Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

09 - Recurrent Neural Networks

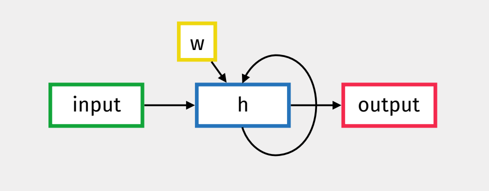
François Pitié

Ussher Assistant Professor in Media Signal Processing
Department of Electronic & Electrical Engineering, Trinity College Dublin

[4C16] Deep Learning and its Applications — 2019/2020

Recurrent Neural Networks (RNN) are special type of neural architectures designed to be used on **sequential data**.

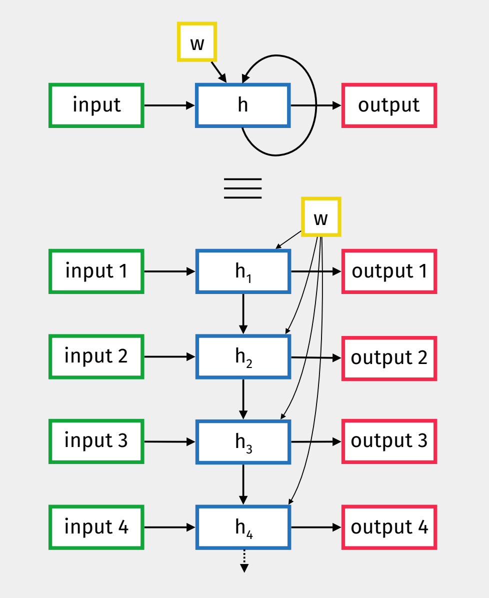
Sequential data can be found in any time series such as audio signal, stock market prices, vehicle trajectory but also in natural language processing (text). In fact, RNNs have been particularly successful with Machine Translation tasks.



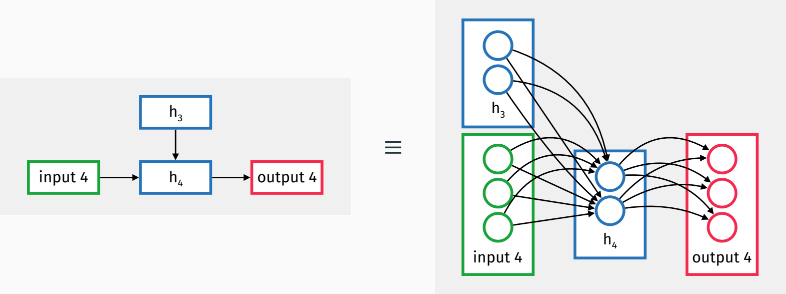
Recurrent Networks define a recursive evaluation of a function. The input stream feeds a context layer (denoted by h in the diagram). The context layer then re-use the previously computed context values to compute the output values.

The best analogy in signal processing would be to say that if convolutional layers where similar to FIR filters, RNNs are similar to IIR filters.

In the next slide, the RNN is unfolded to produce a classic feedforward neural net.


w
input
h
output
≡
w
input 1
h1
output 1
input 2
h2
output 2
input 3
h3
output 3
input 4
h4
output 4

A key aspect of RNNs is that the network parameters w are shared across all the iterations. That is w is fixed in time.



In its simplest form, the inner structure of the hidden layer block is simply a dense layer of neurons with $\tanh$ activation. This is called a simple RNN architecture or Elman network.

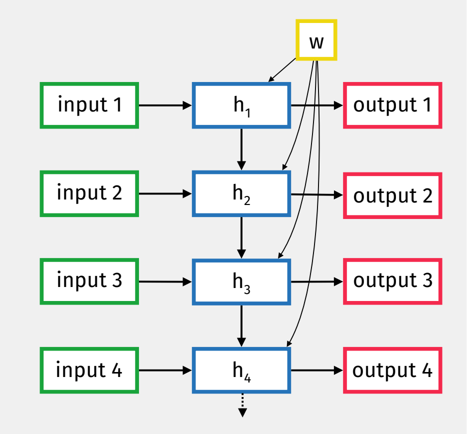
We usually take a $\tanh$ activation as it can produce positive or negative values, allowing for increases and decreases of the state values. Also $\tanh$ bounds the state values between -1 and 1, and thus avoids a potential explosion of the state values.

The equations for this network are as follows:

$$\mathbf{h}_t = \tanh(\mathbf{W}_h \mathbf{x}_t + \mathbf{U}_h \mathbf{h}_{t-1} + \mathbf{b}_h)$$
$$\mathbf{y}_t = \sigma_y(\mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y)$$

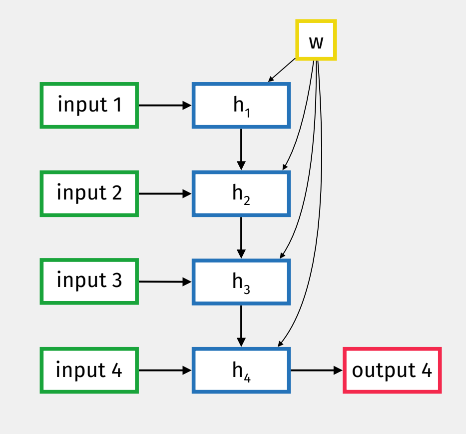
where $\mathbf{x}$ is the input vector, $\mathbf{h}$ the vector of the hidden layer states, $\mathbf{x}$ is the output vector, σ_y is the output's activation function, $\mathbf{W}_h$ and $\mathbf{b}_h$ the matrix stacking the parameters for h, $\mathbf{U}_h$ the matrix stacking the feedback parameters for h and $\mathbf{W}_y$ and $\mathbf{b}_y$ the matrix and vector stacking the parameters for the output.

The parameters $\mathbf{W}_h$, $\mathbf{W}_y$, $\mathbf{b}_h$, $\mathbf{b}_y$ are shared by all input vectors x_t.



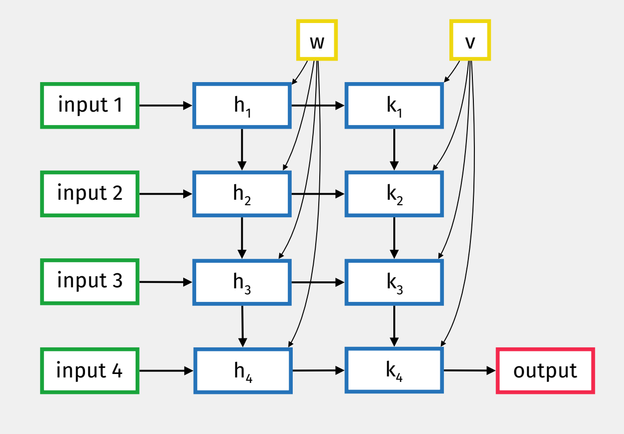
In Keras, we can define a simple RNN layer as follows:

```
input = Input(shape=(n, p))
h = SimpleRNN(hsize, return_sequences=True)(input)
output = Dense(osize, Activation='softmax')(h)
```



Note that we can choose to produce a single output for the entire sequence instead of an output at each timestamp. In Keras, this would be defined as:

```
input = Input(shape=(n, p))
h = SimpleRNN(hs, return_sequences=False)(input)
output = Dense(os, Activation='softmax')(h)
```



And we can stack multiple RNN layers. For instance:

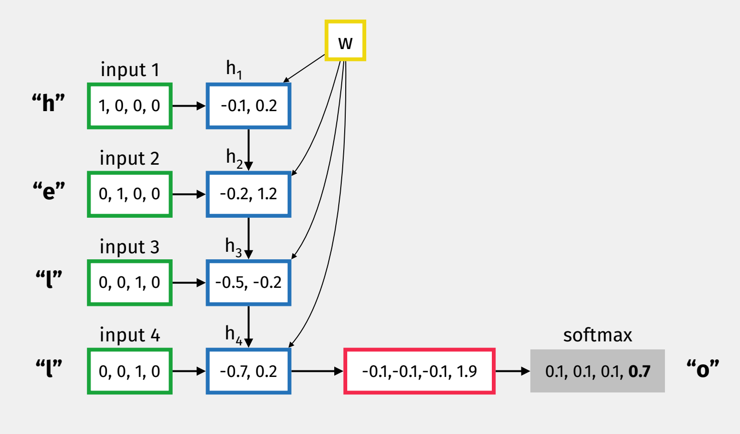
```
input = Input(shape=(n, p))
h = SimpleRNN(hs, return_sequences=True)(input)
k = SimpleRNN(ks, return_sequences=False)(h)
output = Dense(os, Activation='softmax')(k)
```

Application Example: Character-Level Language Modelling

In the next slide is presented an example application of RNNs where we try to predict next character given a sequence of previous characters. The idea is to give the RNN a large corpus of text to train on and try to model the text inner dynamics (a bit similar to the idea of Word2Vec).

Training. We start from a character one-hot encoding. Each input of the RNNs is a character from the sequence. The RNN then is used for a **classification** task: we try to classify the output of the sequence $\mathbf{x}_1, \cdots, \mathbf{x}_{n-1}$ as the next character $\mathbf{y} = \mathbf{x}_n$.

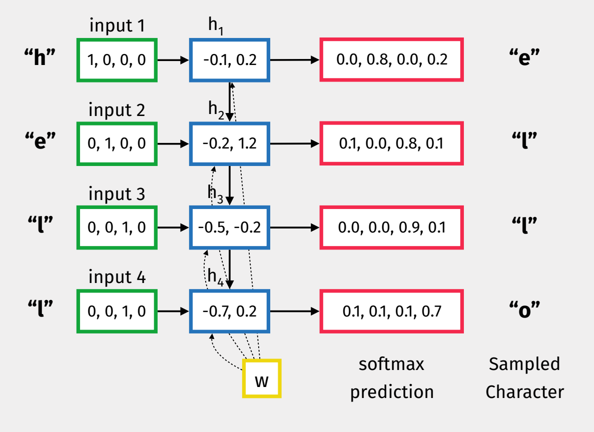
Since we are using cross-entropy and softmax, the network returns back the vector of probability distribution for the next character.



We are training for a classification task: can you predict the next character based on the previous characters?

Once we have trained the RNN, we can then generate whole sentences, one character at a time. We achieve this by providing an initial sentence fragment, or seed. Then we can use our RNN to predict the probability distribution of the next character. To generate the next character, we simply sample the next character based from these probabilities. This character is then appended to the sentence and the process is repeated.

Diagram of the text generation process is illustrated in the next slide.


input 1
input 2
input 3
input 4
h1
h2
h3
h4
“h”
“e”
“l”
“l”
1, 0, 0, 0
0, 1, 0, 0
0, 0, 1, 0
0, 0, 1, 0
-0.1, 0.2
-0.2, 1.2
-0.5, -0.2
-0.7, 0.2
0.0, 0.8, 0.0, 0.2
0.1, 0.0, 0.8, 0.1
0.0, 0.0, 0.9, 0.1
0.1, 0.1, 0.1, 0.7
“e”
“l”
“l”
“o”
w
softmax prediction
Sampled Character

This fun application is taken from this seminal blog post by Karpathy:

http://karpathy.github.io/2015/05/21/rnn-effectiveness/#fun-with-rnns

Check this link for results and more insight about the RNN!

Training

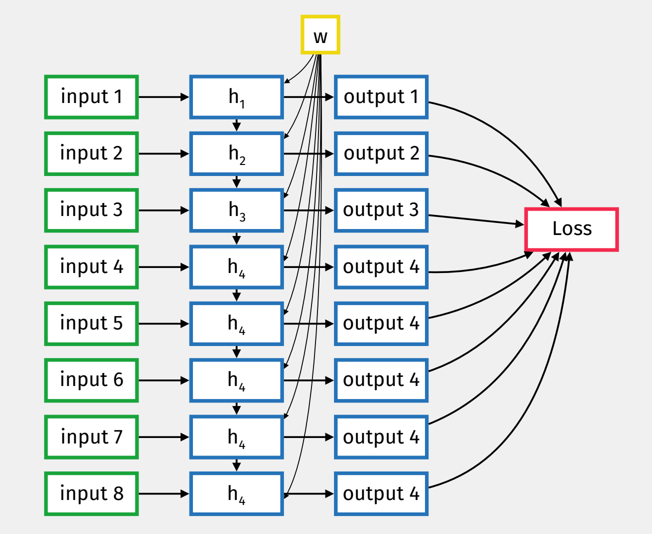
To train a RNN, we can unroll the network to expand it into a standard feedforward network and then apply back-propagation as per usual.

This process is called Back-Propagation Through Time (BPTT).

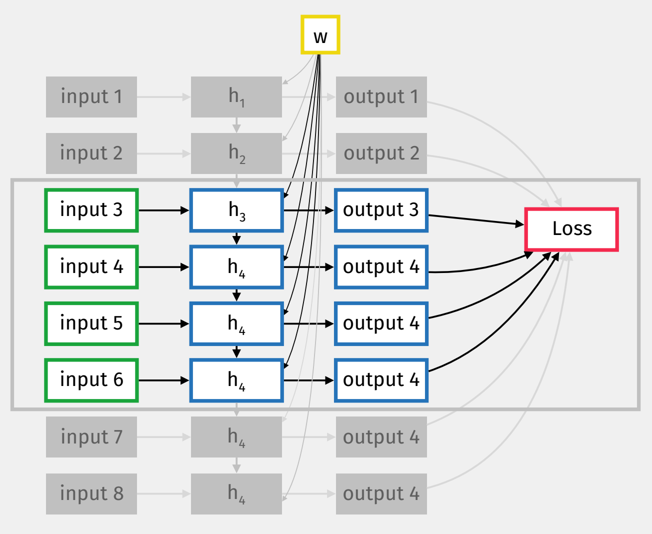
Training

Note that the unrolled network can grow very large and might be hard to fit into the GPU memory. Also, the process is very sequential in nature and it is thus difficult to avail of parallelism.

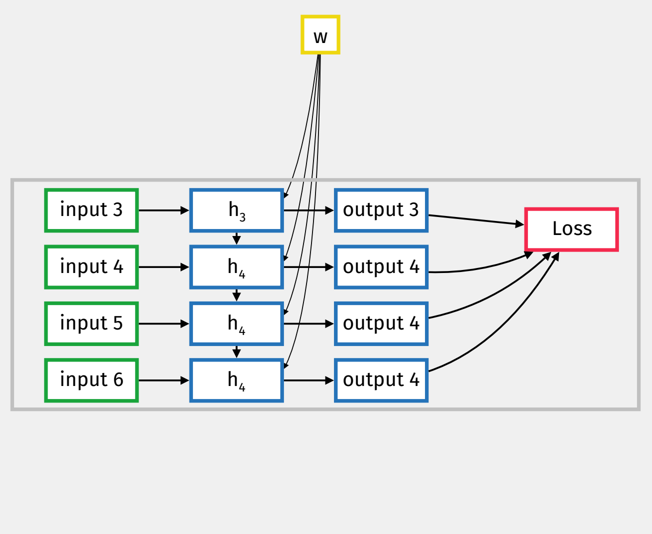
Sometimes, a strategy to speed up learning is to split the sequence into chunks and train apply BPTT on these truncated parts. This process is called **Truncated Back-Propagation Through Time**.



Example of unrolling the RNN with BPTT.



It is possible to split the sequence into chunks.



and train each chunk separately (truncated BPTT)

Training

When unrolled, recurrent networks can grow very deep.

As with any deep network, the main problem with using gradient descent is then that the error gradients can vanish (or explode) exponentially quickly.

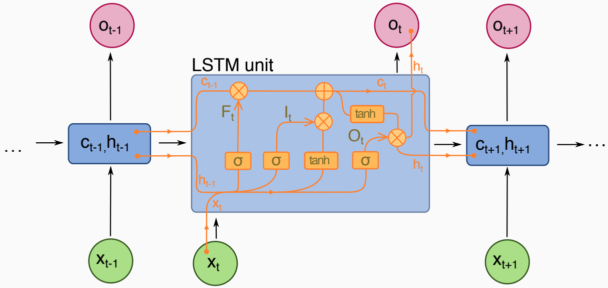
Therefore we rarely use the Simple RNN layer architecture as they are very difficult to train. Instead, we usually resort to two alternative RNN layer architectures: LSTM and GRU.

LSTM (Long Short-Term Memory) was specifically proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber to deal with the exploding and vanishing gradient problem. LSTM blocks are a special type of network that is used for the recurrent hidden layer. LSTM block can be used as a direct replacement for the dense layer structure of simple RNNs.

As of 2017, major technology companies including Google, Apple, and Microsoft are using LSTM in their speech recognition or Machine Translation products.

S. Hochreiter and J. Schmidhuber (1997). "Long short-term memory". [https://goo.gl/hhBNRE]

`Keras: https://keras.io/layers/recurrent/#lstm`
See also Brandon's Rohrer's video: `https://youtu.be/WCUNPb-5EYI`
and colah's blog [https://goo.gl/uc7gbn]



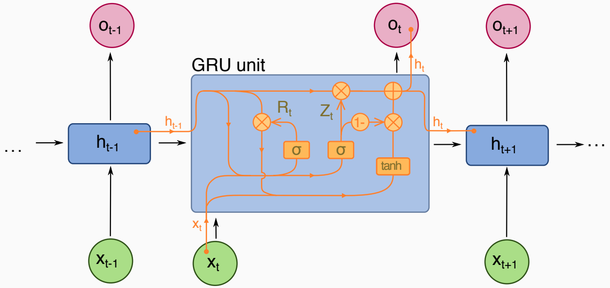
Architecture of LSTM Cell. *[Figure by François Deloche]*

non-examinable material

GRU (Gated Recurrent Units) were introduced in 2014 as a simpler alternative to the LSTM block. Their performance is reported to be similar to the one of LSTM (maybe slightly better on smaller problems and slightly worse on bigger problems). As they have fewer parameters than LSTM, GRUs are quite a bit faster to train.

> J. Chung, C. Gulcehre, K. Cho and Y. Bengio (2014). "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". [https://arxiv.org/abs/1412.3555]

> `Keras: https://keras.io/layers/recurrent/#gru`



Architecture of Gated Recurrent Cell. *[Figure by François Deloche]*

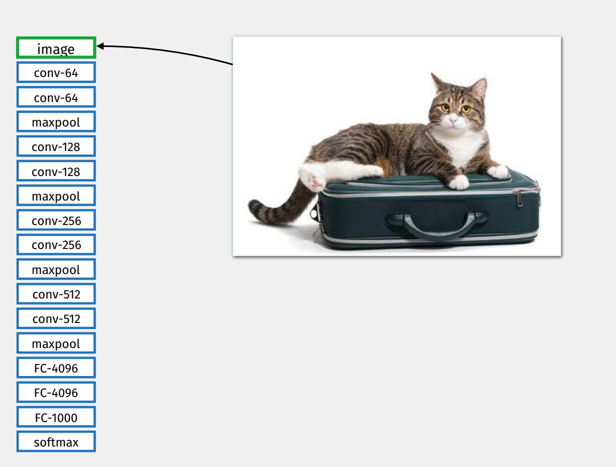
non-examinable material

Application: Image Caption Generator

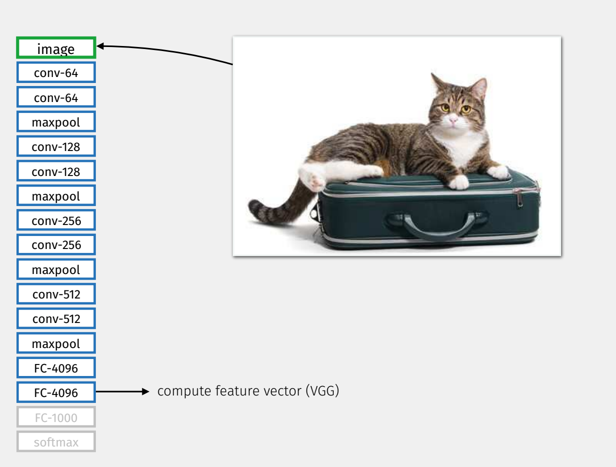
A nice application showing how to merge picture and text processing is **Image Caption Generator**, which aims at automatically generating text that describes a picture.

O. Vinyals, A. Toshev, S. Bengio and D. Erhan (2015). "Show and tell: A neural image caption generator" [https://arxiv.org/abs/1411.4555]

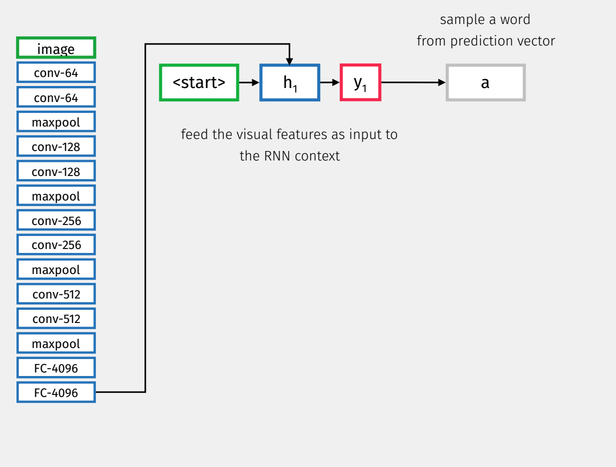
Google Research Blog [https://goo.gl/U88bDQ]



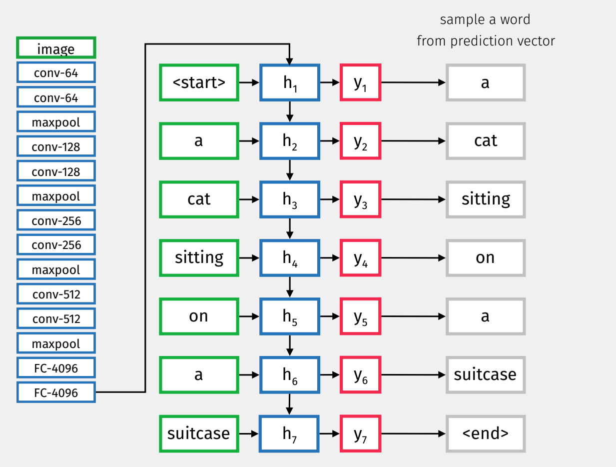
We start by building visual features using an off-the-shelf CNN (in this case VGG).



We don’t need the classification part so we only used the second to last Fully Connected layer.



We then feed this tensor as an input to a RNN that predicts the next word.



We then continue sampling the next word from the predictions till we generate the `<end>` word token.