Natural Language Understanding, Generation, and Machine Translation

Lecture 5: Recurrent Neural Networks

Alexandra Birch

22 January 2025 (week 2)

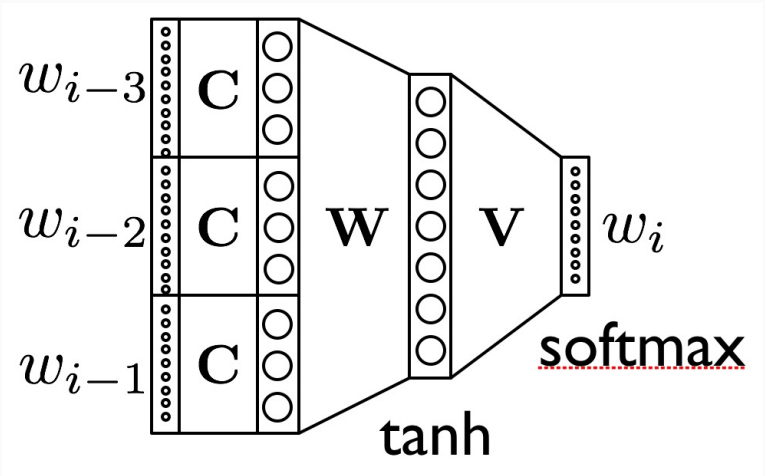
School of Informatics
University of Edinburgh
a.birch@ed.ac.uk

Overview

- Recurrent networks for language modeling
  - From Feedforward to Recurrent Networks
  - Recurrent Networks as Recursive Functions
  - Backpropagation through Time
  - Vanishing Gradients
- Long Short-term Memory
  - Architecture
  - Training

Reading: Section 6 of 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.

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!

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 $P : V \to \mathbb{R}_+$

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

Markov assumptions are wrong for language

The **roses** are red.

Markov assumptions are wrong for language

The **roses** are red.

The **roses** in the vase are red.

Markov assumptions are wrong for language

The **roses** are red.

The **roses** in the vase are red.

The **roses** in the vase by the door are red.

Markov assumptions are wrong for language

The **roses** are red.

The **roses** in the vase are red.

The **roses** in the vase by the door are red.

The **roses** in the vase by the door to the kitchen are red.

Markov assumptions are wrong for language

The **roses** are red.

The **roses** in the vase are red.

The **roses** in the vase by the door are red.

The **roses** in the vase by the door to the kitchen are red.

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

Markov assumptions are wrong for language

The **roses** are red.

The **roses** in the vase are red.

The **roses** in the vase by the door are red.

The **roses** in the vase by the door to the kitchen are 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 ___

Recurrent networks for language modeling

We need to model arbitrary context. How?

Context is important in language modeling:

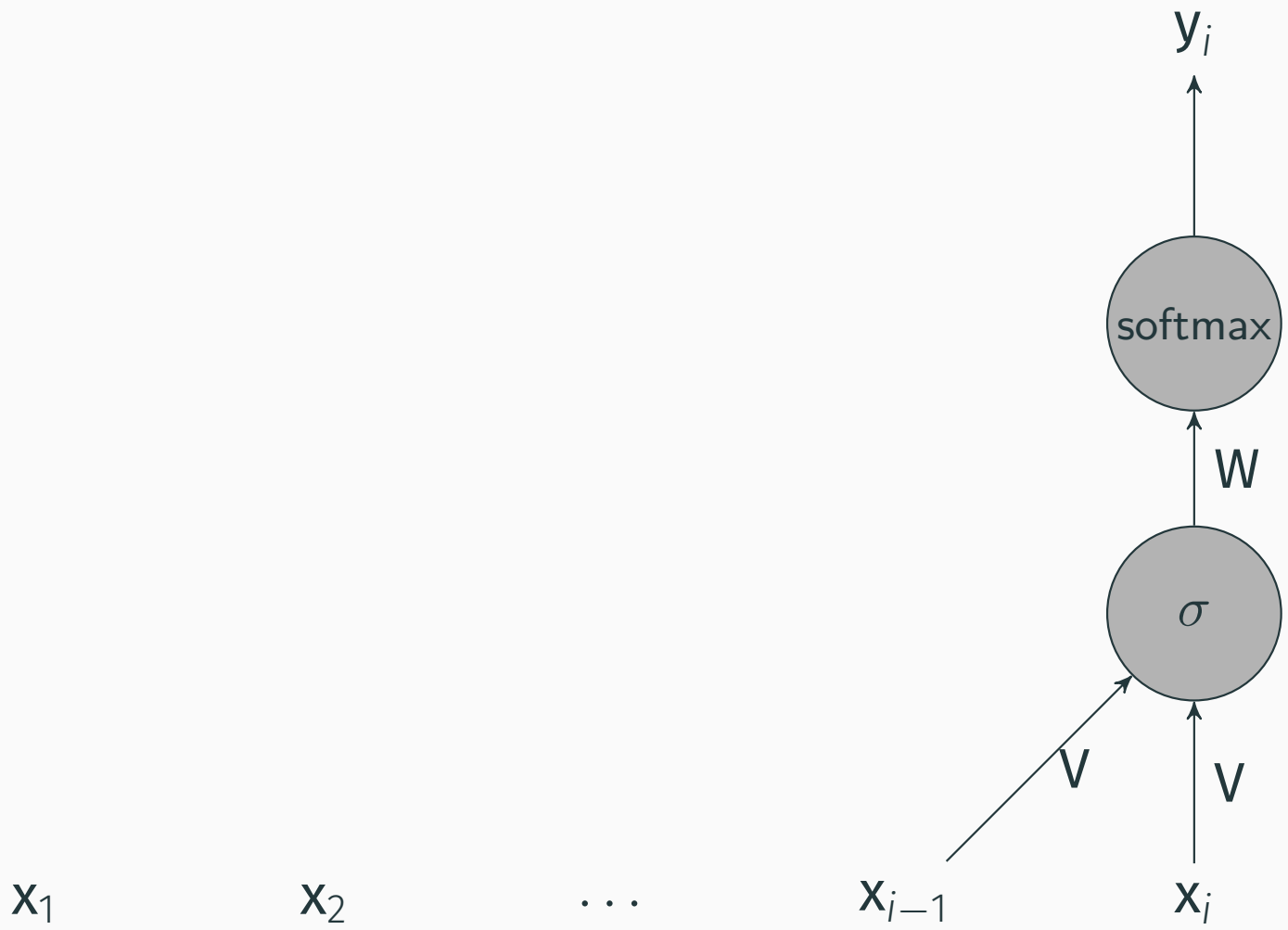
- But n-gram language models use a fixed context window.
- Feedforward networks also use a fixed context window...
- but linguistic dependencies can be arbitrarily long!

This is where *recurrent neural networks* come in!

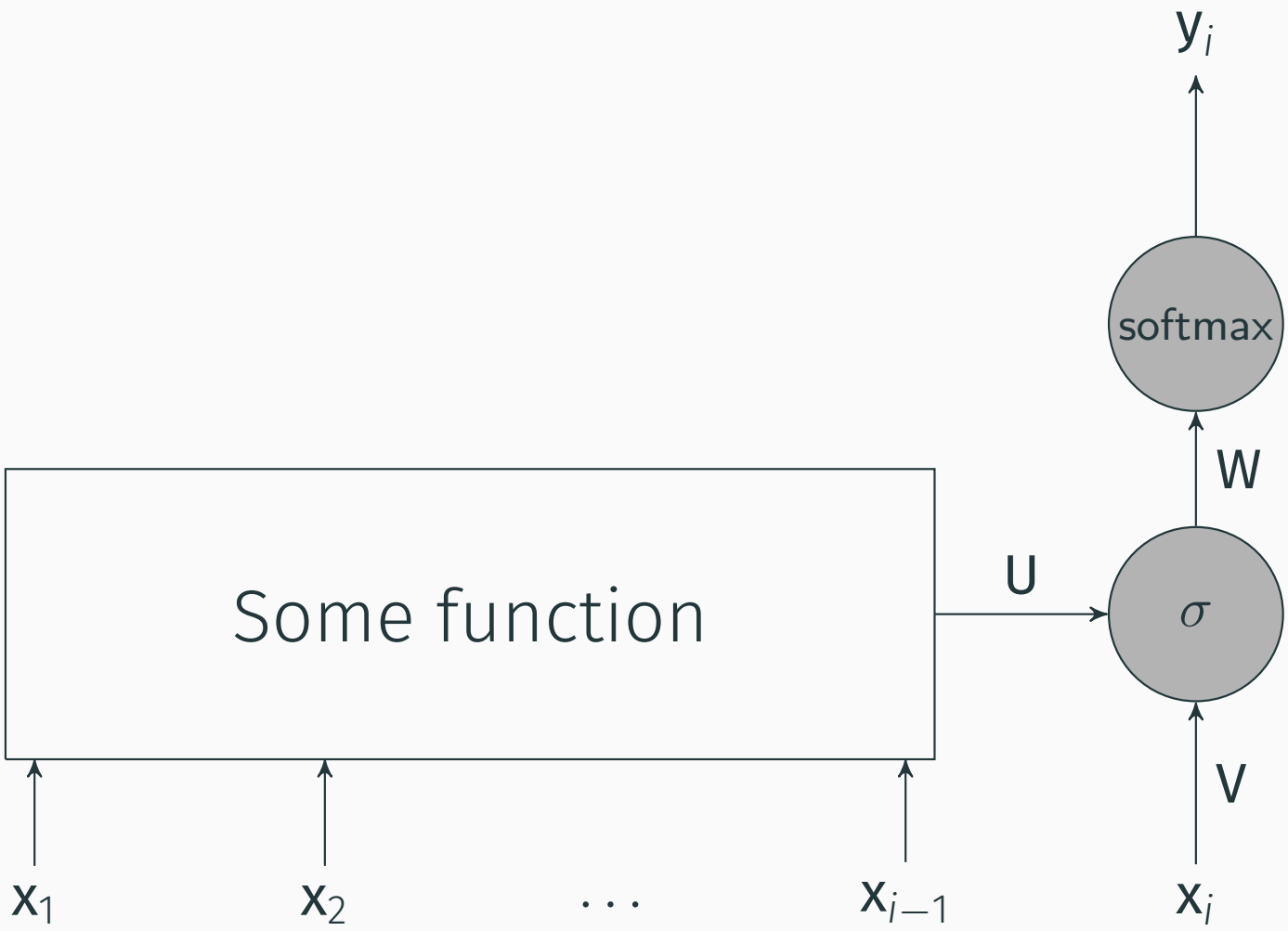
Glossary

- $\mathbf{x}_i$: 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

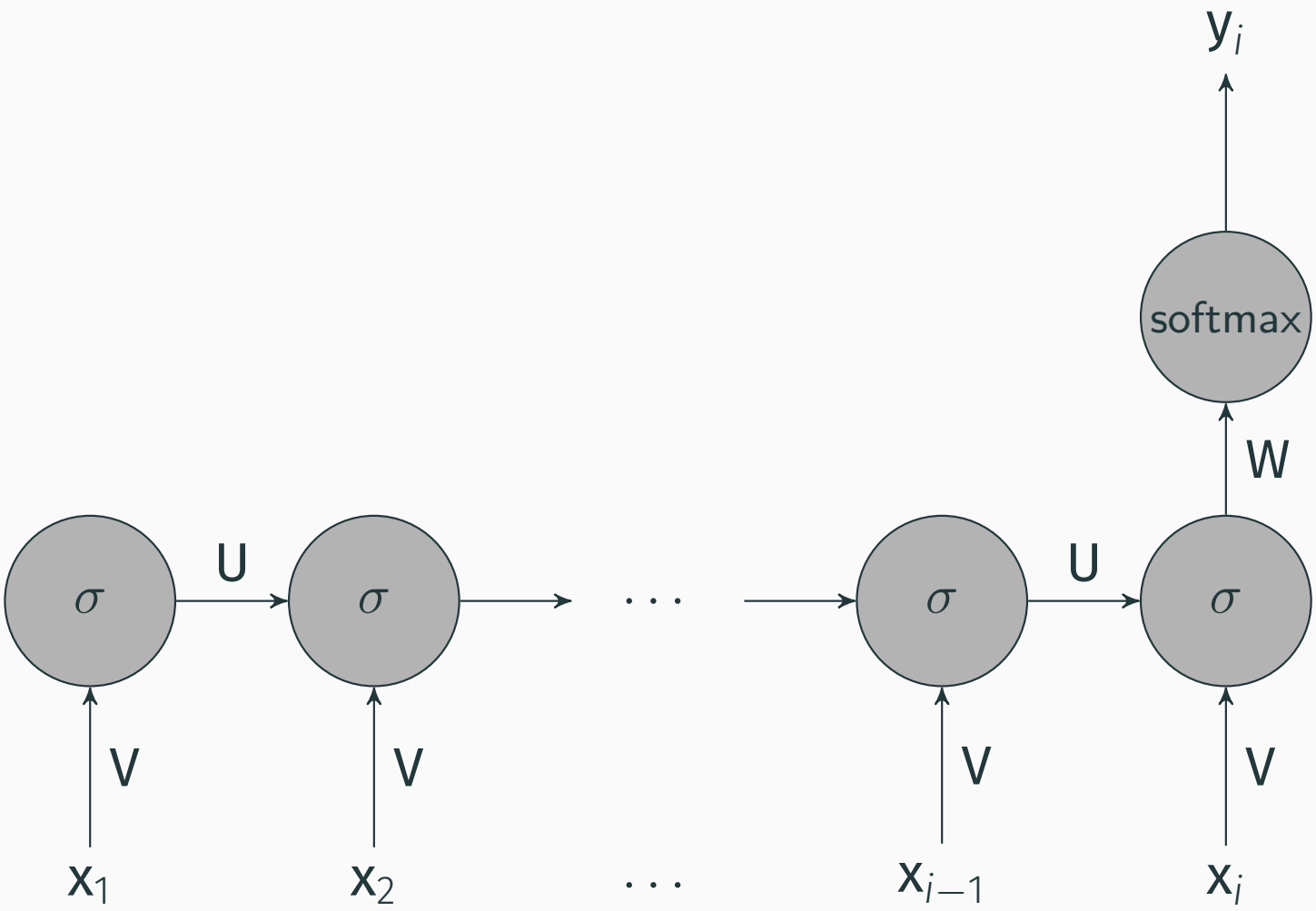
From feedforward to recurrent neural networks



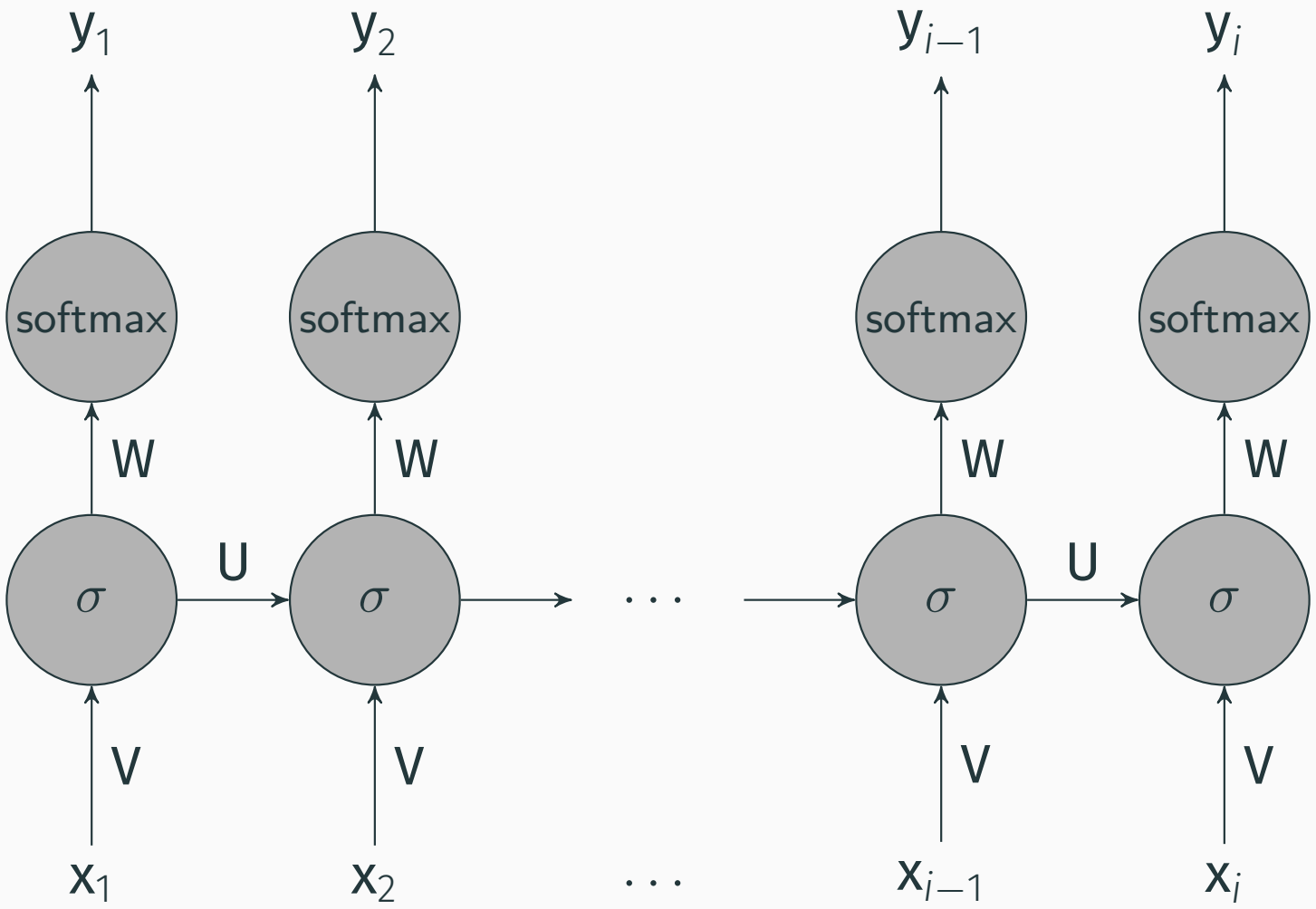
From feedforward to recurrent neural networks



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 $\mathbf{h}_i$, 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

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

Now P is a function of x_i, and, recursively through $\mathbf{h}_i$, 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

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

Note change of notation reflects BPTT reading Guo (2013)
Hidden layer is now $\mathbf{s}(t)$ (for state), showing weight multiplication separately to activation, removing bias
Input layer x to hidden layer:

$$\mathbf{h} = \tanh(\mathbf{V}\mathbf{x} + \mathbf{b}_1)$$

$$\begin{aligned} s_j(t) &= f(net_j(t)) \\ net_j(t) &= \sum_i^l x_i(t) v_{ji} \end{aligned}$$

Forward Propagation computes the output

The input at time t is $\mathbf{x}(t)$, output is $\mathbf{y}(t)$, and hidden layer $\mathbf{s}(t)$.
Input layer $\mathbf{x}$ to hidden layer $\mathbf{s}$:

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

$$net_j(t) = \sum_i^l x_i(t) v_{ji} \tag{2}$$

Hidden layer $\mathbf{s}$ to output $\mathbf{y}$:

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

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

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

Forward Propagation computes the output

The input at time t is $\mathbf{x}(t)$, output is $\mathbf{y}(t)$, and hidden layer $\mathbf{s}(t)$.
Input layer $\mathbf{x}$ to hidden layer $\mathbf{s}$:

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

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

Hidden layer $\mathbf{s}$ to output $\mathbf{y}$:

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

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

where $f(z) = \sigma(z)$, and $g(z) = \mathbf{softmax}(z)$.
So far this was a standard feedforward network.
Now we add the recurrence.

Backpropagation computes the gradient

For output units, we update the weights $\mathbf{W}$ 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 $\mathbf{V}$ 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:

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

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

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

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

Note: Guo (2013) incorrectly writes $f'(g_{pj}(t-1))$ instead of $f'(net_{pj}(t-1))$.

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 **U** using backprop through time. For timestep t:

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

For timestep $t-1$:

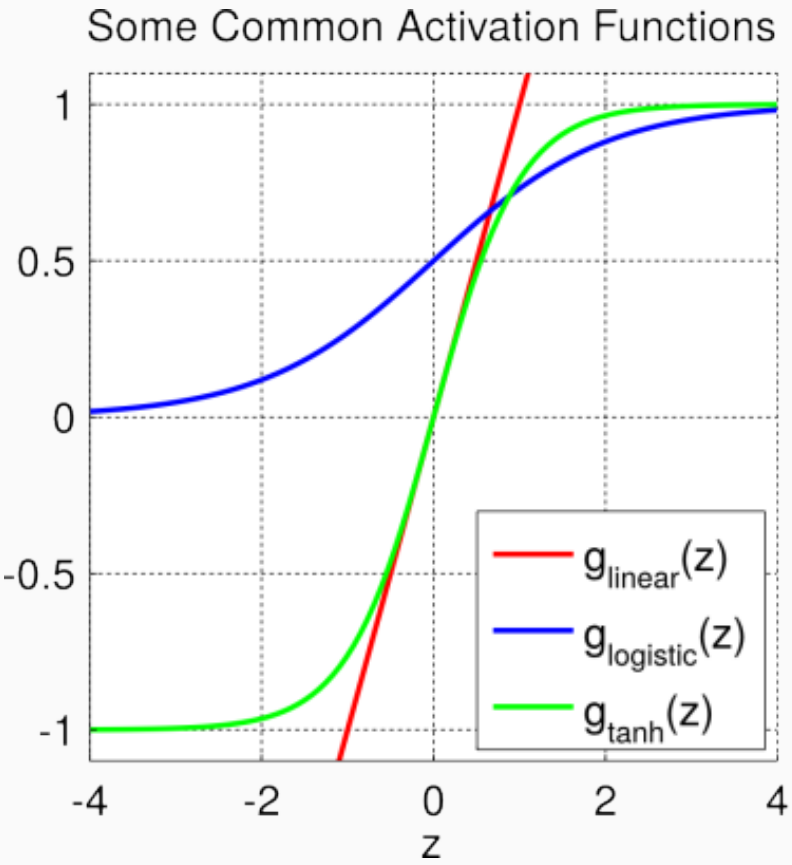
$$\delta_{pj}(t-1) = \sum_{h}^{m} \delta_{ph}(t) u_{hj} f'(net_{pj}(t-1))$$

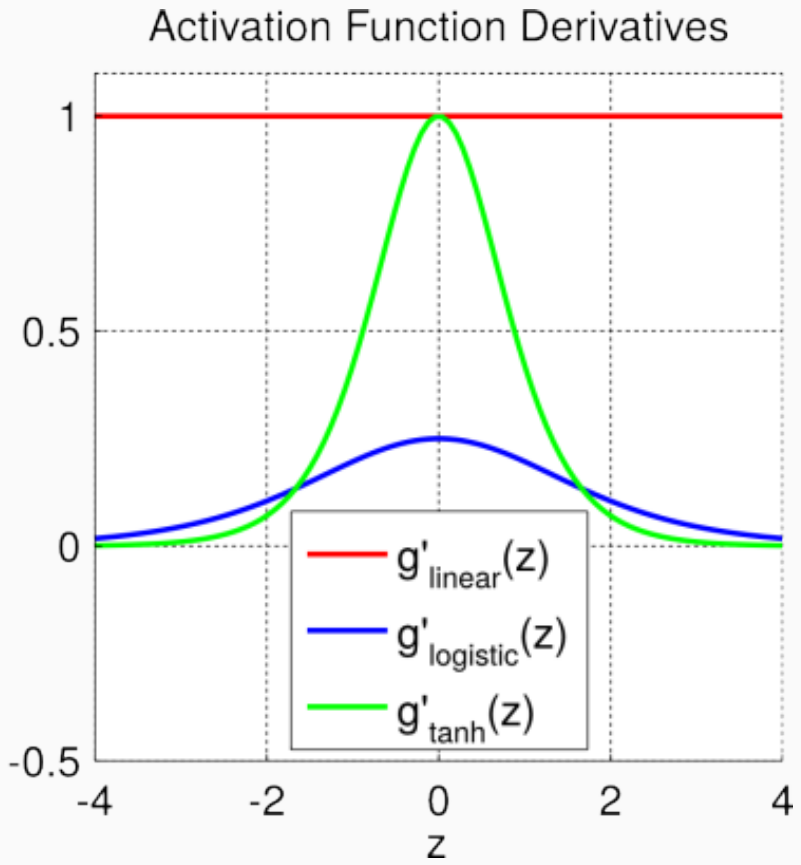
For time step $t-2$:

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

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.

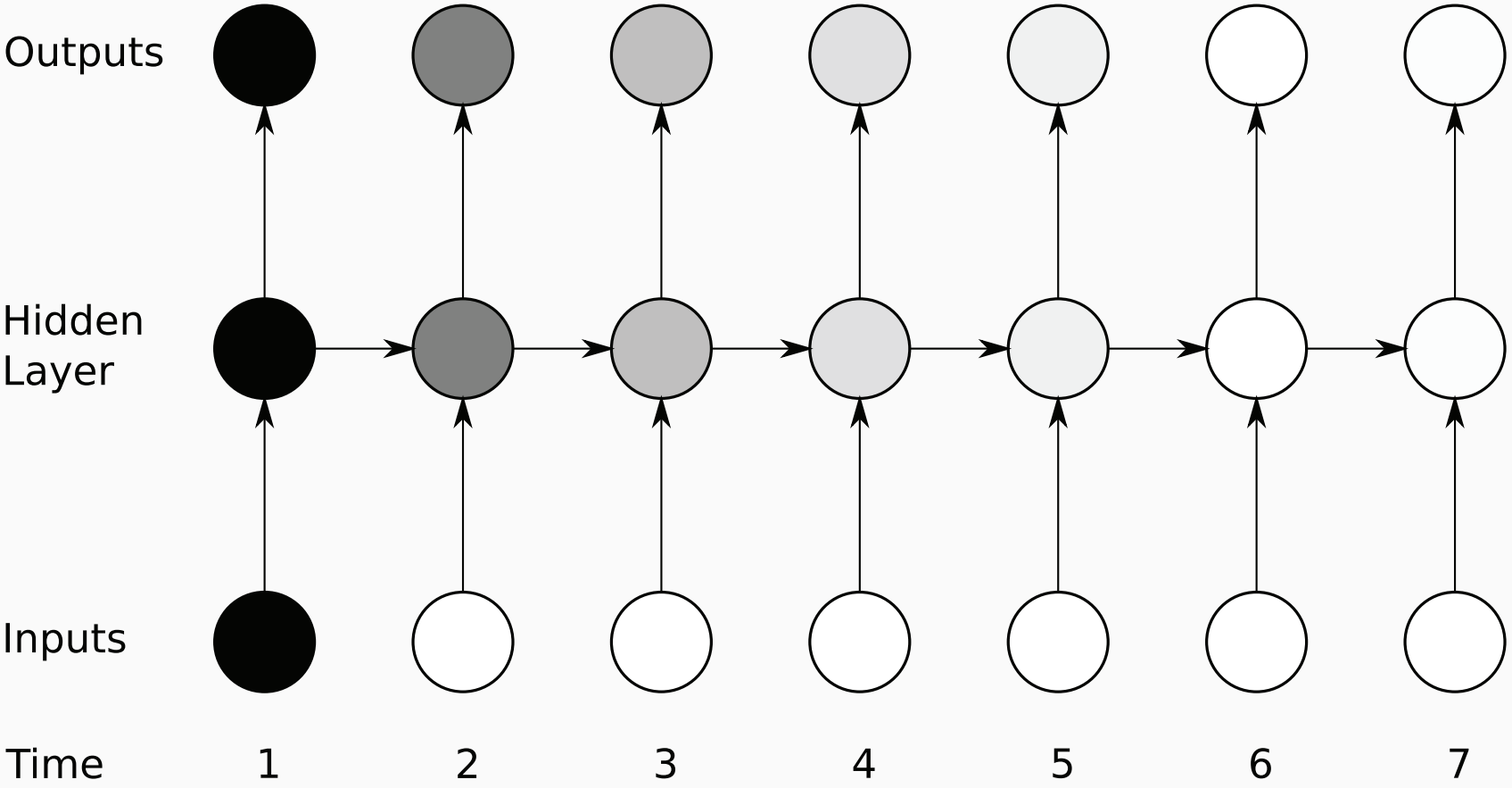




[Source: https://theclevermachine.wordpress.com/]

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).]

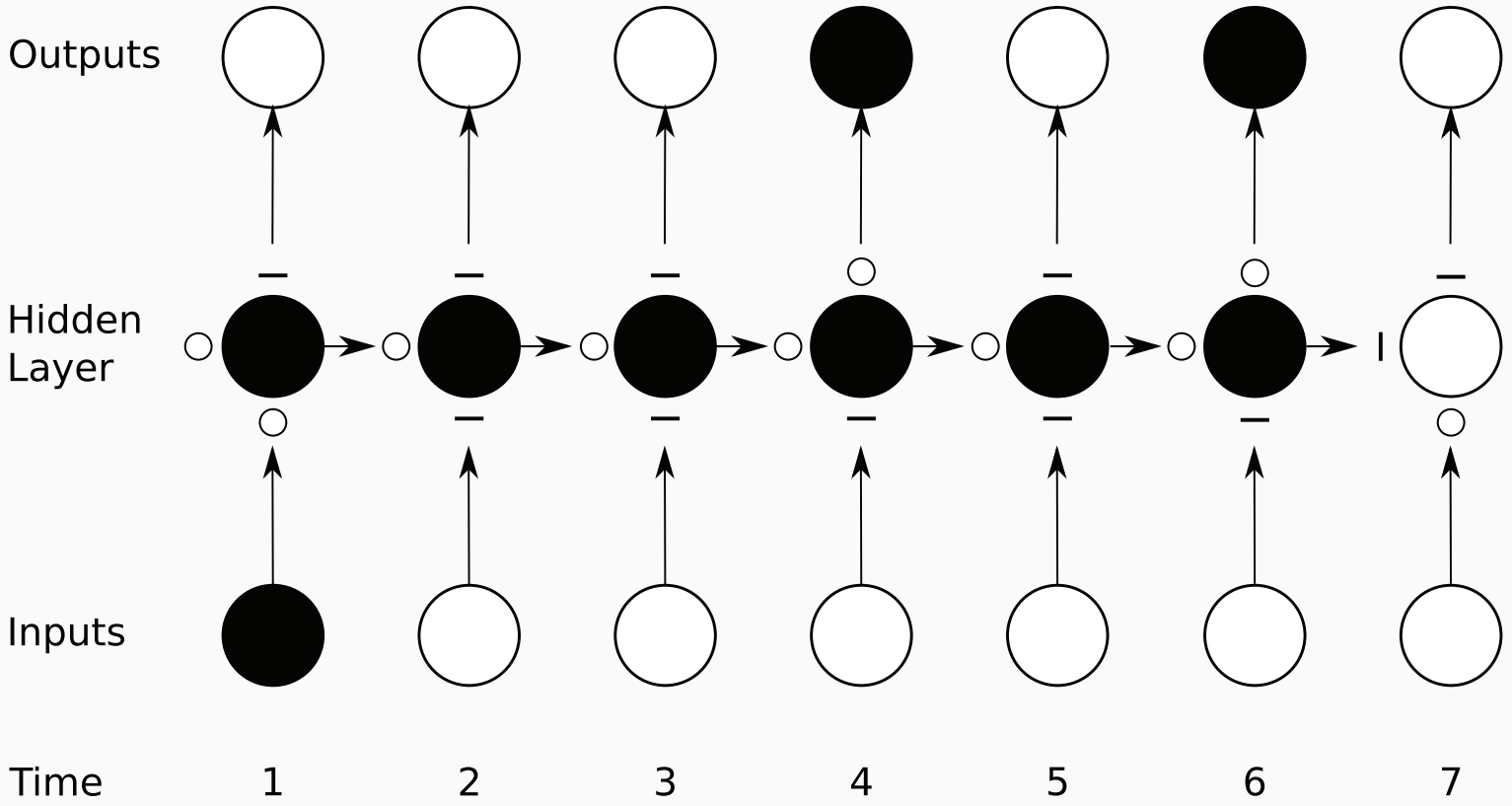
Summary

- 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.

Long Short-term Memory

Long Short-term Memory

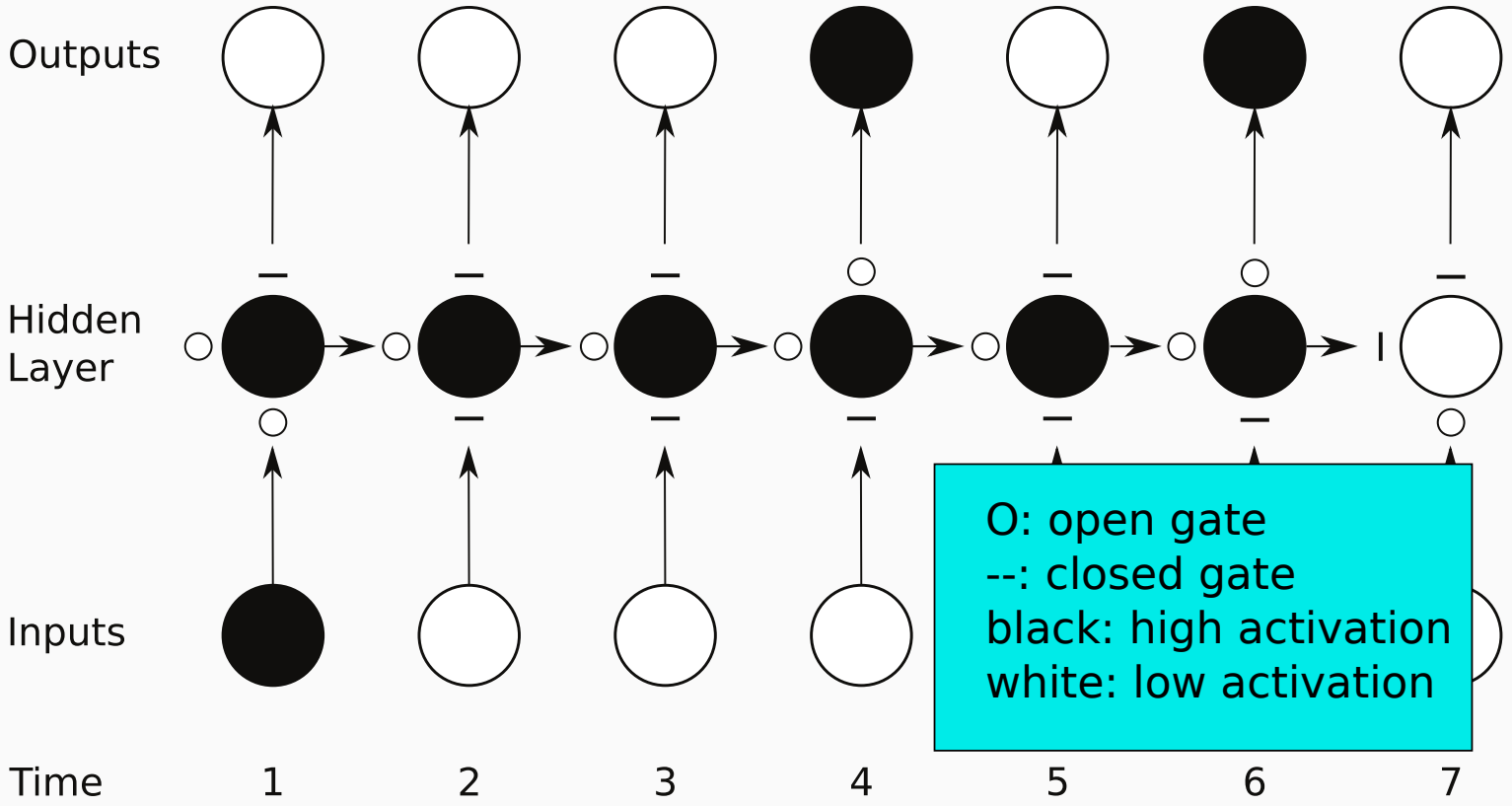
Solution: network can sometimes pass on information from previous time steps unchanged, so that it can learn from distant inputs:



[Source: Graves (2012).]

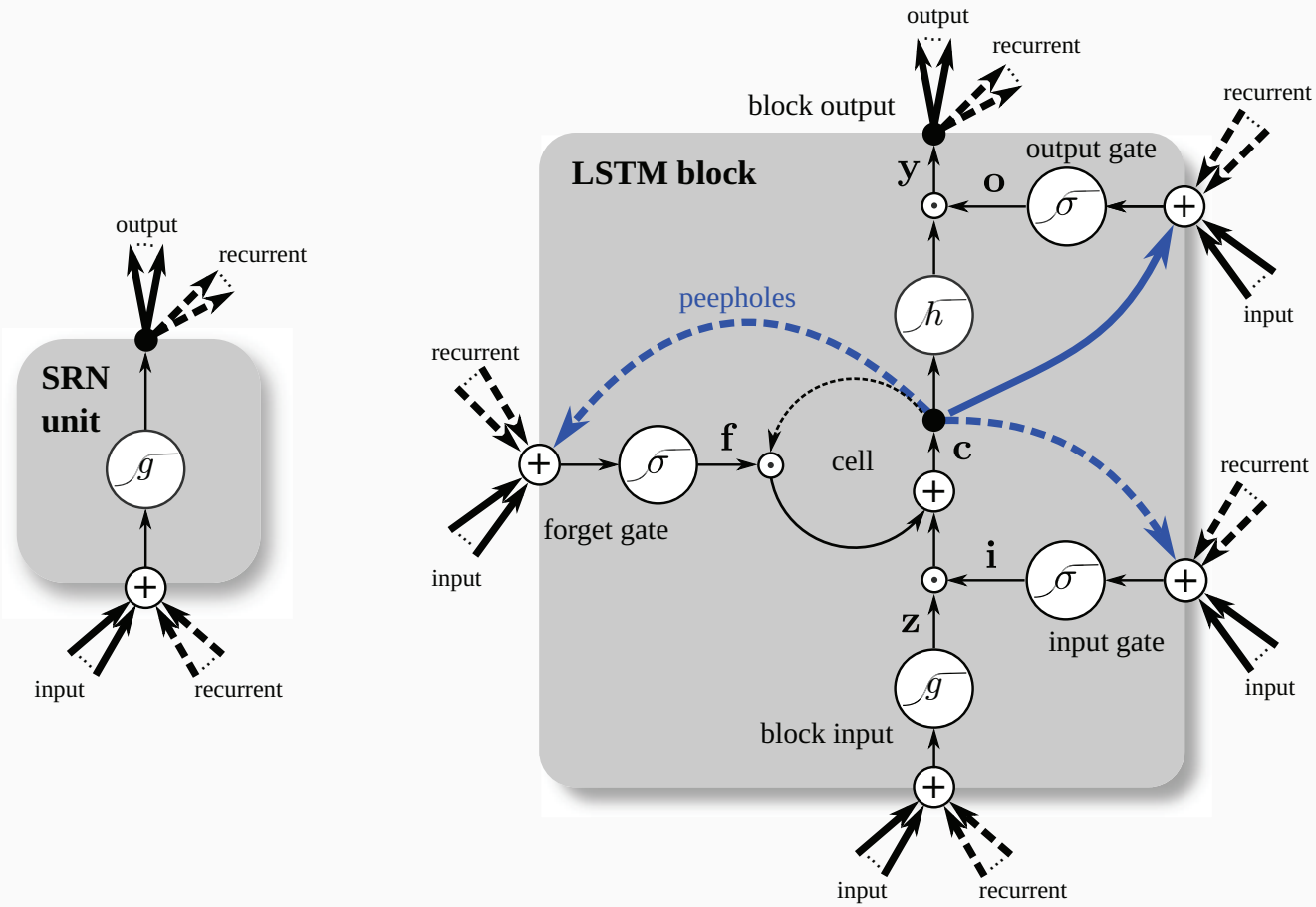
Long Short-term Memory

Solution: network can sometimes pass on information from previous time steps unchanged, so that it can learn from distant inputs:



[Source: Graves (2012).]

RNN Unit compared to LSTM Memory Block



[Source: Greff et al. (2016).]

The Gates and the Memory Cell

- Gates are *regular hidden units:* they sum their input and pass it through a sigmoid activation function;
- all four inputs to the block are the same: the *input layer and the recurrent layer* (hidden layer at previous time step);
- all gates have *multiplicative connections:* if the activation is close to zero, then the gate doesn't let anything through;
- the *memory cell* itself is linear: it has no activation function;
- but the block as a whole has input and output activation functions (can be tanh or sigmoid);
- all connections within the block are *unweighted:* they just pass on information (i.e., copy the incoming vector);
- the only output that the rest of the network sees is what the output gate lets through.

Vanishing Gradients Again

Why does this solve the vanishing gradient problem?

- Additive connections between memory cells, so its gradient doesn't vanish;
- an LSTM block can retain information indefinitely: if the forget gate is open (close to 1) and the input gate is closed (close to 0), then the cell just passes on the activation from the previous time step
- in addition, the block can decide when to output information by opening the output gate;
- the block can therefore retain information over an arbitrary number of time steps before it outputs it;
- the block learns when to accept input, produce output, and forget information: the gates have trainable weights.
- Empirically, research has found that trained LSTMs can be sensitive to hundreds of previous words.

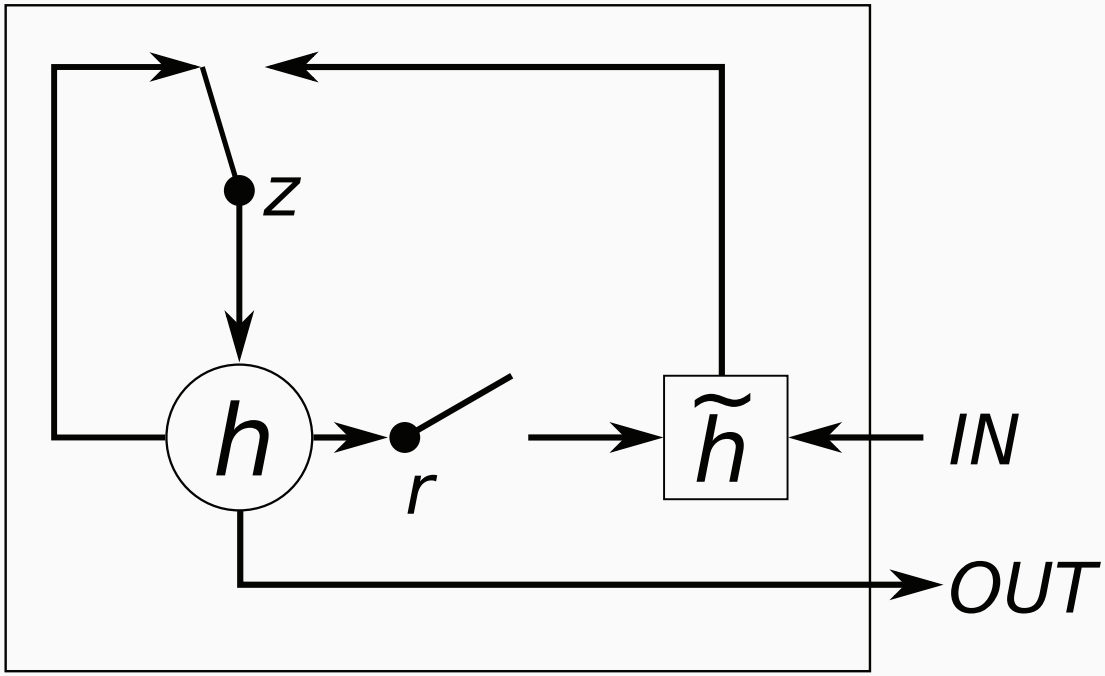
Training

- The original LSTM (Hochreiter and Schmidhuber, 1997) was trained with backprop through time;
- but BPTT was truncated after one step, assuming that long-range dependencies are dealt with by LSTM blocks;
- recall that the activation of the memory cell doesn't vanish (unlike the activation of standard recurrent connections);
- recent work has used untruncated BPTT, resulting in more accurate gradients.

We follow Graves's notation, which extends the RNN notation introduced in the last lecture.

The Gated Recurrent Unit

The Gated Recurrent Unit (GRU) is similar to the LSTM unit. In most cases it performs comparably, but it has fewer parameters and is quicker to train.



Source: Cho et al. (2014).

We will use the GRU in Coursework 1.

Applications

Language modeling: use the LSTM just like a standard RNN:

Method	Perplexity
LSTM (512 units)	68.8
IRNN (4 layers, 512 units)	69.4
IRNN (1 layer, 1024 units + linear projection)	70.2
RNN (4 layers, 512 tanh units)	71.8
RNN (1 layer, 1024 tanh units + linear projection)	72.5

[Source: Le et al. (2015).]

IRNN: network of ReLUs (rectified linear units) initialized with identify matrix. The ReLU activation function is $f(z) = \max(z, 0)$.

Summary

- Backprop through time with RNNs has the problem that gradients vanish with increasing timesteps;
- the LSTM is a way of addressing this problem;
- it replaces additive hidden units with complex memory blocks;
- a linear memory cell is the core of the each block;
- input gate controls whether cell receives input; output gate controls whether the cell state is passed on; forget gate determines whether recurrent input is passed on;
- the LSTM can be trained with standard backprop or BPTT;
- applications include sequence labeling and sequence-to-sequence mapping tasks.

References

Cho, K., van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111.

Graves, A. (2012). Supervised sequence labelling with recurrent neural networks. pages 37–45. Springer.

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2016). Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232.

Guo, J. (2013). Backpropagation through time unpublished material. In *Harbin Institute of Technology*.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Le, Q. V., Jaitly, N., and Hinton, G. E. (2015). A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*.