Time series forecasting - with deep learning

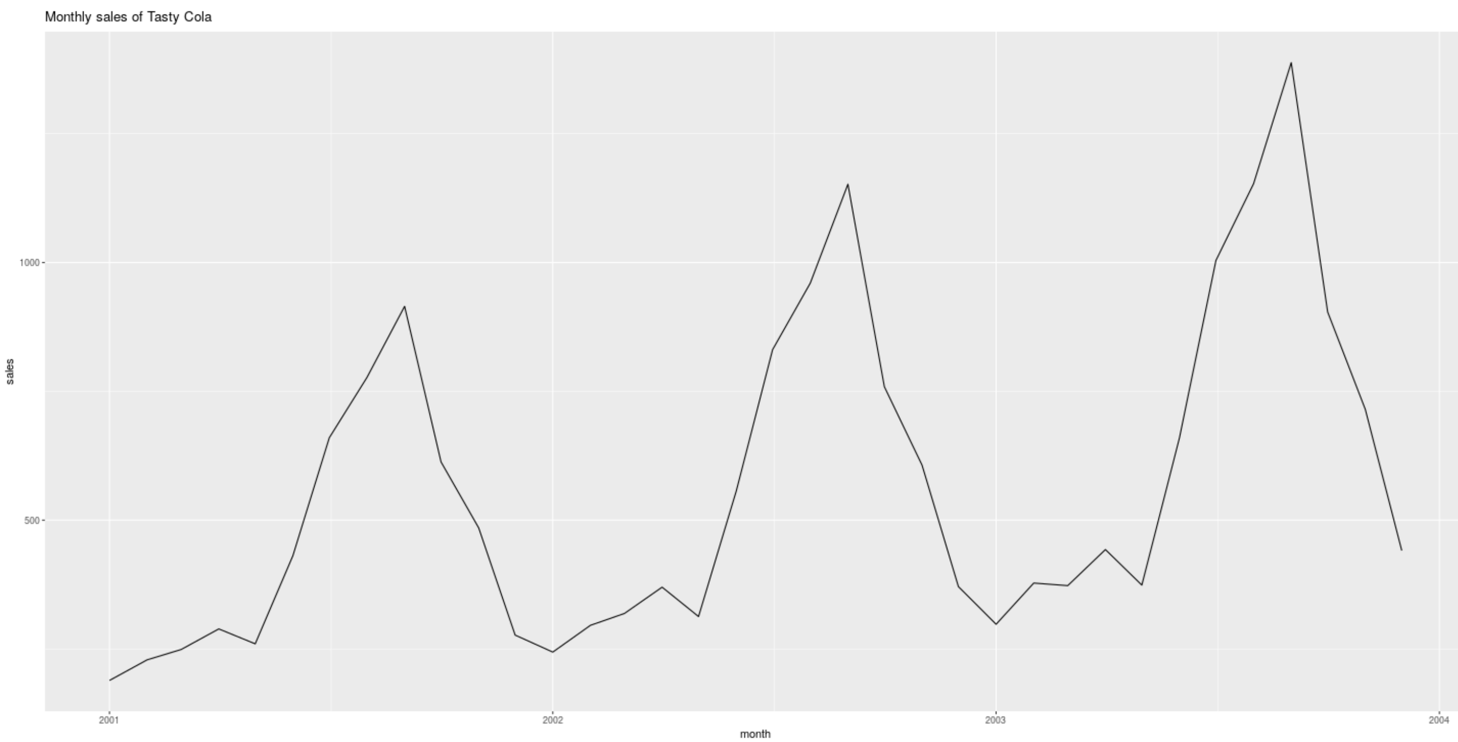
Sigrid Keydana, Trivadis GmbH

2017-05-23

Time series forecasting: the classical approach

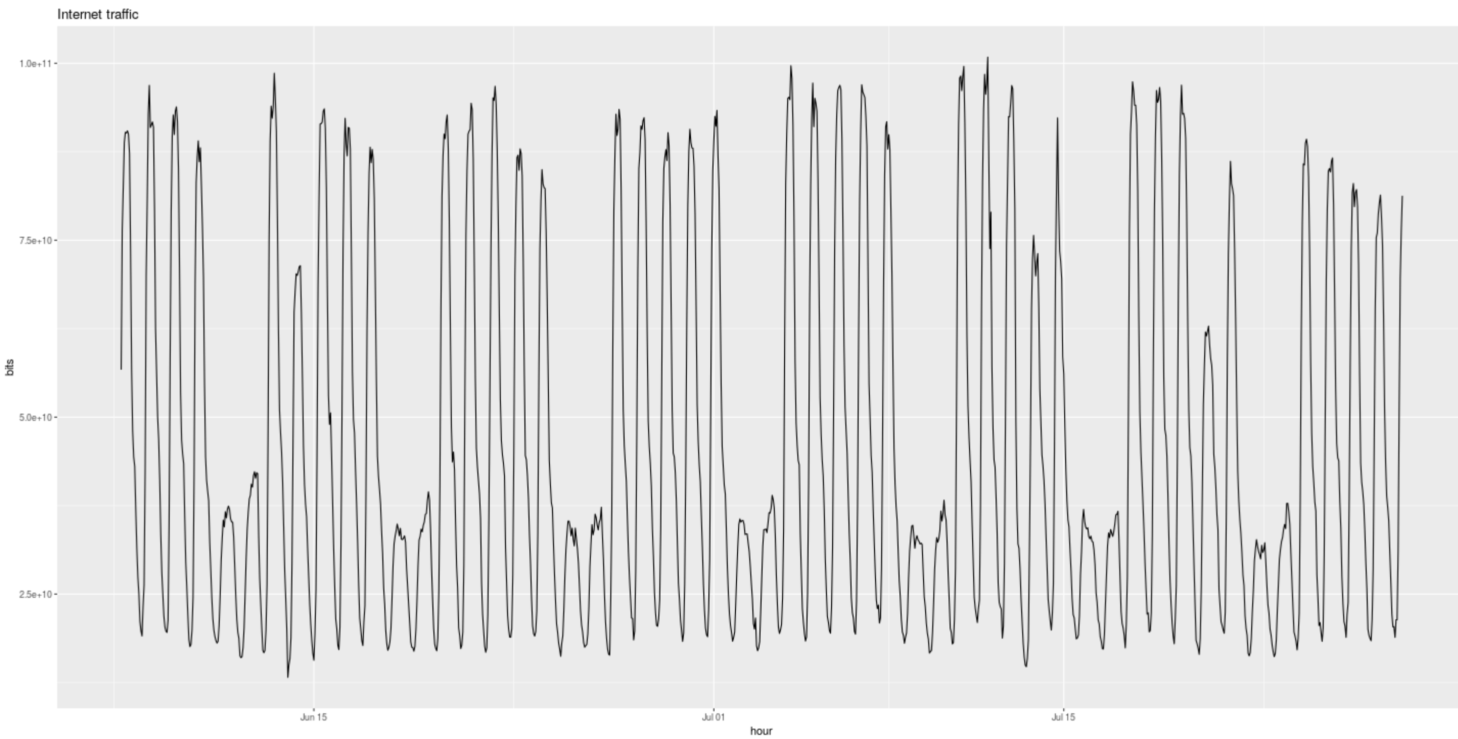
A time series

```
cola_df <- read_csv("monthly-sales-of-tasty-cola.csv", col_names = c("month", "sales"), skip = 1,
                    col_types = cols(month = col_date("%y-%m")))
ggplot(cola_df, aes(x = month, y = sales)) + geom_line() + ggtitle("Monthly sales of Tasty Cola")
```



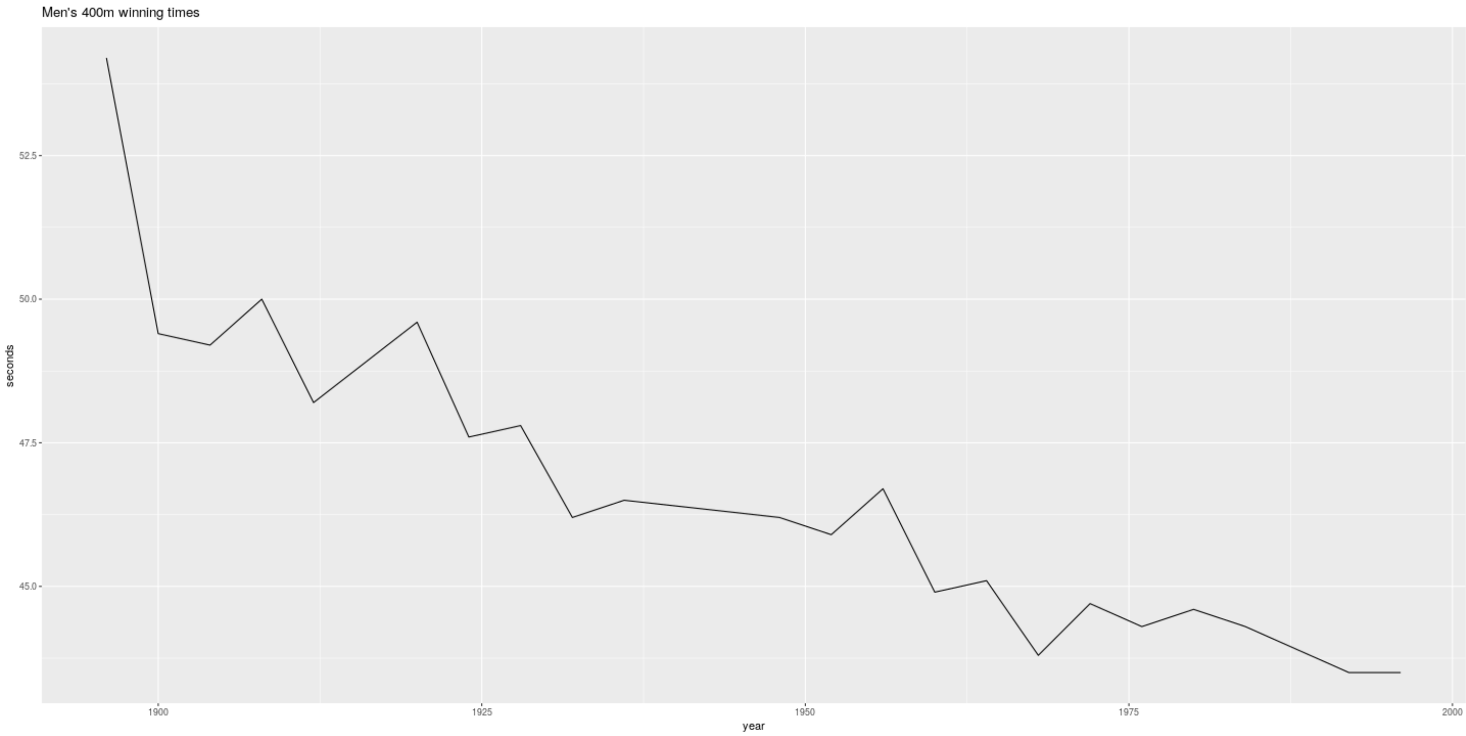
Another one ...

```
traffic_df <- read_csv("internet-traffic-data-in-bits-fr.csv", col_names = c("hour", "bits"), skip
= 1)
ggplot(traffic_df, aes(x = hour, y = bits)) + geom_line() + ggtitle("Internet traffic")
```



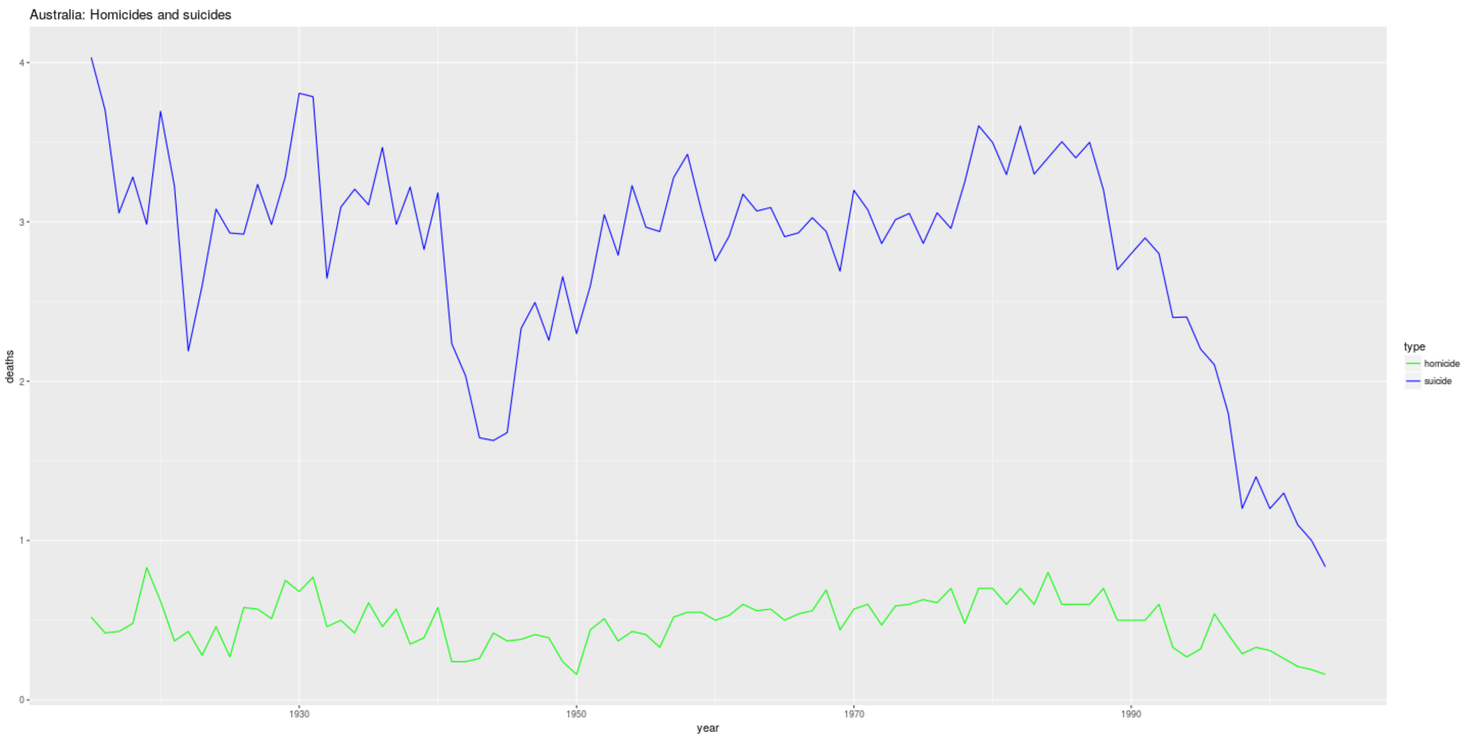
And another.

```
win_df <- read_csv("winning-times-for-the-mens-400-m.csv", col_names = c("year", "seconds"), skip =
1)
ggplot(win_df, aes(x = year, y = seconds)) + geom_line() + ggtitle("Men's 400m winning times")
```



Sometimes we may look at several time series together.

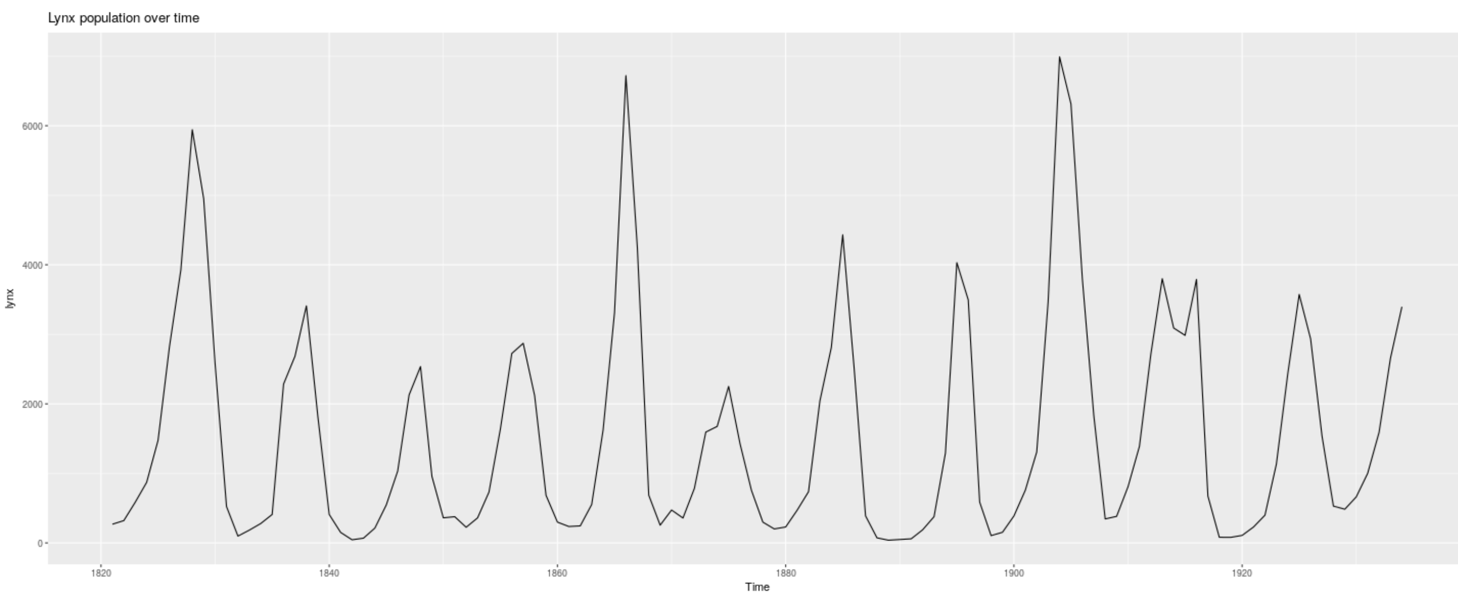
```
deaths_df <- read_csv("deaths-from-homicides-and-suicid.csv", col_names = c("year", "homicide",
"suicide"), skip = 1)
deaths_df <- gather(deaths_df, key = 'type', value = 'deaths', homicide:suicide)
ggplot(deaths_df, aes(x = year, y = deaths, color = type)) + geom_line() +
scale_colour_manual(values=c("green","blue")) + ggtitle("Australia: Homicides and suicides")
```



Sometimes there's more to the picture than we might think.

So far, this is nothing but a univariate time series of lynx population.

```
data("lynx")
autoplot(lynx) + ggtitle("Lynx population over time")
```

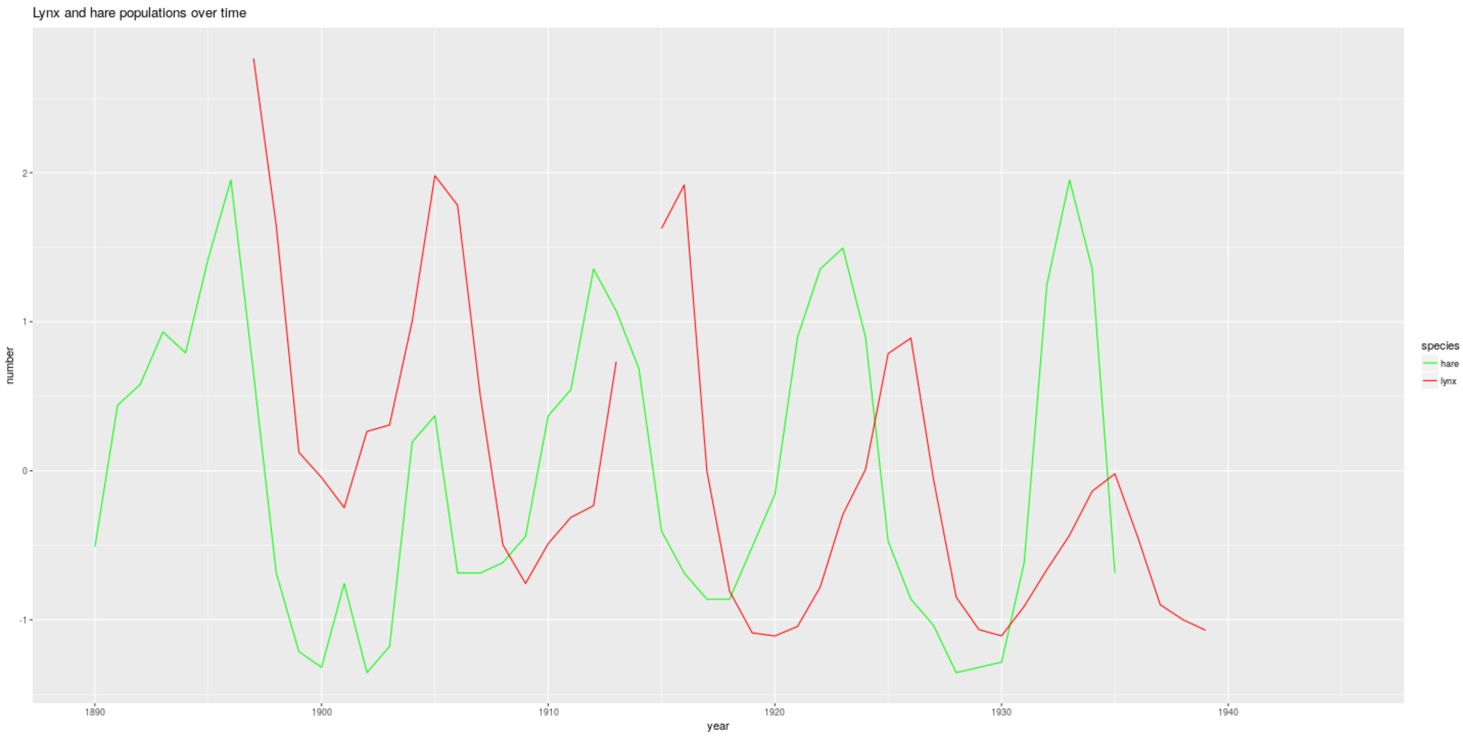


However …

Source: Rudolfo's Usenet Animal Pictures Gallery (link no longer exists)

Lynx and hare

```
lynx_df <- read_delim("lynxhare.csv", delim = ";") %>% select(year, hare, lynx) %>%
filter(between(year, 1890, 1945)) %>% mutate(hare = scale(hare), lynx = scale(lynx))
lynx_df <- gather(lynx_df, key = 'species', value = 'number', hare:lynx)
ggplot(lynx_df, aes(x = year, y = number, color = species)) + geom_line() +
scale_colour_manual(values=c("green","red")) + ggtitle("Lynx and hare populations over time")
```



Concepts in classical time series modeling

- Stationarity
- Decomposition
- Autocorrelation

Wait. This will be about deep learning

… why would the classical approach even matter?

Stationarity (1)

- We want to forecast future values of a time series
- We need fundamental statistical properties like mean, variance …
- What *is* the mean, or the variance, of a time series?

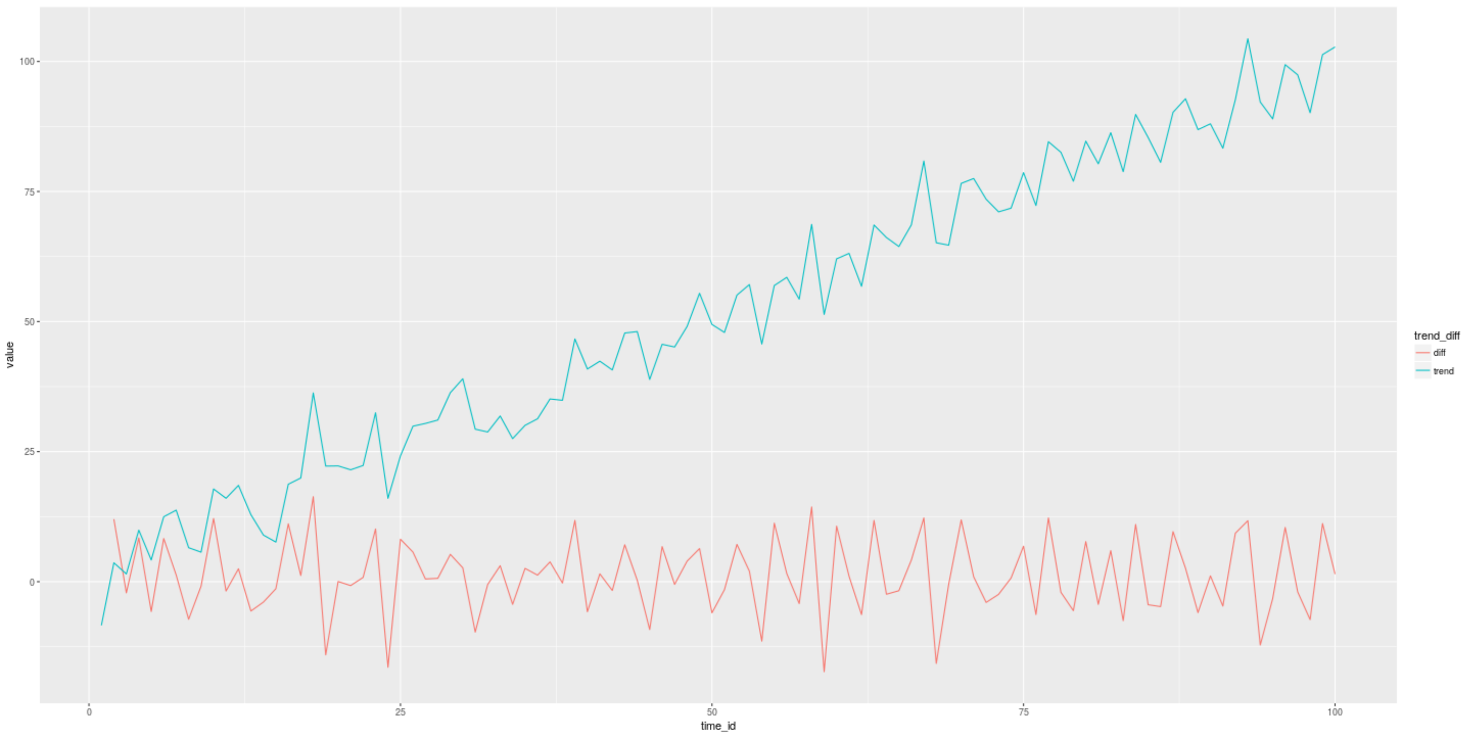
Stationarity (2)

- If a time series y_t is stationary, then for all s, the distribution of (y_t, y_{t+s}) does not depend on t
- By ergodicity, after we remove any trend and seasonality effects, we may assume that the residual series is stationary in the mean: $\mu(t) = t$

Differencing

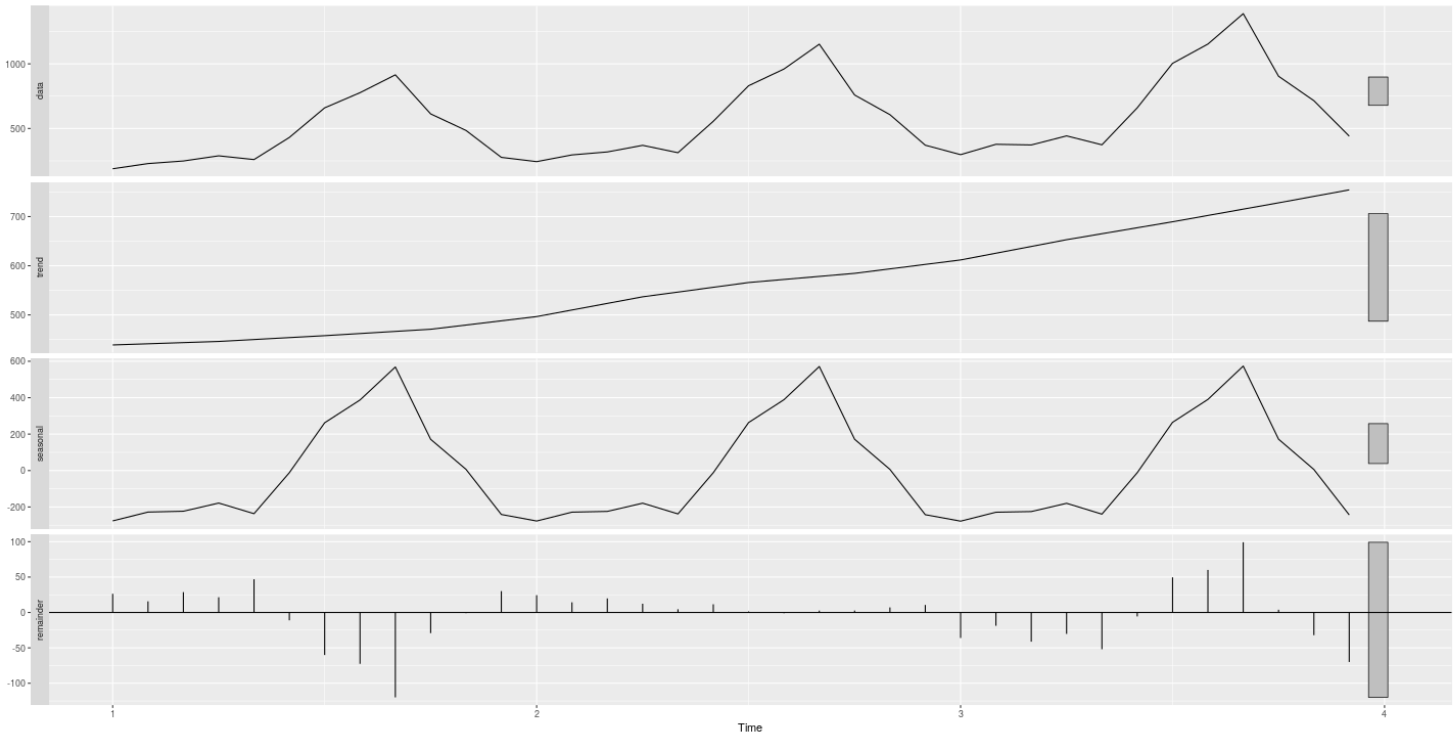
- The trend is usually removed using differencing (forming the differences of neighboring values)

```
set.seed(7777)
trend <- 1:100 + rnorm(100, sd = 5)
diff <- diff(trend)
df <- data_frame(time_id = 1:100,
                 trend = trend,
                 diff = c(NA, diff))
df <- df %>% gather(key = 'trend_diff', value = 'value', -time_id)
ggplot(df, aes(x = time_id, y = value, color = trend_diff)) + geom_line()
```



Time series decomposition

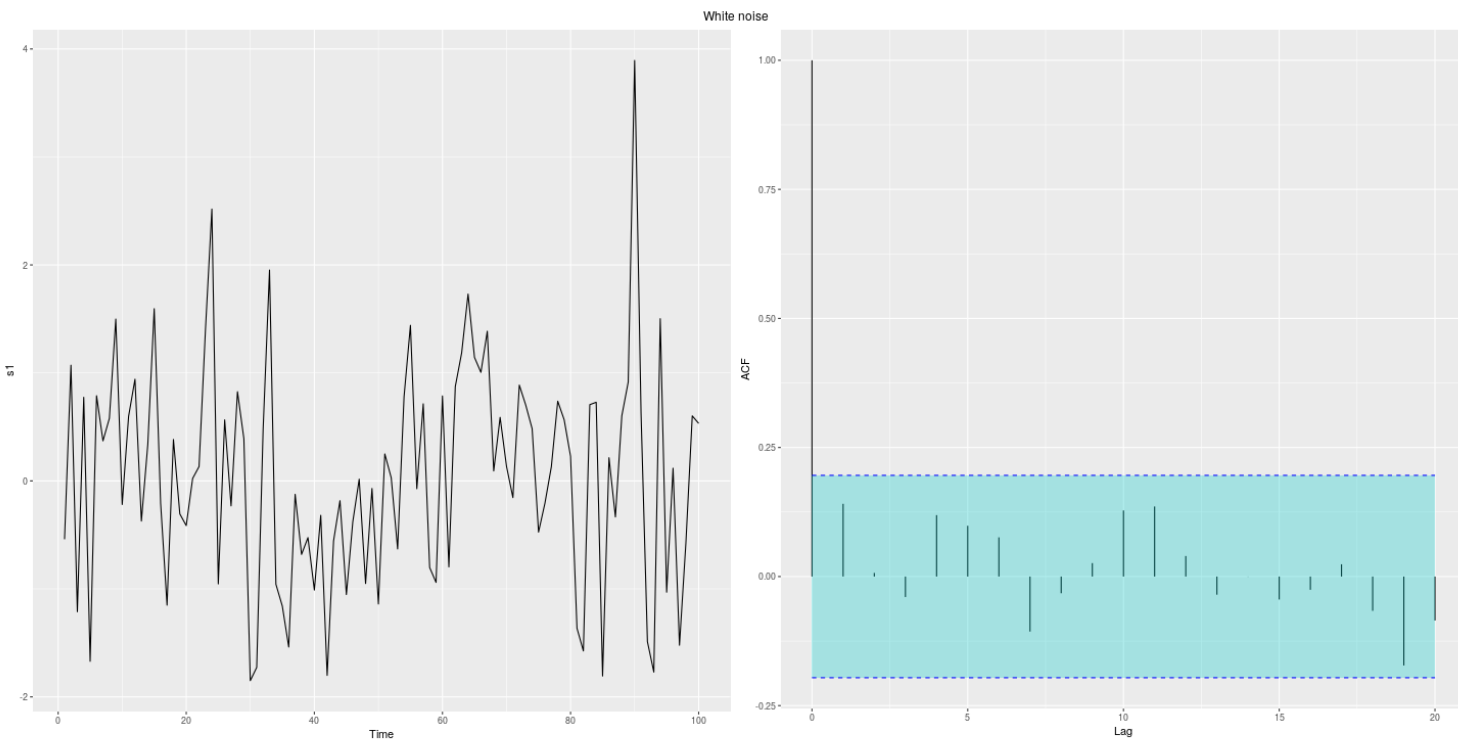
```
autoplot(stl(ts(cola_df$sales, frequency=12), s.window = 12))
```



Autocorrelation - Case 1: White noise

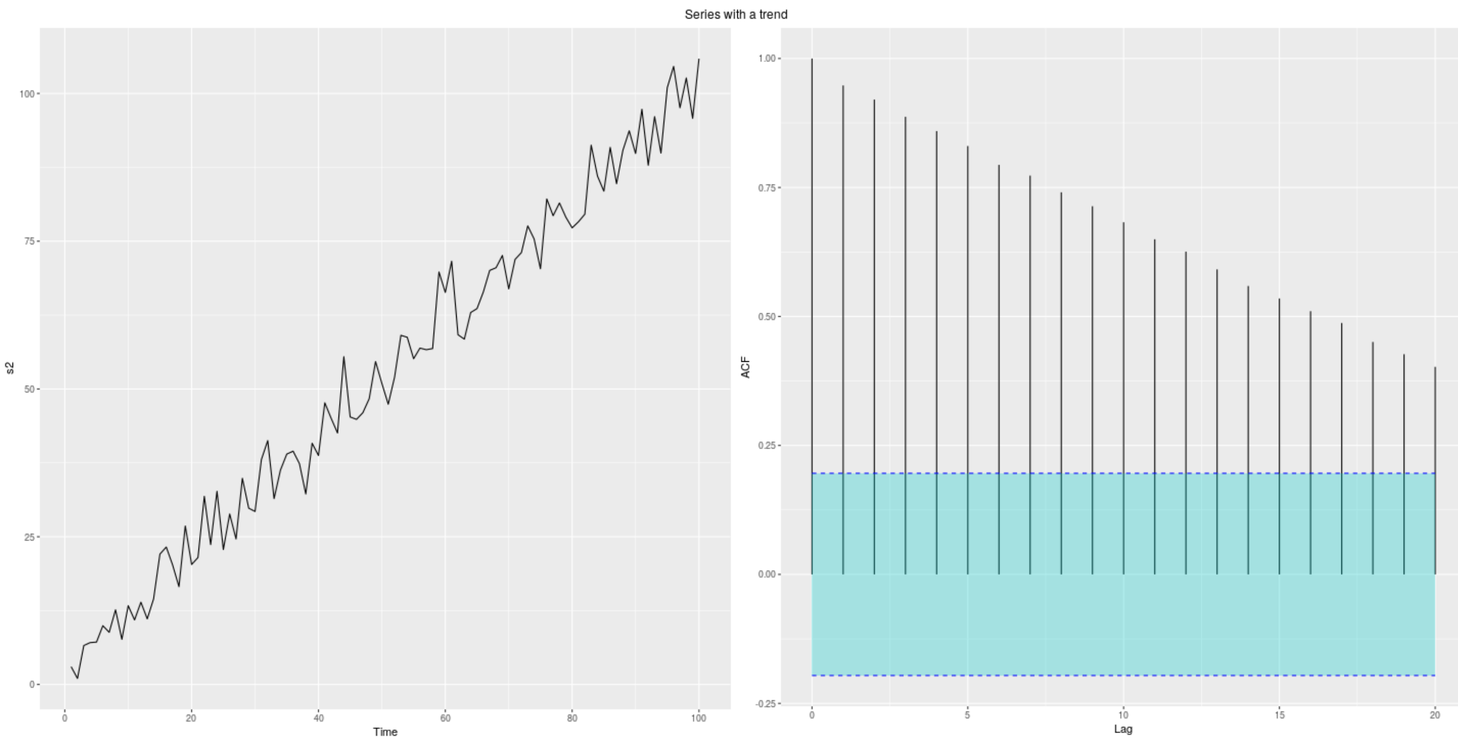
If consecutive values were not related, there'd be no way of forecasting future values

```
s1 <- ts(rnorm(100))
ts1 <- autoplot(s1)
acf1 <- ggfortify:::autoplot.acf(acf(s1, plot = FALSE), conf.int.fill = '#00cccc', conf.int.value =
0.95)
do.call('grid.arrange', list('grobs' = list(ts1, acf1), 'ncol' = 2, top = "White noise"))
```



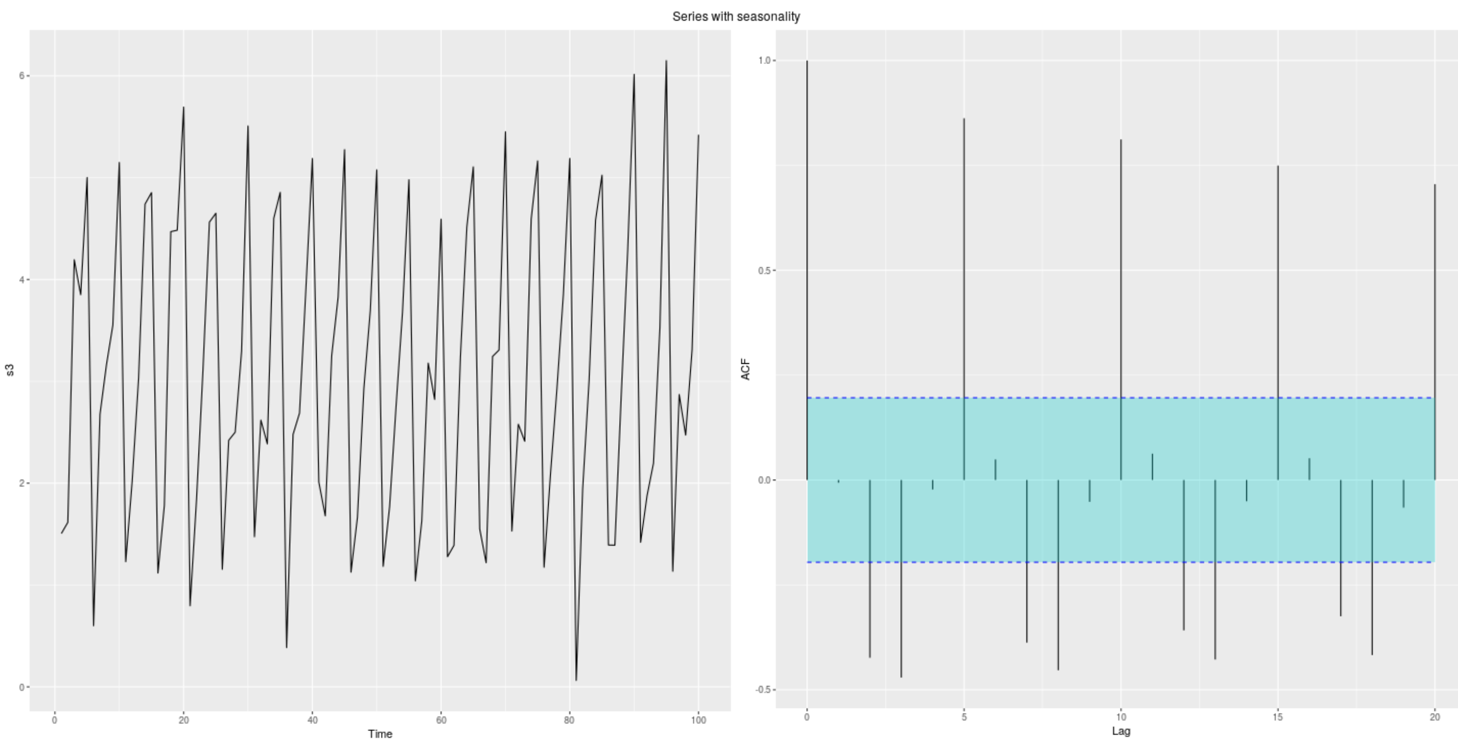
Autocorrelation - Case 2: Linear trend

```
s2 <- ts(1:100 + rnorm(100, 2, 4))
ts2 <- autoplot(s2)
acf2 <- ggfortify:::autoplot.acf(acf(s2, plot = FALSE), conf.int.fill = '#00cccc', conf.int.value =
0.95)
do.call('grid.arrange', list('grobs' = list(ts2, acf2), 'ncol' = 2, top = "Series with a trend"))
```



Autocorrelation - Case 3: Seasonality

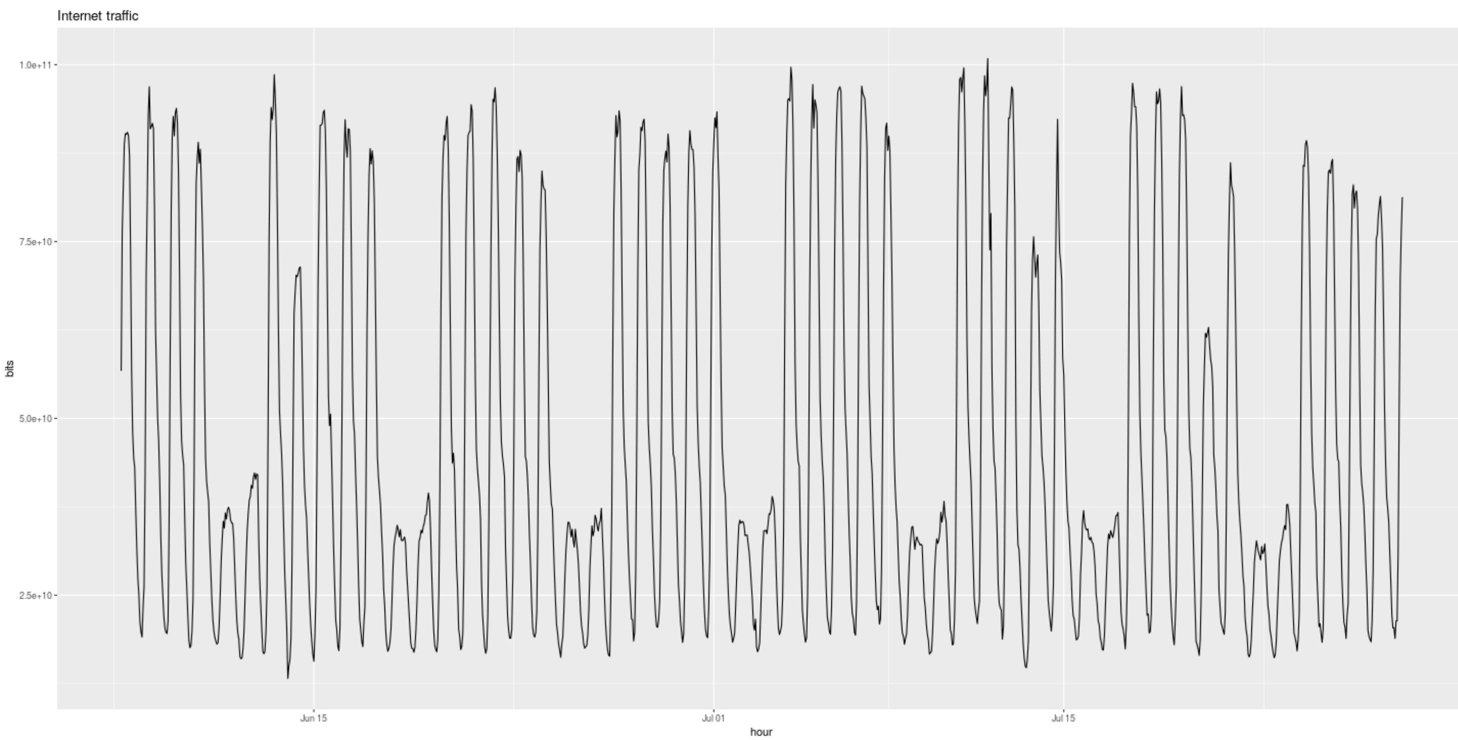
```
s3 <- ts(rep(1:5,20) + rnorm(100, sd= 0.5))
ts3 <- autoplot(s3)
acf3 <- ggfortify:::autoplot.acf(acf(s3, plot = FALSE), conf.int.fill = '#00cccc', conf.int.value =
0.95)
do.call('grid.arrange', list('grobs' = list(ts3, acf3), 'ncol' = 2, top = "Series with
seasonality"))
```



Forecasting internet traffic, the classical way

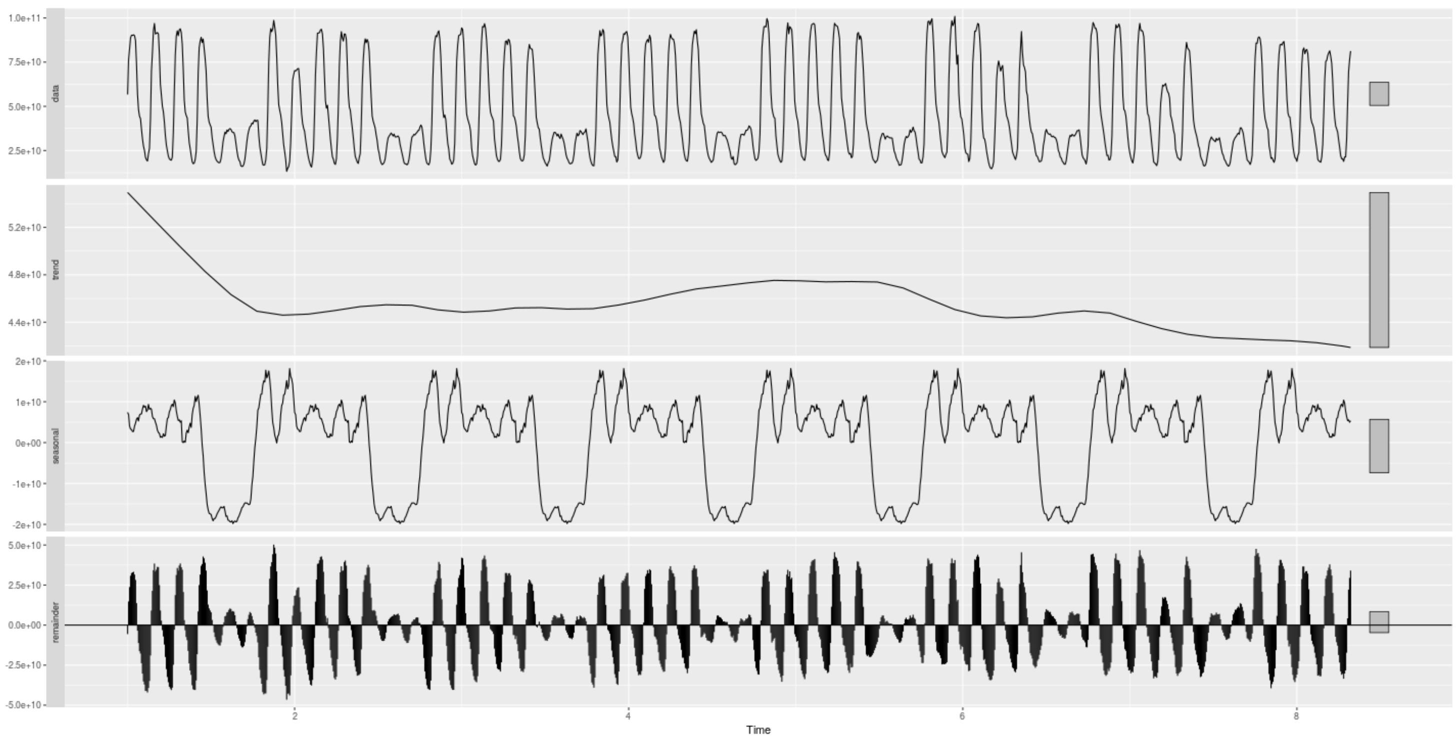
Here's the traffic time series again.

```
ggplot(traffic_df, aes(x = hour, y = bits)) + geom_line() + ggtitle("Internet traffic")
```



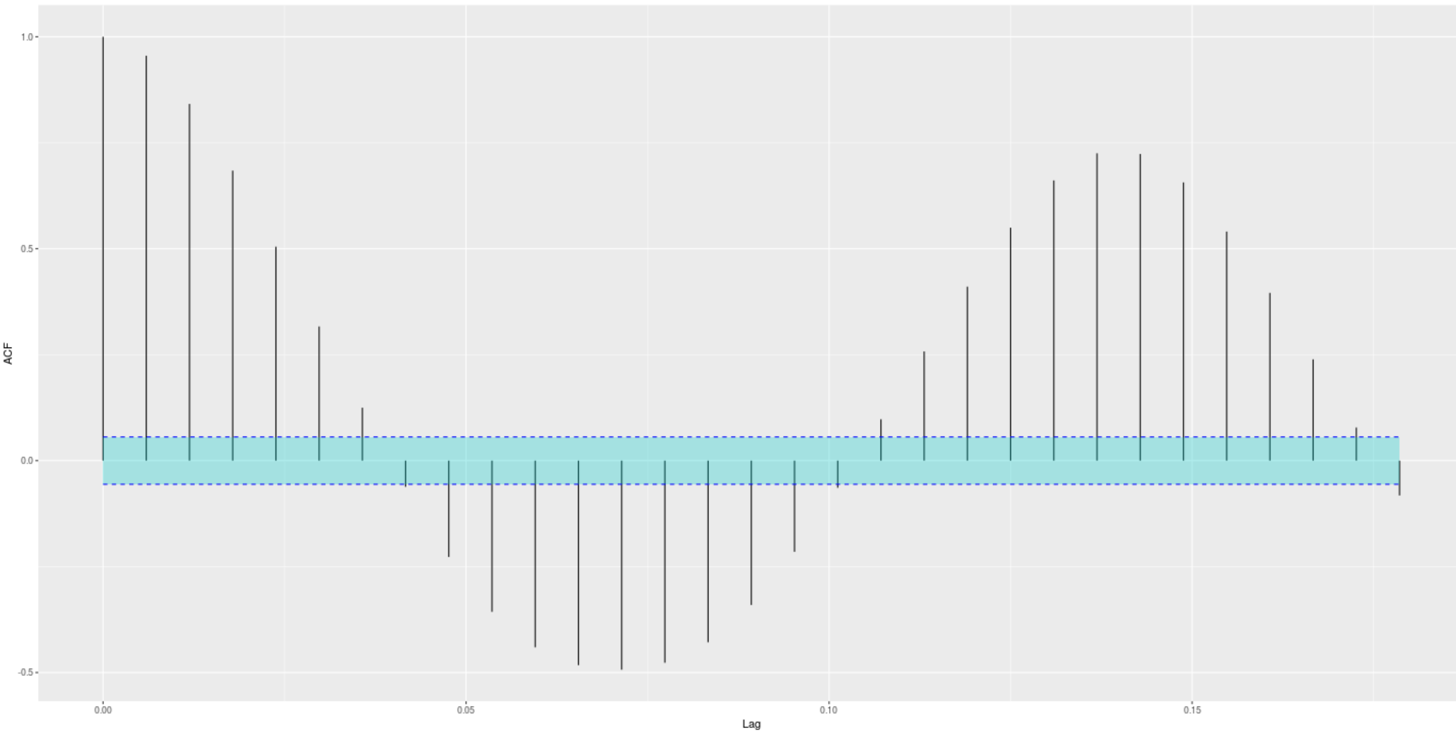
Let's first look at decomposition.

```
traffic_ts <- msts(traffic_df$bits,seasonal.periods = c(24, 24*7))
autoplot(stl(traffic_ts, s.window = 7 * 24))
```



How about autocorrelation?

```
ggfortify:::autoplot.acf(acf(traffic_ts, plot = FALSE), conf.int.fill = '#00cccc', conf.int.value =
0.95)
```



The usual ARIMA won't work...

```
arima_fit <- auto.arima(traffic_ts, stepwise = FALSE, max.order = 10, trace = TRUE)
```

```
 ARIMA(0,0,0) with zero mean     : 60774.59
 ARIMA(0,0,0) with non-zero mean : Inf *
 ARIMA(0,0,0)(0,0,1)[168] with zero mean     : Inf
 ARIMA(0,0,0)(0,0,1)[168] with non-zero mean : Inf *
```

```
Error in myarima(x, order = c(i, d, j), seasonal = c(I, D, J), constant = (K == : root finding code
failed
```

… will regression with ARIMA errors?

Let's add an indicator variable for whether it's weekend.

```
traffic_df_wd <- traffic_df %>% mutate(weekend = if_else(wday(hour) %in% c(7,1), 1, 0))
ggplot(traffic_df_wd, aes(x=hour, y=bits, color=factor(weekend))) + geom_point()
```

No.

This will run forever and you'll have to kill it.

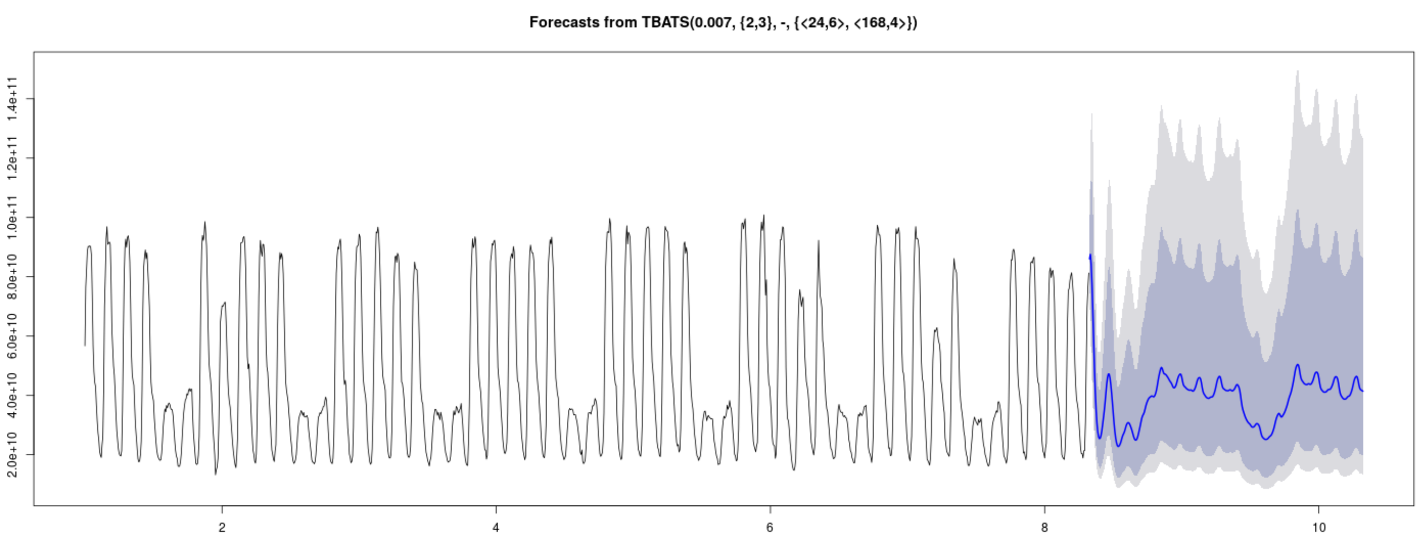
```
# arima_fit <- auto.arima(ts(traffic_df_wd$bits, frequency = 24 * 7), xreg = traffic_df_wd$weekend,
#                         stepwise = FALSE, max.order = 10, trace = TRUE)
```

Trying TBATS

TBATS ("Exponential smoothing state space model with Box-Cox transformation, ARMA errors, Trend and Seasonal components") does *not* fail…

But, look at the forecast.

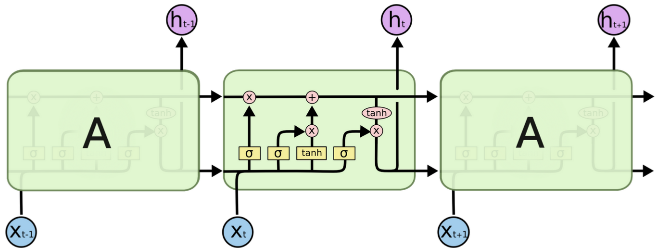
```
tbats_fit <- tbats(traffic_ts)
plot(forecast(tbats_fit, h=14*24))
```



Will deep learning do any better for this time series?

- Let's step back a little though.
- How does deep learning even do time series?

Enter: LSTM (Long Short Term Memory)



$$
\begin{aligned}
i^{(t)} &= \sigma(W^{(i)}x^{(t)} + U^{(i)}h^{(t-1)}) && \text{(Input gate)}\\
f^{(t)} &= \sigma(W^{(f)}x^{(t)} + U^{(f)}h^{(t-1)}) && \text{(Forget gate)}\\
o^{(t)} &= \sigma(W^{(o)}x^{(t)} + U^{(o)}h^{(t-1)}) && \text{(Output/Exposure gate)}\\
\tilde{c}^{(t)} &= \tanh(W^{(c)}x^{(t)} + U^{(c)}h^{(t-1)}) && \text{(New memory cell)}\\
c^{(t)} &= f^{(t)} \circ \tilde{c}^{(t-1)} + i^{(t)} \circ \tilde{c}^{(t)} && \text{(Final memory cell)}\\
h^{(t)} &= o^{(t)} \circ \tanh(c^{(t)})
\end{aligned}
$$

Source: **Christopher Olah's post on LSTM**

New world, new rules?

- Do we still care about stationarity and decomposition?
- How does DL handle trends, or seasonality?

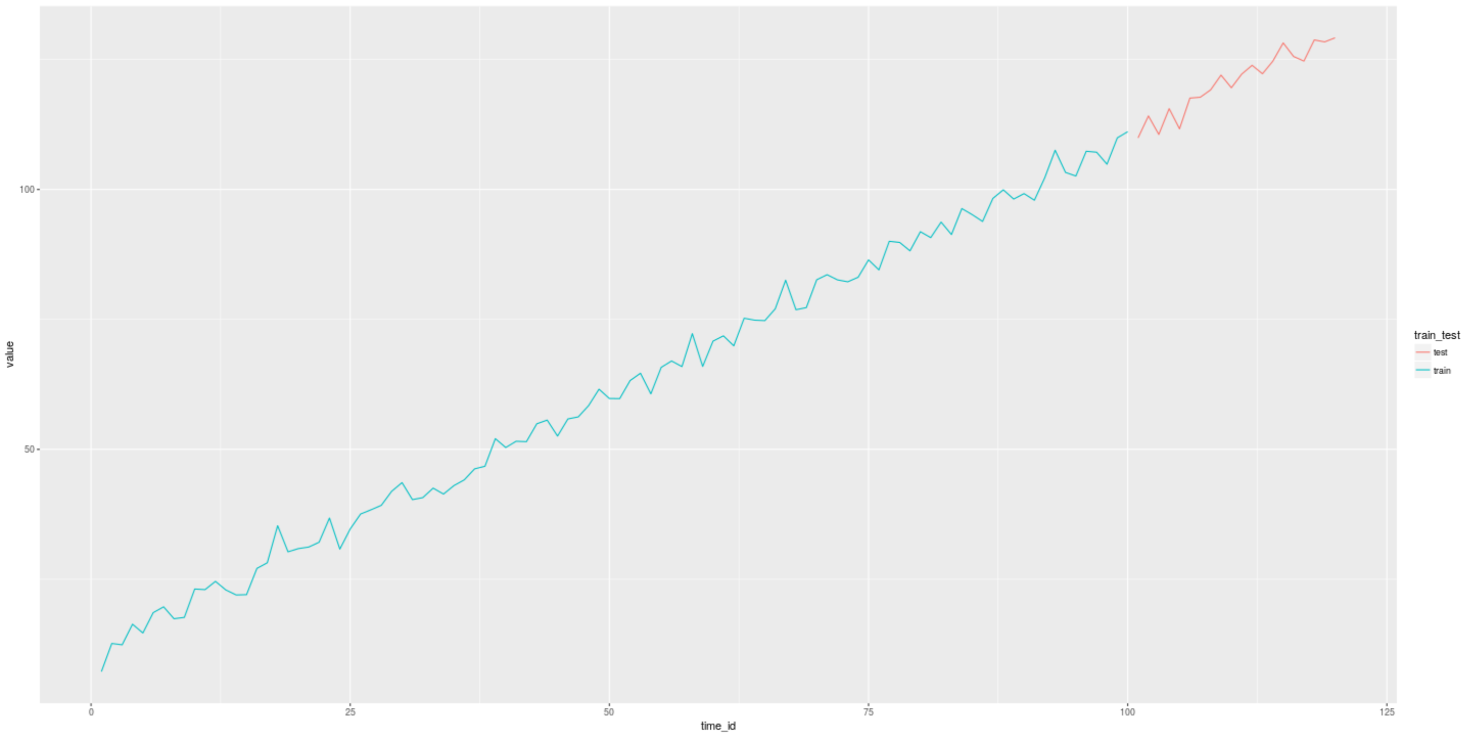
Let's compare ARIMA vs. LSTM on a little set of benchmarks

- synthetic dataset, with trend only, test data out-of-range
- synthetic dataset, with trend only, test data in-range
- synthetic dataset, seasonal only

ARIMA vs. LSTM, Round 1: Trend-only dataset, test data out-of-range

Trend-only dataset

```r
set.seed(7777)
trend_train <- 11:110 + rnorm(100, sd = 2)
trend_test <- 111:130 + rnorm(20, sd =2)
df <- data_frame(time_id = 1:120,
                 train = c(trend_train, rep(NA, length(trend_test))),
                 test = c(rep(NA, length(trend_train)), trend_test))
df <- df %>% gather(key = 'train_test', value = 'value', -time_id)
ggplot(df, aes(x = time_id, y = value, color = train_test)) + geom_line()
```



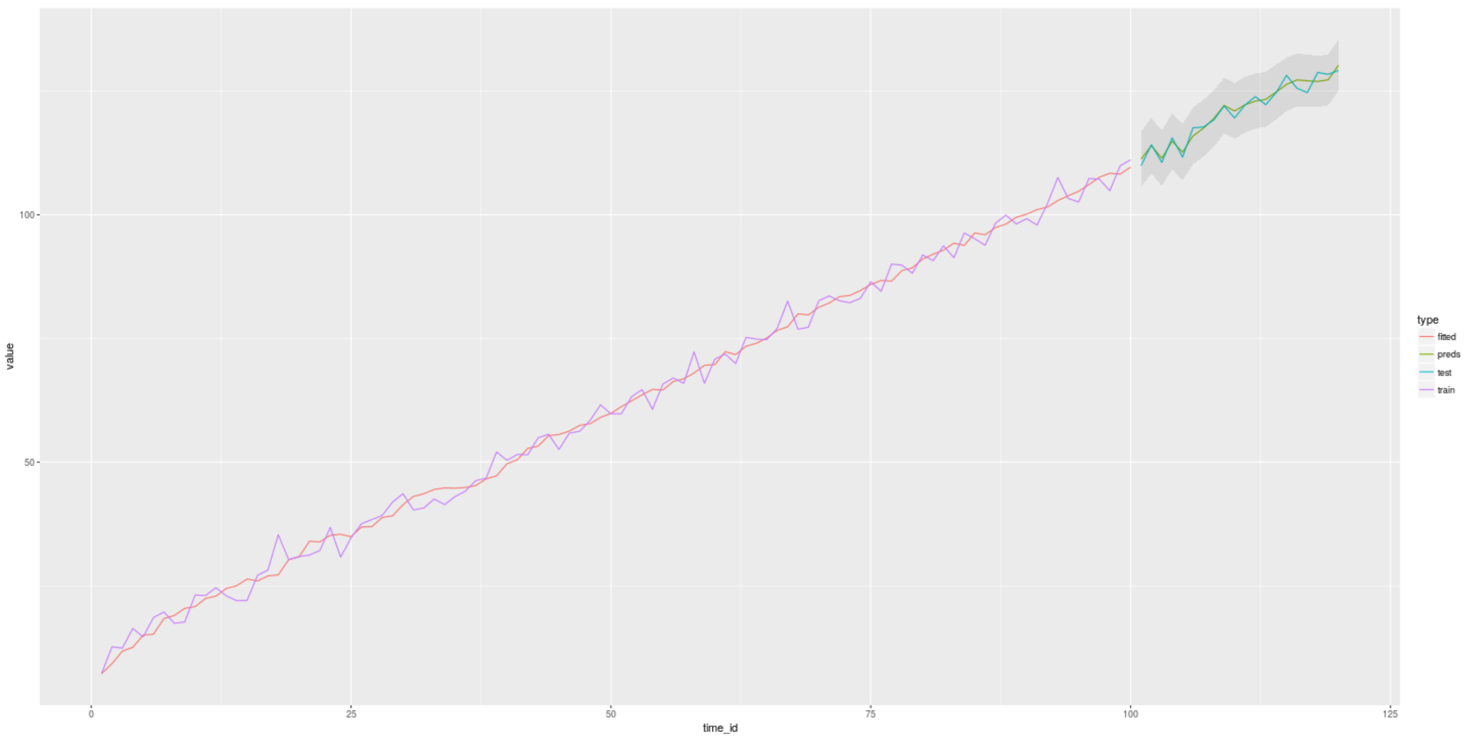
Trend-only dataset: Enter: ARIMA (1)

```
set.seed(7777)
trend_train <- 11:110 + rnorm(100, sd = 2)
trend_test <- 111:130 + rnorm(20, sd =2)
h <- 1
n <- length(trend_test) - h + 1
fit <- auto.arima(trend_train)

# re-estimate the model as new data arrives, as per https://robjhyndman.com/hyndsight/rolling-
forecasts/
order <- arimaorder(fit)
predictions <- matrix(0, nrow=n, ncol=h)
lower <- matrix(0, nrow=n, ncol=h) # 95% prediction interval
upper <- matrix(0, nrow=n, ncol=h)
for(i in 1:n) {
  x <- c(trend_train[(1+i):length(trend_train)], trend_test[1:i])
  refit <- Arima(x, order=order[1:3], seasonal=order[4:6])
  predictions[i,] <- forecast(refit, h=h)$mean
  lower[i,] <- unclass(forecast(refit, h=h)$lower)[,2]
  upper[i,] <- unclass(forecast(refit, h=h)$upper)[,2]
}

(test_rsme <- sqrt(sum((trend_test - predictions)^2)))
```

```
[1] 5.31686
```

Trend-only dataset: Enter: ARIMA (2)

```
df <- data_frame(time_id = 1:120,
  train = c(trend_train, rep(NA, length(trend_test))),
  test = c(rep(NA, length(trend_train)), trend_test),
  fitted = c(fit$fitted, rep(NA, length(trend_test))),
  preds = c(rep(NA, length(trend_train)), predictions),
  lower = c(rep(NA, length(trend_train)), lower),
  upper = c(rep(NA, length(trend_train)), upper))
df <- df %>% gather(key = 'type', value = 'value', train:preds)
ggplot(df, aes(x = time_id, y = value)) + geom_line(aes(color = type)) + geom_ribbon(aes(ymin =
lower, ymax = upper), alpha = 0.1)
```



Trend-only dataset: Enter: LSTM

- Let's first show what we have to do for time series prediction with LSTM networks.
- We'll choose the **Keras** framework, and the R bindings provided by **kerasR**.

Background: Data preparation for LSTM in Keras (1)

Firstly, LSTMs work with a sliding window of input, so we need to provide the data (train and test) in "window form":

```
# example given for training set, - do the same for test set
# length(trend_train)
lstm_num_timesteps <- 5
X_train <- t(sapply(1:(length(trend_train) - lstm_num_timesteps), function(x) trend_train[x:(x +
lstm_num_timesteps - 1)]))
# dim(X_train)
X_train[1:5, ]
```

```
         [,1]     [,2]     [,3]     [,4]     [,5]
[1,]  7.24409 12.65291 12.39629 16.36328 14.67253
[2,] 12.65291 12.39629 16.36328 14.67253 18.59298
[3,] 12.39629 16.36328 14.67253 18.59298 19.70279
[4,] 16.36328 14.67253 18.59298 19.70279 17.41485
[5,] 14.67253 18.59298 19.70279 17.41485 17.67259
```

```
y_train <- sapply((lstm_num_timesteps + 1):(length(trend_train)), function(x) trend_train[x])
y_train[1:5]
```

```
[1] 18.59298 19.70279 17.41485 17.67259 23.13023
```

Background: Data preparation for LSTM in Keras (2)

Keras LSTMs expect the input array to be shaped as (no. samples, no. time steps, no. features)

```
# example given for training set, - do the same for test set
# add 3rd dimension
dim(X_train)
```

```
[1] 95  5
```

```
X_train <- expand_dims(X_train, axis = 2)
dim(X_train)
```

```
[1] 95  5  1
```

```
# LSTM input shape: (samples, time steps, features)
num_samples <- dim(X_train)[1]
num_steps <- dim(X_train)[2]
num_features <- dim(X_train)[3]
c(num_samples, num_steps, num_features)
```

```
[1] 95  5  1
```

Background: Keras - build the model

```
model <- Sequential()
model$add(LSTM(units = 4, input_shape=c(num_steps, num_features)))
model$add(Dense(1))
keras_compile(model, loss='mean_squared_error', optimizer='adam')
```

Background: Keras - fit the model!

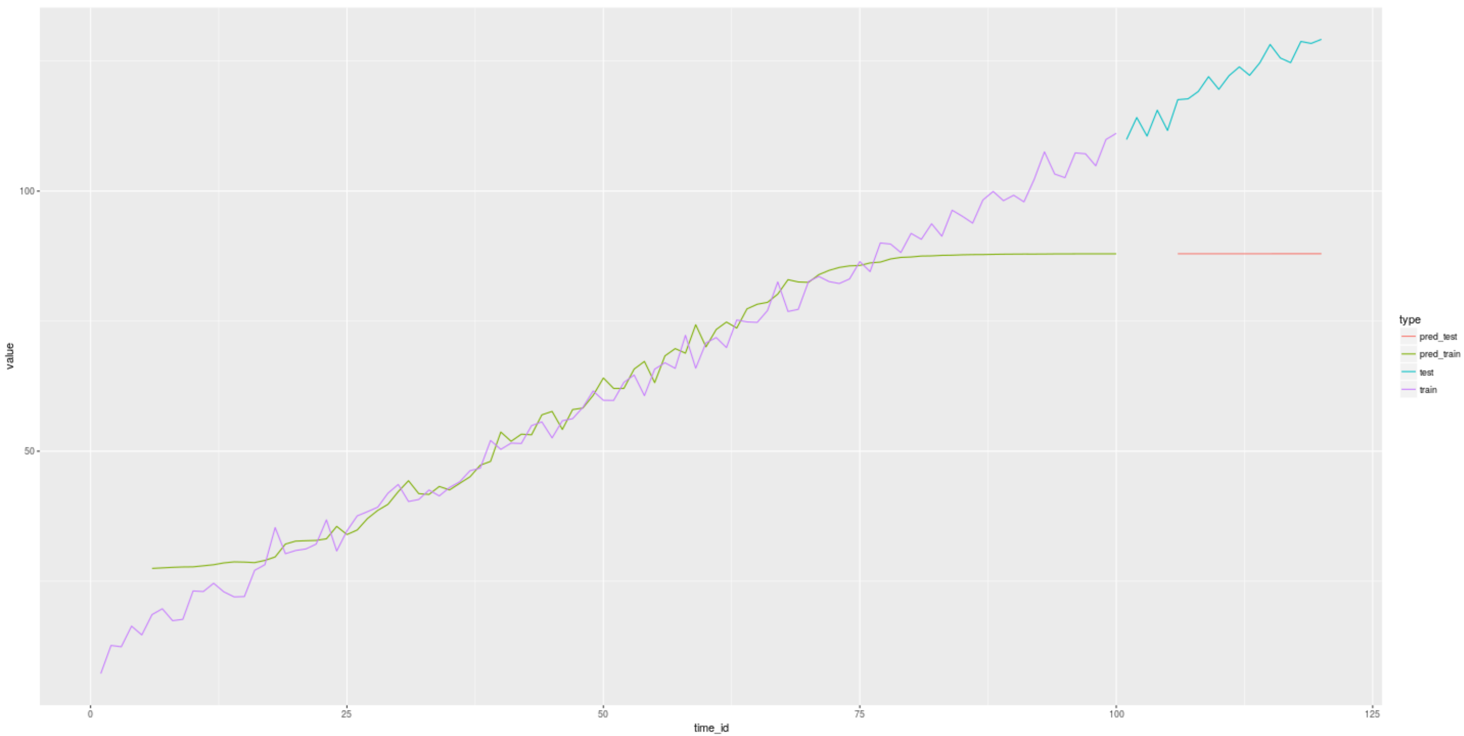
```
# not executed "live" ;-)
# keras_fit(model, X_train, y_train, batch_size = 1, epochs = 500, verbose = 1)
```

We'll load the fitted model instead, and get the predictions.

```
# we'll load the fitted model instead...
# keras_fit(model, X_train, y_train, batch_size = 1, epochs = 500, verbose = 1)

model <- keras_load('trend_nodiff.h5')
pred_train <- keras_predict(model, X_train, batch_size = 1)
pred_test <-keras_predict(model, X_test, batch_size = 1)
```

Hm. Whatever happened to predicting the test data?

```
df <- data_frame(time_id = 1:120,
  train = c(trend_train, rep(NA, length(trend_test))),
  test = c(rep(NA, length(trend_train)), trend_test),
  pred_train = c(rep(NA, lstm_num_timesteps), pred_train, rep(NA, length(trend_test))),
  pred_test = c(rep(NA, length(trend_train)), rep(NA, lstm_num_timesteps), pred_test))
df <- df %>% gather(key = 'type', value = 'value', train:pred_test)
ggplot(df, aes(x = time_id, y = value)) + geom_line(aes(color = type))
```



Anything we could have done to our data to make this work better?

- What if we had worked with the value differences, instead of the original values (learning from ARIMA & co.)?
- Or even the relative differences?
- What if we had scaled the data?

What if we work with differenced data?

```
set.seed(7777)
trend_train <- 11:110 + rnorm(100, sd = 2)
trend_test <- 111:130 + rnorm(20, sd =2)

trend_train_start <- trend_train[1]
trend_test_start <- trend_test[1]

trend_train_diff <- diff(trend_train)
trend_test_diff <- diff(trend_test)
trend_test_diff
```

```
 [1]  4.2235013 -3.5657054  4.9708362 -3.8893841  5.9185922  0.1646659
 [7]  1.4168194  2.8387000 -2.4343835  2.6326364  1.6875305 -1.6268218
[13]  2.4204479  3.5124616 -2.6026255 -0.8882357  4.0684488 -0.3788169
[19]  0.7841282
```

```
lstm_num_timesteps <- 4
```

Get the predictions

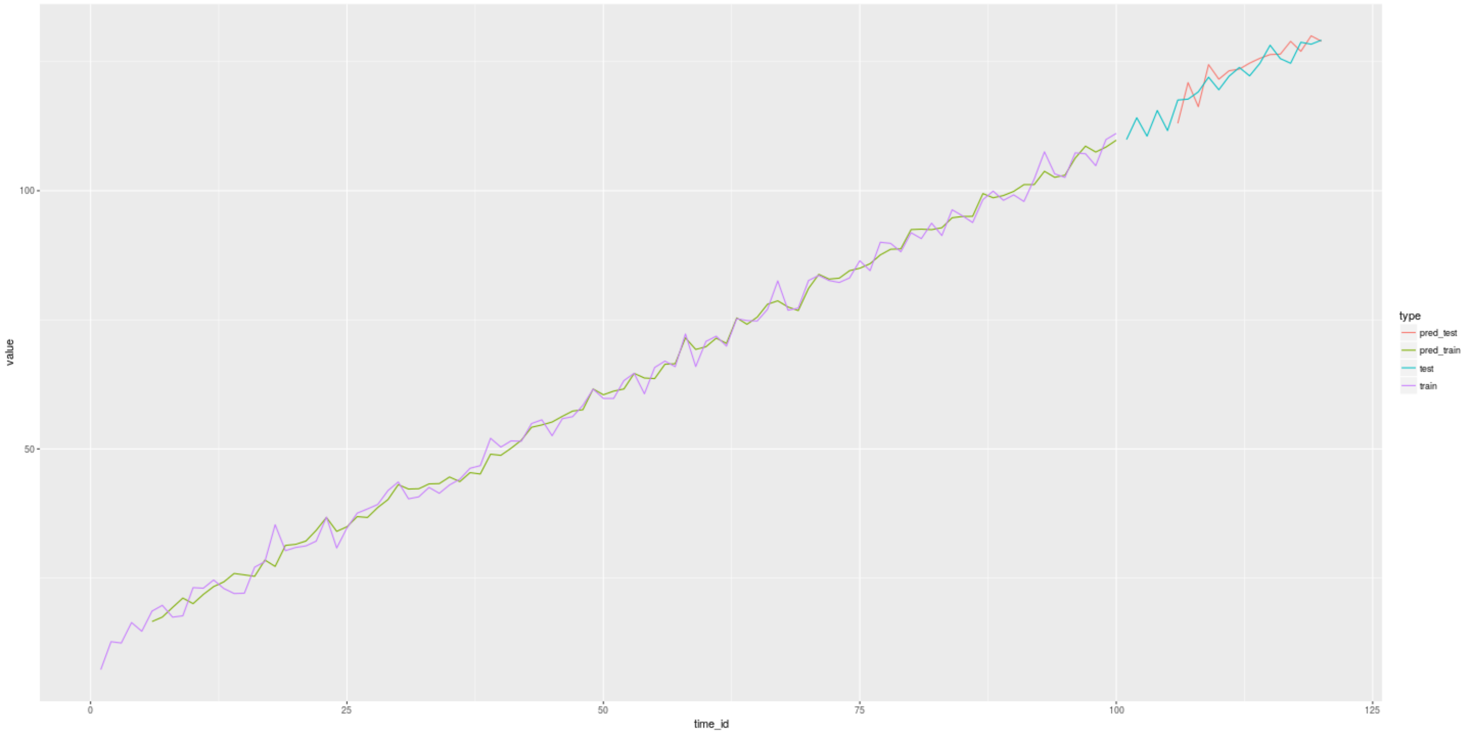
```
# we'll load the fitted model instead...
model <- keras_load('trend_diff.h5')
pred_train <- keras_predict(model, X_train, batch_size = 1)
pred_test <-keras_predict(model, X_test, batch_size = 1)

# "undiff"
pred_train_undiff <- pred_train + trend_train[(lstm_num_timesteps+1):(length(trend_train)-1)]
pred_test_undiff <- pred_test + trend_test[(lstm_num_timesteps+1):(length(trend_test)-1)]
```

Differencing makes the difference…

```
df <- data_frame(time_id = 1:120,
   train = c(trend_train, rep(NA, length(trend_test))),
   test = c(rep(NA, length(trend_train)), trend_test),
   pred_train = c(rep(NA, lstm_num_timesteps+1), pred_train_undiff, rep(NA, length(trend_test))),
   pred_test = c(rep(NA, length(trend_train)), rep(NA, lstm_num_timesteps+1), pred_test_undiff))
df <- df %>% gather(key = 'type', value = 'value', train:pred_test)
(test_rsme <- sqrt(sum((tail(trend_test,length(trend_test) - lstm_num_timesteps - 1) -
pred_test_undiff)^2)))
```

```
[1] 9.231277
```

```
ggplot(df, aes(x = time_id, y = value)) + geom_line(aes(color = type))
```



Just for completeness, let's try relative differences as well

```r
set.seed(7777)
trend_train <- 11:110 + rnorm(100, sd = 2)
trend_test <- 111:130 + rnorm(20, sd =2)

trend_train_start <- trend_train[1]
trend_test_start <- trend_test[1]

trend_train_diff <- diff(trend_train)/trend_train[-length(trend_train)]
trend_test_diff <- diff(trend_test)/trend_test[-length(trend_test)]
trend_test_diff
```

```
 [1]  0.038423558 -0.031238910  0.044953465 -0.033660270  0.053006039
 [6]  0.001400490  0.012033246  0.023822810 -0.019954355  0.022018781
[11]  0.013810047 -0.013131880  0.019798102  0.028172487 -0.020302958
[16] -0.007072681  0.032626255 -0.002941878  0.006107476
```

```r
lstm_num_timesteps <- 4
```

Get the predictions

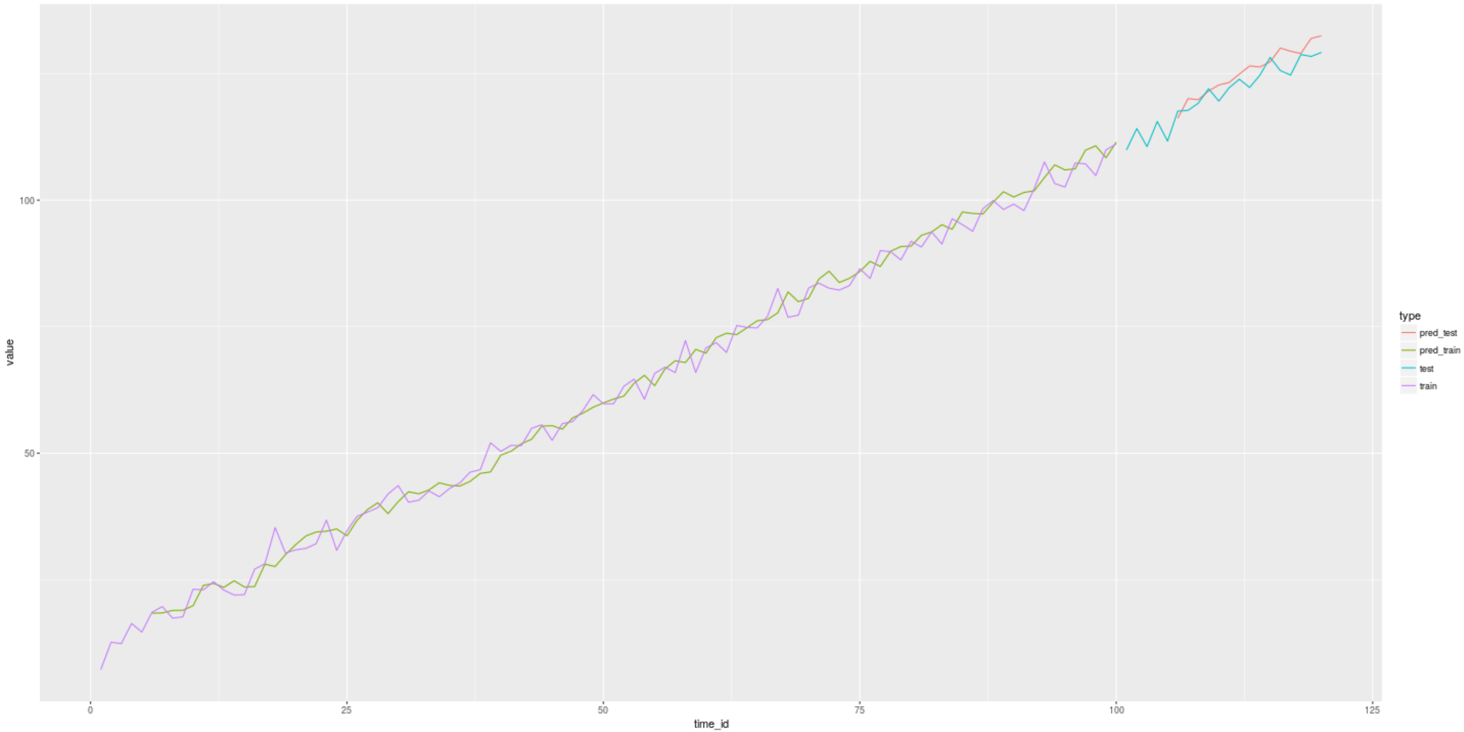
```
# we'll load the fitted model instead...
model <- keras_load('trend_reldiff.h5')

pred_train <- keras_predict(model, X_train, batch_size = 1)
pred_test <-keras_predict(model, X_test, batch_size = 1)
pred_train_undiff <- pred_train * trend_train[(lstm_num_timesteps+1):(length(trend_train)-1)] +
trend_train[(lstm_num_timesteps+1):(length(trend_train)-1)]
pred_test_undiff <- pred_test * trend_test[(lstm_num_timesteps+1):(length(trend_test)-1)] +
trend_test[(lstm_num_timesteps+1):(length(trend_test)-1)]
```

Relative differences: results

```
df <- data_frame(time_id = 1:120,
    train = c(trend_train, rep(NA, length(trend_test))),
    test = c(rep(NA, length(trend_train)), trend_test),
    pred_train = c(rep(NA, lstm_num_timesteps+1), pred_train_undiff, rep(NA, length(trend_test))),
    pred_test = c(rep(NA, length(trend_train)), rep(NA, lstm_num_timesteps+1), pred_test_undiff))
df <- df %>% gather(key = 'type', value = 'value', train:pred_test)
(test_rsme <- sqrt(sum((tail(trend_test,length(trend_test) - lstm_num_timesteps - 1) -
pred_test_undiff)^2)))
```

```
[1] 10.35424
```

```
ggplot(df, aes(x = time_id, y = value)) + geom_line(aes(color = type))
```



What if we difference AND normalize?

```
set.seed(7777)
trend_train <- 11:110 + rnorm(100, sd = 2)
trend_test <- 111:130 + rnorm(20, sd =2)

trend_train_start <- trend_train[1]
trend_test_start <- trend_test[1]

trend_train_diff <- diff(trend_train)
trend_test_diff <- diff(trend_test)

minval <- min(trend_train_diff)
maxval <- max(trend_train_diff)

normalize <- function(vec, min, max) {
  (vec-min) / (max-min)
}
denormalize <- function(vec,min,max) {
  vec * (max - min) + min
}

trend_train_diff <- normalize(trend_train_diff, minval, maxval)
trend_test_diff <- normalize(trend_test_diff, minval, maxval)

lstm_num_timesteps <- 4
```

Get the predictions

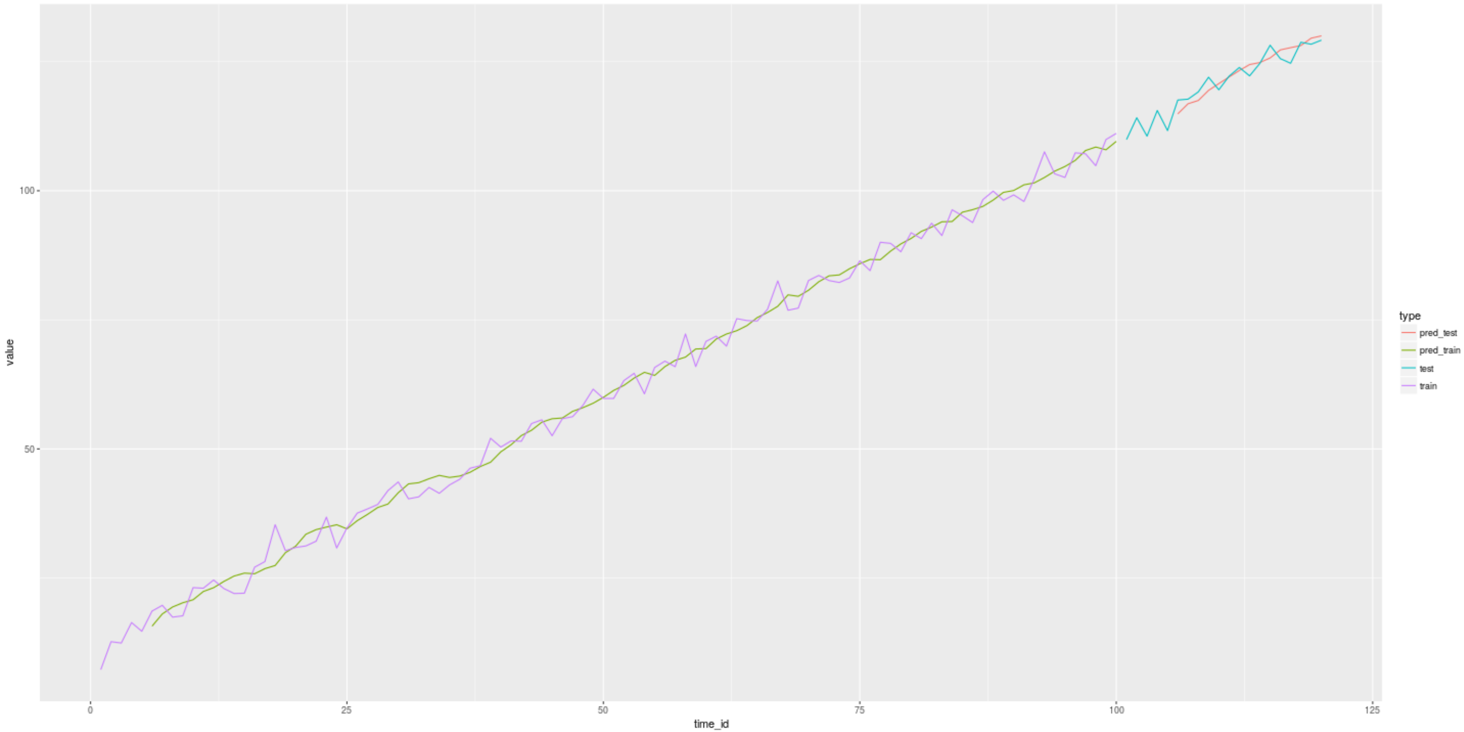
```
# we'll load the fitted model instead...
model <- keras_load('trend_diffnorm.h5')
pred_train <- keras_predict(model, X_train, batch_size = 1)
pred_test <-keras_predict(model, X_test, batch_size = 1)

pred_train <- denormalize(pred_train, minval, maxval)
pred_test <- denormalize(pred_test, minval, maxval)

pred_train_undiff <- pred_train + trend_train[(lstm_num_timesteps+1):(length(trend_train)-1)]
pred_test_undiff <- pred_test + trend_test[(lstm_num_timesteps+1):(length(trend_test)-1)]
```

Difference and normalize: results

```
df <- data_frame(time_id = 1:120,
    train = c(trend_train, rep(NA, length(trend_test))),
    test = c(rep(NA, length(trend_train)), trend_test),
    pred_train = c(rep(NA, lstm_num_timesteps+1), pred_train_undiff, rep(NA, length(trend_test))),
    pred_test = c(rep(NA, length(trend_train)), rep(NA, lstm_num_timesteps+1), pred_test_undiff))
df <- df %>% gather(key = 'type', value = 'value', train:pred_test)
(test_rsme <- sqrt(sum((tail(trend_test,length(trend_test) - lstm_num_timesteps - 1) -
pred_test_undiff)^2)))
```

```
[1] 6.662725
```

```
ggplot(df, aes(x = time_id, y = value)) + geom_line(aes(color = type))
```



ARIMA vs. LSTM, Round 2: Trend-only dataset, test data in-range

Would anything change if the test data were in the range already known by the model?

```
set.seed(7777)
trend_train <- 11:110 + rnorm(100, sd = 2)
trend_test <- 31:50 + rnorm(20, sd =2)
df <- data_frame(time_id = 1:120,
train = c(trend_train, rep(NA, length(trend_test))),
test = c(rep(NA, length(trend_train)), trend_test))
df <- df %>% gather(key = 'train_test', value = 'value', -time_id)
ggplot(df, aes(x = time_id, y = value, color = train_test)) + geom_line()
```

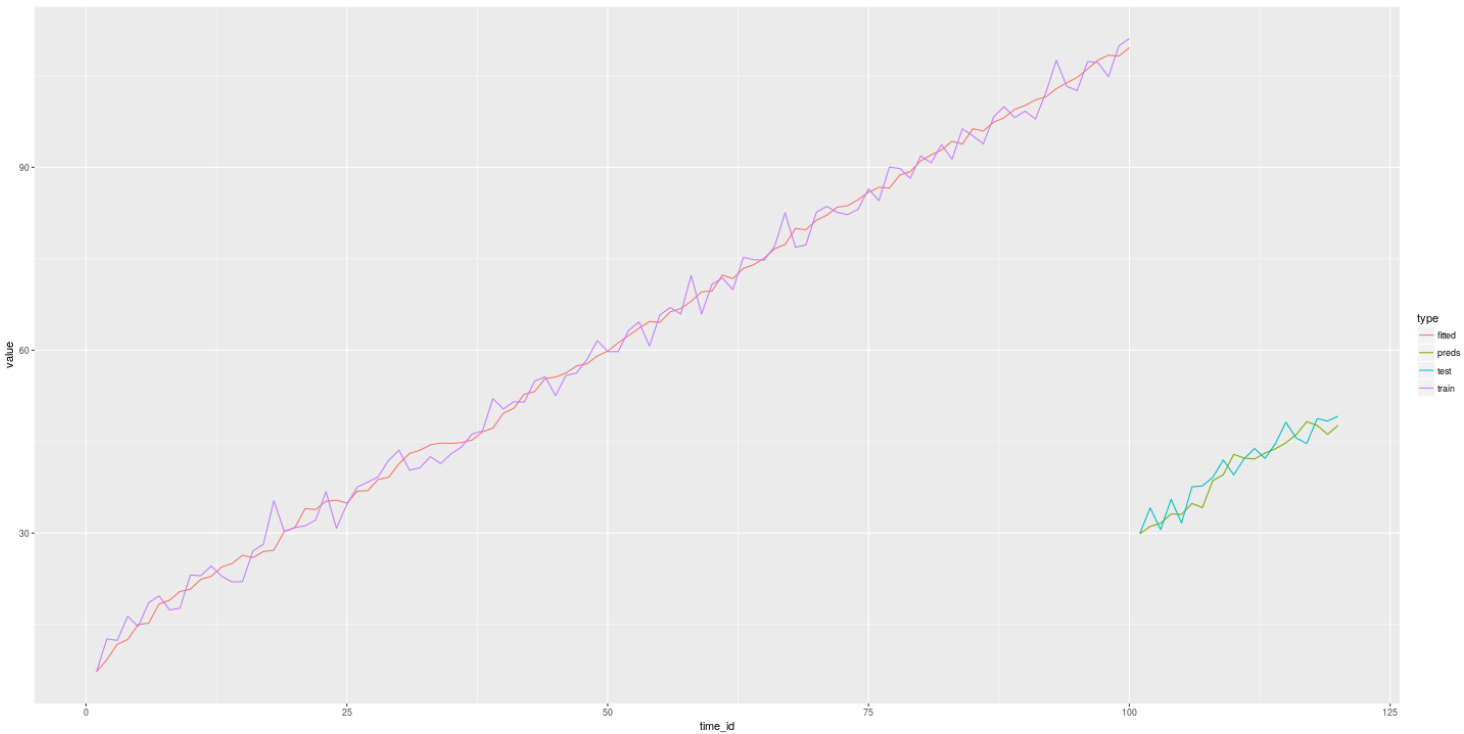
Trend-only dataset, test data in-range: Enter: ARIMA (1)

```
set.seed(7777)
trend_train <- 11:110 + rnorm(100, sd = 2)
trend_test <- 31:50 + rnorm(20, sd =2)
fit <- auto.arima(trend_train)
order <- arimaorder(fit)
# fit on the test set
refit <- Arima(trend_test, order=order[1:3], seasonal=order[4:6])
predictions <- refit$fitted
(test_rsme <- sqrt(sum((trend_test - predictions)^2)))
```

```
[1] 9.629612
```

Trend-only dataset, test data in-range: Enter: ARIMA (2)

```
df <- data_frame(time_id = 1:120,
  train = c(trend_train, rep(NA, length(trend_test))),
  test = c(rep(NA, length(trend_train)), trend_test),
  fitted = c(fit$fitted, rep(NA, length(trend_test))),
  preds = c(rep(NA, length(trend_train)), predictions))
df <- df %>% gather(key = 'type', value = 'value', train:preds)
ggplot(df, aes(x = time_id, y = value)) + geom_line(aes(color = type))
```

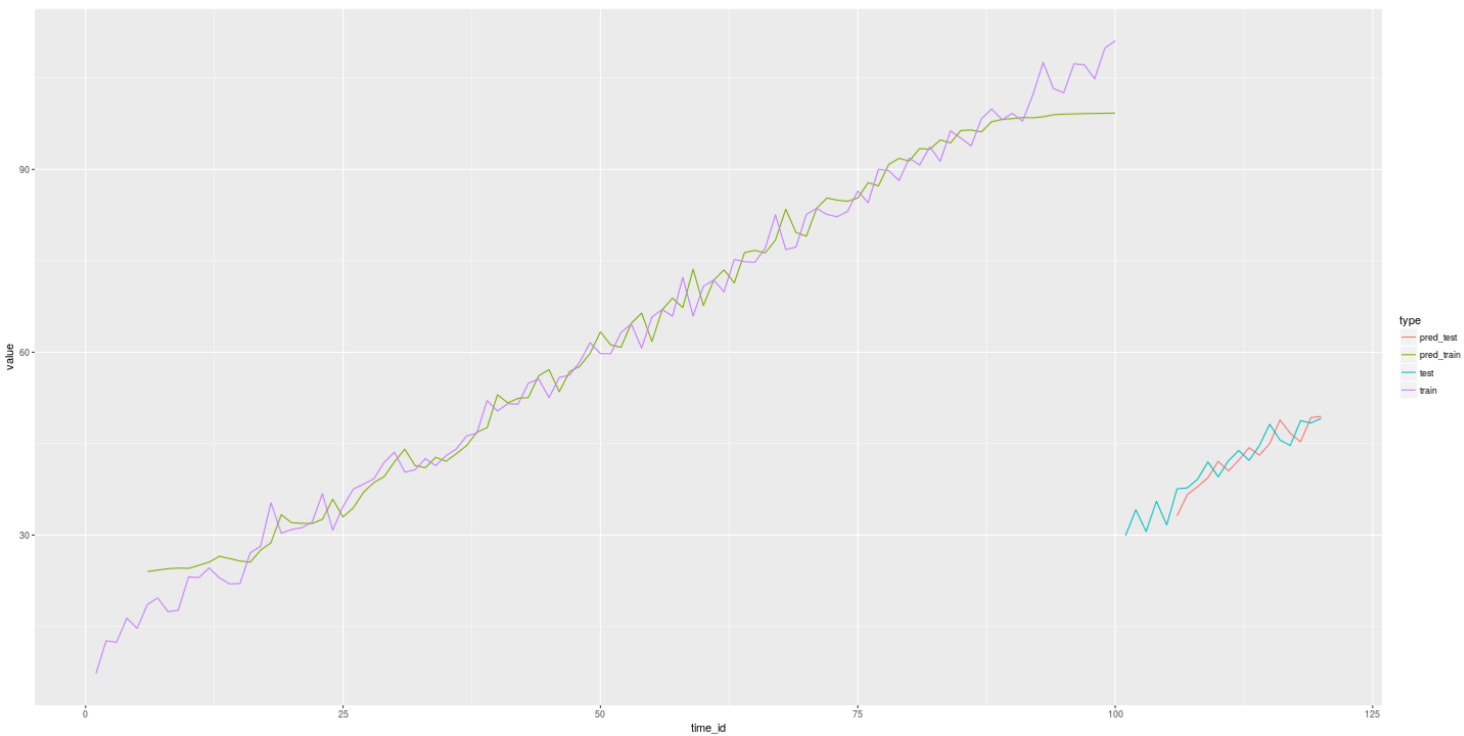


Trend-only dataset, test data in-range: Enter: LSTM (no differencing)

```
(test_rsme <- sqrt(sum((tail(trend_test,length(trend_test) - lstm_num_timesteps) - pred_test)^2)))
```

```
[1] 9.254975
```

```
ggplot(df, aes(x = time_id, y = value)) + geom_line(aes(color = type))
```



Does it get even better with differencing?

```
(test_rsme <- sqrt(sum((tail(trend_test,length(trend_test) - lstm_num_timesteps - 1) -
pred_test_undiff)^2)))
```

```
[1] 6.394894
```

```
ggplot(df, aes(x = time_id, y = value)) + geom_line(aes(color = type))
```

And with relative differencing?

```
(test_rsme <- sqrt(sum((tail(trend_test,length(trend_test) - lstm_num_timesteps - 1) -
pred_test_undiff)^2)))
```

```
[1] 7.491432
```

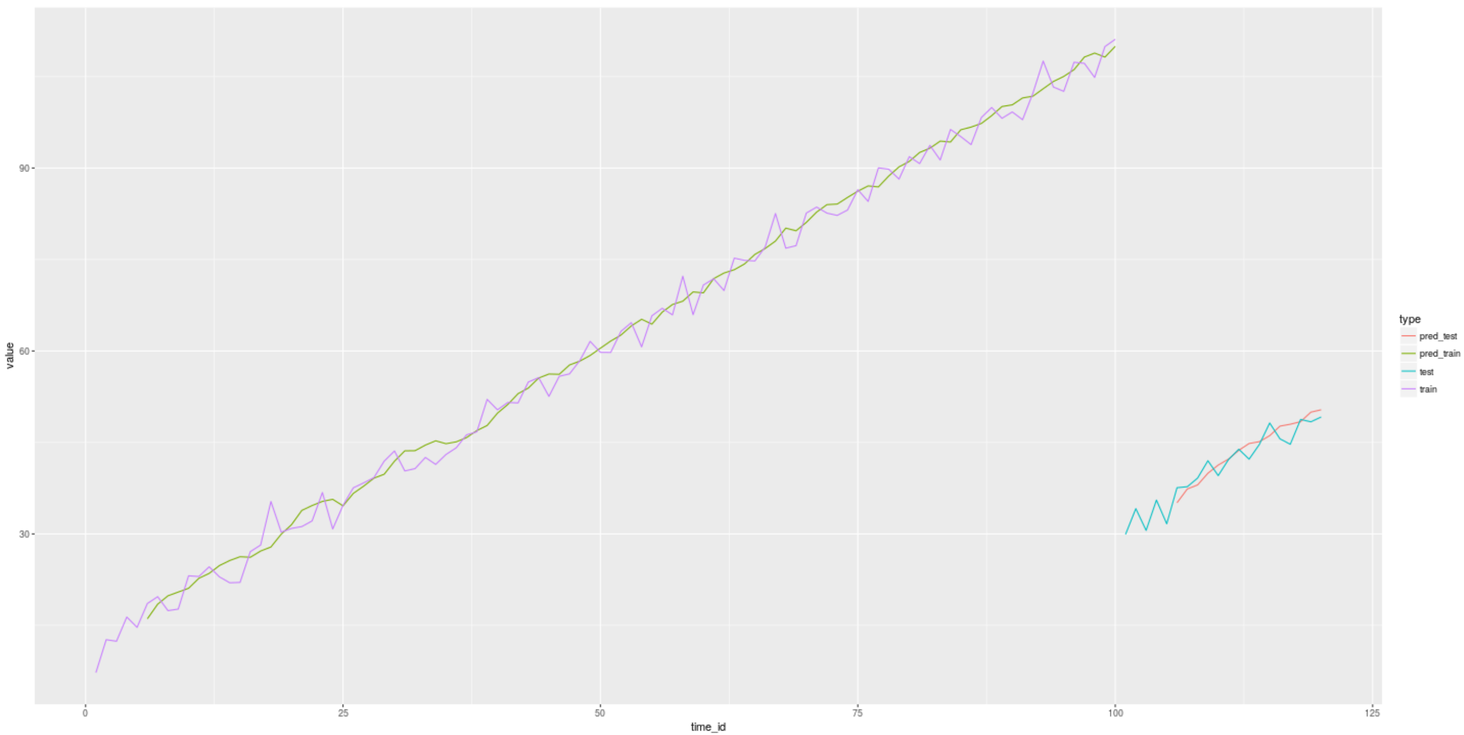
```
ggplot(df, aes(x = time_id, y = value)) + geom_line(aes(color = type))
```

Finally, what about both differencing and normalizing?

```
(test_rsme <- sqrt(sum((tail(trend_test,length(trend_test) - lstm_num_timesteps - 1) -
pred_test_undiff)^2)))
```

```
[1] 6.710867
```

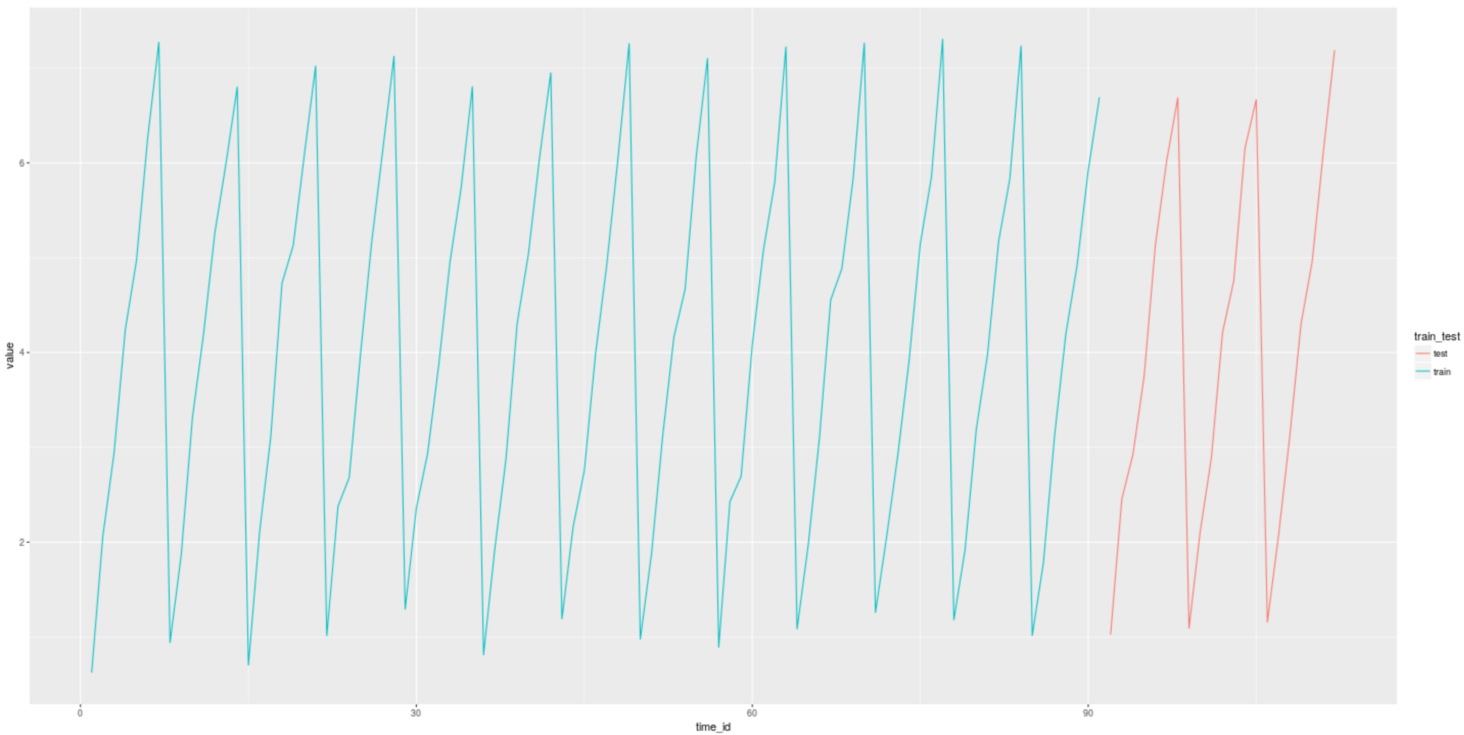
```
ggplot(df, aes(x = time_id, y = value)) + geom_line(aes(color = type))
```



ARIMA vs. LSTM, Round 3: seasonal-only dataset

Seasonal-only dataset

```
set.seed(7777)
seasonal_train <- rep(1:7, times = 13) + rnorm(91, sd=0.2)
seasonal_test <- rep(1:7, times = 3) + rnorm(21, sd=0.2)
df <- data_frame(time_id = 1:112,
train = c(seasonal_train, rep(NA, length(seasonal_test))),
test = c(rep(NA, length(seasonal_train)), seasonal_test))
df <- df %>% gather(key = 'train_test', value = 'value', -time_id)
ggplot(df, aes(x = time_id, y = value, color = train_test)) + geom_line()
```



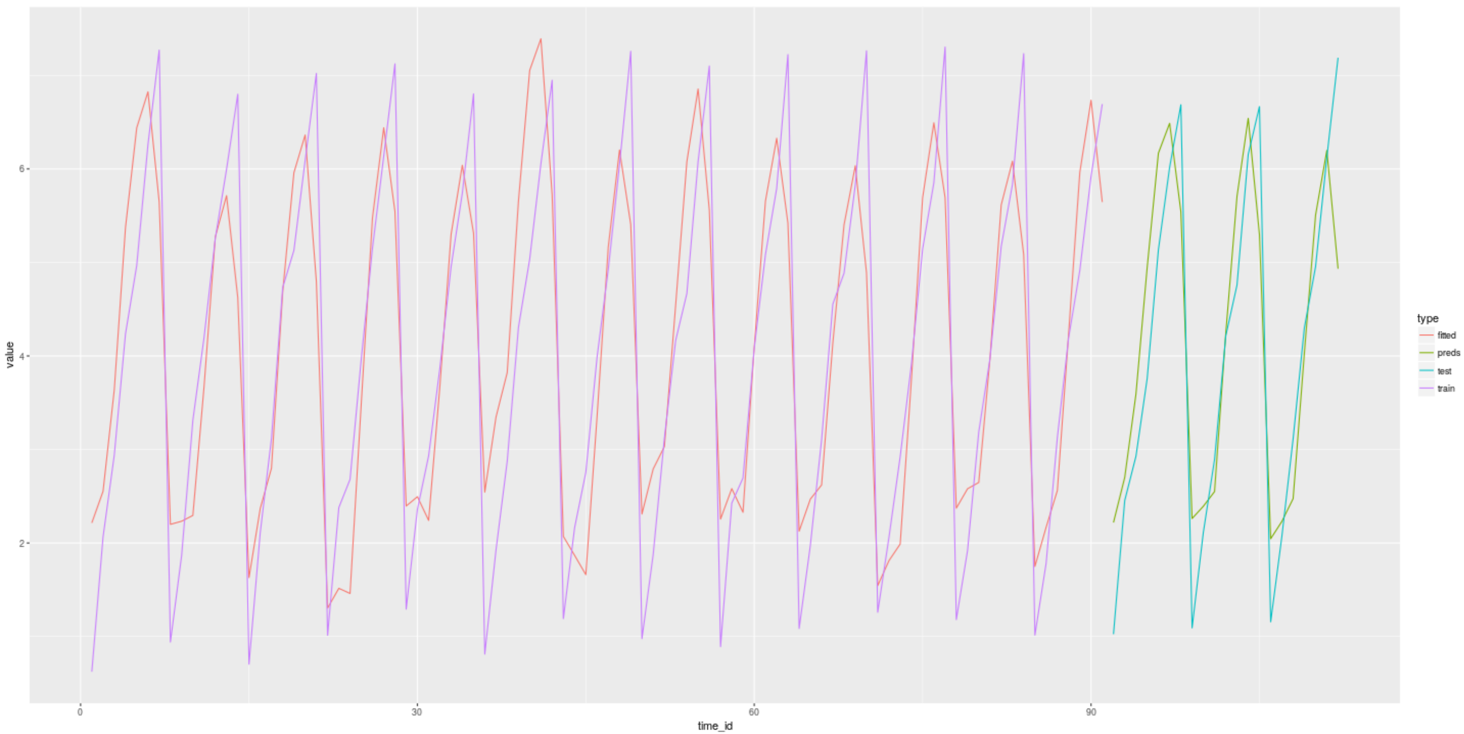
Seasonal-only dataset: Enter: ARIMA (1)

```
set.seed(7777)
seasonal_train <- rep(1:7, times = 13) + rnorm(91, sd=0.2)
seasonal_test <- rep(1:7, times = 3) + rnorm(21, sd=0.2)
h <- 1
n <- length(seasonal_test) - h + 1
fit <- auto.arima(seasonal_train)
order <- arimaorder(fit)
refit <- Arima(seasonal_test, order=order[1:3], seasonal=order[4:6])
predictions <- refit$fitted
(test_rsme <- sqrt(sum((seasonal_test - predictions)^2)))
```

```
[1] 4.136755
```

Seasonal-only dataset: enter: ARIMA (2)

```
df <- data_frame(time_id = 1:112,
  train = c(seasonal_train, rep(NA, length(seasonal_test))),
  test = c(rep(NA, length(seasonal_train)), seasonal_test),
  fitted = c(fit$fitted, rep(NA, length(seasonal_test))),
  preds = c(rep(NA, length(seasonal_train)), predictions))
df <- df %>% gather(key = 'type', value = 'value', train:preds)
ggplot(df, aes(x = time_id, y = value)) + geom_line(aes(color = type))
```

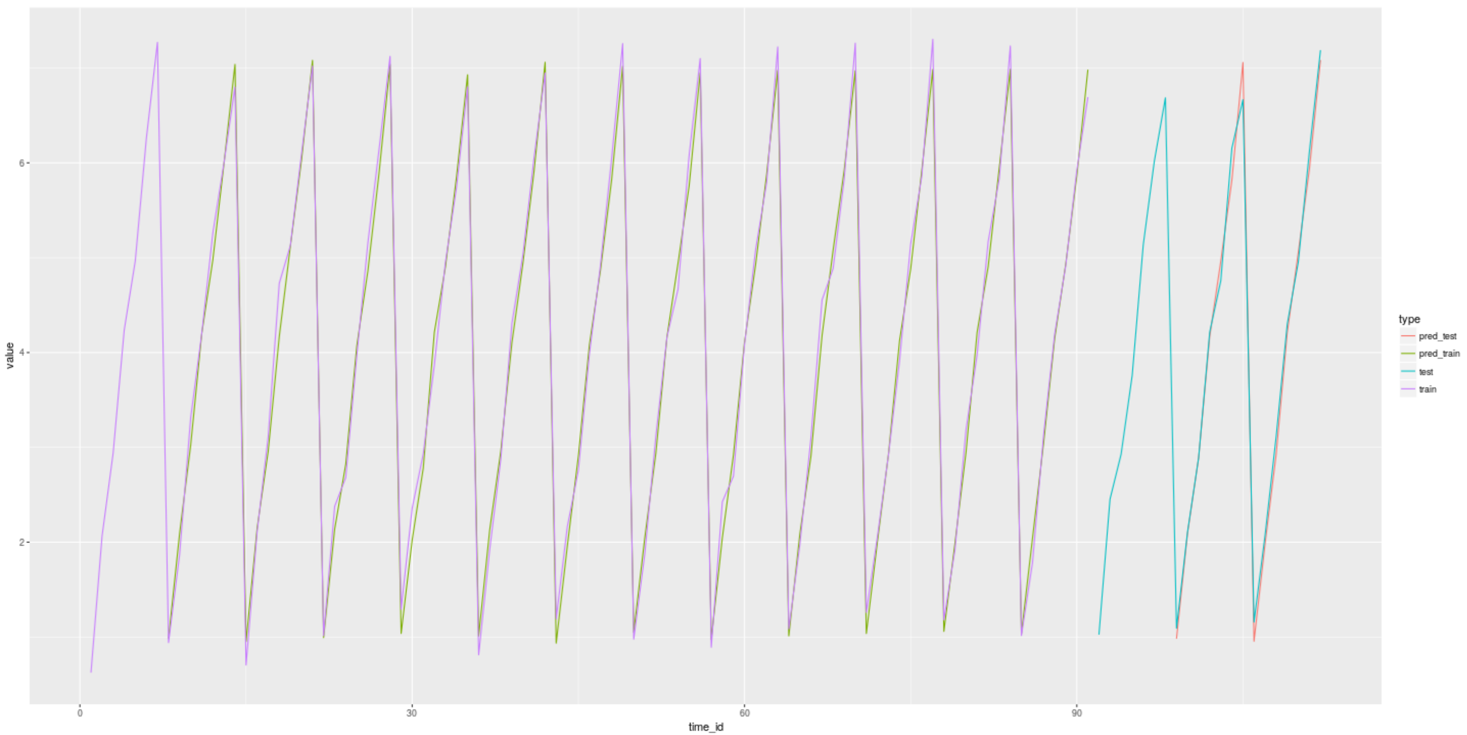


Seasonal-only dataset: Enter: LSTM (no differencing)

```
(test_rsme <- sqrt(sum((tail(seasonal_test,length(seasonal_test) - lstm_num_timesteps) -
pred_test)^2)))
```

```
[1] 0.6890056
```

```
ggplot(df, aes(x = time_id, y = value)) + geom_line(aes(color = type))
```

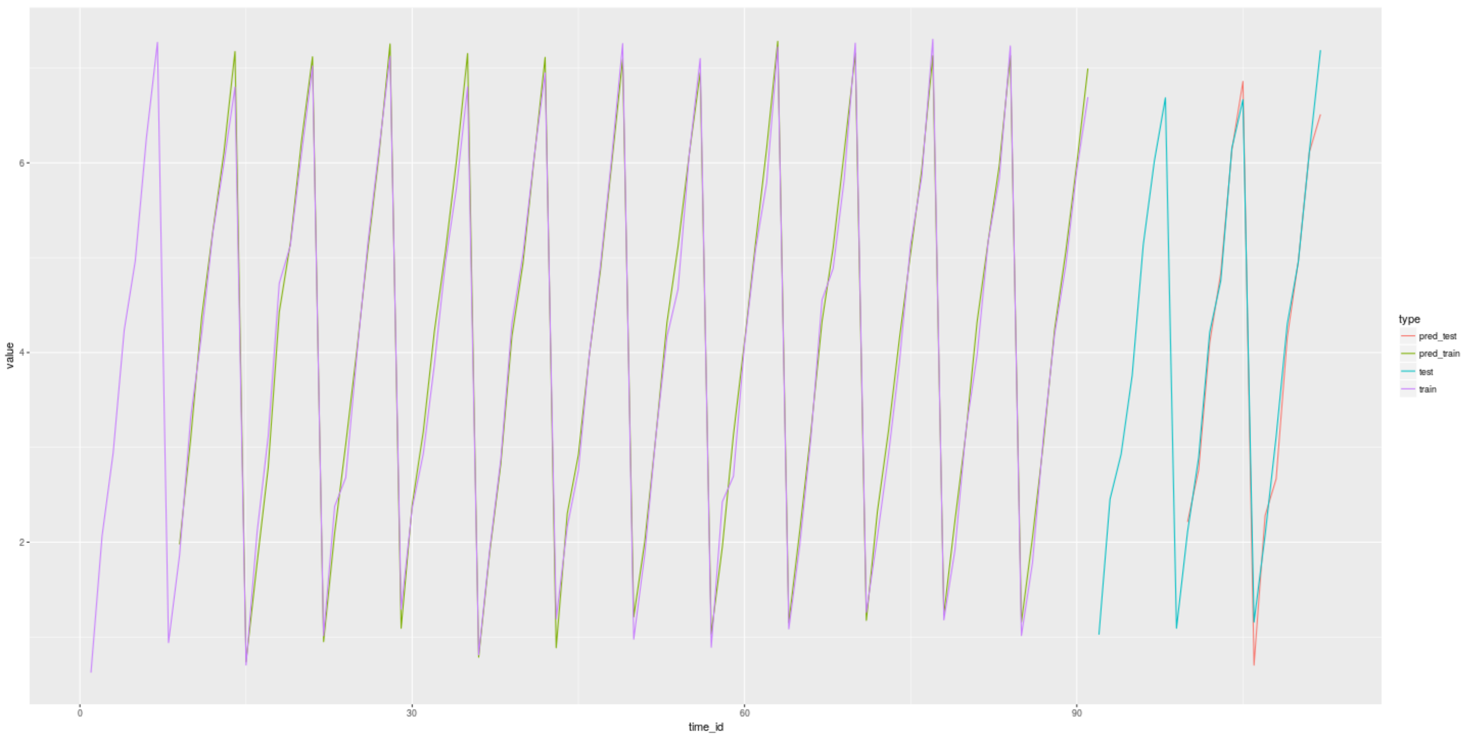


Does it get even better with differencing? (How could it ;-))

```
(test_rsme <- sqrt(sum((tail(seasonal_test,length(seasonal_test) - lstm_num_timesteps - 1) -
pred_test_undiff)^2)))
```

```
[1] 1.005745
```

```
ggplot(df, aes(x = time_id, y = value)) + geom_line(aes(color = type))
```

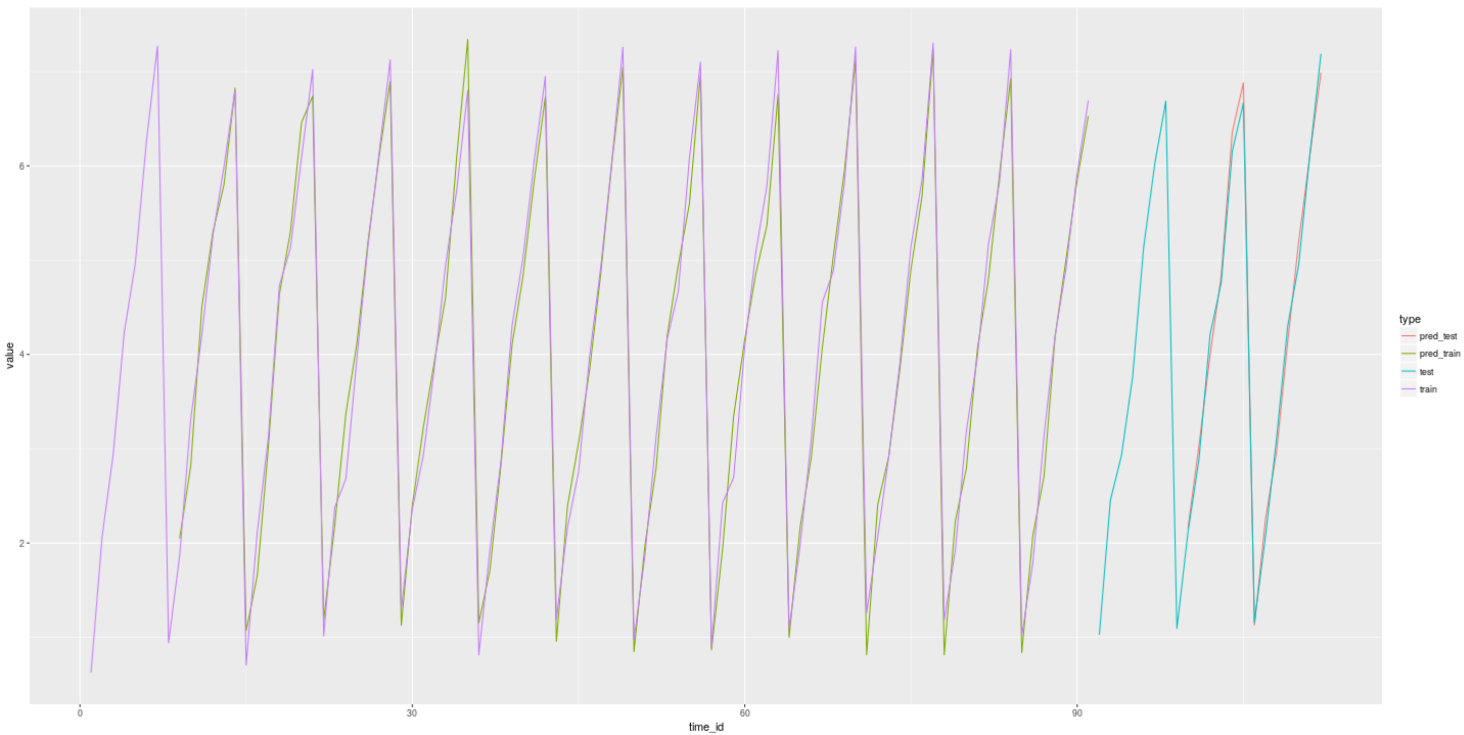


Or with relative differencing (for completeness' sake)?

```
(test_rsme <- sqrt(sum((tail(seasonal_test,length(seasonal_test) - lstm_num_timesteps - 1) -
pred_test_undiff)^2)))
```

```
[1] 0.5890867
```

```
ggplot(df, aes(x = time_id, y = value)) + geom_line(aes(color = type))
```

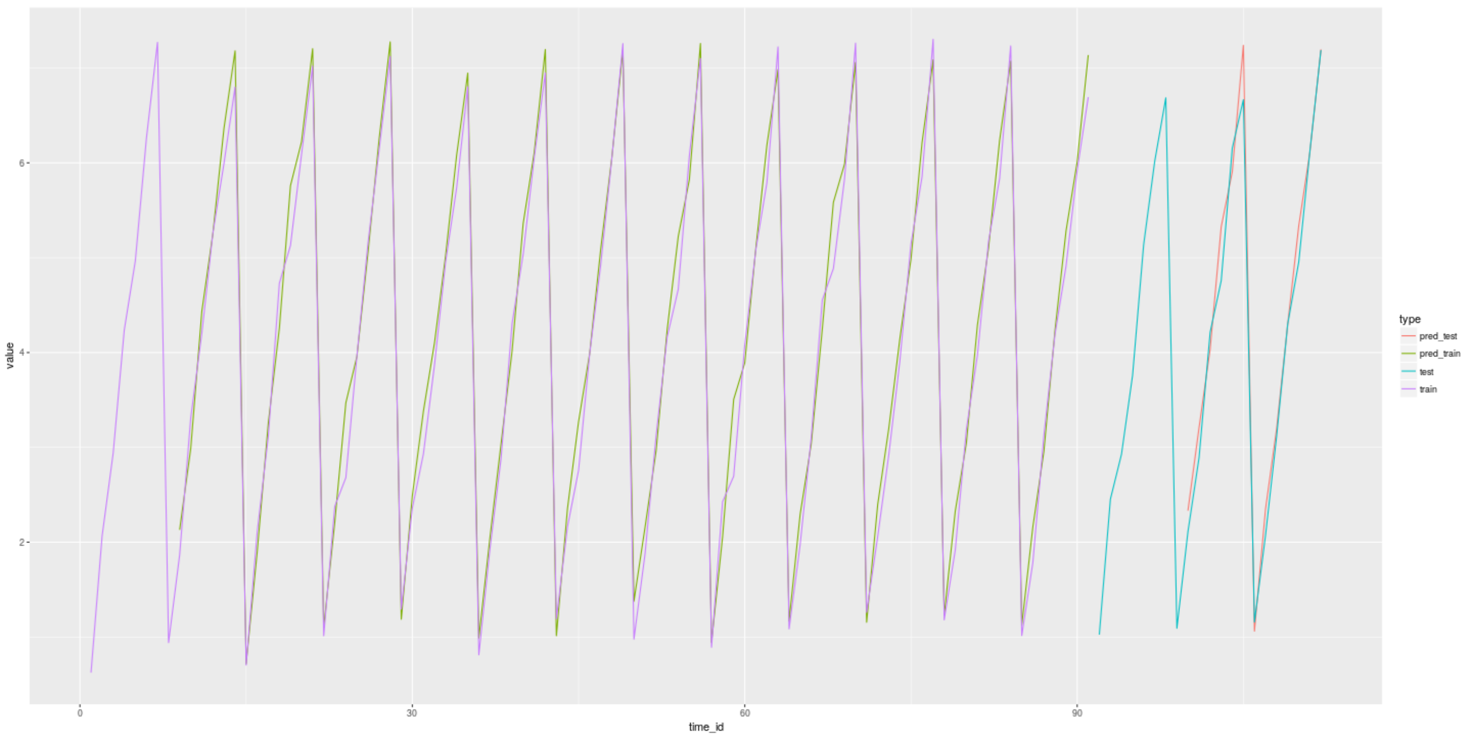


Finally, what about both differencing and normalizing

```
(test_rsme <- sqrt(sum((tail(seasonal_test,length(seasonal_test) - lstm_num_timesteps - 1) -
pred_test_undiff)^2)))
```

```
[1] 1.064543
```

```
ggplot(df, aes(x = time_id, y = value)) + geom_line(aes(color = type))
```



Benchmark: Remarks and conclusions

- We've used a very basic setup for the LSTM
  - just one layer
  - no experiments with number of units, optimization routines, activation functions…
  - no use of dropout, regularization, weight decay…
  - not making use of Keras *stateful* LSTM
- We've only compared both methods on a rolling forecast (not forecasting several periods into the future)

Aside (1): Stateful RNNs in Keras

- With stateful RNNs, states computed for the samples in one batch will be reused as initial states for the *respective* samples in the next batch
- "Makes sense" for a time series, as long as the data is reformatted or batch_size=1 is used
- Presupposes batches arriving in the same order in every epoch (set shuffle = False)
- Not currently implemented in KerasR - time for some Python…

Demo: Stateful LSTM in Keras

Aside (2): Multi-step-ahead forecasts in Keras

- Multi-step-ahead forecasts using LSTM can be done in a number of ways:
  - build predictions on earlier predictions ("low end")
  - seq2seq architecture ("high end", not currently available out of the box in Keras)
  - using TimeDistributed layer (not currently implemented in KerasR)

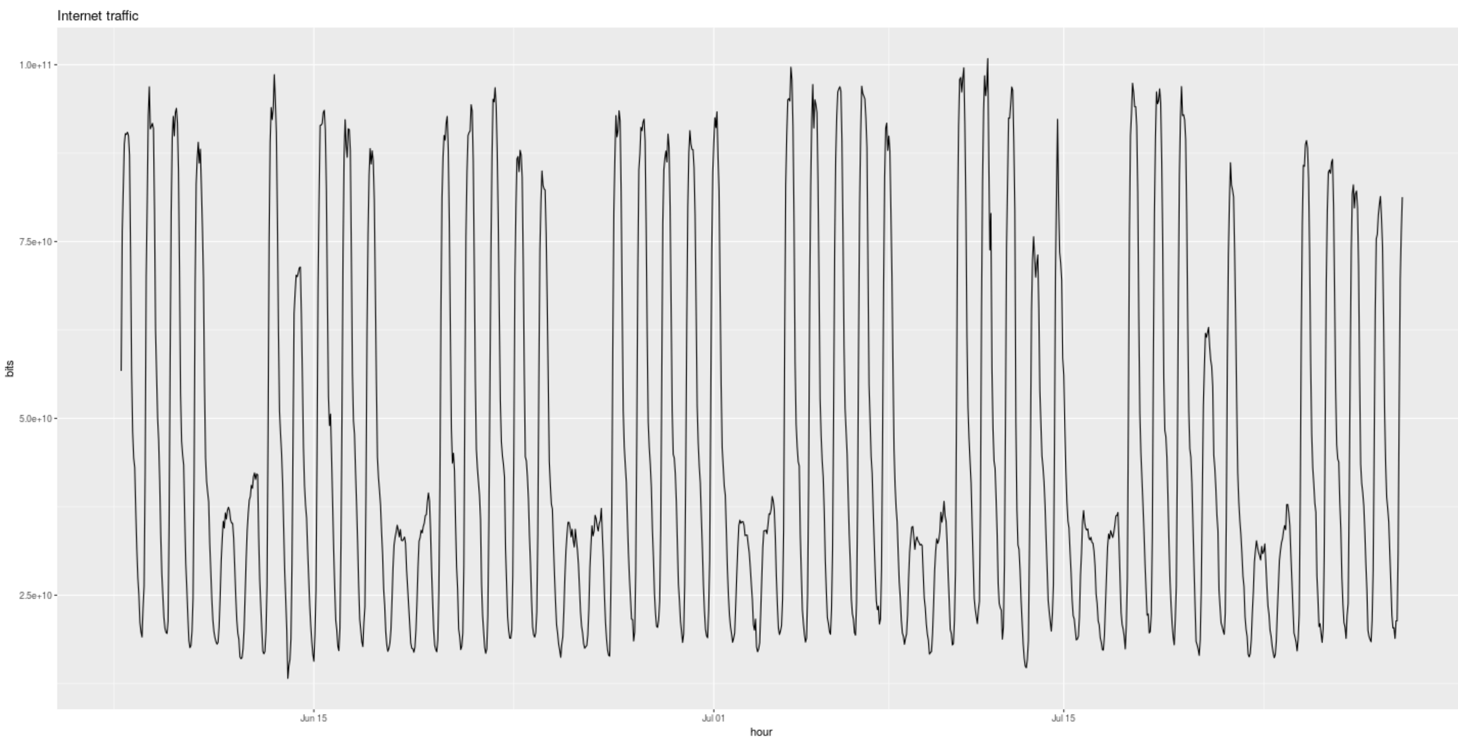
Demo: Multi-step ahead forecasting in Keras using TimeDistributed

Finally…

Forecasting internet traffic using LSTM

Here, again, is our time series...

```
ggplot(traffic_df, aes(x = hour, y = bits)) + geom_line() + ggtitle("Internet traffic")
```



Forecasting internet traffic with LSTM

We apply first-order differencing and normalize the data.

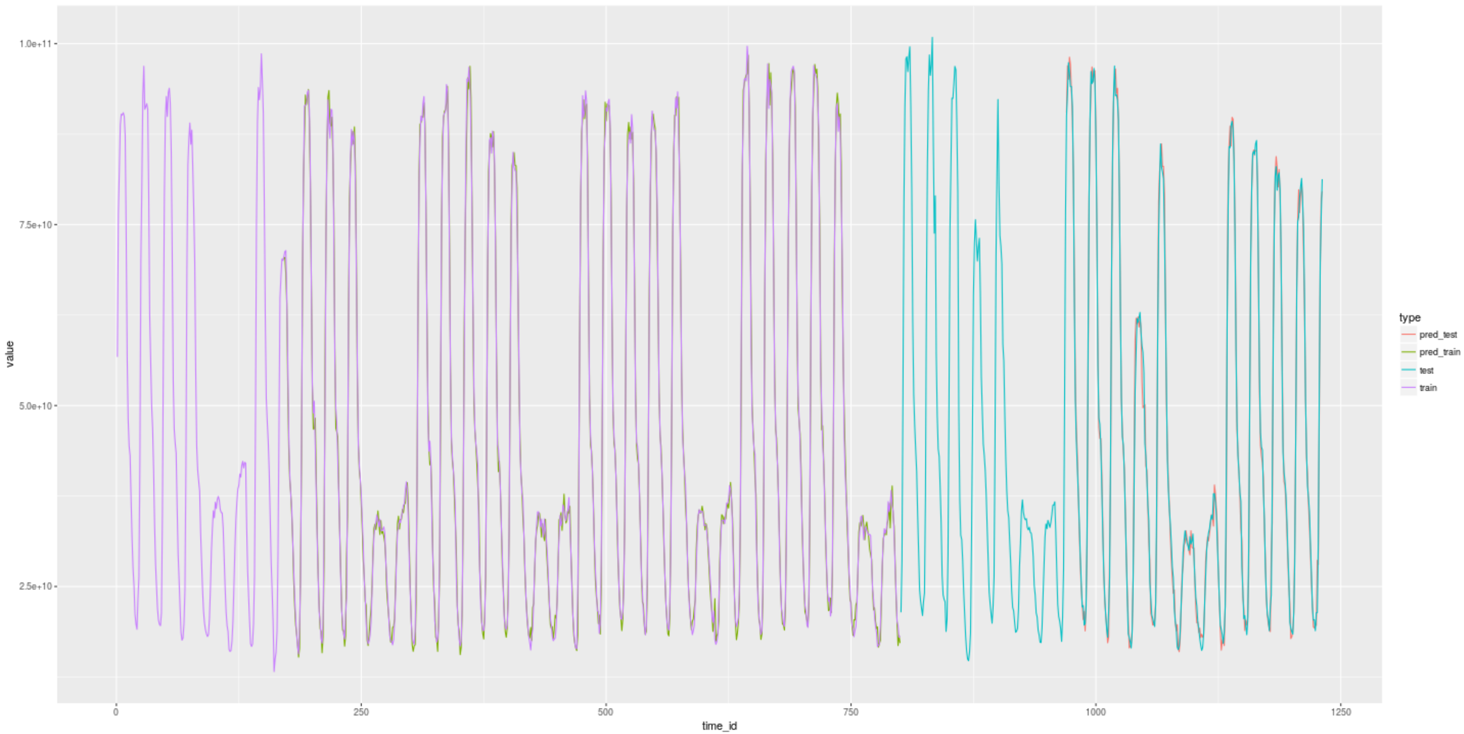
```
traffic_train <- traffic_df$bits[1:800]
traffic_test <- traffic_df$bits[801:nrow(traffic_df)]

traffic_train_start <- traffic_train[1]
traffic_test_start <- traffic_test[1]

traffic_train_diff <- diff(traffic_train)
traffic_test_diff <- diff(traffic_test)

minval <- min(traffic_train_diff)
maxval <- max(traffic_train_diff)

normalize <- function(vec, min, max) {
  (vec-min) / (max-min)
}
denormalize <- function(vec,min,max) {
  vec * (max - min) + min
}

traffic_train_diff <- normalize(traffic_train_diff, minval, maxval)
traffic_test_diff <- normalize(traffic_test_diff, minval, maxval)
```

We choose 7*24=168 for the number of timesteps.

```
lstm_num_timesteps <- 7*24
```

And the results? … Wow!

```
ggplot(df, aes(x = time_id, y = value)) + geom_line(aes(color = type))
```



The end?

- This is more like a beginning
- Lots of things to explore…
  - experiment with parameters, architectures…
  - experiment with different datasets…
- Have fun!

Thanks for your attention!!