Introduction to Recurrent Neural Networks using the Keras package

Saint Louis R User Group

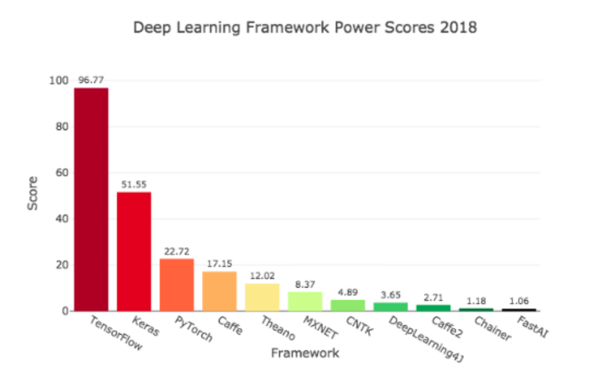
John Snyder

December 13, 2018

Outline

1. Overview of Keras
2. RNN/LSTM
3. Using Generator Functions
4. How to RNN/
5. Temperature Prediction Example (time permitting)
6. Final Thoughts

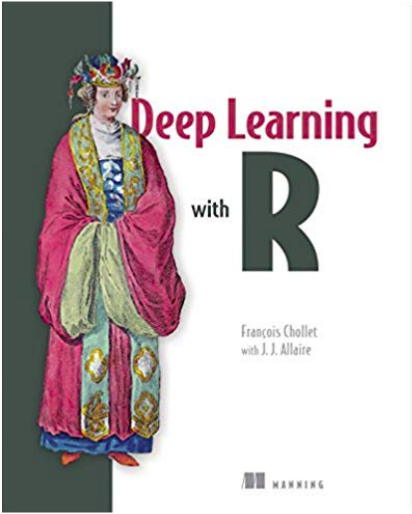
Broad Adoption



Deep learning frameworks ranking computed by Jeff Hale, based on 11 data sources across 7 categories

- Deep learning is cool
- Keras is becoming more and more popular.
  - High level and easy to use

Keras Book


Deep Learning
with R
François Chollet
with J. J. Allaire
MANNING

Installation

- CPU

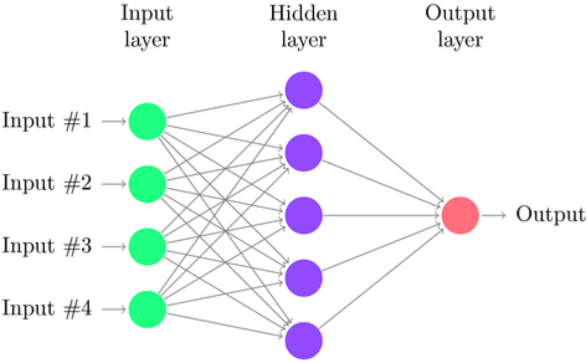
```r
install.packages("keras")
keras::install_keras()
```

- GPU

```r
install.packages("keras")
keras::install_keras(tensorflow = "gpu")
```

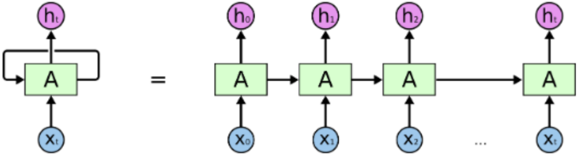
- If prerequisite software is missing, instructions are provided.
- CUDA and cuDNN libraries required for GPU
- NVIDIA GPUs are required
  - Really fast

Neural Networks

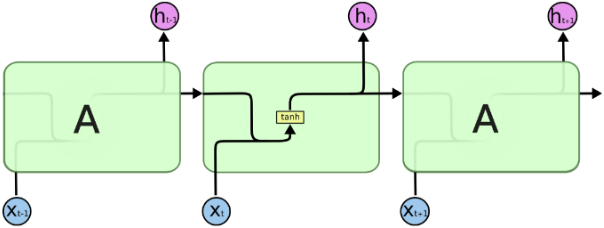


- Doesnt work well for sequential data
  - Inputs are treated separately

Recurrent Neural Network



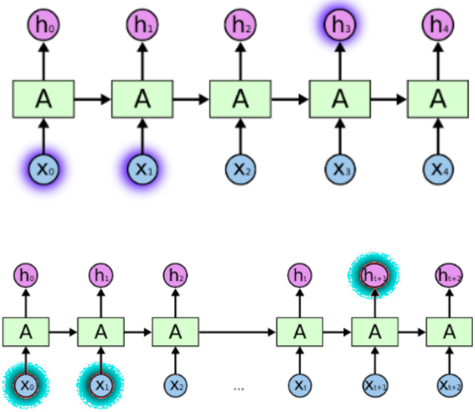
An unrolled recurrent neural network.



The repeating module in a standard RNN contains a single layer.

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Problems with RNNs

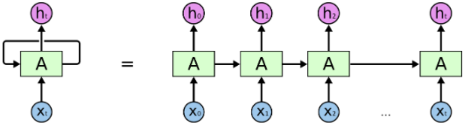


- RNNs can have a *vanishing*(or sometimes *exploding*) gradient problem.

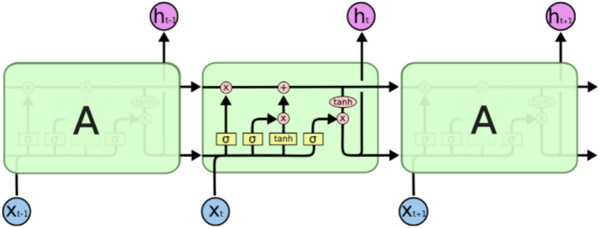
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM

- LSTM = Long Short Term Memory



An unrolled recurrent neural network.



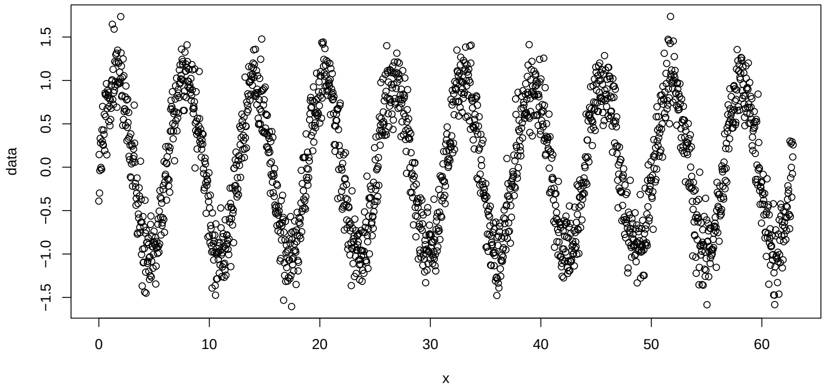
The repeating module in an LSTM contains four interacting layers.

- LSTM cells combat this using robust gates.

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Simulated Data

```
# Generate 10 cycles of sin(x) + noise
n_seq <- 2000
x <- seq(0,10*2*pi,length.out = n_seq)
truth <- sin(x)
data <- truth + rnorm(n_seq,mean=0,sd=.25) %>%  as.matrix(ncol=1)
plot(x,data)
```



Our Data

```r
data[1:8,] %>% round(digits = 4)
```

```
## [1] -0.3903  0.1454 -0.2983 -0.0388  0.3320  0.2799 -0.0024 -0.0220
```

Say we want to predict the "next" value using the previous 3

	t.3	t.2	t.1	t
1	-0.3903	0.1454	-0.2983	-0.0388
2	0.1454	-0.2983	-0.0388	0.3320
3	-0.2983	-0.0388	0.3320	0.2799
4	-0.0388	0.3320	0.2799	-0.0024
5	0.3320	0.2799	-0.0024	-0.0220

- The previous 3 are used to predict the next.
- Keras trains models using *batches* of these data

Aside: Generator Functions

- Function that you call repeatedly to obtain a sequence of values

```
sequence_generator <- function(start) {
  value <- start - 1
  return(
    function() {
      value <<- value + 1
      return(value)
    }
  )
}
```

- Function which returns a function
- Superassignment <<- used to maintain internal state
- Allows us to pass small chunks of data to Keras *on the fly*
  - *Extremly* useful for not wasting resources.

Aside: Generator Functions

```
gen <- sequence_generator(1)
gen()
```

```
## [1] 1
```

```
gen()
```

```
## [1] 2
```

```
gen()
```

```
## [1] 3
```

```
#value  # returns error!
get("value",envir = environment(gen))
```

```
## [1] 3
```

A Generator Function for our Example

```
TS_generator <- function(data, lookback, min_index=1, max_index,
                          batch_size = 4, shuffle=FALSE){
  i <- min_index + lookback
  function(){
    if(shuffle){
      rows <- sample(c((min_index+lookback):max_index), size = batch_size)
    }else{
      if (i + batch_size >= max_index){i <<- min_index + lookback}

      rows <- c(i:min(i+batch_size-1, max_index))
      i <<- i + length(rows)
    }
    #initialize output objects
    samples <- array(0, dim = c(length(rows),lookback))
    targets <- array(0, dim = length(rows))    #one target for each sample

    for(j in 1:length(rows)){
      indices <- seq(rows[j] - lookback, rows[j]-1)
      samples[j,] <- data[indices]
      targets[j] <- data[rows[j]]
    }

    list(samples, targets)
  }
}
```

Time Series Data Generation

```
index_data <- 1:30
train_gen <- TS_generator(index_data,lookback = 3,
                          min_index = 1,max_index = 30,
                          batch_size = 4)
train_gen()
```

```
## [[1]]
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    2    3    4
## [3,]    3    4    5
## [4,]    4    5    6
##
## [[2]]
## [1] 4 5 6 7
```

```
train_gen()
```

```
## [[1]]
##      [,1] [,2] [,3]
## [1,]    5    6    7
## [2,]    6    7    8
## [3,]    7    8    9
## [4,]    8    9   10
##
## [[2]]
## [1]  8  9 10 11
```

Shuffled Time Series Data Generation

```
index_data <- 1:30
train_gen <- TS_generator(index_data,lookback = 9,
                          min_index = 1,max_index = 30,
                          batch_size = 4, shuffle = T)
train_gen()
```

```
## [[1]]
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]   20   21   22   23   24   25   26   27   28
## [2,]   10   11   12   13   14   15   16   17   18
## [3,]    9   10   11   12   13   14   15   16   17
## [4,]   13   14   15   16   17   18   19   20   21
##
## [[2]]
## [1] 29 19 18 22
```

```
train_gen()
```

```
## [[1]]
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    2    3    4    5    6    7    8    9   10
## [2,]   20   21   22   23   24   25   26   27   28
## [3,]   14   15   16   17   18   19   20   21   22
## [4,]   17   18   19   20   21   22   23   24   25
##
## [[2]]
## [1] 11 29 23 26
```

How to RNN: Model Building

Keras is set up to *stack* model layers

```
RNN_model <- keras_model_sequential() %>%
  layer_simple_rnn(input_shape = list(NULL, 1),
                   units=16,
                   return_sequences = TRUE,
                   dropout = .2, recurrent_dropout = .3) %>%
  layer_simple_rnn(units = 8) %>%
  layer_dense(units = 1)
```

- input_shape: Input is *one* time series(feature) of *arbitrary* length
- units: Number of RNN sequences
- return_sequences: Return every output of each RNN sequence
- dropout/recurrent dropout: Randomly set weights to 0 during model fitting
    - Prevents overfitting

How to RNN: Compile

Deep learning models are fit iteratively by making the predictions closer to the observations.

```
RNN_model %>% compile(
  optimizer = optimizer_rmsprop(),
  loss = "mse")
```

- loss: The function which defines how far away our predictions are
- optimizer: Approach used to minimize the loss

Another optional argument is:

- metrics: Other functions evaluated during training, for monitoring

One can even define their own loss functions.

How to RNN: Fit the Model

Keras has special functionality to fit models using the generator functions we defined earlier!

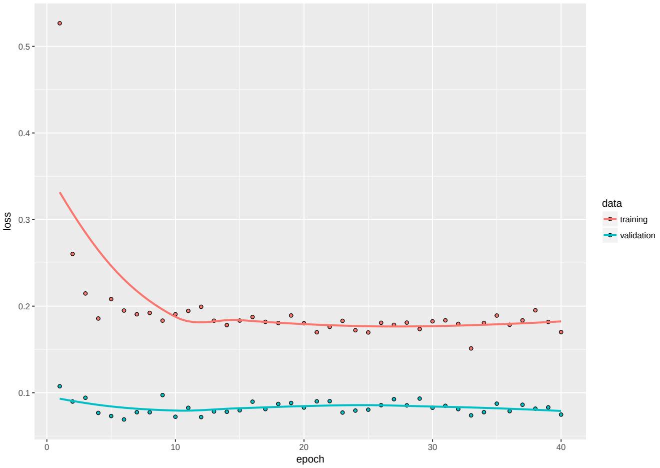
```
# data = sin(x) data from earlier
train_gen <- TS_generator(data,lookback = 100,
                          min_index = 1,max_index = 1100,
                          batch_size = 8,shuffle = T)
val_gen <- TS_generator(data,lookback = 100,
                          min_index = 1101,max_index = 1500,
                          batch_size = 8,shuffle = T)
```

and then...

```
RNNfit <- RNN_model %>% fit_generator(
  train_gen,
  steps_per_epoch = 125,
  epochs = 40,
  validation_data = val_gen,
  validation_steps = 50
)
```

- epochs/steps: Big/Small iterations through the data

How to RNN: The Result

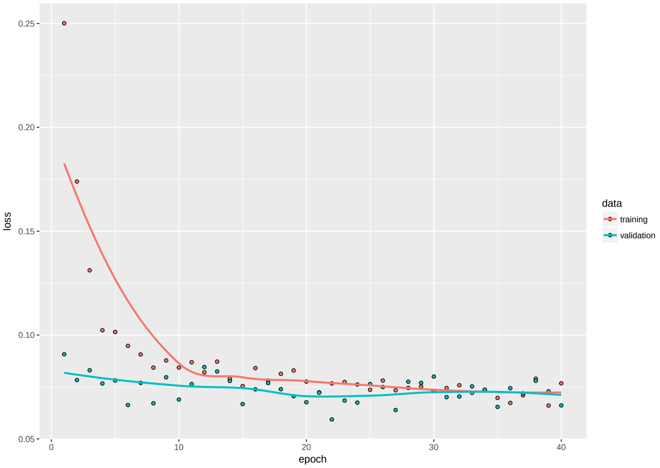


How to LSTM: Everything is the same

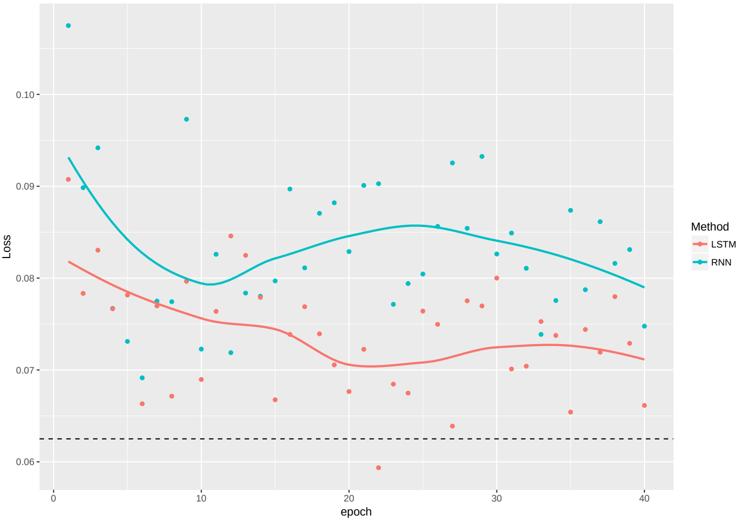
```r
LSTM_model <- keras_model_sequential() %>%
  layer_lstm(units = 16,
                    return_sequences = TRUE,
                    dropout = .2, recurrent_dropout = .3,
                    input_shape = list(NULL, 1)) %>%
  layer_lstm(units = 8) %>%
  layer_dense(units = 1)

LSTM_model %>% compile(
  optimizer = optimizer_rmsprop(),
  loss = "mse")

LSTMfit <- LSTM_model %>% fit_generator(
  train_gen,
  steps_per_epoch = 125,
  epochs = 40,
  validation_data = val_gen,
  validation_steps = 50
)
```

- Everything is basically the same.

How to LSTM: The Result

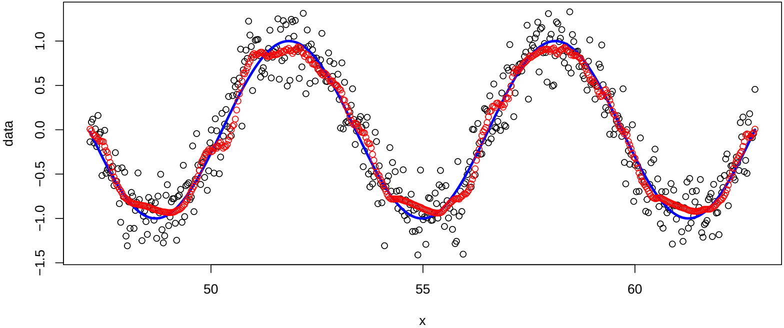


RNN vs LSTM



Predict on the test set

```
test_gen <- TS_generator_test(data,lookback = 100,
                              min_index = 1500-100,max_index = 2000,
                              batch_size = 1,shuffle = F)

pred <- predict_generator(LSTM_model,test_gen,500)

plot(x[1501:2000],data[(1501):2000],type="p")
lines(x[1501:2000],sin(x[1501:2000]),lwd=3,col="blue")
points(x[1501:2000],pred,col="red")
```



Weather example

RNN_Keras.R

Final Thoughts

- Center Variables
- In real applications, always compare against a naive baseline
  - Don't treat these as magic black boxes.
- Use generator functions
  - Relatively recent addition to R-Keras
  - Also useful for image problems
- BTW: Financial Markets have very different statistical characteristics than natural phenomena