

# 1 Recurrent neural networks for sequential processing

*INSERT WIT HERE AND FIX TITLE*

— J. M. G. de Lammens, PhD Dissertation,  
*A Computational Model of Color Perception  
and Color Naming*, State University of New York,  
1994

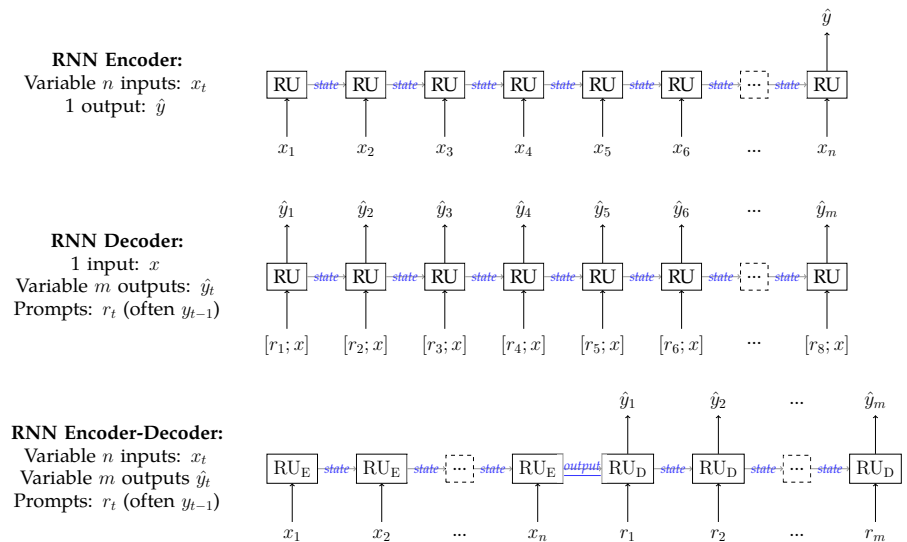
This chapter continues on from the general introduction to machine learning, with a focus on recurrent networks. Recurrent neural networks are the most neural network approach for working with sequences of dynamic size. As with the prior chapter, readers familiar with RNNs can reasonably skip this also; it does not pertain specifically to NLP. However, as NLP tasks are almost always sequential in nature, RNNs are a very important technique

## 1.1 Recurrent Neural Networks

/ A key limitation of a neural network is that the number of inputs and outputs must be known at training time and must always be the same for all cases. This is not true for natural language: If a problem involves processing a sentence, then each input will be made up a varying number of words. Similarly for output, in a text generation case.

Recurrent neural networks (RNN) overcome this by allowing the network to have state that persists over time. Inputs of any size can be handled one constant sized part of the input (e.g. one word) at a time, using the state to remember past inputs.

**Figure 1.1:** The unrolled structure of an RNN for us in Encoding, Decoding and Encoding-Decoding (sequence-to-sequence) problems. RU is the recurrent unit – the neural network which reoccurs at each time step.



A RNN is effectively a chain of feed-forward neural networks, each one being identical in terms of their weight and bias parameters. Whiled identical in terms of parameters, they each acting at a different time-step.

## Time-Step

RNNs are normally described in terms of a time-step. This is the advancement of the system such that the previous output state, is now the input old-state. This doesn't have to be literal time – indeed it can not be, as it is discrete. In most NLP applications it is time analogous: words in the order they are said. In other NLP applications it might actually be words in the reverse order to that in which they are normally said. In other machine learning applications it may not correspond to time at all. For example, using a rotating distance sensor (e.g radar) each time-step corresponds to a different angle of the antenna.

At each time-step the same network is used, with different inputs. For purposes of looking at this in the big picture we will first consider each network a black-box recurrent unit (RU).

The recurrent unit takes at each time-step, an input for that time-step, some representation its of state for the RU at the previous timestep; and it produces an output for this time-step, and its state representation for the next time-step. A diagram of this is shown in Figure 1.2. It is worth distinguishing the unit output is not the same as the overall network output, it is just the output of this sub-network at this time-step. Every-time step can be considered as having two effective inputs (previous state, and actual input at this time-step) and two effective outputs (next state and actual output at this time step).

Not all of these inputs and outputs are actually used at all time steps meaningfully. The initial state for the first time step is normally set to some zero vector, and the final state at end of the sequence of normally discarded. Similar things are done for inputs and outputs

depending on the type of problem (decoder/encoder) that the RNN is being used for.

### 1.1.1 General RNN structures

In general most interested uses of recurrent networks can be considered belongs to one of 4 general types of structure. Matched-sequence, encoder, decoder, and encoder-decoder. These common structures are shown in Figure 1.1. The motivation for using an RNN is if the size of the input and/or the size of the output is not consistent across all cases.

If the input size and the output size is always the same then one can use an matched-sequence RNN structure. This is the most basic RNN structure. At each time-step, there is an input, and a target output. An example of this in natural language processing is part of speech (POS) tagging. Every word is to be labelled as a noun, a verb, an adjective etc. This does require memory as the other words around the word being classified influence the correct POS. Since the same word can occupy a different part of speech depending on the usage. For example 'record' is both a noun and a verb (it is in-fact almost ubiquitous that all verbs have a noun form). This particular example is a good use for a Bidirectional RNN (Section 1.1.4), as both the previous and following words are useful for determining the POS. A key limitation of the matched-sequence structure is that the input and output size must be the same. Often though one would like to process an input that could be any size, but produce just a fixed size output.

For example if one is trying to learn a mapping from a textual color names into a probability distribution in color-space (2017arXiv170909360W) , then different descriptions have different numbers of words. One input might be 'very light green', while another just 'orange'. At each time-step one input is provided – being one of the words in the (potentially multi-word) color name. Another example use is sentiment analysis,

[2017arXiv170909360W](#),  
[2017arXiv170909360W](#)

predicting the sentiment being expressed by a sentence as positive or negative. All the outputs at all time-steps except the last can be ignored. The output of the final time-step can be connected to a further network with a final output layer giving the overall output. This situation with a variable number of inputs but a fixed size of output is described as an encoder network.

## Pseudo-tokens

Pseudo-tokens such as '<EOS>' (end of string), '<START>' (start of string), and similar are common in RNN tasks. As mentioned, outputting '<EOS>' is required to know when the decode is done outputting. '<START>' can be useful to model things that occur near the start of input; and as an initial prompt. In some tasks other non-word tokens might be inserted also. Such as in transcription of recorded speech, a '<PAUSE>' token might be included. Pragmatically, as long as the symbols used to represent these never occur amongst the true word tokens, these can be treated just like regular words by the system.

2016arXiv160603821M,  
2016arXiv160603821M

cho-EtAl:2014:EMNLP2014,  
cho-EtAl:2014:EMNLP2014

The reverse is a decoder network. This means a fixed sized input mapping to a variably sized output. If the system is attempting to learn from a point in color space to the name of that color (2016arXiv160603821M). For example: (144,238,144) 'very light green' (3 outputs); but (255,165,0) might map to just 'orange' (one output). In this decoder type network, there is one true input, at the first time-step, and the output from every time-step is used. Each output can be connected to a softmax layer giving probability for possible words. Some inputs at each time-step after the first in such a network inputs must also be provided for the decoding. We call this input a prompt, as it is not providing new information to the network, merely driving it to produce the next output. It is common to include an end of string marker token (often literally "<EOS>"), so as to know when to stop prompting for additional outputs.

The encoder-decoder RNN is the generalise structure for sequence to sequence learning (cho-EtAl:2014:EMNLP2014). While the matched-sequence RNN take a sequence as an input and produces a sequence as an output those sequences must be the same size. In an application such as machine translation, or question answering, or automatic captioning, sequential input and output is required, but the sequences have different lengths. A sentence in one language will not normally translate to a sentence with the exact same number of word in another. The solution to this is the encoder-decoder RNN. In this structure an encoder RNN is used to take the input, its final output is then connected as the input to so a separate decoder RNN. Thus separating the input processing from the output generation.

### 1.1.1.1 Prompts in decoder RNNs

Prompts are required for decoders as the network must have some input to cause it to give an output. In theory this could be a constant: all inputs and all outputs for those inputs, can be learned and remembered by the network's memory (encoded in its state). In practice is very common to include part of the output of the previous step as part of the prompt. When generating a sequence of words for example, one can use the previous word. At training time this can be the targeted previous output. At test time (and in real deployment) this is normally the most-likely predicted output. This effectively gives direct bigram state information to the model, allowing the memory to focus on higher level tasks. It also allows the outputs to be explored, for example by providing the second most-likely word as the first prompt a different sequence can be generated.

It is also common to include in prompts the original input to the decoder. For example in a caption generator, including the a vector representation of the image (for example an Inception Image Embedding); in the color decoder example this would be including the original color representation. In general the prompt can be used to add information to the network ensuring that each time step can do as well as possible, even if the state does not capture all the information desired. Ensuring the state is able to capture all information is part of designing the internals of the RU.

#### Terminology: Prompt

The word prompt is our own terminology here. We are not aware of a consistent term used in literature for the input at each time step to a decoder RNN. Dummy input would also work, although as discussed the choice of prompt can allow useful information to be added. Simplifying the learning problem.

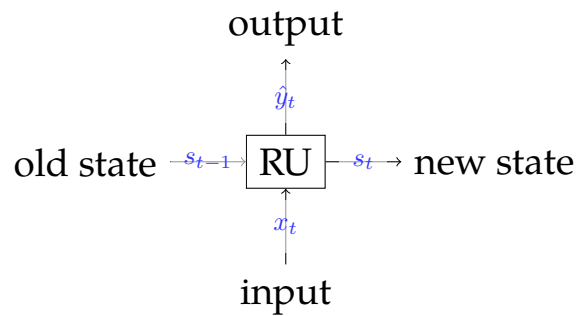
### 1.1.2 Inside the Recurrent Unit

In this section we discuss the various different types of recurrent unit. This is the difference between Elman networks, GRU networks and LSTM networks. As discussed every recurrent unit from the outside has the previous state, the next state, the unit input and the unit output. What differs is how they are connected

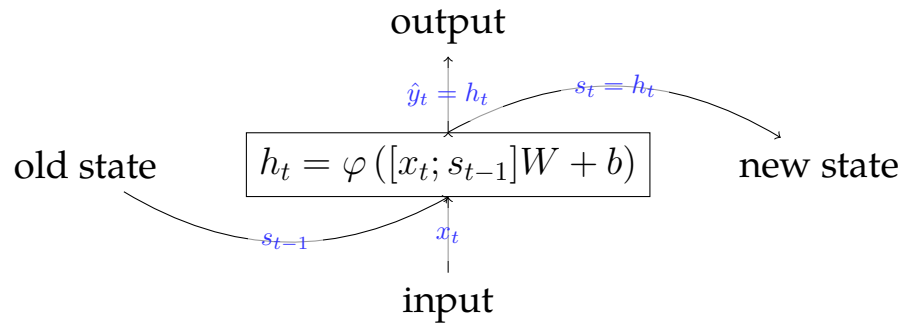
#### Output/Input layers

Extra feedforward layers are often applied between the overall network's input and output and the RU's input and output. The extra layers could be considered as part of the RU, as they occur at

each time-step (even if not used). Alternatively they can be considered as something that surrounds the RU, only at certain time-steps. This depends on depending on point of view. For out purposes we will consider the core of the RU in isolation.



**Figure 1.2:** A recurrent unit with its 2 inputs and 2 outputs. Not shown are the internal functioning which may be a complex (or simple) neural network of its own.



**Figure 1.3:** A basic recurrent unit.

and what controls the information with in them. Every recurrent unit is itself a neural network.

In many types of RU (e.g. GRU, Basic RU) the output and the state are always equal. This particularly makes sense when they should be capturing the same kinds of information (as in a decoder-encoder). Further-more as in general there will be additional feed-forward layers on top of used outputs (if nothing else an output layer is often required), the need to differentiate output from state is lessened. However, it is a distinction made in the very well known LSTM unit (Section 1.1.2.3) so we preserve it here.

## 1.1.2.1 Basic Recurrent Unit

### Jordan RU

The formulation is not commonly used today, however analogously to considering the Basic RU as an Elman RU, a similar formulation can be done for a Jordan network (jordan1986rnnTR). The Jor-

The most basic recurrent unit, is a single layer  $h_t$ , with the unit output, and the unit state both being the value of this layer. The layer is not hidden from the perspective of the recurrent unit, but is from the perspective of the whole network. The network inside the basic



recurrent unit is given by:

$$h_t = \varphi(W[x_t; s_{t-1}] + b) \quad (1.1)$$

$$s_t = h_t \quad (1.2)$$

$$\hat{y}_t = h_t \quad (1.3)$$

dan RU would be given by:

$$h_t = \varphi(W[x_t; s_{t-1}] + b)$$

$$o_t = \varphi(Vh_t + c)$$

$$s_t = o_t$$

$$\hat{y}_t = o_t$$

As shown in Figure 1.3 Where  $W$  and  $b$  are the weight matrix and bias vector that are trained, for this single hidden layer.

It would not need an additional output layer. One of these RUs alone is a Jordan network.

One could call this an Elman RU: the networks considered by in **elman1990finding** are such a basic RU, with an extra over-all output layer on-top.

**elman1990finding,**  
**elman1990finding**

Basic RNN units are not used in many modern works. It is difficult for these networks to propagate error information across many time-steps (**bengio1994learning**). This results in the state not capturing information from many time-steps ago. Thus the network as a very short memory.

**bengio1994learning,**  
**bengio1994learning**

Other more advanced recurrent units solved this problem by placing more explicit controls on the state.

### 1.1.2.2 Gated Recurrent Unit

The Gated Recurrent Unit (GRU) was introduced by **cho2014properties**, GRU is actually a simplification of the much older and better known Long Short Term Memory (LSTM), which will be discussed in Section 1.1.2.3).

**cho2014properties,**  
**cho2014properties**

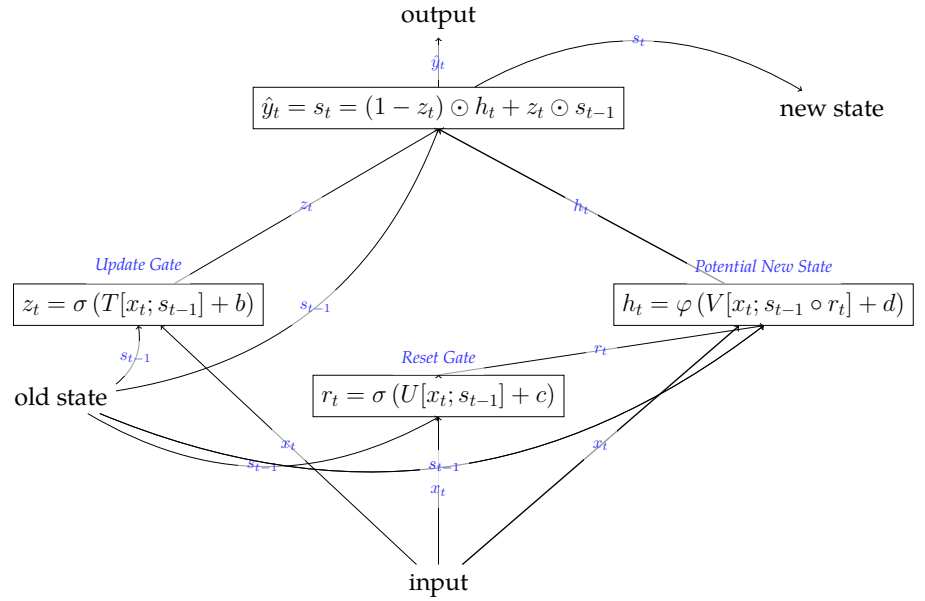
We discuss GRU first as it is the simpler system. In the evaluations of **chung2014empirical** and **jozefowicz2015empirical** it was found to perform very well – similar to LSTM for most tasks.

**chung2014empirical,**  
**chung2014empirical**

**jozefowicz2015empirical,**  
**jozefowicz2015empirical**

The core insight is to gate the changes in the state. There are subnetworks within the RU that we call gates. These subnetworks learn how the state should change. Thus helping the overall network preserve better information in the state.

**Figure 1.4:** A Gated Recurrent Unit



The gated recurrent unit (GRU) is defined by:

$$z_t = \sigma(T[x_t; s_{t-1}] + b) \quad (1.4)$$

$$r_t = \sigma(U[x_t; s_{t-1}] + c) \quad (1.5)$$

$$h_t = \varphi(V[x_t; s_{t-1} \odot r_t] + d) \quad (1.6)$$

$$s_t = (1 - z_t) \odot h_t + z_t \odot s_{t-1} \quad (1.7)$$

$$\hat{y}_t = s_t \quad (1.8)$$

Where  $T, U$  and  $V$  are the weight-vectors, and  $b, c$  and  $d$  are the bias vectors for the 3 separate layers.

#### Trade-off formula

It may be stating the obvious, but a standard way (and the only sensible way) to trade-off a value between two possible choices:  $a$  and  $b$ , when given a preference level for the first as  $p$  (a number between zero and one) is  $(p)a + (1 - p)b$ . If  $p$  is interpreted as the probability of  $a$  being the correct value of a random variable, with  $b$  as the other possible value, then this is the expected value of that random variable.

This may look complex, but it can be broken down into parts. First we can see that like in the basic RU, the output and the state are the same value.

$h_t$  is our core layer, as in the basic cell. However, unlike in the basic cell, it does not immediately become the state:  $s_t$ . There is a trade-off between the state keeping its old value  $s_{t-1}$ , and getting the hidden layer value  $h_t$ . This trade-off is element-wise controlled by  $z_t$  which is valued between 0 and 1 (due to the sigmoid unit). When an element of  $z_t$  is 1, then the value of  $s_{t-1}$  for the corresponding element is kept. Conversely, when the  $z_t$  element is 0, then the element of the new state is fully given by  $h_t$ .

$z_t$  is often called the *update gate* as it controls ("gates") how much the state is updated using the new value in  $h_t$ .



The update-gate sub-network uses the previous state  $s_{t-1}$  and present input  $x_t$  to make this determination.

The *reset gate* is loosely similar.  $r_t$  controls how much influence the past state  $s_{t-1}$  has on calculating the new value of  $h_t$  – which is the new potential state/output as discussed. It is perhaps clearer if  $h_t$  is reformulated to split  $V$  into the terms that multiply with  $x_t$  and the terms that multiply with  $s_{t-1}$ :

$$h_t = \varphi(\sigma(V[x_t; s_{t-1}] \odot r_t) + d) \quad (1.9)$$

$$= \varphi(\sigma(V_x x_t + V_s r_t \odot s_{t-1} + d)) \quad (1.10)$$

It  $r_t$  is called the reset-gate because it wipes the effect of the old-state in calculating the potential new state  $h_t$ . When  $r_t$  is reduced to zero, then the updated value for  $h_t$  is as if  $s_{t-1}$  was just like for the initial zero-vector state – it is zeroed out by  $r_t$ . If the update gate  $z_t$  is high then it would fully reset the system.

Notice that when  $z_t = 0$  and  $r_t = 1$  then the system is identical to the Basic RU. This could be achieved by setting suitably large biases. However, the system is more flexible than that, since  $z_t$  and  $r_t$  are themselves very similar to Basic RUs – though they do not control their own state. The gates can behave differently based on the inputs and states to recognise important information that must be stored.

### 1.1.2.3 LSTM Recurrent Unit

The LSTM is most well known RNN unit, the term is very nearly interchangeable with RNN today. The original form was proposed by [hochreiter1997long](#). The form in current use is a variant from [gers1999learning](#).

LSTM uses a compound state, comprised of the units previous output, and an addition state vector called the cell. we write  $s_t = (c_t, \hat{y}_t)$ , to fit the normal formulation, though for maths it is easier to work with these in parts.

### Multiplication, Concatenation, and Addition

It is important to grasp that the product of a matrix and the concatenation of two vectors can also be expressed as the sum of the product of a block of that matrix. The area the same thing.

$W \cdot [a; b] = U \cdot a + V \cdot b$  if  $W = [U \ V]$

For  $[a; b]$  being the vertical concatenation of the vectors (considered as column matrices) and  $[U \ V]$  the horizontal concatenation of the matrices respectively.

The difference is purely notation.

[hochreiter1997long](#),  
[hochreiter1997long](#)

[gers1999learning](#),  
[gers1999learning](#)

### Biasing the Forget Gate

Practically, it is important to initially set the forget gate bias to 1. Unlike all the other initialisation in the networks to 0 (or a small random number eg  $\text{randn}(0.01)$ ). Equivalently one can also just add a constant additional bias term of +1 to the forget gate equations. This has been show to benefit almost all used of LSTM (gers1999learning; jozefowicz2015empirical). This is the default in most frameworks (though only recently patched in some).

$$s_t = (c_t, \hat{y}_t) \quad (1.11)$$

$$i_t = \sigma(W[x_t; \hat{y}_{t-1}] + d) \quad (1.12)$$

$$f_t = \sigma(T[x_t; \hat{y}_{t-1}] + b) \quad (1.13)$$


$$o_t = \sigma(U[x_t; \hat{y}_{t-1}] + k) \quad (1.14)$$

$$h_t = \tanh(V[x_t; \hat{y}_{t-1}] + a) \quad (1.15)$$

$$c_t = i_t \odot h_t + f_t \odot c_{t-1} \quad (1.16)$$

$$\hat{y}_t = o_t \odot \varphi(c_t) \quad (1.17)$$

In LSTM there are 3 gate sub-networks: the input gate  $i_t$ , the forget gate  $f_t$ , and the output gate  $o_t$ .

Together the input and forget-gates take the purpose of the GRU's update-gate. Consider the case if  $o_t = 1$  and  $\varphi$  is the identify function, and with  $i_t = 1 - f_t$ , that would make the value for  $c_t$  very similar to GRU's  $s_t$  (and  $\hat{y}_t$  identical) 

Individually, the forget gate  $f_t$  controls the extent that previous value of the cell is used, and the input gate  $i_t$  controls the extent to which the new potential value  $h_t$  is used for the new value of  $c_t$ .

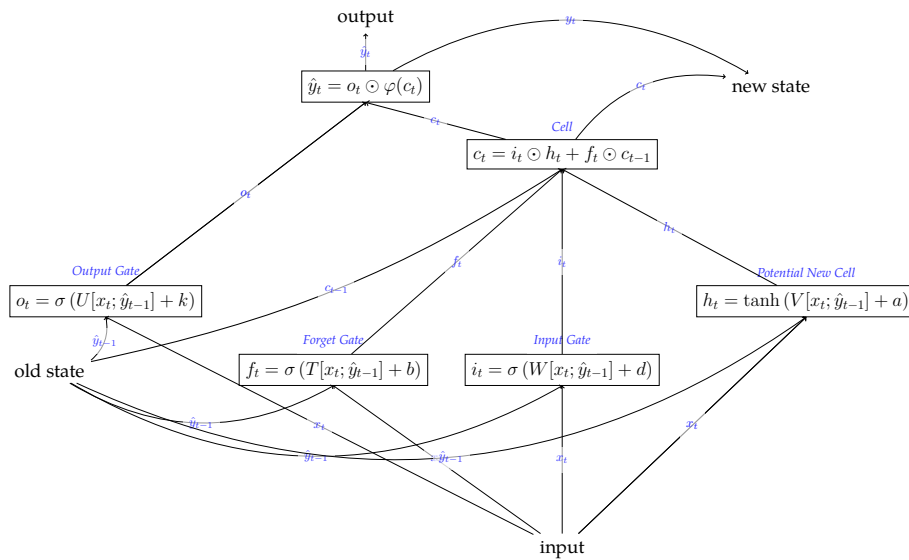
The output-gates obvious purpose is to gate the output  $y_t$ . However, as the output forms part of the state, this has an effect on the networks next time step. Loosely this can be seen as similar to the GRU's reset gate. If we substitute the value for  $\hat{y}_{t-1} = o_{t-1} \cdot \varphi(\odot c_{t-1})$  into  $h_t$ :

$$h_t = \tanh(V[x_t; \hat{y}_{t-1}] + a) \quad (1.18)$$

$$= \tanh(V[x_t; o_{t-1} \cdot \varphi(\odot c_{t-1})] + a) \quad (1.19)$$

Similar for  $o_t$ ,  $f_t$  and  $o_t$  as for  $h_t$ . It can be seen that at the previous time step setting an element of  $o_{t-1}$  to zero effectively removes the effect of the previous cell-state from the equations for all the gates and the the potential new cell  $h_t$ . Thus resetting the network.

## 1.1 Recurrent Neural Networks



**Figure 1.5:** A LSTM Recurrent unit. It may seem complex, and that is basically because it is. However, it can be broken down into understandable parts.

### 1.1.3 Deep Variants

One can have a deep RNN. This is done by stacking additional recurrent units above the existing ones. Attaching, the unit output as the unit input of the layer above. The result can be interpreted as a single, deep recurrent unit.

### 1.1.4 Bidirection RNNs

Similar to deep variants, a Bidirectional RNN ([schuster1997bidirectional](#))

. This is effectively using using two RNNs so that the input can be processed from both temporal directions. The forward and backwards RNNs both take the same input at each timestep, but are other wise fully distinct. The unit outputs at each time step are concatenated before passing for further processing.

[schuster1997bidirectional](#),  
[schuster1997bidirectional](#)

This is not so useful in most decoder RNNs, the number of outputs (time steps) of a decoder RNN is not known except by running them until an end-marker is output. However, there are absolutely no issues when using them for an encoder as all the inputs are known in advance as it is processed one full input at a time. Similar is true for a matched-sequence RNN, when real-time output is not required, and the sequence can be broken up into blocks.

### 1.1.5 Other RNNs

[mikolovLongerSTM](#),  
[mikolovLongerSTM](#)

[stacklstm](#), [stacklstm](#)

[MemoryNN](#), [MemoryNN](#)

[DBLP:journals/corr/GravesWD14](#),  
[DBLP:journals/corr/GravesWD14](#)

There also exist many other RNNs. Some are similar in structure to those discussed and can be analysed similarly via their recurrent units. Such as the two models of **mikolovLongerSTM** , which incorporate a decaying sum of past states into the current state (either gated, or constant). Other networks incorporate various other data-structures (or analogies to such structures) into there recurrent structure: such as stacks **stacklstm** , pointers/arrays **MemoryNN** , or tapes **DBLP:journals/corr/GravesWD14** .

At natural language is primarily a sequential task, RNNs in various forms will be discussed in almost every chapter of this book.