Predictive Analysis with Intel Limit Order Book Data

By: Jovan Krcadinac
STAT 430 - Spring 2019

Date: 9 May 2019

In this project, I attempt to predict the price movement based on imbalances between the buy and the sell volumes, using different CNN and RNN models to identify which patterns have predictive power.

Introduction:

A Limit Order Book, or LOB, is a trading method used by traders world wide. It is a transparent system that matches customer orders based on a price time priority basis. It consists of different levels of bid/ asking prices along with the bid/asking volumes. It also includes the highest, 'best', bid order and the lowest, 'cheapest', offer which constitutes the best market in a given security or swamp contract. Customers can routinely cross the bid/ask price spread to make an effective, low cost execution. Traders can also see market depth, or stack, in which customers can view bid orders for various prices and volumes on one side versus viewing offers at various volumes and prices on the other side of the transaction. The LOB is by definition fully transparent, real-time, anonymous and low cost in execution which makes it a great tool for traders.

Data Description:

The dataset was found on Lobster. The Intel limit order book data that Lobster reconstructed originates from NASDAQ's historical data. It contains 10 levels of ask price, bid price, ask volume, and bid volume. This led to 624,040 observations and 40 variables with no missing observations.

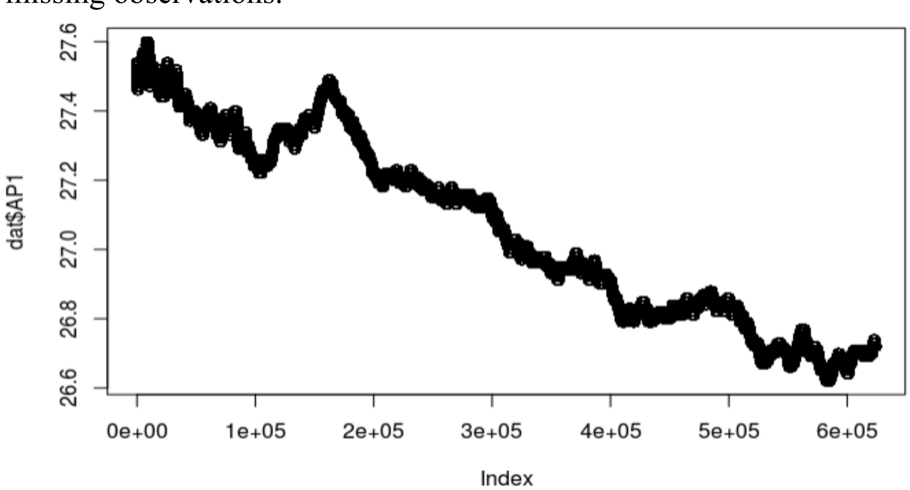
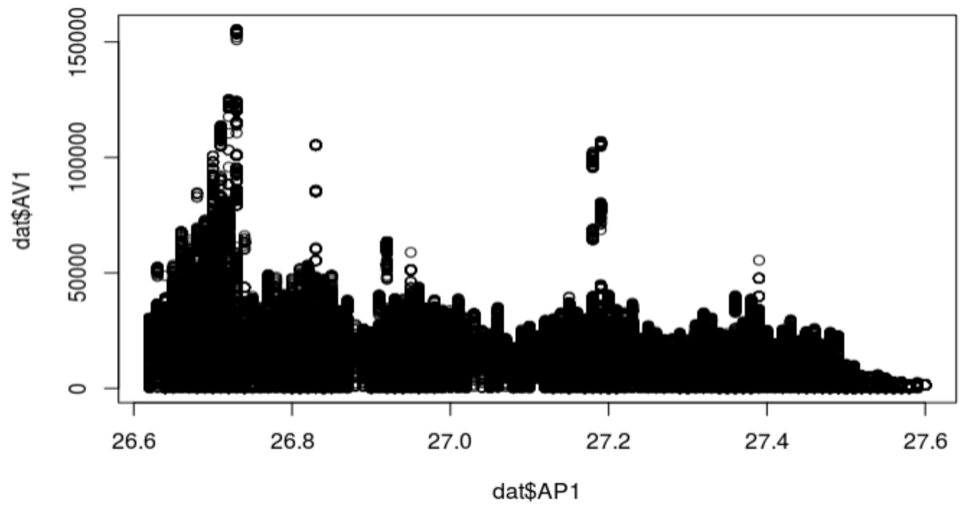


FIG 1: The 'best' asking price

FIG 2: ‘Best’ asking price in comparison to the asking volume



Labeling:

Since the goal of the project is to predict the price movement, I needed to label the data based on on the mid-price. In order to do this, I used the previous 100 mid-price level for the previous labels as well as the future 100 mid-price level for the future price labels. This was accomplished by using a rollover mean in the ‘zoo’ package. A threshold of 0.00005 was used for labeling the direction. I used -1 to detect no change, 0 to detect a decrease, and 1 to detect an increase. After labeling the direction, I split my data into 3 categories using a 3-1-1 ratio. This split was stored as my training, test, and validation data respectively. Below are the sums of the labeled observations before and after I split them. Note the first 99 observations do not have labels.

```
    -1      0      1
376263 129024 118554
```

FIG3: Sums of the labels before the split

```
    -1      0      1
208648  86274  79382

   -1     0     1
84770 20495 19503

   -1     0     1
82845 22255 19669
```

FIG4: Sums of the labels after the split

Features:

In order to create my feature matrix, I used only the direction and volumes as my features, since we are only concerned with the imbalance between asking and bidding volumes. The feature matrix made it easier for me to access the features because the data set was so large and because the data was labeled.

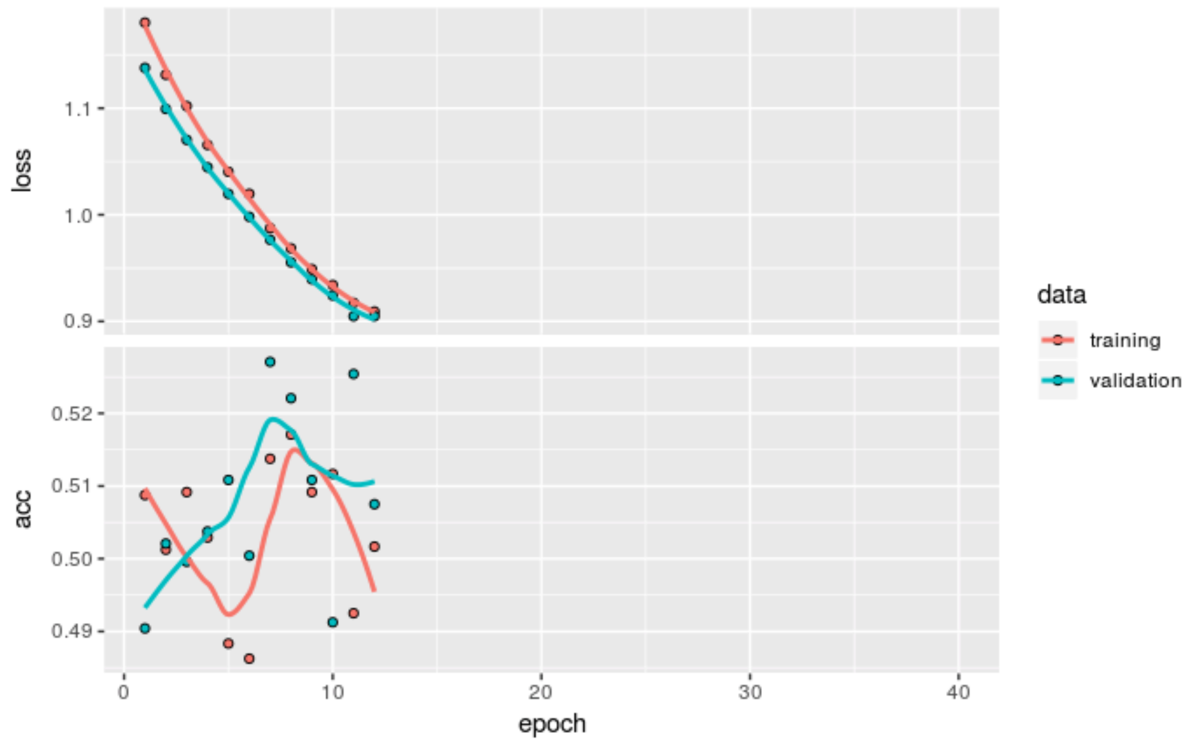
Analysis:

In order to compare my models, I used the given Keras function, binary crossentropy, for my loss function to measure the accuracy of my models. I also used multiple callback functions to make sure I don't waste any unnecessary time and to save the best results. The callback functions are listed below:

1.) Early stop – Interrupts the training when the validation accuracy has been improving for more than 5 epochs
2.) Check point – Makes sure I do not overwrite the model file unless validation loss has improved.
3.) Reduce Lr – The function is triggered after the validation accuracy has been improving for more than 4 epochs. Then the learning rate is reduced to lr * 0.1
4.) Logger – This function saves the best result in another csv file in my work folder

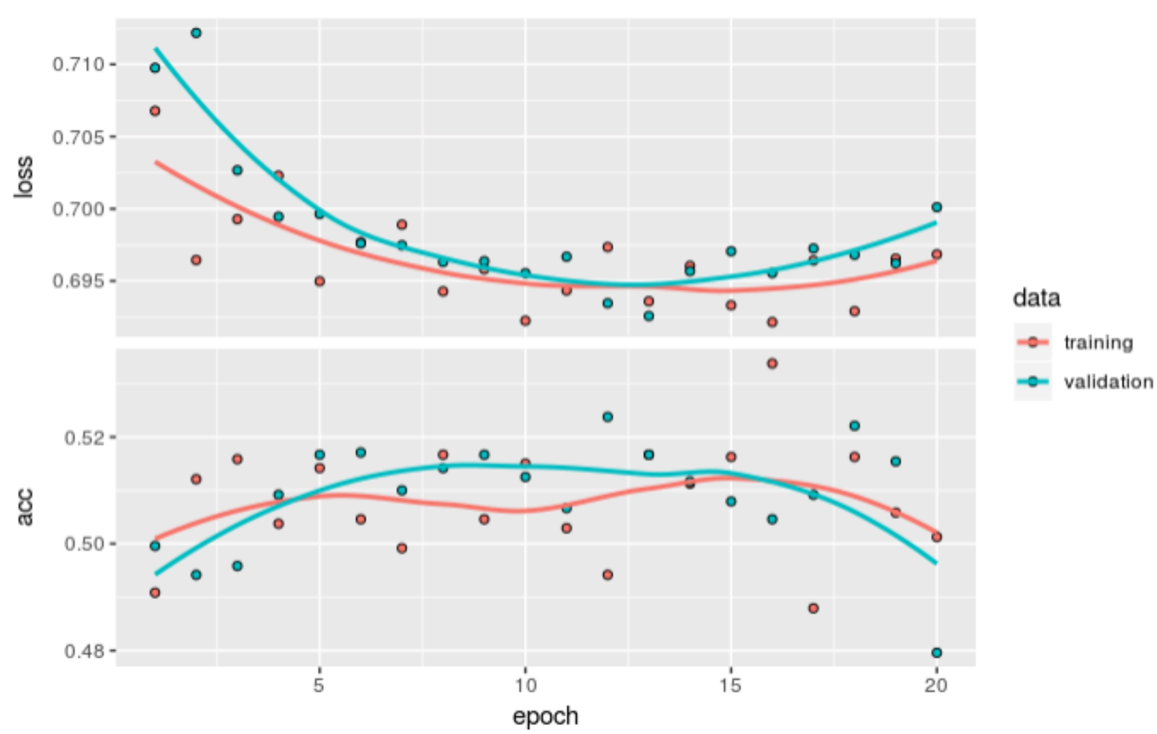
The first model, Model_A, I ran was a CNN model with dense layers. Model_A had 40 epochs with 100 steps per epoch. For this model, one of the callback functions was triggered around the 12th epoch. This model resulted in a good loss result, however after viewing the accuracy graph, we can see that the model might be over-fitting/ under-fitting the data. Below is the plotted results of my model.

FIG5: Model_A results



The second model, Model_B, was an RNN model with dense layers and a feature dimension of 20. This lead to an epoch size of 20 with 100 steps per epoch. This model yielded disappointing results since the RNN model had a lower accuracy and loss. The RNN works on the principle of saving the output of a layer and feeding it back into the input in order to predict the output of the layer. However, this model yielded in little to no improvement in comparison of the first model. No callbacks were triggered.

FIG6: Model_B results



The final model I ran, Model_C was a 1 dimension CNN and RNN model with a feature dimension of 20. This model had the lowest accuracy, however it seemed to not over-fit the data. A callback function was triggered around the 10th epoch.

FIG7: Model_C results

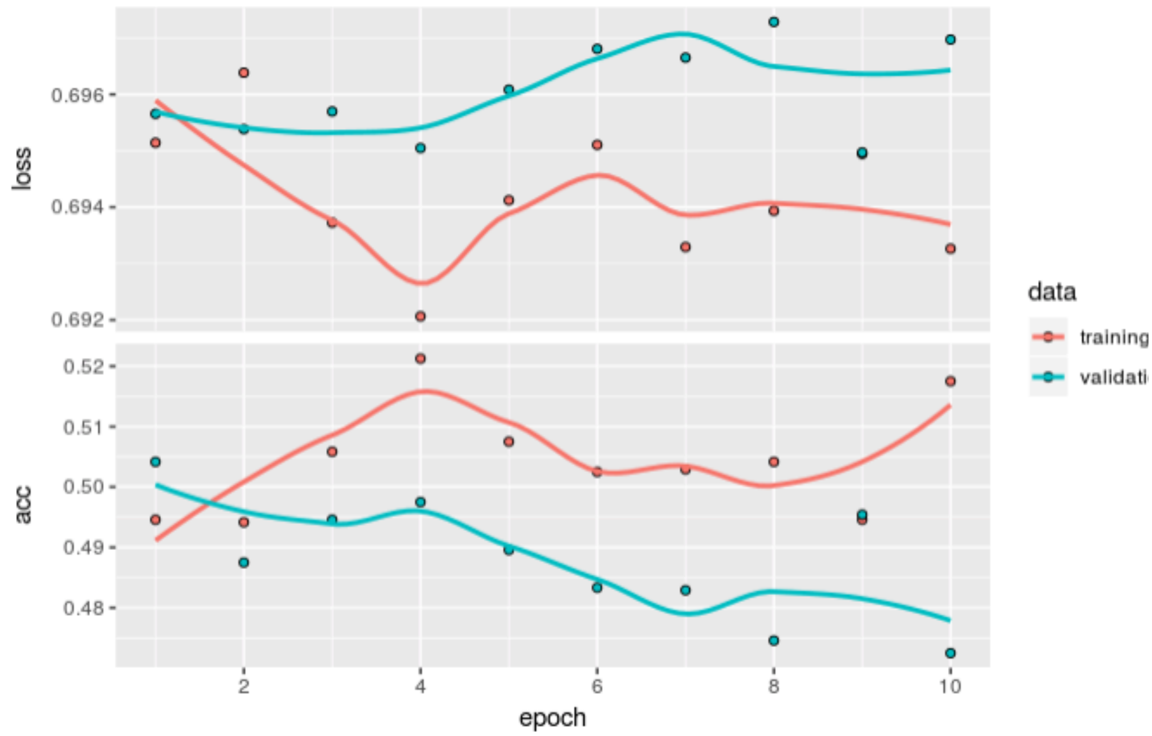


FIG8: Comparison of all the model summaries in order: Model_A, Model_B, Model_C

A.)

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 98, 18, 6)	60
max_pooling2d (MaxPooling2D)	(None, 49, 9, 6)	0
conv2d_1 (Conv2D)	(None, 47, 7, 8)	440
max_pooling2d_1 (MaxPooling2D)	(None, 23, 3, 8)	0
flatten (Flatten)	(None, 552)	0
dropout (Dropout)	(None, 552)	0
dense (Dense)	(None, 16)	8848
dense_1 (Dense)	(None, 1)	17

Total params: 9,365
Trainable params: 9,365
Non-trainable params: 0

B.)

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
cu_dnngru (CuDNNGRU)         (None, None, 8)           1200
_________________________________________________________________
cu_dnngru_1 (CuDNNGRU)       (None, 16)                1248
_________________________________________________________________
dense (Dense)                (None, 1)                 17
=================================================================
Total params: 2,465
Trainable params: 2,465
Non-trainable params: 0
_________________________________________________________________
```

C.)

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d (Conv1D)              (None, None, 8)           808
_________________________________________________________________
cu_dnngru (CuDNNGRU)         (None, None, 8)           432
_________________________________________________________________
cu_dnngru_1 (CuDNNGRU)       (None, 16)                1248
_________________________________________________________________
dense (Dense)                (None, 1)                 17
=================================================================
Total params: 2,505
Trainable params: 2,505
Non-trainable params: 0
_________________________________________________________________
```

Conclusion:

After running and comparing my three models, it seemed like Model_C yielded the best results. Not only did it have the best accuracy, but also it did not tend to over fit the data. In order to get better results from my models, more layers were added to the RNN models. Finally, the comparison of the accuracy and loss of each model are summarized below:

Model	Accuracy	Loss
A	0.5075833	1.08652
B	0.5104583	0.696358
C	0.5248344	0.695375

Appendix:

Data:

https://lobsterdata.com/info/DataSamples.php

Reading:

https://www.investopedia.com/terms/l/limitorderbook.asp

Code:

```
#install_keras(tensorflow = '1.12-gpu')


library(dplyr)
library(keras)
library(zoo)
library(readr)
library(abind)

work_folder = 'work'

dat = read_csv('INTC_2012-06-21_34200000_57600000_orderbook_10.csv', col_names = F)

dat <- apply(dat, 2, as.numeric)
for(i in seq(1,40, 2)) dat[,i] <- dat[,i] / 10000
dat <- data.frame(dat)
names(dat) <- c("AP1", "AV1", "BP1", "BV1", "AP2", "AV2", "BP2", "BV2", "AP3", "AV3", "BP3", "BV3", 
"AP4", "AV4",
            "BP4", "BV4", "AP5", "AV5", "BP5", "BV5","AP6", "AV6", "BP6", "BV6", "AP7", "AV7", "BP7", 
"BV7",
            "AP8", "AV8", "BP8", "BV8", "AP9", "AV9", "BP9", "BV9", "AP10", "AV10", "BP10", "BV10")

nrow(dat)
ncol(dat)

sum(is.na(dat))

# mid price
dat$mPrice <- (dat$AP1 + dat$BP1)/2

w <- 100
avgMprice <- c(rep(NA, w-1), zoo::rollmean(dat$mPrice, k=w, align="left"))
dat1 <- dat[-c(((nrow(dat)-w+1):nrow(dat))), ] # remove last w observations
```

```
dat1$preMP <- avgMprice[1:nrow(dat1)]
dat1$postMP <- avgMprice[(w+1):(nrow(dat1)+w)]
dat1 <- dat1[-(1:(w-1)),] # remove first (w-1) observations
head(dat1)

a <- 0.00005
chg <- dat1$postMP / dat1$preMP - 1


dat1$direction <- -1 # stable
dat1$direction[chg > a] <- 1 # increase
dat1$direction[chg < -a] <- 0 # decrease
table(dat1$direction)

head(dat1)

col_used <- c("direction",
        "AV10", "AV9", "AV8", "AV7", "AV6", "AV5", "AV4", "AV3", "AV2", "AV1",
        "BV1", "BV2", "BV3", "BV4", "BV5", "BV6", "BV7", "BV8", "BV9", "BV10")
dat1 <- dat1[, names(dat1) %in% col_used]
dat1 <- dat1[, sapply(col_used, function(x){which(x==names(dat1))})]
head(dat1)

fMat = as.matrix(dat1)

data_train <- dat1[(1:floor(nrow(dat1)/5*3)),]
data_val <- dat1[((floor(nrow(dat1)/5*3)+1):floor(nrow(dat1)/5*4)),]
data_test <- dat1[((floor(nrow(dat1)/5*4)+1):nrow(dat1)),]
dim(data_train); dim(data_val); dim(data_test); dim(dat1)
table(data_train$direction)
table(data_val$direction)
table(data_test$direction)

col_volume <- (2:21)
up_down_train <- (data_train$direction != -1)
me_volume_train <- mean(as.matrix(data_train[up_down_train, col_volume]))  # <----------------- try log
sd_volume_train <- sd(as.matrix(data_train[up_down_train, col_volume]))

for(i in col_volume) data_train[,i] <- scale(data_train[,i], center = me_volume_train, scale =
sd_volume_train)
X_data_train <- data_train[, col_volume]
Y_data_train <- data_train$direction


for(j in col_volume) data_val[,j] <- scale(data_val[,j], center = me_volume_train, scale =
sd_volume_train)
X_data_val <- data_val[, col_volume]
Y_data_val <- data_val$direction
```

```
for(k in col_volume) data_test[,k] <- scale(data_test[,k], center = me_volume_train, scale =
sd_volume_train)
X_data_test <- data_test[, col_volume]
Y_data_test <- data_test$direction
```

CNN Model

```
k_clear_session()

model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 6, kernel_size = c(3, 3), activation = "relu", input_shape = c(100, 20, 1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 8, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 16, activation = "relu", kernel_regularizer = regularizer_l1(0.001)) %>%
  layer_dense(units = 1, activation = "sigmoid")

summary(model)

model %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(lr = 1e-4),
  metrics = c("accuracy")
)

sampling_generator <- function(X_data, Y_data, batch_size, w)
{
  function()
  {
    rows_with_up_down <- w:nrow(X_data)
    rows_with_up_down <- intersect(rows_with_up_down, which( Y_data %in% c(0,1)))  # only use labels
0 and 1

    rows <- sample( rows_with_up_down, batch_size, replace = TRUE )

    Y <- X <- NULL
    Xlist <- list()
    for(i in rows)
    {
      Xlist[[i]] <- X_data[(i-w+1):i,]
      Y <- c(Y, Y_data[i])
    }
    X <- array(abind::abind(Xlist, along = 0), c(batch_size, w, ncol(X_data), 1))
```

```
   list(X, Y)
  }
}

w = 100
batch_size = 24
epochs = 40
rows_with_up_down_train <- w:nrow(X_data_train)
rows_with_up_down_train <- intersect(rows_with_up_down_train, which( Y_data_train %in% c(0,1)))
sample_size_up_down_train <- length(rows_with_up_down_train)
rows_with_up_down_val <- w:nrow(X_data_val)
rows_with_up_down_val <- intersect(rows_with_up_down_val, which( Y_data_val %in% c(0,1)))
sample_size_up_down_val <- length(rows_with_up_down_val)

# Interrupts training when validation accuracy has stopped improving for more than 5 epoch
earlyStop <- callback_early_stopping(monitor = "val_acc", patience = 5)

# do not overwrite the model file unless val_loss has improved
checkPoint <- callback_model_checkpoint(filepath = file.path(work_folder, "LOB_CNN_INTEL.h5"),
                              monitor = "val_acc", save_best_only = TRUE)

# The callback is triggered after the val_acc has stopped improving for 4 epochs
# Then learning rate is reduced to lr*0.1
reduceLr <- callback_reduce_lr_on_plateau(monitor = "val_acc", factor = 0.1, patience = 4)

# runtime csv loggers
logger <- callback_csv_logger(file.path(work_folder, "LOB_CNN_INTEL_callback.csv"))

#run CNN model w callbacks

his <- model %>% fit_generator(sampling_generator(X_data_train, Y_data_train, batch_size =
batch_size, w=w),
                    steps_per_epoch = 100, epochs = epochs,
                    callbacks = list(logger, earlyStop, checkPoint, reduceLr),
                    validation_data = sampling_generator(X_data_val, Y_data_val, batch_size = batch_size,
                                        w=w), validation_steps = 100)

summary(model)
str(his)
fitted <- load_model_hdf5(file.path(work_folder, "LOB_CNN_INTEL.h5"))

results <- fitted %>% evaluate_generator(sampling_generator(X_data_test, Y_data_test, batch_size =
batch_size,w=w),steps = 1000)

results

plot(his)
```

```
#RNN Model

sampling_generator <- function(X_data, Y_data, batch_size, w)
 {
  function()
  {
   rows_with_up_down <- w:nrow(X_data)
   rows_with_up_down <- intersect(rows_with_up_down, which( Y_data %in% c(0,1)))
   rows <- sample( rows_with_up_down, batch_size, replace = TRUE )

   Y <- X <- NULL
   Xlist <- list()
   for(i in rows)
   {
    Xlist[[i]] <- X_data[(i-w+1):i,]
    Y <- c(Y, Y_data[i])
   }
   X <- array(abind::abind(Xlist, along = 0), c(batch_size, w, ncol(X_data)))
   list(X, Y)
  }
}


 k_clear_session()

 model_B <- keras_model_sequential() %>%
  layer_cudnn_gru(unit=8, input_shape = list(NULL, 20), return_sequences = TRUE) %>%
  layer_cudnn_gru(unit=16) %>%
  layer_dense(units = 1, activation = "sigmoid")
 summary(model_B)

 model_B %>% compile(
   loss = "binary_crossentropy",
   optimizer = optimizer_rmsprop(lr = 1e-4),
   metrics = c('accuracy')
 )

batch_size = 24
 epochs = 20

 his_B <- model_B %>% fit_generator(sampling_generator(X_data_train, Y_data_train, batch_size =
batch_size, w=w),
                steps_per_epoch = 100, epochs = epochs,
                callbacks = list(checkPoint, reduceLr, logger),
                validation_data = sampling_generator(X_data_val, Y_data_val, batch_size
                                  = batch_size, w=w),validation_steps
                                  = 100)
```

```
  plot(his_B)
  fitted_B <- load_model_hdf5(file.path(work_folder, "LOB_CNN_INTEL.h5_B"))

}


results <- model_B %>% evaluate_generator(sampling_generator(X_data_test, Y_data_test, batch_size =
batch_size, w=w), steps = 1000)

results

plot(dat$AP1, dat$AV1)


# 1d CNN and RNN
sampling_generator <- function(X_data, Y_data, batch_size, w)
 {
  function()
  {
   rows_with_up_down <- w:nrow(X_data)
   rows_with_up_down <- intersect(rows_with_up_down, which( Y_data %in% c(0,1)))

   rows <- sample( rows_with_up_down, batch_size, replace = TRUE )

   Y <- X <- NULL
   Xlist <- list()
   for(i in rows)
   {
    Xlist[[i]] <- X_data[(i-w+1):i,]
    Y <- c(Y, Y_data[i])
   }
   X <- array(abind::abind(Xlist, along = 0), c(batch_size, w, ncol(X_data)))
   list(X, Y)
  }
 }

 k_clear_session()
 model_C <- keras_model_sequential() %>%
  layer_conv_1d(filters = 8, kernel_size = 5, activation = "relu",
         input_shape = list(NULL, 20)) %>%
  layer_cudnn_gru(unit=8, return_sequences = TRUE) %>%
  layer_cudnn_gru(unit=16) %>%
  layer_dense(units = 1, activation = "sigmoid")

 summary(model_C)

 model_C %>% compile(
```

```
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(lr = 1e-4),
  metrics = c('accuracy')
 )

 #############
 # run model #
 #############
 batch_size <- 24
 his_C <- model_C %>% fit_generator(sampling_generator(X_data_train, Y_data_train, batch_size =
batch_size, w=w),
                    steps_per_epoch = 100, epochs = 10,
                    callbacks = list(checkPoint, reduceLr, logger),
                    validation_data = sampling_generator(X_data_val, Y_data_val, batch_size =
batch_size, w=w),
                    validation_steps = 100)
 plot(his_C)

results <- model_C %>% evaluate_generator(sampling_generator(X_data_test, Y_data_test, batch_size =
batch_size, w=w), steps = 1000)

results
```