Finance 3 Final Paper: Forecasting the Volatility of a Stock Price Index

Vincent Jia
vjia@uwaterloo.ca

December 16, 2024

Contents

1 Introduction

1.1 Objective of Study

In *Forecasting the volatility of stock price index: A hybrid model integrating LSTM with multiple GARCH-type models* [5] by Kim and Won, the authors attempt to forecast volatility in the Korean stock market. Volatility plays a crucial role in the financial market, such as in derivative pricing, portfolio risk management, and hedging strategies. Therefore, if we can accurately predict volatility, we can gain an advantage in the market.

To predict volatility, Kim and Won employ a "hybrid model" which uses both the concepts of Long Short-Term Memory (LSTM) neural networks and Generalized Auto-Regressive Conditional Heteroscedasticity (GARCH) models. However, the LSTM model which Kim and Won are referring to is actually a LSTM model combined with a Deep Feed-forward Neural (DFN) network. Whenever they refer to LSTM, they actually are referring to "LSTM-DFN". The specific GARCH type models they are referring to in their "hybrid model" are the standard GARCH, Exponential GARCH (EGARCH), and Exponentially Weighted Moving Average (EWMA).

Using data from January 1, 2001 to January 2, 2017, Kim and Won test their models on the Korean Composite Stock Price Index (KOSPI) 200 index. The KOSPI 200 index is the index of the 200 largest companies in Korea by market capitalization. Out of the models they tested, they find the least mis-specified model out of the ones they tested is the GEW-LSTM model. GEW-LSTM stands for "Garch-Egarch-Ewma-LSTM" model. This model outperforms all the other models in the metrics of Mean Absolute Error (MAE), Mean Squared Error (MSE), Heteroscedasticity Adjusted MAE (HMAE), Heteroscedasticity Adjusted MSE (HMSE).

We will attempt to recreate some of the results of Kim and Won, specifically by creating the GEW-LSTM model, and testing it over new data to validate if it truly is effective. We will ultimately create and test four models. These are the DFN, GEW-DFN, LSTM, and GEW-LSTM and train them over minimizing MSE. The DFN and LSTM model will not contain parameters from the GARCH-type models, and thus are not "hybrid models". The GEW-DFN model was not tested by Kim and Won.

2 Review of Literature

2.1 History of Previous Volatility Models

The idea of applying Machine Learning and GARCH type models to forecast volatility is not a new one. Kim and Won have done an extensive literature review.

If we were to start at the beginning, we would start at Engle's 1982 ARCH model [3]. Engle proposed a model where the current volatility is a function of the "shocks" of past time periods.

After ARCH, Bollerslev created the GARCH model in 1986 [1]. Bollerslev's GARCH model is effectively an ARCH(∞) model. In the GARCH model, the current volatility is a function of past "shocks" and past volatility. The GARCH model captures "volatility clustering" in the market. "Volatility clustering" means that periods of high volatility are followed by periods of high volatility, and periods of low volatility are followed by periods of low volatility.

However, the GARCH model does not capture "leverage effect". This is why need the Nelson's 1991 EGARCH model [7]. The "leverage effect" in financial markets is the observation that sudden drops in stock market prices will cause more increases in volatility than sudden rises in stock market prices.

LSTM has also been applied to forecast financial patterns. One of the first papers cited by Kim and Won to apply LSTM to financial modeling is Maknickiene and Maknickas' 2012 paper [6] *Application of neural network for forecasting of exchange rates and forex trading.* In this paper, the authors appply LSTM to predict changes in exchange rates for foreign currencies.

Another paper which directly applies LSTM to predict financial trends, is the 2015 paper *A LSTM-based method for stock returns prediction: A case study of China stock market* [2] by Chen, Zhou and Dai. The authors use LSTM to predict returns in the Chinese stock market

Finally, Deep Neural Networks have been used to predict volatility before. In 2008, in the paper *Artificial neural network model of the hybrid EGARCH volatility of the Taiwan stock index option prices*[8], Tseng et al use a "hybrid model" of an integrated EGARCH model and a feed forward neural nwtork to estimate the volatility of Taiwan stock index option prices.

It's clear that Kim and Won are building on the work of prior authors.

3 Key Concepts to Understand Paper

3.1 GARCH Type Models

3.1.1 ARCH Model

The model ARCH(q) is commonly written as:

$$Y_t = \mu + \epsilon_t \tag{1}$$

$$\epsilon_t = \sigma_t \eta_t \quad \text{where } \eta_t \sim iid(0,1) \tag{2}$$

$$\sigma_t^2 = \omega + \sum_{i=1}^{q} \alpha_i \epsilon_{t-i}^2 \tag{3}$$

We see that $\eta_t \sim iid(0,1)$. At first, this definition may seem a bit vague. What $\eta_t \sim iid(0,1)$ means is that η_t is an iid random variable with predefined, fixed first and second moments. We can change the distribution of η_t to what suits us, whether that is a normal distribution, a student's t distribution, or a skewed student's t distribution. In one of the more common cases, $\eta_t \sim iid\, N(0,1)$ so $\epsilon_t \sim N(0, \sigma_t^2)$.

We can think of Y_t as a process with a long term mean μ and some changing "noise" terms ϵ_t. In our particular case, we can think of Y_t as the log returns of a stock market index. We can call ϵ_t a "shock" or "innovation" in the model. We can think of σ_t as the standard deviation of the "volatility". From now on, we will call σ_t^2 as the volatility. This volatility has a long term mean level ω, and depends on the "noise" of past squared shocks ϵ_{t-i}^2. If we are speaking loosely, we can also call ϵ_{t-i}^2 a "shock". Written more explicitly:

$$\sigma_t^2 = \omega + \alpha_1 \epsilon_{t-1}^2 + \alpha_2 \epsilon_{t-2}^2 + ... + \alpha_q \epsilon_{t-q}^2$$

At each time $t-i$, we call ϵ_{t-i}^2 the squared shock at time $t-i$. The time $t-i$ is also referred to as the "lag" at i periods back. So, the ARCH(q) model looks back q lags in the past to fit the volatility. Since σ_t^2 must be non-negative (in order for σ_t^2 to have a real world interpretation), we must have that $\omega > 0$, and each $\alpha_i \geq 0$.

3.1.2 GARCH Model

The GARCH(p,q) model can be written as:

$$Y_t = \mu + \epsilon_t \tag{4}$$

$$\epsilon_t = \sigma_t \eta_t \quad \text{where } \eta_t \sim iid(0,1) \tag{5}$$

$$\sigma_t^2 = \omega + \sum_{i=1}^{q} \alpha_i \epsilon_{t-i}^2 + \sum_{j=1}^{p} \beta_j \sigma_{t-j}^2 \tag{6}$$

We can see that the current volatility σ_t^2 depends not only on the past squared shocks ϵ_{t-i}^2, but also the past volatility values σ_{t-j}^2. Thus, the GARCH model has the nice property of "volatility clustering". This means that periods of high volatility will generally be followed by periods of high volatility. Periods of low volatility will be generally be followed by periods of low volatility. Volatility clustering is a good property to have in a volatility model because volatility clustering is a "stylized fact" (observed phenomenon) in the financial market.

3.1.3 EGARCH Model

The EGARCH Model can be written as:

$$\ln \sigma_t^2 = \omega + \sum_{i=1}^{p} \alpha_i \left(|e_{t-i}| - \sqrt{2/\pi} \right) + \sum_{j=1}^{o} \gamma_j e_{t-j} + \sum_{k=1}^{q} \beta_k \ln \sigma_{t-k}^2$$

This is how the EGARCH is written in the the "arch models" package in python. A more interpretable way to write the EGARCH is given by the rugarch package in R. The EGARCH(1,1) given by rugarch is:

$$Y_t = \mu + \epsilon_t \tag{7}$$

$$\epsilon_t = \sigma_t \eta_t \quad \text{where } \eta_t \sim iid(0,1) \tag{8}$$

$$\ln(\sigma_t^2) = \omega + \alpha \frac{\epsilon_{t-1}}{\sigma_{t-1}} + \gamma \left(\left| \frac{\epsilon_{t-1}}{\sigma_{t-1}} \right| - E\left[\left| \frac{\epsilon_{t-1}}{\sigma_{t-1}} \right| \right] \right) + \beta \ln(\sigma_{t-1}^2) \tag{9}$$

Notice that $\frac{\epsilon_{t-1}}{\sigma_{t-1}} = \eta_{t-1} = \eta_t$. We know that $\frac{\epsilon_{t-1}}{\sigma_{t-1}} = \eta_{t-1}$ because we are given that $\epsilon_t = \sigma_t \eta_t$. We know $\eta_{t-1} = \eta_t$ in distribution because η_t are iid random variables. If we want to rewrite the last line of the EGARCH model more explicitly, we have:

$$\ln(\sigma_t^2) = \omega + \alpha \eta_{t-1} + \gamma(|\eta_{t-1}| - E[|\eta_{t-1}|]) + \beta \ln(\sigma_{t-1}^2)$$

The key difference in the EGARCH model from the GARCH model are the γ term and the $(|\eta_{t-1}| - E[|\eta_{t-1}|])$ term. When $\gamma < 0$, negative shocks have a greater effect on increasing volatility σ_t^2 than positive shocks. When $\gamma > 0$, then positive shocks have a greater effect on increasing volatility σ_t^2 than negative shocks. Thus, γ captures the "leverage effect" which is a desirable property. The term $(|\eta_{t-1}| - E[|\eta_{t-1}|])$ measures how far the magnitude of a certain shock $|\eta_{t-1}|$ is away from the expected magnitude of all shocks $E[|\eta_{t-1}|]$.

3.1.4 EWMA Model

The EWMA model given by Kim and Won is $\sigma_t^2 = \rho \sigma_{t-1}^2 + (1-\rho)\epsilon_{t-1}^2$ where $0 < \rho < 1$. Basically, the EWMA predicts that current volatility σ_t^2 is a function of volatility 1 lag back σ_{t-1}^2 and the shock 1 lag back ϵ_{t-1}^2.

If we just look at this equation, we see the EWMA looks like a highly simplified GARCH model. The EWMA looks like a GARCH(1,1) with an unit root.

3.2 LSTM Models

The LSTM model is fairly complex. If I had to explain it simply, the LSTM model is a model which can retain memory far into the past.

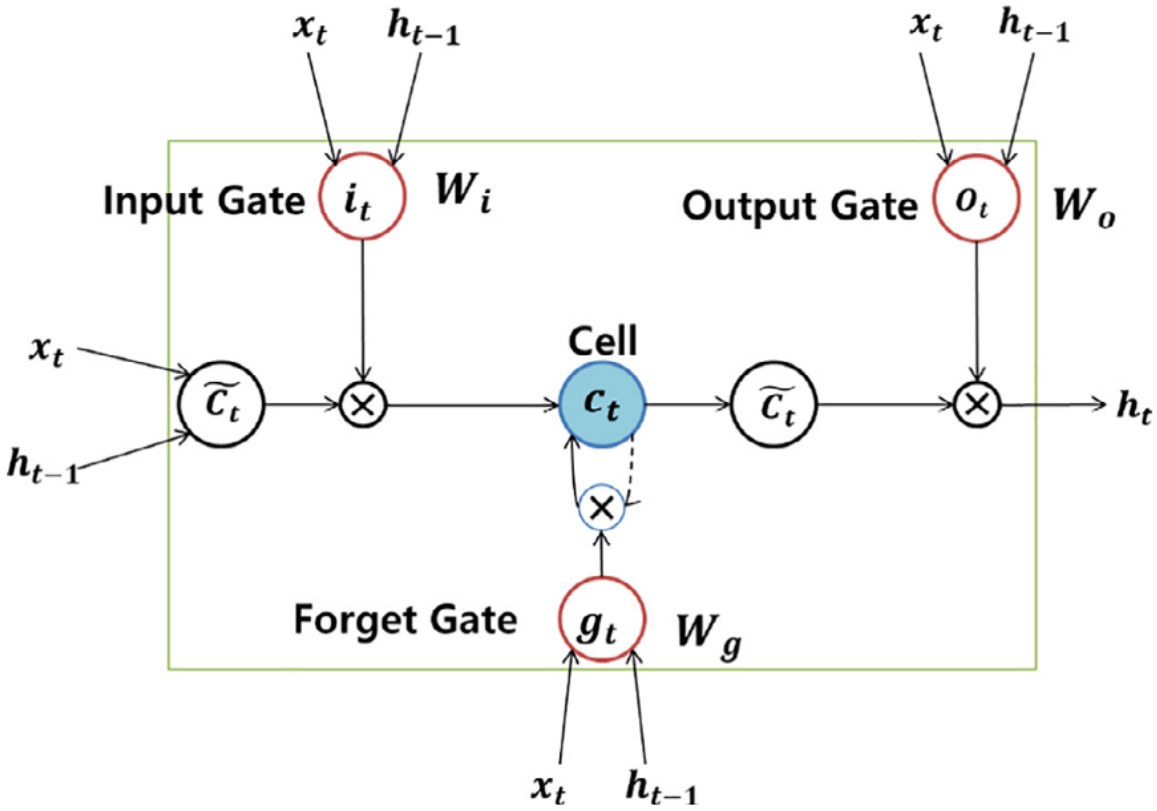


Fig. 5. An LSTM memory block.

One way we can write the LSTM model as:

$$r_t = \epsilon_t \tag{10}$$

$$\epsilon_t|\mathcal{F}_{t-1} \sim N(0, \sigma_t^2) \tag{11}$$

$$(\epsilon_t|\mathcal{F}_{t-1})N(0, \sigma_t^2) \Leftrightarrow \sigma_t\eta_t \text{ where } \sigma_t\eta_t = \epsilon_t \tag{12}$$

$$\text{Where:} \tag{13}$$

$$\sigma_t^2 = o_t C_t \tag{14}$$

$$C_t = f_t C_{t-1} + i_t(\omega + \alpha\epsilon_{t-1}^2 + \beta\sigma_{t-1}^2) \tag{15}$$

$$f_t = \sigma(w_f + \alpha_f\epsilon_{t-1}^2 + \beta_f\sigma_{t-1}^2) \quad \text{Where } \sigma \text{ is the activation function} \tag{16}$$

$$i_t = \sigma(w_i + \alpha_i\epsilon_{t-1}^2 + \beta_i\sigma_{t-1}^2) \quad \text{Where } \sigma \text{ is the activation function} \tag{17}$$

$$o_t = \sigma(w_o + \alpha_o\epsilon_{t-1}^2 + \beta_o\sigma_{t-1}^2) \quad \text{Where } \sigma \text{ is the activation function} \tag{18}$$

The LSTM model can capture the idea of the GARCH(p,q) model with infinite lags.

Let us explain the model in depth. f_t, i_t, and o_t are known as gates. f stands for "forget gate". We can think of f as determining the proportion of past information stored in C_{t-1} to retain.

i_t stands for "input gate". We can think of i_t as determining how much new information $(\omega + \alpha\epsilon_{t-1}^2 + \beta\sigma_{t-1}^2)$ should be stored in C_t.

o_t stands for output gate. We can think of o_t as controlling how much C_t is being used to determine σ_t^2

Both f_t and i_t are sigmoid functions and so their values are restricted to lie between 0 and 1. C_t is a weighted sum between it's previous period value C_{t-1} and the current information learned at time t. Thus, each C_t can retain some of the information from previous C_{t-i} terms for $i = 1, 2, 3....$ This is how the LSTM has "long memory".

We can compare the LSTM to the GARCH memory. The LSTM fixes an issue with the GARCH model because past information eventually decays away at a fixed rate under the GARCH model. Under the GARCH(1,1) model, the weights for ϵ_t^2 decay at the rate of β for every period, and is given by the equation $\sigma_t^2 = \sum_{i=0}^{\infty}\beta^i(\omega + \alpha\epsilon_{t-1-i}^2)$. If $\beta = 0.97$, then every period t, only 97% of information of ϵ_{t-1}^2 is retained.

3.3 DFN Models

The DFN model is designed to mimic the structure of neurons in the brain. The DFN has "nodes" which are connected to each other, and when information goes from one node to another, we apply a weight, a bias, and an activation function to that information.

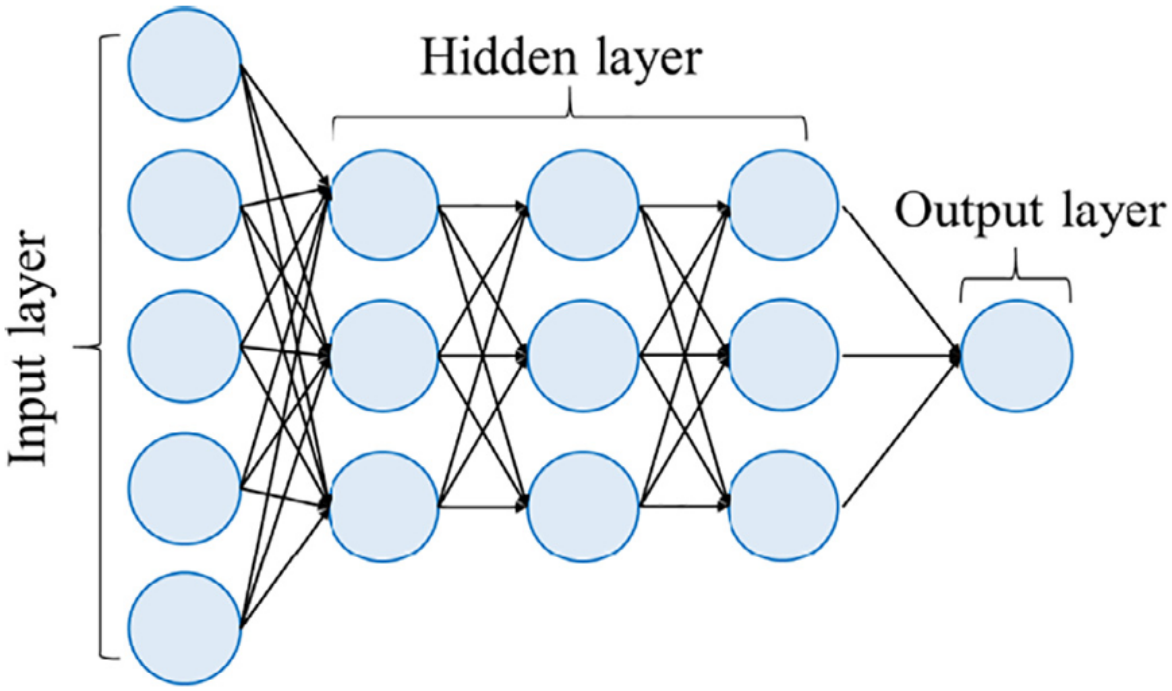


Fig. 4. A deep feedforward network.

An example of how information changes going from the input layer to the output layer is given by:

$$\hat{v}_t = f\Big(f\Big(f\Big(x_i w_{ji} + b_j\Big)w_{mj} + b_m\Big)w_{km} + b_k\Big)$$

The main idea is that we start with information x_i which is passed to one of the nodes in the input layer. We multiply x_i by some weight w_{ji} and add a bias b_j to it. Then we input $x_i w_{ji} + b_j$ into some "activation function" $f()$. In our particular case in the paper, $f()$ would be the sigmoid function.

We pass the information through the nodes until we reach the output layer. We call $\hat{v}_t$ the predicted value which is created by putting information into the input layer. Ultimately, we want $\hat{v}_t$ to get closer and closer to v_t which is the true known value we want to predict.

Over time, the model can be trained, where a backprogation algorithm is used to find more optimal weights and biases to reduce a specific criteria (MSE, MAE, HMSE, HMAE) using gradient descent.

3.4 Realized Volatility

In the paper, we are given the equation of realized volatility. It is given as:

$$RV_t = \sqrt{\frac{1}{\rho_t}\sum_{t=1}^{\rho_t}(s_t - \bar{s_t})^2}$$

Let us explain the equation. I personally chose ρ_t to be a fixed size rolling window. I did this to be consistent with my algorithm which looks back a fixed set amount of time from the current period to find the realized volatility in the next period.

In the author's case, $\rho_t = 22$. In my implementation, $\rho_t = 7$ due to hardware constraints.

The process s_t is the log return rate of the KOSPI 200 index at time t. $\bar{s_t}$ is the average of the log return rates of the KOSPI 200 index over the rolling window.

3.5 Metrics to Measure Error

These are loss functions that we can use to train our machine learning model.

3.5.1 MAE

This can be written as

$$\frac{1}{T}\sum|\hat{v}_t - RV_t|$$

where T is the length of the testing set.

3.5.2 MSE

This can be written as

$$\frac{1}{T}\sum(\hat{v}_t - RV_t)^2$$

where T is the length of the testing set.

3.5.3 HMAE

This can be written as

$$\frac{1}{T}\sum\left|1 - \frac{\hat{v}_t}{RV_t}\right|$$

where T is the length of the testing set.

3.5.4 HMAE

This can be written as

$$\frac{1}{T}\sum\left(1 - \frac{\hat{v}_t}{RV_t}\right)^2$$

where T is the length of the testing set.

4 Brief Overview of Method

We can finally discuss the overview of the method used to implement the LSTM models. We will use the data set and window size used by the authors'. Let's say that we have data from January 1st, 2001 to January 2nd, 2017.

The data will consist of 6 data sets. These are the stock price of the KOSPI 200 Index, the log return of the KOSPI 200 Index, the interest rate of 3-year Korea Treasury Bond (KTB), the interest rate on AA grade corporate bonds, and the price of crude oil and gold.

We will split our data into a training set and a testing set. The training set will be approximately 66% of the total data, while the testing set will be approximately 33% of the data. In the Kim and Won's case, the training data is 2665 data points from January 1, 2001 to September 30, 2011. The testing data is 1298 data from September 31, 2011 to January 2, 2017.

Over the training set, we will fit a GARCH, EGARCH, and EWMA model. When we fit the GARCH model, the specific values which we want to extract are the ω, α and β parameters. When we fit the EGARCH model, we want to extract ω, α, β and γ. The specific EGARCH model Kim and Won use is:

$$r_t = X_t \mathrm{M} + \varepsilon_t$$

$$\ln \sigma_t^2 = \alpha_0' + \beta \ln \sigma_{t-1}^2 + \omega\left(\frac{\varepsilon_{t-1}}{\sigma_{t-1}}\right) + \gamma\left|\frac{\varepsilon_{t-1}}{\sigma_{t-1}}\right|$$

Here, X_t represents an explanatory variable, and M is the parameter for that specific explanatory variable. For the EWMA model, Kim and Won actually don't fit the model themselves. They actually just set $\rho = 0.97$ because that is a "good" value from past papers. So we have found ρ and $1 - \rho$

Below is the photo of Kim and Won's values.

Table 2
Estimated parameters of GARCH variant models.

Models	Trend	ε_{t-1}^2	σ_{t-1}^2	$(\frac{\varepsilon_{t-1}}{\sigma_{t-1}})$	$\lvert\frac{\varepsilon_{t-1}}{\sigma_{t-1}}\rvert$
GARCH	0.000000251	0.067415	0.926795		
EGARCH	− 0.234291		0.987650	− 0.062557	0.138948
EWMA		0.03	0.97		

These values don't change over time. From the table, we can extract 9 different parameters.

Actually, **as of 12/16/24**, I realized that Kim and Won only extracted 7 values from the table. They don't extract ω (called "trend" on the table) for the GARCH and EGARCH models. However, this doesn't change the main idea of the paper. Also, the number of DFN nodes on the LSTM model is 5 and 1 nodes, not 10, 5 and 1 nodes. However, adding more nodes should not significantly hurt our results. The plain DFN model is still 10, 5 and 1 nodes.

We can now add these constant GARCH parameters to the training and testing data sets. If we are following Kim and Won's method, we should now have 13 features (6 explanatory variables + 7 garch parameters) in our dataset. Using our knowledge of the whole stock path, we can take the log return of the stock path and then use the realized volatility equation to calculate realized volatility over the whole data set.

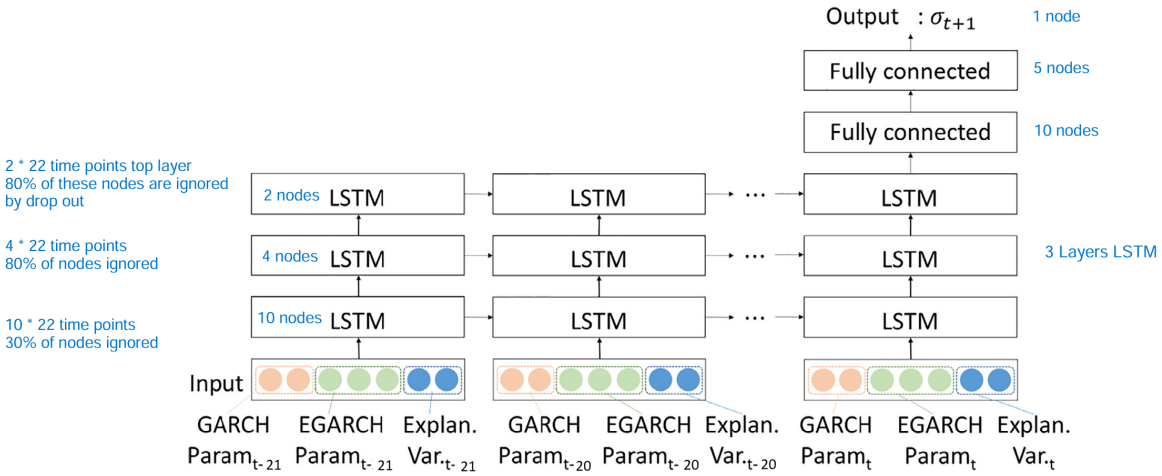


Fig. 6. Architecture of the proposed GE-LSTM hybrid model, created by integrating the GARCH, EGARCH, and LSTM models. *Note:* "Param" refers to parameters, and "Explan. Var." to explanatory variables.

Now, we can explain the full LSTM-DFN model. Basically, Kim and Won have taken a LSTM model and stacked a DFN model over it. In the specific diagram above, we have a GE-LSTM-DFN model. The G stands for Garch, and E stands for Egarch. The number of nodes drawn on the input layer is not to scale. In total, we have 2 Garch parameters, 3 Egarch parameters, and 6 explanatory variables. So in the GE-LSTM-DFN model we have 11 nodes in the input layer. So, our GE-LSTM will have 11 features.

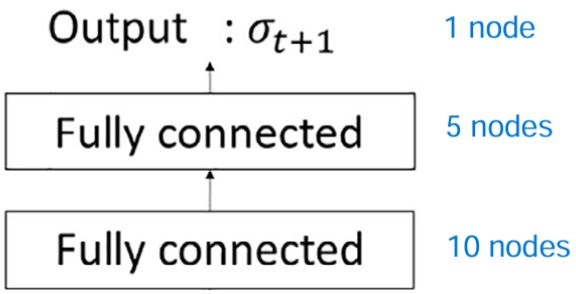
The whole LSTM model takes in data 22 time steps at a time. We need to all the data from time t to time $t-21$. In total, this is 22 days (3 weeks and a day). We will use the past 22 days to predict the volatility of the next day.

The LSTM part of the model has 3 layers. At the layer directly above the input layer, we have 10 nodes in each LSTM "box". At this point, we apply dropout to 30% of the nodes. The LSTM layer above that has 4 nodes in each LSTM "box". At this point, we apply an 80% drop out. The final LSTM layer on the top only has 2 nodes per "box". At this point, we also apply another 80% dropout. Though the dropout rate may see high, Kim and Won

claim these dropout values produce the best empirical results, and were found by trial and error.

On top of the LSTM is DFN. The DFN is fully connected (dense), and has 10 nodes. the next layer over that has 5 nodes. The final layer has 1 node. The activation over the DFN should be the sigmoid function, because that was suggested by the authors in the paper.

The authors also propose a plain DFN model. This model can be described as just the DFN on top of the LSTM-DFN model.



For both the LSTM-DFN and DFN models, we will use the Adam optimizer, set the learning rate to 0.0001, and the number of epochs to 150.

5 Authors' Implementation and Results

The authors come to some interesting results. For the purposes of clarity, we note that G stands for Garch, E stands for Egarch, and W stands for Ewma. The Garch parameters add 2 features, the Egarch parameters add 3 features, and the Ewma parameters add 2 features.

Table 3
Input variables of the models used in this study.

Input Variables	LSTM (DFN)	G-LSTM (DFN)	E-LSTM (DFN)	W-LSTM (DFN)	GE- LSTM	GW- LSTM	EW- LSTM	GEW- LSTM (DFN)
Explanatory variables	O	O	O	O	O	O	O	O
GARCH param.(2)	X	O	X	X	O	O	X	O
EGARCH param.(3)	X	X	O	X	O	X	O	O
EWMA param.(2)	X	X	X	O	X	O	O	O

The above table describes the naming convention for the machine learning models. O's represent inclusion, and X's exclusion.

Table 4
Results of out-of-sample forecasts with various loss functions.

		Model	MAE	MSE	HMAE	HMSE
Single model		EWMA	0.06559	0.00936	1.82616	3.87996
		GARCH	0.05925	0.00693	1.56660	2.87190
		EGARCH	0.05898	0.00641	1.11166	1.55857
		DFN	0.02083	0.00570	0.65977	0.78589
		LSTM	0.01498	0.00302	0.51864	0.44231
DFN-based hybrid model	with one GARCH-type	W-DFN	0.01943	0.00495	0.63632	0.77037
		G-DFN	0.01941	0.00429	0.60864	0.60309
		E-DFN	0.01701	0.00350	0.56996	0.45170
LSTM-based hybrid model	with one GARCH-type	W-LSTM	0.01322	0.00279	0.49125	0.32835
		G-LSTM	0.01271	0.00252	0.45258	0.29479
		E-LSTM	0.01191	0.00216	0.44651	0.28760
	with two GARCH-type	GW-LSTM	0.01156	0.00160	0.44188	0.28379
		EW-LSTM	0.01136	0.00158	0.43396	0.25292
		GE-LSTM	0.01089	0.00150	0.43279	0.23519
	with three GARCH-type	GEW-LSTM	0.01069	0.00149	0.42911	0.23492

The above table documents the quality of fit of the various machine learning models on the testing set. The general trend is that we see the more complex the model, the better the fit. The model with the best fit is the GEW-LSTM. Accross all metrics, MAE, MSE, HMAE and HMSE, it ourperforms all the other models.

We see that LSTM models with two Garch type model's parameters added to them outperform LSTM models with only one Garch type model's parameter added to them (E.g. GW-LSTM outperforms G-LSTM). We also see that the LSTM models with only one GARCH type outperform the DFN models with only one GARCH type (E.g. G-LSTM outperforms G-DFN). We also see that models without any GARCH types perform the worse.

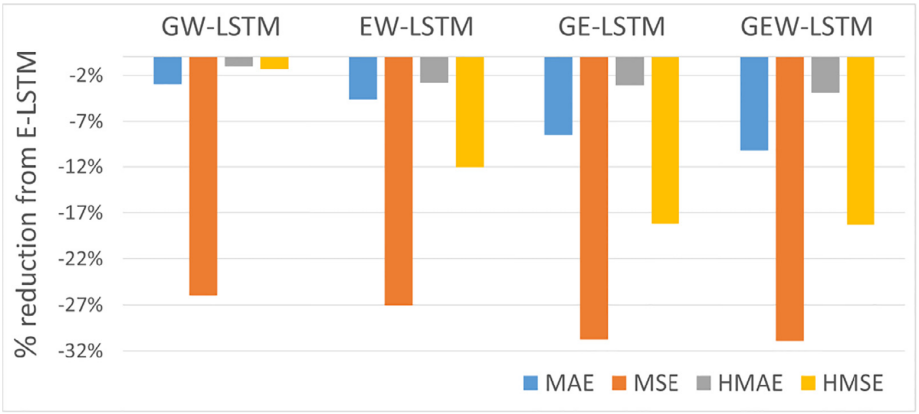
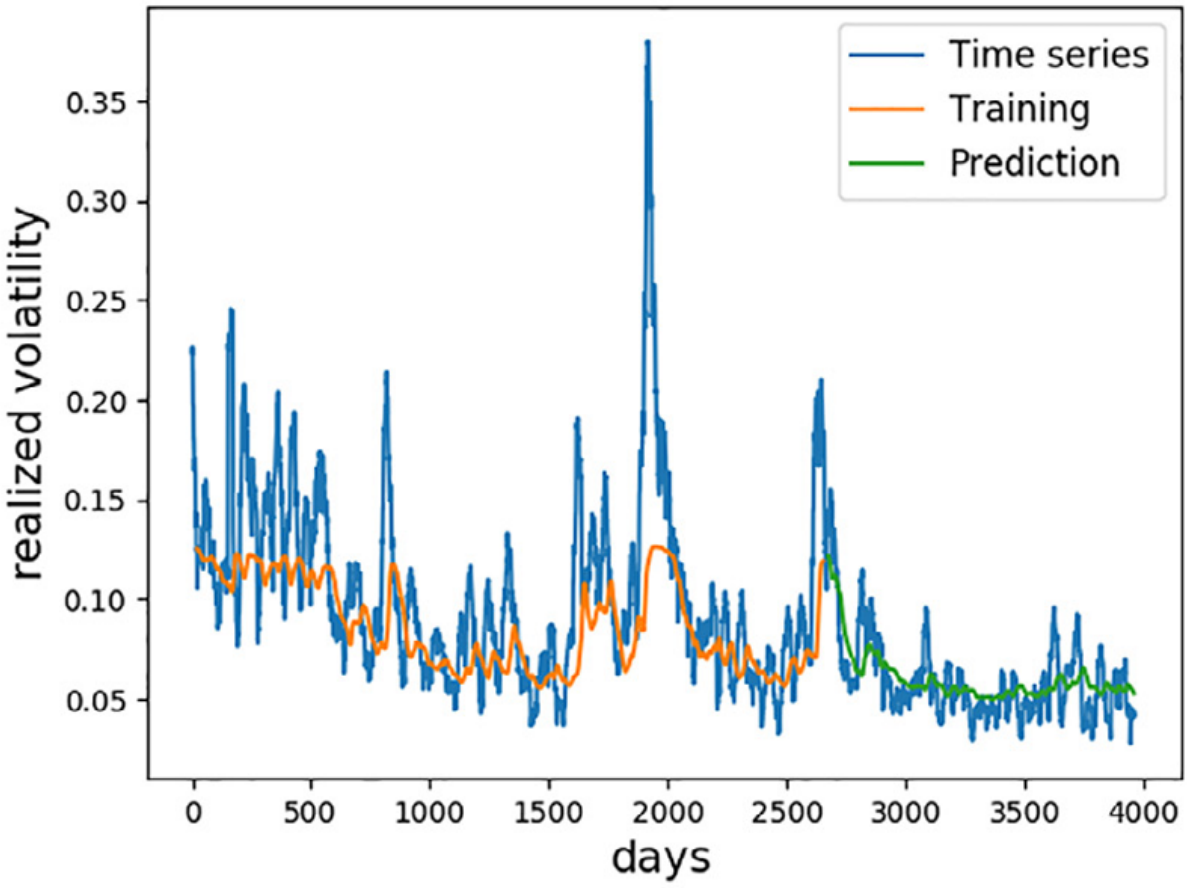


Fig. 7. Comparison of prediction errors of hybrid models combining the LSTM with two or three GARCH models with those of the best-performing LSTM-based single GARCH hybrid model (E-LSTM).

From the above diagram, Kim and Won provide more illustration that the more complex LSTM models outperform the simple LSTM models. The bars are comparing the reduction in error of the GW-LSTM, EW-LSTM, GE-LSTM and GEW-LSTM when compared to the E-LSTM.



Finally, we can see the exact prediction output of the GEW-LSTM model. It seems to follow the realized volatility quite well.

6 My Own Implementation and Results

6.1 Part 1 - Cleaning the Data

I will attempt to implement the algorithm of Kim and Won, but on a different data set. Instead of using data from January 1, 2001 to January 2, 2017, I will use data from Feb 1st, 2010 to Feb 1st 2024.

Using Bloomberg, I can grab the relevant data. Specifically, the data I downloaded was KOSPI 200 data, 3 year AA- corporate bond spread over government rates, 3 year Korean bond rate, the brent crude oil price, and the global commodity gold price. To get the log return of the KOSPI 200, I have to calculate that manually. I also have to manually add the corporate spread to the government interest rate to get the corporate bond interest rate.

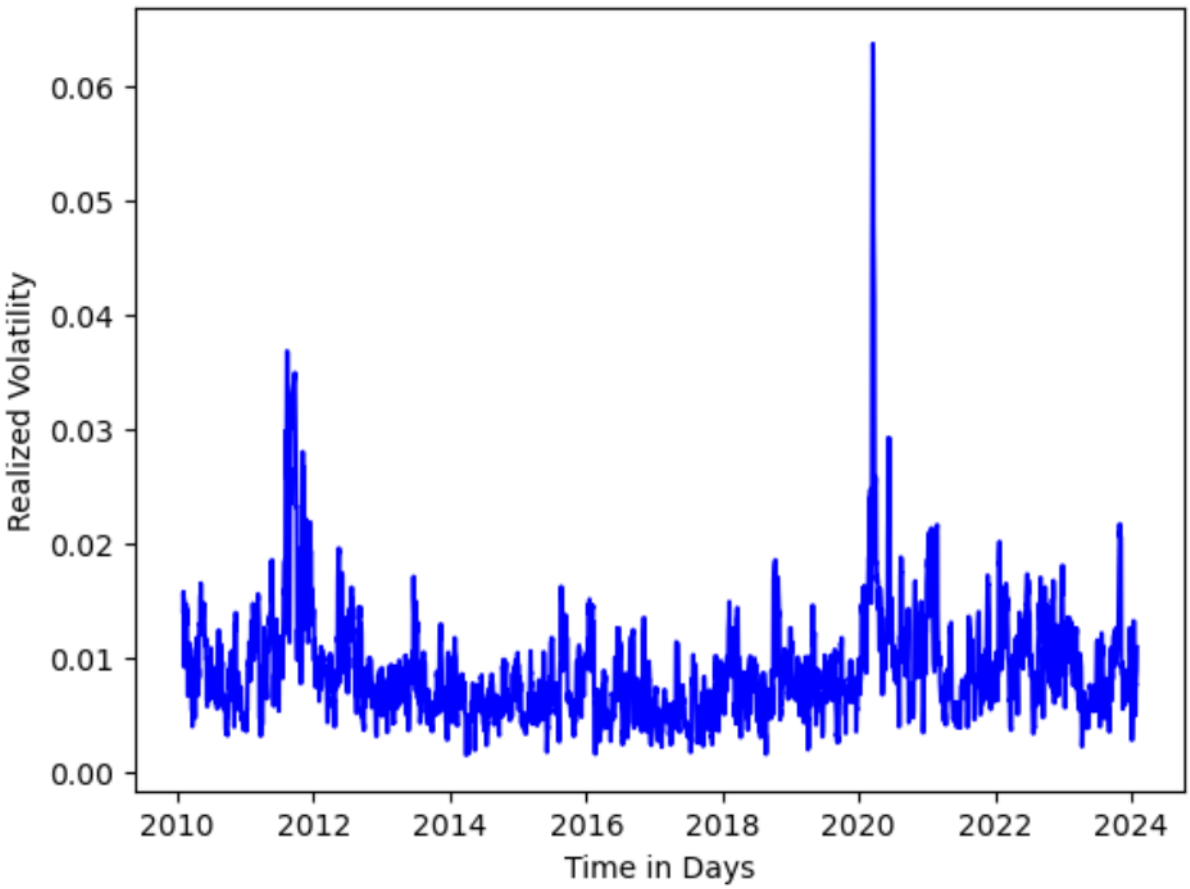
I will also calculate realized volatility using a rolling window of 7 days in the past.

At this point, our table should look like:

	date	price_kospi_raw	log_return_kospi	interest_rate_government	interest_rate_corporate	price_oil	price_gold	realized_volatility
0	2024-01-31	336.24	-0.003681	3.261	3.991	75.85	2039.52	0.010932
1	2024-01-30	337.48	-0.001066	3.267	4.007	77.82	2037.01	0.009749
2	2024-01-29	337.84	0.010803	3.304	4.044	76.78	2033.23	0.008876
3	2024-01-26	334.21	0.001228	3.262	4.002	78.01	2018.52	0.008171
4	2024-01-25	333.80	0.001589	3.314	4.054	77.36	2020.84	0.007573
...	...	...	...	...	...	...	...	...
3306	2010-02-08	203.37	-0.008276	4.230	5.260	71.89	1062.85	0.009932
3307	2010-02-05	205.06	-0.031962	4.225	5.265	71.19	1066.25	0.015218
3308	2010-02-04	211.72	-0.000661	4.280	5.320	73.14	1063.53	0.015026
3309	2010-02-03	211.86	0.012300	4.270	5.320	76.98	1109.80	0.015767
3310	2010-02-02	209.27	-0.007095	4.280	5.330	77.23	1114.45	0.014052

3304 rows × 8 columns

The graph of our realized volatility over the full data set looks like:



When looking at the graph, we see it is consistent with historical data. We can even see the spike in 2020 due to the Covid-19 pandemic.

Next, we need to split the data into the training data and testing data. I will split the data 66% training data, and 33% testing data.

The training set will be from 2010/02/02 to 2019/05/17. The testing set will be from 2019/05/20 to 2024/01/31. The reason why there is a gap from 2019/05/17 to 2019/05/20 is because the data only encompasses trading days.

We also need to calculate and fit the Garch, Egarch, and Ewma parameters over the training data. To do this, I will use the "arch" package in python. For each of our Garch type models, we will assume $p = 1, q = 1$ and $\eta_t \sim N(0, 1)$. The exact code is included in the appendix. My parameters are:

omega_garch	alpha_garch	beta_garch	omega_egarch	alpha_egarch	beta_egarch	rho_ewma	one_minus_rho_ewma
0.000002	0.05	0.93	-0.1462	0.1077	0.9839	0.985	0.015

In total, I will use 8 different Garch type parameters. We will append these parameters to both the testing and training set. We will save the testing and training set as an excel file to pass to the next section of code.

6.2 Part 2 - Format Correct Input for Machine Learning

The most difficult part of this project is actually formatting our data in a way that our machine learning algorithms can process it. We have to manually format our training and testing data. We will also save that data as a numpy array.

Kim and Won created 18 data sets. The breakdown is as follows. For each training / testing data set, we will need to create "input parameters (x)" and "output parameters (y)". In total, there are 8 ways to group the data sets.

1. Explanatory Variables
2. Explanatory Variables + Garch Params
3. Explanatory Variables + Egarch Params
4. Explanatory Variables + Ewma Params
5. Explanatory Variables + Garch Params + Egarch Params
6. Explanatory Variables + Garch Params + Ewma Params
7. Explanatory Variables + Egarch Params + Ewma Params
8. Explanatory Variables + Garch Params + Egarch Params + Ewma Params

For each of the 8 ways to group the data, we need to format the data so the training and testing data for the input variable x is in the right shape. So that is 8*2 = 16 models. The y output data will always be the same for the training and testing sets. So we only need 2 y data sets, "y_testing" and "y_training". In total, this is 16 + 2 = 18 datasets.

Creating all of the data sets would be incredibly tedious. For the sake of time, we will create data set 8 (Explanatory Variables + Garch Params + Egarch Params + Ewma Params), and data set 1 (Explanatory Variables) only. In total, we will create 6 datasets. We will call these data sets "data_8_training_x", "data_8_testing_x", "data_1_training_x", "data_1_testing_x", "training_y", and "testing_y".

For "data_8_training_x", we want to get our data into a numpy array with dimensions (2174, 7, 14). That is 2174 samples, 7 time steps and 14 features. We will also normalize the explanatory variables to have mean 0 and standard deviation 1, but not the Garch parameters. We will normalize all the x data the same way.

For "data_8_testing_x", we want to get our data into a numpy array with dimensions (1116, 7, 14). That is 1116 samples, 7 time steps and 14 features.

For "data_1_training_x" we want to get our data into a numpy array with dimensions (2174,7,6).

For "data_1_testing_x" we want to get our data into a numpy array with dimensions (1116,7,6).

For "training_y" we want to get our data into dimension (2174,1). For "testing_y", we want to get our data into dimension (1116,1). I will not normalize the data for y, because Kim and Won did not either.

6.3 Part 3 - Fitting the Machine Learning Model

We will design the machine learning models are specified by the paper. We will implement four different models. These are the GEW-LSTM, LSTM, GEW-DFN, and DFN models. The GEW-DFN model was not tested in the original paper. The GEW models will use the "data_8" training data, and the plain models will use the "data_1" training data. We will use the "Adam" optimizer, with training rate 0.0001, epochs equal to 150 and MSE loss function.

Below is the code.

GEW-LSTM (Explanatory Variables + GARCH + EGARCH + EWMA) (LSTM + DFN)

```
# GEW-LSTM
gew_lstm_model = Sequential()
gew_lstm_model.add(Input(shape=[7, 14])) # 7 time steps, 14 features
# 3 + 3 + 2 + 6 Garch / Egarch / EWMA / changing parameters
gew_lstm_model.add(LSTM(10, return_sequences=True))
gew_lstm_model.add(Dropout(0.3))
gew_lstm_model.add(LSTM(4, return_sequences=True))
gew_lstm_model.add(Dropout(0.8))
gew_lstm_model.add(LSTM(2))
gew_lstm_model.add(Dropout(0.8))
gew_lstm_model.add(Dense(10,activation='sigmoid')) # to be consistent with the paper
gew_lstm_model.add(Dense(5,activation='sigmoid'))
gew_lstm_model.add(Dense(1)) # no activation function (default is just linear)
```

LSTM (Explanatory Variables) (LSTM + DFN)

```
# Explanatory Variables-LSTM
lstm_model = Sequential()
lstm_model.add(Input(shape=[7, 6])) # 6 explanatory varaibles
lstm_model.add(LSTM(10, return_sequences=True))
lstm_model.add(Dropout(0.3))
lstm_model.add(LSTM(4, return_sequences=True))
lstm_model.add(Dropout(0.8))
lstm_model.add(LSTM(2))
lstm_model.add(Dropout(0.8))
lstm_model.add(Dense(10,activation='sigmoid')) # to be consistent with the paper
lstm_model.add(Dense(5,activation='sigmoid'))
lstm_model.add(Dense(1)) # no activation function (default is just linear)
```

GEW-DFN(Explanatory Variables + GARCH + EGARCH + EWMA)

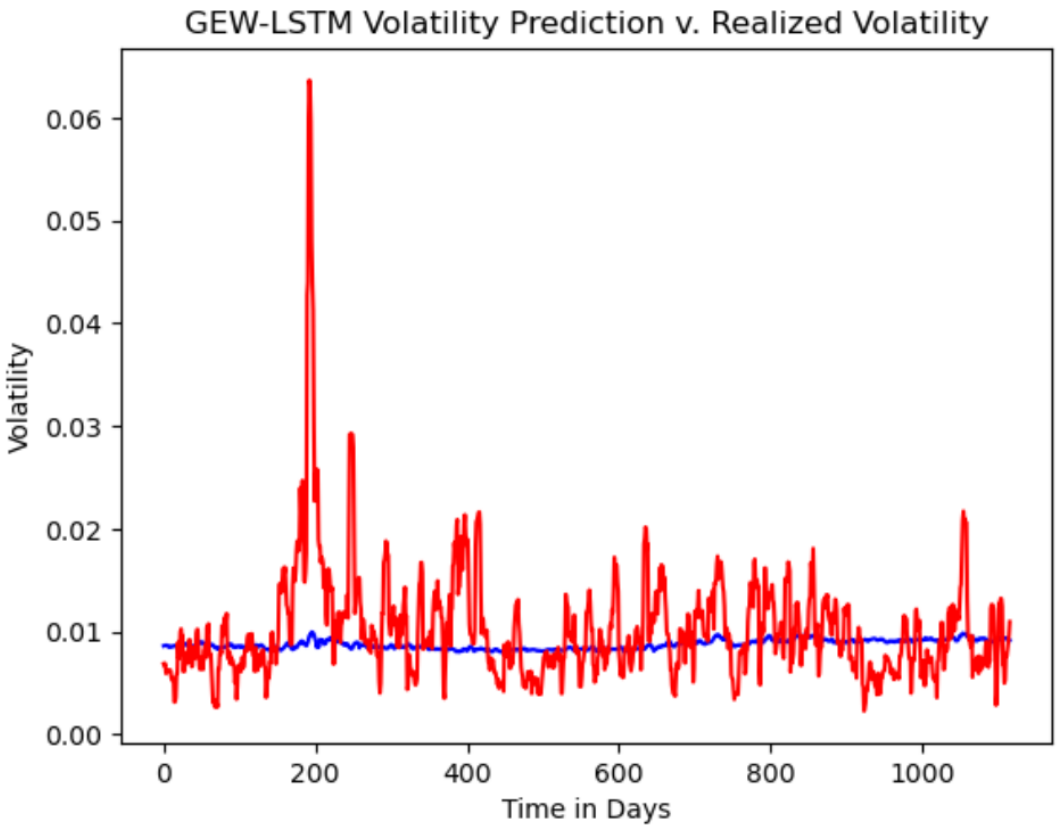
```
# Explanatory Variables-GEW-DNN
gew_dnn_model = Sequential()
gew_dnn_model.add(Input(shape = [7, 14])) # 7 past time steps / 14 explanatory variables
# 3 + 3 + 2 + 6 Garch / Egarch / EWMA / changing parameters
gew_dnn_model.add(Dense(10,activation='sigmoid'))
gew_dnn_model.add(Dense(5,activation='sigmoid'))
gew_dnn_model.add(GlobalAveragePooling1D())  # Flatten the time-step dimension
gew_dnn_model.add(Dense(1))
```

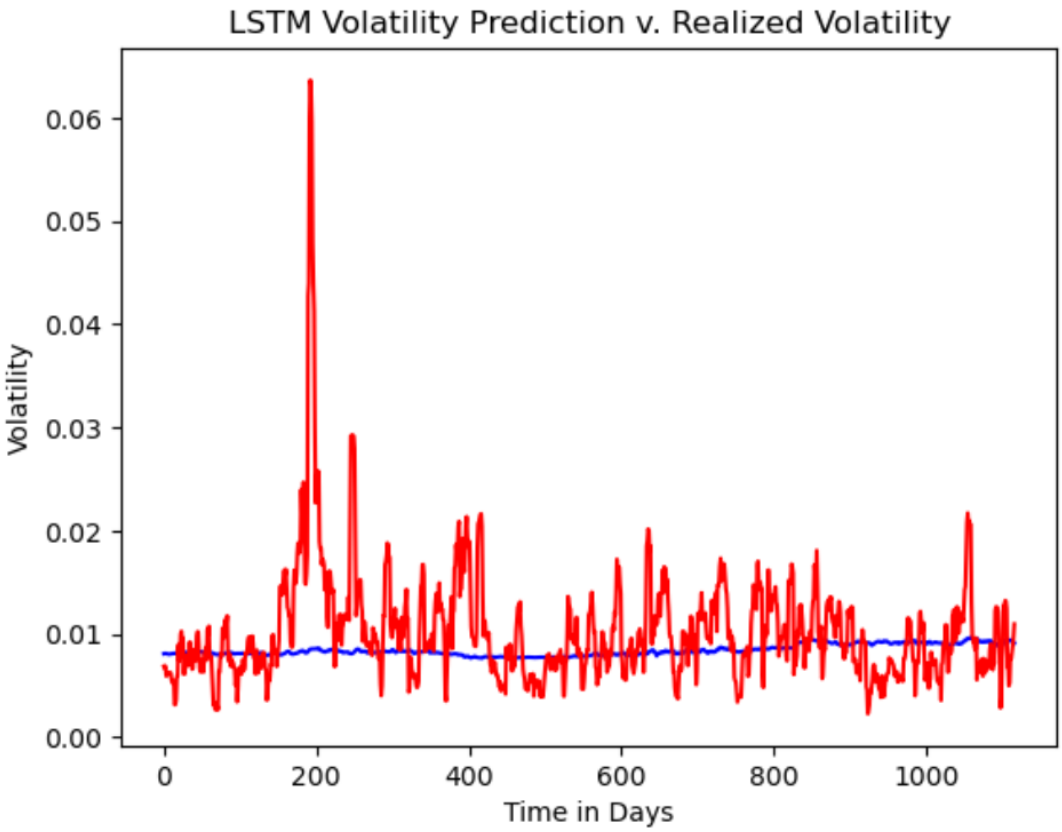
DFN (Explanatory Variables)

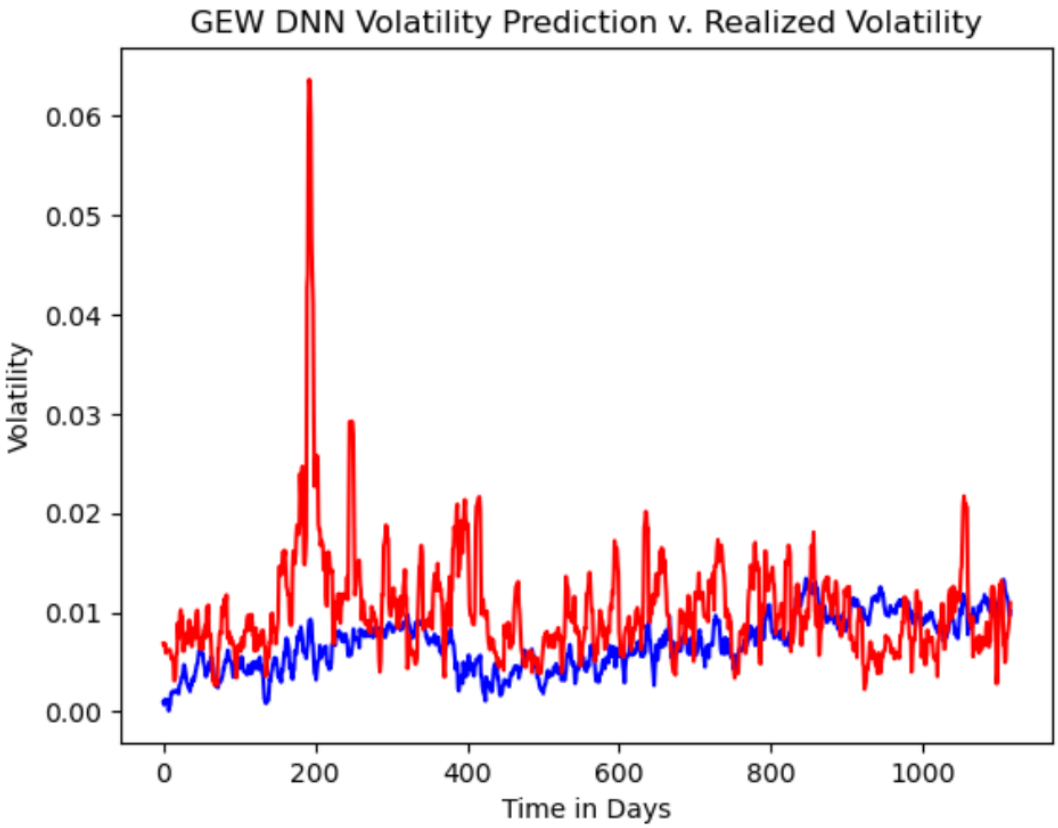
```
# Explanatory Variables-DNN
dnn_model = Sequential()
dnn_model.add(Input(shape = [7, 6])) # 7 past time steps / 6 explanatory variables
dnn_model.add(Dense(10,activation='sigmoid'))
dnn_model.add(Dense(5,activation='sigmoid'))
dnn_model.add(GlobalAveragePooling1D())  # Flatten the time-step dimension
dnn_model.add(Dense(1))
```

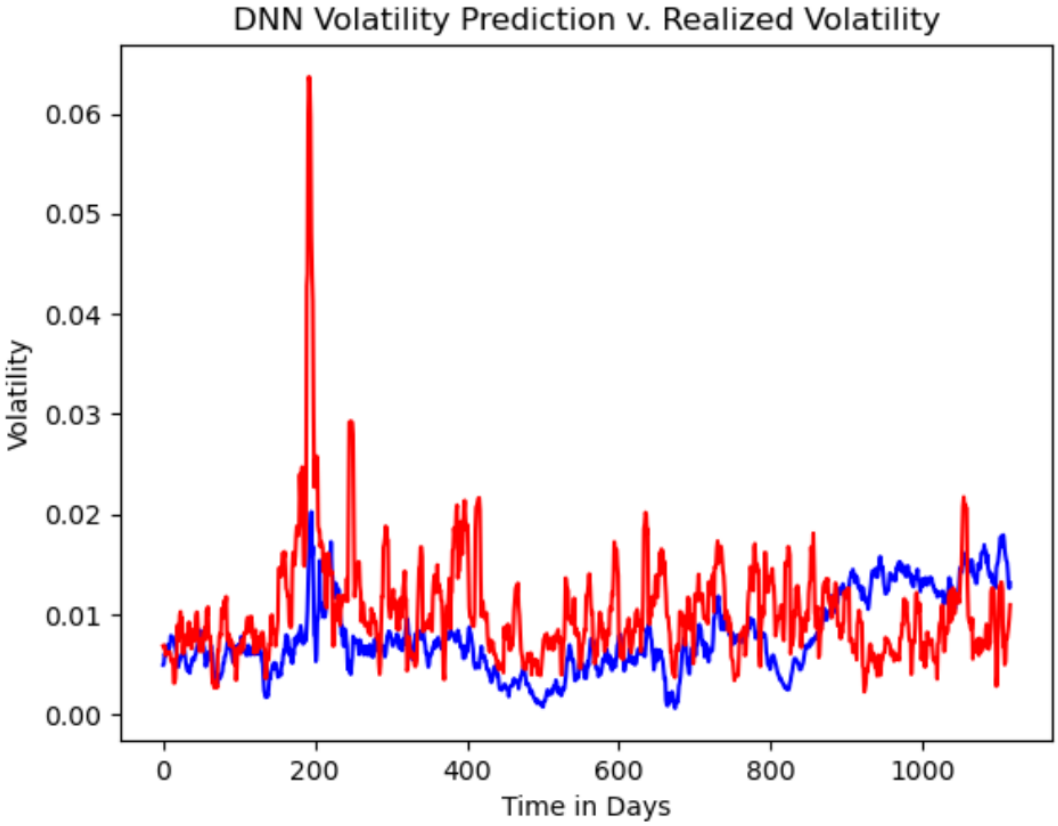
6.4 Part 4 - Predict Using Machine Learning

Here, we just use our trained machine learning models for GEW-LSTM, LSTM, GEW-DFN, and DFN to predict volatility. We can graph the results and find the MSE of the predicted values from the real values. The blue line is the forecasted realized volatility. The red line is the realized volatility.


GEW-LSTM Volatility Prediction v. Realized Volatility
Volatility
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0
200
400
600
800
1000
Time in Days


LSTM Volatility Prediction v. Realized Volatility
Volatility
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0
200
400
600
800
1000
Time in Days


GEW DNN Volatility Prediction v. Realized Volatility
Volatility
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0
200
400
600
800
1000
Time in Days


DNN Volatility Prediction v. Realized Volatility
Volatility
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0
200
400
600
800
1000
Time in Days

We can make some evaluations of the models from the forecasting. We see that the LSTM type models more or less forecasts a straight line. This result is surprising because the Kim and Won got a line that more closely mirrored the peaks in the realized volatility.

There are some factors which can explain this. One, this could be because I used a smaller rolling window of 7 days instead of 22 days. Another reason could because our training set does not look like our testing set. The testing set has more "shocks" in it while the training set has relatively few. The data set by Kim and Won had more "shocks" in the training set and less "shocks" in the testing set.

If we look at the GEW-DFN and DFN models, we can see that the blue line varies much more than in the LSTM model. This is a promising result. However, near the end of the forecasting period for both DFN type models, the forecasted line seems to develop an upward drift.

Finally, we can compare the MSE of all four models.

```
Mean Squared Error for DNN Model (6 parameters) is:  4.7453318846387845e-05
Mean Squared Error for GEW-DNN Model (14 parameters) is:  4.9151567465172275e-05
Mean Squared Error for LSTM Model (6 parameters) is:  3.661743213685297e-05
Mean Squared Error for GEW-LSTM Model (14 parameters) is:  3.461197825444811e-05
```

Our expectation is that the GEW-LSTM Model should have the lowest MSE for the same amount of epochs among all the models. We can see this is the case. This also means that GEW-LSTM out performs the plain LSTM, which is expected. More parameters on the same model should perform better. We also see that the LSTM model is superior to the DNN model which is also expected. However, we see that the MSE of the GEW-DNN model is actually worse than the MSE of the DNN model. This is not expected, but the MSE values for both models are fairly close. We also see that the GEW-LSTM model is only slightly better than the LSTM model in terms of MSE.

7 Critical Evaluation of Methods

The biggest issue with the GEW-LSTM model is justifying the reason why adding the Garch, Egarch, Ewma parameters improves volatility prediction. Empirically, we see that the GEW-LSTM outperforms the LSTM model in terms of MSE. In intuition is that we are doing something analogous to "supervised learning" by feeding in parameters of models which already model volatility.

Another issue with our method is that it most likely forecasts better on data sets with less "shocks". In our data set, we had a significant shock in 2020 which corresponded with Covid-19. Our LSTM type models forecasted a straight line over the the testing set, and while the LSTM models had lower MSE than the DFNs, the results would not be exactly helpful in a practical sense. If a financial institution were to rely solely on the GEW-LSTM model, the model could fail to predict a single period of high volatility and the financial institution would take a huge loss.

In my particular case, I had a hardware constraint. I couldn't set my window size to 22 because my laptop could not handle it. I'm also training my models using just my CPU. For a financial institution, this would not be too much of an issue. Kim and Won have shown in their paper that forecasted values fit better for larger window sizes.

Another problem with the GEW-LSTM method is the lack of "big data". Kim and Won only had around 4000 data points, and I had only around 3000 data points. Serious machine learning applications, like self-driving cars, can take in millions and millions of different pieces of information. For some data in Bloomberg, it is not possible to find financial data taken by the minute or by the hour for datasets measured in years.

Finally, one last hurdle that can face this model is that the model itself is completely opaque. It is a complete black box that neither I nor a regulator would fully understand. In the US, I don't see this model being approved for practical use.

However, the model has one great strength. Empirically, it significantly outperforms traditional econometrics techniques like fitting a GARCH or EGARCH model to a dataset. For Kim and Won's testing set, the GARCH model had a MAE of 0.05925 while the GEW-LSTM had a MAE of 0.01069.

8 Suggestions for Improvement

There are numerous ways we can improve upon the model suggested by Kim and Won. The most obvious way is to use a rolling window over the training set to fit the Garch, Egarch, Ewma parameters. In fact, there have been papers written using this exact technique.

We can also attempt to collect more granular data. Instead of using daily data, we can collect hourly data. For popular stocks in liquid stock exchanges, like the NYSE, this is possible. We can attempt to create a model using only assets whose price updates frequently.

We can also attempt to forecast fewer days forward in time. The issue with forecasting over long periods is that "shocks" can happen in the market. Prof. Tony has talked extensively about this issue. "Shocks" in the market could render a GARCH model completely useless in the long term. For example, no one could have definitively predicted Covid-19 or the Ukraine-Russia war. Instead of forecasting 1000 days ahead, we can attempt to only forecast 24 hours ahead. That way, we can attempt to avoid the "shocks" in the market.

Kim and Won have shown that increasing the size of your rolling window will result in a better fitted model. Perhaps if we attempted increasing our rolling window size from 22 to 252 we could produce better results.

There have also been papers where the authors attempt to create even more complex models. In *LSTM–GARCH Hybrid Model for the Prediction of Volatility in Cryptocurrency Portfolios* [4], Medina and Moreno use the past 72 periods (in hours) to forecast volatility for the next 24 periods (in hours)

We can also attempt to use different Garch type models. One particular model we can use instead of Egarch could be Aparch which is an even more flexible model which captures leverage effect. Other potential inputs could be the day of the week. In some markets, there is the "day of the week effect" where during the beginning of the trading week (Monday in most countries) and at the end of the trading week (Friday in most countries), there is increased trading. This could lead to an increase in volatility.

We can also attempt to explain what drives the volatility in the models. If possible, we can attempt to apply LIME or SHAP to the models.

Finally, we can attempt to collect more qualitative data rather than quantitative data. If the only inputs into our model are things that can be measured, we will miss crucial information. One thing we could track is "investor sentiment". If people feel pessimistic about the economy, maybe that could signal an incoming recession and high volatility.

9 Conclusion

In conclusion, we have thoroughly discussed the method by Kim and Won to implement volatility forecasting using "hybrid models" which combine the properties of LSTM models and Garch type models.

By replicating their approach on a new dataset spanning 2010 to 2024, we demonstrated that the GEW-LSTM model outperforms the LSTM, GEW-DFN, and DFN models when trained to minimize MSE. Thus, we conclude that the ideas of Kim and Won have merit.

We have also created our own GEW-DFN model. We see that it actually performs worse than the DFN model. This is an interesting fact to explore, because I'm curious why it would actually perform worse.

There are many practical challenges with using LSTM models for financial forecasting. These include possible hardware constraints, a lack of data to train the model, and the inherent opacity of neural networks. Despite this, we see that the GEW-LSTM significantly outperforms traditional econometrics techniques for volatility forecasting. Overall, we see that LSTM models show great promise for financial forecasting, and deserves more research.

10 All The Code

You can find the code at this link: Github Code For Final Finance 3 Project

References

[1] Tim Bollerslev. Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, 1986.

[2] Kai Chen, Yi Zhou, and Fangyan Dai. A lstm-based method for stock returns prediction: A case study of china stock market. *Proceedings of the 2015 IEEE International Conference on Big Data*, 2015.

[3] Robert F. Engle. Autoregressive conditional heteroscedasticity with estimates of the variance of united kingdom inflation. *Econometrica*, 1982.

[4] Andres Garcia-Medina and Aguayo-Moreno Ester. Lstm–garch hybrid model for the prediction of volatility in cryptocurrency portfolios. *Computational Economics*, 2023.

[5] H. Y. Kim and C. H. Won. Forecasting the volatility of stock price index: A hybrid model integrating lstm with multiple garch-type models. *Expert Systems with Applications*, 2018.

[6] N Maknickiene and A Maknickas. Application of neural network for forecasting of exchange rates and forex trading. *Proceeedings of the 7th International Scientific Conference on Business and Management*, 2012.

[7] Daniel B. Nelson. Conditional heteroskedasticity in asset returns: A new approach. *Econometrica*, 1991.

[8] Chih-Hsiung Tseng, Sheng-Tzong Cheng, Yi-Hsien Wang, and Peng Jin-Tang. Artificial neural network model of the hybrid egarch volatility of the taiwan stock index option prices. *Pysica A: Statistical Mechanics and its Applications*, 2008.

Part 1 - Clean Raw Data

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from arch import arch_model

df1 = pd.read_excel("./excel_data/kospi_200_index.xlsx") # korea stock exchange index
df2 = pd.read_excel("./excel_data/3_year_gov_bond_rates.xlsx") # 3 year gov bond interest rates
df3 = pd.read_excel("./excel_data/corporate_bond_spreads.xlsx") # excess return on korean corporate bonds over government bonds
df4 = pd.read_excel("./excel_data/crude_oil_prices.xlsx") # brent crude oil prices
df5 = pd.read_excel("./excel_data/gold_prices.xlsx") # gold prices

# convert all "date" columns to date-time object
df1['date'] = pd.to_datetime(df1['date'])
df2['date'] = pd.to_datetime(df2['date'])
df3['date'] = pd.to_datetime(df3['date'])
df4['date'] = pd.to_datetime(df4['date'])
df5['date'] = pd.to_datetime(df5['date'])

# reverse the dataframe
df1_reversed = df1.iloc[::-1].reset_index(drop=True)

# calculate log return
df1_reversed['log_return_kospi'] = np.log(df1_reversed['price'] / df1_reversed['price'].shift(1))

# drop the top value of the dataframe because it is "NaN"
df1_reversed = df1_reversed.drop(index=0).reset_index(drop=True)

# restore original order of the dataframe
df1 = df1_reversed.iloc[::-1].reset_index(drop=True)

# rename 'price' columns to be unique
df1 = df1.rename(columns={'price': 'price_kospi_raw'})
df2 = df2.rename(columns={'price': 'interest_rate_government'})
```

```
df3 = df3.rename(columns={'price':
    'interest_rate_corporate_spread'})
df4 = df4.rename(columns={'price': 'price_oil'})
df5 = df5.rename(columns={'price': 'price_gold'})

# merge df1, df2, df3, df4, df5
# since df3 has the fewest rows, we will merge on df3 first
merged_df1 = pd.merge(df3, df2, on = "date", how = "inner")
# add corporate spread to interest rate
merged_df1['interest_rate_corporate'] =
    merged_df1['interest_rate_corporate_spread'] +
    merged_df1['interest_rate_government']
# drop the old column "interest_rate_corporate_spread"
merged_df1.drop(columns='interest_rate_corporate_spread',
    inplace=True)

# merge everything else
merged_df2 = pd.merge(merged_df1, df1, on = "date", how = "inner")
merged_df3 = pd.merge(merged_df2, df4, on = "date", how = "inner")
merged_df4 = pd.merge(merged_df3, df5, on = "date", how = "inner")

# re-arrange column order for clarity
df6 = merged_df4[['date', 'price_kospi_raw', 'log_return_kospi',
    'interest_rate_government',
                  'interest_rate_corporate', 'price_oil',
                     'price_gold' ]]

# we need to flip the dataframe so oldest results are at the top
df6_reversed = df6.iloc[::-1].reset_index(drop=True)

# Define the maximum window size
# we're not following the author's window size because our computer
    isn't as good as their's
max_window_size = 7

# Initialize a list to store realized volatility values
realized_volatility = []

# Loop over each row in df6
# remember that i starts from 0
```

```
for i in range(len(df6_reversed)):
    # Get the subset of log returns starting from the current row
    # remaining returns is a subset of the dataframe df6_reversed
    remaining_returns = df6_reversed['log_return_kospi'].iloc[i:]

    # Limit the number of rows to the max window size
    if len(remaining_returns) > max_window_size:
        # here, we truncate our data so we only get to the 7th value
        # inclusive of the 7th value (because index starts at 0)
        remaining_returns = remaining_returns.iloc[:max_window_size]

    # Number of days in the current window (rho_t)
    rho_t = len(remaining_returns)

    if rho_t > 1:  # Ensure at least two data points to compute
       variance
        # Calculate the mean of the remaining returns
        mean_return = remaining_returns.mean()

        # Calculate realized volatility (standard deviation of
           mean-centered returns)
        rv_t = np.sqrt((1 / rho_t) * np.sum((remaining_returns -
           mean_return) ** 2))
    else:
        rv_t = np.nan  # Not enough data to compute volatility

    realized_volatility.append(rv_t)

# Use .loc to avoid SettingWithCopyWarning
df6_reversed.loc[:, 'realized_volatility'] = realized_volatility

# reverse the dataframe so newest results are on the top
df6 = df6_reversed.iloc[::-1].reset_index(drop=True)

# drop the top value because it is a "NaN"
df6 = df6.drop(index=0).reset_index(drop=True)
# drop all rows with "NaN"
df6 = df6.dropna()
df6

```

```
# now our data goes from 2010/02/02 to 2024/01/31

# lets graph the realized volatility
plt.plot(df6_reversed['date'], df6_reversed['realized_volatility'],
    color='blue')
plt.xlabel("Time in Days")
plt.ylabel("Realized Volatility")

num_rows = df6.shape[0]
print(num_rows)

split_index = int(0.34 * num_rows)
row_34 = df6.iloc[split_index]
print(row_34)

df_testing_data =  df6.iloc[:split_index] # data from 2019/05/20 to
    2024/01/31 (only trading days in dataset)
df_training_data = df6.iloc[split_index:] # data from 2010/02/02 to
    2019/05/17

# put oldest data on top
df_testing_data = df_testing_data.iloc[::-1].reset_index(drop=True)
df_training_data =
    df_training_data.iloc[::-1].reset_index(drop=True)

garch11 = arch_model(df_training_data['log_return_kospi'],
    vol='GARCH', p=1, q=1, rescale = False)
fitted_model = garch11.fit()
print(fitted_model.summary())
# omega = 1.9716e-06 = 0.0000019716
# alpha = 0.0500
# beta = 0.9300

# EGARCH Model
egarch11 = arch_model(df_training_data['log_return_kospi'],
    vol='EGARCH',  p=1, q=1, mean='Constant', dist='Normal',rescale
    = False)
fitted_modelv2 = egarch11.fit()
print(fitted_modelv2.summary())
# omega = -0.1462
```

```
# alpha = 0.1077
# beta = 0.9839

# Find optimal EWMA parameters
from statsmodels.tsa.holtwinters import SimpleExpSmoothing
ewma_model =
   SimpleExpSmoothing(df_training_data['log_return_kospi']).fit()
optimal_alpha = ewma_model.model.params['smoothing_level']
# we can think of alpha as 1 - rho
print("rho = ", 1 - optimal_alpha)
# thus we know the rho = lambda parameter as defined by the paper
# rho = 0.985 Round for convenience
# 1 - rho = 0.015

# add parameters into the dataframes
df_training_data = df_training_data.assign (
    omega_garch = 0.0000019716,
    alpha_garch = 0.0500,
    beta_garch = 0.9300,
    omega_egarch = -0.1462,
    alpha_egarch = 0.1077,
    beta_egarch = 0.9839,
    rho_ewma = 0.985,
    one_minus_rho_ewma = 0.015
)
df_training_data

df_testing_data = df_testing_data.assign (
    omega_garch = 0.0000019716,
    alpha_garch = 0.0500,
    beta_garch = 0.9300,
    omega_egarch = -0.1462,
    alpha_egarch = 0.1077,
    beta_egarch = 0.9839,
    rho_ewma = 0.985,
    one_minus_rho_ewma = 0.015
)
df_testing_data

df_testing_data.to_excel("./excel_data/testing_data.xlsx",
```

```
    index=False)
df_training_data.to_excel("./excel_data/training_data.xlsx",
    index=False)
```

Part 2 - Format Correct Input to ML

```
import numpy as np
import pandas as pd
import os

df_training = pd.read_excel("./excel_data/training_data.xlsx") #
    training dataframe
window_size = 7

x_training = df_training.drop(columns =
    ['date','realized_volatility'])
x_training_numpy = x_training.to_numpy()
subsets_x_training = [x_training_numpy[i:i+window_size] for i in
    range(len(x_training_numpy) - window_size)]
subsets_x_training_numpy = np.array(subsets_x_training)
print("The shape of subsets_x_training_numpy is: ",
    subsets_x_training_numpy.shape)

head = subsets_x_training_numpy[:1]
print(head)

# Extract the first 6 features and last 8 features
first_6_features = subsets_x_training_numpy[:, :, :6]
last_8_features = subsets_x_training_numpy[:, :, 6:]

# Compute mean and std for the first 6 features across samples and
    timesteps
# this normalizes across each of the first 6 features
mean = np.mean(first_6_features, axis=(0, 1), keepdims=True)
std = np.std(first_6_features, axis=(0, 1), keepdims=True)

# Normalize the first 6 features
normalized_first_6 = (first_6_features - mean) / std

# Concatenate normalized and unchanged features
data_8_training_x = np.concatenate([normalized_first_6,
```

```
    last_8_features], axis=-1)

print("\nNormalized Data:")
head = data_8_training_x[:1] # results look correct
print(head)

y_training = df_training[['realized_volatility']]
y_training_numpy = y_training['realized_volatility'].to_numpy()
y_training_truncated = y_training_numpy[window_size:]

# save numpy data to folder
folder_path = "./numpy_data"
os.makedirs(folder_path, exist_ok=True)

# save data_8_training_x
file_path = os.path.join(folder_path, "data_8_training_x.npy")
np.save(file_path, data_8_training_x)

# save y_training_truncated
file_path = os.path.join(folder_path, "training_y.npy")
np.save(file_path, y_training_truncated)

df_testing = pd.read_excel("./excel_data/testing_data.xlsx") #
    training dataframe

x_testing = df_testing.drop(columns =
    ['date','realized_volatility'])
x_testing_numpy = x_testing.to_numpy()
subsets_x_testing = [x_testing_numpy[i:i+window_size] for i in
    range(len(x_testing_numpy) - window_size)]
subsets_x_testing_numpy = np.array(subsets_x_testing)
print("The shape of subsets_x_testing_numpy is: ",
    subsets_x_testing_numpy.shape)

# Extract the first 6 features and last 8 features
first_6_features = subsets_x_testing_numpy[:, :, :6]
last_8_features = subsets_x_testing_numpy[:, :, 6:]

# Compute mean and std for the first 6 features across samples and
    timesteps
```

```
# this normalizes across each of the first 6 features
mean = np.mean(first_6_features, axis=(0, 1), keepdims=True)
std = np.std(first_6_features, axis=(0, 1), keepdims=True)

# Normalize the first 6 features
normalized_first_6 = (first_6_features - mean) / std

# Concatenate normalized and unchanged features
data_8_testing_x = np.concatenate([normalized_first_6,
   last_8_features], axis=-1)

y_testing = df_testing[['realized_volatility']]
y_testing_numpy = y_testing['realized_volatility'].to_numpy()
y_testing_truncated = y_testing_numpy[window_size:]

y_testing_truncated.shape

# save data_8_testing_x
file_path = os.path.join(folder_path, "data_8_testing_x.npy")
np.save(file_path, data_8_testing_x)

# save y_testing_truncated
file_path = os.path.join(folder_path, "testing_y.npy")
np.save(file_path, y_testing_truncated)

x_training = df_training.drop(columns =
   ['date','realized_volatility'])
x_training = x_training.drop(x_training.columns[-8:], axis=1)
x_training.head()

x_training_numpy = x_training.to_numpy()
subsets_x_training = [x_training_numpy[i:i+window_size] for i in
   range(len(x_training_numpy) - window_size)]
subsets_x_training_numpy = np.array(subsets_x_training)
print("The shape of subsets_x_training_numpy is: ",
   subsets_x_training_numpy.shape)

head = subsets_x_training_numpy[:1]
print(head)

```

```
first_6_features = subsets_x_training_numpy[:, :, :6]

# Compute mean and std for the first 6 features across samples and timesteps
# this normalizes across each of the first 6 features
mean = np.mean(first_6_features, axis=(0, 1), keepdims=True)
std = np.std(first_6_features, axis=(0, 1), keepdims=True)

# Normalize the first 6 features
data_1_training_x = (first_6_features - mean) / std

print("\nNormalized Data:")
head = data_1_training_x[:1] # results look correct
print(head)

file_path = os.path.join(folder_path, "data_1_training_x.npy")
np.save(file_path, data_1_training_x)

df_testing = pd.read_excel("./excel_data/testing_data.xlsx") # training dataframe
x_testing = df_testing.drop(columns = ['date','realized_volatility'])
x_testing = x_testing.drop(x_testing.columns[-8:], axis=1)
x_testing_numpy = x_testing.to_numpy()
subsets_x_testing = [x_testing_numpy[i:i+window_size] for i in range(len(x_testing_numpy) - window_size)]
subsets_x_testing_numpy = np.array(subsets_x_testing)
print("The shape of subsets_x_testing_numpy is: ", subsets_x_testing_numpy.shape)

# normalize
first_6_features = subsets_x_testing_numpy[:, :, :6]

# Compute mean and std for the first 6 features across samples and timesteps
# this normalizes across each of the first 6 features
mean = np.mean(first_6_features, axis=(0, 1), keepdims=True)
std = np.std(first_6_features, axis=(0, 1), keepdims=True)

# Normalize the first 6 features
```

```
data_1_testing_x = (first_6_features - mean) / std

head = data_1_testing_x[:1] # results look correct
print(head)

# save data_1_testing_x
file_path = os.path.join(folder_path, "data_1_testing_x.npy")
np.save(file_path, data_1_testing_x)
```

Part 3 - Fit ML Models

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.layers import LSTM, Input, Dropout,
    Normalization, GlobalAveragePooling1D
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dense
from tensorflow.keras.optimizers import Adam
import os.path

# GEW-LSTM
gew_lstm_model = Sequential()
gew_lstm_model.add(Input(shape=[7, 14])) # 7 time steps, 14 features
# 3 + 3 + 2 + 6 Garch / Egarch / EWMA / changing parameters
gew_lstm_model.add(LSTM(10, return_sequences=True))
gew_lstm_model.add(Dropout(0.3))
gew_lstm_model.add(LSTM(4, return_sequences=True))
gew_lstm_model.add(Dropout(0.8))
gew_lstm_model.add(LSTM(2))
gew_lstm_model.add(Dropout(0.8))
gew_lstm_model.add(Dense(10,activation='sigmoid')) # to be
    consistent with the paper
gew_lstm_model.add(Dense(5,activation='sigmoid'))
gew_lstm_model.add(Dense(1)) # no activation function (default is
    just linear)

# Explanatory Variables-LSTM
lstm_model = Sequential()
lstm_model.add(Input(shape=[7, 6])) # 6 explanatory varaibles
```

```
lstm_model.add(LSTM(10, return_sequences=True))
lstm_model.add(Dropout(0.3))
lstm_model.add(LSTM(4, return_sequences=True))
lstm_model.add(Dropout(0.8))
lstm_model.add(LSTM(2))
lstm_model.add(Dropout(0.8))
lstm_model.add(Dense(10,activation='sigmoid')) # to be consistent with the paper
lstm_model.add(Dense(5,activation='sigmoid'))
lstm_model.add(Dense(1)) # no activation function (default is just linear)

# Explanatory Variables-GEW-DNN
gew_dnn_model = Sequential()
gew_dnn_model.add(Input(shape = [7, 14])) # 7 past time steps / 14 explanatory variables
# 3 + 3 + 2 + 6 Garch / Egarch / EWMA / changing parameters
gew_dnn_model.add(Dense(10,activation='sigmoid'))
gew_dnn_model.add(Dense(5,activation='sigmoid'))
gew_dnn_model.add(GlobalAveragePooling1D())  # Flatten the time-step dimension
gew_dnn_model.add(Dense(1))

# Explanatory Variables-DNN
dnn_model = Sequential()
dnn_model.add(Input(shape = [7, 6])) # 7 past time steps / 6 explanatory variables
dnn_model.add(Dense(10,activation='sigmoid'))
dnn_model.add(Dense(5,activation='sigmoid'))
dnn_model.add(GlobalAveragePooling1D())  # Flatten the time-step dimension
dnn_model.add(Dense(1))

data_1_testing_x = np.load("./numpy_data/data_1_testing_x.npy")
data_1_training_x = np.load("./numpy_data/data_1_training_x.npy")
data_8_testing_x = np.load("./numpy_data/data_8_testing_x.npy")
data_8_training_x = np.load("./numpy_data/data_8_training_x.npy")
testing_y = np.load("./numpy_data/testing_y.npy")
training_y = np.load("./numpy_data/training_y.npy")

```

```
gew_lstm_model.compile(optimizer = Adam(learning_rate = 0.0001),
    loss = 'mse', metrics = ['mse'])
gew_lstm_model.fit(x = data_8_training_x, y = training_y,
    validation_split = 0.2, batch_size =20, epochs = 150, verbose =
    1)

# save model
if os.path.isfile('ml_models/gew_ltsm_model.keras') is False:
    gew_lstm_model.save('ml_models/gew_lstm_model.keras',
        include_optimizer=False)

lstm_model.compile(optimizer = Adam(learning_rate = 0.0001), loss =
    'mse', metrics = ['mse'])
lstm_model.fit(x = data_1_training_x, y = training_y,
    validation_split = 0.2, batch_size =20, epochs = 150, verbose =
    1)

# save model
if os.path.isfile('ml_models/ltsm_model.keras') is False:
    lstm_model.save('ml_models/lstm_model.keras',
        include_optimizer=False)

gew_dnn_model.compile(optimizer = Adam(learning_rate = 0.0001),
    loss = 'mse', metrics = ['mse'])
gew_dnn_model.fit(x = data_8_training_x, y = training_y,
    validation_split = 0.2, batch_size =20, epochs = 150, verbose =
    1)

# save model
if os.path.isfile('ml_models/gew_dnn_model.keras') is False:
    gew_dnn_model.save('ml_models/gew_dnn_model.keras',
        include_optimizer=False)

dnn_model.compile(optimizer = Adam(learning_rate = 0.0001), loss =
    'mse', metrics = ['mse'])
dnn_model.fit(x = data_1_training_x, y = training_y,
    validation_split = 0.2, batch_size =20, epochs = 150, verbose =
    1)

# save model
```

```
if os.path.isfile('ml_models/dnn_model.keras') is False:
    dnn_model.save('ml_models/dnn_model.keras',
       include_optimizer=False)
```

Part 4 - Predict Using ML

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.layers import LSTM, Input, Dropout,
   Normalization, GlobalAveragePooling1D
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dense
from tensorflow.keras.optimizers import Adam
import os.path
from keras.models import load_model
import matplotlib.pyplot as plt

data_1_testing_x = np.load("./numpy_data/data_1_testing_x.npy")
data_1_training_x = np.load("./numpy_data/data_1_training_x.npy")
data_8_testing_x = np.load("./numpy_data/data_8_testing_x.npy")
data_8_training_x = np.load("./numpy_data/data_8_training_x.npy")
testing_y = np.load("./numpy_data/testing_y.npy")
training_y = np.load("./numpy_data/training_y.npy")

dnn_model = load_model('ml_models/dnn_model.keras')
gew_dnn_model = load_model('ml_models/gew_dnn_model.keras')
lstm_model = load_model('ml_models/lstm_model.keras')
gew_lstm_model = load_model('ml_models/gew_lstm_model.keras')

y_pred_dnn_model = dnn_model.predict(data_1_testing_x)
y_pred_gew_dnn_model = gew_dnn_model.predict(data_8_testing_x)
y_pred_lstm_model = lstm_model.predict(data_1_testing_x)
y_pred_gew_lstm_model = gew_lstm_model.predict(data_8_testing_x)

time = np.arange(len(testing_y))

plt.plot(time, y_pred_dnn_model, label='Predicted DNN Volatility',
   color='blue', linestyle='-')
plt.plot(time, testing_y, label='Realized Volatility', color='red',
```

```
    linestyle='-')
plt.title("DNN Volatility Prediction v. Realized Volatility")
plt.xlabel("Time in Days")
plt.ylabel("Volatility")

plt.plot(time, y_pred_gew_dnn_model, label='Predicted GEW DNN
    Volatility', color='blue', linestyle='-')
plt.plot(time, testing_y, label='Realized Volatility', color='red',
    linestyle='-')
plt.title("GEW DNN Volatility Prediction v. Realized Volatility")
plt.xlabel("Time in Days")
plt.ylabel("Volatility")

plt.plot(time, y_pred_lstm_model, label='Predicted LSTM
    Volatility', color='blue', linestyle='-')
plt.plot(time, testing_y, label='Realized Volatility', color='red',
    linestyle='-')
plt.title("LSTM Volatility Prediction v. Realized Volatility")
plt.xlabel("Time in Days")
plt.ylabel("Volatility")

plt.plot(time, y_pred_gew_lstm_model, label='Predicted GEW-LSTM
    Volatility', color='blue', linestyle='-')
plt.plot(time, testing_y, label='Realized Volatility', color='red',
    linestyle='-')
plt.title("GEW-LSTM Volatility Prediction v. Realized Volatility")
plt.xlabel("Time in Days")
plt.ylabel("Volatility")

from sklearn.metrics import mean_squared_error

mse_dnn_model = mean_squared_error(y_pred_dnn_model, testing_y)
mse_gew_dnn_model = mean_squared_error(y_pred_gew_dnn_model,
    testing_y)
mse_lstm_model = mean_squared_error(y_pred_lstm_model,testing_y)
mse_gew_lstm_model = mean_squared_error(y_pred_gew_lstm_model,
    testing_y)

print("Mean Squared Error for DNN Model (6 parameters) is: ",
    mse_dnn_model)
```

```
print("Mean Squared Error for GEW-DNN Model (14 parameters) is: ",
    mse_gew_dnn_model)
print("Mean Squared Error for LSTM Model (6 parameters) is: ",
    mse_lstm_model)
print("Mean Squared Error for GEW-LSTM Model (14 parameters) is: ",
    mse_gew_lstm_model)
```