Blockhouse

ML Engineer

Work Trial Report

GitHub Link:

https://github.com/Vimarsh07/Blockhouse-Work-Trial-Task.git

Setup and Environment:

I have used Python, TensorFlow, Numpy and Pandas for my implementation.
Visual Studio Code as my IDE

Model Implementation:

I went through the sample code given to use in the Python file. I tried running the given code and model but I was getting issues while importing the PPO library. Some version error was stopping me from implementing PPO and RL and after lots of installation of different version of gymnasium, stable_baselines3 I decided to go with a pure transformer model.

Also I was not able to figure out stable_baselines3 for TensorFlow. I think it is only compatible with PyTorch.

Thus, my final model is a pure Transformer Model with the Trading Environment.

Since I was not able to use PPO, I had to define my own conditions for the Transformer on the basis of which it would decide Buy, Sell or Hold recommendations.

We were given the code to calculate Technical Indicators. I used that code with the dataset to calculate those indicators and gave some conditions on the basis of which I created a new column 'recommend' and created sequences.

Creating sequences for training models, especially in the context of time series data or sequences like financial market data, is crucial for several reasons:

Temporal Dependency: Time series data inherently has temporal dependencies where past events influence future events. By creating sequences, you provide the model with a structured way to understand these dependencies. For instance, past stock prices might influence future prices, and modelling this requires the model to see a sequence of prices rather than isolated instances.

Contextual Information: Sequences provide contextual information that helps the model make more accurate predictions. In financial markets, trends and patterns over time, such as moving averages or momentum indicators, can significantly influence trading decisions. By training the model on sequences, it learns to recognize and utilize these patterns.

Transformer Model:

Components of the Transformer Model:

Input Layer:

Shape: (60, 17) where 60 represents the sequence length or time steps, and 17 represents the number of features per step. This structure allows the model to consider a history of 60 time points, each with 17 distinct measurements or indicators (17 feature columns)

Transformer Encoder Layer:

- Multi-Head Attention: This component helps the model to focus on different positions of the input sequence simultaneously. It is crucial for capturing the nuanced dependencies between different time steps and features. In financial data, this might mean understanding how past prices influence future prices or how various indicators like volume and RSI interact.
- Normalization and Dropout: These are used to stabilize and regularize the learning, helping to prevent overfitting, which is common in complex models trained on noisy financial data.
- Feed-Forward Network: This portion of the layer processes the output of the attention mechanism, allowing for further abstraction and representation learning, which is critical for capturing complex patterns in the data.

Global Average Pooling 1D:

This layer averages over the sequence dimension, reducing it to a single vector that summarizes the entire sequence. This is useful for making predictions based on the entire input sequence rather than based on individual time steps.

Output Layer:

The model outputs a vector of size 3, corresponding to three classes (buy, hold, sell), using a SoftMax activation function. This setup is typical for classification tasks where each class represents a specific decision or prediction.

Model Compilation:

The model uses the Adam optimizer and sparse categorical cross entropy loss, which are standard choices for multi-class classification tasks.

The sequenced dataset was divided into train, validation and test datasets.

Evaluation:

After training the model, the test accuracy of the model is:

Test Loss: 0.27560245990753174

Test Accuracy: 0.8834065794944763

	precision	recall	f1-score	support
1.0	0.90	0.98	0.94	7796
2.0	0.56	0.21	0.31	1081
accuracy			0.88	8877
macro avg	0.73	0.59	0.62	8877
weighted avg	0.86	0.88	0.86	8877

Confusion Matrix:
[[7613 183]
[852 229]]

Fine-Tuning:

Early Stopping is used to stop training prematurely if a monitored metric (like validation loss) stops improving for a defined number of epochs (patience). This helps prevent overfitting, which occurs when a model learns the detail and noise in the training data to an extent that it negatively impacts the performance of the model on new data.

Parameters:

monitor: The metric to be monitored (e.g., val_loss). Here, val_loss is used, which is the loss on the validation dataset after each epoch.

patience: The number of epochs with no improvement after which training will be stopped. Here, it's set to 10, meaning the training will stop if there is no improvement in validation loss for 10 consecutive epochs.

LearningRateScheduler adjusts the learning rate during training according to a predefined function or schedule. This approach can help in converging to a better or more robust model by reducing the learning rate as training progresses, which can help in fine-tuning the model's weights, especially when approaching a minimum in the loss landscape.

The function defined here dynamically changes the learning rate based on the current epoch. For the first 10 epochs, it maintains the initial learning rate, promoting a faster convergence during the initial phase when the gradients are typically large.

After 10 epochs, the learning rate is exponentially decayed by multiplying it with tf.math.exp(-0.1). This reduces the learning rate, allowing the model to make smaller updates to the weights, which can be crucial for fine-tuning and settling into a minimum.

Trade Recommendations:

Finally the Transformer was used with the Trading Environment and trade recommendations were saved in a csv file.