Visualization of RNN hidden state

It's nice that we have update equations telling us the mechanics of an RNN layer.

But *what* is an RNN layer *really* doing ? How does it make the magic happen ?

One plausible theory is that

- The individual elements of the latent state $\mathbf{h}$
- Are acting like *counters*
- Incrementing/Decrementing according to the input

A visualization can confirm this theory (in some cases).

- Pick one element $\mathbf{h}_j$ of the latent state
- Examine the sequence $[\mathbf{h}_{(t),j} | 1 \le t \le T]$ of this element
- Correlate changes in $\mathbf{h}_{(t),j}$ with the input sequence $[\mathbf{x}_{(t)} | 1 \le t \le T]$

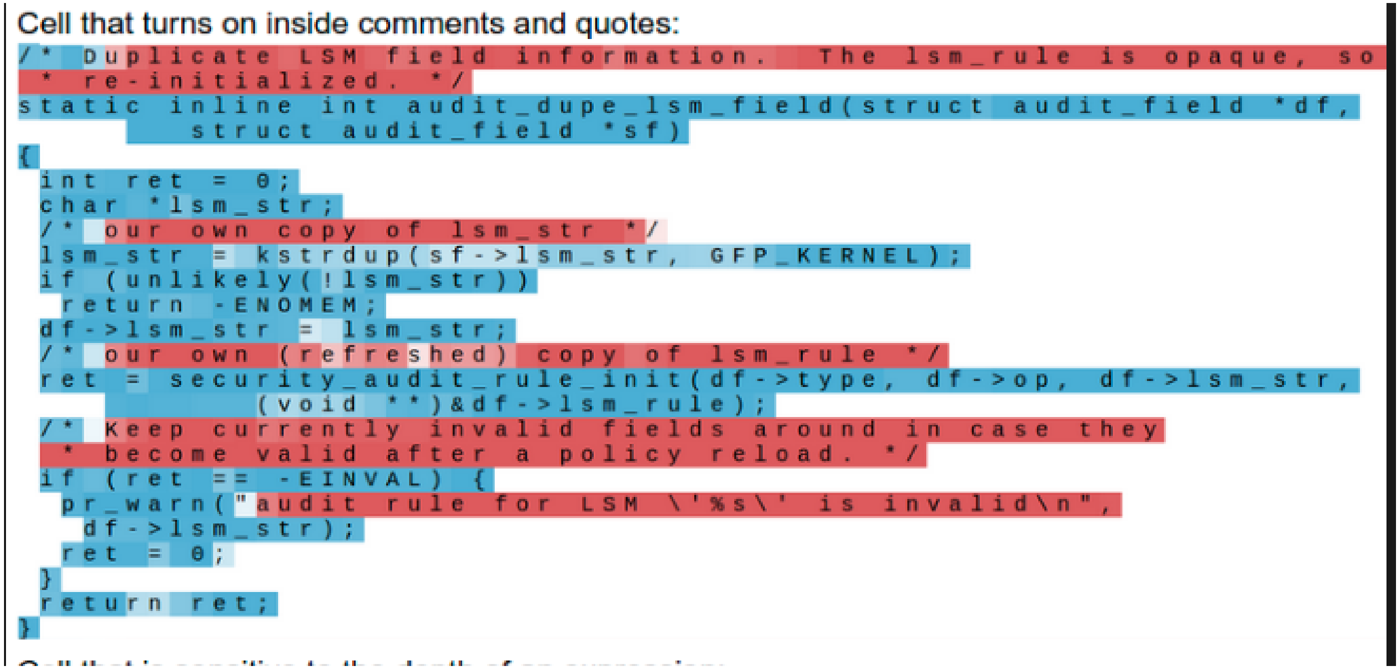
Below is a [visualization (http://karpathy.github.io/2015/05/21/rnn-effectiveness/#visualizing-the-predictions-and-the-neuron-firings-in-the-rnn)](http://karpathy.github.io/2015/05/21/rnn-effectiveness/#visualizing-the-predictions-and-the-neuron-firings-in-the-rnn)

- Of several elements of the hidden state
- Where the value of the element is color-coded
    - Red: High; Blue: Low
- And overlaid on the corresponding element of $\mathbf{x}_{(t)}$
- On an RNN trained on a "predict the next character" in the sequence task

Here is an element ("cell") that becomes active

- Inside quotes (" .. ")
- Inside code comments (/ ... /)

State activations after seeing prefix of input



Here is a cell that seems to be

- Counting the *depth* of nesting of code

State activations after seeing prefix of input

Cell that is sensitive to the depth of an expression:

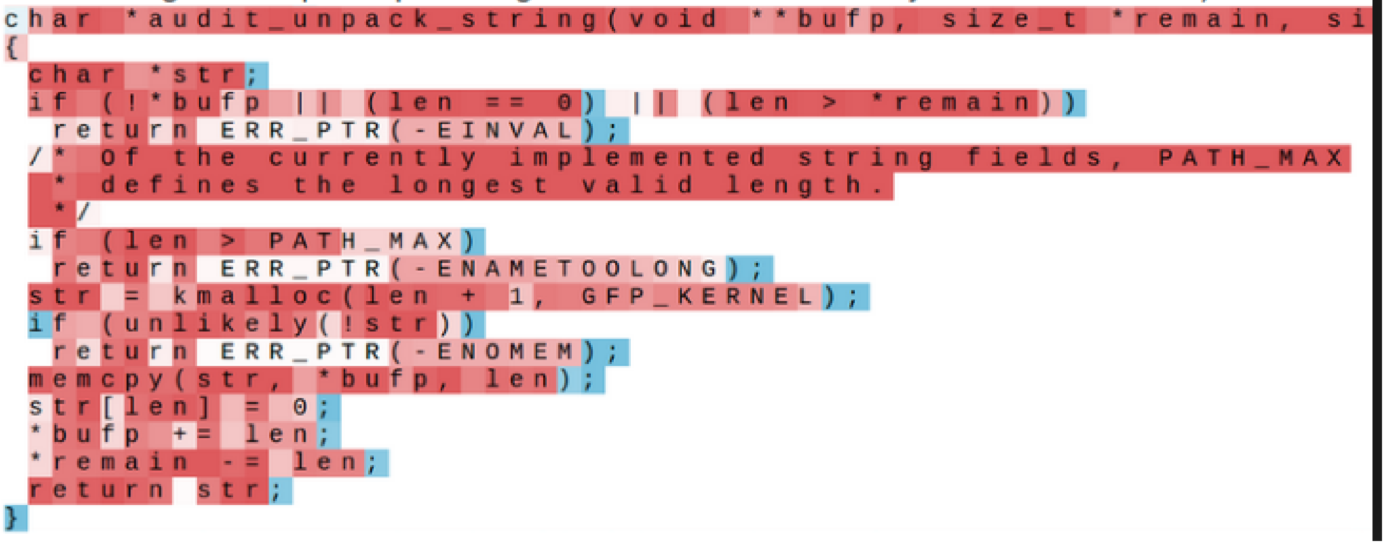
```
#ifdef CONFIG_AUDITSYSCALL
static inline int audit_match_class_bits(int class, u32 *mask)
{
 int i;
 if (classes[class]) {
  for (i = 0; i < AUDIT_BITMASK_SIZE; i++)
   if (mask[i] & classes[class][i])
    return 0;
 }
 return 1;
}
```

And here is a cell that has been interpretted

- As predicting end-of-line characers

State activations after seeing prefix of input

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



Of course, this is a matter of interpretation rather than mathematics.

Still: there is some logic in believing that counters

- Can capture structure
- Sufficient to encode the probability of the next character (our target)

In a later module

- We will study a more advanced Recurrent layer called an LSTM
- It's internal workings are closely aligned with the notion of implementing counters

In [2]:

```
print("Done")
```

Done