Recurrent Neural Networks

By: Collin Brown

Sequential Models and Computational Graphs

- Thinking about neural network architectures that are designed to process sequence (time-series) data.
- Want to process a sequence of inputs $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(\tau)}$. A "training example" that the RNN operates on is therefore a sequence of these $\mathbf{x}^{(t)}$ vectors, $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(\tau)}\}$, where t is the time step index that ranges from $1 ... \tau$.
- In practice, RNNs typically operate on mini-batches of these sequences. Note that each sequence in the mini-batch may have a different sequence length, τ.
- Note that the index t can just refer to the position of an element in a sequence. It does not need to correspond literally to time.
- The RNN architecture can be applied to 2D data as well (e.g. sequences of images).
- We can utilize parameter sharing to generalize across examples of different forms (in the context of RNNs, this means showing sequences of different lengths).

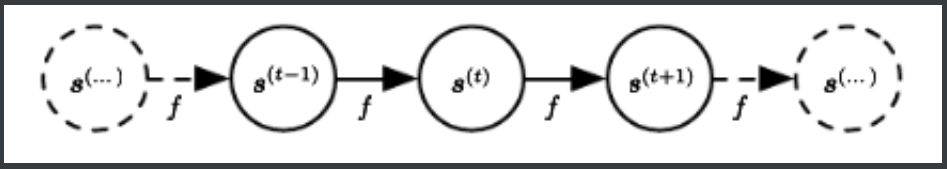
Parameter Sharing: Convolutional vs. Recurrent Networks

Convolutional: We "convolve" a kernel over some region of the input space and map it to a region in the output space. So each member of the output is a function of some neighboring set of inputs.

Recurrent: The previous outputs become arguments for the current outputs. So each member of the output is produced by using the same update rule applied to previous outputs.

Dynamical Systems as Unfolded Computational Graphs

- Consider a simple dynamical system, where the current state $s^{(t)}$ is determined as a function of (i) the previous state $s^{(t-1)}$ and (ii) some parameter set, θ. That is, $s^{(t)} = f(s^{(t-1)}, \theta)$.



- The above figure (Figure 10.1 from Deep Learning Book, Chapter 10) illustrates the dynamical system governed by f as an unfolded computational graph.

- As long as the sequence is finite (i.e. $\tau < \infty$), a cyclic graph representing a recurrent equation can be unfolded to yield a traditional directed acyclic graph (DAG), such as Figure 10.1 above.
- This is equivalent to showing that we can unfold a recursive equation into a big function composition.

$$s^{(\tau)} = f(s^{(\tau-1)}, \theta)$$

$$s^{(\tau)} = f(f(s^{(\tau-2)}, \theta), \theta)$$

$$s^{(\tau)} = f(f(f(\dots f(f(s^{(0)}, \theta), \theta), \dots, \theta), \theta), \theta)$$

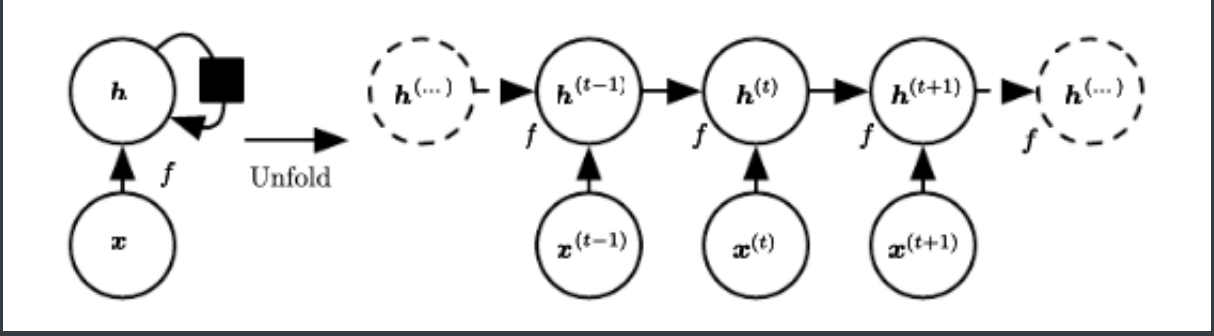
- We can expand the above idea by allowing $s^{(t)}$ to depend not only on its previous state $s^{(t-1)}$ and a shared parameter set θ, but also on some external signal $x^{(t)}$ received in each period t.

$$s^{(t)} = f(s^{(t-1)}, x^{(t)}, \theta)$$

- This allows us to think about situations where we want the current state to account for information received throughout the whole past sequence as well as the information received in the current step in the sequence.
- In the context of recurrent neural networks, we want to use an equation similar to the above to define the network's hidden units. So, we rewrite as follows:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}, \theta)$$

- The above equation can be drawn as a computational graph in two different ways. First (left), we can draw it as a circuit diagram with a black square used to indicate a one-step time delay (e.g. from time t to $t+1$). Second (right), we can unfold it and draw it as a directed acyclic graph. The below figure is Figure 10.2 from Deep Learning Book, Chapter 10.

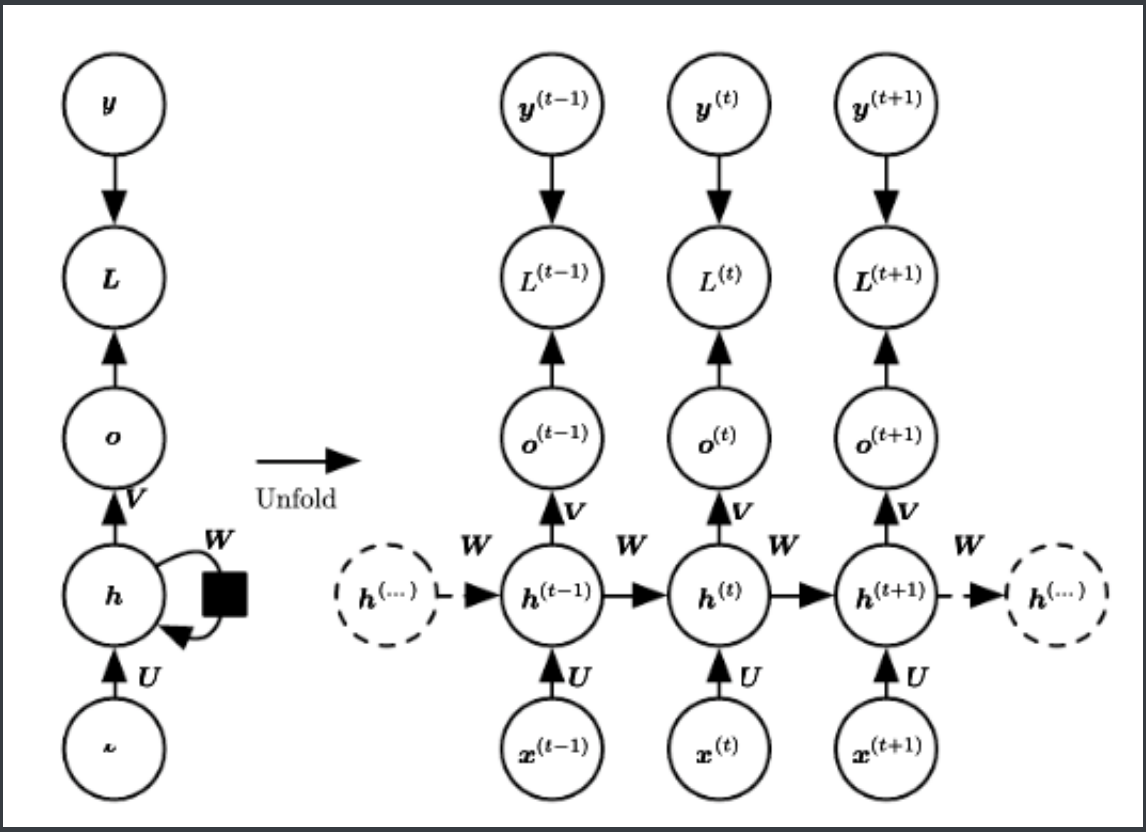


- Note that the model always has the same input size, because it is specified in terms of transitions between states, which is independent of the number of variables in the sequence.
- Note also that it is possible to use the same transition function f with the same parameters at each time step.
- These two advantages allow a single model, f, to be learned that generalizes to all sequence

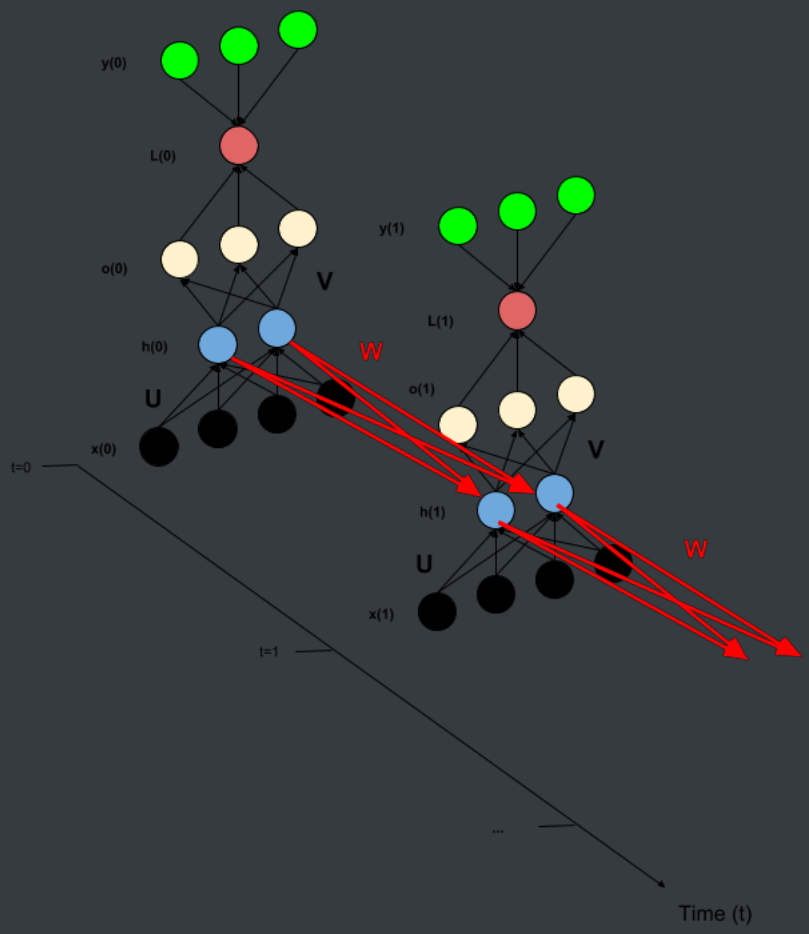
lengths. Also, because of the parameter sharing, significantly fewer parameters need to be estimated, which means that significantly fewer training examples are required to learn f than would be required in the absence of parameter sharing.

Recurrent Neural Networks that Produce an Output at Each Time Step

- The idea behind this architecture is to simply map a sequence of input vectors $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(\tau)}\}$ to a sequence of output vectors $\{\mathbf{o}^{(1)}, \ldots, \mathbf{o}^{(\tau)}\}$.
- A loss function $\mathscr{L}^{(t)}$ measures how far off the predicted value $o^{(t)}$ was from the true value $y^{(t)}$. Note that there is a loss for each t in $1...\tau$.



- The connections between the input layer and the hidden layer are parameterized by $\mathbf{U}$, the hidden-to-hidden recurrent connections are parameterized by $\mathbf{W}$, and the connections between the hidden layer and the output layer are parameterized by $\mathbf{V}$. This architecture is illustrated in the simple example below.



- In the example above, there are simple input vectors each containing 4 elements. Each input vector corresponds to a 3-element output vector. There is a hidden recurrent layer that has 2 elements, a 3 element output layer, and the loss function takes the predicted output and the true output in order to compute the loss associated with that particular time step. Notice that the same parameter tensors $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$ get reused at each time step. Note also that the above diagram does not illustrate the entry of bias vectors.

Forward propagation of this RNN

- The following system of equations defines forward propagation in the above architecture:

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)}$$

$$\mathbf{h}^{(t)} = tanh(\mathbf{a}^{(t)})$$

$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{o}^{(t)})$$

-

Further Reading

Deep Learning Book, Chapter 10

Supervised Sequence Labelling with Recurrent Neural Networks