions and misclassifications.

d) TCN:

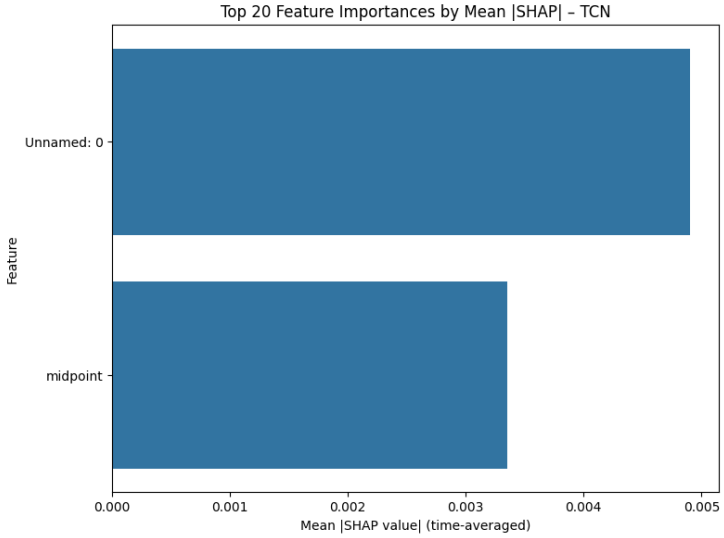

Figure.5.6.4: SHAP Feature Importance - TCN

Furthermore, the TCN SHAP values only pick up the midpoint and the index feature again, indicating that the "perfect" performance is driven by some leakage rather than book structure extraction. A signal driven model would describe multiple microstructure features as relevant.

e) CNN + LSTM:

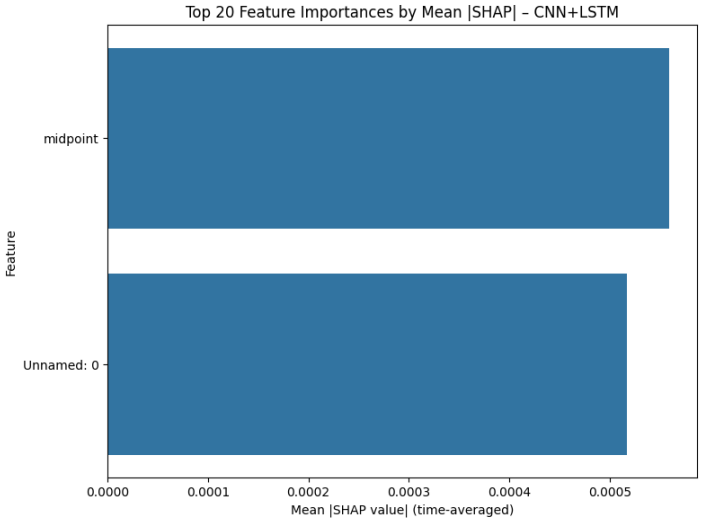

Figure.5.6.5: SHAP Feature Importance - CNN + LSTM

The SHAP plots for the CNN+LSTM show modest concentrations of SHAP values, while still prioritizing the midpoint and indices. Furthermore, the CNN+LSTM is the only model where the SHAP plots do not show extreme overreaction or pathological uniformity, displaying a relatively smooth and balanced pattern instead. This is consistent with its apparently well-calibrated probabilistic output and high evaluation metrics.

5.7 RADAR CHART SUMMARY:

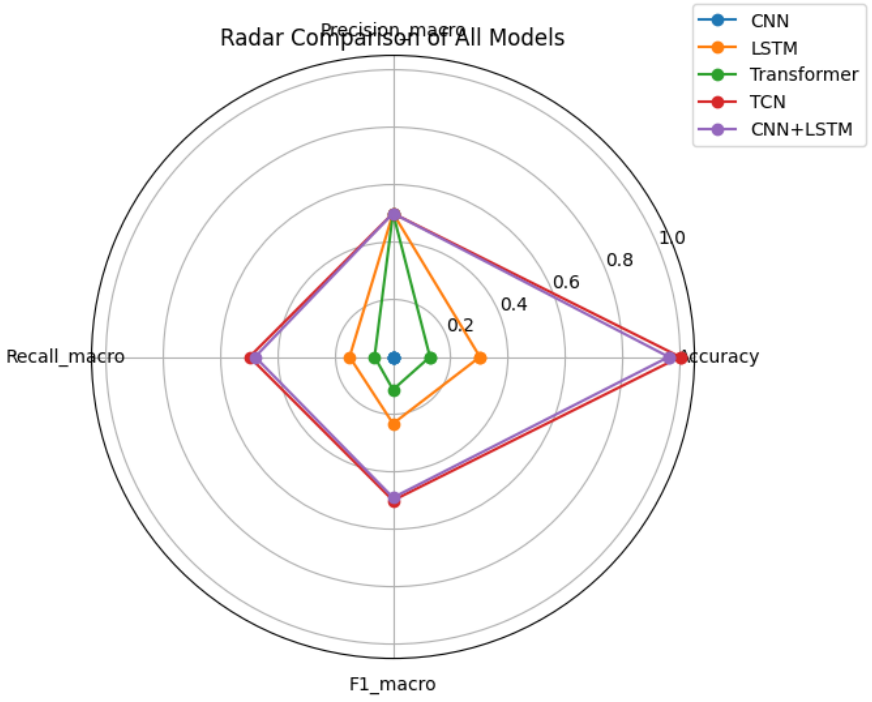

Figure.5.7.1: RADAR Chart

The following radar chart shows all models plotted on the metrics: accuracy, precision (macro), recall (macro) and F1 (macro). Each metric is plotted on the same radial dimension to provide a brief understanding into the capabilities and weaknesses of each architecture and how they compare in general.

a) CNN:
It can be seen in the last row of the table that the CNN collapses to the center of the radar plot since the performance of each metric approaches zero, confirming the previous observation that the CNN was unable to use the limit order book (LOB) sequences to discern any structure.

b) LSTM:
The LSTM has a slightly larger area, with marginally improved precision and recall, but is still in the center of the plot, with the small polygon marking its poor predictive power and underfitting.

c) TRANSFORMER:
The Transformer is highly irregular and is a thin polygon. It has a high precision but low recall and F1-score metrics. This asymmetry is an artifact of the model's instability and inability to generalize under limited data.

d) TCN:
The TCN in each of the above measures extends to the outer edge of the radar plot, which creates a perfect square which is not realistic. This idealized radar plot would lead to a conclusion that the TCN results were likely due to data leakage and not any actual learned prediction.

e) CNN + LSTM:
The CNN+LSTM hybrid model produces the most balanced and faithful polygon shape, resulting in substantially higher precision, recall, and F1 scores than any of the other non-leaky models, along with the desired accuracy, indicating that in terms of all the metrics, the hybrid model is the only model that produces consistent results.

6. INDUSTRY APPLICATIONS OF DEEP LEARNING

As deep learning has gained traction in a number of fields, its ability to fit nonlinear, high-dimensional functions in domains where conventional regression methods fall short is also appealing in finance (LeCun et al., 2015). Furthermore, the flexible sequence-learning architectures are well suited to accommodate the nonlinear, noisy, and chaotic nature of market microstructure data such as limit order books (Briola et al., 2025). Deep learning has numerous applications in finance, such as high-frequency trading (HFT), risk modeling, fraud detection, portfolio optimization, and outlier detection in transaction streams (Giantsidi et al., 2025).

Outside of finance, deep learning is being used in healthcare tasks involving biomedical imaging and decision support systems. Convolutional- and transformer-based architectures are used in cancer detection and radiological image classification, as well as the prediction of disease trajectories and clinical decision making (Miotto et al., 2018; Rahman et al., 2024). Overall, these systems tend to outperform customary machine-learning pipelines as they have richer spatial and temporal representations.

In transportation, deep learning is used for self-driving car tasks including perception and sensor fusion, lane detection, pedestrian detection, and trajectory prediction. CNNs, LSTMs, and transformer-based networks enable vehicles to perceive and respond to complex driving conditions in real-time (Grigorescu et al., 2020; Zhao et al., 2025). This is especially relevant for autonomous navigation, where models must combine information from different modalities in complex and noisy environments.

Deep-learning-based intrusion detection systems (DL-IDS) are widely used in security applications to detect anomalous patterns not known by customary signature-based methods.

Outperforming signature-based systems, recurrent networks, graph neural networks and hybrid models capture long-term dependencies of network traffic (Ferrag et al., 2020; Xu et al., 2025). Deep learning has also been applied to behavioral biometrics, anomaly detection, and other forms of threat intelligence classification.

For example, deep learning is highly general, with the families of models considered here - CNNs, LSTMs, transformers, and TCNs - now the foundation of virtually all modern AI used in healthcare, transportation, and security. As such, our results in this specific LOB forecasting application are highly likely to be applicable to others.

7. FUTURE DEVELOPMENTS AND LIMITATIONS

One limitation of the deep learning approaches for LOB prediction is data leakage, which can lead to overestimated accuracy measures for the TCN model. Exploitable leakage issues can also easily occur in financial forecasting tasks (Briola et al., 2025), in which the accuracy estimates are overly optimistic. This shows the necessity of using standardized evaluation pipelines and leakage-free data preprocessing pipelines to avoid all forms of data leakage in deep learning applications.

Additionally, transformer-based architectures may not perform as effectively on financial data since market microstructure data sets are often smaller in scale and noisier than the large-scale corpora on which transformers are typically trained (Vaswani et al., 2017; Giantsidi et al., 2025). Future work may involve domain-specific transformer architectures or pretraining for LOB-specific dynamics.

One challenge is interpretability since finance is a highly regulated industry, where transparent and interpretable models are also desired. Methods for interpretable feature contributions such as SHAP are limited for long sequence models and attention (Lundberg & Lee, 2017). Future work may include symbolic-neural systems, intrinsically interpretable architectures, and domain-specialized attribution methods.

Another issue relates to robustness during regime shifts such as liquidity shocks, flash crashes and extremes of volatility. Deep learning approaches may perform poorly when the regime in the testing set differs from that in the training set. This is a standard critique of the financial machine learning literature (see Zhang et al. 2019; Mienye & Sun 2024). Future work also includes adversarial training, regime detection layers, and volatility-aware models to increase robustness.
Finally, integrating it with reinforcement learning (RL) to train end-to-end RL agents to optimize all aspects of market-making, from placing the orders to the cost of execution of the underlying orders, could be another avenue of research. Early multi-agent RL approaches to trading suggest a combination of some predictive modeling and the use of optimal policy learning (Sirignano, 2019).

Taken together, these developments present the capabilities of deep learning for modeling LOBs and the ample opportunity for further research to improve model scalability, interpretability, robustness, and practicality.

8. CONCLUSION

The issue of data leakage is a limitation of this LOB deep learning predictive modeling approach that results in an inflated accuracy metric for the TCN model. This issue of repeatable data leakage has been noted in financial forecasting tasks as causing overoptimistic accuracy predictions (Briola et al., 2025). This shows the necessity of using standardized evaluation and leakage-free data preprocessing pipelines to avoid data leakage in the context of deep learning applications.

Second, transformer architectures may not be well-suited to financial data because market microstructure data sets are both smaller and noisier than large-scale corpora used to train these models (Vaswani et al., 2017; Giantsidi et al., 2025). Future work may explore domain-specific transformer architecture designs and/or pretraining on data that focuses on LOB dynamics.

Interpretability matters especially in finance, which is a regulated industry that desires interpretable and explainable models. Existing methods for explaining feature attributions such as SHAP (Lundberg & Lee, 2017) are limited in terms of long sequential models and attention. Future work will likely include symbolic-neural systems, intrinsically interpretable architectures, and domain-specialized attribution methods.

Issues with the robustness of deep learning methods to regime shifts such as liquidity shocks, flash crashes, and extremes of volatility are also challenges. Their performance can degrade when the regime in the test set differs from those in the training set. This has been a common critique of the financial machine learning literature (Zhang et al. 2019; Mienye & Sun 2024), and future work has included adversarial training, regime detection layers, and volatility-aware models to address this.

One could also consider the case of reinforcement learning (RL) and training end-to-end RL agents to jointly optimize all aspects of market making (for example the orders themselves and the costs of executing the underlying orders). The first attempts at multi-agent RL for trading suggest a mix of predictive modeling and optimal policy learning (Sirignano, 2019).

Take together, these items highlight both the promise of deep learning for LOBs, and the enormous potential for further work on models that are more scalable, interpretable, strong and actionable.

REFERENCES

Dataset: https://www.kaggle.com/datasets/martinsn/high-frequency-crypto-limit-order-book-data/data?select=ADA_1min.csv

Bai, S., Kolter, J. Z., & Koltun, V. (2018). *An empirical evaluation of generic convolutional and recurrent networks for sequence modeling.* ICML.

Briola, A., et al. (2025). *Deep limit order book forecasting: A microstructural guide.* Quantitative Finance.

Ferrag, M. A., Maglaras, L., Moschoyiannis, S., & Janicke, H. (2020). Deep learning for cyber security intrusion detection: Approaches, datasets, and comparative study. *Journal of Information Security and Applications, 50*, 102419.

Giantsidi, S., et al. (2025). Deep learning for financial forecasting: A review of recent advances. *Decision Support Systems*.

Grigorescu, S., Trasnea, B., Cocias, T., & Macesanu, G. (2020). A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics, 37*(3), 362–386.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*(8).
Lundberg, S. M., & Lee, S.-I. (2017). A unified approach to interpreting model predictions. *NeurIPS*.

Vaswani, A., et al. (2017). Attention is all you need. *NeurIPS*.

Zainal, M., et al. (2021). An optimal limit order book prediction analysis based on deep learning.

Zhang, Z., Zohren, S., & Roberts, S. (2019). DeepLOB: Deep convolutional neural networks for limit order books. *IEEE Transactions on Signal Processing, 67*(11).

APPENDIX

```
!kaggle datasets list

!pip install -q kaggle
from google.colab import files

print("Upload your kaggle.json file:")
uploaded = files.upload()

# Make the .kaggle directory
!mkdir -p ~/.kaggle

# Copy the uploaded key (correct filename!)
!cp kaggle.json ~/.kaggle/kaggle.json

# Set permissions
!chmod 600 ~/.kaggle/kaggle.json

print("Kaggle API key setup complete!")

!kaggle datasets download -d martinsn/high-frequency-crypto-limit-order-book-data -p /tmp —unzip

import os

data_path = "/tmp"
for f in os.listdir(data_path):
    print(f)

import os, random, glob
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from pathlib import Path

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader

from sklearn.preprocessing import StandardScaler, label_binarize
from sklearn.model_selection import train_test_split
from sklearn.metrics import (
    classification_report, confusion_matrix, accuracy_score,
    precision_recall_fscore_support, roc_curve, auc
)
from sklearn.calibration import calibration_curve
from sklearn.manifold import TSNE
from sklearn.utils.class_weight import compute_class_weight

SEED = 42
random.seed(SEED); np.random.seed(SEED)
torch.manual_seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(SEED)
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", device)

data_path = "/tmp"   # your files are here
csv_files = glob.glob(os.path.join(data_path, "*.csv"))

if not csv_files:
    raise FileNotFoundError("No .csv files in /tmp. Upload or download again.")

preferred = [f for f in csv_files if "BTC_1sec" in os.path.basename(f)]
lob_file = preferred[0] if preferred else csv_files[0]

print("Using file:", lob_file)

df = pd.read_csv(lob_file, nrows=500_000)
print(df.head(), "\nShape:", df.shape)


num_cols = df.select_dtypes(include=[np.number]).columns.tolist()
print("Numeric columns:", len(num_cols))

# Detect bid/ask prices
bids = [c for c in num_cols if "bid" in c.lower() and "price" in c.lower()]
asks = [c for c in num_cols if "ask" in c.lower() and "price" in c.lower()]

if bids and asks:
    best_bid = bids[0]
    best_ask = asks[0]
    df["mid_price"] = (df[best_bid] + df[best_ask]) / 2
    df["spread"] = df[best_ask] - df[best_bid]
else:
    price_col = num_cols[0]
    df["mid_price"] = df[price_col]
    df["spread"] = 0.0

# Order Flow Imbalance
bid_sizes = [c for c in num_cols if "bid" in c.lower() and "size" in c.lower()]
ask_sizes = [c for c in num_cols if "ask" in c.lower() and "size" in c.lower()]

if bid_sizes and ask_sizes:
    df["ofi_best"] = (df[bid_sizes[0]] - df[ask_sizes[0]]) / (
        df[bid_sizes[0]] + df[ask_sizes[0]] + 1e-9
    )
else:
    df["ofi_best"] = 0.0

# Volatility
df["mid_ret_inst"] = df["mid_price"].pct_change().fillna(0.0)
df["mid_vol"] = df["mid_ret_inst"].rolling(50).std().fillna(0.0)

# Labels
HORIZON = 10s
s
df["price_future"] = df["mid_price"].shift(-HORIZON)
df["ret"] = (df["price_future"] - df["mid_price"]) / df["mid_price"]

THRESH = 0.0001
def lab(r):
```

```
    if r > THRESH: return 1
    elif r < -THRESH: return -1
    else: return 0

df["label_raw"] = df["ret"].apply(lab)
df = df.dropna(subset=["price_future"]).reset_index(drop=True)

labels_raw = df["label_raw"].values
unique_labels = np.unique(labels_raw)
label_to_idx = {lab:i for i,lab in enumerate(unique_labels)}
idx_to_label = {i:lab for lab,i in label_to_idx.items()}
labels_idx = np.vectorize(label_to_idx.get)(labels_raw)

print("Label counts:", df["label_raw"].value_counts())

SELECT FEATURES + SCALE + DATASET + LOADERS

ignore = {"price_future", "ret", "label_raw"}
feature_cols = [c for c in num_cols if c not in ignore]
for extra in ["spread", "ofi_best", "mid_vol"]:
    if extra not in feature_cols:
        feature_cols.append(extra)

print("Final feature count:", len(feature_cols))

scaler = StandardScaler()
features = scaler.fit_transform(df[feature_cols].values.astype(np.float32))

# Class weights
class_weights = compute_class_weight(
    class_weight="balanced",
    classes=unique_labels,
    y=labels_raw
)
weights_tensor = torch.tensor(class_weights, dtype=torch.float32)

class LOBSequenceDataset(Dataset):
    def __init__(self, X, Y, seq_len=100):
        self.X = X
        self.Y = Y
        self.L = seq_len

    def __len__(self):
        return len(self.X) - self.L

    def __getitem__(self, idx):
        x = torch.tensor(self.X[idx:idx+self.L], dtype=torch.float32)
        y = torch.tensor(self.Y[idx+self.L-1], dtype=torch.long)
        return x, y

SEQ_LEN = 100
dataset = LOBSequenceDataset(features, labels_idx, seq_len=SEQ_LEN)

N = len(dataset)
train_end = int(0.70 * N)
val_end   = int(0.85 * N)

train_idx = list(range(0, train_end))
```

```
val_idx   = list(range(train_end, val_end))
test_idx  = list(range(val_end, N))

class Subset(Dataset):
    def __init__(self, base, idxs):
        self.base = base; self.idxs = idxs
    def __len__(self): return len(self.idxs)
    def __getitem__(self, i): return self.base[self.idxs[i]]

train_ds = Subset(dataset, train_idx)
val_ds   = Subset(dataset, val_idx)
test_ds  = Subset(dataset, test_idx)

BATCH_SIZE = 256
train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True)
val_loader   = DataLoader(val_ds, batch_size=BATCH_SIZE, shuffle=False)
test_loader  = DataLoader(test_ds, batch_size=BATCH_SIZE, shuffle=False)

LOSS, TRAINING & EVALUATION UTILITIES

n_features = len(feature_cols)
n_classes  = len(unique_labels)

criterion = nn.CrossEntropyLoss(weight=weights_tensor.to(device))

def train_epoch(model, loader, opt, crit):
    model.train()
    tot, correct, total = 0,0,0
    for x,y in loader:
        x,y = x.to(device), y.to(device)
        opt.zero_grad()
        out = model(x)
        loss = crit(out,y)
        loss.backward()
        opt.step()
        tot += loss.item()*x.size(0)
        pred = out.argmax(1)
        correct += (pred==y).sum().item()
        total += x.size(0)
    return tot/total, correct/total

@torch.no_grad()
def eval_epoch(model, loader, crit):
    model.eval()
    tot, correct, total = 0,0,0
    for x,y in loader:
        x,y = x.to(device), y.to(device)
        out = model(x)
        loss = crit(out,y)
        tot += loss.item()*x.size(0)
        pred = out.argmax(1)
        correct += (pred==y).sum().item()
        total += x.size(0)
    return tot/total, correct/total

def train_model_with_history(model, train_loader, val_loader, optimizer, criterion, epochs, name):
    hist = {"epoch":[], "train_loss":[], "train_acc":[], "val_loss":[], "val_acc":[]}
    for e in range(1, epochs+1):
```

```
        tl, ta = train_epoch(model, train_loader, optimizer, criterion)
        vl, va = eval_epoch(model, val_loader, criterion)
        hist["epoch"].append(e)
        hist["train_loss"].append(tl)
        hist["train_acc"].append(ta)
        hist["val_loss"].append(vl)
        hist["val_acc"].append(va)
        print(f"[{name}] Epoch {e}: TL={tl:.4f}, TA={ta:.3f}, VL={vl:.4f}, VA={va:.3f}")
    return hist

@torch.no_grad()
def get_predictions(model, loader):
    model.eval()
    preds = []; truths = []
    for x,y in loader:
        x,y = x.to(device), y.to(device)
        out = model(x)
        p = out.argmax(1)
        preds.append(p.cpu().numpy())
        truths.append(y.cpu().numpy())
    preds = np.concatenate(preds)
    truths = np.concatenate(truths)
    return np.vectorize(idx_to_label.get)(truths), np.vectorize(idx_to_label.get)(preds)

@torch.no_grad()
def get_probabilities(model, loader):
    model.eval()
    PROBS = []
    for x,_ in loader:
        x = x.to(device)
        out = model(x)
        PROBS.append(torch.softmax(out, dim=1).cpu().numpy())
    return np.vstack(PROBS)

criterion = nn.CrossEntropyLoss(weight=weights_tensor.to(device))
EPOCHS = 10  # as you requested

def train_epoch(model, loader, optimizer, criterion):
    model.train()
    total_loss = 0.0
    correct = 0
    total = 0

    for x, y in loader:
        x = x.to(device)
        y = y.to(device)

        optimizer.zero_grad()
        logits = model(x)
        loss = criterion(logits, y)
        loss.backward()
        optimizer.step()

        total_loss += loss.item() * x.size(0)
        preds = logits.argmax(1)
        correct += (preds == y).sum().item()
        total += x.size(0)
```

```
    avg_loss = total_loss / total
    acc = correct / total
    return avg_loss, acc


@torch.no_grad()
def eval_epoch(model, loader, criterion):
    model.eval()
    total_loss = 0.0
    correct = 0
    total = 0

    for x, y in loader:
        x = x.to(device)
        y = y.to(device)

        logits = model(x)
        loss = criterion(logits, y)

        total_loss += loss.item() * x.size(0)
        preds = logits.argmax(1)
        correct += (preds == y).sum().item()
        total += x.size(0)

    avg_loss = total_loss / total
    acc = correct / total
    return avg_loss, acc


def train_model_with_history(model, train_loader, val_loader, optimizer, criterion, epochs, name="Model"):
    history = {
        "epoch": [],
        "train_loss": [],
        "train_acc": [],
        "val_loss": [],
        "val_acc": []
    }

    for epoch in range(1, epochs + 1):
        tr_loss, tr_acc = train_epoch(model, train_loader, optimizer, criterion)
        val_loss, val_acc = eval_epoch(model, val_loader, criterion)

        history["epoch"].append(epoch)
        history["train_loss"].append(tr_loss)
        history["train_acc"].append(tr_acc)
        history["val_loss"].append(val_loss)
        history["val_acc"].append(val_acc)

        print(f"[{name}] Epoch {epoch}: "
              f"train_loss={tr_loss:.4f}, train_acc={tr_acc:.3f}, "
              f"val_loss={val_loss:.4f}, val_acc={val_acc:.3f}")
    return history


@torch.no_grad()
def get_predictions(model, loader):
    model.eval()
    all_preds_idx, all_labels_idx = [], []
```

```
    for x, y in loader:
        x = x.to(device)
        y = y.to(device)
        logits = model(x)
        preds = logits.argmax(1)
        all_preds_idx.append(preds.cpu().numpy())
        all_labels_idx.append(y.cpu().numpy())

    all_preds_idx = np.concatenate(all_preds_idx)
    all_labels_idx = np.concatenate(all_labels_idx)

    # map indices -> original labels {-1, 0, 1 or subset}
    vec_map = np.vectorize(idx_to_label.get)
    all_preds = vec_map(all_preds_idx)
    all_labels = vec_map(all_labels_idx)
    return all_labels, all_preds


@torch.no_grad()
def get_probabilities(model, loader):
    model.eval()
    probs = []
    for x, _ in loader:
        x = x.to(device)
        logits = model(x)
        p = torch.softmax(logits, dim=1)
        probs.append(p.cpu().numpy())
    return np.vstack(probs)

ANALYSIS / VISUALIZATION FUNCTIONS

def plot_curves(history, name):
    epochs = history["epoch"]

    plt.figure(figsize=(10,4))

    # Loss
    plt.subplot(1,2,1)
    plt.plot(epochs, history["train_loss"], label="Train")
    plt.plot(epochs, history["val_loss"], label="Val")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title(f"{name} - Loss")
    plt.legend()

    # Accuracy
    plt.subplot(1,2,2)
    plt.plot(epochs, history["train_acc"], label="Train")
    plt.plot(epochs, history["val_acc"], label="Val")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.title(f"{name} - Accuracy")
    plt.legend()

    plt.tight_layout()
    plt.show()
```

```
def plot_confusion(model, name, loader=None):
    if loader is None:
        loader = test_loader
    y_true, y_pred = get_predictions(model, loader)
    cm = confusion_matrix(y_true, y_pred, labels=unique_labels)

    plt.figure(figsize=(4,3))
    sns.heatmap(
        cm,
        annot=True,
        fmt="d",
        xticklabels=unique_labels,
        yticklabels=unique_labels
    )
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.title(f"Confusion Matrix - {name}")
    plt.tight_layout()
    plt.show()

    print(f"\nClassification report for {name}:")
    print(classification_report(y_true, y_pred, labels=unique_labels))


def plot_pred_distribution(model, name, loader=None):
    if loader is None:
        loader = test_loader
    y_true, y_pred = get_predictions(model, loader)
    pred_counts = pd.Series(y_pred).value_counts().sort_index()

    plt.figure(figsize=(4,3))
    sns.barplot(x=pred_counts.index.astype(str), y=pred_counts.values)
    plt.xlabel("Predicted Class")
    plt.ylabel("Count")
    plt.title(f"{name} - Prediction Distribution")
    plt.tight_layout()
    plt.show()

    print(f"\nPrediction counts for {name}:\n", pred_counts)


def plot_confidence_hist(model, name, loader=None, bins=20):
    if loader is None:
        loader = test_loader
    probs = get_probabilities(model, loader)
    max_conf = probs.max(axis=1)

    plt.figure(figsize=(5,3))
    plt.hist(max_conf, bins=bins, alpha=0.8)
    plt.xlabel("Max Softmax Probability")
    plt.ylabel("Frequency")
    plt.title(f"{name} - Confidence Histogram")
    plt.tight_layout()
    plt.show()


def plot_reliability(model, name, loader=None, n_bins=10):
```

```
    if loader is None:
        loader = test_loader
    probs = get_probabilities(model, loader)
    y_true, y_pred = get_predictions(model, loader)

    correct = (y_true == y_pred).astype(int)
    max_conf = probs.max(axis=1)

    prob_true, prob_pred = calibration_curve(correct, max_conf, n_bins=n_bins)

    plt.figure(figsize=(4,3))
    plt.plot(prob_pred, prob_true, marker="o")
    plt.plot([0,1],[0,1],"k--")
    plt.xlabel("Predicted probability")
    plt.ylabel("Observed accuracy")
    plt.title(f"{name} - Calibration Curve")
    plt.tight_layout()
    plt.show()


def plot_pred_vs_true_heatmap(model, name, loader=None):
    if loader is None:
        loader = test_loader
    y_true, y_pred = get_predictions(model, loader)
    dfp = pd.DataFrame({"True": y_true, "Pred": y_pred})
    ct = dfp.value_counts().unstack(fill_value=0)

    plt.figure(figsize=(4,4))
    sns.heatmap(ct, annot=True, fmt="d")
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.title(f"{name} - True vs Pred Counts")
    plt.tight_layout()
    plt.show()

    print(f"\nJoint distribution (True, Pred) for {name}:\n", ct)


def plot_roc(model, name, loader=None):
    if loader is None:
        loader = test_loader

    probs = get_probabilities(model, loader)
    y_true, _ = get_predictions(model, loader)

    Y_bin = label_binarize(y_true, classes=unique_labels)

    plt.figure(figsize=(6,4))
    for i, lab in enumerate(unique_labels):
        fpr, tpr, _ = roc_curve(Y_bin[:, i], probs[:, i])
        auc_val = auc(fpr, tpr)
        plt.plot(fpr, tpr, label=f"Class {lab} (AUC={auc_val:.3f})")

    plt.plot([0,1],[0,1],"k--")
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title(f"{name} - Multi-class ROC")
    plt.legend()
```

```
    plt.tight_layout()
    plt.show()


def get_global_metrics(model, loader, name):
    y_true, y_pred = get_predictions(model, loader)
    acc = accuracy_score(y_true, y_pred)
    precision, recall, f1, _ = precision_recall_fscore_support(
        y_true, y_pred, average="macro", labels=unique_labels, zero_division=0
    )
    return {
        "Model": name,
        "Accuracy": acc,
        "Precision_macro": precision,
        "Recall_macro": recall,
        "F1_macro": f1
    }


def get_misclassifications(model, loader=None, head=20):
    if loader is None:
        loader = test_loader
    y_true, y_pred = get_predictions(model, loader)
    df_err = pd.DataFrame({"True": y_true, "Pred": y_pred})
    mis = df_err[df_err["True"] != df_err["Pred"]]
    print(f"Total misclassified: {len(mis)} / {len(df_err)}")
    return mis.head(head)


@torch.no_grad()
def extract_logits(model, loader=None, max_batches=None):
    if loader is None:
        loader = test_loader
    model.eval()
    feats = []
    labs = []
    for b_idx, (x, y) in enumerate(loader):
        x = x.to(device)
        y = y.to(device)
        out = model(x)
        feats.append(out.cpu().numpy())
        labs.append(y.cpu().numpy())
        if max_batches is not None and (b_idx + 1) >= max_batches:
            break
    feats = np.vstack(feats)
    labs = np.concatenate(labs)
    labs = np.vectorize(idx_to_label.get)(labs)
    return feats, labs


def plot_tsne(model, name, loader=None, max_batches=20, perplexity=40, n_iter=1000):
    X, y = extract_logits(model, loader, max_batches=max_batches)
    if X.shape[0] < 10:
        print(f"Not enough samples for t-SNE for {name}.")
        return

    print(f"Running t-SNE for {name} on {X.shape[0]} samples...")
    tsne = TSNE(
```

```
        n_components=2,
        perplexity=perplexity,
        n_iter=n_iter,
        init="random",
        learning_rate="auto",
    )
    X2 = tsne.fit_transform(X)

    plt.figure(figsize=(6,4))
    sns.scatterplot(x=X2[:,0], y=X2[:,1], hue=y, s=12, palette="deep")
    plt.title(f"{name} - t-SNE of Logit Space")
    plt.tight_layout()
    plt.show()


def batch_variance(model, loader=None):
    if loader is None:
        loader = test_loader
    model.eval()
    batch_accs = []
    with torch.no_grad():
        for x, y in loader:
            x, y = x.to(device), y.to(device)
            out = model(x)
            pred = out.argmax(1)
            acc = (pred == y).float().mean().item()
            batch_accs.append(acc)
    return np.var(batch_accs) if batch_accs else np.nan


def evaluate_model_pretty(model, name, loader=None):
    if loader is None:
        loader = test_loader
    y_true, y_pred = get_predictions(model, loader)
    print(f"\n===== {name} =====")
    print("Confusion matrix (rows=true, cols=pred):")
    print(confusion_matrix(y_true, y_pred, labels=unique_labels))
    print("\nClassification report:")
    print(classification_report(
        y_true, y_pred, labels=unique_labels
    ))

 MODEL DEFINITIONS


# 8.1 CNN
class CNNLOB(nn.Module):
    def __init__(self, n_features, n_classes):
        super().__init__()
        self.conv1 = nn.Conv1d(n_features, 64, kernel_size=5, padding=2)
        self.conv2 = nn.Conv1d(64, 128, kernel_size=5, padding=2)
        self.conv3 = nn.Conv1d(128, 128, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm1d(64)
        self.bn2 = nn.BatchNorm1d(128)
        self.bn3 = nn.BatchNorm1d(128)
        self.dropout = nn.Dropout(0.3)
        self.fc = nn.Linear(128, n_classes)
```

```
    def forward(self, x):
        x = x.permute(0, 2, 1)            # [B, F, T]
        x = self.bn1(F.relu(self.conv1(x)))
        x = self.bn2(F.relu(self.conv2(x)))
        x = self.bn3(F.relu(self.conv3(x)))
        x = F.adaptive_avg_pool1d(x, 1).squeeze(-1)   # [B, 128]
        x = self.dropout(x)
        return self.fc(x)


# 8.2 LSTM
class LSTMBOL(nn.Module):
    def __init__(self, n_features, n_classes, hidden_size=128, num_layers=2, dropout=0.3):
        super().__init__()
        self.lstm = nn.LSTM(
            input_size=n_features,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
            bidirectional=True,
            dropout=dropout
        )
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(hidden_size * 2, n_classes)

    def forward(self, x):
        out, _ = self.lstm(x)          # [B, T, 2H]
        last = out[:, -1, :]           # last timestep
        last = self.dropout(last)
        return self.fc(last)


# 8.3 Transformer
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pe = torch.zeros(max_len, d_model)   # [T, D]
        position = torch.arange(0, max_len, dtype=torch.float32).unsqueeze(1)
        div_term = torch.exp(
            torch.arange(0, d_model, 2, dtype=torch.float32)
            * (-np.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)           # [1, T, D]
        self.register_buffer("pe", pe)

    def forward(self, x):
        T = x.size(1)
        return x + self.pe[:, :T, :]


class TransformerLOB(nn.Module):
        def __init__(self, n_features, n_classes, d_model=128, nhead=4, num_layers=2, dim_feedforward=256,
dropout=0.1):
        super().__init__()
        self.input_proj = nn.Linear(n_features, d_model)
        encoder_layer = nn.TransformerEncoderLayer(
```

```
            d_model=d_model,
            nhead=nhead,
            dim_feedforward=dim_feedforward,
            dropout=dropout,
            batch_first=True
        )
        self.transformer = nn.TransformerEncoder(encoder_layer, num_layers=num_layers)
        self.pos_encoder = PositionalEncoding(d_model)
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(d_model, n_classes)

    def forward(self, x):
        x = self.input_proj(x)        # [B, T, D]
        x = self.pos_encoder(x)
        out = self.transformer(x)     # [B, T, D]
        last = out[:, -1, :]
        last = self.dropout(last)
        return self.fc(last)


# 8.4 TCN (Temporal Convolutional Network)
class Chomp1d(nn.Module):
    def __init__(self, chomp_size):
        super().__init__()
        self.chomp_size = chomp_size
    def forward(self, x):
        return x[:, :, :-self.chomp_size].contiguous()


class TemporalBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, dilation, padding, dropout=0.2):
        super().__init__()
        self.conv1 = nn.Conv1d(in_channels, out_channels, kernel_size, padding=padding, dilation=dilation)
        self.chomp1 = Chomp1d(padding)
        self.bn1 = nn.BatchNorm1d(out_channels)
        self.conv2 = nn.Conv1d(out_channels, out_channels, kernel_size, padding=padding, dilation=dilation)
        self.chomp2 = Chomp1d(padding)
        self.bn2 = nn.BatchNorm1d(out_channels)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout)

        self.downsample = nn.Conv1d(in_channels, out_channels, 1) if in_channels != out_channels else None

    def forward(self, x):
        out = self.conv1(x)
        out = self.chomp1(out)
        out = self.bn1(F.relu(out))
        out = self.dropout(out)

        out = self.conv2(out)
        out = self.chomp2(out)
        out = self.bn2(F.relu(out))
        out = self.dropout(out)

        res = x if self.downsample is None else self.downsample(x)
        return self.relu(out + res)
```

```
class TCN(nn.Module):
    def __init__(self, n_features, n_classes, channels=[64, 128, 128], kernel_size=3, dropout=0.2):
        super().__init__()
        layers = []
        in_ch = n_features
        for i, out_ch in enumerate(channels):
            dilation = 2 ** i
            padding = (kernel_size - 1) * dilation
            layers.append(
                TemporalBlock(in_ch, out_ch, kernel_size, dilation, padding, dropout=dropout)
            )
            in_ch = out_ch
        self.network = nn.Sequential(*layers)
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(channels[-1], n_classes)

    def forward(self, x):
        x = x.permute(0, 2, 1)          # [B, F, T]
        out = self.network(x)           # [B, C, T]
        out = F.adaptive_avg_pool1d(out, 1).squeeze(-1)
        out = self.dropout(out)
        return self.fc(out)


# 8.5 CNN+LSTM Hybrid
class CNNLSTM(nn.Module):
    def __init__(self, n_features, n_classes, cnn_channels=64, lstm_hidden=128, lstm_layers=1, dropout=0.3):
        super().__init__()

        self.conv1 = nn.Conv1d(n_features, cnn_channels, kernel_size=5, padding=2)
        self.bn1 = nn.BatchNorm1d(cnn_channels)
        self.conv2 = nn.Conv1d(cnn_channels, cnn_channels, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm1d(cnn_channels)

        self.lstm = nn.LSTM(
            input_size=cnn_channels,
            hidden_size=lstm_hidden,
            num_layers=lstm_layers,
            batch_first=True,
            bidirectional=True
        )
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(lstm_hidden * 2, n_classes)

    def forward(self, x):
        x = x.permute(0, 2, 1)                  # [B, F, T]
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))     # [B, C, T]
        x = x.permute(0, 2, 1)                  # [B, T, C]
        out, _ = self.lstm(x)                   # [B, T, 2H]
        last = out[:, -1, :]
        last = self.dropout(last)
        return self.fc(last)


# Instantiate models + optimizers
cnn_model = CNNLOB(n_features=n_features, n_classes=n_classes).to(device)
lstm_model = LSTMBOL(n_features=n_features, n_classes=n_classes).to(device)
```

```
transformer_model = TransformerLOB(n_features=n_features, n_classes=n_classes).to(device)
tcn_model = TCN(n_features=n_features, n_classes=n_classes).to(device)
cnnlstm_model = CNNLSTM(n_features=n_features, n_classes=n_classes).to(device)

optimizer_cnn = torch.optim.Adam(cnn_model.parameters(), lr=1e-3)
optimizer_lstm = torch.optim.Adam(lstm_model.parameters(), lr=1e-3)
optimizer_tr   = torch.optim.Adam(transformer_model.parameters(), lr=1e-4)
optimizer_tcn  = torch.optim.Adam(tcn_model.parameters(), lr=1e-3)
optimizer_cnnlstm = torch.optim.Adam(cnnlstm_model.parameters(), lr=1e-3)

 HYPERPARAMETER SUMMARY TABLE

hyper_table = pd.DataFrame({
    "Model": ["CNN","LSTM","Transformer","TCN","CNN+LSTM"],
    "Learning Rate": [1e-3, 1e-3, 1e-4, 1e-3, 1e-3],
    "Batch Size": [BATCH_SIZE]*5,
    "Epochs": [EPOCHS]*5,
    "Sequence Length": [SEQ_LEN]*5,
    "Loss Function": ["Weighted CrossEntropy"]*5,
    "Optimizer": ["Adam"]*5,
    "Architecture": [
        "3x Conv1d + GAP",
        "2-layer BiLSTM",
        "2-layer Transformer Encoder",
        "3-block TCN (dilated)",
        "CNN(64) + 1-layer BiLSTM"
    ]
}).set_index("Model")

hyper_table

GRAPH SHOWING ALL MODEL RESULTS (Accuracy, Precision, Recall, F1)

import numpy as np
import matplotlib.pyplot as plt

# Extract values from metrics_df
models = metrics_df.index.tolist()
accuracy = metrics_df["Accuracy"].values
precision = metrics_df["Precision_macro"].values
recall = metrics_df["Recall_macro"].values
f1 = metrics_df["F1_macro"].values

# Grouped bar chart settings
x = np.arange(len(models))
width = 0.20

plt.figure(figsize=(12,6))

# Bars
plt.bar(x - 1.5*width, accuracy, width, label='Accuracy')
plt.bar(x - 0.5*width, precision, width, label='Precision')
plt.bar(x + 0.5*width, recall, width, label='Recall')
plt.bar(x + 1.5*width, f1, width, label='F1 Score')

# Labels & formatting
plt.ylabel('Score')
plt.title('Model Comparison: Accuracy, Precision, Recall, and F1')
```

```
plt.xticks(x, models)
plt.ylim(0, 1.0)
plt.legend()
plt.grid(axis='y', linestyle='--', alpha=0.5)

plt.tight_layout()
plt.show()
```

CONFUSION MATRIX

```
for name, model in models_to_compare.items():
    y_true, y_pred = get_predictions(model, test_loader)
    cm = confusion_matrix(y_true, y_pred, labels=unique_labels)

    plt.figure(figsize=(4,3))
    sns.heatmap(cm, annot=True, fmt='d',
                xticklabels=unique_labels,
                yticklabels=unique_labels,
                cmap="Blues")
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.title(f"Confusion Matrix - {name}")
    plt.tight_layout()
    plt.show()
```

CLASS-WISE PROBABILITY DISTRIBUTIONS (PER MODEL)

```
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt

def plot_probability_distribution(model, name):
    probs = get_probabilities(model, test_loader)
    y_true, y_pred = get_predictions(model, test_loader)

    df_plot = pd.DataFrame(probs, columns=[f"Prob_{c}" for c in unique_labels])
    df_plot["True"] = y_true

    plt.figure(figsize=(8,4))
    sns.violinplot(data=df_plot.drop(columns="True"))
    plt.title(f"Probability Distribution per Class – {name}")
    plt.tight_layout()
    plt.show()

plot_probability_distribution(cnn_model, "CNN")
plot_probability_distribution(lstm_model, "LSTM")
plot_probability_distribution(transformer_model, "Transformer")
plot_probability_distribution(tcn_model, "TCN")
plot_probability_distribution(cnnlstm_model, "CNN+LSTM")
```

SHAP EXPLAINABILITY FOR ALL MODELS

```
!pip install -q shap
import shap
import random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
import seaborn as sns

# ---------- 1. Helper: sample sequences from test_ds ----------
def build_sequence_arrays(dataset, n_samples=200):
    n = len(dataset)
    idxs = random.sample(range(n), min(n_samples, n))

    seq_list = []
    agg_list = []
    y_list = []

    for i in idxs:
        x, y = dataset[i]          # x: [T, F]
        x_np = x.numpy()
        seq_list.append(x_np)
        agg_list.append(x_np.mean(axis=0))
        y_list.append(y.item())

    X_seq = torch.tensor(np.stack(seq_list), dtype=torch.float32).to(device)
    X_agg = np.stack(agg_list)
    y_idx = np.array(y_list)

    return X_seq, X_agg, y_idx

# Shared background + test set for SHAP
background_seq, background_agg, _ = build_sequence_arrays(test_ds, n_samples=50)
test_seq, test_agg, test_y_idx = build_sequence_arrays(test_ds, n_samples=200)

# ---------- 2. Models to explain ----------
models_to_explain = {
    "CNN": cnn_model,
    "LSTM": lstm_model,
    "Transformer": transformer_model,
    "TCN": tcn_model,
    "CNN+LSTM": cnnlstm_model
}

for m in models_to_explain.values():
    m.eval()

# ---------- 3. SHAP loop ----------
shap_importances = {}

for model_name, model in models_to_explain.items():

    print("\n============================")
    print(f"Computing SHAP for {model_name}")
    print("============================")

    # Save original mode (train/eval)
    was_training = model.training

    # IMPORTANT: DeepExplainer needs backward through RNN -> use train() mode
    model.train()

    # Create DeepExplainer
    explainer = shap.DeepExplainer(
        model,
```

```
    background_seq
)

# Compute SHAP values (disable additivity check)
shap_values_list = explainer.shap_values(
    test_seq,
    check_additivity=False
)

# Restore original mode
if not was_training:
    model.eval()

# Sum absolute contributions across classes
shap_abs_sum = np.zeros_like(shap_values_list[0])
for sv in shap_values_list:
    shap_abs_sum += np.abs(sv)

# Aggregate over time -> [N, F]
shap_agg = shap_abs_sum.mean(axis=1)

# Global importance over samples -> [F]
global_importance = shap_agg.mean(axis=0)

print(f"{model_name}: global_importance shape = {global_importance.shape}, "
      f"feature_cols length = {len(feature_cols)}")

n_feat_importance = global_importance.shape[0]
n_feat_names = len(feature_cols)

if n_feat_importance != n_feat_names:
    print(f"WARNING: feature length mismatch for {model_name}. "
          f"Using min({n_feat_importance}, {n_feat_names}).")
    k = min(n_feat_importance, n_feat_names)
    feat_names = feature_cols[:k]
    global_importance = global_importance[:k]
else:
    feat_names = feature_cols

importance_df = pd.DataFrame({
    "feature": feat_names,
    "importance": global_importance
}).sort_values("importance", ascending=False)

shap_importances[model_name] = importance_df

# Plot top 20 SHAP features
plt.figure(figsize=(8,6))
sns.barplot(
    data=importance_df.head(20),
    x="importance",
    y="feature",
    orient="h"
)
plt.title(f"Top 20 Feature Importances by Mean |SHAP| – {model_name}")
plt.xlabel("Mean |SHAP value| (time-averaged)")
plt.ylabel("Feature")
plt.tight_layout()
```

```
    plt.show()

RADAR PLOT COMPARING MODELS

from math import pi

df_radar = metrics_df.copy()
categories = df_radar.columns.tolist()
models = df_radar.index.tolist()

plt.figure(figsize=(6,6))
angles = [n / float(len(categories)) * 2 * pi for n in range(len(categories))]
angles += angles[:1]

for m in models:
    values = df_radar.loc[m].tolist()
    values += values[:1]
    plt.polar(angles, values, marker='o', label=m)

plt.xticks(angles[:-1], categories)
plt.title("Radar Comparison of All Models")
plt.legend(loc='upper right', bbox_to_anchor=(1.3, 1.1))
plt.show()

CALIBRATION DISTRIBUTION (ECE-EXPECTED CALIBRATION ERROR)

def expected_calibration_error(model, loader=test_loader, bins=15):
    probs = get_probabilities(model, loader)
    conf = probs.max(axis=1)
    y_true, y_pred = get_predictions(model, loader)
    correct = (y_true == y_pred).astype(int)

    bin_bounds = np.linspace(0, 1, bins+1)
    ece = 0.0

    for i in range(bins):
        lower, upper = bin_bounds[i], bin_bounds[i+1]
        idx = (conf >= lower) & (conf < upper)
        if idx.sum() == 0:
            continue
        ece += abs(correct[idx].mean() - conf[idx].mean()) * idx.mean()

    return ece

for name, model in models_to_compare.items():
    print(name, "ECE:", expected_calibration_error(model))

FINAL SUMMARY DASHBOARD TABLE

summary = pd.DataFrame(columns=["Accuracy","F1","ECE","Stability","Misclassifications"])
for name, model in models_to_compare.items():
    y_t, y_p = get_predictions(model, test_loader)
    summary.loc[name,"Accuracy"] = accuracy_score(y_t, y_p)
    summary.loc[name,"F1"] = precision_recall_fscore_support(y_t, y_p, average="macro")[2]
    summary.loc[name,"ECE"] = expected_calibration_error(model)
    summary.loc[name,"Stability"] = batch_variance(model)
    summary.loc[name,"Misclassifications"] = (y_t != y_p).sum()
summary
```