Natural Language Processing

Anoop Sarkar
anoopsarkar.github.io/nlp-class

Simon Fraser University

November 6, 2019

Natural Language Processing

Anoop Sarkar
anoopsarkar.github.io/nlp-class

Simon Fraser University

Part 1: Long distance dependencies

Long distance dependencies

Example

- He doesn't have very much confidence in himself
- She doesn't have very much confidence in herself

n-gram Language Models: $P(w_i \mid w_{i-n+1}^{i-1})$

$$
\begin{aligned}
P(\text{himself} \quad &| \quad \text{confidence}, \text{in}) \\
P(\text{herself} \quad &| \quad \text{confidence}, \text{in})
\end{aligned}
$$

What we want: $P(w_i \mid w_{<i})$

$$
\begin{aligned}
P(\text{himself} \quad &| \quad \text{He}, \ldots, \text{confidence}) \\
P(\text{herself} \quad &| \quad \text{She}, \ldots, \text{confidence})
\end{aligned}
$$

Long distance dependencies

Other examples

- **Selectional preferences**: *I ate lunch with a fork* vs. *I ate lunch with a backpack*
- **Topic**: *Babe Ruth was able to touch the home plate yet again* vs. *Lucy was able to touch the home audiences with her humour*
- **Register**: Consistency of register in the entire sentence, e.g. informal (Twitter) vs. formal (scientific articles)

Language Models

Chain Rule and ignore some history: the trigram model

$$
\begin{aligned}
& p(w_1, \ldots, w_n) \\
& \quad \approx \quad p(w_1)p(w_2 \mid w_1)p(w_3 \mid w_1, w_2) \ldots p(w_n \mid w_{n-2}, w_{n-1}) \\
& \quad \approx \quad \prod_t p(w_{t+1} \mid w_{t-1}, w_t)
\end{aligned}
$$

How can we address the long-distance issues?

- Skip n-gram models. Skip an arbitrary distance for n-gram context.
- Variable n in n-gram models that is adaptive
- **Problems**: Still "all or nothing". Categorical rather than soft.

Natural Language Processing

Anoop Sarkar
anoopsarkar.github.io/nlp-class

Simon Fraser University

Part 2: Neural Language Models

Neural Language Models

Use Chain rule and approximate using a neural network

$$p(w_1, \ldots, w_n) \approx \prod_t p(w_{t+1} \mid \underbrace{\phi(w_1, \ldots, w_t)}_{\text{capture history with vector } s(t)})$$

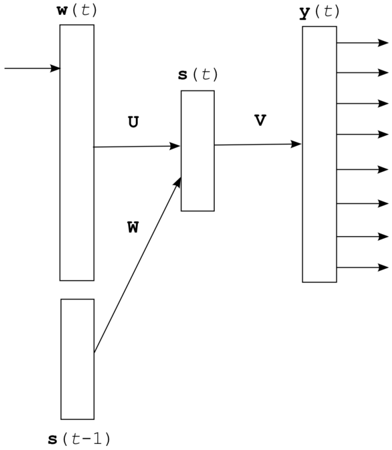
Recurrent Neural Network

- Let y be the output w_{t+1} for current word w_t and history $w_1, \ldots, w_t$
- $s(t) = f(U_{xh} \cdot w(t) + W_{hh} \cdot s(t-1))$ where f is sigmoid / tanh
- $s(t)$ encapsulates history using single vector of size h
- Output word at time step w_{t+1} is provided by $y(t)$
- $y(t) = g(V_{hy} \cdot s(t))$ where g is softmax

Neural Language Models

Recurrent Neural Network

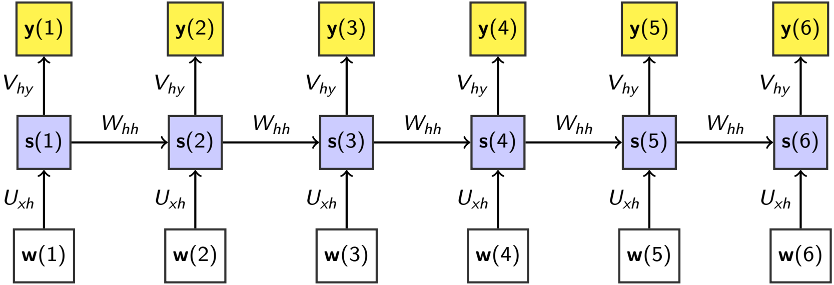
Single time step in RNN:



- Input layer is a one hot vector and output layer **y** have the same dimensionality as vocabulary (10K-200K).
- One hot vector is used to look up word embedding **w**
- "Hidden" layer **s** is orders of magnitude smaller (50-1K neurons)
- U is the matrix of weights between input and hidden layer
- V is the matrix of weights between hidden and output layer
- Without recurrent weights W, this is equivalent to a bigram feedforward language model

Neural Language Models

Recurrent Neural Network



What is stored and what is computed:

- Model parameters: $\mathbf{w} \in \mathbb{R}^{x}$ (word embeddings);
  $U_{xh} \in \mathbb{R}^{x \times h}$; $W_{hh} \in \mathbb{R}^{h \times h}$; $V_{hy} \in \mathbb{R}^{h \times y}$ where $y = |\mathcal{V}|$.
- Vectors computed during forward pass: $\mathbf{s}(t) \in \mathbb{R}^{h}$; $\mathbf{y}(t) \in \mathbb{R}^{y}$ and each $\mathbf{y}(t)$ is a probability over vocabulary $\mathcal{V}$.

Natural Language Processing

Anoop Sarkar
anoopsarkar.github.io/nlp-class

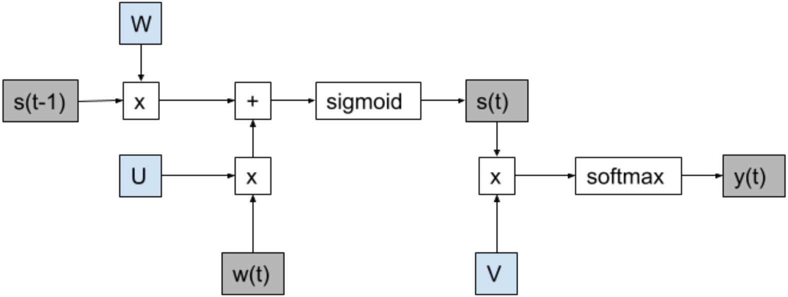
Simon Fraser University

Part 3: Training RNN Language Models

Neural Language Models

Recurrent Neural Network

Computational Graph for an RNN Language Model



Training of RNNLM

- The training is performed using Stochastic Gradient Descent (SGD)
- We go through all the training data iteratively, and update the weight matrices U, W and V (after processing every word)
- Training is performed in several "epochs" (usually 5-10)
- An epoch is one pass through the training data
- As with feedforward networks we have two passes:

  Forward pass : collect the values to make a prediction (for each time step)

  Backward pass : back-propagate the error gradients (through each time step)

Training of RNNLM

Forward pass

- In the forward pass we compute a hidden state $s(t)$ based on previous states $1, \ldots, t-1$
  - $s(t) = f(U_{xh} \cdot w(t) + W_{hh} \cdot s(t-1))$
  - $s(t) = f(U_{xh} \cdot w(t) + W_{hh} \cdot f(U_{xh} \cdot w(t) + W_{hh} \cdot s(t-2)))$
  - $s(t) = f(U_{xh} \cdot w(t) + W_{hh} \cdot$ $f(U_{xh} \cdot w(t) + W_{hh} \cdot f(U_{xh} \cdot w(t) + W_{hh} \cdot s(t-3))))$
  - etc.
- Let us assume f is linear, e.g. $f(x) = x$.
- Notice how we have to compute $W_{hh} \cdot W_{hh} \cdot \ldots = \prod_i W_{hh}$
- By examining this repeated matrix multiplication we can show that the norm of $W_{hh} \to \infty$ (explodes)
- This is why f is set to a function that returns a bounded value (sigmoid / tanh)

Training of RNNLM

Backward pass

- Gradient of the error vector in the output layer $\mathbf{e}_o(t)$ is computed using a cross entropy criterion:

$$\mathbf{e}_o(t) = \mathbf{d}(t) - \mathbf{y}(t)$$

- $\mathbf{d}(t)$ is a target vector that represents the word $w(t+1)$ represented as a one-hot (1-of-$\mathcal{V}$) vector

Training of RNNLM

Backward pass

- Weights V between the hidden layer $s(t)$ and the output layer $y(t)$ are updated as

$$V^{(t+1)} = V^{(t)} + \mathbf{s}(t) \cdot \mathbf{e}_o(t) \cdot \alpha$$

- where α is the learning rate

Training of RNNLM

Backward pass

- Next, gradients of errors are propagated from the output layer to the hidden layer

$$\mathbf{e}_h(t) = d_h(\mathbf{e}_o \cdot V, t)$$

- where the error vector is obtained using function $d_h()$ that is applied element-wise:

$$d_{hj}(x, t) = x \cdot s_j(t)(1 - s_j(t))$$

Training of RNNLM

Backward pass

- Weights U between the input layer $w(t)$ and the hidden layer $s(t)$ are then updated as

$$U^{(t+1)} = U^{(t)} + \mathbf{w}(t) \cdot \mathbf{e}_h(t) \cdot \alpha$$

- Similarly the word embeddings **w** can also be updated using the error gradient.

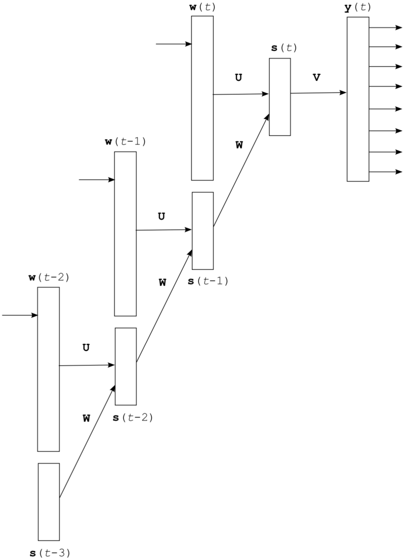
Training of RNNLM: Backpropagation through time

Backward pass

- The recurrent weights W are updated by unfolding them in time and training the network as a deep feedforward neural network.
- The process of propagating errors back through the recurrent weights is called Backpropagation Through Time (BPTT).

Training of RNNLM: Backpropagation through time

Fig. from [1]: RNN unfolded as a deep feedforward network 3 time steps back in time



Training of RNNLM: Backpropagation through time

Backward pass

- Error propagation is done recursively as follows (it requires the states of the hidden layer from the previous time steps τ to be stored):

$$\mathbf{e}(t-\tau-1) = d_h(\mathbf{e}_h(t-\tau) \cdot W, t-\tau-1)$$

- The error gradients quickly vanish as they get backpropagated in time (less likely if we use sigmoid / tanh)
- We use gated RNNs to stop gradients from vanishing or exploding.
- Popular gated RNNs are *long short-term memory* RNNs aka LSTMs and *gated recurrent units* aka GRUs.

Training of RNNLM: Backpropagation through time

Backward pass

- The recurrent weights W are updated as:

$$W^{(t+1)} = W^{(t)} + \sum_{z=0}^{T} \mathbf{s}(t-z-1) \cdot \mathbf{e}_h(t-z) \cdot \alpha$$

- Note that the matrix W is changed in one update at once, not during backpropagation of errors.

Natural Language Processing

Anoop Sarkar
anoopsarkar.github.io/nlp-class

Simon Fraser University

Part 4: Gated Recurrent Units

Interpolation for hidden units

u: use history or forget history

- For RNN state $s(t) \in \mathbb{R}^h$ create a binary vector $u \in \{0,1\}^h$

$$u_i = \begin{cases} 1 & \text{use the new hidden state (standard RNN update)} \\ 0 & \text{copy previous hidden state and ignore RNN update} \end{cases}$$

- Create an intermediate hidden state $\tilde{s}(t)$ where f is tanh:

$$\tilde{s}(t) = f(U_{xh} \cdot w(t) + W_{hh} \cdot s(t-1))$$

- Use the binary vector u to interpolate between copying prior state $s(t-1)$ and using new state $\tilde{s}(t)$:

$$s(t) = (1-u) \odot s(t-1) + u \odot \tilde{s}(t)$$

$\odot$ is elementwise multiplication

Interpolation for hidden units

r: reset or retain each element of hidden state vector

- For RNN state $s(t-1) \in \mathbb{R}^h$ create a binary vector $r \in \{0,1\}^h$

$$r_i = \begin{cases} 1 & \text{if } s_i(t-1) \text{ should be used} \\ 0 & \text{if } s_i(t-1) \text{ should be ignored} \end{cases}$$

- Modify intermediate hidden state $\tilde{s}(t)$ where f is tanh:

$$\tilde{s}(t) = f(U_{xh} \cdot w(t) + W_{hh} \cdot (r \odot s(t-1)))$$

- Use the binary vector u to interpolate between $s(t-1)$ and $\tilde{s}(t)$:

$$s(t) = (1-u) \odot s(t-1) + u \odot \tilde{s}(t)$$

Interpolation for hidden units

Learning u and r

- Instead of binary vectors $u \in \{0,1\}^h$ and $r \in \{0,1\}^h$ we want to *learn* u and r
- Let $u \in [0,1]^h$ and $r \in [0,1]^h$
- Learn these two h dimensional vectors using equations similar to the RNN hidden state equation:

$$
\begin{aligned}
u(t) &= \sigma\left(U^u_{xh} \cdot w(t) + W^u_{hh} \cdot s(t-1)\right) \\
r(t) &= \sigma\left(U^r_{xh} \cdot w(t) + W^r_{hh} \cdot s(t-1)\right)
\end{aligned}
$$

- The sigmoid function σ ensures that each element of u and r is between $[0,1]$
- The *use history* u and *reset element* r vectors use different parameters U^u, W^u and U^r, W^r

Interpolation for hidden units

Gated Recurrent Unit (GRU)

- Putting it all together:

$$
\begin{aligned}
u(t) &= \sigma\left(U^u_{xh} \cdot w(t) + W^u_{hh} \cdot s(t-1)\right) \\
r(t) &= \sigma\left(U^r_{xh} \cdot w(t) + W^r_{hh} \cdot s(t-1)\right) \\
\tilde{s}(t) &= \tanh(U_{xh} \cdot w(t) + W_{hh} \cdot (r(t) \odot s(t-1))) \\
s(t) &= (1 - u(t)) \odot s(t-1) + u(t) \odot \tilde{s}(t)
\end{aligned}
$$

Interpolation for hidden units

Long Short-term Memory (LSTM)

- Split up $u(t)$ into two different gates $i(t)$ and $f(t)$:

$$
\begin{aligned}
i(t) &= \sigma\left(U^i_{xh} \cdot w(t) + W^i_{hh} \cdot s(t-1)\right) \\
f(t) &= \sigma\left(U^f_{xh} \cdot w(t) + W^f_{hh} \cdot s(t-1)\right) \\
r(t) &= \sigma\left(U^r_{xh} \cdot w(t) + W^r_{hh} \cdot s(t-1)\right) \\
\tilde{s}(t) &= \tanh(U_{xh} \cdot w(t) + W_{hh} \cdot \underbrace{s(t-1)}_{\text{GRU:}r(t)\odot s(t-1)}) \\
\hat{s}(t) &= \underbrace{f(t) \odot s(t-1) + i(t) \odot \tilde{s}(t)}_{\text{GRU:}(1-u(t))\odot s(t-1)+u(t)\odot\tilde{s}(t)} \\
s(t) &= r(t) \odot \tanh\left(\hat{s}(t)\right)
\end{aligned}
$$

- So LSTM is a GRU plus an extra U_{xh}, W_{hh} and tanh.
- **Q**: what happens if $f(t)$ is set to $1 - i(t)$? **A**: read [3]

Natural Language Processing

Anoop Sarkar
anoopsarkar.github.io/nlp-class

Simon Fraser University

Part 5: Sequence prediction using RNNs

Representation: finding the right parameters

Problem: Predict ?? using context, $P(?? \mid \texttt{context})$

Profits/N soared/V at/P Boeing/?? Co. , easily topping forecasts on Wall Street , as their CEO Alan Mulally announced first quarter results .

Representation: history

- The input is a tuple: $(x_{[1:n]}, i)$ [ignoring y_{-1} for now]
- $x_{[1:n]}$ are the n words in the input
- i is the index of the word being tagged
- For example, for $x_4 =$ Boeing
- We can use an RNN to summarize the entire context at $i = 4$
  - $x_{[1:i-1]} =$ (Profits, soared, at)
  - $x_{[i+1:n]} =$ (Co., easily, ..., results, .)

Locally normalized RNN taggers

Log-linear model over history, tag pair (h, t)

$$\log \Pr(y \mid h) = \mathbf{w} \cdot \mathbf{f}(h, y) - log \sum_{y'} \exp\left(\mathbf{w} \cdot \mathbf{f}(h, y')\right)$$

$\mathbf{f}(h, y)$ is a vector of feature functions

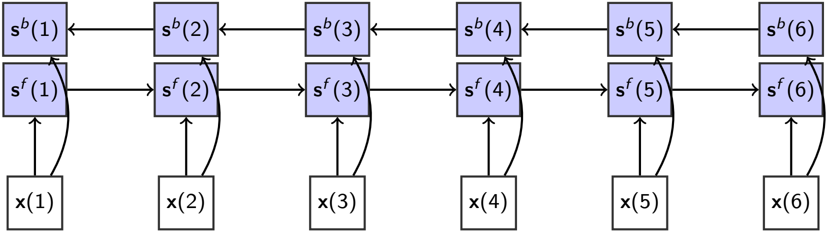
RNN for tagging

- Replace $\mathbf{f}(h, y)$ with RNN hidden state $s(t)$
- Define the output logprob: $\log \Pr(y \mid h) = \log y(t)$
- $y(t) = g(V \cdot s(t))$ where g is softmax
- In neural LMs the output $y \in \mathcal{V}$ (vocabulary)
- In sequence tagging using RNNs the output $y \in \mathcal{T}$ (tagset)

$$\log \Pr(y_{[1:n]} \mid x_{[1:n]}) = \sum_{i=1}^{n} \log \Pr(y_i \mid h_i)$$

Bidirectional RNNs

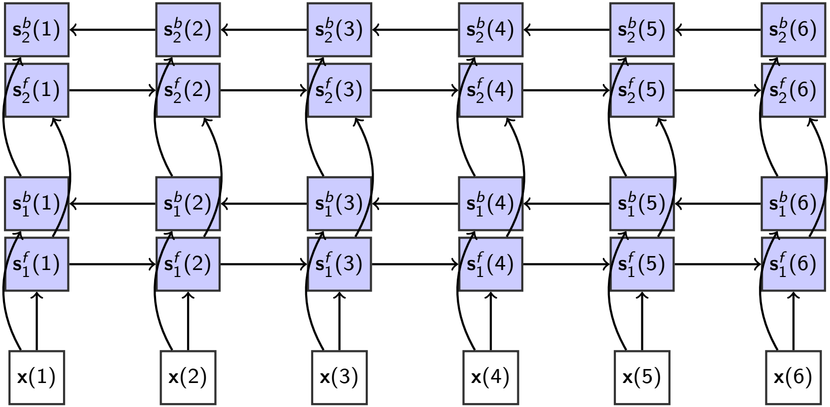
Fig. from [2]



Bidirectional RNN

Bidirectional RNNs can be Stacked

Fig. from [2]



Two Bidirectional RNNs stacked on top of each other

Natural Language Processing

Anoop Sarkar
anoopsarkar.github.io/nlp-class

Simon Fraser University

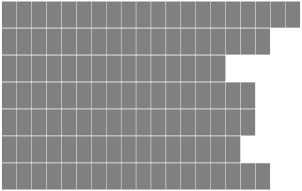
Part 6: Training RNNs on GPUs

Parallelizing RNN computations

Fig. from [2]

Apply RNNs to *batches* of sequences

Present the data as a 3D tensor of $(T \times B \times F)$. Each dynamic update will now be a matrix multiplication.



Binary Masks

Fig. from [2]

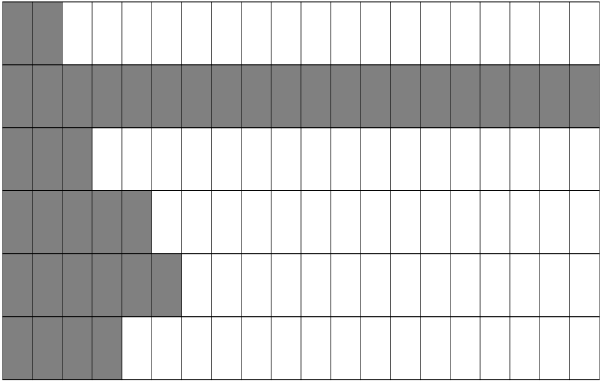
A *mask* matrix may be used to aid with computations that ignore the padded zeros.

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0

Binary Masks

Fig. from [2]

It may be necessary to (partially) sort your data.



[1] Tomas Mikolov
Recurrent Neural Networks for Language Models. Google Talk.
2010.

[2] Philemon Brakel
MLIA-IQIA Summer School notes on RNNs
2015.

[3] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, Jürgen Schmidhuber
LSTM: A Search Space Odyssey
2017.

Acknowledgements

Many slides borrowed or inspired from lecture notes by Michael Collins, Chris Dyer, Kevin Knight, Chris Manning, Philipp Koehn, Adam Lopez, Graham Neubig, Richard Socher and Luke Zettlemoyer from their NLP course materials.

All mistakes are my own.

A big thank you to all the students who read through these notes and helped me improve them.