Cash Flow Forecasting Using Deep Learning

The Problem

Morris Corfield has very uneven cash flow. This is primarily for two reasons:

1. Customers tend to be farms, whose own cash flow is seasonal and especially dependent upon clement weather
2. The sales mix is of parts, commonly £50 per unit, and machinery, commonly £50,000 per unit, (up to £300,000/unit) Approximately 1,000 machines will be sold in a year, approx 10,000 parts will be sold in the same time.

Previous approaches to this problem

The classical solution is to associate a 'time to pay' to each purchase (creditor) or sale (debtor) and model when cash will transfer, hence moving the bank balance. This is reasonable and provides a base line for alternative forecasting models.

In this notebook we explore a deep learning model. It applies a stack of 1D convolutions, these are 'causal' (they respect time dependency) and dilated, aka 'Wavenet'. A very similar model was used for parts forecasting and is applicable to most time series problems.

This notebook will use Rstudio v1.2 (not yet stable!) in order to benefit from python and R is a single notebook.

Model Input and Output

The data we need is in the accounting SQL database and can be extracted via ODBC

```r
library(dplyr)
library(tidyr)
library(RODBC)
library(reticulate)
library(knitr)

#lots of packages, ensure dplyr gets priority
select    <- dplyr::select
filter    <- dplyr::filter
summarise <- dplyr::summarise

# Extract the required cash flow data.
# The SQL used in the stored procedure is in an appendix at the end of the notebook
srcData <- sqlQuery(connection, "EXEC [sp_GetCashFlowModellingData]", stringsAsFactors=FALSE)

# This is the kind of data where it helps to set NA to zero
srcData[is.na(srcData)] <- 0

# The reference is the 'Calendar Day' and the final 8 columns are the target data, the 'y'
y_r <- srcData %>% select(CalendarDay,
                          RollingBalance_Bank14Hence,
                          RollingBalance_Bank30Hence,
                          MinFortniteHence, AvgFortniteHence, MaxFortniteHence,
                           MinMonthHence,    AvgMonthHence,    MaxMonthHence)

# Which leaves the rest as the predictors, the 'x'
x_r <- srcData %>% select(-RollingBalance_Bank14Hence,
                           -RollingBalance_Bank30Hence,
                           -MinFortniteHence, -AvgFortniteHence, -MaxFortniteHence,
                           -MinMonthHence,    -AvgMonthHence,    -MaxMonthHence)

# Clear out the imported file
rm(srcData)

# Close connection
odbcClose(connection)
```

The data is comprised of ten rows per calendar day. Each row per day represents a department posting its records into the accounts system; machinery sales, parts sales, parts purchases, machinery purchases etc. Each of these movement is a movement in debtors or creditors. It is not necessarily a cash movement.

Each row is a summary of all the transactions in that department for the day. This results in a total transaction value for the day, and *value_weighted* averages for the customer/supplier's known days to pay. Note, the resulting value can be -v eor +ve for either purchases or sales depending on whether the largest transaction in the day was an invoice or a credit. Finally we have the closing bank balance for the day. This is only available once for the day, so is repeated for each row within the day.

Also, we have cash movements which do not result from either a purchase or a sale: + Wages + Pensions, + Corporation Tax, + Dividends, Share buybacks and Share purchases + Loan payments + VAT payments to/from HMRC + etc Note, fixed asset sales and purchases would normally move through the debtor and creditor records in the acctg system being used.

Whereas movements in debtors and creditors takes time to become cash (ie for the bill to be paid), wages, dividends and tax tend to be paid immediately and on a schedule.

In some companies wages are fairly even throughout the year and could be ignored, being a constant bias, but this business is highly seasonal.

All the above are exactly the kind of figures an accountant would seek in order to estimate future movements of the bank balance.

```r
#let's see the predictors for a specific day
kable(head(x_r %>% filter(CalendarDay==as.Date('2016-03-04')), 11))
```

CalendarDay	PostingDelayDays_Weighted	SaleOrPurchase	Category	TransactionType	
2016-03-04	0.000000	NonSaleOrPurchase	Not a Purchase	Manual Journal	
2016-03-04	0.000000	Purchase	Finance	0	
2016-03-04	28.000000	Purchase	Machinery	Purchase Journal	
2016-03-04	18.994494	Purchase	Machinery	Purchase Invoice	
2016-03-04	0.000000	Purchase	Parts	Purchase Cash Payment	
2016-03-04	0.000000	Purchase	Parts	Purchase Journal	
2016-03-04	37.809158	Purchase	Parts	Purchase Invoice	
2016-03-04	8.030268	Purchase	Unspecified	Purchase Invoice	
2016-03-04	0.567208	Purchase	Unspecified	Purchase Cash Payment	
2016-03-04	0.000000	Sale	Chargeable Work	0	
2016-03-04	0.000000	Sale	Other	Manual Invoice/Credit	
Let's also see	the targets:				
We are interes	te din forecasting our cash	flow. We'd like to k	now the min, max a	nd average value of our	ba

Therefore, our data includes the following 6 figures, each relating to the bank balance, for every day:

– MinFortniteHence – AvgFortniteHence – MaxFortniteHence

– MinMonthHence – AvgMonthHence – MaxMonthHence

We will be using a sequence to sequence (seq2seq) modelling method, so will not be training our model to match the above scalar values. Instead we need to train the model to forecast a sequence of bank balances for 14-30 days hence. Then we calculate the min, avg and max to see how useful the forecast is. So for training we also have these two values for every day:

– RollingBalance__Bank14Hence – RollingBalance__Bank30Hence

```r
#let's see the targets for a specific day
kable(head(y_r %>% filter(CalendarDay == as.Date('2016-03-04')), 11))
```

CalendarDay	RollingBalance__Bank14Hence	RollingBalance__Bank30Hence	MinFortniteHence	AvgFortniteHence
2016-03-04	-441693.7	-327598.1	-897506.5	-716305.9
2016-03-04	-441693.7	-327598.1	-897506.5	-716305.9
2016-03-04	-441693.7	-327598.1	-897506.5	-716305.9
2016-03-04	-441693.7	-327598.1	-897506.5	-716305.9
2016-03-04	-441693.7	-327598.1	-897506.5	-716305.9
2016-03-04	-441693.7	-327598.1	-897506.5	-716305.9
2016-03-04	-441693.7	-327598.1	-897506.5	-716305.9
2016-03-04	-441693.7	-327598.1	-897506.5	-716305.9
2016-03-04	-441693.7	-327598.1	-897506.5	-716305.9
2016-03-04	-441693.7	-327598.1	-897506.5	-716305.9
2016-03-04	-441693.7	-327598.1	-897506.5	-716305.9

There are a number of factors in the predictor data, these need converting to one-hot encoding for the deep model. The SaleOrPurchase column is convenient for reading but superflous, telling us nothing more than the Category column does.

```r
library(lubridate)

# replace NA or NULL with zero
x_r[is.na(x_r)] <- 0

# Add month of year as a predictor, data will be somewhat seasonal
x_r_processed <- x_r %>% mutate(MonthOfYr = month(CalendarDay))

# concatenate SaleOrPurchase and Category
x_r_processed <- x_r_processed %>%
                  mutate(NewCategory = paste0(SaleOrPurchase, Category)) %>%
                    select(-SaleOrPurchase, -Category)

# spread the NewCategory column into 11 columns, ie as onehot
x_r_processed <- x_r_processed %>%
    mutate(Cat_NonSaleOrPurch = ifelse(NewCategory == 'NonSaleOrPurchaseNot a Purchase', 1, 0),
           Cat_PurchFinance   = ifelse(NewCategory == 'PurchaseFinance',                 1, 0),
           Cat_PurchMachinery = ifelse(NewCategory == 'PurchaseMachinery',               1, 0),
           Cat_PurchParts     = ifelse(NewCategory == 'PurchaseParts',                   1, 0),
           Cat_PurchUnspecif  = ifelse(NewCategory == 'PurchaseUnspecified',             1, 0),
           Cat_SaleChargework = ifelse(NewCategory == 'SaleChargeable Work',             1, 0),
           Cat_SaleOther      = ifelse(NewCategory == 'SaleOther',                       1, 0),
           Cat_SaleParts      = ifelse(NewCategory == 'SaleParts Over the Counter',      1, 0),
           Cat_SaleUnspecif   = ifelse(NewCategory == 'SaleUnspecified',                 1, 0),
           Cat_SaleHire       = ifelse(NewCategory == 'SaleWholegood Hire',              1, 0),
           Cat_SaleWholegood  = ifelse(NewCategory == 'SaleWholegood Sales',             1, 0)) %>%
      select(-NewCategory, -TransactionType)

#view the data
kable(head(x_r_processed %>% filter(CalendarDay == as.Date('2016-03-04')), 11))
```

CalendarDay	PostingDelayDays_Weighted	TransactionValue	AvgDaysToPay_Weighted	AvgValuePaid_Weighted
2016-03-04	0.000000	-600.00	3.00000	5000.00
2016-03-04	0.000000	0.00	0.00000	0.00
2016-03-04	28.000000	-22696.04	43.00000	22959.94
2016-03-04	18.994494	-45410.72	43.00024	22953.45
2016-03-04	0.000000	634.08	3.00000	24820.24
2016-03-04	0.000000	0.00	43.00000	2544.31
2016-03-04	37.809158	-214539.68	43.00000	44604.97
2016-03-04	8.030268	-107181.35	43.00000	15999.85
2016-03-04	0.567208	90510.81	3.00000	64419.86
2016-03-04	0.000000	0.00	0.00000	0.00
2016-03-04	0.000000	124200.00	2.00000	65009.96

Forming Examples for Training

We have one time series, whereas we need many examples to train a deep learning model. An 'example' could be 365 days of data. It should be feasible to forecast the cash position at the end of that period, knowing a year of data and the opening position.

We could then slide our 365 day window forward by one day, thus forming the next example. In fact, we don't need to slide forward by one day, we could randomly sample any 365 day span, so long as the days within the example are in sequence.

There are 9yrs of data in the data, so this approach would yield 9 * 365 = 3,285 examples, either randomly or in sequence. The final year would be reserved for testing, so training examples are reduced to 8 x 365 = 2,920.

Presenting examples in sequence is acceptable, in real life the data will always be revealed 'in sequence'.

Validation data would also be a 365 day window, we cannot use the same window as for training, so will use walk forward validation. It will be shifted forward by the forecast period, which is 30 days. This means we lose another 30 days from the end of the training period, leaving 2,890 training examples total.

The data is in an array of one example, we need a python generator function to yield sequential 365 day spans to the model fitting process.

```
# Transpose and scale
x_matrix <- scale(x_r_processed %>% select (-CalendarDay))

# Separate into train, validate and test.
# Final year is test year. In order to achieve 100 test examples, we need 100 days in addition to 365 d
# NB there are 11 rows for each day of data, each row represents a category of sale/purchase, eg wholeg
train_cols <- seq(from= 1,
                  to   = nrow(x_r_processed)-465*11,
                  by   = 1)
```

```
test_cols  <- seq(from= nrow(x_r_processed)-465*11+1,
                  to   = nrow(x_r_processed),
                  by   = 1)

x_train    <- x_matrix[train_cols, ]
x_test     <- x_matrix[test_cols,  ]
```

Create ‘y’ matrix to match

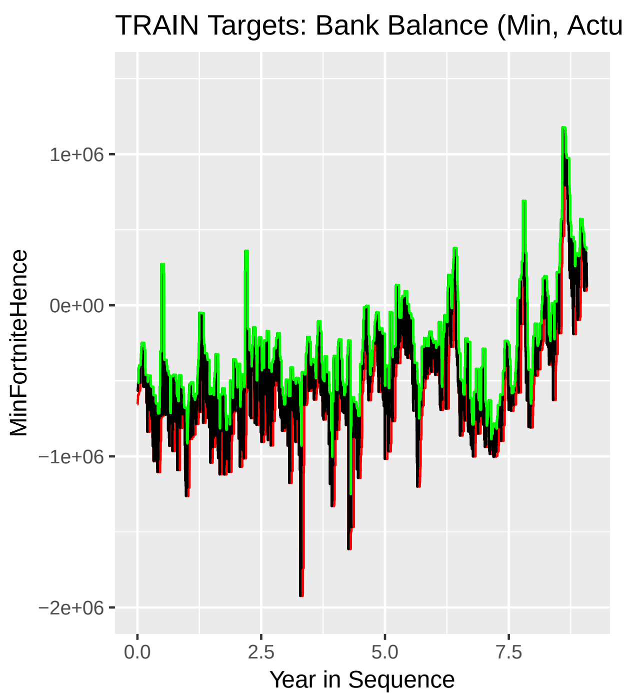
```
#transpose (no need to scale)
y_matrix <- as.matrix(y_r %>% select (-CalendarDay))

# Separate into train and test. Final year is test year
y_train  <- y_matrix[train_cols, ]
y_test   <- y_matrix[test_cols,  ]
```

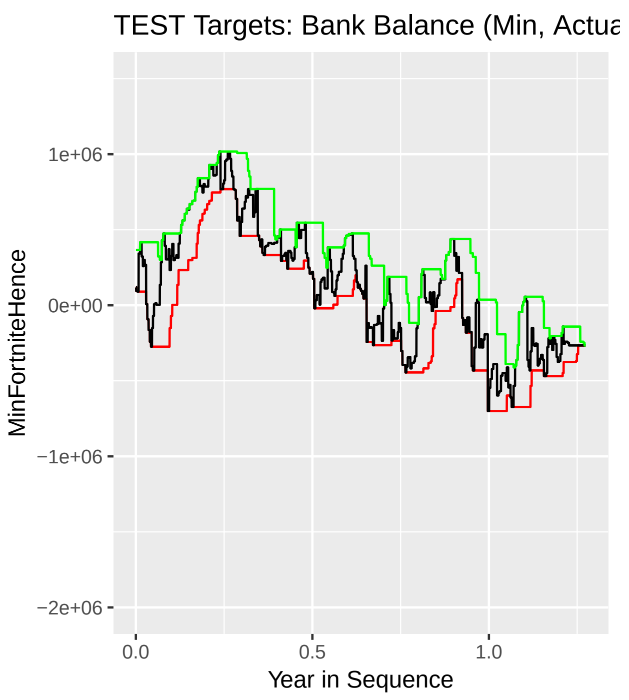
Visualise the Target Data

Let’s see what we’re aiming for.

In our first model we’ll aim to forecast the bank balance for each of the coming 14 days In a less ambitious second model, we’ll attmept to forecast a single figure to represent those 14 days, either: – MinFortniteHence – AvgFortniteHence – MaxFortniteHence

```
library(ggplot2)

y_train_chart <- cbind.data.frame(Seq                          = 1:dim(y_train)[1]/11/365,
                                  RollingBalance_Bank14Hence = y_train[, 1],
                                  MinFortniteHence           = y_train[, 3],
                                  MaxFortniteHence           = y_train[, 5])

y_test_chart  <- cbind.data.frame(Seq                          = 1:dim(y_test)[1]/11/365,
                                  RollingBalance_Bank14Hence = y_test[, 1],
                                  MinFortniteHence           = y_test[, 3],
                                  MaxFortniteHence           = y_test[, 5])

a <- ggplot(data=y_train_chart, aes(x=Seq)) +
          geom_line(aes(y=MinFortniteHence),           col="red")   +
          geom_line(aes(y=RollingBalance_Bank14Hence), col="black") +
          geom_line(aes(y=MaxFortniteHence),           col="green") +
          ggtitle("TRAIN Targets: Bank Balance (Min, Actual, Max)") +
          xlab("Year in Sequence")+ ylim(-2000000,1500000)

b <- ggplot(data=y_test_chart, aes(x=Seq)) +
          geom_line(aes(y=MinFortniteHence),           col="red")   +
          geom_line(aes(y=RollingBalance_Bank14Hence), col="black") +
          geom_line(aes(y=MaxFortniteHence),           col="green") +
          ggtitle("TEST Targets: Bank Balance (Min, Actual, Max)") +
          xlab("Year in Sequence") + ylim(-2000000,1500000)

library(gridExtra)
  grid.arrange(a,b, ncol=2,nrow=1)
```





Transfer to Python and Keras on Tensorflow

This model will be built with Keras on a Tensorflow backend using a GPU. Python is native to Keras, so we'll move to python for Keras tasks. This is straightforward using the R package 'Reticulate':

```
import pandas as pd
import numpy  as np
#dplyr for pandas in python
from   dfply import *
# in order to access the R objects from Python we prefix them with "r."
x_train_py = r.x_train
y_train_py = r.y_train
x_test_py  = r.x_test
y_test_py  = r.y_test
#reformat x into (n_series, n_timesteps, n_features)
def Get3D(array):
  array_3D = array.reshape((1, array.shape[0], array.shape[1]))
  return array_3D

x_train_py = Get3D(x_train_py)
x_test_py  = Get3D(x_test_py)
y_train_py = Get3D(y_train_py)
y_test_py  = Get3D(y_test_py)
print("Train dims: x=", x_train_py.shape, ". y=", y_train_py.shape )
```

```
## Train dims: x= (1, 36465, 20) . y= (1, 36465, 8)
```

```
print("Test  dims: x=", x_test_py.shape, " . y=", y_test_py.shape )
```

```
## Test  dims: x= (1, 5115, 20)  . y= (1, 5115, 8)
```

As you see, we currently have an input with one time series (n_examples=1), 36000 data points in that time series (n_timesteps = 36000) and 20 features (n_features=20).

We'll need to split our data into many examples with which we can train the model.

To confidently make a 14day forecast of cash-flow a human would want approximately one year of previous sales, debtors, creditors, etc. So we'll set each example to be 365 days long and train the model to the subsequent 14 days (14x11 timesteps) of the bank balance. Since each day has 11 records, thats 365*11 timesteps per example.

We will then slide the 365day window along by one day (1x11 timesteps) to create the next example.

20% of the examples will be used for validation, but note they ar enot independent of each other. Each example will overlap by 364 days! This is not a great situation. But we have < 3000 examples for both train and validation (36000 / 11 less final year), this is not much for training a deep model.

```
records_per_example = 365*11
```

Python Generator

We'll use a generator to slide our window across the data for both train and validate. Generators can be a bit awkward as we need to be sure we understand the data they are feeding to the model. Therefore a 'monitor' mode has been added to this function which allows us to view the dims of the tensors it returns.

FYI, for more complex projects which read data from disk and manufacture new examples, common with images, generators can be even more complex. See https://stanford.edu/~shervine/blog/keras-how-to-generate-data-on-the-fly

```
# let's define an example as any window on the data which is 'days_per_example' wide
# we need a function to give us 'batch_size' qty of examples
def yield_xy(sourcearray_x, sourcearray_y, records_per_example, batch_size, step_size = 11,
             monitor_me = False):

  # monitor_me puts the generator into a 'test' mode,
  # cos generators are finnicky and liable to inf loops

  max_records   = sourcearray_x.shape[1]
  max_examples  = int((max_records - records_per_example)/step_size)
  max_batches   = int(max_examples/batch_size)
  example_count = 0
  batch_count   = 1
  # build batches
  while batch_count <= max_batches:

    # build a batch of examples
    for example in range(batch_size):

      start_inclusive = example_count * step_size
      end_exclusive   = start_inclusive + records_per_example

      # if we exceed bounds of data, or deliver more batches than fit_generator expects,
      # then break for loop
      if (end_exclusive > max_records) or (batch_count > max_batches) :
        break # this breaks the for loop, not the parent while loop.

      # get first example
      if example == 0:
        batch_examples_x = sourcearray_x[ : , start_inclusive:end_exclusive, : ]
        batch_examples_y = sourcearray_y[ : , start_inclusive:end_exclusive, : ]
      # concatenate further examples into our batch of examples
      else :
        batch_examples_x = np.concatenate((batch_examples_x,
                                           sourcearray_x[:, start_inclusive:end_exclusive, : ]),
                                           axis=0)

        batch_examples_y = np.concatenate((batch_examples_y,
                                           sourcearray_y[:, start_inclusive:end_exclusive, : ]),
                                           axis=0)

      #increment the example count before looping to the next example
      example_count += 1

      if monitor_me :
        generator_shape = (start_inclusive, end_exclusive)
        generator_count = (example_count)
      # End of for loop around a batch

    # break the while loop if we've exceeded bounds
    if (end_exclusive > max_records) or (batch_count > max_batches) :
      break

    batch_count += 1

    # if in test mode...
    if monitor_me :
      yield (example_count, generator_shape, batch_count-1, max_batches, batch_examples_x.shape)
    # else in production !
    else:
      yield (batch_examples_x, batch_examples_y)

    #End of while loop
```

We can use this generator in a 'test' mode and see what array shapes it returns, we expect 'batch_count' arrays, each of 'batch_size' rows. All column sizes should be the same.

The generator acts a list, with each list item being 'yielded' to us as it has been created

```
batch_size = 64
test_gen   = yield_xy(x_train_py, y_train_py, records_per_example = 365*11, step_size = 11,
                      batch_size =  64, monitor_me = True)
print("example_count, start_and_end_example, batch_count, max_batches, batch.shape")
# interrogate a generator as if it were a list...
```

```
## example_count, start_and_end_example, batch_count, max_batches, batch.shape
```

```
for i in test_gen:
  print(i)
```

```
## (64, (693, 4708), 1, 46, (64, 4015, 20))
## (128, (1397, 5412), 2, 46, (64, 4015, 20))
## (192, (2101, 6116), 3, 46, (64, 4015, 20))
## (256, (2805, 6820), 4, 46, (64, 4015, 20))
## (320, (3509, 7524), 5, 46, (64, 4015, 20))
## (384, (4213, 8228), 6, 46, (64, 4015, 20))
## (448, (4917, 8932), 7, 46, (64, 4015, 20))
## (512, (5621, 9636), 8, 46, (64, 4015, 20))
## (576, (6325, 10340), 9, 46, (64, 4015, 20))
## (640, (7029, 11044), 10, 46, (64, 4015, 20))
## (704, (7733, 11748), 11, 46, (64, 4015, 20))
## (768, (8437, 12452), 12, 46, (64, 4015, 20))
## (832, (9141, 13156), 13, 46, (64, 4015, 20))
## (896, (9845, 13860), 14, 46, (64, 4015, 20))
## (960, (10549, 14564), 15, 46, (64, 4015, 20))
## (1024, (11253, 15268), 16, 46, (64, 4015, 20))
## (1088, (11957, 15972), 17, 46, (64, 4015, 20))
## (1152, (12661, 16676), 18, 46, (64, 4015, 20))
## (1216, (13365, 17380), 19, 46, (64, 4015, 20))
## (1280, (14069, 18084), 20, 46, (64, 4015, 20))
## (1344, (14773, 18788), 21, 46, (64, 4015, 20))
## (1408, (15477, 19492), 22, 46, (64, 4015, 20))
## (1472, (16181, 20196), 23, 46, (64, 4015, 20))
## (1536, (16885, 20900), 24, 46, (64, 4015, 20))
## (1600, (17589, 21604), 25, 46, (64, 4015, 20))
## (1664, (18293, 22308), 26, 46, (64, 4015, 20))
## (1728, (18997, 23012), 27, 46, (64, 4015, 20))
## (1792, (19701, 23716), 28, 46, (64, 4015, 20))
## (1856, (20405, 24420), 29, 46, (64, 4015, 20))
## (1920, (21109, 25124), 30, 46, (64, 4015, 20))
## (1984, (21813, 25828), 31, 46, (64, 4015, 20))
## (2048, (22517, 26532), 32, 46, (64, 4015, 20))
## (2112, (23221, 27236), 33, 46, (64, 4015, 20))
## (2176, (23925, 27940), 34, 46, (64, 4015, 20))
## (2240, (24629, 28644), 35, 46, (64, 4015, 20))
## (2304, (25333, 29348), 36, 46, (64, 4015, 20))
## (2368, (26037, 30052), 37, 46, (64, 4015, 20))
## (2432, (26741, 30756), 38, 46, (64, 4015, 20))
## (2496, (27445, 31460), 39, 46, (64, 4015, 20))
## (2560, (28149, 32164), 40, 46, (64, 4015, 20))
## (2624, (28853, 32868), 41, 46, (64, 4015, 20))
## (2688, (29557, 33572), 42, 46, (64, 4015, 20))
## (2752, (30261, 34276), 43, 46, (64, 4015, 20))
## (2816, (30965, 34980), 44, 46, (64, 4015, 20))
## (2880, (31669, 35684), 45, 46, (64, 4015, 20))
## (2944, (32373, 36388), 46, 46, (64, 4015, 20))
```

Getting the generator right is so important that we will subject it to another test…

```python
# use this import to ensure we can limit the returns from the generator, else we'll be here all day...
import itertools as it
# test by stacking 10 examples in a batch (batch_size=10)
generator_test = yield_xy(x_train_py[:,:,],
                          y_train_py[:,:,],
                          records_per_example=2*11,
                          batch_size = 3)
# get first output from the generator, should be the two arrays stacked on top of each other,
# for both x and y.
```

```
generator_test = list(it.islice(generator_test, 1))
print("First  in tuple should be x, type is: ", type(generator_test[0][0]))
```

```
## First  in tuple should be x, type is:  <class 'numpy.ndarray'>
```

```
print("Dims of x expected: (3, 2*11, 20)")
```

```
## Dims of x expected: (3, 2*11, 20)
```

```
print("Dims of x actually:", generator_test[0][0].shape)
```

```
## Dims of x actually: (3, 22, 20)
```

```
print("\n")
```

```
print("Second in tuple should be y, type is: ", type(generator_test[0][1]))
```

```
## Second in tuple should be y, type is:  <class 'numpy.ndarray'>
```

```
print("Dims of y expected: (3, 2*11,  6)")
```

```
## Dims of y expected: (3, 2*11,  6)
```

```
print("Dims of y actually:", generator_test[0][1].shape)
```

```
## Dims of y actually: (3, 22, 8)
```

Define the Model

Generator and data are ready, let's a define a model. We could use an LSTM, but causal dilated 1D conv nets (aka wavenets) require far fewer weights for the same scope (qty of timesteps). The 'dilation' requires them to be stacked, each layer taking us to the next power of the filter_width, which is typically 2. The power starts from 0, so first layer is 2 to the power of 0 = 1 timestep. But the 11th layer is 2 to the power of 11 = 2048 timesteps. 6months at 11 steps per day.

A key input to the complexity (number of weights) in the model is the number of filters, this has been set to 32 as default. Later code chunks will explore results with 16 filters.

```
from keras.models     import Model
from keras.layers     import Input, Conv1D, Dense, Dropout, Lambda, concatenate
from keras.optimizers import Adam, TFOptimizer
from keras.layers     import TimeDistributed
def create_model_cash(days_forecast, multby11 = True, n_channels = 1,
                      qty_dilations = 13, dropout_between_1Dconvs = False,
                      predictor_qty = 20, n_filters = 32):

  # convolutional layer parameters
  filter_width   = 2 #this is effectively the dilation power, only very long sequences would need >2
  dilation_rates = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4000][0:qty_dilations]

  if(multby11):
    seq_length = days_forecast*11
  else:
    seq_length = days_forecast

  # define an input history series and pass it through a stack of dilated causal convolutions.
  # defined using the functional API (as opposed to sequential, eg model.add())
  input_sequence = Input(shape=(None, predictor_qty))
  x = input_sequence

  # 8x 1D dilated causal convolutions
  # input shape  (batch, steps, channels)
  # output shape (batch, new_steps, filters)
  for dilation_rate in dilation_rates:
    x = Conv1D(filters       = n_filters,
               kernel_size   = filter_width,
               padding       = 'causal',
               dilation_rate = dilation_rate)(x)
    if(dropout_between_1Dconvs):
      x = Dropout(0.2)(x)
  #NB no activation after Conv1D

  # dense layer, ReLU activation
  # We don't have to use TimeDistributed, doesn't make difference.
  # So alternative would be: x = Dense(128, activation='relu')(x)
  x = TimeDistributed(Dense(128, activation='relu'))(x)

  # dropout, 20%
  x = Dropout(0.2)(x)

  # softmax, NB only 1D output.
  # No activation, a linear output unconstrained to any range
  # We don't have to use TimeDistributed, doesn't make difference.
  # So alternative would be: x = Dense(n_channels)(x)
  x = TimeDistributed(Dense(n_channels))(x)
  # extract the last few time step(s) as the training
  # x will be shape [n_examples, n_timesteps, 1]
  def slice(x, seq_length):
    return x[:,-(seq_length):,:]
    #returns as many rows as there were examples in the batch

  output_sequence = Lambda(slice, arguments={'seq_length':seq_length})(x)
  model = Model(inputs  = input_sequence,
                outputs = output_sequence)
  return(model)
```

```
#let's build a model for fortnite forecasts first
model_cash14 = create_model_cash(days_forecast=14)
model_cash14.summary()
```

```
## _________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =================================================================
## input_10 (InputLayer)        (None, None, 20)          0
## _________________________________________________________________
## conv1d_73 (Conv1D)           (None, None, 32)          1312
## _________________________________________________________________
## conv1d_74 (Conv1D)           (None, None, 32)          2080
## _________________________________________________________________
## conv1d_75 (Conv1D)           (None, None, 32)          2080
## _________________________________________________________________
## conv1d_76 (Conv1D)           (None, None, 32)          2080
## ________________________________________________________________
## conv1d_77 (Conv1D)           (None, None, 32)          2080
## ________________________________________________________________
## conv1d_78 (Conv1D)           (None, None, 32)          2080
## ________________________________________________________________
## conv1d_79 (Conv1D)           (None, None, 32)          2080
## ________________________________________________________________
## conv1d_80 (Conv1D)           (None, None, 32)          2080
## ________________________________________________________________
## conv1d_81 (Conv1D)           (None, None, 32)          2080
## ________________________________________________________________
## conv1d_82 (Conv1D)           (None, None, 32)          2080
## ________________________________________________________________
## conv1d_83 (Conv1D)           (None, None, 32)          2080
## ________________________________________________________________
## conv1d_84 (Conv1D)           (None, None, 32)          2080
## ________________________________________________________________
## conv1d_85 (Conv1D)           (None, None, 32)          2080
## ________________________________________________________________
## time_distributed_13 (TimeDis (None, None, 128)         4224
## ________________________________________________________________
## dropout_53 (Dropout)         (None, None, 128)         0
## ________________________________________________________________
## time_distributed_14 (TimeDis (None, None, 1)           129
## ________________________________________________________________
## lambda_7 (Lambda)            (None, None, 1)           0
## ================================================================
## Total params: 30,625
## Trainable params: 30,625
## Non-trainable params: 0
## ________________________________________________________________
```

31,000 params in total, but only 3,000 examples to train, validate and test.

```
from keras import callbacks
######################################################################
## Compile Model with Side by Side Validation
model_cash14.compile(optimizer = Adam(),
                     loss      = 'mean_absolute_error')
# hook up to Tensorboard for keeping records of progress
# to use the Tensorboard, go to command line, set cd "current project directory", then:
# tensorboard --logdir GraphRMS
# then
# http://localhost:6006/
## Tensorboard callback
tbCallback_cash  = callbacks.TensorBoard(log_dir        = './Tensorboard_cash14',
                                         histogram_freq = 0,
                                         write_graph    = True)
## Checkpoint callback
cpCallback_cash  = callbacks.ModelCheckpoint('./Checkpoints/cash_{epoch:02d}.hdf5',
                                             monitor    = 'val_acc',
                                             verbose    = 1,
                                             save_best_only=True,
                                             mode       = 'max')
```

```
##Earlystopping, stop training when validation accuracy ceases to improve
esCallback_cash  = callbacks.EarlyStopping(monitor   = 'val_loss',
                                           min_delta = 0.00005,
                                           patience  = 200,
                                           verbose   = 0,
                                           mode      = 'auto',
                                           baseline  = None)
```

Now we're ready to create train, validation and test instances of the the generator. We can then fit the model. These two tasks are in one code chunk because we must re-instantiate the generator each time we fit the model.

```
## Let's fit using walk forward validation
## NOTE use of model.fit_generator(), not model.fit()
epochs     = 50
batch_size = 64
train_gen = yield_xy(x_train_py, y_train_py[:,:,0], records_per_example, batch_size)
test_gen  = yield_xy(x_test_py,  y_test_py[:,:,0],  records_per_example, batch_size)
history_cash = model_cash14.fit_generator(
                              generator        = train_gen,
                              steps_per_epoch  = int((x_train_py.shape[1] - records_per_example)
                                                     / batch_size/11),
                              epochs           = epochs,
                              validation_data  = valid_gen,
                              validation_steps = int((x_valid_py.shape[1] - records_per_example)
                                                     / batch_size/11))
                             #callbacks        = [tbCallback_cash, esCallback_cash])
```

The above code causes Tensorflow errors.

I can't get past this problem, which is stopping iteration. Only other instance of this problem on Google is other people using RStudio, so it may be an environment problem.

Alternative to fit_generator()

We're Running out of time, so we'll try model.fit and prepare all the data in memory first. To do that we simply consume the results of our generator, for which we need the below function...

```
def get_accumulated_generator(generator_obj, train_valid_test, verbose = True):
  loop_count = 0
  for batch in generator_obj:

    xs = list(batch)[0]
    ys = list(batch)[1]

    if loop_count == 0:
      all_batches_x = xs
      all_batches_y = ys
    else :
      all_batches_x = np.concatenate((all_batches_x, xs), axis=0)
      all_batches_y = np.concatenate((all_batches_y, ys), axis=0)

    loop_count += 1
  if(verbose):
    print(train_valid_test,". x shape, all examples: ", all_batches_x.shape)
    print(train_valid_test,". y shape, all examples: ", all_batches_y.shape)

  return(all_batches_x, all_batches_y)
```

Now we unify all our examples as one large tensor for train and one for test.

```
batch_size = 64
train_gen = yield_xy(x_train_py, y_train_py, records_per_example, batch_size, step_size = 11)
x_train_accum, y_train_accum = get_accumulated_generator(train_gen, "Train")
```

```
## Train . x shape, all examples:  (2944, 4015, 20)
## Train . y shape, all examples:  (2944, 4015, 8)
```

```
test_gen  = yield_xy(x_test_py,  y_test_py,  records_per_example, batch_size = 100, step_size = 11)
x_test_accum,  y_test_accum  = get_accumulated_generator(test_gen,  "Test ")
```

```
## Test  . x shape, all examples:  (100, 4015, 20)
## Test  . y shape, all examples:  (100, 4015, 8)
```

In training we will target only RollingBalance_Bank14Hence, although we could target multiple outputs we probably don't have enough trinaing data for that. RollingBalance_Bank14Hence is the first of the channels in the targets array. NB, the array is of dimensions = n_examples, n_timesteps, n_channels

```
# for 14 day forecasts, channel = 0, ie RollingBalance_Bank14Hence
y_train_finaldate_14 = y_train_accum[:,-14*11:, 0].reshape((y_train_accum.shape[0], 14*11, 1))
y_test_finaldate_14  = y_test_accum[:, -14*11:, 0].reshape((y_test_accum.shape[0],  14*11, 1))
```

… and train a new model accordingly…

```
epochs     =  500
batch_size =   64
history_cash14 = model_cash14.fit(x                = x_train_accum,
                                  y                = y_train_finaldate_14,
                                  batch_size       = batch_size,
                                  epochs           = epochs,
                                  validation_split = 0.2,
                                  shuffle          = True)
# save model
model_cash14.save('./SaveModels/model_cash14.h5')
# save history
import pickle
with open('./SaveModels/model_cash14_history.pickle', 'wb') as file_pi:
  pickle.dump(history_cash14.history, file_pi)
```

Plot Learning Progress

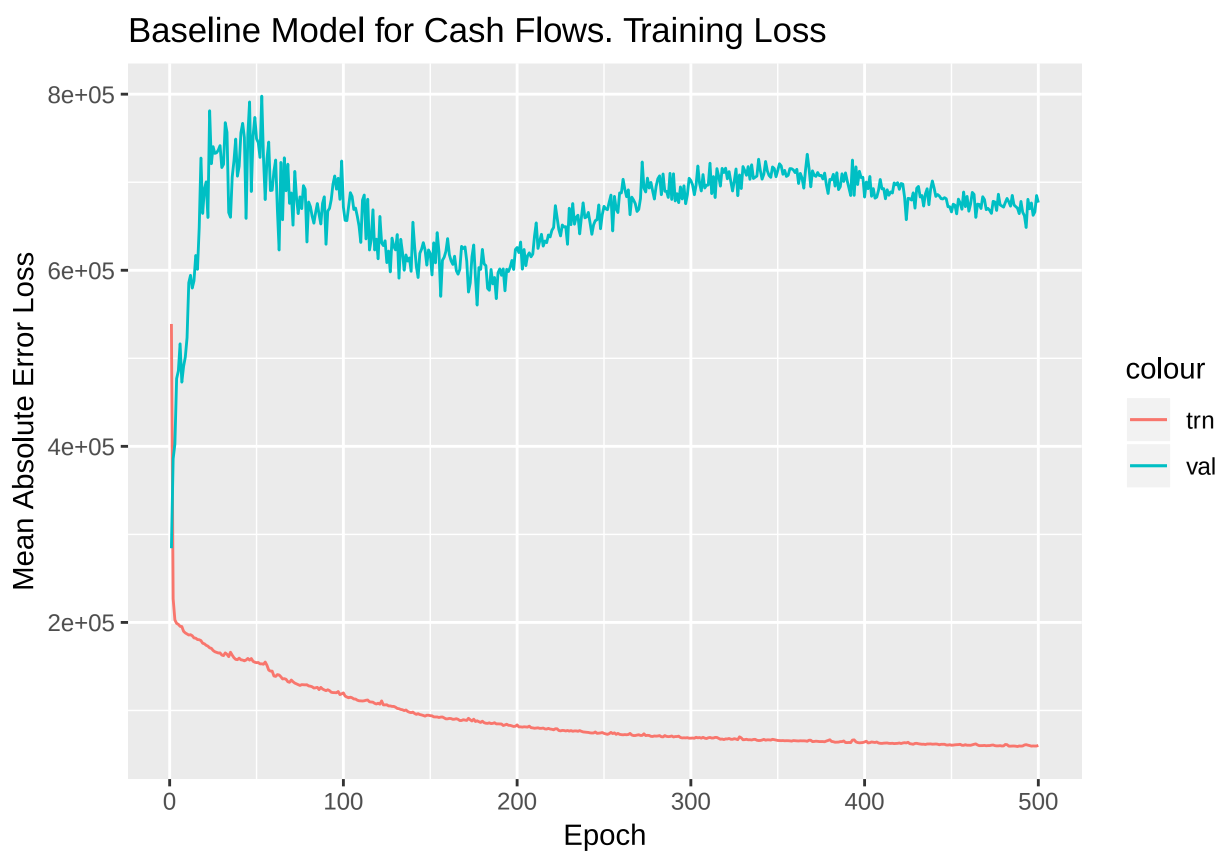
This is a three step process, first we load the model and training history from file, because it takes a long time and the session has probably been restarted since training time!

Secondly, we create a function to plot the training history. Thi sis done in ggplot2 rather than matplotlib because matplotlib is causing problems in the RStudio environment.

```
# the above training takes a long time, so we load results from file
from keras.models import load_model
import pickle
model_cash14   = load_model('./SaveModels/model_cash14.h5')
load_history   = open("./SaveModels/model_cash14_history.pickle","rb")
history_cash14 = pickle.load(load_history)

toggplot_cash14_loss     = history_cash14['loss']
toggplot_cash14_loss_val = history_cash14['val_loss']
```

```
library(ggplot2)

plot_learning <- function(history_py_cash_trn, history_py_cash_val, title){

  history_py_cash_epc <- seq_along(history_py_cash_trn)
  history_py_cash     <- cbind.data.frame(epoch    = history_py_cash_epc,
                                          loss     = unlist(history_py_cash_trn),
                                          val_loss = unlist(history_py_cash_val))

  p <- ggplot(data = history_py_cash, aes(x=epoch)) +
       geom_line(aes(y = loss,     col = "trn"))    +
       geom_line(aes(y = val_loss, col = "val"))    +
       xlab("Epoch")                                +
       ylab("Mean Absolute Error Loss")             +
       ggtitle(title)

  p
}
```

```
history_py_cash14_trn <- py$toggplot_cash14_loss
history_py_cash14_val <- py$toggplot_cash14_loss_val

plot_learning(history_py_cash_trn = history_py_cash14_trn,
              history_py_cash_val = history_py_cash14_val,
              title = "Baseline Model for Cash Flows. Training Loss")
```



Thats not training! We'll need to change our model or our data.

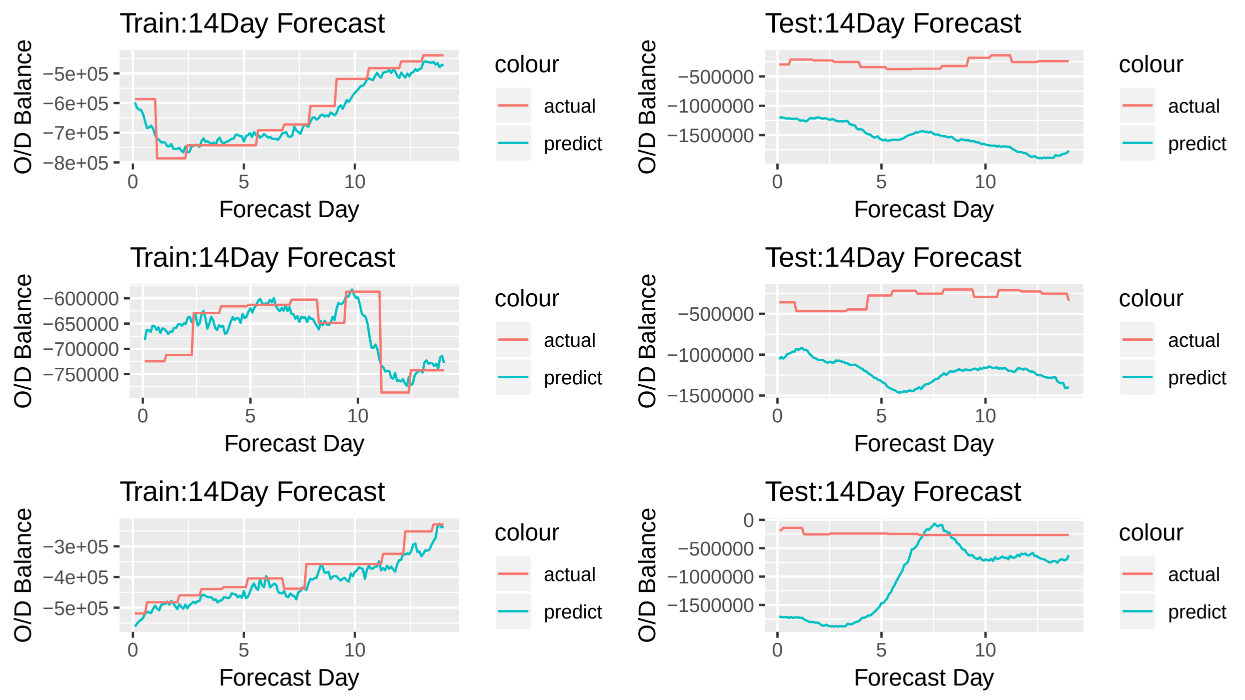
We'll go through the motions of testing on the test set, as it allows us to build the functions we need for later attempts. . .

Plot predictions vs Actuals for Test Set

```
preds_train_14 = model_cash14.predict(x_train_accum)
preds_test_14  = model_cash14.predict(x_test_accum)
```

```
plot_preds_train_vs_test <- function(preds_test,   preds_train,
                                     actuals_test, actuals_train,
                                     chart_examples = c(1),
                                     forecast_period){

  # predictions and y-arrays should be of dims =
  # (n_examples, n_timesteps = days x 11, n_channels = 1)

  chart_dat_test  <- list()
  chart_dat_train <- list()

  for (i in 1:length(chart_examples)){

    #Chart data
    chart_dat_test[[length(chart_dat_test)+1]] <- cbind.data.frame(
                                              x       = seq_along(preds_test[chart_examples[i],,1]),
                                              preds   = preds_test[chart_examples[i],,1],
                                              actuals = actuals_test[chart_examples[i],,1])

    chart_dat_train[[length(chart_dat_train)+1]]<- cbind.data.frame(
                                              x       = seq_along(preds_train[chart_examples[i],,1]),
                                              preds   = preds_train[chart_examples[i],,1],
                                              actuals = actuals_train[chart_examples[i],,1])
  }

  #Min, max and avg over ALL examples
  Error_Min  = mean(apply(actuals_test, 1, FUN=min)  - apply(preds_test, 1, FUN=min))
  Error_Mean = mean(apply(actuals_test, 1, FUN=mean) - apply(preds_test, 1, FUN=mean))
  Error_Max  = mean(apply(actuals_test, 1, FUN=max)  - apply(preds_test, 1, FUN=max))

  # create chart function
  get_chart <- function(chart_dat, type_name, forecast_period){
    the_chart <- ggplot(data = chart_dat , aes(x=x/11))       +
                    geom_line(aes(y = preds,   col = "predict")) +
                    geom_line(aes(y = actuals, col = "actual"))     +
                    xlab("Forecast Day")                          +
                    ylab("O/D Balance")                 +
                    ggtitle(paste0(type_name,":",forecast_period,"Day Forecast"))
    return(the_chart)
  }

  # get charts
  chart_list <- list()

  for (i in 1:length(chart_examples)){

    chart_list[[length(chart_list)+1]] <- get_chart(chart_dat_train[[i]], "Train", forecast_period)

    chart_list[[length(chart_list)+1]] <- get_chart(chart_dat_test[[i]],  "Test",  forecast_period)
  }

return(list(chart_list, c(Error_Min, Error_Mean, Error_Max)))
}
```

```
set.seed(20180905)
random_examples14 <- round(runif(3, 1, dim(py$y_test_finaldate_14)[1]),0)

# Be sure to copy our data from python to R objects, so we can save it in a .RData file.
# Cannot save Python session.

preds_test14     <- py$preds_test_14
actuals_test14   <- py$y_test_finaldate_14

preds_train14    <- py$preds_train_14
actuals_train14  <- py$y_train_finaldate_14

Example_A14 <- plot_preds_train_vs_test(preds_test     = preds_test14,
                                        preds_train    = preds_train14,
                                        actuals_test   = actuals_test14,
                                                  actuals_train  = actuals_train14,
                                                  chart_example  = random_examples14,
                                                  forecast_period= 14)

library(gridExtra)
grid.arrange(Example_A14[[1]][[1]], Example_A14[[1]][[2]], Example_A14[[1]][[3]],
             Example_A14[[1]][[4]], Example_A14[[1]][[5]], Example_A14[[1]][[6]],
             ncol=2, nrow=3)
```



```
 #print min, avg, max errors
 print(paste0("Test set, mean error in 14day forecast of min  bank balance: ",
              round(Example_A14[[2]][1],0)))
```

```
## [1] "Test set, mean error in 14day forecast of min  bank balance: 458335"
```

As we expected, the test results indicate the model is not usable.

A Simpler Problem

Let's ask the model to solve as simple a problem as we can, whats the lowest bank balance we should expect in the forthcoming two weeks? So we'll train to a target of just one time step not fourteen*11. The time step will have just one output channel being:

– MinFortniteHence

It will take only 6 input channels (not 20, as previously):

– TransactionValue – AvgDaysToPay_Weighted – RollingBalance_Creditors – RollingBalance_Debtors – RollingBalance_Bank – MonthOfYr

We'll also simply the model by having two less 1Dconv layers (11 as opposed to 13). This improves the ratio of weights to examples, fewer weights and more examples. Why more examples? With a max dilation of 2^11 we will reach 1024 timesteps, which is approx 3mths, not one year.

Lastly, we will greatly increase the number of examples by having a smaller step between examples. Instead of sliding the window forward by 1 day (11 examples), we will slide it forward by half a day, 6 examples.

Hopefully, this will give us a better result in test

Distribution of MinFortniteHence

First let's extract the data and then plot its distribution, so we can see what an educated guess should look like. Our baseline is such a guess.

```
train_gen_simple = yield_xy(x_train_py,
                            y_train_py,
                            records_per_example = 1024,
                            batch_size          =   64,
                            step_size           =    5)

x_train_simple, y_train_simple = get_accumulated_generator(train_gen_simple,
                                                           "Train",
                                                           verbose = False)
test_gen_simple  = yield_xy(x_test_py,
                            y_test_py,
                            records_per_example = 1024,
                            batch_size          =  100,
                            step_size           =    5)

x_test_simple,  y_test_simple  = get_accumulated_generator(test_gen_simple,
                                                           "Test ",
                                                           verbose = False)
# Use only the followiong predictors:
# TransactionValue, AvgDaysToPay_Weighted, RollingBalance_Creditors,
# RollingBalance_Debtors, RollingBalance_Bank, MonthOfYr
x_train_6channels = x_train_simple[:, :, [1,2,5,6,7,8]]
x_test_6channels  = x_test_simple[:,  :, [1,2,5,6,7,8]]
# get single digit target for each example, MinFortniteHence
y_train_finaldate_1 = y_train_simple[:,-1, 2].reshape((y_train_simple.shape[0], 1, 1))
y_test_finaldate_1  = y_test_simple[:, -1, 2].reshape((y_test_simple.shape[0],  1, 1))
```

The above python generates the data, we'll plot distribution in the R package ggplot

```
# save the data, these tensors take a long time to build.
x_train_6channels   <- py$x_train_6channels
x_test_6channels    <- py$x_test_6channels
y_train_finaldate_1 <- py$y_train_finaldate_1
y_test_finaldate_1  <- py$y_test_finaldate_1

# We can this data if it takes a long time to build:
# save(x_train_6channels, x_test_6channels, y_train_finaldate_1, y_test_finaldate_1,
#      file = "Cash_Flow_Deep_simpler.RData")
# To load the data again
# load("data.RData")

# chart data distributions
chart_dat_MinFortniteHence = data.frame(Seq = 1:dim(y_train_finaldate_1)[1]/365,
                                        MinFortniteHence = y_train_finaldate_1)

f <- ggplot(chart_dat_MinFortniteHence,aes(x = MinFortniteHence)) +
      geom_histogram(fill = "blue") +
      ggtitle("Histogram: Min Bank Balance 14Days Hence, Train")

g <- ggplot(chart_dat_MinFortniteHence, aes(x=Seq, y=MinFortniteHence)) +
      geom_line() + xlab("Year in Sequence") +
      ggtitle("MinFortniteHence Over Time, 2008 to 2017")

library(gridExtra)
grid.arrange(g, f, ncol=2, nrow=1)
```

There is clearly a trend in the data, the bank balance is getting better, hopefully the model can learn this trend. Let's create our model, note it uses only 6 of the predictors in order to simplify the learning process.

```
model_cash1_shuf = create_model_cash(days_forecast =  1, multby11   = False, predictor_qty = 6,
                                     n_filters     = 16, n_channels = 1,     qty_dilations = 11,
                                     dropout_between_1Dconvs = True)
model_cash1_shuf.summary()
```

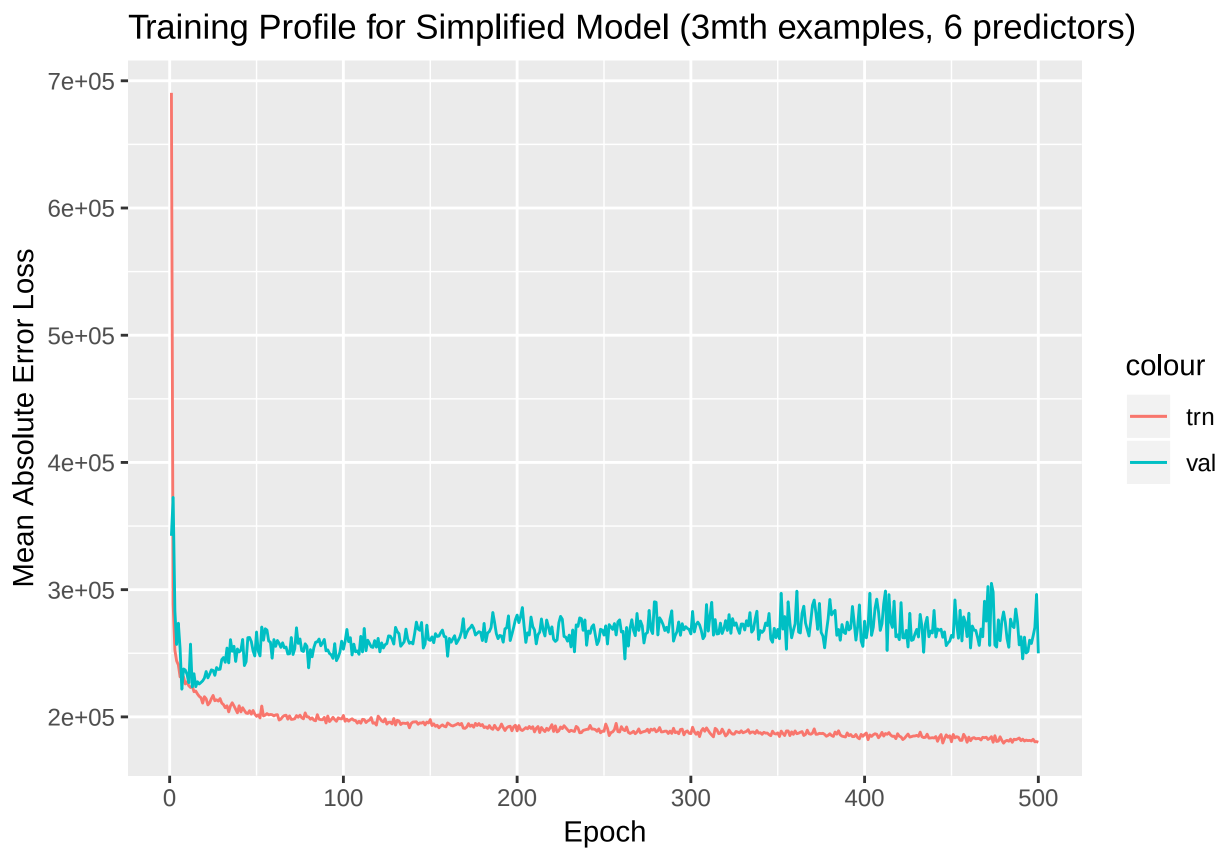
```
## _________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =================================================================
## input_11 (InputLayer)        (None, None, 6)           0
## _________________________________________________________________
## conv1d_86 (Conv1D)           (None, None, 16)          208
## _________________________________________________________________
## dropout_54 (Dropout)         (None, None, 16)          0
## _________________________________________________________________
## conv1d_87 (Conv1D)           (None, None, 16)          528
## _________________________________________________________________
## dropout_55 (Dropout)         (None, None, 16)          0
## _________________________________________________________________
## conv1d_88 (Conv1D)           (None, None, 16)          528
## _________________________________________________________________
## dropout_56 (Dropout)         (None, None, 16)          0
## _________________________________________________________________
## conv1d_89 (Conv1D)           (None, None, 16)          528
## _________________________________________________________________
## dropout_57 (Dropout)         (None, None, 16)          0
## _________________________________________________________________
## conv1d_90 (Conv1D)           (None, None, 16)          528
## _________________________________________________________________
## dropout_58 (Dropout)         (None, None, 16)          0
## _________________________________________________________________
## conv1d_91 (Conv1D)           (None, None, 16)          528
## _________________________________________________________________
## dropout_59 (Dropout)         (None, None, 16)          0
## _________________________________________________________________
## conv1d_92 (Conv1D)           (None, None, 16)          528
## _________________________________________________________________
## dropout_60 (Dropout)         (None, None, 16)          0
## _________________________________________________________________
## conv1d_93 (Conv1D)           (None, None, 16)          528
## _________________________________________________________________
## dropout_61 (Dropout)         (None, None, 16)          0
## _________________________________________________________________
## conv1d_94 (Conv1D)           (None, None, 16)          528
## _________________________________________________________________
## dropout_62 (Dropout)         (None, None, 16)          0
## _________________________________________________________________
## conv1d_95 (Conv1D)           (None, None, 16)          528
## _________________________________________________________________
## dropout_63 (Dropout)         (None, None, 16)          0
## _________________________________________________________________
## conv1d_96 (Conv1D)           (None, None, 16)          528
## _________________________________________________________________
## dropout_64 (Dropout)         (None, None, 16)          0
## _________________________________________________________________
## time_distributed_15 (TimeDis (None, None, 128)         2176
## _________________________________________________________________
## dropout_65 (Dropout)         (None, None, 128)         0
## _________________________________________________________________
## time_distributed_16 (TimeDis (None, None, 1)           129
## _________________________________________________________________
## lambda_8 (Lambda)            (None, None, 1)           0
## =================================================================
## Total params: 7,793
## Trainable params: 7,793
## Non-trainable params: 0
## _________________________________________________________________
```

Only 8,000 weights, a quarter as complex as the last model. Let's train that model.

```
model_cash1_shuf.compile(optimizer = Adam(),
                         loss      = 'mean_absolute_error')
epochs     = 500
batch_size =  64
history_cash1_shuf = model_cash1_shuf.fit(x               = x_train_6channels,
                                          y               = y_train_finaldate_1,
                                          batch_size      = batch_size,
                                          epochs          = epochs,
                                          validation_split= 0.2,
                                          shuffle         = True)
# save model
model_cash1_shuf.save('./SaveModels/model_cash1_shuf.h5')
# save history
import pickle
with open('./SaveModels/model_cash1_shuf_history.pickle', 'wb') as file_pi:
    pickle.dump(history_cash1_shuf.history, file_pi)
```

```
model_cash1_shuf   = load_model('./SaveModels/model_cash1_shuf.h5')
load_history       = open("./SaveModels/model_cash1_shuf_history.pickle","rb")
history_cash1_shuf = pickle.load(load_history)
toggplot_cash1_shuf_loss     = history_cash1_shuf['loss']
toggplot_cash1_shuf_loss_val = history_cash1_shuf['val_loss']
```

```
history_py_cash1_shuf_trn <- py$toggplot_cash1_shuf_loss
history_py_cash1_shuf_val <- py$toggplot_cash1_shuf_loss_val

plot_learning(history_py_cash_trn = history_py_cash1_shuf_trn,
              history_py_cash_val = history_py_cash1_shuf_val,
              title = "Training Profile for Simplified Model (3mth examples, 6 predictors)")
```



A Slightly less Simple Approach !

From the above chart we can see that we succesfully trained a model. Let's see if we can reduce mean error by increasing the duration to 6months per example and increase the predictors from 6 to all 20.

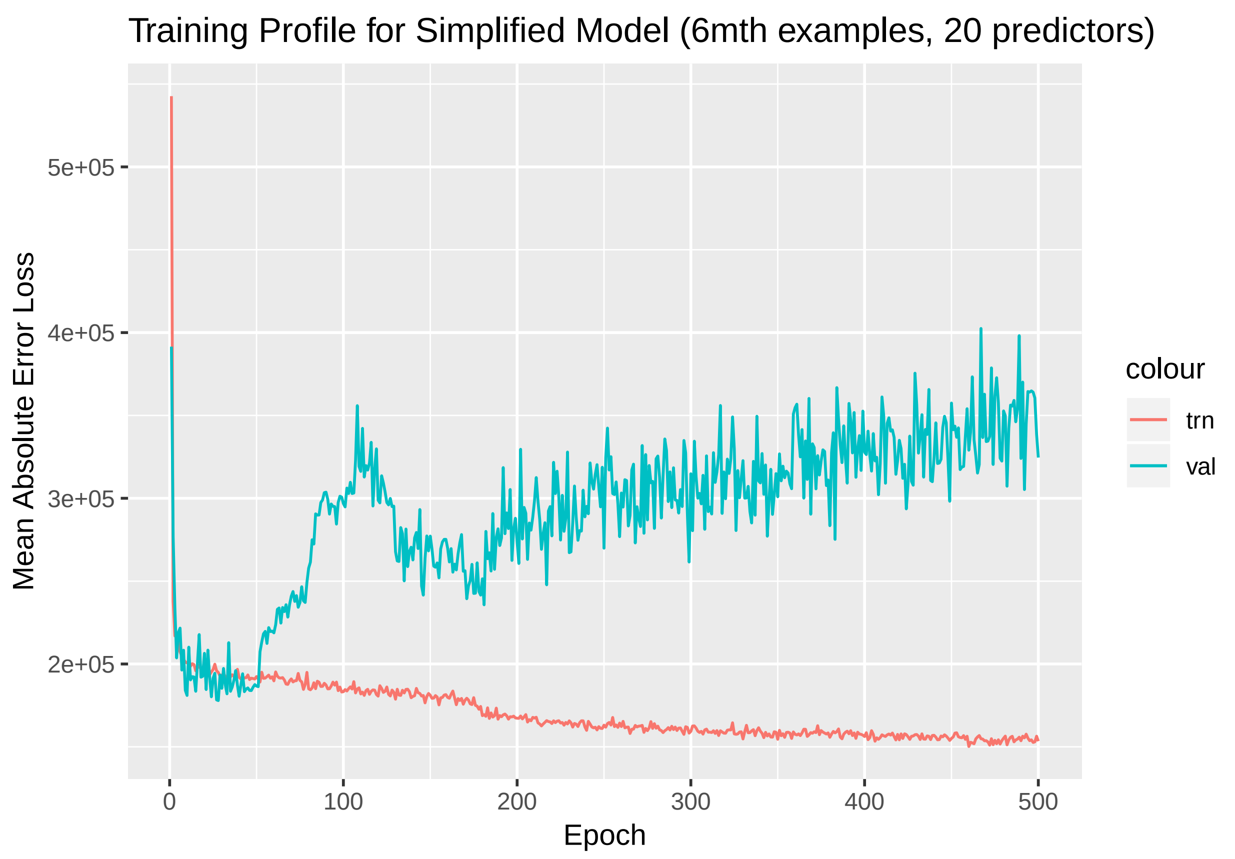
```
train_gen_simple_6mth = yield_xy(x_train_py,
                                 y_train_py,
                                 records_per_example = 2048,
                                 batch_size          =   64,
                                 step_size           =    5)

x_train_simple_6mth, y_train_simple_6mth = get_accumulated_generator(train_gen_simple_6mth,
                                                                     "Train",
                                                                     verbose = False)
test_gen_simple_6mth  = yield_xy(x_test_py,
                                 y_test_py,
                                 records_per_example = 2048,
                                    batch_size          =  100,
                                    step_size           =    5)

x_test_simple_6mth,  y_test_simple_6mth  = get_accumulated_generator(test_gen_simple_6mth,
                                                                     "Test ",
                                                                     verbose = False)
# get single digit target for each example
y_train_finaldate_1_6mth= y_train_simple_6mth[:,-1, 2].reshape((y_train_simple_6mth.shape[0], 1, 1))
y_test_finaldate_1_6mth = y_test_simple_6mth[:, -1, 2].reshape((y_test_simple_6mth.shape[0],  1, 1))
```

```
model_cash1_6mth = create_model_cash(days_forecast =  1, multby11      = False, predictor_qty = 20,
                                     n_filters     = 32, n_channels    = 1,     qty_dilations = 12,
                                     dropout_between_1Dconvs = True)
model_cash1_6mth.compile(optimizer = Adam(),
                         loss      = 'mean_absolute_error')
epochs     = 500
batch_size =  64
history_cash1_6mth = model_cash1_6mth.fit(x                = x_train_simple_6mth,
                                          y                = y_train_finaldate_1_6mth,
                                          batch_size       = batch_size,
                                          epochs           = epochs,
                                          validation_split= 0.2,
                                          shuffle          = True)
# save model
model_cash1_6mth.save('./SaveModels/model_cash1_6mth.h5')
# save history
import pickle
with open('./SaveModels/model_cash1_6mth_history.pickle', 'wb') as file_pi:
    pickle.dump(history_cash1_6mth.history, file_pi)
```

We'll load the results from history and view the training progress.

```
model_cash1_6mth   = load_model('./SaveModels/model_cash1_6mth.h5')
load_history       = open("./SaveModels/model_cash1_6mth_history.pickle","rb")
history_cash1_6mth = pickle.load(load_history)
toggplot_cash1_6mth_loss     = history_cash1_6mth['loss']
toggplot_cash1_6mth_loss_val = history_cash1_6mth['val_loss']
```

```
history_py_cash1_6mth_trn <- py$toggplot_cash1_6mth_loss
history_py_cash1_6mth_val <- py$toggplot_cash1_6mth_loss_val

plot_learning(history_py_cash_trn = history_py_cash1_6mth_trn,
              history_py_cash_val = history_py_cash1_6mth_val,
              title = "Training Profile for Simplified Model (6mth examples, 20 predictors)")
```



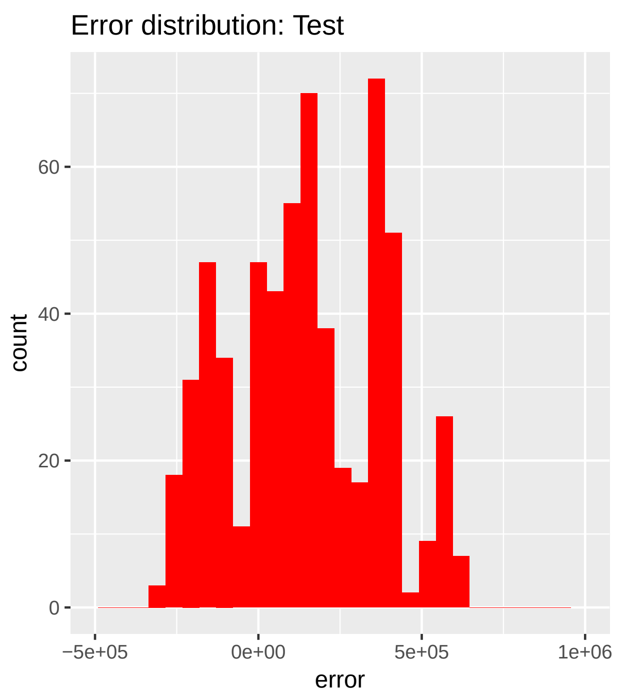
Better than before, at approx 45 epochs. But still not very impressive as the error looks to be around £200k before validation diverges from training. Nevertheless let's rebuild to 45 epochs and then see how it performs on test.

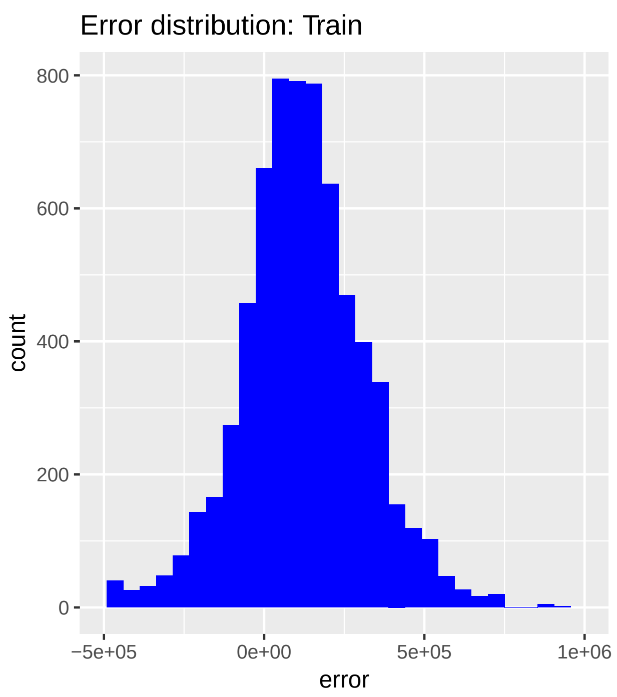
```
model_cash1_6mth45epc = create_model_cash(days_forecast =  1, multby11      = False,
                                          predictor_qty = 20, qty_dilations = 12,
                                          n_filters     = 32, n_channels    = 1,
                                          dropout_between_1Dconvs = True)

model_cash1_6mth45epc.compile(optimizer = Adam(),
                              loss      = 'mean_absolute_error')
epochs     =  50
batch_size =  64
history_cash1_6mth45epc = model_cash1_6mth45epc.fit(x                = x_train_simple_6mth,
                                                    y                = y_train_finaldate_1_6mth,
                                                    batch_size       = batch_size,
                                                    epochs           = epochs,
                                                    validation_split= 0.2,
                                                    shuffle          = True)
# save model
model_cash1_6mth45epc.save('./SaveModels/model_cash1_6mth45epc.h5')
# save history
import pickle
with open('./SaveModels/model_cash1_6mth45epc_history.pickle', 'wb') as file_pi:
    pickle.dump(history_cash1_6mth45epc.history, file_pi)
```

```
model_cash1_6mth45epc   = load_model('./SaveModels/model_cash1_6mth45epc.h5')
load_history            = open("./SaveModels/model_cash1_6mth45epc_history.pickle","rb")
history_cash1_6mth45epc = pickle.load(load_history)
preds_train_1_6mth45epc = model_cash1_6mth45epc.predict(x_train_simple_6mth)
preds_test_1_6mth45epc  = model_cash1_6mth45epc.predict(x_test_simple_6mth)
```

```
preds_test1_6mth45epc    <- py$preds_test_1_6mth45epc
actuals_test1_6mth45epc  <- py$y_test_finaldate_1_6mth

preds_train1_6mth45epc   <- py$preds_train_1_6mth45epc
actuals_train1_6mth45epc <- py$y_train_finaldate_1_6mth

chart_dat_test <- cbind.data.frame(
                              type  = "Test",
                              error = as.vector(preds_test1_6mth45epc  - actuals_test1_6mth45epc)
                              )
chart_dat_train<- cbind.data.frame(
                              type  = "Train",
                              error = as.vector(preds_train1_6mth45epc - actuals_train1_6mth45epc)
                              )
chart_dat      <- rbind(chart_dat_test, chart_dat_train)

u <- ggplot(chart_dat,aes(x=error)) +
        geom_histogram(data=subset(chart_dat,type == 'Test'), fill = "red") +
        ggtitle("Error distribution: Test") +
        xlim(-500000, 1000000)

v <- ggplot(chart_dat,aes(x=error)) +
        geom_histogram(data=subset(chart_dat,type == 'Train'),fill = "blue") +
        ggtitle("Error distribution: Train") +
        xlim(-500000, 1000000)

library(gridExtra)
grid.arrange(u,v, ncol=2, nrow=1)
```

```
## Warning: Removed 1 rows containing missing values (geom_bar).

## Warning: Removed 201 rows containing non-finite values (stat_bin).

## Warning: Removed 1 rows containing missing values (geom_bar).
```





There is some resemblence between these distirbutions. We could do a formal test, but instead, let's simply compare predicitions with actuals on the test set:

```
the_rownames<- c("MinBalance14Days")

results_tab <- cbind.data.frame(
                            Type          = the_rownames[1:dim(preds_train1_shuf)[3]],
                            MeanErr_Train = as.vector(colMeans(preds_train1_shuf - actuals_train1_shuf)),
                            StdDev_Train  = as.vector(sd(preds_train1_shuf - actuals_train1_shuf)),
                            MeanErr_Test  = as.vector(colMeans(preds_test1_shuf  - actuals_test1_shuf)),
                            StdDev_Test   = as.vector(sd(preds_test1_shuf  - actuals_test1_shuf))
                            )

kable(results_tab)
```

Type	MeanErr_Train	StdDev_Train	MeanErr_Test	StdDev_Test
MinBalance14Days	303560.5	226991.9	85846.51	212555.3

A mean error of £140k and SD of £230k is far better than our first model, but remains little better than an educated guess.

Adding dropout between the 1DConv layers causes the model to 'forget' some data which is an odd approach to time series, but it does appear to regularise the results.

Note, we can't chart examples in the same way as our first example because there is only one figure, the minimum forecast for the coming 14 days, to chart.

Conclusion on Wavenet Models for Cash Flow Forecasting

These models are training but are nowhere near providing useful forecasts. We must assume either :

a) the architecture is unsuitable for the problem
b) there are insufficient examples to train 1Dconv nets for this task
c) the data does not include the information needed to make useful forecasts

We don't have much development time, but can attmept to elimiate possibility (a) by considering an LSTM model.

LSTM Architecture

A more common approach to time series models is the LSTM. Introduction to LSTM's in Keras at https://machinelearningmastery.com/keras-functional-api-deep-learning/

Keras LSTM models are straight forward to create, although it will be immediately apparent as to why the 1DConvs were einitially investigated. LSTM's require more weights per time period than dilated 1DConv's. The largest model we could attempt using an LSTM will have size=128, meaning it will look back 128 time steps. This is just 12 days.

```
from keras.models import Model
from keras.layers import Input, Dense
from keras.layers.recurrent import LSTM
from keras.layers.wrappers import TimeDistributed
from keras.optimizers import Adam, TFOptimizer
def create_model_cash_LSTM(n_channels = 1, LSTM_size = 128, stacked = False):

  #parameters
  predictor_qty = 20
  # input layer
  # expected input data shape: (batch_size, timesteps, data_dim)
  input_sequence = Input(shape=(None, predictor_qty))
  x = input_sequence

  # feature extraction
  # params = inputs * outputs + outputs
  if (stacked):
    x = LSTM(LSTM_size, return_sequences=True)(x)
  # stacked layer for more complex feature identification
  x = LSTM(LSTM_size)(x)
  x = Dense(LSTM_size, activation='relu')(x)
  output = Dense(n_channels)(x)

  # output
  model = Model(inputs  = input_sequence,
              outputs = output)
  return(model)
```

```
#let's build a model for fortnite forecasts first
model_cash1_LSTM = create_model_cash_LSTM(n_channels = 1, LSTM_size = 64)
model_cash1_LSTM.summary()
```

```
## _________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =================================================================
## input_12 (InputLayer)        (None, None, 20)          0
## _________________________________________________________________
## lstm_4 (LSTM)                (None, 64)                21760
## _________________________________________________________________
## dense_23 (Dense)             (None, 64)                4160
## _________________________________________________________________
## dense_24 (Dense)             (None, 1)                 65
## =================================================================
## Total params: 25,985
## Trainable params: 25,985
## Non-trainable params: 0
## _________________________________________________________________
```

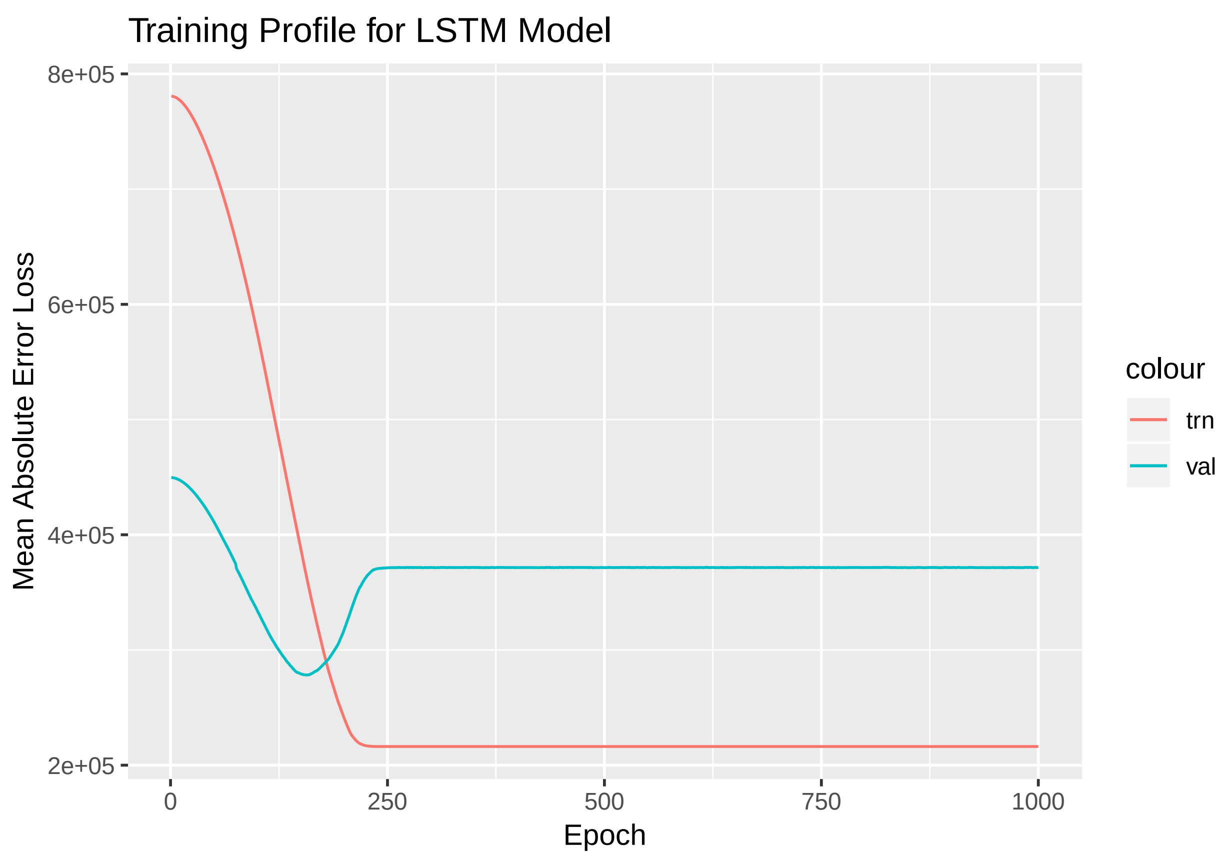
This new model need targets in 2D, not 3D.

```
# get single digit target for each example
y_train_finaldate_1_LSTM = y_train_finaldate_1[:,:,0]
y_test_finaldate_1_LSTM  = y_test_finaldate_1[:,:,0]
```

```
model_cash1_LSTM.compile(optimizer = Adam(),
                         loss      = 'mean_absolute_error')
epochs     = 1000
batch_size =   64
history_cash1_LSTM = model_cash1_LSTM.fit(x                = x_train_simple,
                                          y                = y_train_finaldate_1_LSTM,
                                          batch_size       = batch_size,
                                          epochs           = epochs,
                                          validation_split= 0.2,
                                          shuffle          = True)
# save model
model_cash1_LSTM.save('./SaveModels/model_cash1_LSTM.h5')
# save training history
with open('./SaveModels/model_cash1_LSTM_history.pickle', 'wb') as file_pi:
    pickle.dump(history_cash1_LSTM.history, file_pi)
```

```
model_cash1_LSTM   = load_model('./SaveModels/model_cash1_LSTM.h5')
load_history       = open("./SaveModels/model_cash1_LSTM_history.pickle","rb")
history_cash1_LSTM = pickle.load(load_history)
toggplot_cash1_LSTM_loss     = history_cash1_LSTM['loss']
toggplot_cash1_LSTM_loss_val = history_cash1_LSTM['val_loss']
```

```
history_py_cash1_LSTM_trn <- py$toggplot_cash1_LSTM_loss
history_py_cash1_LSTM_val <- py$toggplot_cash1_LSTM_loss_val

plot_learning(history_py_cash_trn = history_py_cash1_LSTM_trn,
              history_py_cash_val = history_py_cash1_LSTM_val,
              title = "Training Profile for LSTM Model")
```

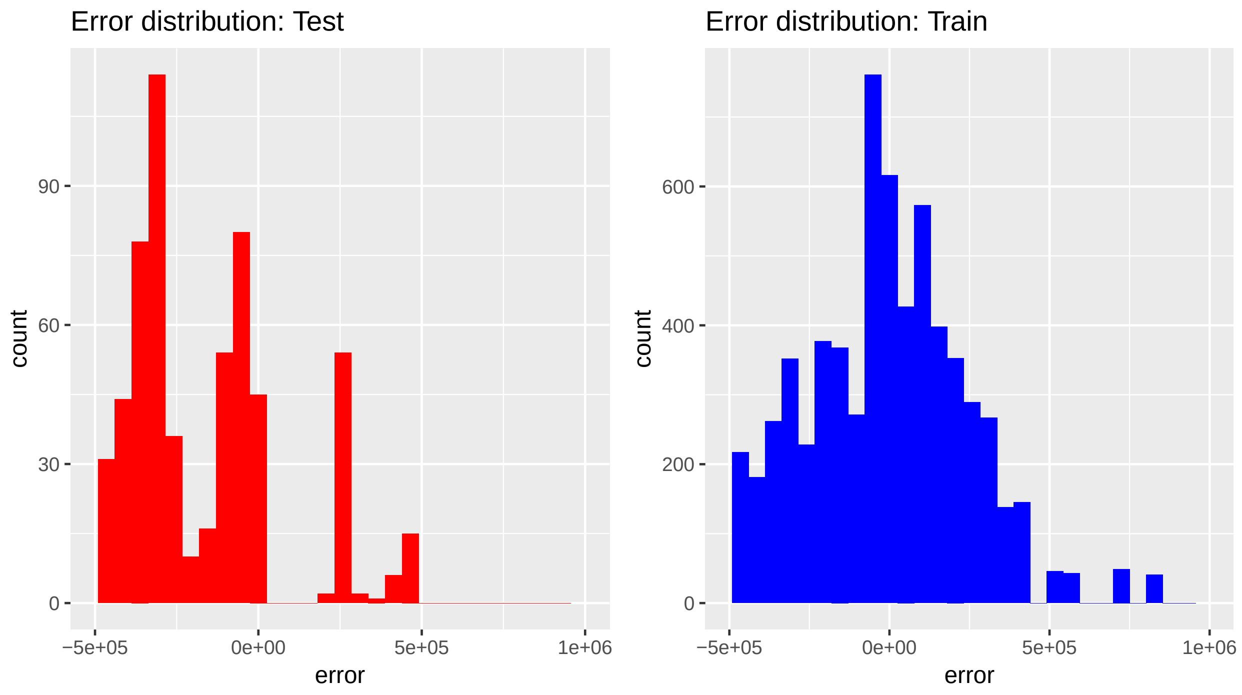


```
preds_train_1_LSTM = model_cash1_LSTM.predict(x_train_simple)
preds_test_1_LSTM  = model_cash1_LSTM.predict(x_test_simple)
```

```
preds_test1_LSTM    <- py$preds_test_1_LSTM
actuals_test1_LSTM  <- py$y_test_finaldate_1_LSTM

preds_train1_LSTM   <- py$preds_train_1_LSTM
actuals_train1_LSTM <- py$y_train_finaldate_1_LSTM

chart_dat_test <- cbind.data.frame(type  = "Test",
                                   error = as.vector(preds_test1_LSTM  - actuals_test1_LSTM))
chart_dat_train<- cbind.data.frame(type  = "Train",
                                   error = as.vector(preds_train1_LSTM - actuals_train1_LSTM))
chart_dat      <- rbind(chart_dat_test, chart_dat_train)

w <- ggplot(chart_dat,aes(x=error)) +
        geom_histogram(data=subset(chart_dat,type == 'Test'), fill = "red") +
        ggtitle("Error distribution: Test") +
        xlim(-500000, 1000000)

x <- ggplot(chart_dat,aes(x=error)) +
        geom_histogram(data=subset(chart_dat,type == 'Train'),fill = "blue") +
        ggtitle("Error distribution: Train") +
        xlim(-500000, 1000000)

library(gridExtra)
```

```r
grid.arrange(w,x, ncol=2, nrow=1)
```

```
## Warning: Removed 212 rows containing non-finite values (stat_bin).

## Warning: Removed 1 rows containing missing values (geom_bar).

## Warning: Removed 566 rows containing non-finite values (stat_bin).

## Warning: Removed 1 rows containing missing values (geom_bar).
```



We could retrain the model to get the weights where the model scores the minimum error, which is at approx 140 epochs. However, even then, the mean absolute error is substantially higher than possible with the 1Dconv models. This first endeavour with LSTM's has not improved upon our existing models.

Overall Conclusion

Insufficient data for training!

We could further simplify the problem by changing it into a binary classification problem. For example we could ask the model "will we exceed some threshold in the coming 14 days?". That may be of some use, but is currently judged not worth the effort.

With regret, this investigation closes without a model which can be deployed to production.

APPENDIX. SQL for Data Extraction from Accounting System

The following T-SQL (for SQL Server 2016) was created to extract the cash transactions, debtors and creditors from the accounting system. The above notebook executes this T-SQL when is calls the SQL stored procedure in the second code chunk.

```
DECLARE @MonthEndBeforeFirstDay DATETIME  = NULL;
DECLARE @OpeningBal_2150100 NUMERIC(12,2) = NULL;
DECLARE @OpeningBal_1850100 NUMERIC(12,2) = NULL;
DECLARE @OpeningBal_1950100 NUMERIC(12,2) = NULL;

--Prepare temp table
IF OBJECT_ID(N'tempdb..#tmp_StartingValues', N'U') IS NOT NULL
DROP TABLE [#tmp_StartingValues];

-- get starting values
SELECT
    ACCOUNT_NO, TRANS_DATE, BC_TRANS_VALUE
INTO
    [#tmp_StartingValues]
FROM
(
    SELECT ACCOUNT_NO, TRANS_DATE, BC_TRANS_VALUE, TRANS_DESC,
            ROW_NUMBER() OVER(PARTITION BY ACCOUNT_NO ORDER BY TRANS_DATE, ACCOUNT_NO) AS Seq
    FROM
        PUBLIC_NOMINAL_TRANSACTION_HISTORY
    WHERE
            TRANS_TYPE IN (N'bf') --bf means 'month ends', literally brought forward
            AND
            ACCOUNT_NO IN('2150100', -- purch ledger ctrl (creditors)
                          '1850100', -- sales ledger ctrl (debtors)
                          '1950100') -- bank current a/c
) As MonthEnds
WHERE
    Seq = 1 --we only want the starting value, the first in the sequence for each nominal account_no.

-- get those starting values...
SELECT @OpeningBal_2150100 = BC_TRANS_VALUE FROM [#tmp_StartingValues] WHERE ACCOUNT_NO = '2150100';
SELECT @OpeningBal_1850100 = BC_TRANS_VALUE FROM [#tmp_StartingValues] WHERE ACCOUNT_NO = '1850100';
SELECT @OpeningBal_1950100 = BC_TRANS_VALUE FROM [#tmp_StartingValues] WHERE ACCOUNT_NO = '1950100';
SELECT @MonthEndBeforeFirstDay = MIN(TRANS_DATE) FROM [#tmp_StartingValues];

--proceed with main task of building table of transactions, day by day

WITH
AllNominals
AS
(
    SELECT Src.ACCOUNT_NO, Src.TRANS_DATE, SYSTEM_DATE, Src.BC_TRANS_VALUE,
           Src.DOCUMENT_REF, Src.TRANS_TYPE, Src.TRANS_DESC
    FROM
        PUBLIC_NOMINAL_TRANSACTION_HISTORY AS Src
        LEFT OUTER JOIN
        [16_NominalTransactionHistoryMajorKey] AS Decode
        ON
        Src.ACCOUNT_NO       = Decode.ACCOUNT_NO
        AND Src.TRANS_DATE   = Decode.TRANS_DATE
        AND Src.DOCUMENT_REF = Decode.DOCUMENT_REF
    WHERE
```

```
            (NOT (Src.TRANS_TYPE IN (N'bf', N'by')))
            AND
            Src.ACCOUNT_NO IN('2150100', -- purch ledger ctrl (creditors)
                              '1850100', -- sales ledger ctrl (debtors)
                              '1950100') -- bank current a/c
            AND
            Decode.InternalOrExternal = 'External'

    UNION ALL

    SELECT Src.ACCOUNT_NO, Src.TRANS_DATE, SYSTEM_DATE, Src.BC_TRANS_VALUE, Src.DOCUMENT_REF,
           Src.TRANS_TYPE, Src.TRANS_DESCR
    FROM
        PUBLIC_NOMINAL_TRANSACTIONS AS Src
        LEFT OUTER JOIN
        [17_NominalTransactionCurrentMajorKey] As Decode
        ON
        Src.ACCOUNT_NO       = Decode.ACCOUNT_NO
        AND Src.TRANS_DATE   = Decode.TRANS_DATE
        AND Src.DOCUMENT_REF = Decode.DOCUMENT_REF
    WHERE
            (NOT (Src.TRANS_TYPE IN (N'bf', N'by')))
            AND
            Src.ACCOUNT_NO IN('2150100', -- purch ledger ctrl (creditors)
                              '1850100', -- sales ledger ctrl (debtors)
                              '1950100') -- bank current a/c
            AND
            Decode.InternalOrExternal = 'External'
)
,
Bank_1
AS
(
    --Get daily transaction total, null days where there is no data
    SELECT DISTINCT
        CASE WHEN ACCOUNT_NO = '1950100'
        -- bank transaction take place that day, disregard reconciliation (system) date
             THEN CalendarDay
             ELSE
                CASE WHEN SYSTEM_DATE > CalendarDay
                -- cannot know about the future for debtors/creditors
                     THEN SYSTEM_DATE
                     ELSE CalendarDay
                     END
             END
             AS CalendarDay,
        ACCOUNT_NO,
        SUM(COALESCE(BC_TRANS_VALUE,0)) AS DailyTotal
    FROM
        Decode_FinancialDays
        LEFT OUTER JOIN
        AllNominals
        ON Decode_FinancialDays.CalendarDay = AllNominals.TRANS_DATE
    WHERE
        CalendarDay > @MonthEndBeforeFirstDay
        --must be after the first month end for which we have starting balances
        AND
        CalendarDay < DATEADD(dd,-30, GETDATE())
        --30 days in past, final month has no useful forecasts
    GROUP BY
        CASE WHEN ACCOUNT_NO = '1950100'
        -- bank transaction take place that day, disregard reconciliation (system) date
             THEN CalendarDay
             ELSE
                CASE WHEN SYSTEM_DATE > CalendarDay
                -- cannot know about the future for debtors/creditors
                     THEN SYSTEM_DATE
                     ELSE CalendarDay
                     END
             END,
        ACCOUNT_NO

)
,
Bank_2
AS
(
    SELECT DISTINCT
        CalendarDay,
        EOMONTH(CalendarDay) AS MonthEnd,
        SUM(CASE WHEN ACCOUNT_NO = '1950100' THEN DailyTotal ELSE 0 END)
            OVER (ORDER BY CalendarDay) + @OpeningBal_1950100
            AS RollingBalance_Bank,

        SUM(CASE WHEN ACCOUNT_NO = '2150100' THEN DailyTotal ELSE 0 END)
            OVER (ORDER BY CalendarDay) + @OpeningBal_2150100
            AS RollingBalance_Creditors,

        SUM(CASE WHEN ACCOUNT_NO = '1850100' THEN DailyTotal ELSE 0 END)
            OVER (ORDER BY CalendarDay) + @OpeningBal_1850100
            AS RollingBalance_Debtors
    FROM
        Bank_1
)
,
Bank_3
AS
(
    SELECT
        CalendarDay,
        RollingBalance_Creditors,
        RollingBalance_Debtors,
        RollingBalance_Bank,
        MAX(RollingBalance_Bank) OVER (ORDER BY CalendarDay ROWS BETWEEN CURRENT ROW AND 14 FOLLOWING)
          AS MaxFortniteHence,
        MIN(RollingBalance_Bank) OVER (ORDER BY CalendarDay ROWS BETWEEN CURRENT ROW AND 14 FOLLOWING)
```

```
          AS MinFortniteHence,
        AVG(RollingBalance_Bank) OVER (ORDER BY CalendarDay ROWS BETWEEN CURRENT ROW AND 14 FOLLOWING)
          AS AvgFortniteHence,
        MAX(RollingBalance_Bank) OVER (ORDER BY CalendarDay ROWS BETWEEN CURRENT ROW AND 30 FOLLOWING)
          AS MaxMonthHence,
        MIN(RollingBalance_Bank) OVER (ORDER BY CalendarDay ROWS BETWEEN CURRENT ROW AND 30 FOLLOWING)
          AS MinMonthHence,
        AVG(RollingBalance_Bank) OVER (ORDER BY CalendarDay ROWS BETWEEN CURRENT ROW AND 30 FOLLOWING)
          AS AvgMonthHence,
        LEAD(RollingBalance_Bank, 14, 0) OVER(ORDER BY CalendarDay) AS RollingBalance_Bank14Hence,
        LEAD(RollingBalance_Bank, 30, 0) OVER(ORDER BY CalendarDay) AS RollingBalance_Bank30Hence
    FROM
        Bank_2
)
,
SaleAndPurchaseTransactions
AS
(
    SELECT
        CASE AllNominals.ACCOUNT_NO
            WHEN '2150100' THEN 'Purchase'
            WHEN '1850100' THEN 'Sale'
            ELSE 'Unknown'
            END AS TransType,
        AllNominals.TRANS_DATE  AS TransactionDate,
        AllNominals.SYSTEM_DATE AS SystemDate,
        Decode_NominalTransactionTypes.[description]   AS TransactionType,
        AllNominals.BC_TRANS_VALUE As TransactionValue,
        AllNominals.DOCUMENT_REF AS TransactionDocRef

    FROM
        AllNominals

        INNER JOIN
        PUBLIC_NOMINAL_MASTER
        ON AllNominals.ACCOUNT_NO = PUBLIC_NOMINAL_MASTER.ACCOUNT_NO

        LEFT OUTER JOIN
        Decode_NominalTransactionTypes
        ON AllNominals.TRANS_TYPE = Decode_NominalTransactionTypes.code
    WHERE
        AllNominals.ACCOUNT_NO IN('2150100','1850100')
)
,
TransactionsWithAccount
AS
(
    SELECT
        TransType,
        TransactionDate,
        SystemDate,
        TransactionType,
        TransactionValue,
        TransactionDocRef,
        INVOICE_TYPE,
        LTRIM(COALESCE(CAST(AllSuppTrans.ACCOUNT_NO AS VARCHAR),
                       CAST(AllCustTrans.ACCOUNT_NO AS VARCHAR))) AS Account
    FROM
        SaleAndPurchaseTransactions
        LEFT OUTER JOIN
        (
            SELECT TRANS_DATE, DOCUMENT_REF, TRANS_TYPE, TRANS_VALUE#1, ACCOUNT_NO
            FROM PUBLIC_SUPPLIER_TRANSACTIONS
            UNION
            SELECT TRANS_DATE, DOCUMENT_REF, TRANS_TYPE, TRANS_VALUE#1, ACCOUNT_NO
            FROM PUBLIC_SUPPLIER_TRANSACTION_HISTORY
        ) AS AllSuppTrans
        ON  SaleAndPurchaseTransactions.TransactionDate   = AllSuppTrans.TRANS_DATE
        AND SaleAndPurchaseTransactions.TransactionDocRef = AllSuppTrans.DOCUMENT_REF
        AND SaleAndPurchaseTransactions.TransactionValue  = AllSuppTrans.TRANS_VALUE#1

        LEFT OUTER JOIN
        (
            SELECT TRANS_DATE, TRANS_REF, TRANS_TYPE, TRANS_VALUE#1, ACCOUNT_NO, INVOICE_TYPE
            FROM PUBLIC_CUSTOMER_TRANSACTIONS
            UNION
            SELECT TRANS_DATE, TRANS_REF, TRANS_TYPE,       VALUE#1, ACCOUNT_NO, INVOICE_TYPE
            FROM PUBLIC_CUSTOMER_TRANSACTION_HISTORY
        ) AS AllCustTrans
        ON  SaleAndPurchaseTransactions.TransactionDate   = AllCustTrans.TRANS_DATE
        AND SaleAndPurchaseTransactions.TransactionDocRef = AllCustTrans.TRANS_REF
        AND SaleAndPurchaseTransactions.TransactionValue  = AllCustTrans.TRANS_VALUE#1
)
,
OtherTransactions --wages, bank charges, finance movements, vat in/out etc.
AS
(
    SELECT DISTINCT
            'NonSaleOrPurchase'        AS TransType,
            CalendarDay                AS TransactionDate,
            CalendarDay                AS SystemDate,
            --otherwise the postingdealy_days is skewed by bank reconciliation dates
            'Manual Journal'           AS TransactionType,
            COALESCE(BC_TRANS_VALUE,0) AS TransactionValue,
            BankNoms.DOCUMENT_REF      AS TransactionDocRef,
            'NA'                       AS INVOICE_TYPE,
            '0000'                     AS Account
        FROM
            Decode_FinancialDays
            LEFT OUTER JOIN
            (SELECT * FROM AllNominals WHERE ACCOUNT_NO = '1950100') AS BankNoms
            ON Decode_FinancialDays.CalendarDay = BankNoms.TRANS_DATE
        WHERE
            CalendarDay > CONVERT(DATETIME, '2009-09-30', 102)
            AND
            CalendarDay < DATEADD(dd,-30, GETDATE()) --30 days in past, final month has no useful forec
            AND
            LEFT(BankNoms.DOCUMENT_REF,1) = 'N'
)
,
FirstTemplate_ExcMeans
AS
(
SELECT
    TransType,
    TransactionDate,
    SystemDate,
    TransactionType,
    TransactionValue,
    TransactionDocRef,

    CASE TransType
         WHEN 'Purchase'
            THEN 'Not a Sale'
         WHEN 'NonSaleOrPurchase'
            THEN 'Not a Sale'
         ELSE COALESCE([Decode_Invoice_Type_Groups].[Group], 'Unspecified')
         END
         AS SalesCategory,

    CASE
        WHEN TransType = 'Sale'
            THEN 'Not a Purchase'
        WHEN TransType = 'NonSaleOrPurchase'
            THEN 'Not a Purchase'
        WHEN Account IN ('8609', '1196')
            THEN 'Finance'
        WHEN Account IN ('2610','2481','1765','1758','1649','1353','1336','1328','1322','1292','1233',
                         '1225','1133','1114','1113','1107')
            THEN 'Machinery'
        WHEN Account IN ('8702','8526','2625','2474','2064','2054','1993','1962','1956','1867','1781',
                         '1773','1760','1742','1727','1494','1391','1290','1262','1261','1259','1258',
                         '1250','1206','1191','1183','1176','1151','1150','1131','1101','1100')
            THEN 'Parts'
        ELSE 'Unspecified'
        END
        AS PurchaseCategory,

    Account,

    CASE TransType
        WHEN 'Sale'     THEN PUBLIC_CUSTOMER_MASTER.AVE_DAYS_TO_PAY
        WHEN 'Purchase' THEN PUBLIC_SUPPLIER_MASTER.AVE_DAYS_TO_PAY
        WHEN 'NonSaleOrPurchase' THEN 3
        ELSE 'Unknown'
        END
        AS AvgDaysToPay,

    CASE TransType
```

```
        WHEN 'Sale'     THEN PUBLIC_CUSTOMER_MASTER.AVE_AMOUNT_PAID
        WHEN 'Purchase' THEN PUBLIC_SUPPLIER_MASTER.AVE_PAID_AMOUNT
        WHEN 'NonSaleOrPurchase' THEN 5000
        ELSE 'Unknown'
        END
        AS AvgValuePaid,

    CASE TransType
        WHEN 'Sale'     THEN PUBLIC_CUSTOMER_MASTER.NO_OF_PAYMENTS
        WHEN 'Purchase' THEN PUBLIC_SUPPLIER_MASTER.NO_OF_PAYMENTS
        WHEN 'NonSaleOrPurchase' THEN 300
        ELSE 'Unknown'
        END
        AS QtyOfPayments
FROM
    (
        SELECT * FROM TransactionsWithAccount
        UNION ALL
        SELECT * FROM OtherTransactions
    )
    AS AllTransactions

    LEFT OUTER JOIN
    PUBLIC_CUSTOMER_MASTER
    ON AllTransactions.Account = LTRIM(PUBLIC_CUSTOMER_MASTER.ACCOUNT_NO)

    LEFT OUTER JOIN
    PUBLIC_SUPPLIER_MASTER
    ON AllTransactions.Account = LTRIM(PUBLIC_SUPPLIER_MASTER.ACCOUNT_NO)

    LEFT OUTER JOIN
    [Decode_Invoice_Type_Groups]
    ON AllTransactions.INVOICE_TYPE = [Decode_Invoice_Type_Groups].INVOICE_TYPE

)
,
Mean_ValuePaidAndQtyOfPayments_ForMissingData
AS
(
    SELECT
        TransType,
        ROUND(AVG(TransactionValue),-1) AS AvgValuePaid_ForMissingData,
        ROUND(COUNT(TransactionValue)
              /
              COUNT(DISTINCT Account)
              ,-1)
              AS QtyOfPayments_ForMissingData
    FROM
        FirstTemplate_ExcMeans
    WHERE
        --missing data is defined as:
        AvgDaysToPay = 0
        OR
        AvgDaysToPay > 65000 -- seems to be an Ibcos default which must be rubbish
        OR
        AvgDaysToPay IS NULL
    GROUP BY
        TransType
)
,
Mean_DaysToPay_ForMissingData
AS
--Cannot calculate indepenently, must simply use average of those transactions where we have data
--(NOT missing)
(
    SELECT
        TransType,
        ROUND(SUM(AvgDaysToPay * ABS(TransactionValue))
              /
              SUM(ABS(TransactionValue))
              ,0)
              AS AvgDaysToPay_ForMissingData
    FROM
        FirstTemplate_ExcMeans
    WHERE
        --NOT missing data is defined as:
        AvgDaysToPay != 0
        AND
        AvgDaysToPay < 65000 -- seems to be an Ibcos default which must be rubbish
        AND
        AvgDaysToPay IS NOT NULL
    GROUP BY
        TransType
)
,
SecondTemplate
AS
(
    SELECT
        FirstTemplate_ExcMeans.TransType AS SaleOrPurchase,
        TransactionDate,
        DATEDIFF(dd, TransactionDate, SystemDate) AS PostingDelayDays,
        TransactionType,
        TransactionValue,
        TransactionDocRef,
        SalesCategory,
        PurchaseCategory,
        Account,

        CASE WHEN TransactionType = 'Sales Cash Received'
                  OR
                  TransactionType = 'Purchase Cash Payment'
             THEN 3  --cash payments clear quick!
             ELSE
                -- handle missing data
                CASE WHEN AvgDaysToPay = 0 OR AvgDaysToPay > 65000 OR AvgDaysToPay IS NULL
```

```
                     THEN Mean_DaysToPay_ForMissingData.AvgDaysToPay_ForMissingData
                     ELSE AvgDaysToPay
                     END
             END
             AS AvgDaysToPay,

        COALESCE(AvgValuePaid,
                 Mean_ValuePaidAndQtyOfPayments_ForMissingData.AvgValuePaid_ForMissingData)
                 AS AvgValuePaid,

        COALESCE(QtyOfPayments,
                 Mean_ValuePaidAndQtyOfPayments_ForMissingData.QtyOfPayments_ForMissingData)
                 AS QtyOfPayments

    FROM
        FirstTemplate_ExcMeans

        LEFT OUTER JOIN
        Mean_DaysToPay_ForMissingData
        ON FirstTemplate_ExcMeans.TransType = Mean_DaysToPay_ForMissingData.TransType

        LEFT OUTER JOIN
        Mean_ValuePaidAndQtyOfPayments_ForMissingData
        ON FirstTemplate_ExcMeans.TransType = Mean_ValuePaidAndQtyOfPayments_ForMissingData.TransType

)
,
DistinctTransactionTypes
AS
(
    SELECT DISTINCT
        SaleOrPurchase,
        SalesCategory,
        PurchaseCategory
    FROM
        SecondTemplate
    WHERE
        NOT(SaleOrPurchase = 'Purchase' AND SalesCategory    = 'Not a Sale')
        OR
        NOT(SaleOrPurchase = 'Sale'     AND PurchaseCategory = 'Not a Purchase')
)
,
DistinctTransactionTypesAndDays
AS
(
    SELECT
        CalendarDay,
        SaleOrPurchase,
        SalesCategory,
        PurchaseCategory
    FROM
        DistinctTransactionTypes
        CROSS JOIN
        Decode_FinancialDays
    WHERE
        CalendarDay > CONVERT(DATETIME, '2009-09-30', 102)
        AND
        CalendarDay < GETDATE()
)

SELECT
    DistinctTransactionTypesAndDays.CalendarDay,

    CASE WHEN SUM(TransactionValue) = 0
         THEN 0
         ELSE SUM(COALESCE(PostingDelayDays,0) * ABS(TransactionValue))
              /
              SUM(ABS(TransactionValue))
         END
         AS PostingDelayDays_Weighted,

    DistinctTransactionTypesAndDays.SaleOrPurchase,

    CASE WHEN DistinctTransactionTypesAndDays.SaleOrPurchase = 'Sale'
         THEN DistinctTransactionTypesAndDays.SalesCategory
         ELSE DistinctTransactionTypesAndDays.PurchaseCategory
         END
         AS Category,
    TransactionType,
    COALESCE(SUM(TransactionValue),0) AS TransactionValue,

    CASE WHEN SUM(TransactionValue) = 0
         THEN AVG(AvgDaysToPay)
         --else weighted average
         ELSE SUM(AvgDaysToPay * ABS(TransactionValue))
                /
              SUM(ABS(TransactionValue))
         END
         AS AvgDaysToPay_Weighted,

    CASE WHEN SUM(TransactionValue) = 0
         THEN AVG(AvgValuePaid)
         --else weighted average
         ELSE SUM(AvgValuePaid * TransactionValue)
                /
              SUM(TransactionValue)
         END
         AS AvgValuePaid_Weighted,

    CASE WHEN SUM(TransactionValue) = 0
         THEN AVG(QtyOfPayments)
         -- else weigthed average
         ELSE SUM(QtyOfPayments* ABS(TransactionValue))
                /
              SUM(ABS(TransactionValue))
         END
        AS QtyOfPayments_Weighted,

    RollingBalance_Creditors,
    RollingBalance_Debtors,
    RollingBalance_Bank,

    RollingBalance_Bank14Hence,
    MinFortniteHence,
    AvgFortniteHence,
    MaxFortniteHence,

    RollingBalance_Bank30Hence,
    MinMonthHence,
    AvgMonthHence,
    MaxMonthHence

FROM
    DistinctTransactionTypesAndDays
    --Teamplate for transaction types ensures we have same trans types for all days,
    --even if zero transaction value
    LEFT OUTER JOIN
    SecondTemplate
    ON  DistinctTransactionTypesAndDays.SaleOrPurchase   = SecondTemplate.SaleOrPurchase
    AND DistinctTransactionTypesAndDays.SalesCategory    = SecondTemplate.SalesCategory
    AND DistinctTransactionTypesAndDays.PurchaseCategory = SecondTemplate.PurchaseCategory
    AND DistinctTransactionTypesAndDays.CalendarDay      = SecondTemplate.TransactionDate

    INNER JOIN
    Bank_3
    ON DistinctTransactionTypesAndDays.CalendarDay = Bank_3.CalendarDay

WHERE
    DistinctTransactionTypesAndDays.CalendarDay < DATEADD(dd,-30, GETDATE())

GROUP BY
    DistinctTransactionTypesAndDays.CalendarDay,
    DistinctTransactionTypesAndDays.SaleOrPurchase,

    CASE WHEN DistinctTransactionTypesAndDays.SaleOrPurchase = 'Sale'
         THEN DistinctTransactionTypesAndDays.SalesCategory
         ELSE DistinctTransactionTypesAndDays.PurchaseCategory
         END,

    TransactionType,
    RollingBalance_Creditors,
    RollingBalance_Debtors,
    RollingBalance_Bank,

    MinFortniteHence,
    AvgFortniteHence,
    MaxFortniteHence,

    MinMonthHence,
    AvgMonthHence,
    MaxMonthHence,

    RollingBalance_Bank14Hence,
    RollingBalance_Bank30Hence
ORDER BY
    CalendarDay,
    SaleOrPurchase,
    Category
```