CIS 8398
Advanced AI Topics in Business

#Recurrent Neural Networks

Yu-Kai Lin

Agenda

- Understanding recurrent neural networks (RNNs)
- Long Short-Term Memory (LSTM)
- Bidirectional LSTM

[**Acknowledgments**] The materials in the following slides are based on the source(s) below:

- Deep Learning with R by Francois Chollet and J.J. Allaire
- R interface to Keras
- Understanding LSTM Networks by Christopher Olah

Prerequisites

```
#install.packages("tidyverse")
#install.packages("keras")

library(tidyverse)
library(keras)
#install_keras() # will take a while; you should have done this already
```

Some of the deep neural networks are too large to train on a normal laptop, so you may use the ones that I have already trained:

```
model <- load_model_hdf5("some_model_filename.h5")
```

Download pretrained models (and their histories) here:
https://www.dropbox.com/s/hpdhvzrx9p9etnl/deeplearning-pretrained.zip?dl=0

If you are using the VM, the pretrained model and history files are stored in C:/CIS8398/data/deeplearning-pretrained/

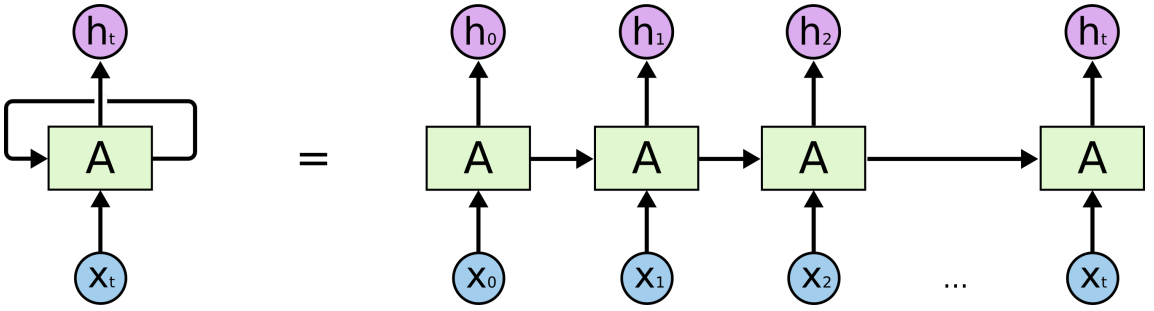
Intro to RNNs

A major characteristic of all neural networks we've seen so far, such as densely connected networks and convnets, is that they have no memory. Each input shown to them is processed independently, with no state kept in between inputs.

In contrast, as we're reading the present sentence, we're processing it word by word while keeping memories of what came before; this gives us a fluid representation of the meaning conveyed by this sentence.

A **recurrent neural network (RNN)** adopts the same principle, albeit in an extremely simplified version: it processes sequences by iterating through the sequence elements and maintaining a *state* containing information relative to what it has seen so far.

A recurrent network: a network with a loop that keeps track of states



Pseudocode RNN

```
state_t = 0 # initial state
for (input_t in input_sequence) { # iterate over sequence elements
  output_t <- f(input_t, state_t) # get output based on current state
  state_t <- output_t             # update current state
}
```

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

A recurrent layer in Keras

```
layer_simple_rnn(units = 32, return_sequences = FALSE)
```

`layer_simple_rnn` processes batches of sequences, like all other Keras layers. This means it takes inputs of shape `(batch_size, timesteps, input_features)`.

The `units` parameter controls the output dimensionality.

`layer_simple_rnn` can be run in two different modes: it can return either the full sequences of successive outputs for each timestep (a 3D tensor of shape `(batch_size, timesteps, output_features)`) or only the last output for each input sequence (a 2D tensor of shape `(batch_size, output_features)`).

These two modes are controlled by the `return_sequences` argument (default is `FALSE` and return only the last output for each input sequence).

Let's look at an example that uses `layer_simple_rnn` and returns only the output at the last timestep:

```
library(keras)
model <- keras_model_sequential() |>
  layer_embedding(input_dim = 10000, output_dim = 32) |>
  layer_simple_rnn(units = 32) # return_sequences = FALSE, by default

summary(model)
```

```
## Model: "sequential"
## _________________________________________________________________
##  Layer (type)                        Output Shape                    Param #
## =================================================================
##  embedding (Embedding)               (None, None, 32)                320000
##  simple_rnn (SimpleRNN)              (None, 32)                      2080
## =================================================================
## Total params: 322080 (1.23 MB)
## Trainable params: 322080 (1.23 MB)
## Non-trainable params: 0 (0.00 Byte)
## _________________________________________________________________
```

The following example returns the full state sequence:

```
model2 <- keras_model_sequential() |>
  layer_embedding(input_dim = 10000, output_dim = 32) |>
  layer_simple_rnn(units = 32, return_sequences = TRUE)

summary(model2) # see the difference in output shape
```

```
## Model: "sequential_1"
## _________________________________________________________________
##  Layer (type)                        Output Shape                    Param #
## =================================================================
##  embedding_1 (Embedding)             (None, None, 32)                320000
##  simple_rnn_1 (SimpleRNN)            (None, None, 32)                2080
## =================================================================
## Total params: 322080 (1.23 MB)
## Trainable params: 322080 (1.23 MB)
## Non-trainable params: 0 (0.00 Byte)
## _________________________________________________________________
```

It's sometimes useful to stack several recurrent layers one after the other in order to increase the representational power of a network. In such a setup, **we have to get all of the intermediate layers to return full sequences**:

```
model3 <- keras_model_sequential() |>
  layer_embedding(input_dim = 10000, output_dim = 32) |>
  layer_simple_rnn(units = 32, return_sequences = TRUE) |>
  layer_simple_rnn(units = 32, return_sequences = TRUE) |>
  layer_simple_rnn(units = 32, return_sequences = TRUE) |>
  layer_simple_rnn(units = 32, return_sequences = FALSE)

summary(model3)
```

```
## Model: "sequential_2"
## _________________________________________________________________
##  Layer (type)                        Output Shape              Param #
## =================================================================
##  embedding_2 (Embedding)             (None, None, 32)          320000
##  simple_rnn_5 (SimpleRNN)            (None, None, 32)          2080
##  simple_rnn_4 (SimpleRNN)            (None, None, 32)          2080
##  simple_rnn_3 (SimpleRNN)            (None, None, 32)          2080
##  simple_rnn_2 (SimpleRNN)            (None, 32)                2080
## =================================================================
## Total params: 328320 (1.25 MB)
## Trainable params: 328320 (1.25 MB)
## Non-trainable params: 0 (0.00 Byte)
## _________________________________________________________________
```

Now, let's apply a simple RNN to the IMDB movie-review-classification problem.

```
max_features <- 10000
maxlen <- 500

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

```
## 25000 train sequences
```

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

```
## 25000 test sequences
```

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

```
## x_train shape: 25000 500
```

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

```
## x_test shape: 25000 500
```

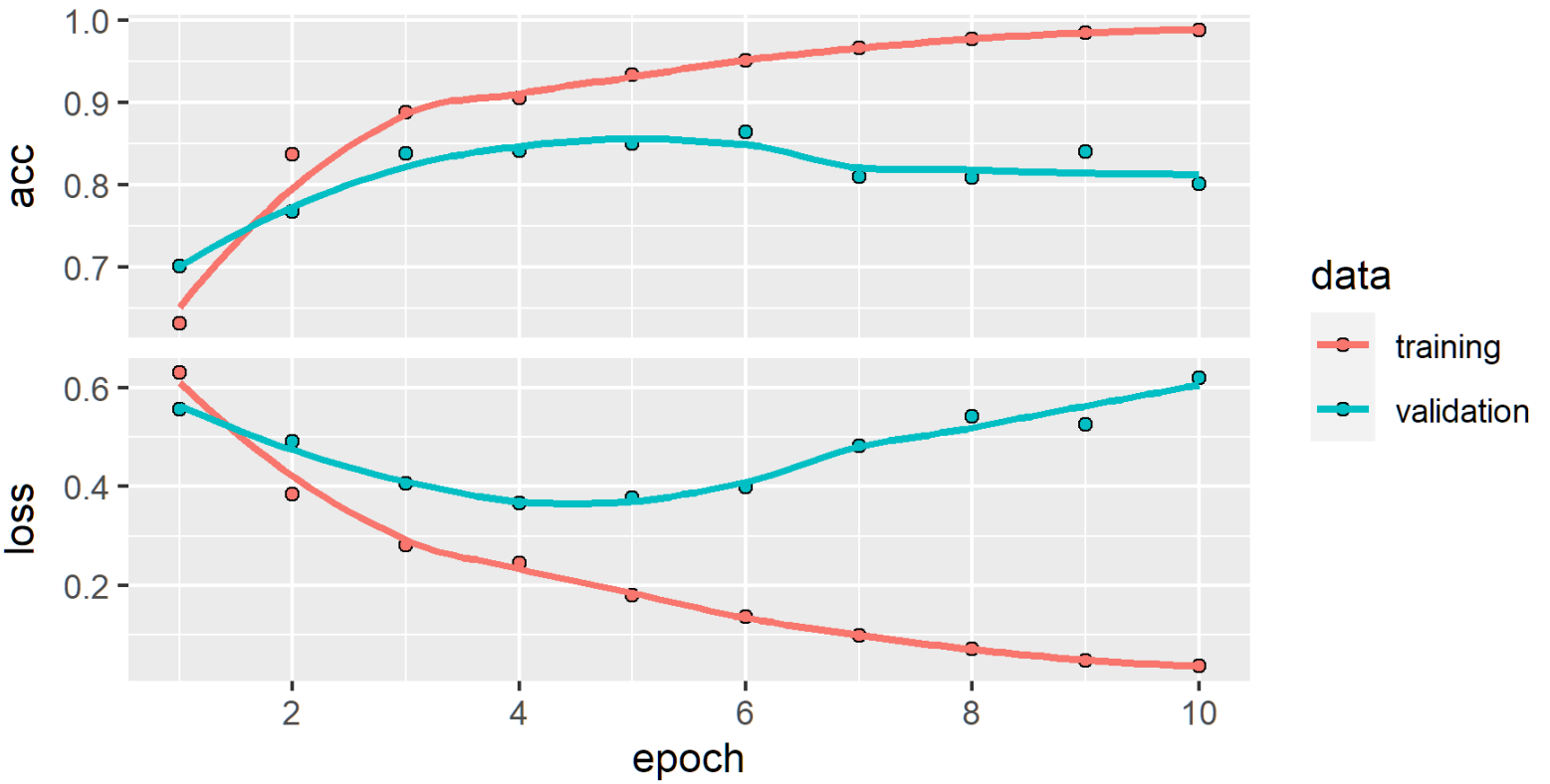
Let's train a simple recurrent network using `layer_embedding` and `layer_simple_rnn`.

```
model <- keras_model_sequential() |>
  layer_embedding(input_dim = max_features, output_dim = 32) |>
  layer_simple_rnn(units = 32) |>
  layer_dense(units = 1, activation = "sigmoid")

model |> compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history <- model |> fit(
  x_train, y_train,
  epochs = 10,
  batch_size = 32,
  validation_split = 0.2
)
```

Alternative, you can skip training and load my pretrainied model/history:

```
model_dir = "data/deeplearning-pretrained/"
model <- load_model_hdf5(str_c(model_dir, "imdb_simple_rnn.h5"))
history <- readRDS(str_c(model_dir,"imdb_simple_rnn.rds"))
```

```
plot(history)
```



The problem of long-term dependencies

One of the appeals of RNNs is the idea that they have short term memory for the information within a sentence.

Sometimes, we only need to look at recent information to perform the present task. In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information. For example,

> the clouds are in the *sky*

But there are also cases where we need more context. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large. As that gap grows, RNNs become unable to learn to connect the information. For example,

> I grew up in France. ... I speak fluent *French*

Beyond simple RNNs

Simple RNNs aren't the only recurrent layers available in Keras.

There are two others: `layer_lstm` and `layer_gru`.

- Long Short-Term Memory (LSTM)
- Gated Recurrent Units (GRU)

In practice, we'll always use one of these, because `layer_simple_rnn` is generally too simplistic to be of real use.

Keep in mind that these advanced layers will take *much* longer time to train.

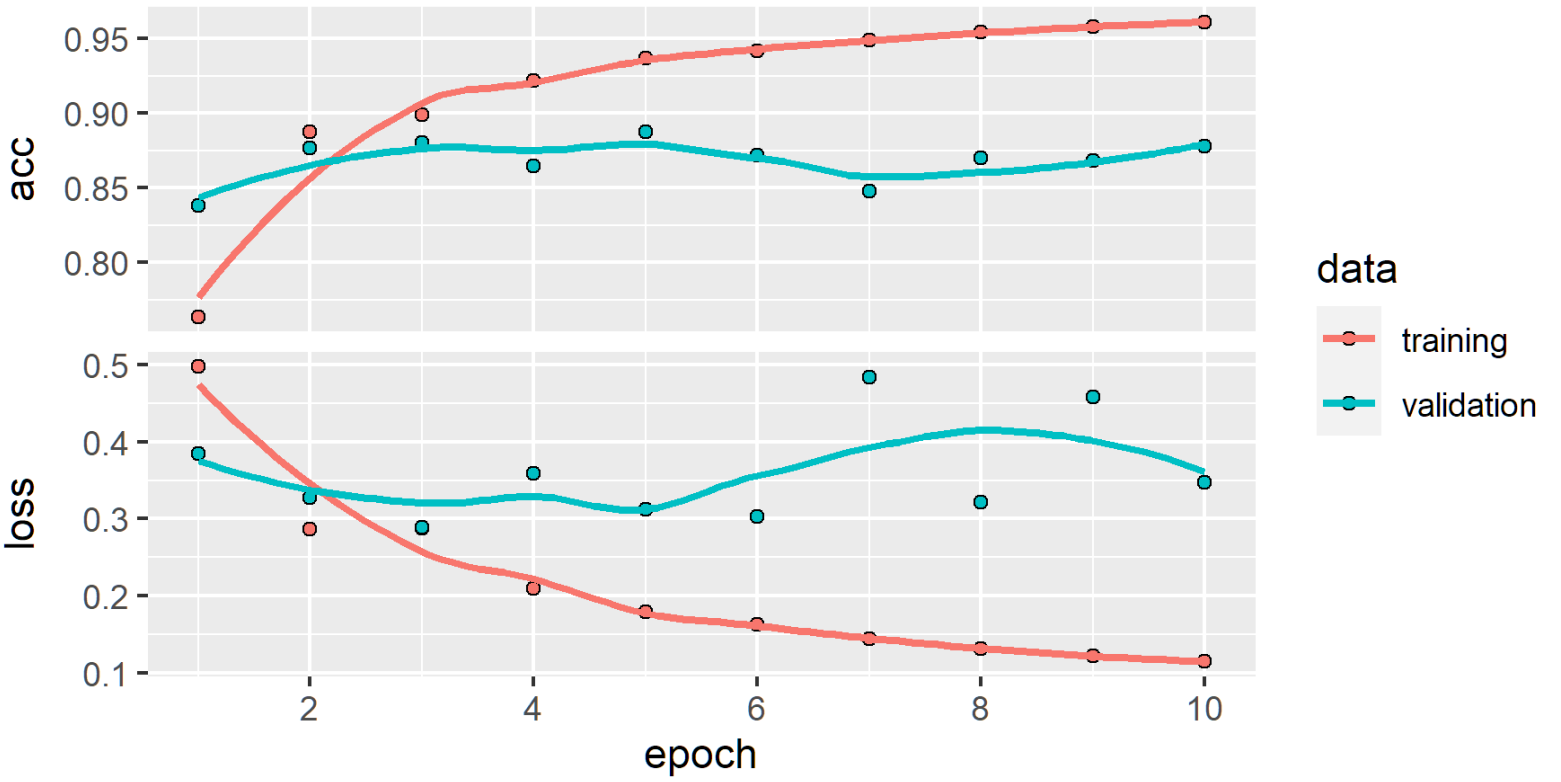
A concrete LSTM example in Keras

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

model |> compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history <- model |> fit(
  x_train, y_train,
  epochs = 10,
  batch_size = 32,
  validation_split = 0.2
)
```

Alternative, you can skip training and load my pretrainied model/history:

```
model_dir = "data/deeplearning-pretrained/"
model <- load_model_hdf5(str_c(model_dir, "imdb_lstm.h5"))
history <- readRDS(str_c(model_dir,"imdb_lstm.rds"))
```

```
plot(history)
```



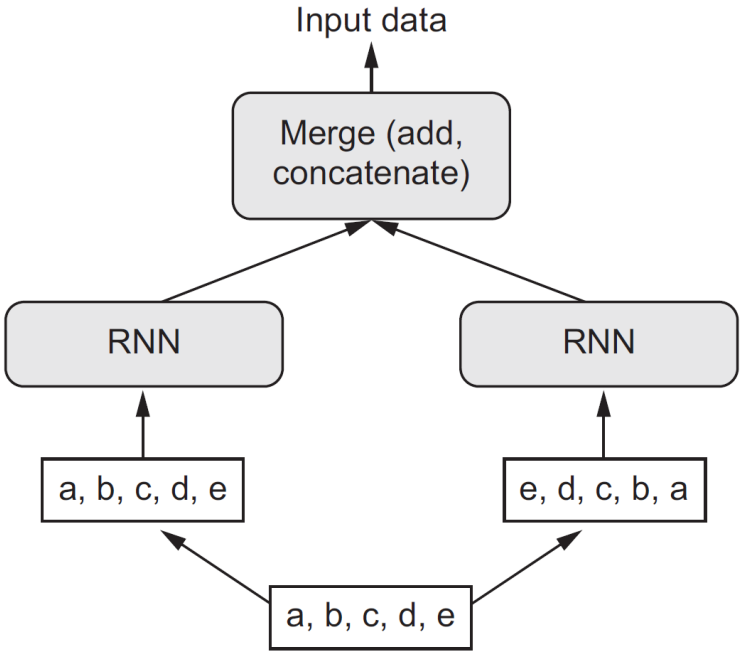
We can see that the accuracy is better than for simple RNNs (from ~0.82 to ~0.875).

Bidirectional RNNs

A bidirectional RNN is a common RNN variant that can offer greater performance than a regular RNN on certain tasks. It's frequently used in NLP.

A bidirectional RNN exploits the order sensitivity of RNNs: it consists of using two regular RNNs, such as `layer_lstm`, each of which processes the input sequence in one direction (forward and backward), and then merging their representations.

By processing a sequence both ways, a bidirectional RNN can catch patterns that may be overlooked by a unidirectional RNN.



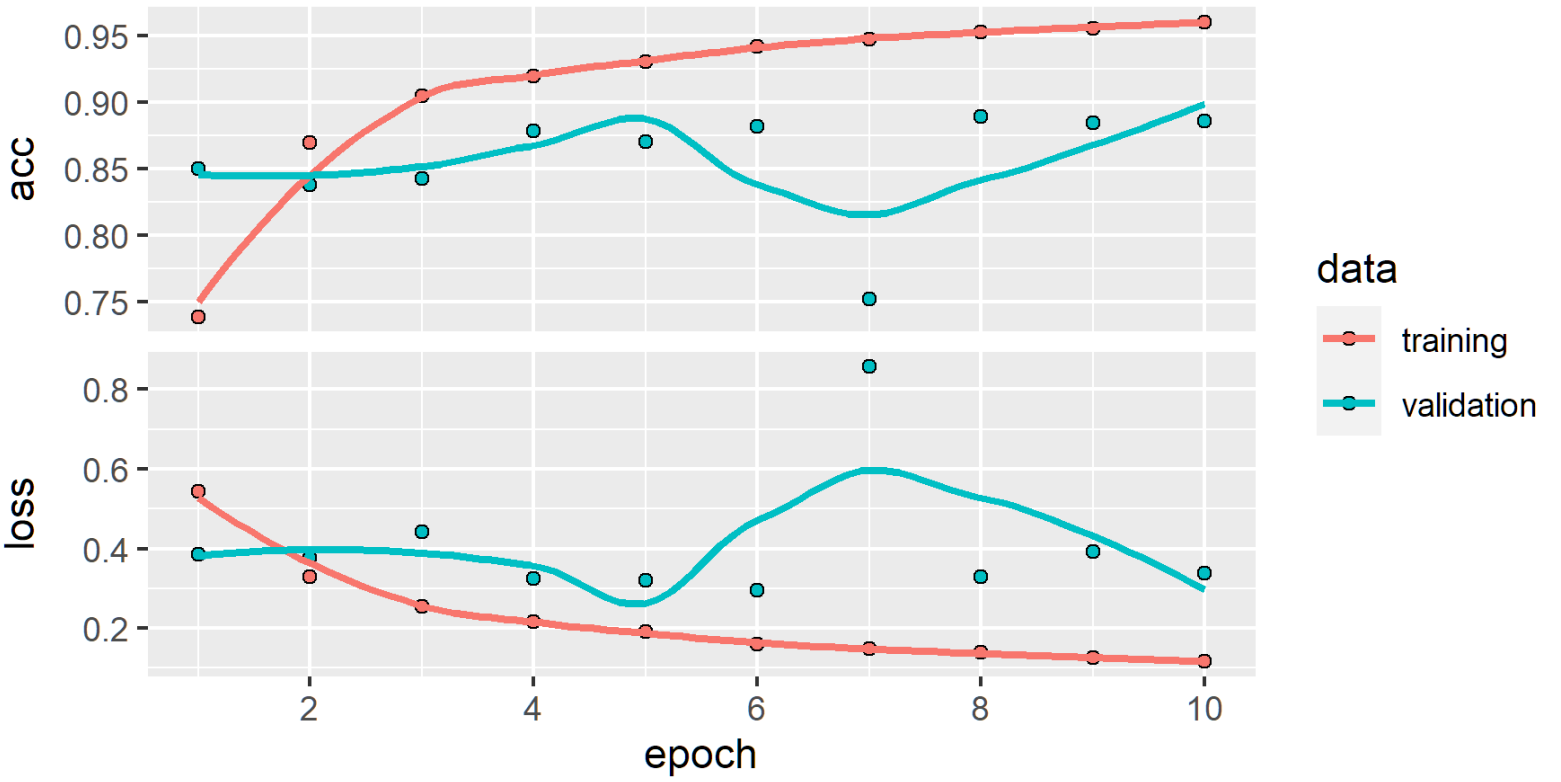
Training and evaluating a bidirectional LSTM

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

model |> compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history <- model |> fit(
  x_train, y_train,
  epochs = 10,
  batch_size = 32,
  validation_split = 0.2
)
```

Alternative, you can skip training and load my pretrainied model/history:

```
model_dir = "data/deeplearning-pretrained/"
model <- load_model_hdf5(str_c(model_dir, "imdb_bidirectional_lstm.h5"))
history <- readRDS(str_c(model_dir,"imdb_bidirectional_lstm.rds"))
```

```
plot(history)
```



We can see that the accuracy is slightly better than it was for our uni-directional LSTM (from ~0.875 to ~0.9).

Your turn

Load the following pre-trained models:

- Simple RNN
- LSTM
- Bidirectional LSTM

Evaluate the models using the testing data and find out their accuracy. Which one is the best?

- See the last few slides from `session-6.1-work-with-text` on how to evaluate a model on the test data.