Intelligent Systems: Reasoning and Recognition

James L. Crowley

MOSIG M1 Winter Semester 2019-2020
Lessons 12 21 March 2020

Recurrent Neural Networks

Outline

Sources

1) Goodfellow, I., Bengio, Y., and Courville, A., Deep learning. MIT press, 2016.
2) Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In Rumelhart, D. E. and McClelland, J. L., editors, Parallel Distributed Processing, volume 1, pages 318–362. MIT Press.

Notation

(Unlike previous lectures, in this lecture we adopt the bold-face matrix-vector notation used in Goodfellow et al 2016.)

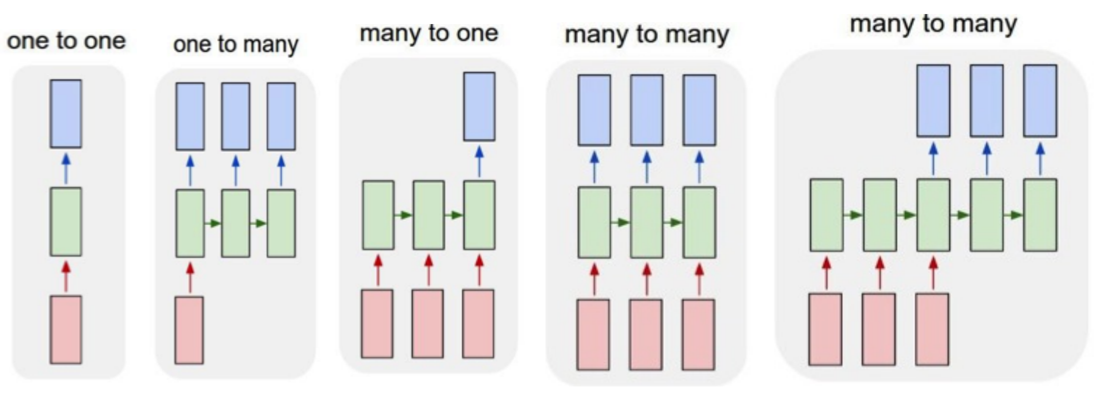
$h^{(t)}$	The hidden recurrent activation vector of the network.
$f(-)$	A process equation that computers $h^{(t+1)}$ from $h^{(t)}$
$\mathbf{x}^{(t)}$	A sequence of τ input vectors. Equivalent to $\{\vec{X}_1,...,\vec{X}_\tau\}$ in earlier lectures.
$o^{(t)}$	The network output. Typically a symbol from a set of symbols.
U	a weight matrix from the input to the hidden unit.
W	a hidden-to-hidden layer recurrent weight matrix
V	a hidden-layer-to-output weight matrix
b	bias vector for hidden units
c	bias vector for the output units.

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are used to discriminate and generate sequences (data that have an intrinsic order relation). Examples of sequences that may be discriminated and generated with RNNs include Speech, Music, Text, and Time Series data. RNNs can be combined with convolutional networks to recognize and generate video sequences of actions. RNNs can be very useful for natural language processing including for understanding written text and machine translation.

Recurrent networks are Turing Universal, which means that any function that can be computed by a Turing machine can be computed by a recurrent network.

Recurrent networks can be used for many kinds of tasks.



History

In the early days of neural networks (1980's), a frequent criticism was that networks have no memory, other than the parameter learning. It was said that because networks did not maintain temporal state, they could not be suitable for tasks involving temporal or spatial sequences.

In the late 1980s, Rumelhart addressed this question by building on a class of completely connected networks proposed by Hopfield, leading to the idea of "unfolding" the network over time. Such networks are now called recurrent neural networks.

A recurrent neural network (RNN) is a neural network where connections between nodes form a directed graph along a temporal sequence. This allows it to exhibit temporal dynamic behavior. RNNs can use internal state (memory) to process variable length sequences of inputs. This makes them applicable to tasks such as un-segmented, connected handwriting recognition or speech recognition.

Finite vs Infinite impulse networks

The term “recurrent neural network” refers to two broad classes of networks finite impulse and infinite impulse. Both classes exhibit temporal dynamic behavior.

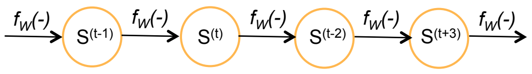
Finite Impulse: A finite impulse recurrent network is a directed acyclic graph that can be unrolled and replaced with a strictly feed-forward neural network. The temporal dynamics are similar to a Finite Impulse Response (FIR) digital filter. In digital signal processing, FIR filters are known to be easy to design, stable, but limited in response.

Infinite impulse: Infinite impulse recurrent network is a directed cyclic graph that cannot be unrolled because of internal feedback. These have similar temporal dynamics to an Infinite Impulse Response digital filter (IIR). In digital signal processing, IIR filters are known to be difficult to design, unstable, but very powerful and efficient. The classic Infinite Inpulse Recurrent network is the LSTM (Long-Short-Term Memory) architecture.

Both finite impulse and infinite impulse recurrent networks can have additional states, and storage can be under direct control by the network. The storage can also be replaced by another network or graph. Such controlled states are referred to as gated state or gated memory, and are a key part of long short-term memory networks (LSTMs) and gated recurrent units.

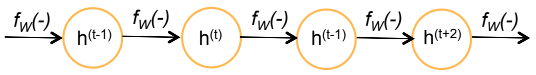
Finite Impulse Recurrent Networks

The classic model for dynamic process is a function, $f(-)$, that predicts the state, s(t) of a system at time *t*, from the state at time *t-1*, using parameters $\vec{w}$. Such as process is known as a "markov" process.



$$s^{(t)} = f_{\vec{w}}(s^{(t-1)})$$

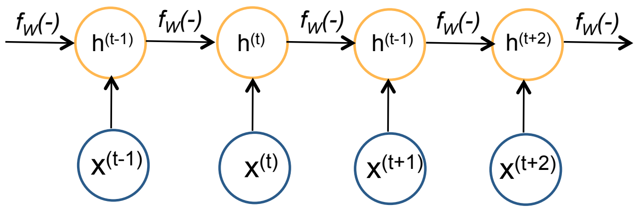
In the case of a recurrent network, the "state" is the activation (or vector of activations) of one or more "hidden" units. These are generally represented with the variable $h^{(t)}$



$$h^{(t)} = f_{\vec{w}}(h^{(t-1)})$$

The time variable is traditionally represented with a superscript, to keep it apart from the unit indices at each level.

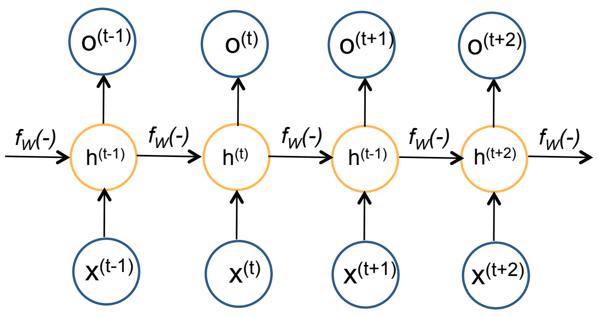
We can model the effects of an external input, $x^{(t)}$ by adding an additional term, $x^{(t)}$ to the temporal transition function.



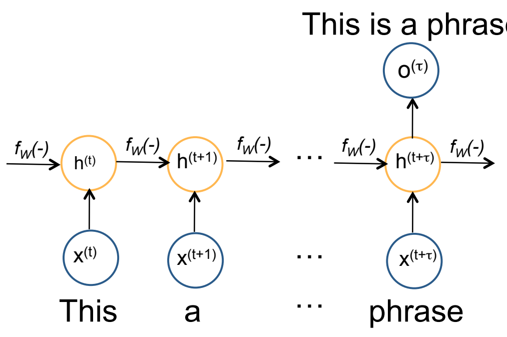
$$h^{(t)} = f_{\vec{w}}(h^{(t-1)}, x^{(t)})$$

The temporal duration of the network is typically represented the the variable τ, so that the network is said to operate on a temporal sequence $x^{(t)}$ from t=1 to τ.

Normally, the network generates an output represented by an output variable, $o^{(t)}$.

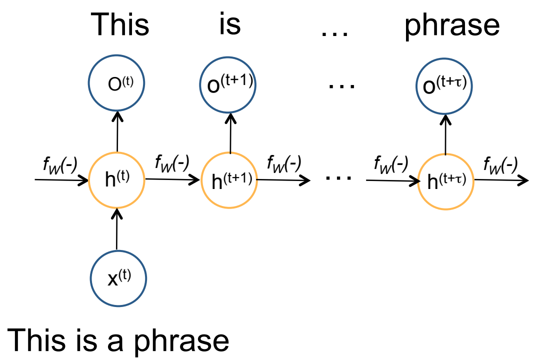


For example, in a many-to-one network, the network would produce a single output after τ time steps. For example, the following network assembles the words "This", "is", "a", and "phrase", into a single output "This is a phrase". In this case, t is the number of words in the phrase, 4.



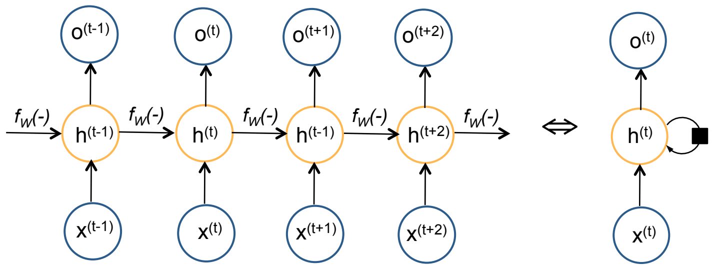
A one-to-many network would produce a sequence of τ outputs from a single input.

For example, a single symbol for "This is a phrase" can be expanded into a sequence of outputs, where $\tau = 4$.

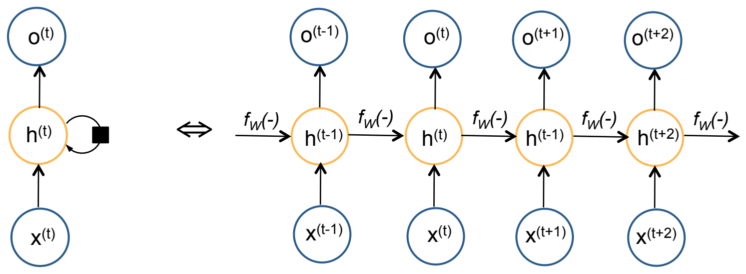


Folding and Unfolding

Recurrent networks are classically "folded" into a recurrent structure:



Where the black square represents a time delay of 1 time unit. The recurrent structure can be unfolded to see the network as a 2-D structure.



Forward Propagation Equations

Networks parameters include:

U a weight matrix,
W a hidden-to-hidden recurrent weight matrix
V a hidden-to-output weight matrix
b bias vector for hidden units
c bias vector for the output units.

The classic forward propagation equations are:

$$\boldsymbol{z}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}x^{(t)}$$
$$\boldsymbol{h}^{(t)} = f(\boldsymbol{z}^{(t))}) = f(\boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}x^{(t)})$$
$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\,\boldsymbol{h}^{(t)}$$

where $\boldsymbol{o}^{(t)}$, $\boldsymbol{h}^{(t),}\, \boldsymbol{z}^{()t,}\, \boldsymbol{b}, \boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}$ are all vectors and ***U**, **V*** and ***W*** are matrices and ***b*** and ***c*** are bias vectors. Such networks typically use a hyperbolic tangent activation function. The equations should be read as expressing summations over as a set of units at the same level. For example for the $j=1$ to N units of $\boldsymbol{h}^{(t)}$ and $\boldsymbol{h}^{(t+1)}$:

$$\boldsymbol{z}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}x^{(t)} \quad \Leftrightarrow \quad z_j^{(t)} = b_j + \sum_{i=1}^{N} W_{ij}^{(t)} h_i^{(t-1)} + \sum_{i=1}^{N} U_{ij}^{(t)} x_i^{(t)}$$

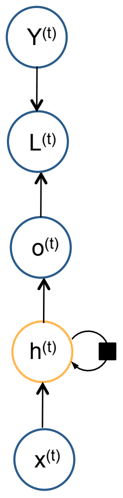
Classically recurrent networks are used to generate symbolic data such as words or characters. In this case, the output vector, $\boldsymbol{o}^{(t)}$ can be seen as an un-normalized log probability of each possible value of the discrete variable. Softmax can then be used to obtain the vector $\hat{y}^{(t)}$ of normalized probabilities for the output.

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

Forward propagation starts from an initial state $\boldsymbol{h}^{(1)}$, and then computes the recurrent states from $t = 1$ to $t = \tau$.

Training

As with classic networks, a recurrent network is trained to minimize the Loss between a target vector $\boldsymbol{Y}^{(t)}$, and an output vector $\boldsymbol{o}^{(t)}$ generated from an input vector $\boldsymbol{X}^{(t)}$. This is represented as $L^{(t)}$:



Where the Loss $L^{(t)}$ measures how far $o^{(t)}$ is from the target $Y^{(t)}$. The loss is internally computed using softmax. $\hat{y}^{(t)} = \mathrm{softmax}(\mathbf{o}^{(t)})$

The total loss L is computed for an input sequence $\left\{x^{(1)},\ldots,x^{(\tau)}\right\}$ and the resulting output sequence $\left\{y^{(1)},\ldots,y^{(\tau)}\right\}$

$$L\left(\left\{\vec{X}^{(1)},\ldots,\vec{X}^{(\tau)}\right\},\left\{y^{(1)},\ldots,y^{(\tau)}\right\}\right)=\sum_{t=1}^{\tau}L^{(t)}=-\sum_{t=1}^{\tau}Log\left(p\left(y^{(t)}\mid\left\{\vec{X}^{(1)},\ldots,\vec{X}^{(\tau)}\right\}\right)\right)$$

Computing the gradient of this loss function with respect to the parameters is an expensive operation.

The gradient computation involves performing a forward propagation pass moving left to right through the unfolded graph of τ units, followed by a backward propagation pass moving right to left through the graph.

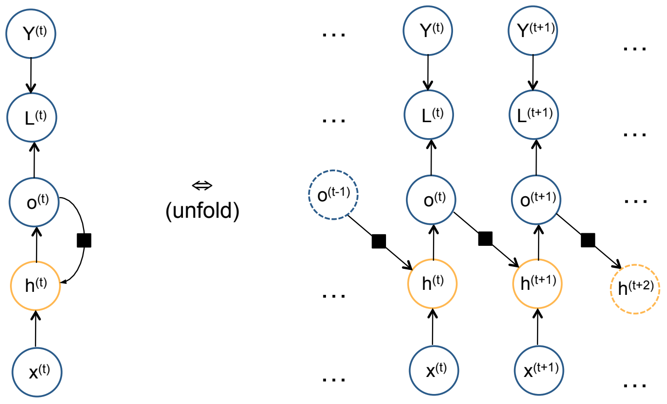
The algorithmic complexity is $O(\tau)$ and cannot be reduced by parallelization because the forward propagation graph is inherently sequential. Each time step may only be computed after the previous one. States computed in the forward pass must be stored until they are reused during the backward pass, so the memory cost is also $O(\tau)$.
(O(-) is the order-of operator)

The back-propagation algorithm applied to the unfolded graph with O(τ) cost is called back-propagation through time or BPTT.

In summary: Computing the gradient of this loss function is expensive because

1) The forward propagation followed by backward propagation operates on all τ samples in parallel.
2) Run-time cost is O(τ) and can not be implemented in parallel.
3) Memory cost is also O(τ).
4) Back-propagation must be applied to the entire unfolded graph. This is called "Back Propagation Through Time" (BPTT).

To reduce the cost of training we can use a network where the recurrence relation is from output to hidden.



The equations for the above network are

$$\boldsymbol{z}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{o}^{(t-1)} + \boldsymbol{U}x^{(t)}$$
$$\boldsymbol{h}^{(t)} = tanh(\boldsymbol{z}^{(t)}) = tanh(\boldsymbol{b} + \boldsymbol{W}\boldsymbol{o}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)})$$
$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\, h^{(t)}$$
$$\hat{y}^{(t)} = \text{softmax}(\mathbf{o}^{(t)})$$

Such a network is less powerful, then the general network described above, but easier to train because each time step can be trained in isolation of the others, allowing for greater parallelization during training.

Computing the Gradient of a Recurrent Network

The gradient of the loss function for a recurrent network is computed with the same back-propagation algorithm described in previous lectures, applied to the unfolded computational graph.

The following development is copied from Goodfellow et al 2016. We adopt there use of bold face for vectors, so that $\mathbf{x}^{(t)}$ is equivalent to $\vec{X}^{(t)}$ used in earlier lectures.

As before network parameters include:

U	a weight matrix,
W	a hidden-to-hidden recurrent weight matrix
V	a hidden-to-output weight matrix
b	bias vector for hidden units
c	bias vector for the output units.

Nodes indexed by t where $\boldsymbol{o}^{(t)}$, $\boldsymbol{h}^{(t),}$ $\boldsymbol{z}^{()t,}$ $\boldsymbol{b}$, $\boldsymbol{h}^{(t-1)}$, $\boldsymbol{x}^{(t)}$ are all vectors. $L^{(t)}$ is the Loss for each unit of time *t*.

For each neural unit we need to recursively compute the gradient $\vec{\nabla} L$ based on the neural units above it in the graph. The gradient of the loss $\vec{\nabla}_{o^{(t)}} L$ is equivalent to the error term , $\delta^{(out)}$, used in deriving back-propagation in earlier lectures.

We compute the loss for each time step, t, as the negative log-likelihood of the true target $y^{(t)}$ given the input sequence $\left\{\mathbf{x}^{(1)},...,\mathbf{x}^{(t)}\right\}$ up to time *t*.

$$L^{(t)} = -Log\left(p\left(y^{(t)} \mid \left\{\mathbf{x}^{(1)},...,\mathbf{x}^{(t)}\right\}\right)\right)$$

We assume that the outputs $o^{(t)}$, are used as the argument to a softmax function to obtain the vector $\hat{y}^{(t)}$ of probabilities over the output.

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

We start by setting:

$$\frac{\partial L}{\partial L^{(t)}} = 1$$

To we can the work backwards from the end of the sequence.

The gradient $\vec{\nabla}_{o^{(t)}} L$ with respect to the output is the outputs at time step t is computed using the chain rule:

$$(\vec{\nabla}_{o^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i,y^{(t)}}$$

At the final time step, τ, $h^{(\tau)}$ has only $o^{(\tau)}$ as a descendent, so the gradient is simple,

$$\vec{\nabla}_{h^{(\tau)}} L = \mathbf{V}^{\mathrm{T}} \vec{\nabla}_{O^{(\tau)}} L$$

We then iterate backwards in time to back-propagate the gradients from $t = \tau - 1$ back to $t=1$, noting that $h^{(t)}$ for $t < \tau$ has as descendants both $o^{(t)}$ and $h^{(t+1)}$.

$$\begin{aligned}\vec{\nabla}_{h^{(t)}} L &= \left(\frac{\partial h^{(t+1)}}{\partial h^{(t)}}\right)^{\mathrm{T}} \left(\vec{\nabla}_{h^{(t+1)}} L\right) + \left(\frac{\partial o^{(t)}}{\partial h^{(t)}}\right)^{\mathrm{T}} \left(\vec{\nabla}_{O^{(\tau)}} L\right) \\ &= \mathbf{W}^{\mathrm{T}} \left(\vec{\nabla}_{h^{(t)}} L\right) diag\left(1 - \left(h^{(t+1)}\right)^2\right) + \mathbf{V}^{\mathrm{T}} \left(\vec{\nabla}_{O^{(\tau)}} L\right)\end{aligned}$$

where $diag\left(1 - \left(h^{(t+1)}\right)^2\right)$ indicates the diagonal matrix containing the elements $1 - \left(h^{(t+1)}\right)^2$.

This is the Jacobian of the hyperbolic tangent associated with the hidden unit i, at time $t+1$. Once we have the gradients of the internal nodes of the computational graph we can obtain the gradients of the parameter nodes.