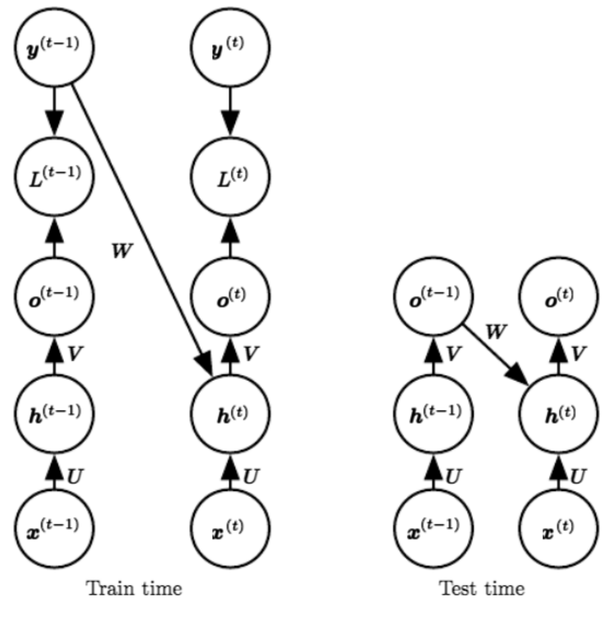
sequence modeling & RNNs

- now its all about sequence processing
- parameter sharing is again a key principle, allowing us to apply it to sequences of variable length.
- In addition, parameter sharing enables generalization and statistical efficiency because a standard net would have to learn the rules of the language separately at each position in the sentence.
- each member of output is a function of previous member of output, with the same parameters applied.
- The reapplication is called unfolding of the computational graph.
- some general variants of RNNs:
  - produce an output at each time step and have recurrent connections between hidden units.
  - produce an output at each time step and have recurrent connections from the output at one time step to hidden units at next time step.
  - recurrent connections between hidden units that read an entire sequence and then produce a single output
- In RNNs, we don't usually use ReLUs → we use a $tanh$ function to act like a binary gate, for reasons that will become clear in a bit.
- the total loss for a given sequence of x values paired with a sequence of y values would then just be the sum of the losses over all time steps.
- Back prop thru time (that happens in RNNs) is really just regular backprop on the unfolded computational graph.
- Because these graphs can get really long, we want to have feedback at each layer level instead of just at the end and then propagated back. this leads to an idea called teacher forcing:



- the kinds of inputs that the network sees during inference might be different from what it saw during training (because it was fed the labels instead of prior time step's output). to overcome this, we can put the label thru a couple cycles before feeding it to a hidden unit.

- the number of params in a RNN may be adjusted to control model capacity but is not forced to scale with sequence length (O(1) as a function of sequence length)
- The assumption is that the conditional probability distribution over the variables at time t + 1 given the variables at time t is stationary.
- RNNs conditioned on context:

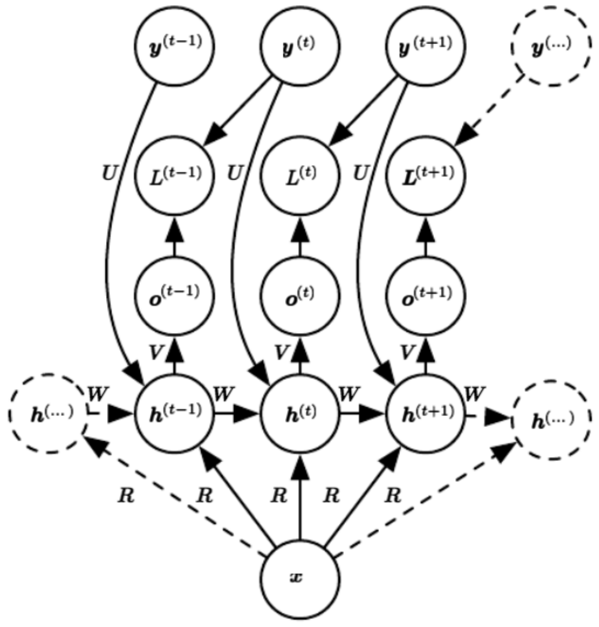


Figure 10.9: An RNN that maps a fixed-length vector $\boldsymbol{x}$ into a distribution over sequences $\mathbf{Y}$. This RNN is appropriate for tasks such as image captioning, where a single image is used as input to a model that then produces a sequence of words describing the image. Each element $\boldsymbol{y}^{(t)}$ of the observed output sequence serves both as input (for the current time step) and, during training, as target (for the previous time step).

- Bi-directional RNNs: we might want to produce a prediction that depends on the whole input sequence, like in speech recognition. This allows the output units to compute a representation that depends on both the past and the future but is most sensitive to the input values around time t, without having to specify a fixed-size window around t.
- how can we design an RNN to map variable sized sequences to variable sized sequences?
- Encoder-Decoder sequence-to-sequence Architectures:
  - (1) An encoder or reader or input RNN processes the input sequence. The encoder emits the context C, usually as a simple function of its final hidden state.
  - (2) A decoder or writer or output RNN is conditioned on that fixed-length vector to generate the output sequence
- the context is a like an encoded summary of the input.
- the downside of above is what if context doesn't have enough dimensions to capture the input properly.
- Recursive neural nets have not graph unfolding → they are just straightforward graphs. they reduce the overall depth of the graph a lot (from τ to $log\ \tau$):

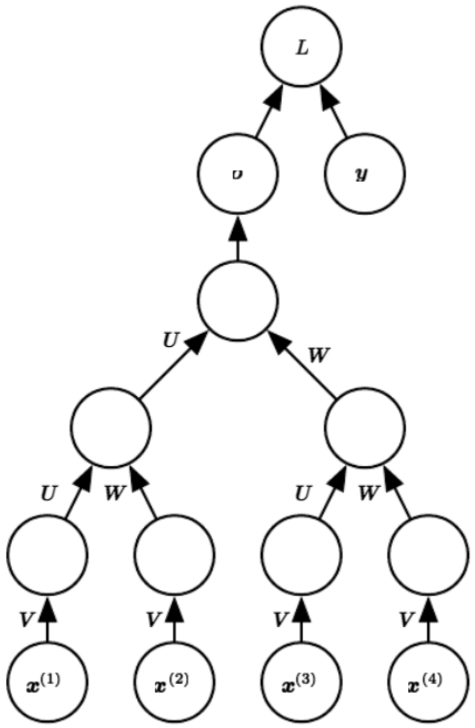


Figure 10.14: A recursive network has a computational graph that generalizes that of the recurrent network from a chain to a tree. A variable-size sequence $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$ can be mapped to a fixed-size representation (the output $\boldsymbol{o}$), with a fixed set of parameters (the weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$). The figure illustrates a supervised learning case in which some target $\boldsymbol{y}$ is provided that is associated with the whole sequence.

- In RNNs, vanishing and exploding gradients is particularly a challenge because of long term dependencies.
- We could add skip connections and do exponential decay of the output of the hidden units over time (leaky units). But the real gold here is LSTMs:

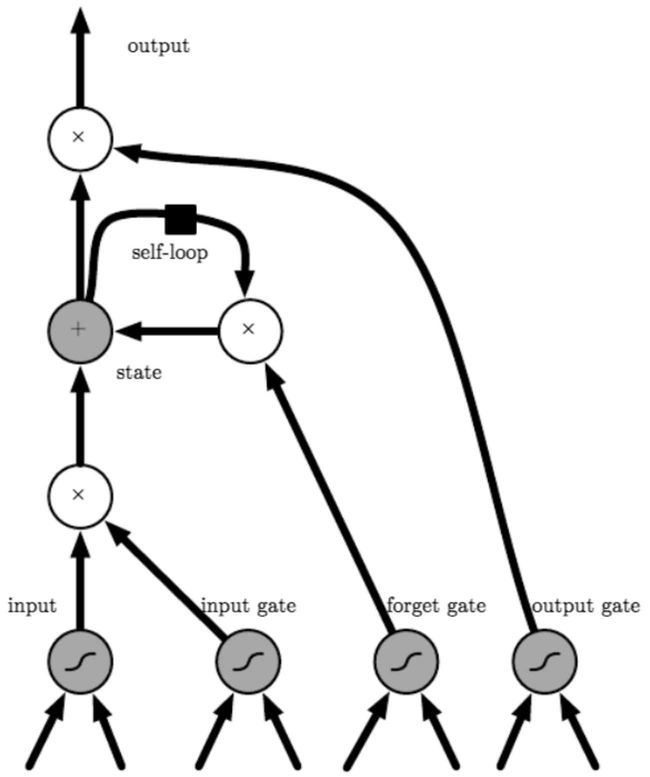


Figure 10.16: Block diagram of the LSTM recurrent network "cell." Cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks. An input feature is computed with a regular artificial neuron unit. Its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. The output of the cell can be shut off by the output gate. All the gating units have a sigmoid nonlinearity, while the input unit can have any squashing nonlinearity. The state unit can also be used as an extra input to the gating units. The black square indicates a delay of a single time step.

- In the above, output is a $tanh$ unit while the gates are sigmoid units. The inputs come with weights $\mathbf{W}$, as usual.
- GRU (gated recurrent unit) is similar except that a single gating unit controls the forgetting factor and the decision to update the state unit.
- LSTMs and GRUs are another example of a model that is easier to optimize rather than a much better optimization algorithm.
- Gradient clipping and other optimization techniques become particularly important with RNNs to get them to behave themselves and work well.
- Neural nets are quite good at implicit knowledge but struggle with explicit facts. What if we gave a neural net explicit memory? Enter neural turing machines.
- To resolve this difficulty, Weston et al. (2014) introduced memory networks that include a set of memory cells that can be accessed via an addressing mechanism. Memory networks originally required a supervision signal instructing them how to use their memory cells. Graves et al. (2014b) introduced the neural Turing machine, which is able to learn to read from and write arbitrary content to memory cells without explicit supervision about which actions to undertake, and allowed end-to-end training without this supervision signal, via the use of a content-based soft attention mechanism
- Each memory cell can be thought of as an extension of the memory cells in LSTMs and GRUs. The difference is that the network outputs an internal state that chooses which cell to read from or write to,

just as memory accesses in a digital computer read from or write to a specific address.

- It is difficult to optimize functions that produce exact integer addresses. To alleviate this problem, NTMs actually read to or write from many memory cells simultaneously. To read, they take a weighted average of many cells. To write, they modify multiple cells by different amounts. The coefficients for these operations are chosen to be focused on a small number of cells, for example, by producing them via a softmax function.
- Explicit memory seems to allow models to learn tasks that ordinary RNNs or LSTM RNNs cannot learn. One reason for this advantage may be that information and gradients can be propagated (forward in time or backward in time, respectively) for very long durations.