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.

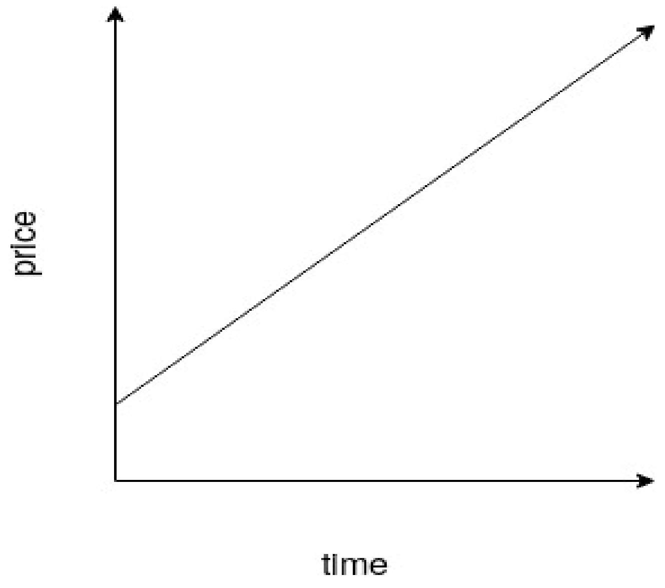
Sequences have order: a permutation of the elements of a sequence is a completely distinct sequence.

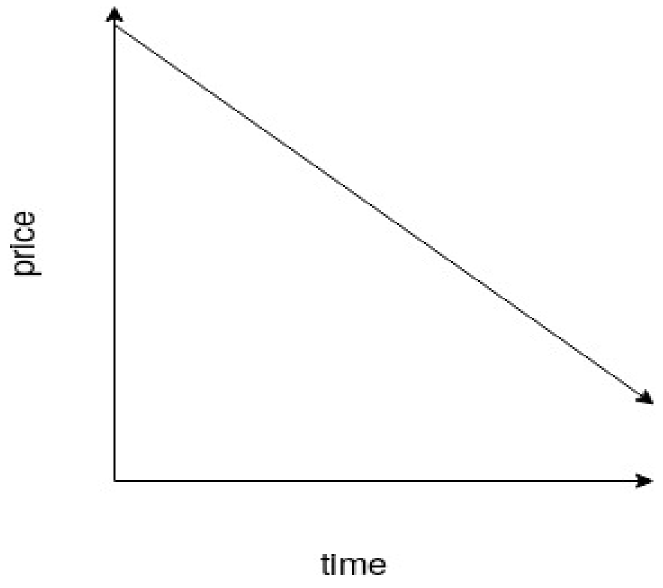
Order matters !

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

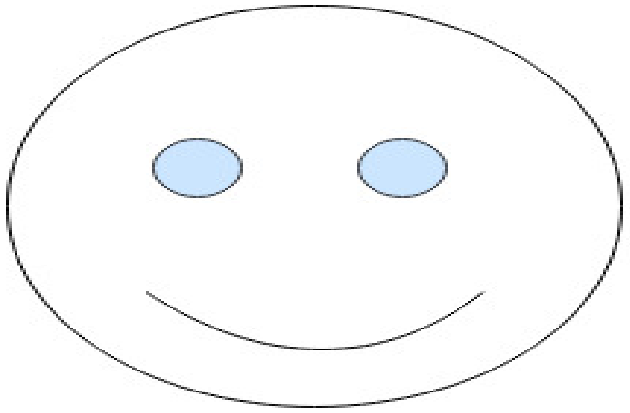


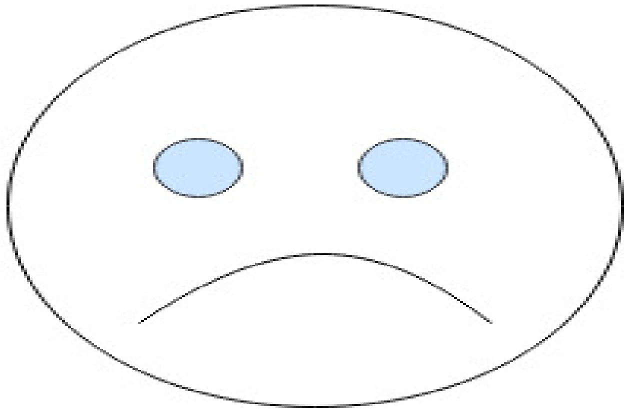


Same words

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

$$p(\mathbf{y}_{(t)} | \mathbf{x}_{(1)} \ldots \mathbf{x}_{(t)}) \quad \text{direct dependence on entire prefix } \mathbf{x}_{(1)} \ldots \mathbf{x}_{(t)}$$

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

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

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

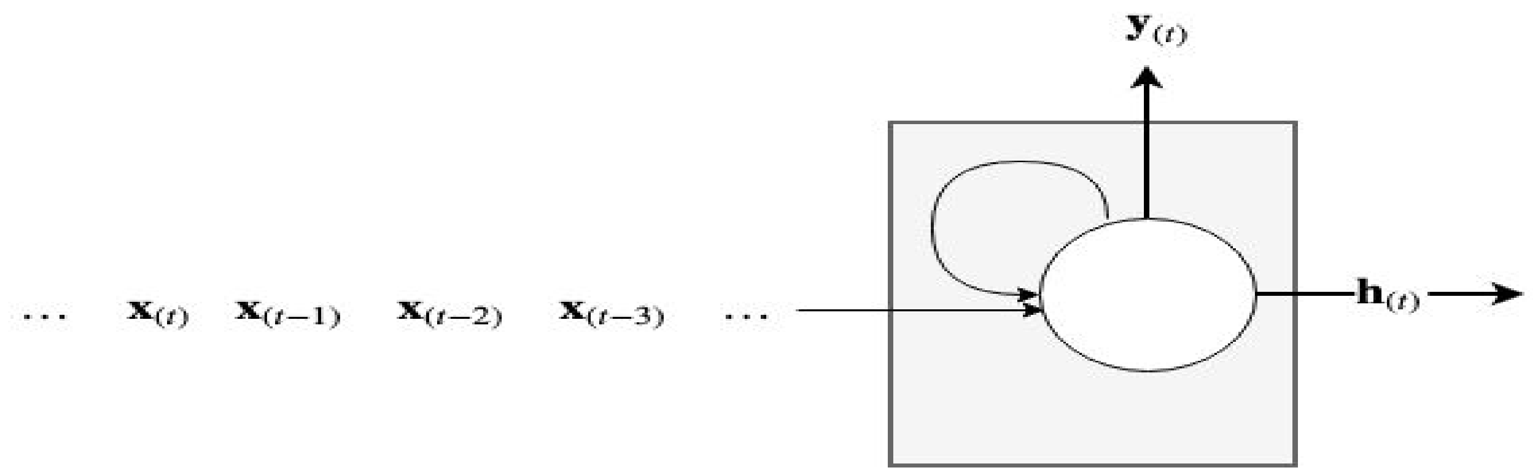
Here is some pseudo-code:

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
```

RNN



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

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.

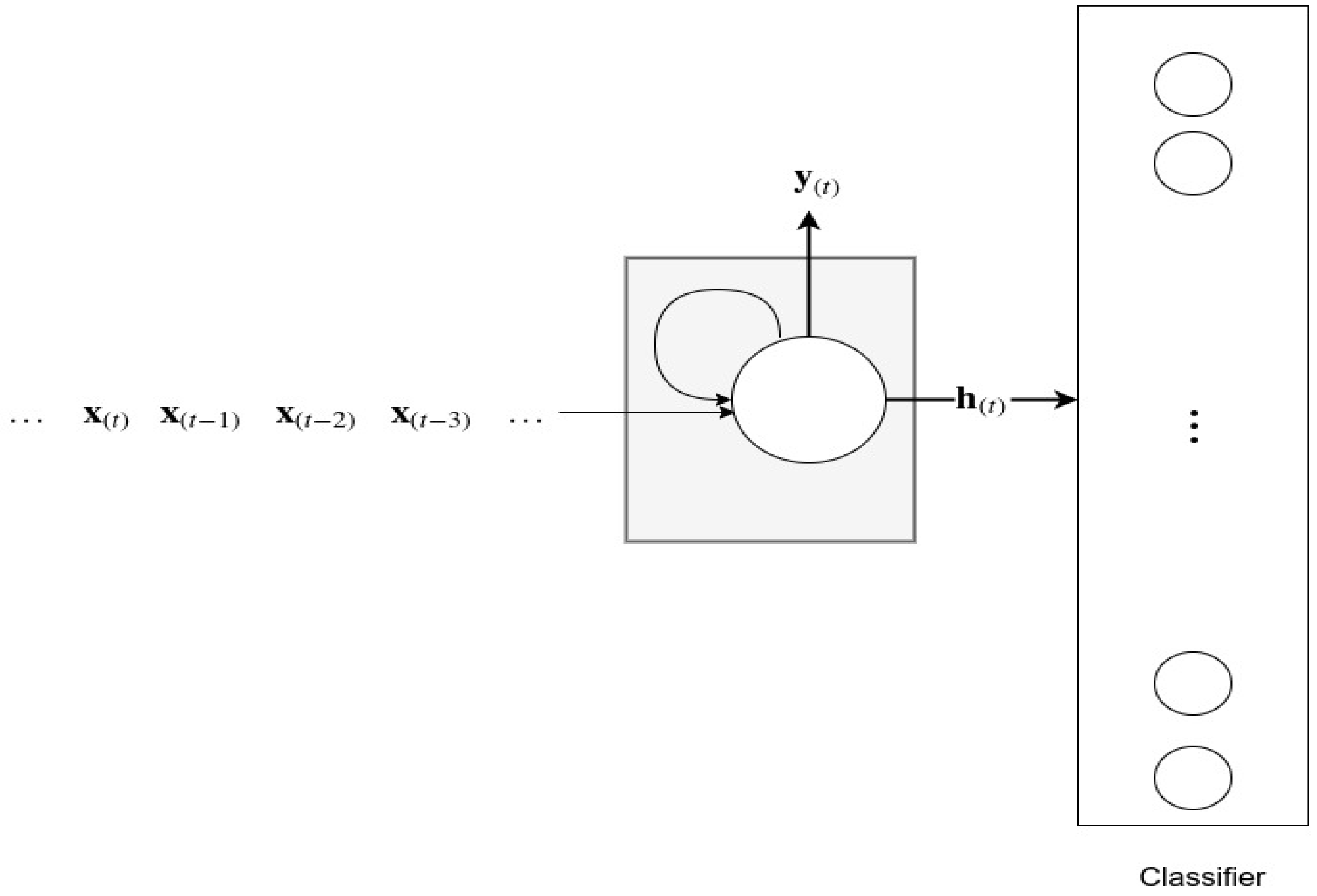
Geron does not distinguish betwee $\mathbf{y}_t$ and $\mathbf{h}_t$ and he uses the single $\mathbf{y}_{(t)}$ to denote the state.

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 $\mathbf{x}_{(1)} \ldots$
    - $\mathbf{h}_{(t)}$ encodes $\mathbf{x}_{(1)} \ldots \mathbf{x}_{(t)}$
    - 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

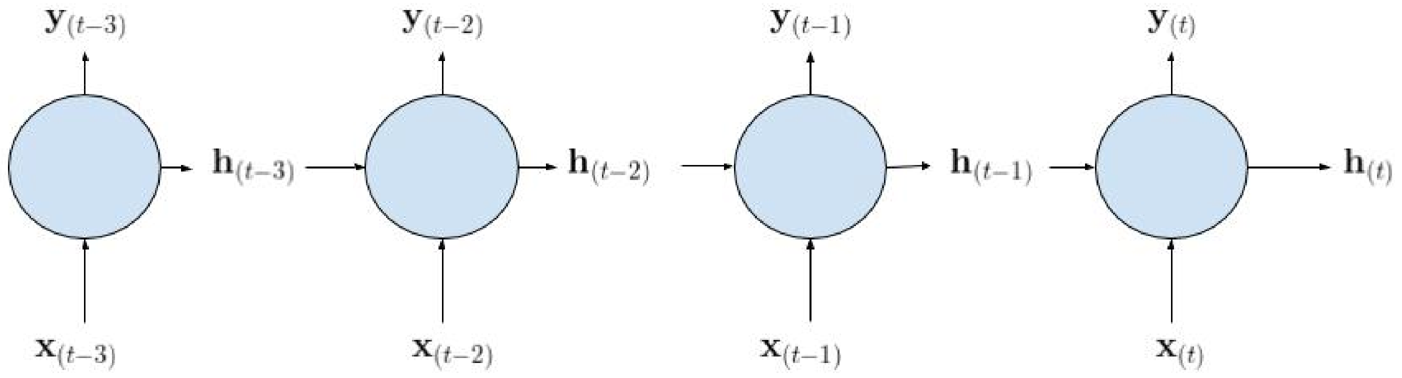
RNN Many to one; followed by classifier


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

Unrolling a loop

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

RNN unrolled



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

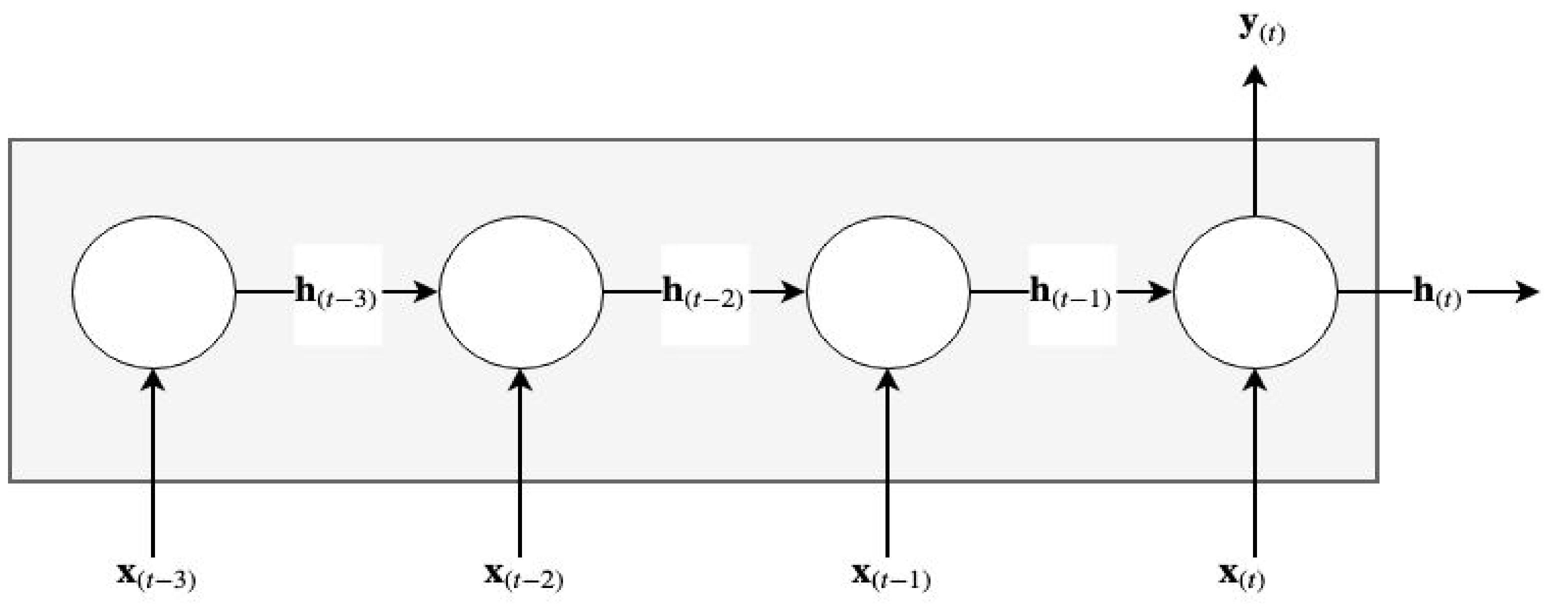
Many to one

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

$$\begin{aligned}&\mathbf{h}_{(1)}, \ldots, \mathbf{h}_{(T)}\\&\mathbf{y}_{(1)}, \ldots, \mathbf{y}_{(T)}\end{aligned}$$

or just

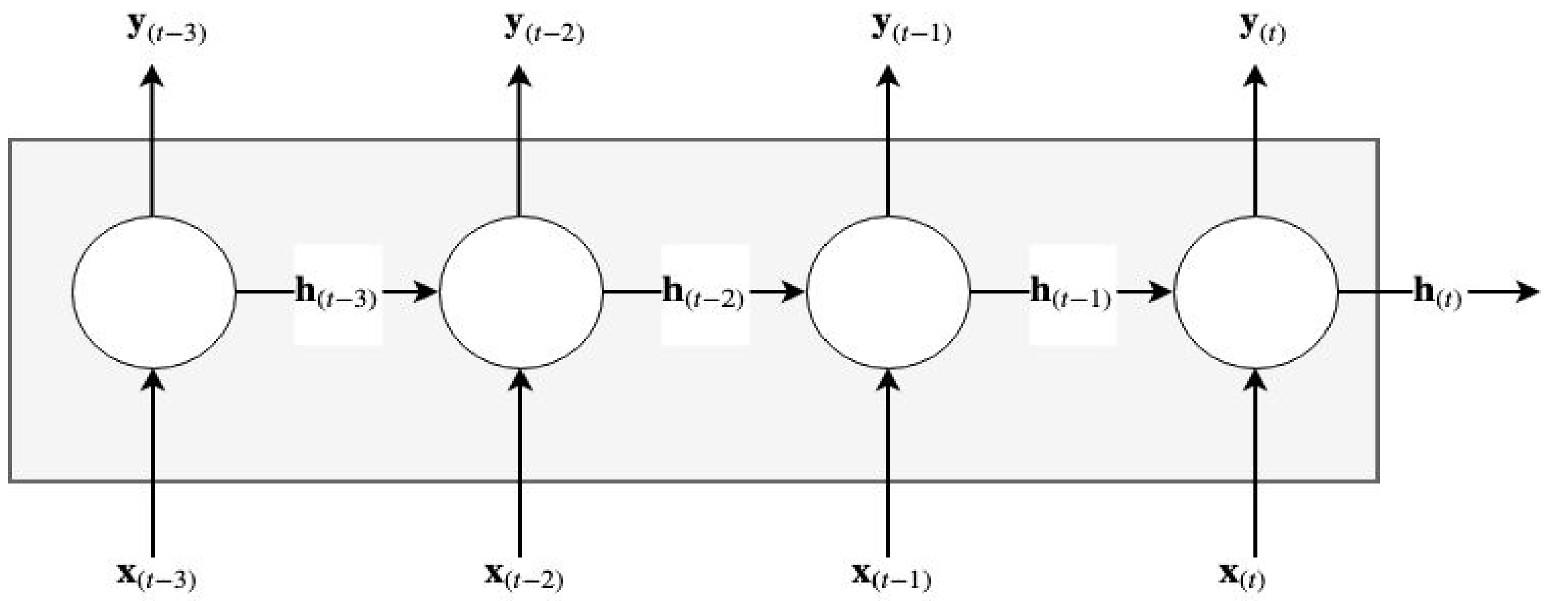
$$\begin{aligned}&\mathbf{h}_{(T)}\\&\mathbf{y}_{(T)}\end{aligned}$$

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



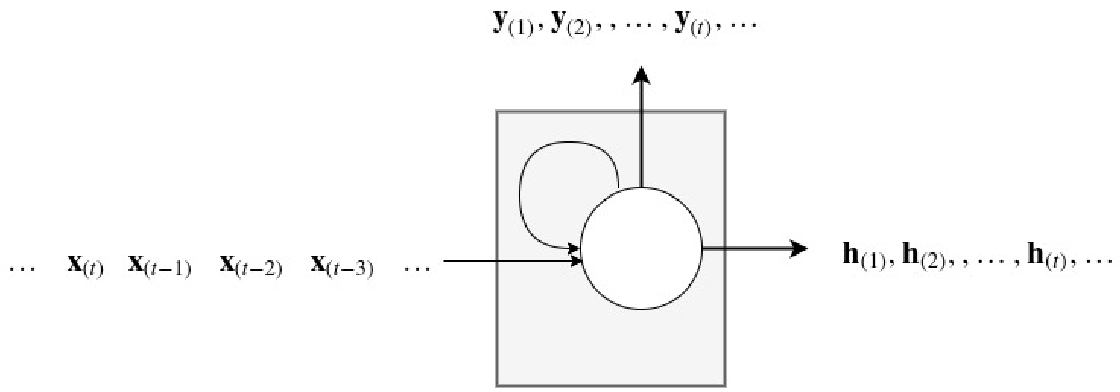
The art-work needs to be clarified

- 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


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

- 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

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

$$
\begin{aligned}
\mathbf{h}_{(t)} &= \mathbf{W}\mathbf{I} + \mathbf{b} \\
\text{with} \\
\mathbf{W} &= \begin{bmatrix} \mathbf{W}_{xh} & \mathbf{W}_{hh} \end{bmatrix} \\
\mathbf{I} &= \begin{bmatrix} \mathbf{x}_{(t)} \\ \mathbf{h}_{(t-1)} \end{bmatrix}
\end{aligned}
$$

Back propagation through time (BPTT)

TL;DR

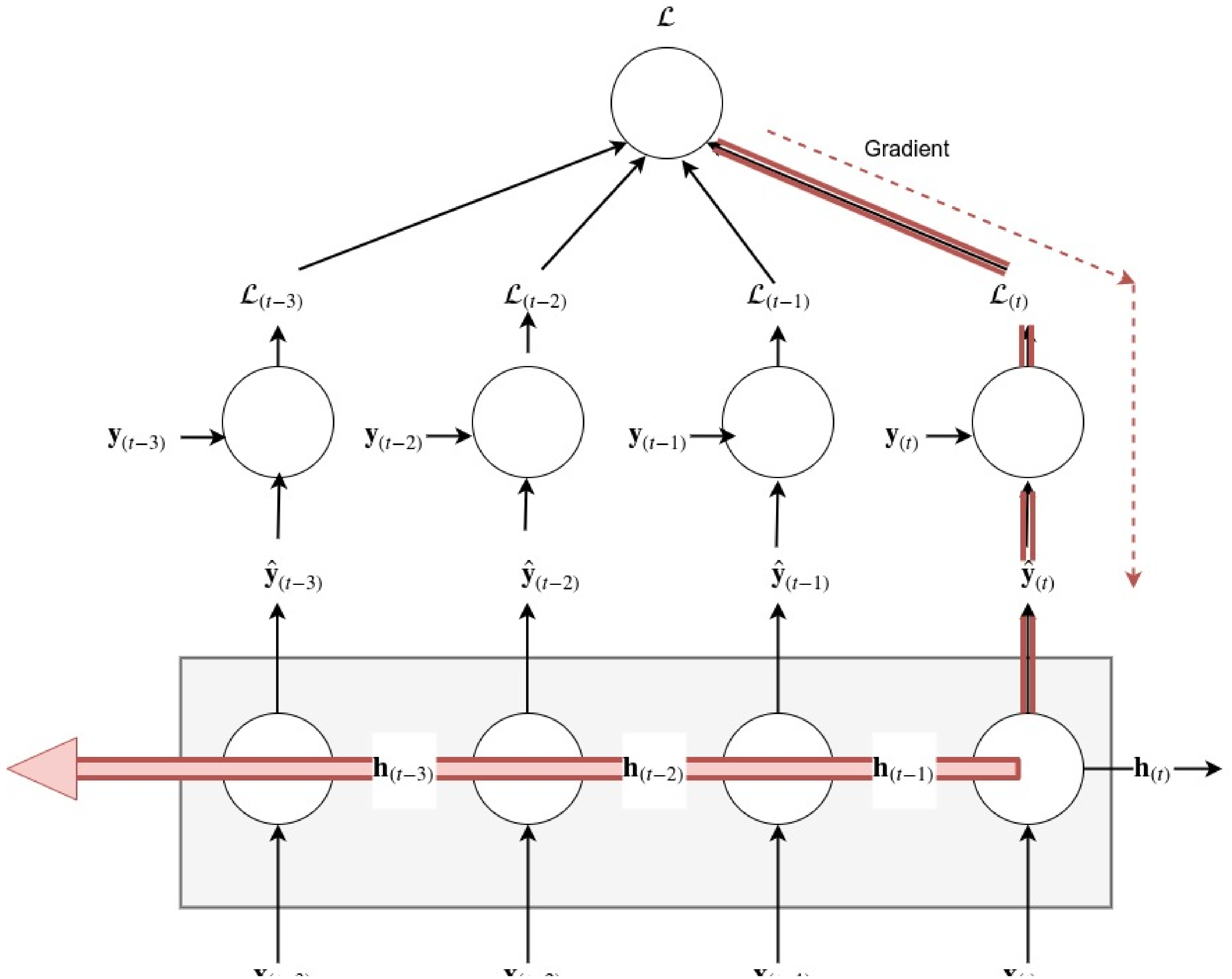
- We can "unroll" the RNN into a sequence of layers, one per time step
- In theory: Back Propagation on the unrolled RNN is the same as for a non-Recurrent Network
- In practice: the unrolled RNN is very deep, which causes issues in Back Propagation.

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.

RNN Loss Gradient Flow



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

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

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

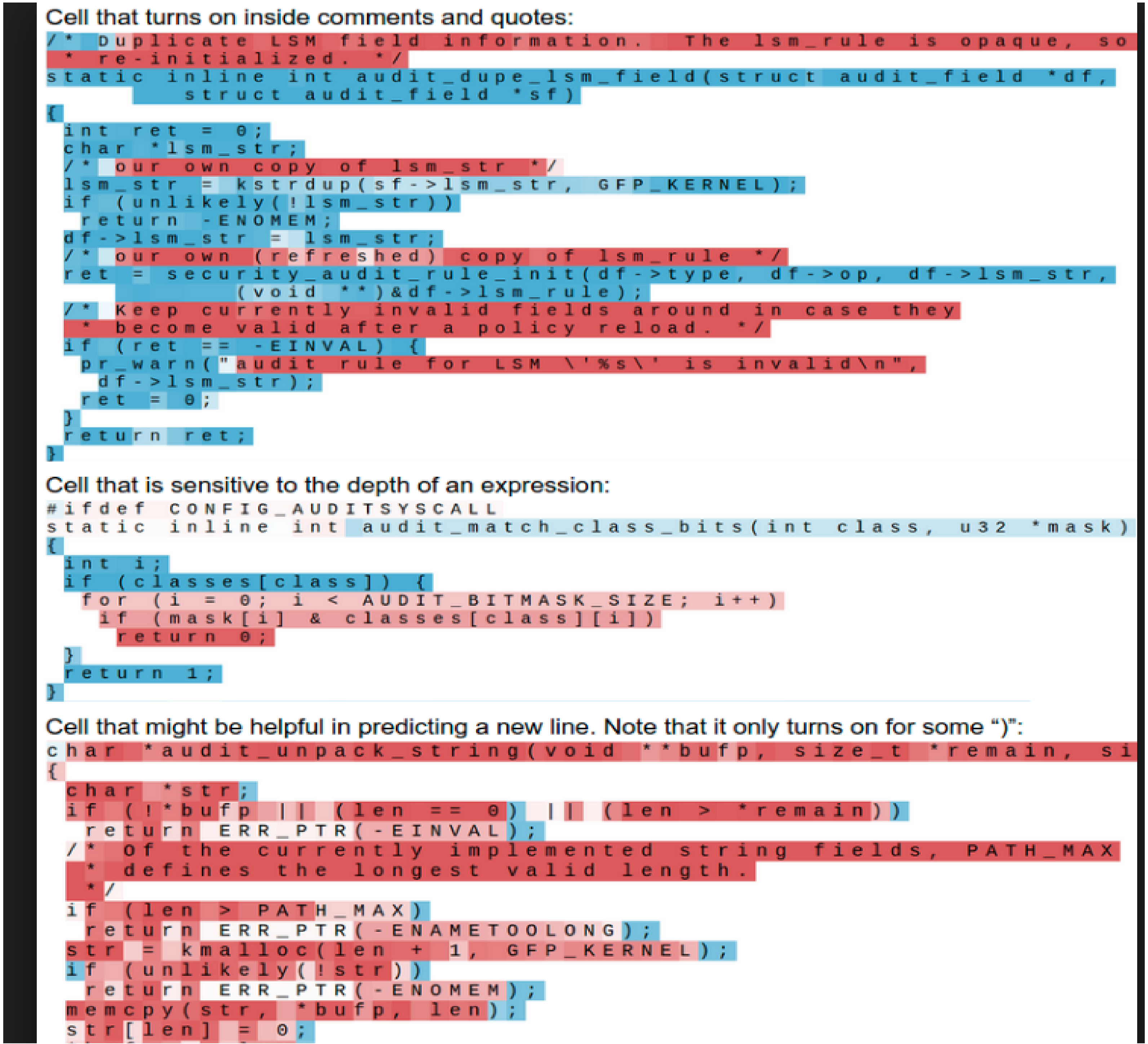
- we can show the same resul with the eigenvalues of the matrix

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)

State activations after seeing prefix of input



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;
```

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

```
Done
```