Brief Introduction to Recurrent Neural Networks

In the past chapter we have looked at **Convolutional Neural Networks** (CNN). Another network architecture that is widely used (for example in natural language processing) is the recurrent one (networks with this architecture are called recurrent neural networks, or RNN for short). This chapter is an extremely short and superficial description of how RNNs work, with one small application that should let you better understand their inner working. RNNs would require one (or multiple) books and the goal of this chapter is to give you a very basic understanding on how they work. I think it is anyway useful, for any machine learning engineer, to have at least an intuitive understanding of how RNNs work. That is the reason for this chapter. I tried really hard to only discuss the very basic components of RNNs to make you see the very fundamental aspects of how they work. I hope you find it useful. At the end of the chapter, I suggest further readings in case you find the subject interesting and you want to better understand RNNs.

# Introduction to RNNs

RNN are very different from CNNs, and typically used when dealing with sequential information, in other words for data for which the order matters. The typical example given is a series of words in a sentence. You can easily understand how the order of words in a sentence can make a big difference. For example, saying "the man eats the rabbit" has a different meaning than "the rabbit eats the man", the only difference being the order of the words, and who gets eaten by whom. You can use RNNs to predict for example the next word in a sentence. Take for example the sentence: "Paris is the capital of", it is easy to complete the sentence with "France", that means that there is information about the final word of the sentence encoded in the previous words, and that information is what RNNs wants to exploit to predict the next terms in the sequence. The name recurrent comes from how they work: the network applies the same operation on each element of the sequence, accumulating information about the previous terms. To summarize:

* RNNs make use of sequential data and uses the information encoded in the order of the terms in the sequence
* RNNs apply the same kind of operation to all terms in the sequence and build a memory of the previous terms in the sequence to predict the next term

Before trying to understand a bit better how they work, let us consider a few important use cases where they can be applied to give you a feeling of the range of applications possible.

* Generating Text: predicting probability of words given previous set of words. For example, you can easily generate text that looks like Shakespeare with RNNs, as A. Karpathy has done in his blog [2]
* Translation: given a set of words in a language you want words in a different language
* Speech recognition: given a series of audio signals (words), we want to predict the sequence of letters forming the words spoken
* Generating image labels: with CNNs, RNNs can be used to generate labels for images. Check the paper "Deep Visual-Semantic Alignments for Generating Image Descriptions" by A. Karpathy on the subject [3]. Be aware that this is a rather advanced paper that requires quite some mathematical background.
* Chatbots: sequence of words given as input, RNNs try to generate answers to the input

As you can imagine, to solve those problems you will need sophisticated architectures that are not easy to describe in a few sentences and that require a deeper (pun intended) understanding of how RNNs are working, things that would go beyond the scope of this chapter and book.

## Notation

Let us consider the sequence: "Paris is the capital of France". This sentence will be fed to a RNN one word at a time: first "Paris", then "is", then "the" and so on. We will have

* "Paris" will be the first word of the sequence: w1 = 'Paris'
* "is" will be the second word of the sequence: w2 = 'is'
* "the" will be the third word of the sequence: w3 = 'the'
* "capital" will be the fourth word of the sequence: w4 = 'capital'
* "of" will be the fifth word of the sequence: w5 = 'of'
* "France" will be the sixth word of the sequence: w6 = 'France'

The words will be fed into the RNN in the following order: w1, w2, w3, w4, w5 and then w6. The different words will be processed by the network one after the other, or as someone like to say at different time points. Usually is said that if word w1 is processed at time , then w2 is processed at time , w3 at time and so on. The time is not related to a real time but is meant to suggest the fact that each element in the sequence is processed sequentially and not in parallel. The time is also not related to computing time or anything related to it. And the increment of in does not have any meaning, it simply means that we are talking about the next element in our sequence. You may find the following notations when reading papers, blogs or books:

* : the input at time . For example, w1 could be the input at time , w2 at time and so on.
* : is the notation with which the internal memory, that we have not defined yet, at time is indicated. This quantity will contain the accumulated information on the previous terms in the sequence we discussed above. An intuitive understanding of it will have to suffice, since a more mathematical definition would require a too detailed explanation.
* is the output of the network at time , or in other words after all the elements of the sequence until , including the element has been fed to the network.

## Basic idea of RNNs

Typically, a RNN is indicated in the literature as the leftmost part of Figure 8-1. The notation used is indicative and has the goal of simply indicating the different elements of the network: the inputs, the internal memory, one set of weights and another set of weights. In reality, this schematic representation is simply a way of depicting the real structure of the network that you can see on the right part of Figure 8-1, sometime called the unfolded version of the network.

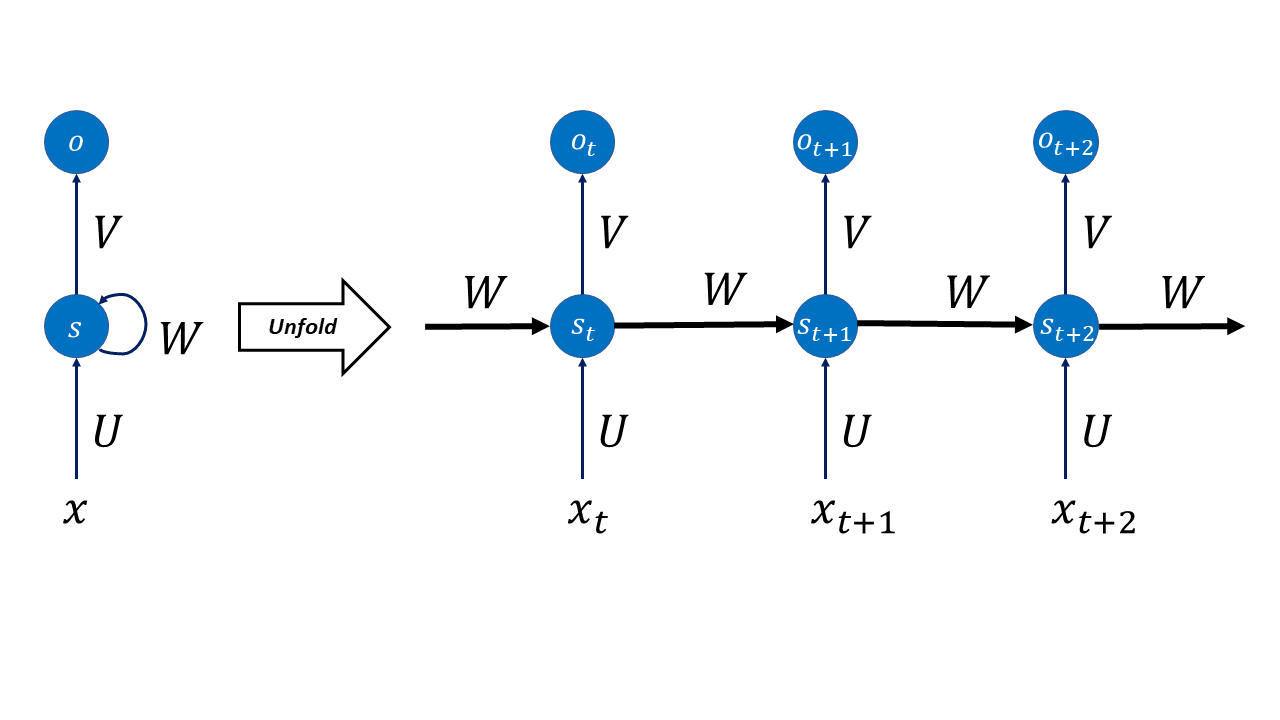


Figure 8-1. A schematic representation of a RNN

The right part of Figure 8-1 should be read left to right. The first neuron in the Figure does its evaluation at an indicative time and produce an output and creates an internal memory state . The second neuron, that does its evaluation at a time , after the first neuron, gets as input both the next element in the sequence and the previous memory state . The second neuron then generates an output and a new internal memory state . The third neuron then (the one at the extreme right of Figure 8-1) gets as input the new element of the sequence and the previous internal memory state and the process proceeds in this way for a finite number of neurons. You can see in Figure 8-1 that there are two sets of weights: and . One set (here indicated with ) is used for the internal memory states and one for the sequence element. Typically, each neuron will generate the new internal memory state with a formula that will look like this

where we have indicated with one of the activation functions we have already seen as ReLU or tanh. Additionally, the previous formula will be of course multi-dimensional. can be understood as the memory of the network at time . The number of neurons (or time steps) that can be used is a new hyperparameter that needs to be tuned, depending on the problem. Research has shown that when this number is too big, the network has big problems during training.

Something very important to note is that at each time step the weights are not changing. We are performing the same operation at each step, simply changing the inputs every time we perform an evaluation. Additionally, In Figure 8-1 we have for every step an output in the diagram (, and ) but typically this is not necessary. In our example where we want to predict the final word in a sentence, we may just need the final output.

## Why the name recurrent

We need to discuss very briefly why the networks are called recurrent. We have said that the internal memory state at a time is given by

The internal memory state at time is evaluated using the same memory state at time , the one at time with the value at time and so on. This is at the origin of the name recurrent.

## Learning to count

To give you an idea of the power of such networks I would like to give you a very basic example of something RNN are very good at, and standard fully connected networks, as the one we saw in the previous chapters, are really bad at. Let us try to teach a network to count. The problem we want to solve is the following: given a certain vector, that we will take being made of 15 elements, containing just 0 and 1 we want to build a neural network that is able to count the amount of 1s we have in the vector. This is a difficult problem for a standard network, but why? To understand intuitively why let us consider the problem we have analyzed of distinguishing the one and two digits in the MNIST dataset. In that case the learning happens because the ones and the twos have black pixels in fundamentally different positions. A digit one will always differ in (at least in the MNIST dataset) the same way from the digit two, so the network will identify those differences and as soon as they are detected a clear identification can be made. In our case this is no more possible. Consider for example a simpler case of a vector with just 5 elements. Consider the case when a one appears exactly one time. We have 5 possible cases: [1,0,0,0,0], [0,1,0,0,0], [0,0,1,0,0], [0,0,0,1,0] and [0,0,0,0,1]. There is no discernable pattern to be detected here. There is no easy weight configuration that could cover those cases at the same time. In case of an image this problem is similar to the problem of detecting the position of a black square in a white image. We can build a network in tensorflow and check how good such networks are. Due to the introductory nature of the chapter, time will not be spent in dealing with hyperparameter discussion, metric analysis and so on. We will simply look at a basic network that can count.

Let us start with creating our vectors. We will create vectors that we will split in training and dev sets.

import numpy as np

import tensorflow as tf

from random import shuffle

from tensorflow import keras

from tensorflow.keras import layers

now let us create our list of vectors. The code is a slightly more complicated, and we will look at it in a bit more details.

nn = 15

ll = 2\*\*15

train\_input = ['{0:015b}'.format(i) for i in range(ll)]

# consider every number up to 2^15 in binary format

shuffle(train\_input) # shuffle inputs

train\_input = [map(int, i) for i in train\_input]

ti = []

for i in train\_input:

temp\_list = []

for j in i:

temp\_list.append([j])

ti.append(np.array(temp\_list))

train\_input = ti

We want to have all possible combinations of 1 and 0 in vectors of 15 elements. So, an easy way to do that is take all numbers up to in binary format. To understand why, let us suppose we want to do this with only 4 elements: we want all possible combinations of four 0 and 1. Consider all number up to in binary, that you can get with this code

['{0:04b}'.format(i) for i in range(2\*\*4)]

The code simply format all numbers that you get with the range(2\*\*4) function from 0 to 2\*\*4 in binary format with {0:04b}, limiting the number of digits to 4. The result is the following

['0000',

'0001',

'0010',

'0011',

'0100',

'0101',

'0110',

'0111',

'1000',

'1001',

'1010',

'1011',

'1100',

'1101',

'1110',

'1111']

As you can easily verify you have all possible combinations in the list. You have all possible combinations of the one appearing one times ([0001], [0010], [0100] and [1000]), of the ones appearing two times, and so on. For our example we will simply do it with 15 digits, that means we will do it with numbers up to . The rest of the code above is there to simply transform a string like '0100' in a list [0,1,0,0] and then concatenate all the lists with all the possible combinations. If you check the dimension of the output array you will notice that you get . Each observation is an array of dimensions . Then we prepare the target variable, a one-hot encoded version of the counts. That means that if we have an input with four 1s in the vector our target vector will look like [0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0]. As expected, the train\_output array will have the dimensions . Now let us split our set in a train and a dev set, as we have done now several times. We will do it here in a dumb way

NUM\_EXAMPLES = ll - 2000

test\_input = train\_input[NUM\_EXAMPLES:]

test\_output = train\_output[NUM\_EXAMPLES:] # everything beyond 10,000

train\_input = train\_input[:NUM\_EXAMPLES]

train\_output = train\_output[:NUM\_EXAMPLES] # till 10,000

remember that this will work since we have shuffled the vectors at the beginning, so we should have a random distribution of cases. We will use 2000 cases for the dev set and the rest (roughly 30000) for the training. The train\_input will have dimensions and the dev\_input will have dimensions .

Now you can build a network with this code, and you should be able to understand almost all of it by now

model = keras.Sequential()

model.add(layers.Embedding(input\_dim = 15, output\_dim = 15))

# Add a LSTM layer with 128 internal units.

model.add(layers.LSTM(24, input\_dim = 15))

# Add a Dense layer with 10 units.

model.add(layers.Dense(16, activation = 'softmax'))

model.compile(loss = 'categorical\_crossentropy', optimizer = 'adam', metrics = ['categorical\_accuracy'])

Then let us train the network

# we need to convert the input and output to numpy array to be used by the network

train\_input = np.array(train\_input)

train\_output = np.array(train\_output)

test\_input = np.array(test\_input)

test\_output = np.array(test\_output)

model.fit(train\_input, train\_output, validation\_data = (test\_input, test\_output), epochs = 10, batch\_size = 100)

For performance reason and to let you realize how efficient RNNs are we are using here a LSTM kind of neuron. This has a special way of calculating the internal state. A discussion would go well beyond the scope of the book. For the moment you should focus on the results and not on the code itself. If you let the code run, you will get the following result

Epoch 1/10

308/308 [==============================] - 4s 9ms/step - loss: 1.9441 - categorical\_accuracy: 0.3063 - val\_loss: 1.1784 - val\_categorical\_accuracy: 0.6840

Epoch 2/10

308/308 [==============================] - 2s 7ms/step - loss: 0.7472 - categorical\_accuracy: 0.8332 - val\_loss: 0.4515 - val\_categorical\_accuracy: 0.9270

Epoch 3/10

308/308 [==============================] - 2s 7ms/step - loss: 0.3311 - categorical\_accuracy: 0.9554 - val\_loss: 0.2360 - val\_categorical\_accuracy: 0.9630

Epoch 4/10

308/308 [==============================] - 2s 7ms/step - loss: 0.1921 - categorical\_accuracy: 0.9658 - val\_loss: 0.1530 - val\_categorical\_accuracy: 0.9675

Epoch 5/10

308/308 [==============================] - 2s 7ms/step - loss: 0.1306 - categorical\_accuracy: 0.9760 - val\_loss: 0.1071 - val\_categorical\_accuracy: 0.9775

Epoch 6/10

308/308 [==============================] - 2s 7ms/step - loss: 0.0937 - categorical\_accuracy: 0.9824 - val\_loss: 0.0778 - val\_categorical\_accuracy: 0.9870

Epoch 7/10

308/308 [==============================] - 2s 7ms/step - loss: 0.0696 - categorical\_accuracy: 0.9905 - val\_loss: 0.0586 - val\_categorical\_accuracy: 0.9930

Epoch 8/10

308/308 [==============================] - 2s 7ms/step - loss: 0.0533 - categorical\_accuracy: 0.9921 - val\_loss: 0.0446 - val\_categorical\_accuracy: 0.9945

Epoch 9/10

308/308 [==============================] - 2s 7ms/step - loss: 0.0422 - categorical\_accuracy: 0.9924 - val\_loss: 0.0367 - val\_categorical\_accuracy: 0.9960

Epoch 10/10

308/308 [==============================] - 2s 7ms/step - loss: 0.0346 - categorical\_accuracy: 0.9943 - val\_loss: 0.0301 - val\_categorical\_accuracy: 0.9955

<tensorflow.python.keras.callbacks.History at 0x7f6b7b3bd990>

After just 10 epochs the network is right in 99% of the cases. Just let it run for more epochs and you can reach incredible precision. An instructive exercise is trying to train a fully connected network (as the ones we have discussed so far) to count. You will see how this is not possible.

# Conclusions

This was a very brief description of RNNs. You should have developed an intuition on how they work and how LSTM neurons are structured. There is a lot more about RNNs to discuss, but that would go way beyond the scope of this book and therefore I have chosen to neglect it here. RNNs are an advanced topic and require a bit more know-how to understand. In the next section I list two sources that are free in internet that you can use to kick-start your learning on RNNs.

# Further Readings

You found the chapter intriguing and would like to learn more about RNNs? No problem. There is a huge amount of material that you can find in internet. Here are two good sources:

* A much more complete and advanced treatment of RNNs can be found at <https://www.deeplearningbook.org/contents/rnn.html>. Be aware that this is more advanced and require a much more advanced mathematics background.
* This review paper is full of information and further references that you can track down and read: <https://arxiv.org/pdf/1808.03314.pdf>