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 $\left[\mathbf{h}_{(t),j} \mid 1 \leq t \leq T\right]$ of this element
- Correlate changes in $\mathbf{h}_{(t),j}$ with the input sequence $\left[\mathbf{x}_{(t)} \mid 1 \leq t \leq T\right]$

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 (/ ... /)

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;
}
```

Here is a cell that seems to be

- Counting the *depth* of nesting of code

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

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

```c
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;
	*bufp += len;
	*remain -= len;
	return str;
}
```

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