Lecture 13: Recurrent Neural Nets

Julia Hockenmaier
juliahmr@illinois.edu
3324 Siebel Center
Brief Recap
1D CNNs for text

Text is a (variable-length) **sequence** of words (word vectors)  
[#channels = dimensionality of word vectors]

We can use a **1D CNN** to slide a window of $n$ tokens across:  

— Filter size $n = 3$, stride = 1, no padding

```
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
```

— Filter size $n = 2$, stride = 2, no padding:

```
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
```
Sequence Labeling

**Input:** a sequence of \( n \) tokens/words:

Pierre Vinken, 61 years old, will join IBM’s board as a nonexecutive director Nov. 29

**Output:** a sequence of \( n \) labels, such that each token/word is associated with a label:

**POS-tagging:** Pierre\_NNP Vinken\_NNP ,\_ , 61\_CD years\_NNS old\_JJ ,\_ , will\_MD join\_VB IBM\_NNP ‘s\_POS board\_NN as\_IN a\_DT nonexecutive\_JJ director\_NN Nov.\_NNP 29\_CD .\_.

**Named Entity Recognition:** Pierre\_B-PERS Vinken\_I-PERS ,\_O 61\_O years\_O old\_O ,\_O will\_O join\_O IBM\_B-ORG ‘s\_O board\_O as\_O a\_O nonexecutive\_O director\_O Nov.\_B-DATE 29\_I-DATE .\_O
Statistical POS tagging

She promised to back the bill

\[ w = w^{(1)} \ldots w^{(6)} \]

\[ t = t^{(1)} \ldots t^{(6)} \]

PRP VBD TO VB DT NN

What is the most likely sequence of tags \( t = t^{(1)} \ldots t^{(N)} \) for the given sequence of words \( w = w^{(1)} \ldots w^{(N)} \) ?

\[ t^* = \text{argmax}_t P(t | w) \]
Hidden Markov Models (HMMs)

HMMs are the most commonly used generative models for POS tagging (and other tasks, e.g. in speech recognition)

HMMs make specific **independence assumptions** in $P(t)$ and $P(w | t)$:

1) $P(t)$ is an $n$-gram (typically **bigram** or **trigram**) model over tags:

   - $P_{\text{bigram}}(t) = \prod_{i} P(t^{(i)} | t^{(i-1)})$
   - $P_{\text{trigram}}(t) = \prod_{i} P(t^{(i)} | t^{(i-1)}, t^{(i-2)})$

   $P(t^{(i)} | t^{(i-1)})$ and $P(t^{(i)} | t^{(i-1)}, t^{(i-2)})$ are called **transition probabilities**

2) In $P(w | t)$, each $w^{(i)}$ depends only on [is generated by/conditioned on] $t^{(i)}$:

   - $P(w | t) = \prod_{i} P(w^{(i)} | t^{(i)})$

   $P(w^{(i)} | t^{(i)})$ are called **emission probabilities**

These probabilities don’t depend on the position in the sentence $(i)$, but are defined over word and tag types.

With subscripts $i,j,k$, to index word/tag types, they become $P(t_i | t_i), P(t_i | t_i, t_k), P(w_i | t_i)$
Today’s lecture

Part 1: Recurrent Neural Nets for various NLP tasks

Part 2: Practicalities:
  Training RNNs
  Generating with RNNs
  Using RNNs in complex networks

Part 3: Changing the recurrent architecture to go beyond vanilla RNNs:
  LSTMs, GRUs
Part 1: Recurrent Neural Nets for various NLP tasks
Recurrent Neural Nets (RNNs)

**Feedforward nets** can only handle inputs and outputs that have a **fixed size**.

Recurrent Neural Nets (RNNs) handle **variable length sequences** (as **input** and as **output**).

There are 3 main variants of RNNs, which differ in their internal structure:

- Basic **RNNs** (Elman nets),
- Long Short-Term Memory cells (**LSTMs**),
- Gated Recurrent Units (**GRUs**).
RNNs in NLP

RNNs are used for...

... language modeling and generation, including...
  ... auto-completion and...
  ... machine translation

... sequence classification (e.g. sentiment analysis)

... sequence labeling (e.g. POS tagging)
Recurrent neural networks (RNNs)

**Basic RNN:** Process a sequence of \( T \) inputs and/or generate a sequence of \( T \) outputs by running a [variant of a feedforward] net \( T \) times.

**Recurrence:**
The **hidden state** computed at the **previous step** \( (h^{(t-1)}) \) is fed into the **hidden state** at the **current step** \( (h^{(t)}) \)

With \( H \) hidden units, this requires additional \( H^2 \) parameters.
Basic RNNs

Basic RNNs are feedforward nets where the **hidden layer** gets its input at **time** $t$…

… from the activations of **the input layer** computed **at the same time** $t$, and

… from the activations of **the same hidden layer** **at the previous time** $t-1$:

The input may change at each time step $t$, but the feedforward net is the same at each time $t$
Basic RNNs

Each time step \( t \) corresponds to a feedforward net whose hidden layer \( h^{(t)} \) gets input from the layer below \( (x^{(t)}) \) and from the output of the hidden layer at the previous time step \( h^{(t-1)} \).

Computing the vector of hidden states at time \( t \)

\[
h^{(t)} = g(Uh^{(t-1)} + Wx^{(t)})
\]

The \( i \)-th element of \( h_i \): \( h_i^{(t)} = g \left( \sum_j U_{ji}h_j^{(t-1)} + \sum_k W_{ki}x_k^{(t)} \right) \)
A basic RNN unrolled in time
RNNs for language modeling

If our vocabulary consists of V words, the output layer (at each time step) has V units, one for each word.

The softmax gives a distribution over the V words for the next word.

To compute the probability of string $w^{(0)}w^{(1)}\ldots w^{(n)}w^{(n+1)}$ (where $w^{(0)} = <s>$, and $w^{(n+1)} = <\text{\textbackslash}s>$), feed in $w^{(i-1)}$ as input at time step $i$ and compute

$$\prod_{i=1}^{n+1} P(w^{(i)} | w^{(0)}\ldots w^{(i-1)})$$
RNNs for language generation

To generate \( w^{(0)}w^{(1)}\ldots w^{(n)}w^{(n+1)} \)
(where \( w^{(0)} = \langle s \rangle \), and \( w^{(n+1)} = \langle \langle s \rangle \rangle \))

...Give \( w^{(0)} \) as first input, and

... Choose the next word \( w^{(i)} \) according to the probability

\[
P(w^{(i)} | w^{(0)} \ldots w^{(i-1)})
\]

... Feed the predicted word \( w^{(i)} \) in as input at the next time step.

... Repeat until you generate \( \langle \langle s \rangle \rangle \)
RNNs for language generation

AKA “autoregressive generation”

In a hole?

Figure 9.7

Autoregressive generation with an RNN-based neural language model.

Task is part-of-speech tagging, discussed in detail in Chapter 8. In an RNN approach to POS tagging, inputs are word embeddings and the outputs are tag probabilities generated by a softmax layer over the tagset, as illustrated in Fig. 9.8.

In this figure, the inputs at each time step are pre-trained word embeddings corresponding to the input tokens. The RNN block is an abstraction that represents an unrolled simple recurrent network consisting of an input layer, hidden layer, and output layer at each time step, as well as the shared U, V, and W weight matrices that comprise the network. The outputs of the network at each time step represent the distribution over the POS tagset generated by a softmax layer.

To generate a tag sequence for a given input, we can run forward inference over the input sequence and select the most likely tag from the softmax at each step. Since we're using a softmax layer to generate the probability distribution over the output

Figure 9.8

Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.
RNN for Autocompletion

Prefix

Autogenerated completion

Sampled Words
Softmax
RNN
Embeddings
An RNN for Machine Translation

Early efforts using this clever approach demonstrated surprisingly good results on standard datasets and led to a series of innovations that were the basis for networks discussed in the remainder of this chapter. Chapter 11 provides an in-depth discussion of the fundamental issues in translation as well as the current state-of-the-art approaches to MT. Here, we’ll focus on the powerful models that arose from these early efforts.
Encoder-Decoder (seq2seq) model

Task: Read an input sequence and return an output sequence
- Machine translation: translate source into target language
- Dialog system/chatbot: generate a response

Reading the input sequence: **RNN Encoder**
Generating the output sequence: **RNN Decoder**
Encoder-Decoder (seq2seq) model

Encoder RNN:
reads in the input sequence
passes its last hidden state to the initial hidden state of the decoder

Decoder RNN:
generates the output sequence
typically uses different parameters from the encoder
may also use different input embeddings
RNNs for sequence classification

If we just want to assign one label to the entire sequence, we don’t need to produce output at each time step, so we can use a simpler architecture.

We can use the hidden state of the last word in the sequence as input to a feedforward net:
Basic RNNs for sequence labeling

Sequence labeling (e.g. POS tagging):
Assign **one label to each element** in the sequence.

RNN Architecture:
Each time step has a distribution over output classes.

Extension: add a CRF layer to capture dependencies among labels of adjacent tokens.
RNNs for sequence labeling

In sequence labeling, we want to assign a label or tag $t^{(i)}$ to each word $w^{(i)}$.

Now the output layer gives a (softmax) distribution over the $T$ possible tags, and the hidden layer contains information about the previous words and the previous tags.

To compute the probability of a tag sequence $t^{(1)}...t^{(n)}$ for a given string $w^{(1)}...w^{(n)}$, feed in $w^{(i)}$ (and possibly $t^{(i-1)}$) as input at time step $i$ and compute $P(t^{(i)} \mid w^{(1)}...w^{(i-1)}, t^{(1)}...t^{(i-1)})$.
Part 2: Recurrent Neural Net Practicalities
RNN Practicalities

This part will discuss how to train and use RNNs. We will also discuss how to go beyond basic RNNs.

The last part used a simple RNN with one layer to illustrate how RNNs can be used for different NLP tasks. In practice, more complex architectures are common.

Three complementary ways to extend basic RNNs:
— Using RNNs in more complex networks (bidirectional RNNs, stacked RNNs) [This Part]
— Modifying the recurrent architecture (LSTMs, GRUs) [Part 3]
— Adding attention mechanisms [Next Lecture]
Using RNNs in more complex architectures
Stacked RNNs

We can create an RNN that has “vertical” depth (at each time step) by stacking multiple RNNs:
Bidirectional RNNs

Unless we need to generate a sequence, we can run two RNNs over the input sequence, one in the forward direction, and one in the backward direction. Their hidden states will capture different context information.

To obtain a single hidden state at time $t$: $h_{bi}^{(t)} = h_{fw}^{(t)} \oplus h_{bw}^{(t)}$

where $\oplus$ is typically concatenation.
Bidirectional RNNs for sequence classification

Combine…
…the forward RNN’s hidden state for the last word, and
…the backward RNN’s hidden state for the first word
into a single vector
Training and Generating Sequences with RNNs
How to generate with an RNN

**Greedy decoding:**
Always pick the word with the highest probability
(if you start from `<s>`, this only generates a single sentence)

**Sampling:**
Sample a word according to the given distribution

**Beam search decoding:**
Keep a number of hypotheses after each time step
— **Fixed-width beam**: keep the top $k$ hypotheses
— **Variable-width beam**: keep all hypotheses whose score is within a certain factor of the best score
Beam Decoding (fixed width $k=4$)

Keep the $k$ best options around at each time step.
Operate breadth-first: keep the $k$ best next hypotheses among the best continuations for each of the current $k$ hypotheses.
Reduce beam width every time a sequence is completed (EOS)
Training RNNs for generation

Maximum likelihood estimation (MLE):
Given training samples \( w^{(1)}w^{(2)}...w^{(T)} \), find the parameters \( \theta^* \) that assign the largest probability to these training samples:
\[
\theta^* = \arg\max_{\theta} P_\theta(w^{(1)}w^{(2)}...w^{(T)}) = \arg\max_{\theta} \prod_{t=1..T} P_\theta(w^{(t)}|w^{(1)}...w^{(t-1)})
\]

Since \( P_\theta(w^{(1)}w^{(2)}...w^{(T)}) \) is factored into \( P_\theta(w^{(t)}|w^{(1)}...w^{(t-1)}) \), we can train models to assign a higher probability to the word \( w^{(t)} \) that occurs in the training data after \( w^{(1)}...w^{(t-1)} \) than any other word \( w_i \in V \):
\[
\forall i=1...|V| P_\theta(w^{(t)}|w^{(1)}...w^{(t-1)}) \geq P_\theta(w_i|w^{(1)}...w^{(t-1)})
\]
This is also called teacher forcing.
Teacher forcing

Each training sequence $w^{(1)}w^{(2)}…w^{(T)}$ turns into $T$ training items:

Give $w^{(1)}w^{(2)}…w^{(t-1)}$ as input to the RNN, and train it to maximize the probability of $w^{(t)}$

(as you would in standard classification, or when training an n-gram language model).
Problems with teacher forcing

Exposure bias:

When we *train* an RNN for sequence generation, the prefix $y^{(1)} \ldots y^{(t-1)}$ that we condition on comes from the original data.

When we *use* an RNN for sequence generation, the prefix $y^{(1)} \ldots y^{(t-1)}$ that we condition on is also generated by the RNN,

— The model is used on data that may look quite different from the data it was trained on.
— The model is not trained to predict the best next token within a generated sequence, or to predict the best sequence.
— Errors at earlier time-steps propagate through the sequence.
Remedies

Minimum risk training:
— define a loss function (e.g. negative BLEU) to compare generated sequences against gold sequences
— Minimize risk (expected loss on training data) such that candidates outputs with a smaller loss (higher BLEU score) have higher probability.

Reinforcement learning-based approaches:
— use BLEU as a reward (i.e. like MRT)
— perhaps pre-train model first with standard teacher forcing.

GAN-based approaches (“professor forcing”)
— combine standard RNN with an adversarial model that aims to distinguish original from generated sequences
Part 3: RNN Variants
RNN variants: LSTMs, GRUs

Long Short-Term Memory networks (LSTMs) are RNNs with a more complex recurrent architecture.

Gated Recurrent Units (GRUs) are a simplification of LSTMs.

Both contain “Gates” to control how much of the input or previous hidden state to forget or remember.
From RNNs to LSTMs

In Vanilla (Elman) RNNs, the current hidden state $h(t)$ is a nonlinear function of the previous hidden state $h(t-1)$ and the current input $x(t)$:

$$h(t) = g(Uh(t-1) + Wx(t) + b_h)$$

With $g=tanh$ (the original definition):
⇒ Models suffer from the vanishing gradient problem: they can’t be trained effectively on long sequences.

With $g=ReLU$
⇒ Models suffer from the exploding gradient problem: they can’t be trained effectively on long sequences.
From RNNs to LSTMs

LSTMs (Long Short-Term Memory networks) were introduced to overcome the vanishing gradient problem. Hochreiter and Schmidhuber, Neural Computation 9(8), 1997
https://www.bioinf.jku.at/publications/older/2604.pdf

Like RNNs, LSTMs contain a hidden state that gets passed through the network and updated at each time step.

LSTMs contain an additional cell state that also gets passed through the network and updated at each time step.

LSTMs contain three different gates (input/forget/output) that read in the previous hidden state and current input to decide how much of the past hidden and cell states to keep.

These gates mitigate the vanishing/exploding gradient problem.
Recap: Activation functions

**Hyperbolic Tangent:** \( \tanh(x) = \frac{\exp(2x) - 1}{\exp(2x) + 1} \in [-1, +1] \)

**Rectified Linear Unit:** \( \text{ReLU}(x) = \max(0, x) \in [0, +\infty] \)

**Sigmoid (logistic function):** \( \sigma(x) = \frac{1}{1 + \exp(-x)} \in [0,1] \)
### RNN variants: LSTMs, GRUs

**Long Short-Term Memory networks (LSTMs)** are RNNs with a more complex recurrent architecture.

**Gated Recurrent Units (GRUs)** are a simplification of LSTMs.

Both contain “**Gates**” to control how much of the input or past hidden state to forget or remember.

---

A **gate** performs **element-wise multiplication** of

- a) a $d$-dimensional **sigmoid** layer $g$ (all elements between 0 and 1), and
- b) a $d$-dimensional **input** vector $u$

**Result:** $d$-dimensional **output** vector $v$ which is like the input $u$, but **elements** where $g_i \approx 0$ are (partially) “**forgotten**”
Gating: element-wise product

\[ v = g \otimes u = [g_1u_1, g_2u_2, \ldots, g_du_d] \]

A gate performs element-wise multiplication of

a) a \(d\)-dimensional sigmoid layer \(g\)
   (all elements between 0 and 1), and
b) a \(d\)-dimensional input vector \(u\)

Result: \(d\)-dimensional output vector \(v\) which is like the input \(u\),
but elements where \(g_i \approx 0\) are (partially) “forgotten”
Gating mechanisms

**Gates** are trainable layers with a **sigmoid** activation function often determined by the current input $x^{(t)}$ and the (last) hidden state $h^{(t-1)}$ eg.:

$$g_k^{(t)} = \sigma(W_k x^{(t)} + U_k h^{(t-1)} + b_k)$$

$g$ is a vector of (Bernoulli) probabilities ($\forall i : 0 \leq g_i \leq 1$)

Unlike traditional (0,1) gates, neural gates are differentiable (we can train them)

$g$ is combined with another vector $u$ (of the same dimensionality) by **element-wise multiplication** (Hadamard product): $v = g \otimes u$

If $g_i \approx 0$, $v_i \approx 0$, and if $g_i \approx 1$, $v_i \approx u_i$

Each $g_i$ has its own set of trainable parameters to determine how much of $u_i$ to keep

Gates can also be used to form **linear combinations of two input vectors** $t, u$:

- **Addition of two independent gates**: $v = g_1 \otimes t + g_2 \otimes u$
- **Linear interpolation (coupled gates)**: $v = g \otimes t + (1 - g) \otimes u$
Long Short-Term Memory Networks (LSTMs)

At time $t$, the LSTM cell reads in
- a $c$-dimensional previous cell state vector $c^{(t-1)}$
- an $h$-dimensional previous hidden state vector $h^{(t-1)}$
- a $d$-dimensional current input vector $x^{(t)}$

At time $t$, the LSTM cell returns
- a $c$-dimensional new cell state vector $c^{(t)}$
- an $h$-dimensional new hidden state vector $h^{(t)}$
- (which may also be passed to an output layer)

https://colah.github.io/posts/2015-08-Understanding-LSTMs/
LSTM operations

Based on the previous cell state $c^{(t-1)}$, previous hidden state $h^{(t-1)}$ and the current input $x^{(t)}$, the LSTM computes:

... A new **intermediate cell state** $\tilde{c}^{(t)}$ that depends on $h^{(t-1)}$ and $x^{(t)}$:

$$\tilde{c}^{(t)} = \tanh \left( W_c x^{(t)} + U_c h^{(t-1)} + b_c \right)$$

... **Three gates** $f^{(t)}$, $i^{(t)}$, $o^{(t)}$, which each depend on $h^{(t-1)}$ and $x^{(t)}$:

- The **forget gate** $f^{(t)} = \sigma \left( W_f x^{(t)} + U_f h^{(t-1)} + b_f \right)$ decides how much of the **last** $c^{(t-1)}$ to remember in the new cell state: $f^{(t)} \otimes c^{(t-1)}$

- The **input gate** $i^{(t)} = \sigma \left( W_i x^{(t)} + U_i h^{(t-1)} + b_i \right)$ decides how much of the **intermediate** $\tilde{c}^{(t)}$ to use in the new cell state: $i^{(t)} \otimes \tilde{c}^{(t)}$

- The **output gate** $o^{(t)} = \sigma \left( W_o x^{(t)} + U_o h^{(t-1)} + b_o \right)$ decides how much of the **new** $c^{(t)}$ to use in the next hidden state: $h^{(t)} = o^{(t)} \otimes c^{(t)}$

The new **cell state** $c^{(t)} = \tanh \left( f^{(t)} \otimes c^{(t-1)} + i^{(t)} \otimes \tilde{c}^{(t)} \right)$ is a **linear combination** of cell states $c^{(t-1)}$ and $\tilde{c}^{(t)}$ that depends on forget gate $f^{(t)}$ and input gate $i^{(t)}$.

The new **hidden state** $h^{(t)} = o^{(t)} \otimes c^{(t)}$ depends on $c^{(t)}$ and the output gate $o^{(t)}$.
Gated Recurrent Units (GRUs)

Based on $h^{(t-1)}$ and $x^{(t)}$, a GRU computes:

- **a reset gate** $r^{(t)}$ to determine how much of $h^{(t-1)}$ to keep in $\tilde{h}^{(t)}$
  
  $r^{(t)} = \sigma(W_r x^{(t)} + U_r h^{(t-1)} + b_r)$

- **an intermediate hidden state** $\tilde{h}^{(t)}$ that depends on $x^{(t)}$ and $r^{(t)} \otimes h^{(t-1)}$
  
  $\tilde{h}^{(t)} = \phi(W_h x^{(t)} + U_h (r^{(t)} \otimes h^{(t-1)}) + b_r)$ [\(\phi = \tanh\) or ReLU]

- **an update gate** $z^{(t)}$ to determine how much of $h^{(t-1)}$ to keep in $h^{(t)}$
  
  $z^{(t)} = \sigma(W_z x^{(t)} + U_z h^{(t-1)} + b_r)$

- **a new hidden state** $h^{(t)}$ as a linear interpolation of $h^{(t-1)}$ and $\tilde{h}^{(t)}$ with weights determined by the (coupled) update gate $z^{(t)}$
  
  $h^{(t)} = z^{(t)} \otimes h^{(t-1)} + (1 - z^{(t)}) \otimes \tilde{h}^{(t)}$

LSTMs vs GRUs

LSTMs are more expressive than GRUs and basic RNNs (they’re better at learning long-range dependencies)

But GRUs are easier to train than LSTMs (useful when training data is limited)