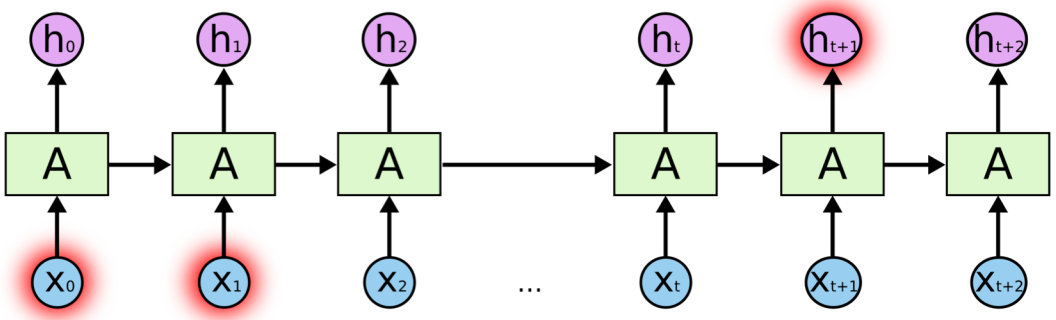
COMP 551 - Applied Machine Learning Lecture 17 — RNNs continued

William L. Hamilton

(with slides and content from Joelle Pineau)

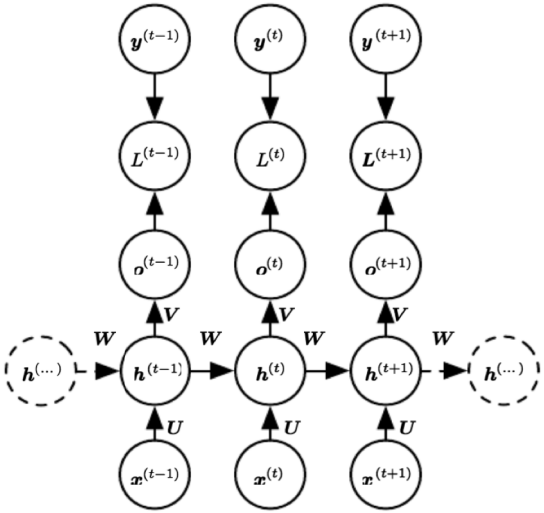
The problem of long-term dependencies

- Let's say we are doing language modelling
- Input paragraph: *"I grew up in France. I worked at […]. I speak fluent French."*
- Want to predict 'French' given words before. This can be hard!
- In practice it is very hard for RNNs to to learn dependencies lasting many time steps.
- Why could this be?



The problem of long-term dependencies

- Because the hidden-to-hidden transition matrix $\mathbf{W}$ is **the same for each time step**, this can cause the gradients to explode or vanish
- Intuition: Imagine multiplying a scalar number w by itself many times. w^k for $k \to \infty$ will either explode (if $w > 1$) or vanish (if $w < 1$)
- Similar behavior occurs if $\mathbf{W}$ is a matrix



$$\mathbf{h}_t = \sigma(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b})$$

$$\text{E.g., } \mathbf{h}_3 = \sigma(\mathbf{W}(\sigma(\mathbf{W}\sigma(\mathbf{W}\mathbf{h}_0 + \mathbf{U}\mathbf{x}_1 + \mathbf{b}) + \mathbf{U}\mathbf{x}_2\mathbf{b}) + \mathbf{U}\mathbf{x}_3 + \mathbf{b})$$

The problem of long-term dependencies

- Recall: a way to intuitively think of backpropagating gradients

- If I change my input by a small amount, what will be the result on the output?

  $\Leftrightarrow$

  If I want my output (loss) to decrease, how do I change my input?

- If input is being multiplied by the same $\mathbf{W}$ many times, this could cause either a huge or tiny effect on the output.

  =>

  The gradient of loss w.r.t parameters could be huge or tiny.

The problem of long-term dependencies

- Perspective from linear algebra (eigendecomposition)
- Consider a simplified "linear" RNN with following recurrence:

$$\mathbf{h}_t = \mathbf{W}\mathbf{h}_{t-1}$$

The problem of long-term dependencies

- Perspective from linear algebra (eigendecomposition)
- Consider a simplified "linear" RNN with following recurrence:

$$\mathbf{h}_t = \mathbf{W}\mathbf{h}_{t-1}$$

- Now, we can get the eigendecomposition of $\mathbf{W}$ as:

$$\mathbf{W} = \mathbf{Q}\mathbf{D}\mathbf{Q}^\top$$

where $\mathbf{Q}$ is an orthogonal matrix of eigenvectors and $\mathbf{D}$ a matrix with eigenvalues on the diagonal.

The problem of long-term dependencies

- Perspective from linear algebra (eigendecomposition)
- Consider a simplified "linear" RNN with following recurrence:

$$\mathbf{h}_t = \mathbf{W}\mathbf{h}_{t-1}$$

- Now, we can get the eigendecomposition of $\mathbf{W}$ as:

$$\mathbf{W} = \mathbf{Q}\mathbf{D}\mathbf{Q}^\top$$

where $\mathbf{Q}$ is an orthogonal matrix of eigenvectors and $\mathbf{D}$ a matrix with eigenvalues on the diagonal.

- And, thus:

$$\begin{aligned}\mathbf{h}_t &= \mathbf{W}^t\mathbf{h}_0 \\ &= (\mathbf{Q}\mathbf{D}\mathbf{Q}^\top\mathbf{Q}\mathbf{D}\mathbf{Q}^\top\ldots)\mathbf{h}_0 \\ &= \mathbf{Q}\mathbf{D}^t\mathbf{Q}^\top\mathbf{h}_0 \qquad \text{since } \mathbf{Q}^\top\mathbf{Q} = \mathbf{I}\end{aligned}$$

- So each eigenvalue is raised to the power of $\mathbf{t}$, causing eigenvalues < 1 to vanish and eigenvalues > 1 to explode.

How to avoid vanishing/exploding gradients?

- Simple way to avoid exploding gradients: gradient clipping

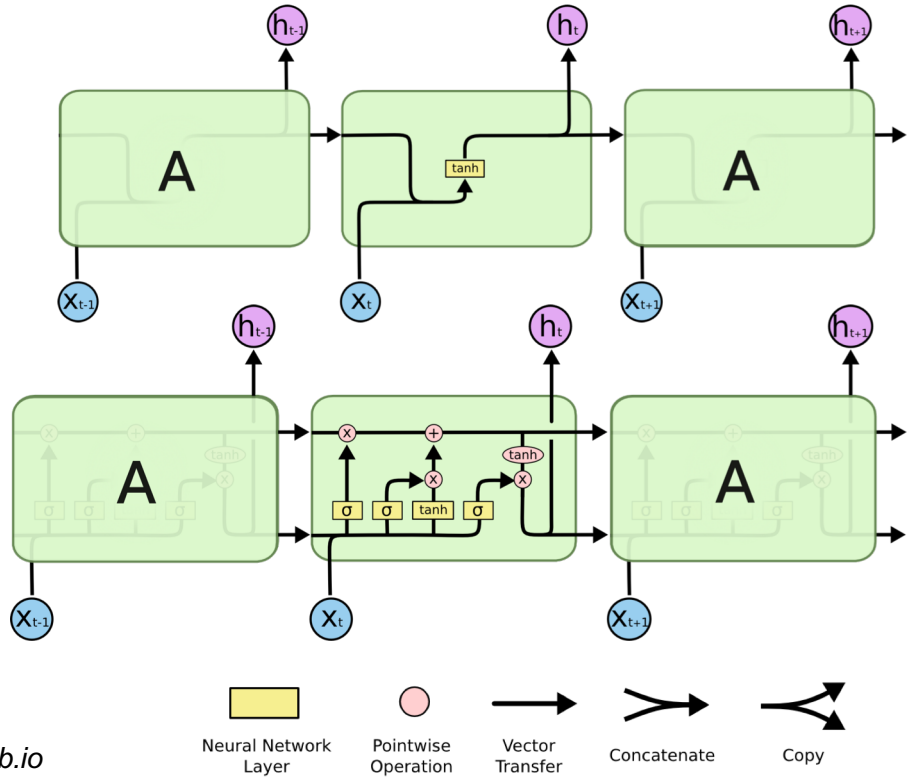
if $|gradient| > threshold$:
    $gradient = threshold * \text{sign}(gradient)$

- Another way: change the architecture of the RNN so there are some non-multiplicative interactions
  - E.g., long short-term memory (LSTM) units

Long short-term memory (LSTM) units

RNN:

LSTM:



LSTM images from: colah.github.io

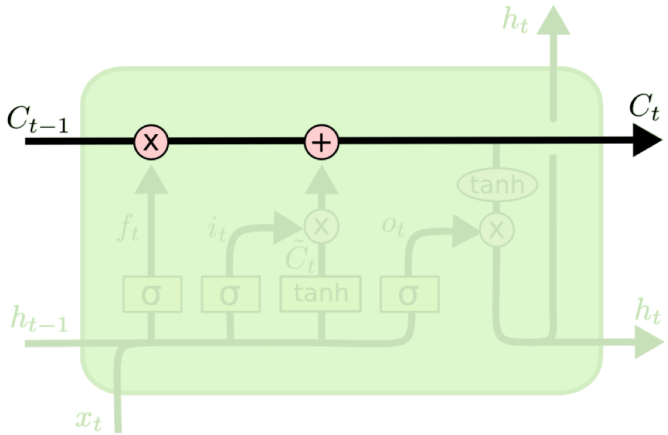
Long short-term memory (LSTM) units

- Much better at dealing with long-term dependencies
- Can think of it as a special 'cell'
- Governed by a set of update equations:

$$
\begin{aligned}
f_t &= \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right) \\
i_t &= \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right) \\
\tilde{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C) \\
C_t &= f_t * C_{t-1} + i_t * \tilde{C}_t \\
o_t &= \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right) \\
h_t &= o_t * \tanh\left(C_t\right)
\end{aligned}
$$

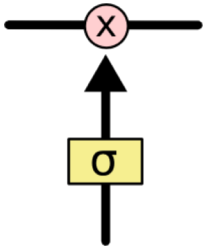
LSTMs

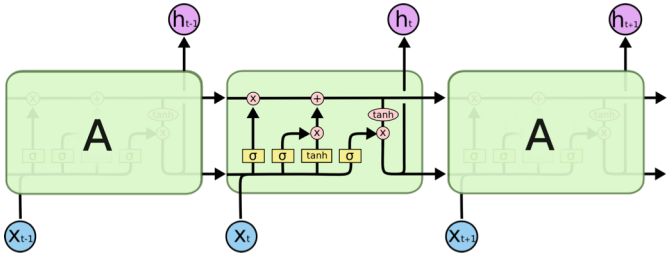
- Core idea: the **cell state is an** an 'information highway'
- Cell state is updated additively based on input, rather than multiplicatively => *less prone to exploding/ vanishing gradients*



LSTMs: Cell states, hidden states, and gating

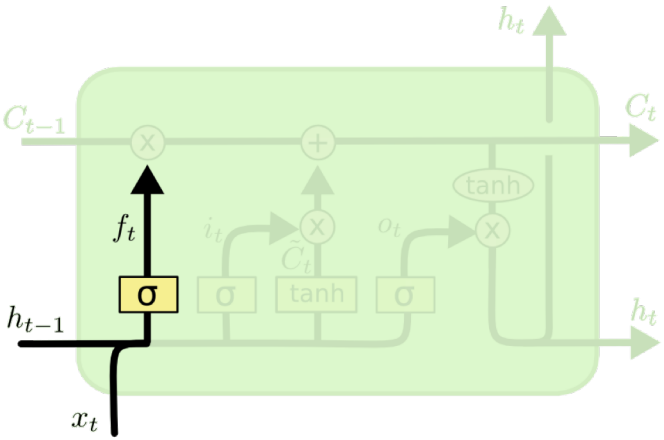
- Cell state vs hidden state (roughly)
  - Hidden state: what info from past do I need **to make my next prediction**?
  - Cell state: what info from past might I need **to make future predictions**?
- For regular RNN, hidden state plays both of these roles
- LSTM uses a set of '**gates**' to control information flow
  - Gate = sigmoid layer + element-wise multiplication. Gives vector of numbers between [0,1] that determine how much of each component to let through:





LSTMs: Forget gate

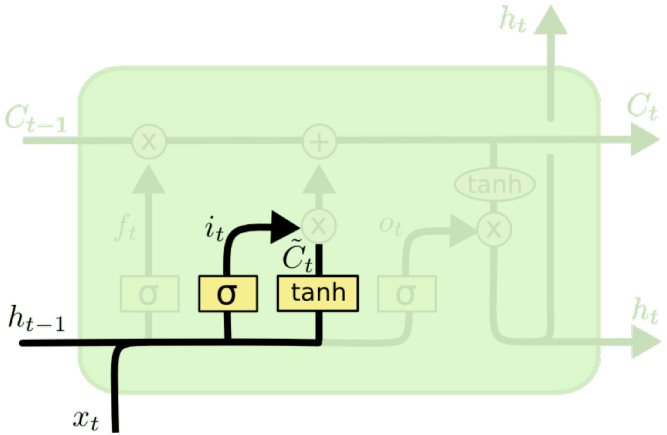
- Forget gate: how much information do we want to keep from the previous cell state?



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

LSTMs: Input gate

- Input gate: what information from the current input (and previous hidden state) do we want to transfer to the cell state?

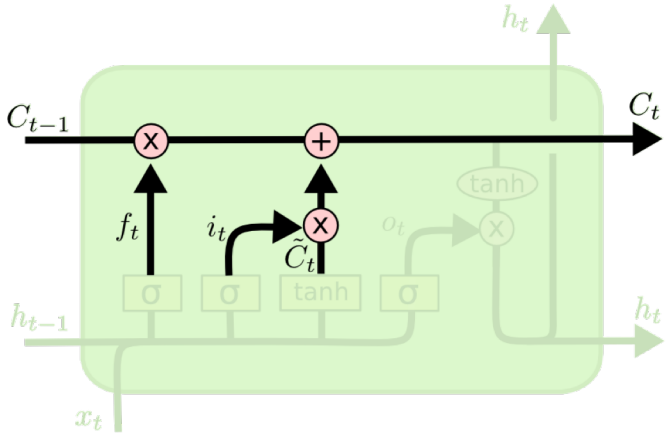


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

LSTMs: Cell update

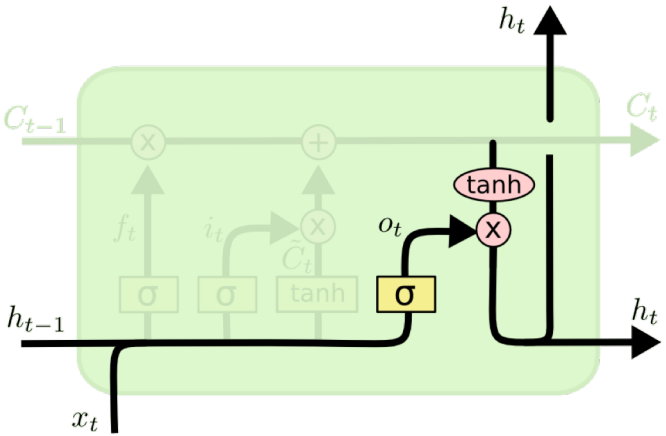
- Cell state updated as an **additive linear combination** of old cell state and processed input:



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

LSTMs: Output gate:

- Output gate: what information from the cell state do we need to make the next prediction?



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

LSTMs

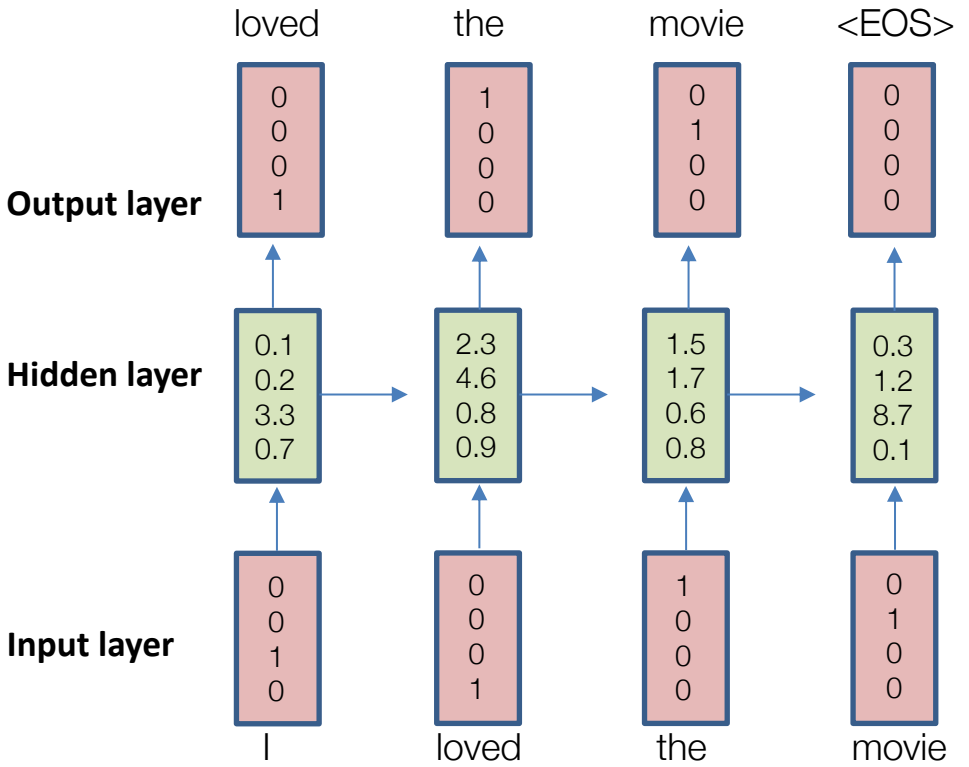
- LSTM architecture has existed for many years (Hochreiter & Schmidhuber 1997).

- Many state-of-the-art results, e.g.,
  - Cursive handwriting recognition (Graves & Schmidhuber, 2009)
  - Speech recognition (Graves, Mohamed & Hinton, 2013)
  - Machine translation (Sutskever, Vinyals & Le, 2014)
  - Question-answer (Weston et al., 2015)
  - Unstructured dialogue response generation (Serban et al., 2016)

- Other similar models can be used (e.g. Gated Recurrent Units)

Recall: Language modeling

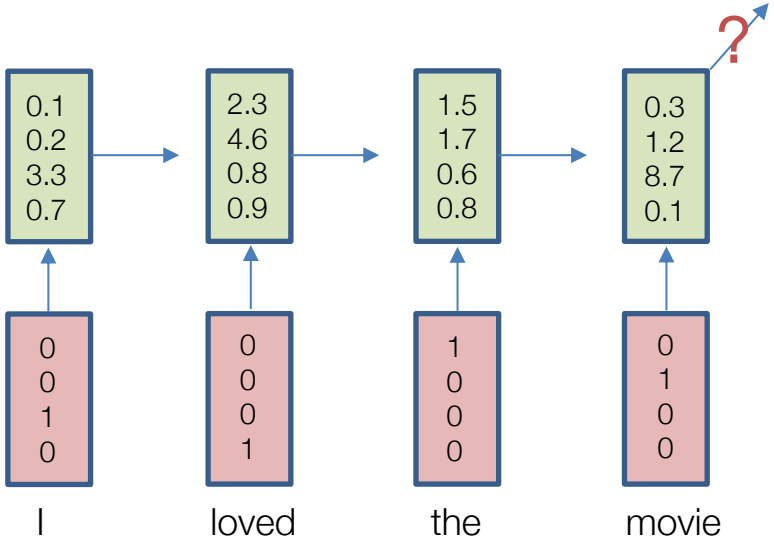
- **Input:** Sequence of words
  - E.g., review, tweet, or news article
- **Output:** Sequence of words
  - E.g., predicting the next word that will occur in a sentence.



Generating sequences with RNNs

- With a single RNN/LSTM, we can produce output sequences that have the same length as the input sequence
- Question: What if we want to go beyond language modelling, and produce an output with a different length from our input?
- E.g.: machine translation, image captioning
- Solution: *Use two RNNs!* One to encode the input, and one to produce the output

Example: Machine translation



J'ai adoré le film

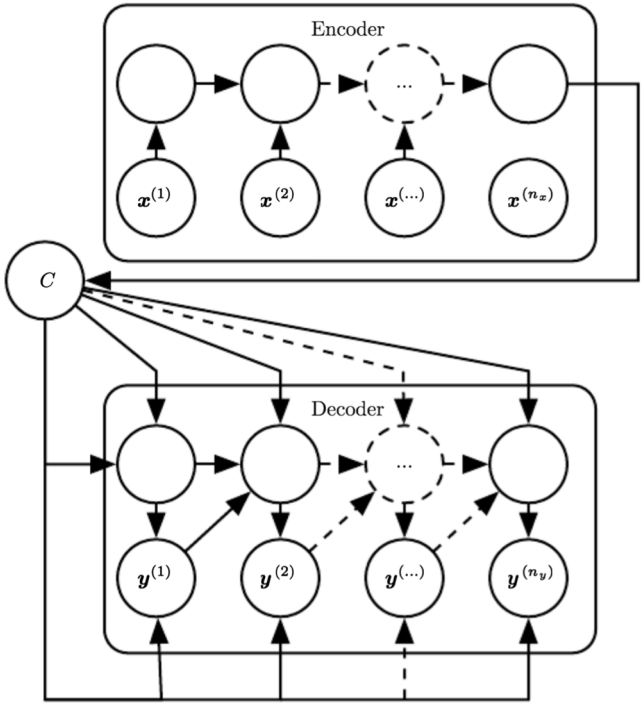
Ich habe den Film geliebt

Jeg elskede filmen

Generating Sequences with RNNs

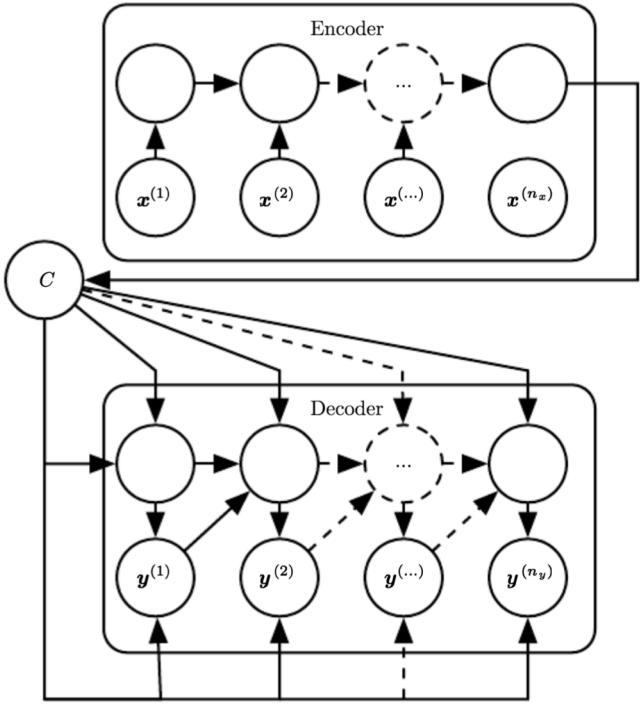
- Basic Idea:
  - Use an 'encoder' RNN to transform the input into a vector!
  - Then, use a 'decoder' RNN to generate a sequence from the encoded representation.

- This is the encoder-decoder model



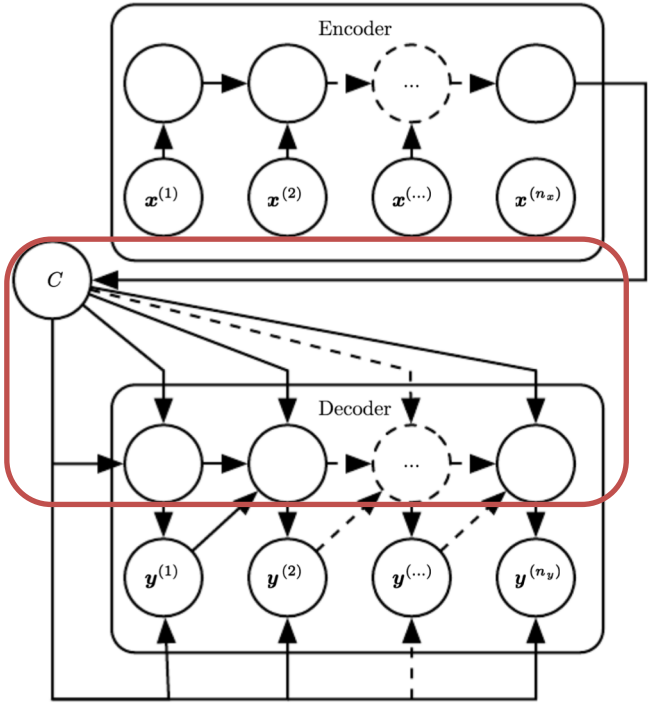
Generating Sequences with RNNs

- The encoder model is essentially a sequence classification RNN, but instead of outputting a prediction/probability, it outputs a vector, $\mathbf{C}$.
  - In the simplest case $\mathbf{C}$ is just the final hidden state of the encoder RNN.
- This vector $\mathbf{C}$ is often called the "context" vector because it encodes the context that is needed for the decoding/generation process.
- E.g., in machine translation the context vector encodes the "underlying meaning" of the input.



A closer look at the decoder

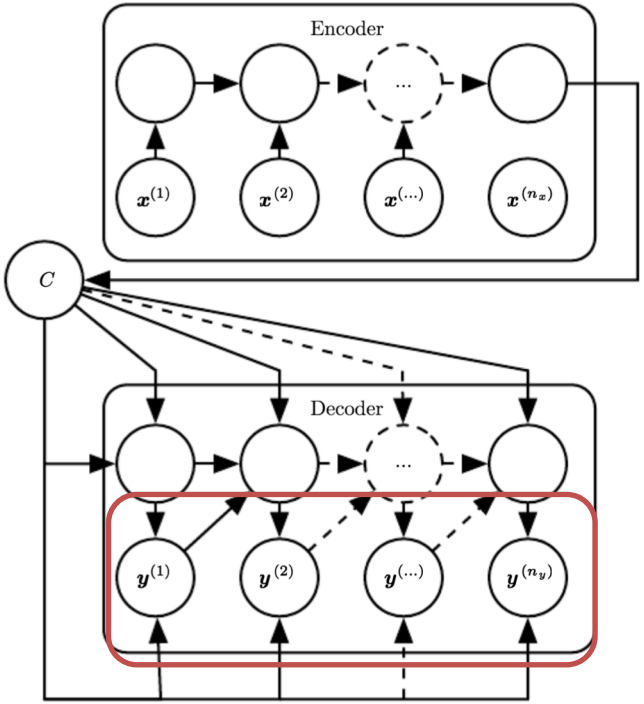
- The input to the decoder at every timestep includes the "context" vector (output by the encoder RNN).



A closer look at the decoder

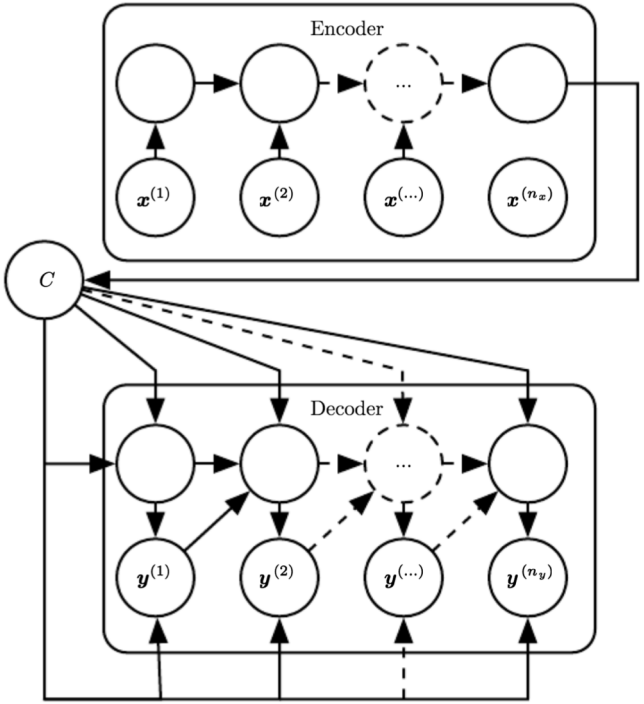
- The input to the decoder at **every** timestep includes the "context" vector (output by the encoder RNN).

- The output $\mathbf{y^{(t)}}$ at timestep $\mathbf{t}$ becomes an input at timestep $\mathbf{t+1}$.



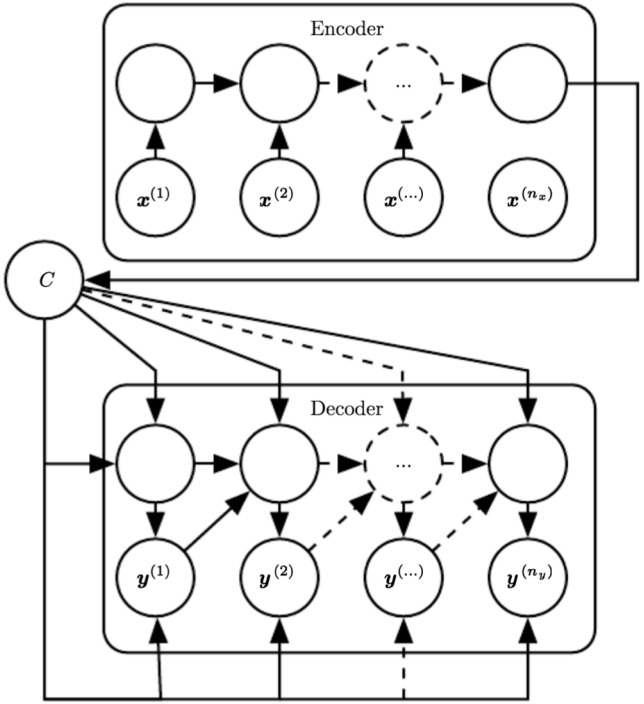
A closer look at the decoder

- The input to the decoder at **every** timestep includes the "context" vector (output by the encoder RNN).
- The output $\mathbf{y}^{(t)}$ at timestep $\mathbf{t}$ becomes an input at timestep $\mathbf{t+1}$.
- Usually the context vector and previous output are simply concatenated, but you could also add them, multiply them, etc.



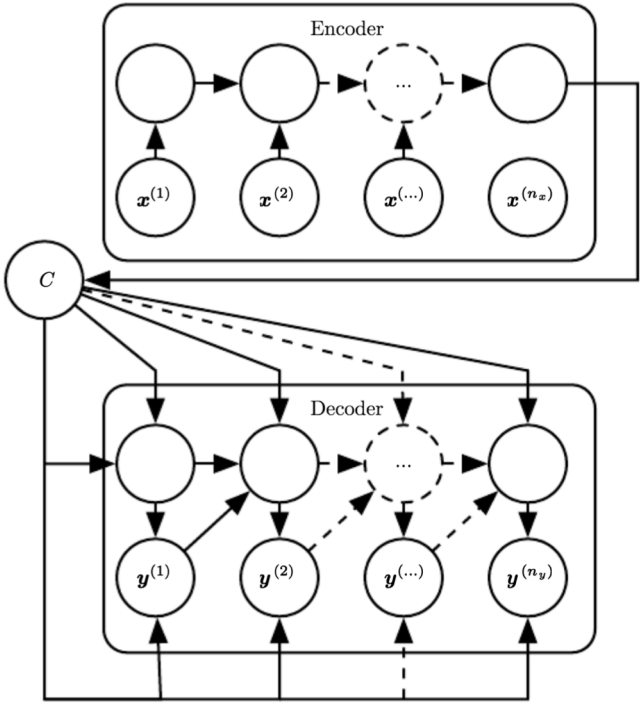
A closer look at the decoder

- The input to the decoder at **every** timestep includes the "context" vector (output by the encoder RNN).
- The output $\mathbf{y}^{(t)}$ at timestep $\mathbf{t}$ becomes an input at timestep $\mathbf{t+1}$.
- Q: How do you know when to stop generating?
- A: Add a special "end-of-sequence" symbol



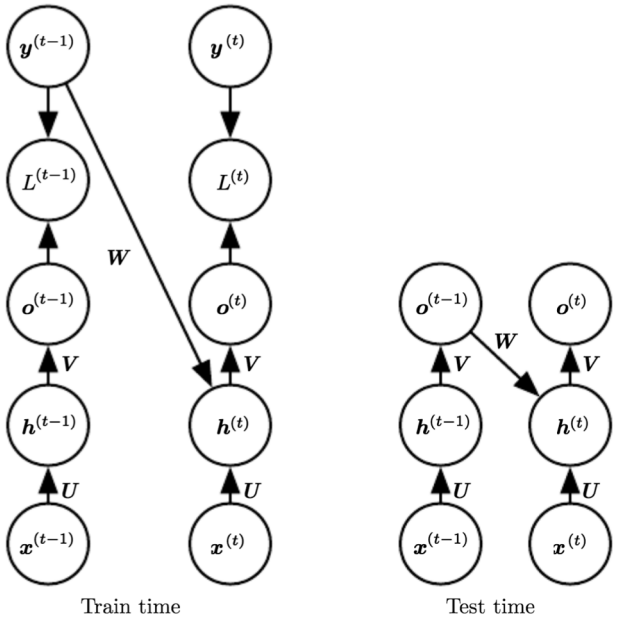
A closer look at the decoder

- The input to the decoder at **every** timestep includes the "context" vector (output by the encoder RNN).
- The output $\mathbf{y}^{(t)}$ at timestep $\mathbf{t}$ becomes an input at timestep $\mathbf{t+1}$.
- Q: How do you train this thing?
- A: Using standard backprop.
  - The loss is computed in the same way as sequence modeling.
  - But remember that the loss is backpropagated all the way into the encoder!

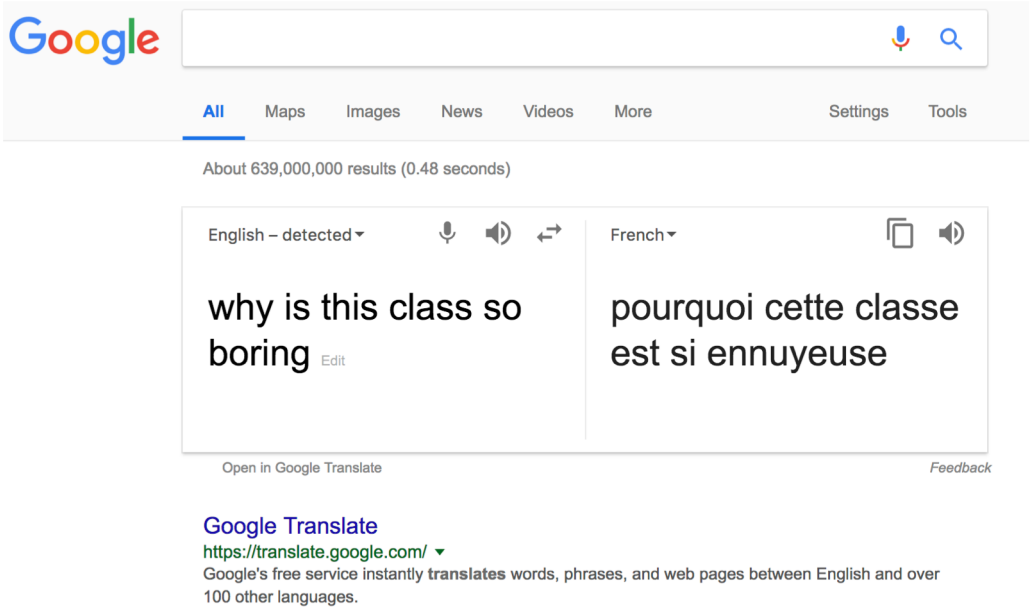


Teacher forcing

- Method of training RNNs if the model receives ground-truth output as input
- Train time: condition on previous ground-truth output
- Test time: don't have true labels, so condition on your own prediction



Application: machine translation

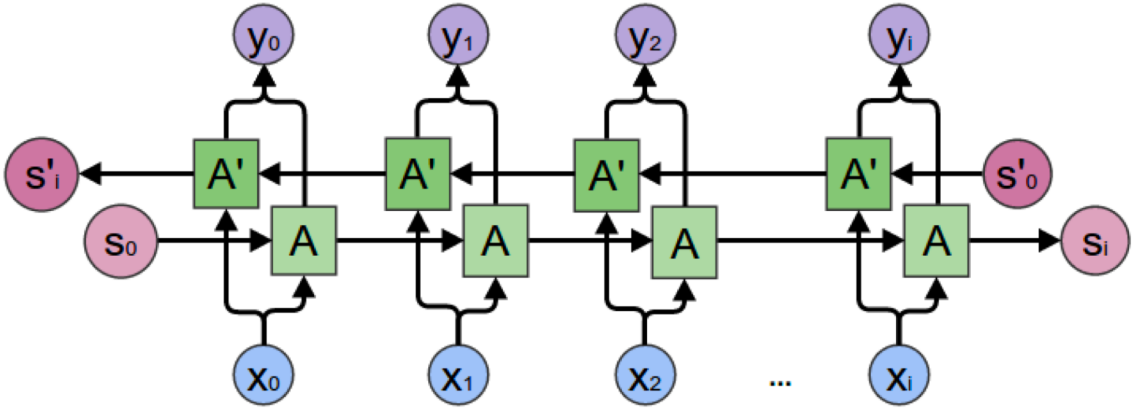


LSTMs: Increasing their complexity

- How can we make LSTMs (and RNNs in general) more expressive?
- Standard approaches:
  - Increase the dimension of the hidden/cell states
  - Increase the depth (i.e., number of layers)
  - Make the LSTMs/RNNs "bidirectional"
  - Add "attention" to encoder-decoder RNNs

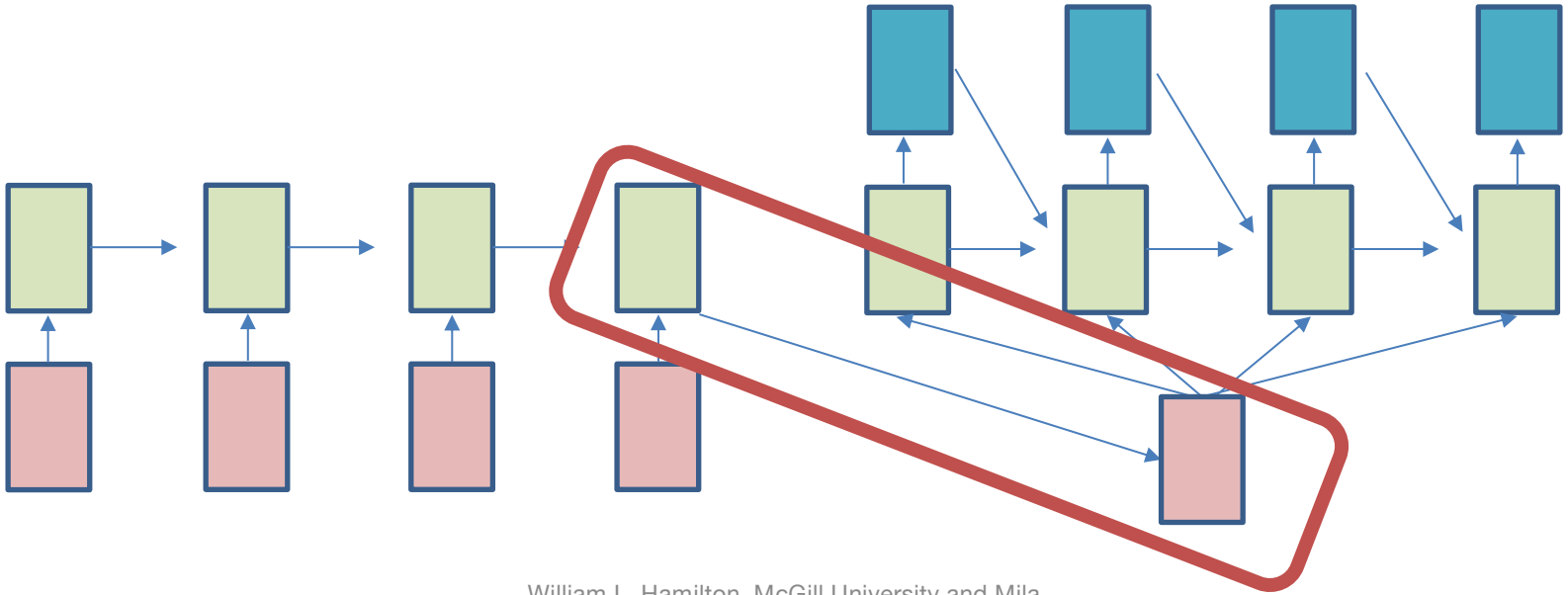
Bidirectional LSTMS

- Simple Idea: Why should we process from left-to-right?
- Bidirectional LSTMs simply encode a sequence by applying on LSTM from left-to-right and another from right-to-left



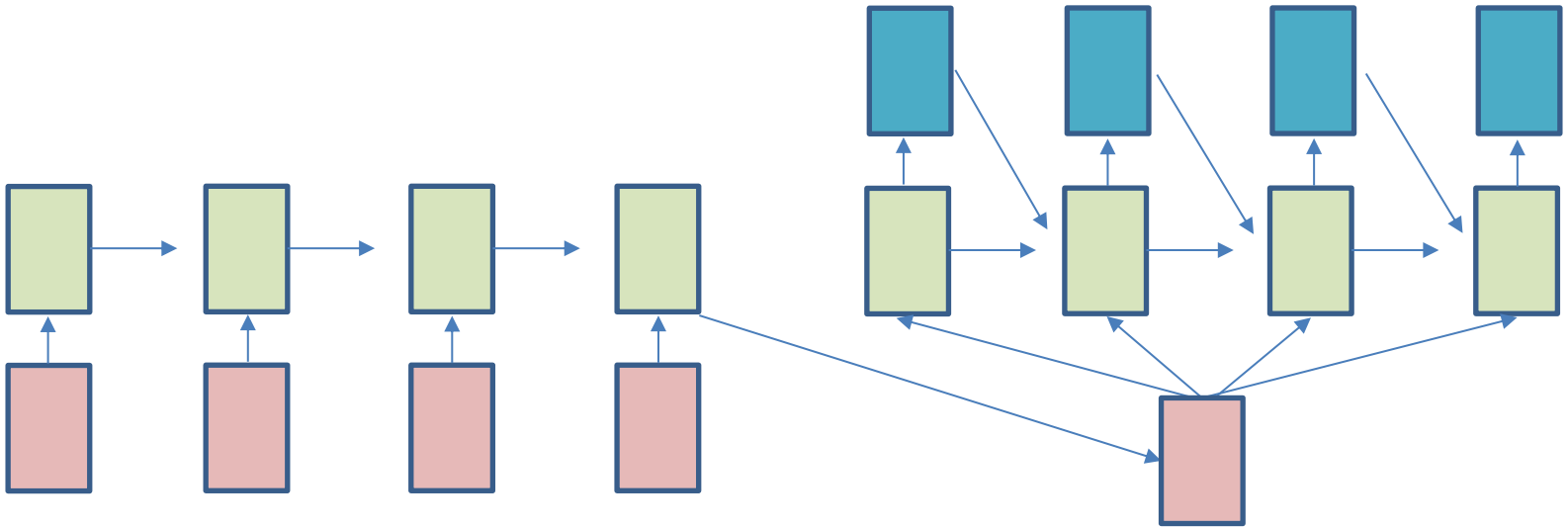
Adding attention

- In the standard encoder-decoder, the “context” input to the decoder only depends on the final output of the encoder RNN.



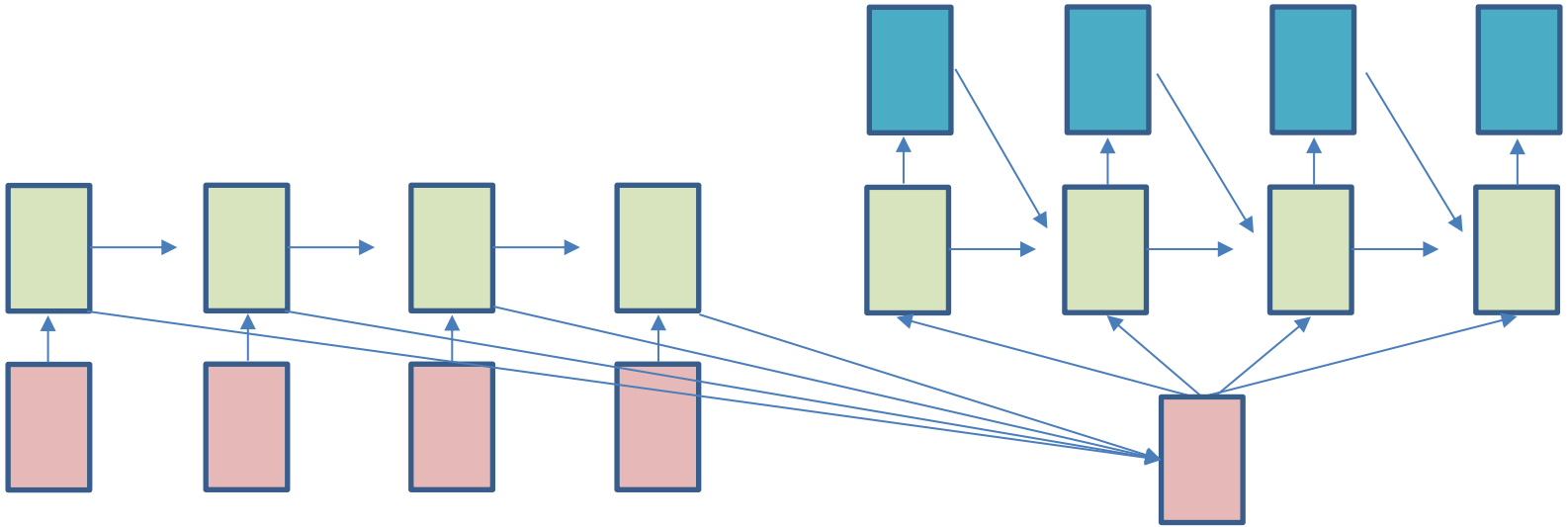
Adding attention

- But what if different parts of the input are useful for generating different parts of the output? (E.g., consider translating place or person names).



Adding attention

- **Idea 1**: Allow the context vector to depend on all the encoder hidden states!
- **Idea 2**: Compute a vector of weights at each time-step of the decoder to determine the contribution of each of the encoder hidden states.



Adding attention

- **Idea 1**: Allow the context vector to depend on all the encoder hidden states!
- **Idea 2**: Compute a vector of weights at each time-step of the decoder to determine the contribution of each of the encoder hidden states.
- The context vector at step s in the decoder is a weighted some of the encoder's hidden states:

$$\mathbf{c}_s = \sum_{t=1}^{T} \alpha_{t,s} \mathbf{h}_t^{\text{ENC}}$$

- The "attention weights" are (usually) computed using a softmax and a "scoring function" that depends on the encoder hidden states and (previous) decoder hidden state:

$$\alpha_{s,t} = \frac{e^{\text{score}(\mathbf{h}_t^{\text{ENC}}, \mathbf{h}_{s-1}^{\text{DEC}})}}{\sum_{j=1}^{T} e^{\text{score}(\mathbf{h}_j^{\text{ENC}}, \mathbf{h}_{s-1}^{\text{DEC}})}}$$

Adding attention

- Some example scoring functions (from Luong et al, 2015):
  - Dot product:

$$\text{score}(\mathbf{h}^{(\text{ENC})}, \mathbf{h}^{(\text{DEC})}) = (\mathbf{h}^{(\text{ENC})})^\top \mathbf{h}^{(\text{DEC})}$$

  - "Bilinear":

$$\text{score}(\mathbf{h}^{(\text{ENC})}, \mathbf{h}^{(\text{DEC})}) = (\mathbf{h}^{(\text{ENC})})^\top \mathbf{W}\mathbf{h}^{(\text{DEC})}$$

  - "Concat":

$$\text{score}(\mathbf{h}^{(\text{ENC})}, \mathbf{h}^{(\text{DEC})}) = \mathbf{v}^\top \tanh(\mathbf{W}[\mathbf{h}^{(\text{ENC})}), \mathbf{h}^{(\text{DEC})}])$$

- There are many, many variations of attention! The architectures I discussed here are just a few classic examples.

Adding attention

- Adding attention often improves empirical results.
- Attention also provides a form of "interpretability".
- E.g., we can visualize which parts of the input the decoder is attending to during generation.

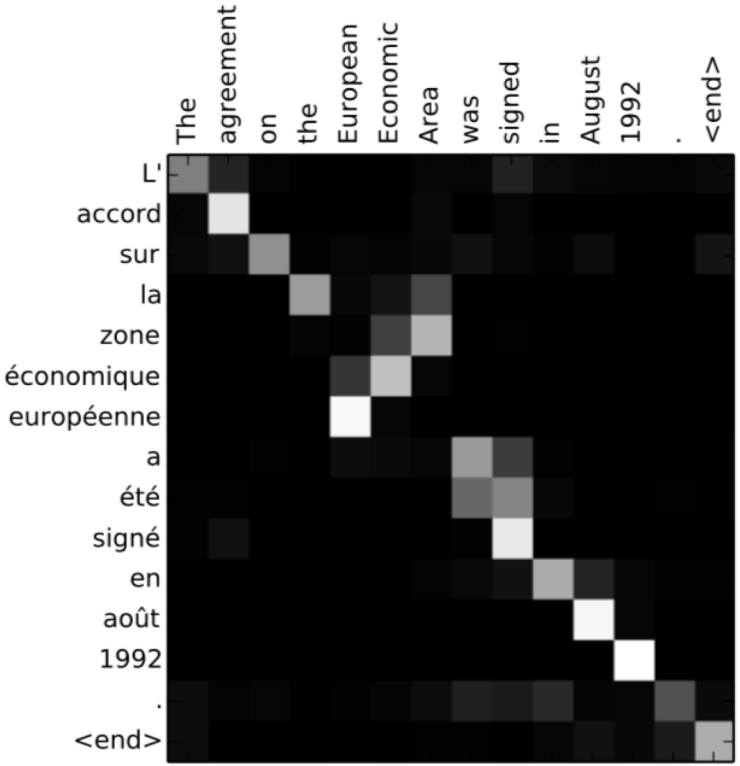


Image captioning

- Can use principle of ‘attention’ over images instead of text
- Combine with CNN to extract features

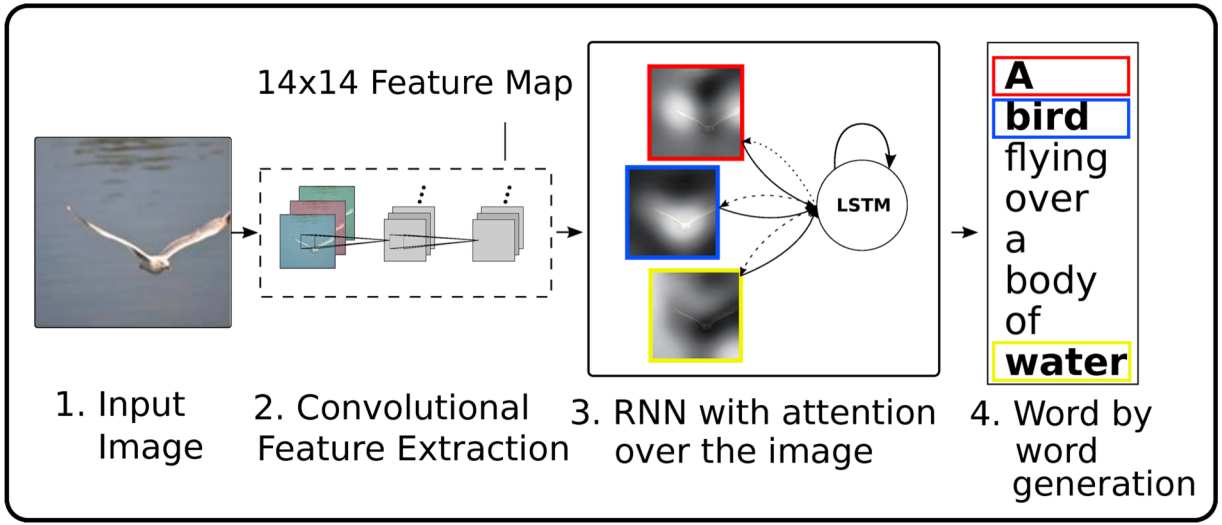


Image captioning

A woman is throwing a **frisbee** in a park.

A **dog** is standing on a hardwood floor.

A **stop** sign is on a road with a mountain in the background

A little **girl** sitting on a bed with a teddy bear.

A group of **people** sitting on a boat in the water.

A giraffe standing in a forest with **trees** in the background.

Image captioning

A large white bird standing in a forest.

A woman holding a clock in her hand.

A man wearing a hat and a hat on a skateboard.

A person is standing on a beach with a surfboard.

A woman is sitting at a table with a large pizza.

A man is talking on his cell phone while another man watches.

Xu et al., (2015)

What you should know

- The problem of vanishing/exploding gradients.
- The LSTM architecture.
- The encoder-decoder architecture and how it's trained (e.g., teacher forcing).
- The concept of attention in encoder-decoder models.