# Natural Language Processing

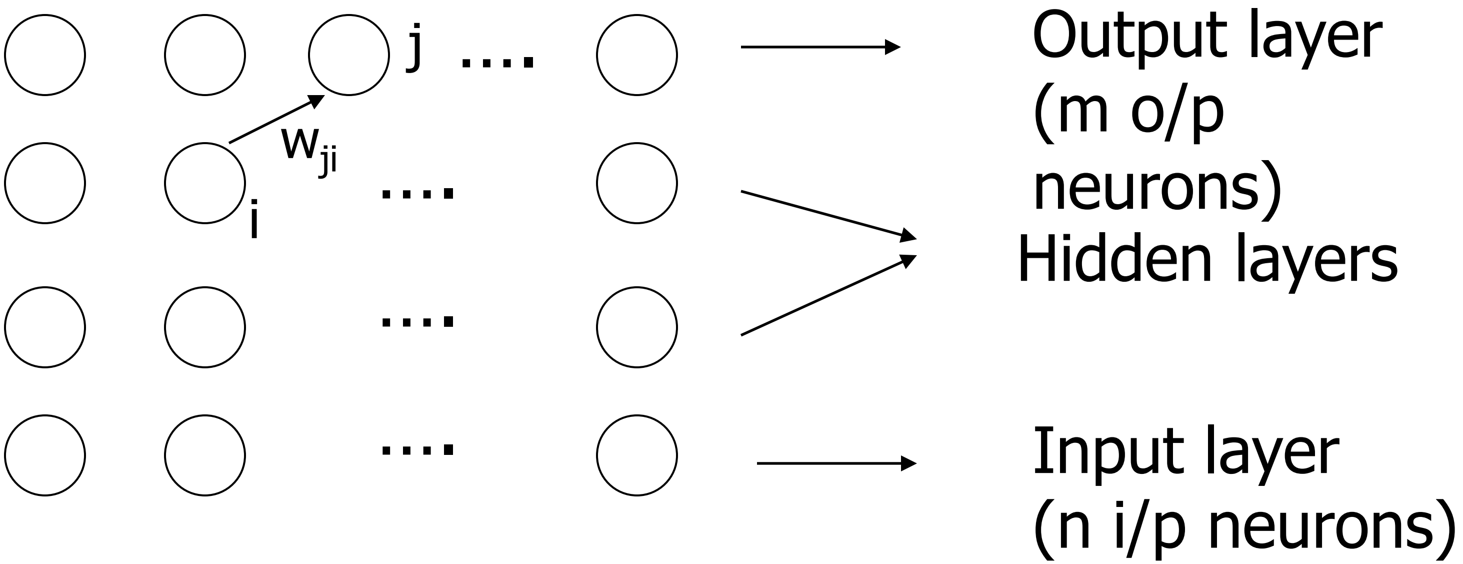
Pushpak Bhattacharyya
CSE Dept,
IIT Patna and Bombay

LSTM

# Recap

# Feedforward Network and Backpropagation

# Backpropagation algorithm



Output layer
(m o/p
neurons)

Hidden layers

Input layer
(n i/p neurons)

- Fully connected feed forward network
- Pure FF network (no jumping of connections over layers)

# General Backpropagation Rule

- General weight updating rule:

$$\Delta w_{ji} = \eta \delta j o_i$$

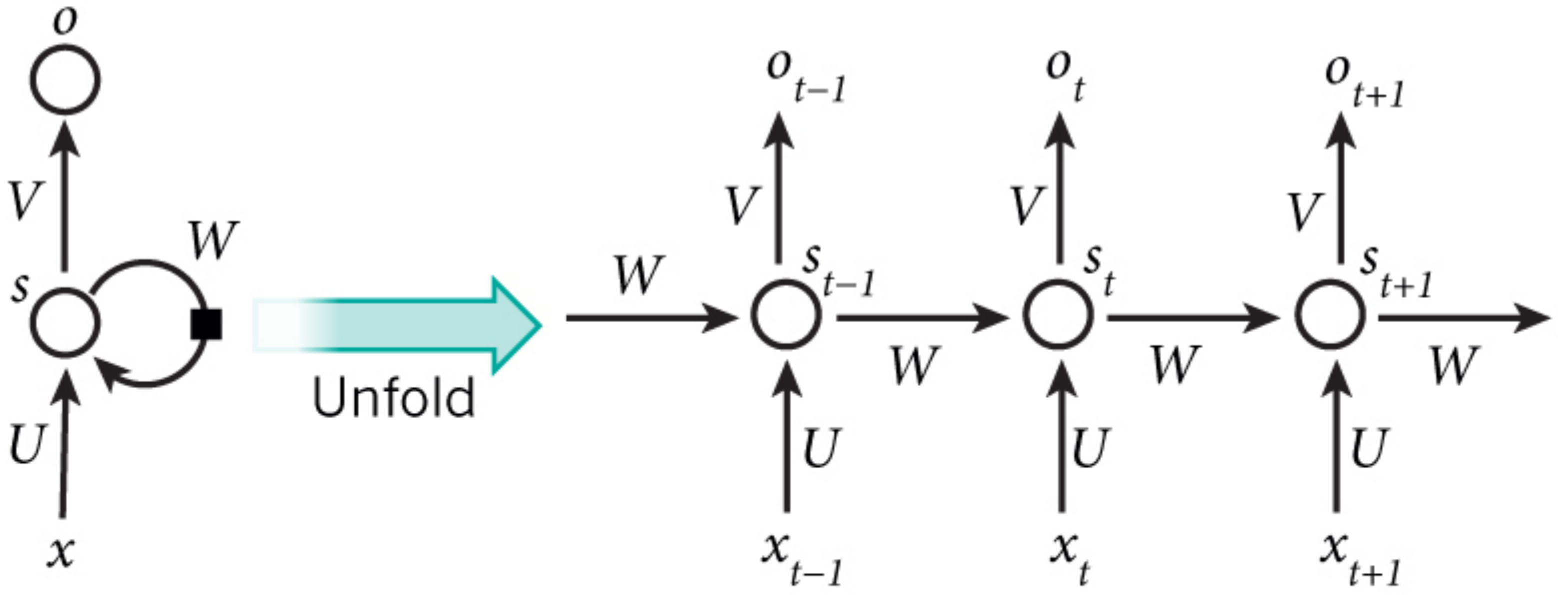
- Where

$$\delta_j = (t_j - o_j) o_j (1 - o_j) \quad \text{for outermost layer}$$

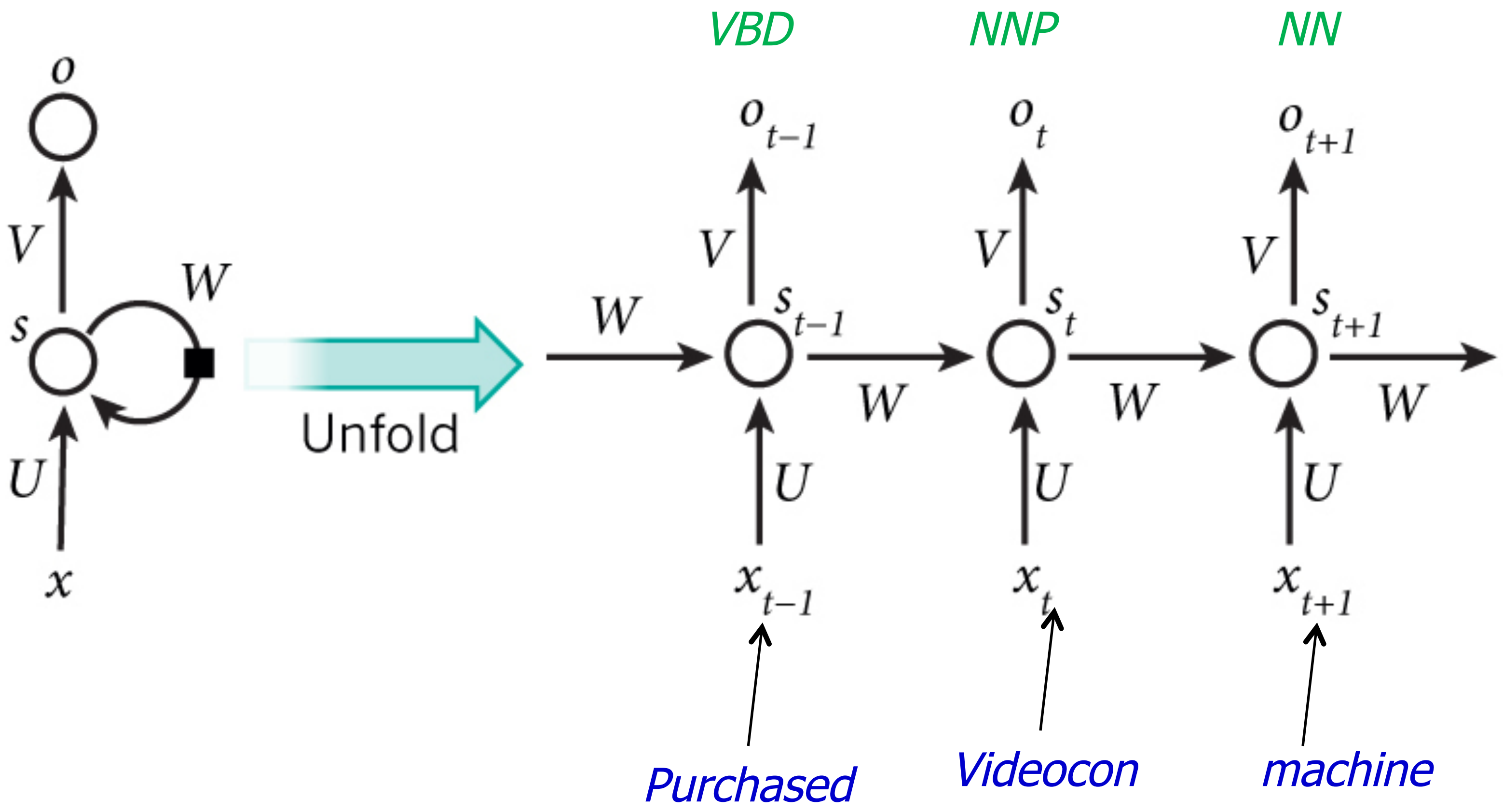
$$= \sum_{k \in \text{next layer}} (w_{kj} \delta_k) o_j (1 - o_j) o_i \text{ for hidden layers}$$

# Recurrent Neural Network

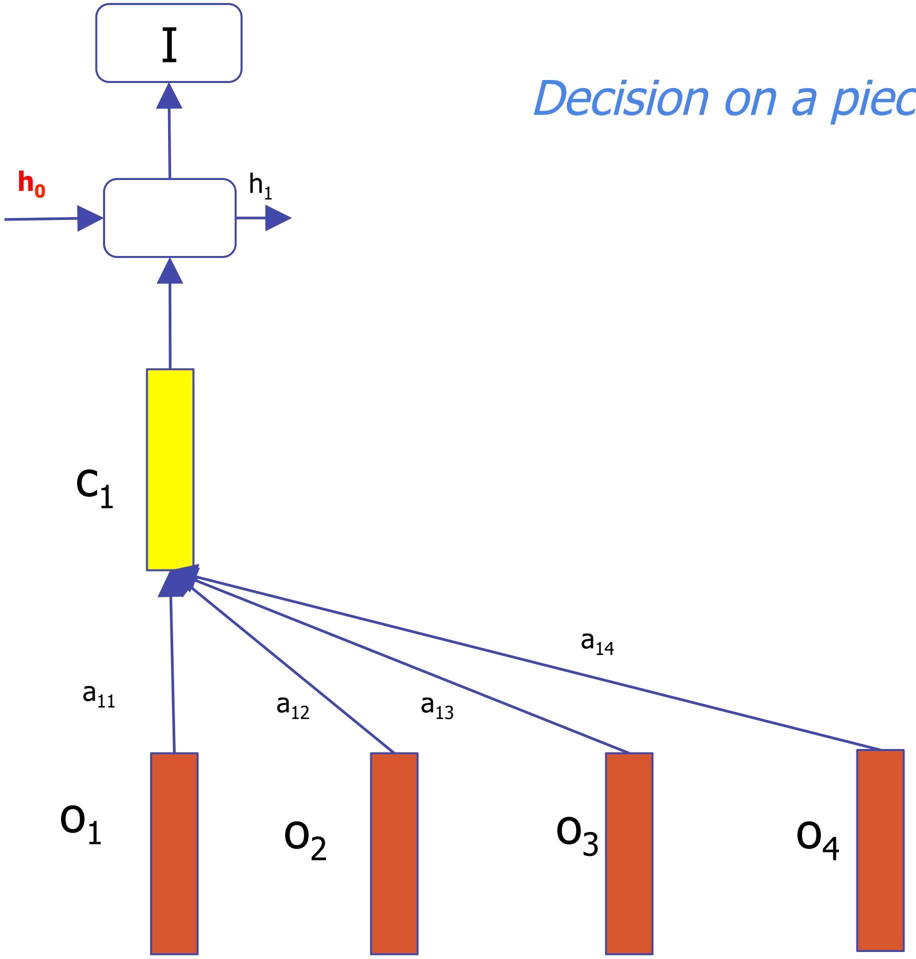
# Sequence processing m/c



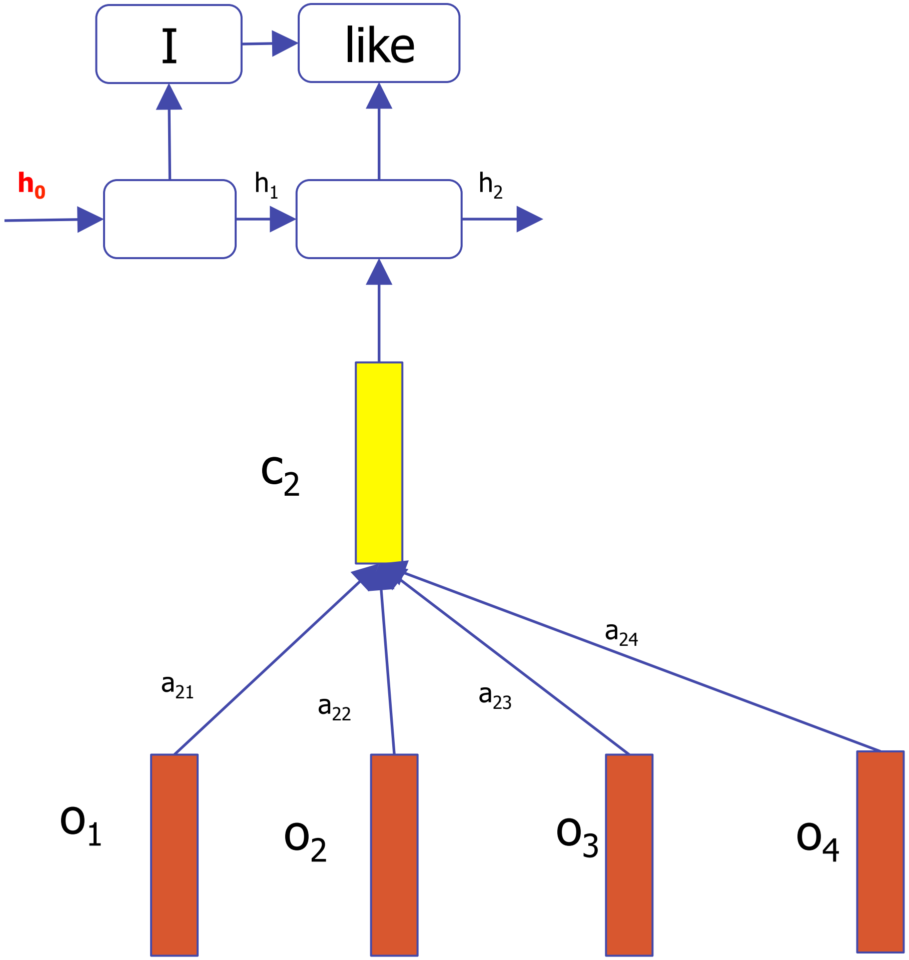
# E.g. POS Tagging

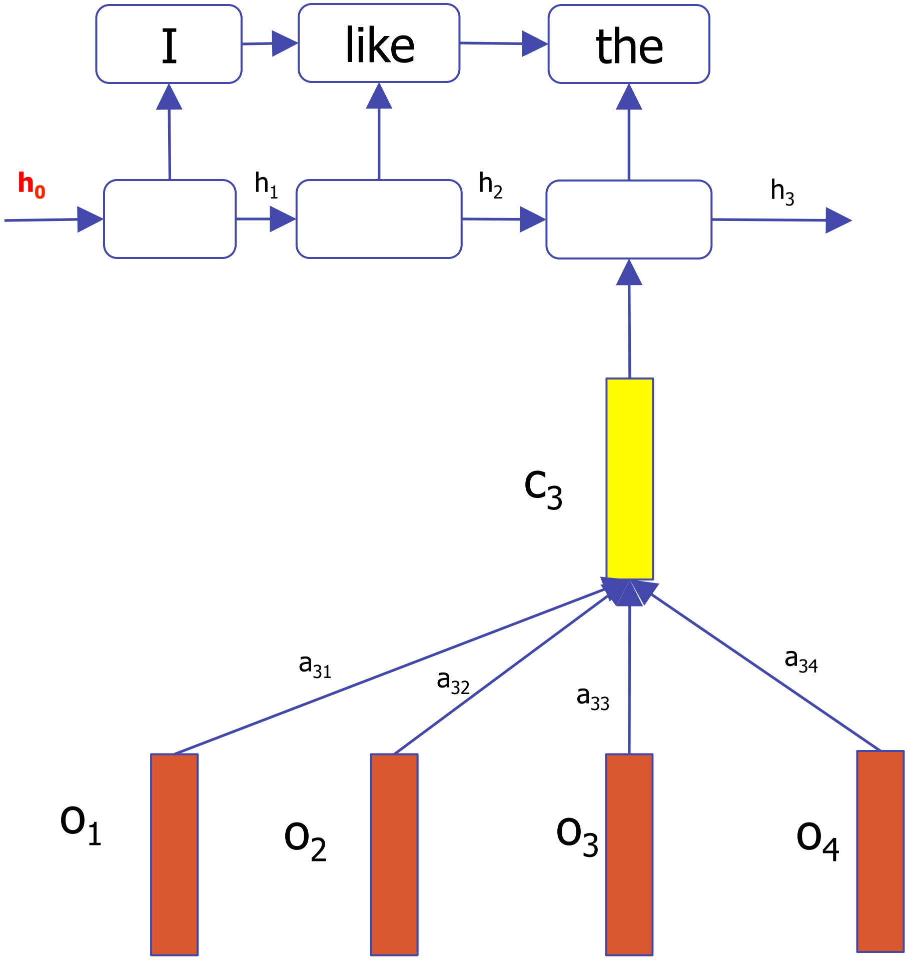


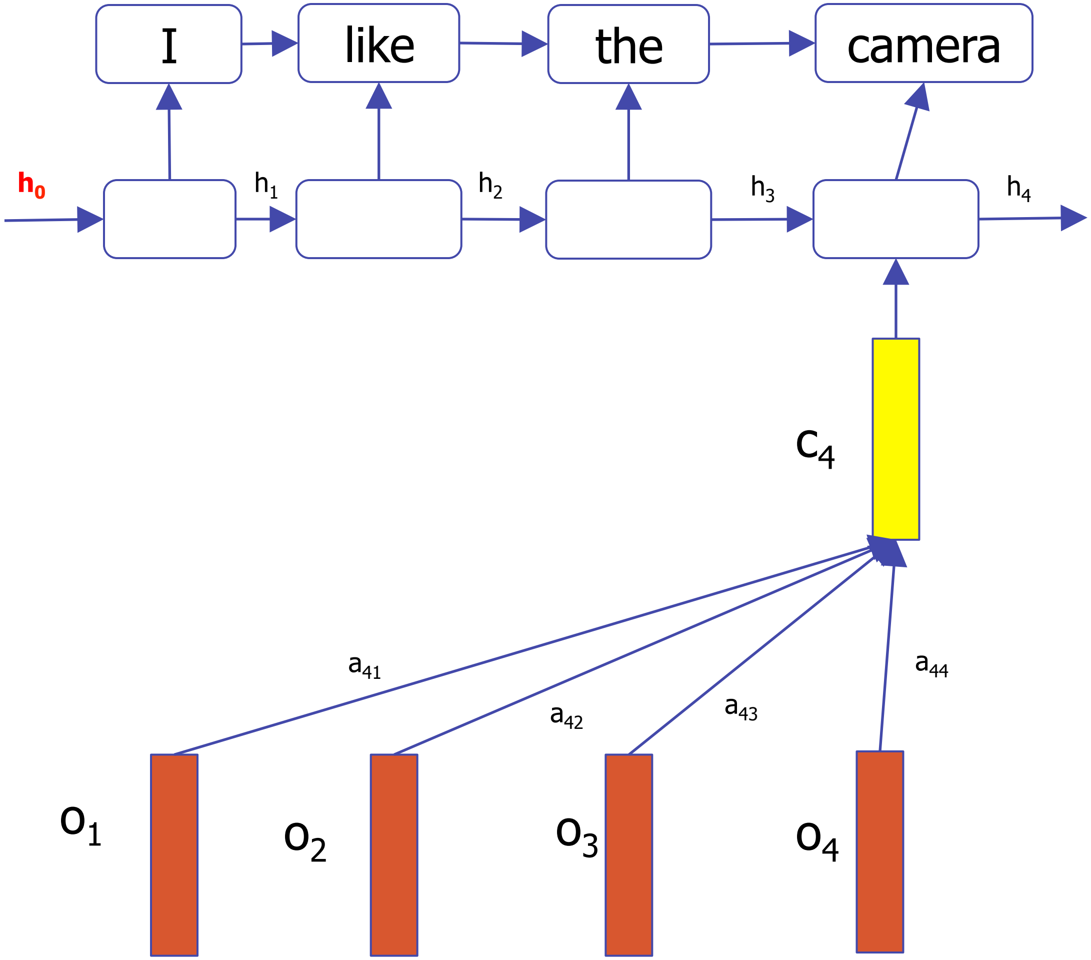
# E.g. Sentiment Analysis

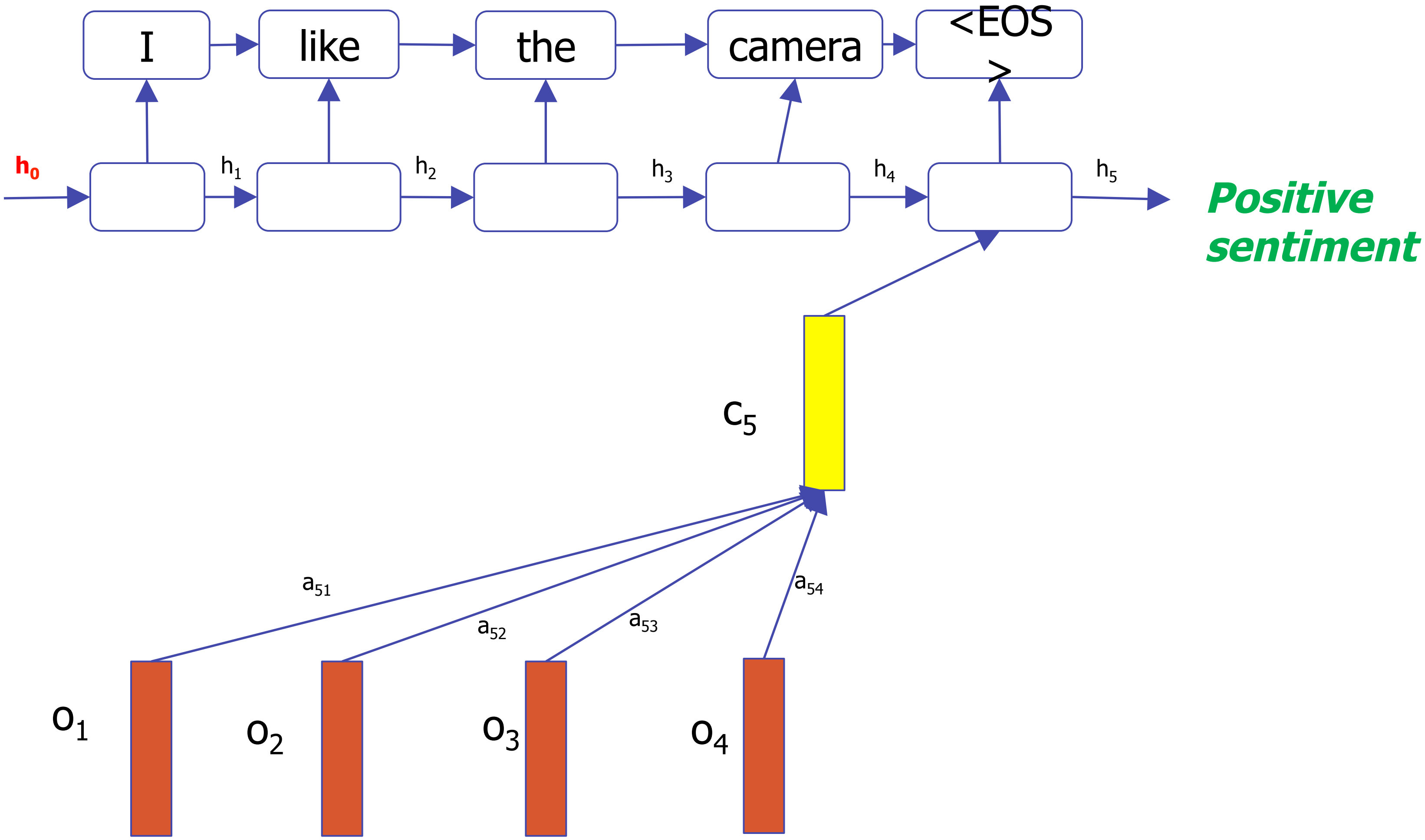


*Decision on a piece of text*


I
like
h0
h1
h2
c2
a24
a21
a22
a23
o1
o2
o3
o4


I
like
the
$h_0$
$h_1$
$h_2$
$h_3$
$c_3$
$a_{31}$
$a_{32}$
$a_{33}$
$a_{34}$
$o_1$
$o_2$
$o_3$
$o_4$


I
like
the
camera
$h_0$
$h_1$
$h_2$
$h_3$
$h_4$
$c_4$
$a_{41}$
$a_{42}$
$a_{43}$
$a_{44}$
$o_1$
$o_2$
$o_3$
$o_4$


I
like
the
camera
<EOS
>
$h_0$
$h_1$
$h_2$
$h_3$
$h_4$
$h_5$
Positive sentiment
$c_5$
$a_{51}$
$a_{52}$
$a_{53}$
$a_{54}$
$o_1$
$o_2$
$o_3$
$o_4$

# Notation: input and state

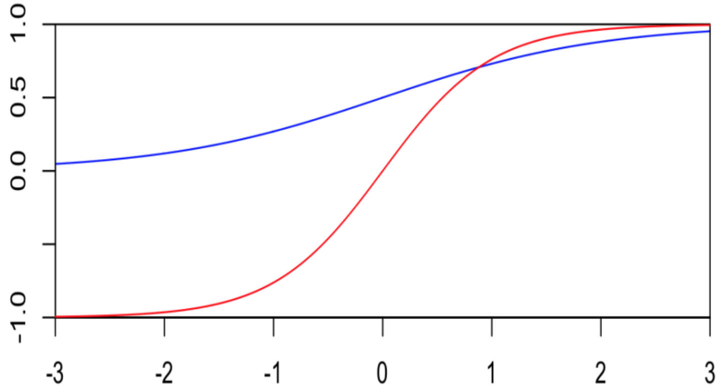
- $x_t$ : input at time step *t*
- $s_t$ : hidden state at time step *t*. It is the "memory" of the network.
- $s_t = f(U.x_t + Ws_{t-1})$ ***U* and *W* matrices are learnt**

- *f* is Usually *tanh* or *ReLU* (approximated by *softplus*)

# *Tanh*, *ReLU* (rectifier linear unit) and Softplus

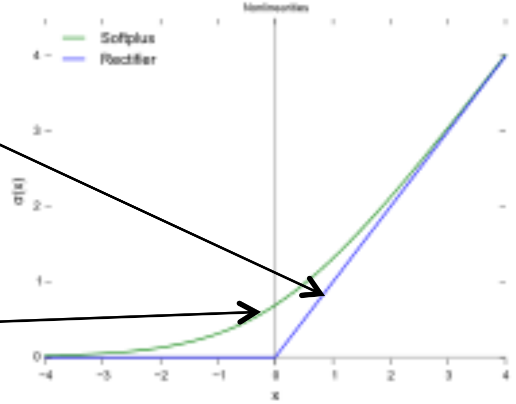
$$\tanh = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

tanh =



$$f(x) = \max(0, x)$$

$$g(x) = \ln(1 + e^{x})$$



# Notation: output

- $o_t$ is the output at step $t$

- For example, if we wanted to predict the next word in a sentence it would be a vector of probabilities across our vocabulary

- $o_t = softmax(V.s_t)$

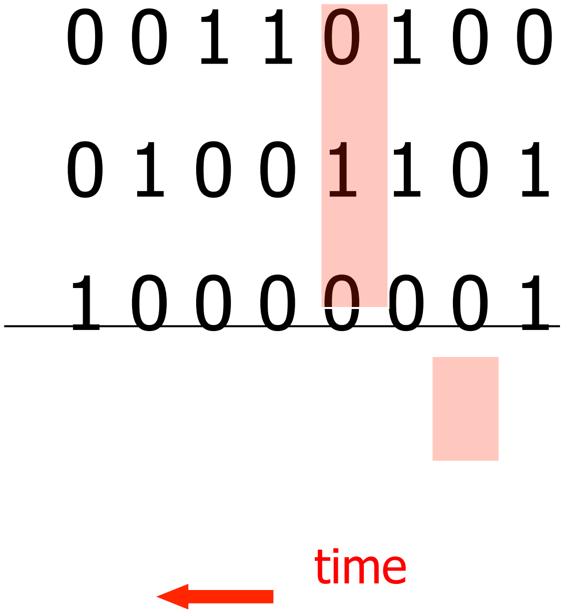
# Backpropagation through time (BPTT algorithm)

- The forward pass at each time step.
-
- The backward pass computes the error derivatives at each time step.

- After the backward pass we add together the derivatives at all the different times for each weight.

# A recurrent net for binary addition

- Two input units and one output unit.
- Given two input digits at each time step.
- The desired output at each time step is the output for the column that was provided as input two time steps ago.
  - It takes one time step to update the hidden units based on the two input digits.
  - It takes another time step for the hidden units to cause the output.



0 0 1 1 0 1 0 0

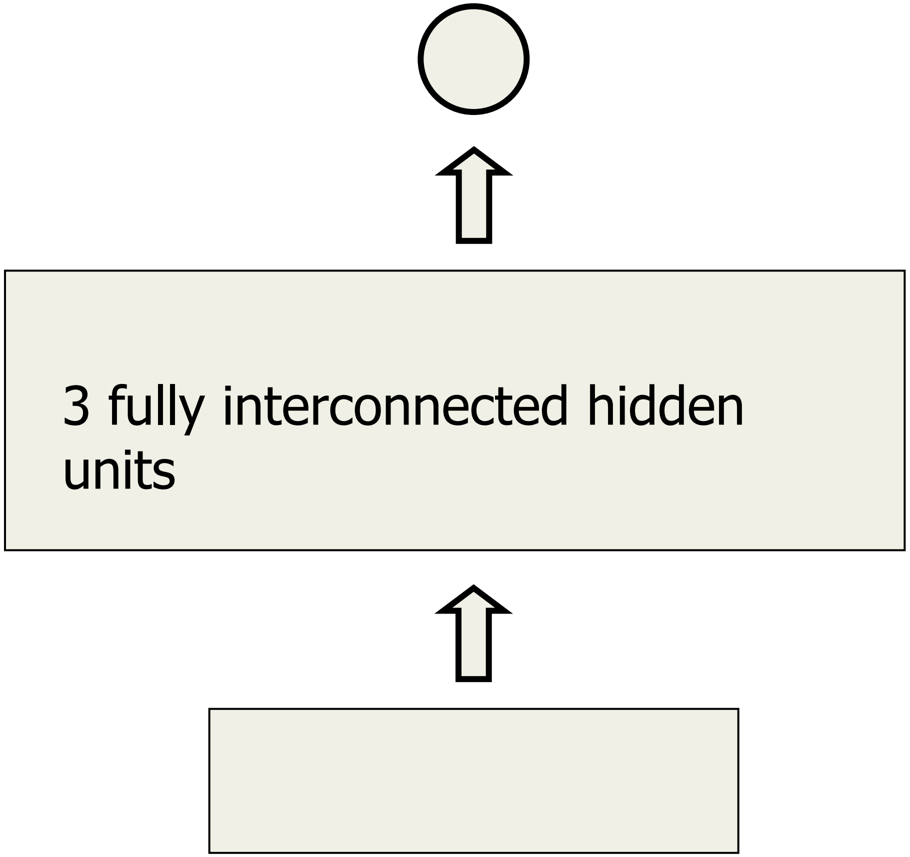
0 1 0 0 1 1 0 1

1 0 0 0 0 0 0 1

# The connectivity of the network

- The input units have feed forward connections

- Allow them to vote for the next hidden activity pattern.



# What the network learns

- Learns four distinct patterns of activity for the 3 hidden units.
- Patterns correspond to the nodes in the finite state automaton
- Nodes in FSM are like activity vectors
- The automaton is restricted to be in exactly one state at each time
- The hidden units are restricted to have exactly one vector of activity at each time.

# Recall: Backpropagation Rule

- General weight updating rule:

$$\Delta w_{ji} = \eta \delta j o_i$$

- Where

$$\delta_j = (t_j - o_j) o_j (1 - o_j)$$ for outermost layer

$$= \sum_{k \in \text{next layer}} (w_{kj} \delta_k) o_j (1 - o_j) o_i$$ for hidden layers

# The problem of exploding or vanishing gradients

- If the weights are  small, the gradients shrink exponentially

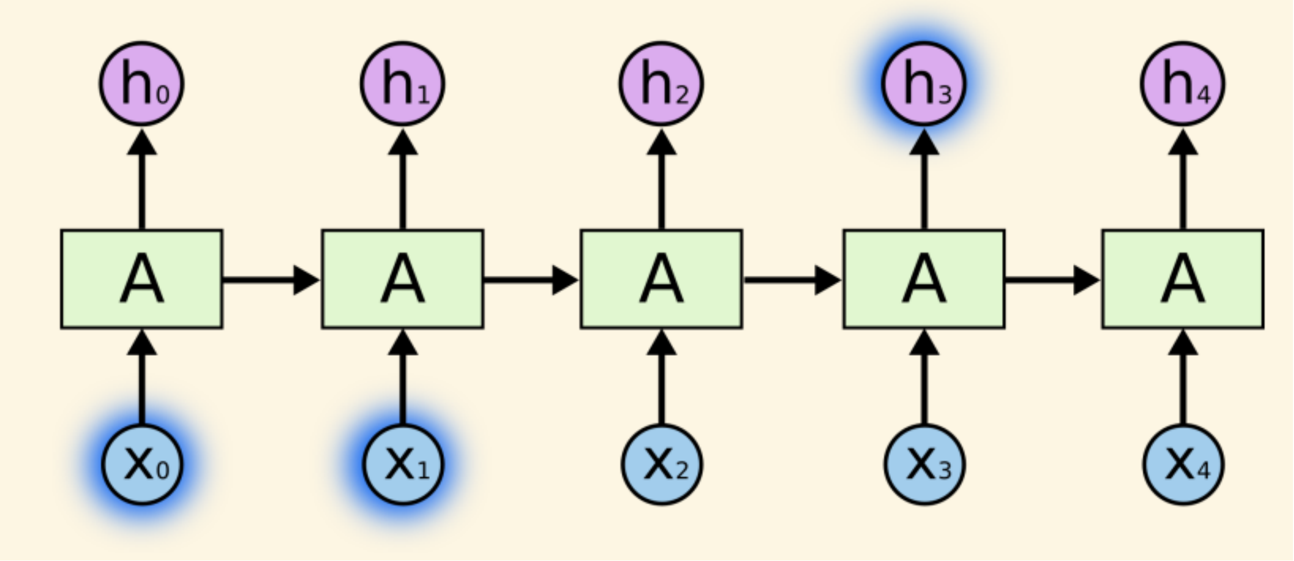
- If the weights are big the gradients grow exponentially.

- Typical feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.

# LSTM

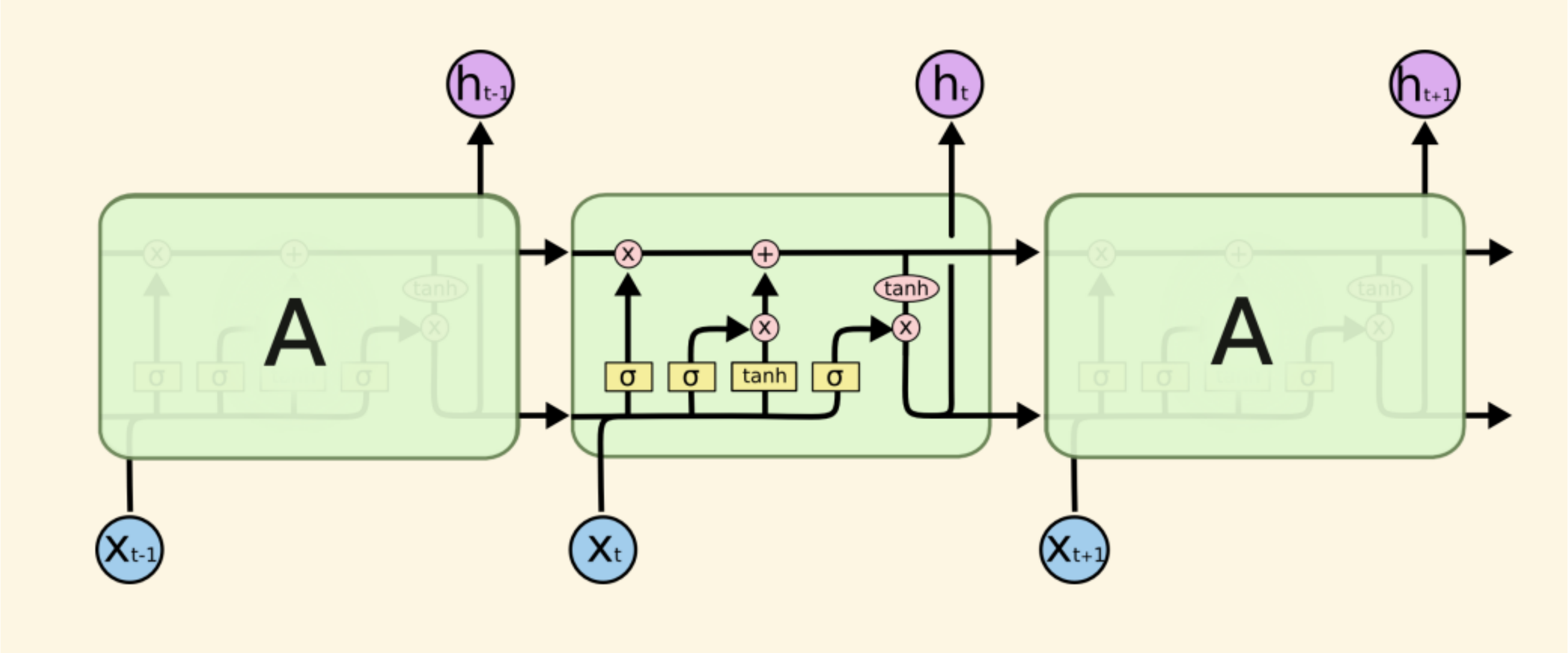
*(Ack: Lecture notes of Taylor Arnold, Yale* and [http://colah.github.io/posts/2015-08-Understanding-LSTMs/](http://colah.github.io/posts/2015-08-Understanding-LSTMs/))

# LSTM: a variation of vanilla RNN



Vanilla RNN

# LSTM: complexity within the block

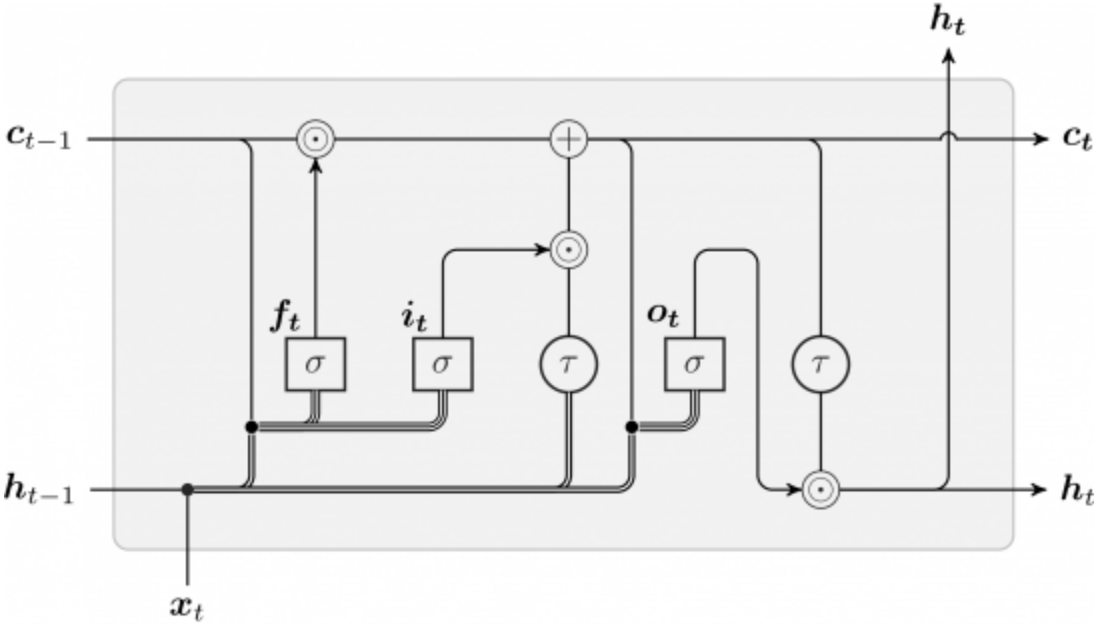


# Central idea

- Memory cell maintains its state over time

- Non-linear gating units regulate the information flow into and out of the cell

# A simple line diagram for LSTM



# Stepping through Constituents of LSTM

# Again: Example of Refrigerator complaint

- *Visiting service person is becoming rarer and rarer,*

  (ambiguous! ‘visit to service person’ OR ‘visit by service person’?)

…

- *and I am regretting/appreciating my decision to have bought the refrigerator from this company*

  (appreciating → ‘to’; regretting → ‘by’)

# Possibilities

- ‘Visiting’: ‘visit to’ or ‘visit by’ (ambiguity, syntactic opacity)
- Problem: solved or unsolved (not known, semantic opacity)
- ‘Appreciating’/’Regretting’: transparent; available on the surface

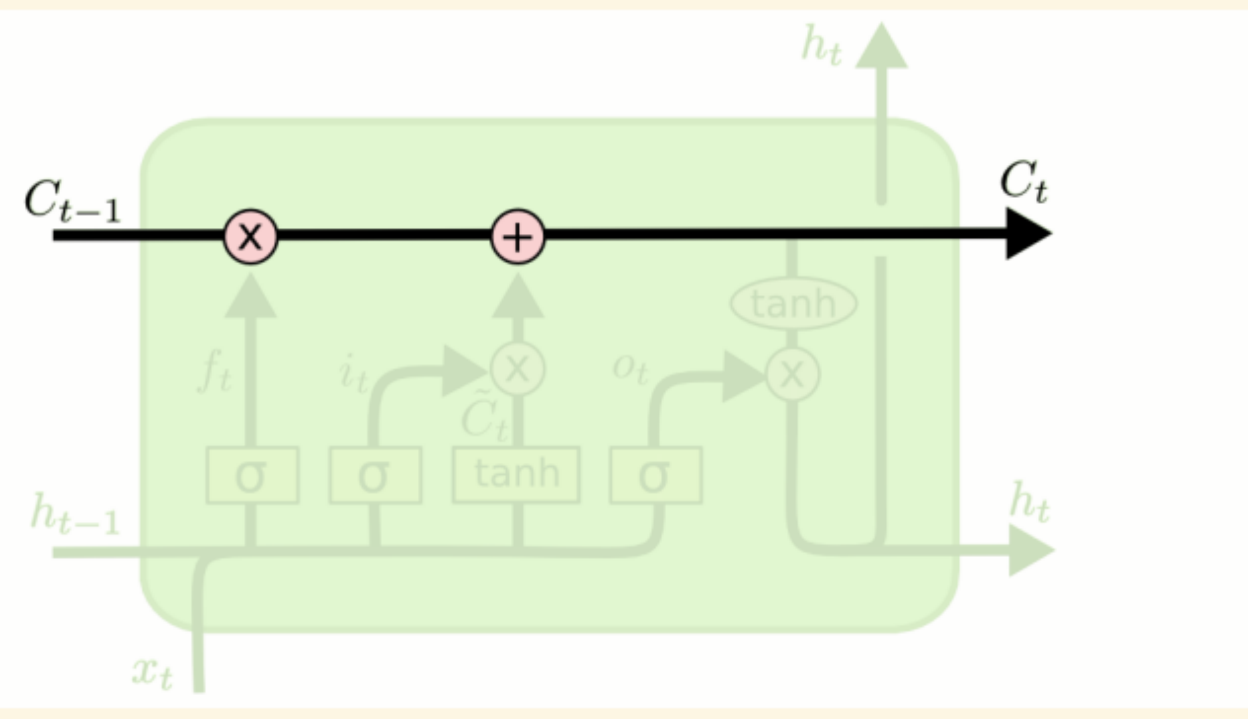
# 4 possibilities (states)

| **Clue-1** | **Clue-2** | **Problem** | **Sentiment** |
|---|---|---|---|
| *Visit to service person* | *Appreciating* | solved | Positive |
| *Visit to service person* | *Appreciating* | Not solved | Not making sense!<br>Incoherent |
| *Visit to service person* | *Regretting* | solved | May be reverse sarcasm |
| *Visit to service person* | *Regretting* | Not solved | Negative |

# 4 possibilities (states)

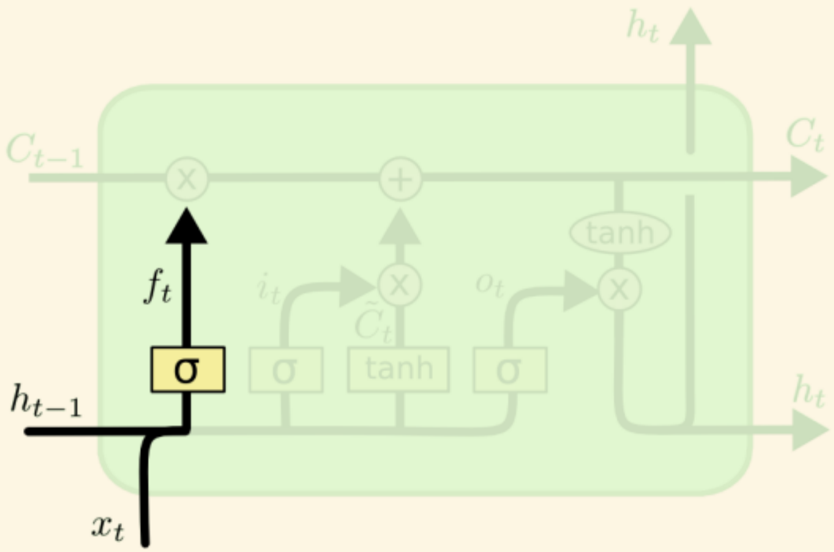
| **Clue-1** | **Clue-2** | **Problem** | **Sentiment** |
|---|---|---|---|
| *Visit by service person* | *Appreciating* | solved | Positive |
| *Visit by service person* | *Appreciating* | Not solved | May be sarcastic |
| *Visit by service person* | *Regretting* | solved | May be reverse sarcasm |
| *Visit by service person* | *Regretting* | Not solved | Negative |

# LSTM constituents: Cell State



The first and foremost component- the controller of flow of information

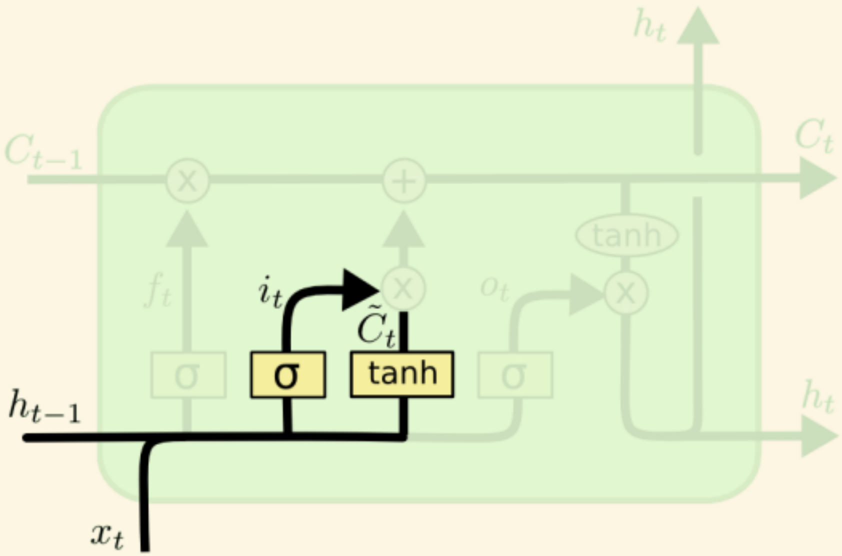
# LSTM constituents- Forget Gate



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

Helps forget irrelevant information. Sigmoid function. Output is between 0 and 1. Because of product, close to 1 will be full pass, close to 0 no pass

# LSTM constituents: Input gate

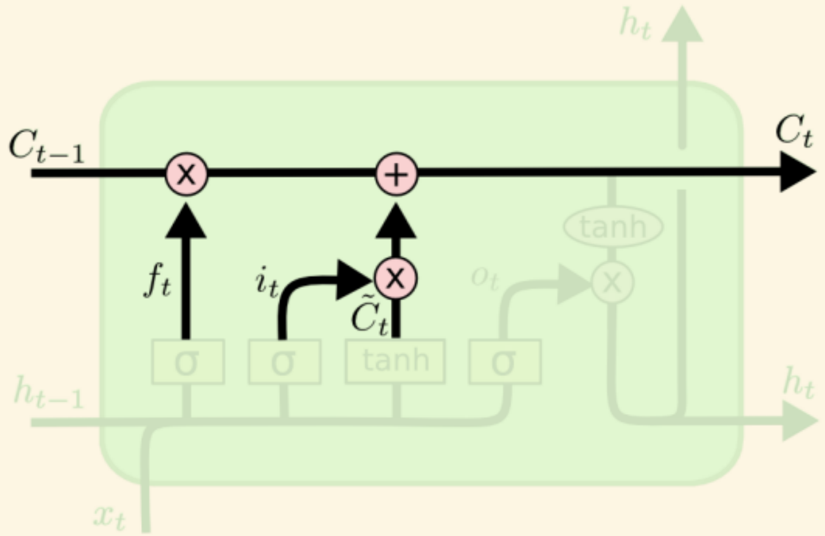


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

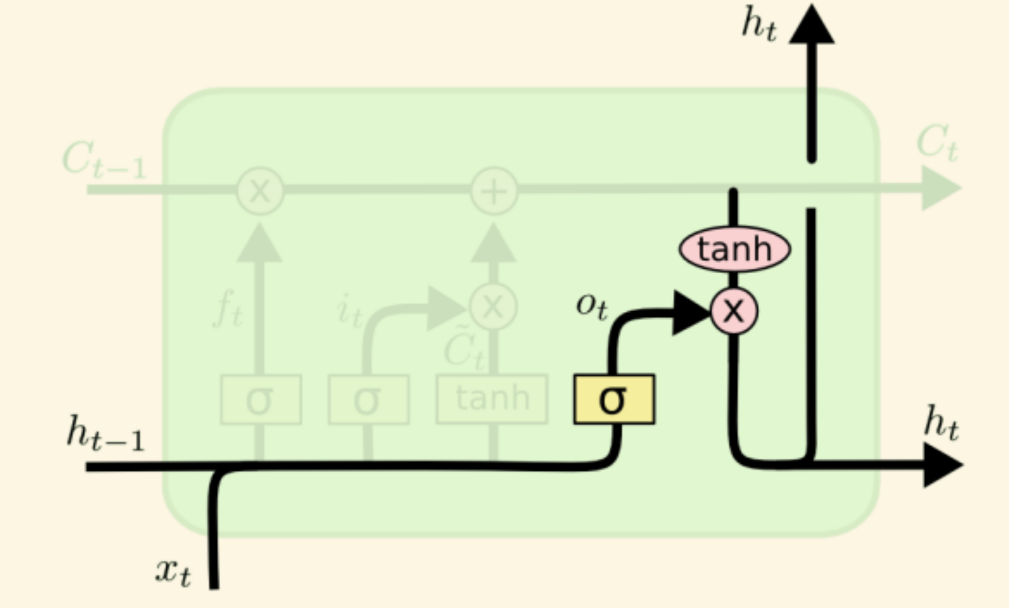
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

*tanh* produces a cell state vector; multiplied with input gate which again 0-1 controls what and how much input goes FOWARD

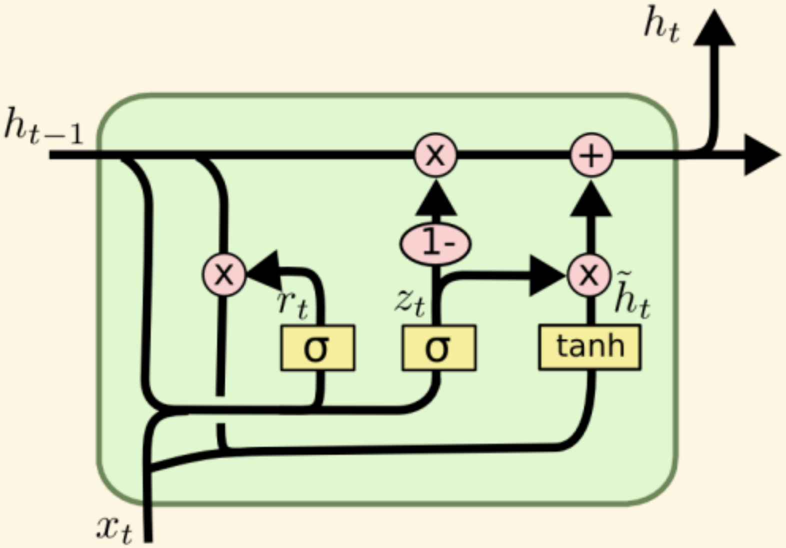
# Cell state operation



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

$h_t$
$C_{t-1}$
$C_t$
tanh
$f_t$
$i_t$
$o_t$
$\tilde{C}_t$
σ
σ
tanh
σ
$h_{t-1}$
$h_t$
$x_t$

# Finally



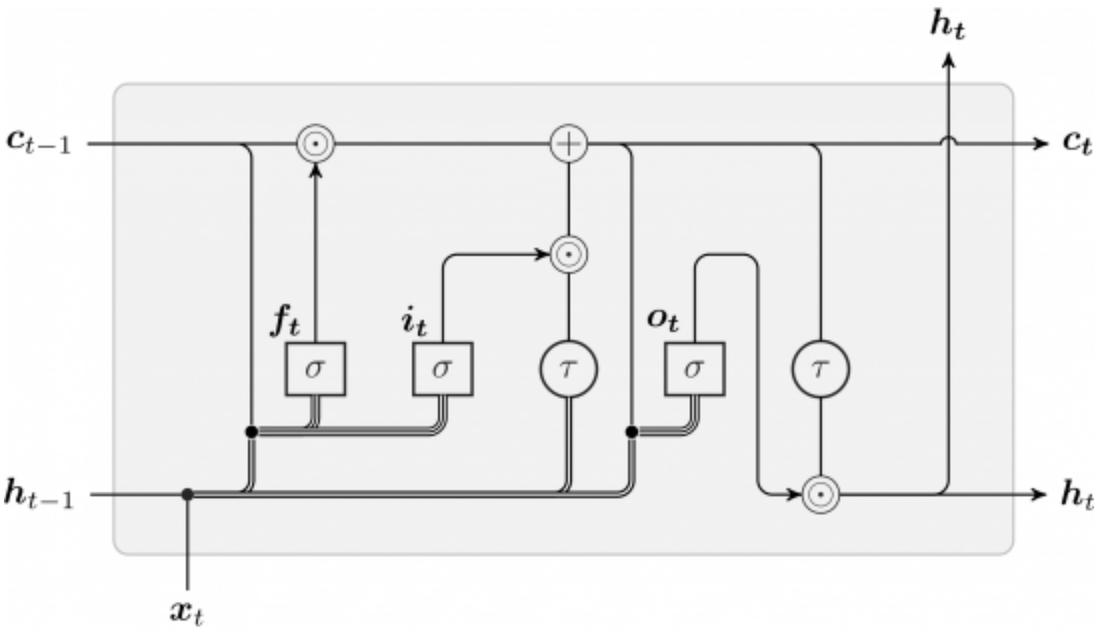
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

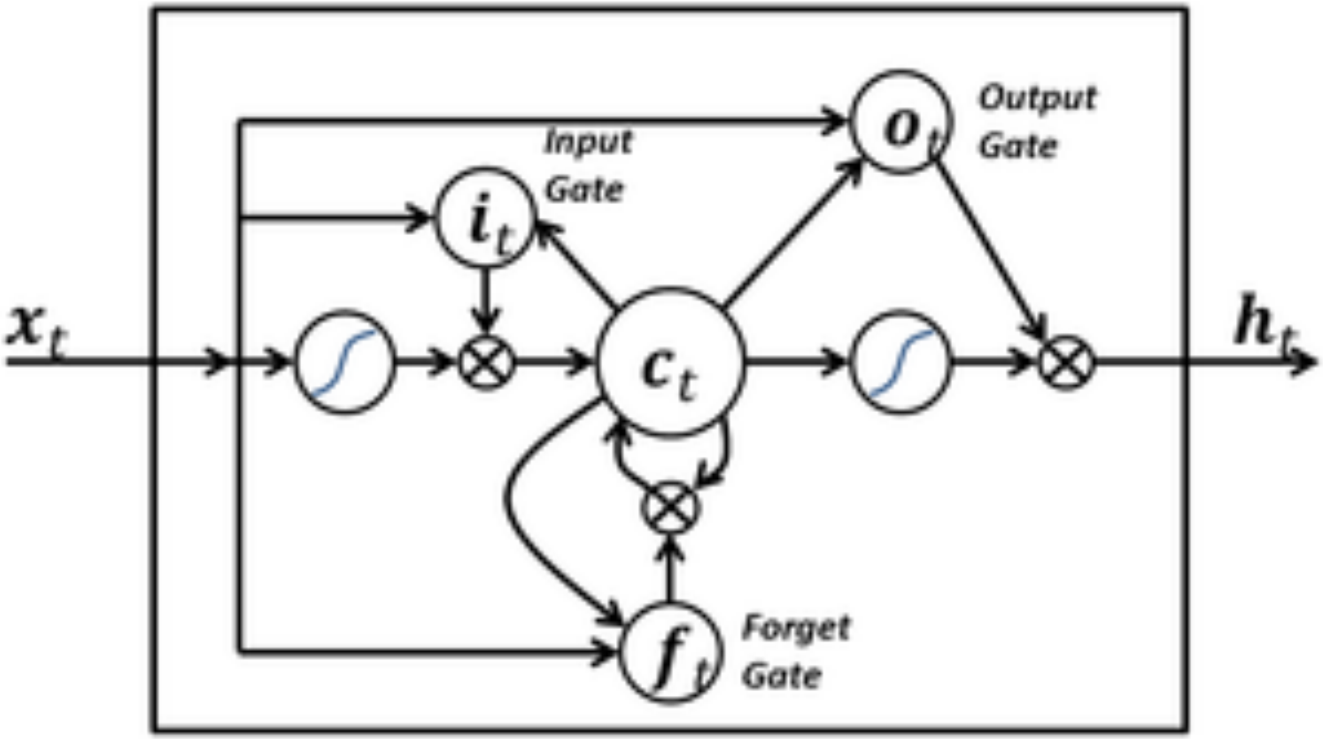
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

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

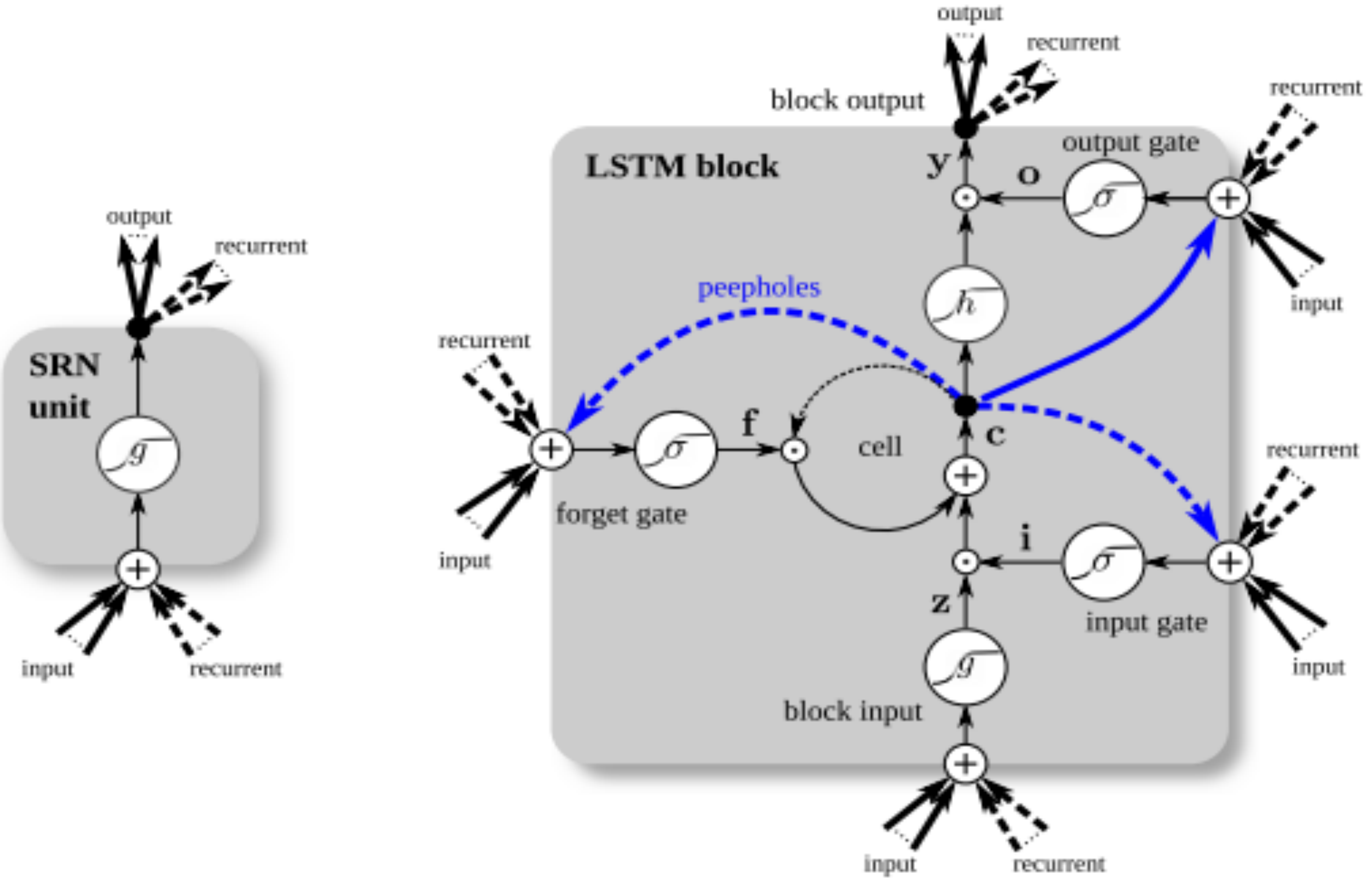
# Better picture (the one we started with)



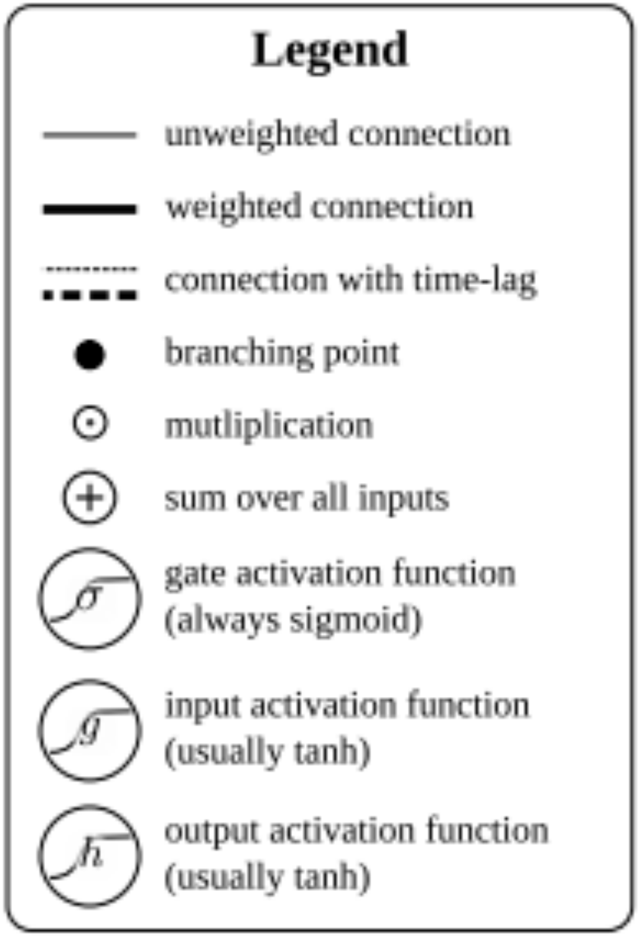
# Another picture



# LSTM schematic greff et al. LSTM a Space Odyssey, arxiv 2015



# Legend



**Legend**

| | |
|---|---|
| — | unweighted connection |
| ━ | weighted connection |
| ┅ | connection with time-lag |
| ● | branching point |
| ⊙ | mutliplication |
| ⊕ | sum over all inputs |
| $\sigma$ | gate activation function<br>(always sigmoid) |
| $g$ | input activation function<br>(usually tanh) |
| $h$ | output activation function<br>(usually tanh) |

# Required mathematics

$$
\begin{aligned}
\mathbf{z}^t &= g(\mathbf{W}_z\mathbf{x}^t + \mathbf{R}_z\mathbf{y}^{t-1} + \mathbf{b}_z) && \textit{block input} \\
\mathbf{i}^t &= \sigma(\mathbf{W}_i\mathbf{x}^t + \mathbf{R}_i\mathbf{y}^{t-1} + \mathbf{p}_i \odot \mathbf{c}^{t-1} + \mathbf{b}_i) && \textit{input gate} \\
\mathbf{f}^t &= \sigma(\mathbf{W}_f\mathbf{x}^t + \mathbf{R}_f\mathbf{y}^{t-1} + \mathbf{p}_f \odot \mathbf{c}^{t-1} + \mathbf{b}_f) && \textit{forget gate} \\
\mathbf{c}^t &= \mathbf{i}^t \odot \mathbf{z}^t + \mathbf{f}^t \odot \mathbf{c}^{t-1} && \textit{cell state} \\
\mathbf{o}^t &= \sigma(\mathbf{W}_o\mathbf{x}^t + \mathbf{R}_o\mathbf{y}^{t-1} + \mathbf{p}_o \odot \mathbf{c}^t + \mathbf{b}_o) && \textit{output gate} \\
\mathbf{y}^t &= \mathbf{o}^t \odot h(\mathbf{c}^t) && \textit{block output}
\end{aligned}
$$

# Training of LSTM

# Many layers and gates

- Though complex, in principle possible to train
- Gates are also *sigmoid* or *tanh* networks

- Remember the FUNDAMENTAL backpropagation rule

# General Backpropagation Rule

- General weight updating rule:

$$\Delta w_{ji} = \eta \delta j o_i$$

- Where

$$\delta_j = (t_j - o_j) o_j (1 - o_j) \quad \text{for outermost layer}$$

$$= \sum_{k \in \text{next layer}} (w_{kj} \delta_k) o_j (1 - o_j) o_i \text{ for hidden layers}$$

# LSTM tools

- *Tensorflow, Ocropus, RNNlib etc.*
- Tools do everything internally
- Still insights and concepts are inevitable

# LSTM applications

# Many applications

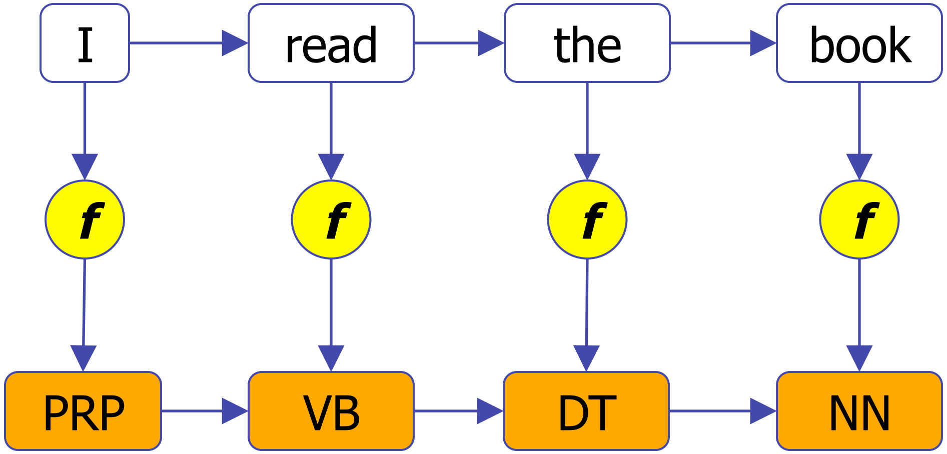
- Language modeling (The tensorflow tutorial on PTB is a good place to start Recurrent Neural Networks) character and word level LSTM's are used
- Machine Translation also known as sequence to sequence learning (https://arxiv.org/pdf/1409.3215.pdf)
- Image captioning (with and without attention, https://arxiv.org/pdf/1411.4555v...)
- Hand writing generation (http://arxiv.org/pdf/1308.0850v5...)
- Image generation using attention models - my favorite ( https://arxiv.org/pdf/1502.04623...)
- Question answering (http://www.aclweb.org/anthology/...)
- Video to text (https://arxiv.org/pdf/1505.00487...)

# Deep Learning Based Seq2Seq Models and POS Tagging

Acknowledgement: Anoop Kunchukuttan, PhD Scholar, IIT Bombay

*So far we are seen POS tagging as a sequence labelling task*

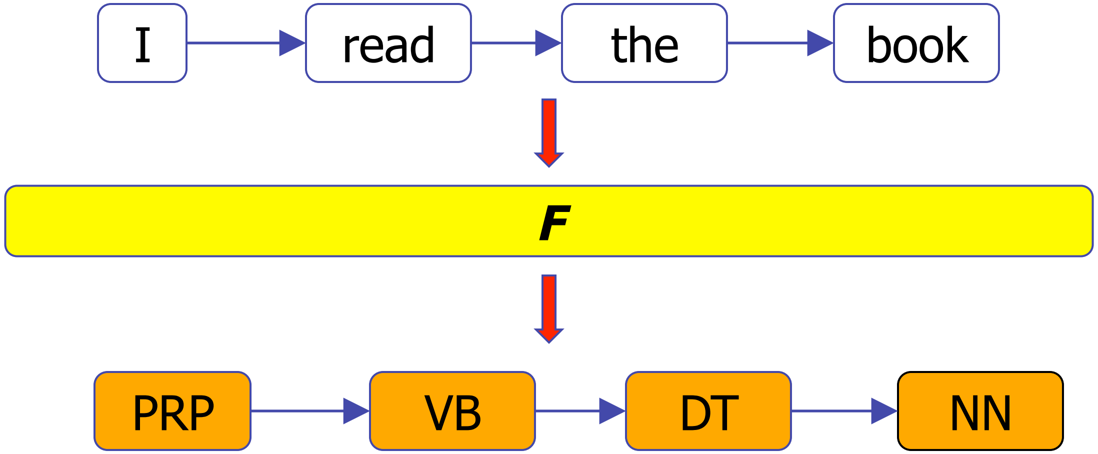
*For every element, predict the tag/label (using function **f** )*



- Length of output sequence is same as input sequence
- Prediction of tag at time *t* can use only the words seen till time *t*

*We can also look at POS tagging as a sequence to sequence transformation problem*

*Read the entire sequence and predict the output sequence (using function **F**)*



- Length of output sequence need not be the same as input sequence
- Prediction at any time step *t* has access to the entire input
- A more general framework than sequence labelling

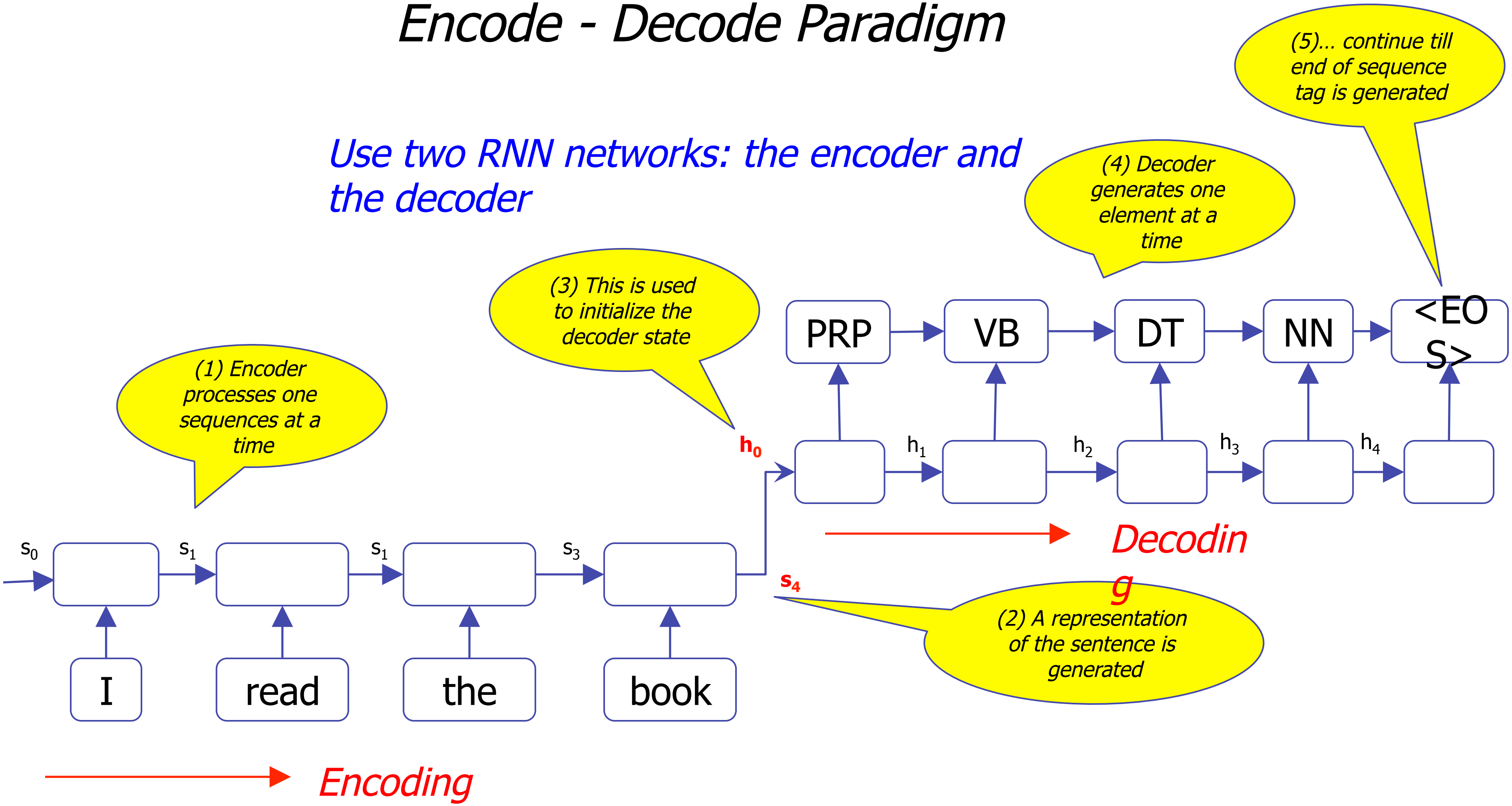
## *Sequence to Sequence transformation is a more general framework than sequence labelling*

- Many other problems can be expressed as sequence to sequence transformation
  - *e.g. machine translation, summarization, question answering, dialog*
- Adds more capabilities which can be useful for problems like MT:
  - many → many mappings: insertion/deletion to words, one-one mappings
  - non-monotone mappings: reordering of words
- For POS tagging, these capabilites are not required

*How does a sequence to sequence model work? Let's see two paradigms*

# *Encode - Decode Paradigm*

*Use two RNN networks: the encoder and the decoder*



*(1) Encoder processes one sequences at a time*

*(2) A representation of the sentence is generated*

*(3) This is used to initialize the decoder state*

*(4) Decoder generates one element at a time*

*(5)… continue till end of sequence tag is generated*

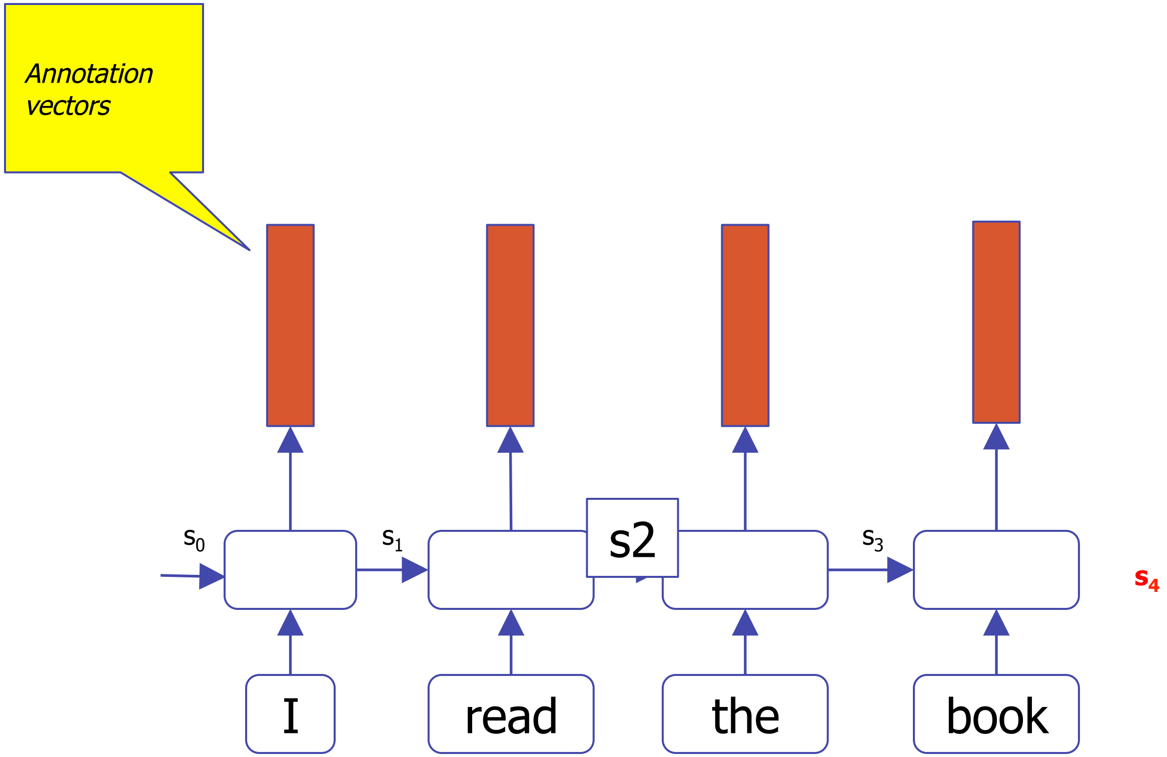
*This approach reduces the entire sentence representation to a single vector*

Two problems with this design choice:

- This is not sufficient to represent to capture all the syntactic and semantic complexities of a sentence
  - ○ *Solution: Use a richer representation for the sentences*
- Problem of capturing long term dependencies: The decoder RNN will not be able to able to make use of source sentence representation after a few time steps
  - ○ *Solution: Make source sentence information when making the next prediction*
  - ○ *Even better, make **RELEVANT** source sentence information available*

*These solutions motivate the next paradigm*

# *Encode - Attend - Decode Paradigm*



Represent the source sentence by the **set of output vectors** from the encoder

Each output vector at time *t* is a contextual representation of the input at time *t*

*Note: in the encoder-decode paradigm, we ignore the encoder outputs*

Let's call these encoder output vectors ***annotation vectors***

## *How should the decoder use the set of annotation vectors while predicting the next character?*

**Key Insight:**

(1)Not all annotation vectors are equally important for prediction of the next element

(2)The annotation vector to use next depends on what has been generated so far by the decoder

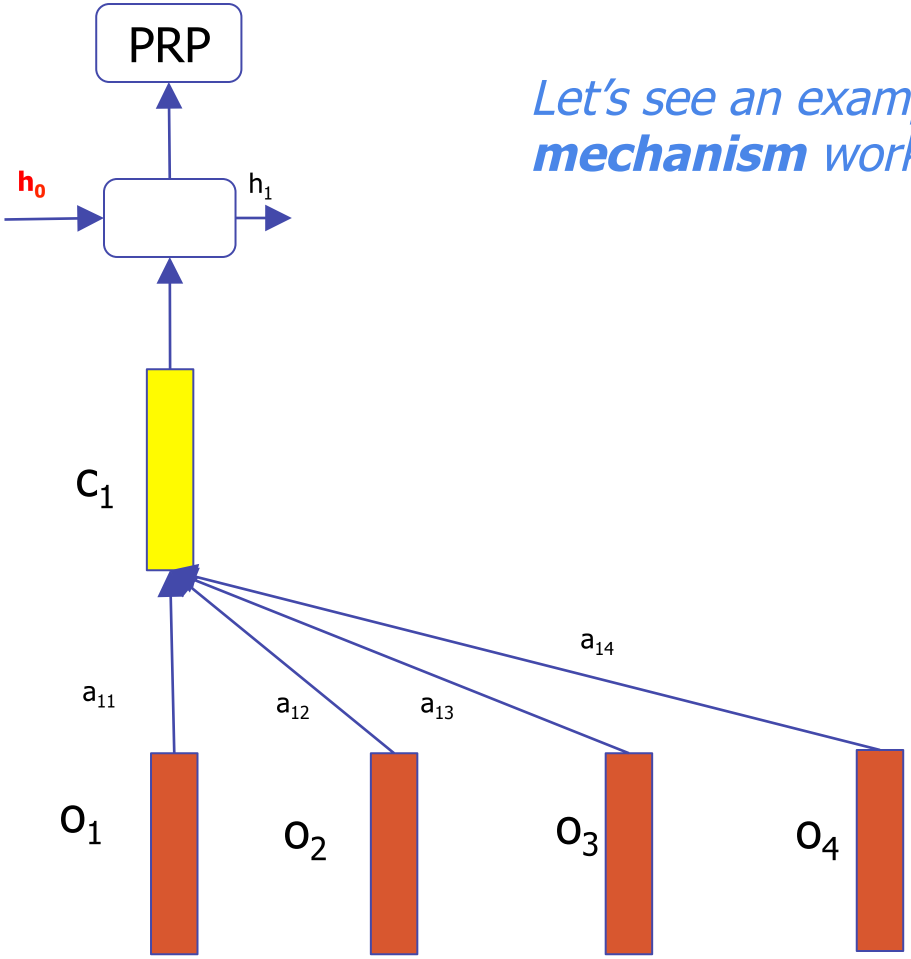
*eg.* To generate the 3rd POS tag, the 3rd annotation vector (hence 3rd word) is most important

One way to achieve this:
Take a weighted average of the annotation vectors, with more weight to annotation vectors which need more **focus or attention**

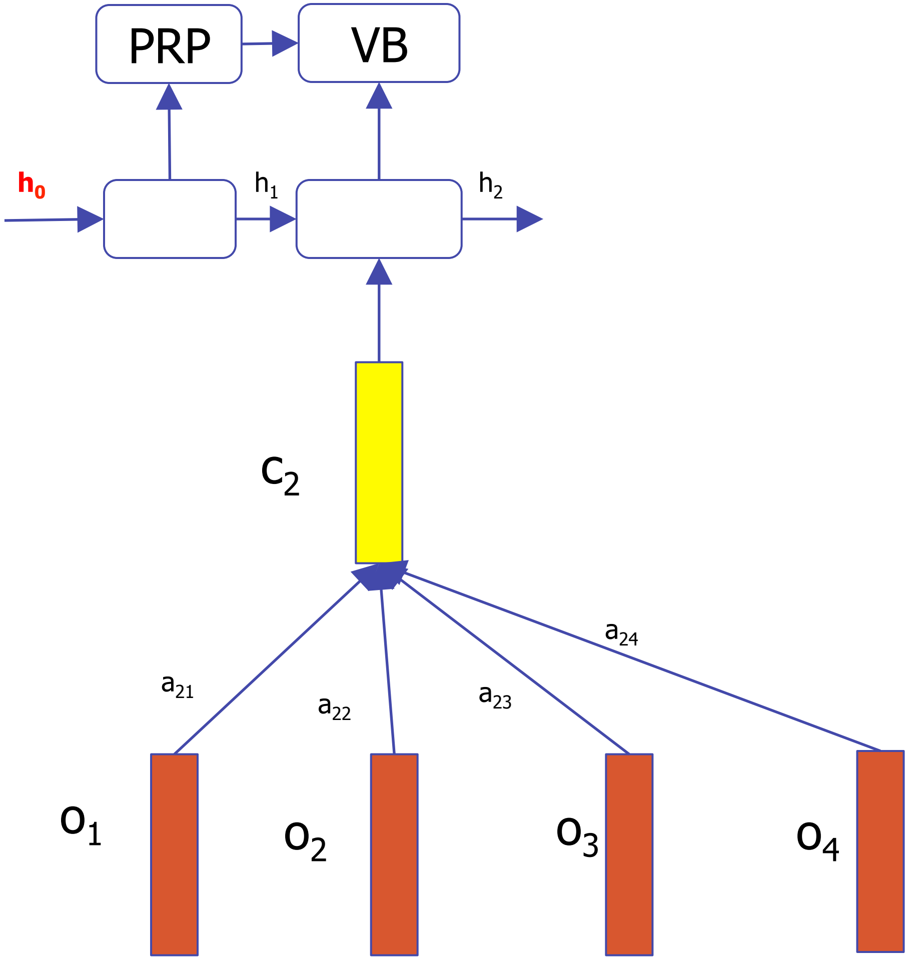
This averaged ***context vector*** is an input to the decoder

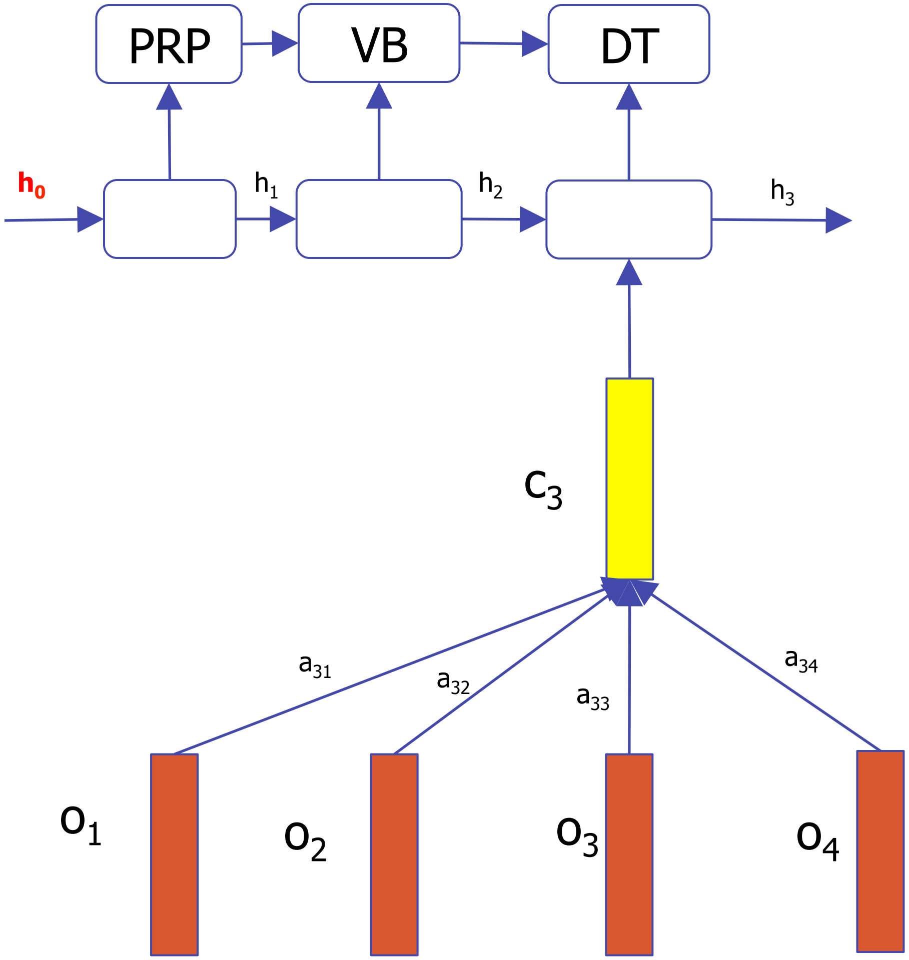
$$c_i = \sum_{j=1}^{n} a_{ij} o_j$$

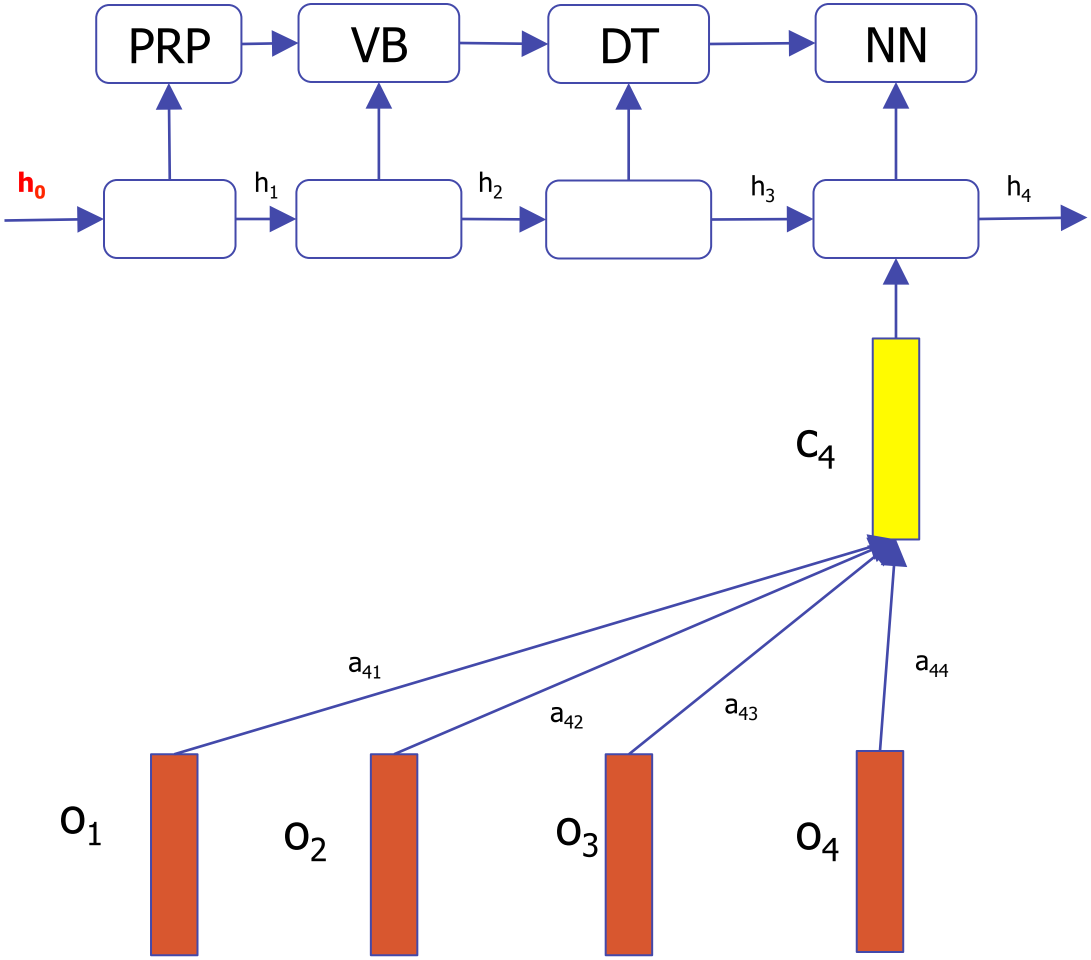
*For generation of $i^{th}$ output character:*
*$c_i$ : context vector*
*$a_{ij}$ : annotation weight for the $j^{th}$ annotation vector*
*$o_j$: $j^{th}$ annotation vector*

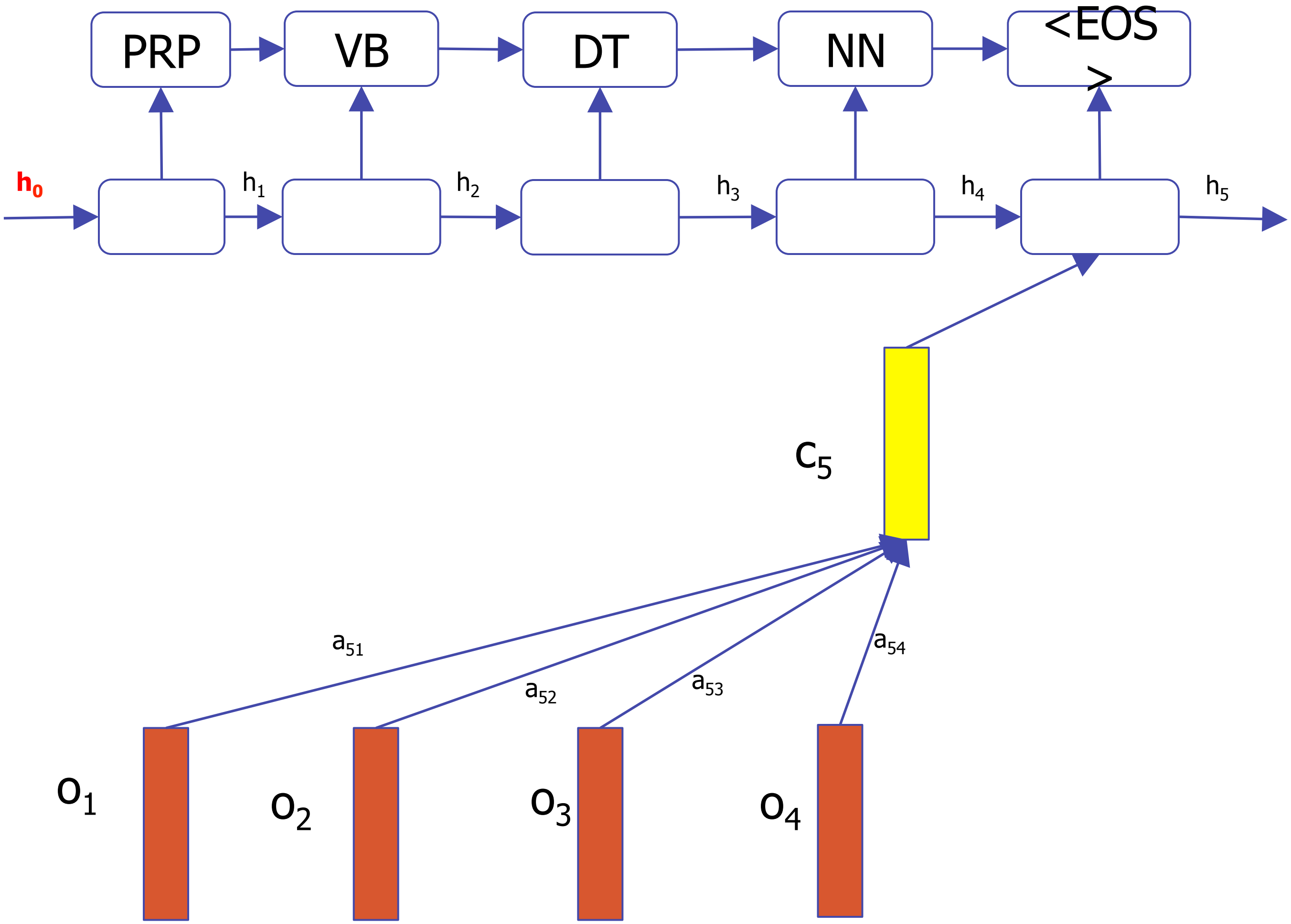


*Let's see an example of how the* ***attention mechanism*** *works*


PRP
VB
$h_0$
$h_1$
$h_2$
$c_2$
$a_{21}$
$a_{22}$
$a_{23}$
$a_{24}$
$o_1$
$o_2$
$o_3$
$o_4$


PRP
VB
DT
h0
h1
h2
h3
c3
a31
a32
a33
a34
o1
o2
o3
o4


PRP
VB
DT
NN
$h_0$
$h_1$
$h_2$
$h_3$
$h_4$
$c_4$
$a_{41}$
$a_{42}$
$a_{43}$
$a_{44}$
$o_1$
$o_2$
$o_3$
$o_4$


PRP
VB
DT
NN
<EOS>
$h_0$
$h_1$
$h_2$
$h_3$
$h_4$
$h_5$
$c_5$
$a_{51}$
$a_{52}$
$a_{53}$
$a_{54}$
$o_1$
$o_2$
$o_3$
$o_4$

*But we do not know the attention weights?*
*How do we find them?*

*Let the training data help you decide!!*

**Idea:** Pick the attention weights that maximize the POS tagging accuracy

*(more precisely, decrease training data loss)*

Have an attention function that predicts the attention weights:

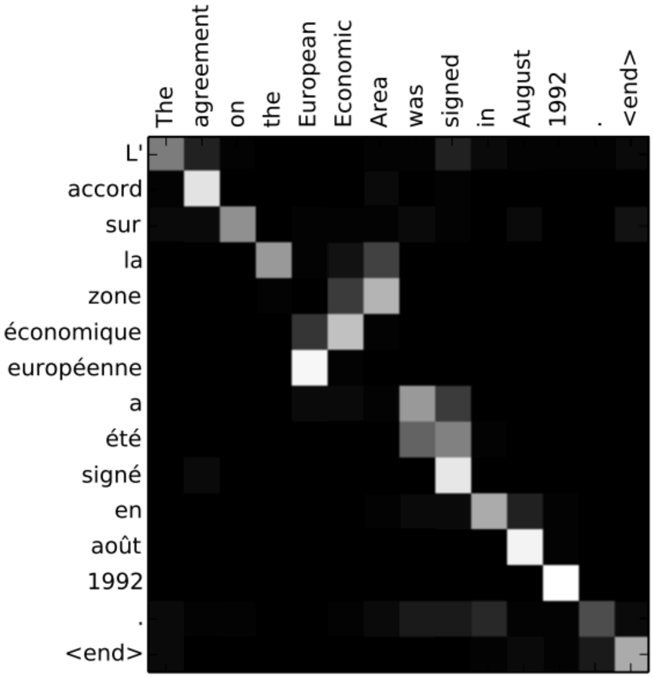
$a_{ij} = \mathbf{A}(o_j,h_i;\mathbf{o})$

**A** could be implemented as a feedforward network which is a component of the overall network

Then training the attention network with the rest of the network ensures that the attention weights are learnt to minimize the translation loss

## *OK, but do the attention weights actually show focus on certain parts?*

Here is an example of how attention weights represent a soft alignment for machine translati



## *Let's go back to the encoder. What type of encoder cell should we use there?*

- Basic RNN: models sequence history by maintaining state information
  - But, cannot model long range dependencies
- LSTM: can model history and is better at handling long range dependencies

The RNN units model only the sequence seen so far, cannot see the sequence ahead

- Can use a bidirectional RNN/LSTM
- This is just 2 LSTM encoders run from opposite ends of the sequence and resulting output vectors are composed

Both types of RNN units process the sequence sequentially, hence parallelism is limited

Alternatively, we can use a **CNN**

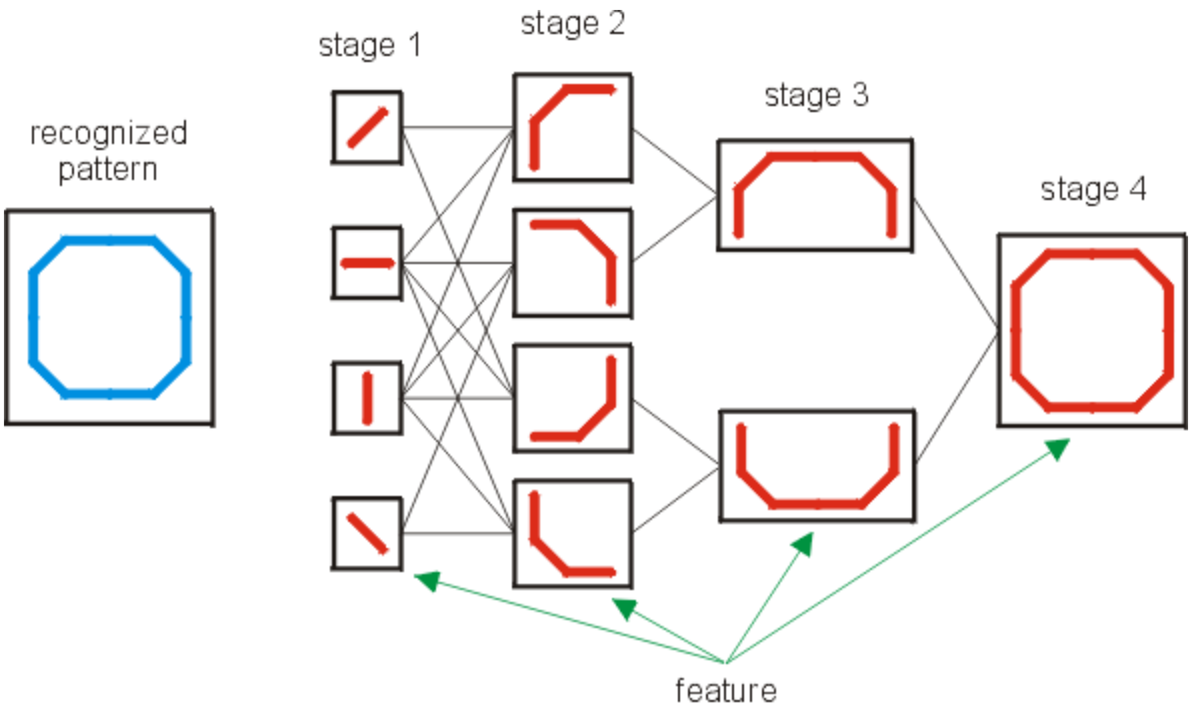
- Can operate on a sequence in parallel
- However, cannot model entire sequence history
- Model only a short local context. This may be sufficient for some applications or deep CNN layers can overcome the problem

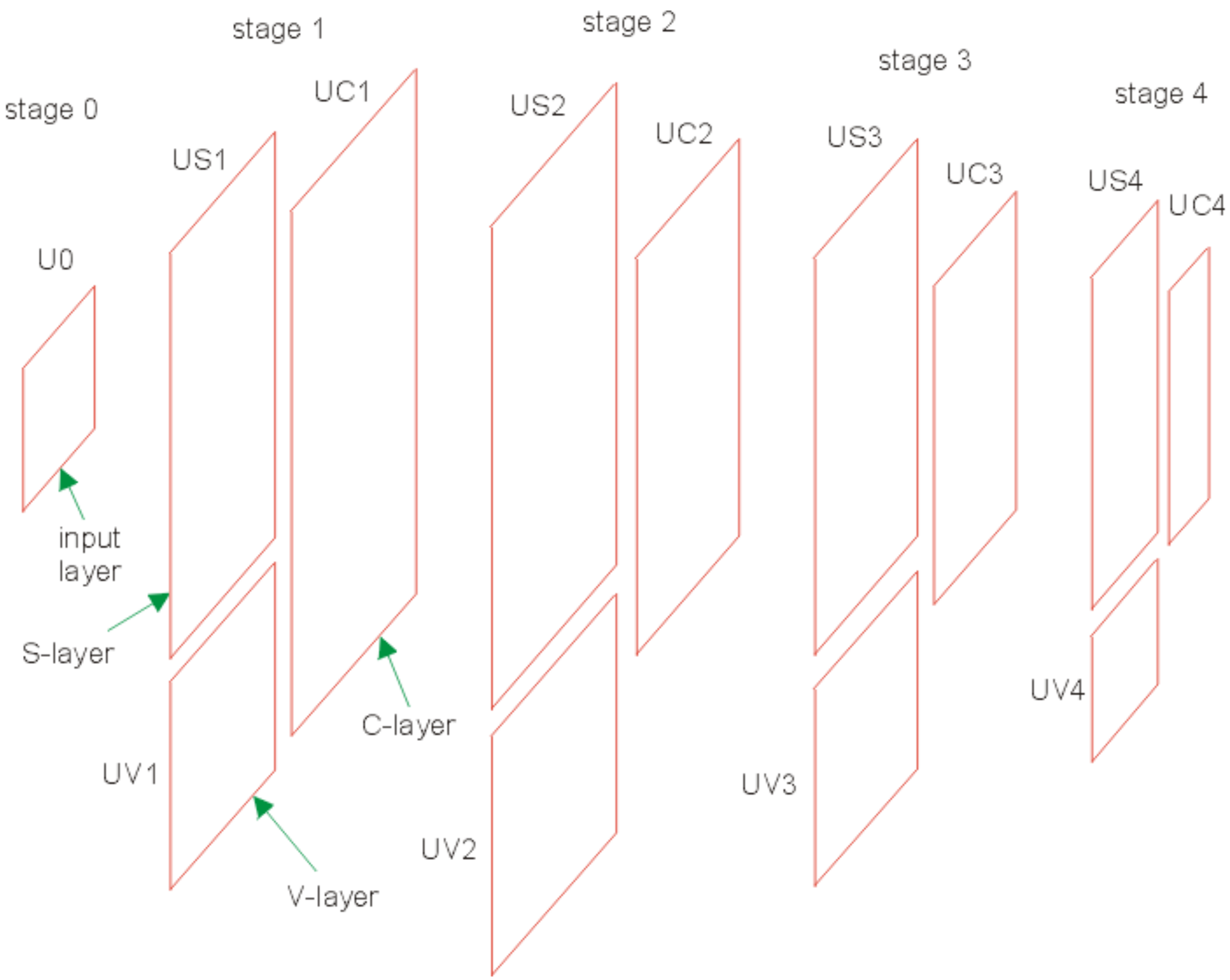
# Convolutional Neural Network (CNN)

# CNN= feedforward + recurrent!

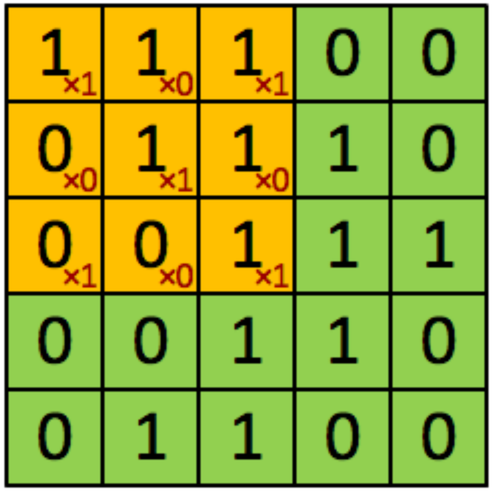
- Whatever we learnt so far in FF-BP is useful to understand CNN
- So also is the case with RNN (and LSTM)
- Input divided into regions and fed forward
- Window slides over the input: input changes, but ‘filter’ parameters remain same
- That is RNN

# Remember Neocognitron



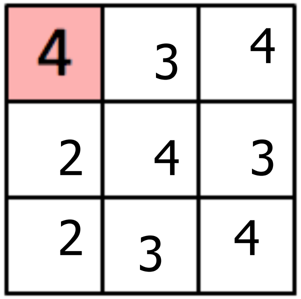

stage 0
stage 1
stage 2
stage 3
stage 4
U0
US1
UC1
US2
UC2
US3
UC3
US4
UC4
input layer
S-layer
C-layer
V-layer
UV1
UV2
UV3
UV4

# Convolution



| | | | | |
|---|---|---|---|---|
| $1_{\times1}$ | $1_{\times0}$ | $1_{\times1}$ | 0 | 0 |
| $0_{\times0}$ | $1_{\times1}$ | $1_{\times0}$ | 1 | 0 |
| $0_{\times1}$ | $0_{\times0}$ | $1_{\times1}$ | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |

Image



| | | |
|---|---|---|
| 4 | 3 | 4 |
| 2 | 4 | 3 |
| 2 | 3 | 4 |

Convolved Feature

- Matrix on the left represents an black and white image.

- Each entry corresponds to one pixel, 0 for black and 1 for white (typically it's between 0 and 255 for grayscale images).

- The sliding window is called a *kernel, filter,* or *feature detector.*

- Here we use a 3×3 filter, multiply its values element-wise with the original matrix, then sum them up.

- To get the full convolution we do this for each element by sliding the filter over the whole matrix.

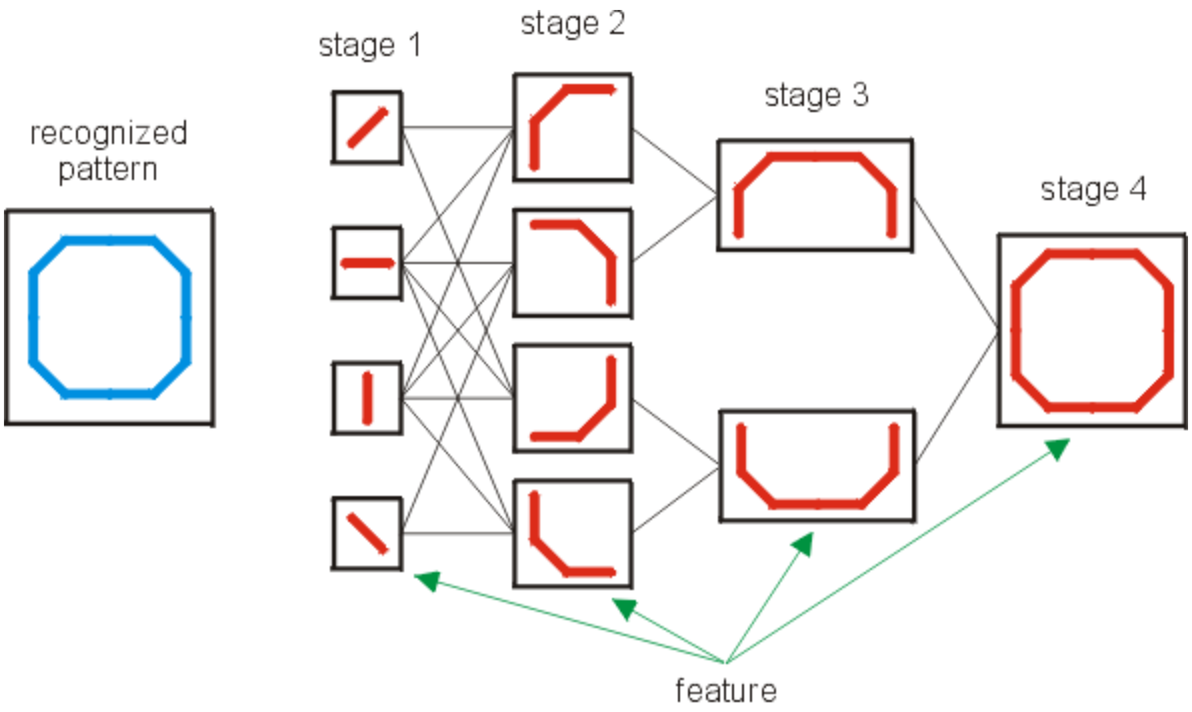
# CNN architecture

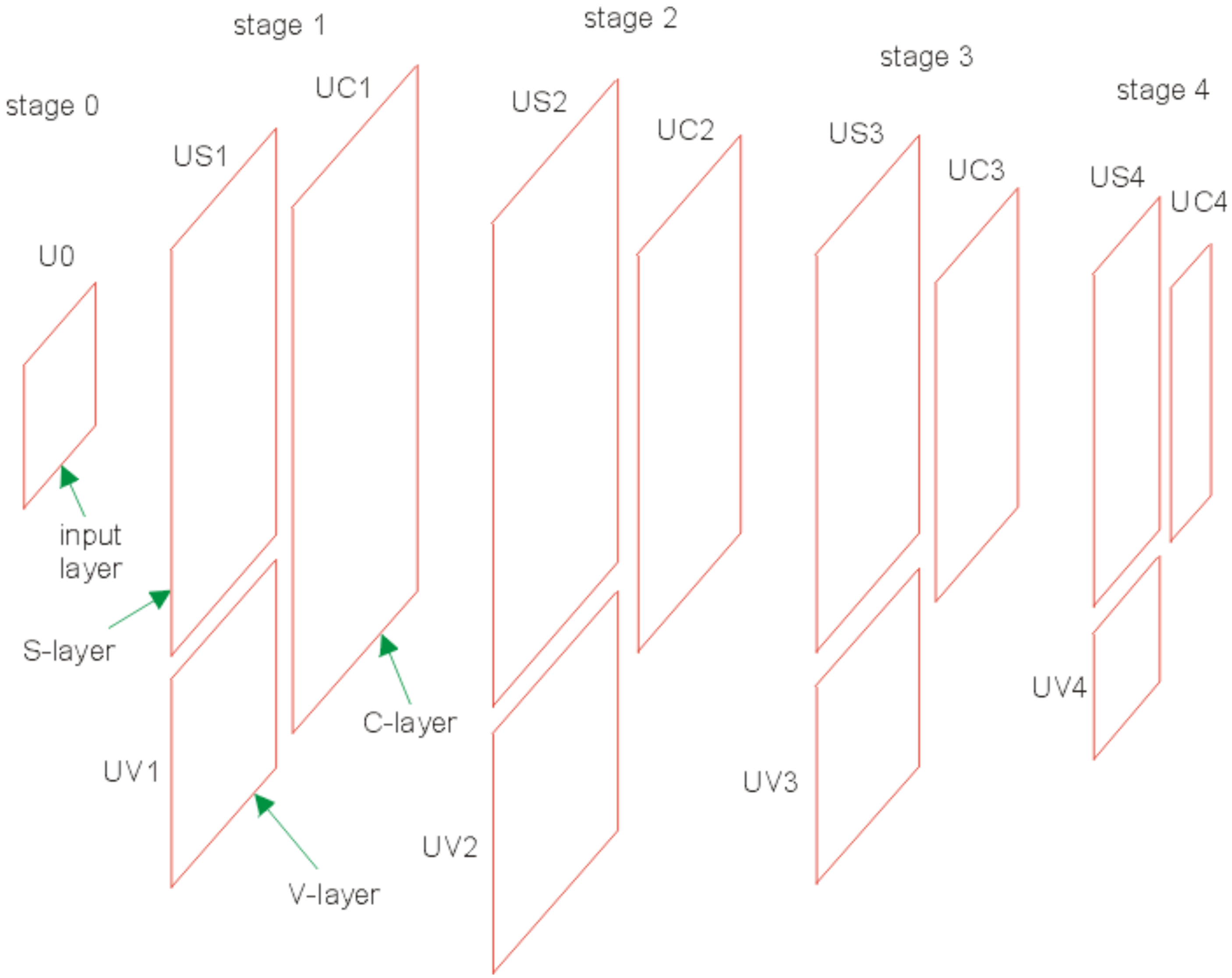
- Several layers of convolution with *tanh* or *ReLU* applied to the results
- In a traditional feedforward neural network we connect each input neuron to each output neuron in the next layer. That's also called a fully connected layer, or affine layer.
- In CNNs we use convolutions over the input layer to compute the output.
- This results in local connections, where each region of the input is connected to a neuron in the output

# Learning in CNN

- **Automatically learns the values of its filters**
- For example, in Image Classification learn to
  - detect edges from raw pixels in the first layer,
  - then use the edges to detect simple shapes in the second layer,
  - and then use these shapes to deter higher-level features, such as facial shapes in higher layers.
  - The last layer is then a classifier that uses these high-level features.

# Remember Neocognitron



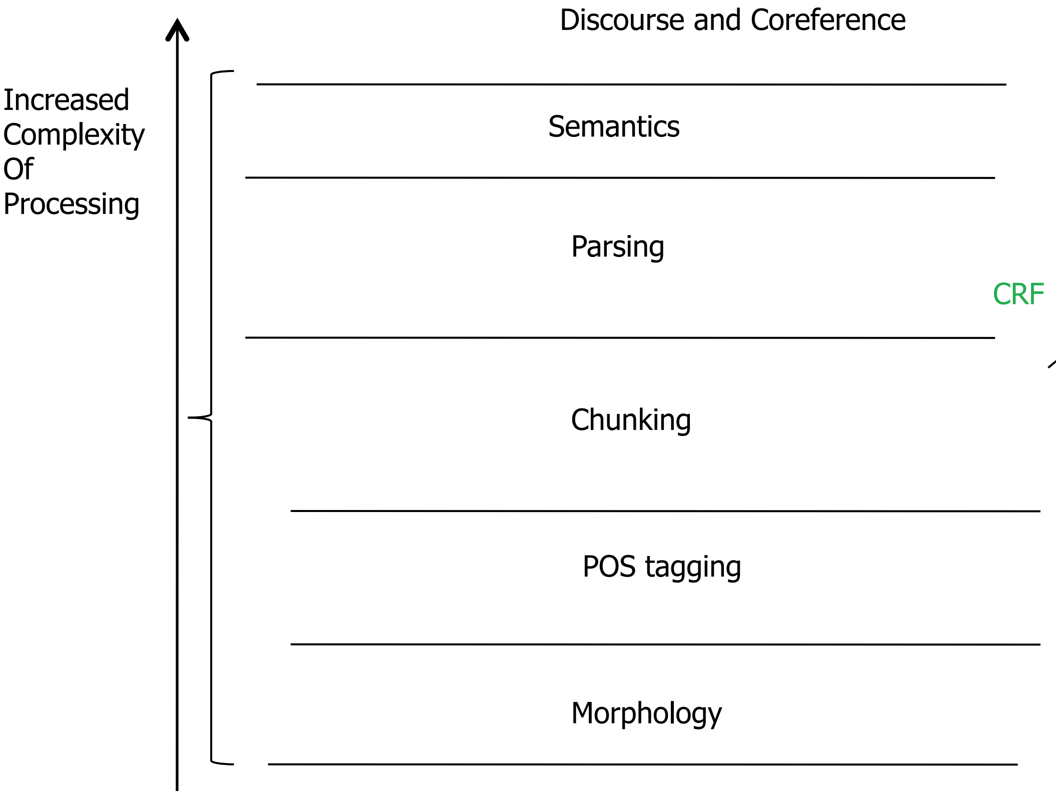

stage 1
stage 2
stage 3
stage 4
stage 0
UC1
US2
UC2
US3
UC3
US4
UC4
US1
U0
input
layer
S-layer
C-layer
UV1
V-layer
UV2
UV3
UV4

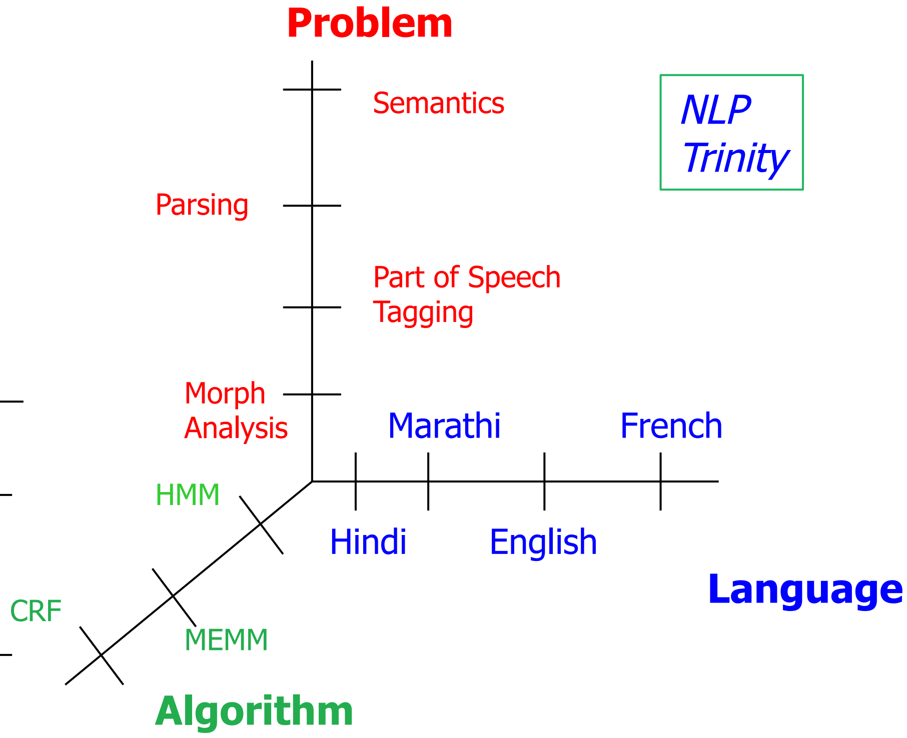
# What about NLP and CNN?

- Natural Match!

- NLP happens in layers

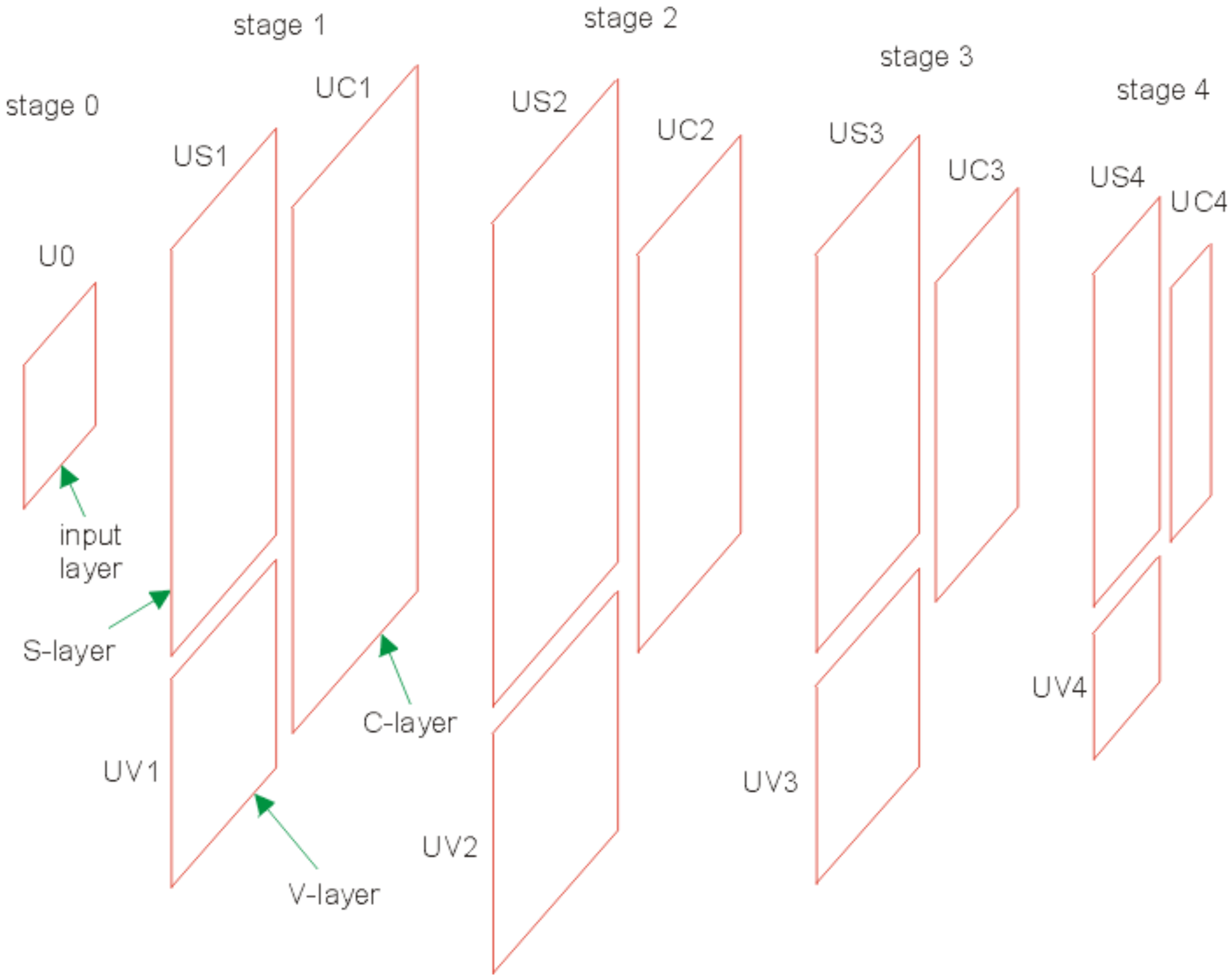
# NLP: multilayered, multidimensional

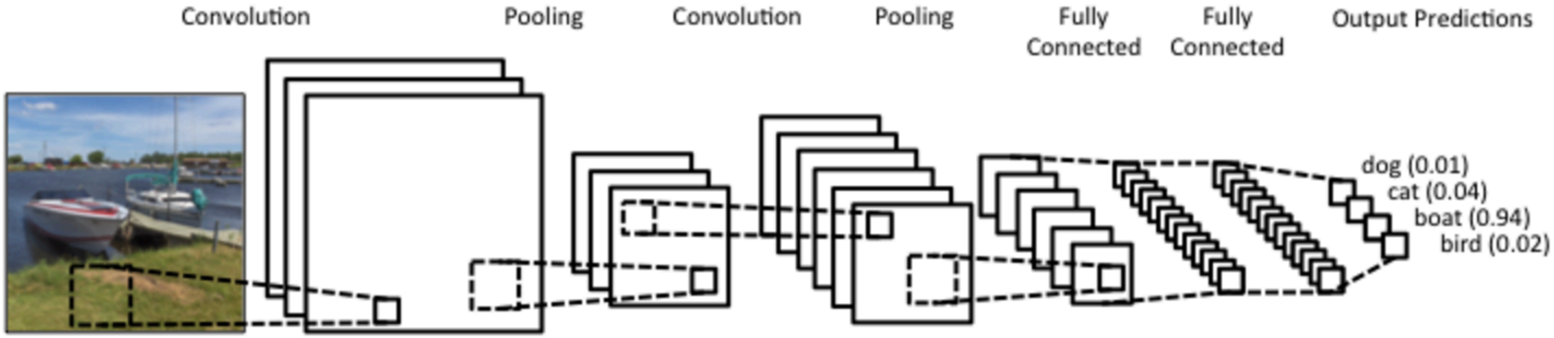




# NLP layers and CNN

- Morph layer →
- POS layer →
- Parse layer →
- Semantics layer


stage 0
stage 1
stage 2
stage 3
stage 4
U0
US1
UC1
US2
UC2
US3
UC3
US4
UC4
input layer
S-layer
C-layer
UV1
UV2
UV3
UV4
V-layer



http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/

# Pooling

- Gives invariance in translation, rotation and scaling

- Important for image recognition

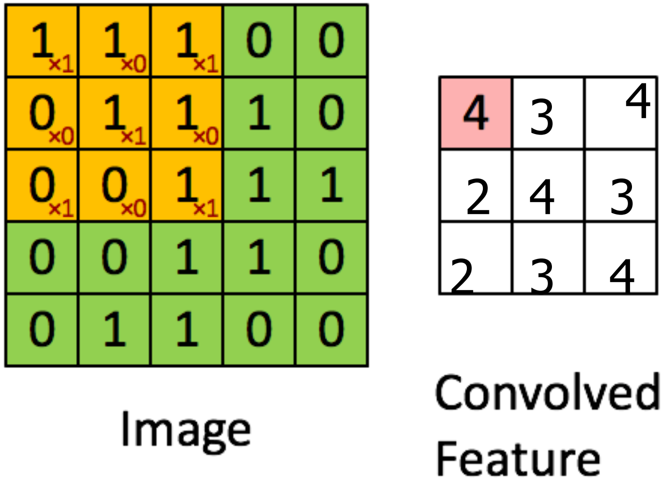
- Role in NLP?

# Input matrix for CNN: NLP

- “image” for NLP ←→ word vectors
- in the rows

- For a 10 word sentence using a 100-dimensional Embedding,

- we would have a 10×100 matrix as our input

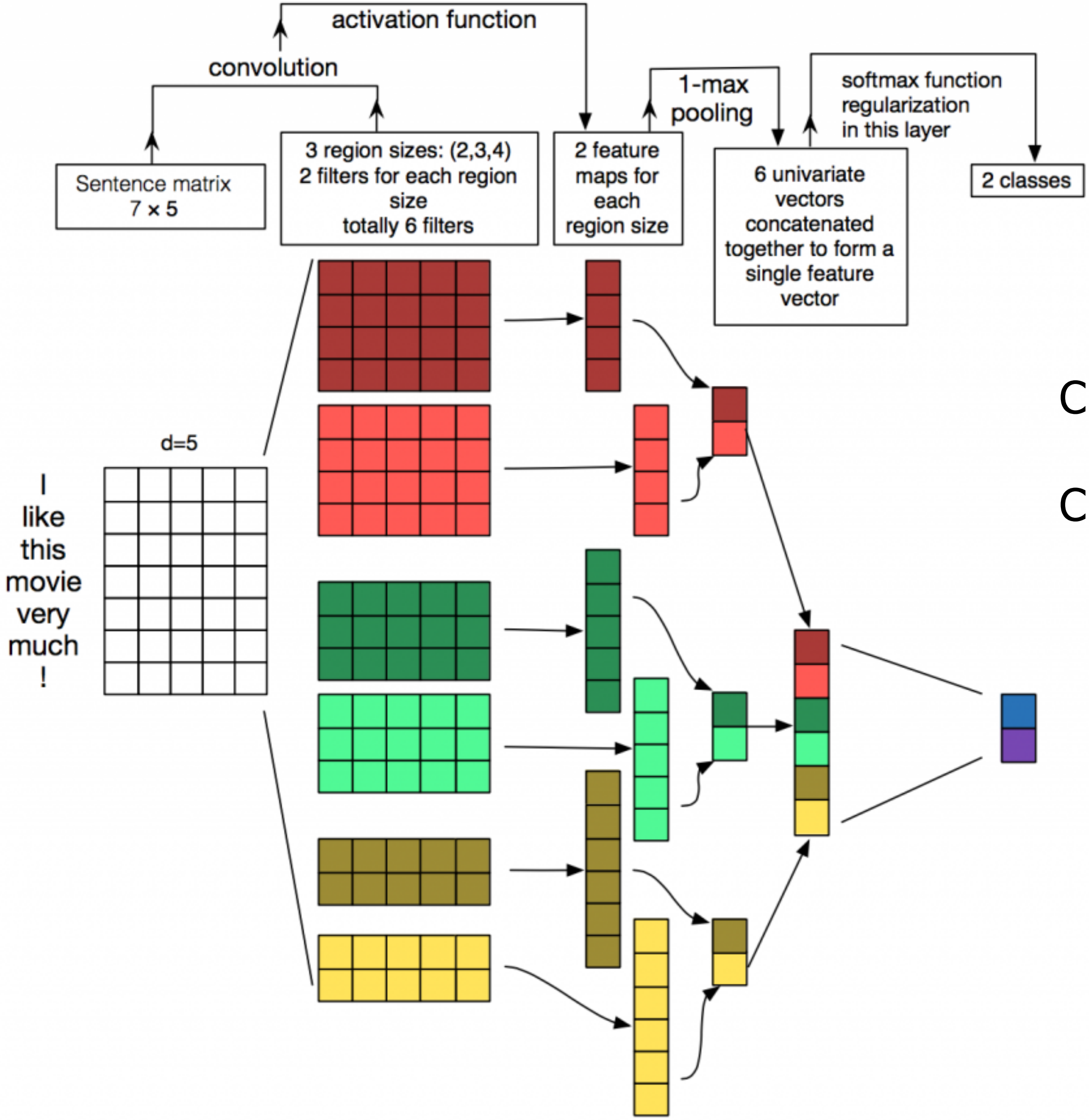


|  |  |  |  |  |
|---|---|---|---|---|
| $1_{\times1}$ | $1_{\times0}$ | $1_{\times1}$ | 0 | 0 |
| $0_{\times0}$ | $1_{\times1}$ | $1_{\times0}$ | 1 | 0 |
| $0_{\times1}$ | $0_{\times0}$ | $1_{\times1}$ | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |

Image

|  |  |  |
|---|---|---|
| 4 | 3 | 4 |
| 2 | 4 | 3 |
| 2 | 3 | 4 |

Convolved Feature


activation function
convolution
1-max pooling
softmax function regularization in this layer
Sentence matrix 7 × 5
3 region sizes: (2,3,4) 2 filters for each region size totally 6 filters
2 feature maps for each region size
6 univariate vectors concatenated together to form a single feature vector
2 classes
d=5
I
like
this
movie
very
much
!

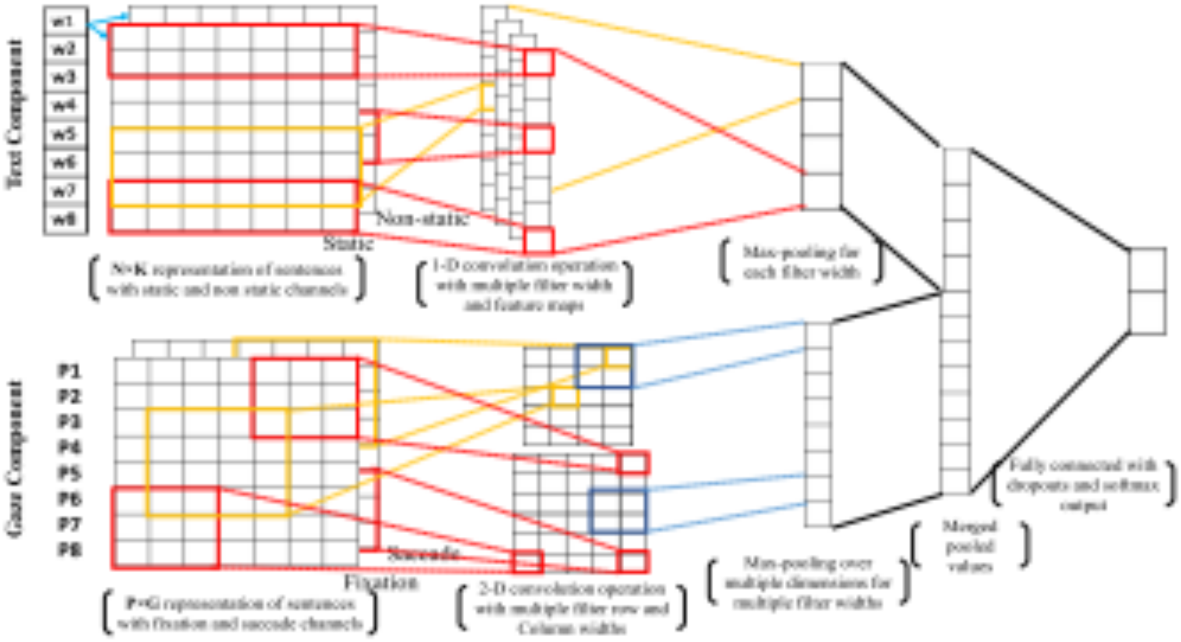
Credit: Denny Britz

CNN for NLP

# CNN Hyper parameters

- Narrow width vs. wide width
- Stride size
- Pooling layers
- Channels

Abhijit Mishra, Kuntal Dey and Pushpak Bhattacharyya, *Learning Cognitive Features from Gaze Data for Sentiment and Sarcasm Classification Using Convolutional Neural Network*, **ACL 2017**, Vancouver, Canada, July 30-August 4, 2017.


Text Component
w1
w2
w3
w4
w5
w6
w7
w8
Non-static
Static
N×K representation of sentences with static and non static channels
1-D convolution operation with multiple filter width and feature maps
Max-pooling for each filter width
Gaze Component
P1
P2
P3
P4
P5
P6
P7
P8
Saccade
Fixation
P×G representation of sentences with fixation and saccade channels
2-D convolution operation with multiple filter row and Column widths
Max-pooling over multiple dimensions for multiple filter widths
Merged pooled values
Fully connected with dropouts and softmax output

# Learning Cognitive Features from Gaze Data for Sentiment and Sarcasm Classification

- In complex classification tasks like sentiment analysis and sarcasm detection, even the extraction and choice of features should be delegated to the learning system
- CNN learns features from both gaze and text and uses them to classify the input text