One of the ways **Deep Learning can be used in business** is to improve the accuracy of time series forecasts (prediction). We recently showed how a [Long Short Term Memory (LSTM) Models developed with the Keras library in R](http://www.business-science.io/timeseries-analysis/2018/04/18/keras-lstm-sunspots-time-series-prediction.html) could be used to take advantage of *autocorrelation* to predict the next 10 years of monthly Sunspots (a solar phenomenon that’s tracked by NASA). In this article, we teamed up with RStudio to take another look at the Sunspots data set, this time implementing some really advanced Deep Learning functionality available with [TensorFlow for R](https://tensorflow.rstudio.com/). Sigrid Keydana, TF Developer Advocate at RStudio, put together an amazing Deep Learning tutorial using keras for [implementing Keras in R](https://tensorflow.rstudio.com/keras/) and tfruns, a [suite of tools](https://tensorflow.rstudio.com/blog/tfruns.html) for trackingtracking, visualizing, and managing TensorFlow training runs and experiments from R. Sounds amazing, right? It is! **Let’s get started with this Deep Learning Tutorial!**

**Related Articles In This Series**

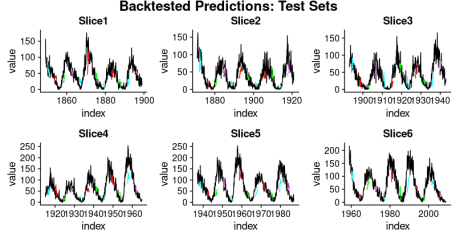
* [Time Series Deep Learning: Forecasting Sunspots With Keras Stateful LSTM In R](http://www.business-science.io/timeseries-analysis/2018/04/18/keras-lstm-sunspots-time-series-prediction.html)
* You can also find this article on [RStudio’s TensorFlow Blog](https://tensorflow.rstudio.com/blog/sunspots-lstm.html).

**Learning Trajectory**

In this **DEEP LEARNING TUTORIAL**\_, you will learn:

* [How Time Series Deep Learning can be used in business](http://www.business-science.io/timeseries-analysis/2018/07/01/keras-lstm-sunspots-part2.html#dl-in-business)
* [Forecasting Sunspots With Deep Learning](http://www.business-science.io/timeseries-analysis/2018/07/01/keras-lstm-sunspots-part2.html#dl-sunspots)
* [How To Build The LSTM](http://www.business-science.io/timeseries-analysis/2018/07/01/keras-lstm-sunspots-part2.html#lstm)
* [Backtesting The LSTM Model](http://www.business-science.io/timeseries-analysis/2018/07/01/keras-lstm-sunspots-part2.html#backtesting)

In fact, one of the coolest things you’ll develop is this plot of backtested LSTM forecasts.



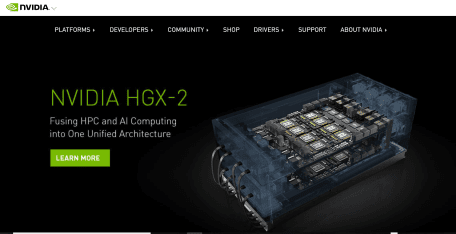
Backtested LSTM Forecasts

**Time Series Deep Learning In Business**

Introduction by Matt Dancho, Founder of Business Science

Time Series Forecasting is a key area that can lead to Return On Investment (ROI) in a business. Think about this: **A 10% improvement in forecast accuracy can save an organization millions of dollars**. How is this possible? Let’s find out.

We’ll take [NVIDIA](http://www.nvidia.com/page/home.html), a semiconductor manufacturer that manufactures state-of-the-art chips for *Artificial Intelligence (AI)* and *Deep Learning (DL)*, as an example. *NVIDIA* builds [Graphics Processing Unitis or GPUs](https://en.wikipedia.org/wiki/Graphics_processing_unit), which are necessary for the computational intensitity resulting from the massive number of numerical calculations required in high-performance Deep Learning. The chips look like this.



Source: [NVIDIA USA](http://www.nvidia.com/page/home.html)

Like all manufacturers, *NVIDIA* needs to ***forecast demand*** for their products. Why? So they can build the right amount of chips to supply their customers. This forecast is critical and it takes a lot of skill and some luck to get this right.

What we are talking about is the ***Sales Forecast***, which drives every manufacturing decision that NVIDIA makes. This includes how much raw material to purchase, how many people to have on staff to build the chips, and how much machining and assembly operations to budget. The more error in the sales forecast, the more unnecessary cost NVIDIA experiences because all of these activities (supply chain, inventory management, financial planning, etc) get thrown off!

**Time Series Deep Learning For Business**

***Time Series Deep Learning*** is amazingly accurate on data that has a high presence of autocorrelation because algorithms including [LSTMs](https://en.wikipedia.org/wiki/Long_short-term_memory) and [GRUs](https://en.wikipedia.org/wiki/Gated_recurrent_unit) can learn information from sequences regardless of when the pattern occurred. These special [RNNs](https://en.wikipedia.org/wiki/Recurrent_neural_network) are designed to have a long memory meaning that they excel at learning patterns from both recent observations and observeration that occurred long ago. This makes them perfect for time series! But are they great for sales data? Maybe. Let’s discuss.

Sales data comes in all flavors, but often it has ***seasonal patterns*** and ***trend***. The ***trend*** can be flat, linear, exponential, etc. This is typically not where the LSTM will have success, but other traditional forecasting methods can detect trend. However, seasonality is different. ***Seasonality*** in sales data is a pattern that can happen at multiple frequencies (annual, quarterly, monthly, weekly, and even daily). The LSTM is great at detecting seasonality because it often has autocorrelation. Therefore, LSTM’s and GRU’s can be great options to help improve seasonality detection, and therefore reduce overall forecast error in the ***Sales Forecast***.

**About The Authors**

This ***DEEP LEARNING TUTORIAL*** was a combined effort: [Sigrid Keydana](https://www.linkedin.com/in/sigrid-keydana-9a16b410/) and [Matt Dancho](https://www.linkedin.com/in/mattdancho/).

Sigrid is the *TensorFlow Developer Advocate at RStudio*, where she develops amazing deep learning using the [R TensorFlow API](https://tensorflow.rstudio.com/). She’s passionate about deep learning, machine learning and statistics, R, Linux and functional programming. In a short period of time, I’ve developed a tremendous respect for Sigrid. You’ll see why when you walk through the R TensorFlow code in this tutorial

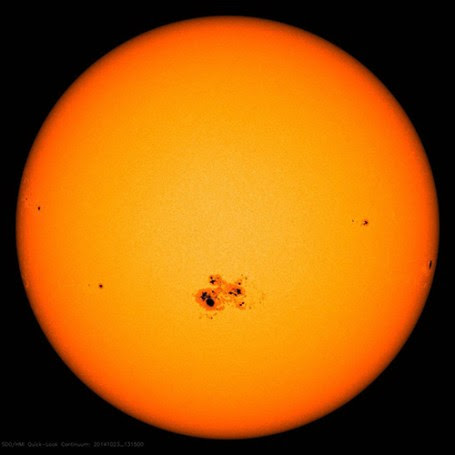
You know me (Matt). I’m the Founder of [Business Science](http://www.business-science.io/) where I strive for one mission: To empower Data scientists interested in Business and Finance.

**Deep Learning for Time Series Forecasting: Predicting Sunspot Frequency with Keras**

By Sigrid Keydana, TensorFlow Developer Advocate at RStudio,  
And Matt Dancho, Founder of Business Science

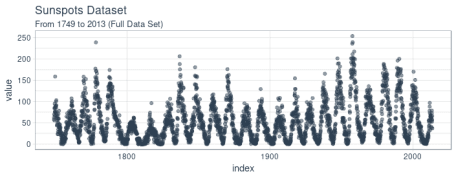
**Forecasting sunspots with deep learning**

In this post we will examine making time series predictions using the [sunspots](https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/sunspot.month.html) dataset that ships with base R. Sunspots are dark spots on the sun, associated with lower temperature. Here’s an image from NASA showing the solar phenomenon.



Source: [NASA](https://www.nasa.gov/content/goddard/largest-sunspot-of-solar-cycle)

We’re using the monthly version of the dataset, sunspots.month (there is a yearly version, too).  
It contains 265 years worth of data (from 1749 through 2013) on the number of sunspots per month.



Forecasting this dataset is challenging because of high short term variability as well as long-term irregularities evident in the cycles. For example, maximum amplitudes reached by the low frequency cycle differ a lot, as does the number of high frequency cycle steps needed to reach that maximum low frequency cycle height.

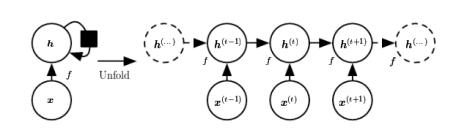
Our post will focus on two dominant aspects: how to apply deep learning to time series forecasting, and how to properly apply cross validation in this domain.  
For the latter, we will use the [rsample](https://cran.r-project.org/package=rsample) package that allows to do resampling on time series data.  
As to the former, our goal is not to reach utmost performance but to show the general course of action when using recurrent neural networks to model this kind of data.

**Recurrent neural networks**

When our data has a sequential structure, it is recurrent neural networks (RNNs) we use to model it.

As of today, among RNNs, the best established architectures are the GRU (Gated Recurrent Unit) and the LSTM (Long Short Term Memory). For today, let’s not zoom in on what makes them special, but on what they have in common with the most stripped-down RNN: the basic recurrence structure.

In contrast to the prototype of a neural network, often called Multilayer Perceptron (MLP), the RNN has a state that is carried on over time. This is nicely seen in this diagram from [Goodfellow et al.](http://www.deeplearningbook.org/), a.k.a. the “bible of deep learning”:



At each time, the state is a combination of the current input and the previous hidden state. This is reminiscent of autoregressive models, but with neural networks, there has to be some point where we halt the dependence.

That’s because in order to determine the weights, we keep calculating how our loss changes as the input changes.  
Now if the input we have to consider, at an arbitrary timestep, ranges back indefinitely – then we will not be able to calculate all those gradients.  
In practice, then, our hidden state will, at every iteration, be carried forward through a fixed number of steps.

We’ll come back to that as soon as we’ve loaded and pre-processed the data.

**Setup, pre-processing, and exploration**

**Libraries**

Here, first, are the libraries needed for this tutorial.

# Core Tidyverse

library(tidyverse)

library(glue)

library(forcats)

# Time Series

library(timetk)

library(tidyquant)

library(tibbletime)

# Visualization

library(cowplot)

# Preprocessing

library(recipes)

# Sampling / Accuracy

library(rsample)

library(yardstick)

# Modeling

library(keras)

library(tfruns)

If you have not previously run Keras in R, you will need to install Keras using the install\_keras() function.

# Install Keras if you have not installed before

install\_keras()

**Data**

sunspot.month is a ts class (not tidy), so we’ll convert to a tidy data set using the tk\_tbl() function from timetk. We use this instead of as.tibble() from tibble to automatically preserve the time series index as a zoo yearmon index. Last, we’ll convert the zoo index to date using lubridate::as\_date() (loaded with tidyquant) and then change to a tbl\_time object to make time series operations easier.

sun\_spots <- datasets::sunspot.month %>%

tk\_tbl() %>%

mutate(index = as\_date(index)) %>%

as\_tbl\_time(index = index)

sun\_spots

# A time tibble: 3,177 x 2

# Index: index

index value

1 1749-01-01 58

2 1749-02-01 62.6

3 1749-03-01 70

4 1749-04-01 55.7

5 1749-05-01 85

6 1749-06-01 83.5

7 1749-07-01 94.8

8 1749-08-01 66.3

9 1749-09-01 75.9

10 1749-10-01 75.5

# ... with 3,167 more rows

**Exploratory data analysis**

The time series is long (265 years!). We can visualize the time series both in full, and zoomed in on the first 10 years to get a feel for the series.

**Visualizing sunspot data with cowplot**

We’ll make two ggplots and combine them using cowplot::plot\_grid(). Note that for the zoomed in plot, we make use of tibbletime::time\_filter(), which is an easy way to perform time-based filtering.

p1 <- sun\_spots %>%

ggplot(aes(index, value)) +

geom\_point(color = palette\_light()[[1]], alpha = 0.5) +

theme\_tq() +

labs(

title = "From 1749 to 2013 (Full Data Set)"

)

p2 <- sun\_spots %>%

filter\_time("start" ~ "1800") %>%

ggplot(aes(index, value)) +

geom\_line(color = palette\_light()[[1]], alpha = 0.5) +

geom\_point(color = palette\_light()[[1]]) +

geom\_smooth(method = "loess", span = 0.2, se = FALSE) +

theme\_tq() +

labs(

title = "1749 to 1759 (Zoomed In To Show Changes over the Year)",

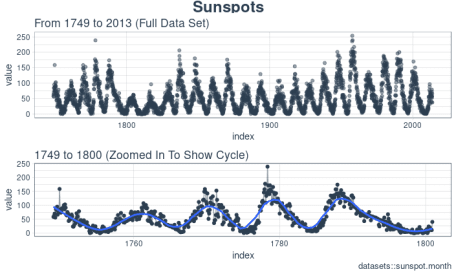
caption = "datasets::sunspot.month"

)

p\_title <- ggdraw() +

draw\_label("Sunspots", size = 18, fontface = "bold", colour = palette\_light()[[1]])

plot\_grid(p\_title, p1, p2, ncol = 1, rel\_heights = c(0.1, 1, 1))



**Backtesting: time series cross validation**

When doing cross validation on sequential data, the time dependencies on preceding samples must be preserved. We can create a cross validation sampling plan by offsetting the window used to select sequential sub-samples. In essence, we’re creatively dealing with the fact that there’s no future test data available by creating multiple synthetic “futures” – a process often, esp. in finance, called “backtesting”.

As mentioned in the introduction, the [rsample](https://cran.r-project.org/package=rsample) package includes facitlities for backtesting on time series. The vignette, [“Time Series Analysis Example”](https://topepo.github.io/rsample/articles/Applications/Time_Series.html), describes a procedure that uses the rolling\_origin() function to create samples designed for time series cross validation. We’ll use this approach.

**Developing a backtesting strategy**

The sampling plan we create uses 50 years (initial = 12 x 50 samples) for the training set and ten years (assess = 12 x 10) for the testing (validation) set. We select a skip span of about twenty years (skip = 12 x 20 – 1) to approximately evenly distribute the samples into 6 sets that span the entire 265 years of sunspots history. Last, we select cumulative = FALSE to allow the origin to shift which ensures that models on more recent data are not given an unfair advantage (more observations) over those operating on less recent data. The tibble return contains the rolling\_origin\_resamples.

periods\_train <- 12 \* 100

periods\_test <- 12 \* 50

skip\_span <- 12 \* 22 - 1

rolling\_origin\_resamples <- rolling\_origin(

sun\_spots,

initial = periods\_train,

assess = periods\_test,

cumulative = FALSE,

skip = skip\_span

)

rolling\_origin\_resamples

# Rolling origin forecast resampling

# A tibble: 6 x 2

splits id

1 Slice1

2 Slice2

3 Slice3

4 Slice4

5 Slice5

6 Slice6

**Visualizing the backtesting strategy**

We can visualize the resamples with two custom functions. The first, plot\_split(), plots one of the resampling splits using ggplot2. Note that an expand\_y\_axis argument is added to expand the date range to the full sun\_spots dataset date range. This will become useful when we visualize all plots together.

# Plotting function for a single split

plot\_split <- function(split, expand\_y\_axis = TRUE, alpha = 1, size = 1, base\_size = 14) {

# Manipulate data

train\_tbl <- training(split) %>%

add\_column(key = "training")

test\_tbl <- testing(split) %>%

add\_column(key = "testing")

data\_manipulated <- bind\_rows(train\_tbl, test\_tbl) %>%

as\_tbl\_time(index = index) %>%

mutate(key = fct\_relevel(key, "training", "testing"))

# Collect attributes

train\_time\_summary <- train\_tbl %>%

tk\_index() %>%

tk\_get\_timeseries\_summary()

test\_time\_summary <- test\_tbl %>%

tk\_index() %>%

tk\_get\_timeseries\_summary()

# Visualize

g <- data\_manipulated %>%

ggplot(aes(x = index, y = value, color = key)) +

geom\_line(size = size, alpha = alpha) +

theme\_tq(base\_size = base\_size) +

scale\_color\_tq() +

labs(

title = glue("Split: {split$id}"),

subtitle = glue("{train\_time\_summary$start} to {test\_time\_summary$end}"),

y = "", x = ""

) +

theme(legend.position = "none")

if (expand\_y\_axis) {

sun\_spots\_time\_summary <- sun\_spots %>%

tk\_index() %>%

tk\_get\_timeseries\_summary()

g <- g +

scale\_x\_date(limits = c(sun\_spots\_time\_summary$start,

sun\_spots\_time\_summary$end))

}

return(g)

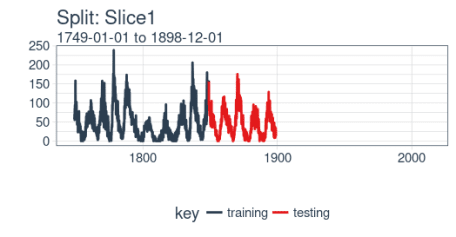
}

The plot\_split() function takes one split (in this case Slice01), and returns a visual of the sampling strategy. We expand the axis to the range for the full dataset using expand\_y\_axis = TRUE.

rolling\_origin\_resamples$splits[[1]] %>%

plot\_split(expand\_y\_axis = TRUE) +

theme(legend.position = "bottom")



The second function, plot\_sampling\_plan(), scales the plot\_split() function to all of the samples using purrr and cowplot.

# Plotting function that scales to all splits

plot\_sampling\_plan <- function(sampling\_tbl, expand\_y\_axis = TRUE,

ncol = 3, alpha = 1, size = 1, base\_size = 14,

title = "Sampling Plan") {

# Map plot\_split() to sampling\_tbl

sampling\_tbl\_with\_plots <- sampling\_tbl %>%

mutate(gg\_plots = map(splits, plot\_split,

expand\_y\_axis = expand\_y\_axis,

alpha = alpha, base\_size = base\_size))

# Make plots with cowplot

plot\_list <- sampling\_tbl\_with\_plots$gg\_plots

p\_temp <- plot\_list[[1]] + theme(legend.position = "bottom")

legend <- get\_legend(p\_temp)

p\_body <- plot\_grid(plotlist = plot\_list, ncol = ncol)

p\_title <- ggdraw() +

draw\_label(title, size = 14, fontface = "bold", colour = palette\_light()[[1]])

g <- plot\_grid(p\_title, p\_body, legend, ncol = 1, rel\_heights = c(0.05, 1, 0.05))

return(g)

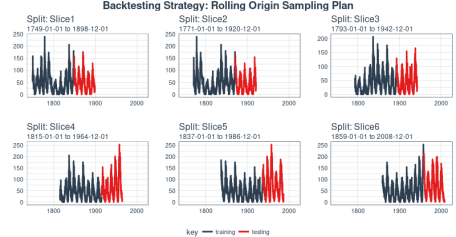
}

We can now visualize the entire backtesting strategy with plot\_sampling\_plan(). We can see how the sampling plan shifts the sampling window with each progressive slice of the train/test splits.

rolling\_origin\_resamples %>%

plot\_sampling\_plan(expand\_y\_axis = T, ncol = 3, alpha = 1, size = 1, base\_size = 10,

title = "Backtesting Strategy: Rolling Origin Sampling Plan")

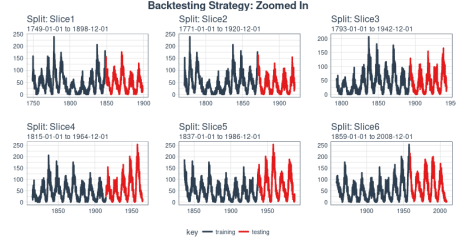


And, we can set expand\_y\_axis = FALSE to zoom in on the samples.

rolling\_origin\_resamples %>%

plot\_sampling\_plan(expand\_y\_axis = F, ncol = 3, alpha = 1, size = 1, base\_size = 10,

title = "Backtesting Strategy: Zoomed In")



We’ll use this backtesting strategy (6 samples from one time series each with 50/10 split in years and a ~20 year offset) when testing the veracity of the LSTM model on the sunspots dataset.

**The LSTM model**

To begin, we’ll develop an LSTM model on a single sample from the backtesting strategy, namely, the most recent slice. We’ll then apply the model to all samples to investigate modeling performance.

example\_split <- rolling\_origin\_resamples$splits[[6]]

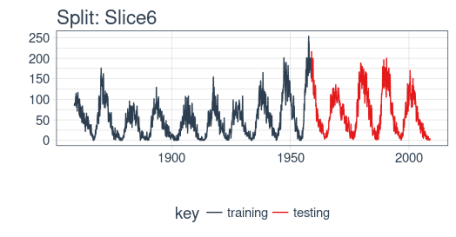
example\_split\_id <- rolling\_origin\_resamples$id[[6]]

We can reuse the plot\_split() function to visualize the split. Set expand\_y\_axis = FALSE to zoom in on the subsample.

plot\_split(example\_split, expand\_y\_axis = FALSE, size = 0.5) +

theme(legend.position = "bottom") +

ggtitle(glue("Split: {example\_split\_id}"))



**Data setup**

To aid hyperparameter tuning, besides the training set we also need a validation set.  
For example, we will use a callback, callback\_early\_stopping, that stops training when no significant performance is seen on the validation set (what’s considered significant is up to you).

We will dedicate 2 thirds of the analysis set to training, and 1 third to validation.

df\_trn <- analysis(example\_split)[1:800, , drop = FALSE]

df\_val <- analysis(example\_split)[801:1200, , drop = FALSE]

df\_tst <- assessment(example\_split)

First, let’s combine the training and testing data sets into a single data set with a column key that specifies where they came from (either “training” or “testing)”. Note that the tbl\_time object will need to have the index respecified during the bind\_rows() step, but [this issue](https://github.com/tidyverse/dplyr/issues/3259) should be corrected in dplyr soon.

df <- bind\_rows(

df\_trn %>% add\_column(key = "training"),

df\_val %>% add\_column(key = "validation"),

df\_tst %>% add\_column(key = "testing")

) %>%

as\_tbl\_time(index = index)

df

# A time tibble: 1,800 x 3

# Index: index

index value key

1 1849-06-01 81.1 training

2 1849-07-01 78 training

3 1849-08-01 67.7 training

4 1849-09-01 93.7 training

5 1849-10-01 71.5 training

6 1849-11-01 99 training

7 1849-12-01 97 training

8 1850-01-01 78 training

9 1850-02-01 89.4 training

10 1850-03-01 82.6 training

# ... with 1,790 more rows

**Preprocessing with recipes**

The LSTM algorithm will usually work better if the input data has been centered and scaled. We can conveniently accomplish this using the recipes package. In addition to step\_center and step\_scale, we’re using step\_sqrt to reduce variance and remov outliers. The actual transformations are executed when we bake the data according to the recipe:

rec\_obj <- recipe(value ~ ., df) %>%

step\_sqrt(value) %>%

step\_center(value) %>%

step\_scale(value) %>%

prep()

df\_processed\_tbl <- bake(rec\_obj, df)

df\_processed\_tbl

# A tibble: 1,800 x 3

index value key

1 1849-06-01 0.714 training

2 1849-07-01 0.660 training

3 1849-08-01 0.473 training

4 1849-09-01 0.922 training

5 1849-10-01 0.544 training

6 1849-11-01 1.01 training

7 1849-12-01 0.974 training

8 1850-01-01 0.660 training

9 1850-02-01 0.852 training

10 1850-03-01 0.739 training

# ... with 1,790 more rows

Next, let’s capture the original center and scale so we can invert the steps after modeling. The square root step can then simply be undone by squaring the back-transformed data.

center\_history <- rec\_obj$steps[[2]]$means["value"]

scale\_history <- rec\_obj$steps[[3]]$sds["value"]

c("center" = center\_history, "scale" = scale\_history)

center.value scale.value

6.694468 3.238935

**Reshaping the data**

Keras LSTM 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.

Here, num\_samples is the number of observations in the set. This will get fed to the model in portions of batch\_size. The second dimension, num\_timesteps, is the length of the hidden state we were talking about above. Finally, the third dimension is the number of predictors we’re using. For univariate time series, this is 1.

How long should we choose the hidden state to be? This generally depends on the dataset and our goal.  
If we did one-step-ahead forecasts – thus, forecasting the following month only – our main concern would be choosing a state length that allows to learn any patterns present in the data.

Now say we wanted to forecast 12 months instead, as does [SILSO](http://sidc.be/silso/home), the *World Data Center for the production, preservation and dissemination of the international sunspot number*.  
The way we can do this, with Keras, is by wiring the LSTM hidden states to sets of consecutive outputs of the same length. Thus, if we want to produce predictions for 12 months, our LSTM should have a hidden state length of 12.

These 12 time steps will then get wired to 12 linear predictor units using a time\_distributed() wrapper.  
That wrapper’s task is to apply the same calculation (i.e., the same weight matrix) to every state input it receives.

Now, what’s the target array’s format supposed to be? As we’re forecasting several timesteps here, the target data again needs to be 3-dimensional. Dimension 1 again is the batch dimension, dimension 2 again corresponds to the number of timesteps (the forecasted ones), and dimension 3 is the size of the wrapped layer.  
In our case, the wrapped layer is a layer\_dense() of a single unit, as we want exactly one prediction per point in time.

So, let’s reshape the data. The main action here is creating the sliding windows of 12 steps of input, followed by 12 steps of output each. This is easiest to understand with a shorter and simpler example. Say our input were the numbers from 1 to 10, and our chosen sequence length (state size) were 4. Tthis is how we would want our training input to look:

1,2,3,4

2,3,4,5

3,4,5,6

And our target data, correspondingly:

5,6,7,8

6,7,8,9

7,8,9,10

We’ll define a short function that does this reshaping on a given dataset.  
Then finally, we add the third axis that is formally needed (even though that axis is of size 1 in our case).

# these variables are being defined just because of the order in which

# we present things in this post (first the data, then the model)

# they will be superseded by FLAGS$n\_timesteps, FLAGS$batch\_size and n\_predictions

# in the following snippet

n\_timesteps <- 12

n\_predictions <- n\_timesteps

batch\_size <- 10

# functions used

build\_matrix <- function(tseries, overall\_timesteps) {

t(sapply(1:(length(tseries) - overall\_timesteps + 1), function(x)

tseries[x:(x + overall\_timesteps - 1)]))

}

reshape\_X\_3d <- function(X) {

dim(X) <- c(dim(X)[1], dim(X)[2], 1)

X

}

# extract values from data frame

train\_vals <- df\_processed\_tbl %>%

filter(key == "training") %>%

select(value) %>%

pull()

valid\_vals <- df\_processed\_tbl %>%

filter(key == "validation") %>%

select(value) %>%

pull()

test\_vals <- df\_processed\_tbl %>%

filter(key == "testing") %>%

select(value) %>%

pull()

# build the windowed matrices

train\_matrix <-

build\_matrix(train\_vals, n\_timesteps + n\_predictions)

valid\_matrix <-

build\_matrix(valid\_vals, n\_timesteps + n\_predictions)

test\_matrix <- build\_matrix(test\_vals, n\_timesteps + n\_predictions)

# separate matrices into training and testing parts

# also, discard last batch if there are fewer than batch\_size samples

# (a purely technical requirement)

X\_train <- train\_matrix[, 1:n\_timesteps]

y\_train <- train\_matrix[, (n\_timesteps + 1):(n\_timesteps \* 2)]

X\_train <- X\_train[1:(nrow(X\_train) %/% batch\_size \* batch\_size), ]

y\_train <- y\_train[1:(nrow(y\_train) %/% batch\_size \* batch\_size), ]

X\_valid <- valid\_matrix[, 1:n\_timesteps]

y\_valid <- valid\_matrix[, (n\_timesteps + 1):(n\_timesteps \* 2)]

X\_valid <- X\_valid[1:(nrow(X\_valid) %/% batch\_size \* batch\_size), ]

y\_valid <- y\_valid[1:(nrow(y\_valid) %/% batch\_size \* batch\_size), ]

X\_test <- test\_matrix[, 1:n\_timesteps]

y\_test <- test\_matrix[, (n\_timesteps + 1):(n\_timesteps \* 2)]

X\_test <- X\_test[1:(nrow(X\_test) %/% batch\_size \* batch\_size), ]

y\_test <- y\_test[1:(nrow(y\_test) %/% batch\_size \* batch\_size), ]

# add on the required third axis

X\_train <- reshape\_X\_3d(X\_train)

X\_valid <- reshape\_X\_3d(X\_valid)

X\_test <- reshape\_X\_3d(X\_test)

y\_train <- reshape\_X\_3d(y\_train)

y\_valid <- reshape\_X\_3d(y\_valid)

y\_test <- reshape\_X\_3d(y\_test)

**Building the LSTM model**

Now that we have our data in the required form, let’s finally build the model.  
As always in deep learning, an important, and often time-consuming, part of the job is tuning hyperparameters. To keep this post self-contained, and considering this is primarily a tutorial on how to use LSTM in R, let’s assume the following settings were found after extensive experimentation (in reality experimentation *did* take place, but not to a degree that performance couldn’t possibly be improved).

Instead of hard coding the hyperparameters, we’ll use [tfruns](https://tensorflow.rstudio.com/tools/tfruns/articles/tuning.html) to set up an environment where we could easily perform grid search.

We’ll quickly comment on what these parameters do but mainly leave those topics to further posts.

FLAGS <- flags(

# There is a so-called "stateful LSTM" in Keras. While LSTM is stateful per se,

# this adds a further tweak where the hidden states get initialized with values

# from the item at same position in the previous batch.

# This is helpful just under specific circumstances, or if you want to create an

# "infinite stream" of states, in which case you'd use 1 as the batch size.

# Below, we show how the code would have to be changed to use this, but it won't be further

# discussed here.

flag\_boolean("stateful", FALSE),

# Should we use several layers of LSTM?

# Again, just included for completeness, it did not yield any superior performance on this task.

# This will actually stack exactly one additional layer of LSTM units.

flag\_boolean("stack\_layers", FALSE),

# number of samples fed to the model in one go

flag\_integer("batch\_size", 10),

# size of the hidden state, equals size of predictions

flag\_integer("n\_timesteps", 12),

# how many epochs to train for

flag\_integer("n\_epochs", 100),

# fraction of the units to drop for the linear transformation of the inputs

flag\_numeric("dropout", 0.2),

# fraction of the units to drop for the linear transformation of the recurrent state

flag\_numeric("recurrent\_dropout", 0.2),

# loss function. Found to work better for this specific case than mean squared error

flag\_string("loss", "logcosh"),

# optimizer = stochastic gradient descent. Seemed to work better than adam or rmsprop here

# (as indicated by limited testing)

flag\_string("optimizer\_type", "sgd"),

# size of the LSTM layer

flag\_integer("n\_units", 128),

# learning rate

flag\_numeric("lr", 0.003),

# momentum, an additional parameter to the SGD optimizer

flag\_numeric("momentum", 0.9),

# parameter to the early stopping callback

flag\_integer("patience", 10)

)

# the number of predictions we'll make equals the length of the hidden state

n\_predictions <- FLAGS$n\_timesteps

# how many features = predictors we have

n\_features <- 1

# just in case we wanted to try different optimizers, we could add here

optimizer <- switch(FLAGS$optimizer\_type,

sgd = optimizer\_sgd(lr = FLAGS$lr, momentum = FLAGS$momentum))

# callbacks to be passed to the fit() function

# We just use one here: we may stop before n\_epochs if the loss on the validation set

# does not decrease (by a configurable amount, over a configurable time)

callbacks <- list(

callback\_early\_stopping(patience = FLAGS$patience)

)

After all these preparations, the code for constructing and training the model is rather short!  
Let’s first quickly view the “long version”, that would allow you to test stacking several LSTMs or use a stateful LSTM, then go through the final short version (that does neither) and comment on it.

This, just for reference, is the complete code.

model <- keras\_model\_sequential()

model %>%

layer\_lstm(

units = FLAGS$n\_units,

batch\_input\_shape = c(FLAGS$batch\_size, FLAGS$n\_timesteps, n\_features),

dropout = FLAGS$dropout,

recurrent\_dropout = FLAGS$recurrent\_dropout,

return\_sequences = TRUE,

stateful = FLAGS$stateful

)

if (FLAGS$stack\_layers) {

model %>%

layer\_lstm(

units = FLAGS$n\_units,

dropout = FLAGS$dropout,

recurrent\_dropout = FLAGS$recurrent\_dropout,

return\_sequences = TRUE,

stateful = FLAGS$stateful

)

}

model %>% time\_distributed(layer\_dense(units = 1))

model %>%

compile(

loss = FLAGS$loss,

optimizer = optimizer,

metrics = list("mean\_squared\_error")

)

if (!FLAGS$stateful) {

model %>% fit(

x = X\_train,

y = y\_train,

validation\_data = list(X\_valid, y\_valid),

batch\_size = FLAGS$batch\_size,

epochs = FLAGS$n\_epochs,

callbacks = callbacks

)

} else {

for (i in 1:FLAGS$n\_epochs) {

model %>% fit(

x = X\_train,

y = y\_train,

validation\_data = list(X\_valid, y\_valid),

callbacks = callbacks,

batch\_size = FLAGS$batch\_size,

epochs = 1,

shuffle = FALSE

)

model %>% reset\_states()

}

}

if (FLAGS$stateful)

model %>% reset\_states()

Now let’s step through the simpler, yet better (or equally) performing configuration below.

# create the model

model <- keras\_model\_sequential()

# add layers

# we have just two, the LSTM and the time\_distributed

model %>%

layer\_lstm(

units = FLAGS$n\_units,

# the first layer in a model needs to know the shape of the input data

batch\_input\_shape = c(FLAGS$batch\_size, FLAGS$n\_timesteps, n\_features),

dropout = FLAGS$dropout,

recurrent\_dropout = FLAGS$recurrent\_dropout,

# by default, an LSTM just returns the final state

return\_sequences = TRUE

) %>% time\_distributed(layer\_dense(units = 1))

model %>%

compile(

loss = FLAGS$loss,

optimizer = optimizer,

# in addition to the loss, Keras will inform us about current MSE while training

metrics = list("mean\_squared\_error")

)

history <- model %>% fit(

x = X\_train,

y = y\_train,

validation\_data = list(X\_valid, y\_valid),

batch\_size = FLAGS$batch\_size,

epochs = FLAGS$n\_epochs,

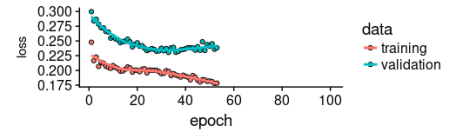
callbacks = callbacks

)

As we see, training was stopped after ~55 epochs as validation loss did not decrease any more.  
We also see that performance on the validation set is way worse than performance on the training set – normally indicating overfitting.

This topic too, we’ll leave to a separate discussion another time, but interestingly regularization using higher values of dropout and recurrent\_dropout (combined with increasing model capacity) did not yield better generalization performance. This is probably related to the characteristics of this specific time series we mentioned in the introduction.

plot(history, metrics = "loss")



Now let’s see how well the model was able to capture the characteristics of the training set.

pred\_train <- model %>%

predict(X\_train, batch\_size = FLAGS$batch\_size) %>%

.[, , 1]

# Retransform values to original scale

pred\_train <- (pred\_train \* scale\_history + center\_history) ^2

compare\_train <- df %>% filter(key == "training")

# build a dataframe that has both actual and predicted values

for (i in 1:nrow(pred\_train)) {

varname <- paste0("pred\_train", i)

compare\_train <-

mutate(compare\_train,!!varname := c(

rep(NA, FLAGS$n\_timesteps + i - 1),

pred\_train[i,],

rep(NA, nrow(compare\_train) - FLAGS$n\_timesteps \* 2 - i + 1)

))

}

We compute the average RSME over all sequences of predictions.

coln <- colnames(compare\_train)[4:ncol(compare\_train)]

cols <- map(coln, quo(sym(.)))

rsme\_train <-

map\_dbl(cols, function(col)

rmse(

compare\_train,

truth = value,

estimate = !!col,

na.rm = TRUE

)) %>% mean()

rsme\_train

21.01495

How do these predictions really look? As a visualization of all predicted sequences would look pretty crowded, we arbitrarily pick start points at regular intervals.

ggplot(compare\_train, aes(x = index, y = value)) + geom\_line() +

geom\_line(aes(y = pred\_train1), color = "cyan") +

geom\_line(aes(y = pred\_train50), color = "red") +

geom\_line(aes(y = pred\_train100), color = "green") +

geom\_line(aes(y = pred\_train150), color = "violet") +

geom\_line(aes(y = pred\_train200), color = "cyan") +

geom\_line(aes(y = pred\_train250), color = "red") +

geom\_line(aes(y = pred\_train300), color = "red") +

geom\_line(aes(y = pred\_train350), color = "green") +

geom\_line(aes(y = pred\_train400), color = "cyan") +

geom\_line(aes(y = pred\_train450), color = "red") +

geom\_line(aes(y = pred\_train500), color = "green") +

geom\_line(aes(y = pred\_train550), color = "violet") +

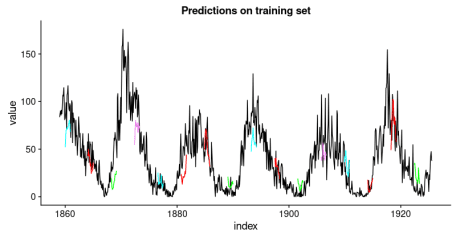
geom\_line(aes(y = pred\_train600), color = "cyan") +

geom\_line(aes(y = pred\_train650), color = "red") +

geom\_line(aes(y = pred\_train700), color = "red") +

geom\_line(aes(y = pred\_train750), color = "green") +

ggtitle("Predictions on the training set")



This looks pretty good. From the validation loss, we don’t quite expect the same from the test set, though.

Let’s see.

pred\_test <- model %>%

predict(X\_test, batch\_size = FLAGS$batch\_size) %>%

.[, , 1]

# Retransform values to original scale

pred\_test <- (pred\_test \* scale\_history + center\_history) ^2

pred\_test[1:10, 1:5] %>% print()

compare\_test <- df %>% filter(key == "testing")

# build a dataframe that has both actual and predicted values

for (i in 1:nrow(pred\_test)) {

varname <- paste0("pred\_test", i)

compare\_test <-

mutate(compare\_test,!!varname := c(

rep(NA, FLAGS$n\_timesteps + i - 1),

pred\_test[i,],

rep(NA, nrow(compare\_test) - FLAGS$n\_timesteps \* 2 - i + 1)

))

}

compare\_test %>% write\_csv(str\_replace(model\_path, ".hdf5", ".test.csv"))

compare\_test[FLAGS$n\_timesteps:(FLAGS$n\_timesteps + 10), c(2, 4:8)] %>% print()

coln <- colnames(compare\_test)[4:ncol(compare\_test)]

cols <- map(coln, quo(sym(.)))

rsme\_test <-

map\_dbl(cols, function(col)

rmse(

compare\_test,

truth = value,

estimate = !!col,

na.rm = TRUE

)) %>% mean()

rsme\_test

31.31616

ggplot(compare\_test, aes(x = index, y = value)) + geom\_line() +

geom\_line(aes(y = pred\_test1), color = "cyan") +

geom\_line(aes(y = pred\_test50), color = "red") +

geom\_line(aes(y = pred\_test100), color = "green") +

geom\_line(aes(y = pred\_test150), color = "violet") +

geom\_line(aes(y = pred\_test200), color = "cyan") +

geom\_line(aes(y = pred\_test250), color = "red") +

geom\_line(aes(y = pred\_test300), color = "green") +

geom\_line(aes(y = pred\_test350), color = "cyan") +

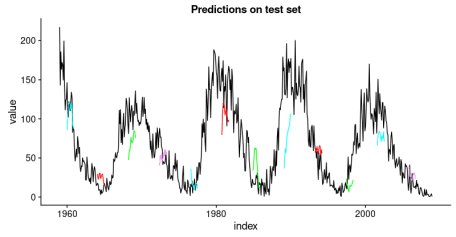
geom\_line(aes(y = pred\_test400), color = "red") +

geom\_line(aes(y = pred\_test450), color = "green") +

geom\_line(aes(y = pred\_test500), color = "cyan") +

geom\_line(aes(y = pred\_test550), color = "violet") +

ggtitle("Predictions on test set")



That’s not as good as on the training set, but not bad either, given this time series is quite challenging.

Having defined and run our model on a manually chosen example split, let’s now revert to our overall re-sampling frame.

**Backtesting the model on all splits**

To obtain predictions on all splits, we move the above code into a function and apply it to all splits.  
First, here’s the function. It returns a list of two dataframes, one for the training and test sets each, that contain the model’s predictions together with the actual values.

obtain\_predictions <- function(split) {

df\_trn <- analysis(split)[1:800, , drop = FALSE]

df\_val <- analysis(split)[801:1200, , drop = FALSE]

df\_tst <- assessment(split)

df <- bind\_rows(

df\_trn %>% add\_column(key = "training"),

df\_val %>% add\_column(key = "validation"),

df\_tst %>% add\_column(key = "testing")

) %>%

as\_tbl\_time(index = index)

rec\_obj <- recipe(value ~ ., df) %>%

step\_sqrt(value) %>%

step\_center(value) %>%

step\_scale(value) %>%

prep()

df\_processed\_tbl <- bake(rec\_obj, df)

center\_history <- rec\_obj$steps[[2]]$means["value"]

scale\_history <- rec\_obj$steps[[3]]$sds["value"]

FLAGS <- flags(

flag\_boolean("stateful", FALSE),

flag\_boolean("stack\_layers", FALSE),

flag\_integer("batch\_size", 10),

flag\_integer("n\_timesteps", 12),

flag\_integer("n\_epochs", 100),

flag\_numeric("dropout", 0.2),

flag\_numeric("recurrent\_dropout", 0.2),

flag\_string("loss", "logcosh"),

flag\_string("optimizer\_type", "sgd"),

flag\_integer("n\_units", 128),

flag\_numeric("lr", 0.003),

flag\_numeric("momentum", 0.9),

flag\_integer("patience", 10)

)

n\_predictions <- FLAGS$n\_timesteps

n\_features <- 1

optimizer <- switch(FLAGS$optimizer\_type,

sgd = optimizer\_sgd(lr = FLAGS$lr, momentum = FLAGS$momentum))

callbacks <- list(

callback\_early\_stopping(patience = FLAGS$patience)

)

train\_vals <- df\_processed\_tbl %>%

filter(key == "training") %>%

select(value) %>%

pull()

valid\_vals <- df\_processed\_tbl %>%

filter(key == "validation") %>%

select(value) %>%

pull()

test\_vals <- df\_processed\_tbl %>%

filter(key == "testing") %>%

select(value) %>%

pull()

train\_matrix <-

build\_matrix(train\_vals, FLAGS$n\_timesteps + n\_predictions)

valid\_matrix <-

build\_matrix(valid\_vals, FLAGS$n\_timesteps + n\_predictions)

test\_matrix <-

build\_matrix(test\_vals, FLAGS$n\_timesteps + n\_predictions)

X\_train <- train\_matrix[, 1:FLAGS$n\_timesteps]

y\_train <-

train\_matrix[, (FLAGS$n\_timesteps + 1):(FLAGS$n\_timesteps \* 2)]

X\_train <-

X\_train[1:(nrow(X\_train) %/% FLAGS$batch\_size \* FLAGS$batch\_size),]

y\_train <-

y\_train[1:(nrow(y\_train) %/% FLAGS$batch\_size \* FLAGS$batch\_size),]

X\_valid <- valid\_matrix[, 1:FLAGS$n\_timesteps]

y\_valid <-

valid\_matrix[, (FLAGS$n\_timesteps + 1):(FLAGS$n\_timesteps \* 2)]

X\_valid <-

X\_valid[1:(nrow(X\_valid) %/% FLAGS$batch\_size \* FLAGS$batch\_size),]

y\_valid <-

y\_valid[1:(nrow(y\_valid) %/% FLAGS$batch\_size \* FLAGS$batch\_size),]

X\_test <- test\_matrix[, 1:FLAGS$n\_timesteps]

y\_test <-

test\_matrix[, (FLAGS$n\_timesteps + 1):(FLAGS$n\_timesteps \* 2)]

X\_test <-

X\_test[1:(nrow(X\_test) %/% FLAGS$batch\_size \* FLAGS$batch\_size),]

y\_test <-

y\_test[1:(nrow(y\_test) %/% FLAGS$batch\_size \* FLAGS$batch\_size),]

X\_train <- reshape\_X\_3d(X\_train)

X\_valid <- reshape\_X\_3d(X\_valid)

X\_test <- reshape\_X\_3d(X\_test)

y\_train <- reshape\_X\_3d(y\_train)

y\_valid <- reshape\_X\_3d(y\_valid)

y\_test <- reshape\_X\_3d(y\_test)

model <- keras\_model\_sequential()

model %>%

layer\_lstm(

units = FLAGS$n\_units,

batch\_input\_shape = c(FLAGS$batch\_size, FLAGS$n\_timesteps, n\_features),

dropout = FLAGS$dropout,

recurrent\_dropout = FLAGS$recurrent\_dropout,

return\_sequences = TRUE

) %>% time\_distributed(layer\_dense(units = 1))

model %>%

compile(

loss = FLAGS$loss,

optimizer = optimizer,

metrics = list("mean\_squared\_error")

)

model %>% fit(

x = X\_train,

y = y\_train,

validation\_data = list(X\_valid, y\_valid),

batch\_size = FLAGS$batch\_size,

epochs = FLAGS$n\_epochs,

callbacks = callbacks

)

pred\_train <- model %>%

predict(X\_train, batch\_size = FLAGS$batch\_size) %>%

.[, , 1]

# Retransform values

pred\_train <- (pred\_train \* scale\_history + center\_history) ^ 2

compare\_train <- df %>% filter(key == "training")

for (i in 1:nrow(pred\_train)) {

varname <- paste0("pred\_train", i)

compare\_train <-

mutate(compare\_train, !!varname := c(

rep(NA, FLAGS$n\_timesteps + i - 1),

pred\_train[i, ],

rep(NA, nrow(compare\_train) - FLAGS$n\_timesteps \* 2 - i + 1)

))

}

pred\_test <- model %>%

predict(X\_test, batch\_size = FLAGS$batch\_size) %>%

.[, , 1]

# Retransform values

pred\_test <- (pred\_test \* scale\_history + center\_history) ^ 2

compare\_test <- df %>% filter(key == "testing")

for (i in 1:nrow(pred\_test)) {

varname <- paste0("pred\_test", i)

compare\_test <-

mutate(compare\_test, !!varname := c(

rep(NA, FLAGS$n\_timesteps + i - 1),

pred\_test[i, ],

rep(NA, nrow(compare\_test) - FLAGS$n\_timesteps \* 2 - i + 1)

))

}

list(train = compare\_train, test = compare\_test)

}

Mapping the function over all splits yields a list of predictions.

all\_split\_preds <- rolling\_origin\_resamples %>%

mutate(predict = map(splits, obtain\_predictions))

Calculate RMSE on all splits:

calc\_rmse <- function(df) {

coln <- colnames(df)[4:ncol(df)]

cols <- map(coln, quo(sym(.)))

map\_dbl(cols, function(col)

rmse(

df,

truth = value,

estimate = !!col,

na.rm = TRUE

)) %>% mean()

}

all\_split\_preds <- all\_split\_preds %>% unnest(predict)

all\_split\_preds\_train <- all\_split\_preds[seq(1, 11, by = 2), ]

all\_split\_preds\_test <- all\_split\_preds[seq(2, 12, by = 2), ]

all\_split\_rmses\_train <- all\_split\_preds\_train %>%

mutate(rmse = map\_dbl(predict, calc\_rmse)) %>%

select(id, rmse)

all\_split\_rmses\_test <- all\_split\_preds\_test %>%

mutate(rmse = map\_dbl(predict, calc\_rmse)) %>%

select(id, rmse)

How does it look? Here’s RMSE on the training set for the 6 splits.

all\_split\_rmses\_train

# A tibble: 6 x 2

id rmse

1 Slice1 22.2

2 Slice2 20.9

3 Slice3 18.8

4 Slice4 23.5

5 Slice5 22.1

6 Slice6 21.1

all\_split\_rmses\_test

# A tibble: 6 x 2

id rmse

1 Slice1 21.6

2 Slice2 20.6

3 Slice3 21.3

4 Slice4 31.4

5 Slice5 35.2

6 Slice6 31.4

Looking at these numbers, we see something interesting: Generalization performance is much better for the first three slices of the time series than for the latter ones. This confirms our impression, stated above, that there seems to be some hidden development going on, rendering forecasting more difficult.

And here are visualizations of the predictions on the respective training and test sets.

First, the training sets:

plot\_train <- function(slice, name) {

ggplot(slice, aes(x = index, y = value)) + geom\_line() +

geom\_line(aes(y = pred\_train1), color = "cyan") +

geom\_line(aes(y = pred\_train50), color = "red") +

geom\_line(aes(y = pred\_train100), color = "green") +

geom\_line(aes(y = pred\_train150), color = "violet") +

geom\_line(aes(y = pred\_train200), color = "cyan") +

geom\_line(aes(y = pred\_train250), color = "red") +

geom\_line(aes(y = pred\_train300), color = "red") +

geom\_line(aes(y = pred\_train350), color = "green") +

geom\_line(aes(y = pred\_train400), color = "cyan") +

geom\_line(aes(y = pred\_train450), color = "red") +

geom\_line(aes(y = pred\_train500), color = "green") +

geom\_line(aes(y = pred\_train550), color = "violet") +

geom\_line(aes(y = pred\_train600), color = "cyan") +

geom\_line(aes(y = pred\_train650), color = "red") +

geom\_line(aes(y = pred\_train700), color = "red") +

geom\_line(aes(y = pred\_train750), color = "green") +

ggtitle(name)

}

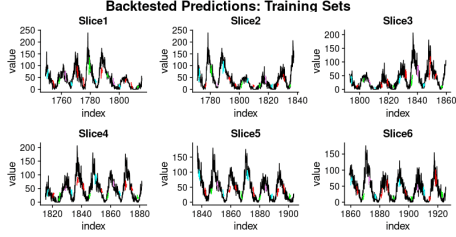
train\_plots <- map2(all\_split\_preds\_train$predict, all\_split\_preds\_train$id, plot\_train)

p\_body\_train <- plot\_grid(plotlist = train\_plots, ncol = 3)

p\_title\_train <- ggdraw() +

draw\_label("Backtested Predictions: Training Sets", size = 18, fontface = "bold")

plot\_grid(p\_title\_train, p\_body\_train, ncol = 1, rel\_heights = c(0.05, 1, 0.05))



And the test sets:

plot\_test <- function(slice, name) {

ggplot(slice, aes(x = index, y = value)) + geom\_line() +

geom\_line(aes(y = pred\_test1), color = "cyan") +

geom\_line(aes(y = pred\_test50), color = "red") +

geom\_line(aes(y = pred\_test100), color = "green") +

geom\_line(aes(y = pred\_test150), color = "violet") +

geom\_line(aes(y = pred\_test200), color = "cyan") +

geom\_line(aes(y = pred\_test250), color = "red") +

geom\_line(aes(y = pred\_test300), color = "green") +

geom\_line(aes(y = pred\_test350), color = "cyan") +

geom\_line(aes(y = pred\_test400), color = "red") +

geom\_line(aes(y = pred\_test450), color = "green") +

geom\_line(aes(y = pred\_test500), color = "cyan") +

geom\_line(aes(y = pred\_test550), color = "violet") +

ggtitle(name)

}

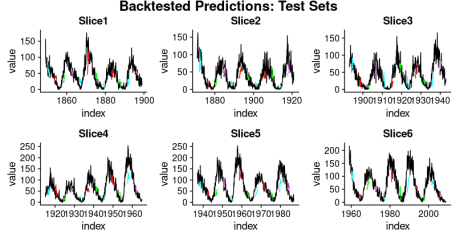
test\_plots <- map2(all\_split\_preds\_test$predict, all\_split\_preds\_test$id, plot\_test)

p\_body\_test <- plot\_grid(plotlist = test\_plots, ncol = 3)

p\_title\_test <- ggdraw() +

draw\_label("Backtested Predictions: Test Sets", size = 18, fontface = "bold")

plot\_grid(p\_title\_test, p\_body\_test, ncol = 1, rel\_heights = c(0.05, 1, 0.05))



This has been a long post, and necessarily will have left a lot of questions open, first and foremost: How do we obtain good settings for the hyperparameters (learning rate, number of epochs, dropout)?  
How do we choose the length of the hidden state? Or even, can we have an intuition how well LSTM will perform on a given dataset (with its specific characteristics)?