Natural Language Understanding

Lecture 12: Recurrent Neural Networks

Frank Keller

School of Informatics
University of Edinburgh
keller@inf.ed.ac.uk

March 7, 2017

1. Introduction
2. Simple Recurrent Networks
   - Architecture
   - Training
   - Results
3. Backpropagation Through Time
   - Architecture
   - Standard Backpropagation
   - Going Back in Time

Reading: Mikolov et al. (2010).
Additional background: Guo (2013).

Language Modeling

A language model assigns probabilities to sequences of words:

- often a simple *n*-gram model is used; trigram models perform well at this task;
- more structured language models include syntactic information; they can be derived from parsers;
- applications include:
    - speech recognition;
    - machine translation;
    - text completion;
    - grammar checking.

We will discuss Mikolov et al.'s (2010) language model based on *recurrent neural networks* (RNNs).

n-gram Language Models

We want to predict the probability of a sequence of words $w_1 \dots w_k$. Using the chain rule, this can be decomposed as:

$$P(w_1 \dots w_k) = P(w_k|w_1 \dots w_{k-1})P(w_{k-1}|w_1 \dots w_{k-2}) \cdots P(w_2|w_1)P(w_1)$$

If we now limit the history (the words in the context that are relevant) to n, we get:

$$P(w_1 \dots w_k) = \prod_{i=1}^{k} P(w_i|w_{i-n+1} \dots w_{i-1})$$

This is the *n-gram approximation* of $P(w_1 \dots w_k)$.

Applications of Language Modeling

Machine translation:

- word ordering: $P(\text{the cat is small}) > P(\text{small the is cat})$;
- word choice: $P(\text{walking home after school}) >$ $P(\text{walking house after school})$.

Grammar checking:

- word substitutions: $P(\text{the principal resigned}) > P(\text{the principle resigned})$;
- agreement errors: $P(\text{the cats sleep in the basket}) >$ $P(\text{the cats sleeps in the basket})$.

n-gram Language Models

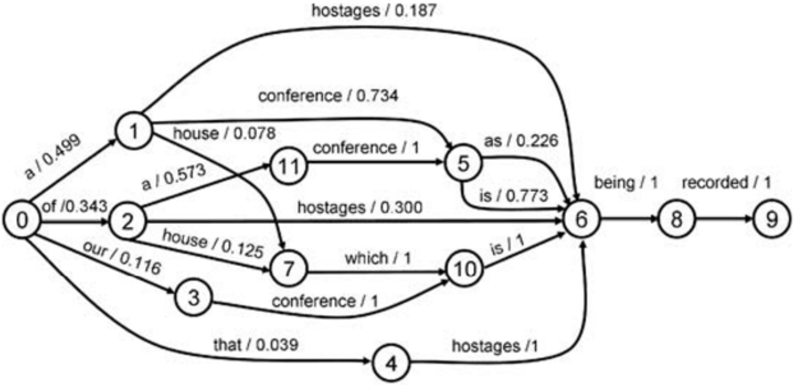
If we have a sequence of words $w_1 \dots w_k$ then we can use the language model to predict the next word w_{k+1}:

$$\hat{w}_{k+1} = \underset{w_{k+1}}{\operatorname{argmax}} P(w_{k+1} | w_1 \dots w_k)$$

Being able to predict the next word is useful for applications that process input in real time (word-by-word).

Applications of Language Modeling

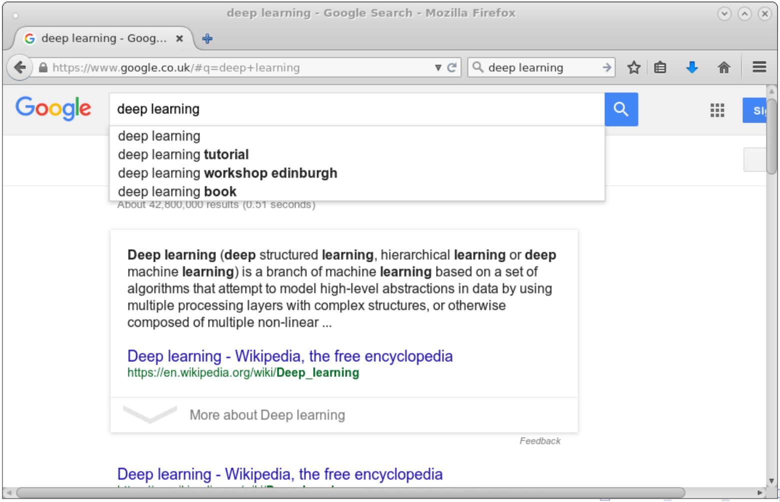
Speech recognition:



a conference is being recorded

Applications of Language Modeling

Text completion:



Estimating n-gram Probabilities

We can get maximum likelihood estimates for the conditional probabilities from n-gram counts in a corpus:

$$P(w_2|w_1) = \frac{n_{(w_1,w_2)}}{n_{(w_1)}} \qquad P(w_3|w_1, w_2) = \frac{n_{(w_1,w_2,w_3)}}{n_{(w_1,w_2)}}$$

But building good n-gram language models can be difficult:

- the higher the n, the better the performance;
- but higher-order n-grams are very sparse;
- good models need to be trained on billions of words;
- this entails large memory requirements;
- fancy smoothing and backoff techniques are required.

Modeling Context

Context is important in language modeling:

- *n*-gram language models use a limited context (fixed *n*);
- feedforward networks can be used for language modeling, but their input is also of fixed size;
- but linguistic dependencies can be arbitrarily long.

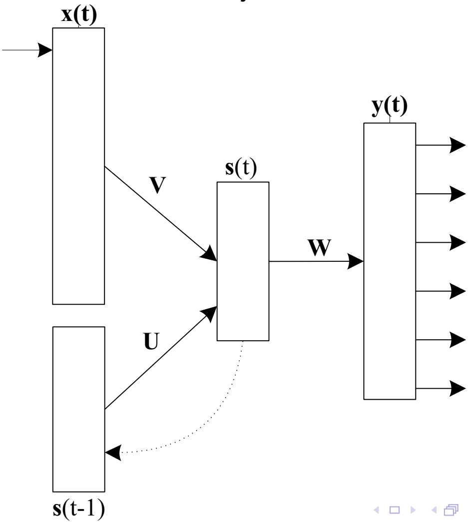
This is where *recurrent neural networks* come in:

- the input of an RNN includes a copy of the previous hidden layer of the network;
- effectively, the RNN buffers all the inputs it has seen before;
- it can thus model context dependencies of arbitrary length.

We will look at *simple recurrent networks* first.

Architecture

The simple recurrent networks only looks back one time step:



Architecture

We have input layer x, hidden layer s (state), output layer y. The input at time t is $x(t)$, output is $y(t)$, and hidden layer $s(t)$.

$$s_j(t) = f(net_j(t)) \tag{1}$$

$$net_j(t) = \sum_i^l x_i(t)v_{ji} + \sum_h^m s_h(t-1)u_{jh} \tag{2}$$

$$y_k(t) = g(net_k(t)) \tag{3}$$

$$net_k(t) = \sum_j^m s_j(t)w_{kj} \tag{4}$$

where $f(z)$ is the sigmoid, and $g(z)$ the softmax function:

$$f(z) = \frac{1}{1+e^{-z}} \qquad g(z_m) = \frac{e^{z_m}}{\sum_k e^{z_k}}$$

Input and Output

- For initialization, set s and x to small random values;
- for each time step, copy $s(t-1)$ and use it to compute $s(t)$;
- input vector $x(t)$ uses 1-of-N (one hot) encoding over the words in the vocabulary;
- output vector $y(t)$ is a probability distribution over the next word given the current word $w(t)$ and context $s(t-1)$;
- size of hidden layer is usually 30–500 units, depending on size of training data.

Training

We can use standard backprop with stochastic gradient descent:

- simply treat the network as a feedforward network with $s(t-1)$ as additional input;
- backpropagate the error to adjust weight matrices $\mathbf{U}$ and $\mathbf{V}$;
- present all of the training data in each epoch;
- test on validation data to see if log-likelihood of training data improves;
- adjust learning rate if necessary.

Error signal for training:

$$\text{error}(t) = \text{desired}(t) - y(t)$$

where $\text{desired}(t)$ is the one-hot encoding of the correct next word.

Experimental Set-up

Evaluations: language modeling in speech recognition:

- interpolate RNN with a standard 5-gram language model with Kneser-Ney smoothing;
- replace rare words with a special rare word token;
- measure perplexity and word error rate (WER);
- try hidden layers of different sizes and different thresholds for rare words;
- test on Wall Street Journal and NIST RT05 corpora.

Experimental Set-up

The perplexity of a language model that defines a probability distribution P on a corpus $w_1 \dots w_N$ is:

$$\begin{aligned} ppl(w_1 \dots w_N) &= \sqrt[N]{\frac{1}{P(w_1 \dots w_N)}} \\ &= \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-n+1} \dots w_{i-1})}} \end{aligned}$$

Perplexity is the inverse of the probability of the test corpus, normalized by the number of words. Smaller is better!

Results: Wall Street Journal

Model	Training data	ppl	WER
KN5 LM	200K	336	16.4
KN5 LM + RNN 90/2	200K	271	15.4
KN5 LM	1M	287	15.1
KN5 LM + RNN 90/2	1M	225	14.0
KN5 LM	6.4M	221	13.5
KN5 LM + RNN 250/5	6.4M	156	11.7

Results: Wall Street Journal

Size of the training data fixed at 6.4M.

	ppl		WER	
Model	RNN	RNN+KN	RNN	RNN+KN
KN5 - baseline	-	221	-	13.5
RNN 60/20	229	186	13.2	12.6
RNN 90/10	202	173	12.8	12.2
RNN 250/5	173	155	12.3	11.7
RNN 250/2	176	156	12.0	11.9
RNN 400/10	171	152	12.5	12.1

Results on NIST RT05

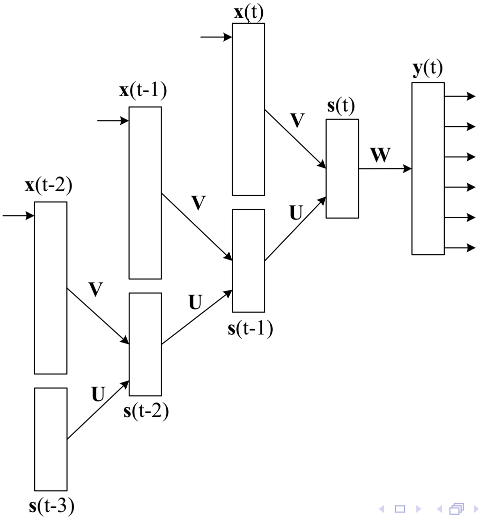
Model	WER
RT05 LM	24.5
RT09 LM - baseline	24.1
KN5 in-domain	25.7
RNN 500/10 in-domain	24.2
RNN 500/10 + RT09 LM	23.3
RNN 800/10 in-domain	24.3
RNN 800/10 + RT09 LM	23.4
RNN 1000/5 in-domain	24.2
RNN 1000/5 + RT09 LM	23.4

From Simple to Full RNNs

- Let's drop the assumption that only the hidden layer from the previous time step is used;
- instead use all previous time steps;
- we can think of this as *unfolding over time:* the RNN is unfolded into a sequence of feedforward networks;
- we need a new learning algorithm: backpropagation through time (BPTT).

Architecture

The full RNN looks at all the previous time steps:



Standard Backpropagation

For output units, we update the weights **W** using:

$$\Delta w_{kj} = \eta \sum_{p}^{n} \delta_{pk} s_{pj} \qquad \delta_{pk} = (d_{pk} - y_{pk}) g'(net_{pk})$$

where d_{pk} is the desired output of unit k for training pattern p.
For hidden units, we update the weights **V** using:

$$\Delta v_{ji} = \eta \sum_{p}^{n} \delta_{pj} x_{pi} \qquad \delta_{pj} = \sum_{k}^{o} \delta_{pk} w_{kj} f'(net_{pj})$$

This is just standard backprop, with notation adjusted for RNNs!

Going Back in Time

If we only go back one time step, then we can update the recurrent weights **U** using the standard delta rule:

$$\Delta u_{ji} = \eta \sum_{p}^{n} \delta_{pj}(t) s_{ph}(t-1) \qquad \delta_{pj}(t) = \sum_{k}^{o} \delta_{pk} w_{kj} f'(net_{pj})$$

However, if we go further back in time, then we need to apply the delta rule to the previous time step as well:

$$\delta_{pj}(t-1) = \sum_{h}^{m} \delta_{ph}(t) u_{hj} f'(s_{pj}(t-1))$$

where h is the index for the hidden unit at time step t, and j for the hidden unit at time step $t-1$.

Going Back in Time

We can do this for an arbitrary number of time steps τ, adding up the resulting deltas to compute Δu_{ji}.

The RNN effectively becomes a deep network of depth τ. For language modeling, Mikolov et al. show that increased τ improves performance.

But: if the network becomes to deep, the gradients tend towards zero: *problem of the vanishing gradients.*

More about this next lecture.

Summary

- Language models assign string probabilities;
- useful for word prediction in speech, MT, text completion;
- simple recurrent networks have one hidden layer, which is copied at each time step;
- can be trained with standard backprop;
- good performance in language modeling: provides an arbitrarily long context;
- we can also unfold an RNN over time and train it with backpropagation through time;
- turns the RNN into a deep network; even better language modeling performance.

References

Guo, Jiang. 2013. Backpropagation through time. Unpubl. ms., Harbin Institute of Technology.

Mikolov, Tomáš, Martin Karafiát, Lukáš Burget, Jan "Honza" Černocky, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Proceedings of Interspeech*. Makuhari, Chiba, Japan, pages 1045–1048.