

Natural Language Understanding, Generation, and Machine Translation

Lecture 6: Recurrent Neural Networks and LSTMs

Adam Lopez

Credits: Mirella Lapata and Frank Keller

24 January 2019

School of Informatics

University of Edinburgh

`alopez@inf.ed.ac.uk`

Recap: probability, language models, and feedforward networks

Simple Recurrent Networks

Backpropagation Through Time

Long short-term memory

Recap: probability, language models, and feedforward networks

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

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

Rules of probability (remember: vocabulary V is finite)

Probabilities are non-negative

$$P : V \rightarrow \mathcal{R}_+$$

... and sum to one

$$\sum_{w \in V} P(w \mid w_{i-n+1}, \dots, w_{i-1}) = 1$$

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!

Rules of probability (remember: vocabulary V is finite)

Probabilities are non-negative

$$P : V \rightarrow \mathcal{R}_+$$

... and sum to one

$$\sum_{w \in V} P(w \mid w_{i-n+1}, \dots, w_{i-1}) = 1$$

A neural network is a function from a vector to a vector

- Matrix multiplication: converts an m -element vector to an n -element vector. Parameters are usually of this form.
- Sigmoid, exp, tanh, RELU, etc: elementwise nonlinear transform from m -element vector to m -element vector.
- Concatenate an m -element and n -element vector into an $(m + n)$ -element vector.
- Functions can have multiple outputs: if we need an m -element output and an n -element output, create a function with $m + n$ -element output.

Input, parameter matrices, and whole subfunctions can be reused.

i.e. if g is any of these and f is a function with two (vector) arguments, then $f(g, g)$ is fine.

Learning: generic operations

1. Decide a **loss** function. Goal: minimize loss w.r.t. parameters on training data.
2. Single learning algorithm: stochastic gradient descent or one of its variants.
3. To compute gradients, use backpropagation / automatic differentiation.

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

Probability distributions are vectors!

Summer is hot winter is _____


cold  0.6

grey  0.3

winter  0.1

is  0

hot  0

summer  0

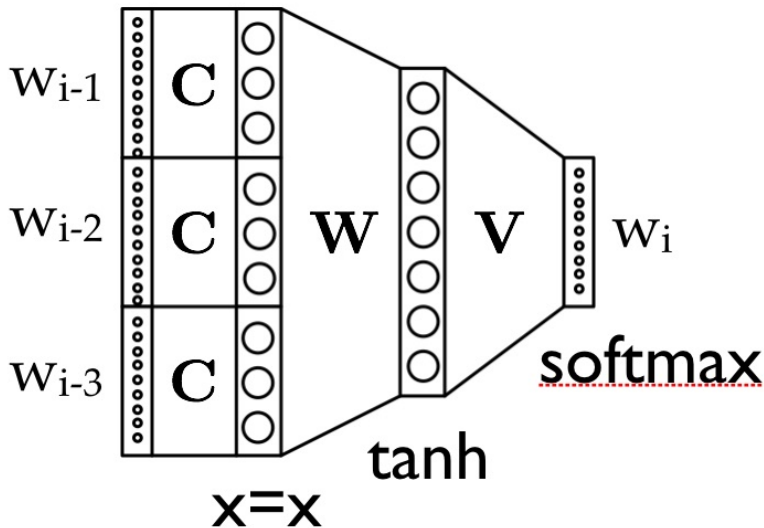
Softmax will convert any vector to a probability distribution.

Elements of discrete vocabularies are vectors!

	Summer	is	hot	winter	is
is	0	1	0	0	1
cold	0	0	0	0	0
grey	0	0	0	0	0
hot	0	0	1	0	0
summer	1	0	0	0	0
winter	0	0	0	1	0

Use *one-hot encoding* to represent any element of a finite set.

Feedforward LM: function from a vectors to a vector



How much context do we need?

The **roses** are red.

How much context do we need?

The **roses** are red.

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

How much context do we need?

The **roses** are red.

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

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

How much context do we need?

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.

How much context do we need?

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 _____

How much context do we need?

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 _____

Simple Recurrent Networks

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.

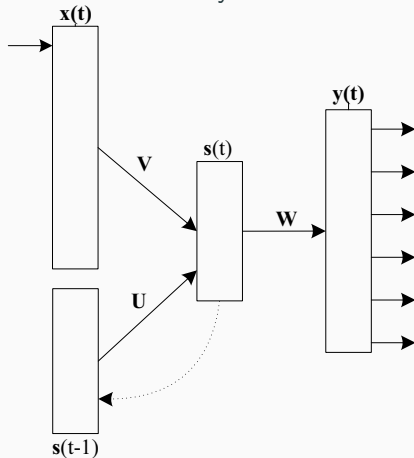
This is where *recurrent neural networks* come in:

- 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:



Only the dependence on the previous hidden layer is new.
For now, pretend it is observed as input.

Input and Output

- To initialize, 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 $x(t)$ is the one-hot encoding of word w_t ;
- size of hidden layer is usually 30–500 units, depending on size of training data.
- output vector $y(t)$ is a probability distribution over the next word w_{t+1} given the current word w_t and context $s(t - 1)$;

Input and Output

- To initialize, 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 $x(t)$ is the one-hot encoding of word w_t ;
- size of hidden layer is usually 30–500 units, depending on size of training data.
- output vector $y(t)$ is a probability distribution over the next word w_{t+1} given the current word w_t and context $s(t-1)$;

Since $s(t-1)$ is a function of $x(t-1)$ —and recursively, of all previous time steps—this function computes $P(w_i \mid w_1, \dots, w_{i-1})$ **with no Markov assumption!**

Training

We can use standard backprop with stochastic gradient descent:

- simply treat the network as a feedforward network with $s(t - 1)$ as additional **observed** input;
- backpropagate the error to adjust weight matrices **U** and **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.

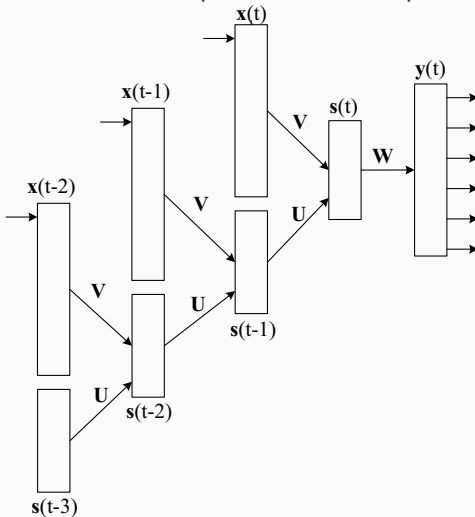
Backpropagation Through Time

From Simple to Full RNNs

- Let's drop the assumption that only the hidden layer from the previous time step is observed; it is computed as a function of previous inputs words!
- instead use all previous time steps;
- we can think of this as *unfolding over time*: the RNN is unfolded into a sequence of *connected* 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 \mathbf{W} using:

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

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

For hidden units, we update the weights \mathbf{V} using:

$$\Delta v_{ji} = \eta \sum_p^n \delta_{pj} x_{pi} \quad \delta_{pj} = \sum_k^o \delta_{pk} w_{kj} f'(net_{pj})$$

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 . 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 :

$$\Delta u_{ji} = \eta \sum_p^n \delta_{pj}(t) s_{ph}(t-1) \quad \delta_{pj}(t) = \sum_k^o \delta_{pk} w_{kj} f'(net_{pj})$$

For timestep $t-1$:

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

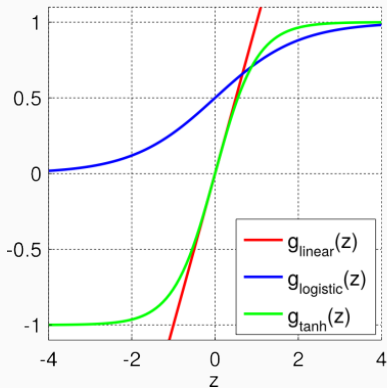
For time step $t-2$:

$$\begin{aligned} \delta_{pj}(t-2) &= \sum_h^m \delta_{ph}(t-1) u_{hj} f'(s_{pj}(t-2)) \\ &= \sum_h^m \sum_{h_1}^m \delta_{ph_1}(t) u_{h_1j} f'(s_{pj}(t-1)) u_{hj} f'(s_{pj}(t-2)) \end{aligned}$$

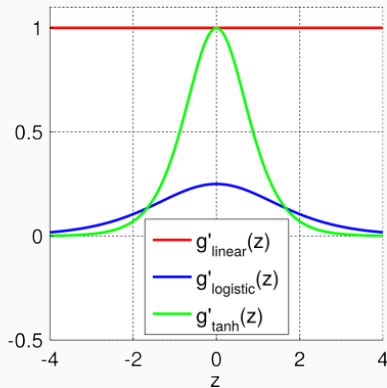
As we backpropagate through time, gradients tend toward 0

At every time step, we multiply the weights with another gradient. The gradients are < 1 so the deltas become smaller and smaller.

Some Common Activation Functions



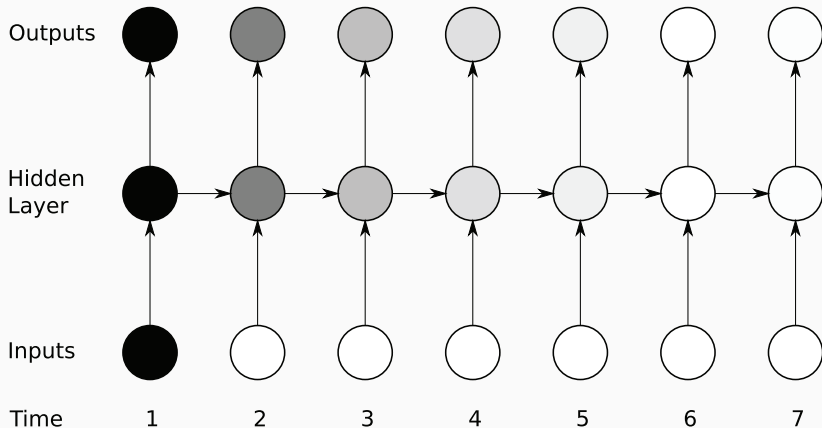
Activation Function Derivatives



[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:

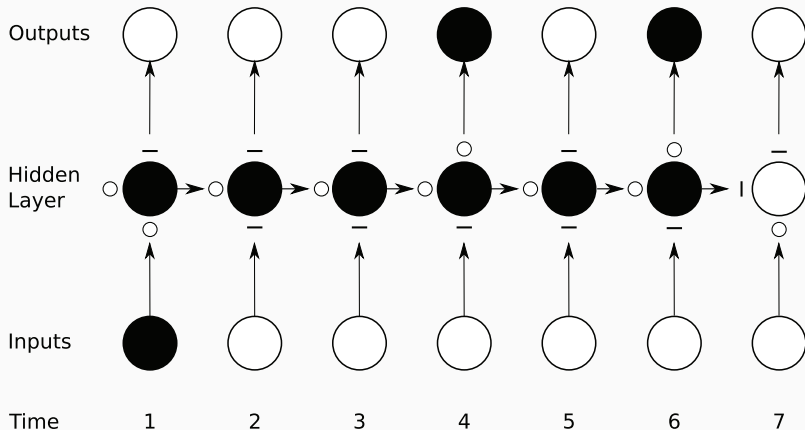


[Source: Graves, Supervised Sequence Labelling with RNNs, 2012.]

Long short-term memory

A better RNN: Long Short-term Memory

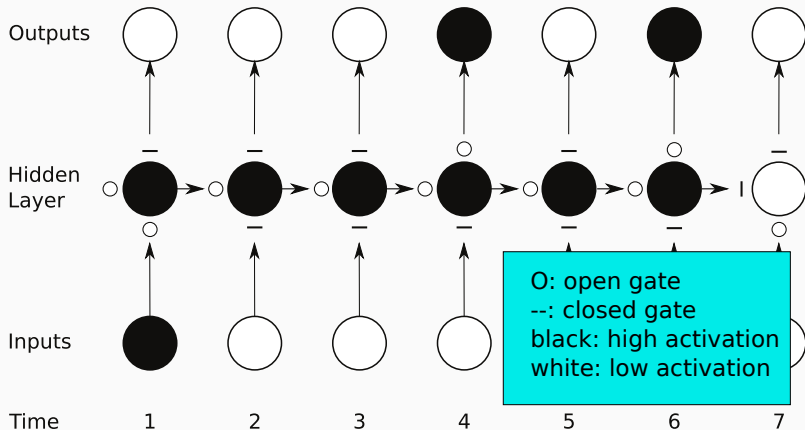
Solution: network can sometimes pass on information from previous time steps unchanged, so that it can learn from distant inputs:



[Source: Graves, Supervised Sequence Labelling with RNNs, 2012.]

A better RNN: 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, Supervised Sequence Labelling with RNNs, 2012.]

Architecture of the LSTM

To achieve this, we need to make the units of the network more complicated:

- LSTMs have a hidden layer of *memory blocks*;
- each block contains a recurrent *memory cell* and three multiplicative units: the *input, output and forget gates*;
- the gates are trainable: each block can learn whether to keep information across time steps or not.

In contrast, the RNN uses simple hidden units, which just sum the input and pass it through an activation function.

The Gates and the Memory Cell

Each memory block consists of four units:

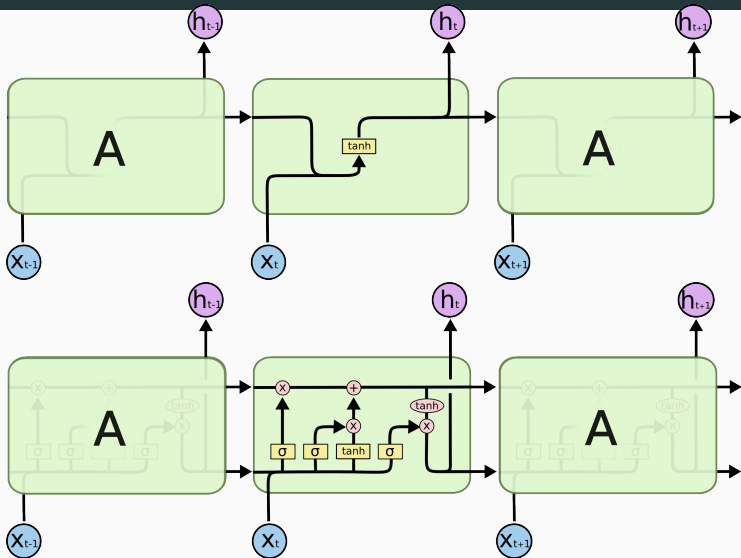
Input gate: controls whether the input to is passed on to the memory cell or ignored;

Output gate: controls whether the current activation vector of the memory cell is passed on to the output layer or not;

Forget gate: controls whether the activation vector of the memory cell is reset to zero or maintained;

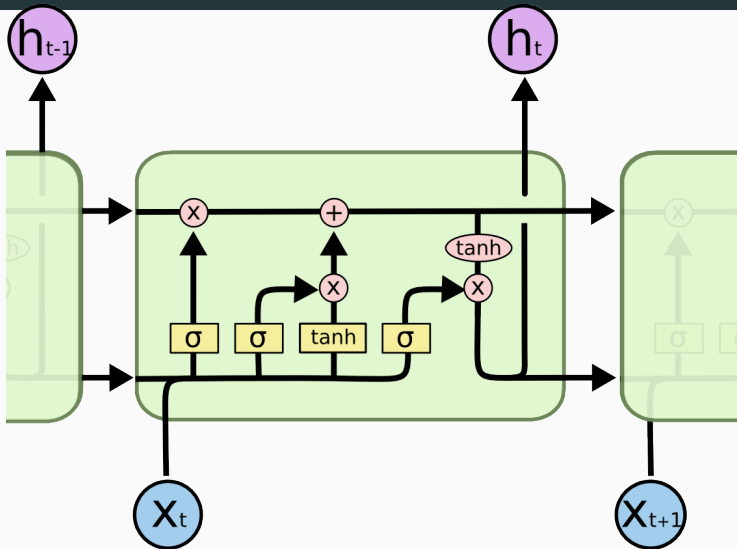
Memory cell: stores the current activation vector; with recurrent connection to itself controlled by forget gate.

LSTM vs. Simple RNN Memory block



[Source: Olah, Understanding LSTMs, 2015.]

LSTM Memory block



[Source: Olah, Understanding LSTMs, 2015.]

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

Does this solve the vanishing gradient problem?

- the memory cell is linear, 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 activation of the cell persists;
- 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.
- In practice: For language modeling, trained LSTMs are sensitive to *hundreds* of previous words.

Applications

LSTMs are the workhorse of modern NLP. You will find them anywhere the input or output of a learned function is a sequence:

- part of speech tagging and parsing;
- semantic role labeling;
- parsing.

With *attention* (introduced next week!) they are widely used for sequence-to-sequence problems:

- machine translation
- summarization;
- dialogue;

We will focus these applications in the rest of the course. There will not be many new architectures/ maths/ ML details.

Summary of key points (i.e. examinable content)

- Recurrent networks can encode a complete sequence, and thus we can use them for language modeling *without* making any Markov assumption!
- RNNs can be trained with standard backprop.
- 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.
- 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.

Coursework: RNNs and backpropagation

- Due 7th of February (two weeks from today) at 4pm (right before lecture).
- **Strongly** encouraged to work in pairs. Team details due 31st of January (one week from today).
- Part I: Implement crucial parts of an RNN from mathematical description, then train and apply to a psycholinguistic modeling task.
- Part II: **Optional** open-ended experimentation for additional points—only do if you are comfortable with Part I.