Warning: Higher dimensions ahead !

A Fully Connected/Dense layer is insensitive to the order of features.

This is just a property of the dot product

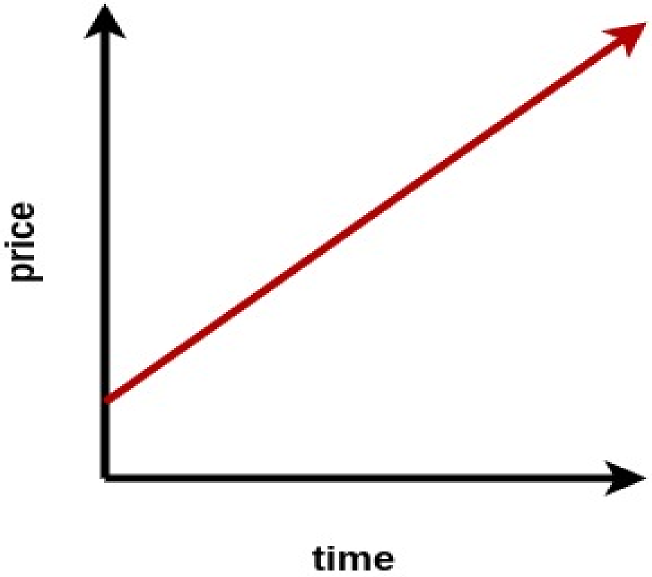
$$\Theta^T \cdot \mathbf{x} = \Theta[\text{perm}]^T \cdot \mathbf{x}[\text{perm}]$$

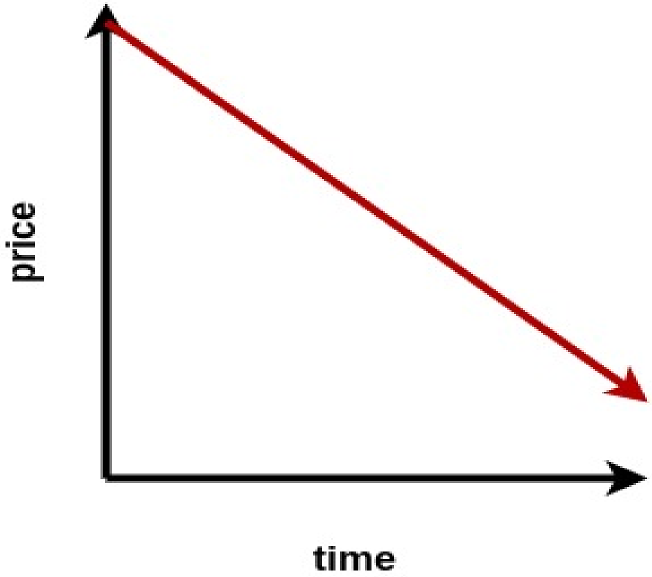
where $\Theta[\text{perm}]^T$ and $\mathbf{x}[\text{perm}]$ are permuations of $\Theta, \mathbf{x}$.

But there are many problems in which order is important.

Consider the following examples

Same prices

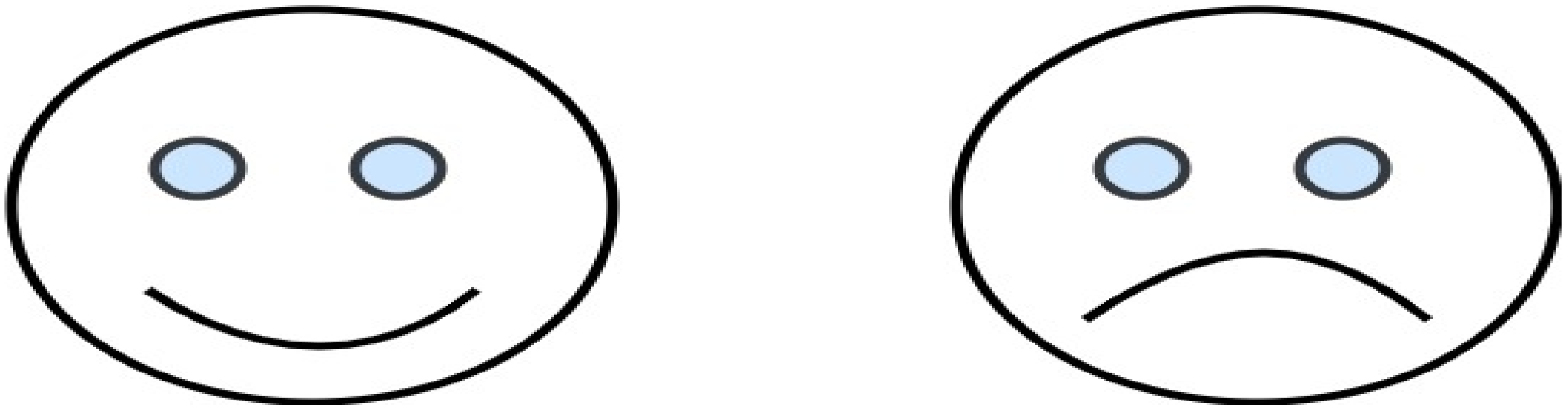




Same words

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

Same pixels



In this lecture, we will be dealing with examples that are *sequences*.

That is, we will add a new dimension to each example which we will call the *temporal dimension*.

To make this concrete, consider the difference between a snapshot and a movie

- A movie is a sequence of snapshots

We have already encountered (when introducing CNN's) data with a *spatial dimension*

- location of a feature within a 1D or 2D space.

The main difference between the spatial and temporal dimensions:

- We have some degree of freedom to alter the spatial dimension without affecting the problem
    - e.g., rotating an image
- There is *no* ability to rearrange data in the temporal dimension
    - Time flows forward and we can't peek ahead.

A single example $\mathbf{x^{(i)}}$ will now be written as

$$[\mathbf{x}^{\mathbf{(i)}}_{(t)} \mid 1 \le t \le T]$$

Using the movie analogy

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

Functions on sequence

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

- Computed functions from vectors to vectors

With a temporal 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:
$$p(\mathbf{y}_{(t)} | \mathbf{x}_{(1)} \dots \mathbf{x}_{(t)})$$
- Uses a "latent state" that is updated with each element of the sequence, then predict the output

$$\begin{array}{ll}
p(\mathbf{h}_{(t)} | \mathbf{x}_{(t)}, \mathbf{h}_{(t-1)}) & \text{latent variable } \mathbf{h}_{(t)} \text{encodes } [\mathbf{x}_{(1)} \dots \mathbf{x}_{(t)}] \\
p(\mathbf{y}_{(t)} | \mathbf{h}_{(t)}) & \text{prediction contingent on latent variable}
\end{array}$$

The Recurrent Neural Network (RNN) adopts the latter approach.

Here is some pseudo-code:

In [2]:

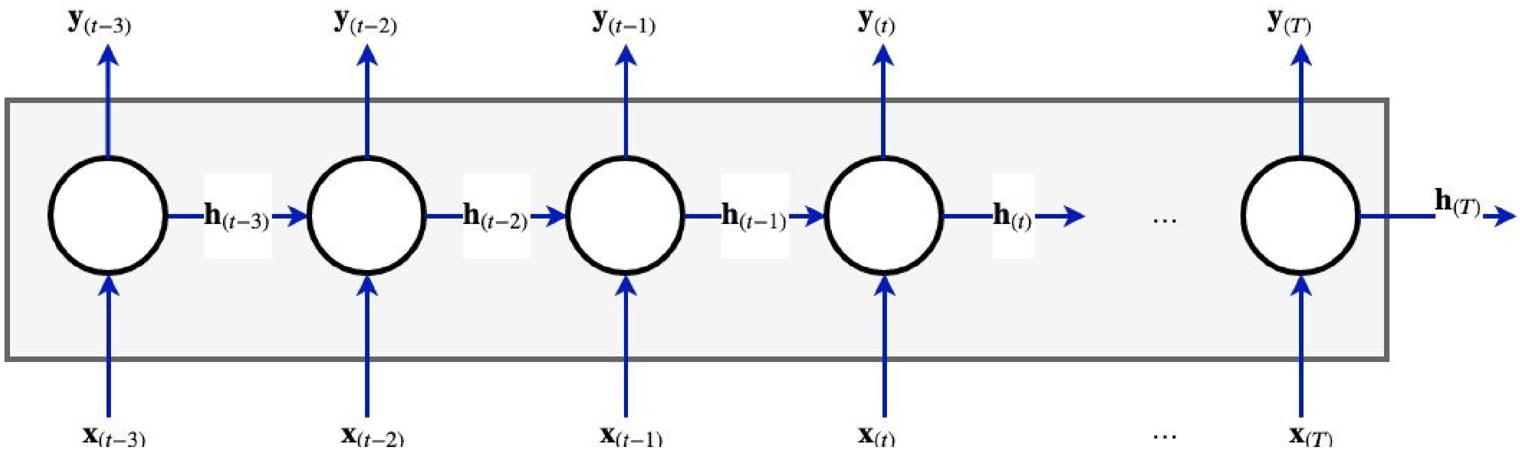
```python
def RNN( input_sequence, state_size ):
    state = np.random.uniform(size=state_size)

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

    return out
```

and a picture/movie

RNN many to many API



At each time step t

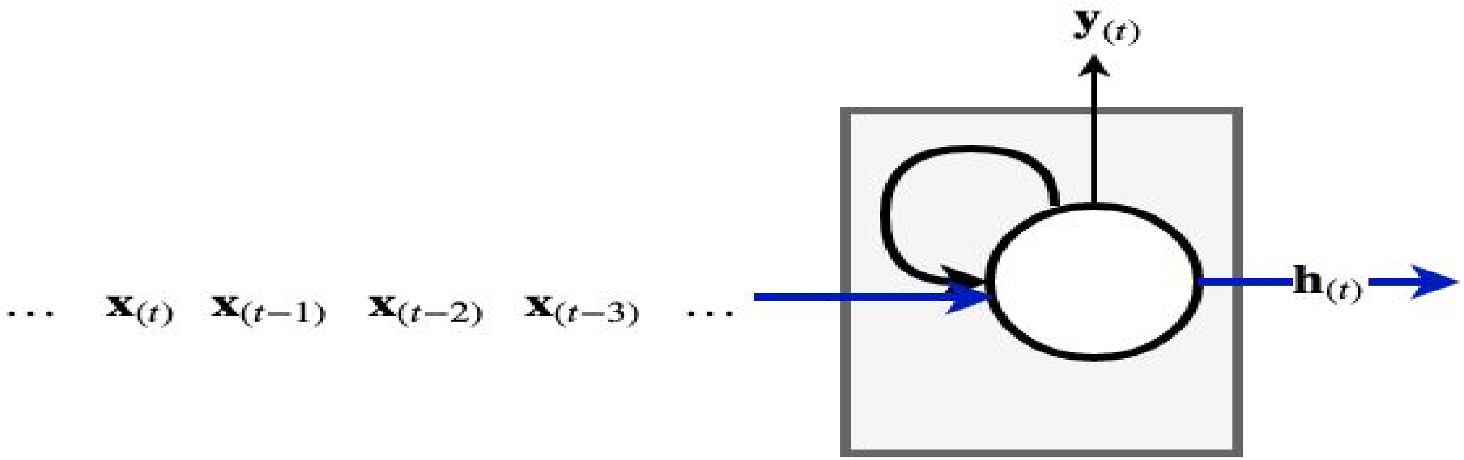
- Input $\mathbf{x}_{(t)}$ is processed
- Causes latent state $\mathbf{h}$ to update from $\mathbf{h}_{(t-1)}$ to $\mathbf{h}_{(t)}$
    - We use the same sequence notation to record the sequence of latent states $[\mathbf{h}_{(1)}, \ldots,]$
- Optionally outputs $\mathbf{y}_{(t)}$ (for outputs that 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 $\left[\mathbf{x}_{(1)} \ldots \mathbf{x}_{(t-1)}\right]$
- 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 short-hand picture for the movie that you will often see is



. . .
$\mathbf{x}_{(t)}$
$\mathbf{x}_{(t-1)}$
$\mathbf{x}_{(t-2)}$
$\mathbf{x}_{(t-3)}$
. . .
$\mathbf{y}_{(t)}$
$\mathbf{h}_{(t)}$

The movie version is a little more direct and is often referred to as "unrolling the loop" in the short-hand version.

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

- The unrolled graph looks just like an ordinary graph
- Because it resembles a non-loop computation, our logic and intuition for computing gradients transfers directly

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

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

- to record information about the entire prefix of the input.

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 $\mathbf{W}$ 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 be just a linear 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".

$\mathbf{h}_{(t)}$ latent state

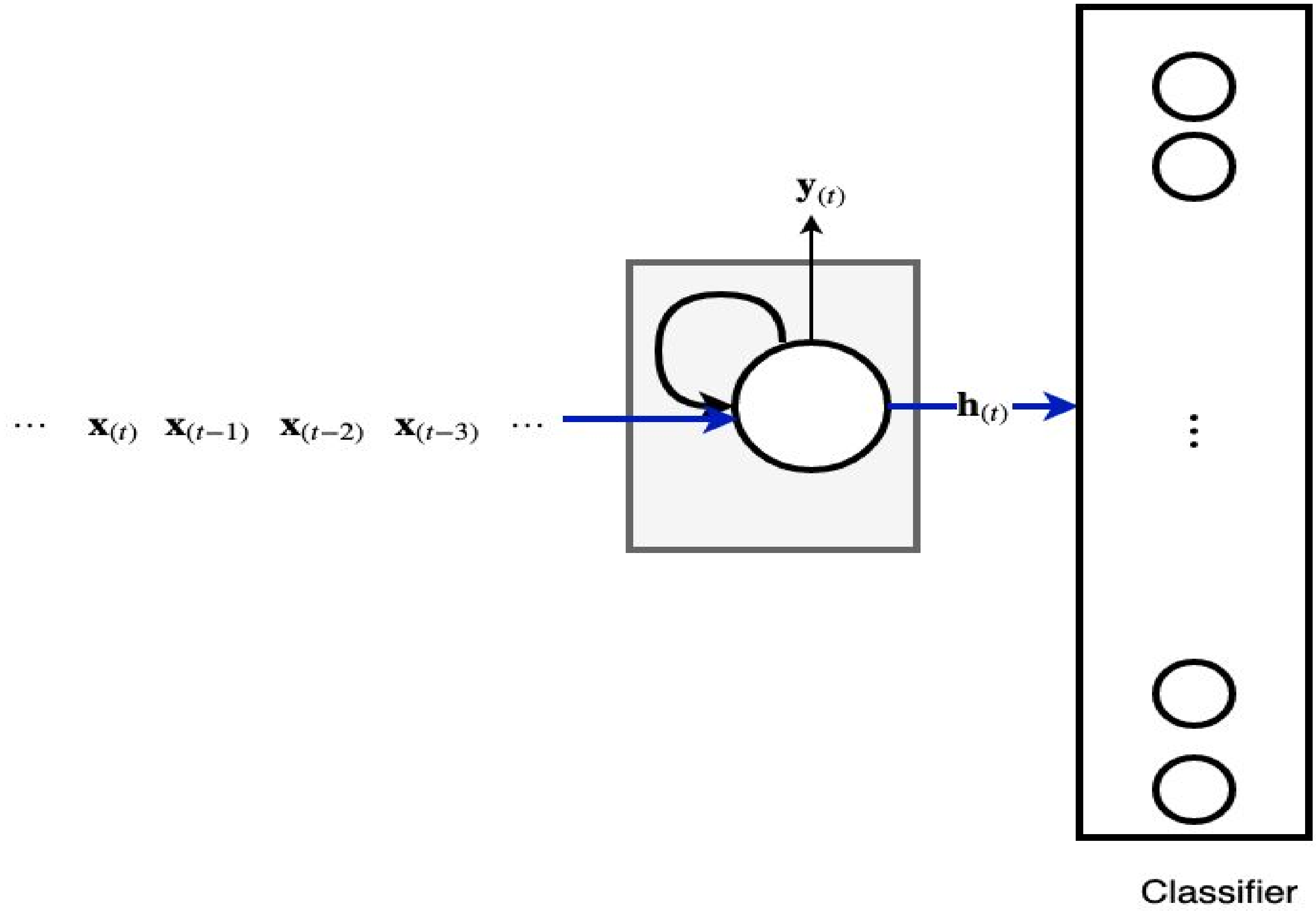
$h_{(t)}$ is the latent state (sometimes called the *hidden state* as it is not visible outside the layer).

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

- 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

Having a fixed length encoding for a variable length input is crucial

- We can feed the (fixed length representation of a) sequence to a Classical ML Classifier/Regressor
- Which have fixed length inputs


$\cdots$ $\mathbf{x}_{(t)}$ $\mathbf{x}_{(t-1)}$ $\mathbf{x}_{(t-2)}$ $\mathbf{x}_{(t-3)}$ $\cdots$
$\mathbf{y}_{(t)}$
$\mathbf{h}_{(t)}$
Classifier

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
```