Realized Volatility Prediction - Executive Summary (Aug/10/2025)

Equal contributors: Martin Molina-Fructuoso, Yuan Zhang

Motivation and Goal

Volatility in the stock market captures price fluctuations and is therefore an important quantitative indicator in financial markets. In this project, we aim to use order book and trade data to predict the Realized Volatility (RV) over the immediate next 10 mins. The universal loss function used throughout the project is the Root Mean Squared Percentage Error (RMSPE). This project is based on the Optiver Realized Volatility Prediction Kaggle competition (2021).

The Data

The raw data consists of trade and order book records, organized by `stock_id` (identifying the stock) and `time_id` (identifying the time bucket). Another common identifier is the `row_id`, formatted as "`stock_id`-`time_id`"; for example, `row_id` "0-5" refers to stock 0 at time bucket 5. The target variable is the future realized volatility (RV) over the next 10 minutes for each combination of `stock_id` and `time_id` in the dataset.

Some key indicators include (formulas provided by the kaggle competition host):

1. **WAP (weighted Average Price),** calculated using the best bid and best ask prices:

$$\frac{(\text{bidprice1})(\text{asksize1}) + (\text{askprice1})(\text{bidsize1})}{(\text{bidsize1}) + (\text{asksize1})}$$

2. The log return of a stock between time t and later time t':

$$r_{t,t'} = \log\left(\frac{P_{t'}}{P_t}\right)$$

Here, t and t' refer to actual time points, measured as "seconds in bucket" within a given `time_id` (not the `time_id` values themselves).

3. The current RV of a given `time_id`:

$$\text{RV} = \sqrt{\sum r_{t,t'}^2}$$

Calculated using consecutive time points t and t' within that `time_id`.

It should be noted that the pre-segregated test set is reserved by the competition host, and we do not have direct access to it. The test set can only be accessed by submitting a model to kaggle.

ETL Pipeline

You can run the py file `./ETLpipeline/Default_Extract_Transform.py` to download raw data from Kaggle, then extract and transform it into processed data for model training. This process includes, in order:

1. Downloading raw data from Kaggle.
2. Recovering `time_id` order (credit of this part goes to this Kaggle post and the top solution).
3. Creating realized volatility from raw order book data.
4. Creating processed trade data from raw trade data.
5. Creating time series data from raw order book data.

To see an example of loading the data for training, refer to `./ETLpipeline/Loading_Example.ipynb`. The main components include:

0.5. Creating the training set and validation set, with the latest 10 % of the `time_id` values used for validation.

1. Using the custom `RVdataset` PyTorch `Dataset` subclass to initialize a PyTorch dataset for the Pytorch `Dataloader`.

2. Loading the values for training by feeding the prepared PyTorch `Dataloaders` to the custom training loop `reg_training_loop_rmspe`.

Base line model

Our base line model is a linear regression that uses the current realized volatility (RV) as the predictor, with the future RV over the immediate next 10 minutes as the target. This model achieved training and validation losses of 0.3019 and 0.2678, respectively.

The training loop

Our custom training loop comes with a scheduler option that updates training rate according to validation loss plateau, and an early stopping option which reloads the best weight dictionary with the best validation loss.

In practice, we would apply the model with the best weight dictionary received from the training loop and fine tuning on the pre-segregated test set for a truly fair model comparison. However, we are experiencing issues with submitting the models to kaggle to access the test set (reserved by kaggle and not made available to us), so we are comparing models with their best validation loss for now. It should be noted that this gives a slight unfair advantage to the neural network models against the baseline model when comparing them: The neural network model is tuned with the scheduler and the early stopping decided by the validation loss, while the baseline model is fully segregated from the validation set.

Neural Network Models

It should be noted that we still could not fully submit the models for testing on kaggle (who reserves the test set), so all the "loss" provided in below are the "best validation loss".

Frozen convolution layer for “derivative of time series” feature creation

We use an untrainable (frozen) convolution layer to create "derivatives of time series" features. This layer applies a kernel `(-1,1)` to produce a derivative feature for a time series (appending 0 at the end). For example, the input `(1., 2., 3., 1., 2., 3., 1., 2., 3.)` yields derivatives `(1., 1., -2., 1., 1., -2., 1., 1., 0.)`. This process can be applied multiple times to produce n-th order derivative features.

Transformer building blocks

We developed custom attention-based transformer components to serve as building blocks for various models, including:

1. A custom encoder with self-attention.
2. A custom decoder with options for masking and/or shifting the ground-truth target input.

3. A positional embedding layer via cross-attention.

All the above are implemented to mimic the architecture described in the seminal paper *Attention is all you need*.

Time series base models

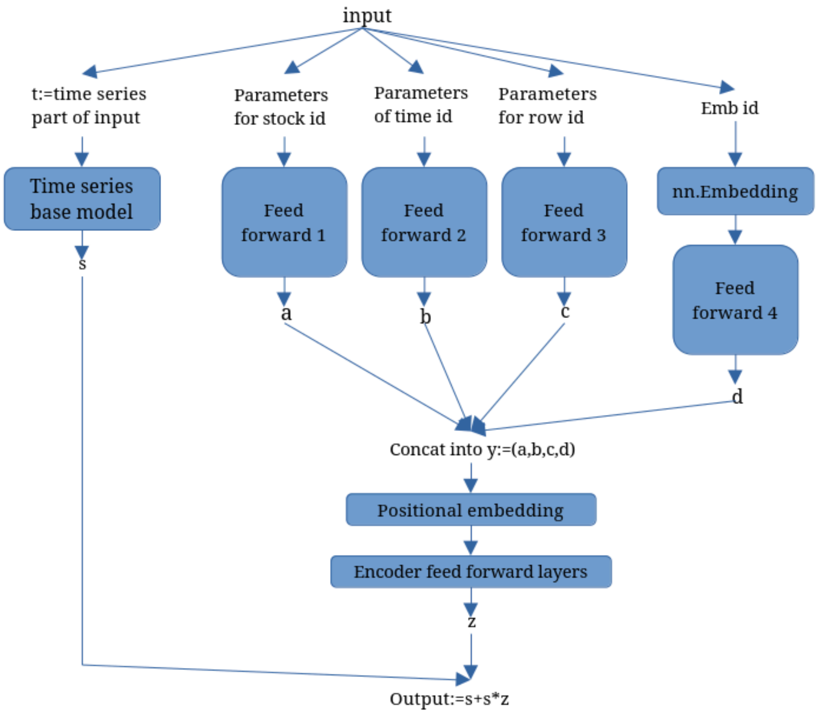
The following models take only time series as input, with the frozen convolution derivative feature layer always serving as the first layer:

1. **RNN-based models** (RNN, LSTM, GRU), achieving best losses of: RNN - 0.2348, LSTM - 0.2301, GRU - 0.2308.
2. **Encoder-only transformer model**, using the current time series as input, with a best loss of 0.2332.
3. **Encoder-decoder teacher forcing model**, using the current time series as input. Since we do not have a connecting time series as the target, we use the encoder memory in place of the ground-truth target (the "teacher"), with no shifting and the causal masking, achieving a best loss of 0.2296.

Adjustment models

These models adjust the output of time series-based models (referred to here as "base models") with values associated with categories including time, stock, and row id. The base models are trained as submodels within full models. In this context, `emb_id` (embedding ID) is created to "replace" the `stock_id`. The only difference is that `emb_id` is a consecutive list of integers with no gaps, whereas `stock_id` skips some integer values.

1. **Adjustment by pre-appending**. After passing `emb_id` through an `nn.Embedding` layer, it is pre-appended to the time series to form a new sequence, which is then fed into the base model. This approach achieved a best loss of 0.2233 with LSTM or GRU as the base model.
2. **Adjustment by multiplication**. After the `emb_id` passes through the `nn.Embedding` layer, it is processed by a feedforward layer to produce a scalar value, which is then multiplied with the output of the chosen base model. This approach achieved a best loss of 0.2228 with a GRU base model.
3. **Adjustment by cross attention**. After pasing `emb_id` through the `nn.Embedding` layer followed by a feedforward layer, the result is used as both the Key and Value in a cross-attention layer, with the base model output serving as the Query. The output of this cross-attention layer is then combined with the base model output to produce the adjusted result. This approach achieved a best loss of 0.2249 with a GRU base model.
4. **Adjustment with mixed identifiers**. This model uses any subset of `row_id`, `stock_id`, `time_id`, and `emb_id`. For details, please refer to the diagram on the left. This approach achieved a best loss of 0.2153 with a GRU base model.



Fine Tuning

We fine-tuned two encoder-decoder models, one with and one without a stock identifier embedding. The optimization strategy and heuristically defined search space, both critical to making Optuna work effectively, were developed using domain knowledge and targeted experimentation. We used the Optuna library to methodically explore the hyperparameter space, combined with custom early termination of underperforming trials. In addition, we designed reusable functions and a modular structure so that the same optimization pipeline can be applied to any custom model, ensuring that the optimization strategy is transparent, models are self-contained, and reproducibility of models for inference is guaranteed.

Future

As of August 10th of 2025, this project is 3 months old. Both contributors have honed their skills and understanding in data transformation and machine learning with PyTorch. They feel that they are just getting started, and there are many opportunities to further improve the models:

1. **Model Architectures**. Most models currently use feedforward layers composed mainly of custom made encoder and decoder blocks, or `nn.Linear` layers. We plan to investigate alternatives, including convolutional layers, particularly for higher-dimensional data.
2. **Regularization of adjustment models**. The adjustment model using all of `row_id`, `stock_id`, `time_id`, and `emb_id` is overfitting quickly (despite having very good validation loss). This suggests weak regularization and possibly noisy input parameters. We plan to test methods such as increasing dropout, adjusting optimizer weight decay(analogous to ridge/L2 regression), and reducing input parameters (similar to lasso/L1 regression).
3. **Teacher forcing adjustments**. The current teacher forcing model uses the encoder output as both encoder memory and the ground-trugh target (the “teacher”). We plan to experiment with splitting the input time series in half, using the first half as the “true input” and the second half as the “teacher”.
4. **Fine-tuning methods**. Explore additional fine-tuning strategies.
5. **Data processing improvements**. While pandas is an excellent tool, its performance can be a bottleneck with massive datasets, even with parallelization. We plan to evaluate tools better suited for big data processing, such as PySpark.
6. **Using pretrained base models**. Currently, all adjustment models train the time series base models as submodels. We plan to investigate using outputs from pretrained base models as parameters to adjust instead.