Deep Learning for NLP

Ed Grefenstette

Lecture Plan

- Word Vectors
- Language Modelling
- Sequence to Sequence + Attention
- Applications to NLP
- Composition and Classification

Word Vectors

How to represent words

Natural language text = sequences of **discrete symbols** (e.g. words).

Naive representation: one hot vectors in $R^{|vocabulary|}$ (very large).

Classical IR: document and query vectors are superpositions of word vectors.

$$\hat{d}_q = \arg\max_d sim(\mathbf{d}, \mathbf{q})$$

Similarly for **word classification problems** (e.g. document classification).

Issues: sparse, orthogonal representations, semantically weak.

How to represent words

We want **richer representations** expressing **semantic similarity**.

Distributional semantics:
"You shall know a word by the company it keeps." — J.R. Firth (1957)

Idea: produce **dense** vector representations based on the **context/use** of words.

Three main approaches: **count-based**, **predictive**, and **task-based**.

Count-based methods

Define a **basis vocabulary** *C* of context words.

Define a **word window** size *w*.

Count the basis vocabulary words occurring *w* words to the left or right of each instance of a **target word** in the corpus.

Form a **vector representation** of the target word based on these counts.

Count-based methods

... and the *cute* **kitten** *purred* and then ...
... the *cute furry* **cat** *purred* and *miaowed* ...
... that the *small* **kitten** *miaowed* and she ...
... the *loud furry* **dog** *ran* and *bit* ...

Example **basis vocabulary**: *{bit, cute, furry, loud, miaowed, purred, ran, small}*.

kitten context words: {cute, purred, small, miaowed}.
cat context words: {cute, furry, miaowed}.
dog context words: {loud, furry, ran, bit}.

Count-based methods

... and the *cute* **kitten** *purred* and then ...
... the *cute furry* **cat** *purred* and *miaowed* ...
... that the *small* **kitten** *miaowed* and she ...
... the *loud furry* **dog** *ran* and *bit* ...

Example **basis vocabulary**: *{bit, cute, furry, loud, miaowed, purred, ran, small}*.

$$\mathbf{kitten} = [0, 1, 0, 0, 1, 1, 0, 1]^\top$$
$$\mathbf{cat} = [0, 1, 1, 0, 1, 0, 0, 0]^\top$$
$$\mathbf{dog} = [1, 0, 1, 1, 0, 0, 1, 0]^\top$$

Count-based methods

Use inner product or cosine as **similarity kernel**. E.g.:

$$sim(\text{kitten}, \text{cat}) = cosine(\mathbf{kitten}, \mathbf{cat}) \approx 0.58$$
$$sim(\text{kitten}, \text{dog}) = cosine(\mathbf{kitten}, \mathbf{dog}) = 0.00$$
$$sim(\text{cat}, \text{dog}) = cosine(\mathbf{cat}, \mathbf{dog}) \approx 0.29$$

Reminder: $cosine(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \times \|\mathbf{v}\|}$

Cosine has the advantage that it's a *norm-invariant* metric.

Count-based methods

Not all features are equal: we must distinguish counts that are high because the are *informative* from those that are just *independently frequent contexts*.

Many **normalisation methods**: TF-IDF, PMI, etc.

Some **remove the need** for norm-invariant similarity metrics.

But... perhaps there are easier ways to address this problem of count-based methods (and others, e.g. choice of basis context).

Neural Embedding Models

Learning count based vectors produces an **embedding matrix** in $R^{|vocab| \times |context|}$:

$$\mathbf{E} = \begin{array}{c} \\ \text{kitten} \\ \text{cat} \\ \text{dog} \\ \vdots \end{array} \begin{array}{c} \begin{array}{cccc} \text{bit} & \text{cute} & \text{furry} & \dots \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & \dots \\ 0 & 1 & 1 & \dots \\ 1 & 0 & 1 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \end{array}$$

Rows are word vectors, so we can retrieve them with **one hot vectors** in $\{0,1\}^{|vocab|}$:

$$onehot_{cat} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad \mathbf{cat} = onehot_{cat}^{\top}\mathbf{E}$$

Symbols = unique vectors. Representation = embedding symbols with **E**.

Neural Embedding Models

1. Collect instances $t_i \in inst(t)$ of a word t of vocab V.
2. For each instance, collect its context words $c(t_i)$ (e.g. k-word window).
3. Define some score function $score(t_i, c(t_i); \boldsymbol{\theta}, \mathbf{E})$ with upper bound on output.
4. Define a loss:

$$L = -\sum_{t \in V} \sum_{t_i \in inst(t)} score(t_i, c(t_i); \theta, \mathbf{E})$$

5. Estimate:

$$\hat{\theta}, \hat{\mathbf{E}} = \arg\min_{\theta, \mathbf{E}} L$$

6. Use the estimated **E** as your embedding matrix.

Neural Embedding Models

Scoring function matters!

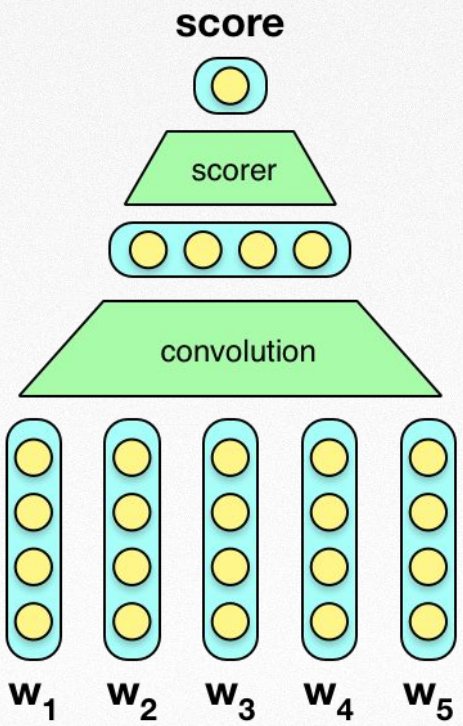
Easy to design a **useless scorer** (e.g. ignore input, output upper bound).

Ideally, scorer:

- Embeds t_i with **E** (obviously).
- Produces a score which is a function of how well t_i is accounted for by c(t_i), and/or vice versa.
- Requires the word to account for the context (or the reverse) more than another word in the same place.
- Produces a loss which is differentiable w.r.t. **θ** and **E**.

Neural Embedding Models: C&W

(Collobert et al. 2011)



Embed all words in a sentence with **E**.

Shallow convolution over embeddings.

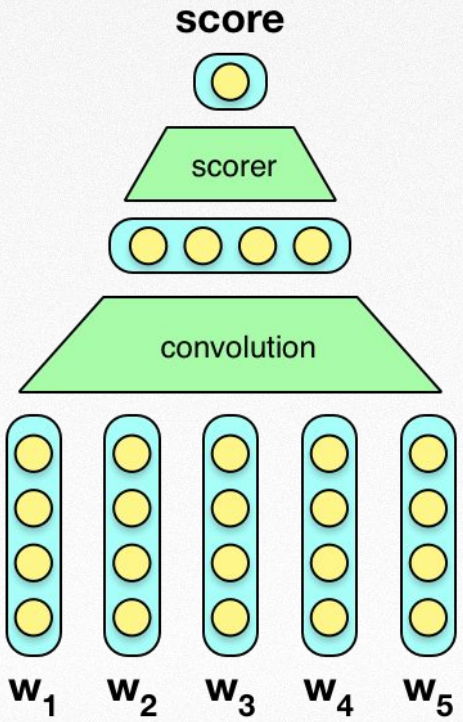
MLP projects output of convolution to a scalar score.

Convolutions and MLP are parameterised by a set of weights **θ**.

Overall network models a function over sentences s: $g_{\theta,E}(s) = f_{\theta}(embed_{E}(s))$

Neural Embedding Models: C&W

(Collobert et al. 2011)



What prevents the network from **ignoring input** and outputting high scores?

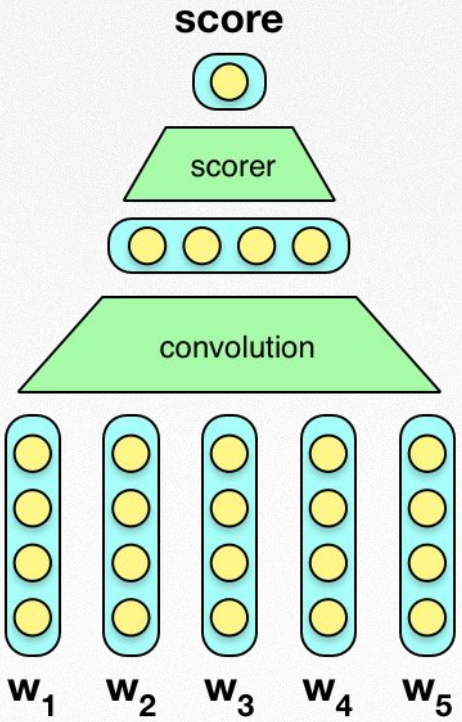
During training, for each sentence s we sample a distractor sentence z by **randomly corrupting** words of s.

Minimise **hinge loss**:

$$L = max(0, 1 - (g_{\theta,\mathbf{E}}(s) - g_{\theta,\mathbf{E}}(z)))$$

Neural Embedding Models: C&W

(Collobert et al. 2011)



Interpretation: representations carry information about what **neighbouring representations** should look like.

A lot like the **distributional hypothesis**?

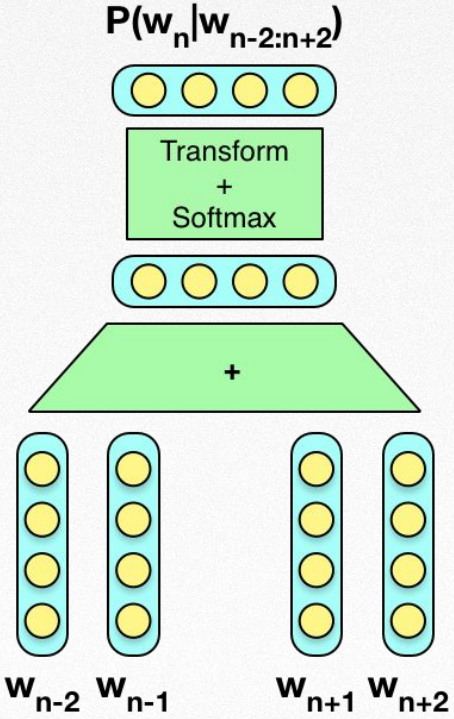
A sensible model but:

Fairly **deep**, so not cheap to train.

Convolutions capture very **local** information.

Neural Embedding Models: CBoW

(Mikolov et al. 2013)



Embed context words. Add them.

Project back to vocabulary size. Softmax.

$$softmax(\mathbf{l})_i = \frac{e^{l_i}}{\sum_j e^{l_j}}$$

$$P(t_i|context(t_i)) = softmax\left(\sum_{t_j \in context(t_i)} onehot_{t_j}^{\top}\ \mathbf{E}\ W_v\right)$$

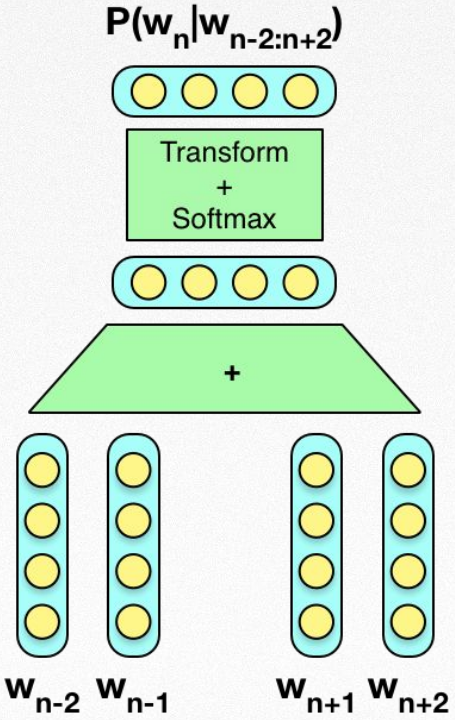
$$= softmax\left(\left(\sum_{t_j \in context(t_i)} onehot_{t_j}^{\top}\ \mathbf{E}\right) W_v\right)$$

Minimize Negative Log Likelihood:

$$L_{data} = -\sum_{t_i \in data} logP(t_i|context(t_i))$$

Neural Embedding Models: CBoW

(Mikolov et al. 2013)



All linear, so very fast. Basically a cheap way of applying one matrix to all inputs.

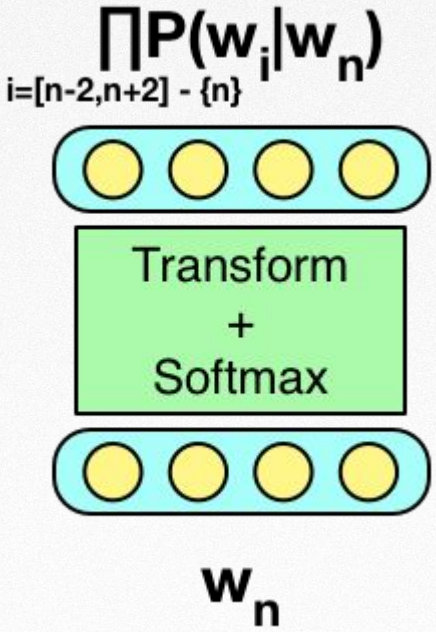
Historically, negative sampling used instead of expensive softmax.

NLL minimisation is more stable and is fast enough today.

Variants: position specific matrix per input (Ling et al. 2015).

Neural Embedding Models: Skip-gram

(Mikolov et al. 2013)



Target word predicts context words.

Embed target word. Project into vocabulary. Softmax.

$$P(t_j|t_i) = softmax(onehot_{t_i}^{\top}\ \mathbf{E}\ W_v)$$

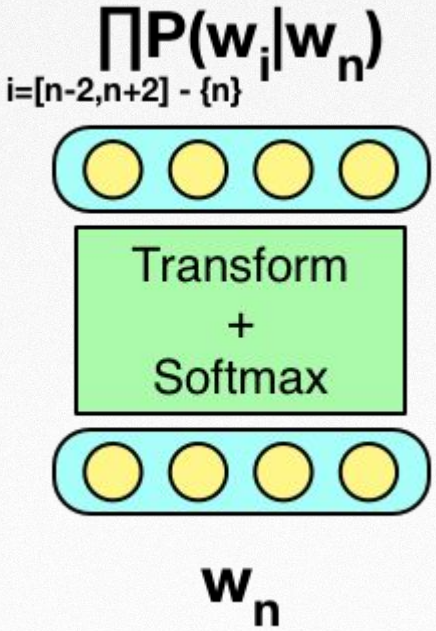
Learn to estimate likelihood of context words.

$$-logP(context(t_i)|t_i) = -log \prod_{t_j \in context(t_i)} P(t_j|t_i)$$

$$- \sum_{t_j \in context(t_i)} logP(t_j|t_i)$$

Neural Embedding Models: Skip-gram

(Mikolov et al. 2013)



Fast: One embedding versus |C| embeddings.

Just read off probabilities from softmax.

Similar variants to CBoW possible: position specific projections.

Trade off between efficiency and more structured notion of context.

Discrete Language Models

Language Modelling Objective

Goal: estimate the **joint probability** of a sequence of symbols (words, characters):

$$P(s_1, s_2, \dots s_n)$$

Usually decomposed left-to-right as follows (right-to-left also doable):

$$P(s_1, s_2, \dots s_n) = \prod_{i=1}^{n} P(s_i | s_1, s_2, \dots s_{i-1}) = \prod_{i=1}^{n} P(s_i | s_{1:i-1})$$

In practice we want to learn a function which models the conditional probability

$$P(s_i | s_{1:i-1})$$

N-gram Language Models

Problem: this probability depends on a *variable length* (unbounded) history of the sequence. How do we model this?

(One) Solution: introduce an order-k markov assumption:

$$P(s_i|s_{1:i-1}) \approx P(s_i|s_{i-k:i-1})$$

Predicting the next word only depends on the last k words.

N-gram Language Models

We can estimate $P(s_i|s_{i-k:i-1})$ using counts. Collect all k+1 grams $s_{i-k:i}$ in the corpus, and then compute:

$$P(s_i|s_{i-k:i-1}) = \frac{count(s_{i-k:i})}{count(s_{i-k:i-1})}$$

Problem: if k is too small, the language model is terrible. If k is too large, the counts are too low to properly estimate probability.

Some solutions address this, such as **smoothing** (e.g. Kneser and Ney, 1995), but overall count-based N-gram models suffer from **sparsity**.

Continuous Conditional Language Models

LBL/NLM

We can estimate the probability using co-learned word representations instead of counts.

$$P(s_i|s_{i-k:i-1}) = softmax\left(W_V \cdot \sigma\left(\sum_{j=i-k}^{i-1} W_j \cdot embed_E(s_j) + b_j\right) + b_V\right)$$

Idea: embed words and project into hidden layer with position-specific matrices and biases, then project into vocabulary and softmax.

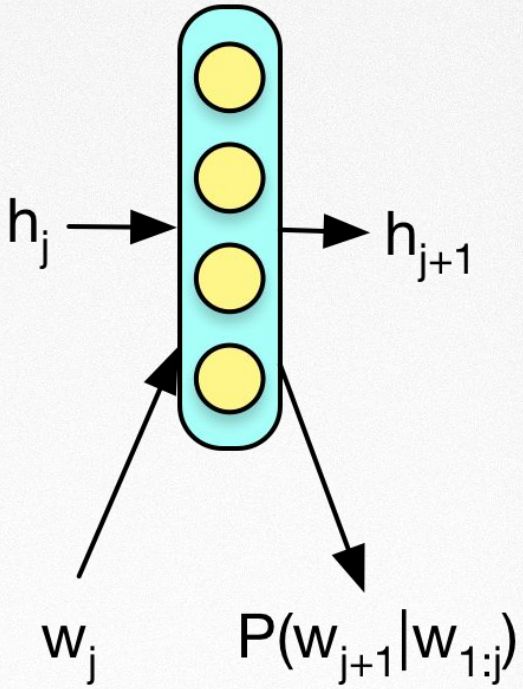
If σ is a non-linearity, this is a **neural language model** (Bengio et al. 2003). If it is the identity, this is a **log-bilinear language model** (Mnih and Hinton, 2007).

RNN LMs

NLM/LBL a step in the right direction, but still rely on order-k Markov assumption.

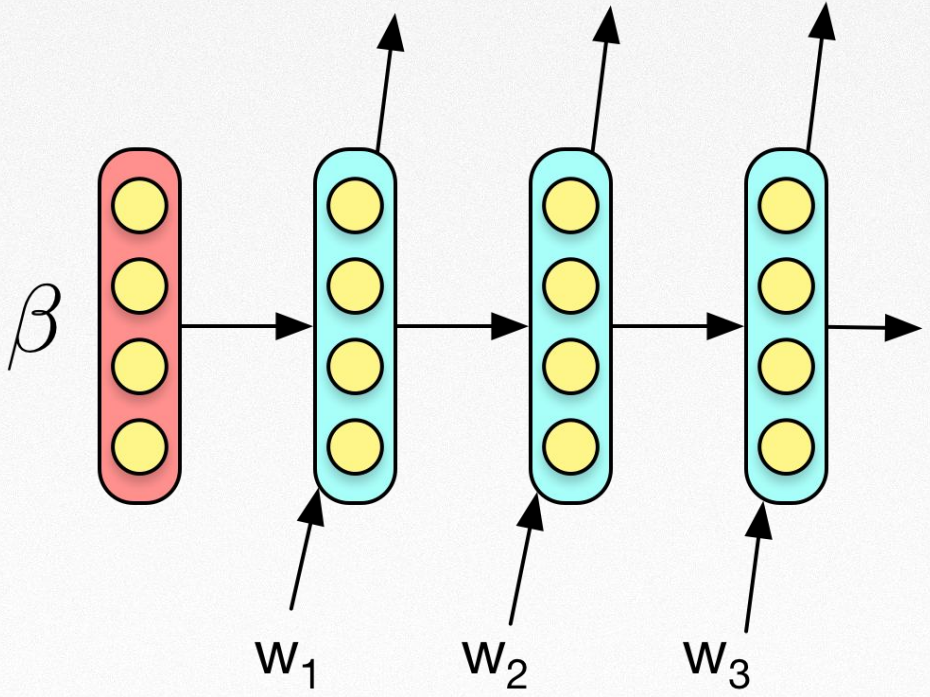
RNNs allow us to relax this assumption by not specifying k. Technically hidden layer activations model the complete history of the sequence.

In practice, they still model some (softly) bounded history, the length of which is a factor of the data, cell size, recurrent update structure, data, etc.



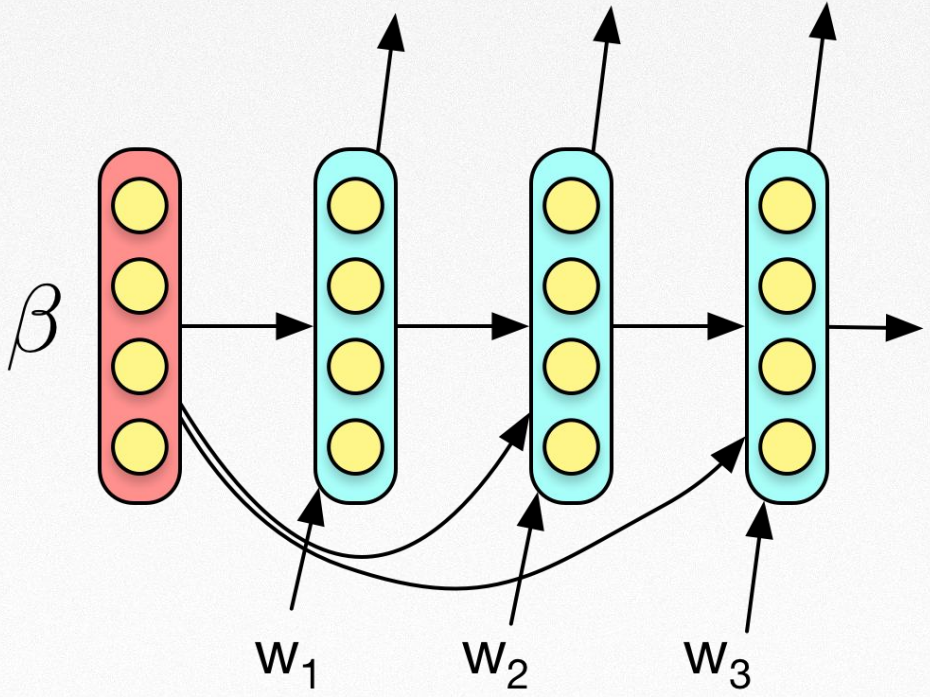
Transduction with Conditional Models

$$P(\mathbf{w}|\beta)$$



Transduction with Conditional Models

$$P(\mathbf{w}|\beta)$$



Sequence to Sequence Mapping with RNNs

Many NLP (and other!) tasks are castable as transduction problems. E.g.:

Translation: English to French transduction

Parsing: String to tree transduction

Computation(?!): Input data to output data transduction

Sequence to Sequence Mapping with RNNs

Generally, goal is to transform some source sequence

$$\mathbf{s} = s_1, s_2, \ldots, s_m$$

into some target sequence

$$\mathbf{t} = t_1, t_2, \ldots, t_n$$

Sequence to Sequence Mapping with RNNs

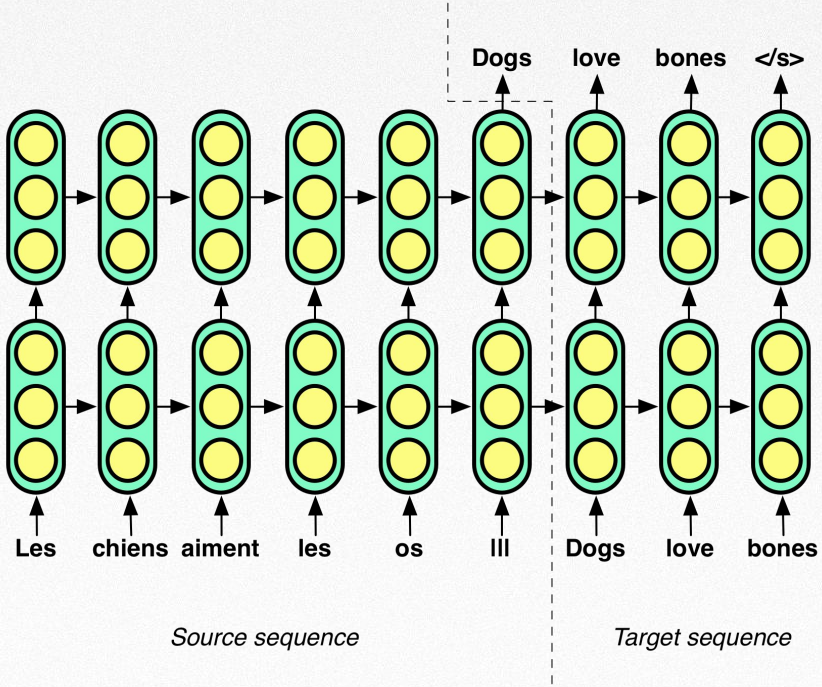
Represent **s** and model $P(t_{i+1}|t_1...t_n; \mathbf{s})$ with RNNs:

1. Read in source sequence to produce **s**.
2. Train model to maximise the likelihood of **t** given **s**.
3. Test time: Generate target sequence **t** (greedily, beam search, etc) from **s**.

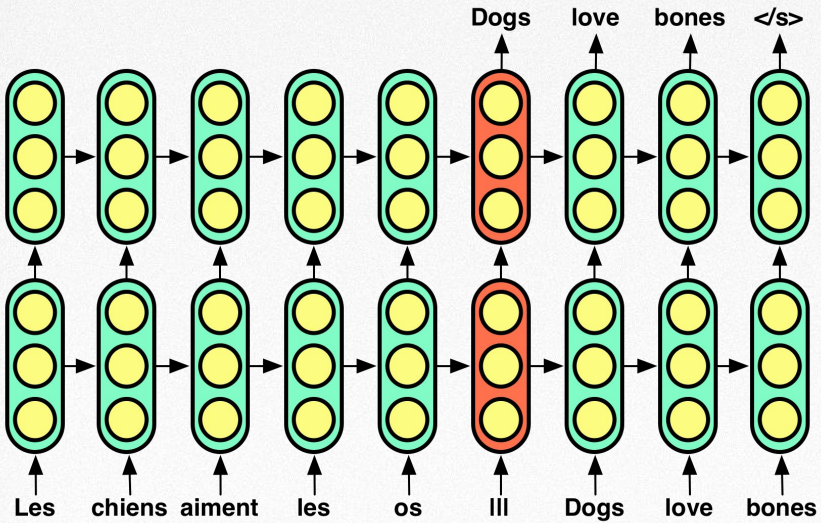
Deep LSTMs for Translation

(Sutskever *et al.* 2014)

$$P(\text{some english}|\text{du français})$$



The Bottleneck for Simple RNNs

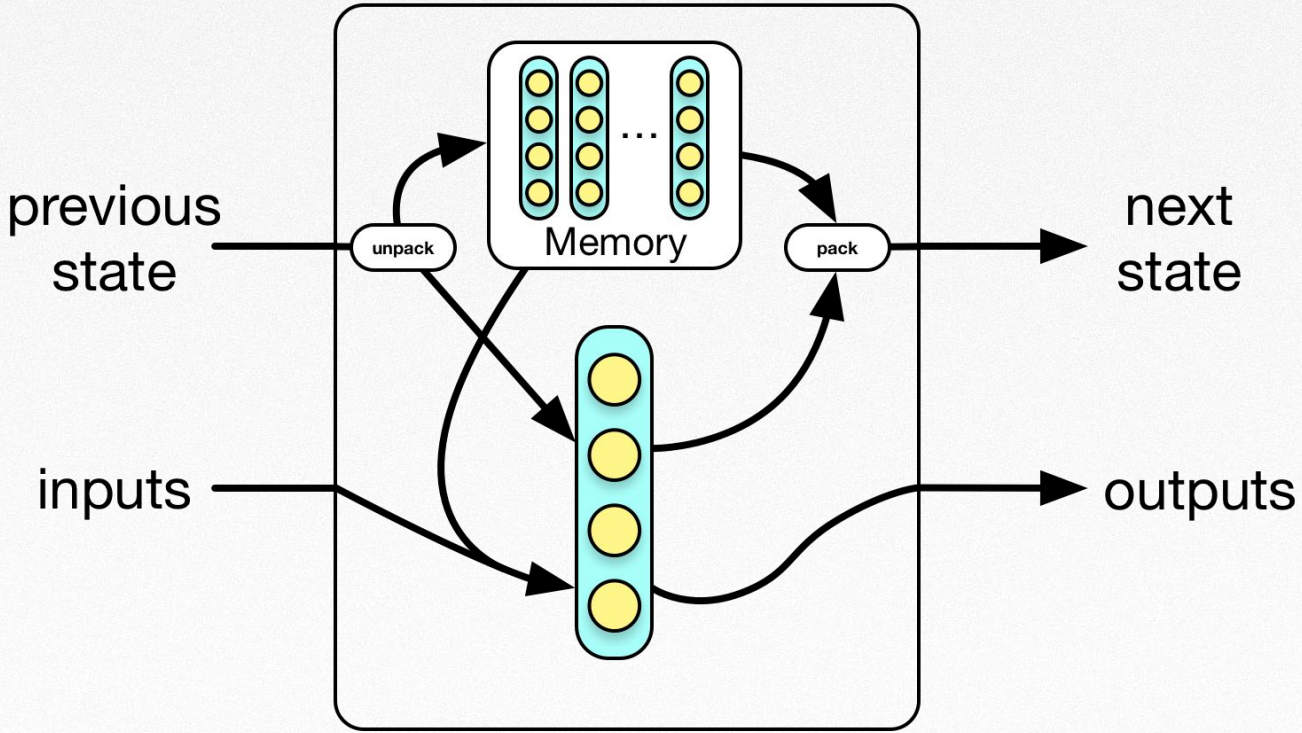


- Non-adaptive capacity
- Target sequence modelling dominates training
- Gradient-starved encoder
- Fixed size considered harmful?

Attention

- You have an array of vectors (produced by encoder) representing some data.
- Your controller (decoder cell) deals with I/O logic.
- You want to read your data array at each timestep.
- You want to accumulate gradients in your memory.

Attention (Early Fusion)



RNN: $X \times P \rightarrow Y \times N$

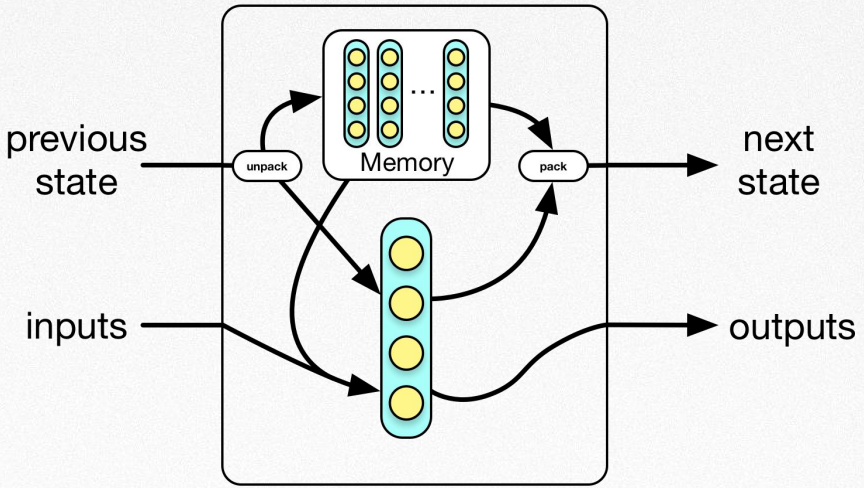
$p_t = (h_{t-1}, M)$

$y_t, h_t = \text{controller}(xs, h_{t-1})$

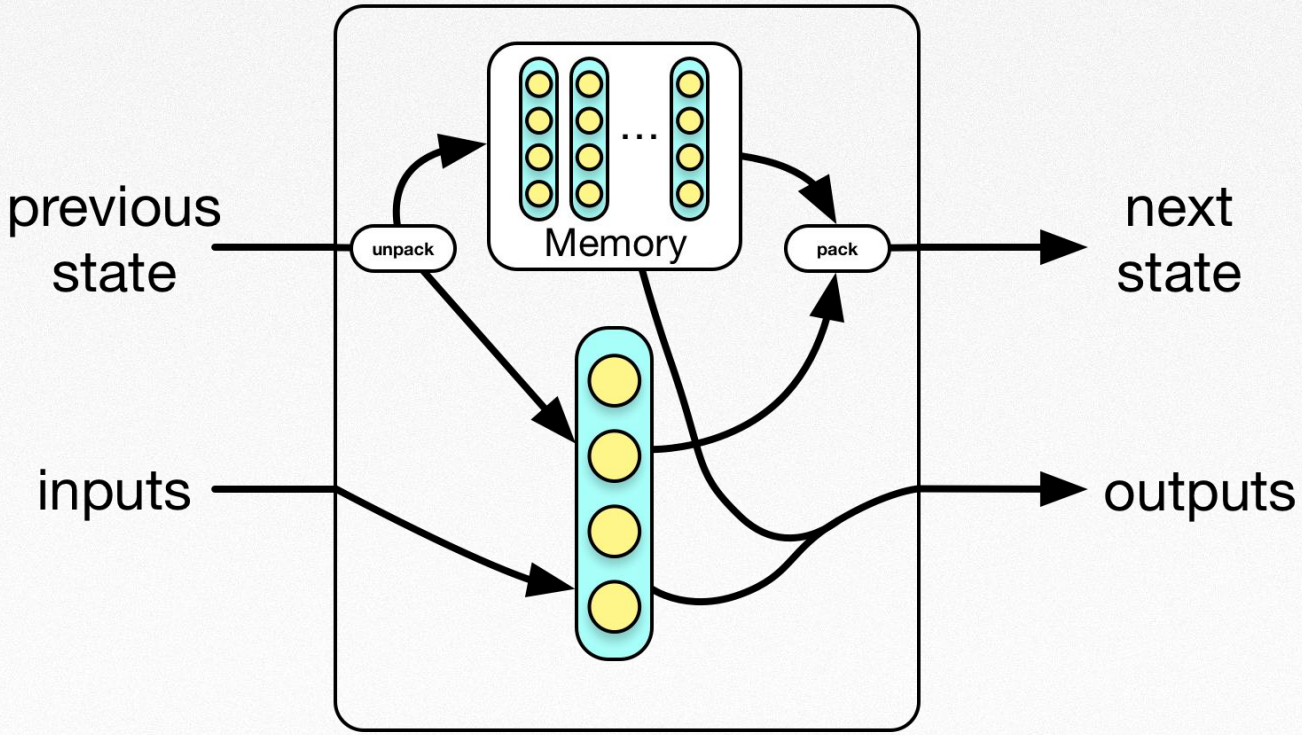
$n_t = (h_t, M)$

$xs = x_t \oplus f_{att}(h_{t-1}, M)$

e.g. $f_{att}(h, M) = M \times \text{softmax}(h \times W_{att} \times M)$



Attention (Late Fusion)



RNN: $X \times P \rightarrow Y \times N$

$p_t = (h_{t-1}, M)$

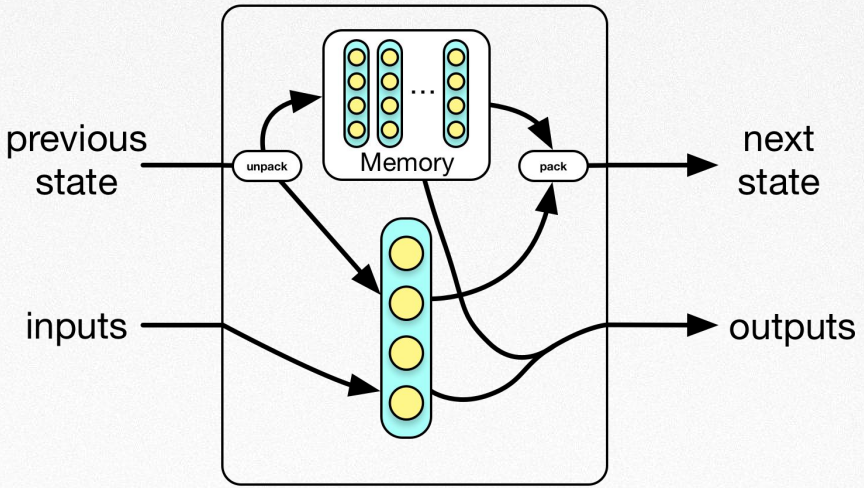
$n_t = (h_t, M)$

$w_t, h_t = \text{controller}(x_t, h_{t-1})$

$y_t = f_{comp}(w_t, f_{att}(h_t, M))$

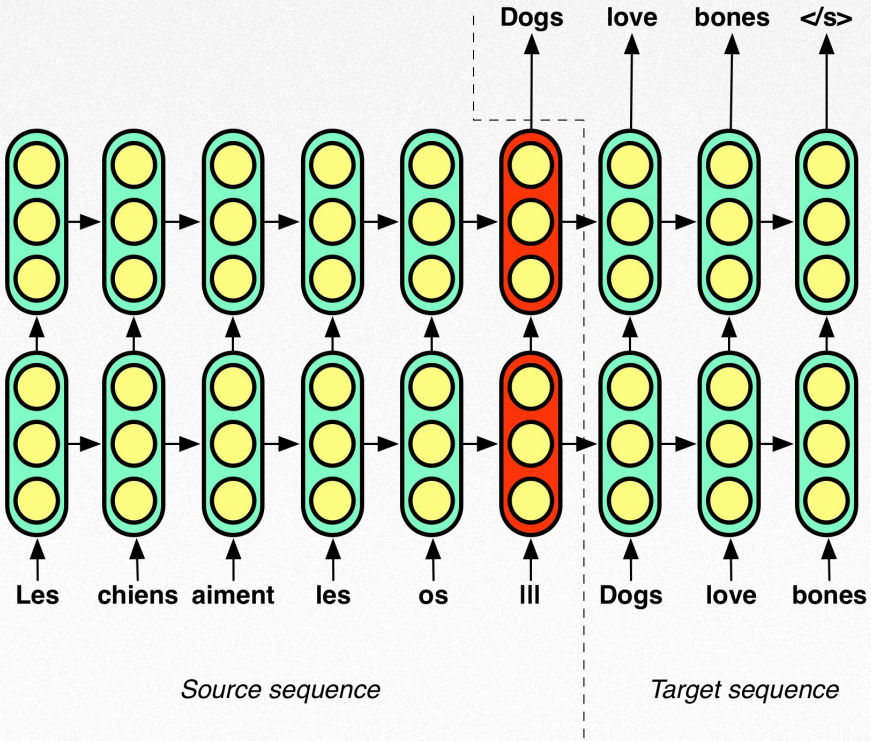
Alternatively:

$y_t = f_{att}(h_t, M)$ if f_{att} yields a distribution over memory positions.



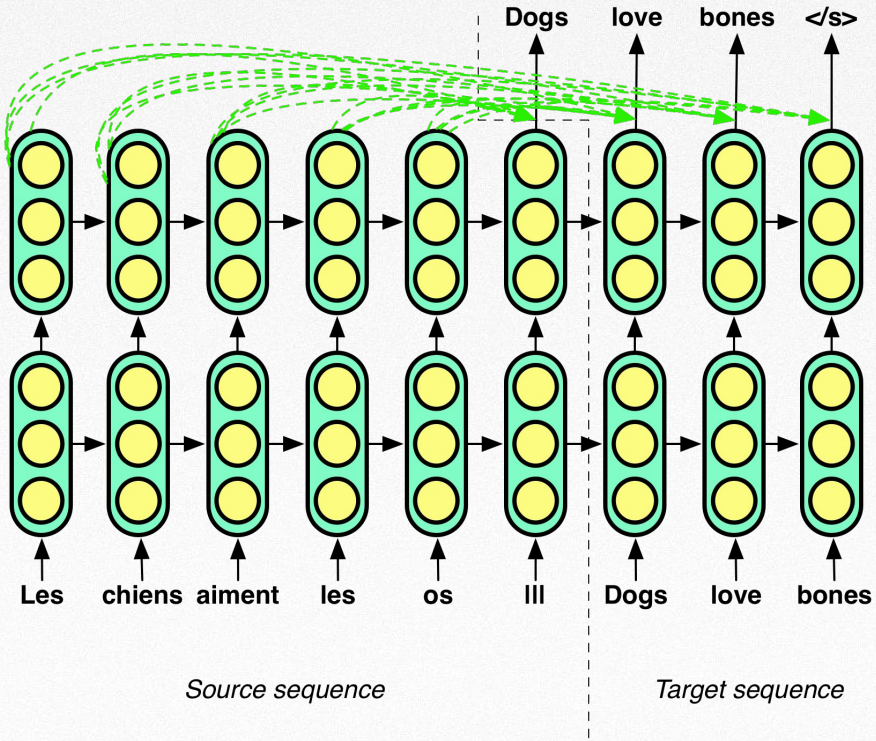
Applications of Seq2Seq + Attention in NLP

Skipping the bottleneck



Skipping the bottleneck

(Bahdanau *et al.* 2014)



Recognizing Textual Entailment (RTE)

(Bowman *et al.* 2015)

A wedding party is taking pictures

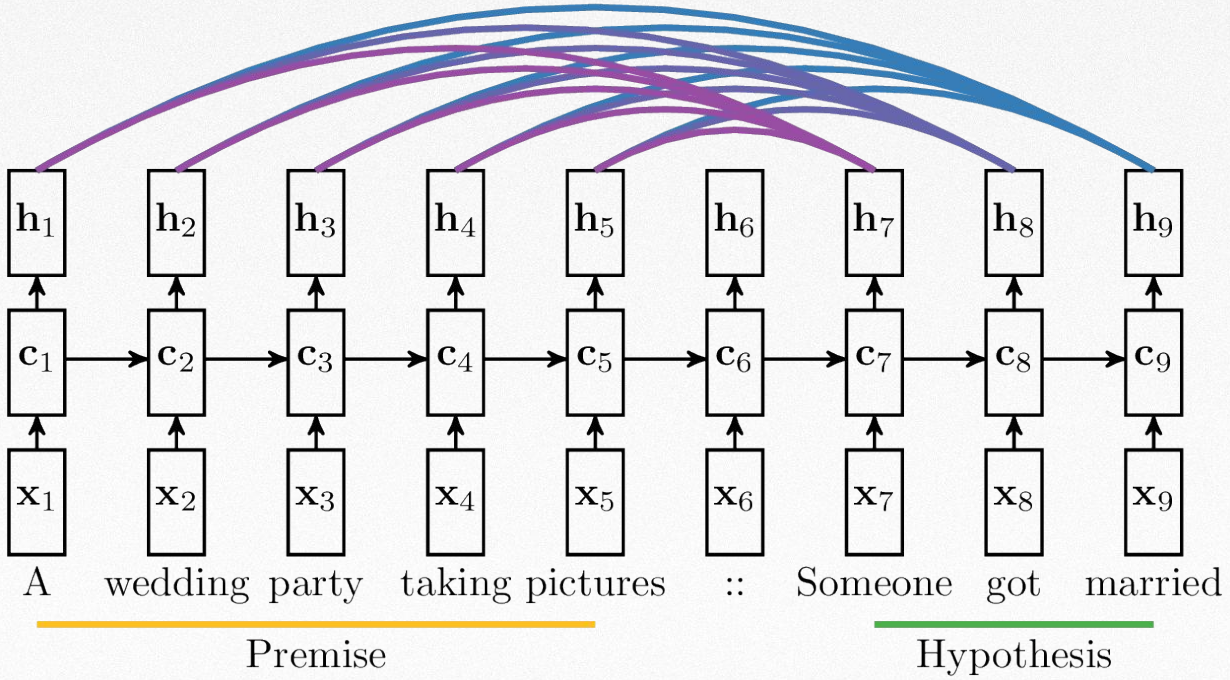
- There is a funeral — **Contradiction**
- They are outside — **Neutral**
- Someone got married — **Entailment**

A man is crowd surfing at a concert

- The man is at a football game — **Contradiction**
- The man is drunk — **Neutral**
- The man is at a concert — **Entailment**

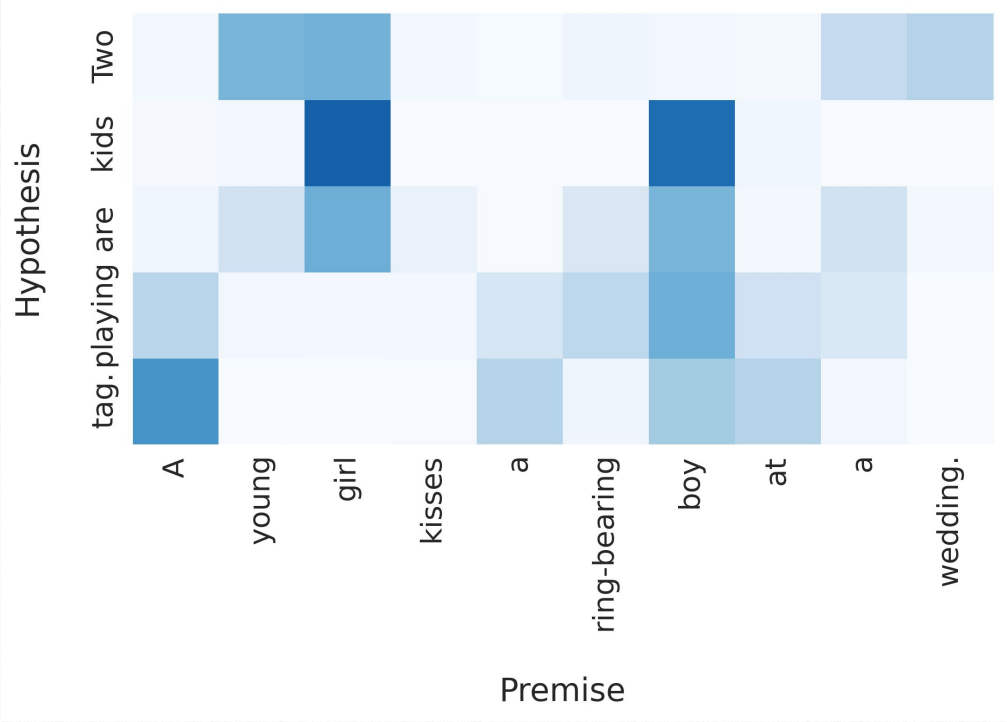
Word-by-Word Attention

(Rocktäschel *et al.* 2015)



Girl + Boy = Kids

(Rocktäschel *et al.* 2015)



Large-scale Supervised Reading Comprehension

(Hermann *et al.* 2015)

The BBC producer allegedly struck by Jeremy Clarkson will not press charges against the "Top Gear" host, his lawyer said Friday. Clarkson, who hosted one of the most-watched television shows in the world, was dropped by the BBC Wednesday after an internal investigation by the British broadcaster found he had subjected producer Oisin Tymon "to an unprovoked physical and verbal attack." …

Cloze-style question:

Query: Producer **X** will not press charges against Jeremy Clarkson, his lawyer says.

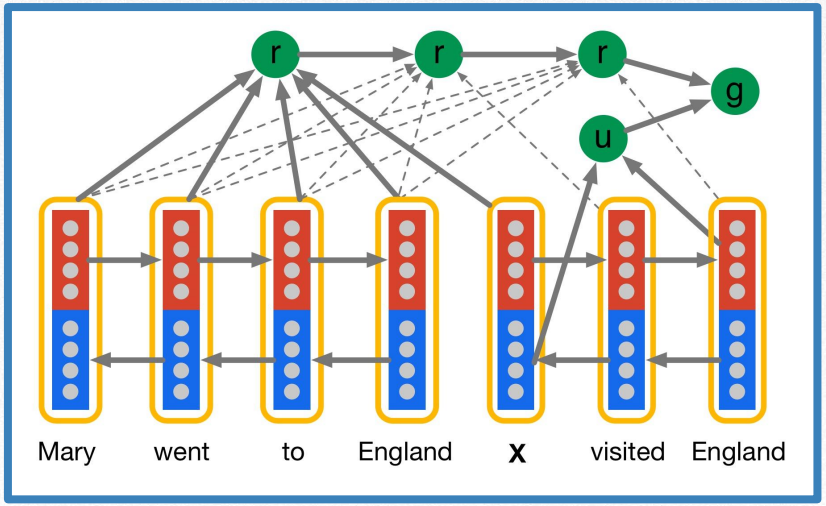
Answer: Oisin Tymon

Machine Reading with Attention

(Hermann *et al.* 2015)

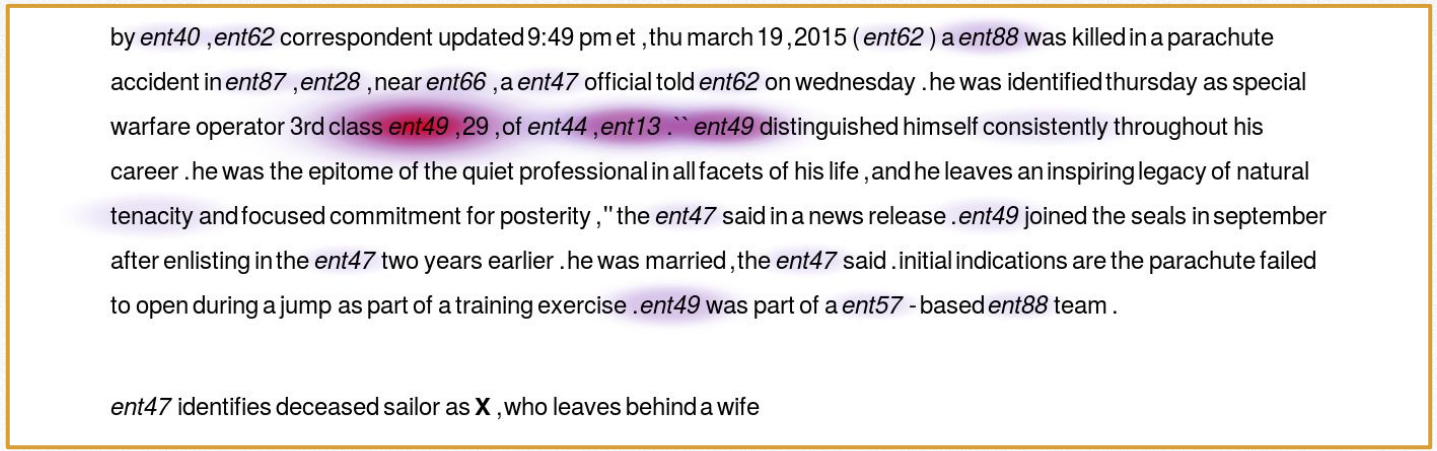
- Create embedding for each token in document
- Read query word by word
- Attend over document at each step through query
- Iteratively combine attention distribution
- Predict answer with increased accuracy

The Impatient Reader



Example QA Heatmap

(Hermann *et al.* 2015)



Correct prediction (***ent49***) - Requires anaphora resolution

Semantic Composition

Classification Tasks

Text classification tasks involve the prediction of a **label** or **scalar value** from a sentence, set of sentences, documents, etc.

Typically, sentence or document representations are produced using a differentiable **composition function** over the embeddings of constituent words.

Sentence/document representations are then fed to a **differentiable classifier** or **regressor** (e.g. linear map, MLP, etc.).

Classification/regression loss used to **update** classifier parameters, composition function parameters, embeddings.

Algebraic Methods

Obtain sentence/document representation by algebraic operations. E.g.:

$$\mathbf{s}_{1:T} = \sum_{i=1}^{T} \mathbf{s}_i \qquad \mathbf{s}_{1:T} = \bigodot_{i=1}^{T} \mathbf{s}_i \qquad \mathbf{s}_{1:T} = \sum_{i=1}^{T-1} \tanh\left(\mathbf{s}_i + \mathbf{s}_{i+1}\right)$$

Pros: No added parameters. **Fast** and simple. **Representations** do the work.

Cons: Not very flexible. Large strain on representations.

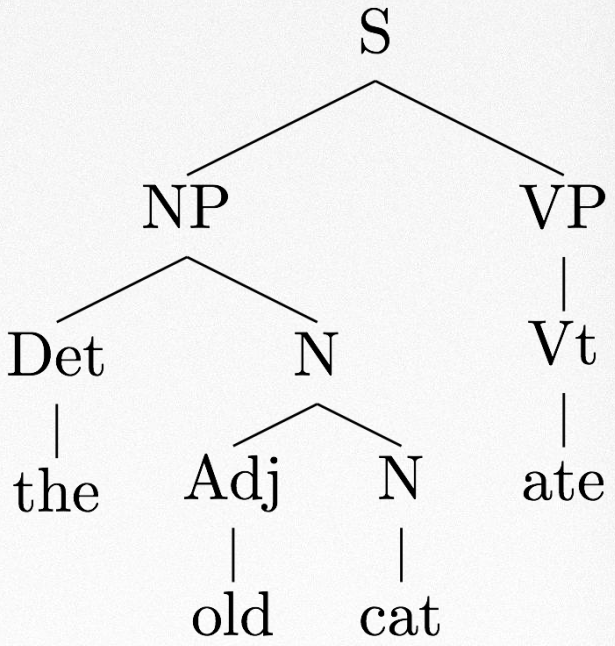
Solution: RNNs? Sure, but what other options are there?

TreeRNNs

Many linguists argue that language is **tree-structured**.

Each sentence has a latent **parse tree**.

There exist parsers that will give us a **distribution over possible parse trees** which we could exploit to guide composition.



TreeRNNs

Basic idea: use sampled or maximally likely parse tree to guide composition:

$$\mathbf{parent} = f_\theta(\mathbf{child}_1, \ldots, \mathbf{child}_n)$$

Perhaps additionally condition on the production rule:

$$\mathbf{parent} = f_\theta(\mathbf{child}_1, \ldots, \mathbf{child}_n; \mathrm{A} \rightarrow \mathrm{B\ C})$$

f can be any differentiable function. There are RNN-style and LSTM-style updates in various papers.

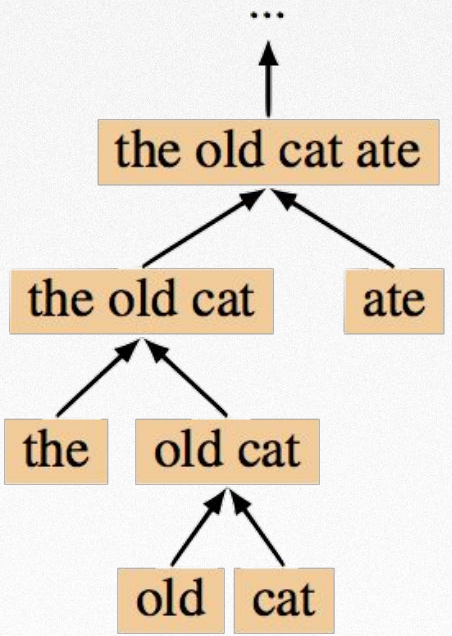


Figure due to Sam Bowman.
Reproduced with author's permission.

TreeRNNs

We have **parse tree-annotated** text classification corpora (e.g. Stanford Sentiment Treebank) and can use parsers to **annotate** other text classification corpora (and preferably do some manual cleaning).

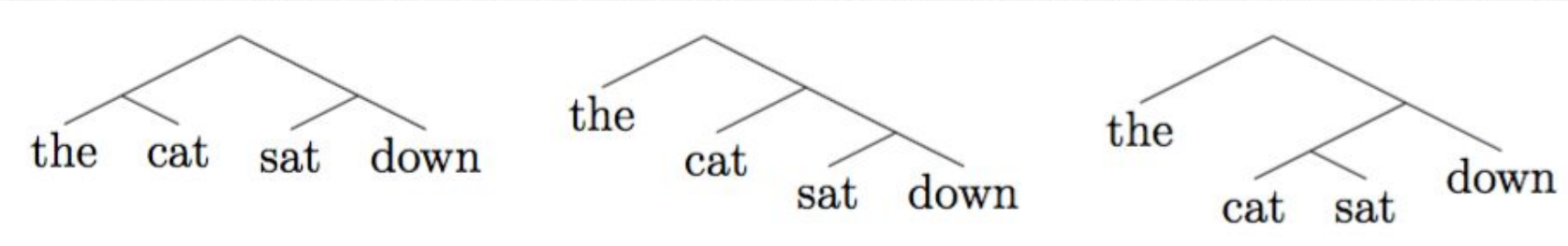
But we need trees at test time too! Annoying to have to **parse target data**, and sometimes quite **difficult** (e.g. tweets).

Solution: jointly train composition operations with "parser"-like composer.

SPINN

(Bowman *et al*. 2016)

Figure due to Sam Bowman.
Reproduced with author's permission.

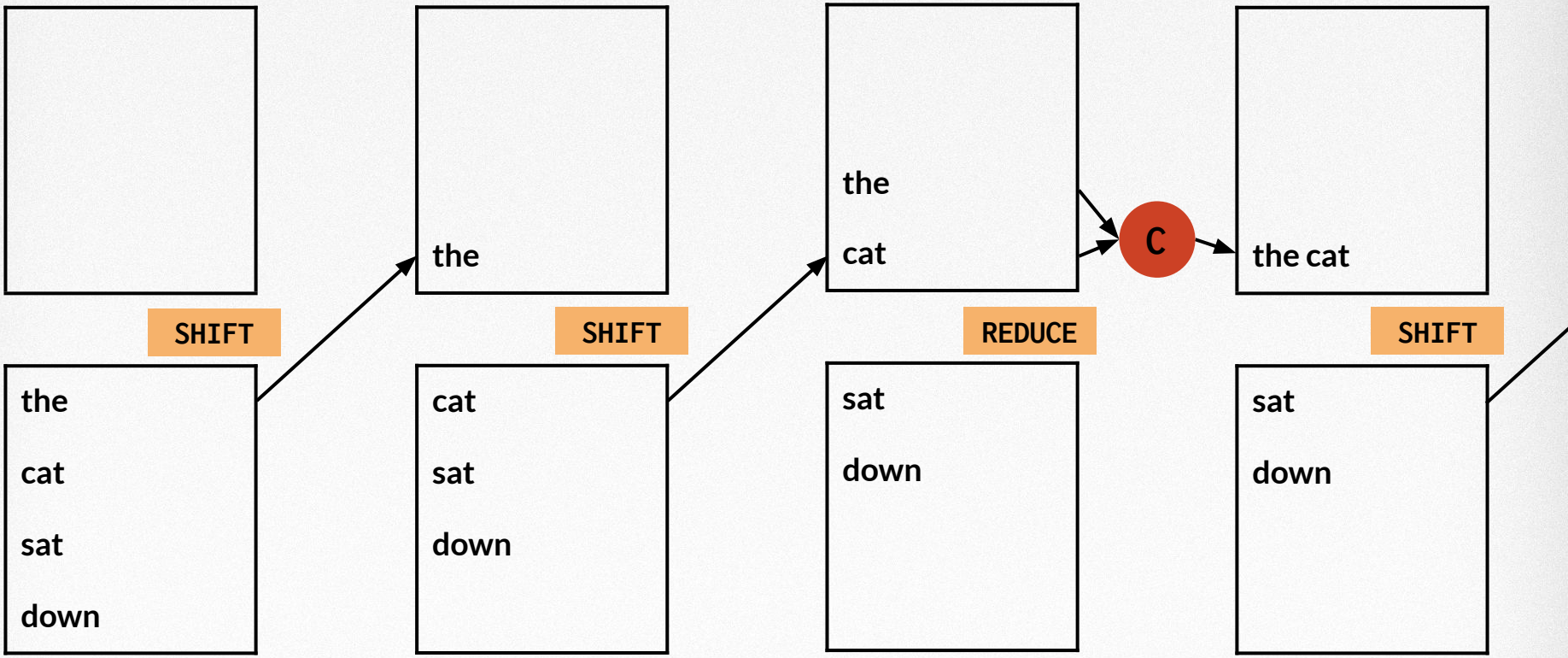


```
[SHIFT, SHIFT,
REDUCE, SHIFT,
SHIFT, REDUCE,
REDUCE]
```

```
[SHIFT, SHIFT,
SHIFT, SHIFT,
REDUCE, REDUCE, REDUCE]
```

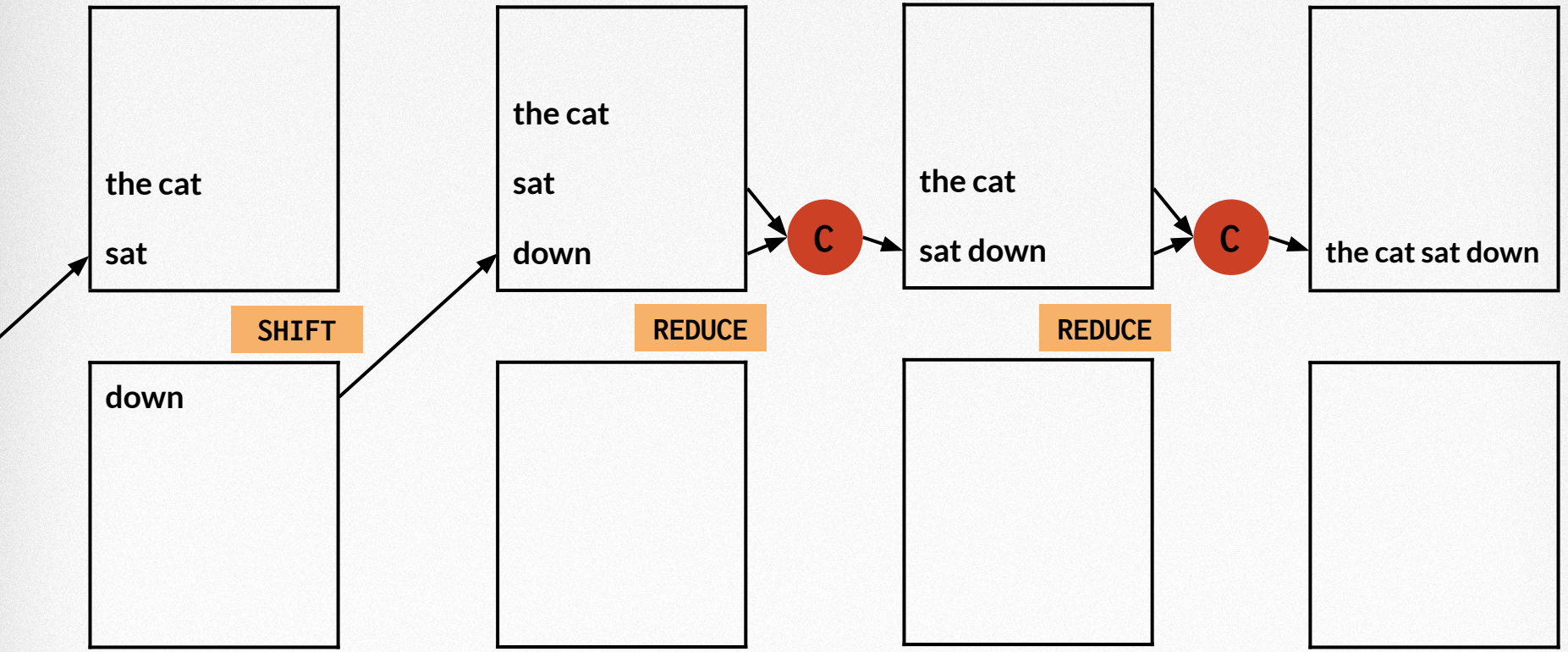
```
[SHIFT, SHIFT,
SHIFT, REDUCE,
SHIFT, REDUCE, REDUCE]
```

SPINN


Stack
Figure due to Sam Bowman.
Reproduced with author's permission.
SHIFT
SHIFT
REDUCE
SHIFT
C
the
cat
sat
down
the
cat
sat
down
the
cat
sat
down
the cat
sat
down
Buffer

SPINN

Figure due to Sam Bowman.
Reproduced with author's permission.



SPINN

(Bowman *et al*. 2016)

High level idea: decide shift or reduce as a function of buffer head and top two elements of the stack.

If shift, move the **vector embedding** of the **top buffer word** to the stack.

If reduce: **remove and compose** top two stack vectors, and **push result** to stack.

Stack head after **2T-1 steps** (for length T seq.) is the sentence representation.

Learn sentence representation from classification loss, and shift-reduce decisions from parse. Both jointly **conditioned on word representations**.

RL-SPINN

(Yogatama et al. 2016)

What if we don't want to rely on trees during training? **Solution:** RL-SPINN.

Idea: learn a policy network $\pi(a|s, W_R)$ from which we sample $a \in \{\text{shift}, \text{reduce}\}$ using your favourite RL method, e.g. REINFORCE (Williams, 1992).

Jointly learn to produce **parse trees** and **exploit them during composition**, based on how well the composed representations serve the end-task.

Often, these trees do not resemble those linguists suggest, but are **better suited to the specific task**.

Wrap-Up

Conclusions

Language is a **difficult topic**. Many interesting high-level research problems.

Deep Learning for NLP is **relatively new**. Many opportunities for progress.

Many Deep Learning for NLP courses are putting their slides/videos online:
https://github.com/oxford-cs-deepnlp-2017/lectures (slides, videos)
http://cs287.fas.harvard.edu/ (notes, slides)
https://github.com/stanfordnlp/cs224n-winter17-notes (notes, slides)

THANK YOU