Introductory Overview Lecture

The Deep Learning Revolution

Christopher Manning & Russ Salakhutdinov

Stanford University ❀ Carnegie Mellon University

JSM, 2018-07-29

Plan for Part 4

Big ideas for bigger, structured neural models

a. Convolutional models
b. Recurrent models
c. Gated and residual connections
d. Attention
e. Final thoughts

4a. Vision: Convolutional models

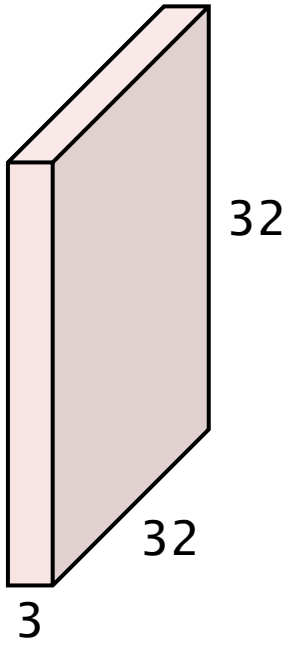
- For computer vision, a key property that we usually wish to capture is translation invariance
- We would like to have visual “feature detectors” that find something in an image regardless of precisely where it is located
- We do this with a convolutional layer

[Slides from Karpathy/Johnson/Yeung/Fei-Fei, cs231n.stanford.edu]

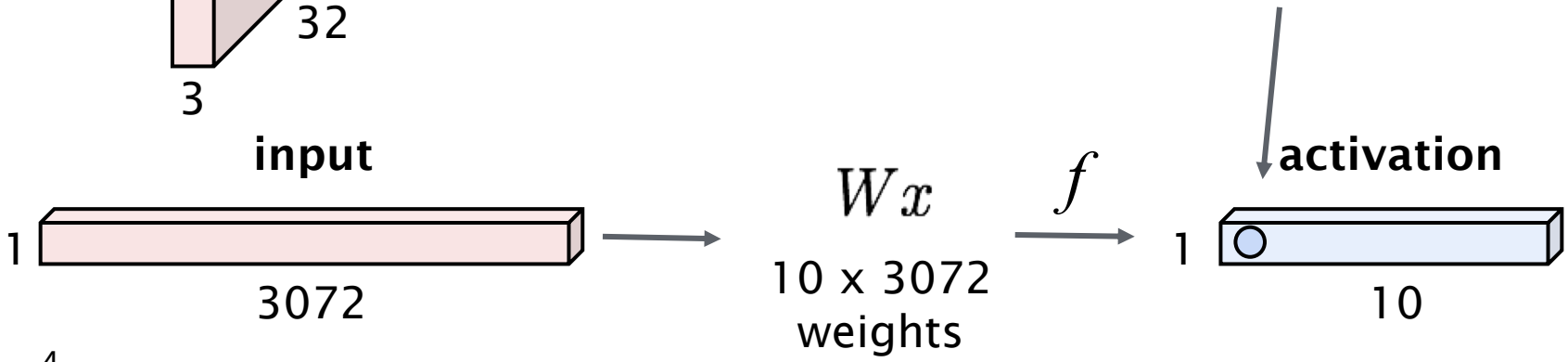
Motivating a Convolution Layer

32x32x3 image → 10 classes

Could stretch to 3072 x 1 and use an FC layer



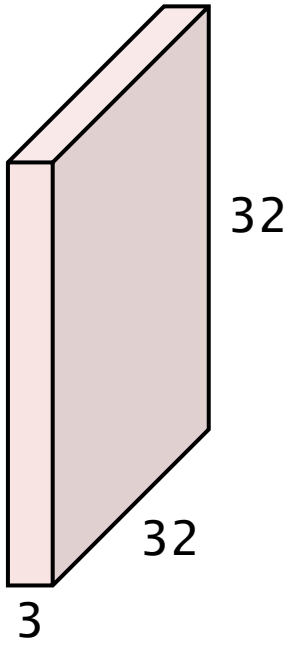
A number
A probability of the object being a certain class



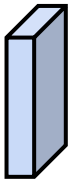
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image



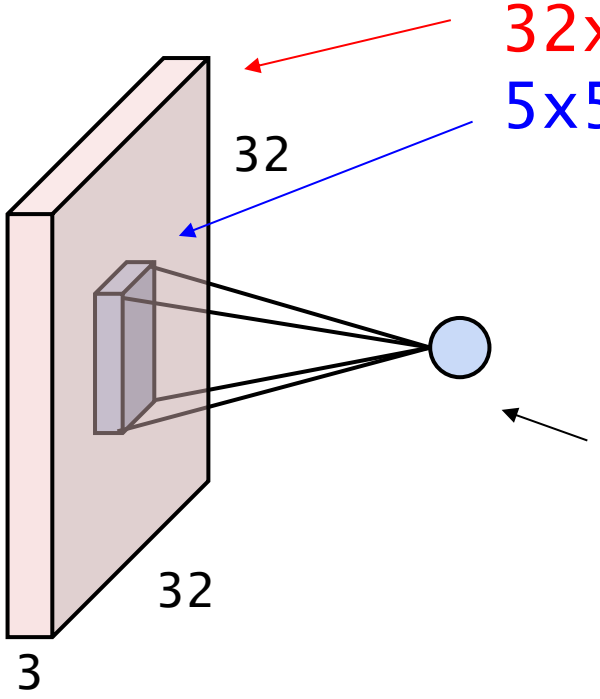
5x5x3 filter



Convolve a filter with the image i.e. “slide over the image spatially, computing dot products”

[Slides from Karpathy/Johnson/Yeung/Fei-Fei, cs231n.stanford.edu

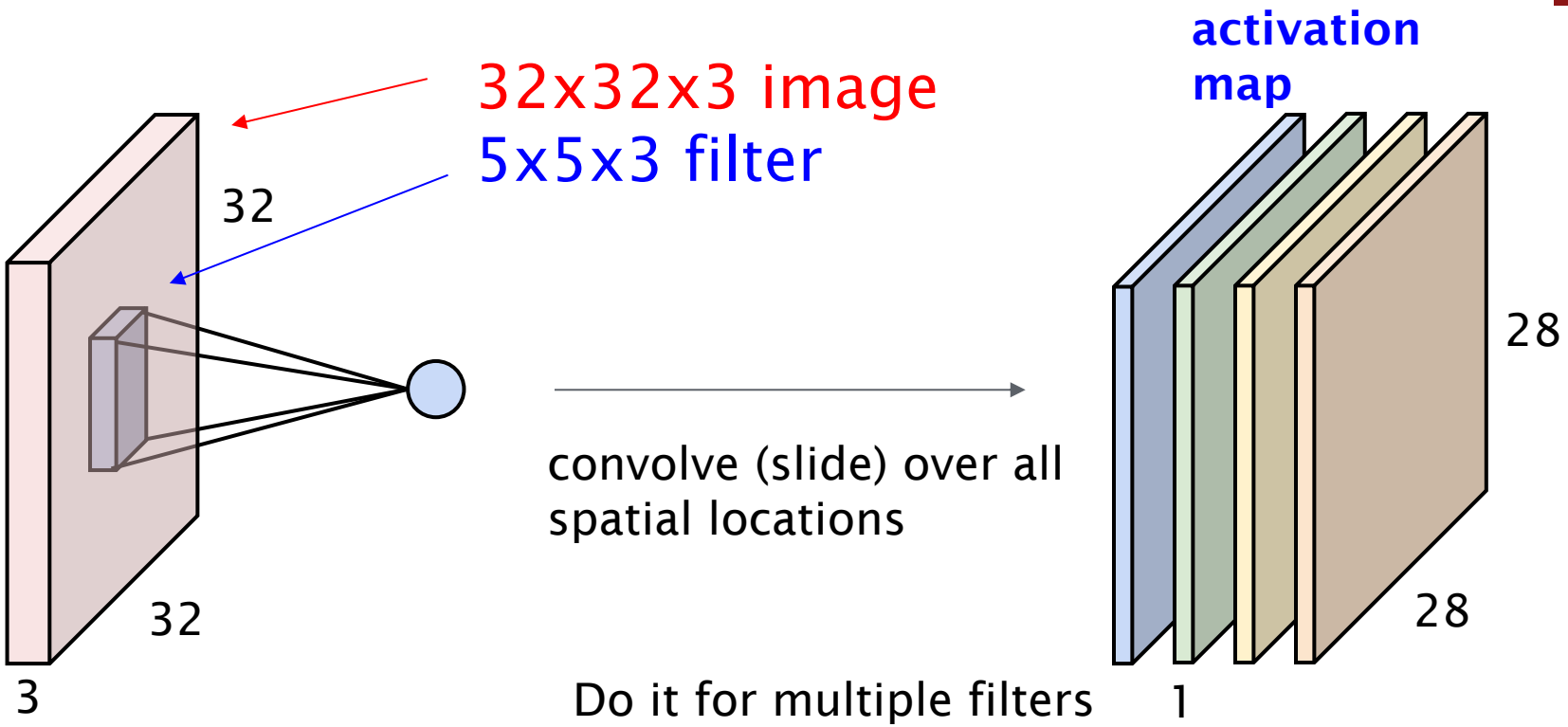
Convolution Layer



1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

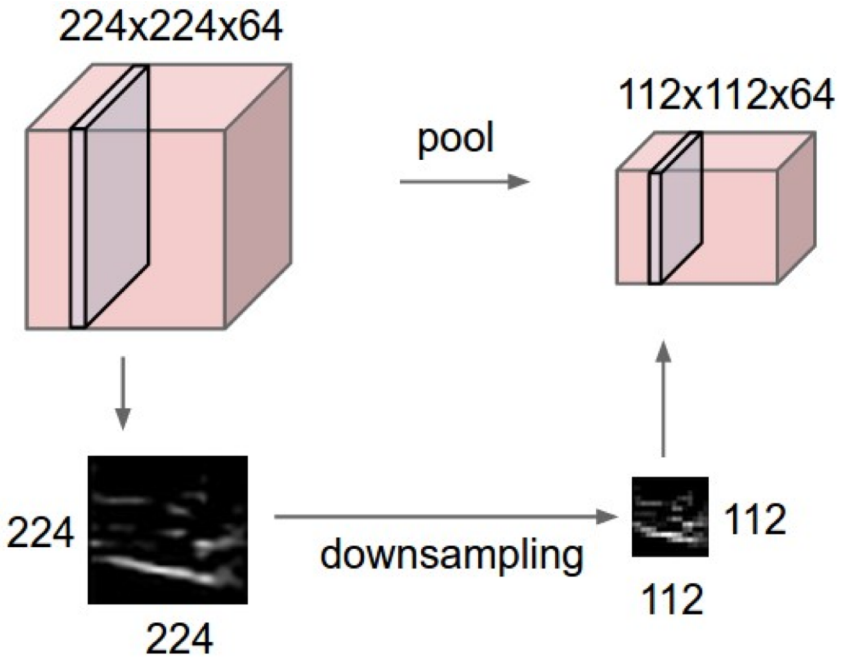
$$w^T x + b$$

Convolution Layer

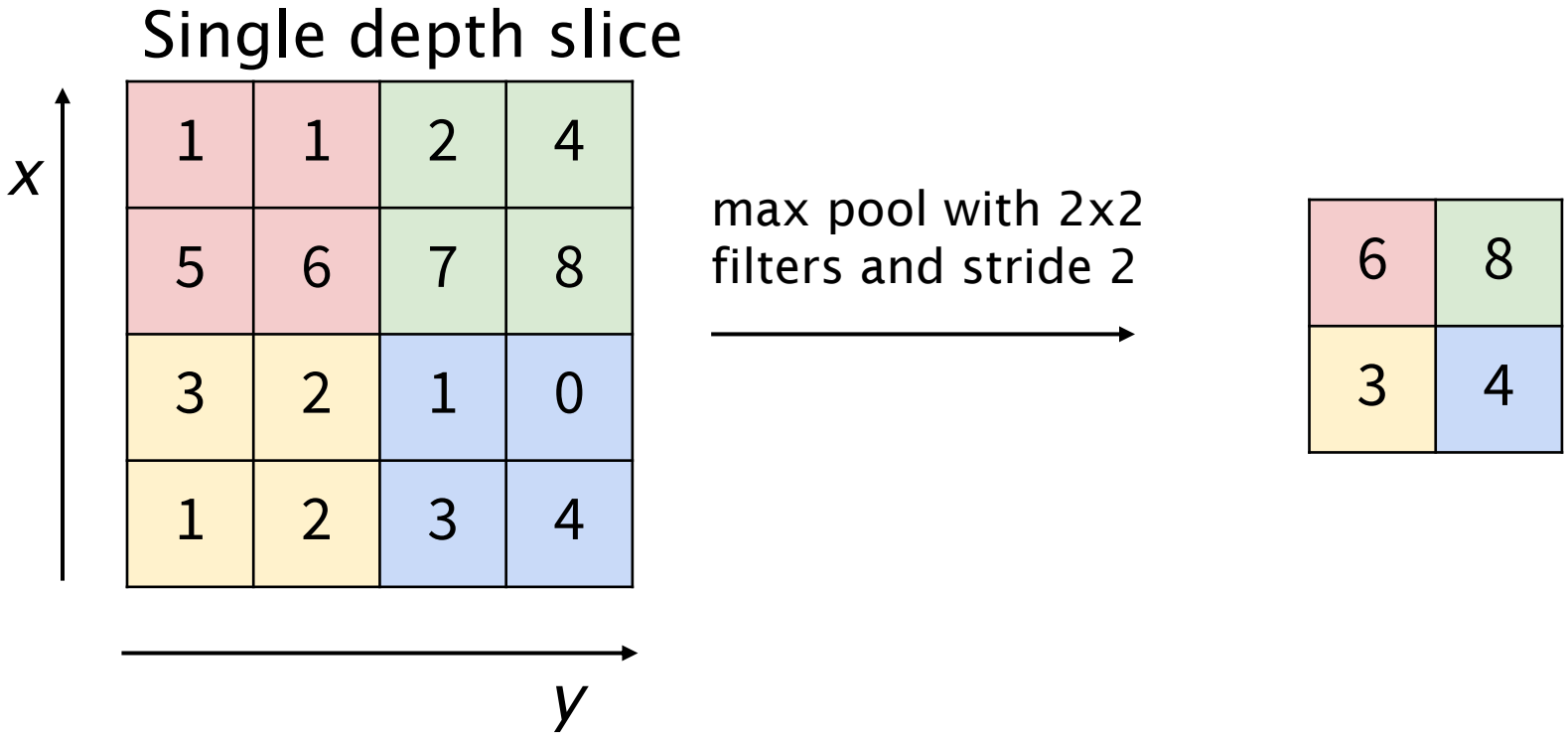


Pooling layer

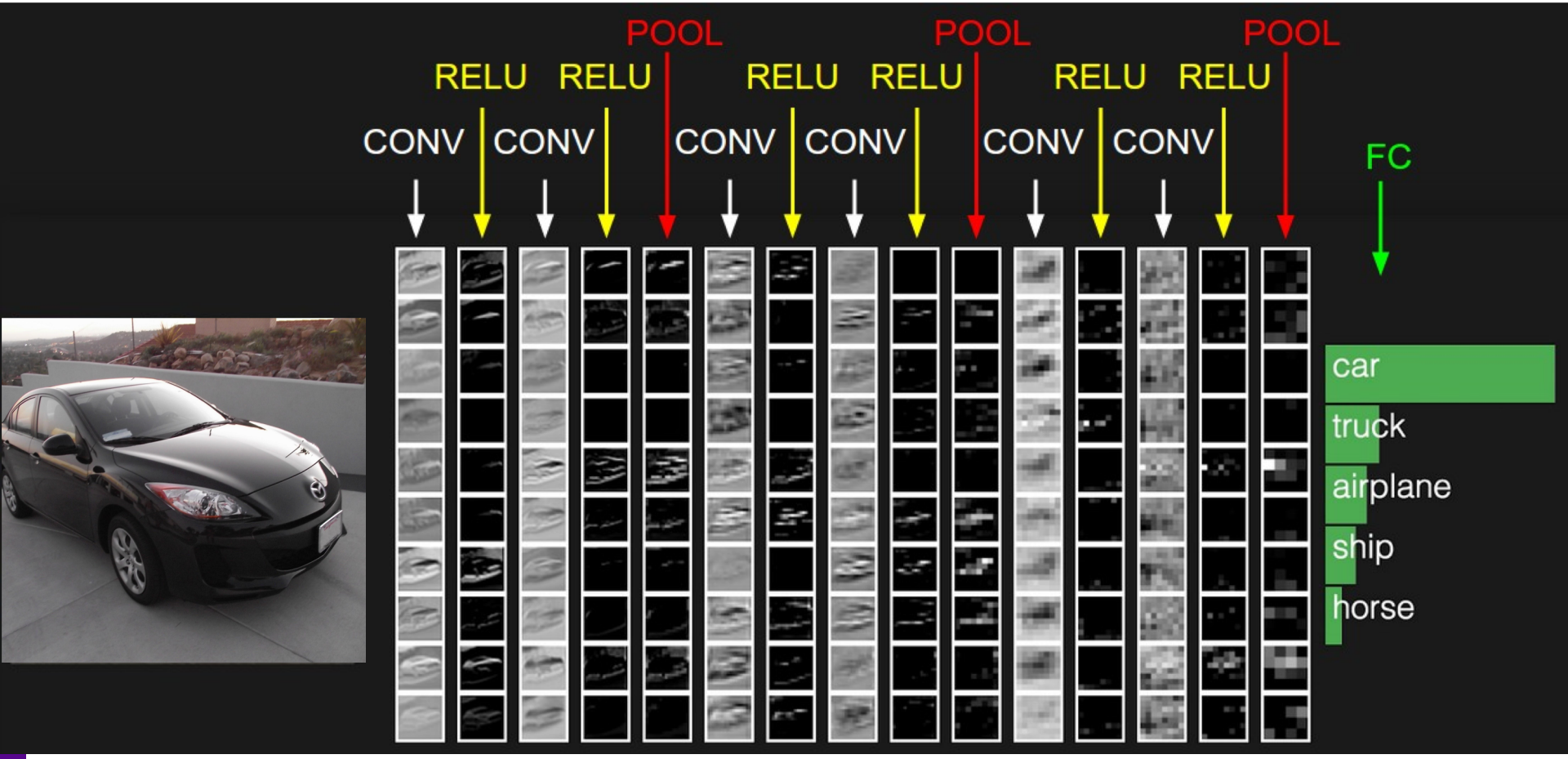
- Allows convolutional features with a broader view without a lot of parameters
- Can make representations more manageable



Pooling layer

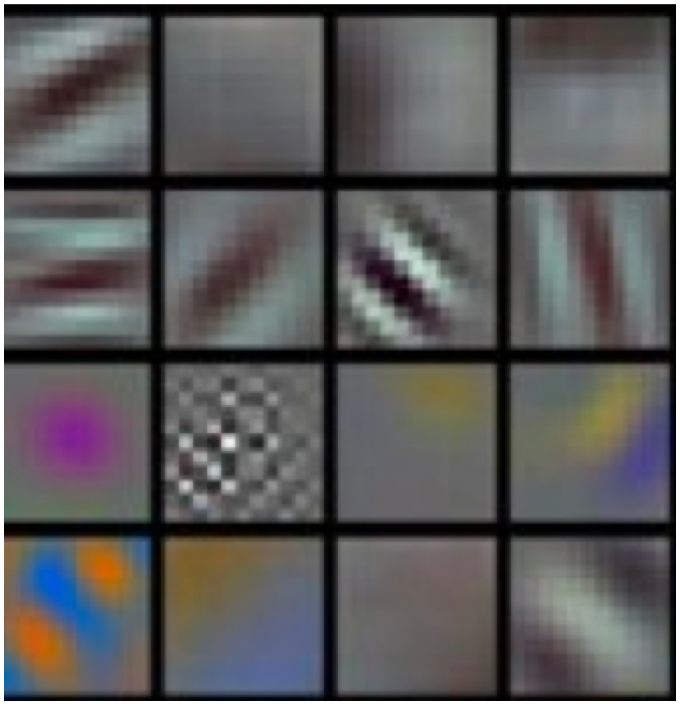


A (Very Simple) Modern ConvNet

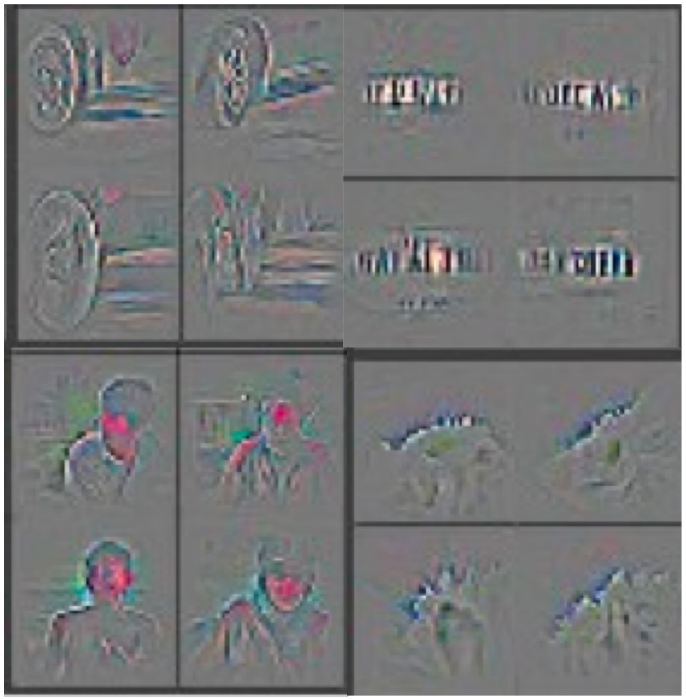


ConvNet Representation Learning

- At the beginning of the ConvNet, features are edge and color/texture detectors



- High in the ConvNet, features detect parts and object types

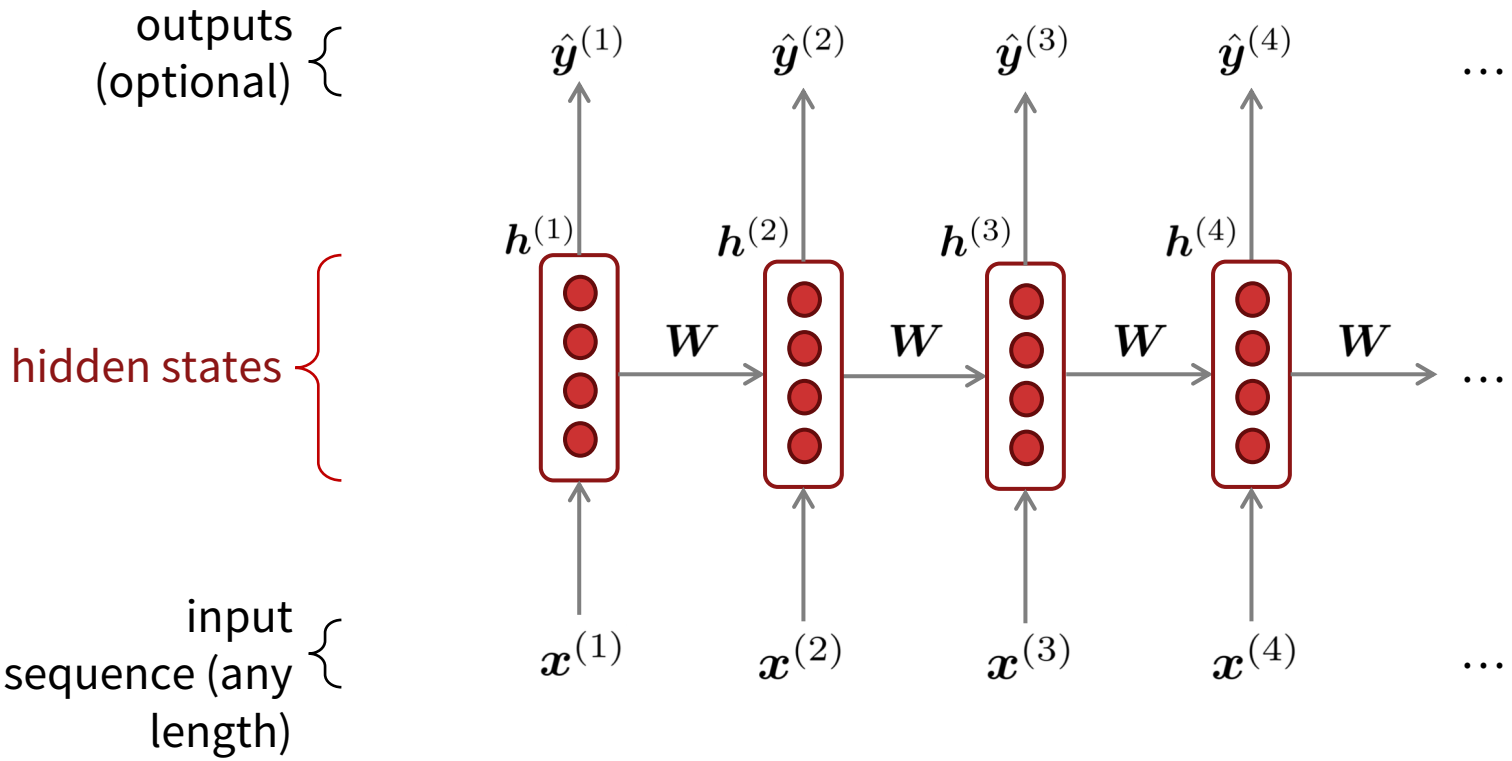


4b. Language: Recurrent models

- Until now, we've dealt with classifying/generating fixed-size objects.
  - We just resized images to our procrustean bed!
- How can we deal with variable-size inputs, such as the word sequences in human language text or bioinformatic gene sequences?
- We do this with a recurrent layer

Recurrent Neural Networks (RNN)

Core idea: Apply the same weights W *repeatedly*



Read, update, predict

Example: a Recurrent "Language Model" that generates sentences

Transition Function $h_t = f(h_{t-1}, x_t)$

Inputs

i. Current word $x_t \in \{1, 2, \ldots, |V|\}$

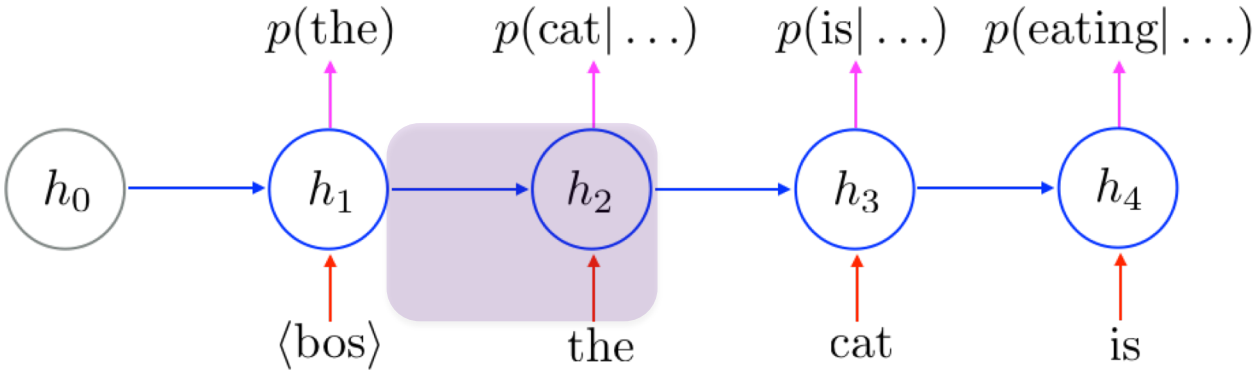
ii. Previous state $h_{t-1} \in \mathbb{R}^d$

Parameters

i. Input weight matrix $W \in \mathbb{R}^{|V| \times d}$

ii. Transition weight matrix $U \in \mathbb{R}^{d \times d}$

iii. Bias vector $b \in \mathbb{R}^d$



Building a Recurrent Language Model

Transition Function $h_t = f(h_{t-1}, x_t)$

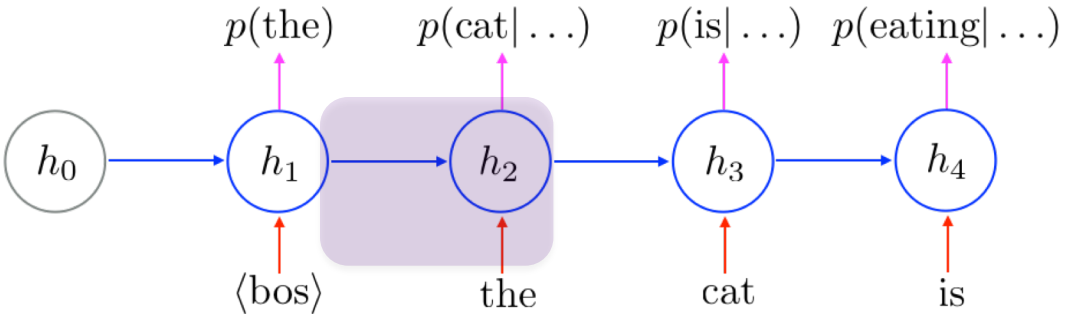
Naïve Transition Function

$$f(h_{t-1}, x_t) = \tanh(W\,[x_t] + U h_{t-1} + b)$$

Element-wise nonlinear transformation

Trainable word vector

Linear transformation of previous state



Building a Recurrent Language Model

Prediction Function $p(x_{t+1} = w | x_{\leq t}) = g_w(h_t)$

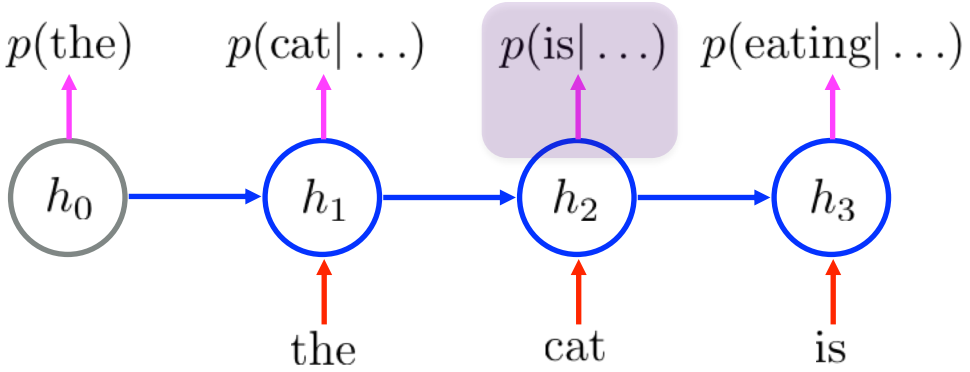
Inputs

i. Current state $h_t \in \mathbb{R}^d$

Parameters

i. Softmax matrix $R \in \mathbb{R}^{|V| \times d}$

ii. Bias vector $c \in \mathbb{R}^{|V|}$



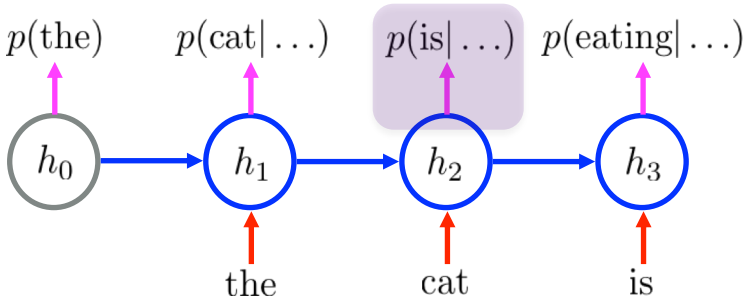
Building a Recurrent Language Model

Prediction Function $p(x_{t+1} = w | x_{\leq t}) = g_w(h_t)$

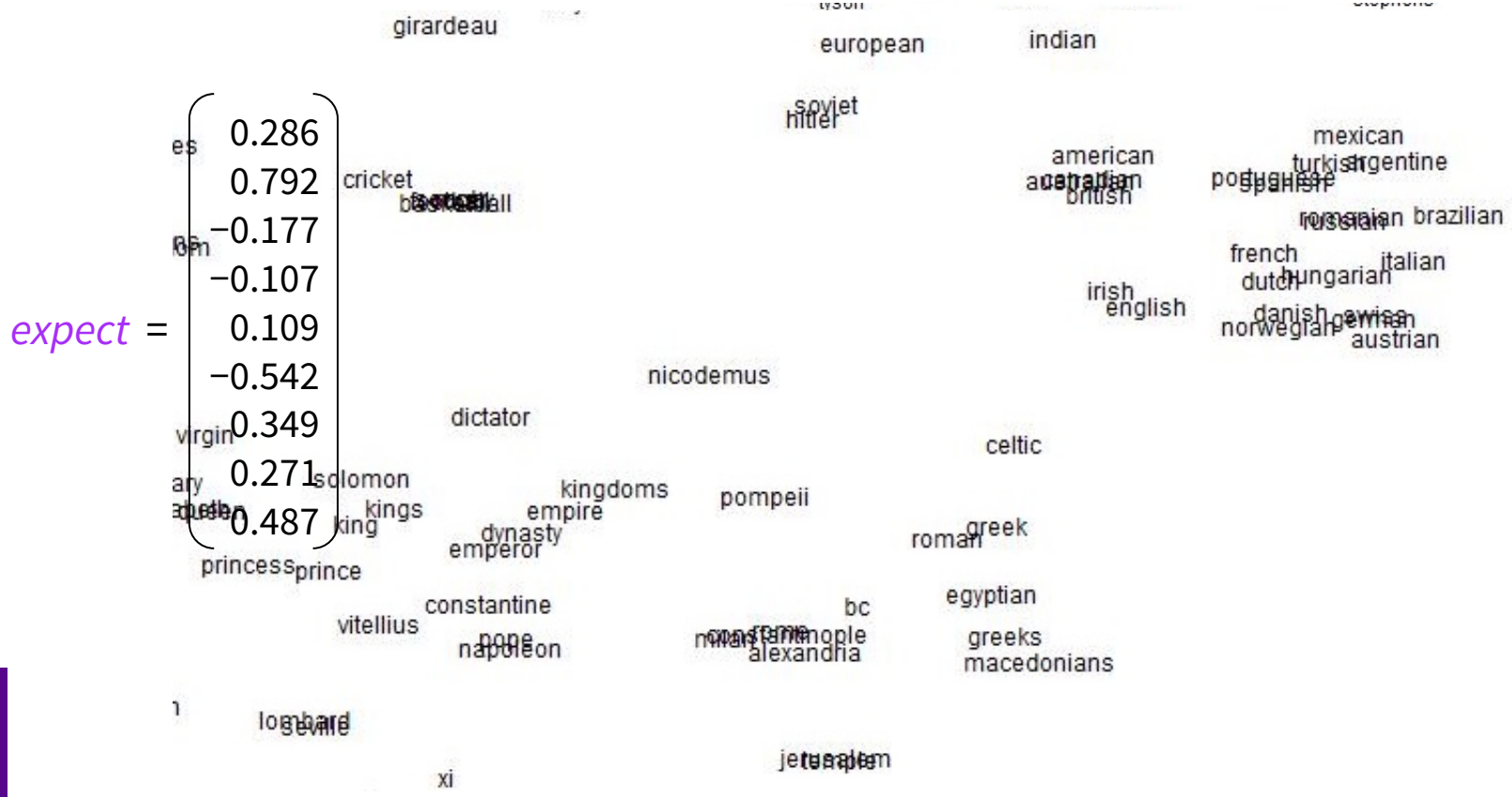
$$p(x_{t+1} = w | x_{\leq t}) = g_w(h_t) = \frac{\exp(R[w]^\top h_t + c_w)}{\sum_{i=1}^{|V|} \exp(R[i]^\top h_t + c_i)}$$

Normalize

This gives a probability distribution over next words. To generate text we take max. prob. word (or sample a word), and then use what we generated as the input at the next time step



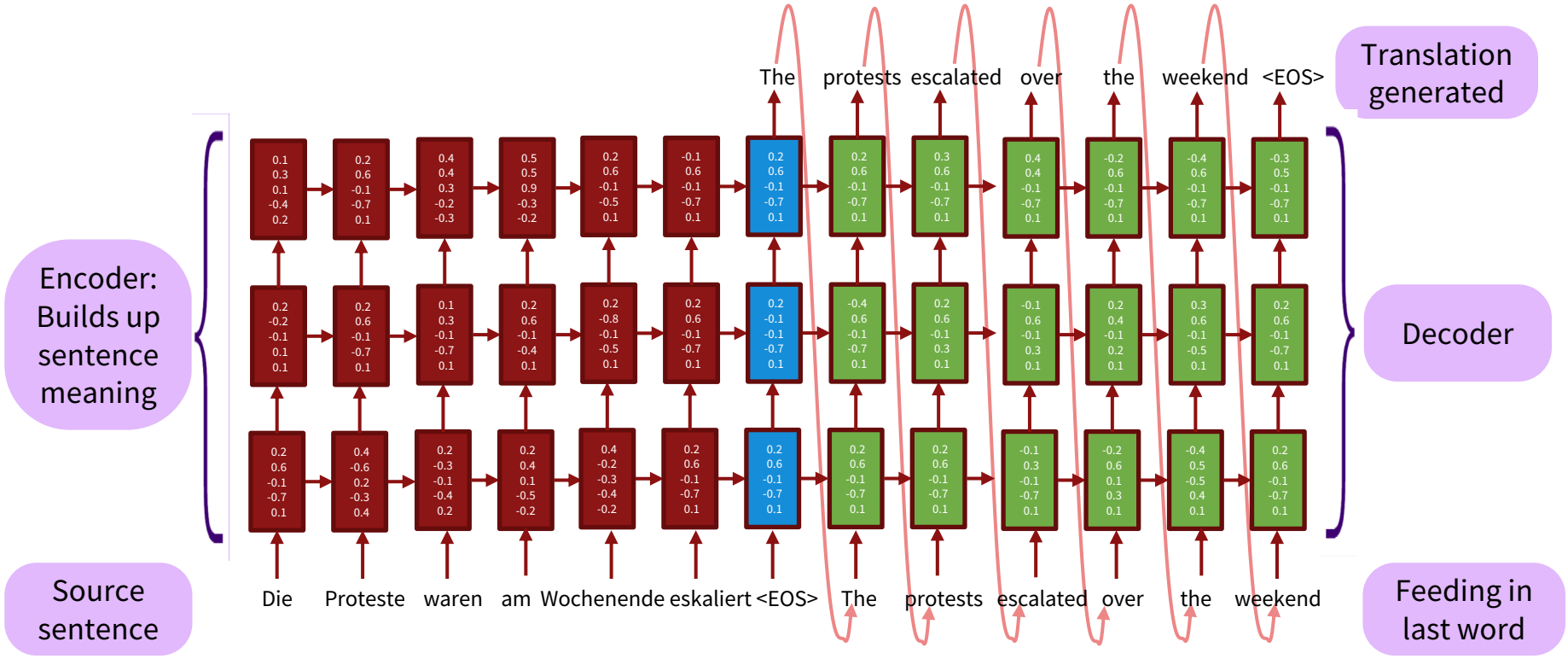
Learned word representations show words' semantic similarity



$$expect = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \\ 0.487 \end{pmatrix}$$

Neural Machine Translation

Source sentence is mapped to vector, then output sentence generated [Sutskever et al. 2014, Bahdanau et al. 2014, Luong and Manning 2016]



2017+: Almost every company is now using Neural MT in production, at least for many language pairs

Google
Microsoft
facebook

4c. Gated Units

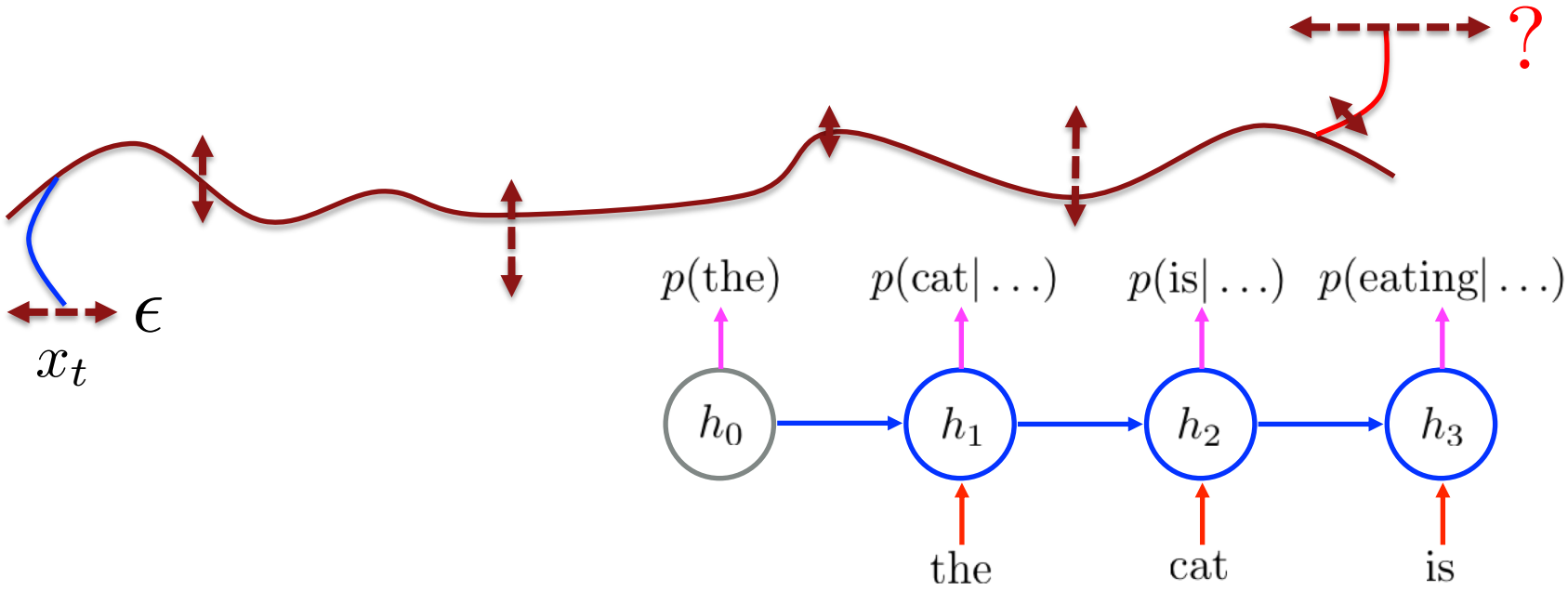
Fixing Backpropagation through Time

Intuitively, what happens with RNNs?

1. Measure the influence of the past on the future

$$\frac{\partial \log p(x_{t+n}|x_{<t+n})}{\partial h_t} = \frac{\partial \log p(x_{t+n}|x_{<t+n})}{\partial g} \frac{\partial g}{\partial h_{t+n}} \frac{\partial h_{t+n}}{\partial h_{t+n-1}} \cdots \frac{\partial h_{t+1}}{\partial h_t}$$

2. How does the perturbation at t affect $p(x_{t+n}|x_{<t+n})$?



Backpropagation through Time

Problem: Often get a vanishing gradient which is super-problematic

- With the naïve transition function

$$f(h_{t-1}, x_t) = \tanh(W\,[x_t] + U h_{t-1} + b)$$

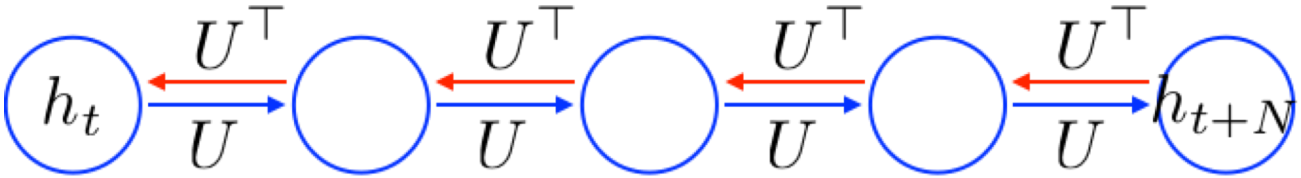
- The temporal derivative is $\dfrac{\partial h_{t+1}}{\partial h_t} = U^\top \dfrac{\partial \tanh(a)}{\partial a}$
- When we only observe

$$\left\| \frac{\partial h_{t+N}}{\partial h_t} \right\| = \left\| \prod_{n=1}^{N} U^\top \operatorname{diag}\left(\frac{\partial \tanh(a_{t+n})}{\partial a_{t+n}} \right) \right\| \to 0$$

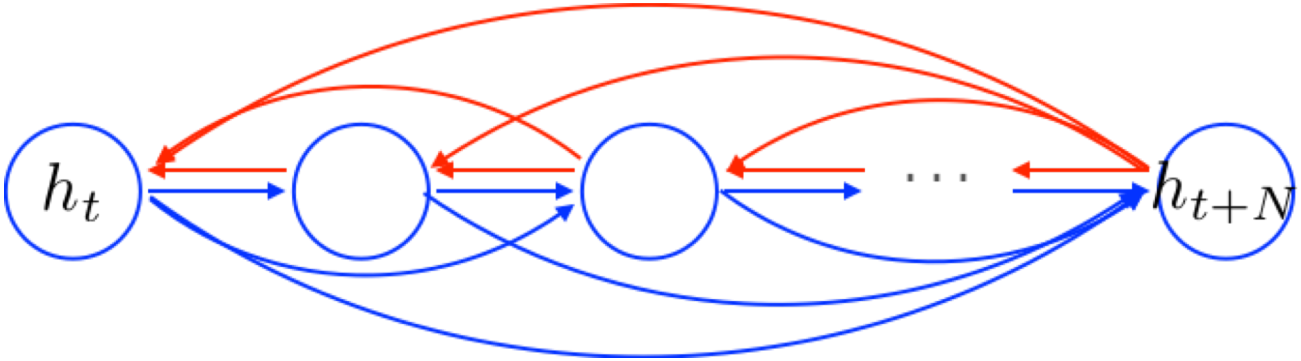
- We cannot tell whether there is
  1. No dependency between *t* and *t+N* in data, or
  2. Small weights (leading eigenvalue < 1) → gradient vanishes

Gated Recurrent Unit

- With a simple recurrent model, the error must backpropagate through all the intermediate nodes:

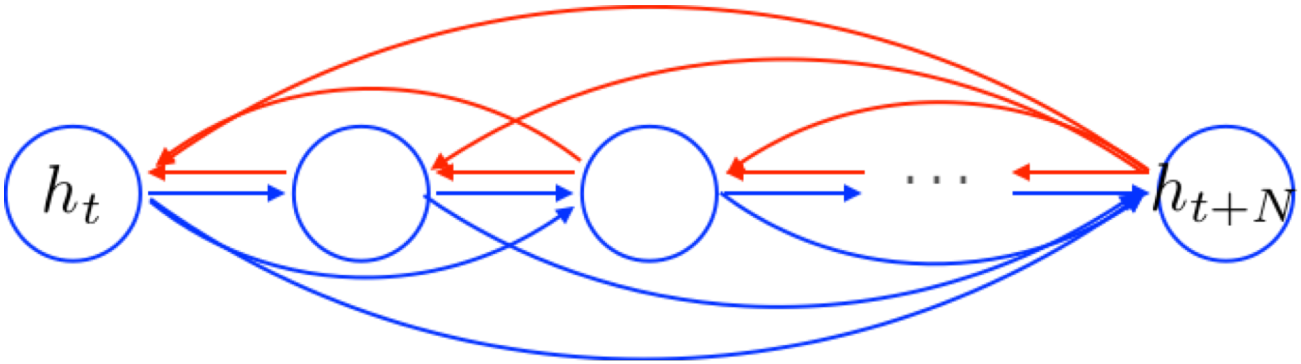


- Perhaps we can create shortcut connections



Gated Recurrent Unit

- Perhaps we can create *adaptive* shortcut connections



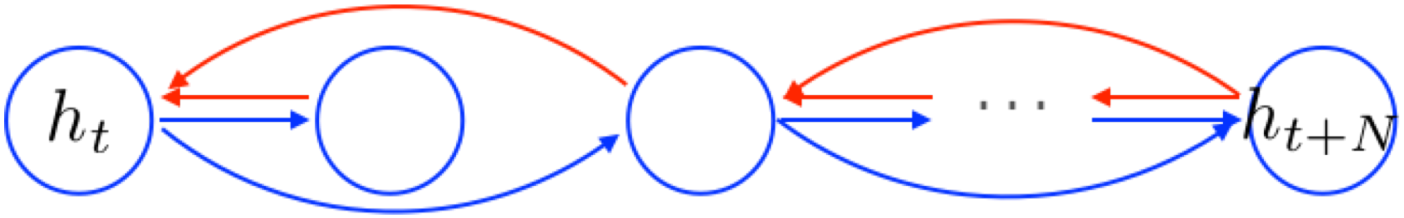
- Candidate Update $\tilde{h}_t = \tanh(W[x_t] + Uh_{t-1} + b)$
- Update gate $u_t = \sigma(W_u[x_t] + U_u h_{t-1} + b_u)$

$$f(h_{t-1}, x_t) = u_t \odot \tilde{h}_t + (1 - u_t) \odot h_{t-1}$$

$\odot$: element-wise multiplication

Gated Recurrent Unit

- Let the net prune unnecessary connections *adaptively*

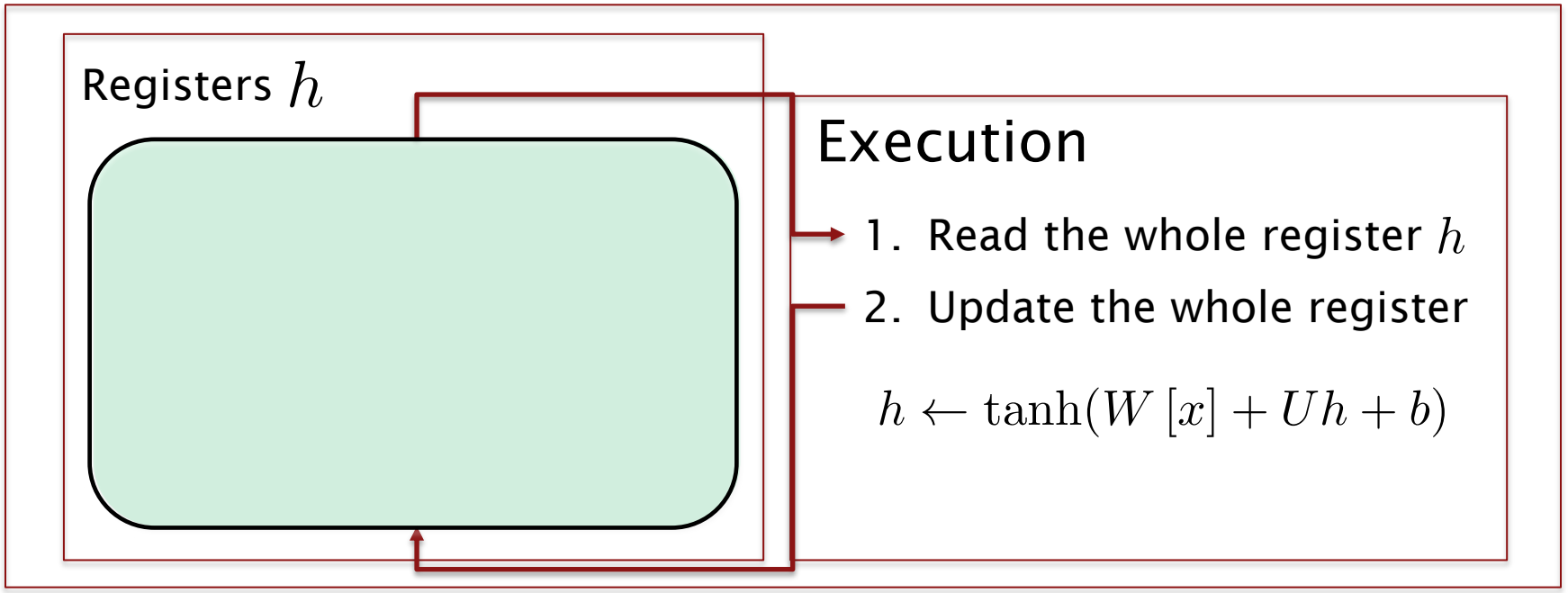


- Candidate Update $\tilde{h}_t = \tanh(W\,[x_t] + U(r_t \odot h_{t-1}) + b)$
- Reset gate $r_t = \sigma(W_r\,[x_t] + U_r h_{t-1} + b_r)$
- Update gate $u_t = \sigma(W_u\,[x_t] + U_u h_{t-1} + b_u)$

$$f(h_{t-1}, x_t) = u_t \odot \tilde{h}_t + (1 - u_t) \odot h_{t-1}$$

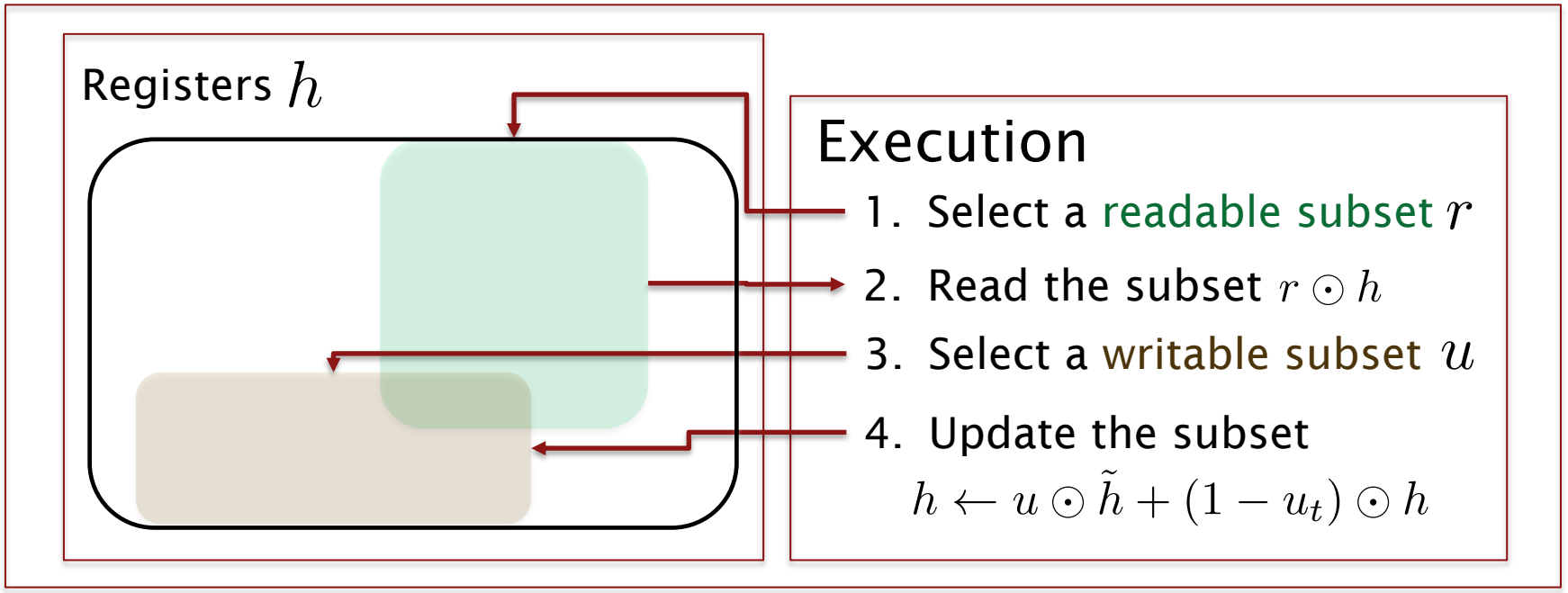
Gated Recurrent Unit

tanh-RNN ….



Gated Recurrent Unit

GRU …



Gated recurrent units are much more realistic!

Gated Recurrent Units

Two most widely used gated recurrent units

Gated Recurrent Unit

[Cho et al., EMNLP2014; Chung, Gulcehre, Cho, Bengio, DLUFL2014]

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

$$\tilde{h}_t = \tanh(W\,[x_t] + U(r_t \odot h_{t-1}) + b)$$

$$u_t = \sigma(W_u\,[x_t] + U_u h_{t-1} + b_u)$$

$$r_t = \sigma(W_r\,[x_t] + U_r h_{t-1} + b_r)$$

Long Short-Term Memory

[Hochreiter & Schmidhuber, NC1999; Gers, Thesis2001]

$$h_t = o_t \odot \tanh(c_t)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

$$\tilde{c}_t = \tanh(W_c\,[x_t] + U_c h_{t-1} + b_c)$$

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

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

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

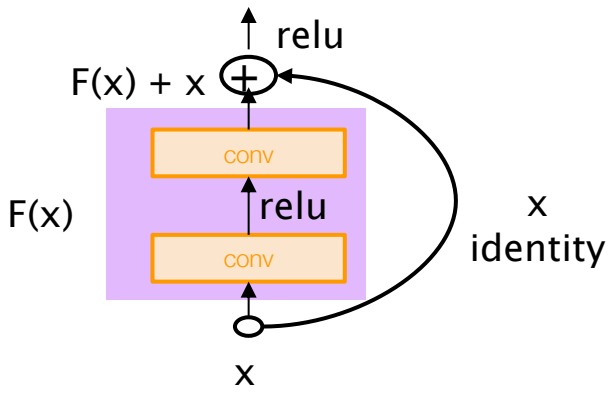
In many ways similar, but the LSTM is actually more powerful: Its cell can count

Gated units

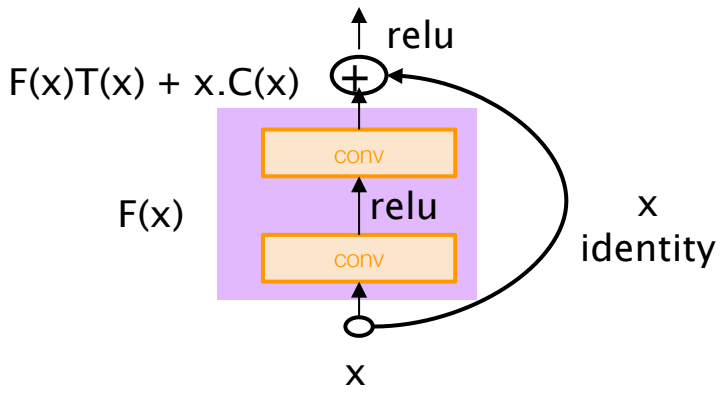
Gating is a general idea, which is now used in a whole bunch of places

You can also gate vertically

- Indeed the key idea – summing candidate update with shortcut connection – is needed for very deep networks to work



Residual block

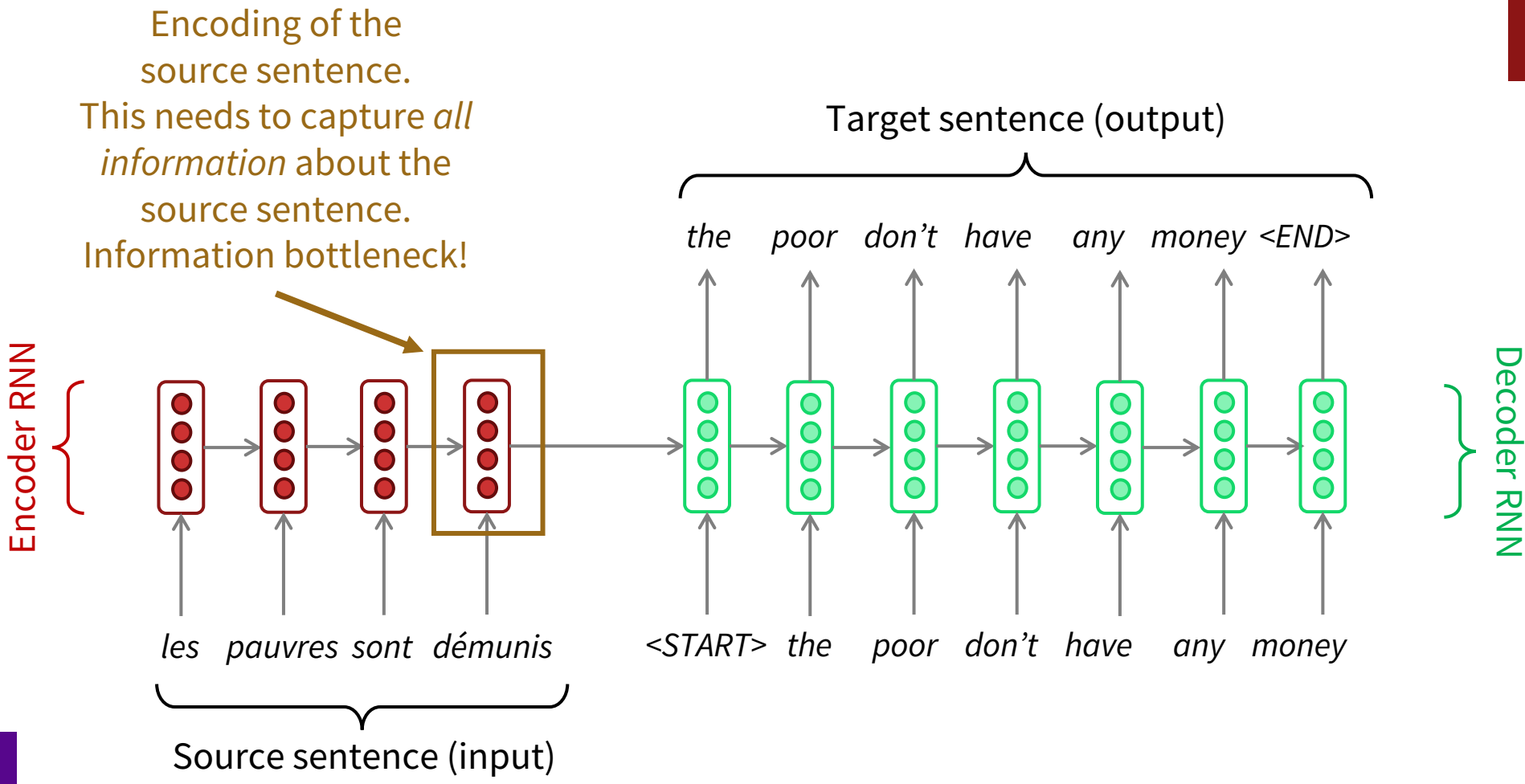


Highway block

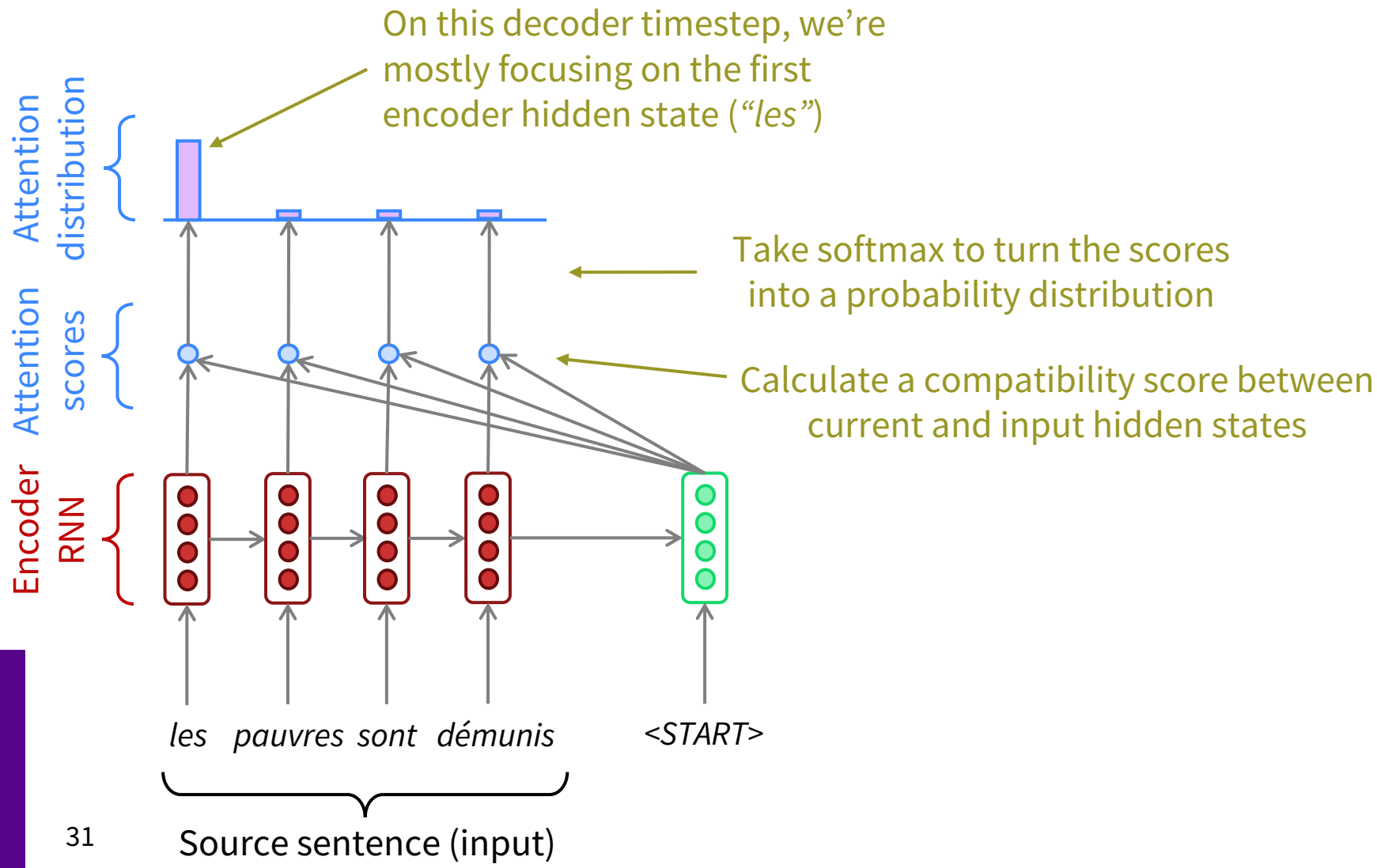
4d. (Selective) Attention

- Problem: The NMT system had to remember the entire input in one hidden vector … regardless of how long the input sentence was
- And in general we might like to have a large external memory and to selectively look at certain bits of information in it at various times
- Solution: Neural **Attention**

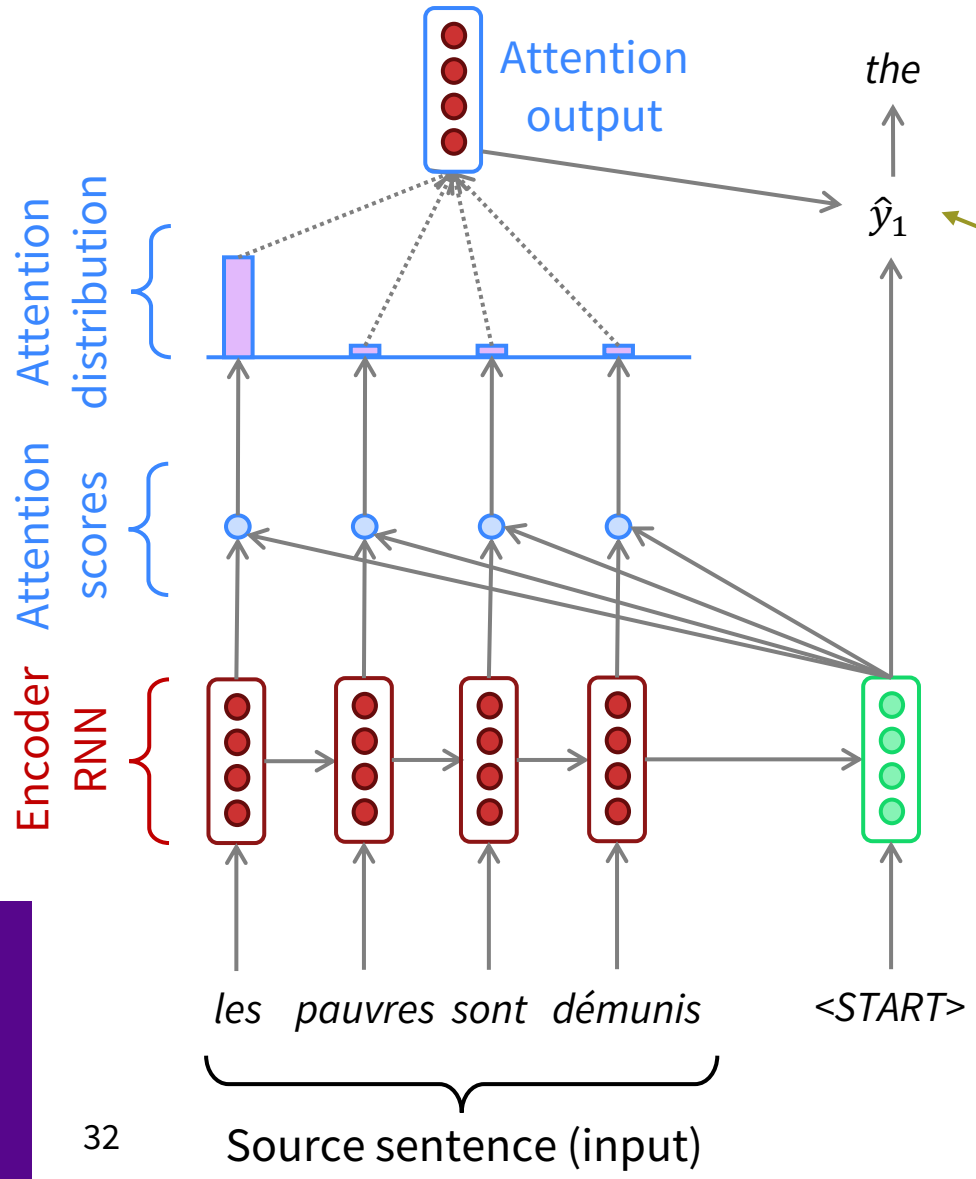
Sequence-to-sequence: bottleneck problem



Sequence-to-sequence with attention



Sequence-to-sequence with attention



Use the attention distribution to take a **weighted sum** of the encoder hidden states.

Concatenate attention output with decoder hidden state, then use to compute $\hat{y}_1$ as before

$$\boldsymbol{e}^t = [\boldsymbol{s}_t^T \boldsymbol{h}_1, \ldots, \boldsymbol{s}_t^T \boldsymbol{h}_N] \in \mathbb{R}^N$$

$$\alpha^t = \text{softmax}(\boldsymbol{e}^t) \in \mathbb{R}^N$$

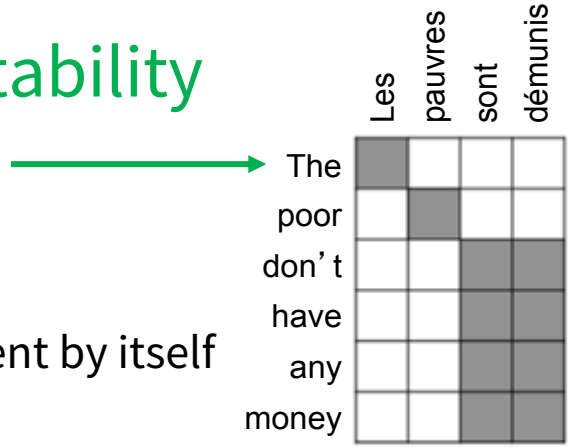
$$\boldsymbol{a}_t = \sum_{i=1}^{N} \alpha_i^t \boldsymbol{h}_i \in \mathbb{R}^h$$

Output: $g_w([\boldsymbol{a}_t, \boldsymbol{s}_t])$

Decoder RNN

Attention is great

- Attention significantly improves NMT performance
  - It's very useful to allow decoder to focus on certain parts of the source
- Attention solves the bottleneck problem
  - Attention allows decoder to look directly at source; bypass bottleneck
- Attention helps with vanishing gradient problem
  - Rather than traditional optimization problems, attention provides an alternative shortcut to faraway states
- Attention provides some interpretability
  - By inspecting attention distribution, we can see what the decoder was focusing on
  - We get alignment in MT for free!
  - This is cool because the network learned alignment by itself



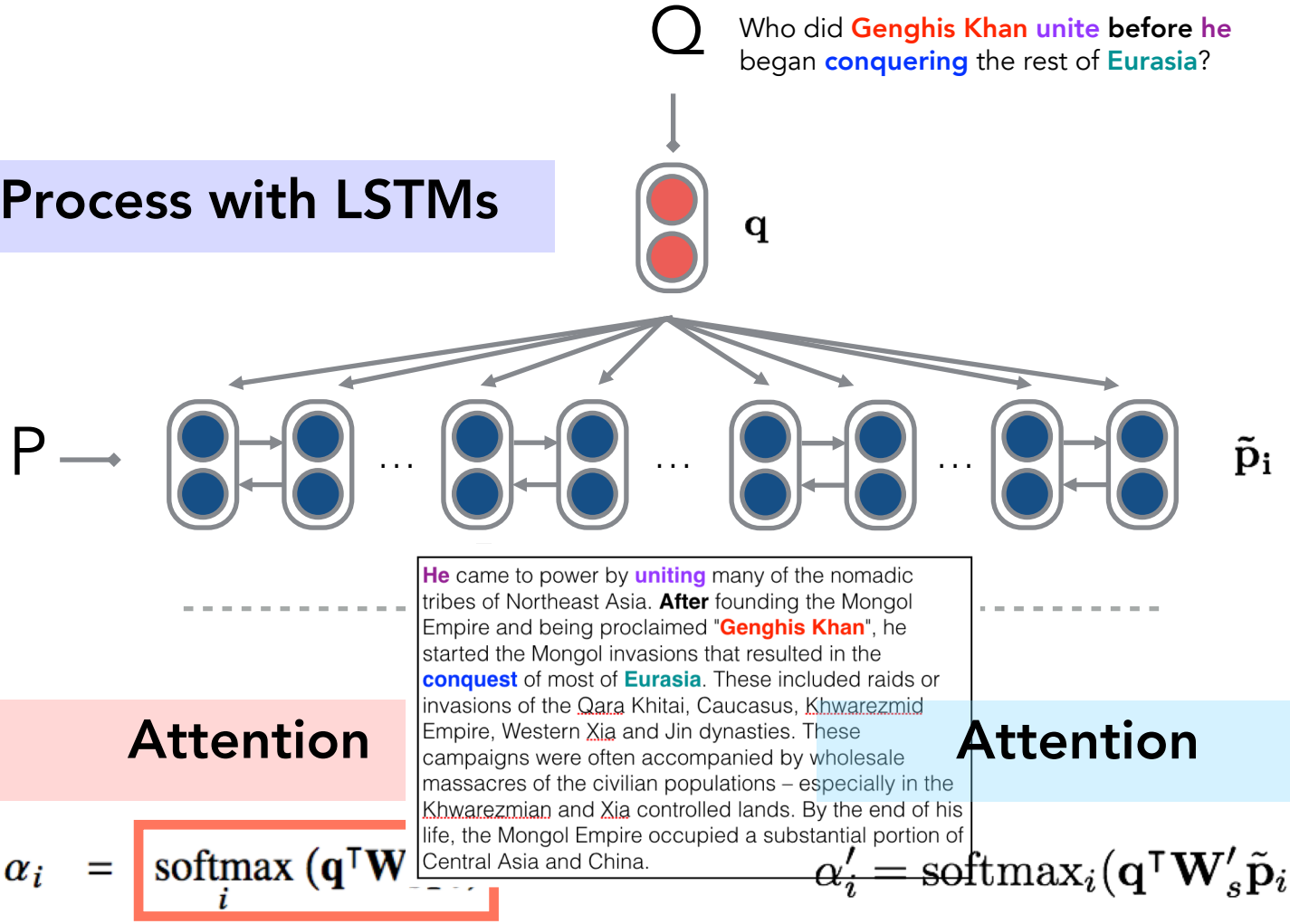
Attention is a *general* Deep Learning technique

- **Variant attention functions**:
  - Basic dot-product attention: $\boldsymbol{e}_i = \boldsymbol{s}^T \boldsymbol{h}_i \in \mathbb{R}$
  - Multiplicative attention: $\boldsymbol{e}_i = \boldsymbol{s}^T \boldsymbol{W} \boldsymbol{h}_i \in \mathbb{R}$
  - Additive attention: $\boldsymbol{e}_i = \boldsymbol{v}^T \tanh(\boldsymbol{W}_1 \boldsymbol{h}_i + \boldsymbol{W}_2 \boldsymbol{s}) \in \mathbb{R}$

- **More general definition of attention**:
  - Given a set of vector *keys* with associated vector *values*, and a vector *query*, **attention** is a technique to compute a weighted sum of the values, dependent on the query
  - We measure a score between the query and each key and then return an aggregate value which is a score-weighted average of the values
    - E.g. Key-value Memory Networks: (Miller et al. 2016) https://arxiv.org/abs/1606.03126

Attention in Question Answering

(Stanford Attentive Reader: Chen et al. 2016, 2017)



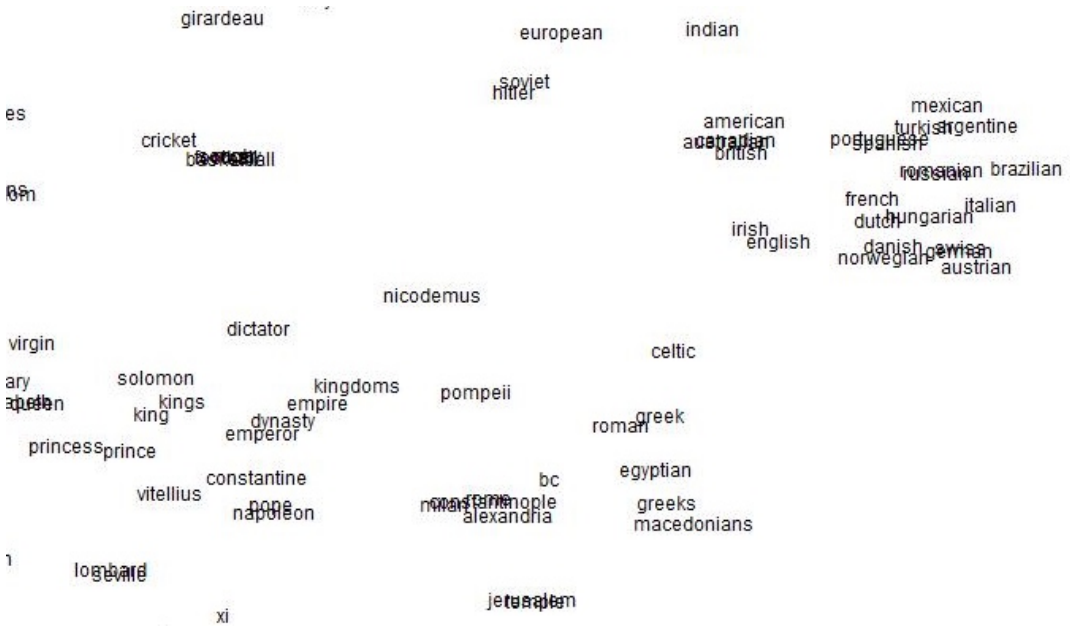
4e. Final thoughts

Here are a few “empirical understandings” of a decade of deep learning, some of which deserve better formal understanding

Non-convex models really are more powerful than convex models

- Extremely deep models using the residual connections (ResNets) idea, have powered the 2015–2017 gains on ImageNet
  - But what are they doing? With residual connections it doesn’t seem to be quite the traditional idea of layers of representation

Use of distributed representations of categorical values like words allows very effective modeling and sharing of dimensions of similarity



Models with orders of magnitude more parameters than there are training data available, if trained with ample amounts of regularization (including new forms like dropout), will *outperform* simpler models – *generalizing better* to new data

- Practitioner's recipe: build a sufficiently high-capacity model that it can be trained to 0% training error, and then increase its regularization until it doesn't overfit on dev

When we were young, our parents used to warn us with tales of bad local minima, but it turns out that bad local minima typically don’t exist and any different minima seem to be roughly equivalently good. No need to worry!

Rather than a clean separation between an objective (loss) function and an optimizer that is trying to minimize it, we find that in current neural network successes, the optimizer is mixed up into the loss function – small batch SGD is acting as a form of regularizer

Thanks!

Questions?