Deep Learning for sequence data (3)

Contents

IMDB data set

```
max_features <- 10000
maxlen <- 500
batch_size <- 32
cat("Loading data...\n")
```

```
## Loading data...
```

```
imdb <- dataset_imdb(num_words = max_features)
c(c(input_train, y_train), c(input_test, y_test)) %<-% imdb
cat(length(input_train), "train sequences\n")
```

```
## 25000 train sequences
```

```
cat(length(input_test), "test sequences")
```

```
## 25000 test sequences
```

```
cat("Pad sequences (samples x time)\n")
```

```
## Pad sequences (samples x time)
```

```
input_train <- pad_sequences(input_train, maxlen = maxlen)
input_test <- pad_sequences(input_test, maxlen = maxlen)
cat("input_train shape:", dim(input_train), "\n")
```

```
## input_train shape: 25000 500
```

```
cat("input_test shape:", dim(input_test), "\n")
```

```
## input_test shape: 25000 500
```

```
dim(input_train)
```

```
## [1] 25000   500
```

Long Short Term Memory (LSTM layer)

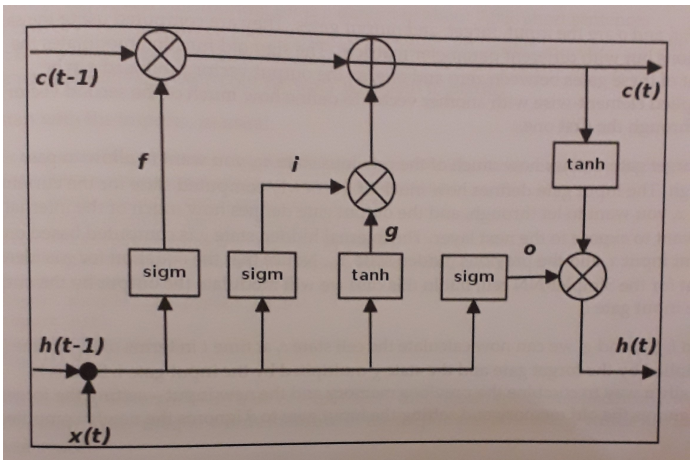
In order to combat the problem of vanishing gradients, Sepp Hochreiter and Jürgen Schmidhuber introduced the long short-term memory (LSTM) architecture. The basic principle behind the architecture was that the

network would be designed for the purpose of reliably transmitting important information many time steps into the future.

An LSTM cell maintains a **cell state**, c_t, which can be thought of as the cell's internal beliefs about current status of the sequence. It adds a way to carry information across many timesteps. This is distinct from the **hidden state**, h_t, which is ultimately output by the cell after final timestep. The cell state is the same length as the hidden state (the number of units in the cell).

Imagine a conveyor belt running parallel to the sequence you're processing. Information from the sequence can jump onto the conveyor belt at any point, be transported to later timestep, and jump off, intact, when you need it. This is essentially what a LSTM does: it saves information for later, thus preventing older signals from gradually vanishing during processing.

The diagram looks complicated, but let us look at it component by component. The line across the top of the diagram is the **cell state**, c_t, and represent the internal memory of the unit. The line across the bottom is the **hidden state**, h_t, and the i, f, o and g gates are the mechanism by which the LSTM works around the vanishing gradient problem. During training, the LSTM learns the parameters for these gates.



In order to gain deeper understanding of how these gates modulate the LSTM's hidden state, let us consider the equations that show how it calculates the hidden state h_t at timestep t from the hidden state h_{t-1} at the previous timestep:

$$
\begin{aligned}
i &= \sigma(W_i H_{t-1} + U_i x_t + b_i) \\
f &= \sigma(W_f H_{t-1} + U_f x_t + b_f) \\
o &= \sigma(W_o H_{t-1} + U_o x_t + b_o) \\
g &= \tanh(W_g H_{t-1} + U_g x_t + b_g) \\
c_t &= (c_{t-1} \otimes f) \oplus (g \otimes i) \\
h_t &= o \otimes \tanh(c_t)
\end{aligned}
$$

Here i, f, and o are input, forget and output gates. They are computed using the same equation but with different parameter matrices. The sigmoid function modulates the output of these gates between 0 and 1, so the output vector produced can be multiplied element-wise with another vector to define how much of the second vector can pass through the first one.

-The forget gate defines how much of the previous hidden state h_{t-1} you want to allow to pass through.

-The input gate defines how much of the newly computed state for the current input x_t you want to let though.

-The output gate defines how much of the internal state you want to expose to next step.

-The internal hidden state g is computed based on the current input x_t and the previous hidden state h_{t-1}. Notice that the equation for g is identical to that for simple RNN cell, but in this case we will modulate the output but the output of the input gate i.

-Given i, f, o and g, we can now calculate the cell state c_t at time t in terms of c_{t-1} multiplied by the forget gate and the state g multiplied by the input gate i.

-So this is basically a way to combine the previous memory and the new input - setting the keep gate to 0 ignores the old memory and setting the write gate to 0 ignores the newly computed state.

-Finally, the hidden state h_t at time t is computed by multiplying the memory c_t with the output gate.

One thing to realize is that an LSTM is a drop-in replacement for a simpleRNN layer, the only difference is that LSTM are resistant to the vanishing gradient problem. You can replace an RNN layer in a network with an LSTM without worrying about any side effects. You should generally see better results along with longer training times.

```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = max_features, output_dim = 32) %>%
  layer_lstm(units = 28) %>%
  layer_dense(units = 1, activation = "sigmoid")
summary(model)
```

```
## Model: "sequential"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## embedding (Embedding)        (None, None, 32)          320000
## _________________________________________________________________________
## lstm (LSTM)                  (None, 28)                6832
## _________________________________________________________________________
## dense (Dense)                (None, 1)                 29
## =========================================================================
## Total params: 326,861
## Trainable params: 326,861
## Non-trainable params: 0
## _________________________________________________________________________
```

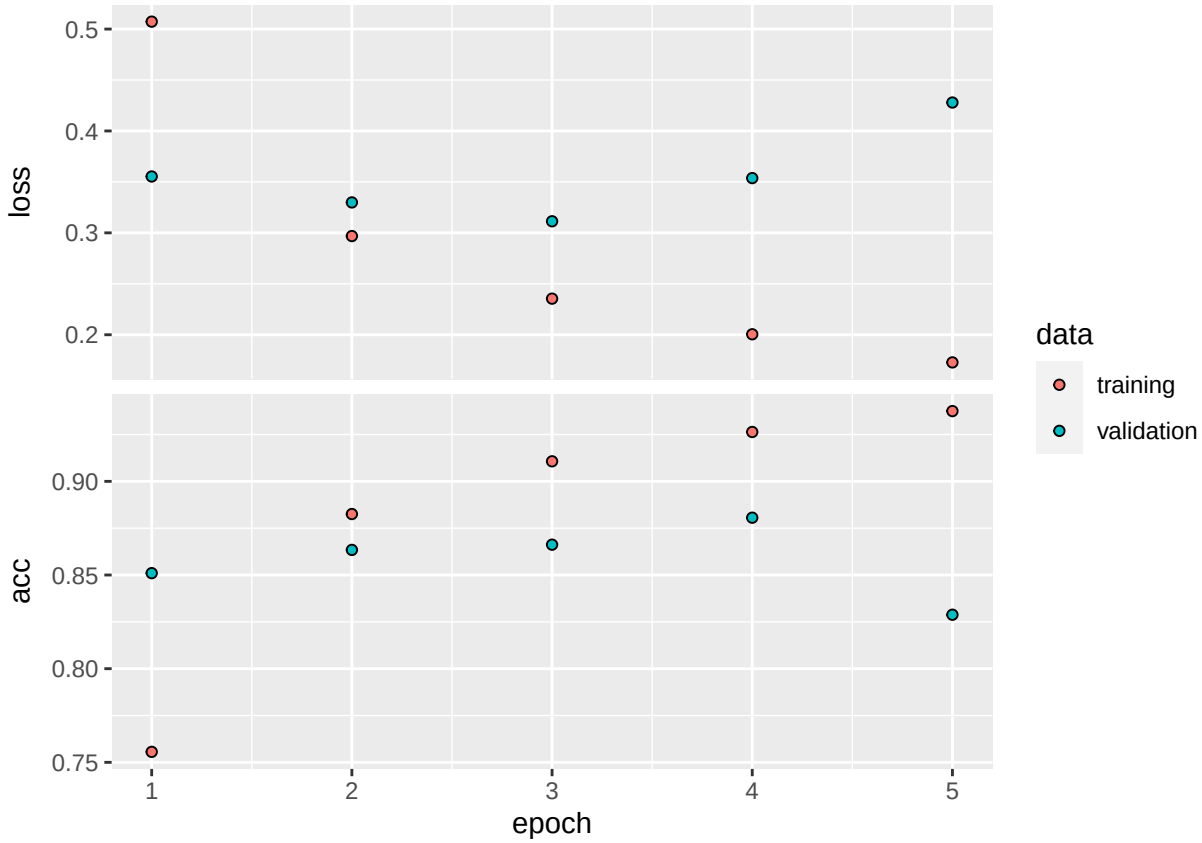
Observe that number of parameters required to train the LSTM layer are:

$$4(units \times units) + 4(units \times input) + 4(units) = 4 * 1708 = 6832$$

```
model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("acc")
)
```

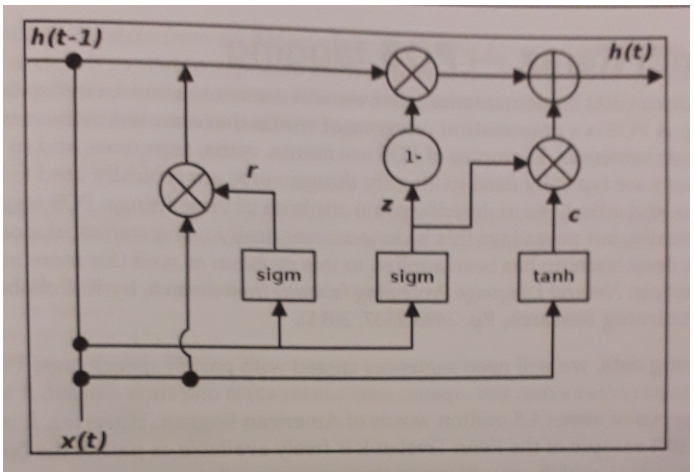
```
history <- model %>% fit(
  input_train, y_train,
  epochs = 5,
  batch_size = 128,
  validation_split = 0.2
)
```

```
plot(history)
```



Gated recurrent unit - GRU

The GRU is a variant of the LSTM introduced by K. Cho. It retains the LSTM's resitance to the vanishing gradient problem, but its internal structure is simpler, and therefore is faster to train. The gates for GRU cell are illustrated in the following diagram:



GRU has two gates, an update gate z and a reset gate r:

- The update gate defines how much previous memory to keep around
- The reset gate defines how to combine the new input with the previous memory

There is no persistent cell state distinct from the hidden state as in LSTM. The following equations define

the gating mechanism in a GRU

$$
\begin{aligned}
z &= \sigma(W_z H_{t-1} + U_z x_t + b_z) \\
r &= \sigma(W_r H_{t-1} + U_r x_t + b_r) \\
c &= \tanh(W_c(H_{t-1} \otimes r) + U_c x_t + b_c) \\
h_t &= (z \otimes c) \oplus ((1 - z) \otimes H_{t-1})
\end{aligned}
$$

GRU and LSTM have comparable performance and there is no simple way to recommend ono or the other for specific task. While GRUs are faster to train and need less data to generalize, in situtaions where there is enough data, an LSTM's greater expressive power may lead better results. Like LSTMs, GRUs are drop-in replacements for simple RNN cell.

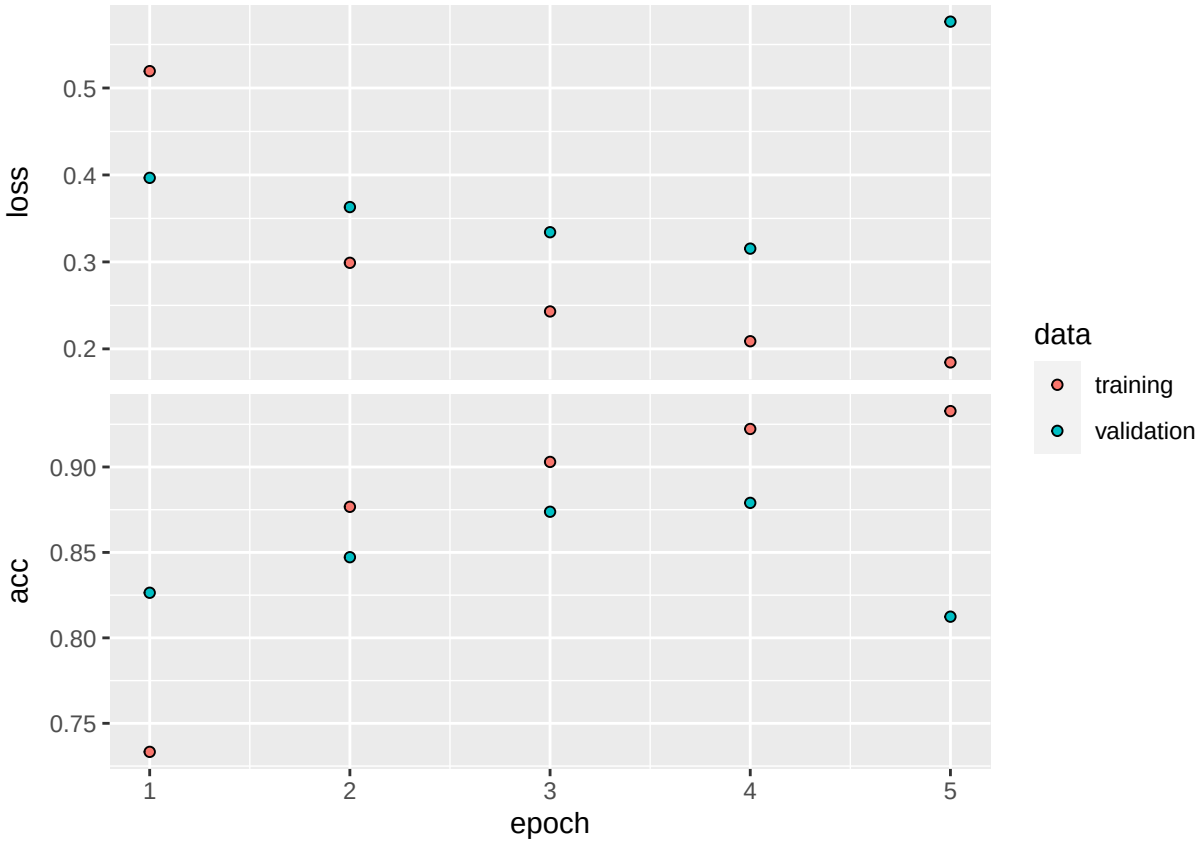
```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = max_features, output_dim = 32) %>%
  layer_gru(units = 28) %>%
  layer_dense(units = 1, activation = "sigmoid")
summary(model)
```

```
## Model: "sequential_1"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## embedding_1 (Embedding)      (None, None, 32)          320000
## _________________________________________________________________________
## gru (GRU)                    (None, 28)                5124
## _________________________________________________________________________
## dense_1 (Dense)              (None, 1)                 29
## =========================================================================
## Total params: 325,153
## Trainable params: 325,153
## Non-trainable params: 0
## _________________________________________________________________________
```

```
model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("acc")
)
```

```
history <- model %>% fit(
  input_train, y_train,
  epochs = 5,
  batch_size = 128,
  validation_split = 0.2
)
```

```
plot(history)
```


loss
0.5
0.4
0.3
0.2
acc
0.90
0.85
0.80
0.75
1
2
3
4
5
epoch
data
training
validation