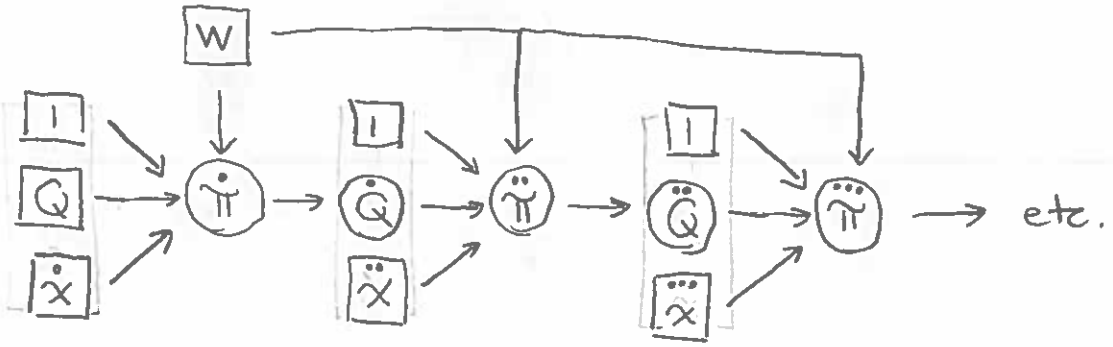
Long Short-Term Memory Networks

① You can view an RNN as a fancy way of representing a for-loop. For instance, we previously represented a loop that determines whether there are at least 3 Gs in a DNA sequence:

```
for x in X:
    if x == "G" and not Q[1]:
        Q[1] = True
    elif x == "G" and not Q[2]:
        Q[2] = True
    elif x == "G" and not Q[3]:
        Q[3] = True
```

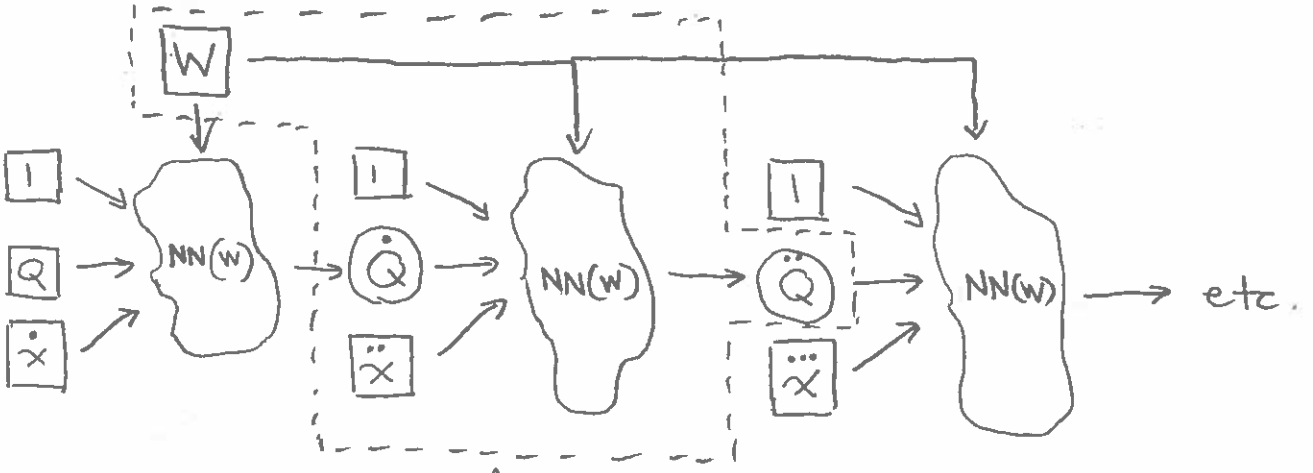
---

② We used the following neural net:



where: $\overset{(k)}{\pi} = W^T \begin{bmatrix} 1 \\ \overset{(k-1)}{Q} \\ \overset{(k)}{x} \end{bmatrix}$ and $\overset{(k)}{Q} = a(\overset{(k)}{\pi})$

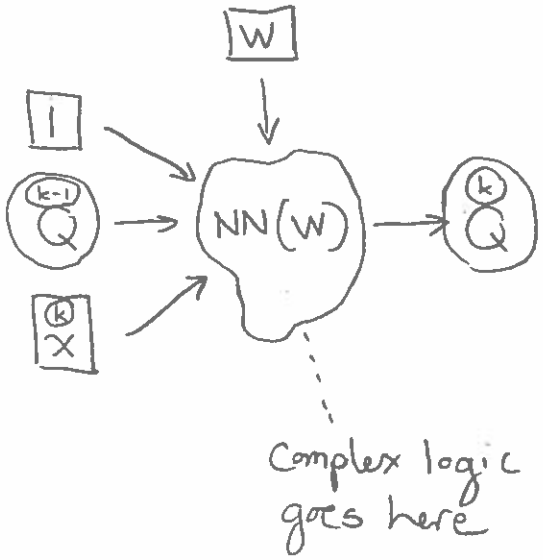
③ For each iteration, we use a single fully connected layer, followed by an activation function. But there's nothing stopping us from using a more complicated neural network to compute the next state $Q^{(k)}$ at each iteration:



by defining different NN(W), we can make different, more powerful RNNs.

④ Notice that we can structure NN(W) to enable more complex logic in each iteration of the loop:

```
for x in X:
    | complex logic
    | goes here
```



⑤ Let's try encoding a somewhat more complicated function. Suppose we want to determine whether a bitstring contains at least four sequences of three consecutive ones, e.g.

0 1 1 1 1 0 1 1 1 0 1 1 1 ✓

(1, 2, 3, 4)

Note that overlapping sequences still count.

---

⑥ One way to write this as a function:

```
def f(X: bitstring):
    count = 0
    streak = 0

    for ch in X:
        if ch == 0:
            streak = 0
        else:
            streak += 1
        if streak >= 3:
            count += 1
    return count ≥ 4
```

Long Short-Term Memory Networks

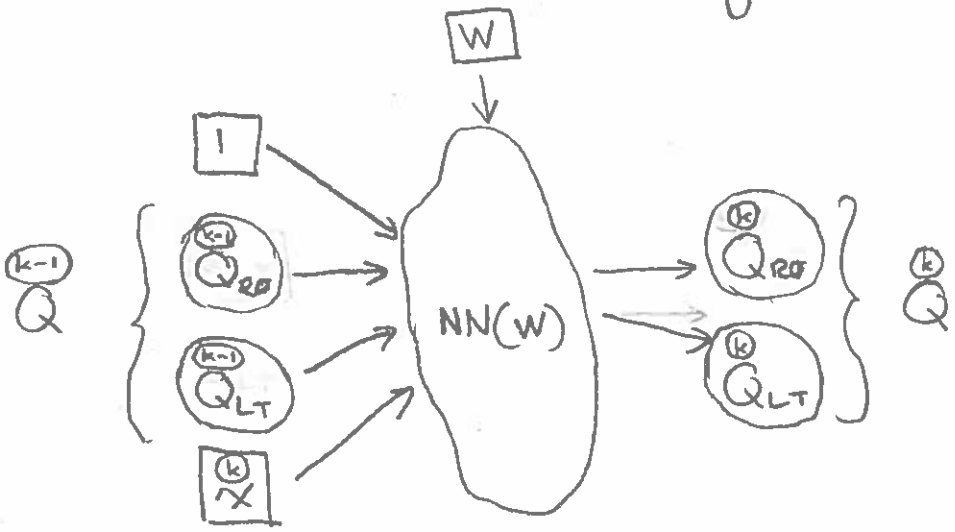
7) For our state variables Q, we'll use our variables count and streak.

$$Q = \begin{bmatrix} \text{count} \\ \text{streak} \end{bmatrix}$$

But we'll actually make two copies of each: a "long-term" version and a "read-only" version:

$$Q = \begin{bmatrix} \text{count}_{RO} \\ \text{streak}_{RO} \\ \text{count}_{LT} \\ \text{streak}_{LT} \end{bmatrix} \quad \left(\text{or } Q_{RO} = \begin{bmatrix} \text{count}_{RO} \\ \text{streak}_{RO} \end{bmatrix}, \; Q_{LT} = \begin{bmatrix} \text{count}_{LT} \\ \text{streak}_{LT} \end{bmatrix} \right)$$

So we'll be trying to find a good neural network architecture of the following form:



Long Short-Term Memory Networks

(8) Let's trace out the desired functionality of NN(W) in terms of the long-term state variables. Suppose our bitstring is 0111011110 and we're starting the fifth iteration, i.e.

0 1 1 1 0 1 1 1 1 0
^

At this point, hopefully $Q_{LT} = \begin{bmatrix} count_{LT} \\ streak_{LT} \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$

---

(9) We'd like to go through the following steps:

$$\begin{matrix} count_{LT} \\ streak_{LT} \end{matrix} \begin{bmatrix} 1 \\ 3 \end{bmatrix} \xrightarrow[\text{streak}]{\text{reset}} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \xrightarrow[\text{streak and count}]{\text{increment}} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

(if applicable) (if applicable)

---

(10) Similarly, for the next iteration:

0 1 1 1 0 1 1 1 1 0
^

$$\begin{matrix} count \\ streak \end{matrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \xrightarrow{\text{reset}} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \xrightarrow{\text{increment}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

(11) Going through the next few iterations:

0 1 1 1 0 1 1 1 1 0
^

count [1] streak [1] → reset → [1 / 1] → increment → [1 / 2]

streak gets incremented to 2 because we encountered another one in the bitstring

0 1 1 1 0 1 1 1 1 0
  ^

count [1] streak [2] → reset → [1 / 2] → increment → [1 / 3]

0 1 1 1 0 1 1 1 1 0
    ^

count [1] streak [3] → reset → [1 / 3] → increment → [2 / 4]

count gets incremented to 2 because the streak was ≥ 3 at the iteration's start

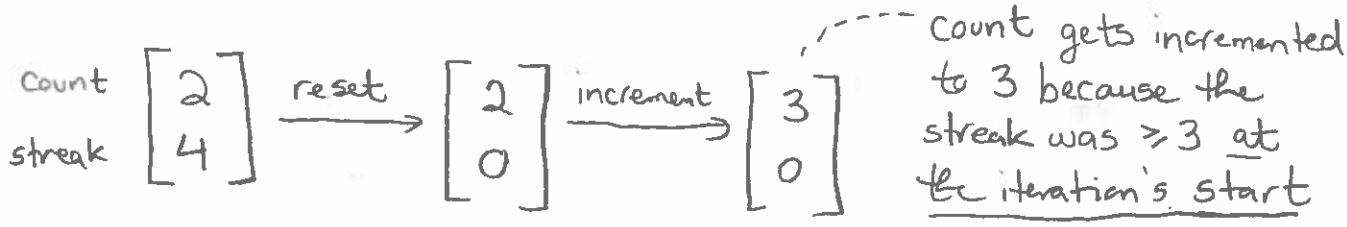
0 1 1 1 0 1 1 1 1 0
      ^

count [2] streak [4] → reset → [2 / 0] → increment → [3 / 0]

streak gets reset to 0 because we encountered a zero in the bitstring

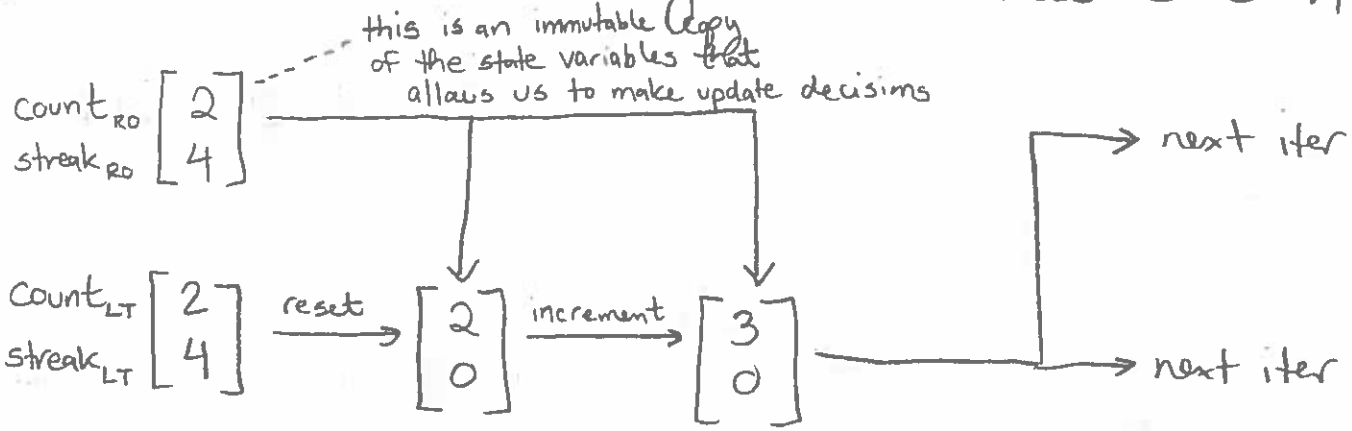
Long Short-Term Memory Networks

(12) Notice particularly the last iteration:

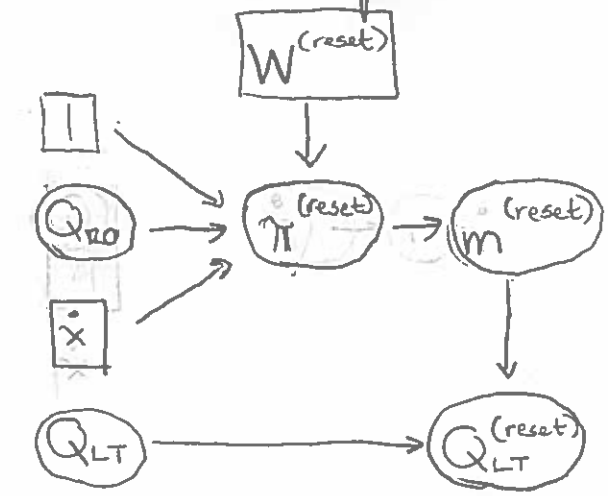


But we wiped out the streak during the reset, so how do we implement the increment of count?

(13) This is where the read-only state variables come in:

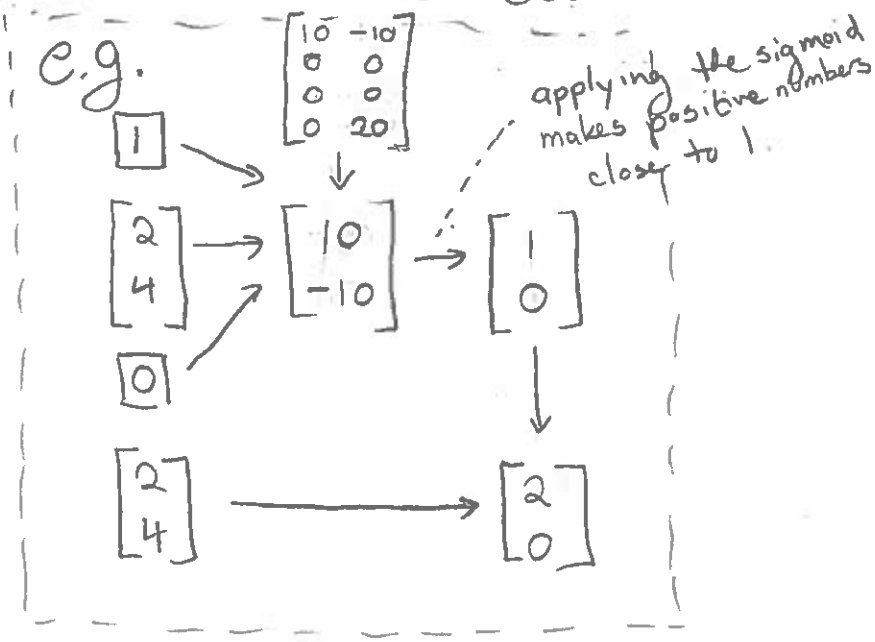


(14) How do we implement "reset" as a neural network?

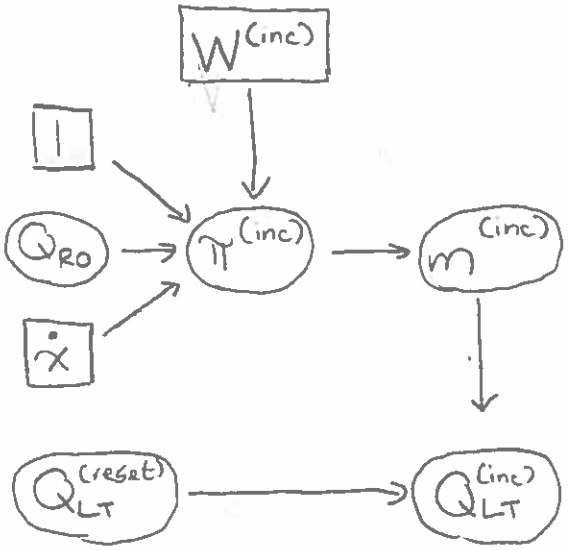


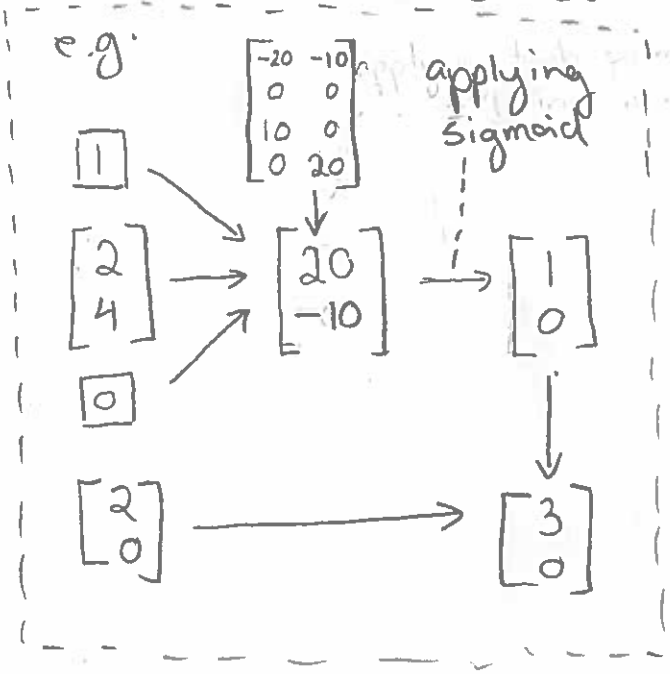
where: $\pi = W^T \begin{bmatrix} 1 \\ Q_{RO} \\ \dot{x} \end{bmatrix}$, $m = \sigma(\pi)$

$Q_{LT}^{(reset)} = Q_{LT} \oplus m^{(reset)}$



(15) How do we implement "increment" as a neural network?





where: $\pi = W^T \begin{bmatrix} 1 \\ Q_{RO} \\ \dot{x} \end{bmatrix}$

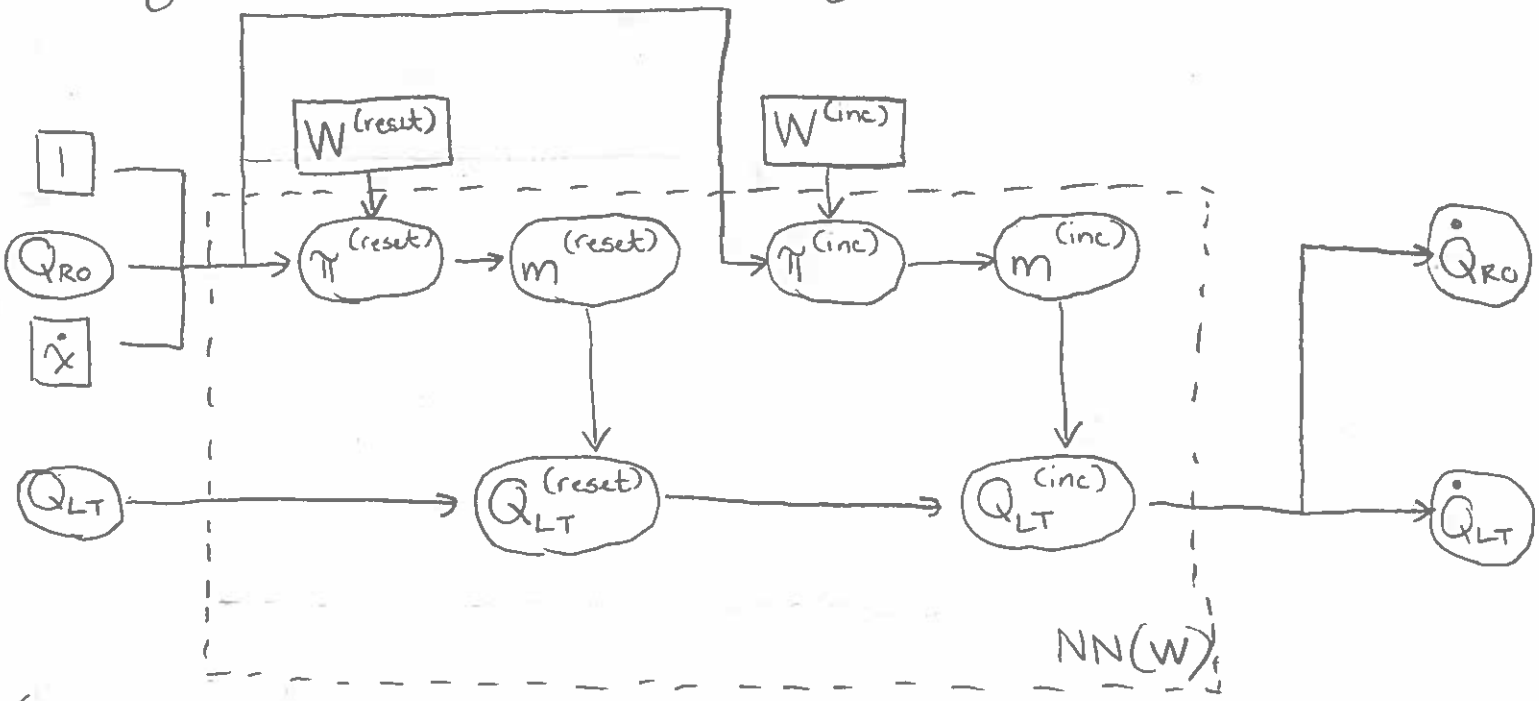
$m = \sigma(\pi)$

$Q_{LT}^{(inc)} = Q_{LT}^{(reset)} + m^{(inc)}$

"elementwise "plus", not Hadamard product

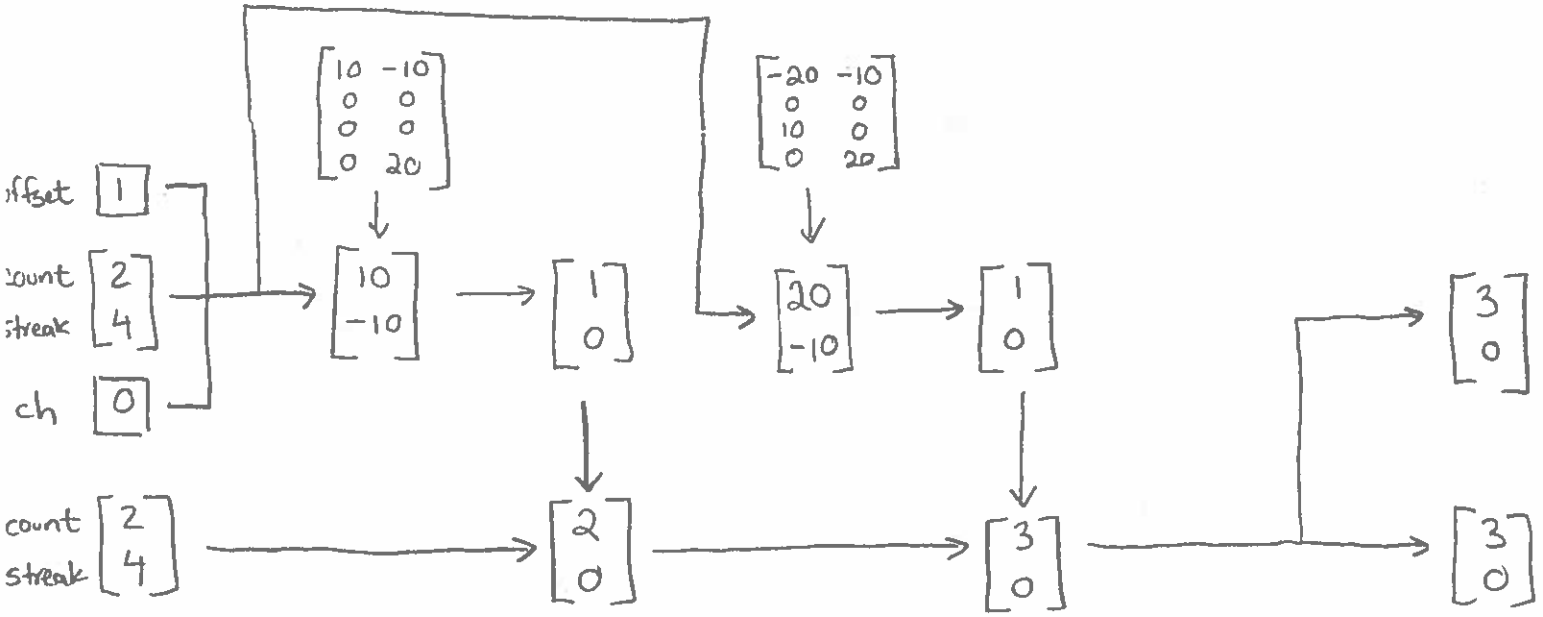
Long Short-term Memory Networks

(16) Putting "reset" and "increment" together:



(Compare this to (7))

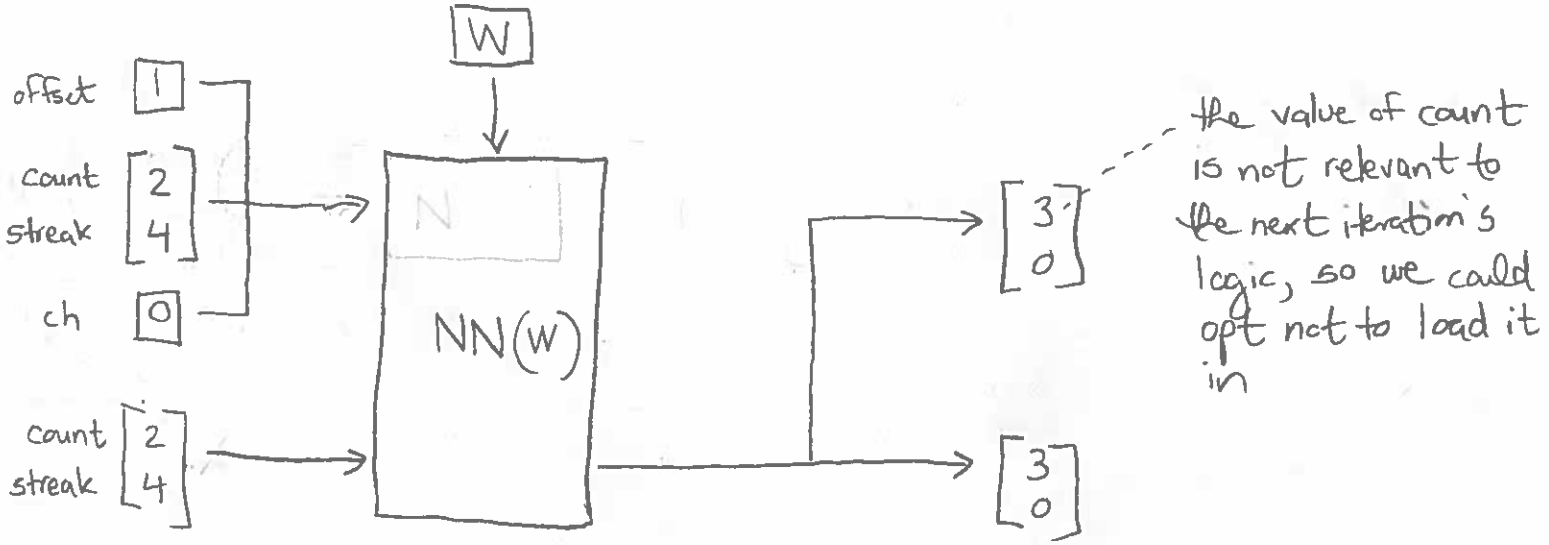
(17) For example:



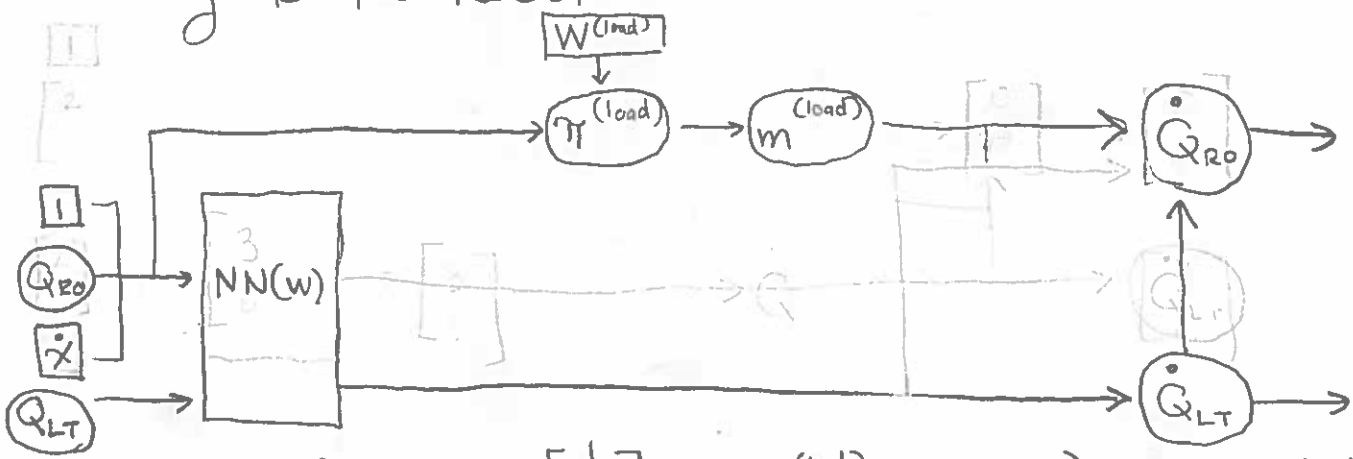
Long Short-Term Memory Networks

(18) The RNN depicted in (16) is close to something called a Long Short-Term Memory network (LSTM). Full LSTMs have a caple additional extensions.

First, the LSTM does not have to load <u>everything</u> from the long-term memory back into the read-only buffer for the next iteration.



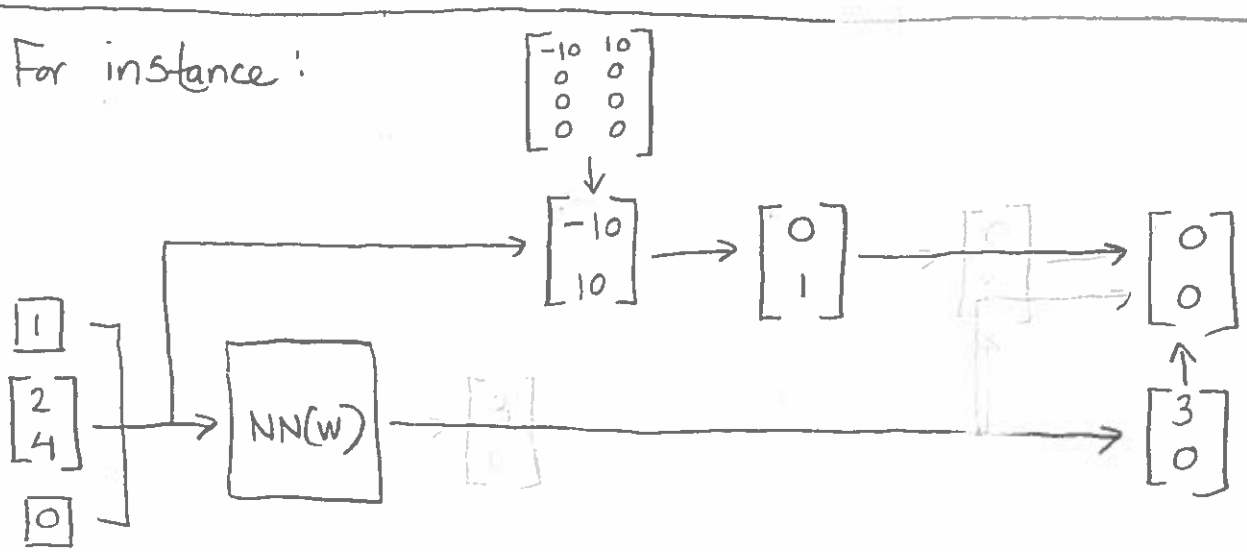
(19) Zeroing out irrelevant state variables can be achieved the same way as the reset.



where: $\pi^{(load)} = (W^{(load)})^T \begin{bmatrix} 1 \\ Q_{RO} \\ \dot{x} \end{bmatrix}$; $m^{(load)} = \sigma(\pi^{(load)})$; $\dot{Q}_{RO} = m^{(load)} \oplus \dot{Q}_{LT}$

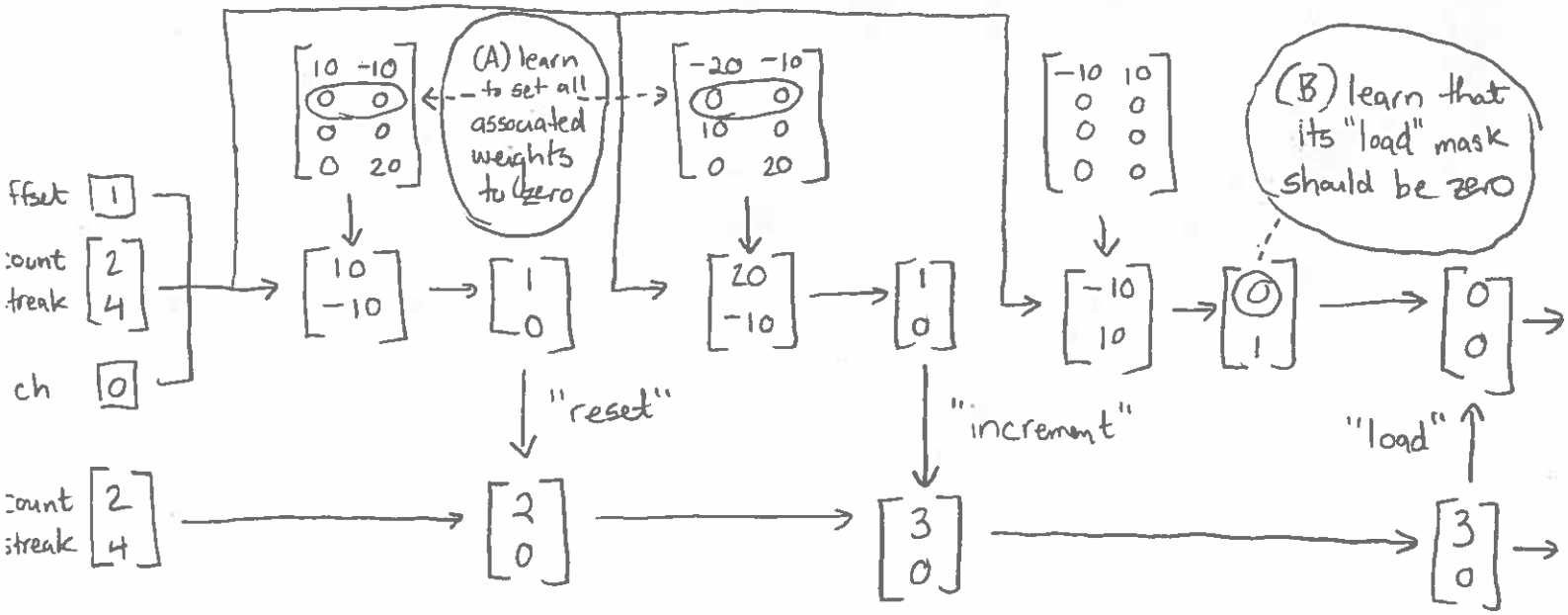
Long Short-Term Networks

(20) For instance:



(21) Why load selectively? It certainly doesn't harm things (from a representational perspective) to load the entire long-term memory into the read-only buffer every iteration.

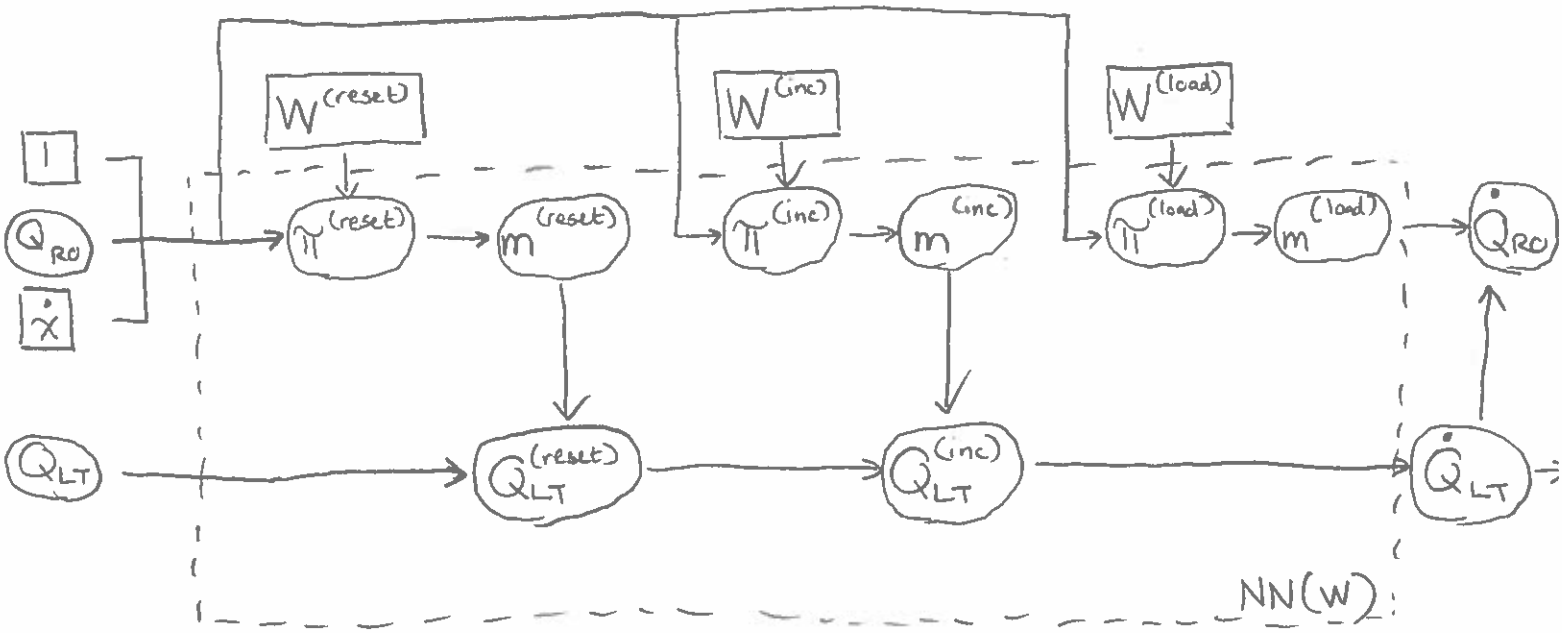
However, it may make it easier to learn the function. Consider the count variable, which is never needed in the read-only buffer. In order to ignore it, you can either:



Option B often requires less parameters to learn.

Long Short-Term Memory Networks

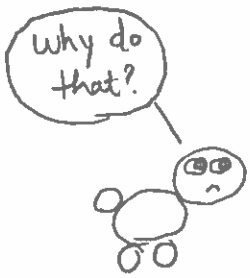
(22) Adding the "load" to our RNN gives us:



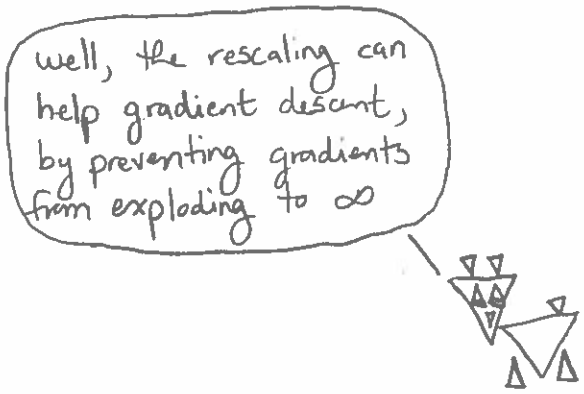
where:

$\pi^{(reset)} = (W^{(reset)})^T \begin{bmatrix} 1 \\ Q_{RO} \\ \dot{x} \end{bmatrix}$	$m^{(reset)} = \sigma(\pi^{(reset)})$	$Q_{LT}^{(reset)} = m^{(reset)} \oplus Q_{LT}$ (not $\oplus$!)
$\pi^{(inc)} = (W^{(inc)})^T \begin{bmatrix} 1 \\ Q_{RO} \\ \dot{x} \end{bmatrix}$	$m^{(inc)} = \sigma(\pi^{(inc)})$	$Q_{LT}^{(inc)} = m^{(inc)} + Q_{LT}^{(reset)}$ $\dot{Q}_{LT} = Q_{LT}^{(inc)}$
$\pi^{(load)} = (W^{(load)})^T \begin{bmatrix} 1 \\ Q_{RO} \\ \dot{x} \end{bmatrix}$	$m^{(load)} = \sigma(\pi^{(load)})$	$\dot{Q}_{RO} = m^{(load)} \oplus \dot{Q}_{LT}$

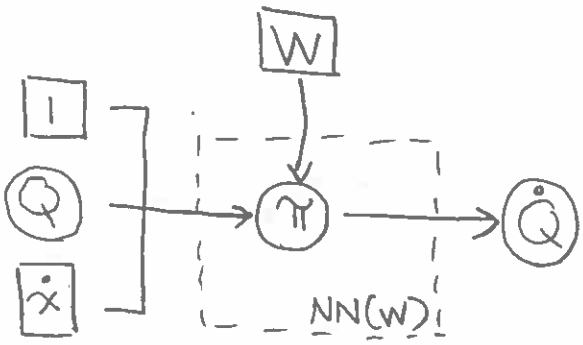
(23) The final extension needed to turn this into a standard LSTM is the addition of a couple tanh activations in strategic places, e.g. to rescale the values in the read-only buffer to be between -1 and 1, i.e.



$$\dot{Q}_{RO} = \tanh\left(m^{(load)} \oplus \dot{Q}_{LT}\right)$$



(24) Anyways, even comparing our LSTM-lite with the vanilla RNN:



it's clear that it's much more complicated. However it does tend to work better, particularly for long sequential iterations. This is generally attributed to the long-term state, which allows the network to "remember" information that may only be relevant much later in the sequence.

(25) As an example of when these "long term relevancies" might occur, consider the task of coreference resolution in English.

> "Suppose two integers sum to seven, and moreover, the product of the numbers is twelve. If so, what are the two numbers?"

If an RNN is processing this passage word by word, then it needs the capacity to store information about the "two integers" and retrieve it much later in the passage (both when it is stated that the product of the numbers is twelve, and when it is asked what the two numbers are).