Recurrent Neural Network - Cryptocurrencies - v2 **26-Feb-2020**

G. Belford

Overview

This paper describes the implementation of RNN (Recurrent Neural Network) to predict future price movements of a cryptocurrency based on a sequence. Using 4 crypto/fiat pair historical (last 60 minutes) of 1 minute tick prices and volumes, the RNN predicts if the selected crypto/fiat pair price will rise or fall 3 minutes in the future (classification problem). This is a deep learning exercise using Python, Tensorflow & Keras. Crypto/Fiat pairs used in this analysis are BTC/USD, BCH/USD, ETH/USD & LTC/USD.

Background

RNN is ideal for processing sequence data for predictions (called sequential memory). As viewed in Jupyter Notebook "RNN-Crypto-ver3.ipynb", the program contains the following code segments:

1. Import individual crypto/fiat historical 1minute tick data (csv) & merge
2. Create targets
3. Separate out-of-sample data*
4. Make sequences, balance, normalize & scale the data.
5. Statistics – output allows visual confirmation that data is balanced.
6. Build & train Sequential Model
7. Define 2 Callbacks (Tensorboard, ModelCheckpoint)
8. Tensorboard output files saved down into /logs/ dir.
9. ModelCheckpoint output files saved down to /models/ dir.

Tensorboard GUI: graphic view of real-time training & validation loss/accuracy results.

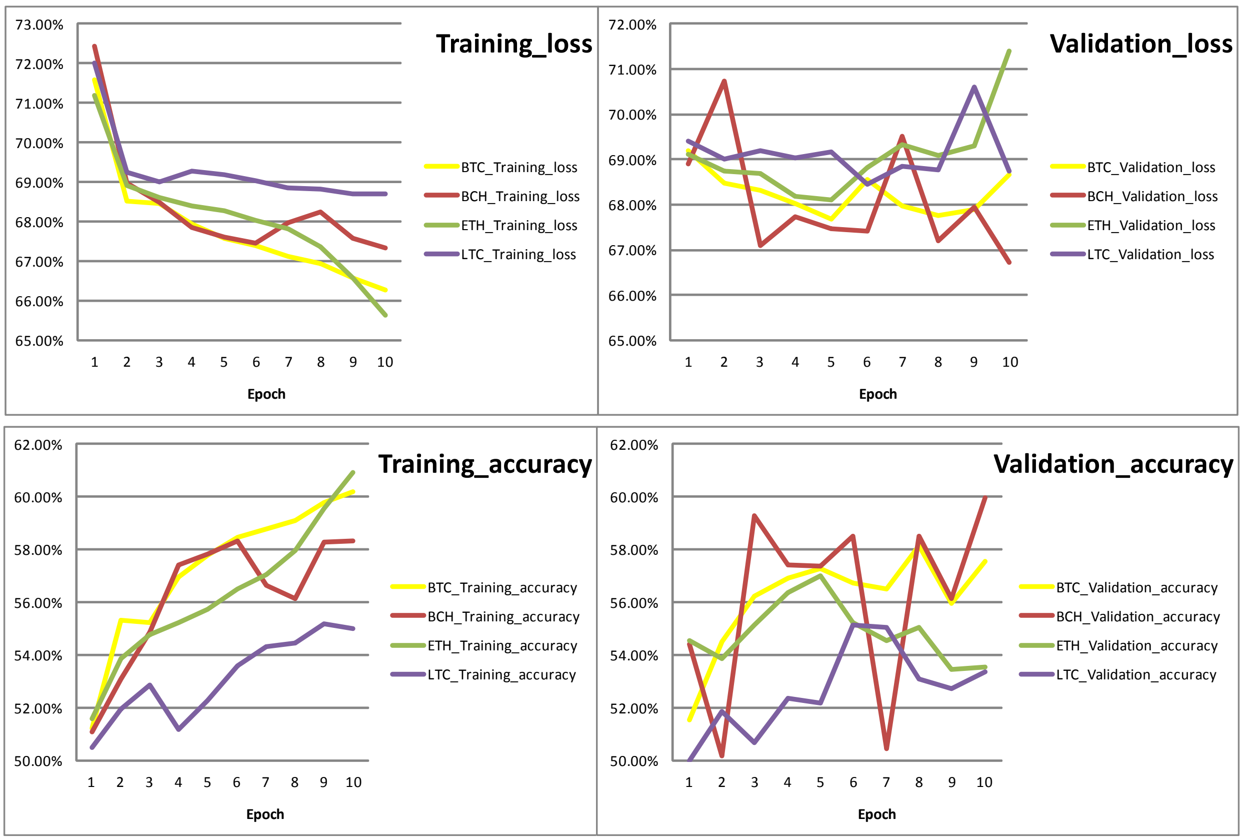
*For temporal (aka time-series or sequential) data – cannot just shuffle & take a random 5% of data as our sequences are 60 minutes long (ie. 60 units in each sequence – predicting out 3 minutes). The problem is if we shuffle the data and take random 5% - our out-of-sample samples would all have very close examples in-sample. It would then be easy for our model as it **overfits** for in-sample data to let this overfitting pour over into **validation** (aka out-of-sample data). Solution: take a segment of the time-series data (ie. last 5% of historical data) and separate this out as our **validation (out-of-sample)** data.

Out-of-sample testing is important as if it is not performed feedback will erroneously comment how great the model fits – if a big enough NN and enough epochs are used you will of course overfit your data to a perfect fit.

Model Results
The goal is to see accuracy increase & losses decrease over the epochs.

BTC-USD
By Epoch # 10 (scale 1-10) (final) Validation_accuracy = 57.57% & Validation_loss = 68.67%. BTC-USD shows promise. Next step: Try & get BTC-USD and other pairs over 60% Validation accuracy (via introduction of other factors).

BCH-USD
Best performer in terms of validation loss & accuracy by final epoch.

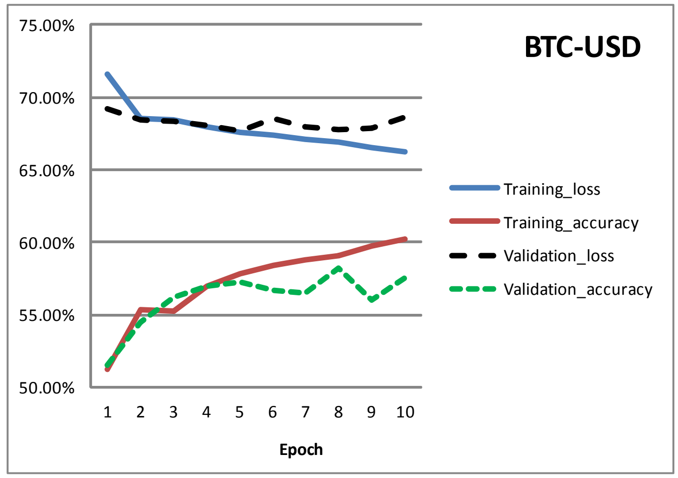


Individual pair results:

SEQ_LEN 60 (minutes)
FUTURE_PERIOD_PREDICT 3 (minutes in future)
RATIO_TO_PREDICT BTC-USD
EPOCHS 10

Training on 74112 samples
Validate on 3592 samples

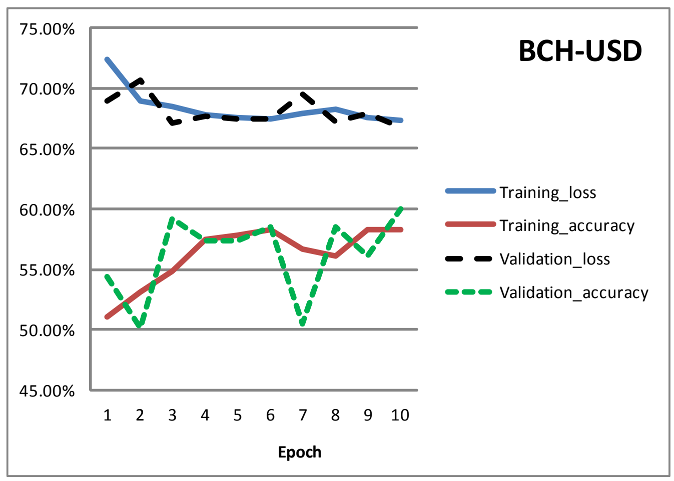
Epoch	Training_ loss	Validation_ loss	Training_ accuracy	Validation_ accuracy
1	71.57%	69.18%	51.24%	51.53%
2	68.51%	68.47%	55.34%	54.48%
3	68.47%	68.31%	55.24%	56.21%
4	67.94%	68.02%	56.96%	56.93%
5	67.58%	67.68%	57.80%	57.29%
6	67.39%	68.56%	58.44%	56.71%
7	67.12%	67.98%	58.80%	56.51%
8	66.93%	67.76%	59.10%	58.16%
9	66.56%	67.88%	59.76%	55.96%
10	66.26%	68.67%	60.17%	57.57%



SEQ_LEN 60 (minutes)
FUTURE_PERIOD_PREDICT 3 (minutes in future)
RATIO_TO_PREDICT BCH-USD
EPOCHS 10

Training on 65728 samples
Validate on 2484 samples 4%

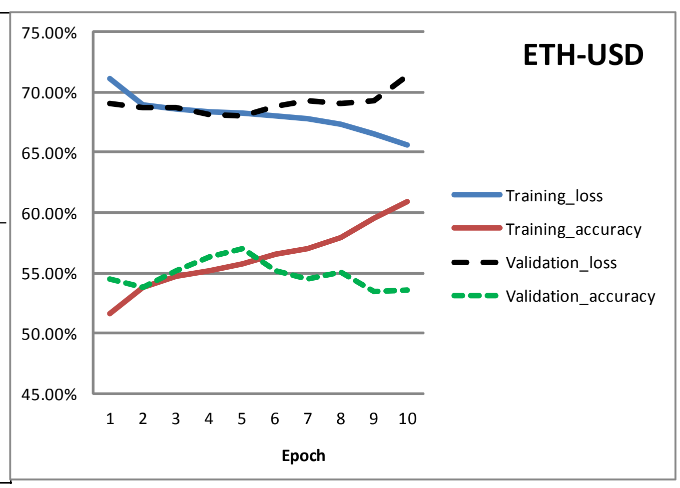
Epoch	Training_ loss	Validation_ loss	Training_ accuracy	Validation_ accuracy
1	72.44%	68.91%	51.09%	54.39%
2	68.96%	70.72%	53.07%	50.16%
3	68.50%	67.10%	54.88%	59.26%
4	67.85%	67.74%	57.43%	57.41%
5	67.60%	67.46%	57.84%	57.37%
6	67.46%	67.42%	58.33%	58.49%
7	67.96%	69.50%	56.66%	50.44%
8	68.25%	67.19%	56.15%	58.49%
9	67.56%	67.95%	58.30%	56.16%
10	67.32%	66.72%	58.32%	59.98%



SEQ_LEN 60 (minutes)
FUTURE_PERIOD_PREDICT 3 (minutes in future)
RATIO_TO_PREDICT ETH-USD
EPOCHS 10

Training on 74196 samples
Validate on 3260 samples

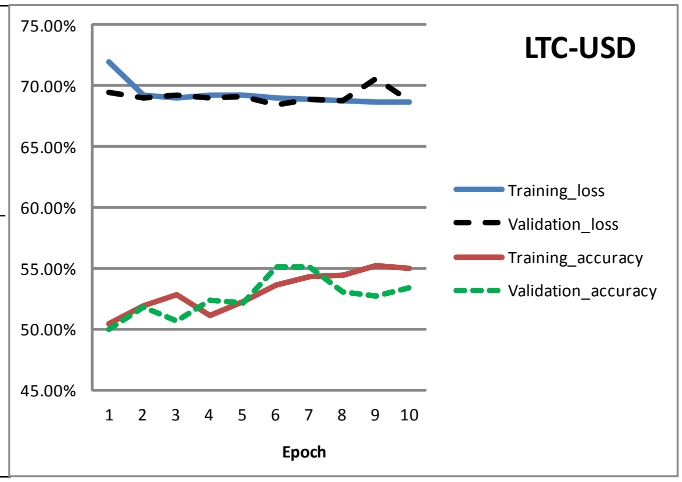
Epoch	Training_ loss	Validation_ loss	Training_ accuracy	Validation_ accuracy
1	71.19%	69.10%	51.60%	54.54%
2	68.91%	68.75%	53.84%	53.87%
3	68.61%	68.69%	54.78%	55.15%
4	68.38%	68.19%	55.24%	56.35%
5	68.26%	68.09%	55.74%	56.99%
6	68.03%	68.82%	56.51%	55.21%
7	67.83%	69.33%	57.03%	54.54%
8	67.35%	69.08%	57.95%	55.06%
9	66.57%	69.29%	59.54%	53.47%
10	65.64%	71.41%	60.92%	53.56%



nb. If see Training_accuracy (in-sample) < Validation_accuracy (out-of-sample) – normally not expected to be the case but may occur if the NN is learning alot per epoch as Training_accuracy is calc'd over the entire epoch run (so weighed down by initial part of epoch run) whereas Validation_accuracy is only calc'd at the end of the epoch run. **ETH-USD** is an interesting example where this behaviour is seen for the 1st 5 epochs before switching to "normal state" for the remaining 5 epochs.

SEQ_LEN	60 (minutes)			
FUTURE_PERIOD_PREDICT	3 (minutes in future)			
RATIO_TO_PREDICT	LTC-USD			
EPOCHS	10			
Training on	69188 samples			
Validate on	3062 samples			

Epoch	Training_ loss	Validation_ loss	Training_ accuracy	Validation_ accuracy
1	72.00%	69.41%	50.50%	49.97%
2	69.24%	69.01%	51.96%	51.86%
3	69.00%	69.19%	52.86%	50.69%
4	69.28%	69.04%	51.16%	52.38%
5	69.19%	69.17%	52.27%	52.19%
6	69.03%	68.45%	53.59%	55.13%
7	68.86%	68.84%	54.34%	55.06%
8	68.82%	68.76%	54.44%	53.10%
9	68.70%	70.59%	55.19%	52.74%
10	68.69%	68.73%	54.98%	53.36%



Viewing Generated Results in Tensorboard
Tensorboard may be used to view the epoch_loss & epoch_accuracy results. Launch (from Anaconda Prompt) & viewing instructions can be found at the bottom of RNN-Crypto-ver3.ipynb file.

Further Research
1. This study has considered a classification problem (will selected Crypto/Fiat pair price rise or fall). Program could be extended to predict a regression (ie. try to predict future price and/or a % change (normalized).
2. Program performance improvement. On GB current maching running 10 epochs takes several hours. GB's computer doesn't have NVIDIA - needed to invoke GPU processes.
3. Test other Crypto/Fiat pairs.
4. Adding additional factors (ex. sentiment) to crypto price/volume to try & improve accuracy/reduce loss. Sentdex notes if you tokenize the words (use a word vector), the sequence of words is input and output is sentiment (RNN for NLP sentiment analysis). Convert sentiment data to numerical values, scale and include in training dataset.
5. For production usage – pipe in real-time 1 minute tick data – deque fn (used in Step5) can keep funnel limited to 60 (current setting) observations. GB starting to look at using model.predict on live data from an exchange. Retrain nightly via a cronjob and update the model to make predictions with.

Document Version History
v2. 26-Feb-2020. Added BCH-USD, ETH-USD & LTC-USD RNN Training/Validation results
v1. 24-Feb-2020.

Reference Files
RNN-Crypto-ver3.ipynb

Source
Sentdex (H. Kulick) youtube videos (publicly available)

Appendix A. Historical Data
Crypto tick data for the 4 pairs used in this analysis is accessible from the following site:
https://pythonprogramming.net/static/downloads/machine-learning-data/crypto_data.zip

Appendix B. Recurrent Neural Network (RNN) – Overview
Traditional Neural Networks (aka Feed Forward NN) have a standard input layer, hidden layer & output layer where the information only flows in forward direction. A challenge which then arises is how to get this framework to use previous info to affect subsequent info. The solution is to add a **loop** in the NN & this is what a RNN does (loops in the hidden middle layer).

Vanishing Gradient
If, for example, we were to feed in the statement ***"What time is it?"*** to the RNN and examine the output, we would notice an odd distribution at the end of the hidden state (ie. the words "What" and "time" would have small slices/impact compared to subsequent words in the above expression. This is an issue with RNN known as *short-term memory* which is caused by the *vanishing gradient problem* (which is also prevalent in other NN architectures). As the RNN processes more steps, it has <u>problems retaining info</u> from the previous steps.

Short-term memory & *vanishing gradient* is due to the nature of *backpropagation* – which is an algo used to train/optimize NN. Let's examine the effects of *backpropagation* on "Deep deep Forward NN". Training this NN involves 3 steps:

> **1/3.** NN performs a forward pass & makes a prediction.
> **2/3.** Using a Loss Function – NN compares prediction vs. the "ground truth". The Loss function outputs an error value (estimate of how badly the NN is performing).
> **3/3.** NN uses Error Value to do *backpropagation* – which calcs gradients for each node in NN. Gradient is a value used to adjust NN internal weights which allow NN to learn. – bigger the gradient – bigger the adjustment.

Here is the problem – when doing *backpropagation* – each node in a layer calcs its gradient with respect to effects of gradients in previous layer. So if the adjustments in earlier layer are small, adjustments in current layer will be even smaller. This causes gradients to exponentially shrink as it backpropagates down – earlier layers fail to do any learning as internal weights are barely being adjusted due to extremely small gradient (*vanishing gradient problem*).

Solutions
Now lets see how above applies to RNN – think of each timestep in RNN as a layer – to train a RNN use application of backpropagation: called "b*ackpropagation through time*" – gradients value expotentially shrink as backpropagates through each timestep. As noted above, due to *vanishing gradient* – the RNN doesn't learn long-range dependencies across timesteps (ie. so terms "What" & "time" are not considered when tyring to predict users intention. NN has to make best guess with "is", "it" – difficult. So not being able to learn on early timesteps causes RNN to have short-term memory. To combat this problem:

- **Solution1. RNN-LSTM** – capable of learning long-term dependencies using mechanism called gates.
- **Solution2. RNN-GRU** – gates are different tensor operations that can learn what info to add/remove to hidden state.

Appendix C. Supervised Machine Learning (ML)
In this RNN cryptocurrency analysis we are solving a classification problem which fits under Supervised ML (**Classification**/Regression).

A. **Classification ML Algos:** Output variable (upward/downward px *movement)* is categorical (or discrete). Typically use a rough rule of thumb where the threshold results need to be =>60% to classify as a viable signal. Examples include logistic regression, naive bayes, decision trees, K nearest neighbours.
B. **Regression ML Algos:** output variable is *numerical (or continuous)*. Examples include Linear Regression, SVR, Regression Trees. Could look at modifying .ipynb to consider a regression problem (predict px or a % px change).

Appendix D. RNN-Crypto-ver3.ipynb

```
import numpy as np
import pandas as pd
import datetime
import os
import matplotlib.pyplot as plt
import sklearn
from sklearn import preprocessing
from collections import deque
import random
import time
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, LSTM, BatchNormalization

# CuDNNLSTM: Fast LSTM implementation backed by CuDNN. Can only be run on GPU, w/ TensorFlow backend.
# Error was because from TensorFlow 2 you do not need to specify CuDNNLSTM
# Use LSTM with no Activation Function & it will automatically use CuDNN version
```

 # Historical Data available at:

```
# https://pythonprogramming.net/static/downloads/machine-learning-data/crypto_data.zip

from tensorflow.keras.callbacks import TensorBoard, ModelCheckpoint
# Tensorboard - callback
# ModelCheckpoint - callback where you can set various params as to when you want to save certain checkpoints

SEQ_LEN=60                          # Look at last 60 minutes of data (4 pairs px/volume..
FUTURE_PERIOD_PREDICT=3             # Predict 3 minutes in future
RATIO_TO_PREDICT="LTC-USD"          # User to change this...ex. LTC-USD by watching other prices
EPOCHS = 10                         # How many EPOCHS want to train the Model for.
BATCH_SIZE = 64
NAME = f"{RATIO_TO_PREDICT}-{SEQ_LEN}-SEQ-{FUTURE_PERIOD_PREDICT}-PRED-{int(time.time())}"
# import tensorflow stuff above

def classify(current,future):          # Take current & future price
    if float(future)>float(current):   # if Px higher in future than now in our Training Data
        return 1                       # Those features then are 1.
    else:                              # With this sequence of features...hoping Model can learn this relnship
        return 0

 #Step. Define conversion function for native timestamps in csv file
def dateparse (time_in_secs):
    return datetime.datetime.fromtimestamp(float(time_in_secs))

print('Data listing...')
print(os.listdir('../gb_Jupyter_Notebook_Repository/raw_csv/crypto_data'))

df=pd.read_csv("../gb_Jupyter_Notebook_Repository/raw_csv/crypto_data/LTC-USD.csv",names=["time",
"low","high","open","close","volume"])
# raw csv has no column titles so specify above.
print(df.head()) # Printing head of this dataframe
df.info()
```

 # Step. Merge Data. Now just concern w Close & Volume....Data in 4 csv - but all share same Time - so join all 4 csv on time axis

```
main_df=pd.DataFrame()
ratios=["BTC-USD", "LTC-USD", "ETH-USD", "BCH-USD"]
# nb. RATIO-TO-PREDICT defined above
for ratio in ratios:
    #   Want to iterate over 4 ratios
    dataset =  f"../gb_Jupyter_Notebook_Repository/raw_csv/crypto_data/{ratio}.csv"  #ratio means all 4 pairs
```

```
    df=pd.read_csv(dataset,names=["time", "low","high","open","close","volume"])
   # print(df.head())
```

#Now merge above 4 separate dataframes. Close & Volume now become f strings

```
    df.rename(columns={"close":f"{ratio}_close","volume":f"{ratio}_volume"},inplace=True)
    #True means dont hv to redefine df
    df.set_index("time", inplace=True)
    df=df[[f"{ratio}_close",f"{ratio}_volume"]]
    # print(df.head())
    if len(main_df)==0:   #ie. it is empty
        main_df = df
    else:
        main_df=main_df.join(df)

print(main_df.head())

#for c in main_df.columns:
#    print(c)

# Now need to set Targets - so specify Sequence Length above, Predict, RATIO_TO_PREDICT (defined above)
# .shift will just shift columns for us, -ve shift will shift them "up"
# so shifting up 3 will give us px 3min in the future & we are just assigning this to new column
# Now we hv future values, we can use them to make a Target using above fn
main_df['future']=main_df[f"{RATIO_TO_PREDICT}_close"].shift(-FUTURE_PERIOD_PREDICT)

print(main_df[[f"{RATIO_TO_PREDICT}_close", "future"]].head())

# Now we hv future px, now want to map this fn to a new column "TARGET"
# map(): used to map a fn
# Param1: classify - this is fn we want to map
# Param2: are params to above fn (ie. current close px & then future px)
# map() part is what allows us to do this row-by-row for these columns
# list() part converts end result to a list, which we can just set as a column
main_df['target']=list(map(classify,main_df[f"{RATIO_TO_PREDICT}_close"], main_df["future"]))
print(main_df[[f"{RATIO_TO_PREDICT}_close", "future", "target"]].head(10))

# So above is 3 period ahead prediction (Future 96.50 does appear 3 time later)
# Now ready to build sequences, balance the data, normalize the data, scale the data, Sample)

# Step below (Out-of-Sample separation) Last 5% of data: last_5pct =
times = sorted(main_df.index.values)
# data should be in order - but making sure in order / .index references index / .values converts to numpy array
```

 # Now want to find Threshold (actual unix time) of last 5% of data

```
last_5pct = times[-int(0.05*len(times))]  #Could be main_df of time
print(last_5pct)
# Above returns Unix timestamp "1534922100" - this is Threshold for last 5%
```

 # Now separate Training Data (in-sample data) from Validation Data (out-of-sample Data)

```
# Can create problem with normalization, scaling - but can fix this later
# GB notes - keep this order to prevent error below (...cannot scale)
validation_main_df=main_df[(main_df.index>=last_5pct)]      # 2/2 Validation Data (out-of-sample data)
main_df = main_df[(main_df.index < last_5pct)]              # 1/2 Training Data (in-sample data)

# Now have split up data
# Now need to create sequences, balance, scale
```

 # Step. Create preprocess_df function

```
def preprocess_df(df):        # Fn will take in a dataframe (df) as a param. Now work on Scaling
    df = df.drop('future',1)   # Drop 'future' column as only earlier needed it to generate target
    for col in df.columns:
        if col != "target":
```

```
        df[col]=df[col].pct_change()    # This normalizes data-px mvmt (% change) (ex. BTC px vs. LTC px, BTC
Volume vs. LTC volume
        df.dropna(inplace=True)         # (% change normalizes above). We are interested in movements (trends of
movements)
.
        df[col]=preprocessing.scale(df[col].values)

        df.dropna(inplace=True) # Drops NaN
```

 # Step. Now lets stuff in sequential data

```
        sequential_data=[]      # Sequential data = empty list
        prev_minutes=deque(maxlen=SEQ_LEN) # deque - like a list - keep appending to list - this is our sequence -
                                # if hit max 60 - gets rid of earlier
                                # Wait to prev_days gets 60 values and then keep populating it
        # print(df.head())  # Sanity check
        # for c in df.columns:
        #     print(c)
        for i in df.values:                   # df.values just converts dataframe to list of lists
           prev_minutes.append([n for n in i[:-1]])    # it won't contain time anymore but will still be order of index
           if len (prev_minutes)==SEQ_LEN:            # Now iterate over the columns.

              sequential_data.append([np.array(prev_minutes),i[-1]]) # Here we are appending our features and labels
              # It will however contain target so be careful. For i (row of all 8 columns)
              # prev_minutes.append(a list) - up to -1 means excluding target!
              # Current label i[-1] hoping model could predict  or 10
        random.shuffle(sequential_data)    #Step can take awhile as building sequences
```

#Step. Now hv our sequences, targets...closing in on ability to feed it thru a Model (Sequential)

```
# We hv got preprocessing happening, we hv built sequential data & we hv separated out our Validation data
# We normalized the data,  df[col]=df[col]./pct_change()....
# We scaled the data here, df[col]=preprocessing.scale(df[col].values)
# Now we need to *balance* data (ie. buy/sells 48/52 ok ) but if > 60/40 split - need to balance dataset
```

 # Step. Balance the data

```
# Buys is a list
   buys = []
# Sells is a list
   sells = []

   for seq, target in sequential_data:
      if target ==0:  # Means a sell
         sells.append([seq,target])
      elif target==1:
          buys.append([seq,target])

   random.shuffle(buys)
   random.shuffle(sells)
    lower = min(len(buys),len(sells))

   buys = buys[:lower] #if len was 30k would say 30k
   sells = sells[:lower]

   sequential_data=buys+sells
   random.shuffle(sequential_data)

  # now we want to split out into x and y...going to invoke some model.fit(x,y)
   X = []   # X = a list
   y = []   # y = a list
   # now iterate over sequential data
   for seq, target in sequential_data:
      X.append(seq)
      y.append(target)
```

```
    return np.array(X),y  #Now our preprocessing dataframe function should be complete

# Make Fn to allow us to do above on both 1/2 and 2/2
train_x, train_y = preprocess_df(main_df)  # Pass main_df
validation_x, validation_y = preprocess_df(validation_main_df)
# Now go to top and define preprocess fn
# Now run preprocess - should see Target & everything should be converted to % change and scaled
```

 # Step. Add statistics

```
print(f"train data: {len(train_x)} validation data: {len(validation_x)}")
print(f"Train_Dont_buys: {train_y.count(0)}, Train_buys: {train_y.count(1)}")
print(f"Validation_Dont_buys: {validation_y.count(0)}, Validation_buys: {validation_y.count(1)}")
```

#Step. Build Model & train Model

```
# Create Model(model=Sequential, model.add(), Compile Model (model.compile(), Fit Model (model.fit()
# Above: now import time lib & add Epochs
# Sequential Model
model = Sequential()

# LSTM layer needs 3D input. (Samples, time steps, features)
```

#Layer 1/5

```
model.add(LSTM(128,input_shape=(train_x.shape[1:]), return_sequences=True))
# 128 nodes in this layer
model.add(Dropout(0.2))
model.add(BatchNormalization()) # Add BatchNormalization but with no params
```

#Layer 2/5

```
model.add(LSTM(128,input_shape=(train_x.shape[1:]), return_sequences=True))
model.add(Dropout(0.2))
model.add(BatchNormalization())
```

#Layer 3/5

```
model.add(LSTM(128,input_shape=(train_x.shape[1:]))) # so remove return_sequences
model.add(Dropout(0.2))
model.add(BatchNormalization())
```

#Layer 4/5. Add another Dense layer

```
model.add(Dense(32,activation="relu")) #Rectified Linear..
# nb. If can't use CuDDNLSTM - throw in some activations - "tanh" (which is what CuDDNLSTM uses) or "relu"
model.add(Dropout(0.2))
```

#Layer 5/5. Final Dense layer - output

```
model.add(Dense(2,activation="softmax")) #Binary choice means use 2  (2 options)
# As output layer, activation choice is softmax.
```

#Specify Optimizer

```
opt = tf.keras.optimizers.Adam(lr=0.001, decay=1e-6)  #lr = learning rate
```

#Compile Model

```
# Keras doc: compile(loss,optimizer,metrics,loss_weights,sample_weight_mode,weighted_metrics,target_tensors)
model.compile(loss='sparse_categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
```

 # Define Callbacks

```
# Callback 1/2. Tensorboard
tensorboard=TensorBoard(log_dir=f'logs\\{NAME}')
# GB changed syntax from /  to \\

filepath = "RNN_Final-{epoch:02d}-{val_accuracy:.3f}"
```

```
#Callback 2/2. Checkpoint
checkpoint = ModelCheckpoint("models\\{}.model".format(filepath, monitor='val_accuracy', verbose=2,
save_best_only=True, mode='max'))

train_y = np.asarray(train_y)
validation_y = np.asarray(validation_y)
```

 # Fit Model

```
history=model.fit(
    train_x, train_y,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    validation_data=(validation_x, validation_y),
    callbacks=[tensorboard,checkpoint])
```