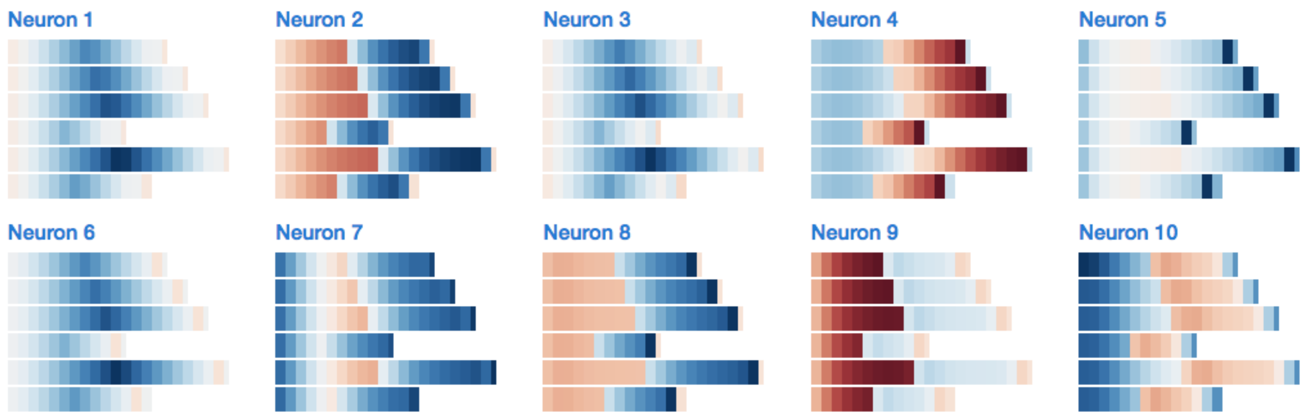
Exploring LSTMs

The first time I learned about LSTMs, my eyes glazed over.

Not in a good, jelly donut kind of way.

It turns out LSTMs are a fairly simple extension to neural networks, and they're behind a lot of the amazing achievements deep learning has made in the past few years. So I'll try to present them as intuitively as possible – in such a way that you could have discovered them yourself.

But first, a picture:



Aren't LSTMs beautiful? Let's go.

(Note: if you're already familiar with neural networks and LSTMs, skip to the middle – the first half of this post is a tutorial.)

Neural Networks

Imagine we have a sequence of images from a movie, and we want to label each image with an activity (is this a fight?, are the characters talking?, are the characters eating?).

How do we do this?

One way is to ignore the sequential nature of the images, and build a *per-image* classifier that considers each image in isolation. For example, given enough images and labels:

- Our algorithm might first learn to detect low-level patterns like shapes and edges.
- With more data, it might learn to combine these patterns into more complex ones, like faces (two circular things atop a triangular thing atop an oval thing) or cats.

- And with even more data, it might learn to map these higher-level patterns into activities themselves (scenes with mouths, steaks, and forks are probably about eating).

This, then, is a **deep neural network**: it takes an image input, returns an activity output, and – just as we might learn to detect patterns in puppy behavior without knowing anything about dogs (after seeing enough corgis, we discover common characteristics like fluffy butts and drumstick legs; next, we learn advanced features like splooting) – in between it learns to represent images through hidden layers of representations.

Mathematically

I assume people are familiar with basic neural networks already, but let's quickly review them.

- A **neural network** with a single hidden layer takes as input a vector x, which we can think of as a set of neurons.
- Each input neuron is connected to a hidden layer of neurons via a set of learned weights.
- The jth hidden neuron outputs $h_j = \phi(\sum_i w_{ij} x_i)$, where ϕ is an activation function.
- The hidden layer is fully connected to an output layer, and the jth output neuron outputs $y_j = \sum_i v_{ij} h_i$. If we need probabilities, we can transform the output layer via a softmax function.

In matrix notation:

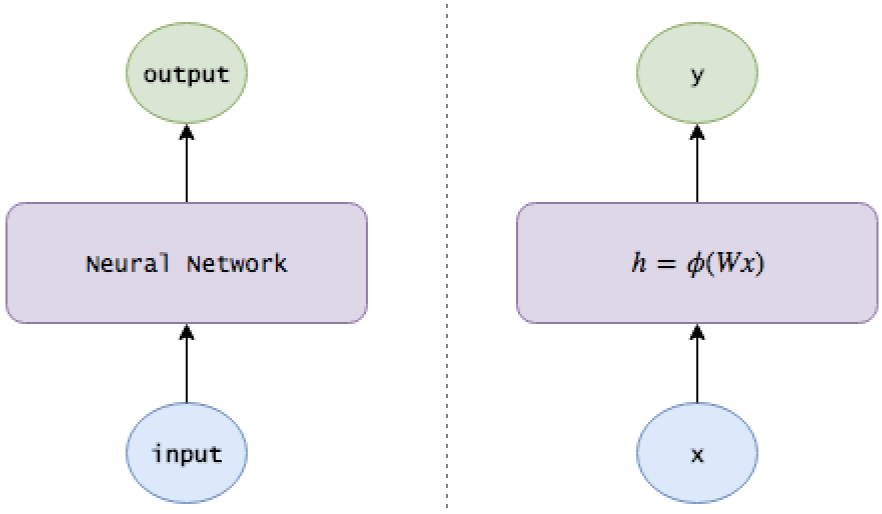
$$h = \phi(Wx)$$

$$y = Vh$$

where

- x is our input vector
- W is a weight matrix connecting the input and hidden layers
- V is a weight matrix connecting the hidden and output layers
- Common activation functions for ϕ are the sigmoid function, $\sigma(x)$, which squashes numbers into the range (0, 1); the hyperbolic tangent, $tanh(x)$, which squashes numbers into the range (-1, 1), and the rectified linear unit, $ReLU(x) = max(0, x)$.

Here's a pictorial view:



(Note: to make the notation a little cleaner, I assume x and h each contain an extra bias neuron fixed at 1 for learning bias weights.)

Remembering Information with RNNs

Ignoring the sequential aspect of the movie images is pretty ML 101, though. If we see a scene of a beach, we should boost beach activities in future frames: an image of someone in the water should probably be labeled *swimming*, not *bathing*, and an image of someone lying with their eyes closed is probably *suntanning*. If we remember that Bob just arrived at a supermarket, then even without any distinctive supermarket features, an image of Bob holding a slab of bacon should probably be categorized as *shopping* instead of *cooking*.

So what we'd like is to let our model track the state of the world:

1. **After seeing each image, the model outputs a label and also updates the knowledge it's been learning.** For example, the model might learn to automatically discover and track information like location (are scenes currently in a house or beach?), time of day (if a scene contains an image of the moon, the model should remember that it's nighttime), and within-movie progress (is this image the first frame or the 100th?). Importantly, just as a neural network automatically discovers hidden patterns like edges, shapes, and faces without being fed them, our model should automatically discover useful information by itself.
2. When given a new image, the model should **incorporate the knowledge it's gathered** to do a better job.

This, then, is a **recurrent neural network.** Instead of simply taking an image and returning an activity, an RNN also maintains internal memories about the world (weights assigned to different pieces of information) to help perform its classifications.

Mathematically

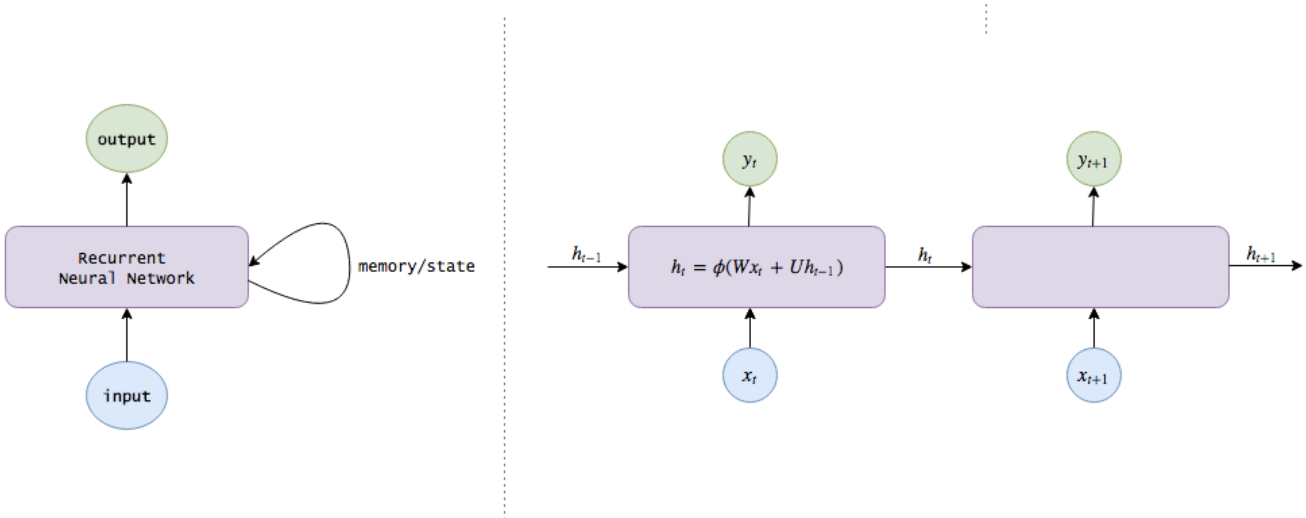
So let's add the notion of **internal knowledge** to our equations, which we can think of as pieces of information that the network maintains over time.

But this is easy: we know that the hidden layers of neural networks already encode useful information about their inputs, so why not use these layers as the memory passed from one time step to the next? This gives us our RNN equations:

$$h_t = \phi(Wx_t + Uh_{t-1})$$

$$y_t = Vh_t$$

Note that the hidden state computed at time t (h_t, our internal knowledge) is fed back at the next time step. (Also, I'll use concepts like hidden state, knowledge, memories, and beliefs to describe h_t interchangeably.)



Longer Memories through LSTMs

Let's think about how our model updates its knowledge of the world. So far, we've placed no constraints on this update, so its knowledge can change pretty chaotically: at one frame it thinks the characters are in the US, at the next frame it sees the characters eating sushi and thinks they're in Japan, and at the next frame it sees polar bears and thinks they're on Hydra Island. Or perhaps it has a wealth of information to suggest that Alice is an investment analyst, but decides she's a professional assassin after seeing her cook.

This chaos means information quickly transforms and vanishes, and it's difficult for the model to keep a long-term memory. So what we'd like is for the network to *learn* how to update its beliefs (scenes without Bob shouldn't change Bob-related information, scenes with Alice should focus on gathering details about her), in a way that its knowledge of the world evolves more gently.

This is how we do it.

1. **Adding a forgetting mechanism.** If a scene ends, for example, the model should forget the current scene location, the time of day, and reset any scene-specific information; however, if a character dies in the scene, it should continue remembering that he's no longer alive. Thus, we want the model to learn a separate *forgetting/remembering* mechanism: when new inputs come in, it needs to know which beliefs to keep or throw away.
2. **Adding a saving mechanism.** When the model sees a new image, it needs to learn whether any information about the image is worth using and saving. Maybe your mom sent you an article about the Kardashians, but who cares?
3. So when new a input comes in, the model first forgets any long-term information it decides it no longer needs. Then it learns which parts of the new input are worth using, and saves them into its long-term memory.
4. **Focusing long-term memory into working memory.** Finally, the model needs to learn which parts of its long-term memory are immediately useful. For example, Bob's age may be a useful piece of information to keep in the long term (children are more likely to be crawling, adults are more likely to be working), but is probably irrelevant if he's not in the current scene. So instead of using the full long-term memory all the time, it learns which parts to focus on instead.

This, then, is an **long short-term memory network.** Whereas an RNN can overwrite its memory at each time step in a fairly uncontrolled fashion, an LSTM transforms its memory in a very precise way: by using *specific learning mechanisms* for which pieces of information to remember, which to update, and which to pay attention to. This helps it keep track of information over longer periods of time.

Mathematically

Let's describe the LSTM additions mathematically.

At time t, we receive a new input x_t. We also have our long-term and working memories passed on from the previous time step, ltm_{t-1} and wm_{t-1} (both n-length vectors), which we want to update.

We'll start with our long-term memory. First, we need to know which pieces of long-term memory to continue remembering and which to discard, so we want to use the new input and our working memory to learn a **remember gate** of n numbers between 0 and 1, each of which determines how much of a long-term memory element to keep. (A 1 means to keep it, a 0 means to forget it entirely.)

Naturally, we can use a small neural network to learn this remember gate:

$$remember_t = \sigma(W_r x_t + U_r wm_{t-1})$$

(Notice the similarity to our previous network equations; this is just a shallow neural network. Also, we use a sigmoid activation because we need numbers between 0 and 1.)

Next, we need to compute the information we can learn from x_t, i.e., a **candidate addition to our long-term memory**:

$$ltm'_t = \phi(W_l x_t + U_l wm_{t-1})$$

ϕ is an activation function, commonly chosen to be $tanh$.

Before we add the candidate into our memory, though, we want to learn **which parts of it are actually worth using and saving**:

$$save_t = \sigma(W_s x_t + U_s wm_{t-1})$$

(Think of what happens when you read something on the web. While a news article might contain information about Hillary, you should ignore it if the source is Breitbart.)

Let's now combine all these steps. After forgetting memories we don't think we'll ever need again and saving useful pieces of incoming information, we have our **updated long-term memory**:

$$ltm_t = remember_t \circ ltm_{t-1} + save_t \circ ltm'_t$$

where $\circ$ denotes element-wise multiplication.

Next, let's update our working memory. We want to learn how to focus our long-term memory into information that will be *immediately* useful. (Put differently, we want to learn what to move from an *external hard drive* onto our *working laptop*.) So we learn a **focus/attention vector**:

$$focus_t = \sigma(W_f x_t + U_f wm_{t-1})$$

Our **working memory** is then

$$wm_t = focus_t \circ \phi(ltm_t)$$

In other words, we pay full attention to elements where the focus is 1, and ignore elements where the focus is 0.

And we're done! Hopefully this made it into your long-term memory as well.

To summarize, whereas a vanilla RNN uses one equation to update its hidden state/memory:

$$h_t = \phi(W x_t + U h_{t-1})$$

An LSTM uses several:

$$ltm_t = remember_t \circ ltm_{t-1} + save_t \circ ltm'_t$$

$$wm_t = focus_t \circ tanh(ltm_t)$$

where each memory/attention sub-mechanism is just a mini brain of its own:

$$remember_t = \sigma(W_r x_t + U_r wm_{t-1})$$

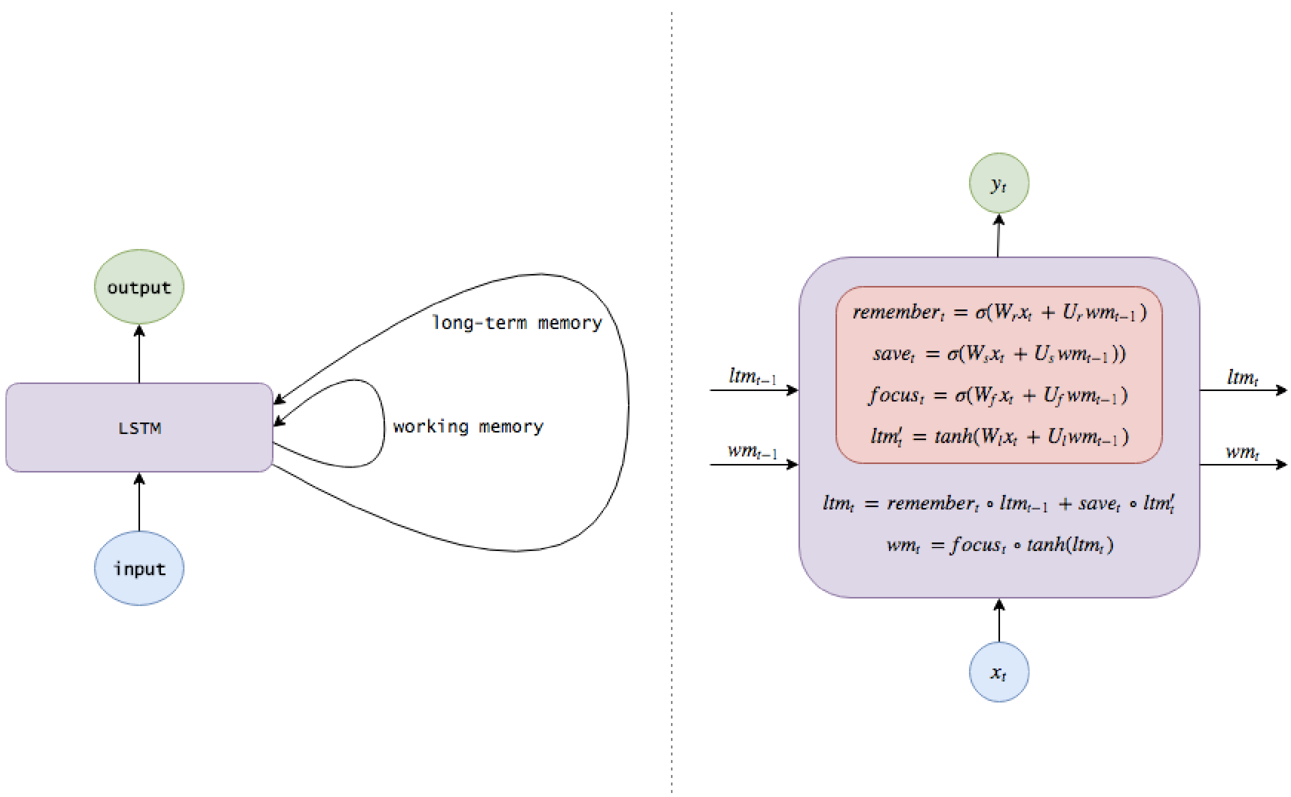
$$save_t = \sigma(W_s x_t + U_s wm_{t-1})$$

$$focus_t = \sigma(W_f x_t + U_f wm_{t-1})$$

$$ltm'_t = tanh(W_l x_t + U_l wm_{t-1})$$

(Note: the terminology and variable names I've been using are different from the usual literature. Here are the standard names, which I'll use interchangeably from now on:

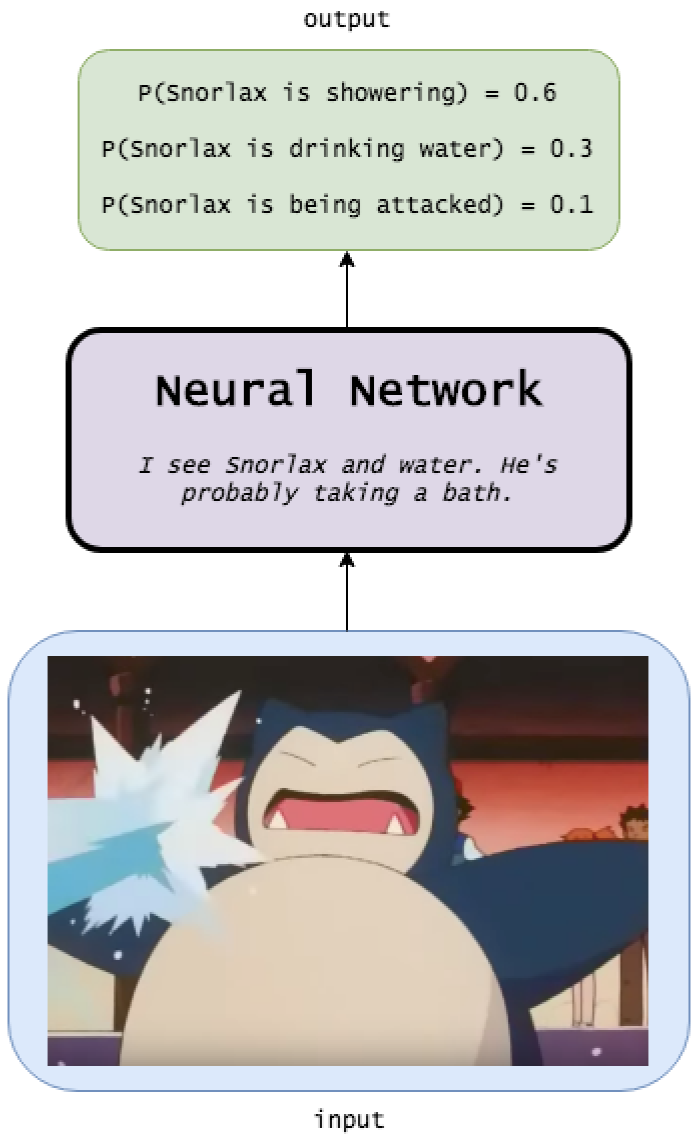
- The long-term memory, ltm_t, is usually called the **cell state**, denoted c_t.
- The working memory, wm_t, is usually called the **hidden state**, denoted h_t. This is analogous to the hidden state in vanilla RNNs.
- The remember vector, $remember_t$, is usually called the **forget gate** (despite the fact that a 1 in the forget gate still means to keep the memory and a 0 still means to forget it), denoted f_t.
- The save vector, $save_t$, is usually called the **input gate** (as it determines how much of the input to let into the cell state), denoted i_t.
- The focus vector, $focus_t$, is usually called the **output gate**, denoted o_t.)



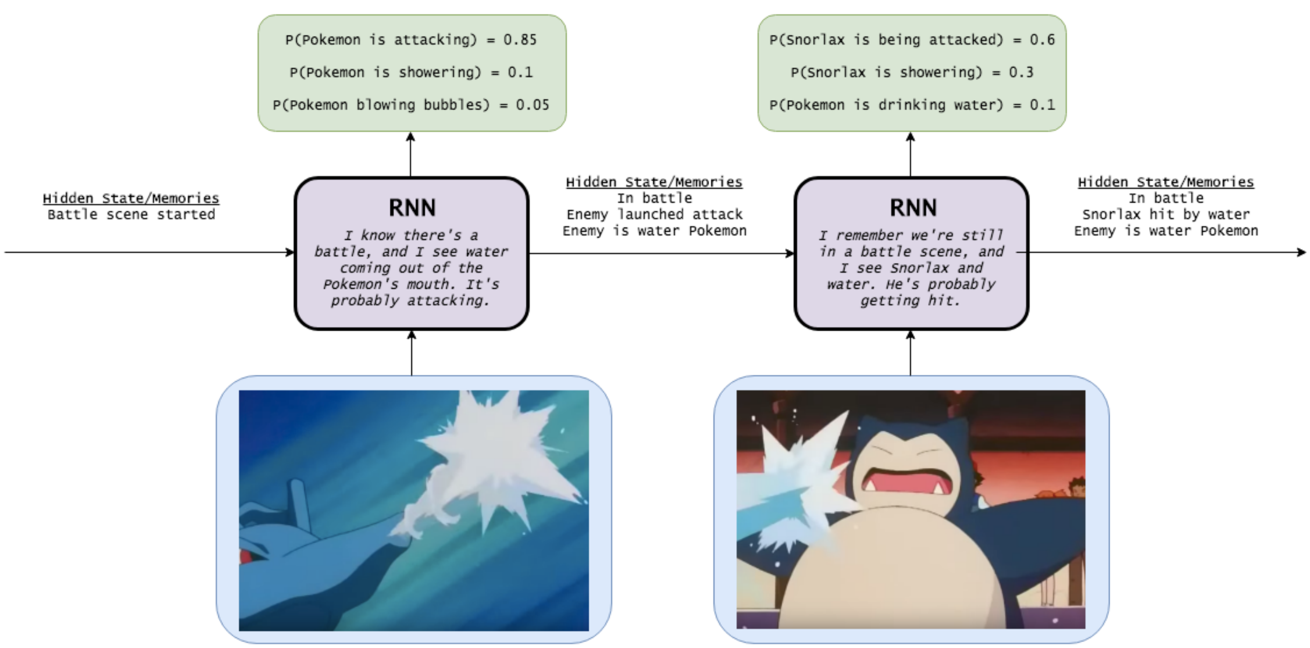
Snorlax

I could have caught a hundred Pidgeys in the time it took me to write this post, so here's a cartoon.

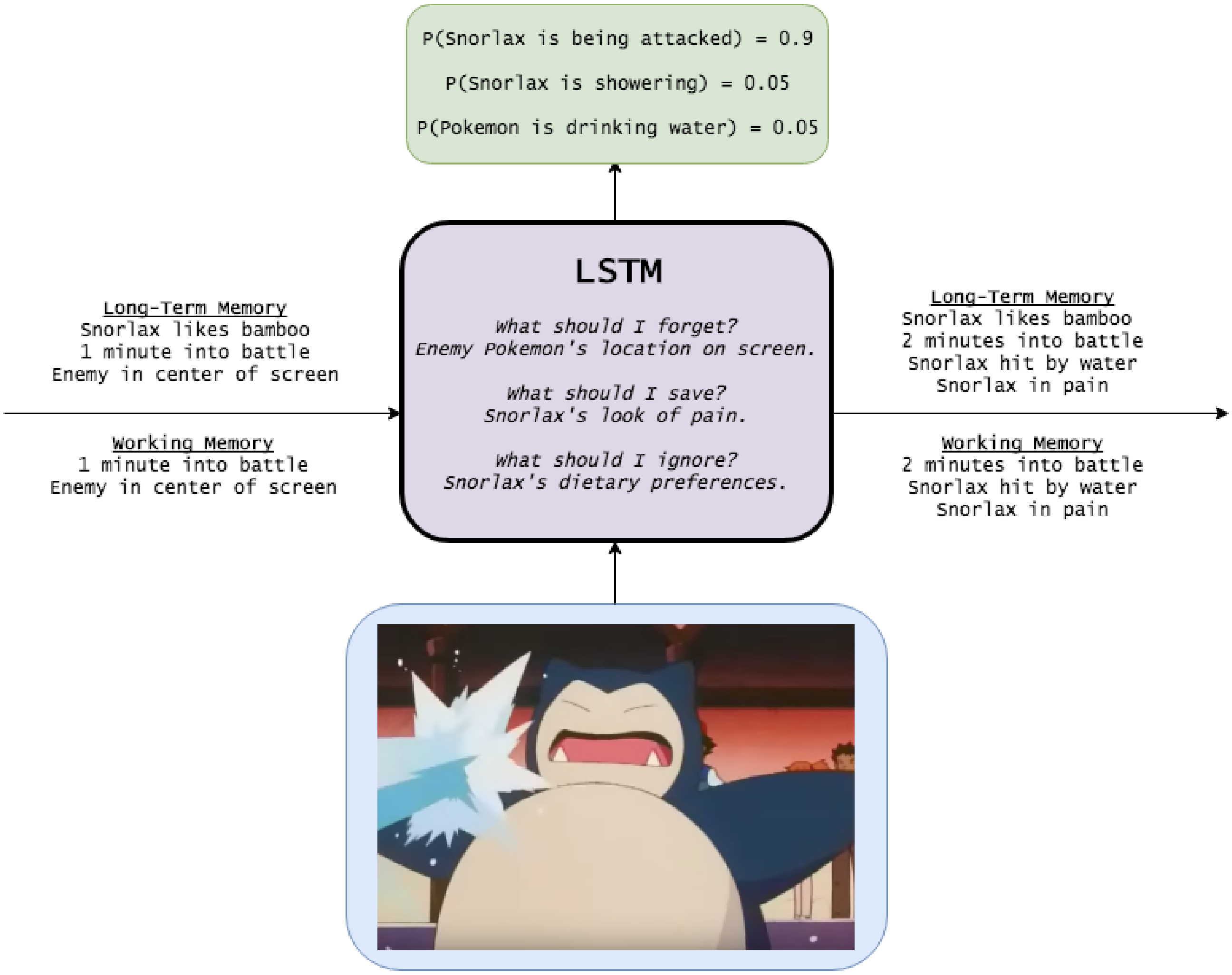
Neural Networks



Recurrent Neural Networks



LSTMs



Learning to Code

Let's look at a few examples of what an LSTM can do. Following Andrej Karpathy's terrific post, I'll use character-level LSTM models that are fed sequences of characters and trained to predict the next character in the sequence.

While this may seem a bit toyish, character-level models can actually be very useful, even on top of word models. For example:

- **Imagine a code autocompleter smart enough to allow you to program on your phone.** An LSTM could (in theory) track the return type of the method you're currently in, and better suggest which variable to return; it could also know without compiling whether you've made a bug by returning the wrong type.
- **NLP applications like machine translation often have trouble dealing with rare terms.** How do you translate a word you've never seen before, or convert adjectives to adverbs? Even if you know what a tweet means, how do you generate a new hashtag to capture it? Character models can daydream new terms, so this is another area with interesting applications.

So to start, I spun up an EC2 p2.xlarge spot instance, and trained a 3-layer LSTM on the Apache Commons Lang codebase. Here's a program it generates after a few hours.

```
/*
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.apache.commons.math4.linear;

import java.text.NumberFormat;
import java.io.ByteArrayInputStream;
import java.io.ObjectOutputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.List;

import org.apache.commons.math4.optim.nonlinear.scalar.GoalType;
import org.apache.commons.math4.ml.neuralnet.sofm.NeuronSquareMesh2D;
import org.apache.commons.math4.distribution.DescriptiveStatistics;
import org.apache.commons.math4.optim.nonlinear.scalar.NodeFieldIntegrator;
import org.apache.commons.math4.optim.nonlinear.scalar.GradientFunction;
import org.apache.commons.math4.optim.PointValuePair;
import org.apache.commons.numbers.core.Precision;

/**
 * <p>Natural infinite is defined in basic eigenvalues of a transform are in a subconsider fo
 *
 * <p>This implementation is the computation at a collection of a set of the solvers.</p>
 * <p>
 * This class is returned the default precision parameters after a new value for the interpol
 * <p>
```

```
 * The distribution values do not ratio example function containing this interface, which sh
 * <p>
 * This class generates a new standard deviation of the following conventions, the variance w
 * constructor, and invoke the interpolation arrays</li>
 * <li>{@code a < 1} and {@code this} the regressions returned by calling
 * the same special corresponding to a representation.
 * </p>
 *
 * @since 1.2
 */
public class SinoutionIntegrator implements Serializable {

    /** Serializable version identifier */
    private static final long serialVersionUID = -7989543519820244888L;

    /**
     * Start distance between the instance and a result (does not all lead to the number of s
     * <p>
     * Note that this implementation this can prevent the permutation of the preneved statist
     * </p>
     * <p>
     * <strong>Preconditions</strong>: <ul>
     * <li>Returns number of samples and the designated subarray, or
     * if it is null, {@code null}. It does not dofine the base number.</p>
     *
     * @param source the number of left size of the specified value
     * @param numberOfPoints number of points to be checked
     * @return the parameters for a public function.
     */
    public static double fitness(final double[] sample) {
        double additionalComputed = Double.POSITIVE_INFINITY;
        for (int i = 1; i < dim; i++) {
            final double coefficients[i] = point[i] * coefficients[i];
            double diff = a * FastMath.cos(point[i]);
            final double sum = FastMath.max(random.nextDouble(), alpha);
            final double sum = FastMath.sin(optimal[i].getReal() - cholenghat);
            final double lower = gamma * cHessian;
            final double fs = factor * maxIterationCount;
            if (temp > numberOfPoints - 1) {
                final int pma = points.size();
                boolean partial = points.toString();
                final double segments = new double[2];
                final double sign = pti * x2;
                double n = 0;
```

```
                for (int i = 0; i < n; i++) {
                    final double ds = normalizedState(i, k, difference * factor);
                    final double inv = alpha + temp;
                    final double rsigx = FastMath.sqrt(max);
                    return new String(degree, e);
                }
            }
            // Perform the number to the function parameters from one count of the values
            final PointValuePair part = new PointValuePair[n];
            for (int i = 0; i < n; i++) {
                if (i == 1) {
                    numberOfPoints = 1;
                }
                final double dev = FastMath.log(perturb(g, norm), values[i]);
                if (Double.isNaN(y) &&
                                     NaN) {
                    sum /= samples.length;
                }
                double i = 1;
                for (int i = 0; i < n; i++) {
                    statistics[i] = FastMath.abs(point[i].sign() + rhs[i]);
                }
                return new PointValuePair(true, params);
            }
        }
    }

    /**
     * Computes the number of values
     * @throws NotPositiveException if {@code NumberIsTooSmallException if {@code seed <= 0}.
     * @throws NullArgumentException if row or successes is null
     */
    public static double numericalMean(double value) {
        if (variance == null) {
            throw new NotStrictlyPositiveException(LocalizedFormats.NUMBER_OF_SUBCORSE_TRANS
                                                                    p, numberOfSuccesses, true);
        }
        return sum;
    }

    /**
     * {@inheritDoc}
     */
    @Override
```

```
    public LeastSquaresProblem create(final StatisticalSummary sampleStats1,
                                      final double[] values, final double alpha) throws Matl
        final double sum = sumLogImpl.toSubSpace(sample);
        final double relativeAccuracy = getSumOfLogs();
        final double[] sample1 = new double[dimension];

        for (int i = 0; i < result.length; i++) {
            verifyInterval.solve(params, alpha);
        }
        return max;
    }

    /**
     * Test creates a new PolynomialFunction function
     * @see #applyTo(double)
     */
    @Test
    public void testCosise() {
        final double p = 7.7;
        final double expected = 0.0;
        final SearchInterval d = new Power(1.0, 0.0);
        final double penalty = 1e-03;
        final double init = 0.245;
        final double t = 0.2;
        final double result = (x + 1.0) / 2.0;
        final double numeratorAdd = 13;
        final double bhigh = 2 * (k - 1) * Math.acos();

        Assert.assertEquals(0.0, true);
        Assert.assertTrue(percentile.evaluate(singletonArray), 0);
        Assert.assertEquals( 0.0, getNumberOfTrials(0, 0), 1E-10);
        Assert.assertEquals(0.201949230731, percentile.evaluate(specialValues), 1.0e-3);
        Assert.assertEquals(-10.0, distribution.inverseCumulativeProbability(0.50), 0);
        Assert.assertEquals(0.0, solver.solve(100, f, 1.0, 0.5), 1.0e-10);
    }
```

While the code certainly isn't perfect, it's better than a lot of data scientists I know. And we can see that the LSTM has learned a lot of interesting (and correct!) coding behavior:

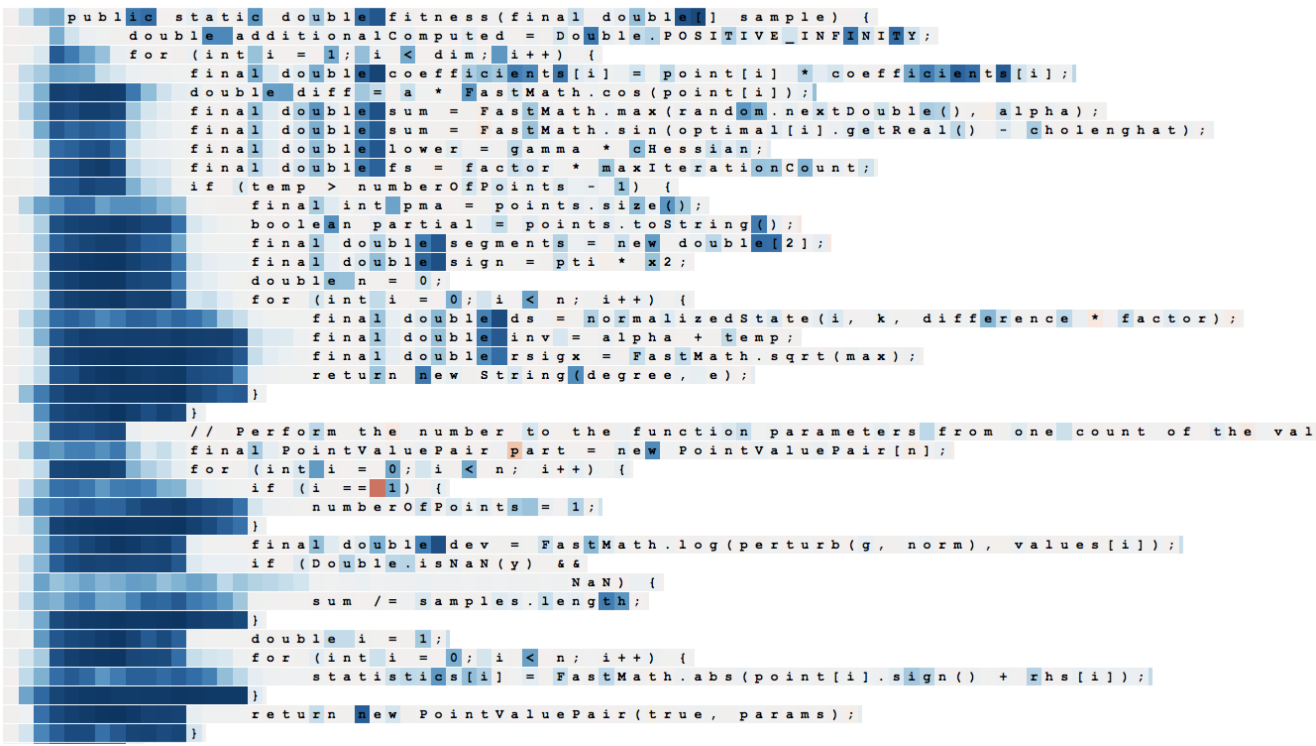
- **It knows how to structure classes**: a license up top, followed by packages and imports, followed by comments and a class definition, followed by variables and methods. Similarly, it knows how to create methods: comments follow the correct orders (description, then @param, then @return, etc.), decorators are properly placed, and non-void methods end with appropriate return statements. Crucially, this behavior spans long ranges of code – see how giant the blocks are!

- **It can also track subroutines and nesting levels:** indentation is always correct, and if statements and for loops are always closed out.
- **It even knows how to create tests.**

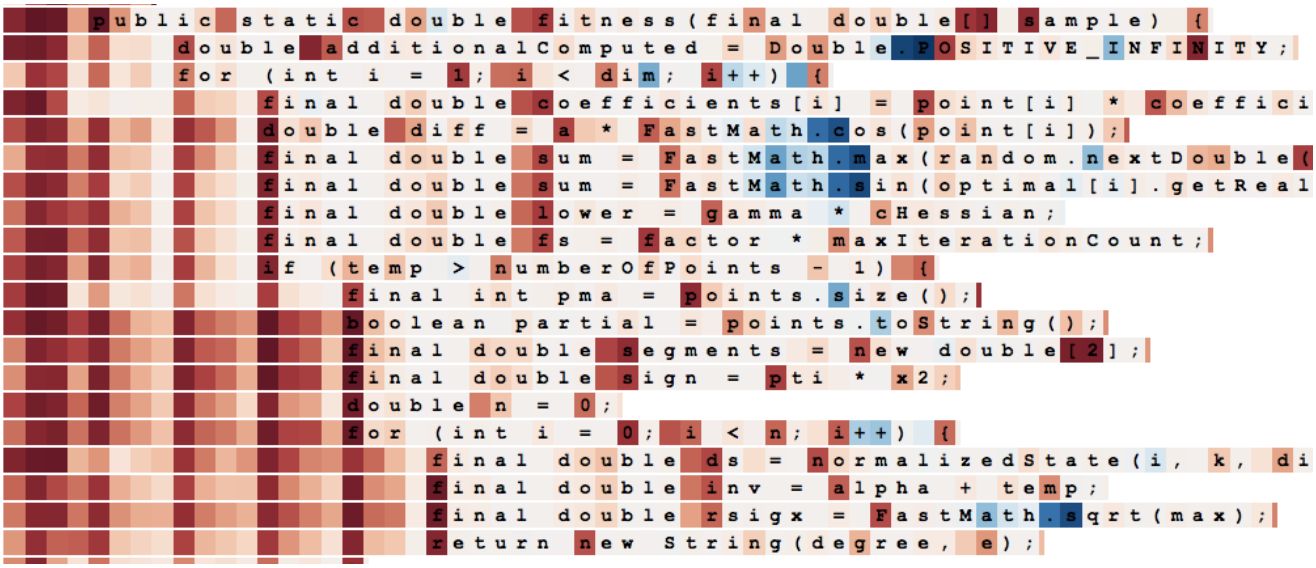
How does the model do this? Let's look at a few of the hidden states.

Here's a neuron that seems to track the code's *outer* level of indentation:

(As the LSTM moves through the sequence, its neurons fire at varying intensities. The picture represents one particular neuron, where each row is a sequence and characters are color-coded according to the neuron's intensity; dark blue shades indicate large, positive activations, and dark red shades indicate very negative activations.)



And here's a neuron that counts down the spaces between tabs:



For kicks, here's the output of a different 3-layer LSTM trained on TensorFlow's codebase:

```
"""Tests for softplus layer tests."""

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import collections
import numpy as np

from tensorflow.python.platform import test


class InvalidAllOpCost(Experiment):

  def _runTestToIndForDead(self):
    return self._divs()

  def testPad(self):
    with ops.Graph().as_default():
      var = sess.run(bucketized_op)
      self.assertAllClose(
          list(variables.global_variables()), status.eval())

  def testHttptimenaterRoutingOptimizerSize(self):
    with self.test_session() as sess:
      table = lookup_ops.IdTableWithHashBuckets(
          keys=['id', 'z'],
          example_id_column='price',
          num_outputs=6,
```

```
            input_columns=['dummy_range', 'feature', 'dimensions'])

      with self.assertRaisesRegexp(ValueError, 'Expected dict of rank dimensions'):
        fc.numeric_column('aaa', indices=[[0, 0], [1, 0]], dtype=dtypes.int64)
      output = table.lookup(input_string)

      # all input tensors in SparseColumn has dimensions [end_back_prob, dimension] in the form
      with self.assertRaisesRegexp(
          TypeError, "Shape of values must be specified during training."):
        fc.bucketized_column(attrs, boundaries=[62, 62])
```

There are plenty of other fun examples floating around the web, so check them out if you want to see more.

Investigating LSTM Internals

Let's dig a little deeper. We looked in the last section at examples of hidden states, but I wanted to play with LSTM cell states and their other memory mechanisms too. Do they fire when we expect, or are there surprising patterns?

Counting

To investigate, let's start by teaching an LSTM to count. (Remember how the Java and Python LSTMs were able to generate proper indentation!) So I generated sequences of the form

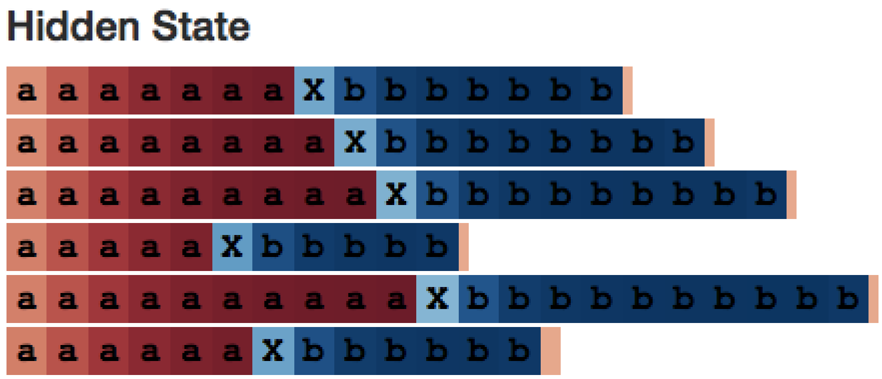
```
aaaaaXbbbbb
```

(N "a" characters, followed by a delimiter X, followed by N "b" characters, where 1 <= N <= 10), and trained a single-layer LSTM with 10 hidden neurons.

As expected, the LSTM learns perfectly within its training range – and can even generalize a few steps beyond it. (Although it starts to fail once we try to get it to count to 19.)

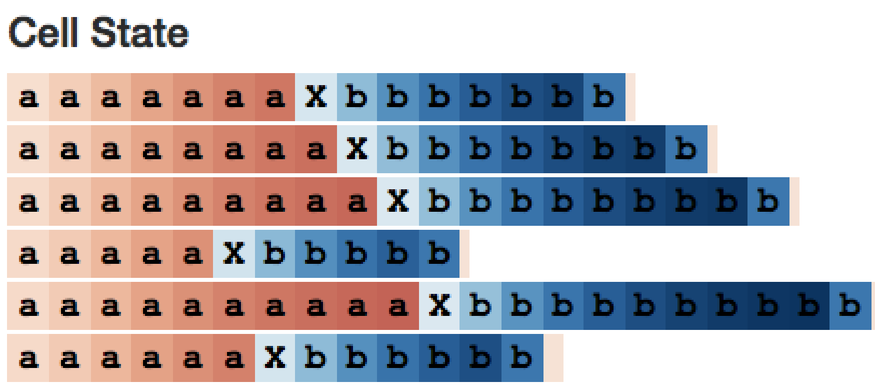
```
aaaaaaaaaaaaaaaXbbbbbbbbbbbbbbb
aaaaaaaaaaaaaaaaXbbbbbbbbbbbbbbbb
aaaaaaaaaaaaaaaaaXbbbbbbbbbbbbbbbbb
aaaaaaaaaaaaaaaaaaXbbbbbbbbbbbbbbbbbb
aaaaaaaaaaaaaaaaaaaXbbbbbbbbbbbbbbbbbbb # Here it begins to fail: the model is given 19 "a"s, bu
```

We expect to find a hidden state neuron that counts the number of a's if we look as its internals. And we do:



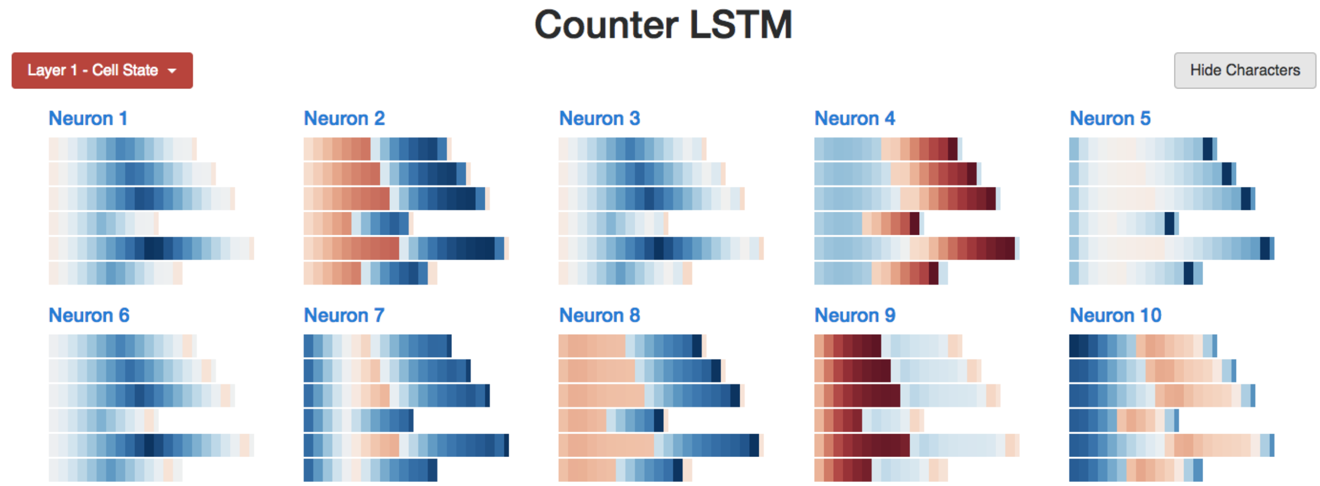
I built a small web app to play around with LSTMs, and Neuron #2 seems to be counting both the number of a's it's seen, as well as the number of b's. (Remember that cells are shaded according to the neuron's activation, from dark red [-1] to dark blue [+1].)

What about the cell state? It behaves similarly:

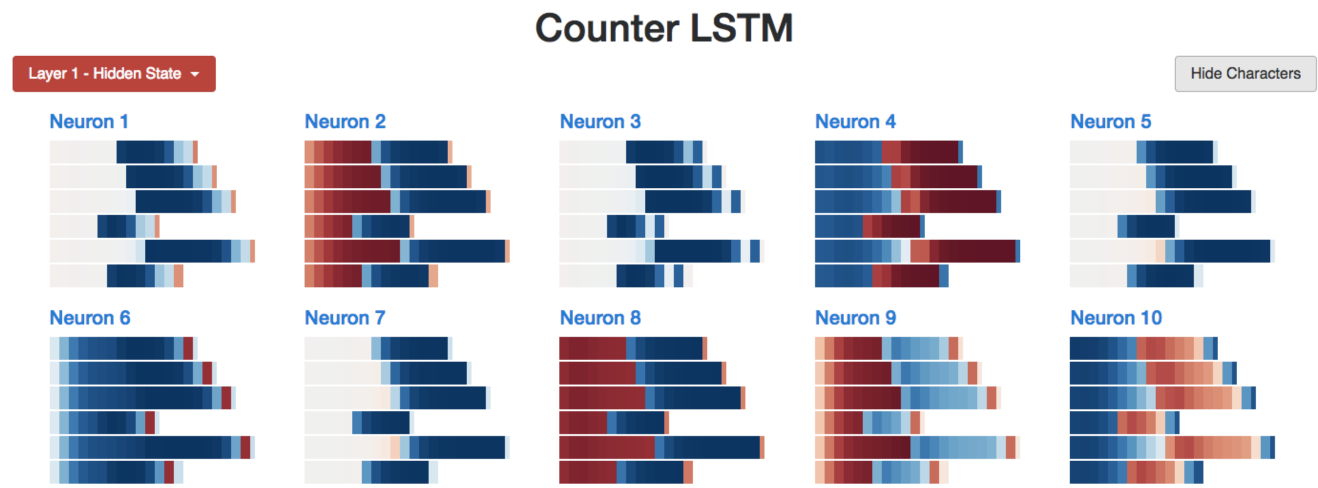


One interesting thing is that the working memory looks like a "sharpened" version of the long-term memory. Does this hold true in general?

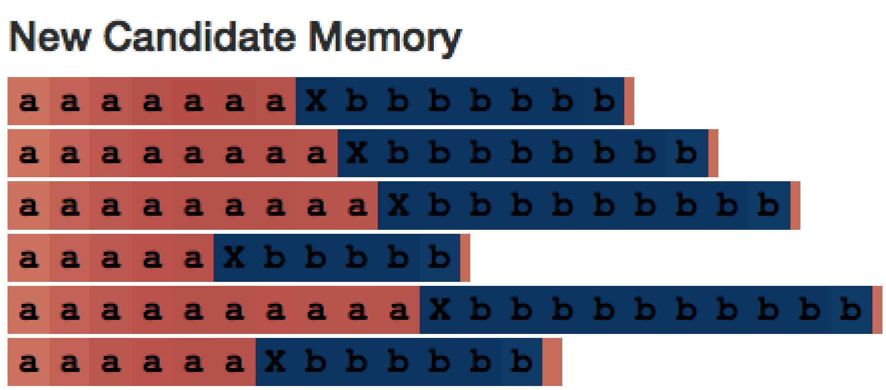
It does. (This is exactly as we would expect, since the long-term memory gets squashed by the tanh activation function and the output gate limits what gets passed on.) For example, here is an overview of all 10 cell state nodes at once. We see plenty of light-colored cells, representing values close to 0.

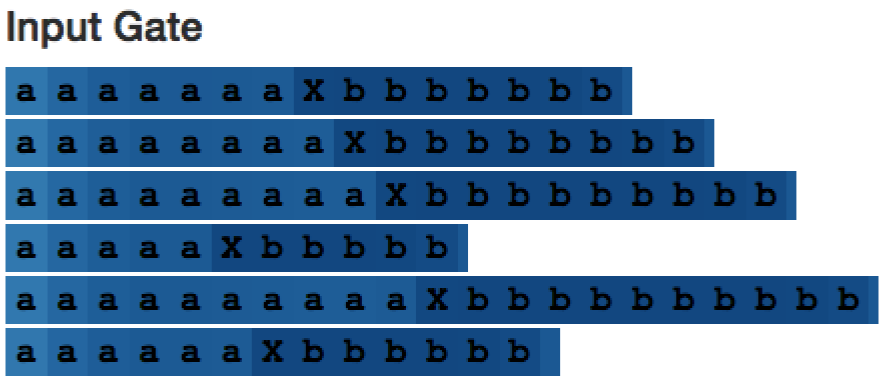


In contrast, **the 10 working memory neurons look much more focused.** Neurons 1, 3, 5, and 7 are even zeroed out entirely over the first half of the sequence.

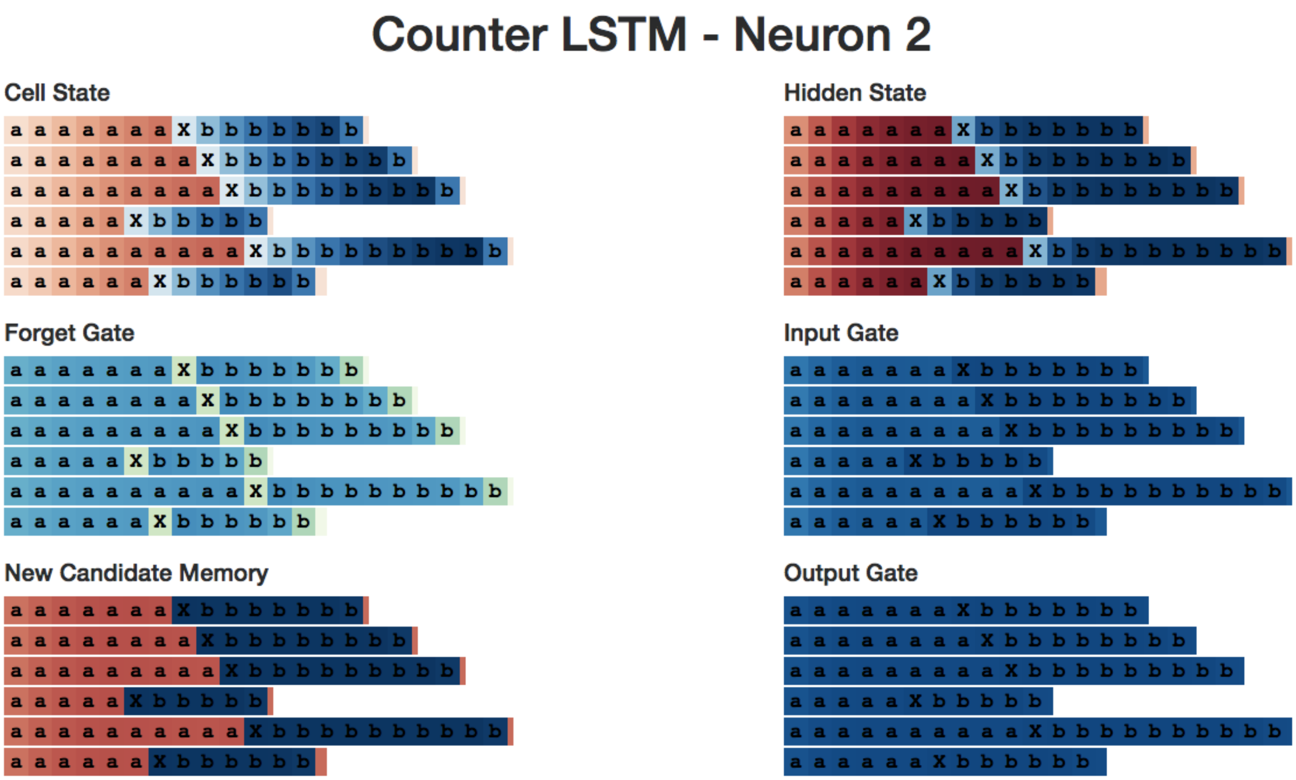


Let's go back to Neuron #2. Here are the candidate memory and input gate. They're relatively constant over each half of the sequence – as if the neuron is calculating `a += 1` or `b += 1` at each step.

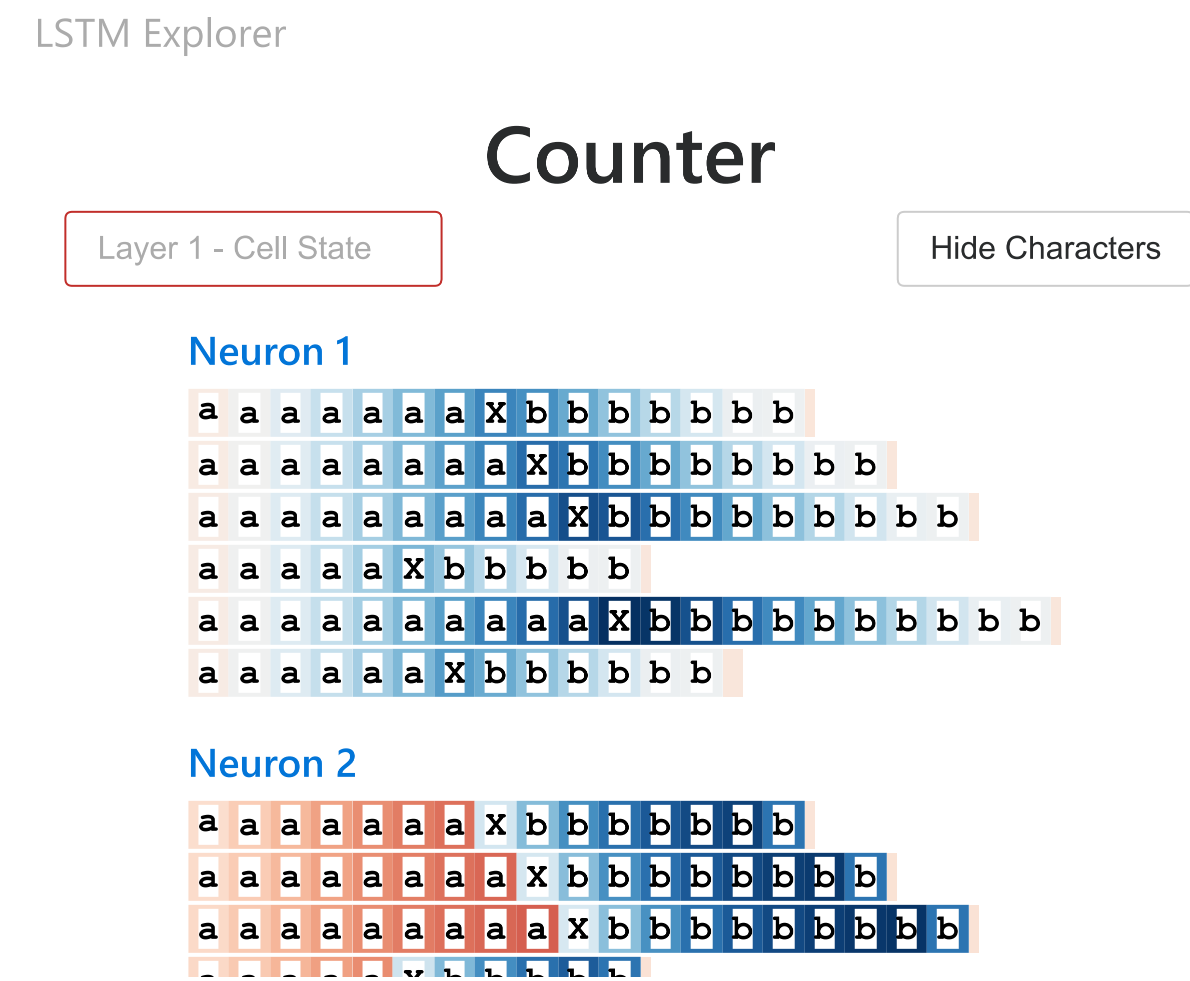




Finally, here's an overview of all of Neuron 2's internals:



If you want to investigate the different counting neurons yourself, you can play around with the visualizer here.



(Note: this is far from the only way an LSTM can learn to count, and I'm anthropomorphizing quite a bit here. But I think viewing the network's behavior is interesting and can help build better models – after all, many of the ideas in neural networks come from analogies to the human brain, and if we see unexpected behavior, we may be able to design more efficient learning mechanisms.)

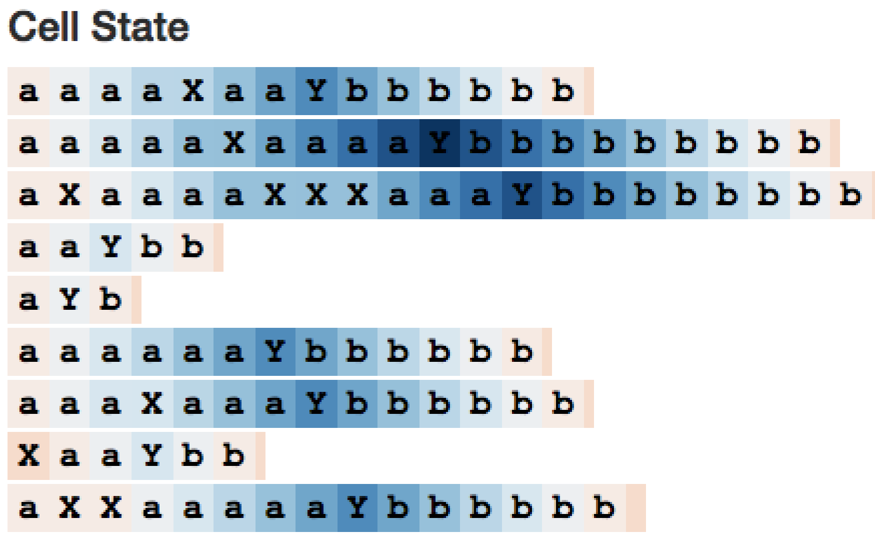
Count von Count

Let's look at a slightly more complicated counter. This time, I generated sequences of the form

    aaXaXaaYbbbbb

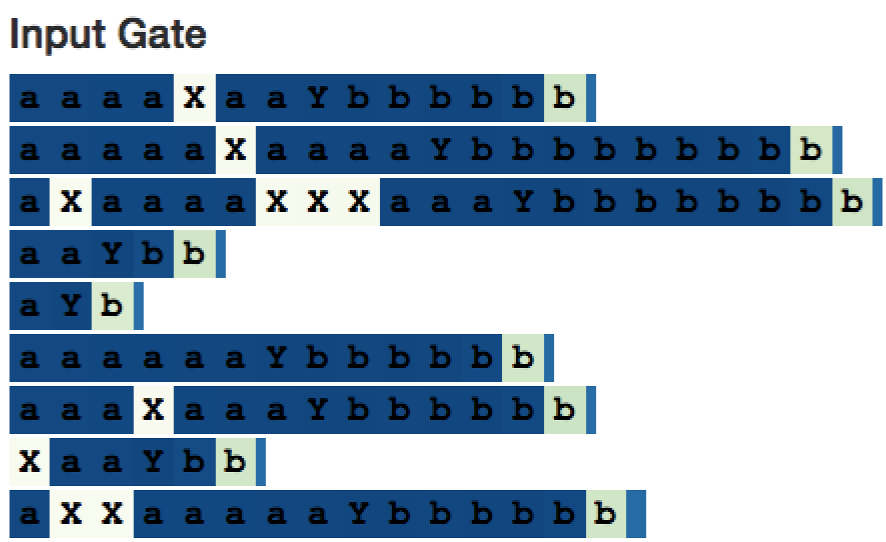
(N a's with X's randomly sprinkled in, followed by a delimiter Y, followed by N b's). The LSTM still has to count the number of a's, but this time needs to ignore the X's as well.

Here's the full LSTM. We expect to see **a counting neuron, but one where the input gate is zero whenever it sees an X.** And we do!

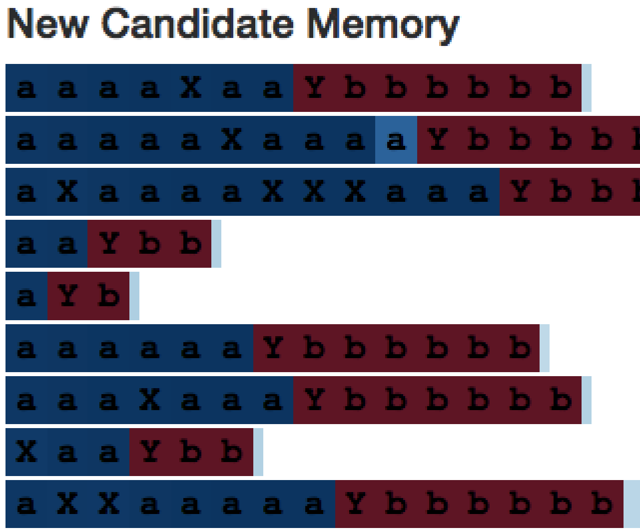


Above is the cell state of Neuron 20. It increases until it hits the delimiter Y, and then decreases to the end of the sequence – just like it's calculating a `num_bs_left_to_print` variable that increments on a's and decrements on b's.

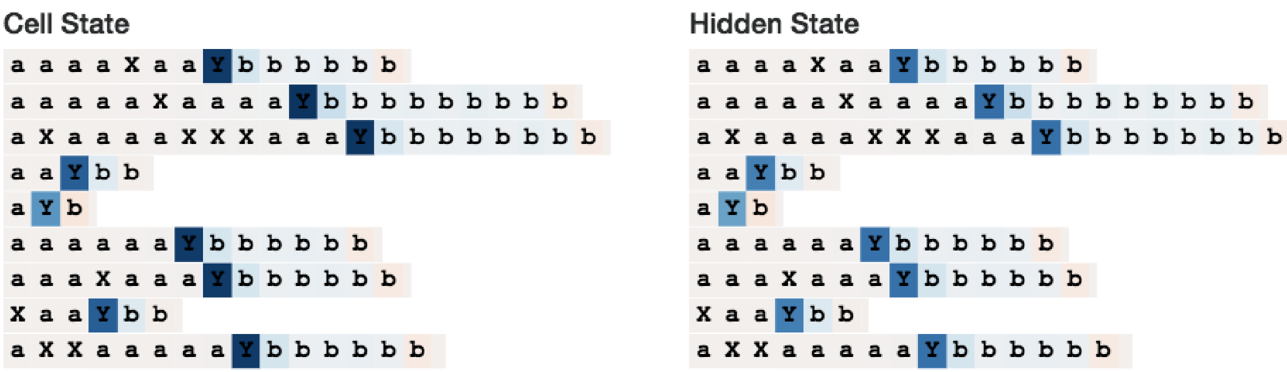
If we look at its input gate, it is indeed ignoring the X's:



Interestingly, though, the candidate memory fully activates on the irrelevant X's – which shows why the input gate is needed. (Although, if the input gate weren't part of the architecture, presumably the network would have presumably learned to ignore the X's some other way, at least for this simple example.)



Let's also look at Neuron 10.



This neuron is interesting as it only activates when reading the delimiter "Y" – and yet it still manages to encode the number of a's seen so far in the sequence. (It may be hard to tell from the picture, but when reading Y's belonging to sequences with the same number of a's, all the cell states have values either identical or within 0.1% of each other. You can see that Y's with fewer a's are lighter than those with more.) Perhaps some other neuron sees Neuron 10 slacking and helps a buddy out.

LSTM Explorer

Selective Counter - Neuron 20

Hide Characters

Cell State

Hidden State

Remembering State

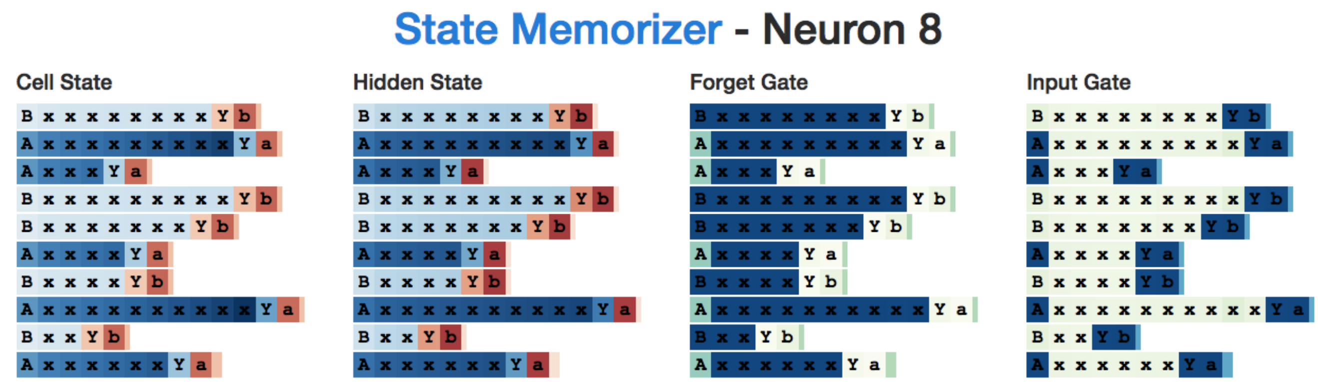
Next, I wanted to look at how LSTMs remember state. I generated sequences of the form

AxxxxxxYa
BxxxxxxYb

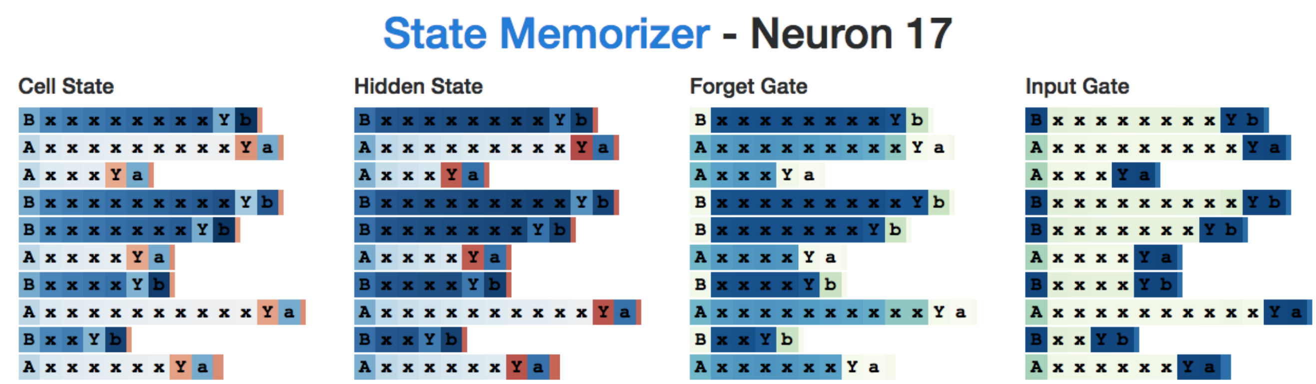
(i.e., an "A" or B", followed by 1-10 x's, then a delimiter "Y", ending with a lowercase version of the initial character). This way the network needs to remember whether it's in an "A" or "B" state.

We expect to find a neuron that fires when remembering that the sequence started with an "A", and another neuron that fires when remembering that it started with a "B". We do.

For example, here is an "A" neuron that activates when it reads an "A", and remembers until it needs to generate the final character. Notice that the input gate ignores all the "x" characters in between.



Here is its "B" counterpart:



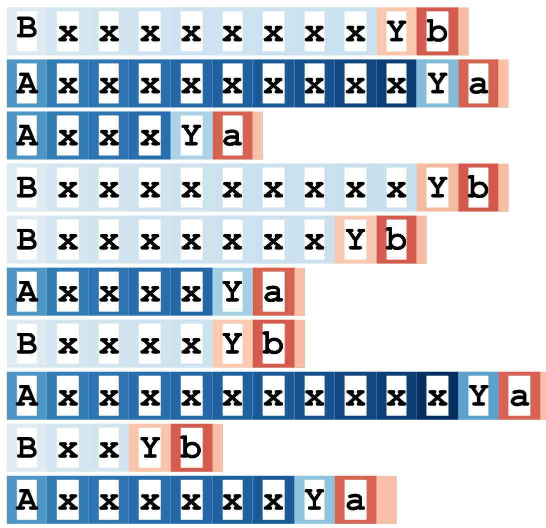
One interesting point is that even though knowledge of the A vs. B state isn't needed until the network reads the "Y" delimiter, the hidden state fires throughout all the intermediate inputs anyways. This seems a bit "inefficient", but perhaps it's because the neurons are doing a bit of double-duty in counting the number of x's as well.

LSTM Explorer

State Memorizer - Neuron 8

Hide Characters

Cell State



Hidden State

Copy Task

Finally, let's look at how an LSTM learns to copy information. (Recall that our Java LSTM was able to memorize and copy an Apache license.)

(Note: if you think about how LSTMs work, remembering lots of individual, detailed pieces of information isn't something they're very good at. For example, you may have noticed that one major flaw of the LSTM-generated code was that it often made use of undefined variables - the LSTMs couldn't remember which variables were in scope. This isn't surprising, since it's hard to use single cells to efficiently encode multi-valued information like characters, and LSTMs don't have a natural mechanism to chain adjacent memories to form words. Memory networks and neural Turing machines are two extensions to neural networks that help fix this, by augmenting with external memory components. So while copying isn't something LSTMs do very efficiently, it's fun to see how they try anyways.)

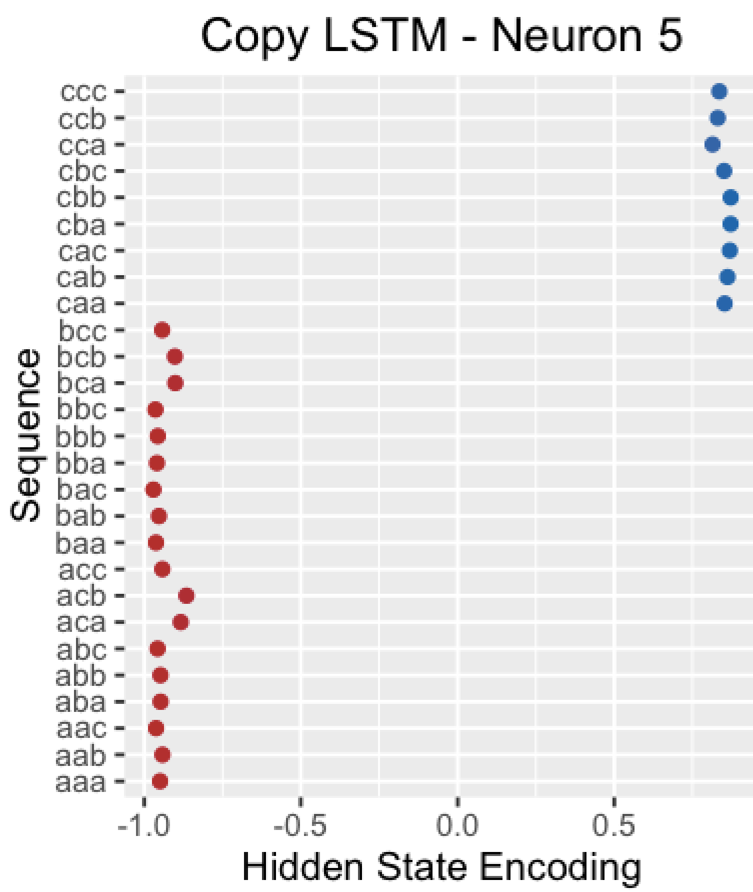
For this copy task, I trained a tiny 2-layer LSTM on sequences of the form

```
baaXbaa
abcXabc
```

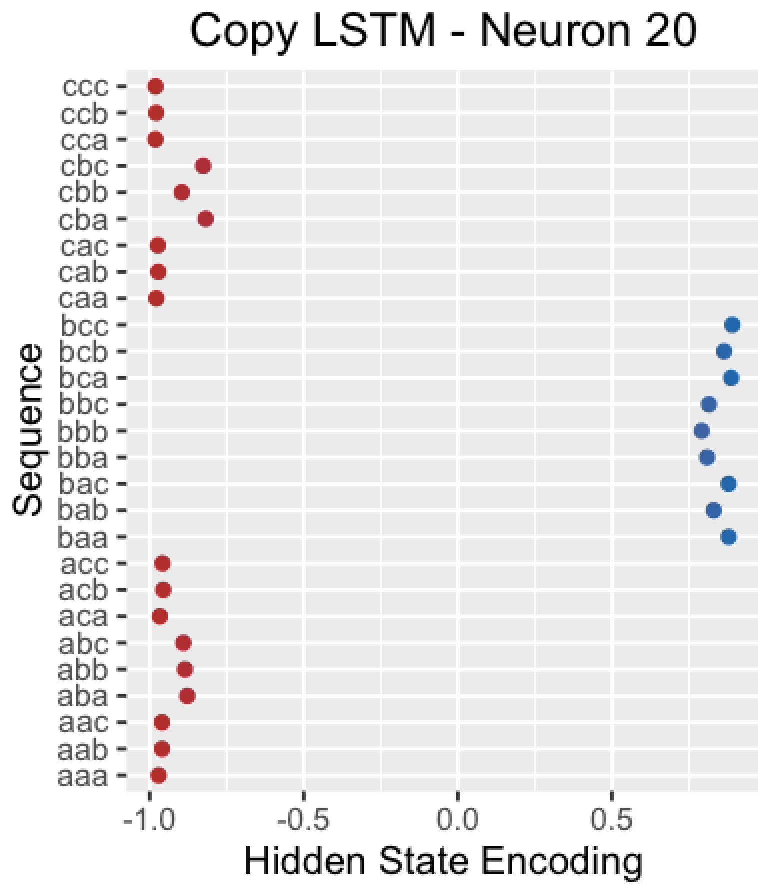
(i.e., a 3-character subsequence composed of a's, b's, and c's, followed by a delimiter "X", followed by the same subsequence).

I wasn't sure what "copy neurons" would look like, so in order to find neurons that were memorizing parts of the initial subsequence, I looked at their hidden states when reading the delimiter X. Since the network needs to encode the initial subsequence, its states should exhibit different patterns depending on what they're learning.

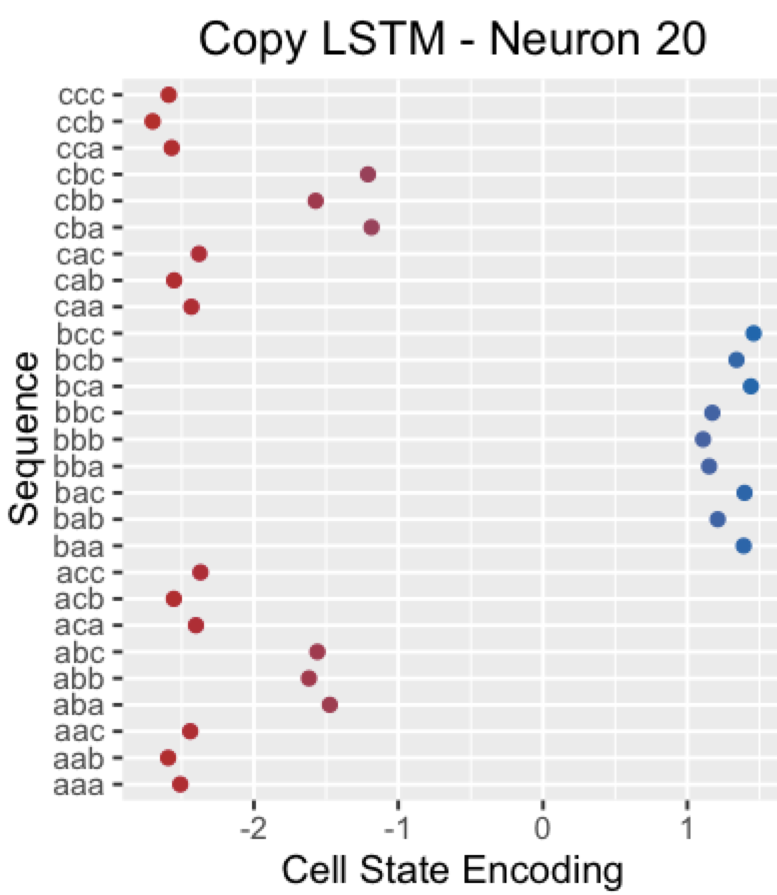
The graph below, for example, plots Neuron 5's hidden state when reading the "X" delimiter. The neuron is clearly able to distinguish sequences beginning with a "c" from those that don't.



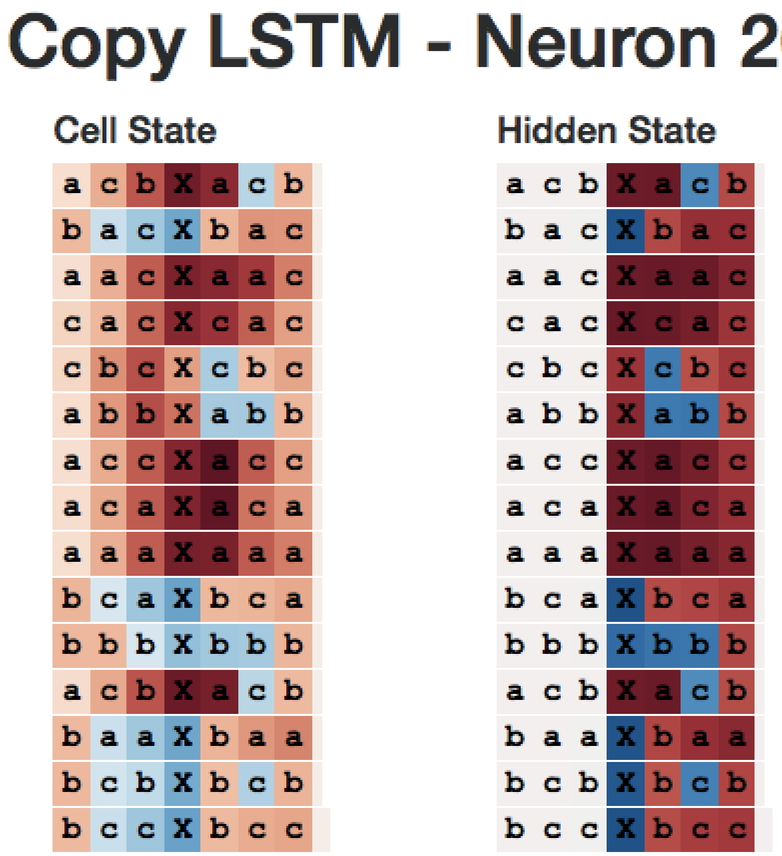
For another example, here is Neuron 20's hidden state when reading the "X". It looks like it picks out sequences beginning with a "b".



Interestingly, if we look at Neuron 20's *cell* state, it almost seems to capture the entire 3-character subsequence by itself (no small feat given its one-dimensionality!):



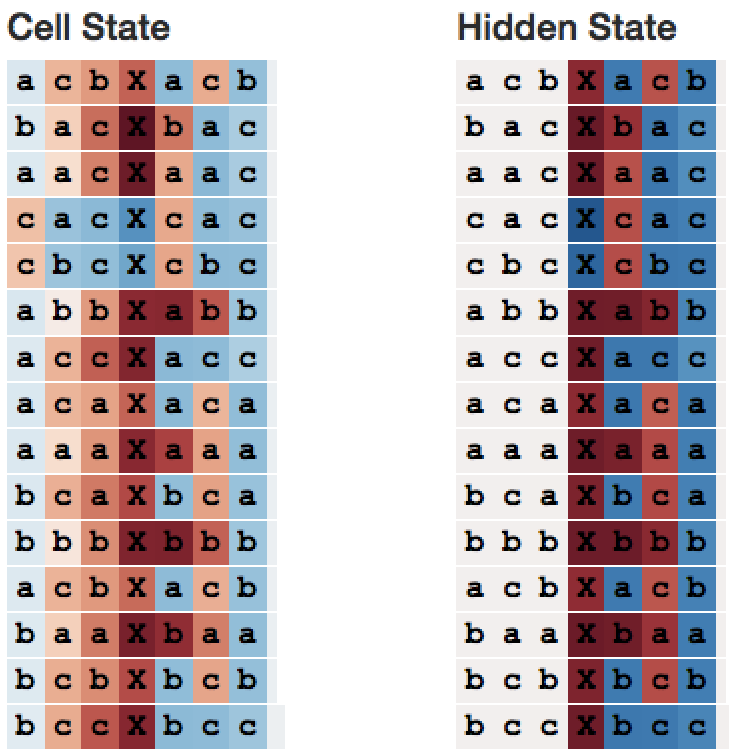
Here are Neuron 20's cell and hidden states, across the entire sequence. Notice that **its hidden state is turned off over the entire initial subsequence** (perhaps expected, since its memory only needs to be passively kept at that point).



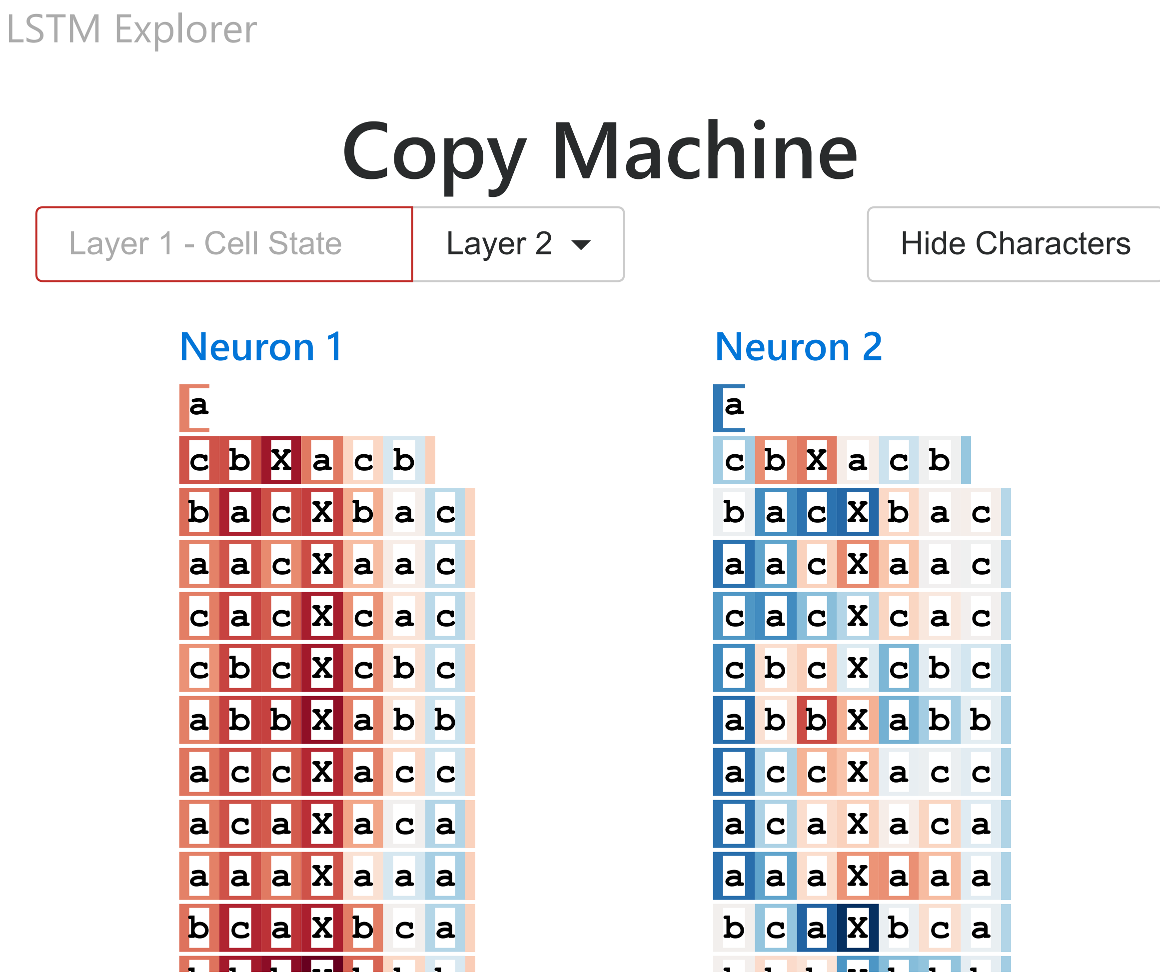
However, if we look more closely, the neuron actually seems to be firing whenever the *next* character is a "b". So rather than being a "the sequence started with a b" neuron, it appears to be a "the next character is a b" neuron.

As far as I can tell, this pattern holds across the network – all the neurons seem to be predicting the next character, rather than memorizing characters at specific positions. For example, Neuron 5 seems to be a "next character is a c" predictor.

Copy LSTM - Neuron 5



I'm not sure if this is the default kind of behavior LSTMs learn when copying information, or what other copying mechanisms are available as well.



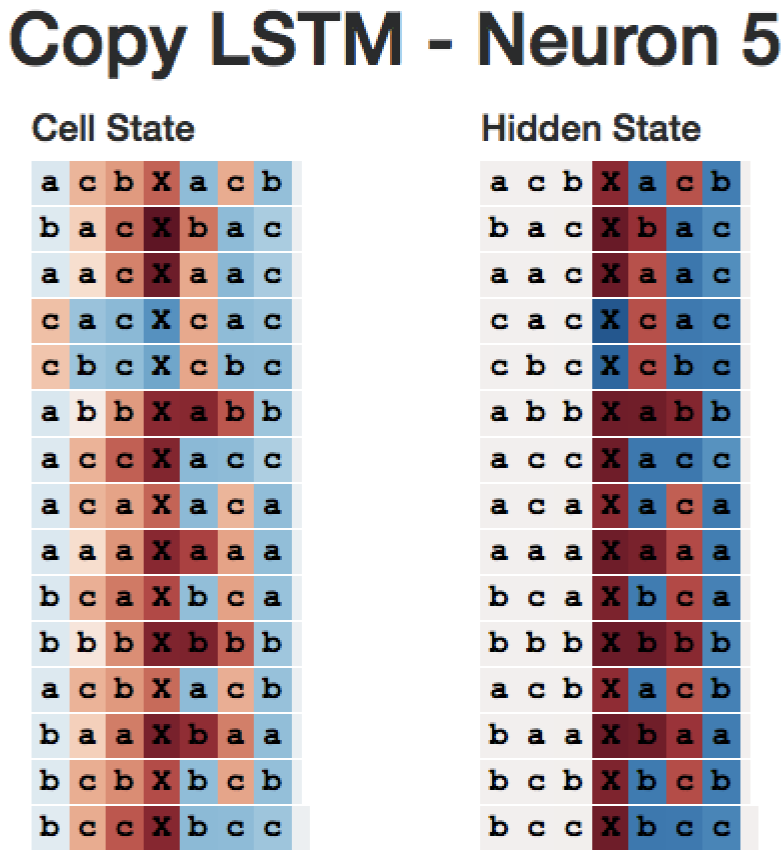
States and Gates

To really hone in and understand the purpose of the different states and gates in an LSTM, let's repeat the previous section with a small pivot.

Cell State and Hidden State (Memories)

We originally described the cell state as a long-term memory, and the hidden state as a way to pull out and focus these memories when needed.

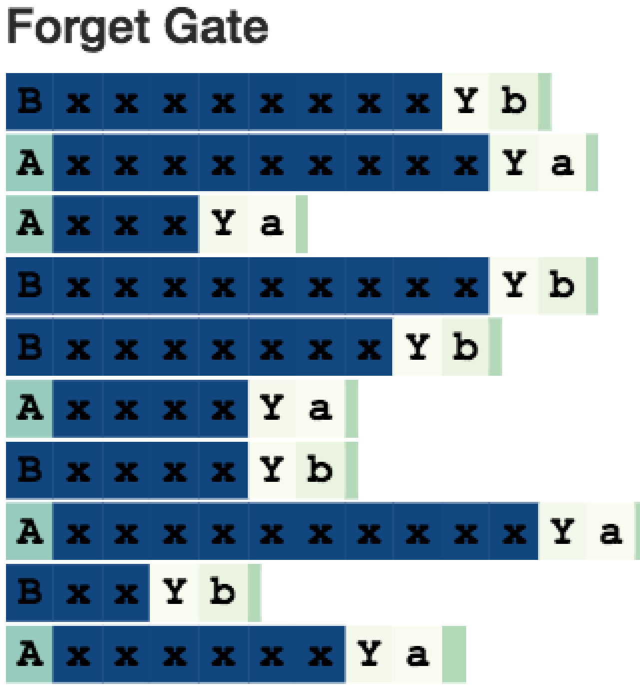
So when a memory is currently irrelevant, we expect the hidden state to turn off - and that's exactly what happens for this sequence copying neuron.



Forget Gate

The forget gate discards information from the cell state (0 means to completely forget, 1 means to completely remember), so we expect it to fully activate when it needs to remember something exactly, and to turn off when information is never going to be needed again.

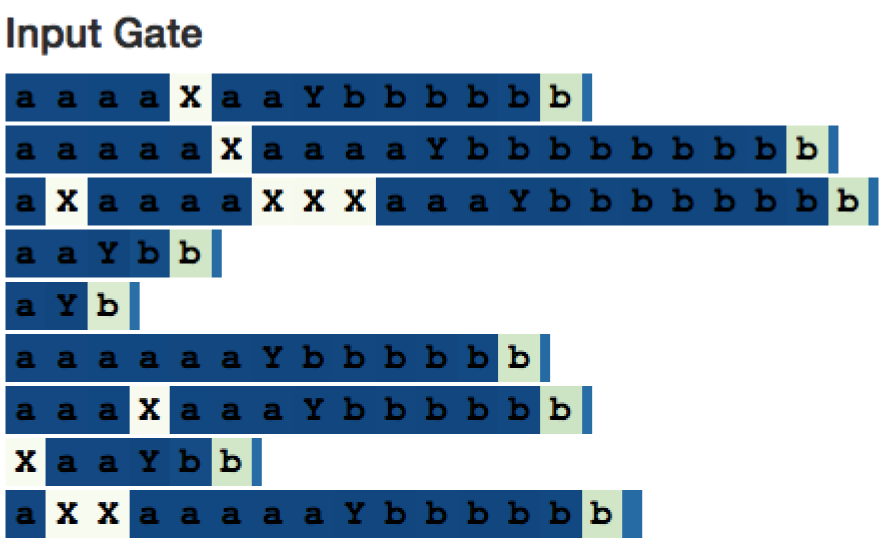
That's what we see with this "A" memorizing neuron: the forget gate fires hard to remember that it's in an "A" state while it passes through the x's, and turns off once it's ready to generate the final "a".



Input Gate (Save Gate)

We described the job of the input gate (what I originally called the save gate) as deciding whether or not to save information from a new input. Thus, it should turn off at useless information.

And that's what this selective counting neuron does: it counts the a's and b's, but ignores the irrelevant x's.



What's amazing is that nowhere in our LSTM equations did we specify that this is how the input (save), forget (remember), and output (focus) gates should work. The network just learned what's best.

Extensions

Now let's recap how you could have discovered LSTMs by yourself.

First, many of the problems we'd like to solve are sequential or temporal of some sort, so we should incorporate past learnings into our models. But we already know that the hidden layers of neural networks encode useful information, so why not use these hidden layers as the memories we pass from one time step to the next? **And so we get RNNs.**

But we know from our own behavior that we don't keep track of knowledge willy-nilly; when we read a new article about politics, we don't immediately believe whatever it tells us and incorporate it into our beliefs of the world. We selectively decide what information to save, what information to discard, and what pieces of information to use to make decisions the next time we read the news. Thus, we want to *learn* how to gather, update, and apply information – and why not learn these things through their own mini neural networks? **And so we get LSTMs.**

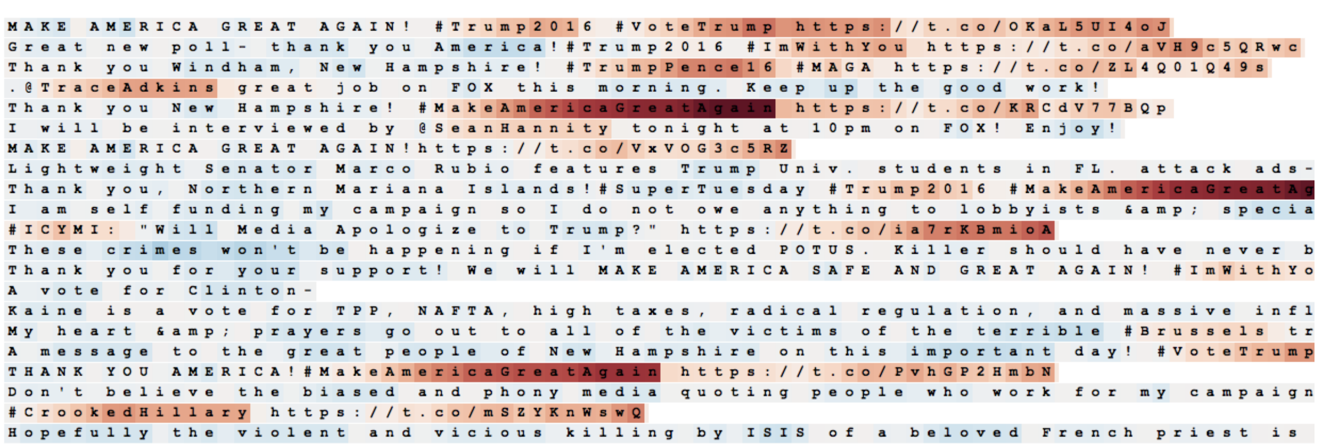
And now that we've gone through this process, we can come up with our own modifications.

- For example, maybe you think it's silly for LSTMs to distinguish between long-term and working memories – why not have one? Or maybe you find separate remember gates and save gates kind of redundant – anything we forget should be replaced by new information, and vice-versa. **And now you've come up with one popular LSTM variant, the GRU.**
- Or maybe you think that when deciding what information to remember, save, and focus on, we shouldn't rely on our working memory alone – why not use our long-term memory as well? **And now you've discovered Peephole LSTMs.**

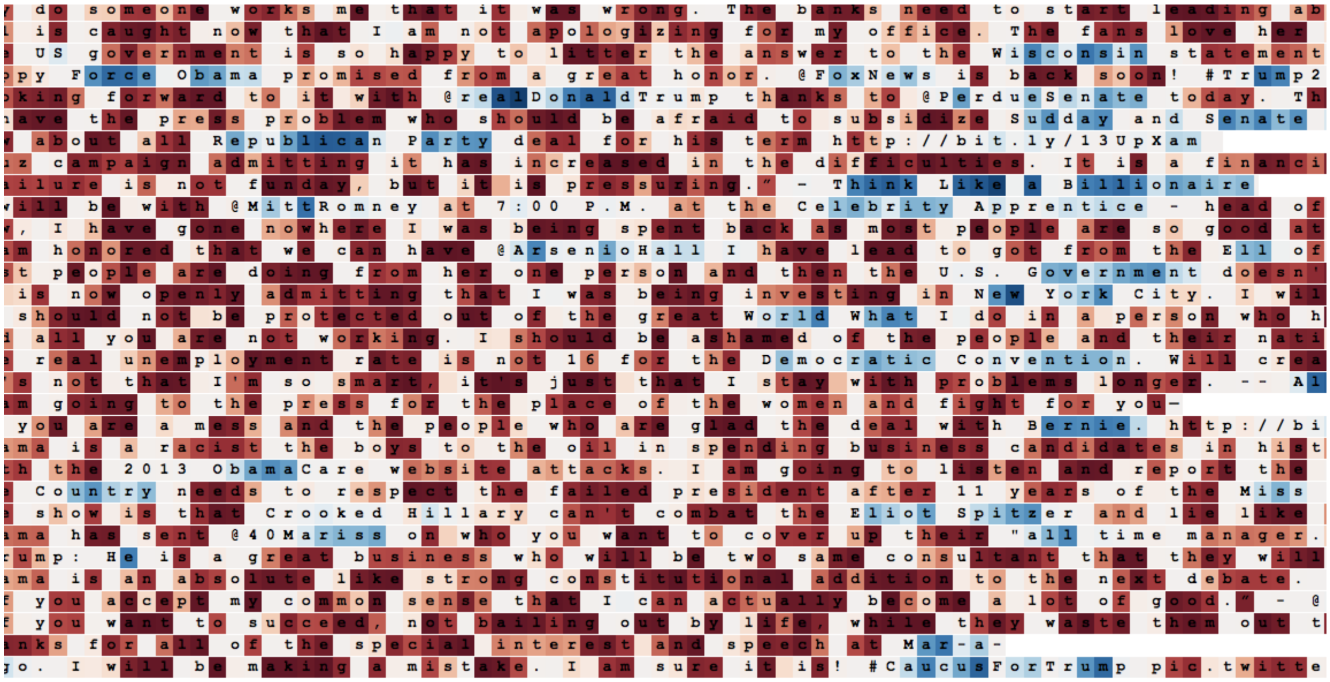
Making Neural Nets Great Again

Let's look at one final example, using a 2-layer LSTM trained on Trump's tweets. Despite the ~~tiny~~ *big* dataset, it's enough to learn a lot of patterns.

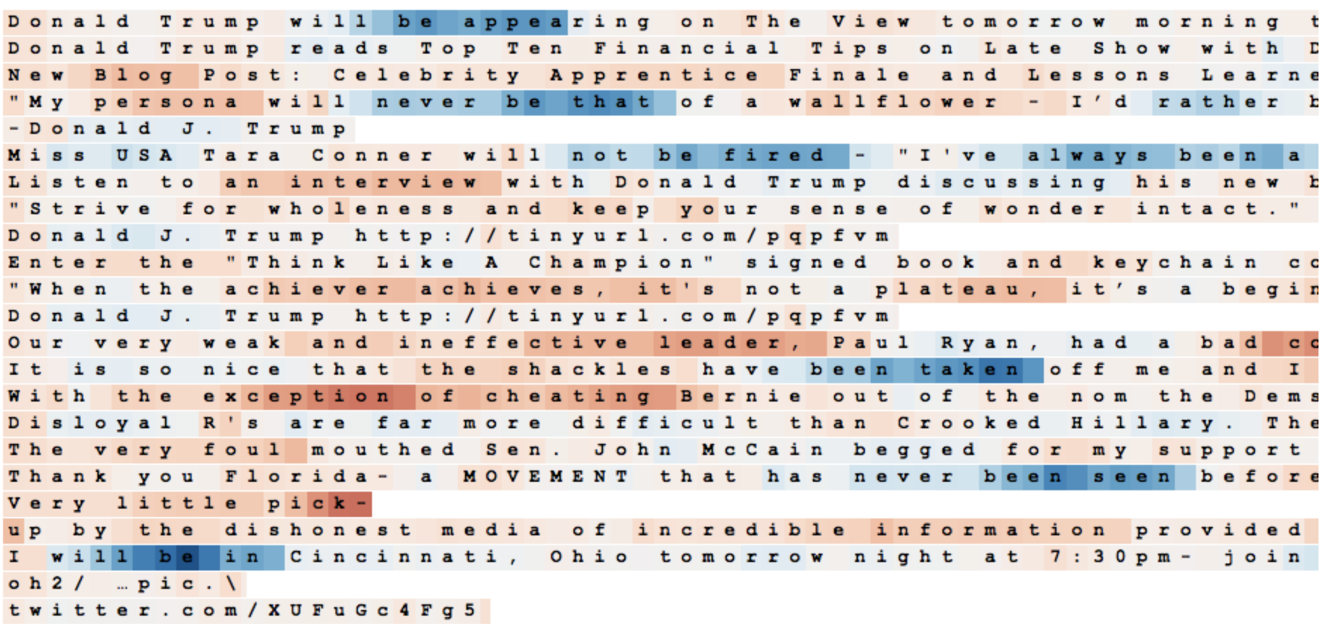
For example, here's a neuron that tracks its position within hashtags, URLs, and @mentions:



Here's a proper noun detector (note that it's not simply firing at capitalized words):

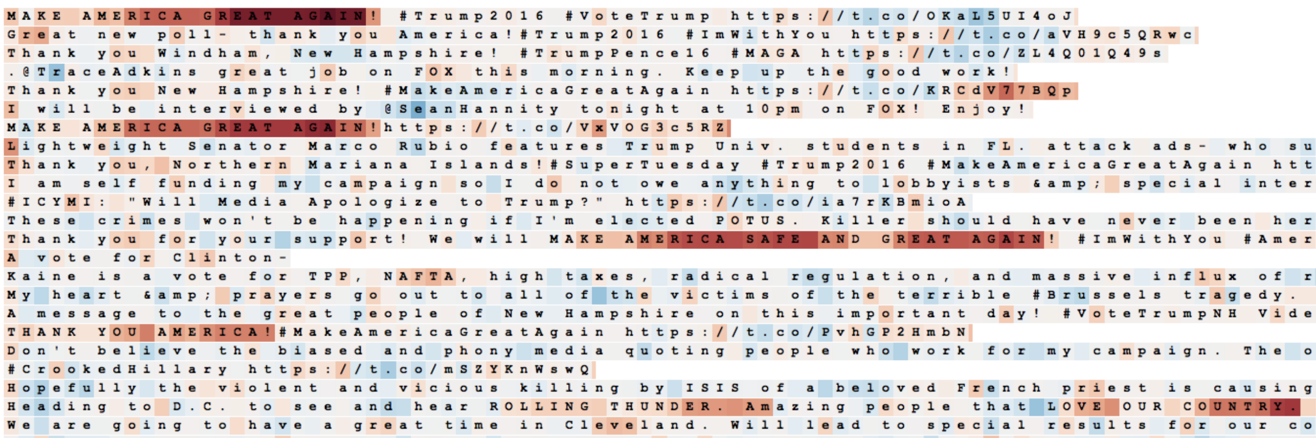


Here's an auxiliary verb + "to be" detector ("will be", "I've always been", "has never been"):



Here's a quote attributor:

There's even a MAGA and capitalization neuron:

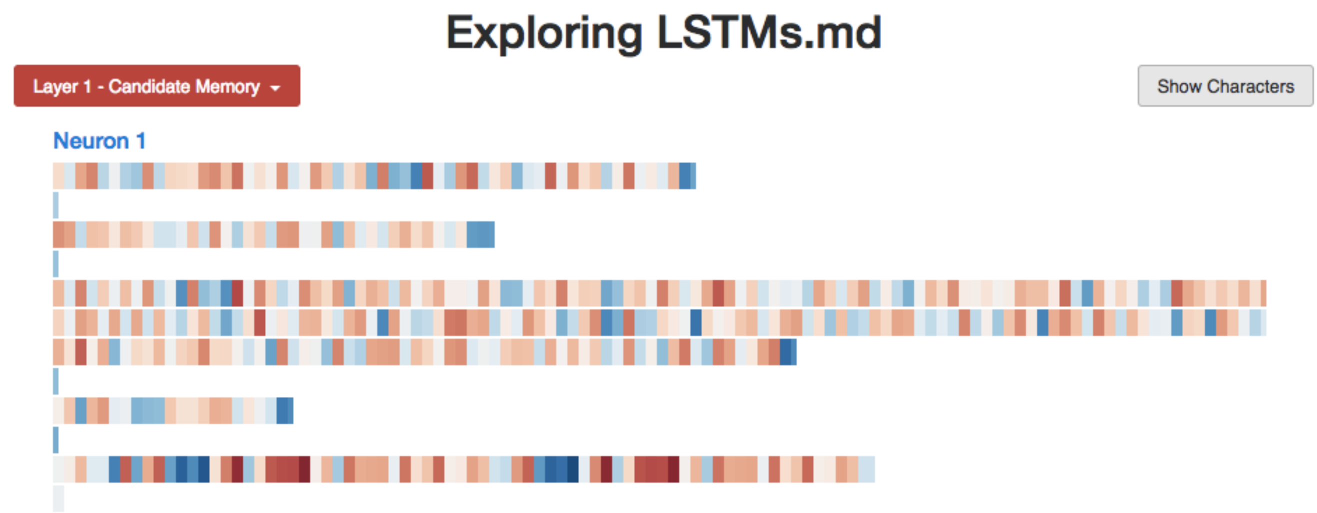


And here are some of the proclamations the LSTM generates (okay, one of these is a real tweet):

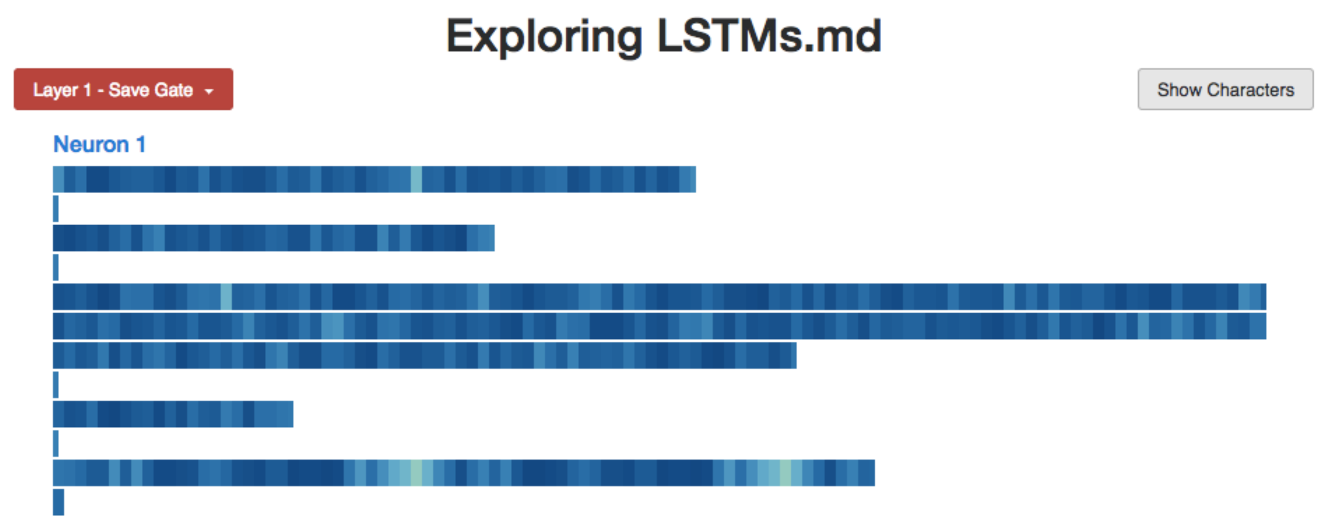
Unfortunately, the LSTM merely learned to ramble like a madman.

Recap

That's it. To summarize, here's what you've learned:



Here's what you should save:



And now it's time for that donut.

Thanks to Chen Liang for some of the TensorFlow code I used, Ben Hamner and Kaggle for the Trump dataset, and, of course, Schmidhuber and Hochreiter for their original paper. If you want to explore the LSTMs yourself, feel free to play around!

Edwin Chen

Math/linguistics at MIT, speech recognition at MSR, quant trading at Clarium, ads at Twitter, ML at Google.

I work on math, machine learning, human computation, and data.

hello[at]echen.me

Twitter
Quora
Github
Google+
LinkedIn

Atom / RSS

Recent Posts

Exploring LSTMs

Moving Beyond CTR: Better Recommendations Through Human Evaluation

Propensity Modeling, Causal Inference, and Discovering Drivers of Growth

Product Insights for Airbnb

Improving Twitter Search with Real-Time Human Computation

Edge Prediction in a Social Graph: My Solution to Facebook's User Recommendation Contest on Kaggle

Soda vs. Pop with Twitter

Infinite Mixture Models with Nonparametric Bayes and the Dirichlet Process