Sequence Modeling: Recurrent and Recursive Nets

박태준

March 3, 2023

- Deep Learning (Goodfellow et al., 2016) Chapter 10.
- Recurrent neural networks or RNNs (Rumelhart et al., 1986a) are a family of neural networks for processing sequential data.
- Such sharing is particularly important when a specific piece of information can occur at multiple positions within the sequence.
- By comparison, a recurrent neural network shares the same weights across several time steps.
- This chapter extends the idea of a computational graph to include cycles.
    - ▷ These cycles represent the influence of the present value of a variable on its own value at a future time step.
    - ▷ Such computational graphs allow us to define recurrent neural networks.

1 Unfolding Computational Graphs

- **A computational graph** is a way to formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs and loss.
- In this section we explain the idea of **unfolding** a recursive or recurrent computation into a computational graph that has a repetitive structure, typically corresponding to a chain of events.
- Unfolding this graph results in the sharing of parameters across a deep network structure.

  ▷ Example 1) if we unfold for 3 time steps, we obtain

$$\begin{aligned}\mathbf{s}^{(3)} &= f(\mathbf{s}) \\ &= f(f(\mathbf{s}^{(1)};\boldsymbol{\theta});\boldsymbol{\theta})\end{aligned}$$

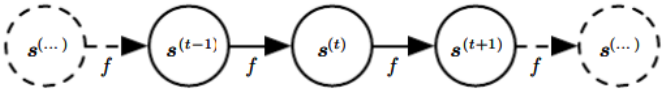

Figure 10.1: The classical dynamical system described by Eq. 10.1, illustrated as an unfolded computational graph. Each node represents the state at some time$t$ and the function f maps the state at t to the state at $t+1$. The same parameters (the same value of $\boldsymbol{\theta}$ used to parametrize f) are used for all time steps.

  ▷ Example 2) Let us consider a dynamical system driven by an external signal $\mathbf{x}^{(t)}$,

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

  where we see that the state now contains information about the whole past sequence.

- **Recurrent neural networks** can be built in many different ways: Any function involving recurrence can be considered a recurrent neural network.
- Using the variable $\mathbf{h}$ to represent the state:

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

- Typical RNNs will add extra architectural features such as output layers that read information out of the state $\mathbf{h}$ to make prediction.

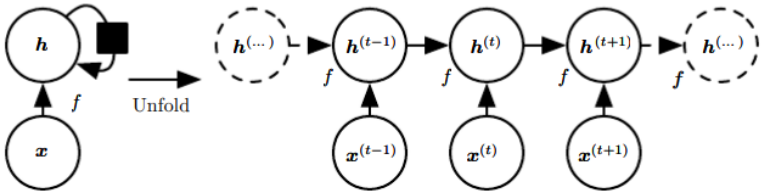

Figure 10.2: A recurrent network with no outputs. This recurrent network just processes information from the input $\boldsymbol{x}$ by incorporating it into the state $\boldsymbol{h}$ that is passed forward through time. *(Left)* Circuit diagram. The black square indicates a delay of 1 time step. *(Right)* The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance.

- It maps an arbitrary length sequence

$$(\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t-2)}, \cdots, \mathbf{x}^{(2)}, \mathbf{x}^{(1)})$$

  to a fixed length vector $\mathbf{h}^{(t)}$.

- The unfolded graph now has a size that depends on the sequence length.

- We can represent the unfolded recurrence after t steps with a function $g^{(t)}$:

$$\begin{aligned}\mathbf{h}^{(t)} &= g^{(t)}(\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t-2)}, \cdots, \mathbf{x}^{(2)}, \mathbf{x}^{(1)}) \\ &= f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \boldsymbol{\theta})\end{aligned}$$

- the unfolded recurrent structure allows us to factorize $g^{(t)}$ into repeated application of a function f.

- The unfolding process thus introduces two major advantages:

  1. Regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state to another state, rather than specified in terms of a variable-length history of states.
  2. It is possible to use the same transition function f with the same parameters at every time step.

2 Recurrent Neural Networks

- Armed with the graph unrolling and parameter sharing ideas, we can design a wide variety of recurrent neural networks.
- Some examples of **important design patterns for recurrent neural networks** include the following:
  - ▷ Recurrent networks that produce an output at each time step and have recurrent connections between hidden units. (Figure 10.3)
  - ▷ Recurrent networks that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step. (Figure 10.4)
  - ▷ Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output.

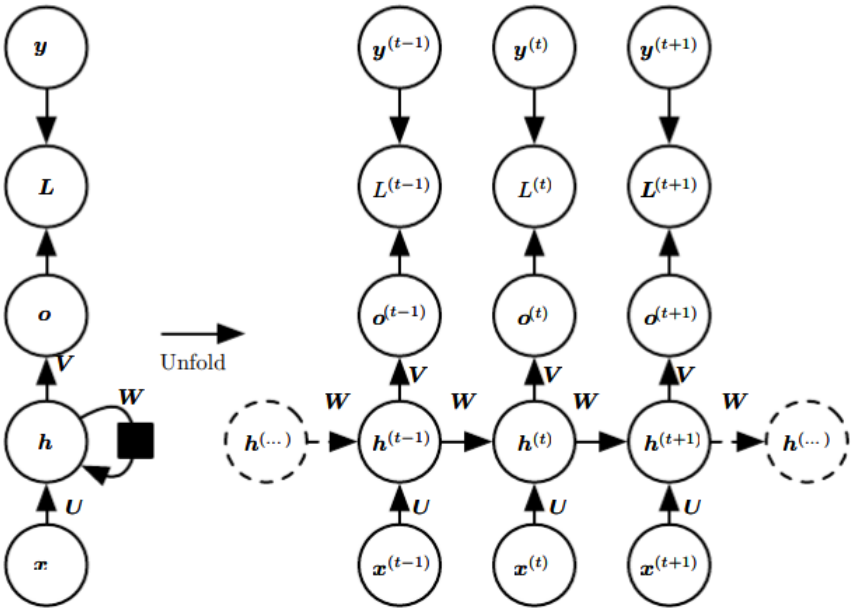

Figure 10.3: The computational graph to compute the training loss of a recurrent network that maps an input sequence of $\boldsymbol{x}$ values to a corresponding sequence of output $\boldsymbol{o}$ values. A loss L measures how far each $\boldsymbol{o}$ is from the corresponding training target $\boldsymbol{y}$. When using softmax outputs, we assume $\boldsymbol{o}$ is the unnormalized log probabilities. The loss L internally computes $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{o})$ and compares this to the target $\boldsymbol{y}$. The RNN has input to hidden connections parametrized by a weight matrix $\boldsymbol{U}$, hidden-to-hidden recurrent connections parametrized by a weight matrix $\boldsymbol{W}$, and hidden-to-output connections parametrized by a weight matrix $\boldsymbol{V}$. Eq. 10.8 defines forward propagation in this model. *(Left)* The RNN and its loss drawn with recurrent connections. *(Right)* The same seen as an time-unfolded computational graph, where each node is now associated with one particular time instance.

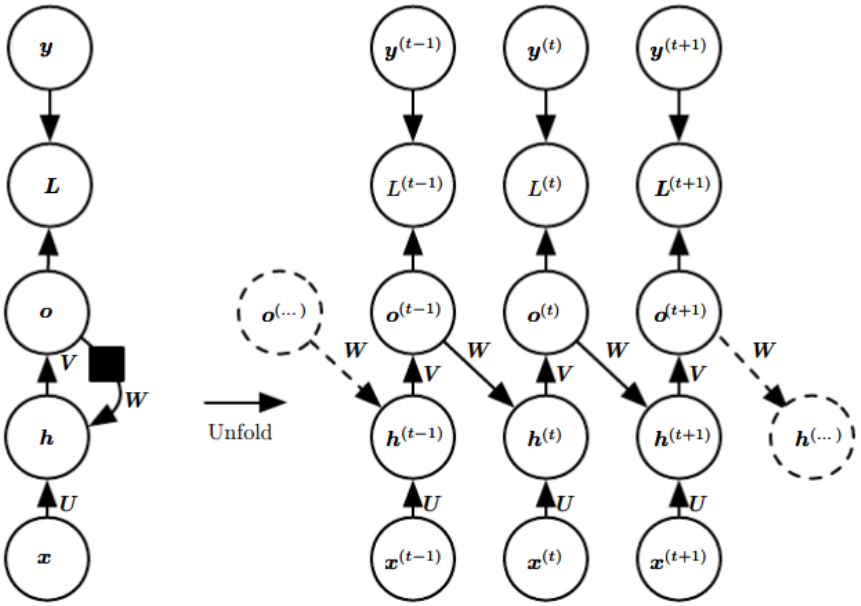

Figure 10.4: An RNN whose only recurrence is the feedback connection from the output to the hidden layer. At each time step t, the input is $\boldsymbol{x}_t$, the hidden layer activations are $\boldsymbol{h}^{(t)}$, the outputs are $\boldsymbol{o}^{(t)}$, the targets are $\boldsymbol{y}^{(t)}$ and the loss is $L^{(t)}$. *(Left)* Circuit diagram. *(Right)* Unfolded computational graph. Such an RNN is less powerful (can express a smaller set of functions) than those in the family represented by Fig. 10.3. The RNN in Fig. 10.3 can choose to put any information it wants about the past into its hidden representation $\boldsymbol{h}$ and transmit $\boldsymbol{h}$ to the future. The RNN in this figure is trained to put a specific output value into $\boldsymbol{o}$, and $\boldsymbol{o}$ is the only information it is allowed to send to the future. There are no direct connections from $\boldsymbol{h}$ going forward. The previous $\boldsymbol{h}$ is connected to the present only indirectly, via the predictions it was used to produce. Unless $\boldsymbol{o}$ is very high-dimensional and rich, it will usually lack important information from the past. This makes the RNN in this figure less powerful, but it may be easier to train because each time step can be trained in isolation from the others, allowing greater parallelization during training, as described in Sec. 10.2.1.

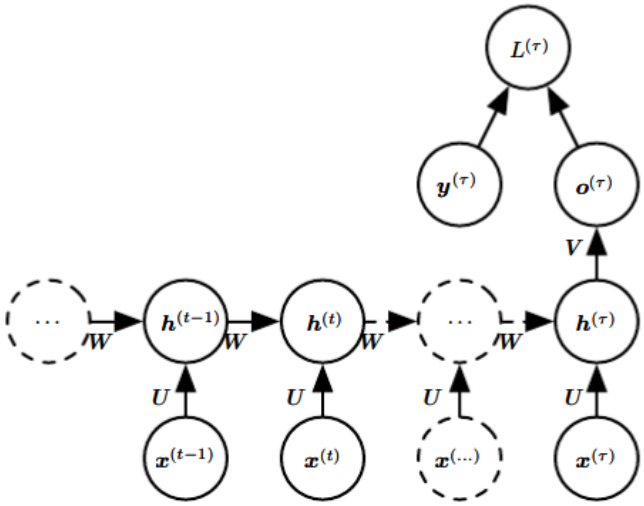

Figure 10.5: Time-unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (as depicted here) or the gradient on the output $\boldsymbol{o}^{(t)}$ can be obtained by back-propagating from further downstream modules.

- The recurrent neural network of Fig. 10.3 and Eq. 10.8 is universal in the sense that any function computable by a Turing machine can be computed by such a recurrent network of a finite size.
- We now develop the forward propagation equations for the RNN.
  - ▷ Here we assume the hyperbolic tangent activation function.
  - ▷ Here we assume that the output is discrete, as if the RNN is used to predict words or characters.
  - ▷ A natural way to represent discrete variables is to regard the output $\mathbf{o}$ as giving the unnormalized log probabilities of each possible value of the discrete variable.
  - ▷ We can then apply the softmax operation as a post-processing step to obtain a vector $\hat{\mathbf{y}}$ of normalized probabilities over the output.
  - ▷ Forward propagation begins with a specification of the initial state $\mathbf{h}^{(0)}$.
  - ▷ Then, for each time step from $t = 1$ to $t = \tau$, we apply the following update equation:

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)}, \tag{10.8}$$

with

$$\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)})$$
$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}$$
$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{o}^{(t)}),$$

where e the parameters are the bias vectors $\mathbf{b}$ and $\mathbf{c}$ along with the weight matrices $\mathbf{U}, \mathbf{V}$ and $\mathbf{W}$, respectively for input-to-hidden, hidden-to-output and hidden-to-hidden connections.

▷ This is an example of a recurrent network that maps an input sequence to an output sequence of the same length.

- The total loss for a given sequence of $\mathbf{x}$ values paired with a sequence of $\mathbf{y}$ values would then be just the sum of the losses over all the time steps.

  ▷ For example, if $L^{(t)}$ is the negative log-likelihood of $y^{(t)}$ given $\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(t)}$, then

$$L\left(\{\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(\tau)}\}, \{\mathbf{y}^{(1)}, \cdots, \mathbf{y}^{(\tau)}\}\right) = \sum_t L^{(t)}$$
$$= -\sum_t \log p_{\text{model}}\left(y^{(t)} | \{\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(t)}\}\right)$$

  where $p_{\text{model}}\left(y^{(t)} | \{\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(t)}\}\right)$ is given by reading the entry for $y^{(t)}$ from the model's output vector $\hat{\mathbf{y}}^{(t)}$.