Recurrent Neural Networks

Jian Tang

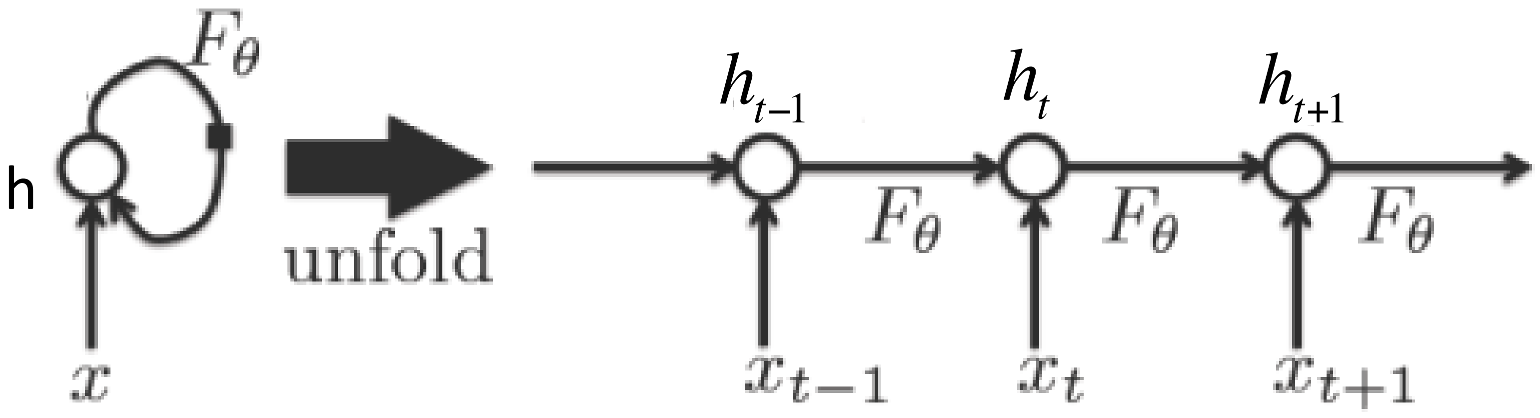
tangjianpku@gmail.com

HEC MONTRĒAL

Institut des algorithmes d'apprentissage de Montréal MILA

RNN: Recurrent neural networks

- Neural networks for sequence modeling
  - Summarize a sequence with fix-sized vector through recursively updating

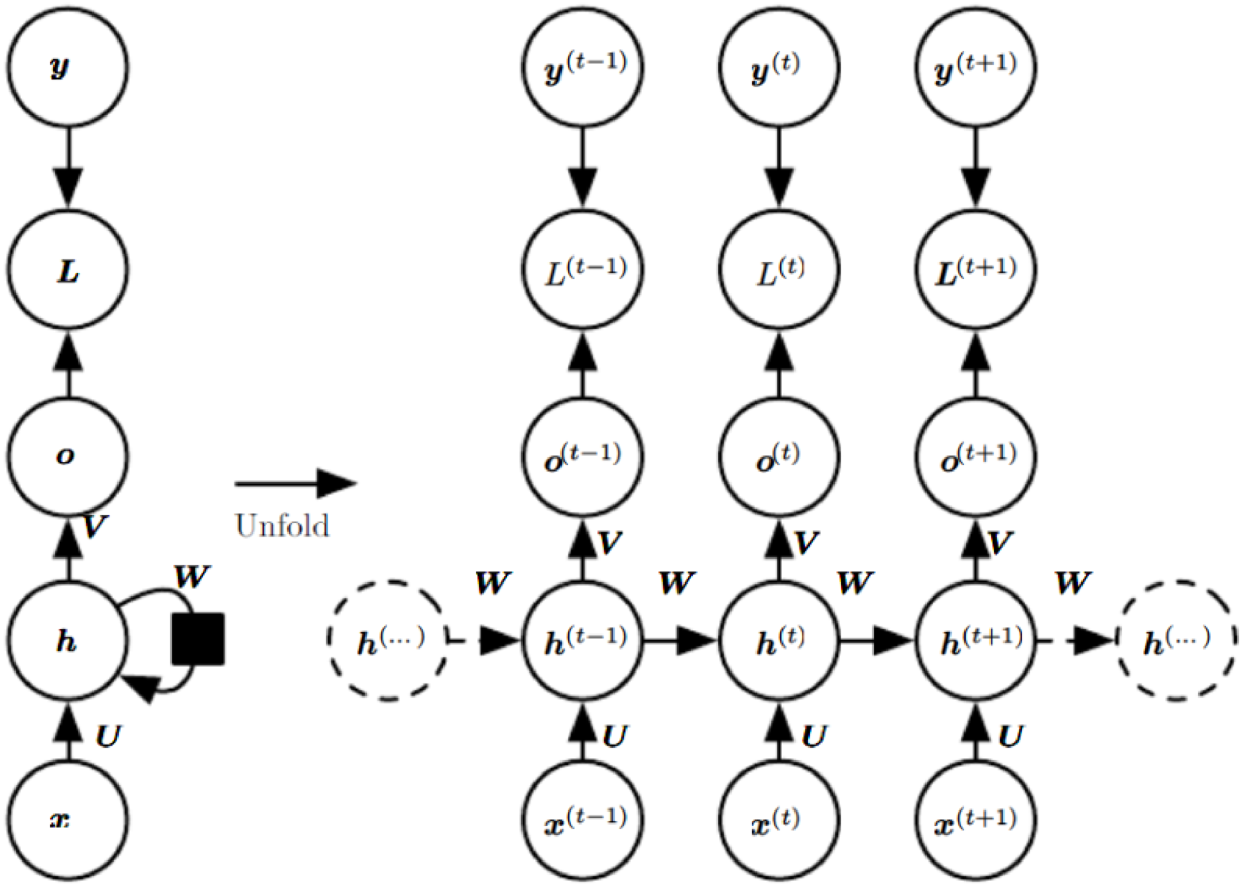


$$h_t = F_\theta(h_{t-1}, x_t)$$

$$h_t = G_t(x_t, x_{t-1}, x_{t-2}, \ldots, x_2, x_1)$$

Recurrent Neural Networks

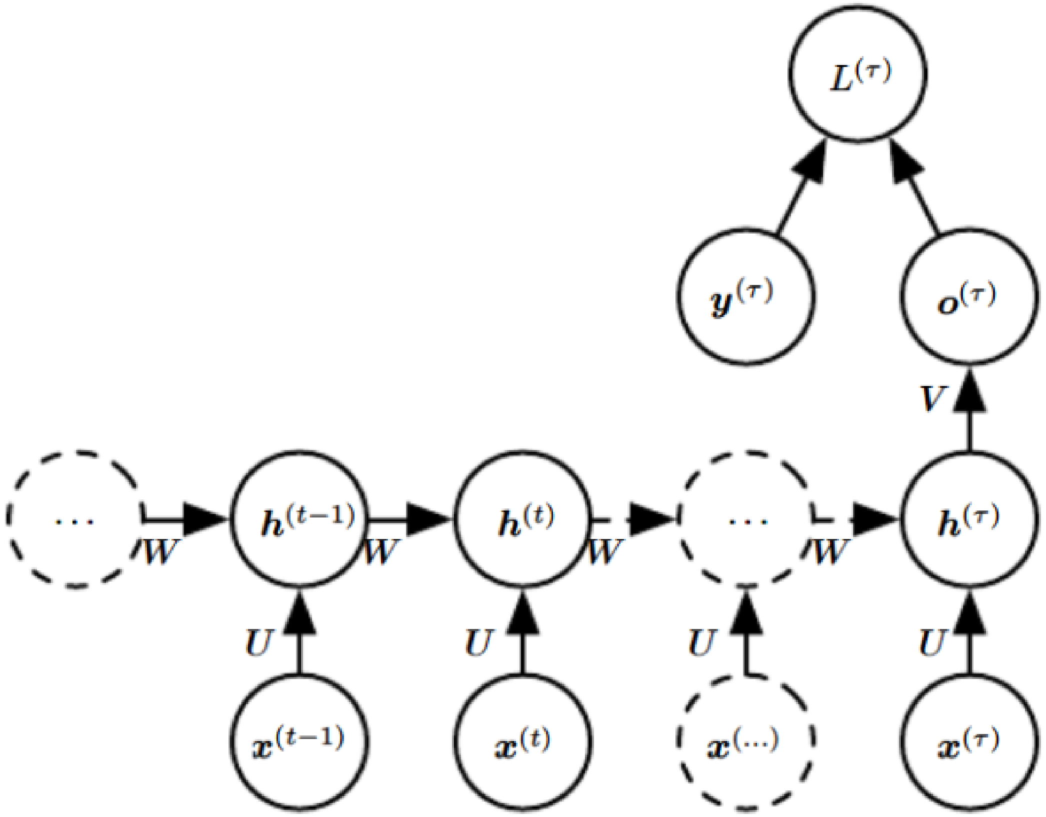
- Can produce an output at each time step: unfolding the graph tell us how to back-prop through time



$$h_t = \tanh(Wh_{t-1} + Ux_t)$$

Recurrent Neural Networks

- Produce a single output at the end of sequence



$$h_t = \tanh(Wh_{t-1} + Ux_t)$$

Language Modeling

A language model computes a probability for a sequence of words: $P(w_1, \ldots, w_T)$

- Useful for machine translation
  - Word ordering:
    p(the cat is small) > p(small the is cat)
  - Word choice:
    p(walking home after school) > p(walking house after school)

RNN for Language Modeling

- Estimate the probability of a sequence $x_1, \ldots, x_{t-1}, x_t, x_{t+1}, \ldots, x_T$

$$P(\mathbf{x}) = P(x_1, \ldots x_T) = \prod_{t=1}^{T} P(x_t | x_{t-1}, x_{t-2}, \ldots x_1)$$

At a single time step:

$$\begin{aligned} h_t &= \sigma\left(W^{(hh)} h_{t-1} + W^{(hx)} x_{[t]}\right) \\ \hat{y}_t &= \mathrm{softmax}\left(W^{(S)} h_t\right) \\ \hat{P}(x_{t+1} = v_j \mid x_t, \ldots, x_1) &= \hat{y}_{t,j} \end{aligned}$$

RNN for Language Modeling

Main idea: we use the same set of W weights at all time steps!

Everything else is the same:

$$\begin{aligned} h_t &= \sigma\left(W^{(hh)} h_{t-1} + W^{(hx)} x_{[t]}\right) \\ \hat{y}_t &= \operatorname{softmax}\left(W^{(S)} h_t\right) \\ \hat{P}(x_{t+1} = v_j \mid x_t, \ldots, x_1) &= \hat{y}_{t,j} \end{aligned}$$

$h_0 \in \mathbb{R}^{D_h}$ is some initialization vector for the hidden layer at time step 0

$x_{[t]}$ is the column vector of L at index [t] at time step t

$W^{(hh)} \in \mathbb{R}^{D_h \times D_h}$ $W^{(hx)} \in \mathbb{R}^{D_h \times d}$ $W^{(S)} \in \mathbb{R}^{|V| \times D_h}$

RNN for Language Modeling

$\hat{y} \in \mathbb{R}^{|V|}$ is a probability distribution over the vocabulary

Same cross entropy loss function but predicting words instead of classes

$$J^{(t)}(\theta) = -\sum_{j=1}^{|V|} y_{t,j} \log \hat{y}_{t,j}$$

RNN for Language Modeling

Evaluation could just be negative of average log probability over dataset of size (number of words) T:

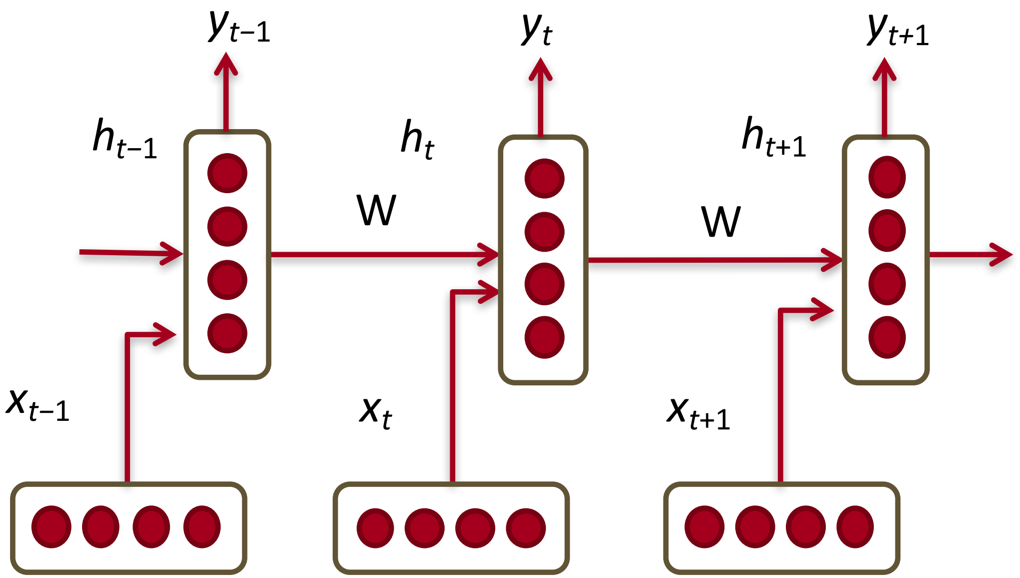
$$J = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{|V|} y_{t,j} \log \hat{y}_{t,j}$$

But more common: Perplexity: 2^J

Lower is better!

Training RNN is very Hard

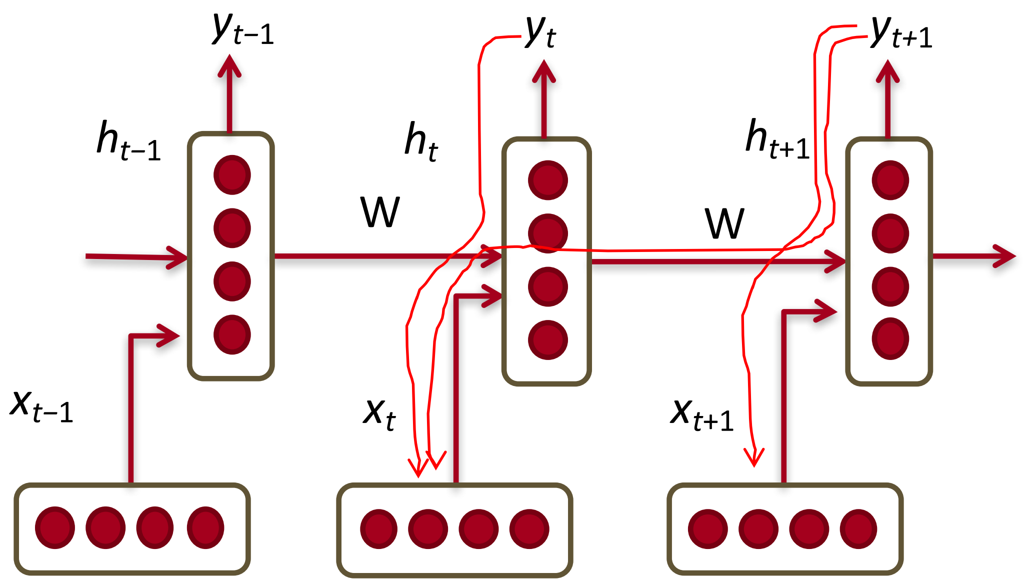
- Multiply the same matrix at each time step during forward prop



- Ideally inputs from many time steps ago can modify output y
- Take $\frac{\partial E_2}{\partial W}$ for an example RNN with 2 time steps! Insightful!

Gradient Vanishing/Exploding

Multiply the same matrix at each time step during backprop



Details

- Similar but simpler RNN formulation:

$$\begin{aligned} h_t &= W f(h_{t-1}) + W^{(hx)} x_{[t]} \\ \hat{y}_t &= W^{(S)} f(h_t) \end{aligned}$$

- Total error is the sum of each error at time steps t

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W}$$

- Hardcore chain rule application:

$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}$$

Details

- Similar to backprop but less efficient formulation
- Useful for analysis we'll look at:

$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \boxed{\frac{\partial h_t}{\partial h_k}} \frac{\partial h_k}{\partial W}$$

- Remember: $h_t = W f(h_{t-1}) + W^{(hx)} x_{[t]}$
- More chain rule, remember:

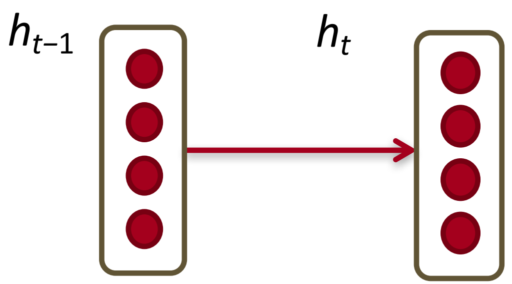
$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}$$

- Each partial is a Jacobian:

$$\frac{d\mathbf{f}}{d\mathbf{x}} = \left[\frac{\partial \mathbf{f}}{\partial x_1} \cdots \frac{\partial \mathbf{f}}{\partial x_n} \right] = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Details

- From previous slide: $\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}$



- Remember: $h_t = W f(h_{t-1}) + W^{(hx)} x_{[t]}$

- To compute Jacobian, derive each element of matrix: $\frac{\partial h_{j,m}}{\partial h_{j-1,n}}$

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{t} W^T \mathrm{diag}[f'(h_{j-1})]$$

- Where: $\mathrm{diag}(z) = \begin{pmatrix} z_1 & & & 0 & \\ & z_2 & & & \\ & & \ddots & & \\ & 0 & & z_{n-1} & \\ & & & & z_n \end{pmatrix}$

Check at home that you understand the diag matrix formulation

Details

- Analyzing the norms of the Jacobians, yields:

$$\left\| \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq \|W^T\| \|\mathrm{diag}[f'(h_{j-1})]\| \leq \beta_W \beta_h$$

- Where we defined ¯‘s as upper bounds of the norms
- The gradient is a product of Jacobian matrices, each associated with a step in the forward computation.

$$\left\| \frac{\partial h_t}{\partial h_k} \right\| = \left\| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq (\beta_W \beta_h)^{t-k}$$

- This can become very small or very large quickly [Bengio et al 1994], and the locality assumption of gradient descent breaks down. → **Vanishing or exploding gradient**

Long-short Term Memory (LSTM)

- From *multiplication* to *summation*
  - Input gate (current cell matters) $i_t = \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1}\right)$
  - Forget (gate 0, forget past) $f_t = \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1}\right)$
  - Output (how much cell is exposed) $o_t = \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1}\right)$
  - New memory cell $\tilde{c}_t = \tanh\left(W^{(c)}x_t + U^{(c)}h_{t-1}\right)$

Final memory cell: $c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$

Final hidden state: $h_t = o_t \circ \tanh(c_t)$

Gated Recurrent Unit (GRU, Cho et al. 2014)

Update gate $$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$

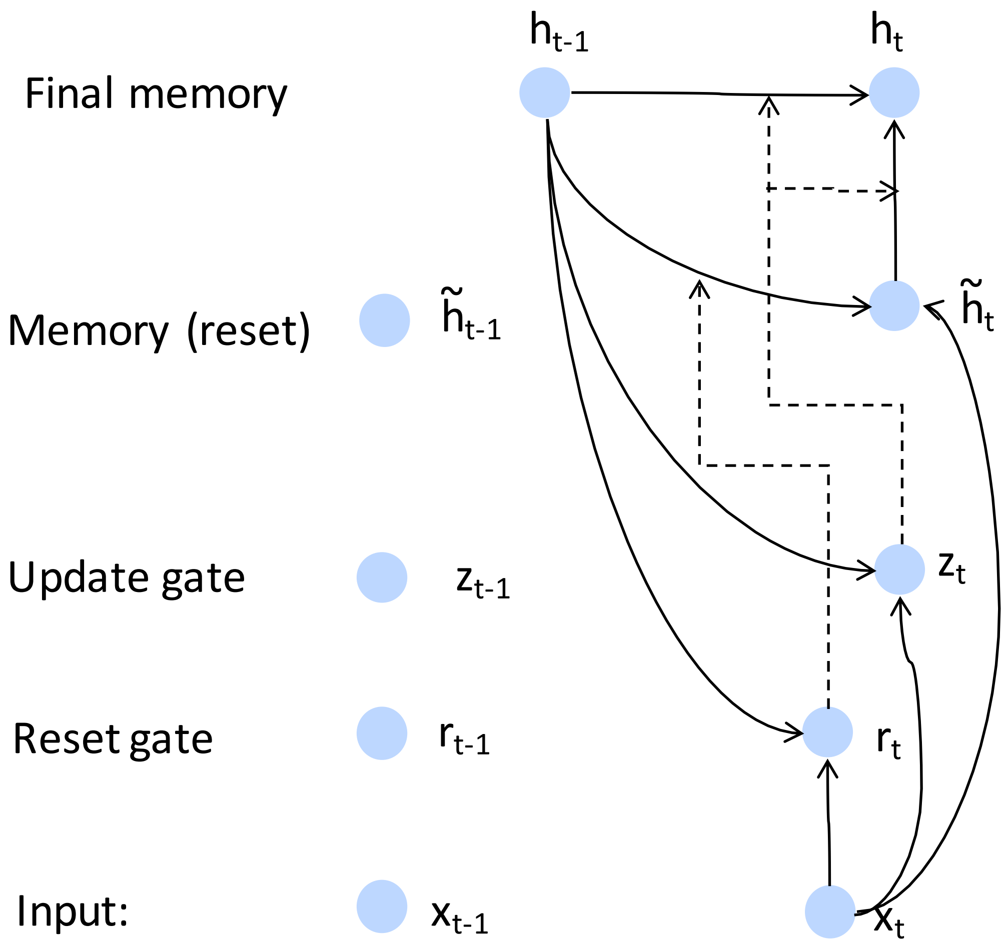
Reset gate $$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$

New memory content: $$\tilde{h}_t = \tanh\left(Wx_t + r_t \circ Uh_{t-1}\right)$$

If reset gate unit is ~0, then this ignores previous memory and only stores the new word information

Final memory at time step combines current and previous time steps: $$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$$

Gated Recurrent Unit (GRU, Cho et al. 2014)



$$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$

$$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$

$$\tilde{h}_t = \tanh\left(Wx_t + r_t \circ Uh_{t-1}\right)$$

$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$$

Gated Recurrent Unit (GRU, Cho et al. 2014)

If reset is close to 0,
ignore previous hidden state
→ Allows model to drop information that is irrelevant in the future

$$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$

$$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$

$$\tilde{h}_t = \tanh\left(Wx_t + r_t \circ Uh_{t-1}\right)$$

$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$$

Update gate z controls how much of past state should matter now.

- If z close to 1, then we can copy information in that unit through many time steps! **Less vanishing gradient**!

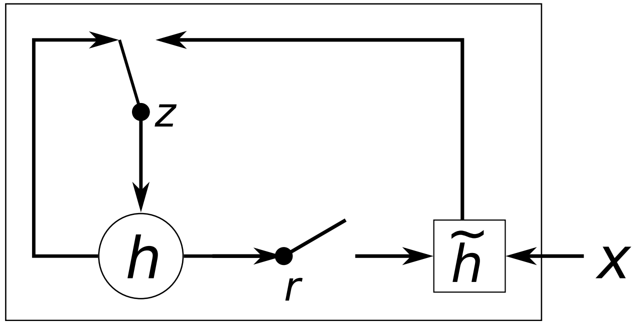
Units with short-term dependencies often have reset gates very active

Gated Recurrent Unit (GRU, Cho et al. 2014)

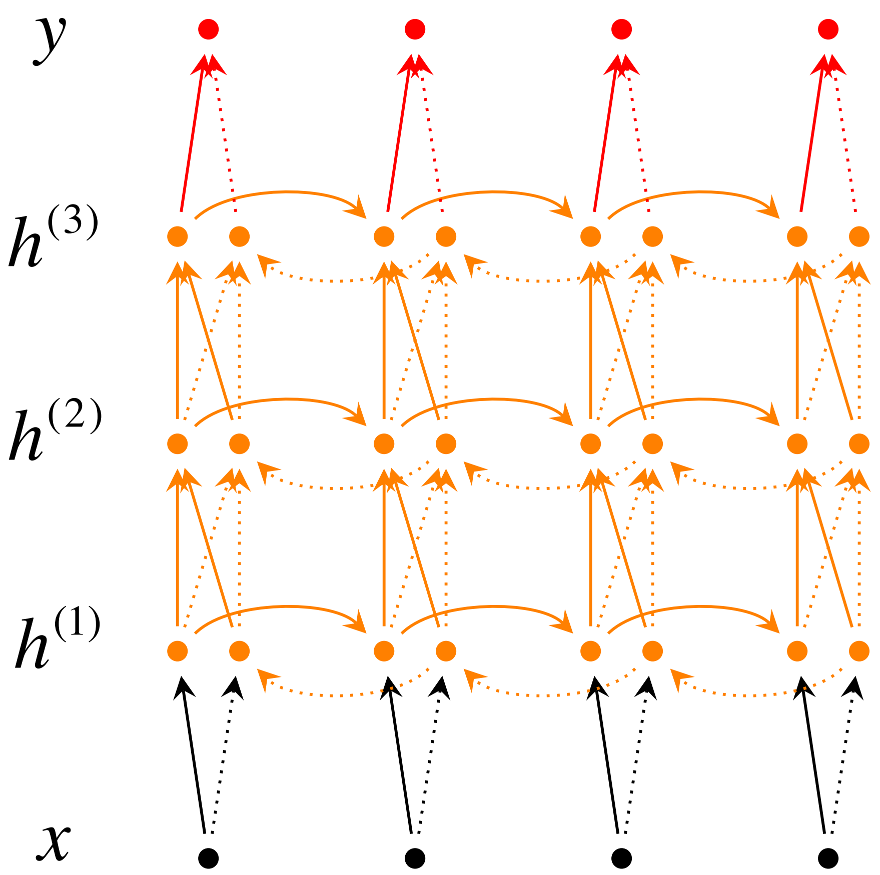
Units with long term dependencies have active update gates z

$$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$
$$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$
$$\tilde{h}_t = \tanh\left(Wx_t + r_t \circ Uh_{t-1}\right)$$
$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$$

Illustration:



Deep Bidirectional RNN (Irsoy and Cardie)



$$\overrightarrow{h}_t^{(i)} = f(\overrightarrow{W}^{(i)} h_t^{(i-1)} + \overrightarrow{V}^{(i)} \overrightarrow{h}_{t-1}^{(i)} + \overrightarrow{b}^{(i)})$$

$$\overleftarrow{h}_t^{(i)} = f(\overleftarrow{W}^{(i)} h_t^{(i-1)} + \overleftarrow{V}^{(i)} \overleftarrow{h}_{t+1}^{(i)} + \overleftarrow{b}^{(i)})$$

$$y_t = g(U[\overrightarrow{h}_t^{(L)}; \overleftarrow{h}_t^{(L)}] + c)$$

Each memory layer passes an intermediate sequential representation to the next.

Optimization for Long-term Dependencies

- Avoiding gradient exploding
  - Clipping Gradients

$$\text{if } ||\boldsymbol{g}|| > v$$

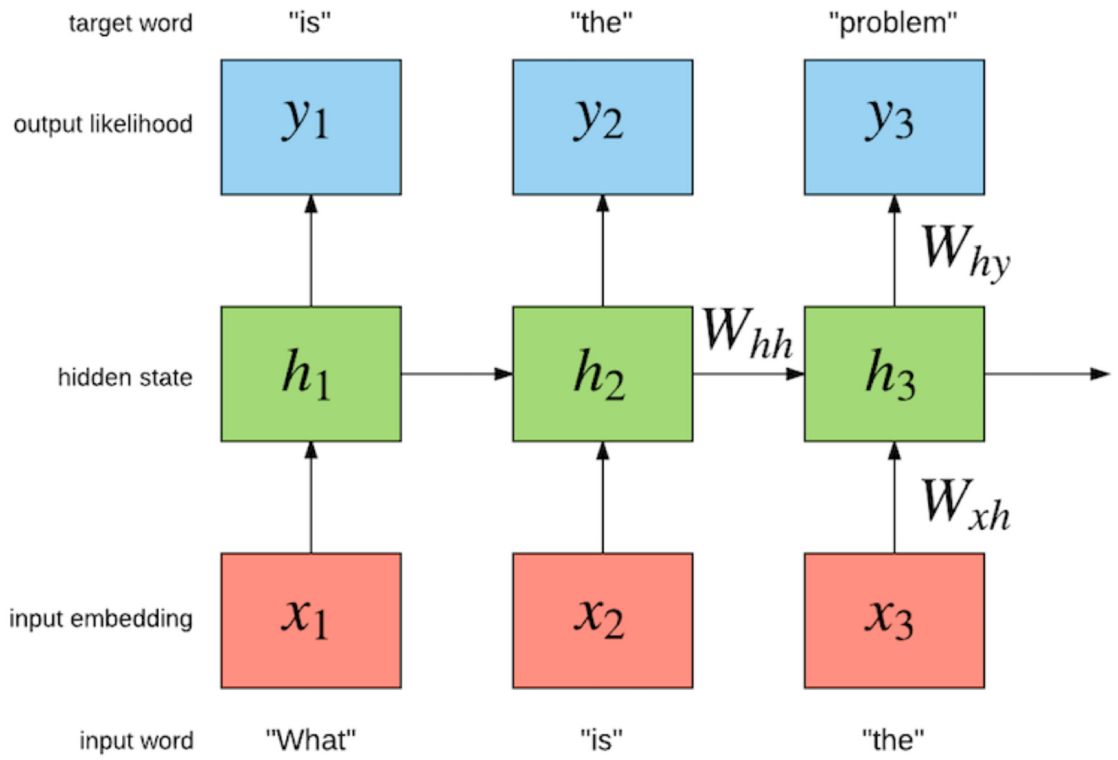
$$\boldsymbol{g} \leftarrow \frac{\boldsymbol{g}v}{||\boldsymbol{g}||}$$

Optimization for Long-term Dependencies

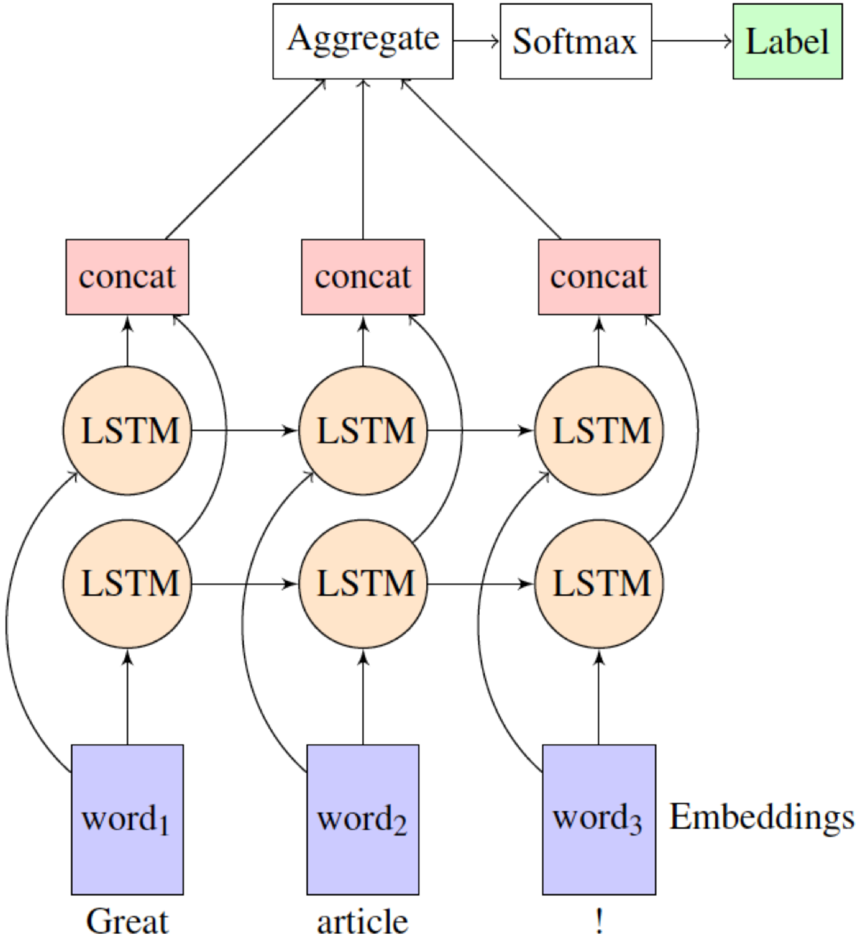
- Avoiding gradient vanishing
  - With LSTM or GRU
  - Or regularize or constrain the parameters so as to encourage “information flow”
- Make $\frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial h_{t-1}}$ close to $\frac{\partial E}{\partial h_t}$. Pascanu et al. (2013a) propose the following regularizer:

$$\Omega = \sum_t \left(\frac{\left|\left| \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial h_{t-1}} \right|\right|}{\left|\left| \frac{\partial E}{\partial h_t} \right|\right|} - 1 \right)^2$$

Applications: Language Modeling



Applications: Sentence Classification



Applications: Sequence Tagging

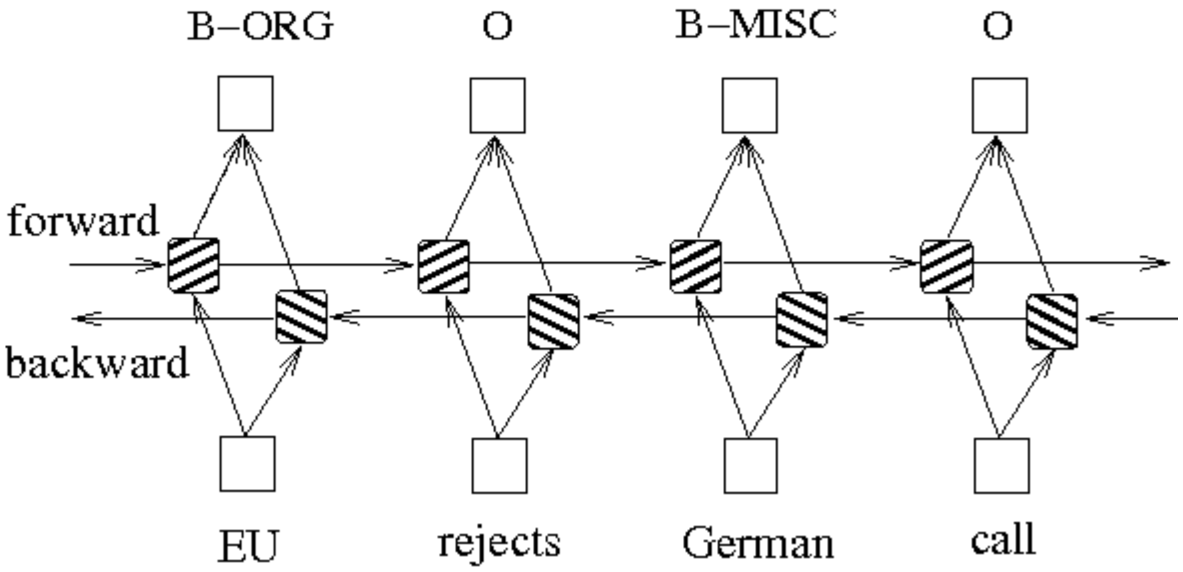


Figure: Bidirectional LSTM-CRF

Applications: Sequential Recommendation

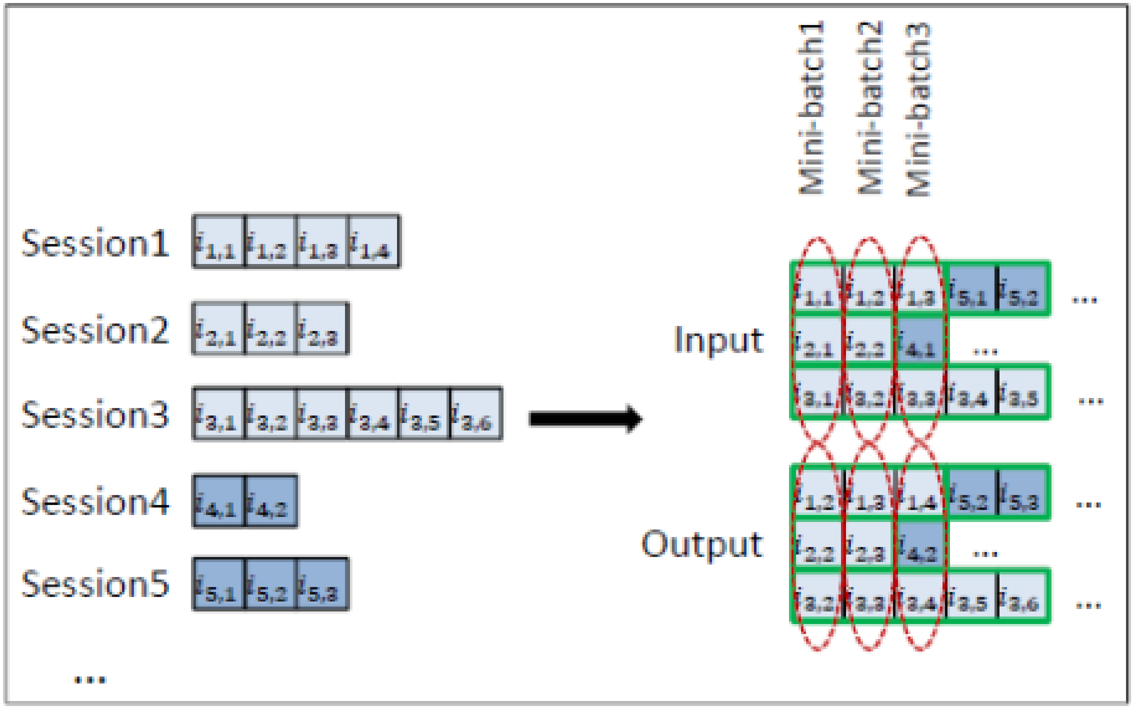


Figure: User sequential behaviors

References

- Chapter 10, Deep Learning Book