The LSTM API

A vanilla RNN uses the latent state vector $\mathbf{h}_{(t)}$ to assist with

- Determining the next output $\mathbf{y}_{(t)}$ (for a many to one RNN)
- Updating the latent state from $\mathbf{h}_{(t)}$ to $\mathbf{h}_{(t+1)}$

An LSTM separates these two tasks, using

- $\mathbf{h}_{(t)}$ as "short-term" memory (control state)
- For controlling state transition from $\mathbf{h}_{(t)}$ to $\mathbf{h}_{(t+1)}$

and an additional vector

- $\mathbf{c}_{(t)}$ as a "long-term" memory

Some analogies for the differing roles of short and long term memory

- Your computer
    - Short term memory: RAM (Random Access Memory)
    - Long term memory: a disk, memory stick, flash card
- Your office
    - Short term memory: the desktop
    - Long term memory: the filing cabinet

Let's introduce the basic elements of an LSTM (LSTM_API.ipynb)

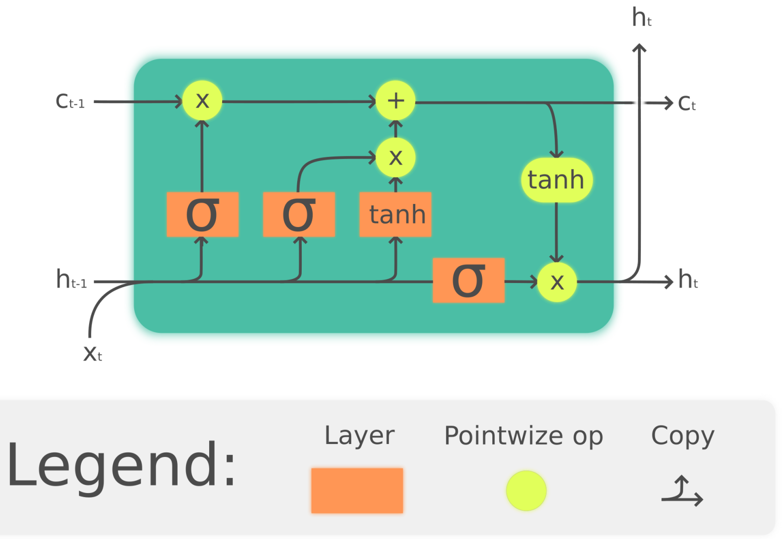
Inside an LSTM Layer

Time to explore the inner workings on an LSTM.

- It will seem complicated at first
- Many small, inter-connected pieces
- Lots of literature with confusing diagrams

Here is a typical diagram from the literature that tries to explain an LSTM

LSTM diagram



Attribution: https://commons.wikimedia.org/wiki/File:The_LSTM_cell.png

We will try to manage the complexity

- Following the "classical" derivation of the original paper that introduced the LSTM (https://www.bioinf.jku.at/publications/older/2604.pdf)
- But influenced by an excellent blog post (http://blog.echen.me/2017/05/30/exploring-lstms/)

Let's go inside an LSTM (LSTM_Workings.ipynb) to see how this happens.

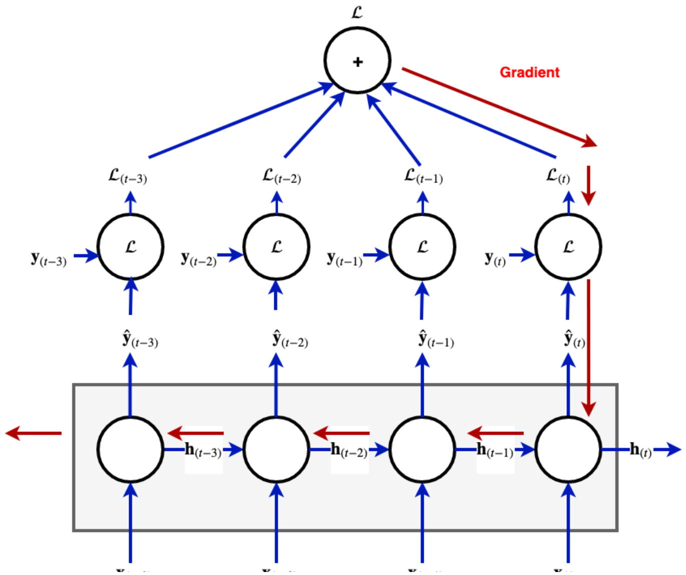
LSTM as gated residual connections

How does an LSTM circumvent the potential problem of vanishing and exploding gradients ?

Recall that a very deep NN (such as an "unrolled" RNN on a long sequence)

- is exposed to the problem of Vanishing/Exploding gradients
- because the Loss Gradient gets moderated by
$$\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}}$$
during Back Propagation
$$\begin{aligned} \mathcal{L}'_{(l-1)} &= \mathcal{L}'_{(l)} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}} \\ &= \mathcal{L}'_{(l)} a'_{(l)} f'_{(l)} \end{aligned}$$
- where $\mathbf{y}_{(l)}$ computes
$$a_{(l)}(f_{(l)}(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l)}))$$
- and $a' < 1$ for many activation functions

RNN Loss: Backward pass



But the LSTM has the ability

- through certain settings of the gates
- to have $\mathbf{y}_{(l)}$ compute
- the identity function

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

When this happens

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

and the Loss Gradient flows backward undiminished.

Examine the update equation for the long term memory of an LSTM:

$$\mathbf{c}_{(t)} = \mathbf{remember}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{save}_{(t)} \otimes \mathbf{c}'_{(t)}$$

and consider the case when

$$\begin{array}{lcll} \mathbf{remember}_{(t),j} & = & 1 & \text{Allow prior value of } j^{th} \text{ element to be updated} \\ \mathbf{save}_{(t),j} & = & 0 & \text{But don't allow the candidate update value } \mathbf{c}'_{(t),j} \text{ to p} \end{array}$$

Then

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

and the "circuit" for this element becomes a skip connection

- Passing through prior value of element j unchanged
- No moderation of the Loss Gradient

By

- Having the ability to turn a layer into the identity transformation
- Implementing a "skip connection" of a residual network

the LSTM can avoid the problem of Vanishing/Exploding gradients.

If $\mathbf{focus}_{(t),j}$ is also equal to 1

$$\mathbf{h}_{(t)} = \mathbf{focus}_t \otimes \tanh(\mathbf{c}_{(t)})$$

is also unchanged.

So, if the Loss function would be minimized by an identity transformation

- The LSTM is able to implement this transformation
- By the appropriate choice of weights $\mathbf{W}$

Initial bias to ”not forget”

A very deep multi-layer network may have trouble learning in early epochs of training

- Uninitialized weights in deep layers
- No specialization of shallow layers to uncover features that would be useful to the deeper layers
- Resulting in large Loss function values
- Large Loss Gradients make weight updates unstable

The ability of the LSTM to implement the identity transformation comes to the rescue !

If we could force $\mathbf{remember}_{(t)}$ to 1 early in training

- We could get the LSTM to implement the identity transformation
- And allow deep LSTM layers to not influence weight updates

We simply set the bias $\mathbf{b}_f$ of the $\mathbf{remember}$ gate to a large positive value

$$\mathbf{remember}_{(t)} = f_{(t)} = \sigma(\mathbf{W}_{x,f}\mathbf{x}_{(t)} + \mathbf{W}_{h,f}\mathbf{h}_{(t-1)} + \mathbf{b}_f)$$

which forces the sigmoid to output 1.

What is *really* going on inside an LSTM

The mechanics of the LSTM feel complicated.

But let's not let that obscure what is going on inside the LSTM.

Let's examine the update equation for the long-term memory $\mathbf{c}_{(t)}$

$$
\begin{aligned}
\mathbf{c}'_{(t)} &= \tanh(\mathbf{W}_{x,c}\mathbf{x}_{(t)} + \mathbf{W}_{h,c}\mathbf{h}_{(t-1)} + \mathbf{b}_c) \\
\mathbf{c}_{(t)} &= \mathbf{remember}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{save}_{(t)} \otimes \mathbf{c}'_{(t)}
\end{aligned}
$$

Note that the candidate update value $\mathbf{c}'_{(t)}$

- Has been squashed by $\tanh$ to the range $[-1, +1]$

When

$$\begin{aligned} \mathbf{remember}_{(t),j} &= 1 \quad \text{Allow prior value of } j^{th} \text{ element to be updated} \\ \mathbf{save}_{(t),j} &= 1 \quad \text{Allow the candidate update value } c'_{(t),j} \text{ to participate} \end{aligned}$$

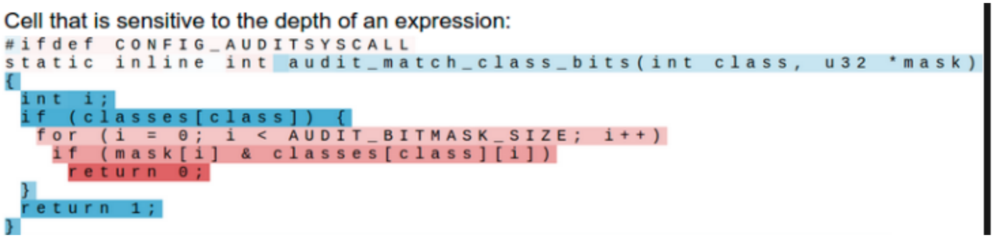
then the j^{th} long-term memory element acts like a counter !

- Incremented/Decremented by $\mathbf{c}_{(t)} \in [-1, +1]$

In our module on RNN Visualization (RNN_Visualization.ipynb)

- We speculated that the elements of the latent state $\mathbf{h}_{(t)}$
- Of a vanilla RNN
- Were implementing counters

State activations after seeing prefix of input



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;
}
```

You can imagine how a counter might be used in handling text input

- As a switch indicating being inside/outside of delimiters like quotation marks
- As a measure of how deeply nested an expression is (e.g., lists)
- As a count of characters in a sentence (increasing probability of seeing an end-of-sentence delimiter)

The update equation for long-term memory

$$\begin{aligned} \mathbf{c}'_{(t)} &= \tanh(\mathbf{W}_{x,c}\mathbf{x}_{(t)} + \mathbf{W}_{h,c}\mathbf{h}_{(t-1)} + \mathbf{b}_c) \\ \mathbf{c}_{(t)} &= \mathbf{remember}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{save}_{(t)} \otimes \mathbf{c}'_{(t)} \end{aligned}$$

is an almost direct operational implementation of the counter concept.

Conclusion

We introduced an advanced Recurrent Layer type call the LSTM.

Similar concepts are present in the Gated Recurrent Unit (GRU), another advanced Recurrent Layer type.

These advanced concepts were designed with the specific intent of dealing with long sequences.

They are thus quite common in domains such as Natural Language Processing, which has long-range dependencies.

```
In [2]: print("Done")
```

```
Done
```