joshvarty.github.io /VisualizingRNNs/

Unknown Title

9-11 minutes

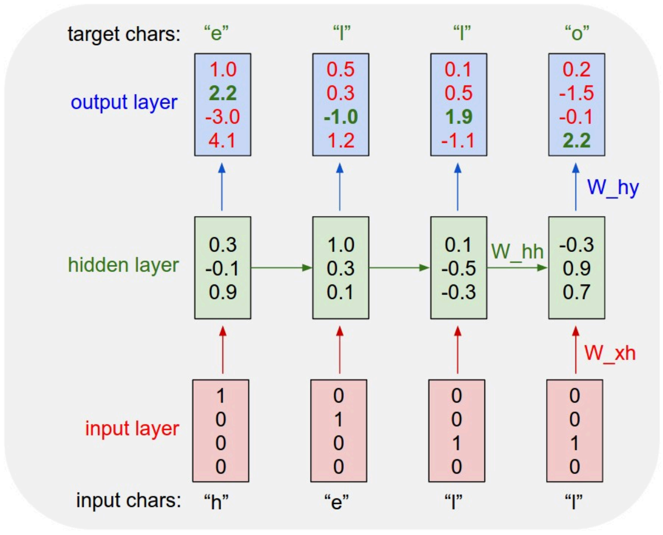
After reading about the unreasonable effectiveness of recurrent neural networks I know:

- They work with sequences (eg. stock data, natural language)
- We can train them with backprop, the same way we do with convolutional nets
- They are flexible and work with a variety of structures
    - **One-to-many**: Take an image and return a caption for it.
    - **Many-to-one**: Take a movie review and return if it's positive or negative
    - **Many-to-Many**: Take an English sentence and return a Spanish sentence

One thing I struggle with is visualizing what's actually going on. In particular I have a lot of trouble reconciling **this code:**

```
class RNN:
# ...
def step(self, x):
# update the hidden state
self.h = np.tanh(np.dot(self.W_hh, self.h) + np.dot(self.W_xh, x))
# compute the output vector
y = np.dot(self.W_hy, self.h)
return y
```

with **this image:**



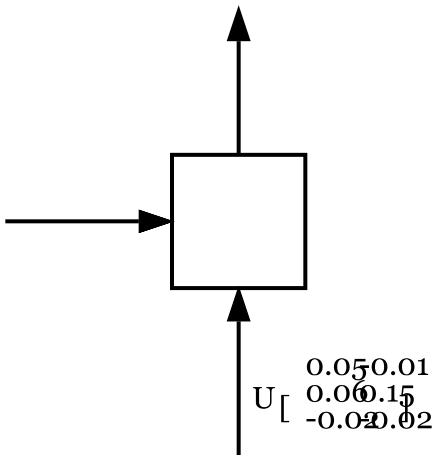
To help me understand exactly what's going on I want visualize each step of the forward pass. What do our inputs look like? What do our weights look like? How do things look at the beginning of training when compared to the end?

In order to tackle this we're going to have to use very small weight matrices or things will get out of hand very quickly. Even Andrej Karpathy's simple Min-Char RNN has weight matrices with **thousands** of parameters. No matter how hard we try, we won't be able to put all those numbers on screen.

An Untrained RNN

With simplicity in mind, instead of predicting the next character in English text (as Min-Char RNN does), let’s build a network that predicts the next output in the sequence:
`1,0,1,0,1,0,1,0,1,0 ...`
This seems simple. Maybe even **too** simple, but a sequence like this will allow us to have small weight matrices, the largest of which will be just `3x3`.
For this visualization we’ll use a modified version of Min-Char RNN. Our `vocab_size` is just `2` (each input character is `1` or `0`) and since we’re predicting such a simple pattern we’ll use a `hidden_size` of `3`.
Our network is fed an input of `1,0,1,0` and is tasked with predicting the target output sequence `0,1,0,1`, one character at a time. Since our network is untrained, note that it outputs probabilities of roughly `0.50` at each time step.
Click "Play" below or step through at your own pace:

```
input_weights_U = np.random.randn(hidden_size, vocab_size) * 0.01
hidden_weights_W = np.random.randn(hidden_size, hidden_size) * 0.01
hidden_bias = np.zeros((hidden_size, 1))
output_weights_V = np.random.randn(vocab_size, hidden_size) * 0.01
output_bias = np.zeros((vocab_size, 1))

xs, hidden_states, outputs, probabilities = {}, {}, {}, {}
loss = 0
hidden_states[-1] = np.copy(hidden_state_prev)
for t in range(len(inputs)):

    xs[t] = np.zeros((vocab_size,1))
    character = inputs[t]
    xs[t][character] = 1
    target = targets[t]

    hidden_states[t] = np.tanh(input_weights_U @ xs[t] + hidden_weights_W @
hidden_states[t-1] + hidden_bias)

    outputs[t] = output_weights_V @ hidden_states[t] + output_bias
    probabilities[t] = np.exp(outputs[t]) / np.sum(np.exp(outputs[t]))

    loss += -np.log(probabilities[t][target,0])
```

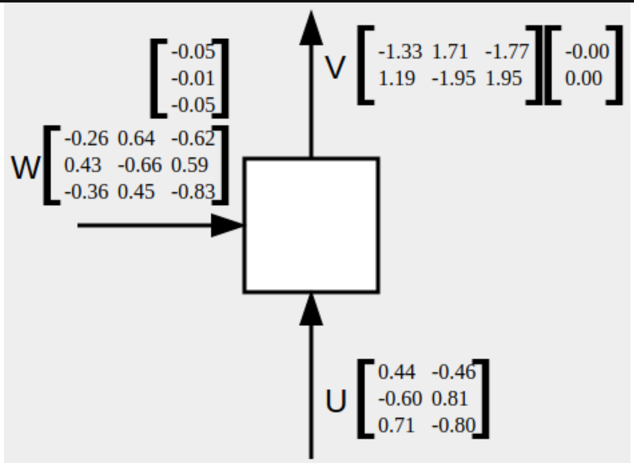


This animation looks pretty cool but you probably won't learn much at first glance. Take a minute to match up the weights in the diagram with the weights in the code. You can hover over them and it will highlight the relationship for you. If you're really feeling up for it, you could work out the matrix multiplications by hand. The big takeaways here are:

- Computing hidden state is most of the work
- We use the same weights for every input of our forward pass
- `hidden_states[t-1]` changes as we go through each step
- We output probabilities for the target character at each step
- Since our network is untrained the output probabilities are:
    - 50% `0`
    - 50% `1`

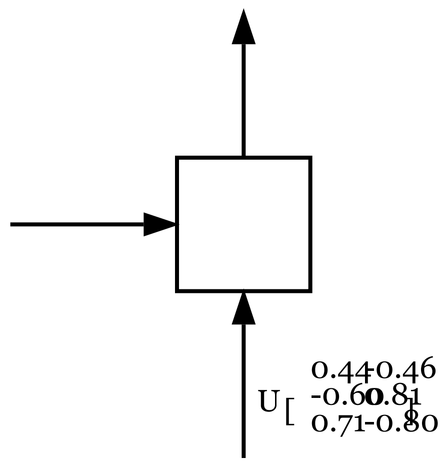
A Trained RNN

So now that we've seen a network that doesn't work, let's take a look at one that does. After training our network with gradient descent (a visualization I'll save for another time) we're left with weights that look like:



So what happens when we run the network using these weights?

```
input_weights_U = ...
hidden_weights_W = ...
hidden_bias = ...
output_weights_V = ...
output_bias = ...

xs, hidden_states, outputs, probabilities = {}, {}, {}, {}
loss = 0
hidden_states[-1] = np.copy(hidden_state_prev)
for t in range(len(inputs)):

    xs[t] = np.zeros((vocab_size,1))
    character = inputs[t]
    xs[t][character] = 1
    target = targets[t]

    hidden_states[t] = np.tanh(input_weights_U @ xs[t] + hidden_weights_W @
hidden_states[t-1] + hidden_bias)

    outputs[t] = output_weights_V @ hidden_states[t] + output_bias
    probabilities[t] = np.exp(outputs[t]) / np.sum(np.exp(outputs[t]))

    loss += -np.log(probabilities[t][target,0])
```



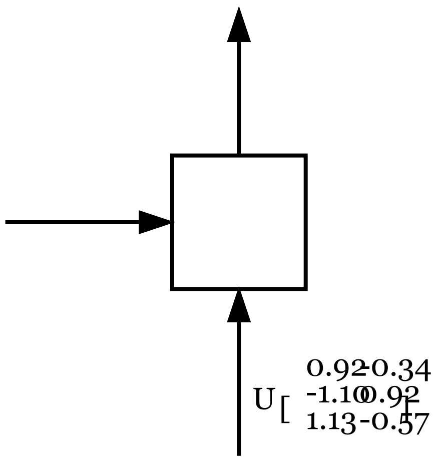
This time our network outputs probabilities for `0` and `1` with almost 100% confidence. If you pay close attention to `hidden_state` you'll notice that it also alternates between `[1,-1,1]` and `[-1,1,-1]`. I can't pretend that I know why this is, but it's interesting to see.
In fact, the more we start thinking about it, the more it's interesting that `hidden_state` learns anything at all. Hidden state is meant to encode useful information about things we've seen in the past, but our problem doesn't depend on past information. Our network should really only care about whatever the current input (`1` or `0`) to our network is.

Making Hidden State Useful

If we want hidden state to be useful, we'll have to give it a problem where it's actually needed. We'll modify our original sequence slightly and have our network predict the next character in `1,1,0,1,1,0,1,1,0, ...`
Now our network can no longer get away with simply predicting the opposite of the input. It will have to take special care to determine whether we're on the first or second `1` when predicting the output.
Below is an RNN trained to respond to input characters `1,1,0,1` with `1,0,1,1`. Keep an eye on `hidden_state` as the animation progresses.

```
input_weights_U = ...
hidden_weights_W = ...
hidden_bias = ...
output_weights_V = ...
output_bias = ...

xs, hidden_states, outputs, probabilities = {}, {}, {}, {}
loss = 0
```

```
hidden_states[-1] = np.copy(hidden_state_prev)
for t in range(len(inputs)):

    xs[t] = np.zeros((vocab_size,1))
    character = inputs[t]
    xs[t][character] = 1
    target = targets[t]

    hidden_states[t] = np.tanh(input_weights_U @ xs[t] + hidden_weights_W @
hidden_states[t-1] + hidden_bias)

    outputs[t] = output_weights_V @ hidden_states[t] + output_bias
    probabilities[t] = np.exp(outputs[t]) / np.sum(np.exp(outputs[t]))

    loss += -np.log(probabilities[t][target,0])
```



This time hidden state is actually encoding useful information we can observe directly. Whenever the network sees the first `1` in the sequence, hidden state is set to `[0.72, 0.58, 0.96]`. You can see this at the first and final steps of the animation. In contrast when the network sees the second `1`, hidden state is set to `[-0.76, 0.89, 0.95]`. Varying hidden state like this allows our network to output the proper probabilities at each step despite our inputs (`1`) and parameters (`U`, `W` and `V`) being the same.

Indeed much of the power of RNNs stems from hidden state and interesting ways of convincing our network to either remember or forget different things about the input sequences. Andrej's original blog post covers this in detail for a number of different domains.

Interested in more about RNNs? Follow me on Twitter at: @ThisIsJoshVarty