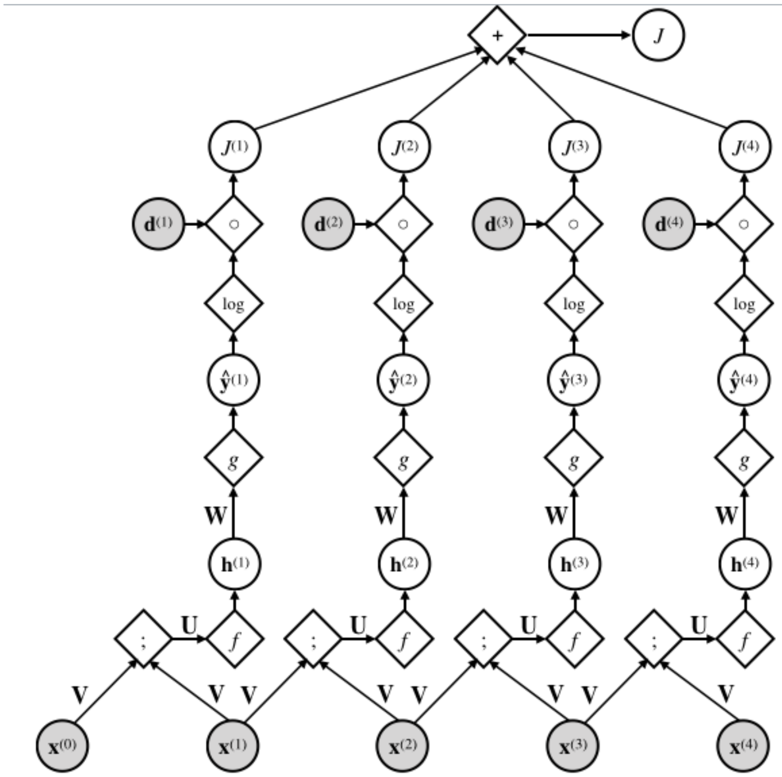
Backpropagation through time, Q1, and Q3

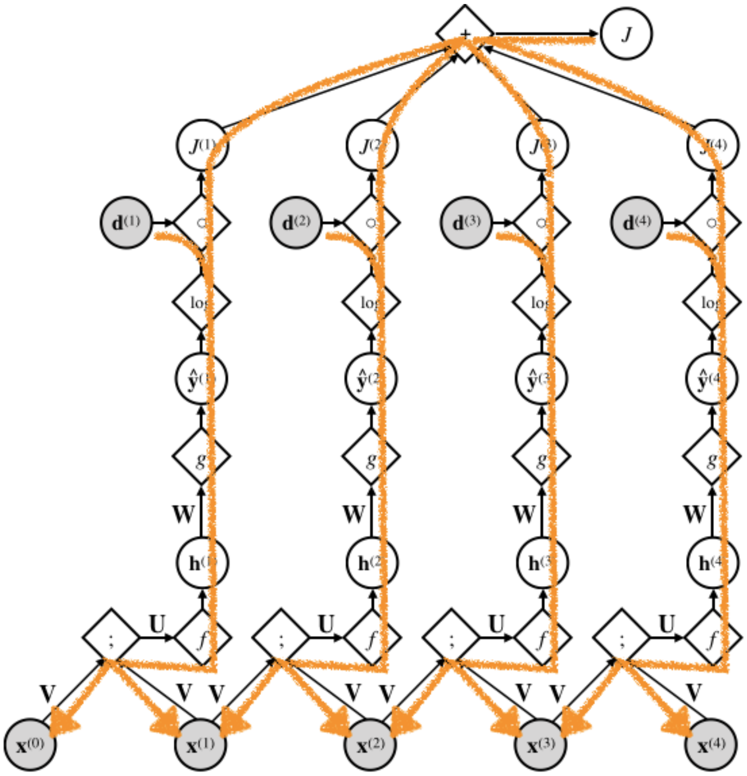
The purpose of this note is to clarify the relationship between backpropagation, backpropagation through time, and **truncated** backpropagation through time, which you are to implement for the coursework.

First, let's talk about backpropagation on a simple feedforward network---in other words, a classifier whose structure is the same for every problem instance. For concreteness, we'll use a feedforward trigram language model, which has almost the same structure as our RNN, except that instead of depending on all history words through a recurrent state, each prediction depends only on the two most recent words. It's illustrated below as a computation graph using the same notation as in your homework where possible. (graphically: shaded circles are observed values; circles are values in general; diamonds are functions; parameter matrices are edge labels, and these edges indicate multiplication by these matrices).

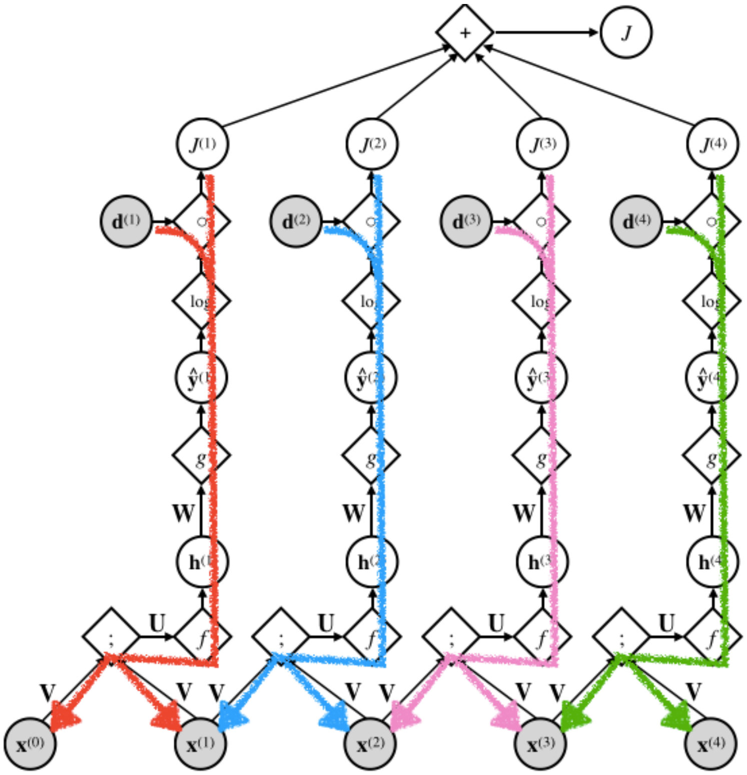


Notice that I've included that overall loss, J, because our goal is to set **U**, **V**, and **W** to minimize J---the entire computation is aimed at minimizing this single function. We do this numerically by taking the partial derivative of J w.r.t. each parameter matrix and then running some variant of a gradient descent algorithm. **The role of backpropagation is to compute the gradient.** For a really clear picture of how that works, I recommend reading the first few sections of Justin Domke's excellent notes on automatic differentiation: https://people.cs.umass.edu/~domke/courses/sml2011/08autodiff_nnets.pdf

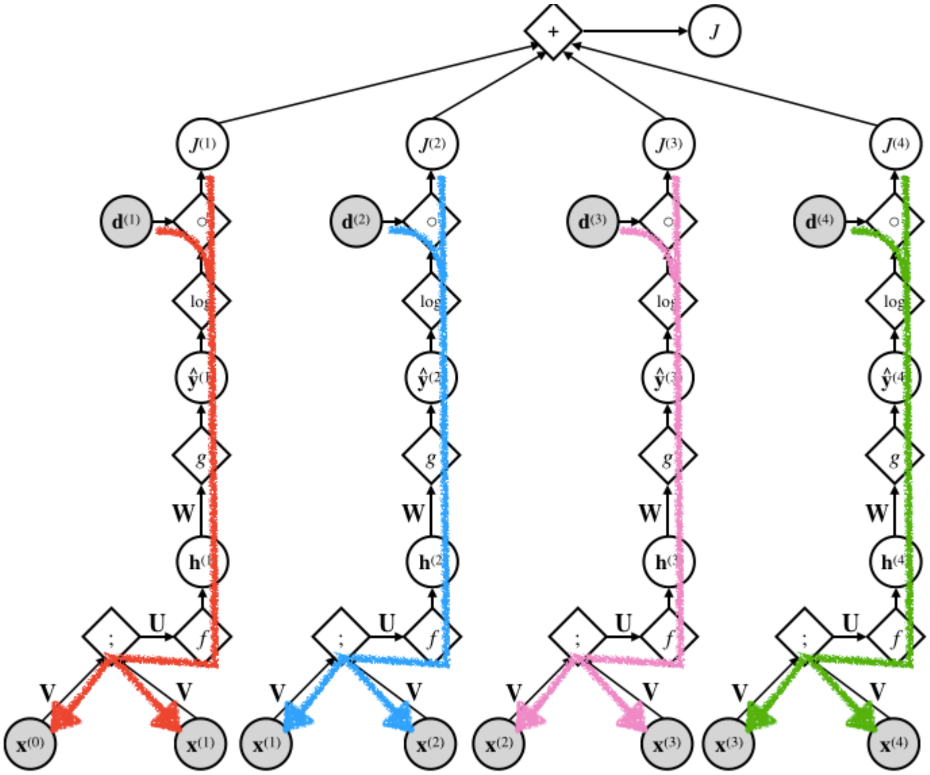
In this case, backpropagation from J (the output) recursively computes partial derivatives in reverse topological order on the computation graph. The partial derivative from backpropagation will also be affected by the **d** nodes. So, its structure looks like this:



One thing to notice here is the structure of the loss J: it's simply the sum of the losses incurred at each individual time step. Since the derivative of a sum of functions is simply the sum of the derivatives of those functions, that means that we can break the computation into four individual pieces. That looks like this:



Just so there's no confusion, notice that the individual losses share absolutely no substructure other than observed variables. If it helps to make that clear, you can also think of the computation like this:

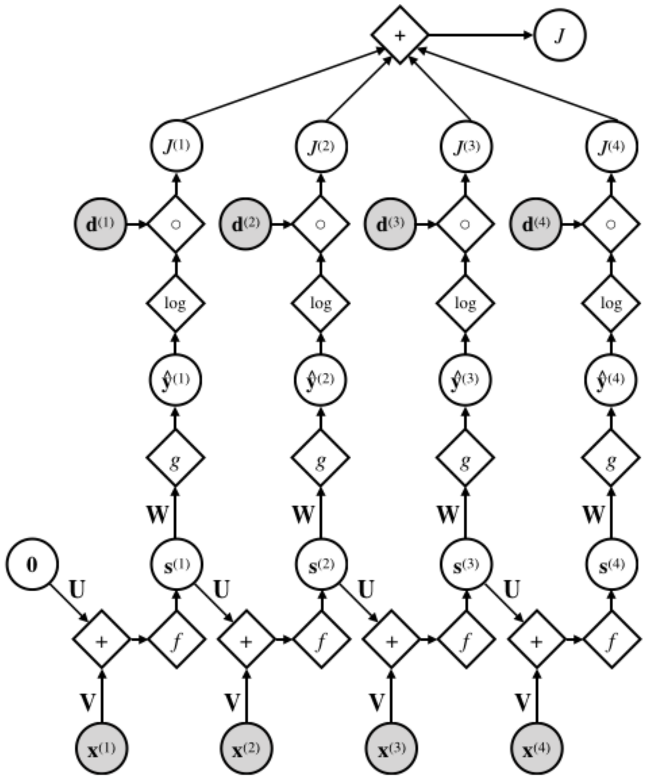


A couple of things are worth pointing out here:

* Your goal is still to compute partial derivatives w.r.t. J. But you can do that by computing partial derivatives w.r.t. the $J^{(i)}$ variables and then summing up as you go.

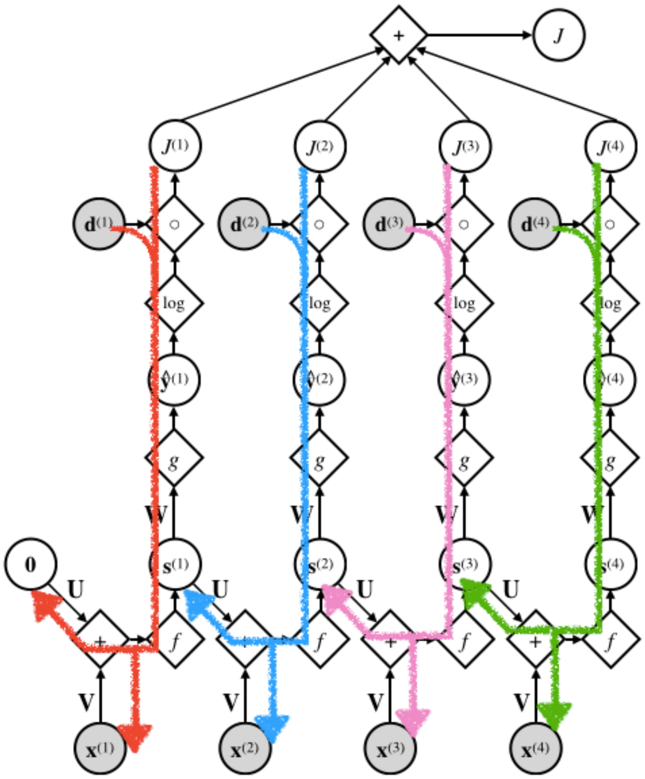
* The structure of these individual computations is always identical. That means that we can construct a small computation subgraph once and run backpropagation on it many times, simply replacing the observed values for each training instance. That makes it easy to implement and train the feedforward network.

The idea in Q1c is to simply port this simple computation to a RNN-LM. First, let's take a look at the structure of that model, using the same notation as in the coursework.

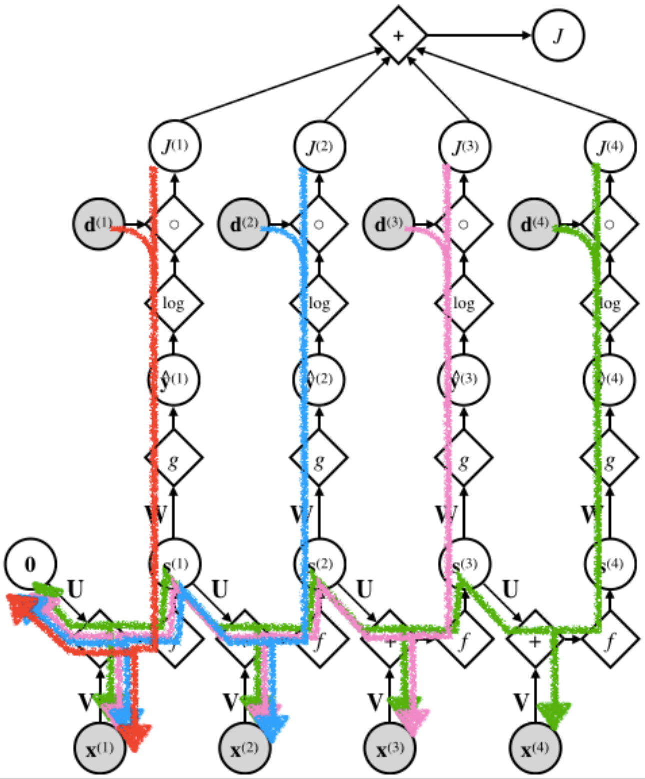


Notice in particular that the individual loss functions share substructure, specifically the **s** nodes. And this means that the indvidual loss functions are now **dynamic**: the function underlying a particular loss depends on the shape of the input. That is, the computation leading to $J^{(1)}$ has a different shape than the computation leading to $J^{(4)}$. A correct implementation of backpropagation would need to account for this, which should be quite clear from Domke's notes above.

Despite this, Q1c just asks you to implement backpropagation on a fixed subgraph that has a common shape for each of the indvidual losses. That is, it asks you to do this:



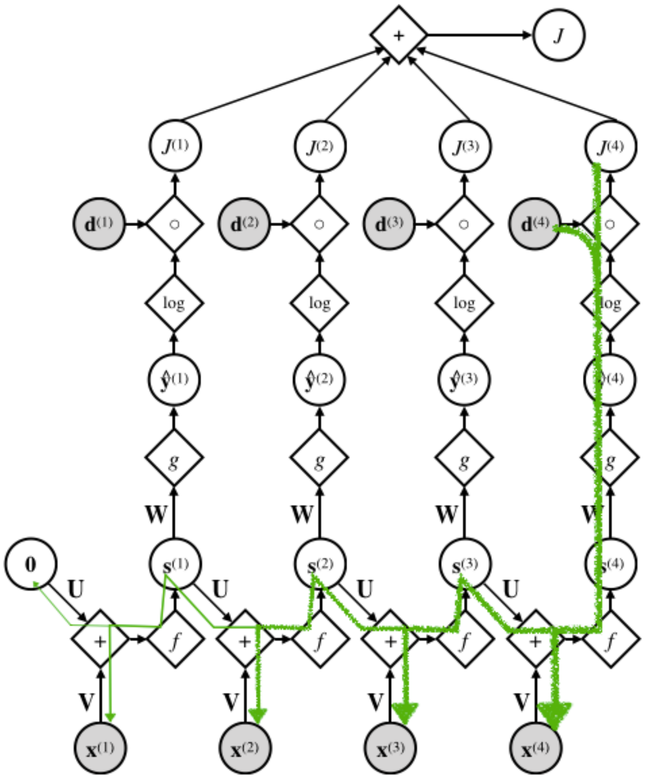
Let's stop and look at a few things about this. First, this is not correctly computing the partial derivative: the loss at t=4 does in fact depend on the input at t={1,...,4}, and we haven't backpropagated that far. So what we're getting here is an approximation. In fact, to compute the correct partial derivatives, we would need to do this:



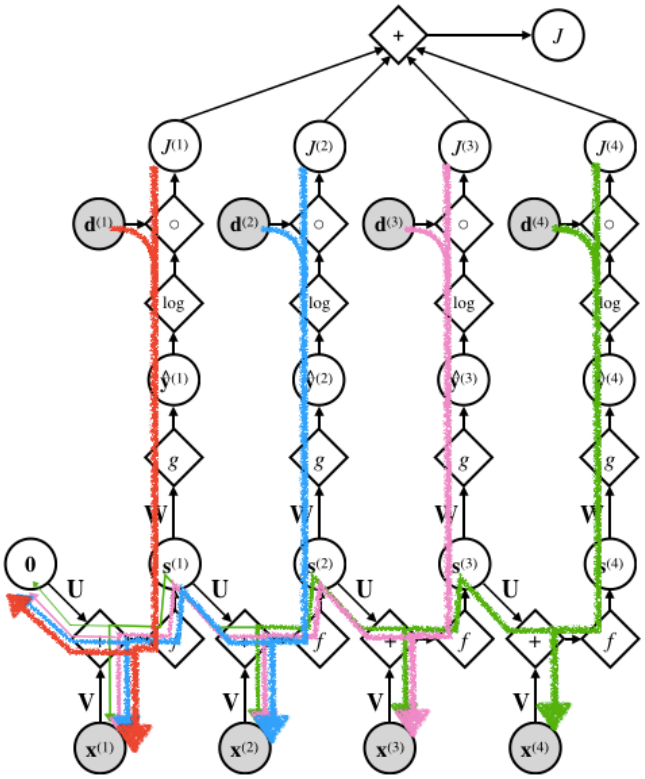
The structure illustrated above is what's often called **backpropagation through time**. But just so we're clear what that means: **it's just backpropagation** on a dynamic computation graph. It's this dynamic computation graph that's different, not the computation of derivatives. In this sense, "backpropgation" and "backpropagation through time" are really the same algorithm, and if any algorithm is really an outlier, it's the version that we ask you to compute in Q1c. That algorithm is properly called **truncated backpropagation through tim**e.

[Two minor parenthetical notes that you can safely skip:
1) There are are a bunch of different variants on truncated BPTT, but they all use roughly the same idea.
2) The structure above implies a quadratic algorithm, which might worry you. But this is just for illustration. You can in fact do this in linear time by propagating the gradients in strict reverse topological order---simply accumulate the partial gradients from each parent node, and only backpropagate once you have all of them. Memory use does grow linearly with the sequence though. See Domke's notes for more intuition about this.]

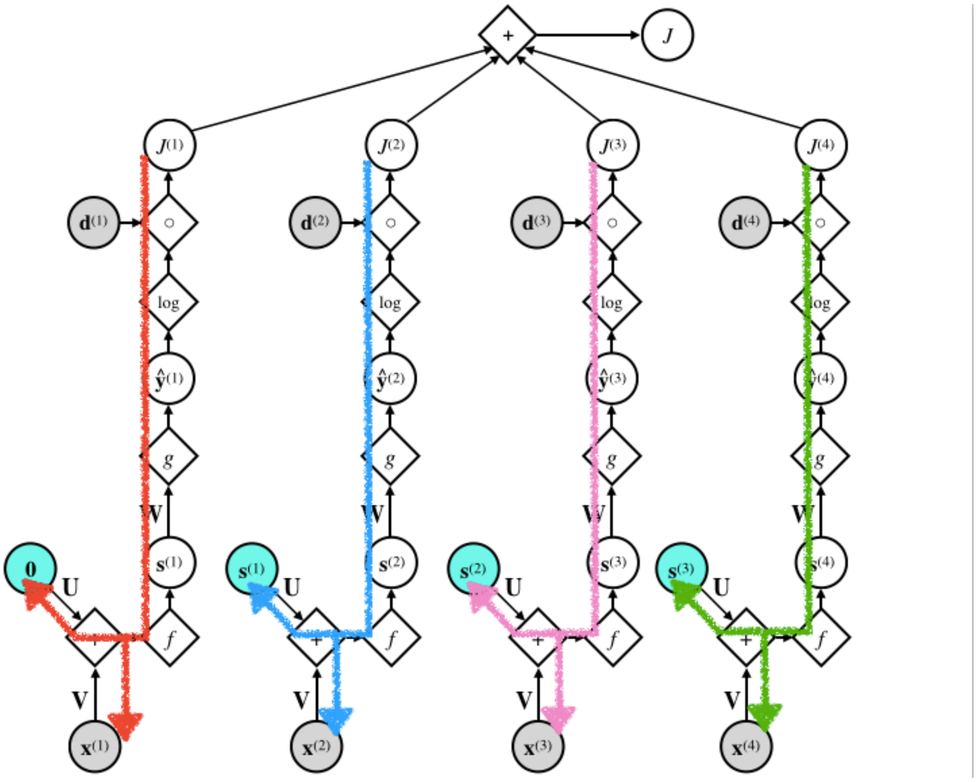
Of course, this isn't the full story. As we discussed in class, the partial derivatives from each individual loss diminish over time: the **vanishing gradient problem** is w.r.t. an indvidual loss.



So, the full BPTT computation behaves more like this:



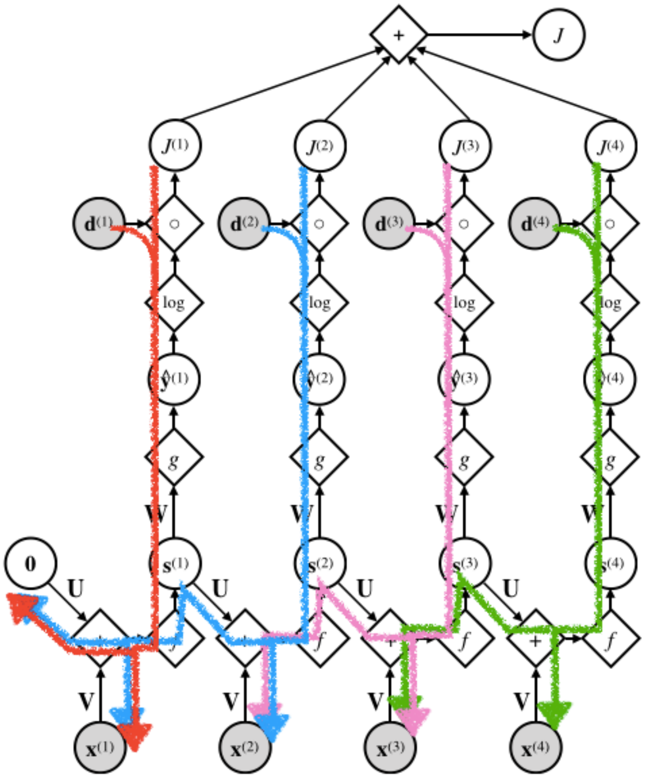
From this perspective, truncated BPTT almost like a reasonable approximation... except that now it might seem as if it has no way to capture long-distance dependencies, which is the whole point of using an RNN! But this is not quite true, and to see why, let's look at the individual computations in truncated BPTT, much as we did for the feedforward case above:



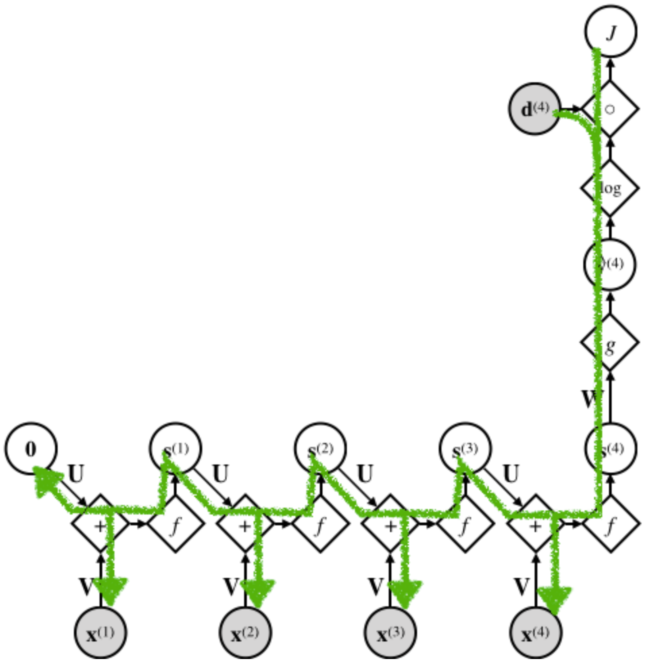
I've highlighted the RNN states here to make a point: allthough TBPTT treats these as if they were observed input variables, they aren't! They are, in fact, the result of first running a forward pass of the RNN. When you run TBTT on this computation, updates to the **U** matrix do reflect dynamic properties of the RNN that depend on the full history. It is true, however, that we cannot backpropagate signal explicitly between words with long-distance dependencies. To have any hope of capturing these, we need for **s** to learn to remember information that enters the network via matrix **V**, and for **V** to be well-trained based on local dependencies. If most dependencies are local, then this might happen.

Of course, whether that really happens is an empirical question, and it depends on the properties of your data and your problem. It is not magic and you should simply assume that it will happen.

Q1d is **also** asking you to implement truncated BPTT, but simply unrolling the network a bit further (this observation may give you a clue about how to simplify your code). For example, if you look back one step, you get this:



Now let's look at what we're asking you to do in Q3. In a single picture, we're just asking you to implement the **truncated** BPTT version of this:



Implementation-wise, if you suspect that this is very similar to your other solutions, you are probably on the right track. There are a lot of interesting questions that this new task raises, so we've given you some space to explore them in Q4.