Venkata Krishna Anirudh Nuti

Blockhouse Take-Home Assignment Report

Table of Contents

Introduction

This report details the implementation and fine-tuning of a transformer-based model designed to generate trade recommendations for the stock market. The project aimed to leverage advanced machine learning techniques to process trade and market data, ultimately providing actionable insights for trading decisions.

Project Overview

The task involved developing a transformer model using PyTorch, fine-tuning it on a provided dataset to forecast market trends and technical indicators, and generating Buy, Sell, or Hold signals. These predictions were then used to execute trades in a trading blotter. The objective was to leverage the transformer model to predict accurate signals, create a trading blotter, and evaluate its performance against the given simple blotter.

Methodology

Data Preparation and Feature Engineering

A crucial step in the project was the creation of a feature set through feature engineering. The following technical indicators were calculated from the existing Close, Open, High, Low, and Volume data:

Trend Indicators:

- Moving Averages (MA)
- Exponential Moving Averages (EMA)
- Ichimoku Cloud (ICH)
- Average Directional Index (ADX)
- Directional Index (DI)

Momentum Indicators:

- Momentum (MOM)
- Rate of Change (ROC)
- Commodity Channel Index (CCI)
- Moving Average Convergence/Divergence (MACD)
- Relative Strength Index (RSI)
- Stochastic Oscillator (Stoch)

Volume Indicators:

- On Balance Volume (OBV)
- Accumulation/Distribution Line (ADL)
- OBV Mean

Volatility Indicators:

- Bollinger Bands (BB)
- Average True Range (ATR)

Price Indicators:

- Time-Weighted Average Price (TWAP)
- Volume-Weighted Average Price (VWAP)

These particular technical indicators were chosen because of their popularity, effectiveness, relevance to trading strategies, data availability (using common stock market data like close, open, high, low, and volume), and computational efficiency.

Label Generation

To create a supervised learning problem, labels were generated using Moving Average and RSI thresholds. Specifically:

- Moving Averages with 50 and 200 time periods were used to capture short-term and long-term trends respectively.

- RSI thresholds of 30 and 70 were chosen to identify oversold and overbought conditions.

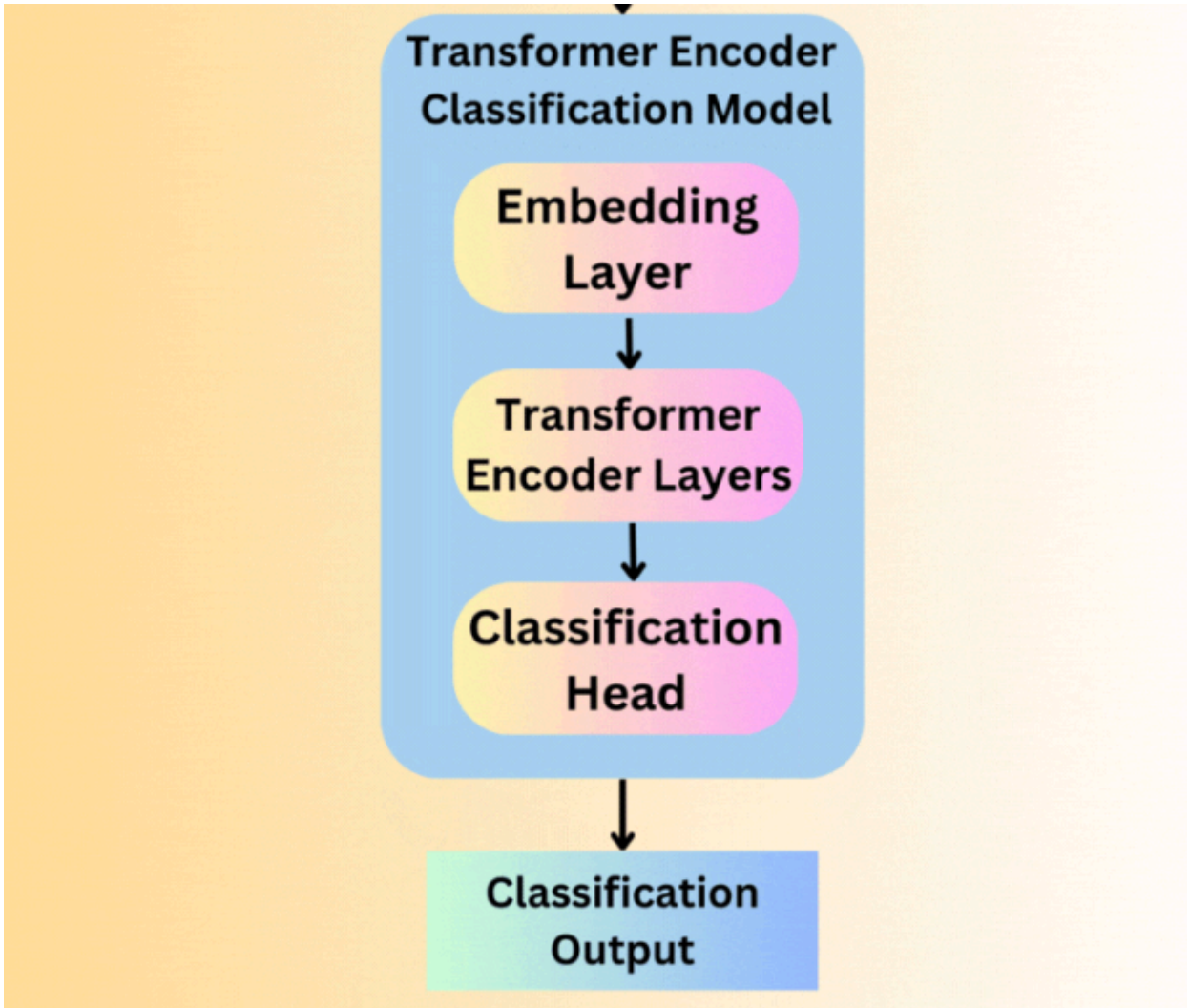
The resulting class distribution was:

- Hold (0): 9,430 instances
- Buy (1): 36,335 instances
- Sell (2): 13,307 instances

This distribution highlighted a class imbalance issue, which was addressed during the model training phase.

Model Architecture

The core of the project is a Transformer Encoder model, fine-tuned with a Classification Head. This architecture was chosen for its ability to capture complex relationships in sequential data, making it well-suited for financial time series analysis.



The architecture consists of:

1. **Input Layer:** The model takes the input of shape (batch_size, seq_len, input_dim). The seq_length represents the number of time steps or elements in each input sequence. In NLP terms, this is like the number of tokens in a sentence.
2. **Embedding Layer:** This linear layer projects the input features to a higher-dimensional space (d_model). It's not a traditional embedding layer used for discrete tokens, but rather a linear projection for continuous input features.
3. **Transformer Encoder:** This is the core of the model. It consists of multiple identical layers (num_layers) of TransformerEncoderLayer. Each layer has:
   - Multi-head self-attention mechanism
   - Feedforward neural network
   - Layer normalization and residual connections
4. **Classification Head:** This is the final layer that maps the transformer's output to class probabilities.
5. **Dropout Layer:** Applied before classification to prevent overfitting.

Steps involved in the forward pass:

- The input x is first passed through the embedding layer.
- The embedded input is then permuted to match the expected input shape of the transformer: (seq_len, batch_size, d_model).
- The transformer encoder processes the sequence.
- The last element of the sequence output is selected (x[-1]). This assumes that the last token's representation captures the necessary information for classification.
- Dropout is applied to this representation.
- Finally, the classifier layer produces the output logits.

Training Process

The training process consists of two stages:

- Data Preprocessing
- Model Training

Data Preprocessing

- Data Loading and Feature Selection: The data is loaded from a CSV file, and relevant features are selected for analysis.
- Scaling and Normalization: The features are scaled using the StandardScaler to ensure all features are on the same scale, preventing feature dominance and improving model convergence.

- Train-Test Split: The preprocessed data is split into training and testing sets (80% for training and 20% for testing) using stratified sampling to maintain class balance.
- Class Weighting: Class weights are calculated to address class imbalance issues, ensuring the model focuses on the minority class during training.
- Data Conversion to Tensors: The data is converted to PyTorch tensors, enabling efficient computation.
- Padding and Sequencing: The PrepareDataset class prepares the data for sequence-based models by padding shorter sequences to a uniform length (seq_length).
- Custom Collate Function: The custom_collate function merges individual data samples into batches, padding sequences to the same length and converting labels to tensors.
- DataLoader: The DataLoader class is used to load data in batches, utilizing the custom collate function to ensure proper padding and formatting.

The preprocessing pipeline ensured that the data was properly prepared for training transformer encoder with classification head model, addressing issues like feature scaling, class imbalance, and sequence padding.

Model Training

Hyperparameters

- Model Architecture: The TransformerEncoder model is used, with hyperparameters such as sequence length, batch size, model complexity (d_model), number of heads (nhead), number of layers (num_layers), and feedforward dimension (dim_feedforward).
- Optimizer: AdamW optimizer is used with a learning rate and weight decay.
- Scheduler: Cosine Annealing LR scheduler is used to adjust the learning rate during training.

Training Process

- Data Loading: Training and testing data are loaded using DataLoaders, with a custom collate function for padding and batching.
- Model Initialization: The model is initialized and moved to the correct device (CPU).
- Training Loop: The model is trained for a specified number of epochs, with each epoch consisting of:
  - Forward pass: Model predictions are made on the training data.
  - Loss calculation: Cross-entropy loss is calculated using class weights.
  - Backward pass: Gradients are computed and weights are updated using the optimizer.
  - Evaluation: Model performance is evaluated on the test data.

Evaluation

- Loss Calculation: Weighted cross-entropy loss is calculated on the test data.
- Accuracy Calculation: Accuracy is calculated by comparing predicted and actual labels.
- Early Stopping: Training is stopped early if no improvement in test loss is seen for a specified number of epochs (patience).

Key Concepts in the Training Process

- Weighted Loss: Class weights are used to address class imbalance issues.
- Masking: Padding masks are created to ignore padded elements during training and evaluation.
- Device Management: Data and models are moved to the correct device (CPU) for efficient computation.

The training and evaluation pipeline ensures that the model is properly trained and evaluated, with techniques like early stopping and weighted loss to address the appropriate challenges.

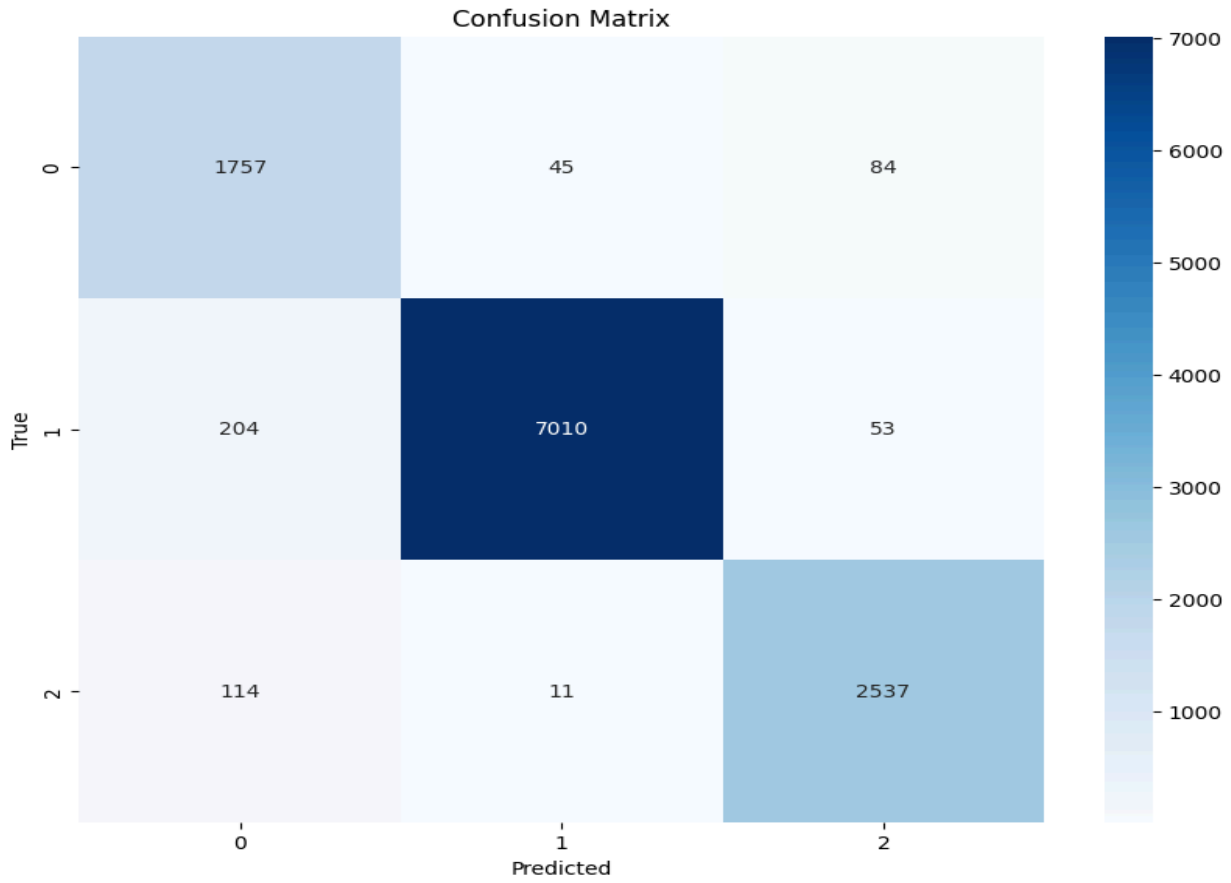
Results

The prediction pipeline uses the trained transformer encoder with a classification model to make predictions on the entire dataset. Below are the steps involved:

- Loaded the trained model from a saved state dictionary.
- Set the model to evaluation mode to disable gradient computation and dropout.
- Iterate over the dataset in batches, making predictions without computing gradients.
- Obtain the predicted class by taking the argmax of the output logits.
- Appended the predictions to the original dataframe under the *Prediction* column for trading blotter.

Metrics calculated for the classification model are:
1. Accuracy (95.6%)
2. Precision (95.91%)
3. Recall (95.67%)
4. F1-score (95.75%)

Confusion matrix



The analysis of the transformer encoder with the classification head model reveals impressive performance, achieving an accuracy of 0.9567. This indicates a strong overall classification ability, particularly for the positive class (class 1), with 7010 true positives, 257 false negatives, and 56 false positives.

While the model excels in identifying class 1, there is room for improvement in distinguishing between classes 0 and 2. As can be seen in the confusion matrix, class 0 had 1757 true negatives but 318 false positives and class 2 had 2537 true negatives with 225 false positives.

The transformer architecture's ability to capture complex dependencies in the data is likely a key factor in these results. The classification head effectively translates these learned representations into accurate class predictions.

Trading Blotter

The trading blotter simulates the trading environment, executing trades based on predictions (actions—HOLD, BUY, SELL) from the transformer model. The blotter calculated various performance metrics, providing insights into the strategy's effectiveness.

Metrics*:*

1. Total Return: Measures the overall return on investment.
2. Sharpe Ratio: Evaluates risk-adjusted performance, considering both return and volatility.
3. Max Drawdown: Calculates the maximum peak-to-trough decline, assessing potential losses.
4. Win Rate: Determines the proportion of profitable trades.
5. Avg Profit/Loss per Trade: Calculates average gains and losses per trade.
6. Profit Factor: Measures the ratio of gross profits to gross losses.
7. Total Portfolio Value: Tracks the overall value of the portfolio.

Trade Recommendation Generation*:*

*Trade Execution***:**

1. BUY: When the model predicts a BUY signal, the blotter executes a buy trade, purchasing shares at the current market price.
2. SELL: Conversely, when a SELL signal is predicted, the blotter executes a sell trade, selling shares at the current market price.

Trade Details:

- Timestamp: Records the exact time of each trade.
- Action: Indicates whether the trade is a BUY or SELL.
- Price: The execution price of the trade.
- Shares: The number of shares traded.
- Symbol: The stock symbol (AAPL in this case).
- Transaction Cost: The cost associated with executing the trade.
- Slippage: The difference between the expected and actual execution price.

Process:

The blotter iterates through the dataset, generating trade recommendations based on the predictive model's output. Each trade is executed, and relevant details are recorded. This process allows for the evaluation of the model's performance and the refinement of the trading strategy.

Example of Generated Recommendation:

Step: 4, Timestamp: 2023–07–03 08:07:20.349525673, Action: BUY, Price: 190.33971812460842, Shares: 147.7617538565381, Symbol: AAPL, Transaction Cost: 0.02537527050553077, Slippage: 3.770281875391589

Step: 5, Timestamp: 2023–07–03 08:07:21.830530495, Action: SELL, Price: 193.44312559988717, Shares: 147.7617538565381, Symbol: AAPL, Transaction Cost: 0.08024365104036953, Slippage: 0.6268744001128255

Step: 170, Timestamp: 2023–07–03 08:15:34.807292509, Action: BUY, Price: 200.1665435797301, Shares: 95.10684475956933, Symbol: AAPL, Transaction Cost: 0.5075054101106153, Slippage: 6.1165435797300916

……till the end of the trades (end of the data)

Final Results and Comparison

Since the simple blotter didn't provide metrics, I calculated similar metrics for it used for the classification model. The results are shown below

Model Metrics

Metric	Value
Total Return	-0.0051
Sharpe Ratio	-0.0474
Max Drawdown	0.0114
Number of Trades	55786.0000
Win Rate	0.0040
Avg Profit per Trade	20240.6386
Avg Loss per Trade	260.5343
Profit Factor	0.3108
Total Portfolio Value	9949360.8643

Simple Blotter Metrics

Metric	Value
Total Return	-0.0004
Sharpe Ratio	-0.0964
Max Drawdown	0.0175
Number of Trades	33142.0000
Win Rate	0.4685
Avg Profit per Trade	3290.8747
Avg Loss per Trade	3471.0963
Profit Factor	0.8372
Total Portfolio Value	9995824.1465

Key Differences

1. Total Return: The simple blotter outperforms the classification model, with a less negative return (-0.0004 vs. -0.0051).
2. Sharpe Ratio: Both models show negative risk-adjusted returns, but the simple blotter's ratio is more negative (-0.0964 vs. -0.0474).
3. Max Drawdown: The classification model experiences a lower maximum drawdown (0.0114 vs. 0.0175).
4. Number of Trades: The classification model executes significantly more trades (55,786 vs. 33,142).
5. Win Rate: The simple blotter has a substantially higher win rate (0.4685 vs. 0.0040).
6. Avg Profit/Loss per Trade: The classification model shows higher average profits (20,240.64 vs. 3,290.87) but lower average losses (260.53 vs. 3,471.10).
7. Profit Factor: The simple blotter has a higher profit factor (0.8372 vs. 0.3108).

Conclusion

Based on the results, the simple blotter approach is better suited for the trading strategy, as it achieves a less negative total return and higher profit factor while maintaining a higher total portfolio value. The classification model's performance is hindered by its high number of trades and low win rate, resulting in negative returns.

Challenges and Improvements

The classification model's negative return can be attributed to its excessive trading activity, caused by a bias towards the BUY label. This imbalance, where the model predicted more BUY than other classes, led to a high volume of trades.

The primary challenge in developing this transformer encoder with a classification head was addressing the class imbalance, which favored the BUY label. While class weights and stratified

sampling were employed to mitigate this issue, exploring additional techniques could further improve the model's performance, leading to a more profitable trading strategy with fewer trades.

Some ways to improve the model with techniques to handle class imbalance are:

- **Oversampling Minority Classes**: While I worked on oversampling minority classes with SMOTE and RandomOverSampler strategies, these methods did not yield the desired results. However, there are other strategies to oversample minority classes that can be utilized.
- **Changing the Loss Function**
- **Adjusting Decision Threshold**: Decision thresholds can be adjusted post-training to favor the minority classes.
- **Ensemble Methods**: Training multiple models with different architectures and combining their results.

Project Structure

GitHub: https://github.com/NvkAnirudh/BH_THAssignment

The project repo is organized into several folders and files.

Relevant (Important) Folders:

- Data
    - data.csv: Original data
    - df_with_predictions.csv: Original data with model predictions
    - trades_blotter.csv: Trades
- Solution_notebooks
    - TEncoder_solution.ipynb: Contains the implementation of the transformer encoder with a classification head architecture.
    - trading_blotter.ipynb: Contains the implementation of trading environment
- Models
    - best_model_checkpoint.pth: Weights of the model stored after early stopping
    - transformer_agent.pth: Weights of the model after full training (20 epochs)

Not so Important Folders:

- Research
    - research.ipynb: Contains feature engineering, implementation of PCA and t-SNE (along with visualizations) for dimensionality reduction and to check if the data had any clusters for label generation.

    - TST.py: Contains the implementation of Time Series Transformer (used mainly for time series forecasting). This architecture was abandoned due to its poor performance even after a lot of fine-tuning.
- Simple_blotters
    - Given implementation of trading blotter with PPO
    - simple_blotter.ipynb: Updated version of the given notebook with calculated metrics

Files:

- requirements.txt: All the used packages in the project
- BH Report.pdf: Documentation of the project