Forecasting corn futures volatility by HAR

Table of contents

HAR-RV

Data:

CN2MIN.csv

When I started working, I realized that only the minutely data is needed. We will be forecasting daily realized variances, which I'll be calculating from the minutely returns as follows:

$$RV_t = \sum_{i=1}^{M} r_{t,i}^2$$

where t represents the current day, and M is the number of minutely return observations on that day.

Methodology:

HARX model from the ARCH package is used to perform the heterogeneous autoregression (HAR) of corn futures' realized variances.

Price is taken to be the closing price.

Returns are calculated as the first difference of the natural logarithm of prices:

p_t: price at time t

p_{t-1}: price at time $t-1$

r_t: return at time t

$$r_t = \ln(p_t) - \ln(p_{t-1}) = \ln(\frac{p_t}{p_{t-1}})$$

I'll provide more details and the full specification of the model later in the document.

Load modules

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from arch.univariate import HARX
import os
```

Set default attributes for plots

In [2]:
```
plt.rc(group = "figure", figsize = (16, 9)) # figure size: 16/9
plt.rc(group = "font", size = 14) # font size of figure annotations
```

Data preparation

Import data

As I mentioned before only the minutely data is needed.

In [3]:
```
price = pd.read_csv("CN2MIN.csv",
                    usecols = ["Date", "CLOSE"], # only the closing prices are needed
                    index_col = 0,
                    parse_dates = True)

price.columns = ['Price'] # change the name of the column from "CLOSE" to "Price"

price
```

Out[3]:

Date	Price
2022-01-18 01:01:00	590.00
2022-01-18 01:02:00	590.00
2022-01-18 01:03:00	589.75
2022-01-18 01:04:00	590.50
2022-01-18 01:05:00	591.00
...	...
2022-04-14 18:16:00	783.75
2022-04-14 18:17:00	784.00
2022-04-14 18:18:00	784.25
2022-04-14 18:19:00	784.25
2022-04-14 18:20:00	784.50

47561 rows × 1 columns

Check for missing observations

In [4]:
```
any(price.isna()) # are there any missing values?
```

Out[4]: True

In [5]:
```
price[price.isna().values] # which are the missing values and how many?
```

Out[5]:

Date	Price
2022-01-18 01:22:00	NaN
2022-01-18 01:25:00	NaN
2022-01-18 01:52:00	NaN
2022-01-18 01:55:00	NaN
2022-01-18 02:07:00	NaN
...	...
2022-04-14 11:27:00	NaN
2022-04-14 11:34:00	NaN
2022-04-14 11:54:00	NaN
2022-04-14 12:13:00	NaN
2022-04-14 12:44:00	NaN

2260 rows × 1 columns

There are missing observations. But does the Date index account for every minute within the trading hours? Also, what are the trading hours?

In [6]:
```
index = price.index.to_frame() # extracting and coverting the index into a dataframe f
```

Aggregating the index series into hourly bins to see it's hourly profile

In [7]:
```
hourly = index.resample('H')
```

In [8]:
```
start = hourly.first().squeeze().dt.time # the first minute of the hour
end = hourly.last().squeeze().dt.time # the last minute of the hour
count = hourly.count() # number of minutely observations in the hour
```

In [9]:
```
hourly_summary = pd.concat((start, end, count), axis = 1)
hourly_summary.columns = ['Start', 'End', 'Count']
hourly_summary.index.name = 'Hour'
hourly_summary
```

Out[9]:

Hour	Start	End	Count
2022-01-18 01:00:00	01:01:00	01:59:00	53
2022-01-18 02:00:00	02:03:00	02:58:00	16
2022-01-18 03:00:00	03:03:00	03:58:00	18
2022-01-18 04:00:00	04:01:00	04:54:00	4
2022-01-18 05:00:00	05:10:00	05:56:00	8
...	...	...	...
2022-04-14 14:00:00	14:00:00	14:59:00	60
2022-04-14 15:00:00	15:00:00	15:59:00	60
2022-04-14 16:00:00	16:00:00	16:59:00	60
2022-04-14 17:00:00	17:00:00	17:59:00	60
2022-04-14 18:00:00	18:00:00	18:20:00	21

2082 rows × 3 columns

As you can see the sequence is irregular. Number of observations is not 60 for each hour. This is a problem since we will be calculating and estimating the variance from minutely returns. The returns must be calculated over equal intervals of 1 minute.

Before we deal with these problems, let's also look at the daily profile.

In [10]:
```
daily = index.resample('B')  # aggregate the index data into bins of Business days
```

In [11]:
```
start = daily.first().squeeze().dt.time # the first minute of the day
end = daily.last().squeeze().dt.time # the last minute of the day
```

In [12]:
```
daily_summary = pd.concat((start, end), axis = 1)
daily_summary.columns = ['Start', 'End']
daily_summary
```

Out[12]:

	Start	End
Date		
2022-01-18	01:01:00	19:20:00
2022-01-19	01:01:00	19:20:00
2022-01-20	01:01:00	19:20:00
2022-01-21	01:01:00	19:20:00
2022-01-24	01:01:00	19:20:00
...	...	...
2022-04-08	00:01:00	18:20:00
2022-04-11	00:01:00	18:20:00
2022-04-12	00:01:00	18:20:00
2022-04-13	00:01:00	18:20:00
2022-04-14	00:01:00	18:20:00

63 rows × 2 columns

The trading hours are not all the same, because of the daylight savings. To avoid unnecessary complexity, I'll change the trading hours so that they are all between 01:01:00 and 19:20:00

I exported the "daily_summary" data into a csv file to see exactly when the daylight savings start.

In [13]:
```
# daily_summary.to_csv("daily_summary.csv") # export into csv file
# os.startfile("daily_summary.csv") # open the csv file
```

The daylight savings start on 2022-03-14

In [14]:
```
dst_start = '2022-03-14'
```

Going back to our price data

In [15]:
```
price
```

Out[15]:

Date	Price
2022-01-18 01:01:00	590.00
2022-01-18 01:02:00	590.00
2022-01-18 01:03:00	589.75
2022-01-18 01:04:00	590.50
2022-01-18 01:05:00	591.00
...	...
2022-04-14 18:16:00	783.75
2022-04-14 18:17:00	784.00
2022-04-14 18:18:00	784.25
2022-04-14 18:19:00	784.25
2022-04-14 18:20:00	784.50

47561 rows × 1 columns

You can see that the Date column is the index. In Pandas DataFrame, an index is immutable, that is, it cannot be changed. So, we first convert the Date index into a regular column.

In [16]:
```
price_1 = price.reset_index()
price_1
```

Out[16]:

	Date	Price
0	2022-01-18 01:01:00	590.00
1	2022-01-18 01:02:00	590.00
2	2022-01-18 01:03:00	589.75
3	2022-01-18 01:04:00	590.50
4	2022-01-18 01:05:00	591.00
...	...	...
47556	2022-04-14 18:16:00	783.75
47557	2022-04-14 18:17:00	784.00
47558	2022-04-14 18:18:00	784.25
47559	2022-04-14 18:19:00	784.25
47560	2022-04-14 18:20:00	784.50

47561 rows × 2 columns

Date is now a column in the dataframe. Now, we add 1 hour to all the dates after "dst_start"

In [17]:
```
mask = price_1.Date > dst_start
price_1.loc[mask, "Date"] = price_1.loc[mask, "Date"] + pd.Timedelta('1 hour')
price_1
```

Out[17]:

	Date	Price
0	2022-01-18 01:01:00	590.00
1	2022-01-18 01:02:00	590.00
2	2022-01-18 01:03:00	589.75
3	2022-01-18 01:04:00	590.50
4	2022-01-18 01:05:00	591.00
...	...	...
47556	2022-04-14 19:16:00	783.75
47557	2022-04-14 19:17:00	784.00
47558	2022-04-14 19:18:00	784.25
47559	2022-04-14 19:19:00	784.25
47560	2022-04-14 19:20:00	784.50

47561 rows × 2 columns

Then we convert the Date column back into the index.

In [18]:
```
price_1 = price_1.set_index('Date')
price_1
```

Out[18]:

	Price
Date	
2022-01-18 01:01:00	590.00
2022-01-18 01:02:00	590.00
2022-01-18 01:03:00	589.75
2022-01-18 01:04:00	590.50
2022-01-18 01:05:00	591.00
...	...
2022-04-14 19:16:00	783.75
2022-04-14 19:17:00	784.00
2022-04-14 19:18:00	784.25
2022-04-14 19:19:00	784.25
2022-04-14 19:20:00	784.50

47561 rows × 1 columns

Now, going back to the problem of the missing minutes in dates.

We first change the index so that it is a regular sequence of minutes.

The following will 'resample' price data so that the Date index is a regular sequence of minutes from the beginning of the sample to the end. But note that it will literally span every minute between these two points of time, regardless of whether they are off trading hours or on weekends.

Of course, for the minutes for which we have no data, it will insert NaNs in their place.

In [19]:
```
price_2 = price_1.resample('T').first()
price_2
```

Out[19]:

	Price
Date	
2022-01-18 01:01:00	590.00
2022-01-18 01:02:00	590.00
2022-01-18 01:03:00	589.75
2022-01-18 01:04:00	590.50
2022-01-18 01:05:00	591.00
...	...
2022-04-14 19:16:00	783.75
2022-04-14 19:17:00	784.00
2022-04-14 19:18:00	784.25
2022-04-14 19:19:00	784.25
2022-04-14 19:20:00	784.50

124940 rows × 1 columns

So, now we need to filter the data to contain only the dates in the original data, and only the minutes between our established trading hours: 01:01:00 to 19:20:00

Getting the list of dates in the original "price" data, and storing it in the variable "dates".

In [20]:
```
dates = np.unique(price.index.date)
dates
```

Out[20]:
```
array([datetime.date(2022, 1, 18), datetime.date(2022, 1, 19),
       datetime.date(2022, 1, 20), datetime.date(2022, 1, 21),
       datetime.date(2022, 1, 24), datetime.date(2022, 1, 25),
       datetime.date(2022, 1, 26), datetime.date(2022, 1, 27),
       datetime.date(2022, 1, 28), datetime.date(2022, 1, 31),
       datetime.date(2022, 2, 1), datetime.date(2022, 2, 2),
       datetime.date(2022, 2, 3), datetime.date(2022, 2, 4),
       datetime.date(2022, 2, 7), datetime.date(2022, 2, 8),
       datetime.date(2022, 2, 9), datetime.date(2022, 2, 10),
       datetime.date(2022, 2, 11), datetime.date(2022, 2, 14),
       datetime.date(2022, 2, 15), datetime.date(2022, 2, 16),
       datetime.date(2022, 2, 17), datetime.date(2022, 2, 18),
       datetime.date(2022, 2, 22), datetime.date(2022, 2, 23),
       datetime.date(2022, 2, 24), datetime.date(2022, 2, 25),
       datetime.date(2022, 2, 28), datetime.date(2022, 3, 1),
       datetime.date(2022, 3, 2), datetime.date(2022, 3, 3),
       datetime.date(2022, 3, 4), datetime.date(2022, 3, 7),
       datetime.date(2022, 3, 8), datetime.date(2022, 3, 9),
       datetime.date(2022, 3, 10), datetime.date(2022, 3, 11),
       datetime.date(2022, 3, 14), datetime.date(2022, 3, 15),
       datetime.date(2022, 3, 16), datetime.date(2022, 3, 17),
       datetime.date(2022, 3, 18), datetime.date(2022, 3, 21),
       datetime.date(2022, 3, 22), datetime.date(2022, 3, 23),
       datetime.date(2022, 3, 24), datetime.date(2022, 3, 25),
       datetime.date(2022, 3, 28), datetime.date(2022, 3, 29),
       datetime.date(2022, 3, 30), datetime.date(2022, 3, 31),
       datetime.date(2022, 4, 1), datetime.date(2022, 4, 4),
       datetime.date(2022, 4, 5), datetime.date(2022, 4, 6),
       datetime.date(2022, 4, 7), datetime.date(2022, 4, 8),
       datetime.date(2022, 4, 11), datetime.date(2022, 4, 12),
       datetime.date(2022, 4, 13), datetime.date(2022, 4, 14)],
      dtype=object)
```

Adding a date component column to the current price ("price_2") data so that we can apply a date-based filter.

In [21]:
```
price_2["DateComp"] = price_2.index.date
price_2
```

Out[21]:

Date	Price	DateComp
2022-01-18 01:01:00	590.00	2022-01-18
2022-01-18 01:02:00	590.00	2022-01-18
2022-01-18 01:03:00	589.75	2022-01-18
2022-01-18 01:04:00	590.50	2022-01-18
2022-01-18 01:05:00	591.00	2022-01-18
...	...	...
2022-04-14 19:16:00	783.75	2022-04-14
2022-04-14 19:17:00	784.00	2022-04-14
2022-04-14 19:18:00	784.25	2022-04-14
2022-04-14 19:19:00	784.25	2022-04-14
2022-04-14 19:20:00	784.50	2022-04-14

124940 rows × 2 columns

Applying the filter.

In [22]:
```
price_3 = price_2[price_2["DateComp"].isin(dates)].copy()
price_3
```

Out[22]:

Date	Price	DateComp
2022-01-18 01:01:00	590.00	2022-01-18
2022-01-18 01:02:00	590.00	2022-01-18
2022-01-18 01:03:00	589.75	2022-01-18
2022-01-18 01:04:00	590.50	2022-01-18
2022-01-18 01:05:00	591.00	2022-01-18
...	...	...
2022-04-14 19:16:00	783.75	2022-04-14
2022-04-14 19:17:00	784.00	2022-04-14
2022-04-14 19:18:00	784.25	2022-04-14
2022-04-14 19:19:00	784.25	2022-04-14
2022-04-14 19:20:00	784.50	2022-04-14

88940 rows × 2 columns

Now, applying the time filter.

In [23]:
```
index = price_3.index.indexer_between_time('01:01:00', '19:20:00')
price_4 = price_3.iloc[index, 0].copy()
price_4
```

Out[23]:
```
Date
2022-01-18 01:01:00    590.00
2022-01-18 01:02:00    590.00
2022-01-18 01:03:00    589.75
2022-01-18 01:04:00    590.50
2022-01-18 01:05:00    591.00
                        ...
2022-04-14 19:16:00    783.75
2022-04-14 19:17:00    784.00
2022-04-14 19:18:00    784.25
2022-04-14 19:19:00    784.25
2022-04-14 19:20:00    784.50
Name: Price, Length: 68200, dtype: float64
```

We are done with regularizing the date index. Let's check the results.

Checking the hourly profile.

In [24]:
```
index = price_4.index.to_frame() # extracting and coverting the index into a dataframe

hourly = index.resample('H')

start = hourly.first().squeeze().dt.time # the first minute in the hour
end = hourly.last().squeeze().dt.time # the last minute in the hour
count = hourly.count() # number of minutely data in the hour

hourly_summary = pd.concat((start, end, count), axis = 1)
hourly_summary.columns = ['Start', 'End', 'Count']
hourly_summary
```

Out[24]:

	Start	End	Count
Date			
2022-01-18 01:00:00	01:01:00	01:59:00	59
2022-01-18 02:00:00	02:00:00	02:59:00	60
2022-01-18 03:00:00	03:00:00	03:59:00	60
2022-01-18 04:00:00	04:00:00	04:59:00	60
2022-01-18 05:00:00	05:00:00	05:59:00	60
...	...	...	...
2022-04-14 15:00:00	15:00:00	15:59:00	60
2022-04-14 16:00:00	16:00:00	16:59:00	60
2022-04-14 17:00:00	17:00:00	17:59:00	60
2022-04-14 18:00:00	18:00:00	18:59:00	60
2022-04-14 19:00:00	19:00:00	19:20:00	21

2083 rows × 3 columns

Every minute is accounted for.

Checking the daily profile.

In [25]:
```
daily = index.resample('B')  # aggregate the index data into bins of Business days

start = daily.first().squeeze().dt.time # the first time of the day
end = daily.last().squeeze().dt.time # the last time of the day
count = daily.count() # number of observations in the day

daily_summary = pd.concat((start, end, count), axis = 1)
daily_summary.columns = ['Start', 'End', 'Count']
daily_summary
```

Out[25]:

Date	Start	End	Count
2022-01-18	01:01:00	19:20:00	1100
2022-01-19	01:01:00	19:20:00	1100
2022-01-20	01:01:00	19:20:00	1100
2022-01-21	01:01:00	19:20:00	1100
2022-01-24	01:01:00	19:20:00	1100
...	...	...	...
2022-04-08	01:01:00	19:20:00	1100
2022-04-11	01:01:00	19:20:00	1100
2022-04-12	01:01:00	19:20:00	1100
2022-04-13	01:01:00	19:20:00	1100
2022-04-14	01:01:00	19:20:00	1100

63 rows × 3 columns

There are equal number of observations in every day.

So, we have ensured that we have a regular time series data. But there are still missing values. We have not dealt with them.

In [26]:
```
price_4[price_4.isna().values]
```

Out[26]:
```
Date
2022-01-18 01:22:00   NaN
2022-01-18 01:25:00   NaN
2022-01-18 01:41:00   NaN
2022-01-18 01:42:00   NaN
2022-01-18 01:43:00   NaN
                       ..
2022-04-14 14:26:00   NaN
2022-04-14 14:27:00   NaN
2022-04-14 14:28:00   NaN
2022-04-14 14:29:00   NaN
2022-04-14 14:30:00   NaN
Name: Price, Length: 22899, dtype: float64
```

Assuming that the values are missing because they haven't changed since the last valid observation, we will '~~backfill~~ forward-fill' the data, that is, replace the NaNs with the last valid observation.

I made a mistake last time. We need to use forward-fill instead of back-fill. Back-fill brings the next valid observation back. But we need to bring the last valid observation forward. (Although this change will have no effect on the results, it was an error that needed correction.)

In [27]:
```
price_5 = price_4.ffill()
price_5
```

Out[27]:
```
Date
2022-01-18 01:01:00    590.00
2022-01-18 01:02:00    590.00
2022-01-18 01:03:00    589.75
2022-01-18 01:04:00    590.50
2022-01-18 01:05:00    591.00
                        ...
2022-04-14 19:16:00    783.75
2022-04-14 19:17:00    784.00
2022-04-14 19:18:00    784.25
2022-04-14 19:19:00    784.25
2022-04-14 19:20:00    784.50
Name: Price, Length: 68200, dtype: float64
```

Check the results

In [28]:
```
price_5[price_5.isna().values]
```

Out[28]:
```
Series([], Name: Price, dtype: float64)
```

Calculations

Calculate the minutely returns

In [29]:
```
r = np.log(price_5).diff() * 100 # in percentage
r
```

Out[29]:
```
Date
2022-01-18 01:01:00         NaN
2022-01-18 01:02:00    0.000000
2022-01-18 01:03:00   -0.042382
2022-01-18 01:04:00    0.127092
2022-01-18 01:05:00    0.084638
                         ...
2022-04-14 19:16:00    0.063816
2022-04-14 19:17:00    0.031893
2022-04-14 19:18:00    0.031883
2022-04-14 19:19:00    0.000000
2022-04-14 19:20:00    0.031873
Name: Price, Length: 68200, dtype: float64
```

Naturally, the first differenial is a NaN. We remove it.

In [30]:
```
r = r.dropna()
r
```

Out[30]:
```
Date
2022-01-18 01:02:00    0.000000
2022-01-18 01:03:00   -0.042382
2022-01-18 01:04:00    0.127092
2022-01-18 01:05:00    0.084638
2022-01-18 01:06:00    0.126823
                         ...
2022-04-14 19:16:00    0.063816
2022-04-14 19:17:00    0.031893
2022-04-14 19:18:00    0.031883
2022-04-14 19:19:00    0.000000
2022-04-14 19:20:00    0.031873
Name: Price, Length: 68199, dtype: float64
```

Calculate the squared returns.

In [31]:
```
r2 = np.square(r)
r2
```

Out[31]:
```
Date
2022-01-18 01:02:00    0.000000
2022-01-18 01:03:00    0.001796
2022-01-18 01:04:00    0.016152
2022-01-18 01:05:00    0.007164
2022-01-18 01:06:00    0.016084
                         ...
2022-04-14 19:16:00    0.004073
2022-04-14 19:17:00    0.001017
2022-04-14 19:18:00    0.001017
2022-04-14 19:19:00    0.000000
2022-04-14 19:20:00    0.001016
Name: Price, Length: 68199, dtype: float64
```

Calculate daily variances.

As mentioned earlier, daily variance is calculated as the sum of squared minutely returns during the day.

In [32]:
```
daily_r2 = r2.resample('B')  # pool the minutely squared returns data into bins of Bus
```

```
RV = daily_r2.sum()  # sum of squared return values in each bin is the daily variance
RV
```

Out[32]:
```
Date
2022-01-18    1.493195
2022-01-19    2.048767
2022-01-20    1.650397
2022-01-21    2.360376
2022-01-24    1.892512
                ...
2022-04-08    3.157537
2022-04-11    2.422808
2022-04-12    1.941506
2022-04-13    1.648066
2022-04-14    1.591114
Freq: B, Name: Price, Length: 63, dtype: float64
```

Note that in the code block above, I used "r2.resample('B')". This will resample the data so that the index contains every business day between the start and the end of the sample. But "business day" definition can vary. I mention it because I noticed one extra date after the resampling. So, we need to remove that extra date.

Remember we stored the list of dates in the original data in the variable "dates".

In [33]:
```
RV = RV[dates]  # this selects only the dates in "dates"
RV  # you'll notice the length has reduced by 1
```

Out[33]:
```
Date
2022-01-18    1.493195
2022-01-19    2.048767
2022-01-20    1.650397
2022-01-21    2.360376
2022-01-24    1.892512
                ...
2022-04-08    3.157537
2022-04-11    2.422808
2022-04-12    1.941506
2022-04-13    1.648066
2022-04-14    1.591114
Name: Price, Length: 62, dtype: float64
```

Change the name of the series to 'RV'

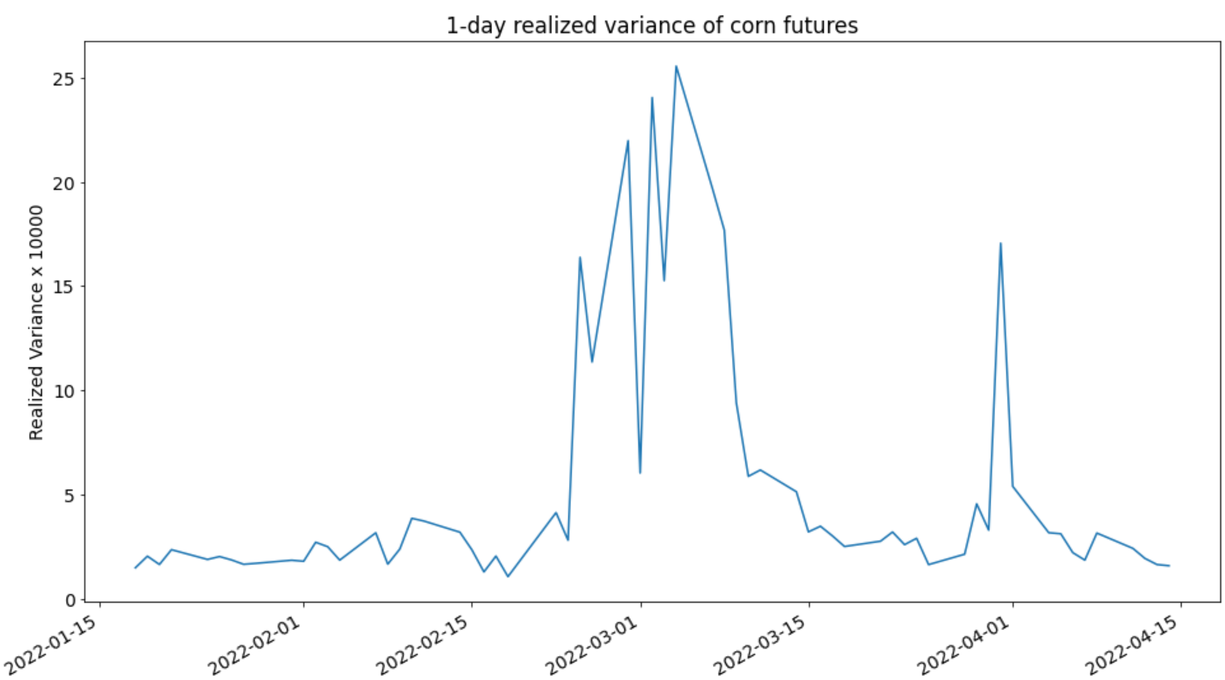
In [34]:
```
RV.name = 'RV'
RV
```

Out[34]:
```
Date
2022-01-18    1.493195
2022-01-19    2.048767
2022-01-20    1.650397
2022-01-21    2.360376
2022-01-24    1.892512
                ...
2022-04-08    3.157537
2022-04-11    2.422808
2022-04-12    1.941506
2022-04-13    1.648066
2022-04-14    1.591114
Name: RV, Length: 62, dtype: float64
```

Plot the daily realized variance

In [35]:
```
RV.plot()
plt.xlabel(None)
plt.ylabel("Realized Variance x 10000")  # remember we multiplied the returns by 100
plt.title("1-day realized variance of corn futures")
```

Out[35]:
```
Text(0.5, 1.0, '1-day realized variance of corn futures')
```



Constructing the model

The HAR model that we will be using is:

$$RV_t = \beta_0 + \beta_1 RV^d_{t-1} + \beta_2 RV^w_{t-1} + \beta_3 RV^m_{t-1} + u_t$$

where

RV_t is the daily realized variance:

RV^d_{t-1} is the daily lagged realized variance: $RV^d_{t-1} = RV_{t-1}$

RV^{w}_{t-1} is the weekly lagged realized variance: $RV^{w}_{t-1} = \frac{1}{5}\sum_{i=1}^{5} RV_{t-i}$

RV^{m}_{t-1} is the monthly lagged realized variance: $RV^{m}_{t-1} = \frac{1}{22}\sum_{i=1}^{22} RV_{t-i}$

To be able to test the model, I'll divide the sample into two sets:

Training set: 2022-01-18 to 2022-04-11, that is, data from all days except the last three.

Testing set: 2022-04-12 to 2022-04-14, data from the last three days.

The training set will be used to estimate the model. The testing set will be used to test its predictions.

In [36]:
```
train_set = RV['2022-01-18':'2022-04-11']
train_set
```

Out[36]:

```
Date
2022-01-18     1.493195
2022-01-19     2.048767
2022-01-20     1.650397
2022-01-21     2.360376
2022-01-24     1.892512
2022-01-25     2.031120
2022-01-26     1.868279
2022-01-27     1.660466
2022-01-28     1.704105
2022-01-31     1.853583
2022-02-01     1.804096
2022-02-02     2.717146
2022-02-03     2.503023
2022-02-04     1.859873
2022-02-07     3.171043
2022-02-08     1.665912
2022-02-09     2.390283
2022-02-10     3.864622
2022-02-11     3.734638
2022-02-14     3.199636
2022-02-15     2.352663
2022-02-16     1.294403
2022-02-17     2.053706
2022-02-18     1.067985
2022-02-22     4.137298
2022-02-23     2.814226
2022-02-24    16.377902
2022-02-25    11.361985
2022-02-28    21.975498
2022-03-01     6.037276
2022-03-02    24.044903
2022-03-03    15.264196
2022-03-04    25.547402
2022-03-07    19.708669
2022-03-08    17.686436
2022-03-09     9.407255
2022-03-10     5.877695
2022-03-11     6.182163
2022-03-14     5.137882
2022-03-15     3.214220
2022-03-16     3.485084
2022-03-17     3.020900
2022-03-18     2.514100
2022-03-21     2.766485
2022-03-22     3.207814
2022-03-23     2.601582
2022-03-24     2.907307
2022-03-25     1.644900
2022-03-28     2.143371
2022-03-29     4.558533
2022-03-30     3.299444
2022-03-31    17.056801
2022-04-01     5.396963
2022-04-04     3.172503
2022-04-05     3.123130
2022-04-06     2.217992
2022-04-07     1.859840
2022-04-08     3.157537
```

```
2022-04-11     2.422808
Name: RV, dtype: float64
```

In [37]:
```
test_set = RV['2022-04-12':'2022-04-14']
test_set
```

Out[37]:
```
Date
2022-04-12    1.941506
2022-04-13    1.648066
2022-04-14    1.591114
Name: RV, dtype: float64
```

Estimating the model using the training set

In [38]:
```
lag_day = 1
lag_week = 5
lag_month = 22

har_model = HARX(y = train_set,
                 lags = [lag_day, lag_week, lag_month],
                 rescale = False)

har_model_fit = har_model.fit()

har_summary = pd.concat([har_model_fit.params, har_model_fit.pvalues], axis = 1)
har_summary.columns = ['Coefficient', 'p-value']
round(har_summary.iloc[:-1, :], 3)
```

Out[38]:

	Coefficient	p-value
Const	6.779	0.020
RV[0:1]	0.288	0.149
RV[0:5]	0.452	0.100
RV[0:22]	-0.717	0.043

All coefficients are significant with p-value < 0.15.

R^2 value of the regression:

In [39]:
```
round(har_model_fit.rsquared, 2)
```

Out[39]:
```
0.42
```

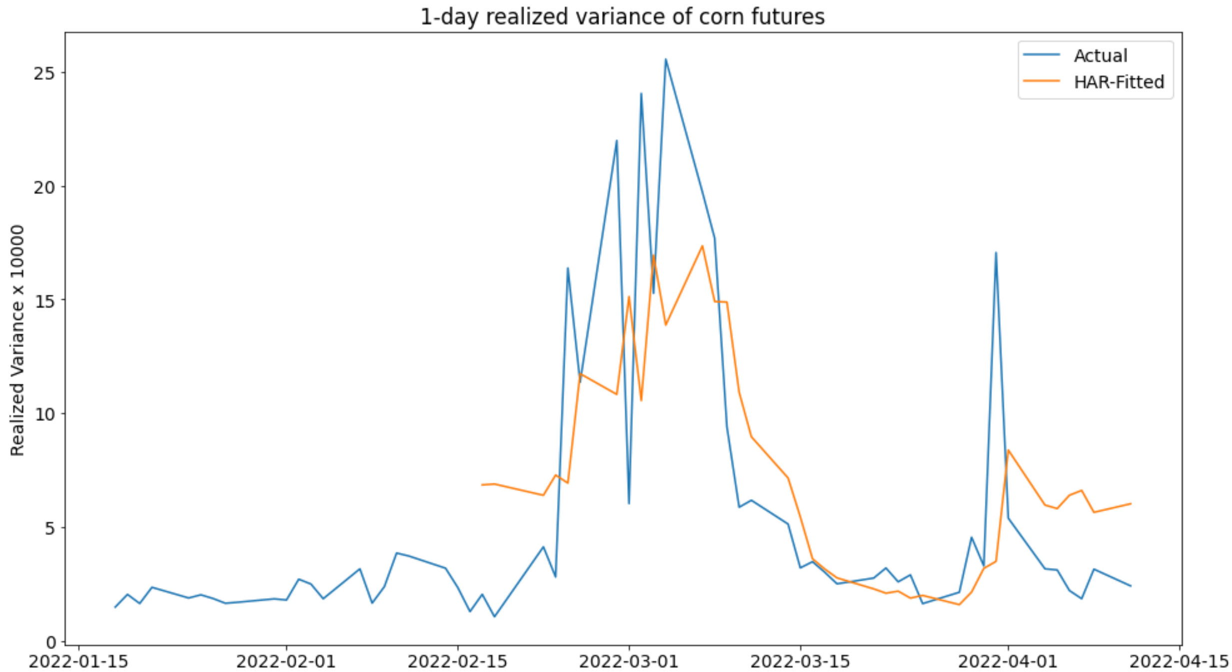
Let's compare the actual values of the dependent variable with the values estimated by the model. (Remember this is not the forecast. We are just comparing the actual values vs. the model-fitted values within the *training set*.)

Actual vs. Fitted values

In [40]:
```
actual_RV = har_model.y  # actual values
fitted_RV = har_model.y - har_model_fit.resid  # fitted values

# Plot of actual vs. fitted values
plt.plot(actual_RV, label = "Actual")
```

```
plt.plot(fitted_RV, label = "HAR-Fitted")
plt.xlabel(None)
plt.ylabel("Realized Variance x 10000")
plt.title("1-day realized variance of corn futures")
plt.legend()
```

Out[40]: <matplotlib.legend.Legend at 0x1eca8026500>



Testing the model

Forecasting daily variance for the next 3 days (the testing set)

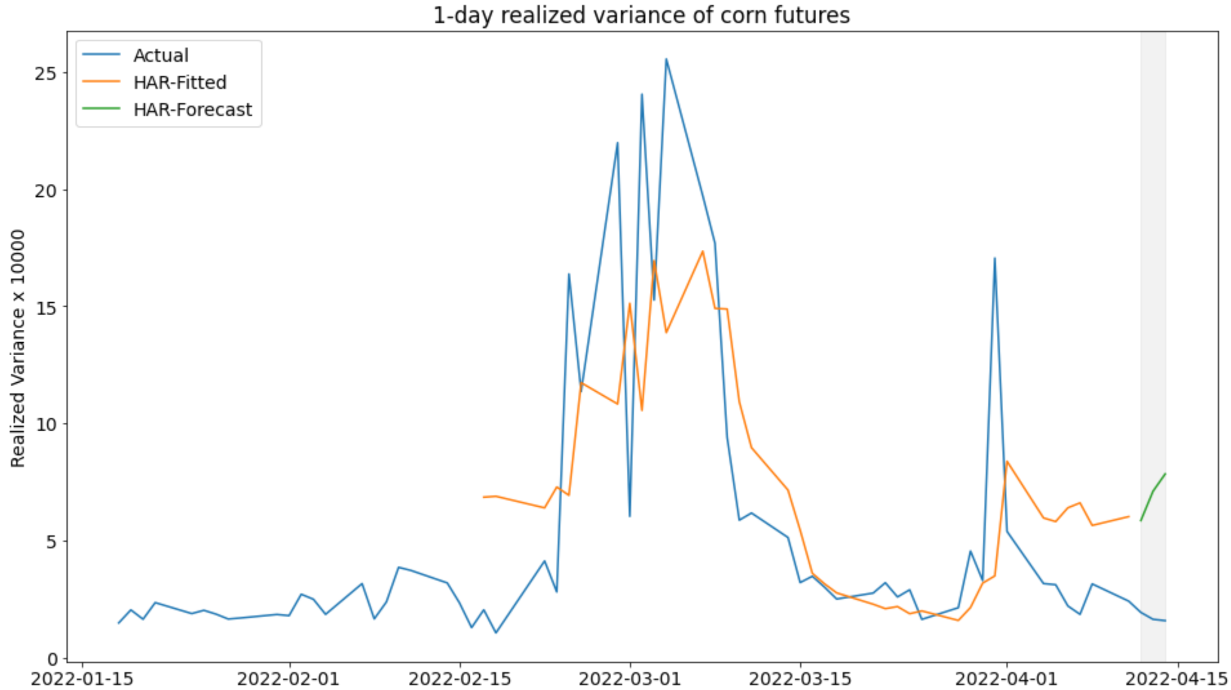
In [41]:
```
forecast_RV = har_model_fit.forecast(horizon = 3, reindex = False)
forecast_RV = forecast_RV.mean.squeeze()
forecast_RV.index = test_set.index
forecast_RV
```

Out[41]:
```
Date
2022-04-12    5.861104
2022-04-13    7.110912
2022-04-14    7.849513
Name: 2022-04-11 00:00:00, dtype: float64
```

Plotting the (Actual) vs. (Fitted + Forecast)

In [42]:
```
plt.plot(RV, label = "Actual")
plt.plot(fitted_RV, label = "HAR-Fitted")
plt.plot(forecast_RV, label = "HAR-Forecast")
plt.legend()
plt.axvspan(xmin = forecast_RV.index.min(),
            xmax = forecast_RV.index.max(),
            color = 'grey',
            alpha = 0.1)
plt.ylabel("Realized Variance x 10000")
plt.title("1-day realized variance of corn futures")
```

Out[42]:
```
Text(0.5, 1.0, '1-day realized variance of corn futures')
```



Comparison with ARCH models

I'll be using the following measure to make the comparison:

Mean Squared Error (MSE): It's the mean of squared deviations of model-estimated values from actual observations.

I'll be calculating an MSE each for

1. the training set: how well the data fits the model?
2. the testing set: how accurate are the forecasts?

Load modules needed for this part of the analysis

In [43]:
```
from arch import arch_model
```

We have calculated and forecasted daily volatility with the HAR model. We will be doing the same with the ARCH models.

To forecast daily volatility with the ARCH models we only need the daily returns. So, for this part of the analysis only "CN2DAY.csv" is used.

In [44]:
```
price = pd.read_csv("CN2DAY.csv",
                    usecols = ["Date", "CLOSE"],
                    index_col = 0,
                    parse_dates = True).squeeze()

price.name = 'Price'
```

```
price
```

Out[44]:
```
Date
2022-01-18    596.50
2022-01-19    607.25
2022-01-20    606.50
2022-01-21    608.50
2022-01-24    610.75
               ...
2022-04-08    760.75
2022-04-11    758.75
2022-04-12    772.50
2022-04-13    778.00
2022-04-14    783.75
Name: Price, Length: 62, dtype: float64
```

Calculate the daily returns

In [45]:
```
r = np.log(price).diff().dropna() * 100
r.name = 'Return'
r
```

Out[45]:
```
Date
2022-01-19    1.786133
2022-01-20   -0.123584
2022-01-21    0.329218
2022-01-24    0.369080
2022-01-25    0.571430
                ...
2022-04-08    1.389830
2022-04-11   -0.263245
2022-04-12    1.795967
2022-04-13    0.709452
2022-04-14    0.736357
Name: Return, Length: 61, dtype: float64
```

Again, dividing the sample into a training set and a testing set

In [46]:
```
r_train = r['2022-01-18':'2022-04-11']
r_test = r['2022-04-12':'2022-04-14']
```

GARCH(1, 1) volatility model

Estimating the model using the training set.

In [47]:
```
garch = arch_model(y = r_train,
                   vol = 'GARCH',
                   p = 1,
                   q = 1)

garch_fit = garch.fit(disp = 'off')

garch_fitted = np.square(garch_fit.conditional_volatility)
```

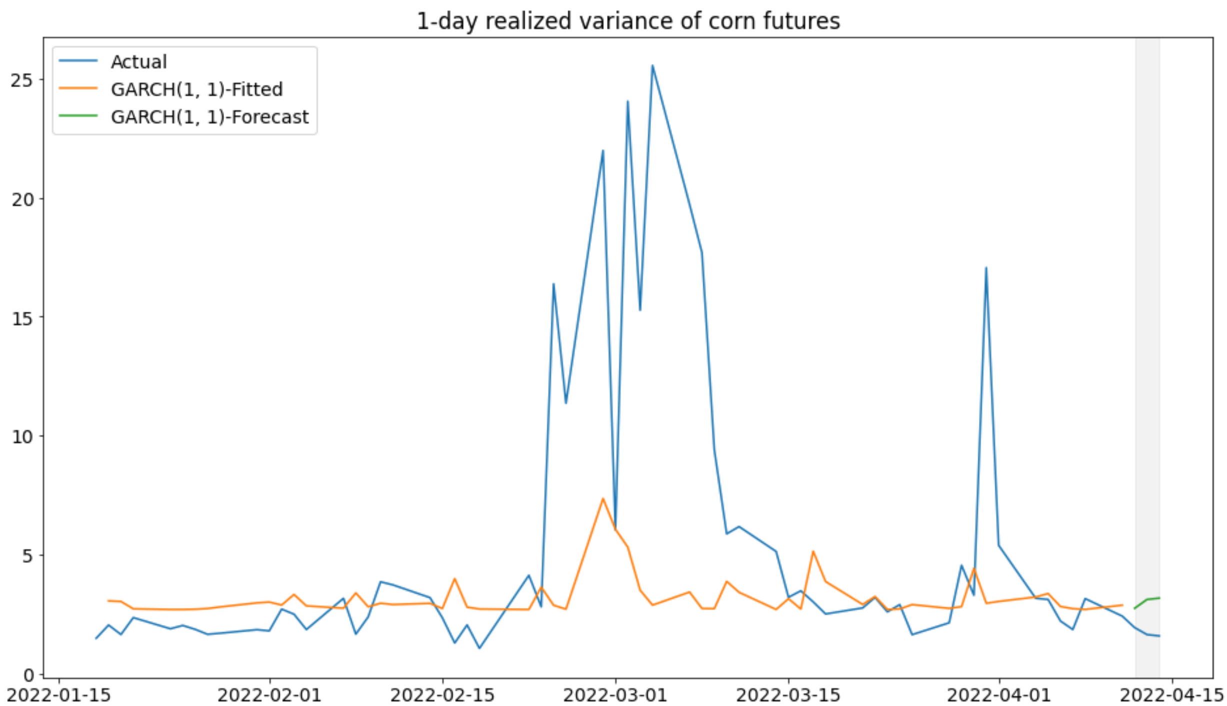
Producing the forecasts

In [48]:
```
garch_forecast = garch_fit.forecast(horizon = 3, reindex = False).variance.squeeze()
garch_forecast.index = r_test.index
```

Plotting the (Actual) vs. (Fitted + Forecast)

In [49]:
```
plt.plot(RV, label = "Actual")
plt.plot(garch_fitted, label = "GARCH(1, 1)-Fitted")
plt.plot(garch_forecast, label = "GARCH(1, 1)-Forecast")
plt.legend()
plt.axvspan(xmin = garch_forecast.index.min(),
            xmax = garch_forecast.index.max(),
            color = 'grey',
            alpha = 0.1)
plt.title("1-day realized variance of corn futures")
```

Out[49]:
```
Text(0.5, 1.0, '1-day realized variance of corn futures')
```



Calculating the MSEs for HAR

In [50]:
```
# HAR: MSE of fit
mse_har_fit = np.mean(np.square(train_set - fitted_RV))

# HAR: MSE of forecast
mse_har_forecast = np.mean(np.square(test_set - forecast_RV))

# Combine the two numbers into a series so we can prepare a table later for comparison
mse_har = pd.Series([mse_har_fit, mse_har_forecast], index = ['Fit', 'Forecast'], name
```

Calculating the MSEs for GARCH(1, 1)

In [51]:
```
# GARCH(1, 1): MSE of fit
mse_garch_fit = np.mean(np.square(train_set-garch_fitted))

# GARCH(1, 1): MSE of forecast
mse_garch_forecast = np.mean(np.square(test_set-garch_forecast))
```

```
# Combine the two numbers into a series so we can prepare a table later for comparison
mse_garch = pd.Series([mse_garch_fit, mse_garch_forecast], index = ['Fit', 'Forecast']
```

Comparison

In [52]:
```
mse_df = round(pd.concat([mse_har, mse_garch], axis = 1).T, 2)
mse_df
```

Out[52]:

	Fit	Forecast
HAR	28.68	28.12
GARCH(1, 1)	39.18	1.78

We get a mixed result. The mean squared error (MSE) of fit from the HAR is model is lower, which says that it provides a better fit for the data. But the mean squared error (MSE) of forecast is much lower for GARCH(1, 1), indicating that GARCH(1, 1) provides a much better forecast. The reasons for this conflicting result could be any of the following.

1. We are forecasting only three values and calculating the mean (squared error) from them. Individual observations are prone to noise. So, for the mean to have any signficance, we need to produce more forecasts.
2. We are working with a small sample size (62 daily returns). It's especially small for HAR because 22 of those are used up in the monthly lag (You'll notice in the HAR plots that the fitted values start around halfway in the middle).
3. I did some reading. According to literature, realized volatilities (RV) tends to follow a lognormal distribution. So, the recommended form of HAR is the one that uses $\ln(RV)$ instead of just RV in the OLS. (I'll send you the reference material.)
4. From the same source, I gather that using too high a frequency is also likely to cause a prediction bias due to something called microstructure effects in the markets. The recommended interval to calculate the returns is 5 mins.

HAR-lnRV

We just need to replace the RV with $\ln RV$ in the HAR-RV model.

$$\ln RV_t = \beta_0 + \beta_1 \ln RV_{t-1}^d + \beta_2 \ln RV_{t-1}^w + \beta_3 \ln RV_{t-1}^m + u_t$$

where

$$\ln RV_{t-1}^d = \ln RV_{t-1}$$

$$\ln RV_{t-1}^w = \frac{1}{5}\sum_{i=1}^{5} \ln RV_{t-i}$$

$$\ln RV_{t-1}^m = \frac{1}{22}\sum_{i=1}^{22} \ln RV_{t-i}$$

In [53]:
```
lnRV = np.log(RV)
```

```
lnRV.name = 'lnRV'
lnRV
```

Out[53]:

```
Date
2022-01-18    0.400918
2022-01-19    0.717238
2022-01-20    0.501016
2022-01-21    0.858821
2022-01-24    0.637905
                ...
2022-04-08    1.149792
2022-04-11    0.884927
2022-04-12    0.663464
2022-04-13    0.499602
2022-04-14    0.464434
Name: lnRV, Length: 62, dtype: float64
```

As before, I'll divide the sample into training and test set

In [54]:

```
lnRV_train = lnRV['2022-01-18':'2022-04-11']
lnRV_test = lnRV['2022-04-12':'2022-04-14']
```

Estimating the model using the training set

In [55]:

```
har_ln = HARX(y = lnRV_train,
              lags = [lag_day, lag_week, lag_month],
              rescale = False)

har_ln_fit = har_ln.fit()

har_ln_summary = pd.concat([har_ln_fit.params, har_ln_fit.pvalues], axis = 1)
har_ln_summary.columns = ['Coefficient', 'p-value']
round(har_ln_summary.iloc[:-1, :], 3)
```

Out[55]:

	Coefficient	p-value
Const	1.175	0.033
lnRV[0:1]	0.550	0.000
lnRV[0:5]	0.191	0.339
lnRV[0:22]	-0.507	0.099

Except for the weekly RV, all coefficients are significant with p-value < 0.10.

R^2: HAR-RV vs. HAR-lnRV

In [56]:

```
R2_df = pd.DataFrame({'$R^2$': [har_model_fit.rsquared, har_ln_fit.rsquared]}, index =
R2_df
```

Out[56]:

	R^2
HAR-RV	0.418981
HAR-lnRV	0.520562

You can see that R^2 has improved.

Calculating the Fitted values and the Forecast

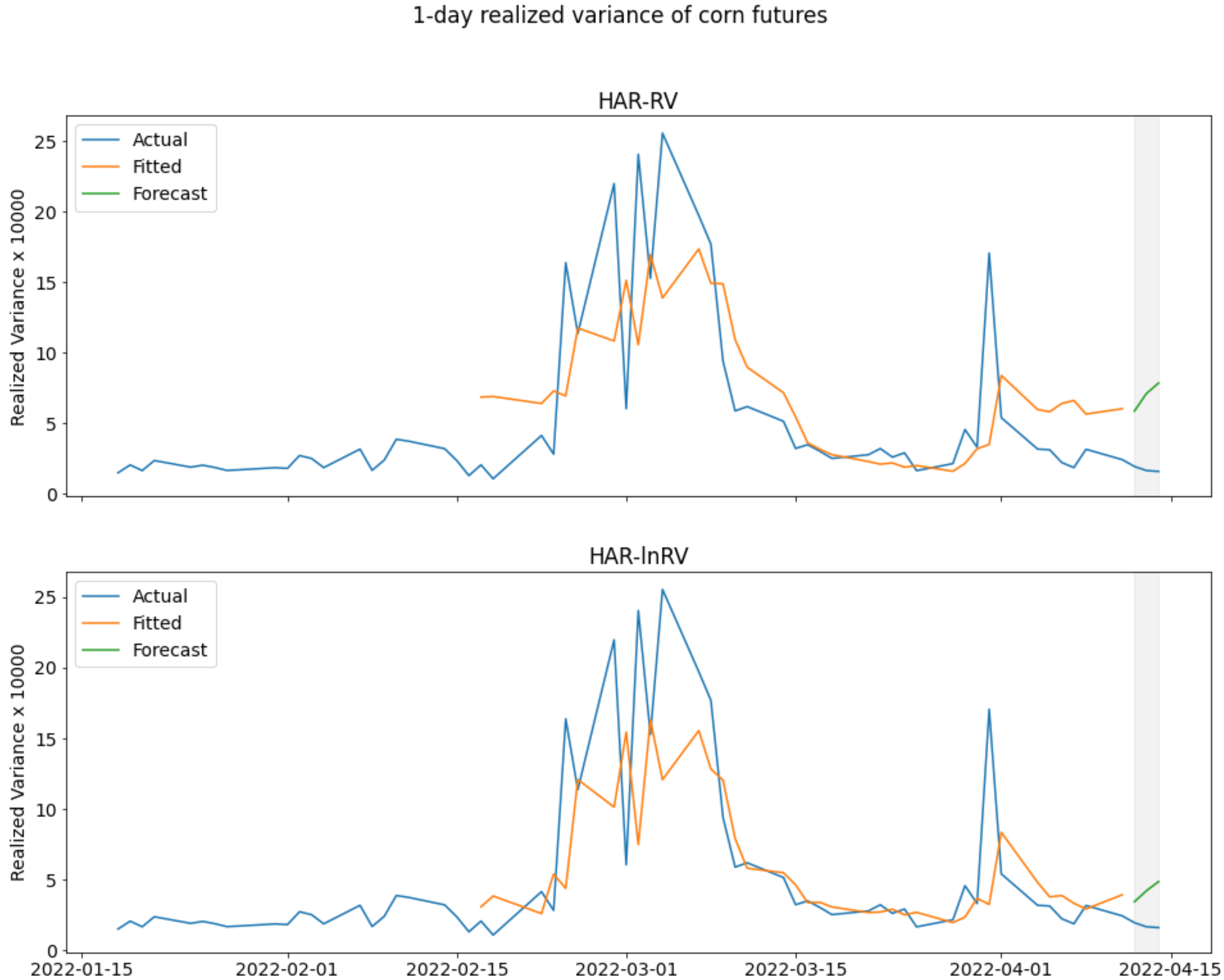
In [57]:
```
# Fitted values
har_ln_fit_RV = np.exp(har_ln.y - har_ln_fit.resid)

# Forecast
har_ln_fc_RV = har_ln_fit.forecast(horizon = 3, reindex = False)
har_ln_fc_RV = np.exp(har_ln_fc_RV.mean).squeeze()
har_ln_fc_RV.index = lnRV_test.index
```

Plotting the (Actual) vs. (Fitted + Forecast)

For comparison, plotting it alongside the plot for HAR-RV

In [58]:
```
fig, axs = plt.subplots(nrows = 2,
                        ncols = 1,
                        sharex = True,
                        sharey = True,
                        figsize = (16, 12))
fig.suptitle("1-day realized variance of corn futures")

# HAR-RV
axs[0].plot(RV, label = "Actual")
axs[0].plot(fitted_RV, label = "Fitted")
axs[0].plot(forecast_RV, label = "Forecast")
axs[0].legend()
axs[0].axvspan(xmin = forecast_RV.index.min(),
               xmax = forecast_RV.index.max(),
               color = 'grey',
               alpha = 0.1)
axs[0].set_title("HAR-RV")
axs[0].set_ylabel("Realized Variance x 10000")

# HAR-lnRV
axs[1].plot(RV, label = "Actual")
axs[1].plot(har_ln_fit_RV, label = "Fitted")
axs[1].plot(har_ln_fc_RV, label = "Forecast")
axs[1].legend()
axs[1].axvspan(xmin = har_ln_fc_RV.index.min(),
               xmax = har_ln_fc_RV.index.max(),
               color = 'grey',
               alpha = 0.1)
axs[1].set_title("HAR-lnRV")
axs[1].set_ylabel("Realized Variance x 10000")
```

Out[58]:
```
Text(0, 0.5, 'Realized Variance x 10000')
```



Calculating the MSEs

MSE of models considered so far

In [59]:
```
mse_df
```

Out[59]:

	Fit	Forecast
HAR	28.68	28.12
GARCH(1, 1)	39.18	1.78

I'll change the name 'HAR' to 'HAR-RV'

In [60]:
```
mse_df = mse_df.rename({'HAR': 'HAR-RV'})
mse_df
```

Out[60]:

	Fit	Forecast
HAR-RV	28.68	28.12
GARCH(1, 1)	39.18	1.78

Calculating the MSE of HAR-lnRV

In [61]:
```
# MSE of fit
mse_har_ln_fit = np.mean(np.square(train_set - har_ln_fit_RV))

# MSE of forecast
mse_har_ln_fc = np.mean(np.square(test_set - har_ln_fc_RV))

# Adding the data to our MSE data frame
mse_df.loc['HAR-lnRV', :] = [mse_har_ln_fit, mse_har_ln_fc]
mse_df
```

Out[61]:

	Fit	Forecast
HAR-RV	28.680000	28.120000
GARCH(1, 1)	39.180000	1.780000
HAR-lnRV	30.270477	6.449163

Fit has worsened slightly, but forecast accuracy has improved drammatically.

HAR-lnRV with 5-min returns

Our minutely prices data was stored in the variable 'price_5' after all the preprocessing.

In [62]:
```
price_5
```

Out[62]:
```
Date
2022-01-18 01:01:00    590.00
2022-01-18 01:02:00    590.00
2022-01-18 01:03:00    589.75
2022-01-18 01:04:00    590.50
2022-01-18 01:05:00    591.00
                        ...
2022-04-14 19:16:00    783.75
2022-04-14 19:17:00    784.00
2022-04-14 19:18:00    784.25
2022-04-14 19:19:00    784.25
2022-04-14 19:20:00    784.50
Name: Price, Length: 68200, dtype: float64
```

To calculate the 5-min returns, I'll resample the data at '5 min' frequency

In [63]:
```
price_5m = price_5.resample('5 min').last().dropna()
price_5m
```

Out[63]:
```
Date
2022-01-18 01:00:00    590.50
2022-01-18 01:05:00    591.50
2022-01-18 01:10:00    591.50
2022-01-18 01:15:00    591.25
2022-01-18 01:20:00    591.75
                        ...
2022-04-14 19:00:00    783.75
2022-04-14 19:05:00    784.25
2022-04-14 19:10:00    783.75
2022-04-14 19:15:00    784.25
2022-04-14 19:20:00    784.50
Name: Price, Length: 13702, dtype: float64
```

Calculating the 5-min returns

In [64]:
```
r_5m = np.log(price_5m).diff().dropna() * 100
r_5m
```

Out[64]:
```
Date
2022-01-18 01:05:00    0.169205
2022-01-18 01:10:00    0.000000
2022-01-18 01:15:00   -0.042274
2022-01-18 01:20:00    0.084531
2022-01-18 01:25:00    0.042239
                         ...
2022-04-14 19:00:00   -0.127510
2022-04-14 19:05:00    0.063776
2022-04-14 19:10:00   -0.063776
2022-04-14 19:15:00    0.063776
2022-04-14 19:20:00    0.031873
Name: Price, Length: 13701, dtype: float64
```

Calculating squared returns

In [65]:
```
r2_5m = np.square(r_5m)
r2_5m
```

Out[65]:
```
Date
2022-01-18 01:05:00    0.028630
2022-01-18 01:10:00    0.000000
2022-01-18 01:15:00    0.001787
2022-01-18 01:20:00    0.007145
2022-01-18 01:25:00    0.001784
                         ...
2022-04-14 19:00:00    0.016259
2022-04-14 19:05:00    0.004067
2022-04-14 19:10:00    0.004067
2022-04-14 19:15:00    0.004067
2022-04-14 19:20:00    0.001016
Name: Price, Length: 13701, dtype: float64
```

Calculating the new RVs

Recall that daily RV is calculated as the sum of squared intraday returns.

In [66]:
```
RV_5m = r2_5m.resample('B').sum().dropna()
RV_5m = RV_5m[dates].copy()
RV_5m
```

Out[66]:
```
Date
2022-01-18    1.221280
2022-01-19    1.759150
2022-01-20    1.288703
2022-01-21    2.393059
2022-01-24    1.762816
                ...
2022-04-08    2.801363
2022-04-11    2.287610
2022-04-12    1.649794
2022-04-13    1.212380
2022-04-14    1.758028
Name: Price, Length: 62, dtype: float64
```

We now have the new RVs calculated using 5-min returns instead of minutely returns. Now, we just need to repeat what we did for HAR-lnRV

In [67]:
```
lnRV_5m = np.log(RV_5m)
lnRV_5m.name = 'lnRV'
lnRV_5m
```

Out[67]:
```
Date
2022-01-18    0.199900
2022-01-19    0.564831
2022-01-20    0.253637
2022-01-21    0.872572
2022-01-24    0.566913
                ...
2022-04-08    1.030106
2022-04-11    0.827507
2022-04-12    0.500650
2022-04-13    0.192586
2022-04-14    0.564193
Name: lnRV, Length: 62, dtype: float64
```

Divide the sample into training and test set

In [68]:
```
lnRV_5m_train = lnRV_5m['2022-01-18':'2022-04-11']
lnRV_5m_test = lnRV_5m['2022-04-12':'2022-04-14']
```

Estimating the model using the training set

In [69]:
```
har_ln_5m = HARX(y = lnRV_5m_train,
                 lags = [lag_day, lag_week, lag_month],
                 rescale = False)

har_ln_5m_fit = har_ln_5m.fit()

har_ln_5m_summary = pd.concat([har_ln_5m_fit.params, har_ln_5m_fit.pvalues], axis = 1)
har_ln_5m_summary.columns = ['Coefficient', 'p-value']
round(har_ln_5m_summary.iloc[:-1, :], 3)
```

Out[69]:

	Coefficient	p-value
Const	1.241	0.028
lnRV[0:1]	0.406	0.001
lnRV[0:5]	0.308	0.122
lnRV[0:22]	-0.585	0.068

All coefficients are significant with p-value < 0.13. This is an improvement on both the HAR models considered so far.

R^2

In [70]:
```
R2_df.loc['HAR-lnRV_5m', :] = har_ln_5m_fit.rsquared
R2_df
```

Out[70]:

	R^2
HAR-RV	0.418981
HAR-lnRV	0.520562
HAR-lnRV_5m	0.448342

Calculating the Fitted values and the Forecast

In [71]:
```
# Fitted values
har_ln_5m_fit_RV = np.exp(har_ln_5m.y - har_ln_5m_fit.resid)

# Forecast
har_ln_5m_fc_RV = har_ln_5m_fit.forecast(horizon = 3, reindex = False)
har_ln_5m_fc_RV = np.exp(har_ln_5m_fc_RV.mean).squeeze()
har_ln_5m_fc_RV.index = lnRV_5m_test.index
```

Plotting the (Actual) vs. (Fitted + Forecast)

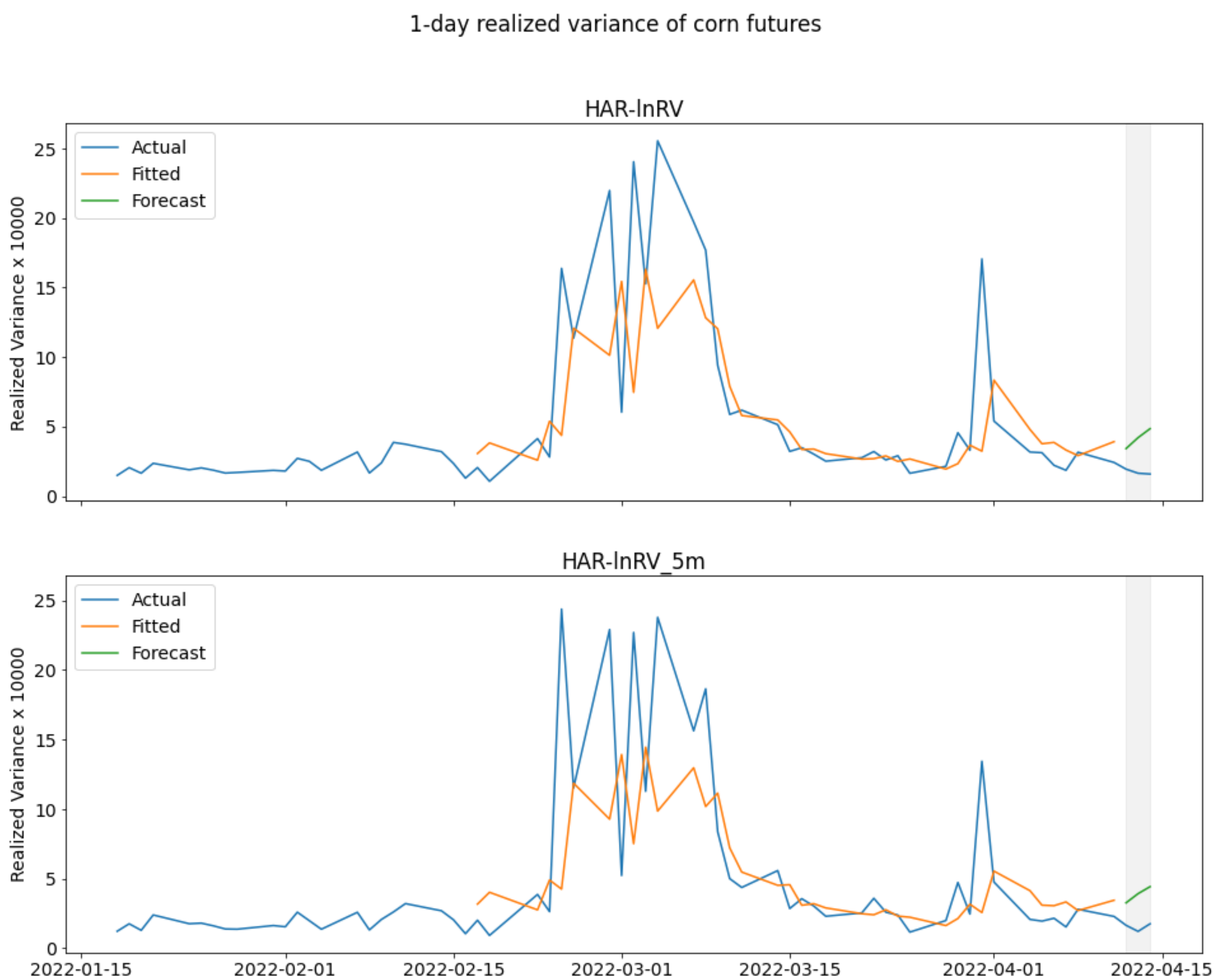
Plotting it alongside the plot for HAR-lnRV. But note that the RV calculated from 5-min returns is different from the RV calculated from the minutely returns. So, basically, the actual RVs are different for the two models. Hence only the gap between actual and fitted/forecast is to be compared between the two plots.

In [72]:
```
fig, axs = plt.subplots(nrows = 2,
                        ncols = 1,
                        sharex = True,
                        sharey = True,
                        figsize = (16, 12))
fig.suptitle("1-day realized variance of corn futures")

# HAR-lnRV
axs[0].plot(RV, label = "Actual")
axs[0].plot(har_ln_fit_RV, label = "Fitted")
axs[0].plot(har_ln_fc_RV, label = "Forecast")
axs[0].legend()
```

```
axs[0].axvspan(xmin = har_ln_fc_RV.index.min(),
               xmax = har_ln_fc_RV.index.max(),
               color = 'grey',
               alpha = 0.1)
axs[0].set_title("HAR-lnRV")
axs[0].set_ylabel("Realized Variance x 10000")

# HAR-LnRV_5m
axs[1].plot(RV_5m, label = "Actual")
axs[1].plot(har_ln_5m_fit_RV, label = "Fitted")
axs[1].plot(har_ln_5m_fc_RV, label = "Forecast")
axs[1].legend()
axs[1].axvspan(xmin = har_ln_5m_fc_RV.index.min(),
               xmax = har_ln_5m_fc_RV.index.max(),
               color = 'grey',
               alpha = 0.1)
axs[1].set_title("HAR-lnRV_5m")
axs[1].set_ylabel("Realized Variance x 10000")
```

Out[72]:

```
Text(0, 0.5, 'Realized Variance x 10000')
```



Calculating the MSEs

MSE of models considered so far

In [73]:

```
mse_df
```

Out[73]:

	Fit	Forecast
HAR-RV	28.680000	28.120000
GARCH(1, 1)	39.180000	1.780000
HAR-lnRV	30.270477	6.449163

Calculating the MSE of HAR-lnRV_5m

In [74]:

```
# MSE of fit
mse_har_ln_5m_fit = np.mean(np.square(train_set - har_ln_5m_fit_RV))

# MSE of forecast
mse_har_ln_5m_fc = np.mean(np.square(test_set - har_ln_5m_fc_RV))

# Adding the data to our MSE data frame
mse_df.loc['HAR-lnRV_5m', :] = [mse_har_ln_5m_fit, mse_har_ln_5m_fc]
mse_df
```

Out[74]:

	Fit	Forecast
HAR-RV	28.680000	28.120000
GARCH(1, 1)	39.180000	1.780000
HAR-lnRV	30.270477	6.449163
HAR-lnRV_5m	33.481681	4.992042

Fit has worsened a little more, but forecast accuracy has improved.

HAR-lnRV with 5-min returns and volume regressor

With volume as a regressor, the model changes to:

$$\ln RV_t = \beta_0 + \beta_1 \ln RV^d_{t-1} + \beta_2 \ln RV^w_{t-1} + \beta_3 \ln RV^m_{t-1} + VOLUME_{t-1} + u_t$$

Since we're modelling daily RV, we only need daily volume. I'll have to reimport 'CN2DAY.csv' as last time I only used the 'CLOSE' column.

In [75]:

```
volume = pd.read_csv("CN2DAY.csv",
                     usecols = ["Date", "VOLUME"],
                     index_col = 0,
                     parse_dates = True).squeeze()

volume.name = 'Volume'

volume = volume/10000 # in 10000s
volume
```

Out[75]:
```
Date
2022-01-18     3.1188
2022-01-19     4.4174
2022-01-20     2.6113
2022-01-21     4.1600
2022-01-24     3.8992
               ...
2022-04-08    15.5533
2022-04-11    15.8908
2022-04-12    15.3025
2022-04-13    18.4615
2022-04-14    11.4603
Name: Volume, Length: 62, dtype: float64
```

Lagging the volume variable

In [76]:
```
volume = volume.shift(1).copy()
volume
```

Out[76]:
```
Date
2022-01-18        NaN
2022-01-19     3.1188
2022-01-20     4.4174
2022-01-21     2.6113
2022-01-24     4.1600
               ...
2022-04-08    14.4673
2022-04-11    15.5533
2022-04-12    15.8908
2022-04-13    15.3025
2022-04-14    18.4615
Name: Volume, Length: 62, dtype: float64
```

Dividing into training and testing set

In [77]:
```
volume_train = volume['2022-01-18':'2022-04-11']
volume_test = volume['2022-04-12':'2022-04-14']
```

Estimating the model using the training set. (We are building the last model i.e., HAR-lnRV_5m, but with a volume regressor. We can use the previously defined lnRV_5m variable and it's training and test set.)

In [78]:
```
har_vo = HARX(y = lnRV_5m_train,
              x = volume_train,
              lags = [lag_day, lag_week, lag_month],
              rescale = False)

har_vo_fit = har_vo.fit()

har_vo_summary = pd.concat([har_vo_fit.params, har_vo_fit.pvalues], axis = 1)
har_vo_summary.columns = ['Coefficient', 'p-value']
round(har_vo_summary.iloc[:-1, :], 3)
```

Out[78]:

	Coefficient	p-value
Const	1.191	0.117
lnRV[0:1]	0.390	0.030
lnRV[0:5]	0.313	0.147
lnRV[0:22]	-0.575	0.096
Volume	0.006	0.872

Alas, the volume variable is insignificant.

R^2

In [79]:
```
R2_df.loc['HAR-lnRV_5m_volume', :] = har_vo_fit.rsquared
R2_df
```

Out[79]:

	R^2
HAR-RV	0.418981
HAR-lnRV	0.520562
HAR-lnRV_5m	0.448342
HAR-lnRV_5m_volume	0.448704

You can see that R^2 has barely changed.

Calculating the Fitted values and the Forecast

In [80]:
```
# Fitted values
har_vo_fit_RV = np.exp(har_vo.y - har_vo_fit.resid)

# Forecast
har_vo_fc_RV = har_vo_fit.forecast(x = volume_test, horizon = 3, reindex = False)
har_vo_fc_RV = np.exp(har_vo_fc_RV.mean).squeeze()
har_vo_fc_RV.index = lnRV_5m_test.index
```

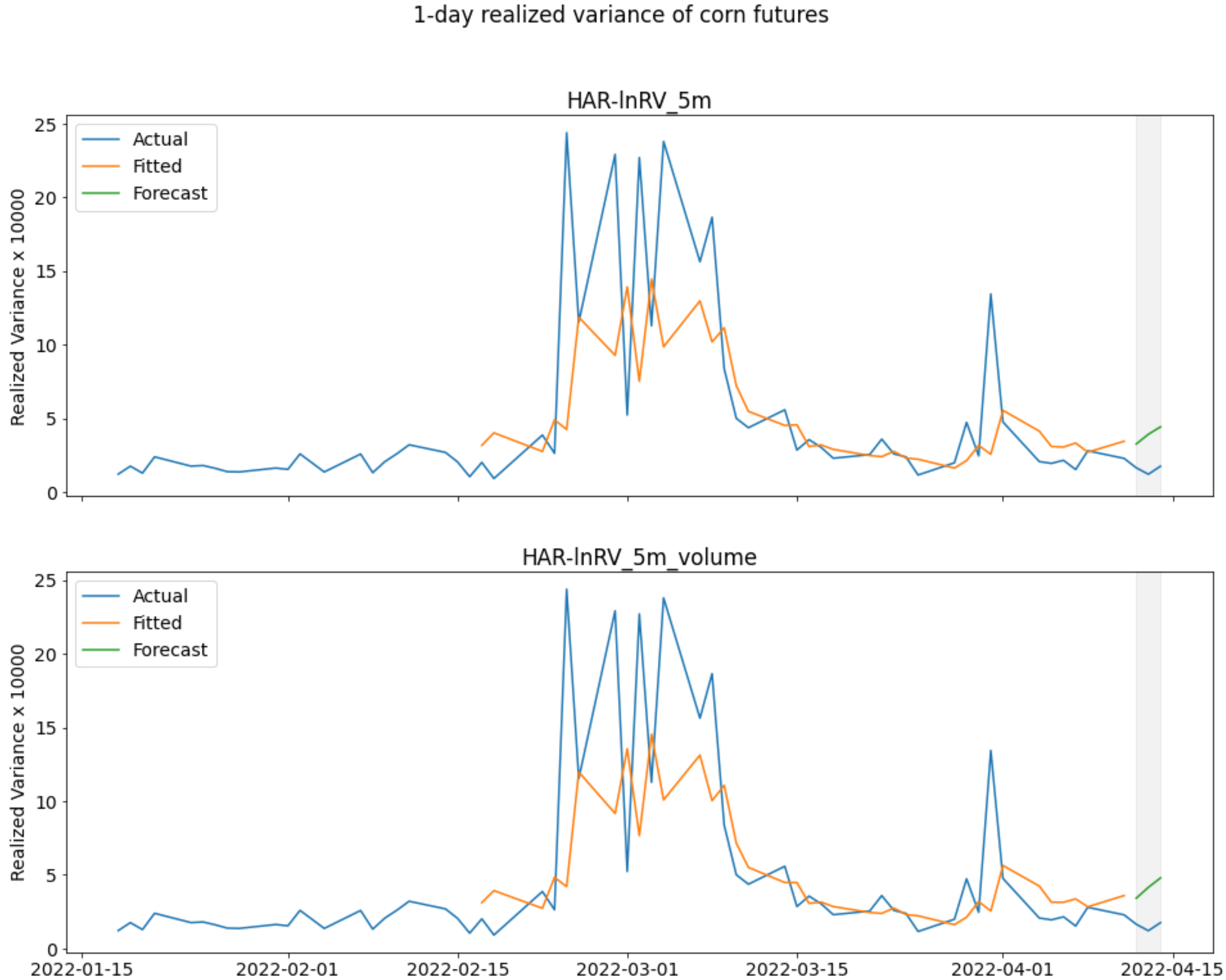
Plotting the (Actual) vs. (Fitted + Forecast)

Plotting it alongside the plot for HAR-lnRV_5m.

In [81]:
```
fig, axs = plt.subplots(nrows = 2,
                        ncols = 1,
                        sharex = True,
                        sharey = True,
                        figsize = (16, 12))
fig.suptitle("1-day realized variance of corn futures")

# HAR-lnRV_5m
axs[0].plot(RV_5m, label = "Actual")
axs[0].plot(har_ln_5m_fit_RV, label = "Fitted")
axs[0].plot(har_ln_5m_fc_RV, label = "Forecast")
axs[0].legend()
```

```
axs[0].axvspan(xmin = har_ln_5m_fc_RV.index.min(),
               xmax = har_ln_5m_fc_RV.index.max(),
               color = 'grey',
               alpha = 0.1)
axs[0].set_title("HAR-lnRV_5m")
axs[0].set_ylabel("Realized Variance x 10000")

# HAR-lnRV_5m_volume
axs[1].plot(RV_5m, label = "Actual")
axs[1].plot(har_vo_fit_RV, label = "Fitted")
axs[1].plot(har_vo_fc_RV, label = "Forecast")
axs[1].legend()
axs[1].axvspan(xmin = har_vo_fc_RV.index.min(),
               xmax = har_vo_fc_RV.index.max(),
               color = 'grey',
               alpha = 0.1)
axs[1].set_title("HAR-lnRV_5m_volume")
axs[1].set_ylabel("Realized Variance x 10000")
```

Out[81]:
```
Text(0, 0.5, 'Realized Variance x 10000')
```



The plots look identical as well.

Calculating the MSEs

MSE of models considered so far

In [82]:
```
mse_df
```

Out[82]:

	Fit	Forecast
HAR-RV	28.680000	28.120000
GARCH(1, 1)	39.180000	1.780000
HAR-lnRV	30.270477	6.449163
HAR-lnRV_5m	33.481681	4.992042

Calculating the MSE of HAR-lnRV_5m_volume

In [83]:
```
# MSE of fit
mse_har_vo_fit = np.mean(np.square(train_set - har_vo_fit_RV))

# MSE of forecast
mse_har_vo_fc = np.mean(np.square(test_set - har_vo_fc_RV))

# Adding the data to our MSE data frame
mse_df.loc['HAR-lnRV_5m_volume', :] = [mse_har_vo_fit, mse_har_vo_fc]
mse_df
```

Out[83]:

	Fit	Forecast
HAR-RV	28.680000	28.120000
GARCH(1, 1)	39.180000	1.780000
HAR-lnRV	30.270477	6.449163
HAR-lnRV_5m	33.481681	4.992042
HAR-lnRV_5m_volume	33.118641	6.235939

Fit accuracy is nearly the same, but forecast accuracy has worsened.

The result is surprising because volume is linked to volatility, both theoretically and empirically. So, just out of curiosity, I'll run the OLS of RV on volume.

In [84]:
```
from statsmodels.api import OLS, add_constant
```

In [85]:
```
OLS(lnRV_5m, add_constant(volume), missing='drop').fit().summary()
```

Out[85]:

OLS Regression Results

Dep. Variable:	lnRV	**R-squared:**	0.124
Model:	OLS	**Adj. R-squared:**	0.109
Method:	Least Squares	**F-statistic:**	8.342
Date:	Fri, 13 May 2022	**Prob (F-statistic):**	0.00541
Time:	21:18:46	**Log-Likelihood:**	-72.316
No. Observations:	61	**AIC:**	148.6
Df Residuals:	59	**BIC:**	152.9
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.5514	0.225	2.454	0.017	0.102	1.001
Volume	0.0749	0.026	2.888	0.005	0.023	0.127

Omnibus:	6.677	**Durbin-Watson:**	0.850
Prob(Omnibus):	0.035	**Jarque-Bera (JB):**	6.045
Skew:	0.756	**Prob(JB):**	0.0487
Kurtosis:	3.302	**Cond. No.**	19.1

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

You can see that the coefficient of volume is highly significant (p-value = 0.005). So, I guess what is happening in the HAR-X model is that the effects of lagged volume are already captured in the lagged realized variances. So, adding the lagged volume has no effect.

Note also that although the coefficient of volume is significant, R^2 is only 0.124, which says that it contributes very little in explaining the variation in RV.

Summary comments

I would like to reiterate that the significance of the results of our analysis is limited by the size of our data set. Since we can't get more minutely or hourly data, as discussed on chat, we could try the approximation for daily RV using only intraday high and low. [Page 4 in "2019 A Practical Guide to Harnessing the HAR Volatility Model", under "2.1.2 Logarithmic range"]