

CPSC 532P / LING 530A: Deep Learning for Natural Language Processing (DL-NLP)

Muhammad Abdul-Mageed

muhammad.mageed@ubc.ca

Natural Language Processing Lab

The University of British Columbia

Table of Contents

1 Recurrent Neural Networks

- Definition
- Parameters Sharing
- RNNs as Computational Graphs
- RNNs as Generative Models
- An Example RNN

RNNs (Rumelhart et al., 1986)

- A family of networks classically specializing in sequential data
- Most RNNs can handle sequences of variable length
- RNNs have the advantage of sharing parameters
- **Parameter sharing**: each member of the output is a function of the previous members of the output.
- Each output member is produced using the same **update rule** applied to the previous outputs.

Parameters Sharing

- **Advantage of parameter sharing**: makes it possible to apply the model to examples of different forms (e.g., different lengths).
- Parameter sharing: **specifically important when a piece of information occurs at different positions in time**

Example

- "In **Vancouver** I live".
- "I live in **Vancouver**".

Comparison to Feedforward Net

- A fully connected **feedforward network** would have separate parameters for each input feature, **needing to learn all the rules of the language separately at each position**
- By comparison, a **recurrent neural network** **shares the same weights across several time steps**

RNNs: Computational Graphs With Cycles

- Operate on sequences of $x^{(t)}$ with the time step t ranging from 1 to τ .
- The time step can be the position of an item in the sequence.
- Can also be applied backward
- An extension of the idea of a **computational graph**, to include **cycles**
- **Cycles** represent the influence of the present value of a variable on its own value at a future time step

Unfolding Computational Graphs I

- Consider the classical form of a dynamical system, with a **state** $s^{(t)}$:

1: A Dynamical System

$$s^{(t)} = f(s^{(t-1)}; \theta).$$

- For example, **unfolding 3 times** would give:

$$s^{(3)} = f(s^{(2)}; \theta).$$

$$= f(f(s^{(1)}; \theta); \theta).$$

- The **equation is recurrent** because the definition of s at time t refers back to the same definition at time $t - 1$.

Unfolding Computational Graphs II

- By repeatedly unfolding, we acquire an expression that does not involve recurrence.
- Such an expression can now be expressed by a traditional directed acyclic graph.

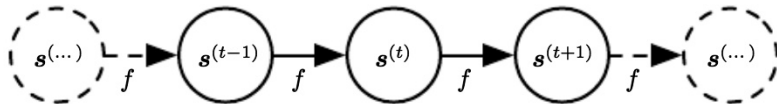


Figure: Each node represents the state at time t , and the function f maps the state at time t to the state at $t+1$. The same parameters are used for all time steps. [From Goodfellow et al., 2016]

RNNs as Computational Graphs

- Consider a **dynamical system driven by an external signal $x^{(t)}$** , where we observe the **state now contains information about the whole past sequence**:

2: Dynamical System With External Signal

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

- Treating the **state as the hidden units** of the network:

3: Hidden State of RNN

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

An RNN (Without Output)

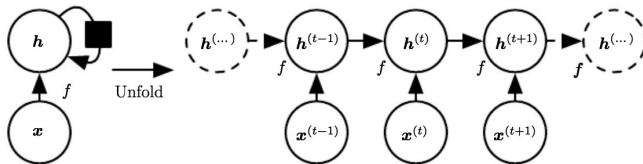


Figure: [From Goodfellow et al., 2016]

More on Unfolding

- **Unfolding** is the operation that maps a circuit as in the **left side** of the figure to a computational graph with repeated pieces as in the **right side**.
- The unfolded graph now has a **size** that depends on the sequence length.

RNNs as Lossy Summarizers

- Typical RNNs would add **extra architectural features** such as **output layers** that read information from the state h to make predictions.
- Can be trained to **predict the future from the past** (e.g., predict the next word in a sequence).
- In these cases, the network typically learns to use $h^{(t)}$ as a kind of **lossy summary** of the task-relevant aspects of the past sequence of inputs up to t .
- Summary is necessarily **lossy**, since it maps an **arbitrary length sequence** $(x^{(t)}, x^{(t-1)}, x^{(t-2)}, \dots, x^{(2)}, x^{(1)})$ to a **fixed length vector** $h^{(t)}$.

Lossy Summarization Illustrated

- Summary might **selectively keep some aspects of the past sequence with more precision than other aspects**, depending on the training criterion
- Consider the case of statistical language modeling where the purpose is to **predict the next word**
- May **not be necessary to keep all information up to time step t** to predict next word
- Most demanding case is when we ask network to predict whole sequence (**auto-encoders**)

Unfolded Recurrence after t steps

- The unfolded recurrence after t steps can be represented with a function $g^{(t)}$:

4: A Function g

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

- The function $g^{(t)}$ takes the whole past sequence $(x^{(t)}, x^{(t-1)}, x^{(t-2)}, \dots, x^{(2)}, x^{(1)})$ as input and produces the current state.
- The unfolded recurrent structure allows us to factorize $g^{(t)}$ into repeated application of a function f .

Learning a Single, Shared Model

- Two factors make it possible to learn a single model f that operates on **all time steps** and **all sequence lengths**:

Learning a Single Model

- 1 Regardless of the sequence length, the **learned model always has the same input size**. Why?
 - 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**.

Advantages of Learning a Single, Shared Model

- **Allows generalization** to sequence lengths that did not appear in the training set
- Allows the model to be **estimated with far fewer training examples** than would be required without parameter sharing.

Mapping input seq \mathbf{x} to output seq \mathbf{o}

- Produce an output at each time step, with recurrent connections between hidden units

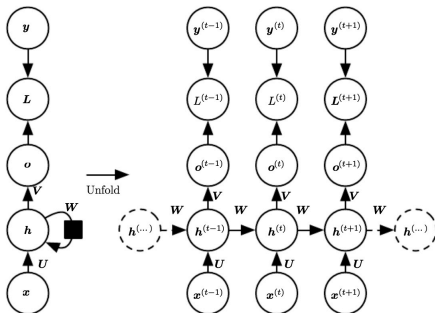


Figure: Three weight matrixes: \mathbf{U} : input-to-hidden connections, \mathbf{U} : hidden-to-hidden recurrent connections, \mathbf{V} : hidden-to-output connections. Loss L compares how far each \mathbf{o} is from its target \mathbf{y} . Loss internally computes $\hat{\mathbf{y}} = \text{softmax}(\mathbf{o})$ [From Goodfellow et al., 2016]

Mapping input seq \mathbf{x} to output seq \mathbf{o} *Cont.*

Notes on Previous Figure

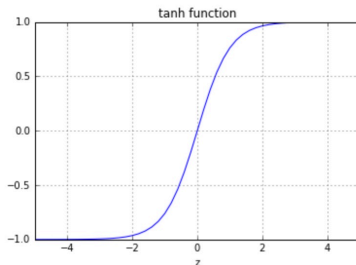
- Let's assume we will use a **hyperbolic tangent (tanh)** as an **activation function**.
- Figure **does not specify** what form the **output** and **loss function** take.
- Assume **output is discrete** (e.g., when the network predicts words or characters).
- Naturally, regard the output \mathbf{o} as giving the **unnormalized log probabilities of each possible value of the discrete variable** (e.g., each word or character).
- **Apply the softmax** as a post-processing step to obtain a vector $\hat{\mathbf{y}}$ of normalized probabilities over the output.

Hyperbolic Tangent Function I

5: Hyperbolic Tangent

$$\tanh(z) = \frac{\sinh z}{\cosh z} = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{e^{2z} - 1}{e^{2z} + 1} = \frac{1 - e^{-2z}}{1 + e^{-2z}}$$

- Strongly neg inputs map to neg outputs; only zero-valued inputs map to near-zero outputs.



Hyperbolic Tangent Function II

```
1 import numpy as np
2 np.set_printoptions(formatter={'float': '{: 0.3f}'.format})
3
4 z = np.arange(-5, 5, .2)
5 t = np.tanh(z)
6 print("Input: values in x\n    {}".format(z))
7 t=np.tanh(z)
8 # Note: Its output is always between -1 and 1
9 print("\nOutput: Tangent Hyperbolic values\n    {}".format(t))
```

Input: values in x

```
[-5.000 -4.800 -4.600 -4.400 -4.200 -4.000 -3.800 -3.600 -3.400 -3.200
-3.000 -2.800 -2.600 -2.400 -2.200 -2.000 -1.800 -1.600 -1.400 -1.200
-1.000 -0.800 -0.600 -0.400 -0.200 0.000 0.200 0.400 0.600 0.800
1.000 1.200 1.400 1.600 1.800 2.000 2.200 2.400 2.600 2.800
3.000 3.200 3.400 3.600 3.800 4.000 4.200 4.400 4.600 4.800]
```

Output: Tangent Hyperbolic values

```
[-1.000 -1.000 -1.000 -1.000 -1.000 -0.999 -0.999 -0.999 -0.998 -0.997
-0.995 -0.993 -0.989 -0.984 -0.976 -0.964 -0.947 -0.922 -0.885 -0.834
-0.762 -0.664 -0.537 -0.380 -0.197 0.000 0.197 0.380 0.537 0.664
0.762 0.834 0.885 0.922 0.947 0.964 0.976 0.984 0.989 0.993
0.995 0.997 0.998 0.999 0.999 0.999 1.000 1.000 1.000 1.000]
```

RNN Update Equations I

- **Forward propagation** begins with a **specification of the initial state $h^{(0)}$** . Then, for each time step from $t = 1$ to $t = \tau$, we apply the following update equations:

6: RNN Equations

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

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

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

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

- where 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.

Mapping Sequences of Same Length

- This RNN **maps an input sequence to an output sequence of the same length.**
- The total loss for a given sequence of x values paired with a sequence of 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 $x^{(1)}, \dots, x^{(t)}$, then: See next slide...

7: RNN Loss

$$\begin{aligned} L &= (x^{(1)}, \dots, x^{(\tau)}, y^{(1)}, \dots, y^{(\tau)}) \\ &= \sum_t L^{(t)} \\ &= - \sum_t \log p_{\text{model}}(y^{(t)} | x^{(1)}, \dots, x^{(t)}) \end{aligned}$$

Expensive Gradient

- Computing the gradients for the loss described earlier involves a forward pass and a backward pass, with a **costly runtime**: $O(\tau)$
- **Cannot be parallelized** due to sequential process: Each time step may be computed only after the previous one

Back-propagation Through Time (BPTT)

BPTT

- States computed during forward pass **must be stored until re-used** during the backward pass
- So, the **memory cost is also $O(\tau)$**
- The back-propagation algorithm applied to this unrolled graph with $O(\tau)$ is called **back-propagation Through Time (BPTT)**.
- To sum up, this network trained with recurrent connections between hidden units is **very powerful**, yet **expensive to train**.
- Are there other options? (Yes, as we will see...)