Neural Language Models

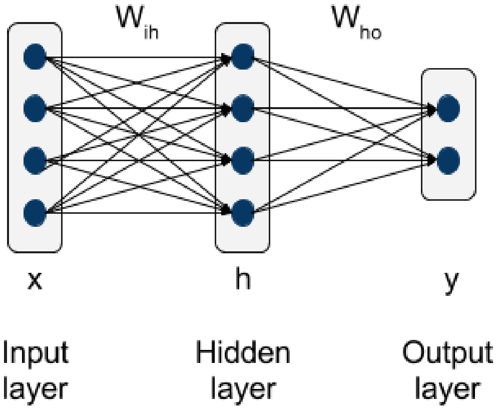
(based on Joost Bastings's slides)

Iacer Calixto

Institute for Logic, Language and Computation
University of Amsterdam

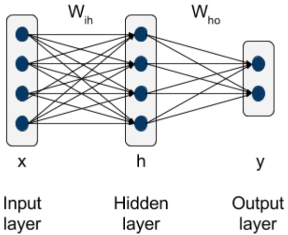
May 18, 2018

Artificial Neural Networks [1]



Let $\boldsymbol{x} \in \mathbb{R}^4, \boldsymbol{h} \in \mathbb{R}^4, \boldsymbol{y} \in \mathbb{R}^2$.

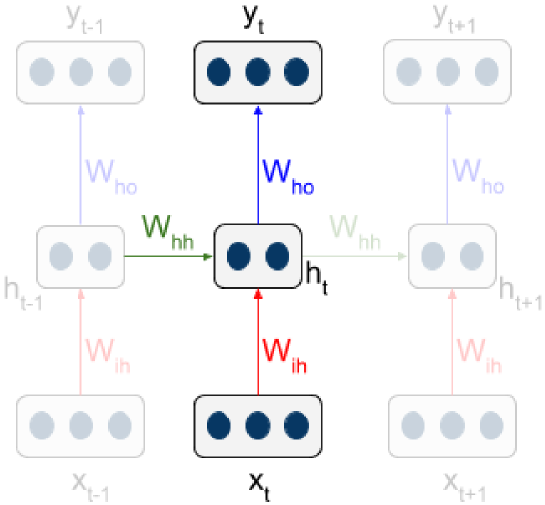
Artificial Neural Networks [2]



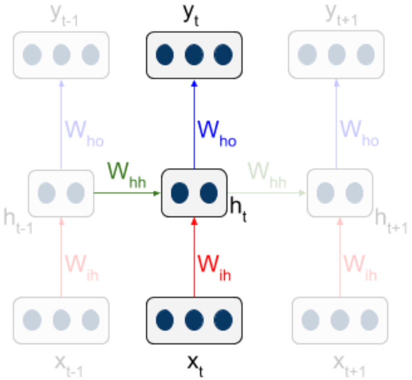
Let $\boldsymbol{x} \in \mathbb{R}^4, \boldsymbol{h} \in \mathbb{R}^4, \boldsymbol{y} \in \mathbb{R}^2$,
$\boldsymbol{W}_{ih} \in \mathbb{R}^{4\times4}$ and $\boldsymbol{b}_{ih} \in \mathbb{R}^4$, and
$\boldsymbol{W}_{ho} \in \mathbb{R}^{4\times2}$ and $\boldsymbol{b}_{ho} \in \mathbb{R}^2$.

$$\boldsymbol{h} = \mathbf{f}(\boldsymbol{x}^T \boldsymbol{W}_{ih} + \boldsymbol{b}_{ih}),$$
$$\boldsymbol{y} = \mathbf{g}(\boldsymbol{h}^T \boldsymbol{W}_{ho} + \boldsymbol{b}_{ho}).$$

Recurrent Neural Networks[1]



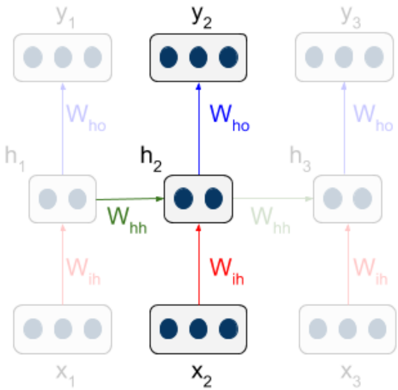
Recurrent Neural Networks[2]



$$\boldsymbol{h}_t = \mathbf{f}(\boldsymbol{W}_{ih}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{ih}),$$
$$\boldsymbol{y}_t = \mathbf{g}(\boldsymbol{W}_{ho}\boldsymbol{h}_t + \boldsymbol{b}_{ho}).$$

Recurrent Neural Networks[3]

- For a **sequence of input vectors** $x = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3\}$, an RNN will compute a **sequence of hidden states** $H = \{\boldsymbol{h}_1, \boldsymbol{h}_2, \boldsymbol{h}_3\}$, and optionally a **sequence of output vectors** $y = \{\boldsymbol{y}_1, \boldsymbol{y}_2, \boldsymbol{y}_3\}$.



you know nothing, jon ______

ground control to major ______

the ______

the quick ______

the quick brown ______

the quick brown fox ______

the quick brown fox jumps ______

the quick brown fox jumps over ______

the quick brown fox jumps over the ______

the quick brown fox jumps over the lazy ______ the quick brown jumps over the lazy dog

Definition

- Language models give us the **probability of a sentence**;
- At any time step, they assign a **probability** to the **next word**.

Applications

- Very useful in a plethora of different tasks:
  - Speech recognition;
  - Spelling correction;
  - Machine translation;
  - etc.
- LMs are useful in almost any tasks that deals with **generating language**.

Different “Types” of Language Models

- N-gram based LMs;
- Log-linear LMs;
- Neural LMs.

N-gram LM[1]

x is a sequence of words

$$\begin{aligned} x &= \{x_1, x_2, x_3, x_4, x_5\} \\ &= \{\text{you}, \text{know}, \text{nothing}, \text{jon}, \text{snow}\} \end{aligned}$$

N-gram LM[2]

To compute the probability of that sentence, we first apply the **chain rule**:

$$P(x_1, x_2, \cdots, x_n) = \prod_i P(x_i | x_1, x_2, x_{i-1})$$

$$\begin{aligned} P(x) =& P(\text{"you know nothing jon snow"}) \\ =& P(\text{"you"}) \cdot \\ & P(\text{"know"} \mid \text{"you"}) \cdot \\ & P(\text{"nothing"} \mid \text{"you know"}) \cdot \\ & P(\text{"jon"} \mid \text{"you know nothing"}) \cdot \\ & P(\text{"snow"} \mid \text{"you know nothing jon"}). \end{aligned}$$

N-gram LM[3]

We then make a Markov assumption of **conditional independence**:

$$
\begin{aligned}
P(x_1, x_2, \cdots, x_n) &= \prod_i P(x_i | x_1, x_2, x_{i-1}) \\
&= \prod_i P(x_i | x_{i-1})
\end{aligned}
$$

$$
\begin{aligned}
P(x) &= P(\text{"you know nothing jon snow"}) \\
&= P(\text{"you know"}) \cdot P(\text{"know nothing"}) \cdot P(\text{"nothing jon"}) \cdot P(\text{"jon sno}
\end{aligned}
$$

N-gram LM[4]

If we didn't observe a certain bigram, then $p(x_i|x_{i-1})$ will be 0.
This makes the probability of the sentence also 0!
MLE:

$$P_{\text{MLE}}(x_i|x_{i-1}) = \frac{\text{count}(x_{i-1}, x_i)}{\text{count}(x_{i-1})}$$

Laplace / add-one smoothing :

$$P_{\text{add1}}(x_i|x_{i-1}) = \frac{\text{count}(x_{i-1}, x_i) + 1}{\text{count}(x_{i-1}) + V}.$$

This doesn't work too well for language modelling.
However, there are more advanced smoothing that could be applied e.g. Kneser-Ney (Kneser and Ney, 1995).

Log-linear LM

$$P_w(Y = y|X = x) = \frac{\exp \boldsymbol{w} \cdot \phi(x, y)}{\sum_{y' \in V_y} \exp \boldsymbol{w} \cdot \phi(x, y')}$$

- Y is the next word and V_y is the vocabulary;
- X is the history;
- ϕ is a feature function that returns an n-dimensional vector;
- $\boldsymbol{w}$ are the model parameters.

Why use a log-linear LM?

- With features of words and histories we can share statistical weight;
- With n-grams, there is no sharing at all!
- We also get smoothing for free; ☺
- We can add arbitrary features!
- We use Stochastic Gradient Descent (SGD) to optimise.

Which features to use?

- n-gram features: "$X_{j-1} = \text{the}$ and $X_j = \text{puppy}$";
- "gappy" n-gram features: "$X_{j-2} = \text{the}$ and $X_j = \text{puppy}$";
- spelling features: "X_j's first letter is capitalised";
- class features: "X_j's belongs to class `ABC`";
- gazetteer features: "X_j is a place name";
- etc.

Neural Language Models - Motivation

- n-gram language models have proven to be effective in various tasks ✓
- log-linear models allow us to share weights through features ✓
- maybe our history is still too limited, e.g. $n-1$ words ✗
- we need to find useful features ✗

Feed-forward Neural Networks

With neural networks we can exploit distributed representations to allow for statistical weight sharing.

How does it work:

1. each word is mapped to an embedding: an m-dimensional feature vector;
2. a probability function over word sequences is expressed in terms of these vectors;
3. we jointly learn the feature vectors and the parameters of the probability function.

How/Why does it work?

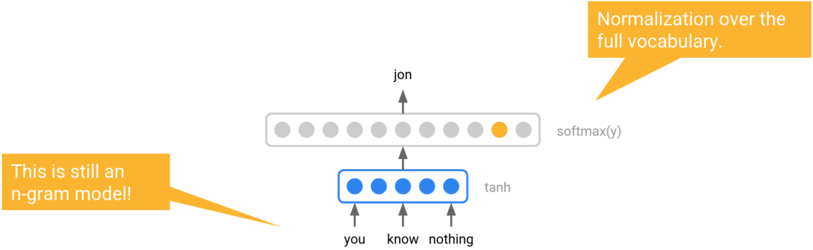
✓ Similar words are expected to have similar feature vectors:
(dog,cat), (running,walking), (bedroom,room)

With this, probability mass is naturally transferred from (1) to (2):
The cat is walking in the bedroom.
The dog is running in the room.
And many other similar sentences...

Take-away message:

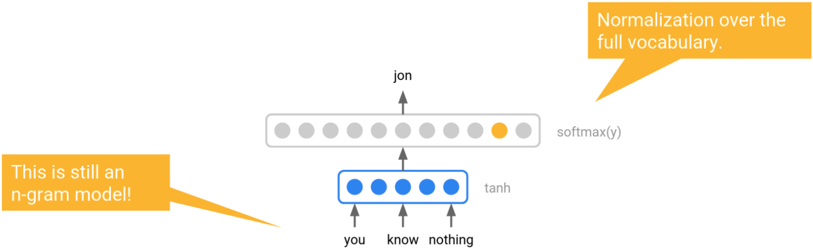
- The presence of only one sentence in the training data will increase the probability of a combinatorial number of "neighbours" in sentence space.

Feed-forward LM



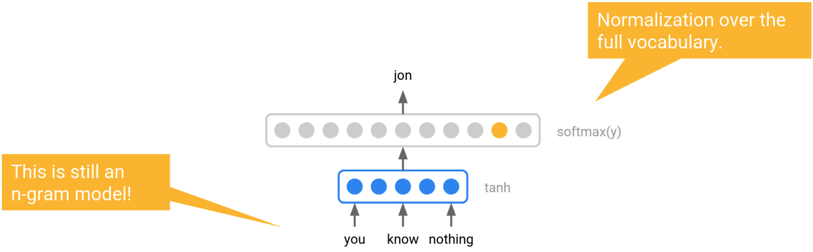
$$
\begin{aligned}
\boldsymbol{E}_{\text{you}}, \boldsymbol{E}_{\text{know}}, \boldsymbol{E}_{\text{nothing}} &\in \mathbb{R}^{100}, \\
\boldsymbol{x} = [\boldsymbol{E}_{\text{you}}; \boldsymbol{E}_{\text{know}}; \boldsymbol{E}_{\text{nothing}}] &\in \mathbb{R}^{300}, \\
\boldsymbol{y} = \boldsymbol{W}_3 \tanh(\boldsymbol{W}_1 x + \boldsymbol{b}_1) &+ \boldsymbol{W}_2 \boldsymbol{x} + \boldsymbol{b}_2.
\end{aligned}
$$

Why does it work?



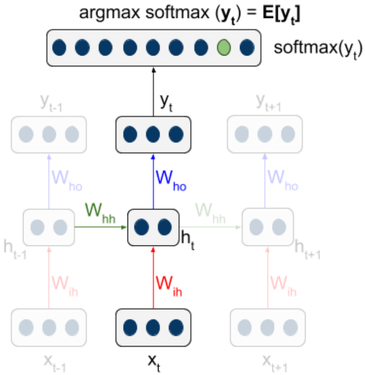
- The non-linear activation functions perform feature combinations that a linear model cannot do;
- End-to-end training on next word prediction.

Continuation...



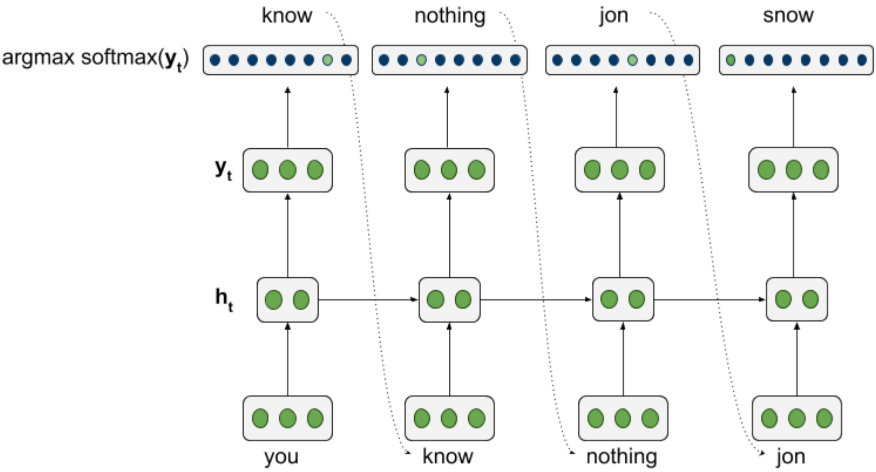
- We now have much better generalisation, but still a limited history/context.
- Recurrent neural networks have unlimited history! ☺

Recurrent Neural Network Language Model



$$\boldsymbol{h}_t = \mathrm{f}(\boldsymbol{W}_{ih}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{ih}),$$
$$\boldsymbol{y}_t = \mathrm{g}(\boldsymbol{W}_{ho}\boldsymbol{h}_t + \boldsymbol{b}_{ho}).$$

Recurrent Neural Network Language Model

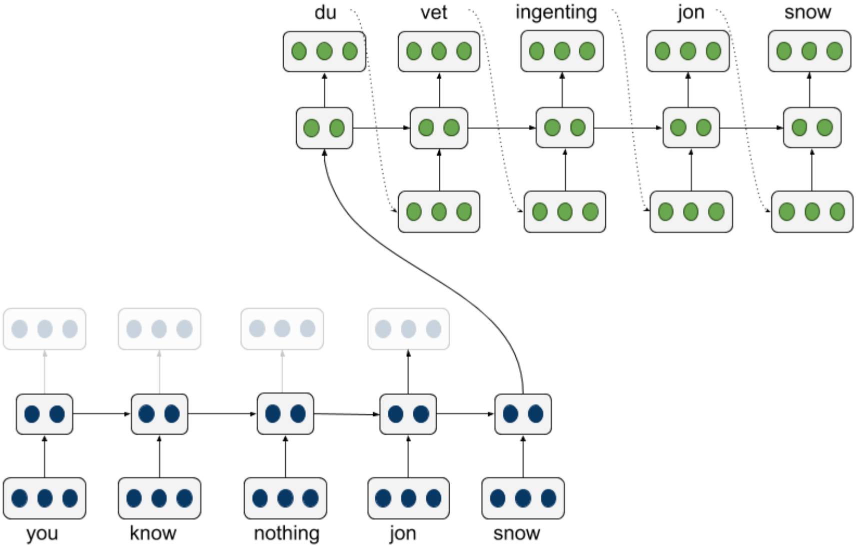


Recurrent Neural Network Language Model

Final notes on neural LMs:

- RNNs suffer from the vanishing gradient problem;
- Many improvements have been proposed insofar:
  - LSTM-based LMs;
  - character-based LMs,
  - etc.

Teaser — Encoder-Decoder



References

Bengio et al. (2003). A neural probabilistic language model. Journal of Machine Learning Research. 3 Feb (2003):1137–1155.

Cho et al. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arxiv:1406.1078 (2014).

Kneser and Ney (1995). Improved Backing-off for N-gram Language Modeling. ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings. 1. 181-184 vol.1. 10.1109/ICASSP.1995.479394.

Mikolov et al., (2010). Recurrent Neural Network-Based Language Model. Interspeech, vol.2, 2010.