Deep Learning for sequence data (2)

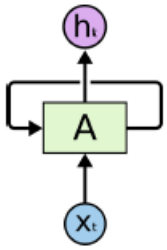
A recurrent layer in Keras

Humans don't start their thinking from scratch every second. As you read this document, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence.

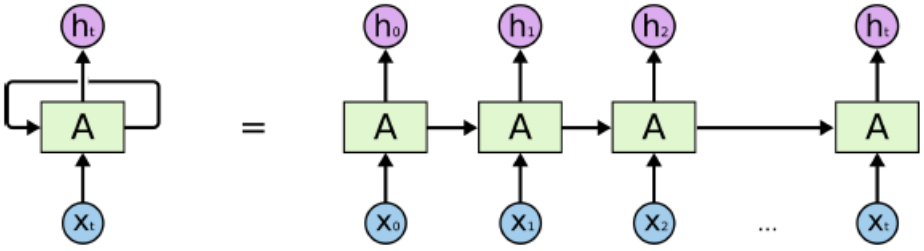
Traditional neural networks can't do this, and it seems like a major shortcoming. For example, imagine you want to classify what kind of event is happening at every point in a movie. It's unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.

RNN

Recurrent neural networks (RNN) address this issue. They are networks with loops in them, allowing information to persist.



In the above diagram, a chunk of neural network, looks at some input and outputs a value. A loop allows information to be passed from one step of the network to the next. These loops make recurrent neural networks seem kind of mysterious. However, if you think a bit more, it turns out that they aren't all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:



An unrolled recurrent neural network.

This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They're the natural architecture of neural network to use for such data. And they certainly are used! In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioning... A recurrent neural network (RNN) processes sequences by iterating through the sequence elements and maintaining a state containing information relative to what

it has seen so far. In effect, an RNN is a type of neural network that has an internal loop. The state of the RNN is reset between processing two different, independent sequences (such as two different IMDB reviews), so you still consider one sequence a single data point: a single input to the network. What changes is that this data point is no longer processed in a single step; rather, the network internally loops over sequence elements.

A recurrent layer has the special property of being able to process sequential input data $[x_1, ..., x_n]$. It consist of a cell that updates the hidden state, h_t as each element of the sequence x_t is passed through it, one timestep at a time. The hidden state is a vector with length equal to the number of `units` in the cell -it can be thought of a cell's current understanding of the sequence. At timestep t the cell uses the previous value of the hidden state h_{t-1} together with the data from current timestep x_t to produce an updated hidden state vector h_t. This recurrent process continues until the end of the sequence. Once the sequence is finished, the layer outputs the final hidden state of the cell, h_n, which is then passed on the next layer of the network.

To explain this in more detail, let's unroll the process so that we can see exactly how a single sequence is fed through the layer

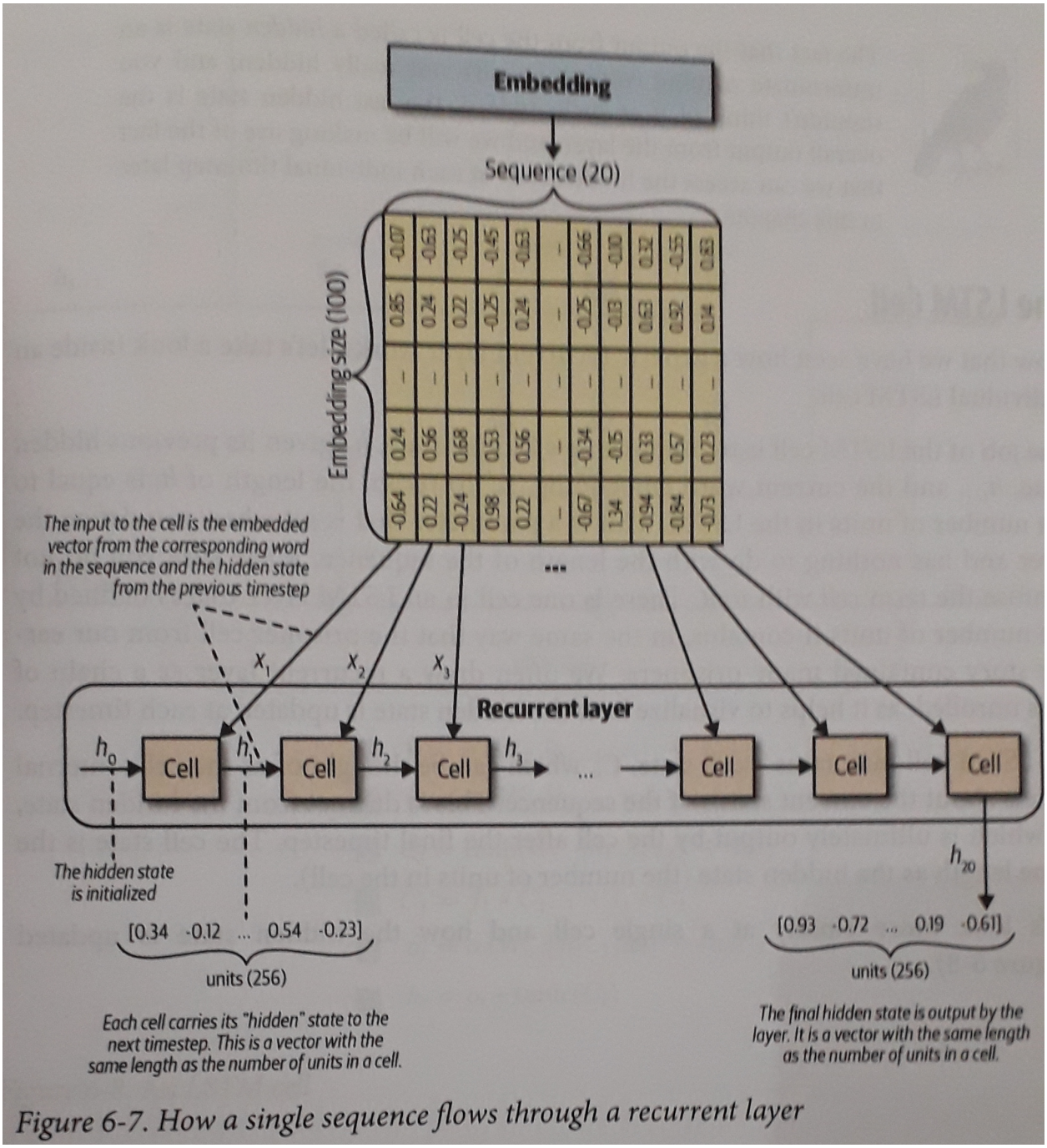
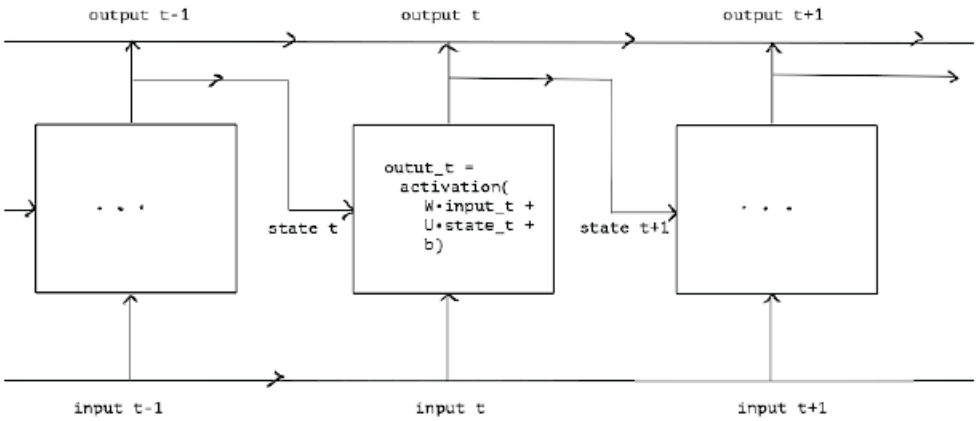


Figure 6-7. How a single sequence flows through a recurrent layer

Here, we represent the recurrent process by drawing a copy of the cell at each timestep and show how the hidden state is constantly being updated as it flows through the cells. We can clearly see how the previous hidden state is blended with the current sequential data point (i.e., the current embedded word vector) to produce the next hidden state. The output from the layer is the final hidden state of the cell. after each word in the input sequence has been processed. It's important to remember that all of the cells in this diagram share the same weights (as they are really the same cell).

The fact that the output from the cell is called hidden state is an unfortunate naming convention. Indeed, the last hidden state is the overall output from the layer, but we can access the hidden state at each individual timestep.

To recap, the length of the hidden state h_t is equal to the number of units in the recurrent layer. This is the parameter that is set when you define the layer and has nothing to do with the length of the sequence. Make sure you do not confuse the term cell with unit. There is one cell in a recurrent layer that is defined by the number of units it contains. We often draw a recurrent layer as a chain of cell unrolled, as it hepls to visualize how the hidden state is updated at each timestep.



The hidden state h_t at time t is given by

$$h_t = \tanh(Wx_t + Uh_{t-1} + b)$$

and the output vector y_t updates by

$$y_t = h_t$$

here, h_t and h_{t-1} are the hidden states from the time t and $t-1$. x_t is the input at time t and y_t is the output at time t. The important thing to notice is that there are two weight matrices U $h \times h$ and W $h \times x$ and one bias term b $h - dim$. Each of these matrices can be thought of as an internal 1 layer neural network with output size as defined in the parameter units, also bias has the same size. y_t is raw h_t and we don't apply another weight matrix here, as suggested by many articles. This represents one individual cell of RNN, and sequential combination of cells (count equal to time-steps in data) creates the complete RNN layer. Remember the same weight matrices and bias are shared across the RNN cells. Finally, we can compute the number of parameters required to train the RNN layer as follows,

$$(units \times input) + (units \times units) + (units)$$

Each timestep t in the output tensor contains information about timesteps 1 to t in the input sequence—about the entire past. For this reason, in many cases, you don't need this full sequence of outputs; you just need the last output (y_t at the end of the loop), because it already contains information about the entire sequence.

```
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
```

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

summary(model)
```

```
## Model: "sequential"
## _________________________________________________________________________________
## Layer (type)                         Output Shape                      Param #
## =================================================================================
## embedding (Embedding)                (None, None, 32)                  320000
## _________________________________________________________________________________
## simple_rnn (SimpleRNN)               (None, 28)                        1708
## _________________________________________________________________________________
## dense (Dense)                        (None, 1)                         29
## =================================================================================
## Total params: 321,737
## Trainable params: 321,737
## Non-trainable params: 0
## _________________________________________________________________________________
```

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

```
history <- model %>% fit(
  input_train, y_train,
  epochs = 10,
  batch_size = 128,
  validation_split = 0.2
)
```

```
plot(history)
```

```
## `geom_smooth()` using formula 'y ~ x'
```