Introduction

Up until now, the layers we have studed (Dense, Convolution) are primarily used to implement functions that are one to one:

- a single input (of fixed length) yields a single output.

Recurrent Neural Networks (RNN) deal with sequences:

- sequences of inputs and sequence of outputs.

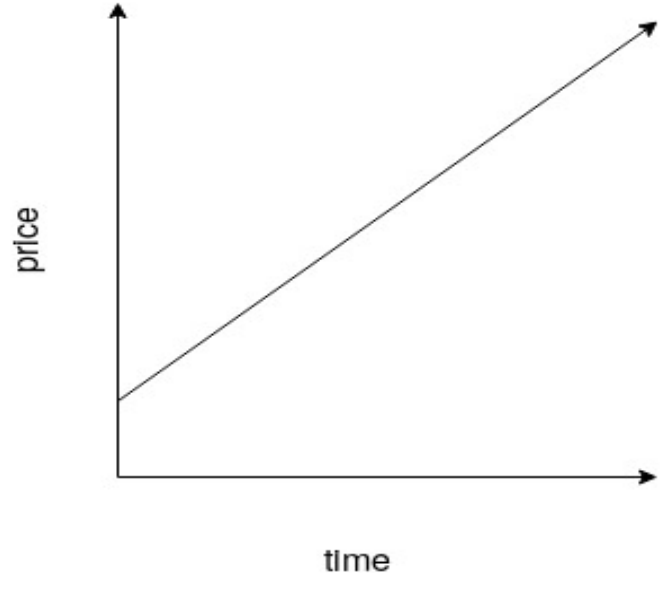
Sequences have order: a permutation of the elements of a sequence is a completely distinct sequence.

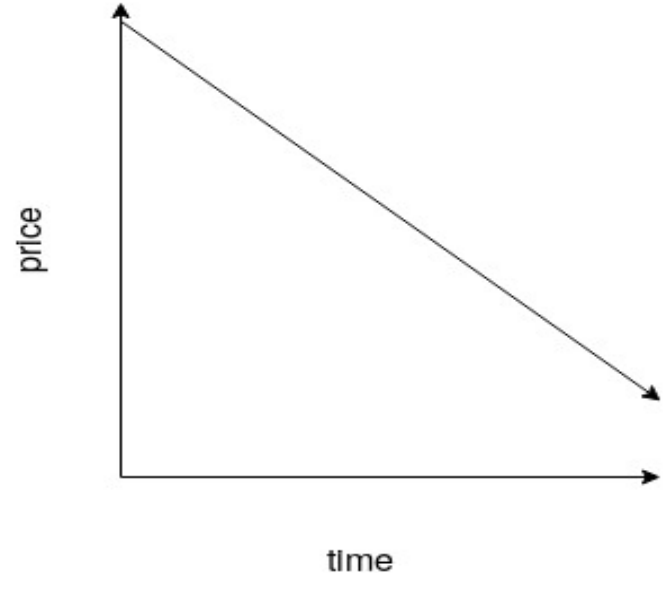
Order matters !

Order matters

- set $\{\mathbf{x}_{(1)} \ldots \mathbf{x}_{(t-1)}\}$ versus sequence $\mathbf{x}_{(1)} \ldots \mathbf{x}_{(t-1)}$
  - order doesn't matter for a set
  - $\{\mathbf{x}_{(1)} \ldots \mathbf{x}_{(t-1)}\}$
    $= \{\mathbf{x}_{t-1)} \ldots \mathbf{x}_{(1)}\}$

Same prices

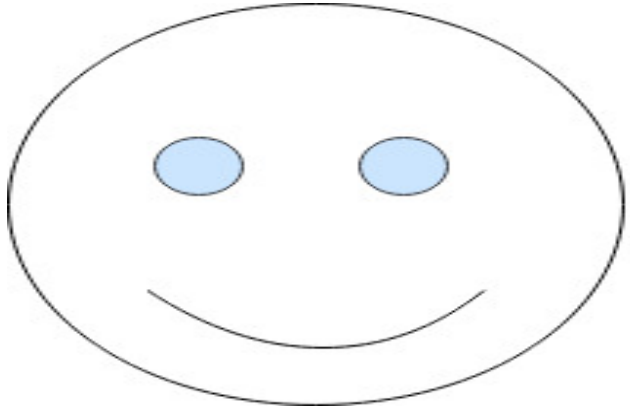


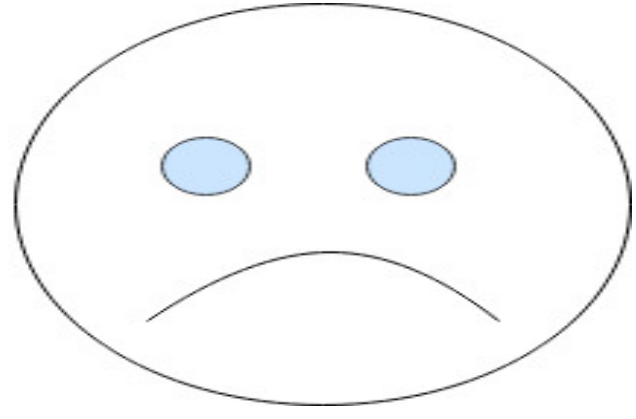


Same words

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

Same pixels





All pairs of inputs in above examples are

- identical as sets
- different as sequences

Note

If the paired examples have different labels:

- NN will find as subset of features that separate them
- the separating feature then becomes an anchor, restricting reordering

Functions on sequence

Examples of functions with sequence as inputs but single output (many to one mapping)

- predict next value in time series
- predict sentiment of text

Examples of functions with single input but sequence as output (one to many mapping)

- text generation (char-rnn) ?

Examples of functions with sequence as inputs and outputs

- translating from one language to another
- captioning a movie

Notation alert

Lot's of dimenions !

- $\mathbf{x^{(i)}}$ is a sequence with elements $\mathbf{x^{(i)}_{(1)}} \ldots \mathbf{x^{(i)}_{(t-1)}}$
- each element $\mathbf{x^{(i)}_{(t)}}$ is a vector
    - $x^{(\mathbf{i})}_{(t),j}$ is a feature
        - examples
            - $\mathbf{x^{(i)}_{(t)}}$ is a frame t in a movie; $x^{(\mathbf{i})}_{(t),j}$ is a pixel in frame t
            - $\mathbf{x^{(i)}_{(t)}}$ are the characteristics of a stock on day t, $x^{(\mathbf{i})}_{(t),j}$ is price or volume
            - $\mathbf{x^{(i)}_{(t)}}$ is a word t in a sentence; $x^{(\mathbf{i})}_{(t),j}$ is element j of the OHE of the word

Choices for how to predict $\mathbf{y}_{(t)}$ that is dependent on $\mathbf{x}_{(1)} \ldots \mathbf{x}_{(t)}$

$p(\mathbf{y}_{(t)} \mid \mathbf{x}_{(1)} \ldots \mathbf{x}_{(t)})$	direct dependence on entire prefix $\mathbf{x}_{(1)} \ldots \mathbf{x}_{(t)}$
$p(\mathbf{h}_{(t)} \mid x_{(t)}, \mathbf{h}_{(t-1)})$	latent variable $\mathbf{h}_{(t)}$encodes $\mathbf{x}_{(1)} \ldots \mathbf{x}_{(t)}$
$p(\mathbf{y}_{(t)} \mid \mathbf{h}_{(t)})$	prediction contingent on latent variable

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

The single layer may also emit an output at step i (for outputs that are sequences).

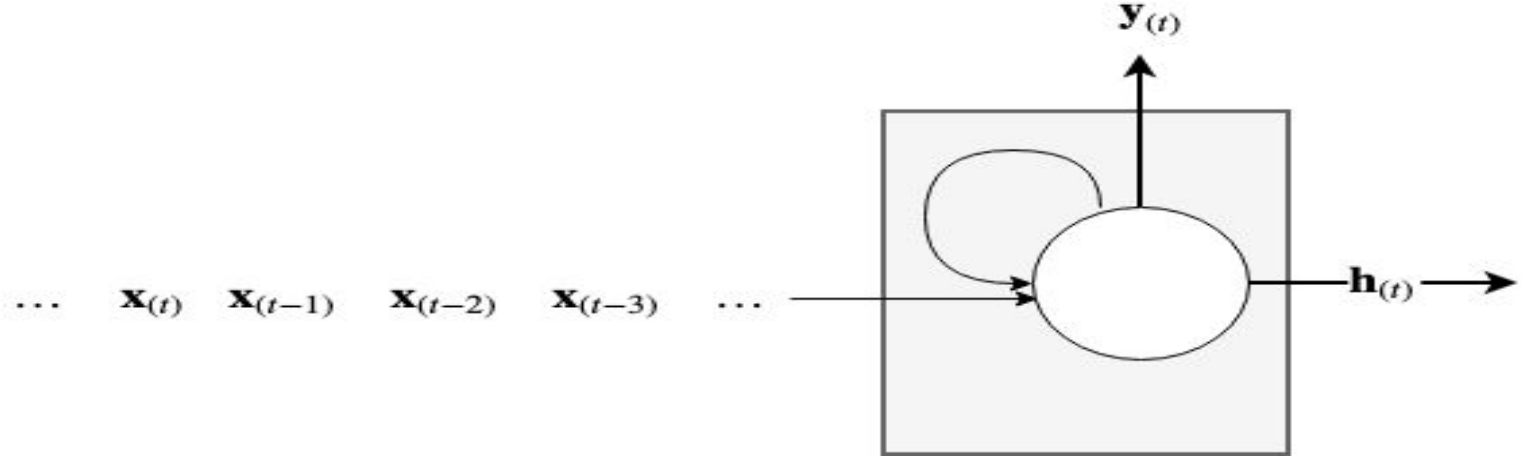
Here is some pseudo-code:

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
```


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

Note that 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.

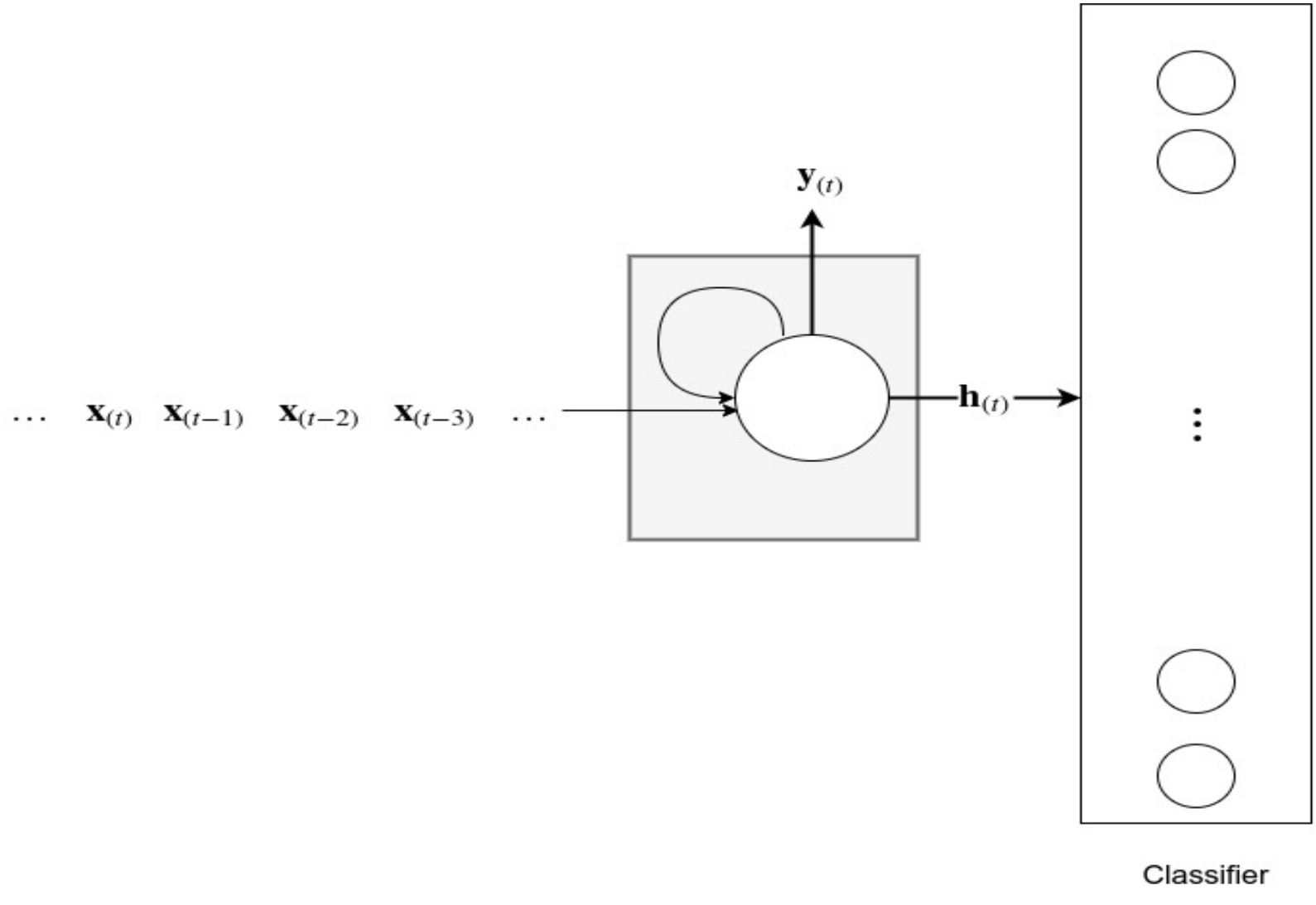
Geron does not distinguish betwee $\mathbf{y}_t$ and $\mathbf{h}_t$ and he uses the single $\mathbf{y}_{(t)}$ to denote the state.

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

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

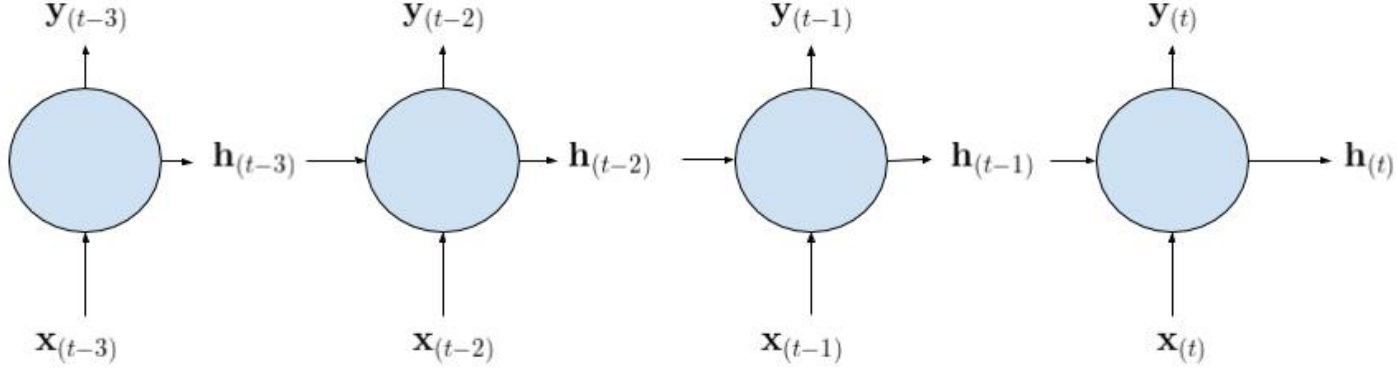
- $\mathbf{h}$ is a *fixed length* encoding of variable length sequence $\mathbf{x}_{(1)}$ ...
    - $\mathbf{h}_{(t)}$ encodes $\mathbf{x}_{(1)}$ ... $\mathbf{x}_{(t)}$
    - gives a way to have variable length input to, e.g., classifiers
- $\mathbf{h}_{(t)}$ is a vector of features
    - captures multiple "dimensions"/concepts of the input sequence

RNN Many to one; followed by classifier



Unrolling a loop

We can "unroll" the loop into the sequence of steps, with time as the horizontal axis


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

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.

The key connection is that the state at time $t-1$ is passed as input to time t.

So when processing $\mathbf{x}_{(t)}$

- the layer can take advantage of a summary ($\mathbf{h}_{(t-1)}$) of every input that preceded it.

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

Essentially, each state $\mathbf{h}$ is replicated per time step.

This view of the computation graph is important

- it shows you the exact computation
- it should tell you how gradients are computed
    - in particular, the loss and gradients flow backwards
        - so the gradients involving $\mathbf{h}$ are updated at *each* time step.
        - this will be important when we discuss
            - vanishing/exploding gradients
            - skip connections

The RNN API

During one time step of computation, the RNN computes 2 values

- new state $\mathbf{h}_{(t)}$
- output $\mathbf{y}_{(t)}$ (sometimes simply taken to be same as shor term state

The state computation is a function of the previous state $\mathbf{h}_{(t-1)}$, and the current input $x_{(t)}$.

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

Note the recursive aspect of the computation of $\mathbf{h}_{(t)}$:

- it implicitly depends on the values of the states at all previous time steps $t' < t$.

RNN as a layer

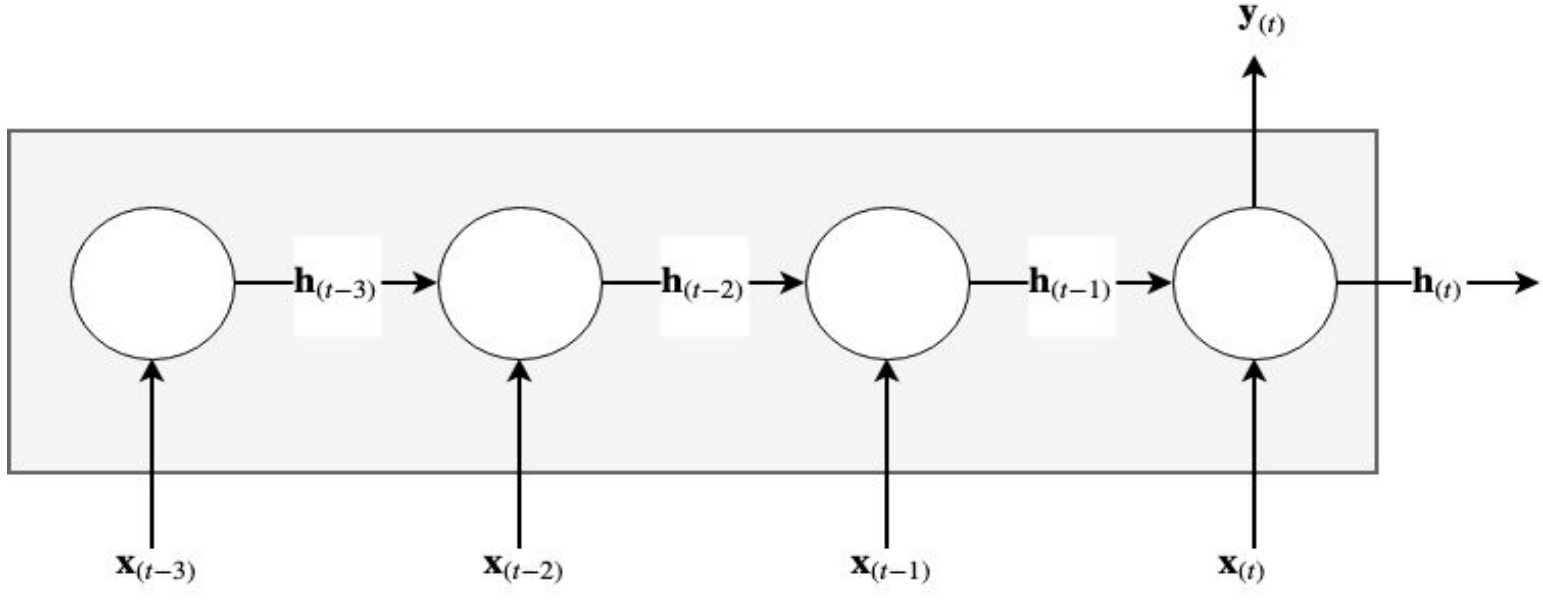
Many to one

Although the unrolled RNN looks confusing, as an "API" the RNN just acts as any other layer

- takes some input $\mathbf{x}$ (which happends to be a sequence)
- produces a single output

If we draw a box around the unrolled RNN, we can see the "API":

RNN many to one



RNN layer as an encoder

The many to one RNN essentially creates a compact encoding of an arbitrarily long sequence.

This can be very useful as we can feed this "summary" (representation) of the entire sequence into layers that can't handle sequence inputs.

Note that there is nothing special about a layer creating a compact encoding (representation) of it's input.

A CNN layer, with outputs flattened to one dimension, creates a compact encoding of an image.

The real power of the RNN is the ability to encode all sequences, regardless of length, into a fixed size representation.

Sequences: variable length input summarized

$\mathbf{h}_{(t)}$ summarizes the length t sequence $\mathbf{x}_{1,\ldots,t}$ in a *fixed size* vector $\mathbf{h}_{(t)}$.

- makes sequences amenable to models that can only deal with fixed size input

To be clear

- the RNN is a layer, just like any other
    - Internally it implements a loop but that is ordinarily hidden
    - The intuition about the "unrolled loop" is to help us to better understand the inner workings, not as a coding matter

- Like any other layer, it produces an output (although after multiple time steps for an RNN versus a single time step for a Dense layer).
- If the length of sequence $\mathbf{x}$ is T, there is ordinarily a **single** output $\mathbf{y}_{(T)}$
    - $\mathbf{y}_{(T)}$ is only available after the entire input sequence has been consumed
    - the intermediate results
      $$\mathbf{h}_{(t)}, \mathbf{y}_{(t)}, \ \ t = 1, \ldots, (T-1)$$
      are not visible through the API

Many to many

The above behavior defines a many to one mapping from input sequence (many) to single output (one).

With a minor change, we can define a many to many mapping:

- each element of the input sequence results in one element of an output sequence.

Many Deep Learning software API's will see recurrent layers with an optional argument

- `return_sequences`
- `return_states`
- both default to `False` in Keras.

This controls the output behavior of the RNN layer, whether it returns one output per time step

$$\mathbf{h}_{(1)}, \cdots, \mathbf{h}_{(T)}$$
$$\mathbf{y}_{(1)}, \cdots, \mathbf{y}_{(T)}$$

or just

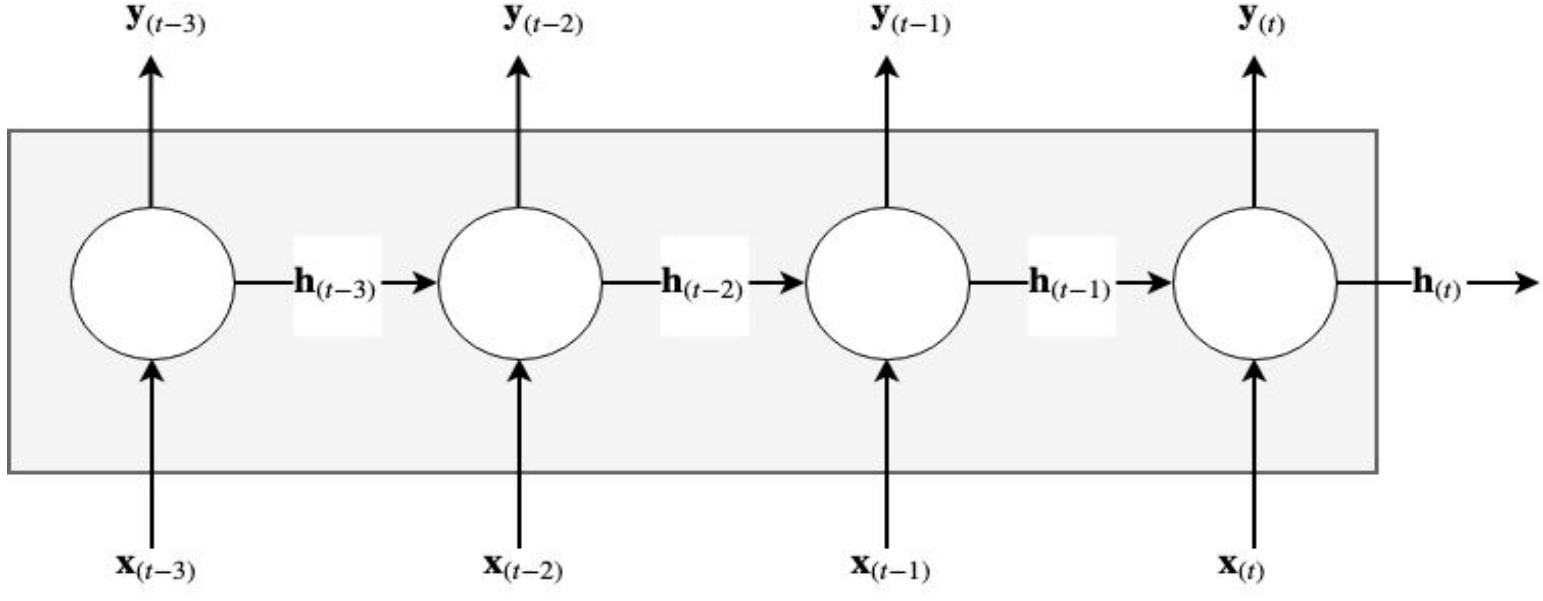
$$\mathbf{h}_{(T)}$$
$$\mathbf{y}_{(T)}$$

This is how any RNN behaves when the function it's implementing is many to many:

- one output per time step.

When the RNN needs to implement a many to one function, the layer looks like

RNN many to many



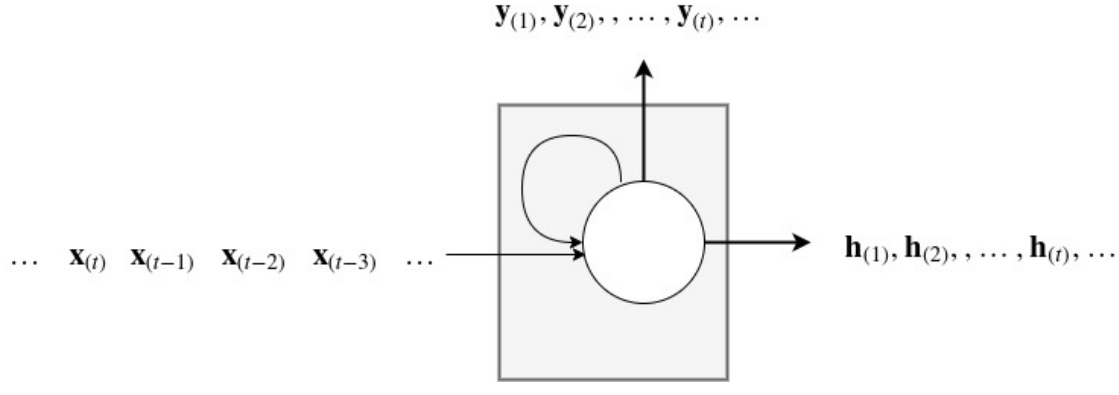
The art-work needs to be clarified

- the RNN layer produces sequences
    - as outputs $\mathbf{y}$
    - as states $\mathbf{h}$

These sequences are available when the RNN layer *completes* its consumption of input $\mathbf{x}$.

The following diagram may clarify

RNN many to many, clarified

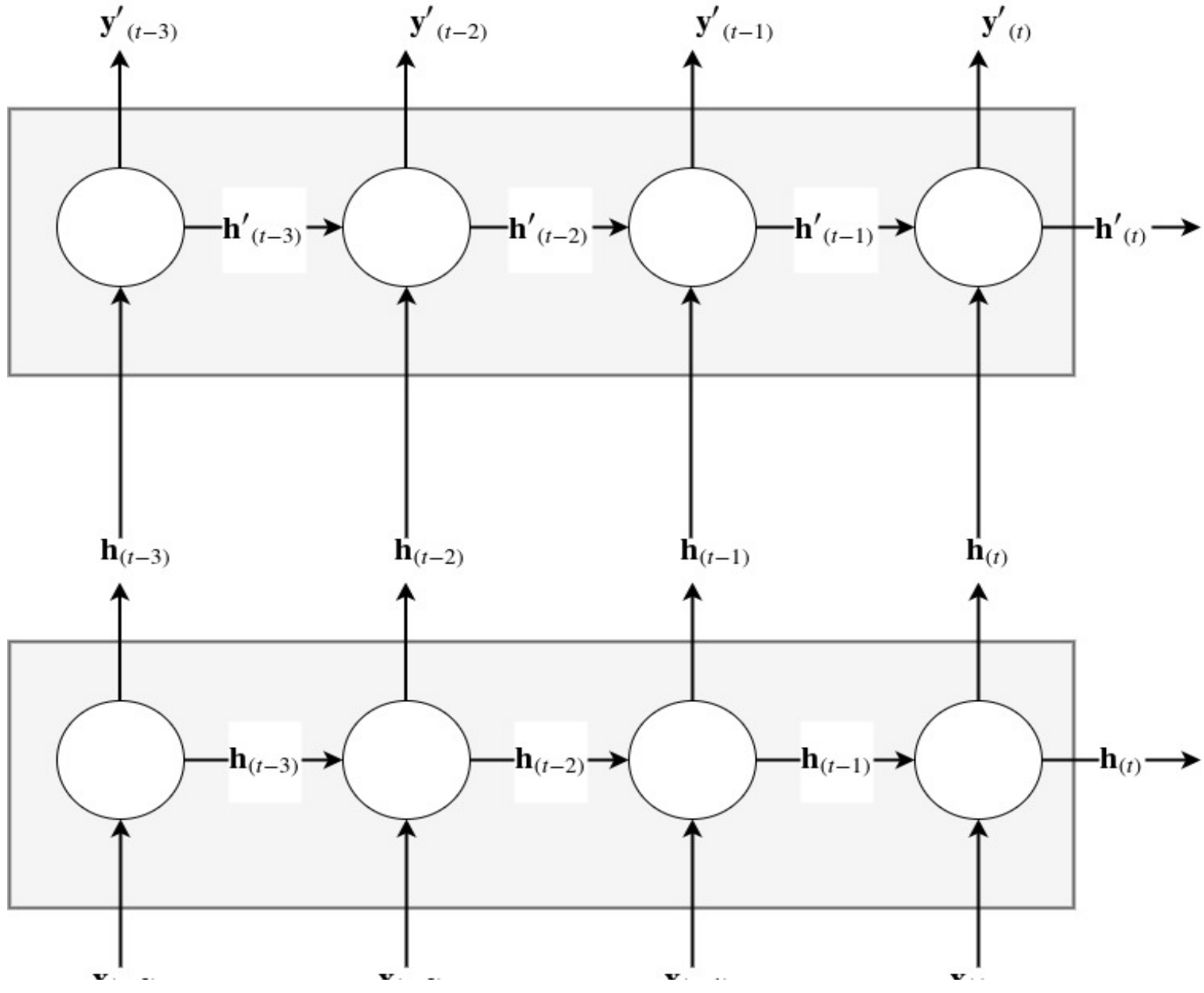


- the `return_sequences` argument instructs the layer to produce a sequence $\mathbf{y}$
    - rather than a scalar, as in the many to one case
- the `return_states` argument instructs the layer to return the state $\mathbf{h}$ as well
    - useful if we stack RNN layers

Stacked RNN layers

One can connect RNN layers into "stacks"

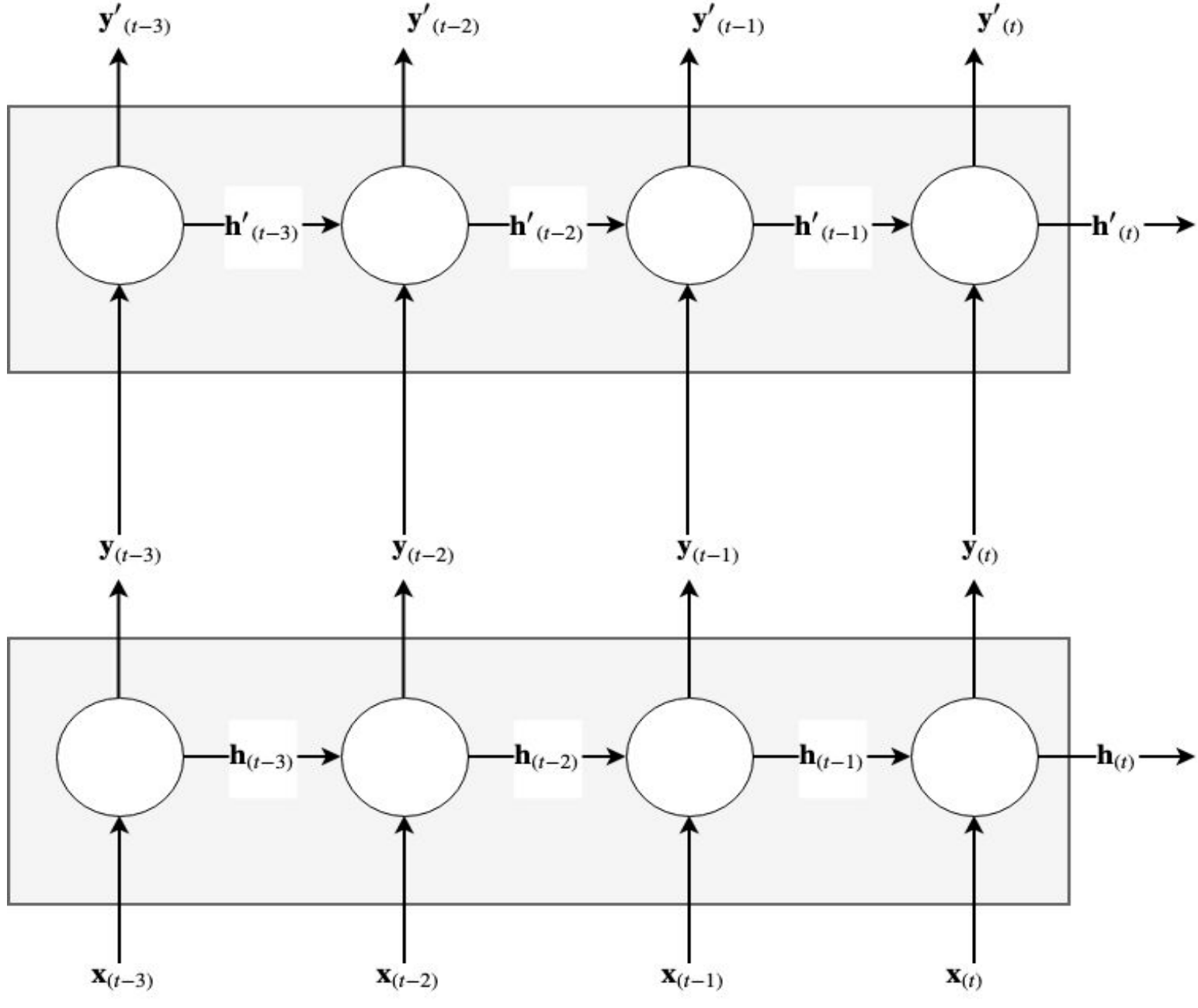
- by feeding the output state of one RNN layer as the input to the successor layer:


$\mathbf{y}'_{(t-3)}$
$\mathbf{y}'_{(t-2)}$
$\mathbf{y}'_{(t-1)}$
$\mathbf{y}'_{(t)}$
$\mathbf{h}'_{(t-3)}$
$\mathbf{h}'_{(t-2)}$
$\mathbf{h}'_{(t-1)}$
$\mathbf{h}'_{(t)}$
$\mathbf{h}_{(t-3)}$
$\mathbf{h}_{(t-2)}$
$\mathbf{h}_{(t-1)}$
$\mathbf{h}_{(t)}$
$\mathbf{h}_{(t-3)}$
$\mathbf{h}_{(t-2)}$
$\mathbf{h}_{(t-1)}$
$\mathbf{h}_{(t)}$

Encoder/Decoder architecture

An *Encoder/Decoder* is a two part Neural Network that is applied to many NLP tasks

- *Encoder* converts sequence (sentence) into intermediate representation (sequence)
- *Decoder* converts intermediate sequence to final sequence


$\mathbf{y}'_{(t-3)}$
$\mathbf{y}'_{(t-2)}$
$\mathbf{y}'_{(t-1)}$
$\mathbf{y}'_{(t)}$
$\mathbf{h}'_{(t-3)}$
$\mathbf{h}'_{(t-2)}$
$\mathbf{h}'_{(t-1)}$
$\mathbf{h}'_{(t)}$
$\mathbf{y}_{(t-3)}$
$\mathbf{y}_{(t-2)}$
$\mathbf{y}_{(t-1)}$
$\mathbf{y}_{(t)}$
$\mathbf{h}_{(t-3)}$
$\mathbf{h}_{(t-2)}$
$\mathbf{h}_{(t-1)}$
$\mathbf{h}_{(t)}$
$\mathbf{x}_{(t-3)}$
$\mathbf{x}_{(t-2)}$
$\mathbf{x}_{(t-1)}$
$\mathbf{x}_{(t)}$

Sequence to Sequence

- Many to one encoder
    - variable length input to fixed length final state $\mathbf{h}$
- One to one decoder
    - *initial* state of decoder set to *final* state of encoder
    - teacher forcing
        - input $(t+1)$ of decoder is out t of decoder

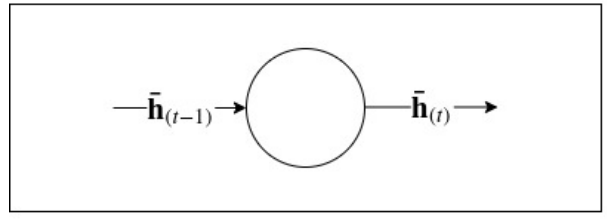
Example: language translation

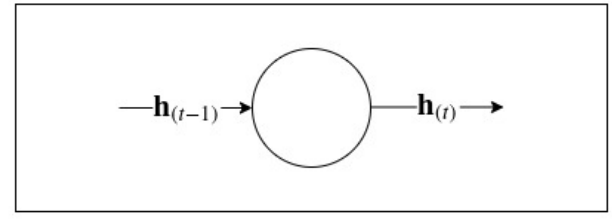
This is useful when there is not an exact one to one correspondence between tokens in the source and target languages.

Sequence to Sequence: inference

Encoder

Decoder





$\hat{\mathbf{y}}_{(1)}\hat{\mathbf{y}}_{(2)} \quad \cdots \quad \hat{\mathbf{y}}_{(t)} \quad \cdots \quad \hat{\mathbf{y}}_{(\bar{T})}$

$\mathbf{h}_{(t)}, \mathbf{s}$

$\mathbf{h}_{(0)} = \bar{\mathbf{h}}_{(T)}$

$\mathbf{s} = \bar{\mathbf{h}}_{(\bar{T})}$

$\mathbf{x}_{(1)}\ \mathbf{x}_{(2)} \quad \cdots \quad \mathbf{x}_{(\bar{T})}$

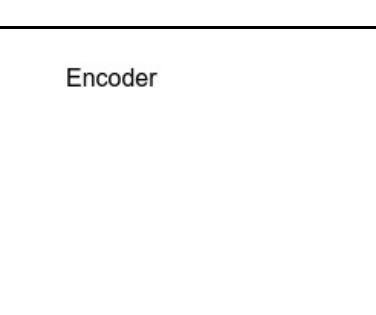
<Start> $\hat{\mathbf{y}}_{(1)}\hat{\mathbf{y}}_{(2)} \quad \cdots \quad \hat{\mathbf{y}}_{(\bar{T})}$

Sequence to Sequence: training (teacher forcing)

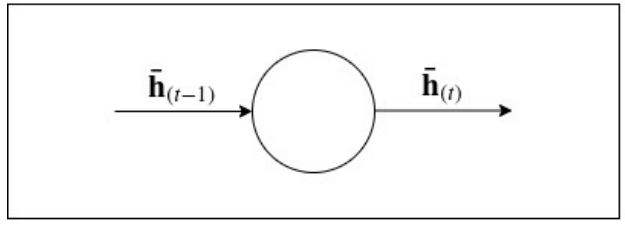
Encoder

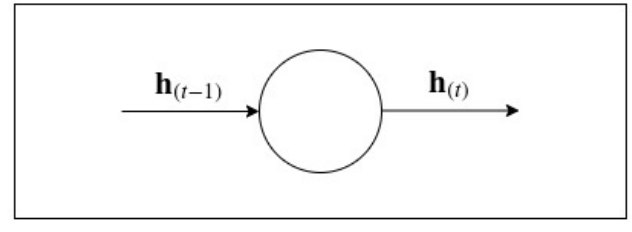
Decoder

$\hat{\mathbf{y}}_{(1)}\,\hat{\mathbf{y}}_{(2)} \quad \cdots \quad \hat{\mathbf{y}}_{(t)} \quad \cdots \quad \hat{\mathbf{y}}_{(\bar{T})}$



$\mathbf{h}_{(t)}, \mathbf{s}$





$\mathbf{h}_{(0)} = \bar{\mathbf{h}}_{(T)}$

$\mathbf{s} = \bar{\mathbf{h}}_{(\bar{T})}$

$\mathbf{x}_{(1)}\,\mathbf{x}_{(2)} \quad \cdots \quad \mathbf{x}_{(\bar{T})}$

<Start> $\mathbf{y}_{(1)}\,\mathbf{y}_{(2)} \quad \cdots \quad \mathbf{y}_{(T)}$

Training: forward/backward pass, cost/loss

Examples

- an example $\mathbf{x}^{(\mathbf{i})}$ is now a *sequence* $\mathbf{x}^{(\mathbf{i})}_{(1)}, \mathbf{x}^{(\mathbf{i})}_{(2)}, \ldots, \mathbf{x}^{(\mathbf{i})}_{(T)}$
    - variable length
    - $\mathbf{x}^{(\mathbf{i})}_{(t)}$ *may* be a vector (doesn't have to be scalar),
        - e.g., word embedding

Per example loss $\mathcal{L}^{(\mathbf{i})}$ *per time step*

- In many to many: there is a loss per time-step.
- Total loss (over which we optimize) is sum, orver time , of the loss per time step
  - $\mathcal{L}^{(\mathbf{i})} = \sum_{t=1}^{n} \mathcal{L}_{(t)}^{(\mathbf{i})}$
- In many to one: loss is single value (per example): depends on final state
  - $\mathcal{L}^{(\mathbf{i})} = \mathcal{L}_{(T)}$

RNN details: update equations

$$\begin{aligned} \mathbf{h}_{(t)} &= \phi(\mathbf{W}_{xh}\mathbf{x}_{(t)} + \mathbf{W}_{hh}\mathbf{h}_{(t-1)} + \mathbf{b}_h) \\ \mathbf{y}_{(t)} &= \mathbf{W}_{hy}\mathbf{h}_{(t)} + \mathbf{b}_y \end{aligned}$$

where ϕ is an activation function (usually $\tanh$)

Note Geron prefers right multiplying weights $\mathbf{x}_{(t)}\mathbf{W}_{xh}$ versus $\mathbf{W}_{xh}\mathbf{x}_{(t)}$

- left multiplying seems more common in literature

Note The equation is for a single example.

In practice, we do an entire minibatch so have m $\mathbf{x}'s$ given as a $(m \times n)$ matrix $\mathbf{X}$.

page 471: mention dimensions of each

Equation in pseudo-matrix form

You will often see a short-hand form of the equation.

Look at $\mathbf{h}_{(t)}$ as a function of two inputs $\mathbf{x}, \mathbf{h}_{(t-1)}$.

We can stack the two inputs into a single matrix.

Stack the two matrices $\mathbf{W}_{xh}, \mathbf{W}_{hh}$ into a single weight matrix

$$\mathbf{h}_{(t)} = \mathbf{W}\mathbf{I} + \mathbf{b}$$

with

$$\mathbf{W} = \begin{bmatrix} \mathbf{W}_{xh} & \mathbf{W}_{hh} \end{bmatrix}$$

$$\mathbf{I} = \begin{bmatrix} \mathbf{x}_{(t)} \\ \mathbf{h}_{(t-1)} \end{bmatrix}$$

Stacked RNN layers revisited

With the benefit of the RNN update equations, we can clarify how stack RNN layers works>

Let superscript $[l]$ denote a stacked layer of RNN.

So the RNN update equation for the bottom layer 1 becomes

$$\mathbf{h}_{(t)}^{[1]} = \phi(\mathbf{W}_{xh}\mathbf{x}_{(t)} + \mathbf{W}_{hh}\mathbf{h}_{(t-1)}^{[1]} + \mathbf{b}_h)$$

The RNN update equation for leyer $[l]$ becomes

$$\mathbf{h}_{(t)}^{[l]} = \phi(\mathbf{W}_{xh}\mathbf{h}_{(t)}^{[l-1]} + \mathbf{W}_{hh}\mathbf{h}_{(t-1)}^{[l]} + \mathbf{b}_h)$$

That is: the input to layer $[l]$ is $\mathbf{h}_{(t)}^{[l-1]}$ rather than $\mathbf{x}_{(t)}$

Back propagation through time (BPTT)

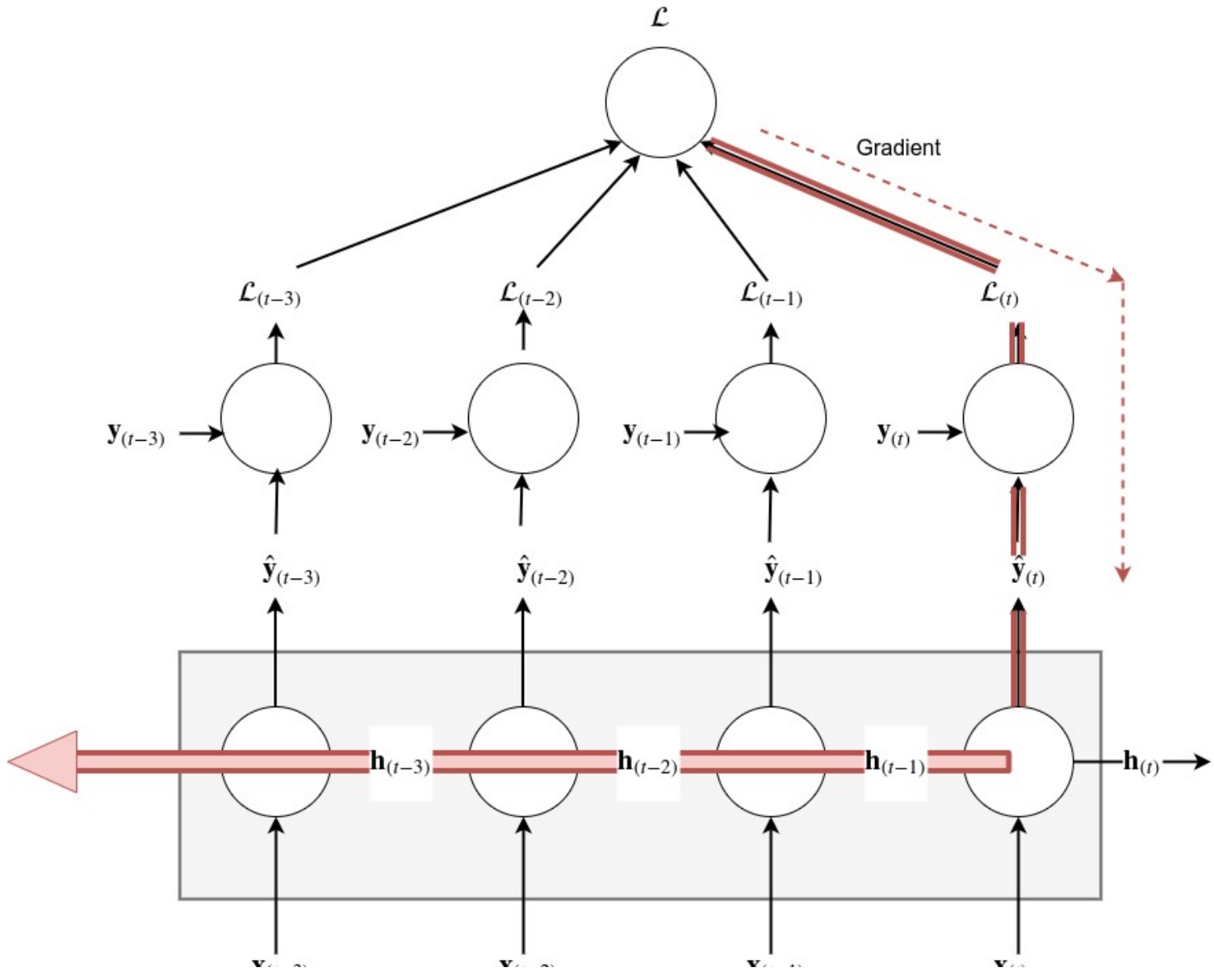
TL;DR

- We can "unroll" the RNN into a sequence of layers, one per time step
- In theory: Back Propagation on the unrolled RNN is the same as for a non-Recurrent Network
- In practice: the unrolled RNN is very deep, which causes issues in Back Propagation.

Back Propagation Through Time (BPTT) refers to

- unrolling the RNN computation into a sequence of layers
- performing ordinary Back Propagation in order to update weights

- In a non-Recurrent network:
    - $\mathbf{W}_{(l)}$, the weights of layer l, affect only layers l and greater.
    - This means the backward flow of the gradient with respect to $\mathbf{W}_{(l)}$ stops at layer l.
- In Recurrent Network:
    - All unrolled "layers" share the *same* weights
    - This means the gradients with respect to shared weight $\mathbf{W}$ must flow backward all the way to the input layer at time 0.


$\mathcal{L}$
Gradient
$\mathcal{L}_{(t-3)}$
$\mathcal{L}_{(t-2)}$
$\mathcal{L}_{(t-1)}$
$\mathcal{L}_{(t)}$
$\mathbf{y}_{(t-3)}$
$\mathbf{y}_{(t-2)}$
$\mathbf{y}_{(t-1)}$
$\mathbf{y}_{(t)}$
$\hat{\mathbf{y}}_{(t-3)}$
$\hat{\mathbf{y}}_{(t-2)}$
$\hat{\mathbf{y}}_{(t-1)}$
$\hat{\mathbf{y}}_{(t)}$
$\mathbf{h}_{(t-3)}$
$\mathbf{h}_{(t-2)}$
$\mathbf{h}_{(t-1)}$
$\mathbf{h}_{(t)}$

The unrolled graph is as deep as the length of $\mathbf{x^{(i)}}$ ($T^{(i)} = |\mathbf{x^{(i)}}|$)

- weights can update only after $T^{(i)}$ input values have been processed, so training can be slow.
- Vanishing Gradients become a concern for large $T^{(i)}$
    - Recall from the Vanishing Gradient lecture: magnitude of gradients diminishes from layer l to layer $(l-1)$ during back propagation

Calculating gradients with BPTT

Back propagation: Refresher

The same math that we used to show how to obtain derivatives (for weight updates in Gradient Descent) will apply to RNN's.

To refresh our memory on notation and results, recall our derivation of back propagation:

Layer l:

- input/output relation of layer l as
$$\mathbf{y}_{(l)} = a_{(l)}(f_{(l)}(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l)}))$$

for

- activation function $a_{(l)}$
- weights $\mathbf{W}_{(l)}$
- $\mathbf{y}_{(l-1)}$ are the outputs of the previous layer
- $f_{(l)}$ is the function computed by layer l
    - function of input $\mathbf{y}_{(l-1)}$ and weights $\mathbf{W}_{(l)}$
    - e.g., `Dense`: $f_{(l)}(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l)}) = \mathbf{y}_{(l)}$
$$= \mathbf{W}_{(l)}\mathbf{y}_{(l-1)} + \mathbf{b}_{(l)}$$

Note We neglect to add $\mathbf{b}_{(l)}$ as an argument to $f_{(l)}$ to simplify notation

Let

- $\mathcal{L}$ denote loss (computed after final layer L)
- $\mathcal{L}'_{(l)} = \frac{\partial \mathcal{L}}{\partial y_{(l)}}$ denote the derivative of $\mathcal{L}$ with respect to the output of layer l, i.e., $y_{(l)}$,
    - refer to as **loss gradient** (at output of layer l)

We showed how to compute

- $\mathcal{L}'_{(l-1)}$ from $\mathcal{L}'_l$
    - so that we can continue this process as the previous layer (i.e, *propogate loss gradient backwards*)

and we showed how to compute the weight update

- $\frac{\partial \mathcal{L}}{\partial W_{(l)}}$, from $\mathcal{L}'_{(l)}$ for $l \in [1, L]$

Note that $\mathbf{y}_{(l)}$ is a function of

- $\mathbf{y}_{(l-1)}$ (the output of the previous layer)
- and $\mathbf{W}_{(l)}$, the parameters of layer l.

We can compute derivatives of $\mathbf{y}_{(l)}$ with respect to each of its inputs

- $\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}}$
- $\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{W}_{(l)}}$

Refer to these as **local gradients**

We used the chain rule to obtain the

- gradient with respect to weights $\mathbf{W}_{(l)}$, given the loss gradient $\mathcal{L}'_{(l)}$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{(l)}} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{W}_{(l)}} = \mathcal{L}'_{(l)} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{W}_{(l)}}$$

That is:

- gradient of $\mathcal{L}$ with respect to weight $\mathbf{W}_{(l)}$
- is the loss gradient (at current step), multiplied by
- a local gradient (with respect to input $W_{(l)}$)

So we have the information required to update $\mathbf{W}_{(l)}$ by Gradient Descent.

BPTT: gradient calculation

Let us adapt these results for the case of a *single layer* RNN

- by "unrolling" this RNN, layer l is equated with "time" (of index into input sequence) t

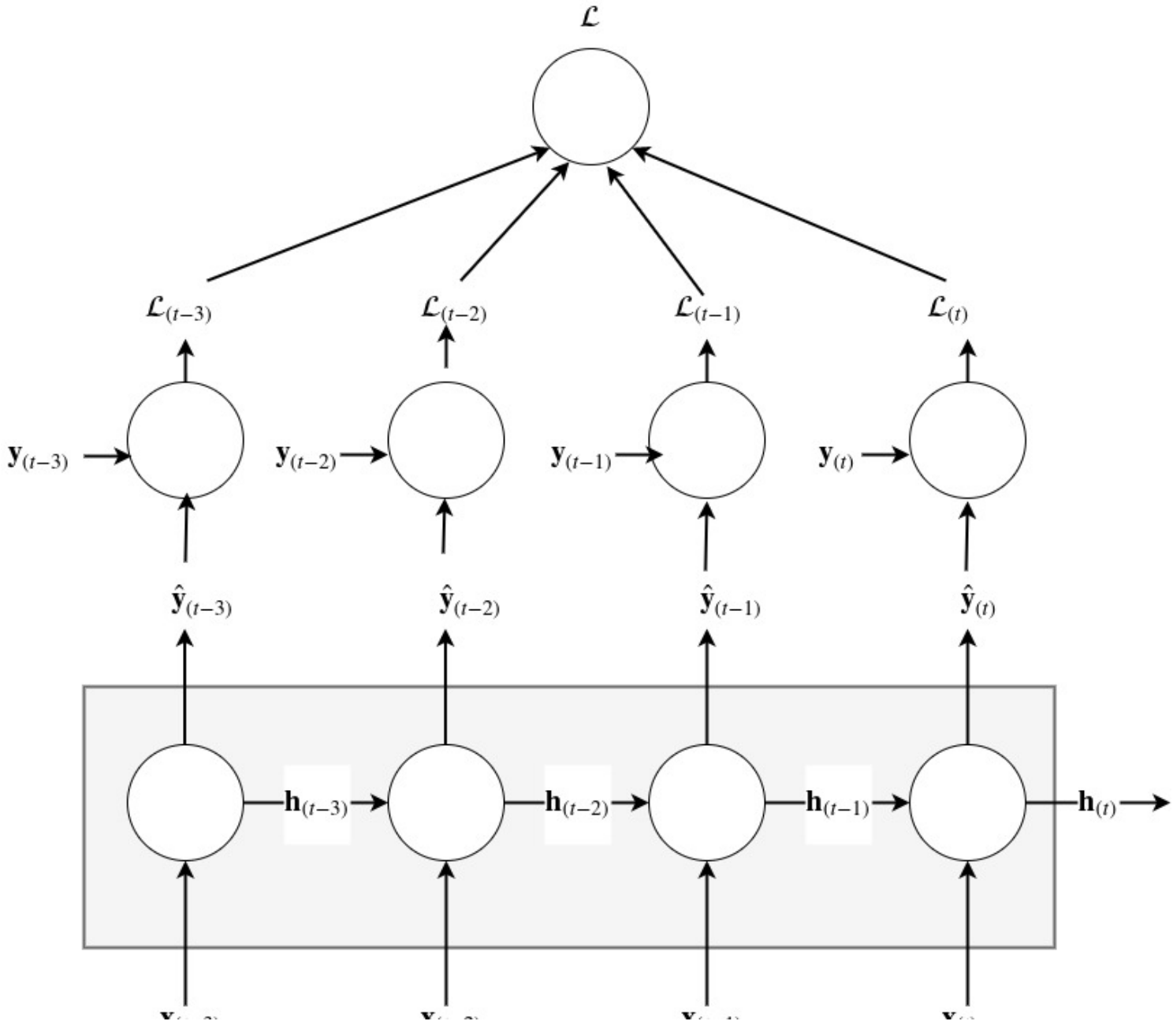
Per example loss $\mathcal{L}^{(\mathbf{i})}$ is now a per example loss *per time step*

$$\mathcal{L}^{(\mathbf{i})}_{(t)}$$

so

$$\mathcal{L}^{(\mathbf{i})} = \sum_{t=1}^{T} \mathcal{L}^{(\mathbf{i})}_{(t)}$$

We will focus on the per example loss for a single time $\mathcal{L}^{(\mathbf{i})}_{(t)}$


$\mathcal{L}$
$\mathcal{L}_{(t-3)}$
$\mathcal{L}_{(t-2)}$
$\mathcal{L}_{(t-1)}$
$\mathcal{L}_{(t)}$
$\mathbf{y}_{(t-3)}$
$\mathbf{y}_{(t-2)}$
$\mathbf{y}_{(t-1)}$
$\mathbf{y}_{(t)}$
$\hat{\mathbf{y}}_{(t-3)}$
$\hat{\mathbf{y}}_{(t-2)}$
$\hat{\mathbf{y}}_{(t-1)}$
$\hat{\mathbf{y}}_{(t)}$
$\mathbf{h}_{(t-3)}$
$\mathbf{h}_{(t-2)}$
$\mathbf{h}_{(t-1)}$
$\mathbf{h}_{(t)}$

As per regular backprop, we can obtain the loss update by multiplying the loss gradient by a local gradient

$$\frac{\partial \mathcal{L}_{(t)}^{(\mathbf{i})}}{\partial \mathbf{W}}$$

but note that we use unsubscripted $\mathbf{W}$ (rather than $\mathbf{W}_{(t)}$ because the *same* $\mathbf{W}$ is used at all timesteps.

$$\frac{\partial \mathcal{L}_{(t)}^{(\mathbf{i})}}{\partial \mathbf{W}} = \mathcal{L}'_{(t)} \frac{\partial \mathbf{y}_{(t)}}{\partial W}$$

but now

$$\frac{\partial \mathbf{y}_{(t)}}{\partial W}$$

becomes more complicated, governed by the RNN Update equations

$$
\begin{aligned}
\mathbf{h}_{(t)} &= \phi(\mathbf{W}_{xh}\mathbf{x}_{(t)} + \mathbf{W}_{hh}\mathbf{h}_{(t-1)} + \mathbf{b}_h) \\
\mathbf{y}_{(t)} &= \mathbf{W}_{hy}\mathbf{h}_{(t)} + \mathbf{b}_y
\end{aligned}
$$

Notes

- In this section we will assume ϕ is the identity function to simplify the presentation.
    - There will be no loss of generality.
- Recall that $\mathbf{W}$ is the matrix with embedded sub-matrices $\mathbf{W}_{xh}, \mathbf{W}_{hh}, \mathbf{W}_{hy}$
    - For clarity: we will add subscripts to $\mathbf{W}$ in the derivatives to show which part of $\mathbf{W}$ is the cause.

The equation defining $\mathbf{y}_{(t)}$

$$\mathbf{y}_{(t)} = \mathbf{W}_{hy}\mathbf{h}_{(t)} + \mathbf{b}_y$$

shows that $\mathbf{y}_{(t)}$ is

- directly depends on $\mathbf{W}$ (through $\mathbf{W}_{hy}$)
- *and* indirectly depends on $\mathbf{W}$ through its dependence on $h_{(t)}$ (which depends on $\mathbf{W}$)

So

$$\frac{\partial \mathbf{y}^{(\mathbf{i})}_{(t)}}{\partial \mathbf{W}} = \frac{\partial \mathbf{y}^{(\mathbf{i})}_{(t)}}{\partial \mathbf{W}_{hy}} + \frac{\partial \mathbf{y}^{(\mathbf{i})}_{(t)}}{\partial \mathbf{h}_{(t)}} \frac{\partial \mathbf{h}_{(t)}}{\partial \mathbf{W}_{hh}}$$

Let's expand the term

$$\frac{\partial \mathbf{h}_{(t)}}{\partial \mathbf{W}_{hh}}$$

Recall the recursive definition of $\mathbf{h}_{(t)}$

$$\mathbf{h}_{(t)} = \mathbf{W}_{xh}\mathbf{x}_{(t)} + \mathbf{W}_{hh}\mathbf{h}_{(t-1)} + \mathbf{b}_h$$

$\mathbf{h}_{(t)}$ depends on $\mathbf{h}_{(t-1)}$, which by recursion depends on $\mathbf{h}_{(t-2)}$ which $\ldots$ depends on $\mathbf{h}_{(0)}$.

- and all $\mathbf{h}_{(t)}$ share the *same* $\mathbf{W}_{hh}$.

This means that $\mathbf{h}_{(t)}$ depends on $\mathbf{W}$ through *each* $\mathbf{h}_{(t-k)}$ for $k = 1, \ldots, t$.

$$\frac{\partial \mathbf{h}_{(t)}}{\partial \mathbf{W}_{hh}} = \sum_{k=1}^{t} \frac{\partial \mathbf{h}_{(t-k)}}{\partial \mathbf{W}_{hh}} \frac{\partial \mathbf{h}_{(t)}}{\partial \mathbf{h}_{(t-k)}}$$

So

$$\frac{\partial \mathcal{L}_{(t)}^{(\mathbf{i})}}{\partial \mathbf{W}} = \mathcal{L}'_{(t)} \frac{\partial \mathbf{y}_{(t)}}{\partial W}$$

and

$$\frac{\partial \mathbf{y}_{(t)}^{(\mathbf{i})}}{\partial W}$$

depends on all time steps from 1 to t.

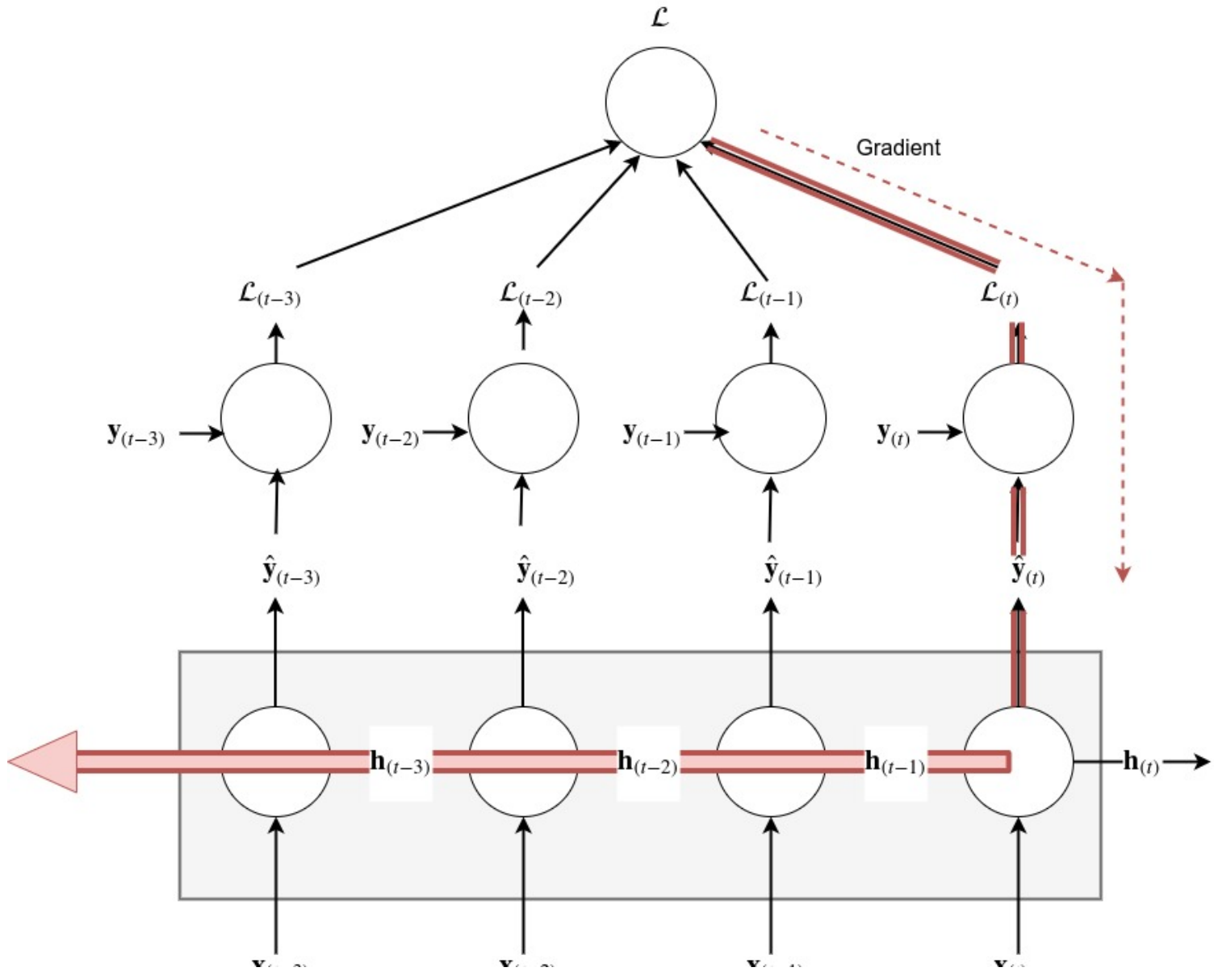
Thus, the derivative update for $\mathbf{W}$ cannot be computed without the gradient (for each time step t) flowing all the way back to time step 0.

Note

Directly expanding the recursion would show

$$\frac{\partial \mathbf{h}_{(t)}}{\partial \mathbf{h}_{(t-k)}} = \prod_{k'=0}^{k-1} \frac{\partial \mathbf{h}_{(t-k')}}{\partial \mathbf{h}_{(t-k'-1)}}$$

It is not necessary now, but will be useful in explaining vanishing/exploding gradients


$\mathcal{L}$
Gradient
$\mathcal{L}_{(t-3)}$
$\mathcal{L}_{(t-2)}$
$\mathcal{L}_{(t-1)}$
$\mathcal{L}_{(t)}$
$\mathbf{y}_{(t-3)}$
$\mathbf{y}_{(t-2)}$
$\mathbf{y}_{(t-1)}$
$\mathbf{y}_{(t)}$
$\hat{\mathbf{y}}_{(t-3)}$
$\hat{\mathbf{y}}_{(t-2)}$
$\hat{\mathbf{y}}_{(t-1)}$
$\hat{\mathbf{y}}_{(t)}$
$\mathbf{h}_{(t-3)}$
$\mathbf{h}_{(t-2)}$
$\mathbf{h}_{(t-1)}$
$\mathbf{h}_{(t)}$

Truncated back progagation through time (TBTT)

TL;DR

- We "unroll" the RNN into a sequence of T layers, one per time step
- We compute the loss at each time step t, for t=1 to T.
- The gradient of the loss of time step t flows backward for a limited number of time steps
    - Rather than flowing backwards al the way to time step 0

This is called *Truncated BPTT (TBPTT)*.

The advantage of TBPTT

- more frequent gradient updates

The disadvantage

- the loss at step t won't affect **all** previous time steps (because of truncation)
- the error signal from time t does not affect any time steps below $t - \tau$.
- this means the RNN has difficulty capturing dependencies longer than τ.

Consider a long piece of text

- The first few words indicate the gender/plurality/age of the subject
- A mis-prediction of, e.g. gender, at word $\tau' > \tau$ causes an error at time step τ'
    - which can't interact with the correct gender in the first few words

Note that there is *no truncation* of the forward pass of the RNN !

Only gradient calculations are truncated.

TBTT: Variations

There are several ways to truncate the Back Propagation.

We will describe them via a function $f(t) = t'$

- describes the earliest time step affecting the gradient of $\mathcal{L}_{(t)}$
- that is, it describes the window τ

- Untruncated BPTT
  - $f(t) = 0$
- k-truncated BPTT
  - $f(t) = \max(0, t - k)$
- subsequence truncated BPTT
  - $f(t) = k * \lfloor t/k \rfloor$

What we refer to as subsequence TBTT seems to be common

- break long sequence $\mathbf{x}^{(i)}$ into subsequences (chunks) of size k
- feed $\mathbf{x}^{(i)}$ forward as usual
    - at the end of a subsequence:
        - immediately compute the loss gradients for all time steps within the chunk

RNN vanishing/exploding gradient problem

TL;DR

- A "single-layer RNN that has been unrolled for T time steps
    - is mathematically equivalent to a simple NN with T layers
    - BUT all layers share the same weights
- This sharing of weights leads to a problem of Vanishing/Exploding gradients
    - Similar to the vanishing gradient problem we derived for simple NN
    - but with a different root cause (weight sharing)

TL;DR

- Why shared weights are different
    - Output y at time t is a function of cell state h at time t
    - Cell state h at time t is recursively defined
        - So it is a function of cell states over all times t' < t as well
        - This means the weight update involves a repeated product: (t - t') times
        - This product tends to 0 (vanishing) or infinity (explode) as (t -t') increases
    - So losses at time step t have difficulty updating gradients for the distant past<
    - RNN has difficulty with long-term dependencies

Returning to the loss gradient we encountered the terms

$$\frac{\partial \mathbf{y}_{(t)}^{(\mathbf{i})}}{\partial \mathbf{W}}$$

We will focus on the part of $\mathbf{W}$ that is $\mathbf{W}_{hh}$

$$\frac{\partial \mathbf{y}_{(t)}}{\partial W_{hh}} = \frac{\partial \mathbf{y}_{(t)}}{\partial \mathbf{h}_{(t)}} \frac{\partial \mathbf{h}_{(t)}}{\partial \mathbf{W}_{hh}}$$

But recursively defined $\mathbf{h}_{(t)}$ is a function of $\mathbf{h}_{(t-1)}, \mathbf{h}_{(t-1)}, \ldots, \mathbf{h}_{(1)}$ so

$$\frac{\partial \mathbf{y}_{(t)}}{\partial W_{hh}} = \frac{\partial \mathbf{y}_{(t)}}{\partial \mathbf{h}_{(t)}} \sum_{k=0}^{t} \frac{\partial \mathbf{h}_{(t)}}{\partial \mathbf{h}_{(t-k)}} \frac{\partial \mathbf{h}_{(t-k)}}{\partial \mathbf{W}_{hh}}$$

The summation: $\frac{\partial \mathbf{h}_{(t)}}{\partial \mathbf{W}_{hh}}$, through all intermediate $\mathbf{h}_{(t-k)}$

The problematic term for us is

$$\frac{\partial \mathbf{h}_{(t)}}{\partial \mathbf{h}_{(t-k)}}$$

It can be computed by the Chain Rule as

$$\frac{\partial \mathbf{h}_{(t)}}{\partial \mathbf{h}_{(t-k)}} = \prod_{u=0}^{t-1} \frac{\partial \mathbf{h}_{(t-u)}}{\partial \mathbf{h}_{(t-u-1)}}$$

Each term

$$\frac{\partial \mathbf{h}_{(t-u)}}{\partial \mathbf{h}_{(t-u-1)}}$$

results in a term $\mathbf{W}_{hh}$ so the repeated product compute matrix $\mathbf{W}_{hh}$ raised to the power k.

For simplicity, suppose $\mathbf{W}_{hh}$ were a scalar

- if $\mathbf{W}_{hh} < 1$ then repeatedly multiply $\mathbf{W}_{hh}$ by itself approaches 0
- if $\mathbf{W}_{hh} > 1$ then repeatedly multiply $\mathbf{W}_{hh}$ by itself approaches ∞

In other words:

- as the distance between time steps t and $(t - k)$ increases
- the gradient (for the weight update) either vanishes or explodes.

Since this term is used in the update for our weights

- updates will either be erratic (too big)
- or non-existent, hampering learning of weights.

This was not necessarily a problem in non-recurrent networks

- because each layer had a different weight matrix.

What an RNN does that helps it be parsimonius in number of parameters

- by sharing the weights across all time steps
- hurts us in learning.

For the general case where $\mathbf{W}_{hh}$ is a matrix

- we can show the same resul with the eigenvalues of the matrix

Controlling exploding gradients by clipping

In theory, we can control the explosion by clipping the gradient $\frac{\partial \mathcal{L}}{\partial W_i}$.

We are still left with the vanishing gradient problem.

This means that we can't learn long-term dependencies (i.e., too many steps backward).

This will be "solved" by introducing recurrent architectures that address this issue.

Long sequences in Tensorflow/Keras

Dealing with something as "simple" as sequences can be surprisingly difficult in Tensorflow/Keras.

- One is required to manually break up long sequences into multiple, shorter subsequences
- The ordering of the examples in a mini-batch now becomes relevant

Consider a long sequence $\mathbf{x}^{(\mathbf{i})}$ of length n.

The "natural" way to represent this $\mathbf{X}$ is

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}_{(1)}^{(1)} & \mathbf{x}_{(2)}^{(1)} & \cdots & \mathbf{x}_{(n^{(1)})}^{(1)} \\ \mathbf{x}_{(1)}^{(2)} & \mathbf{x}_{(2)}^{(2)} & \cdots & \mathbf{x}_{(n^{(2)})}^{(2)} \\ \vdots & & & \end{pmatrix}$$

for equal example lengths $n^{(1)} = n^{(2)} \ldots$

Suppose that the example lengths n is too big.

That is, they areto be broken up into subsequences of length n'.

There will be n/n' such subsequences.

We write $\mathbf{x}^{(i,\alpha)}$ to denote subsequence number α in examples i.

- The elements of this subsequence are $\mathbf{x}_{(t)}^{(i,\alpha)}$ for $1 \leq t \leq n'$.
- So $\mathbf{x}_{(1)}^{(i,\alpha+1)}$ follows $\mathbf{x}_{(n')}^{(i,\alpha)}$

TensorFlow (as of the time of this writing) has limited primitive concepts

- examples *within* batches are unrelated
- example i of one batch *can* be made to be related to example i of the following batch
    - optional flag

To get adjacent subsequences of one sequence to be treated in the proper order by TensorFlow:

- Define the number of minibatches to be n/n', which is the number of subsequences
- Each subsequence of example i should be at the *same position* within each of the n/n' minibatches
- Set RNN optional parameter `stateful=True`
- When fitting the model: set `shuffle=False`

$$\text{Minibatch } 1 = \begin{pmatrix} \mathbf{x}_{(1)}^{(1)} & \mathbf{x}_{(2)}^{(1)} & \cdots & \mathbf{x}_{(n')}^{(1)} \\ \mathbf{x}_{(1)}^{(2)} & \mathbf{x}_{(2)}^{(2)} & \cdots & \mathbf{x}_{(n')}^{(2)} \\ \vdots & & & \end{pmatrix}$$

$$\text{Minibatch 2} = \begin{pmatrix} \mathbf{x}^{(1)}_{(n'+1)} & \mathbf{x}^{(1)}_{(n'+2)} & \cdots & \mathbf{x}^{(1)}_{(n'+n')} \\ \mathbf{x}^{(2)}_{(n'+1)} & \mathbf{x}^{(2)}_{(n'+2)} & \cdots & \mathbf{x}^{(2)}_{(n'+n')} \\ \vdots & & & \end{pmatrix}$$

`stateful=True`

- TensorFlow *will not** reset the state of the RNN at the beginning of a new minibatch (for each example in the minibatch)

This means that

- the example at position i' within each minibatch share state ($\mathbf{h}$)
- we've ordered the minibatches in the correct ordering of subsequences
- so the result is that the entirety of example i's time steps update the same state

The main difference from the simple method (one example with a long sequence)

- we've broken the example up into subsequences
- **that are related by common position within adjacent minibatches**

`shuffle=False`

- Examples in adjacent minibatches must be in the *same* order, so don't shuffle them !

Sequences: Variable length

There are lots of small potholes one encounters with sequences.

What is the examples of my training set have widely varying lengths ?

- Within a batch, short examples may behave differently than long examples:
    - Maybe learn less in short examples, noisier gradient updates
- Padding sequences to make them equal length
    - Pad at the start ? Or at the end ?

The general advice is to arrange your data so that an epoch contains examples of similar lengths.

- You may require multiple fittings, one per length

Residual connections: a gradient highway

Deep Residual Learning (https://arxiv.org/abs/1512.03385)

TL;DR

- We have encountered the Vanishing Gradient problem several times.
- This is a major impediment to training deep (many layers) networks.
- The solution is to give the gradient a path to flow backward undiminished.
- This simple solution is called a Skip or Residual Connection

Consider two layers of a NN

$$\begin{aligned} \mathbf{y}_{(l-1)} &= a_{(l-1)} \left(f_{(l-1)}(\mathbf{y}_{(l-2)}) \right) \\ \mathbf{y}_{(l)} &= a_{(l)} \left(f_{(l)}(\mathbf{y}_{(l-1)}) \right) \end{aligned}$$

Suppose we modify layer $l+1$ by adding $\mathbf{y}_{(l-1)}$ to the output

$$\begin{aligned}\mathbf{y}_{(l-1)} &= a_{(l-1)}\left(f_{(l-1)}(\mathbf{y}_{(l-2)})\right)\\ \mathbf{y}_{(l)} &= a_{(l)}\left(f'_{(l)}(\mathbf{y}_{(l-1)})\right)+\mathbf{y}_{(l-1)}\end{aligned}$$

If the original and modified 2 layer mini-networks compute the same function from $\mathbf{y}_{(l-1)}$ to $\mathbf{y}_{(l+1)}$

$$\begin{aligned} a_{(l)}\left(f_{(l)}(\mathbf{y}_{(l-1)})\right) &= a_{(l)}\left(f'_{(l)}(\mathbf{y}_{(l-1)})\right) + \mathbf{y}_{(l-1)} \\ a_{(l)}\left(f'_{(l)}(\mathbf{y}_{(l-1)})\right) &= a_{(l)}\left(f_{(l)}(\mathbf{y}_{(l-1)})\right) - \mathbf{y}_{(l-1)} \end{aligned}$$

In other words:

- we have forced the modified second layer to learn the "residual" of the unmodified layer with respect to $\mathbf{y}_{(l-1)}$.

This seems strange (and pointless) until you consider the Back Propagation process.

Recall how the loss gradient

$$\mathcal{L}'_{(l)} = \frac{\partial \mathcal{L}}{\partial y_{(l)}}$$

propagates backwards

$$\mathcal{L}'_{(l-1)} = \mathcal{L}'_{(l)} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}}$$

In the unmodified mini-NN the local derivative

$$\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}} = \frac{a_{(l)}\ \left(f_{(l)}(\ldots)\right)}{\partial \mathbf{y}_{(l-1)}}$$

whereas, in the modified mini-NN the local derivative becomes

$$\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}} = \frac{a_{(l)}\ \left(f'_{(l)}(\ldots)\right)}{\partial \mathbf{y}_{(l-1)}} + 1$$

When $\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}}$ is multiplied by $\mathcal{L}'_{(l)}$ to obtain $\mathcal{L}'_{(l-1)}$

- the "upstream" loss gradient $\mathcal{L}'_{(l-1)}$
- flows backwards to layer $l-1$ unmodulate *because of the* $+1$ term in the modified local derivative.

This simple trick vanquishes the vanishing gradient !

It is one of the major reasons that we are able to train extremely deep NN's.

There is another important implication:

- adding an additional layer cannot result in increased loss

This is because there exists a set of weights $\mathbf{W}_{(l)}$ for which

$$a_{(l)}\left(f'_{(l)}(\mathbf{y}_{(l-1)})\right) = 0$$

This means the modified second layer computes the identity functon

$$\mathbf{y}_{(l)} = \mathbf{y}_{(l-1)}$$

So if adding the second layer had the potential of increasing loss relative to a one layer network

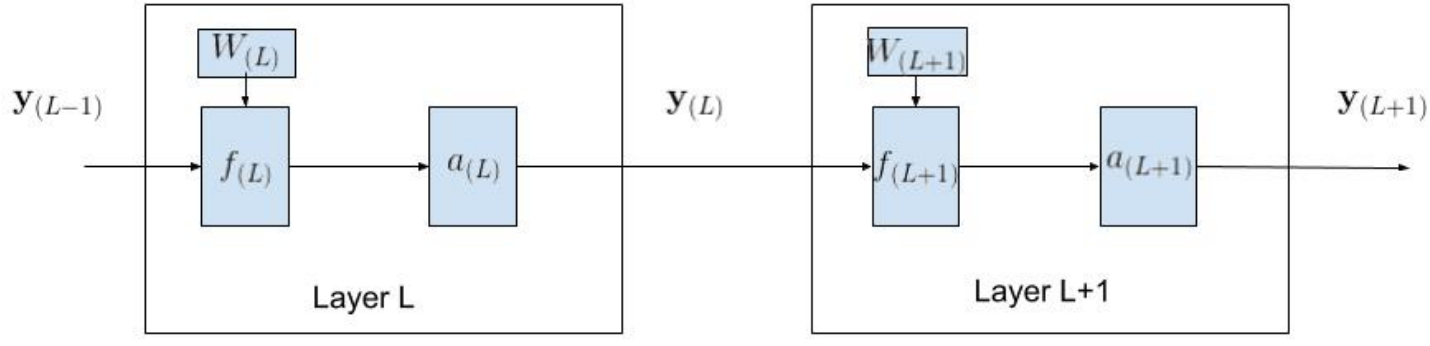
- the optimization would learn the identity function instead, an no increase in loss rsults-

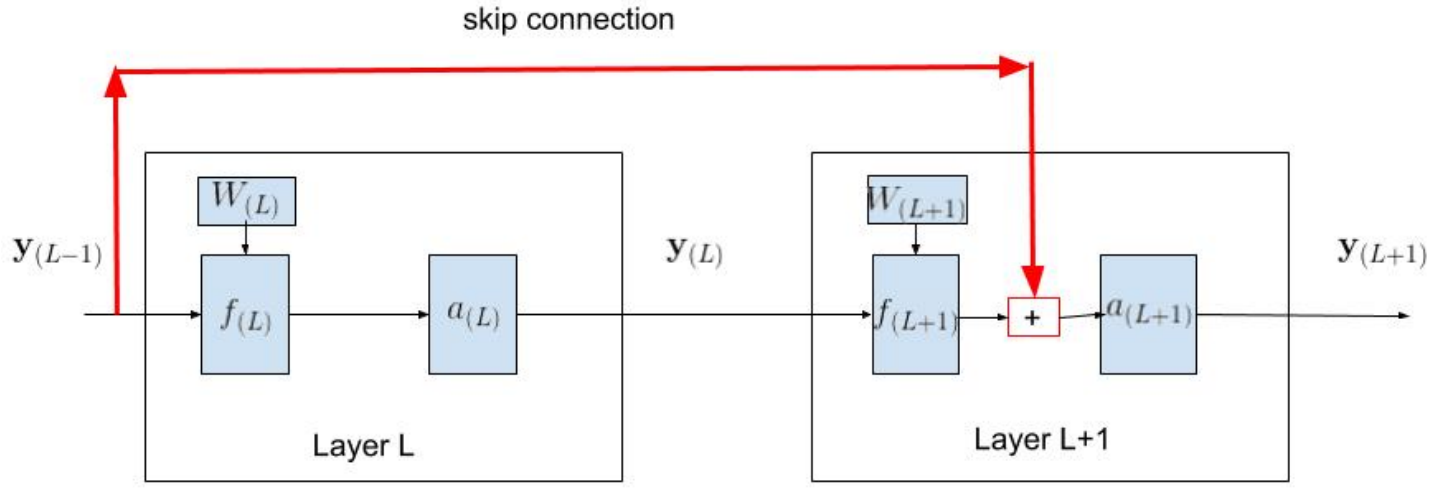
Without the skip connection, it is empircally difficult for a NN to learn an identity function as a layer.

There is an unresolved debate where to place the "head" of the skip connection

- insider the activation function
- outside the activation function

We choose the latter to simplify the derivative expression for the loss gradient.


$\mathbf{y}_{(L-1)}$
$W_{(L)}$
$f_{(L)}$
$a_{(L)}$
Layer L
$\mathbf{y}_{(L)}$
$W_{(L+1)}$
$f_{(L+1)}$
$a_{(L+1)}$
Layer L+1
$\mathbf{y}_{(L+1)}$


skip connection
$W_{(L)}$
$\mathbf{y}_{(L-1)}$
$f_{(L)}$
$a_{(L)}$
$\mathbf{y}_{(L)}$
$W_{(L+1)}$
$f_{(L+1)}$
+
$a_{(L+1)}$
$\mathbf{y}_{(L+1)}$
Layer L
Layer L+1

Preview: skip connections in LSTM's, GRU's

The gradient highway also turns out to be useful in RNN's.

There are more powerful variants of the RNN called LSTM and GRU which avoid vanishing gradients, partially through the use of skip connections.

These variants enhance the power of skip connections by allowing selective skipping via the use of "gates".

We will see this shortly.

Visualization of RNN hidden state

Here is a visualization (http://karpathy.github.io/2015/05/21/rnn-effectiveness/#visualizing-the-predictions-and-the-neuron-firings-in-the-rnn) of single elements within the hidden state, as they consume the input sequence of *single characters*.

The color reflects the intensity (value) of the paricular cell (blue=low, red=high)

Cell that turns on inside comments and quotes:

```
/* Duplicate LSM field information.  The lsm_rule is opaque, so
 * re-initialized. */
static inline int audit_dupe_lsm_field(struct audit_field *df,
                                       struct audit_field *sf)
{
  int ret = 0;
  char *lsm_str;
  /* our own copy of lsm_str */
  lsm_str = kstrdup(sf->lsm_str, GFP_KERNEL);
  if (unlikely(!lsm_str))
    return -ENOMEM;
  df->lsm_str = lsm_str;
  /* our own (refreshed) copy of lsm_rule */
  ret = security_audit_rule_init(df->type, df->op, df->lsm_str,
                                 (void **)&df->lsm_rule);
  /* Keep currently invalid fields around in case they
   * become valid after a policy reload. */
  if (ret == -EINVAL) {
    pr_warn("audit rule for LSM \'%s\' is invalid\n",
      df->lsm_str);
    ret = 0;
  }
  return ret;
}
```

Cell that is sensitive to the depth of an expression:

```
#ifdef CONFIG_AUDITSYSCALL
static inline int audit_match_class_bits(int class, u32 *mask)
{
  int i;
  if (classes[class]) {
    for (i = 0; i < AUDIT_BITMASK_SIZE; i++)
      if (mask[i] & classes[class][i])
        return 0;
  }
  return 1;
}
```

Cell that might be helpful in predicting a new line. Note that it only turns on for some “)”:

```
char *audit_unpack_string(void **bufp, size_t *remain, si
{
  char *str;
  if (!*bufp || (len == 0) || (len > *remain))
    return ERR_PTR(-EINVAL);
  /* Of the currently implemented string fields, PATH_MAX
   * defines the longest valid length.
   */
  if (len > PATH_MAX)
    return ERR_PTR(-ENAMETOOLONG);
  str = kmalloc(len + 1, GFP_KERNEL);
  if (unlikely(!str))
    return ERR_PTR(-ENOMEM);
  memcpy(str, *bufp, len);
  str[len] = 0;
```

RNN as a generative model

Up to now, an RNN's inputs were a prespecificed vector $\mathbf{x}$.

For each example during training, one element of $\mathbf{x}$ was fed into the RNN per time-step.

Similarly for inference time.

This behavior is characteristic of a discriminative network.

Consider: Suppose there were **no** inputs (or more precisely: a very short sequence $\mathbf{x}$ of length t', used to "prime" the RNN).

Instead, let's set the input at time step t to be the output of step $(t-1)$

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

for $t > t'.

Then the RNN would be self-perpetuating, never exhausting its inputs, and generating new outputs *conditional* on previous outputs !

This would be a generative form of the RNN.

Training by teacher forcing

One way to train this RNN is via a supervised task

- given sequence $\mathbf{x}$ up to time t: $\mathbf{x}_{(1,\ldots t)}$
- target is $\mathbf{x}_{(t+1)}$

This is just ordinary supervised training with a specially constructed input derived from a single sequence $\mathbf{x}$.

Of course, we would do this for a training set with many sequences, as usual.

This is similar to a classifier where the class we are trying to predict is the class of the next input element.

The only "trick" is that, at step t, the RNN may output the "wrong" value $\hat{\mathbf{x}} \neq \mathbf{x}_{(t+1)}$. If we fed the wrong $\hat{\mathbf{x}}$ as the next input to the RNN during training, the RNN would remain permanently off-track and never learn.

Instead, during *training*, the next input to the RNN

- is **forced** to be the correct input (**is the input at step t $\mathbf{x}_{(t+1)}$ or is it $\mathbf{x}_{(t+1)}$

This type of supervised learning is called *teacher forcing*.

During *inference* time, we feed back as input whatever the generated output is.

Sampling

We have described a deterministic generation process.

This would be pretty boring, lacking variety

- as well as being problematic for generalization
- we would be encouraging the RNN to memorize inputs.

In producing the single output, what is really happening is

- our classifier has one logit per class
- we arbitarily decide to pick the largest.

But we can properly view the (post-softmax) output

- as a probability vector (elements sum to 1).

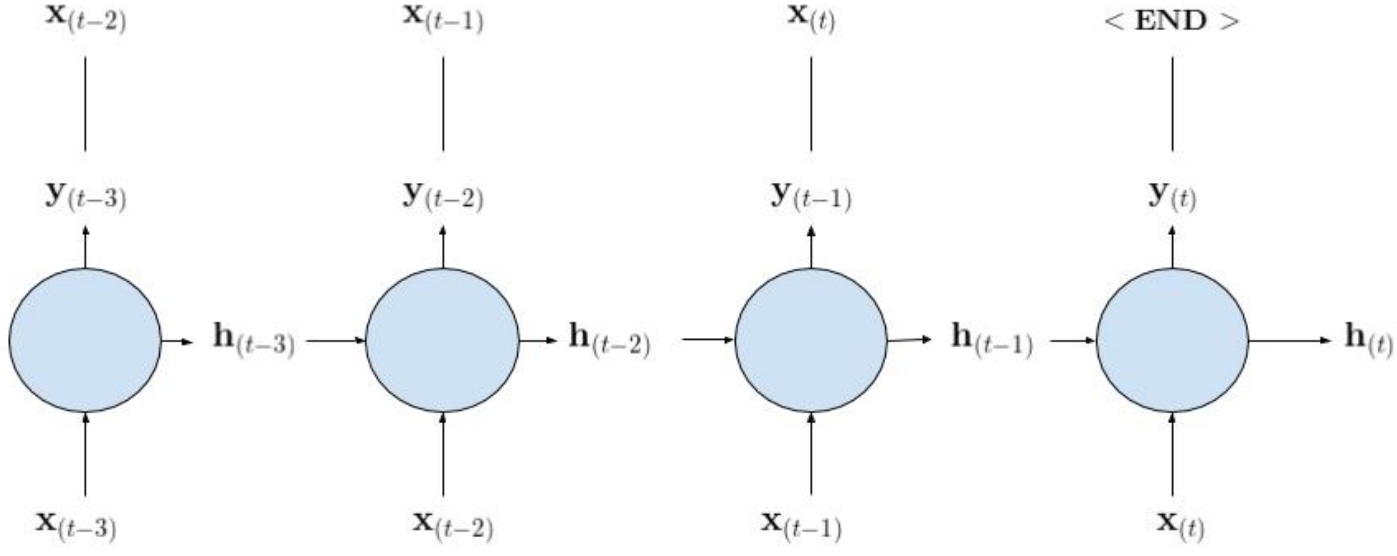
Instead of choosing the class with maximum probability

- we can sample from the probability space defined by this vector
- e.g., if the probability for class c_1 is twice as great as that for class c_2, the probability of sampling c_1 would be twice as great.

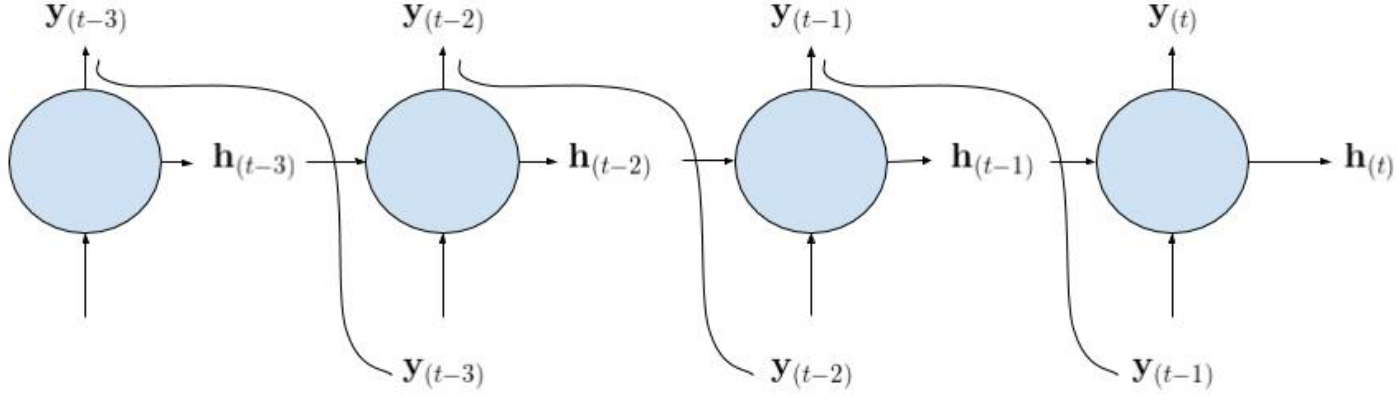
There is still some chance that c_2 is sampled, unlike the deterministic case.

If we do this, the generator can create output sequences different from any training example.

Training targets:



Inference


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

Generating strange things

We haven't specified what each element in sequence $\mathbf{x}$ is.

For text, $x_{(t)}$ could be either a character or a word, for example.

You'll be surprised how successful an RNN can be when it's task is to consume sequences of characters and predict the next character.

Although it hasn't been expicitly programmed to generate valid words, punctuation, etc., it tends to produce realistic text !

Another interesting fact: these "character RNN's" also learn semantically meaningful constructs

- the need for nested things to match
    - multi-level paranthetical phrases, e.g., "(this is (very important) I think)"
    - opening/closing markup
    - indentation/un-indentation of code blocks

This suggests that the hidden state may be learning to "count" certain concepts.

As we will see in a visualization of the hidden state, and in how LSTM's work, this may in fact be true.

RNN's of this type were quite popular and have been used to generate

- Fake Shakespeare (http://karpathy.github.io/2015/05/21/rnn-effectiveness/#shakespeare), or fake politician-speak
- Fake code
- Fake math textbooks (http://karpathy.github.io/2015/05/21/rnn-effectiveness/#algebraic-geometry-latex)
- Click bait headline generator (http://clickotron.com/about)

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

```
Done
```