Parts Ordering Optimization Using Deep Learning

Executive Summary

Variations of the Google Wavenet model are invetgated for forecasting the time series of parts demand for 25,000 different parts. 10yrs of monthly sales data is available for those parts, but most are sold once or twice and then never again. The annual stock order for such parts must be made 9months in advance and is approx GBP 550,000 (inc VAT). Even small improvements over the existing methods of forecasting, being ARIMA and weighted averages, are welcome.

This investigation reveals the wavenet can reduce costs by 28%, this is a surprisingly good result. This result is calculated on a test set isolated from training or validation. Test conditions are the same as would be expected in production.

Notably, this notebook achieves in 1,200 lines what hand crafting of features and use of traditional time series analysis could not achieve in 6,000 lines.

The Problem

Morris Corfield stocks 25,000 lines of parts, approximately 2,000 are active at any one time. Stocking needs to be optimised since obsolescense is a frequent problem in the industry.

Maintaining efficient stocking levels requires reliable demand forecasts. Currently an ARIMA model is used to forecast demand for each part based on previous sales invoices and credits. However, only a small minority of parts are sold sufficiently frequently that we can be confident about the ARIMA modelling. Most are sold infrequently. ARIMA modelling for the infrequently sold parts is likely to be unreliable, or even impossible. Therefore, a simple weighted average of previous year's sales is used.

The company has the opportunity to 'pre-season' order the majority of its stock for a given year, such orders can be made with very substantial discounts. Parts are priced such that it is much better to 'pre-season' order stock, than to wait until the middle of harvest for ordering parts.

So the objective is to use the multiple years of monthly history in order to make a forecast for the coming 12mths.

Previous approaches to this problem

There have been other models employed by Morris Corfield for this problem:

1. A weighted average has been used for a number of years: (3x Last Yr's sales + 2x Prior Yr's Sales + 1x Prior Prior Year's Sales) / 6
2. Forecasting using ARIMA modelling, but this proved no more accurate than the above weighted average.
3. Machine Learning Models for regression: Linear Regression, Bagged CART and XGBoost. All used as regression tools taking the below parameters. No model proved more effective than the weighted average approach!

- RRP
- Month of Forecast

- Years In Use
- Trend (Up, Down, None)
- Cluster (Based on correlation with trends in other parts)

- Sales Qty, mths 01to12
- Sales Qty, mths 13to24
- Sales Qty, mths 25to36
- Event Qty, mths 01to12
- Event Qty, mths 13to24
- Event Qty, mths 25to36

For a full introduction to the data, see the "Parts_Forecasting_Exploration" notebook

Approach of this Notebook

This notebook takes the final approach, being deep learning. It is inspired by Google's Wavenet architecture, which has been applied to a Kaggle competition reminiscent of our problem.

Details of the Kaggle competition at: https://www.kaggle.com/c/web-traffic-time-series-forecasting

The Kaggle competition introduction says: "This competition focuses on the problem of forecasting the future values of multiple time series, as it has always been one of the most challenging problems in the field. More specifically, we aim the competition at testing state-of-the-art methods designed by the participants, on the problem of forecasting future web traffic for approximately 145,000 Wikipedia articles."

This is very similar to our 778 time series of part sales over the prior 117 months. The concern is that we have insufficient data to train a deep learning model, but it is worth trying.

Details of the solution inspiring our approach can be found at: https://github.com/JEddy92/TimeSeries_Seq2Seq/blob/master/notebooks/TS_Seq2Seq_Conv_Intro.ipynb There are substantial differences in our implementation, not least multi-dimensional input (sales qty, invoice qty, etc), and one dimension output (sales qty only). This is not seq2seq. Because of this, we do not use teacher forcing, as it uses the output from step x to feed step x+1, yet output [scalar] has different dimensions to input [x time, y data types] for each series So this is not strictly a seq2seq model

Note, the winning solution used an RNN (stacked GRU) https://www.kaggle.com/c/web-traffic-time-series-forecasting/discussion/43795

Model Input and Output

The input to our model will be a time series of count data, this is a sequence of parts sales for each month over the preceeding X months. X is set by how much history is available. Currently approx 9yrs max, but most parts have not been in circulation for that period.

We actually have multiple time series we can input for each part number. Each is a 'feature'. The first feature is clearly sales quantity, the second is invoice quantity (many parts may be sold on one invoice).

Other features are available, those immediately to hand are: + Sales Qty + Invoice Qty (eg, sales within the month may have been on just one invoice, or many) + Weighted Average Forecast + Unit price

Other features, which could be added at a later date, include: + number of unique customers + month number (seasonality, eg january = month 1) etc

The input will therefore be a tensor: (n_series, n_timesteps, n_features)

The output could be a single figure, or a sequence.

As a single figure it would be the predicted total sales for the 12 months following the end of the data series. As a sequence we could ask the model to forecast each of the 12 forthcoming months. Or, we could ask the model to map the monthly sales to the annual forecasts, taken at each month end.

This final approach, mapping a sequence monthly data to a sequence of 12mth forecasts, will be our primary approach. This is because the monthly data is sparse, ie mostly zeros. Training to mostly zeros leads to problems. Annualised data is much less sparse, better for training.

Model Architecture

We want the model to understand temporal relationships and trends, so a simple network of dense layers will not suffice. The model must be LSTM, RNN or 1D causal convolution.

The google wavenet is derived from 1D conv nets. It achieves 'memory' via 'dilated causal convolutions'. See the above link to github for a full description. This will be the preferred approach for our model because such networks have fewer weights to train than LSTM's for the same scope, we have limited amounts of data.

We'll also be using the new functionality whereby R and Python can work together in one RStudio notebook: https://rstudio.github.io/reticulate/articles/r_markdown.html

Get Data

The data for this model was prepared in the previous notebook, Parts_Forecasting_Exploration.rmd, where it was also fully explored and charted. Now we simply load that data.

```r
library(dplyr)
library(reticulate)

setwd("C:\\Users\\User\\OneDrive\\Oliver\\0_OM\\Training\\DeepLearning_Udacity\\LSTM")

# Get x data
x_r    <- readRDS("pca_parts_cube_x_inclongtail.Rda")

#get dims
dims_x <- dim(x_r)

#replace NA with zero, can't feed NA to a neural net, it will have to learn what 0 means instead
x_r[is.na(x_r)] <- 0

# Occasionally the price is zero. This is duff data, must replace with 1.
unitprices <- x_r[,,3]
unitprices[unitprices == 0] <- 1
x_r[,,3]   <- unitprices

# also remove the final 12 columns of the data because we do not have matching forecasts for those
# months (incomplete year)
x_r_all<- x_r # create a copy before trimming data
x_r    <- x_r[,-seq(from= dims_x[2], to= (dims_x[2]-11), by=-1),]

# Get y data
y_r    <- readRDS("pca_parts_cube_y_inclongtail.Rda")
dims_y <- dim(y_r)
# also remove the final 1 column of the data because incomplete month
# (the final month is the current month)
y_r    <- array(y_r[,-dims_y[2],], c(dims_y[1], dims_y[2]-1, 1))

#extract sales qty dimension for easy access as a pandas dataframe
#doing this in R means we retain all the column and row names without effort
```

```
x_r_salesqty  <- as.data.frame(x_r[,,1]) %>%
                   tibble::rownames_to_column() %>%
                     rename(PART_NUMBER = rowname)

y_r_salesqty  <- as.data.frame(y_r[,,1]) %>%
                   tibble::rownames_to_column() %>%
                     rename(PART_NUMBER = rowname)

#Let' see the data structure:
print("X train data is in this format: (n_examples, n_months, n_features)")
```

```
## [1] "X train data is in this format: (n_examples, n_months, n_features)"
```

```
str(x_r)
```

```
##  num [1:24953, 1:108, 1:4] 0 0 0 0 0 0 0 0 0 1 ...
##  - attr(*, "dimnames")=List of 3
##   ..$ : chr [1:24953] "CL-0000000052" "CL-0000000101" "CL-0000000121" "CL-0000000170" ...
##   ..$ : chr [1:108] "2008-08-31" "2008-09-30" "2008-10-31" "2008-11-30" ...
##   ..$ : chr [1:4] "SalesQty" "EventsQty" "R_R_PRICE" "SimpleForecast"
```

```
print(paste0(
       "Total Data range is...",
       cat("\n"),
       "From :,",attributes(x_r_all)$dimnames[[2]][1],
       ". To : ",attributes(x_r_all)$dimnames[[2]][length(attributes(x_r_all)$dimnames[[2]])]))
```

```
##
## [1] "Total Data range is...From :,2008-08-31. To : 2018-07-31"
```

Let's check we can access that data from a python session…

```
import pandas as pd
import numpy  as np
#dplyr for pandas in python
from   dfply import *
#convert to pandas
df_x = r.x_r_salesqty
df_y = r.y_r_salesqty
# transfer the x (encode) data into a 3D array, (n_series, n_timesteps, n_features)
#   X = PART_NUMBERs   [~2500]
#   Y = Invoice months [~120]
#   Z = Data Types         [7] SalesQty, InvoiceQty, ARIMAforecast, etc
series_array_x = r.x_r
# transfer the y (prediction) data into a 3D array, (n_series, n_timesteps, n_features)
#   X = PART_NUMBERs   [~2500]
#   Y = Invoice months [~120]
#   Z = Data Types         [1] ONLY ONE! SalesQty
series_array_y = r.y_r
print('X data (encode)')
```

```
## X data (encode)
```

```
print(df_x.head())
```

```
##      PART_NUMBER  2008-08-31  2008-09-30  2008-10-31  2008-11-30  2008-12-31  \
## 0  CL-0000000052         0.0         0.0         0.0         0.0         0.0
## 1  CL-0000000101         0.0         0.0         0.0         0.0         0.0
## 2  CL-0000000121         0.0         0.0         2.0         0.0         0.0
## 3  CL-0000000170         0.0         0.0         0.0         0.0         0.0
## 4  CL-0000000211         0.0         0.0         0.0         0.0         0.0
##
##       ...      2017-02-28  2017-03-31  2017-04-30  2017-05-31  2017-06-30  \
## 0     ...             0.0         0.0         0.0         0.0         0.0
## 1     ...             0.0         0.0         0.0         0.0         1.0
## 2     ...             0.0         0.0         0.0         0.0         6.0
## 3     ...             0.0         2.0         0.0         0.0         0.0
## 4     ...             0.0         0.0         0.0         0.0         0.0
##
##    2017-07-31
## 0         0.0
## 1         0.0
## 2         0.0
## 3         0.0
## 4         0.0
##
## [5 rows x 109 columns]
```

```
print('\n')
```

```
print('Y data (prediction)')
```

```
## Y data (prediction)
```

```
print(df_y.head())
#python is zero indexed, first column (col 0) is PART_NUMBER, second col (col 1) is first month
```

```
##   PART_NUMBER   V1   V2   V3   V4   V5  ...   V103  V104  V105  V106  V107  \
## 0           1  0.0  0.0  0.0  0.0  0.0  ...    0.0   0.0   0.0   0.0   0.0
## 1           2  0.0  0.0  0.0  0.0  0.0  ...    1.0   1.0   1.0   1.0   1.0
## 2           3  2.0  4.0  4.0  2.0  2.0  ...    8.0   8.0   8.0   8.0   8.0
## 3           4  0.0  0.0  0.0  0.0  0.0  ...    2.0   2.0   0.0   0.0   0.0
## 4           5  0.0  0.0  0.0  0.0  0.0  ...    0.0   0.0   0.0   0.0   0.0
##
##    V108
## 0   0.0
## 1   0.0
## 2   9.0
## 3   0.0
## 4   0.0
##
## [5 rows x 109 columns]
```

```
months_from = df_x.columns[ 1]
months_to   = df_x.columns[-1]
print('\n')
```

```
print('x data ranges from %s to %s' % (months_from, months_to))
```

```
## x data ranges from 2008-08-31 to 2017-07-31
```

```
print('y data ranges from %s to %s' % (df_y.columns[1], df_y.columns[-1]))
```

```
## y data ranges from V1 to V108
```

Walk Forward Data Split

We will need a training+validation set and a test set. We will also investigate the benefits of using something similar to a 'walk-forward' validation vs traditional side by side validation.

Walk forward is where the validation set spans the same duration as the training set, but shifted forward in time. I say 'similar' because we only have 108 columns (months) of data, so we don't strictly need to 'walk' thru the data a span at a time. Instead, the model will dilate its input across all available data, nevertheless, the validation data will be one year ahead of the training data.

For example, assume our training set is thus: X (Encoding) : from Jan 2009 to Dec 2011 (range) Y (Prediction): sum of qty in 12mths from Jan 2012 to Dec 2012 (scalar) . . . where 'encoding' is the sequence fed into the model and 'prediction' is the sequence coming out.

Then our validation set should span: X (Encoding) : from Jan 2010 to Dec 2012 (range) Y (Prediction): sum of qty in 12mths from Jan 2013 to Dec 2013 (scalar)

This way, we force the model to learn about forecasting the future, a time period it has not been trained on.

The traditional alternative is side-by-side validation, separating X% of the data for use in validation. The upside is that we can use the entire span of available data for training, whereas walk-forward must leave the final year for validation only, not training. The downside is that side-by-side does not validate against future dates, instead, it validates in the same time frame as the training set

We'll try both and see which works best on the test set.

```
from datetime import *
from dateutil.relativedelta import *
month_first_in_import = pd.to_datetime(months_from)
month_last_in_import  = pd.to_datetime(months_to)
# Get start of periods. Validation is same data shunted along by 12months
# (starting at the 13th month)
encoding_firstMth_train = month_first_in_import
encoding_firstMth_valid = month_first_in_import + relativedelta(months=12)
# Get start of periods. Validation is same data shunted along by 12months
# (ending at the last month)
encoding_finalMth_train = month_last_in_import - relativedelta(months=12)
encoding_finalMth_valid = month_last_in_import
encoding_span_total     = relativedelta(encoding_finalMth_train, encoding_firstMth_train)
print('TRAIN & VALIDATION DATA')
```

```
## TRAIN & VALIDATION DATA
```

```
print('Total data span (inclusive):         ', month_first_in_import,  '-', month_last_in_import)
```

```
## Total data span (inclusive):          2008-08-31 00:00:00 - 2017-07-31 00:00:00
```

```
print('TRAINING')
```

```
## TRAINING
```

```
print('Train encoding span (inclusive):     ', encoding_firstMth_train,'-', encoding_finalMth_train)
```

```
## Train encoding span (inclusive):      2008-08-31 00:00:00 - 2016-07-31 00:00:00
```

```
print('VALIDATION')
```

```
## VALIDATION
```

```
print('Validation encoding span (inclusive):', encoding_firstMth_valid,'-', encoding_finalMth_valid)
```

```
## Validation encoding span (inclusive): 2009-08-31 00:00:00 - 2017-07-31 00:00:00
```

Predictions are presented in the 'y' array precalculated for the forthcoming 12months They are presented as scalar values, one value as the sum for the 12mth forecast period They are assigned to the 'ForecastFromMonth', so have the same column index as the final month in a training/valid/test span

Prepare the data

Keras expects the input as well as the target data to be in a specific shape. The input has to be a 3-d array of size num_samples, num_timesteps, num_features.

The first dimension, num_samples is the number of observations in the set. The second dimension, num_timesteps, is the length of the hidden state we were talking about above. The third dimension, num_features, is the number of predictors we're using. - SalesQty - EventQty - Unit Price - Simple Forecast

Note, we are now building a stacked model, our deep learning model will take the predictions of other models in order to calculate its own predictions.

The data be fed to the model in portions of batch_size.

Pull the time series into an array, save a date_to_index mapping as a utility for referencing into the array Create function to extract specified time interval from all the series Create functions to transform all the series.

```python
# data will need to be in array format, but arrays don't hold the month column names
# so, it's useful to have a tool for mapping the array index to month
date_to_index = pd.Series(index = pd.Index([pd.to_datetime(c) for c in df_x.columns[1:]]),
                          data  = [i for i in range(len(df_x.columns[1:]))])
print(series_array_x.shape)
```

```
## (24953, 108, 4)
```

Create train/val and test sets

The test set will be comprised of part numbers on which the model has not been trained or validated. Randomly split data by part number to select which parts are used in train/test and which are used in test.

Note, this code is sleectnig the set by the row number, ie by part_number. The selection by column, ie by month, to ensure walk forward validation, is in a subsequent code chunk.

```python
# x and y should have the same number of parts and the same numbe rof months
assert series_array_x.shape[0] == series_array_y.shape[0], "x has different number of parts to y"
assert series_array_x.shape[1] == series_array_y.shape[1], "x has different number of months to y"
# RANDOMLY SELECT PARTS FOR TRAINING, VALIDATION AND TEST
# first separate 'train and validation' from test
def split_set(series_array_x, series_array_y, proportion_for_A = 0.8):
  no_of_parts_AandB= series_array_x.shape[0]
  no_of_parts_A    = int(round(proportion_for_A * no_of_parts_AandB,0))
  indices_for_A    = np.random.choice(no_of_parts_AandB,
                                      no_of_parts_A,
                                      replace = False).tolist()
  indices_for_B    = [x for x in range(no_of_parts_AandB) if x not in indices_for_A]

  series_array_x_A = series_array_x[indices_for_A,:,:]
  series_array_y_A = series_array_y[indices_for_A,:,:]
  series_array_x_B = series_array_x[indices_for_B,:,:]
  series_array_y_B = series_array_y[indices_for_B,:,:]
  return (series_array_x_A, series_array_y_A, series_array_x_B, series_array_y_B, indices_for_A)
# create test set vs combination of train and validate
series_array_x_train_n_valid, series_array_y_train_n_valid, series_array_x_test, series_array_y_test, in
# repeat the process to separate validation from train
# Note, 75% of the original 80% = 60% of the original data
series_array_x_train, series_array_y_train, series_array_x_valid, series_array_y_valid, indices_train =
print("Total number of parts in data  :", series_array_x.shape[0])
```

```
## Total number of parts in data  : 24953
```

```
print("number of parts in training    :", series_array_x_train.shape[0])
```

```
## number of parts in training    : 14972
```

```
print("number of parts in validation  :", series_array_x_valid.shape[0])
```

```
## number of parts in validation  : 4990
```

```
print("number of parts in testing     :", series_array_x_test.shape[0])
```

```
## number of parts in testing     : 4991
```

We need a function whereby the model can extract the select months of data from the array

```
## X (encode)
def get_time_block_series(series_array, date_to_index, start_date, end_date):
    inds = date_to_index[start_date:end_date]
    return series_array[:,inds,:] # (n_series, n_timesteps, n_features)
```

Here we smooth out the scale by taking log1p and de-meaning each series using the encoder series mean, then reshape to the (n__series, n__timesteps, n__features) tensor format that keras will expect.

Note that if we want to generate true predictions instead of log scale ones, we can easily apply this function: round(exp(prediction__logged)-1,0) = prediction__integer

```
# Normalise by taking log(x+1) and centreing mean at 0
# then ensure tensor shape (ie numpy array) is as expected by keras
# (n_series, n_timesteps, n_features)
def transform(series_array):

    # scale with log(x+1)
    series_array = np.log1p(np.nan_to_num(series_array)) # filling NaN with 0

    # centre mean at 0
    ## get mean for each row and each dim deep (salesqty, invoices, etc).
    ## returns 3D, eg 902 parts, 1 mean for all months, 7 predictors
    series_mean  = np.nanmean(series_array, axis=1, keepdims=True)

    ## subtract mean from each item. Broadcasts across all items in row (row=time series)
    series_array = series_array - series_mean

    # ensure shape is (n_series, n_timesteps, n_features)
    series_array = series_array.reshape((series_array.shape[0], # X = PART_NUMBERs  [~2500]
                                         series_array.shape[1], # Y = Invoice months [~120]
                                         series_array.shape[2]  # Z = Data Types    [max 7]
                                         ))
    return series_array, series_mean
# similar code, but this time for un-normalising and un'log(x+1)'ing of the data, if required
def reverse_transform(series_array, encode_series_mean):
    #series_mean is mean of logged values, so add back before exp()
    series_array = series_array + encode_series_mean
    series_array = np.exp(np.nan_to_num(series_mean)) - 1 # opposite of log(x+1) is exp(x)-1
    series_array = series_array.reshape((series_array.shape[0], # X = PART_NUMBERs  [~2500]
                                         series_array.shape[1], # Y = Invoice months [~120]
                                         series_array.shape[2]  # Z = Data Types    [max 7]
                                         ))
    return series_array
```

Prepare Training Data

```
# ENCODING
# sample of series from encoding_firstMth_train to encoding_finalMth_train
x_train_n_valid_sidebyside = get_time_block_series(series_array  = series_array_x_train_n_valid,
                                                   date_to_index = date_to_index,
                                                   start_date    = encoding_firstMth_train,
                                                   #Note, end date is later than walk forward train
                                                   end_date      = encoding_finalMth_valid)
                                                   #(n_series, n_timesteps, n_features)
x_train_walkfor = get_time_block_series(series_array  = series_array_x_train,
                                        date_to_index = date_to_index,
                                        start_date    = encoding_firstMth_train,
                                        end_date      = encoding_finalMth_train)
                                        #(n_series, n_timesteps, n_features)
x_valid_walkfor = get_time_block_series(series_array  = series_array_x_valid,
                                        date_to_index = date_to_index,
                                        start_date    = encoding_firstMth_valid,
                                        end_date      = encoding_finalMth_valid)
                                        #(n_series, n_timesteps, n_features)
x_train_n_valid_sidebyside_trans,  x_train_n_valid_sidebyside_mean = transform(x_train_n_valid_sidebysi
x_train_walkfor_trans, x_train_walkfor_mean = transform(x_train_walkfor)
x_valid_walkfor_trans, x_valid_walkfor_mean = transform(x_valid_walkfor)
print("For use with 'side by side' validation (25% will be validation):")
```

```
## For use with 'side by side' validation (25% will be validation):
```

```
print("Training & validation dims: ", x_train_n_valid_sidebyside_trans.shape)
```

```
## Training & validation dims:  (19962, 108, 4)
```

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

```
print("For use with 'Walk Forward' validation:")
```

```
## For use with 'Walk Forward' validation:
```

```
print("Training   dims: ", x_train_walkfor_trans.shape)
```

```
## Training   dims:  (14972, 96, 4)
```

```
print("Validation dims: ", x_valid_walkfor_trans.shape)
```

```
## Validation dims:  (4990, 96, 4)
```

```
# PREDICTION
# sample of series of targets, SAME PERIOD as training period because data is presented as
# forecast for 12mths from month end
y_train_n_valid_sidebyside = get_time_block_series(series_array  = series_array_y_train_n_valid,
                                                   date_to_index = date_to_index,
                                                   start_date    = encoding_firstMth_train,
                                                   #Note, end date is later than walk forward train
                                                   end_date      = encoding_finalMth_valid)

y_train_walkfor = get_time_block_series(series_array  = series_array_y_train,
                                        date_to_index = date_to_index,
                                        start_date    = encoding_firstMth_train,
                                        end_date      = encoding_finalMth_train)

y_valid_walkfor = get_time_block_series(series_array  = series_array_y_valid,
                                        date_to_index = date_to_index,
                                        start_date    = encoding_firstMth_valid,
                                        end_date      = encoding_finalMth_valid)

# We don't need to log1p() the the y's, nor to exp() them afterwards,
# coz the model will target the actual integers
# We don't need to use teacher forcing...so commented out
# append a lagged history of the target series to the input data,
# so that we can train with teacher forcing
# NO LONGER NEEDED, COMMENTED OUT
# lagged_target_history = decoder_target_data[:,:-1,:1]
# encoder_input_data    = np.concatenate([encoder_input_data, lagged_target_history], axis=1)
print("For use with 'side by side' validation (25% will be validaiton):")
```

```
## For use with 'side by side' validation (25% will be validaiton):
```

```
print("y train : ", y_train_n_valid_sidebyside.shape)
```

```
## y train :  (19962, 108, 1)
```

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

```
print("For use with 'Walk Forward' validation:")
```

```
## For use with 'Walk Forward' validation:
```

```
print("y train : ", y_train_walkfor.shape)
```

```
## y train :  (14972, 96, 1)
```

```
print("y valid : ", y_valid_walkfor.shape)
```

```
## y valid :  (4990, 96, 1)
```

Prepare the model

Here's what we'll use:

- 8 dilated causal convolutional layers
- 32 filters of width 2 per layer
- Exponentially increasing dilation rate (1, 2, 4, 8, 32). – Ends at 32, ie 32months of data before prediction is made
- 2 (time distributed) fully connected layers to map to final output
- input will be time series, output will be single figure of forecast for forthcoming 12mths

```
from keras.models import Model
from keras.layers import Input, Conv1D, Dense, Dropout, Lambda, concatenate
from keras.optimizers import Adam, TFOptimizer
def create_model_valmethod(use_slice = False):

  # convolutional layer parameters
  batch_size     = 1024  # uses most of available GPU RAM, 2048 maxes out
  n_filters      =   32
  filter_width   =    2
  dilation_rates = [1, 2, 4, 8, 16, 32, 64, 96] # can't go to 128, not enough months of data
  predictor_qty  =    4

  # 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)

  #NB no activation after Conv1D

  # dense layer, ReLU activation
  x = Dense(128, activation='relu')(x)

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

  # softmax, NB only 1D output.
  # So 7 dimensions in to this model, but only 1 dimension out. No activation,
  # a linear output unconstrained to a range
  x = Dense(1)(x)

  ## No teacher forcing
  # extract the last 12 time step(s) as the training target
  def slice(x, seq_length):
    return x[:,-seq_length:,:]
  if(use_slice):
    x = Lambda(slice, arguments={'seq_length':12})(x)
  output_sequence = x

  model = Model(inputs  = input_sequence,
                outputs = output_sequence)
  return(model)
```

```
model_val_sidebyside = create_model_valmethod()
model_val_walkfor    = create_model_valmethod()
model_val_sidebyside.summary()
```

```
## _________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =================================================================
## input_3 (InputLayer)         (None, None, 4)           0
## _________________________________________________________________
## conv1d_17 (Conv1D)           (None, None, 32)          288
## _________________________________________________________________
## conv1d_18 (Conv1D)           (None, None, 32)          2080
## _________________________________________________________________
## conv1d_19 (Conv1D)           (None, None, 32)          2080
## _________________________________________________________________
## conv1d_20 (Conv1D)           (None, None, 32)          2080
## _________________________________________________________________
## conv1d_21 (Conv1D)           (None, None, 32)          2080
## _________________________________________________________________
## conv1d_22 (Conv1D)           (None, None, 32)          2080
## _________________________________________________________________
## conv1d_23 (Conv1D)           (None, None, 32)          2080
## _________________________________________________________________
## conv1d_24 (Conv1D)           (None, None, 32)          2080
## _________________________________________________________________
## dense_5 (Dense)              (None, None, 128)         4224
## _________________________________________________________________
## dropout_3 (Dropout)          (None, None, 128)         0
## _________________________________________________________________
## dense_6 (Dense)              (None, None, 1)           129
## =================================================================
## Total params: 19,201
## Trainable params: 19,201
## Non-trainable params: 0
## _________________________________________________________________
```

Just 20k parameters to train, should be straight forward.

```
from keras import callbacks
batch_size = 1024
epochs     = 2000
# 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_sidebyside = callbacks.TensorBoard(log_dir        = './Tensorboard_sidebyside',
                                              histogram_freq = 0,
                                              write_graph    = True)
## Checkpoint callback
cpCallback_sidebyside = callbacks.ModelCheckpoint('./Checkpoints/sidebyside_{epoch:02d}.hdf5',
                                                  monitor='val_acc', verbose=1,
                                                  save_best_only=True, mode='max')

##Earlystopping, stop training when validation accuracy ceases to improve
esCallback          = callbacks.EarlyStopping(monitor   = 'val_loss', min_delta = 0.00005,
                                              patience = 200,        verbose   = 0,
                                              mode     = 'auto',     baseline  = None)
## Tensorboard callback
tbCallback_walkfor  = callbacks.TensorBoard(log_dir        = './Tensorboard_walkfor',
                                            histogram_freq = 0,
                                            write_graph    = True)
## Checkpoint callback
cpCallback_walkfor  = callbacks.ModelCheckpoint('./Checkpoints/walkforward_{epoch:02d}.hdf5',
                                               monitor        = 'val_acc', verbose = 1,
                                               save_best_only = True,      mode    = 'max')
```

Train the Side by Side Model

```
## Compile Model with Side by Side Validation
model_val_sidebyside.compile(optimizer = Adam(),
                             loss      = 'mean_absolute_error')

## Let's fit using side by side
history_val_sidebyside = model_val_sidebyside.fit(
                              x                = x_train_n_valid_sidebyside_trans,
                              y                = y_train_n_valid_sidebyside,
                              batch_size       = batch_size,
                              epochs           = epochs,
                              # For side by side we ask keras to apply a train/validation split
                              # we'd like 20% of all data, which is 25% of the 80% in train_n_valid
                              validation_split = 0.25,
                              # For side by side validation we may as well shuffle the data
                              shuffle          = True)
                              #callbacks        = [cpCallback_sidebyside,
                              #                    esCallback,
                              #                    tbCallback_sidebyside])
# save model
model_val_sidebyside.save('./SaveModels/model_val_sidebyside.h5')
# save training history
import pickle
with open('./SaveModels/model_val_sidebyside_history.pickle', 'wb') as file_pi:
    pickle.dump(history_val_sidebyside.history, file_pi)
```

Train the Walk Forward Model

Remember, this model has more robust validation for time series, since validation leads train data by 12mths. However, this means there is a shorter period of data available for training, 12mths less.

```
# Compile Model with Walk Forward Validation
model_val_walkfor.compile(optimizer = Adam(),
                          loss      = 'mean_absolute_error')

## Let's fit using walkforward
history_val_walkfor = model_val_walkfor.fit(
                              x               = x_train_walkfor_trans,
                              y               = y_train_walkfor,
                              batch_size      = batch_size,
                              epochs          = epochs,
                              validation_data = (x_valid_walkfor_trans, y_valid_walkfor))
                              #NO shuffle for walk forward validation, defeats the objective
                              #callbacks       = [cpCallback_walkfor,
                              #                     esCallback,
                              #                     tbCallback_walkfor])

# save model
model_val_walkfor.save('./SaveModels/model_val_walkfor.h5')
# save training history
import pickle
with open('./SaveModels/model_val_walkfor_history.pickle', 'wb') as file_pi:
    pickle.dump(history_val_walkfor.history, file_pi)
```

Load Models From File

The above training can take a long time (overnight), so it's a safer to assume we must load the models and their history from file

```
from keras.models import load_model
from os import path
import pickle
def get_model_n_history_file(base_filename):
  the_model_locn   = path.join('.\\SaveModels', base_filename + ".h5")
  the_model        = load_model(the_model_locn)
  the_history_locn = path.join('.\\SaveModels', base_filename + "_history.pickle")
  the_history_file = open(the_history_locn,'rb')
  the_history      = pickle.load(the_history_file)

  return(the_model, the_history)
```

```
model_val_walkfor, history_val_walkfor = get_model_n_history_file(base_filename = "model_val_walkfor")

model_val_sidebyside, history_val_sidebyside = get_model_n_history_file(base_filename = "model_val_side
```

Plot Convergence

```
library(ggplot2)
history_py_trn_sidebyside <- unlist(py$history_val_sidebyside['loss'])
history_py_val_sidebyside <- unlist(py$history_val_sidebyside['val_loss'])
history_py_epc_sidebyside <- seq_along(history_py_trn_sidebyside)
```

```
history_py_trn_walkfo <- unlist(py$history_val_walkfor['loss'])
history_py_val_walkfo <- unlist(py$history_val_walkfor['val_loss'])
history_py_epc_walkfo <- seq_along(history_py_trn_walkfo)

history_py_sidebyside <- cbind.data.frame(epoch    = history_py_epc_sidebyside,
                                          loss     = history_py_trn_sidebyside,
                                          val_loss = history_py_val_sidebyside)

history_py_walkfo <- cbind.data.frame(epoch    = history_py_epc_walkfo,
                                      loss     = unlist(history_py_trn_walkfo),
                                      val_loss = unlist(history_py_val_walkfo))

p <- ggplot(data = history_py_sidebyside, 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") + ylim(0.6,1.4)+
     ggtitle("Side by Side Validation: Training Loss")

q <- ggplot(data = history_py_walkfo, 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") + ylim(0.6,1.4)+
     ggtitle("Walkforward Validation: Training Loss")

library(gridExtra)
grid.arrange(p, q, ncol=2)
```

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

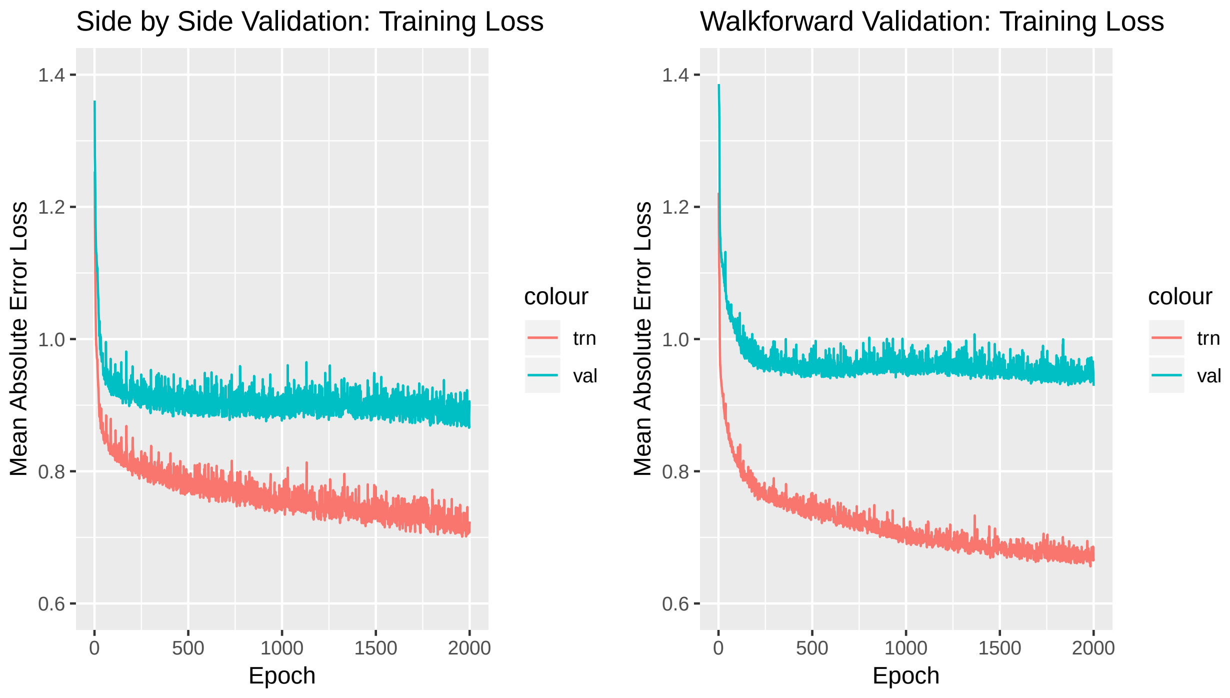


Chart shows training goes well up to approx 200 epochs. After that results on the training set continue to improve, whereas results on validation do not, implying overtraining.

Applying the model on the test set

Run on Test Set

Since we are short of data across time, the test set is defined by the PART_NUMBERS, not the months. Not completely happy with this, but it'll suffice for now.

```
# Prepare the data
# test period will be the entire data available
encoding_firstMth_test   = month_first_in_import
encoding_finalMth_test   = month_last_in_import
# TEST DATA ENCODING
# sample of series same period as training, BUT different parts
x_test = get_time_block_series(series_array  = series_array_x_test,
                               date_to_index = date_to_index,
                               start_date    = encoding_firstMth_test,
                               end_date      = encoding_finalMth_test)
                               #(n_series, n_timesteps, n_features)
x_test_transformed, x_test_mean = transform(x_test)
# TEST DATA TARGETS
# sample of series same period as training, BUT different parts
y_test = get_time_block_series(series_array  = series_array_y_test,
                               date_to_index = date_to_index,
                               start_date    = encoding_firstMth_test,
                               end_date      = encoding_finalMth_test)

# We don't need to log1p() the the y's, nor to exp() them afterwards,
# the model will target the actual integers
predictions_test_sidebyside  = model_val_sidebyside.predict(x_test_transformed)
predictions_test_walkfor     = model_val_walkfor.predict(x_test_transformed)
```

Test Results

We copy our results from python and then analyse in R

```
# get model predictons for test
preds_test_sidebyside<- py$predictions_test_sidebyside
preds_test_walkfor   <- py$predictions_test_walkfor

# get simple weighted averages and prices, to calculate point of comparison, for test
wavg_preds_test_r <- py$x_test[,,4]
unitprice_test_r  <- py$x_test[,,3]

# get actuals for test
actuals_test_r    <- py$y_test
```

```
get_wavg_vs_keras  <- function(keras_preds_test_r, wavg_preds_test_r,
                               unitprice_test_r,   actuals_test_r,
                               byval = FALSE){
```

```r
  # NB: byval=TRUE indicates the prediction is the value to order, not the qty to order

  # force into 2D
  twoD_me <- function(the_array){array(the_array, c(nrow(the_array),ncol(the_array)))}
  keras_preds_test_r <- twoD_me(keras_preds_test_r)
  wavg_preds_test_r  <- twoD_me(wavg_preds_test_r)
  unitprice_test_r   <- twoD_me(unitprice_test_r)
  actuals_test_r     <- twoD_me(actuals_test_r)

  # round to integer, min 0
  if(byval){
    keras_preds_test_r <- round(keras_preds_test_r / unitprice_test_r,0)
  }else{
    keras_preds_test_r <- round(keras_preds_test_r,0)
  }
  keras_preds_test_r   <- ifelse(keras_preds_test_r<0,0,keras_preds_test_r)

  #divider to get average per forecast
  qty_of_forecasts <- ncol(actuals_test_r)
  num_of_parts     <- nrow(actuals_test_r)

  #over order and under order
  keras_money_val_over <- sum(ifelse(keras_preds_test_r > actuals_test_r,
                                     keras_preds_test_r - actuals_test_r, 0)
                              * unitprice_test_r) / qty_of_forecasts

  keras_money_val_under<- sum(ifelse(keras_preds_test_r < actuals_test_r,
                                     keras_preds_test_r - actuals_test_r, 0)
                              * unitprice_test_r) / qty_of_forecasts

  wavg_money_val_over  <- sum(ifelse(wavg_preds_test_r  > actuals_test_r,
                                     wavg_preds_test_r  - actuals_test_r, 0)
                              * unitprice_test_r) / qty_of_forecasts

  wavg_money_val_under <- sum(ifelse(wavg_preds_test_r  < actuals_test_r,
                                     wavg_preds_test_r  - actuals_test_r, 0)
                              * unitprice_test_r) / qty_of_forecasts

  # Summary or total error, over/under order
  keras_money_val_err  <- keras_money_val_over + abs(keras_money_val_under)
  wavg_money_val_err   <- wavg_money_val_over  + abs(wavg_money_val_under)

  print(paste0("Number of parts in analysis     : ", num_of_parts))

  print(paste0("Error with deep model   (GBP/yr): ", round(keras_money_val_err,0),
               ". Over Order  = ", round(keras_money_val_over, 0),
               ". Under Order = ", round(keras_money_val_under,0) ))

  print(paste0("Error with weighted avg (GBP/yr): ", round(wavg_money_val_err,0),
               ". Over Order  = ", round(wavg_money_val_over, 0),
               ". Under Order = ", round(wavg_money_val_under,0) ))
}
```

```
print("Test results with shuffled side by side validation (20%)")
```

```
## [1] "Test results with shuffled side by side validation (20%)"
```

```
cat("\n")
```

```
get_wavg_vs_keras(preds_test_sidebyside, wavg_preds_test_r,
                  unitprice_test_r,  actuals_test_r)
```

```
## [1] "Number of parts in analysis     : 4991"
## [1] "Error with deep model   (GBP/yr): 230989. Over Order  = 37511. Under Order = -193477"
## [1] "Error with weighted avg (GBP/yr): 293015. Over Order  = 8496. Under Order = -284519"
```

```
print("Test results with walk-forward validation")
```

```
## [1] "Test results with walk-forward validation"
```

```
cat("\n")
```

```
get_wavg_vs_keras(preds_test_walkfor, wavg_preds_test_r,
                  unitprice_test_r,   actuals_test_r)
```

```
## [1] "Number of parts in analysis     : 4991"
## [1] "Error with deep model   (GBP/yr): 516670. Over Order  = 355634. Under Order = -161035"
## [1] "Error with weighted avg (GBP/yr): 293015. Over Order  = 8496. Under Order = -284519"
```

The walkforward model is not usable. Curiously the side by side deep model appears to be substantially better than the weighted average. This may be due to the split of data being unfortunate in some way.

Test on Frequently Used Parts

Let's focus on frequently used parts, meaning those which have > 12 sales events, and see how the models compare.

```
salesevents_test_r <- py$x_test[,,2]
salesevents_sum    <- apply(salesevents_test_r, 1, function(x) sum(x))
selected_indices   <- which(salesevents_sum>12)

print("Test results for parts with substantial sales")
```

```
## [1] "Test results for parts with substantial sales"
```

```
cat("\n")
```

```
print("Modelled with Side by side validation")
```

```
## [1] "Modelled with Side by side validation"
```

```
get_wavg_vs_keras(preds_test_sidebyside[selected_indices,,1],
                  wavg_preds_test_r[selected_indices,],
                  unitprice_test_r[selected_indices,],
                  actuals_test_r[selected_indices,,1])
```

```
## [1] "Number of parts in analysis     : 380"
## [1] "Error with deep model   (GBP/yr): 57417. Over Order  = 24409. Under Order = -33008"
## [1] "Error with weighted avg (GBP/yr): 91938. Over Order  = 6561. Under Order = -85377"
```

```
cat("\n")
```

```r
print("Modelled with Walk Forward validation")
```

```
## [1] "Modelled with Walk Forward validation"
```

```r
get_wavg_vs_keras(preds_test_walkfor[selected_indices,,1],
                  wavg_preds_test_r[selected_indices,],
                  unitprice_test_r[selected_indices,],
                  actuals_test_r[selected_indices,,1])
```

```
## [1] "Number of parts in analysis     : 380"
## [1] "Error with deep model   (GBP/yr): 70472. Over Order  = 29239. Under Order = -41233"
## [1] "Error with weighted avg (GBP/yr): 91938. Over Order  = 6561. Under Order = -85377"
```

The deep models outperform the weighted average, a feat which statistical models failed to do! Approx 35% reduction in error vs the weighted average for the side-by-side validation model. The walk forward validation remains less accurate than the side by side. Both deep models achieve this by over ordering more and under-ordering less, ie they broadly know where to be more generous.

Common Sense Check

Overall the deep models under-order less and over-order more than the weighted average model. Are the deep models simply ordering more in total? Has the model learnt to simply up the order suggested by the weighted average by say 10%? If this is true then we should ditch the complex deep learning model and return to a simple weighted average. We can easily test how such an order would fare…

```r
print("Test results for parts with substantial sales")
```

```
## [1] "Test results for parts with substantial sales"
```

```r
cat("\n")
```

```r
print("Modelled with Side by side validation, Weighted Average x 1.1")
```

```
## [1] "Modelled with Side by side validation, Weighted Average x 1.1"
```

```r
get_wavg_vs_keras(preds_test_sidebyside[selected_indices,,1],
                  (wavg_preds_test_r*1.1)[selected_indices,],
                  unitprice_test_r[selected_indices,],
                  actuals_test_r[selected_indices,,1])
```

```
## [1] "Number of parts in analysis     : 380"
## [1] "Error with deep model   (GBP/yr): 57417. Over Order  = 24409. Under Order = -33008"
## [1] "Error with weighted avg (GBP/yr): 91547. Over Order  = 8731. Under Order = -82815"
```

Simply upping the weighted average by 10% does not bridge the difference between the models, it actually worsens it. This further indicates the deep models have learnt something useful.

Alternative Approach to Data

So far we have attempted sequence to sequence modelling where the length of the sequence in matches the length of the sequence out. For example, if we feed the model with 36 months of sales data, we expect 36 forecasts out. Each of those 36 forecasts is an annual forecast as from the equivalent month in the 36 step sequence.

As an alternative approach, we could feed the model with 36months of data and ask the model to forecast 12 monthly steps of sales. In a post modelling process we would then sum the forecasts to get an annual forecast.

This alternative simply needs some data wrangling of the existing x_train data. Note, the y_train data is comprised of annual forecasts, not monthly. We will need to derive new targets from the x_train data.

Process data: log(1+x) of sales and events quantities. Normalise all columns (centre =0, sd=1)

```
# remember n_channels= 1"SalesQty", 2"EventsQty", 3"R_R_PRICE", 4"SimpleForecast"

x_r_all_processed              <- x_r_all
x_r_all_processed[,,c(1,2,4)] <- log(x_r_all_processed[,,c(1,2,4)] + 1)
x_r_all_processed[,,1]        <- scale(x_r_all_processed[,,1])
x_r_all_processed[,,2]        <- scale(x_r_all_processed[,,2])
x_r_all_processed[,,3]        <- scale(x_r_all_processed[,,3])
x_r_all_processed[,,4]        <- scale(x_r_all_processed[,,4])
```

```
# The final 12 months are the final forecast, so cannot be used in training or validation
# Also, as with above approach the validation set will be 12months ahead of the training set
# Therefore, the final 24months are not available to training:
## 12 for final validation target and 12 for final training target

x_train_alt <- x_r_all[py$indices_train_n_valid,
                       seq(from =  1, to = (dim(x_r_all_processed)[2]-24), by = 1),]

x_test_alt  <- x_r_all[-py$indices_train_n_valid,
                       seq(from = 13, to = (dim(x_r_all_processed)[2]-12), by = 1),]

print(paste0("X Train+Valid dims: ", paste(dim(x_train_alt), collapse=" ")))
```

```
## [1] "X Train+Valid dims: 19961 96 4"
```

```
print(paste0("X Test        dims: ", paste(dim(x_test_alt),  collapse=" ")))
```

```
## [1] "X Test        dims: 4992 96 4"
```

…and targets as sequences of 12months of sales…

```
# We'll need a function to convert a 2D array into 3D, depth=1.
# Note, we don't use processed data, raw sales figures required.
threeD_me <- function(the_array){array(the_array, c(nrow(the_array),ncol(the_array), 1))}

y_train_alt <- threeD_me(x_r_all[py$indices_train_n_valid,
                                 seq(from = (dim(x_r_all)[2]-23), to=(dim(x_r_all)[2]-12), by=1),
                                 1])

y_test_alt  <- threeD_me(x_r_all[-py$indices_train_n_valid,
                                 seq(from = (dim(x_r_all)[2]-11), to=dim(x_r_all)[2], by=1),
                                 1])

#for comparison we need the baseline forecast performance, which uses weighted averages
#wavg table is of 12mth forecasts for the current column to 12mths hence
#In order to compare with our monthly forecasts we take the FIRST col only
y_train_alt_wavg <- x_r_all[py$indices_train_n_valid,
                            dim(x_r_all)[2]-24,
                            4]
y_train_alt_rrp  <- x_r_all[py$indices_train_n_valid,
                            dim(x_r_all)[2]-24, #for the rrp, we want same dims as wavg
                            3]
y_test_alt_wavg  <- x_r_all[-py$indices_train_n_valid,
                            dim(x_r_all)[2]-12,
                            4]
y_test_alt_rrp   <- x_r_all[-py$indices_train_n_valid,
                            dim(x_r_all)[2]-12,
                            3]

print(paste0("Y Train+Valid dims: ", paste(dim(y_train_alt), collapse = " ")))
```

```
## [1] "Y Train+Valid dims: 19961 12 1"
```

```
print(paste0("Y Test        dims: ", paste(dim(y_test_alt),  collapse = " ")))
```

```
## [1] "Y Test        dims: 4992 12 1"
```

Train a Model with Alternative Data

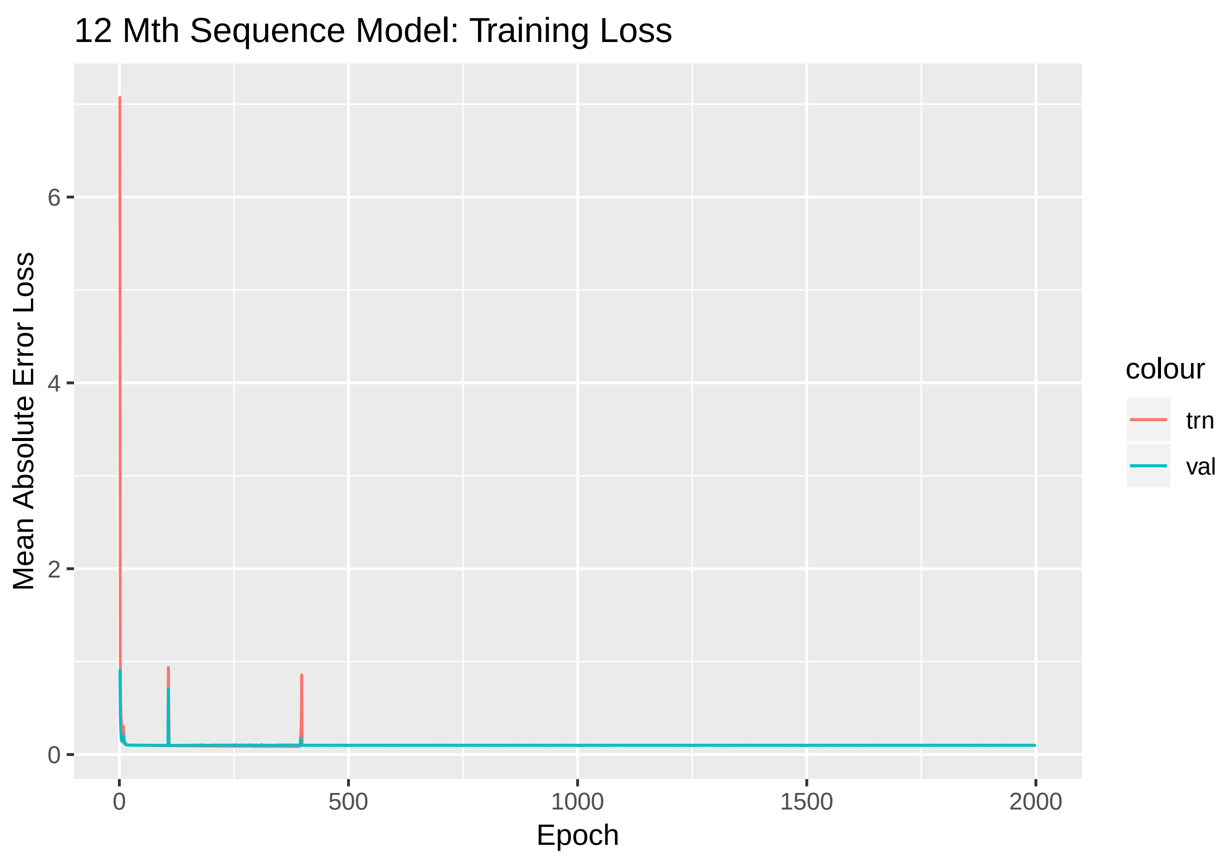
```
# create
model_parts_12mth = create_model_valmethod(use_slice = True)
# compile
model_parts_12mth.compile(optimizer = Adam(),
                          loss      = 'mean_absolute_error')
# fit
history_parts_12mth = model_parts_12mth.fit(x                = r.x_train_alt,
                                            y                = r.y_train_alt,
                                            batch_size       = 1024,
                                            epochs           = epochs,
                                            validation_split = 0.25,
                                            shuffle          = True)
# save model
model_parts_12mth.save('./SaveModels/model_parts_12mth.h5')
# save history
import pickle
with open('./SaveModels/model_parts_12mth_history.pickle', 'wb') as file_pi:
    pickle.dump(history_parts_12mth.history, file_pi)
```

```
## load data from file
model_parts_12mth, history_parts_12mth = get_model_n_history_file('model_parts_12mth')
```

```
history_py_cash <- cbind.data.frame(epoch    = seq_along(unlist(py$history_parts_12mth['loss'])),
                                    loss     = unlist(py$history_parts_12mth['loss']),
                                    val_loss = unlist(py$history_parts_12mth['val_loss']))

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("12 Mth Sequence Model: Training Loss")

p
```



That trained peculiarly quickly, let's inspect results.

```
# Get predictions
preds_train_parts_12mth = np.round(model_parts_12mth.predict(r.x_train_alt),0)
preds_test_parts_12mth  = np.round(model_parts_12mth.predict(r.x_test_alt),0)
```

```
# mean model error on train set vs mean error of weighted average on train set.

mean_wavg_errval_train  <- sum(abs(y_train_alt_wavg - rowSums(y_train_alt)) * y_train_alt_rrp)
mean_model_errval_train <- sum(abs(rowSums(py$preds_train_parts_12mth) - rowSums(y_train_alt))
                               * y_train_alt_rrp)

# mean model error on test set vs mean error of weighted average on test set.

mean_wavg_errval_test   <- sum(abs(y_test_alt_wavg - rowSums(y_test_alt)) * y_test_alt_rrp)
mean_model_errval_test  <- sum(abs(rowSums(py$preds_test_parts_12mth) - rowSums(y_test_alt))
                               * y_test_alt_rrp)

rbind(cbind.data.frame(Type               = "Train",
                       Mean_Model_Err_Val = mean_model_errval_train,
                       Mean_Wavg_Err_Val  = mean_wavg_errval_train),
      cbind.data.frame(Type               = "Test",
                       Mean_Model_Err_Val = mean_model_errval_test,
                       Mean_Wavg_Err_Val  = mean_wavg_errval_test)
)
```

The new approach to the data results in a model worse than the weighted average approach! In fact, on inspection the forecasts from the model are all ‘zero’. This is not unreasonable, the monthly actual sales are very sparse being almost entirely zeros for each part number. The data is so sparse to model in monthly terms that a zero forecast every month is reasonable.

This is a dead end, the previous method, i.e. mapping monthly sales to annual forecasts (which are far less sparse) are better.

Hyper Parameter & Model Architecture Optimisation

Lets see if we can optimise the hyper params and the model architecture. We will use side by side validation for all subsequent models.

We have a good model but need to explore hyper params such as: a. number of dilations b. qty of neurons in dense layer c. number of filters d. maximum dilation, ie max number of months in sequence

We would also like to explore some model architectures, such as: a. How many hidden dense layers should we have? b. Do we need the model to be a stacked model, taking weighted average as a predictor? Does it work fine without? c. Should we be minimsing the value to order, or the qty to order?

To do this using either packages like Hyperopt/Hyperas or our own code, then we need: 1. a function to provide the data 2. a function to provide the model, including instructions to compile and fit 3. to define our model using the Keras Sequential style, not Functional (which we used above) – this is a requirement of Hyperopt 4. Amend our model with the ‘params’ dict

Talos and Hyperas are amongst the most used hyper param tuning tools for keras/python. Both randomly sample the possible combinations of parameters in order to estimate the landscape of all possible solutions. Then they use a similar approach to gradient descent in finding the lowest point in the landscape.

Hyperopt (via Hyperas wrapper) https://github.com/maxpumperla/hyperas Talos https://github.com/autonomio/talos (MIT License for commercial use, no warranty)

Annoyingly, both assume we are using Jupyter and won’t work in this environment Hyperas throws an error demanding Jupyter Talos throws this error, which does not google to any solutions: “local variable ’_hr_out’ referenced before assignment"

So, we’ll take a simpler approach using steps 1-4 above, but fewer combinations, just 32. So we can try all combinations using our own function, no gradient descent for hyper params.

```
from keras.models import Sequential
from keras.layers import Input, Conv1D, Dense, Dropout, Lambda, concatenate
from keras.optimizers import Adam
import talos as ta
def create_model_hpopt(x, y, params):
  """
  Model providing function
  """
  #Set target to optimise, value to order OR qty to order
  if params['target_type']  == 'value_to_order':
    y = y * x[:,:,3].reshape([x.shape[0], x.shape[1], 1])

  # include the weighted average of previous years sales as a predictor? Need we bother?
  if params['weighted_avg'] == 'exclude_wavg':
    x = x[:,:,[0,1,3]]

  # convolutional layer parameters
  predictor_qty  = x.shape[2]
  batch_size     = 1024  # uses most of available GPU RAM, 2048 maxes out
```

```
epochs         = 1000
dense_depth    = params['dense_depth']
n_filters      = params['n_filters']
filter_width   =    2
dilation_max   =    8 #params['dilation_max'] #more is always better
dilation_rates = [1, 2, 4, 8, 16, 32, 64, 96][0:dilation_max] # eg looking back over 96months

# define an input history series and pass it through a stack of dilated causal convolutions.
# defined using the sequential API for the sake of Hyperas
model_hpopt = Sequential()
# first dilated 1Dconv takes the input_shape
model_hpopt.add(Conv1D(filters       = n_filters,
                       kernel_size   = filter_width,
                       padding       = 'causal',
                       dilation_rate = 1,
                       input_shape   = (None, predictor_qty)))
# subsequent 1Dconvs's defined without input shape
for dilation_rate in dilation_rates[1:dilation_max]:
  model_hpopt.add(Conv1D(filters       = n_filters,
                         kernel_size   = filter_width,
                         padding       = 'causal',
                         dilation_rate = dilation_rate))

model_hpopt.add(Dense(dense_depth, activation = 'relu'))

# If we choose 'two_deep', add an additional deep layer
if params['dense_layer_qty'] == 2:
  model_hpopt.add(Dense(dense_depth, activation = 'relu'))

model_hpopt.add(Dropout(0.2))
model_hpopt.add(Dense(1))
#model.add(Lambda(lambda x: x ** 2))

# Compile
model_hpopt.compile(Adam(), loss='mean_absolute_error')

# Callbacks - early stopping only
esCallback = callbacks.EarlyStopping(monitor = 'val_loss', min_delta = 0.00005, patience = 200,
                                     verbose = 0, mode = 'auto', baseline = None)
# Fit
history = model_hpopt.fit(x                = x,
                          y                = y,
                          batch_size       = batch_size,
                          epochs           = epochs,
                          validation_split = 0.25,
                          shuffle          = True,
                          verbose          = 0,
                          callbacks        = [esCallback]) #early stopping
# return {history, model_hpopt} # in dict format for talos
return history, model_hpopt
```

Execute scan of hyper params

```python
params =  {'target_type'    : ['qty_to_order', 'value_to_order'],
           'weighted_avg'   : ['include_wavg', 'exclude_wavg'],
          #'dilation_max'   : [  7,   8], # Deprecated. More is always better
           'dense_depth'    : [128, 64],
           'n_filters'      : [ 32, 64],
           'dense_layer_qty': [  1,   2]}
           # in talos, instead of lists [] you can use tuples () eg (start, end, n)
#If using talos, we would then do this:
#h = ta.Scan(x               = x_train_transformed,
#           y               = y_train,
#           params          = params,
#           dataset_name    = 'first_test',
#           experiment_no   = '1',
#           model           = create_model_hpopt,
#           grid_downsample= 0.5)
#But talos refuses to come quietly, so we'll test all permutations
#blank results
results = pd.DataFrame(columns=['target_type', 'weighted_avg',
                                'dense_depth', 'n_filters', 'dense_layer_qty', #dilation_max,
                                'loss',        'val_loss'])

loop_count = -1
for target_type in params.get('target_type'):
  for weighted_avg in params.get('weighted_avg'):
    for dense_depth in params.get('dense_depth'):
      for n_filters in params.get('n_filters'):
        for dense_layer_qty in params.get('dense_layer_qty'):

          loop_count += 1

          params_current = {'target_type'    : target_type,
                            'weighted_avg'   : weighted_avg,
                            'dense_depth'    : dense_depth,
                            'n_filters'      : n_filters,
                           #'dilation_max'   : dilation_max,
                            'dense_layer_qty': dense_layer_qty}

          # create model
          out_history, out_model = create_model_hpopt(x      = x_train_n_valid_sidebyside_trans,
                                                      y      = y_train_n_valid_sidebyside,
                                                      params = params_current)

          # save the model to current working dir, os.getcwd()
          fileloc = './SaveModels/loop_model_' + str(loop_count) + '.h5'
          out_model.save(fileloc)

          # get mean train and validations error over last 10 epochs
          last10_train      = out_history.history['loss'][-10:]
          last10_train_mean = sum(last10_train) / float(len(last10_train))

          last10_valid      = out_history.history['val_loss'][-10:]
          last10_valid_mean = sum(last10_valid) / float(len(last10_valid))
          #save results to table for comparison
          results.loc[len(results)] = [target_type, weighted_avg,
                                       dense_depth, n_filters, dense_layer_qty, #dilation_max,
                                       last10_train_mean, last10_valid_mean]

          print("loop: ", loop_count)
          print('\n')
          print(results.loc[len(results)-1])
          print('\n')

# save to file because this took a long time (approx 1 day) to complete!
results.to_csv('./SaveModels/model_parts_hpopt.csv')
```

View the results of each test

```
# load from file because above hyper param optimisation with 2000 epochs per model takes ~24hrs
results = pd.read_csv('./SaveModels/model_parts_hpopt.csv')
pd.set_option("display.max_columns",12)
print(results >> arrange(X.val_loss, ascending=True))
```

```
##     Unnamed: 0     target_type  weighted_avg  dense_depth  n_filters  \
## 7            7    qty_to_order  include_wavg           64         64
## 3            3    qty_to_order  include_wavg          128         64
## 15          15    qty_to_order  exclude_wavg           64         64
## 11          11    qty_to_order  exclude_wavg          128         64
## 8            8    qty_to_order  exclude_wavg          128         32
## 5            5    qty_to_order  include_wavg           64         32
## 9            9    qty_to_order  exclude_wavg          128         32
## 1            1    qty_to_order  include_wavg          128         32
## 13          13    qty_to_order  exclude_wavg           64         32
## 10          10    qty_to_order  exclude_wavg          128         64
## 17          17  value_to_order  include_wavg          128         32
## 14          14    qty_to_order  exclude_wavg           64         64
## 25          25  value_to_order  exclude_wavg          128         32
## 0            0    qty_to_order  include_wavg          128         32
## 12          12    qty_to_order  exclude_wavg           64         32
## 4            4    qty_to_order  include_wavg           64         32
## 2            2    qty_to_order  include_wavg          128         64
## 16          16  value_to_order  include_wavg          128         32
## 29          29  value_to_order  exclude_wavg           64         32
## 6            6    qty_to_order  include_wavg           64         64
## 21          21  value_to_order  include_wavg           64         32
## 27          27  value_to_order  exclude_wavg          128         64
## 28          28  value_to_order  exclude_wavg           64         32
## 20          20  value_to_order  include_wavg           64         32
## 24          24  value_to_order  exclude_wavg          128         32
## 30          30  value_to_order  exclude_wavg           64         64
## 22          22  value_to_order  include_wavg           64         64
## 26          26  value_to_order  exclude_wavg          128         64
## 23          23  value_to_order  include_wavg           64         64
## 19          19  value_to_order  include_wavg          128         64
## 31          31  value_to_order  exclude_wavg           64         64
## 18          18  value_to_order  include_wavg          128         64
##
##     dense_layer_qty      loss  val_loss
## 7                   2  0.665890  0.853402
## 3                   2  0.675015  0.868372
## 15                  2  0.684969  0.870714
## 11                  2  0.656104  0.874196
## 8                   1  0.744366  0.875167
## 5                   2  0.709065  0.875519
## 9                   2  0.723422  0.883722
## 1                   2  0.713832  0.886183
## 13                  2  0.707888  0.889304
## 10                  1  0.742253  0.896905
## 17                  2  0.249266  0.900952
## 14                  1  0.767045  0.903791
## 25                  2  0.259687  0.903908
## 0                   1  0.744422  0.904926
## 12                  1  0.772281  0.907492
## 4                   1  0.797846  0.909758
## 2                   1  0.743227  0.910566
## 16                  1  0.292012  0.918178
## 29                  2  0.215560  0.920305
## 6                   1  0.800652  0.925881
## 21                  2  0.271849  0.926230
## 27                  2  0.275874  0.932719
## 28                  1  0.247479  0.933052
## 20                  1  0.320098  0.935827
## 24                  1  0.305618  0.936512
## 30                  1  0.305882  0.939557
## 22                  1  0.299196  0.944960
## 26                  1  0.324222  0.946711
## 23                  2  0.267518  0.961582
## 19                  2  0.276250  0.962394
## 31                  2  0.303313  0.968900
## 18                  1  0.324019  0.971848
```

We must now compare the best model, model 7, with the baseline we prepared earlier.

```
#prep test data
x_test_hpopt = get_time_block_series(series_array  = series_array_x_test[:,:,:],
                                     date_to_index = date_to_index,
                                     start_date    = encoding_firstMth_test,
                                     end_date      = encoding_finalMth_test)
                                     #(n_series, n_timesteps, n_features)
x_test_hpopt_transformed, x_test_hpopt_mean = transform(x_test_hpopt)
```

We load the models from file because the above training took many hours.

```
from keras.models import load_model
model_hpopt_best      = load_model('./SaveModels/loop_model_7.h5')
preds_hpopt_best_test = model_hpopt_best.predict(x_test_hpopt_transformed)
```

now compare the best model architecture with baseline:

```
preds_hpopt_best_test <- py$preds_hpopt_best_test

print("BASELINE: Test results")
```

```
## [1] "BASELINE: Test results"
```

```r
cat("\n")
```

```r
get_wavg_vs_keras(preds_test_sidebyside, wavg_preds_test_r,
                  unitprice_test_r,      actuals_test_r,
                  byval = FALSE)
```

```
## [1] "Number of parts in analysis     : 4991"
## [1] "Error with deep model   (GBP/yr): 230989. Over Order  = 37511. Under Order = -193477"
## [1] "Error with weighted avg (GBP/yr): 293015. Over Order  = 8496. Under Order = -284519"
```

```r
cat("\n")
```

```r
print(paste0("BEST MODEL: Test results "))
```

```
## [1] "BEST MODEL: Test results "
```

```r
cat("\n")
```

```r
get_wavg_vs_keras(preds_hpopt_best_test, wavg_preds_test_r,
                  unitprice_test_r,      actuals_test_r,
                  byval = FALSE)
```

```
## [1] "Number of parts in analysis     : 4991"
## [1] "Error with deep model   (GBP/yr): 212254. Over Order  = 26475. Under Order = -185779"
## [1] "Error with weighted avg (GBP/yr): 293015. Over Order  = 8496. Under Order = -284519"
```

The best model has improved on the baseline by 8% on the test set, well worth having. For reference the architecture of this model is as follows:

```
model_hpopt_best.summary()
```

```
## _________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =================================================================
## conv1d_113 (Conv1D)          (None, None, 64)          576
## _________________________________________________________________
## conv1d_114 (Conv1D)          (None, None, 64)          8256
## _________________________________________________________________
## conv1d_115 (Conv1D)          (None, None, 64)          8256
## _________________________________________________________________
## conv1d_116 (Conv1D)          (None, None, 64)          8256
## _________________________________________________________________
## conv1d_117 (Conv1D)          (None, None, 64)          8256
## _________________________________________________________________
## conv1d_118 (Conv1D)          (None, None, 64)          8256
## _________________________________________________________________
## conv1d_119 (Conv1D)          (None, None, 64)          8256
## _________________________________________________________________
## conv1d_120 (Conv1D)          (None, None, 64)          8256
## _________________________________________________________________
## dense_32 (Dense)             (None, None, 64)          4160
## _________________________________________________________________
## dense_33 (Dense)             (None, None, 64)          4160
## _________________________________________________________________
## dropout_15 (Dropout)         (None, None, 64)          0
## _________________________________________________________________
## dense_34 (Dense)             (None, None, 1)           65
## ================================================================
## Total params: 66,753
## Trainable params: 66,753
## Non-trainable params: 0
## ________________________________________________________________
```

Future Development Possibilities

1. Investigate alternative architectures (LSTM or GRU ?)
2. Improve training spped with "Stochastic Weight Averaging" https://medium.com/@hortonhearsafoo/adding-a-cutting-edge-deep-learning-training-technique-to-the-fast-ai-library-2cd1dba90a49 for example, carry out the first 100 epochs of training as per usual, but in subsequent epochs calculate running average of weights. Harder than it sounds. This leads to better/faster training.