Natural Language Understanding, Generation, and Machine Translation

Lecture 6: Recurrent Neural Networks

Frank Keller

Week of 24 January 2022 (week 2)

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

Overview

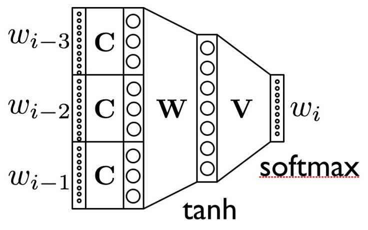
Recap: neural networks, probability, and language models

Recurrent networks for language modeling

- From Feedforward to Recurrent Networks
- Recurrent Networks as Recursive Functions
- Backpropagation through Time
- Vanishing Gradients

Reading: Section 6 Neubig (2017), Guo (2013).

Agenda for Today



Last time we saw that vector-to-vector functions, aka neural networks, can be used to learn *n*-gram probabilities, with some nice side benefits: parameter sharing, word representations, and no zero probabilities in the learned model.

This time we'll see how neural networks can also relieve us from having to deal with a major difficulty in the design of classical probabilistic models: making independence assumptions.

Recap: neural networks, probability, and language models

Recipe for Deep Learning in NLP

1. Design a model that matches the input/output of your training examples.
2. Decide an objective function. For probabilistic models: cross-entropy loss.
3. Choose a learning algorithm: some variant of stochastic gradient descent.
4. Compute gradients of loss w.r.t. model parameters.

Item 2 and 3 are automated for you in modern libraries, so you can focus on 1. This is a design problem! You need to think carefully about input and output, and **most importantly** about your data.

Most models in NLP are probabilistic models

E.g. language model decomposed with chain rule of probability. If $w = w_1 \dots w_{|w|} \in V*$, then:

$$P(w_1 \dots w_{|w|}) = \prod_{i=1}^{|w|+1} P(w_i \mid w_1, \dots, w_{i-1})$$

Modeling decision: Markov assumption

$$P(w_i \mid w_1, \dots, w_{i-1}) \sim P(w_i \mid w_{i-n+1}, \dots, w_{i-1})$$

Most models in NLP are probabilistic models

E.g. language model decomposed with chain rule of probability. If $w = w_1 \dots w_{|w|} \in V*$, then:

$$P(w_1 \dots w_{|w|}) = \prod_{i=1}^{|w|+1} P(w_i \mid w_1, \dots, w_{i-1})$$

Modeling decision: Markov assumption

$$P(w_i \mid w_1, \dots, w_{i-1}) \sim P(w_i \mid w_{i-n+1}, \dots, w_{i-1})$$

Goal for today: remove this assumption!

The real world data input is various in length, Markov method need to fix the input length, which may cause sparse input or cannot have ability to deal with the long dependency (large context).

Most models in NLP are probabilistic models

E.g. language model decomposed with chain rule of probability. If $w = w_1 \dots w_{|w|} \in V*$, then:

$$P(w_1 \dots w_{|w|}) = \prod_{i=1}^{|w|+1} P(w_i \mid w_1, \dots, w_{i-1})$$

Modeling decision: Markov assumption

$$P(w_i \mid w_1, \dots, w_{i-1}) \sim P(w_i \mid w_{i-n+1}, \dots, w_{i-1})$$

Goal for today: remove this assumption!

Must still observe rules of probability:

Probabilities are non-negative $\quad P : V \to \mathbb{R}_+$

. . . and sum to one $\quad \sum_{w \in V} P(w \mid w_1, \dots, w_{i-1}) = 1$

Markov assumptions are wrong for language

The **roses** <u>are</u> red.

Markov assumptions are wrong for language

The **roses** <u>are</u> red.

The **roses** in the vase <u>are</u> red.

Markov assumptions are wrong for language

The **roses** <u>are</u> red.

The **roses** in the vase <u>are</u> red.

The **roses** in the vase by the door <u>are</u> red.

Markov assumptions are wrong for language

The **roses** <u>are</u> red.

The **roses** in the vase <u>are</u> red.

The **roses** in the vase by the door <u>are</u> red.

The **roses** in the vase by the door to the kitchen <u>are</u> red.

Markov assumptions are wrong for language

The **roses** <u>are</u> red.

The **roses** in the vase <u>are</u> red.

The **roses** in the vase by the door <u>are</u> red.

The **roses** in the vase by the door to the kitchen <u>are</u> red.

Captain Ahab nursed his grudge for many years before seeking the White ____

Markov assumptions are wrong for language

The **roses** <u>are</u> red.

The **roses** in the vase <u>are</u> red.

The **roses** in the vase by the door <u>are</u> red.

The **roses** in the vase by the door to the kitchen <u>are</u> red.

Captain Ahab nursed his grudge for many years before seeking the White ____

Donald Trump nursed his grudge for many years before seeking the White ____

whale, house

Recurrent networks for language modeling

We need to model arbitrary context. How?

Context is important in language modeling:

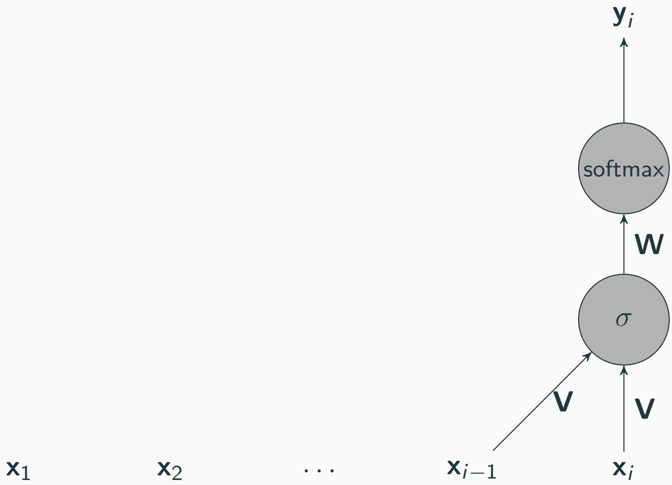
- But *n*-gram language models use a fixed context window.
- Feedforward networks also use a fixed context window...
- but linguistic dependencies can be arbitrarily long!

This is where *recurrent neural networks* come in!

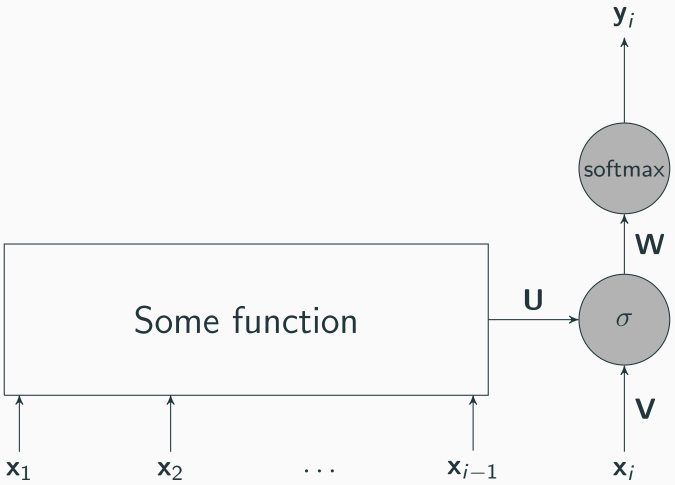
Glossary

- $\mathbf{x}_i$: the input word transformed into one hot encoding
- $\mathbf{y}_i$: the output probability distribution
- $\mathbf{U}$: the weight matrix of the recurrent layer
- $\mathbf{V}$: the weight matrix between the input layer and the hidden layer
- $\mathbf{W}$: is the weight matrix between the hidden layer and the output layer
- σ: the sigmoid activation function
- $\mathbf{h}$: the hidden layer

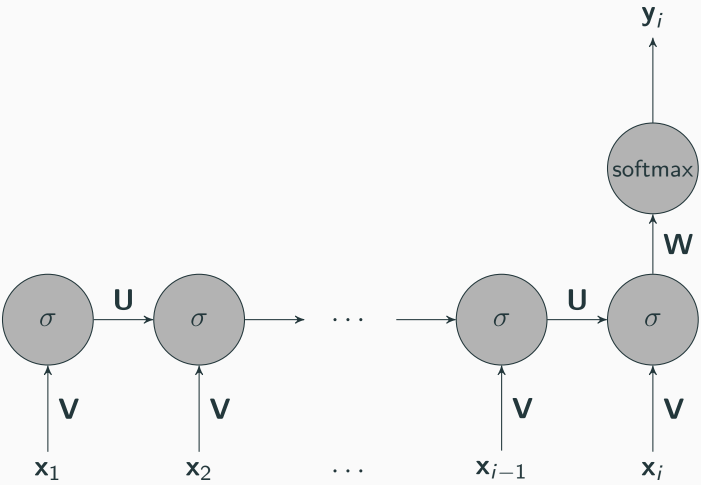
From feedforward to recurrent neural networks



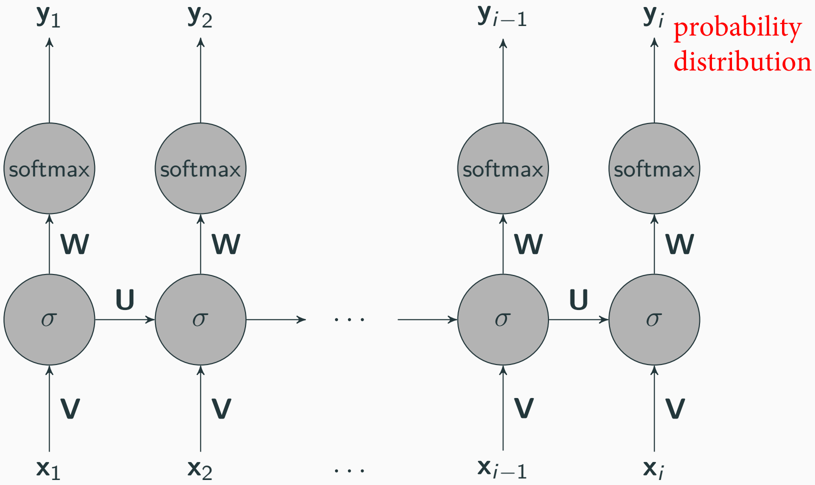
From feedforward to recurrent neural networks



From feedforward to recurrent neural networks



From feedforward to recurrent neural networks



View this complete structure as a *computation graph.*

Recurrent neural networks are simply recursive functions

$$
\begin{aligned}
P(x_{i+1} \mid x_1, \ldots, x_i) = \mathbf{y}_i \\
\mathbf{y}_i &= \text{softmax}(\mathbf{W}\mathbf{h}_i + \mathbf{b}_2) \\
\mathbf{h}_i &= \sigma(\mathbf{V}\mathbf{x}_i + \mathbf{U}\mathbf{h}_{i-1} + \mathbf{b}_1) \\
\mathbf{x}_i &= \text{onehot}(x_i)
\end{aligned}
$$

Now P is a function of x_i, and, recursively through h_i, all of $x_1, \ldots, x_{i-1}$. So it computes $P(x_{i+1} \mid x_1, \ldots, x_{i-1})$ *with no Markov assumption!*

Recurrent neural networks are simply recursive functions

$$
\begin{aligned}
P(x_{i+1} \mid x_1, \ldots, x_i) &= \mathbf{y}_i \\
\mathbf{y}_i &= \text{softmax}(\mathbf{W}\mathbf{h}_i + \mathbf{b}_2) \\
\mathbf{h}_i &= \sigma(\mathbf{V}\mathbf{x}_i + \mathbf{U}\mathbf{h}_{i-1} + \mathbf{b}_1) \\
\mathbf{x}_i &= \text{onehot}(x_i)
\end{aligned}
$$

Now P is a function of x_i, and, recursively through h_i, all of $x_1, \ldots, x_{i-1}$. So it computes $P(x_{i+1} \mid x_1, \ldots, x_{i-1})$ *with no Markov assumption!*

For simplicity, your coursework leaves out $\mathbf{b}_1$ and $\mathbf{b}_2$.

Training

Use stochastic gradient descent, with cross-entropy.

Training

Use stochastic gradient descent, with cross-entropy.

Unlike in a feedforward network, the *structure* of the computation is *dynamic:* $\mathbf{y}_i$ is computed via more nodes than $\mathbf{y}_{i-1}$.

Training

Use stochastic gradient descent, with cross-entropy.

Unlike in a feedforward network, the *structure* of the computation is *dynamic:* $\mathbf{y}_i$ is computed via more nodes than $\mathbf{y}_{i-1}$.

Computing gradients by hand is tedious and error-prone. Most toolkits do this for you via *automatic differentiation,* which computes the gradient in two steps: by computing the current output from the inputs in a *forward pass* on the *computation graph,* and then computing gradients via *backpropagation* from the error (at the output) back to the inputs.

Training

Use stochastic gradient descent, with cross-entropy.

Unlike in a feedforward network, the *structure* of the computation is *dynamic:* $\mathbf{y}_i$ is computed via more nodes than $\mathbf{y}_{i-1}$.

Computing gradients by hand is tedious and error-prone. Most toolkits do this for you via *automatic differentiation,* which computes the gradient in two steps: by computing the current output from the inputs in a *forward pass* on the *computation graph,* and then computing gradients via *backpropagation* from the error (at the output) back to the inputs.

Backpropagation uses the chain rule of derivatives, applied via dynamic programming on the computation graph.

Forward Propagation computes the output

We have input layer $\mathbf{x}$, hidden layer $\mathbf{h}$, output layer $\mathbf{y}$. The input at time t is $\mathbf{x}(t)$, output is $\mathbf{y}(t)$, and hidden layer $\mathbf{h}(t)$.

$$\begin{aligned} y_k(t) &= g(net_k(t)) && (1) \\ net_k(t) &= \sum_j^m h_j(t) w_{kj} && (2) \\ h_j(t) &= f(net_j(t)) && (3) \\ net_j(t) &= \sum_i^l x_i(t) v_{ji} && (4) \end{aligned}$$

where $f(z) = \sigma(z)$, and $g(z) = \text{softmax}(z)$.

Forward Propagation computes the output

We have input layer **x**, hidden layer **h**, output layer **y**. The input at time t is $\mathbf{x}(t)$, output is $\mathbf{y}(t)$, and hidden layer $\mathbf{h}(t)$.

$$
\begin{aligned}
y_k(t) &= g(net_k(t)) && (1)\\
net_k(t) &= \sum_j^m h_j(t) w_{kj} && (2)\\
h_j(t) &= f(net_j(t)) && (3)\\
net_j(t) &= \sum_i^l x_i(t) v_{ji} + \sum_h^m h_h(t-1) u_{jh} && (4)
\end{aligned}
$$

where $f(z) = \sigma(z)$, and $g(z) = \text{softmax}(z)$.

So far this was a standard feedforward network.
Now we add the recurrence.

Backpropagation computes the gradient

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

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

where d_{pk} is the desired output of unit k for training pattern p.

For hidden units, we update the weights **V** using:

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

So far, this is just standard backpropagation!

Backpropagation computes the gradient

At the current time step, we accumulate an update to the recurrent weights **U** using the standard delta rule:

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

adjustment

We backpropagate error through time, applying the delta rule to the previous time step as well:

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

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

Backpropagation computes "back through 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. In theory, τ can (and should) be arbitrarily large.

In practice, it can be set to a small value. This is properly called **truncated backpropagation through time**—what you will implement in the coursework!

As we backpropagate through time, gradients tend toward 0

We adjust $\mathbf{U}$ using backprop through time. For timestep t:

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

For timestep $t-1$:

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

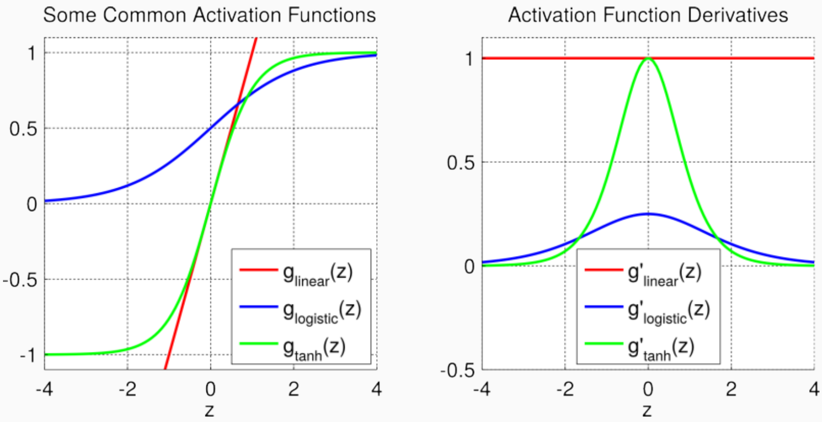
For time step $t-2$:

$$\begin{aligned} \delta_{pj}(t-2) &= \sum_{h}^{m} \delta_{ph}(t-1) u_{hj} f'(h_{pj}(t-2)) \\ &= \sum_{h}^{m} \sum_{h_1}^{m} \delta_{ph_1}(t) u_{h_1 j} f'(h_{pj}(t-1)) u_{hj} f'(h_{pj}(t-2)) \end{aligned}$$

As we backpropagate through time, gradients tend toward 0

At every time step, we multiply the weights with another gradient.

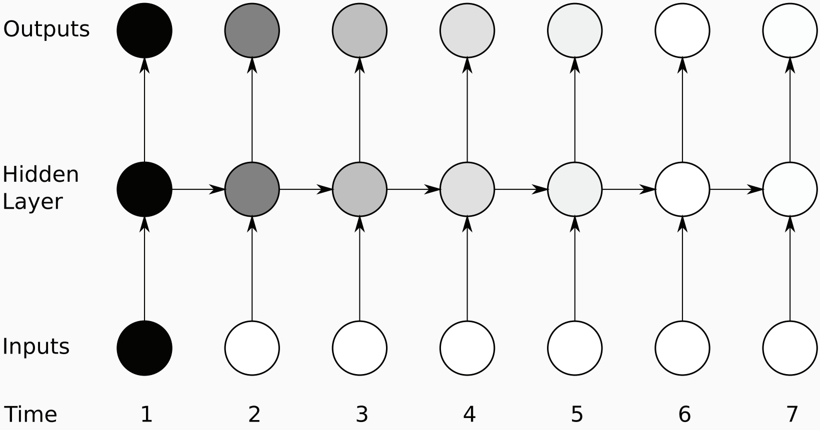
The gradients are < 1 so the deltas become smaller and smaller.



[Source: https://theclevermachine.wordpress.com/]

As we backpropagate through time, gradients tend toward 0

So in fact, the RNN is not able to learn long-range dependencies well, as the gradient vanishes: it rapidly "forgets" previous inputs:



[Source: Graves (2012).]

Summary

- 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 unfold an RNN over time and train it with backpropagation through time;
- effectively turns the RNN into a deep network;
- but: *vanishing gradients* as we propagate through time.

In the next lectures, we will look at Long Short-term Memory networks, which help with vanishing gradients.

References

Graves, Alex. 2012. *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer, Berlin.

Guo, Jiang. 2013. Backpropagation through time. Unpubl. ms., Harbin Institute of Technology, http://ir.hit.edu.cn/~jguo/docs/notes/bptt.pdf.

Neubig, Graham. 2017. Neural machine translation and sequence-to-sequence models: A tutorial. ArXiv:1703.01619.