

# Warning: Higher dimensions ahead !

In the an earlier module we went [beyond the feature dimension](#) ([hIntro to CNN.ipynb#Introduction:-Beyond-the-Feature-dimension](#)) to introduce non-feature dimensions.

Many of the examples we with non-feature dimensions that we visited

- had *spatial* dimensions as the non-feature dimension
  - e.g., images

In this module, we introduce a non-spatial dimension often corresponding to time

- *temporal* non-feature dimension

Strictly speaking, the dimension doesn't have to correspond to time

- just a total ordering of elements

That is, we will deal with inputs that are *sequences*.

Some common sequences

- time-series
- sequences of words: Natural Language Processing

Given an input sequence  $\mathbf{x}$

- let  $\mathbf{x}_{(t)}$  denote the element at index  $t$  in the sequence

There may be non-feature dimensions in addition to the sequence index.

Consider  $\mathbf{x}^{(i)}$ : an example that happens to be a movie.

It is

- a sequence of "frames"
- each frame is a "pixel grid" with two non-feature dimensions indexing the row/column address of a pixel in the frame

So

- $\mathbf{x}^{(i)}$  is a movie: a sequence of frames
- $\mathbf{x}_{(t)}^{(i)}$  is the  $t^{th}$  frame in the movie
- $\mathbf{x}_{(t),j,j'}^{(i)}$  is a particular pixel within the frame  $\mathbf{x}_{(t)}^{(i)}$ 
  - The positional dimension is indexed by  $(t)$  and the spatial dimensions by  $j, j'$

Remember

- the feature dimension has no order
  - permuting the features
    - from Price, Volume, Open, Close
    - to Open, Close, Price, Volume
  - does not change the meaning of the example

But the non-feature dimensions *can* impose ordering relationships among the elements.

Consider

- the indices of the temporal dimension are totally ordered
  - reversing the indices makes time flow backwards rather than forwards
  - words in a sentence are ordered
$$\mathbf{x} = [\text{Machine, Learning, is, easy, not, hard}]$$
$$\mathbf{x}[\text{perm}] = [\text{Machine, Learning, is, hard, not, easy}]$$
  - very different meanings

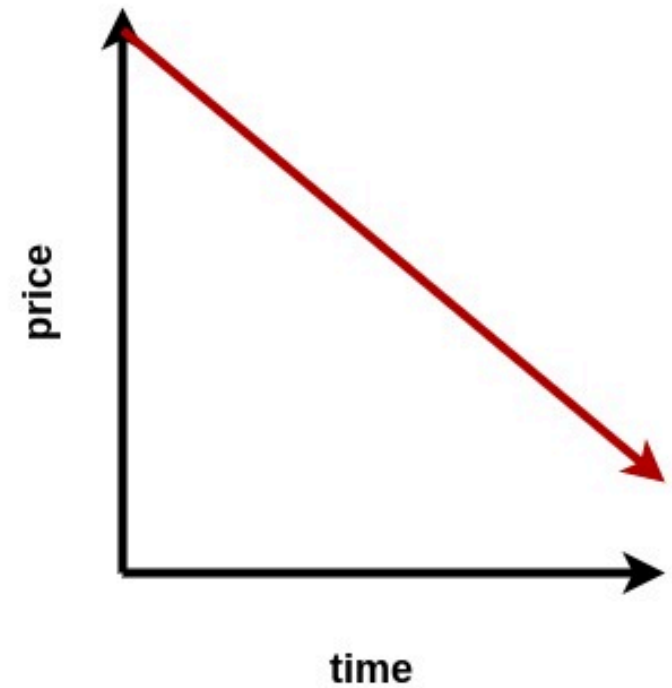
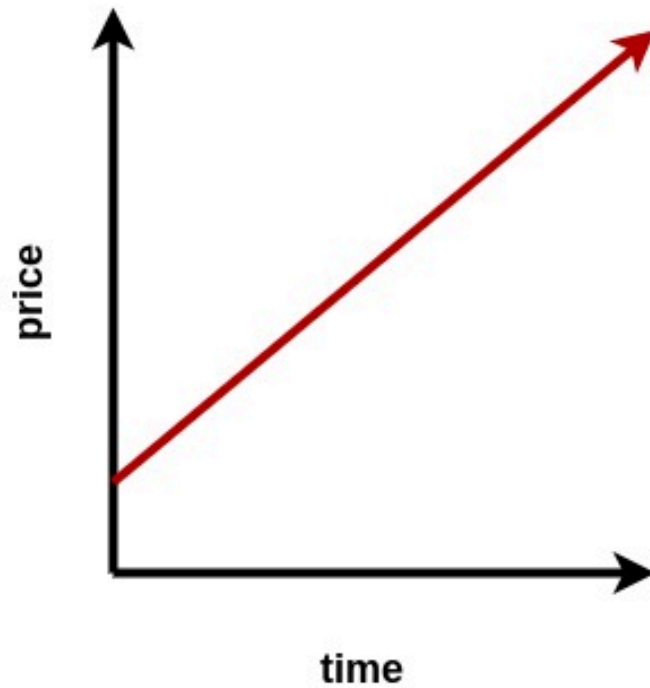
- the indices of the spatial dimension are (at least, partially) *ordered*
  - given an image of a face
    - the eyes are located above the mouth
    - in a horizontal orientation
  - but flipping an image (reversing right and left) may still be meaningful
    - even if not identical to the original

Here are some sequences and permutations

- the meaning clearly changes

Same prices

---





Same words

Machine	Learning	is	easy	not	difficult
Machine	Learning	is	difficult	not	easy

Same pixels

---



# Functions on sequence

In the absence of a positional dimension, our multi-layer networks

- Computed functions from vectors to vectors

With a positional dimension, there are several variants of the function

- Many to one
  - Sequence as input, vector as output
  - Examples:
    - Predict next value in a time series (sequence of values)
    - Summarize the sentiment of a sentence (sequence of words)

- Many to many
  - Sequence as input, sequence of vectors as output
  - Examples
    - Translation of sentence in one language to sentence in second language
    - Caption a movie: sequence of frames to sequence of words

- One to many
  - Single input vector, sequence of vectors as output
  - Examples
    - Generating sentences from seed

# Recurrent Neural Network (RNN) layer

With a sequence  $\mathbf{x}^{(i)}$  as input, and a sequence  $\mathbf{y}$  as a potential output, the questions arises:

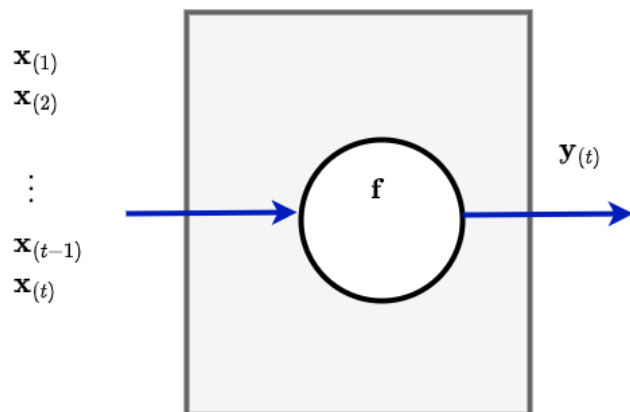
- How does an RNN produce  $\mathbf{y}_{(t)}$ , the  $t^{th}$  output ?

Some choices

- Predict  $\mathbf{y}_{(t)}$  as a direct function of the prefix of  $\mathbf{x}$  of length  $t$ :

$$p(\mathbf{y}_{(t)} | \mathbf{x}_{(1)} \dots \mathbf{x}_{(t)})$$

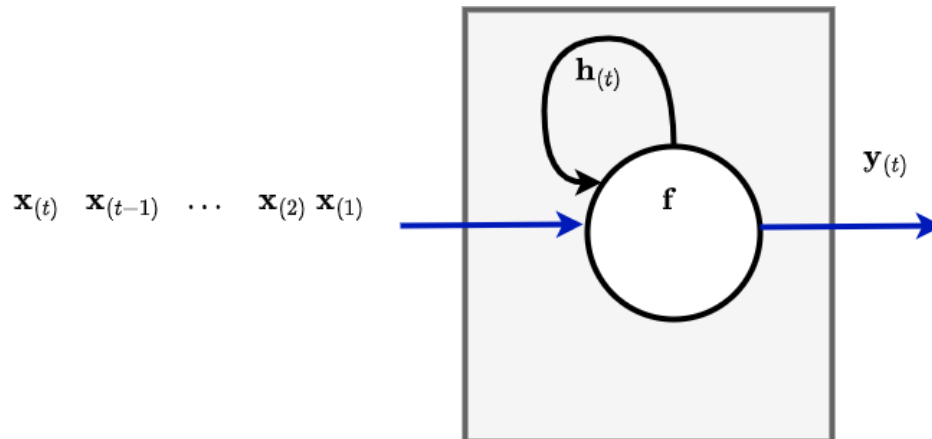
**Direct function**



- Loop
  - Uses a "latent state" that is updated with each element of the sequence, then predict the output

$p(\mathbf{h}_{(t)} | \mathbf{x}_{(t)}, \mathbf{h}_{(t-1)})$     latent variable  $\mathbf{h}_{(t)}$  encodes  $[\mathbf{x}_{(1)} \dots \mathbf{x}_{(t)}]$   
 $p(\hat{\mathbf{y}}_{(t)} | \mathbf{h}_{(t)})$                       prediction contingent on latent variable

Loop with latent state



Since elements of the sequence are presented **one element at a time**

- the latent state  $\mathbf{h}_{(t)}$  must act as a **summary** of all prior elements  $\mathbf{x}_{(1)} \dots \mathbf{x}_{(t)}$   
$$\mathbf{h}_{(t)} = \text{summary}(\mathbf{x}_{([1:t])})$$

Note that  $\mathbf{h}_{(t)}$  is a *vector* of fixed length.

Thus, it is a *fixed length* representation of the key aspects of a sequence  $\mathbf{x}$  of potentially *unbounded* length.



## Example

Let's use an RNN to compute the sum of a sequence numbers

- the latent state  $\mathbf{h}_{(t)}$  can be maintained as

$$\mathbf{h}_{(t)} = \text{summary}(\mathbf{x}_{([1:t])}) = \sum_{t'=1}^t \mathbf{x}_{(t')}$$

- by updating  $\mathbf{h}_{(t)}$  in the loop

$$\mathbf{h}_{(t)} = \mathbf{h}_{(t-1)} + \mathbf{x}_{(t)}$$

The Recurrent Neural Network (RNN) adopts the "latent state" approach.

A prime advantage of the latent state approach

- it can handle sequences of *unbounded* length

Here is some pseudo-code:

```
In [2]: def RNN( input_sequence, state_size ):
        state = np.random.uniform(size=state_size)

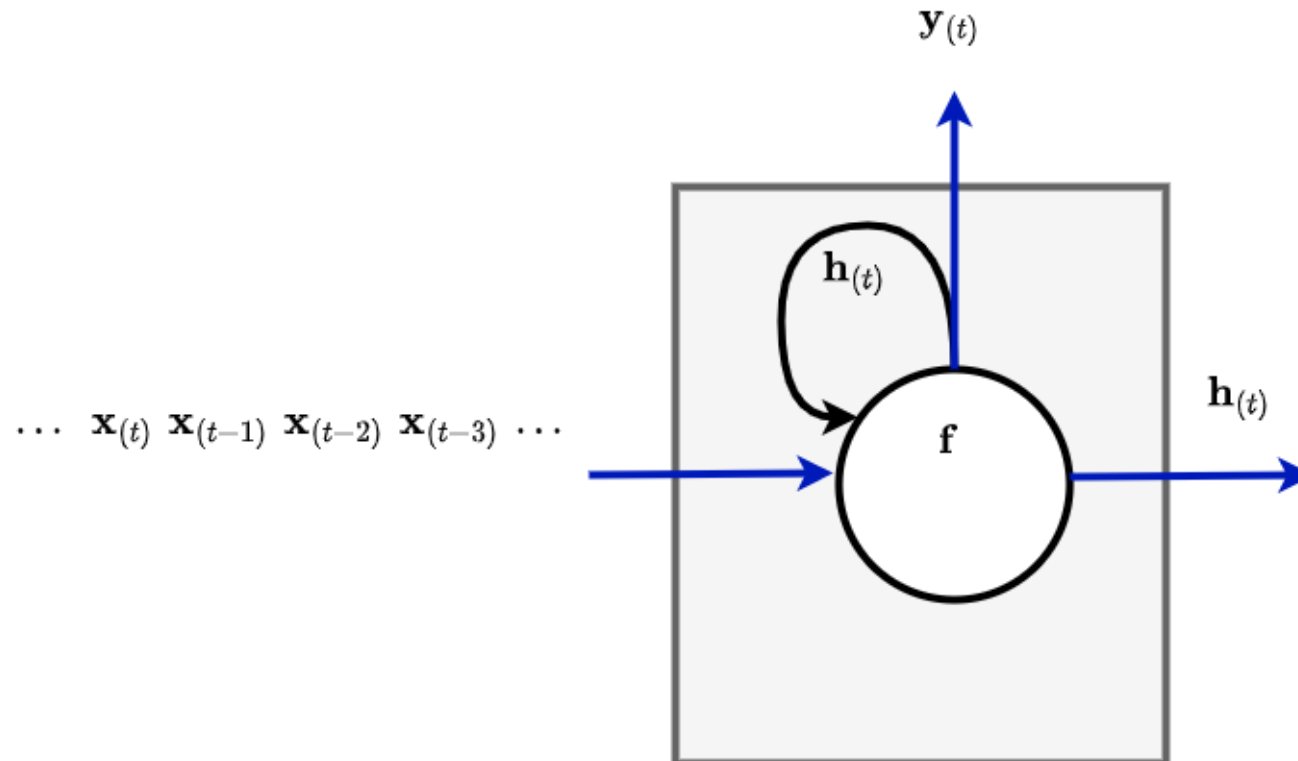
        for input in input_sequence:
            # Consume one input, update the state
            out, state = f(input, state)

        return out
```

And the corresponding diagram, showing the output out ( $\mathbf{y}_{(t)}$ )

RNN

---



## Output $\hat{\mathbf{y}}_{(t)}$ of an RNN

According to our pseudo-code and diagram

$$\hat{\mathbf{y}}_{(t)} = \mathbf{h}_{(t)}$$

That is: the output is the same as the latent state.

It is easy to add another NN to transform  $\mathbf{h}_{(t)}$  into a  $\hat{\mathbf{y}}_{(t)}$  that is different

- we will omit this additional layer for clarity

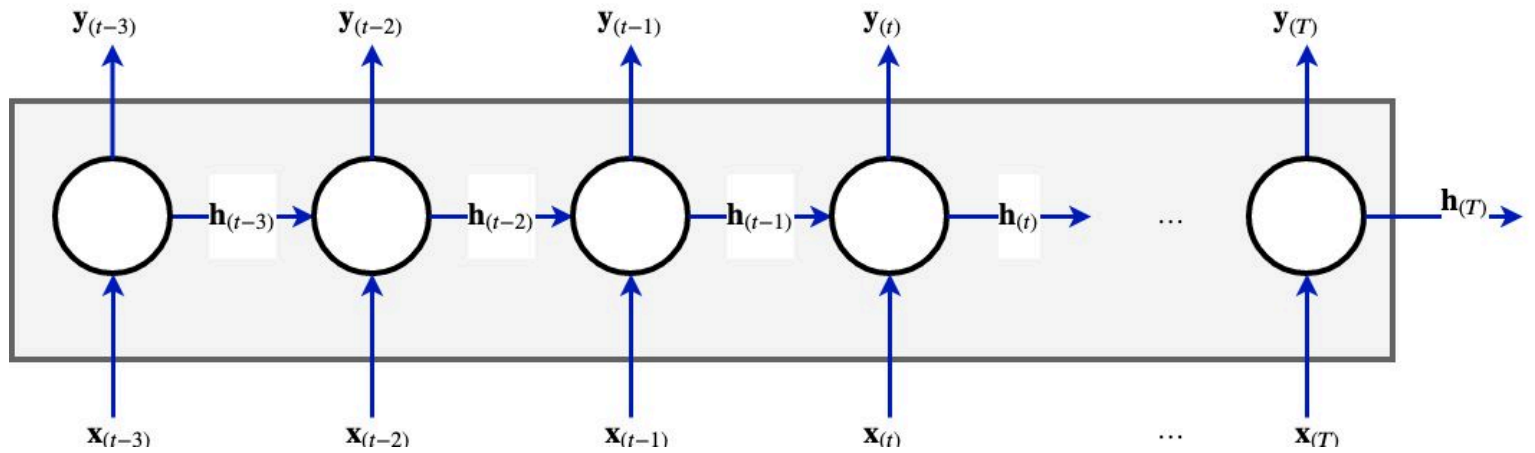
## Unrolled RNN diagram

We can "unroll" the loop into a kind of movie

- a sequence of steps
- step  $t - 1$  arranged to the left of step  $t$

### RNN many to many API

---



At each time step  $t$

- Input  $\mathbf{x}_{(t)}$  is processed
- Causes latent state  $\mathbf{h}$  to update from  $\mathbf{h}_{(t-1)}$  to  $\mathbf{h}_{(t)}$ 
  - We use the same sequence notation to record the sequence of latent states  $[\mathbf{h}_{(1)}, \dots, ]$
- Optionally outputs  $\mathbf{y}_{(t)}$  (for outputs that are of type sequence)



When processing  $\mathbf{x}_{(t)}$

- The function computed takes  $\mathbf{h}_{(t-1)}$  as input
- Latent state  $\mathbf{h}_{(t-1)}$  has been derived by having processed  $[\mathbf{x}_{(1)} \dots \mathbf{x}_{(t-1)}]$
- And is thus a *summary* of the prefix of the input encountered thus far

One can look at this unrolled graph as being a dynamically-created computation graph.

- A sequence of layers
- One layer per time step
- But with an **identical** computation for all layers

The unrolled version will be crucial in understanding how Gradient Descent works when RNN layers are present.

- Just conceptualize the unrolled loop as a sequence of layers
- All our logic and intuition carries over

Note that  $\mathbf{x}$ ,  $\mathbf{y}$ ,  $\mathbf{h}$  are all vectors.

In particular, the state  $\mathbf{h}$  *may have many* elements

- it is a vector of "synthesized" features
- to record information about the entire prefix of the input.

$\mathbf{h}_{(t)}$  is the latent state (sometimes called the *hidden state* as it is not visible outside the layer).

It is essentially a *fixed length* encoding of the variable length sequence  $[\mathbf{x}_{(1)} \dots \mathbf{x}_{(t)}]$

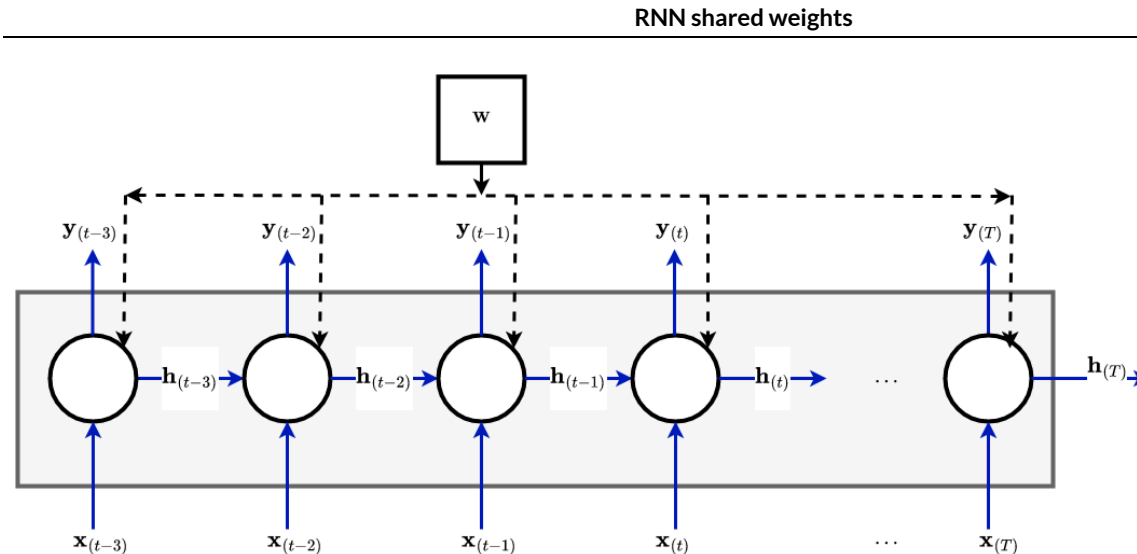
- All essential information about the prefix of  $\mathbf{x}$  ending at step  $t$  is recorded in  $\mathbf{h}_{(t)}$
- Hence, the size of  $\mathbf{h}_{(t)}$  may need to be large

We will shortly attempt to gain some intuition as to what these synthesized features may be.

# All "layers" in the unrolled graph share weights

One extremely important aspect that might not be apparent from the movie version:

- Each unrolled "frame" in the movie shares the *same weights* and computes the *same* function  $F$
- In contrast to a true multi-layer network where each layer has its *own* weights



That is the unrolled RNN computes

$$\begin{aligned}
 \mathbf{y}_{(t)} &= F(\mathbf{y}_{(t-1)}; \mathbf{W}) \\
 &= F( F(\mathbf{y}_{(t-2)}; \mathbf{W}); \mathbf{W} ) \\
 &= F( F( F(\mathbf{y}_{(t-3)}; \mathbf{W}); \mathbf{W} ); \mathbf{W} ) \\
 &= \vdots
 \end{aligned}$$

rather than

$$\begin{aligned}
 \mathbf{y}_{(l)} &= F_{(l)}(\mathbf{y}_{(l-1)}; \mathbf{W}_{(l)}) \\
 &= F_{(l)}( F_{(l-1)}(\mathbf{y}_{(l-2)}; \mathbf{W}_{(l-1)}); \mathbf{W}_{(l)} ) \\
 &= F_{(l)}( F_{(l-1)}( F_{(l-2)}(\mathbf{y}_{(l-3)}; \mathbf{W}_{(l-2)}); \mathbf{W}_{(l-1)} ); \mathbf{W}_{(l)} ) \\
 &= \vdots
 \end{aligned}$$

Note, in particular

- The repeated occurrence of the term **W** will complicate computing the derivative
- As we will see in a subsequent lecture

RNN's are sometimes drawn without separate outputs  $\mathbf{y}_{(t)}$

- in that case,  $\mathbf{h}_{(t)}$  may be considered the output.

The computation of  $\mathbf{y}_{(t)}$  will just be a transformation of  $\mathbf{h}_{(t)}$  so there is no loss in omitting it from the RNN and creating a separate node in the computation graph.

Geron does not distinguish between  $\mathbf{y}_{(t)}$  and  $\mathbf{h}_{(t)}$  and he uses the single  $\mathbf{y}_{(t)}$  to denote the state.

I will use  $\mathbf{h}$  rather than  $\mathbf{y}$  to denote the "hidden state".



# Typical uses of RNN

## Many to one: Creating a fixed length summary of a variable length sequence

A typical Many to One task is predicting the next element in a sequence

For example

- Predict the next word in a sentence
- Predict the next price in a timeseries of prices

These are implemented by a NN (with RNN layers as components) followed by a Head Layer (Classifier or Regressor)

But the Head Layers take **fixed length** inputs and our sequences are of potentially unbounded length !

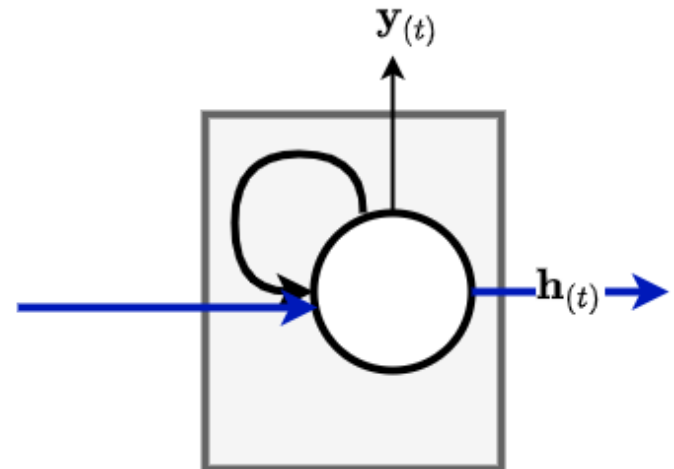
We first need to convert the variable length sequence into a fixed length representation.

Let's make this concrete with an example: a sequence of words

RNN

---

Machine Learning is easy not hard



$\mathbf{h}_{(t)}$  is a **fixed length** vector that "summarizes" the prefix of sequence  $\mathbf{x}$  up to element  $t$ .

The sequence is processed element by element, so order matters.

$$\mathbf{h}_{(0)} = \text{summary}([\text{Machine}])$$

$$\mathbf{h}_{(1)} = \text{summary}([\text{Machine}, \text{Learning}])$$

$$\vdots$$

$$\mathbf{h}_{(t)} = \text{summary}([\mathbf{x}_{(0)}, \dots, \mathbf{x}_{(t)}])$$

$$\vdots$$

$$\mathbf{h}_{(5)} = \text{summary}([\text{Machine}, \text{Learning}, \text{is}, \text{easy}, \text{not}, \text{hard}])$$

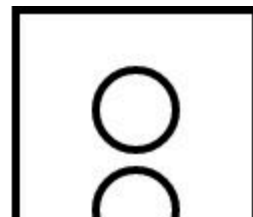
Turning an unbounded length sequence into a fixed length vector is very useful !

- All our other layer types take fixed length input

So we can feed  $\mathbf{h}_{(5)}$  into a Classifier to decide on the sentiment of the sentence.

---

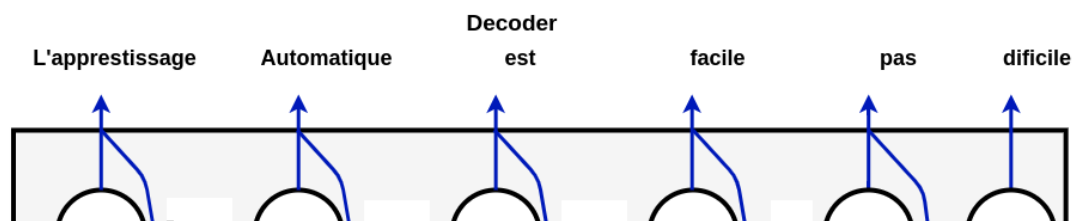
RNN Many to one; followed by classifier



# Many to many: Encoder-Decoder

Another common paradigm using RNN's that we will encounter is the *Encoder-Decoder* which is useful for tasks mapping sequences to sequences

- language translation
- Output and input sequence elements do not have a one to one correspondence
- The Encoder-Decoder decouples the sequences
  - Encoder summarizes the input sequence  $\mathbf{x}_{([1:\bar{T}])}$  with  $\bar{\mathbf{h}}_{(\bar{T})}$
  - Decoder generates output sequence  $\hat{\mathbf{y}}_{([1:T])}$  from the summary



- The final latent state  $\bar{\mathbf{h}}_{(T)}$  of the Encoder "summarizes" the source sentence (English)
- It initializes the latent state of the Decoder which produces the target sentence (French)
- The Decoder implements a one-to-many API
  - source language "summary" as seed

Decoupling means that the length of  $\mathbf{x}$  (length  $\bar{T}$ ) need not be equal to the length of  $\hat{\mathbf{y}}$  (length  $T$ ).



The Sequence to Sequence task is very common in Deep Learning

- Language Translation: source language to target language
- Question answering
  - input: sequence of words (the "question")
  - output: sequence of words (the "answer")
- AI Assistants
  - input: sequence of words (the user prompt)
  - output: sequence of words (the Assistant's response)

We will re-visit the Encoder-Decoder architecture in depth in a future module that introduces Advanced Architectures.

## One to many: Generative ML (generating sequences from a seed)

The two main Machine Learning Tasks we have studied thus far (Regression, Classification) are called *discriminative* tasks

- they learn the relationship between features and targets of an example

We can also use Machine Learning for the *generative* task of creating new examples

- learns the distribution of features
- can sample from the learned distribution to construct new examples

RNN's are often used for generative tasks.

We generate a long sequence that is highly probable (from the learned distribution) given a short sequence as seed.

- The model is initially input a short "seed" sequence.
- The output is a prediction of the **next** element of the sequence
- The input sequence is extended by the prediction
- Repeat !

[Here \(https://app.inferkit.com/demo\)](https://app.inferkit.com/demo) is a demo of creating an entire story from an initial idea comprised of a few words.

# Conclusion

We have introduced the key concepts of Recurrent Neural Networks.

- An unrolled RNN is just a multi-layer network
- In which *all the layers are identical*
- The latent state is a fixed length encoding of the prefix of the input

A more detailed view of sequences and RNN's will be our next topic.

In [3]: `print("Done")`

Done

