ML: Lecture 6

Recurrent Neural Networks

Panagiotis Papapetrou, PhD
Professor, Stockholm University

Syllabus

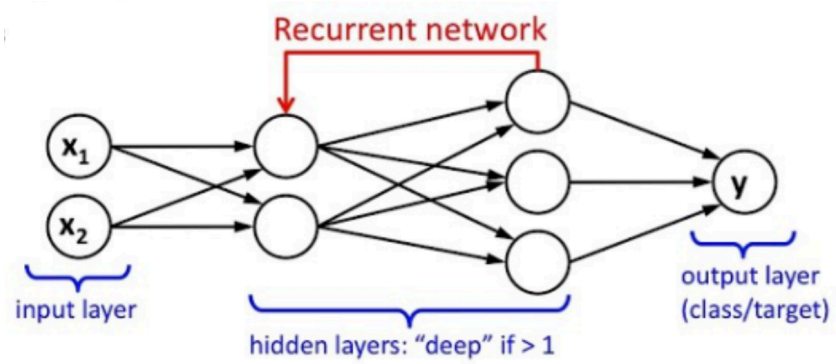
Jan 16	Introduction to machine learning
Jan 18	Regression analysis
Jan 19	Laboratory session 1: numpy and linear regression
Jan 23	Ensemble learning
Jan 25	Deep learning I: Training neural networks
Jan 26	Laboratory session 2: ML pipelines, ensemble learning
Jan 30	Deep learning II: Convolutional neural networks
Feb 1	Laboratory session 3: training NNs and tensorflow
Feb 6	**Deep learning III: Recurrent neural networks**
Feb 8	Laboratory session 4: CNNs and RNs
Feb 13	Deep learning IV: Autoencoders, transformers, and attention
Feb 20	Time series classification

Today

- What are RNNs?
- What are their main principles?
- How to train RNNs?
- Gated RNNs

Recurrent Neural Networks (RNNs) (Rumelhart et al. 1986)

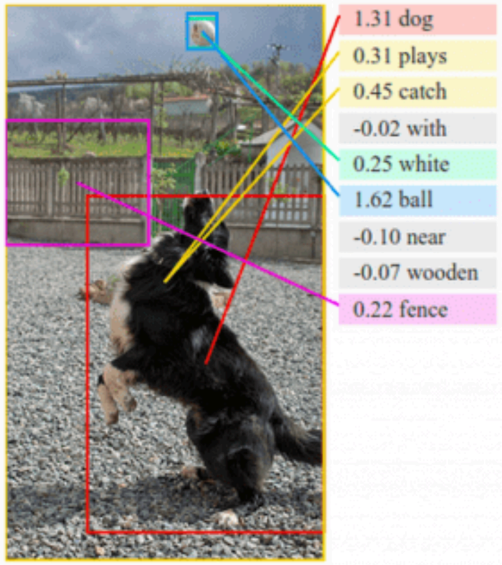
- A family of neural networks for processing sequential data
- Specialized for processing a sequence of values $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(\tau)}$
- Just like CNNs can be applied to images of large width and height, RNNs can scale to much longer sequences and can process sequences of variable length
- **Fully connected feed-forward network:**
  - separate parameters for each input feature
  - learns all the rules of the input separately

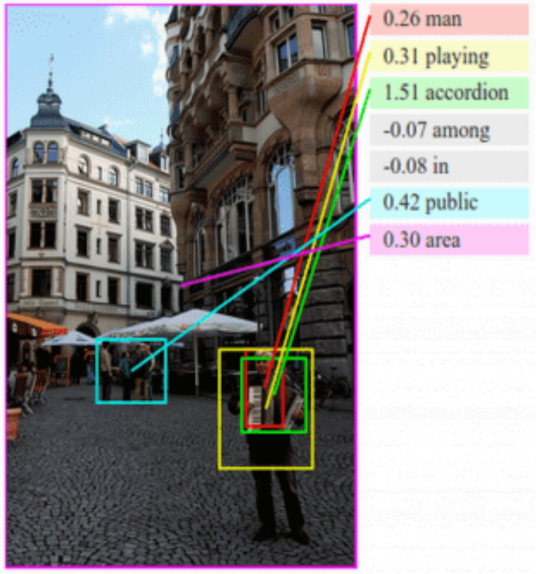


- **RNN:** shares the same weights across several time steps

Applications of RNNs

- Language modelling
- Machine translation
- Speech recognition
- Generating image captions
- Video tagging
- Text summarization
- Learning from medical data





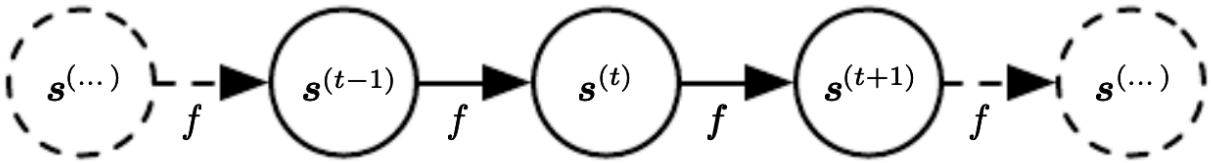
Dynamical Systems

- The classical form:
  - **s(t):** the state of the system at time t

$$s^{(t)} = f(s^{(t-1)}; \theta)$$

  - the definition of **s** at time t refers back to its definition at time t-1
  - for a finite number of steps **τ** the graph can be unfolded by applying function *f* **τ-1** times
  - e.g., for **τ=3**: $s^{(3)} = f(s^{(2)}; \theta)$

$$= f(f(s^{(1)}; \theta); \theta)$$



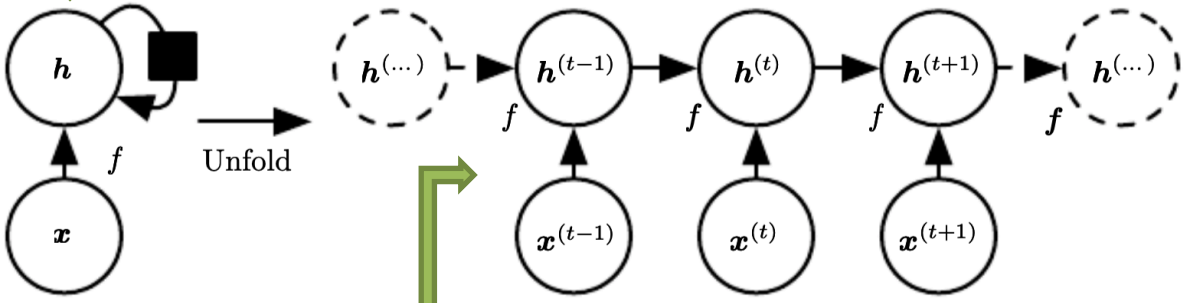
Dynamical Systems

- Consider a dynamical system that is driven by an exogenous signal $\mathbf{x}^{(t)}$

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$

- Any function involving recurrence can be considered an RNN
- Intermediate (i.e., hidden) states are defined by function ***h***:

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$



What is left: addition of extra architectural features such as output layers that read information out of state ***h*** to make predictions

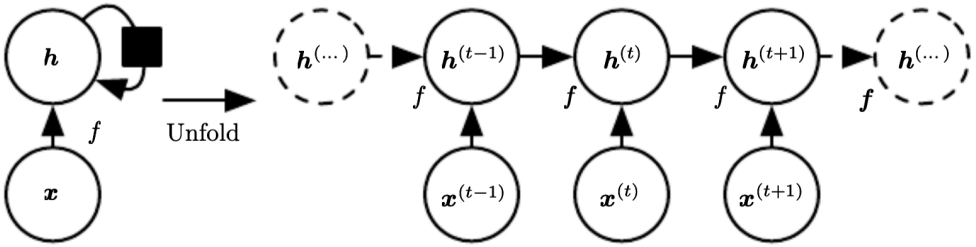
$$\boldsymbol{h}^{(t)} = g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$$

The basic RNN task

- Basic RNN **task**: predict the future from the past
- Map the **past** into a **fixed-length vector** $h^{(t)}$

$$(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}) \longrightarrow \boldsymbol{h}^{(t)}$$

- How much of the past should one store/model in $h^{(t)}$?
- Recurrent vs unfolded version?

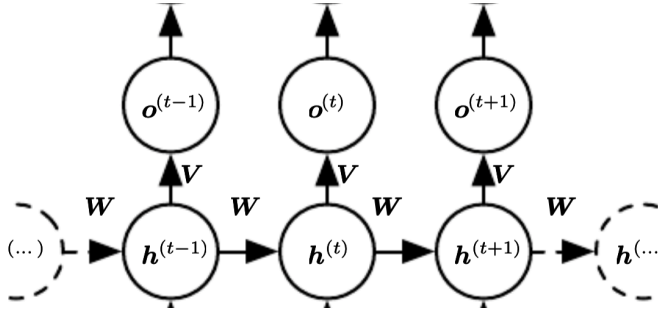


$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$

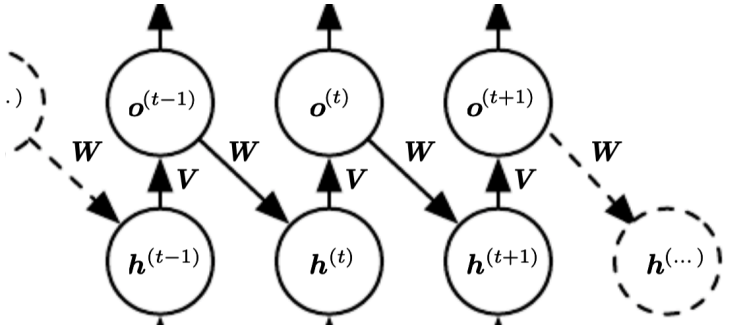
$$\boldsymbol{h}^{(t)} = g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$$

Three common Vanilla RNNs

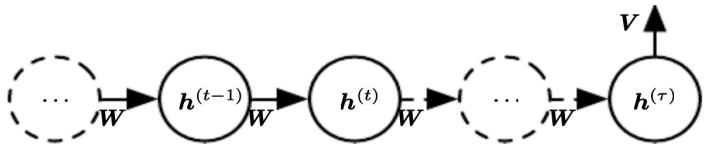
- **Version I:**
  - output at each time step
  - recurrent connections between hidden units



- **Version II:**
  - output at each time step
  - recurrent connections only from the output at one time step to the hidden units at the next

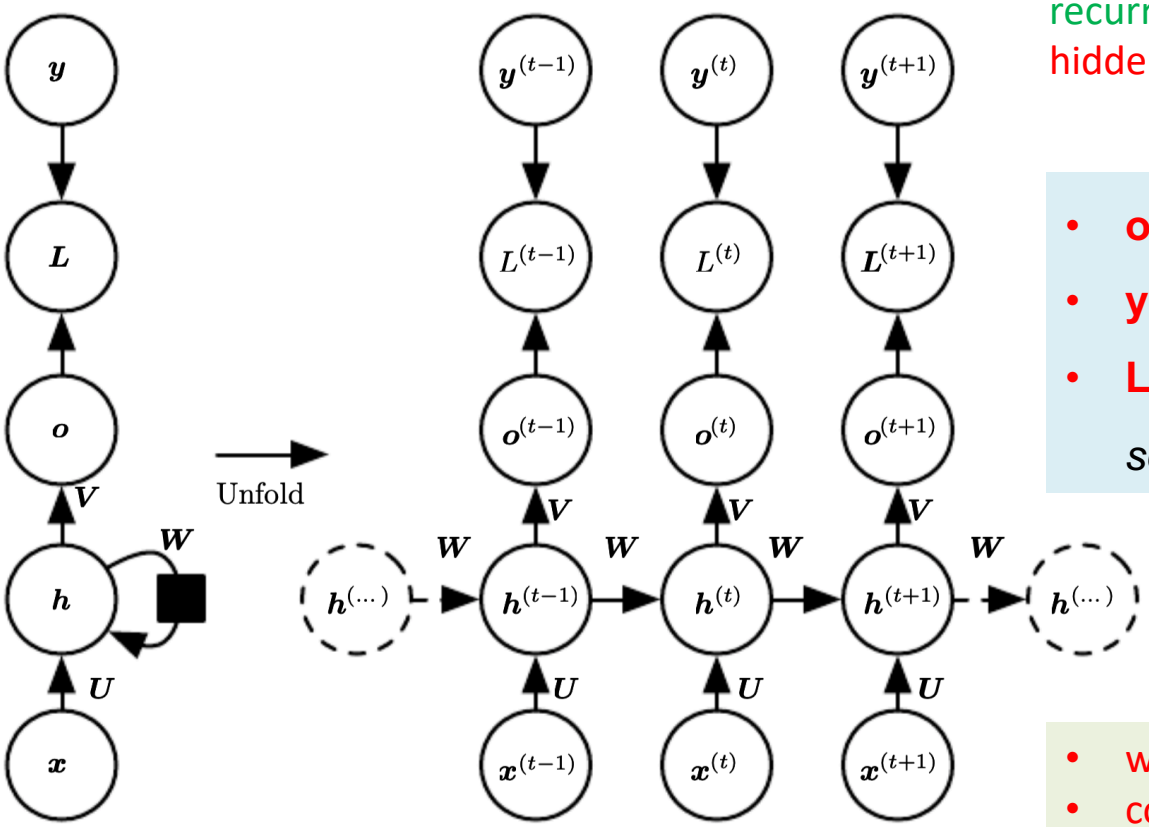


- **Version III:**
  - recurrent connections between hidden units
  - read an entire sequence and produce a single output



The Vanilla RNN – ver 1

output at each time step
recurrent connections between hidden units

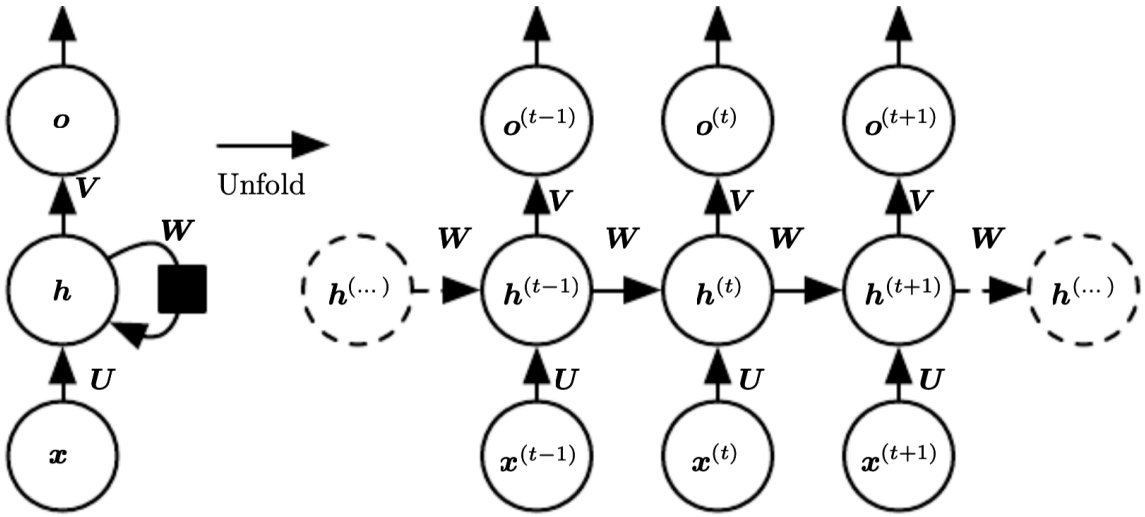


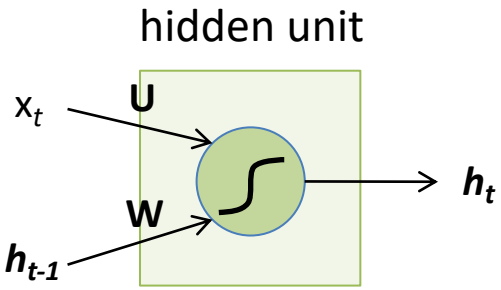
- **o:** output values
- **y:** target values
- **L:** comparison between *softmax(**o**)* and target

- **U:** input-to-hidden connections
- **W:** hidden-to-hidden recurrent connections
- **V:** hidden-to-output connections

- weights are shared over time
- copies of the RNN cell are made over time (unfolding), with different inputs at different time steps

The Vanilla RNN – ver 1





For each time step $t = 1$ to $t = \tau$

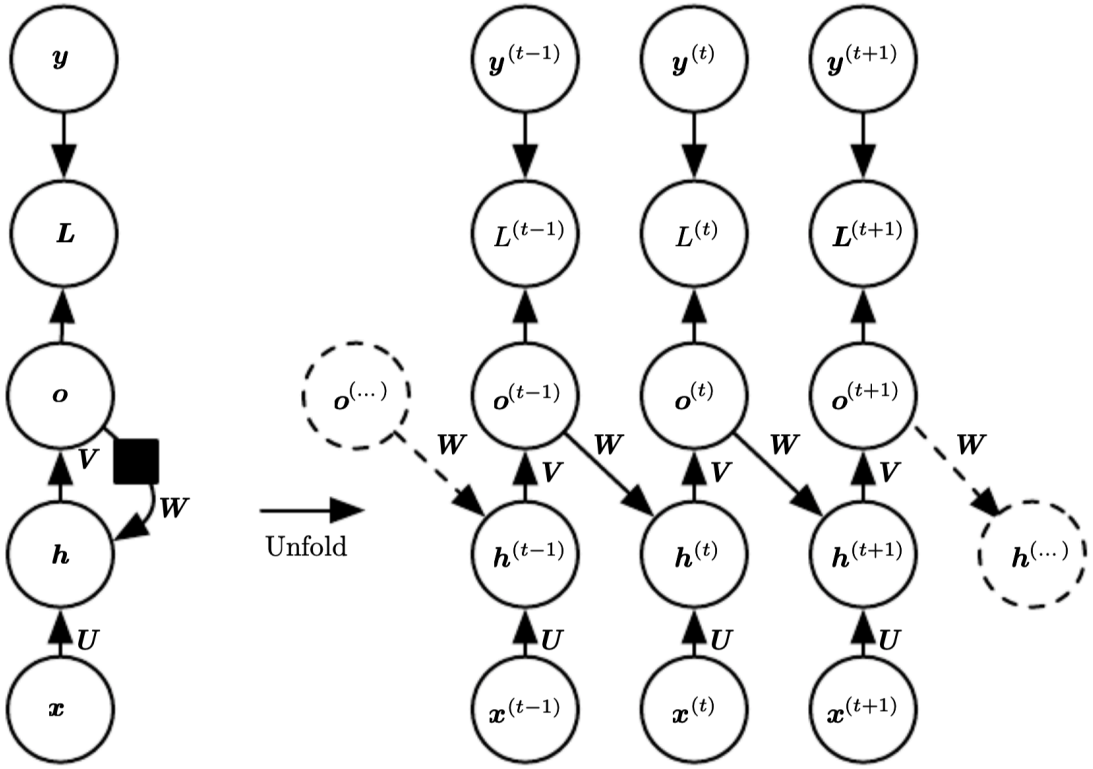
$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}, \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}), \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)}, \\
\hat{\boldsymbol{y}}^{(t)} &= \mathrm{softmax}(\boldsymbol{o}^{(t)}),
\end{aligned}
$$

bias vectors: *b* and ***c***

$$
L\left(\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(\tau)}\}, \{\boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(\tau)}\}\right) = \sum_t L^{(t)}
$$

***L*$^{(t)}$**: cross entropy loss of the predictions given {x$^{(1)}$, . . . , x$^{(t)}$}

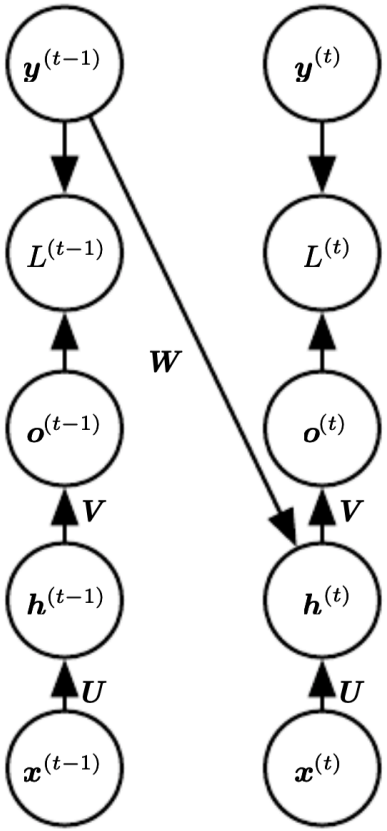
The Vanilla RNN – ver 2



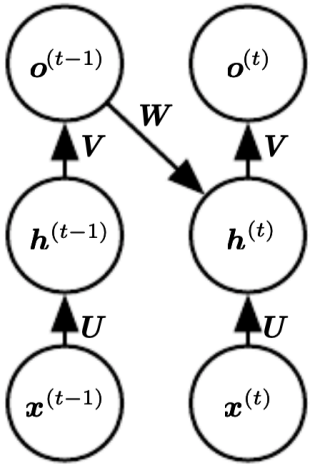
output at each time step
recurrent connections only from the output at one time step to the hidden units at the next

- no direct connections from ***h*** going forward
- it is trained on **output *o*** and this is the only information it is allowed to send to the future
- **easier to train (can train each hidden layer in parallel)**
- no need to compute the output for previous time step first…**why?**

Teacher Forcing



Train time

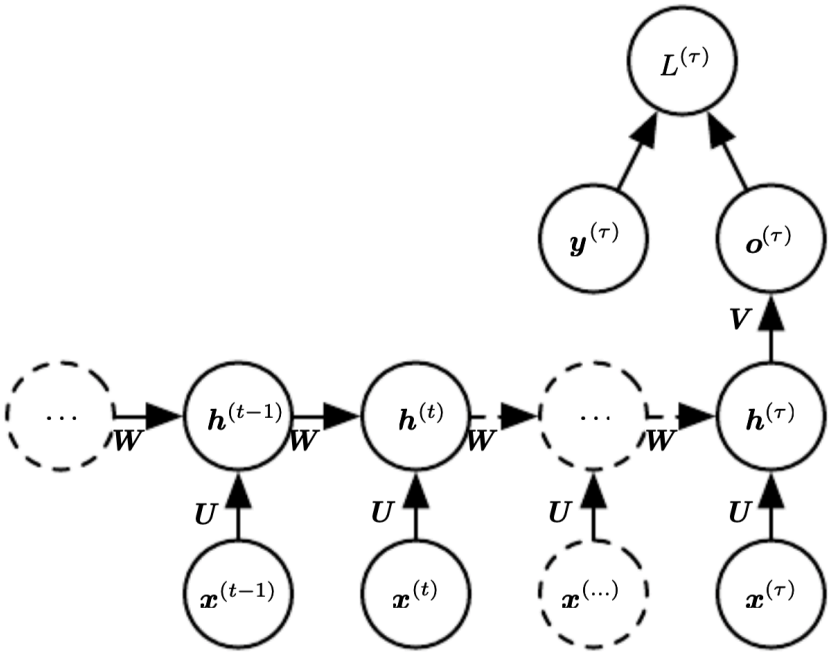


Test time

Applicable to RNNs that have connections from their output to their hidden states at the next time step

At training time, we feed the correct output $\mathbf{y^{(t)}}$ drawn from the training set as input to $\mathbf{h^{(t+1)}}$

The Vanilla RNN – ver 3

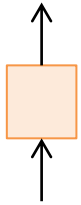


recurrent connections between hidden units
read an entire sequence and produce a single output

Summarize a sequence and produce a fixed-size representation, e.g., as input for further processing

Input – Output Scenarios

Single - Single



Feed-forward Network

Single - Multiple

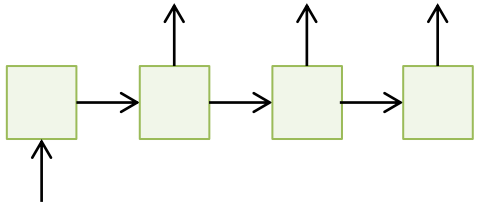
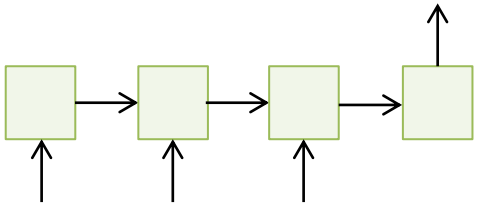


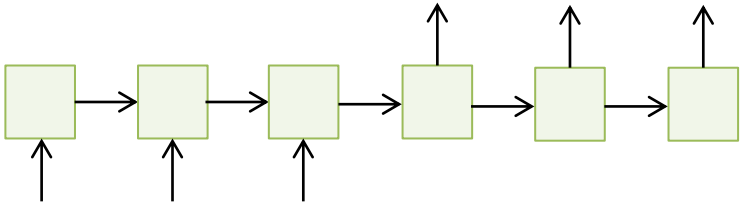
Image Captioning
(image only)

Multiple - Single



Sentiment Classification

Multiple - Multiple



Translation

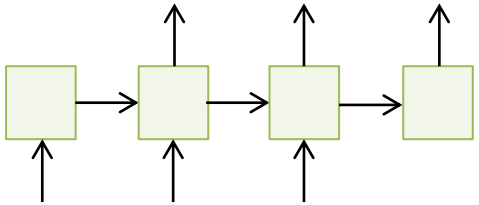
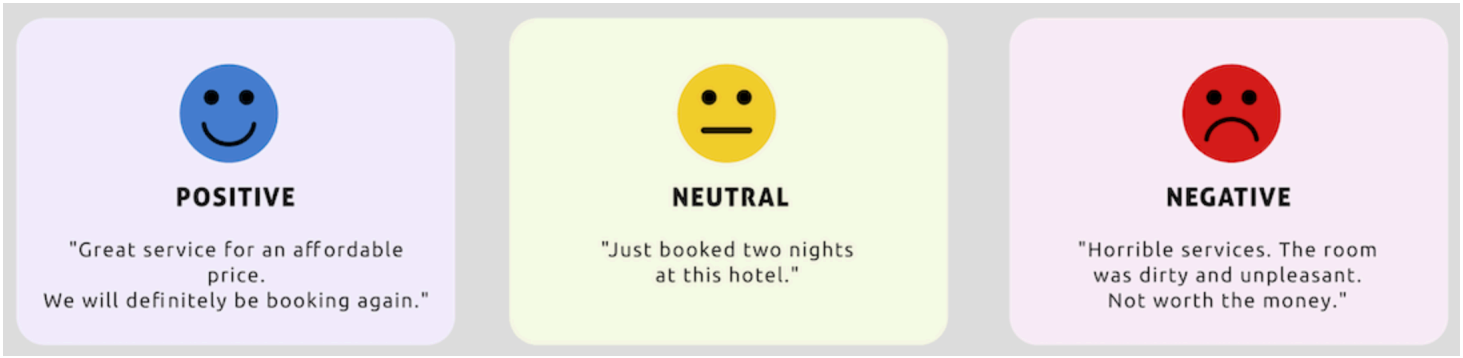
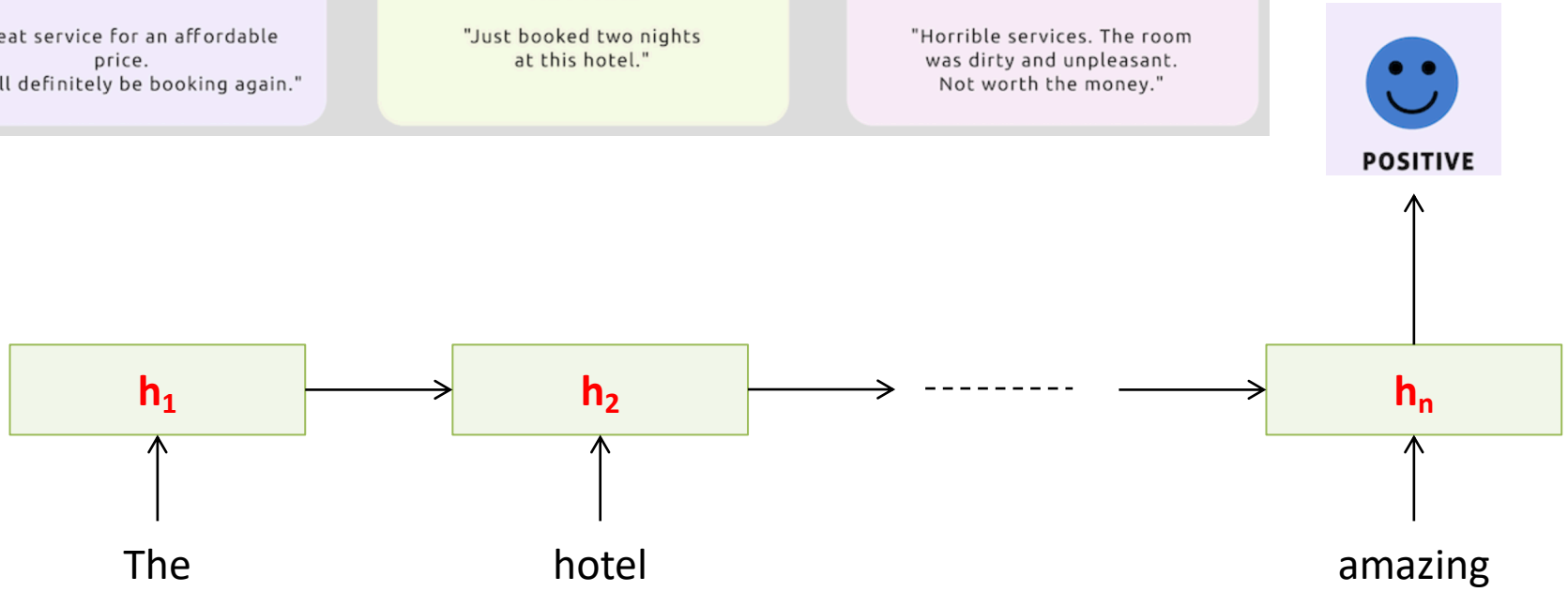


Image Captioning
(previous word too)

Sentiment Classification





Sentiment Classification

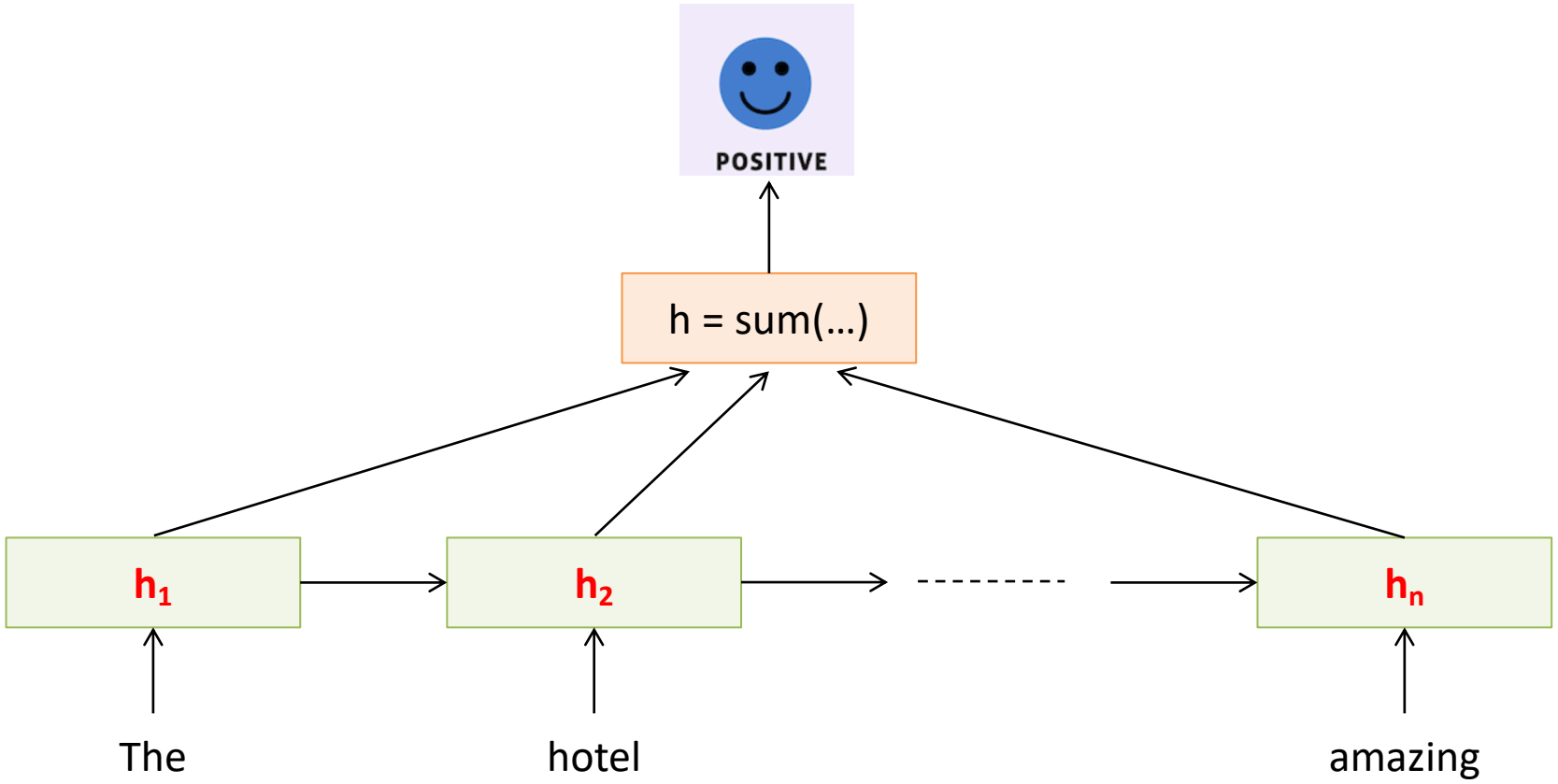


Image Captioning

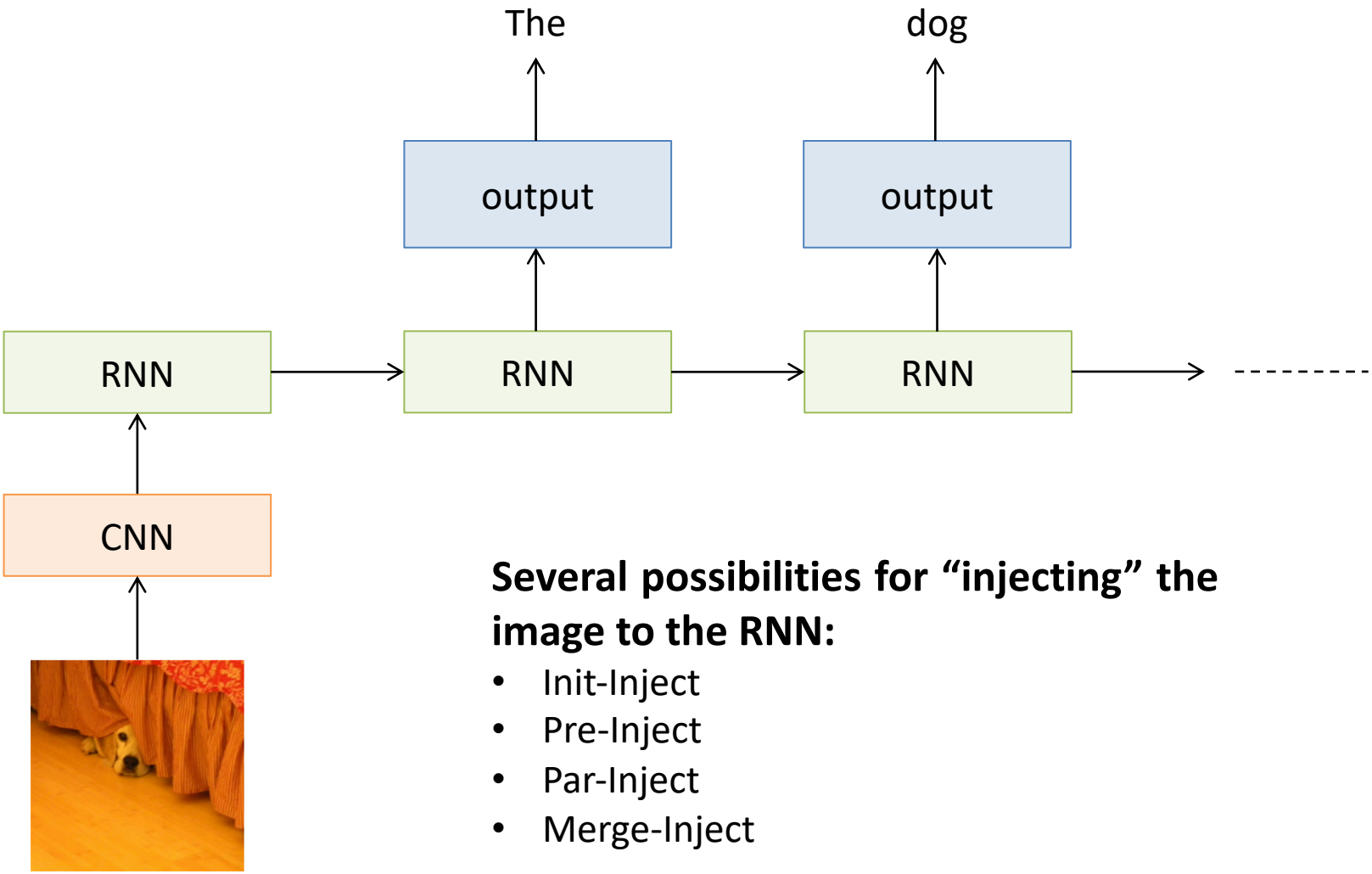
- Given an image, produce a sentence describing its contents

- **Input:** image feature vector (from a CNN)
- **Output:** multiple words (e.g., one sentence)

The dog is hiding

* https://arxiv.org/pdf/1703.09137.pdf

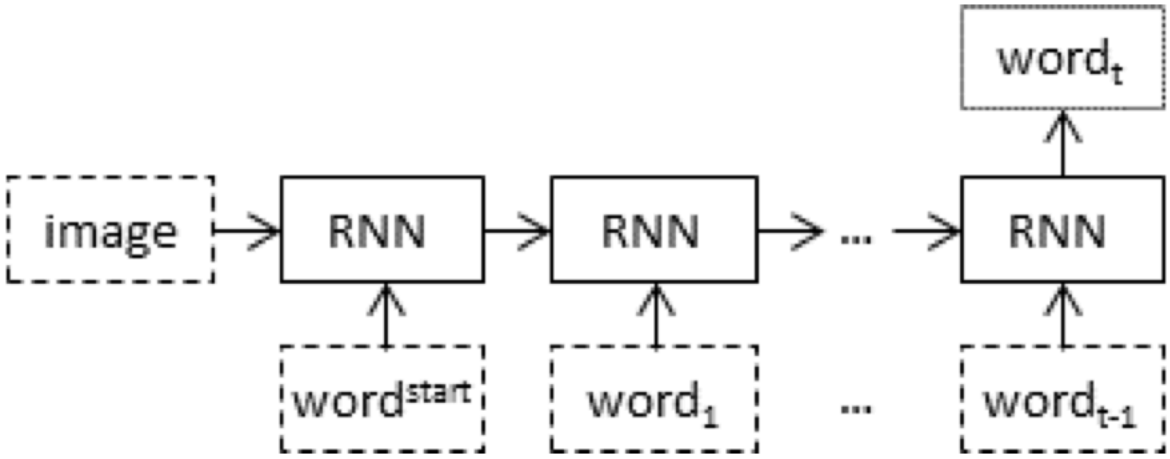
Image Captioning



Several possibilities for "injecting" the image to the RNN:

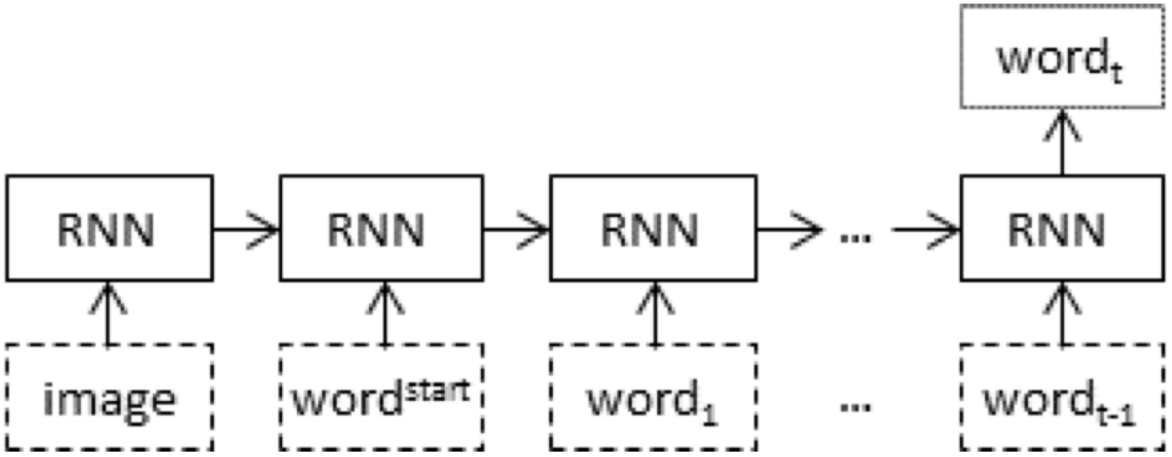
- Init-Inject
- Pre-Inject
- Par-Inject
- Merge-Inject

Init-Inject



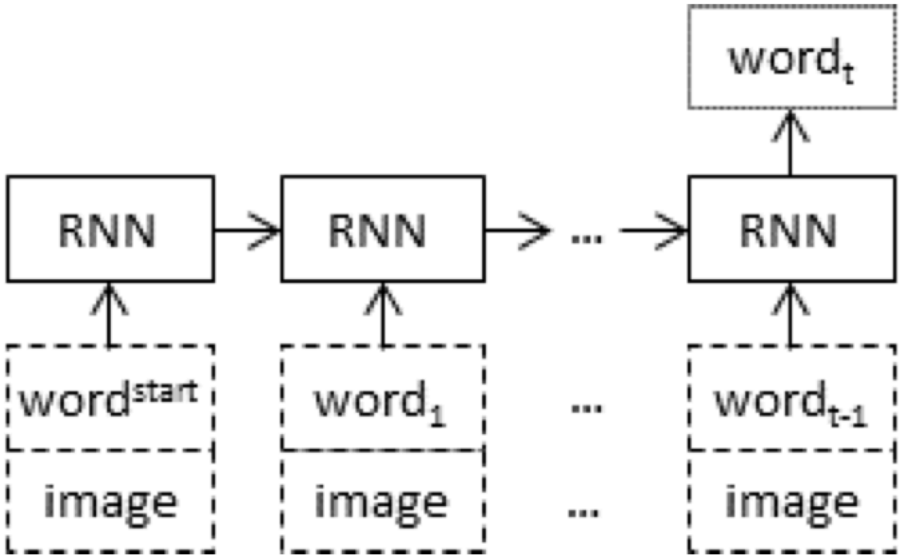
- The RNN's initial hidden state vector is set to be the image vector (or a vector derived from the image vector)
- It requires the image vector to have the same size as the RNN hidden state vector
- This is an early binding architecture and allows the image representation to be modified by the RNN

Pre-Inject



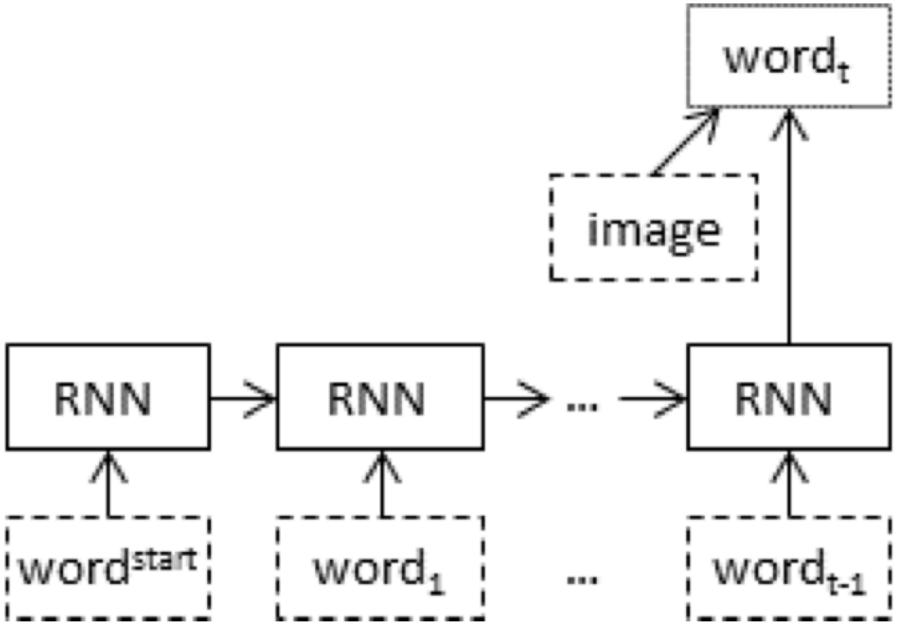
- The image vector is treated as a first word in the prefix
- This requires the image vector to have the same size as the word vectors
- This too is an early binding architecture and allows the image representation to be modified by the RNN

Par-Inject



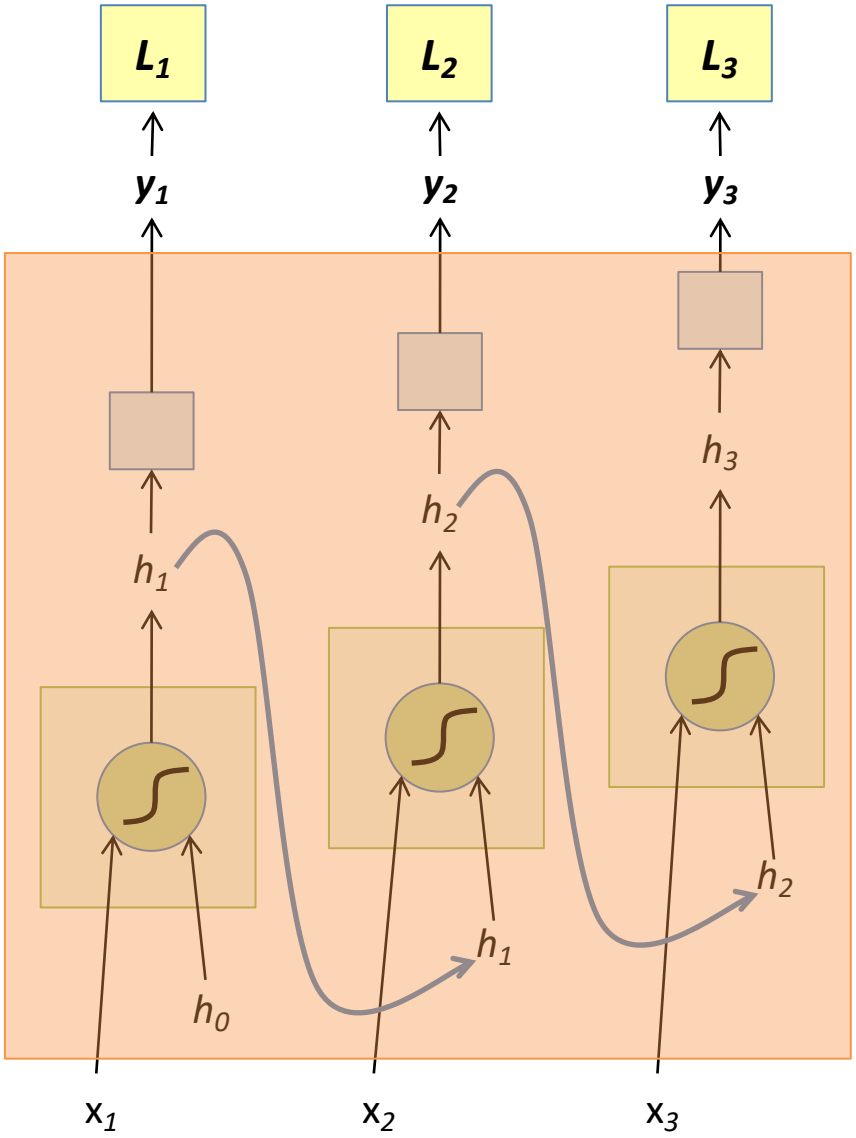
- The image vector is input to the RNN together with the word vectors: (a) two separate inputs or (b) the word vectors are combined with the image vector into a single input
- The image vector does not need to be exactly the same for each word nor does it need to be included with every word
- Allows for modification in the image representation, but it is **harder** for the RNN to do so if the same image is fed to the RNN at every time step due with the original image

Merge-Inject



- The RNN is not exposed to the image vector at any point
- Instead, the image is introduced into the language model after the prefix has been encoded by the RNN
- This is a late binding architecture and it does not modify the image representation with every time step

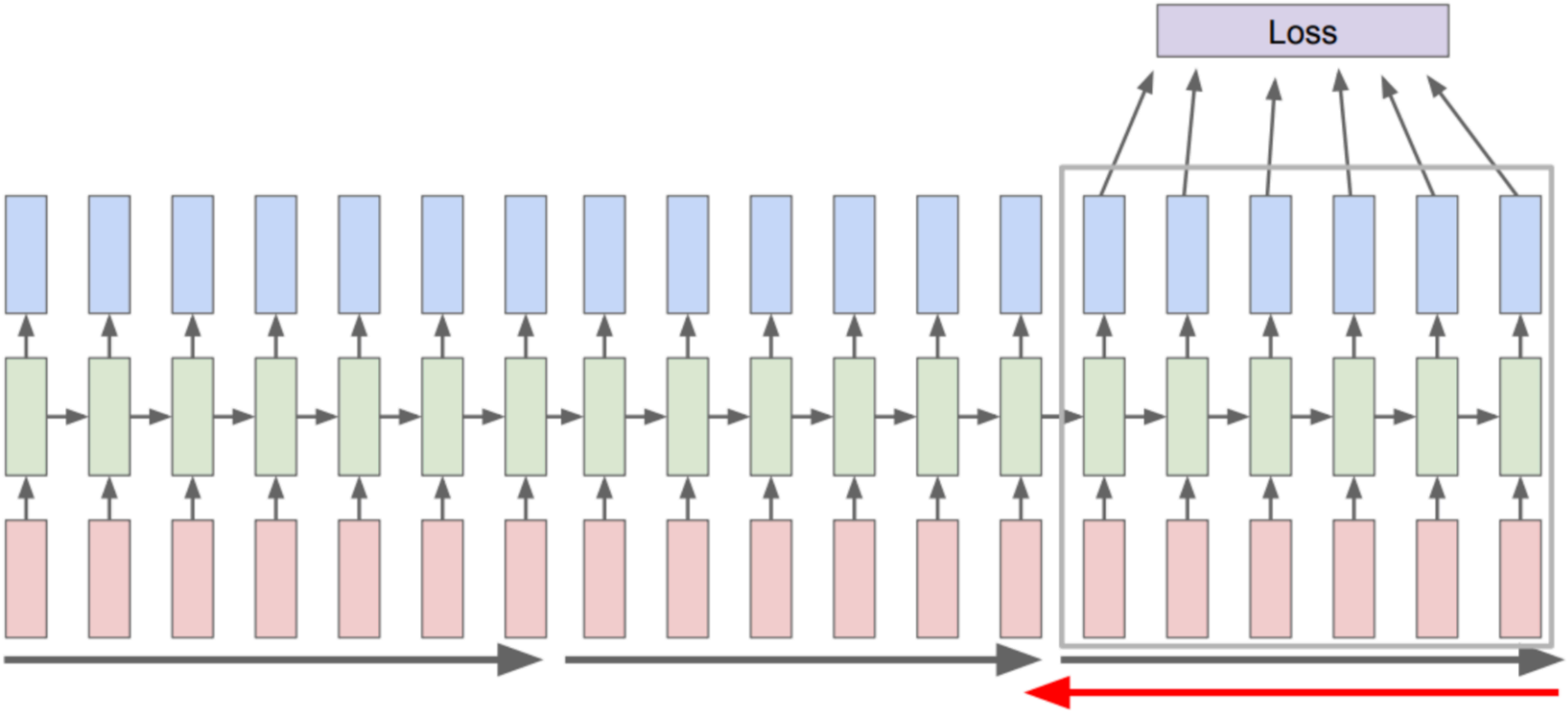
Back-Propagation Through Time (BPTT)



- Treat the unfolded network as one big feed-forward network
- This network takes in the entire sequence as input
- We compute the gradients through the usual back-propagation procedure
- We update the shared weights

The weight updates are computed for each copy in the unfolded network, then summed (or averaged) and then applied to the RNN weights

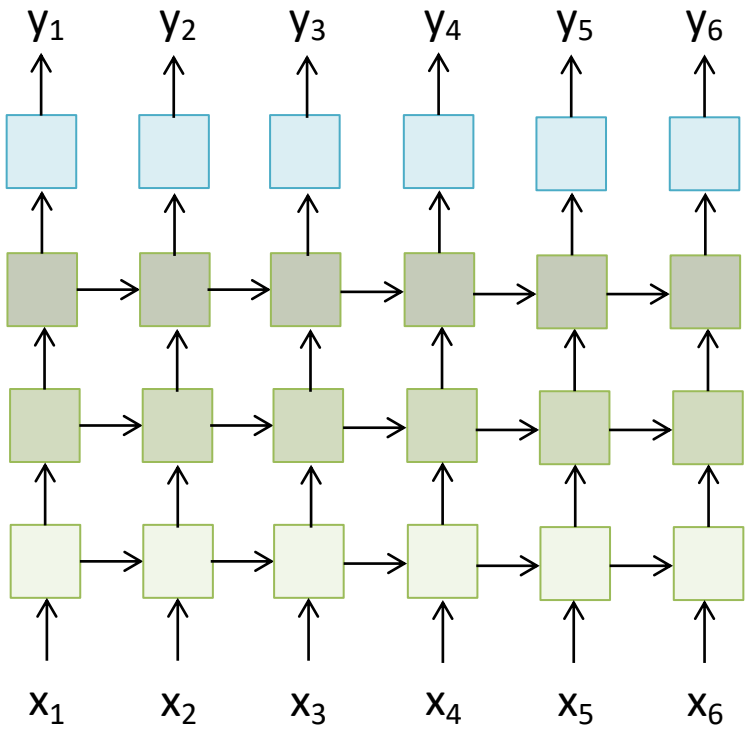
Truncated BPTT



- Run forward and back propagation through chunks of the sequence instead of the whole sequence
- Back-propagate through a chunk (**segment**) and make a gradient step on the weights
- Next batch:
  - still have the hidden state from the previous batch
  - carry the hidden states forward

Multi-layer RNNs

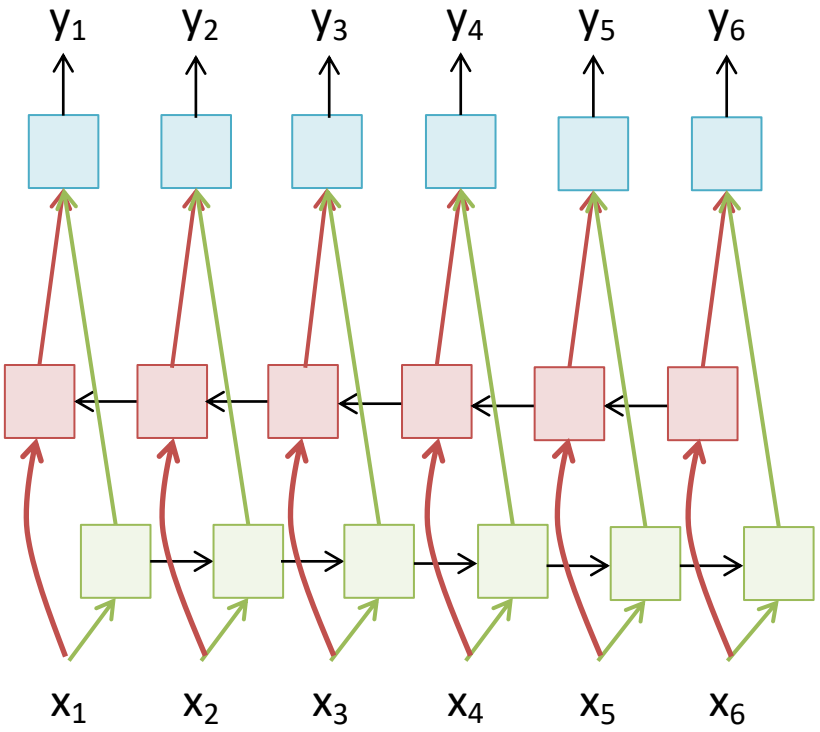
- We can of course design RNNs with multiple hidden layers



- Solve more complex sequential problems

Bi-directional RNNs

- RNNs can process the input sequence in the forward and in the reverse direction



- Popular in speech recognition

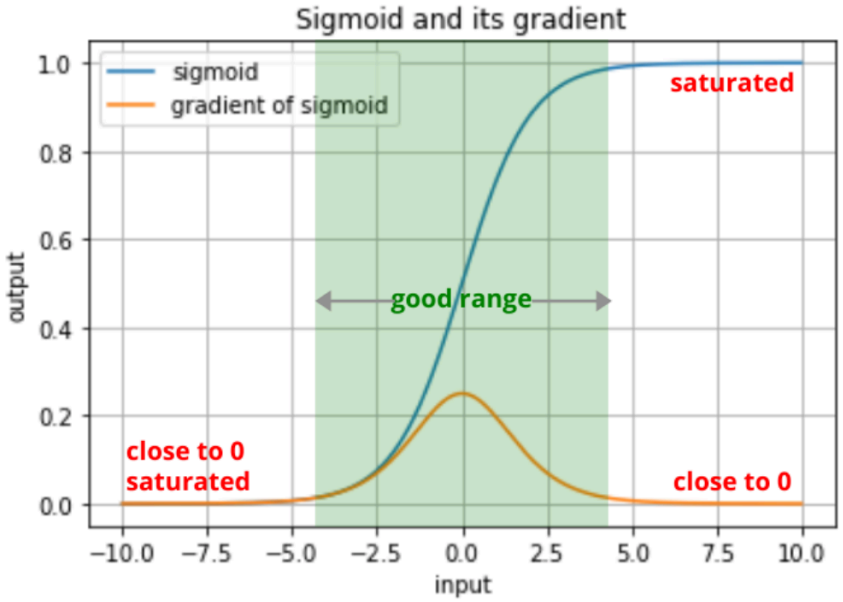
Vanishing or Exploding Gradients

- In the same way a product of ***k*** real numbers can shrink to zero or explode to infinity, so can a product of matrices
- **Vanishing Gradients:** the effect of a change in ***L*** to the weight in some layer (i.e., the *norm* of the *gradient*) becomes so small due to increased model complexity with more hidden units, that **it becomes zero after a certain point**
- **Example:**

Step 1 => 0.3 x 0.2 = 0.06

Step 2 => 0.06 x 0.4 = 0.024

Step 3 => 0.024 x 0.1 = 0.0024



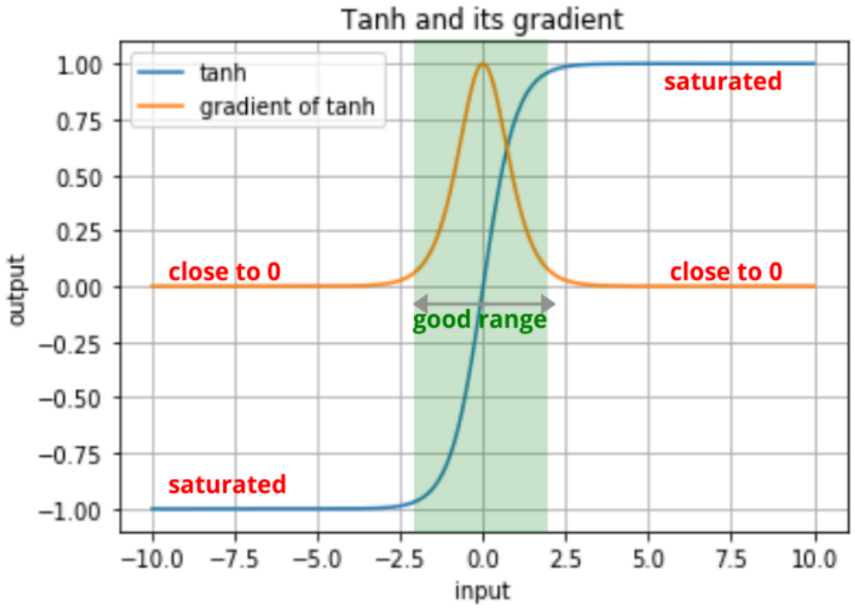
Vanishing or Exploding Gradients

- In the same way a product of ***k*** real numbers can shrink to zero or explode to infinity, so can a product of matrices
- **Vanishing Gradients:** the effect of a change in ***L*** to the weight in some layer (i.e., the *norm* of the *gradient*) becomes so small due to increased model complexity with more hidden units, that **it becomes zero after a certain point**
- **Example:**

Step 1 => 0.3 x 0.2 = 0.06

Step 2 => 0.06 x 0.4 = 0.024

Step 3 => 0.024 x 0.1 = 0.0024



Vanishing or Exploding Gradients

- In the same way a product of ***k*** real numbers can shrink to zero or explode to infinity, so can a product of matrices
- **Vanishing Gradients:** the effect of a change in ***L*** to the weight in some layer (i.e., the *norm* of the *gradient*) becomes so small due to increased model complexity with more hidden units, that **it becomes zero after a certain point**
- **Exploding Gradients:** a **large increase in the norm of the gradient during training** due to an explosion of long-term components, can result in the gradients growing exponentially

Gradient Scaling and Clipping

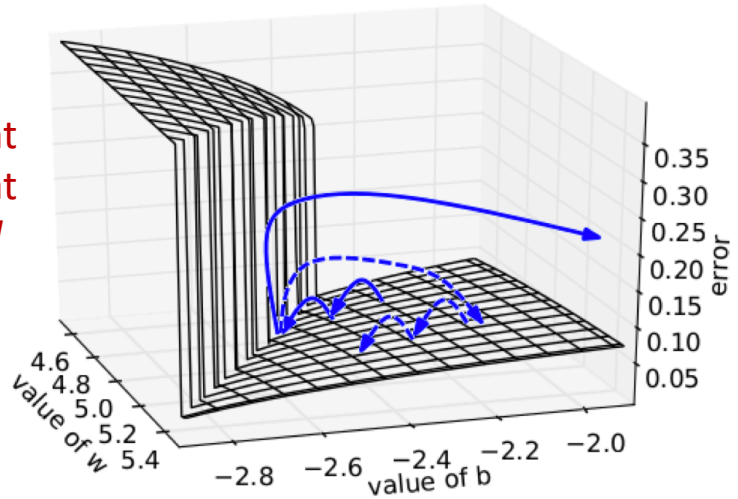
- **Gradient scaling:** **normalizing** the error gradient vector such that the vector norm (magnitude) equals a defined value, such as 1.0
- **Gradient clipping:** **forcing** the gradient values (element-wise) to a specific minimum or maximum value if the gradient exceeds an expected range
- **Together, these methods are often simply referred to as "gradient clipping"**

Algorithm 1 Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}$
end if

ensures the gradient vector $\mathbf{g}$ has norm at most equal to *threshold*

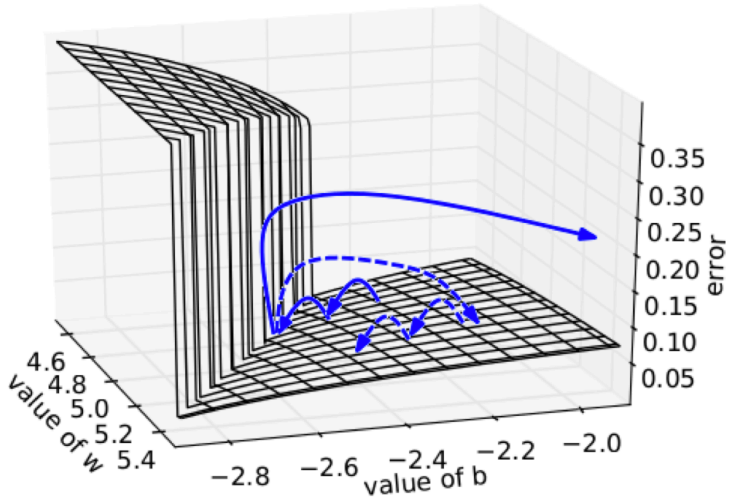
threshold: 0.5 to 10 times the average norm over a sufficiently large number of updates



Gradient Scaling and Clipping

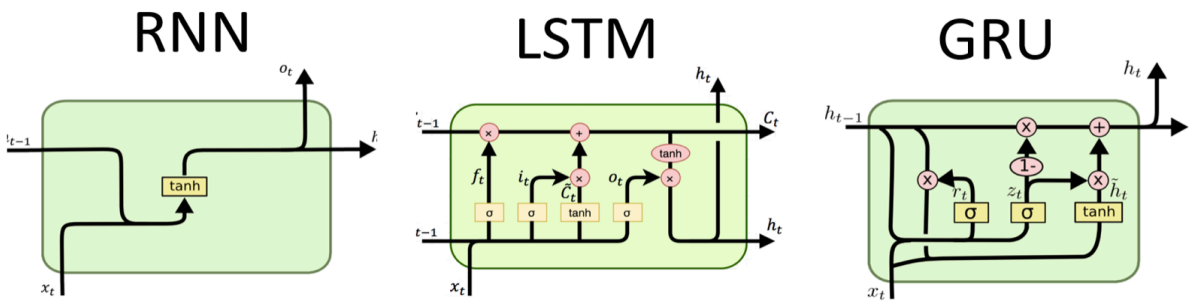
- **Gradient scaling:** **normalizing** the error gradient vector such that the vector norm (magnitude) equals a defined value, such as 1.0
- **Gradient clipping:** **forcing** the gradient values (element-wise) to a specific minimum or maximum value if the gradient exceeds an expected range
- **Together, these methods are often simply referred to as "gradient clipping"**

- Common to use the same gradient clipping configuration for all layers
- In some cases, a larger range of error gradients is permitted in the output layer compared to hidden layers



Gated RNNs

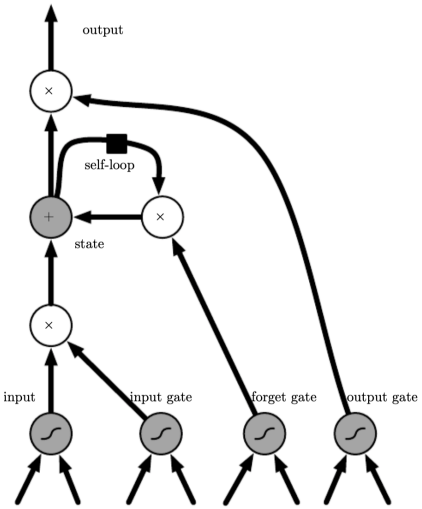
- The most effective sequence models used in practical applications:
  - **long short-term memory (LSTM)**
  - **gated recurrent unit (GRU)**



- **Gated RNNs:** create paths through time that have derivatives that neither vanish nor explode
  - allow the network to accumulate information over a long duration
  - once this information has been used the neural network can forget the old state
  - instead of manually deciding when to clear the state, we want the neural network to learn to decide when to do it

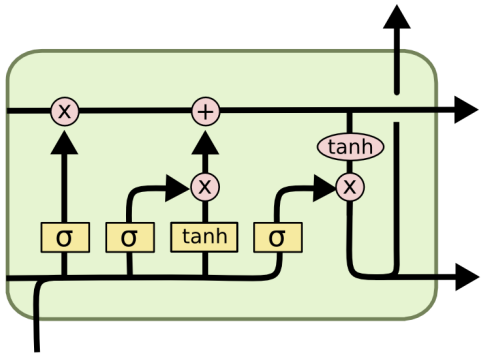
Long Short-Term Memory (LSTM) [Hochreiter et al. 1997]

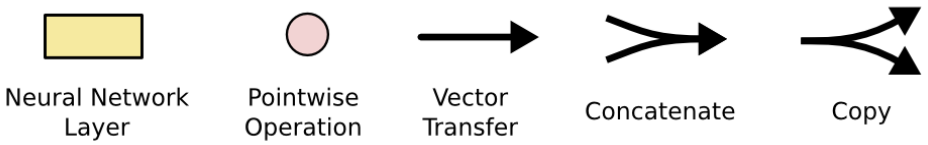
- The LSTM uses this idea of "Constant Error Flow" to create a "Constant Error Carousel" (CEC) which ensures that gradients do not decay
- The key component is a memory cell that acts like an accumulator (contains the identity relationship) over time
- Instead of computing a new state as a matrix product with the old state, it rather computes the difference between them
- Expressivity is the same, but gradients are "better behaved"

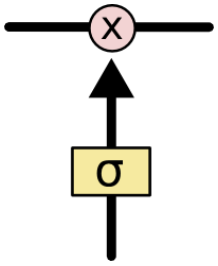


LSTM: structure

- Each cell has the same input and output as an ordinary recurrent network
- But also more parameters and a system of gating units that controls the flow of information







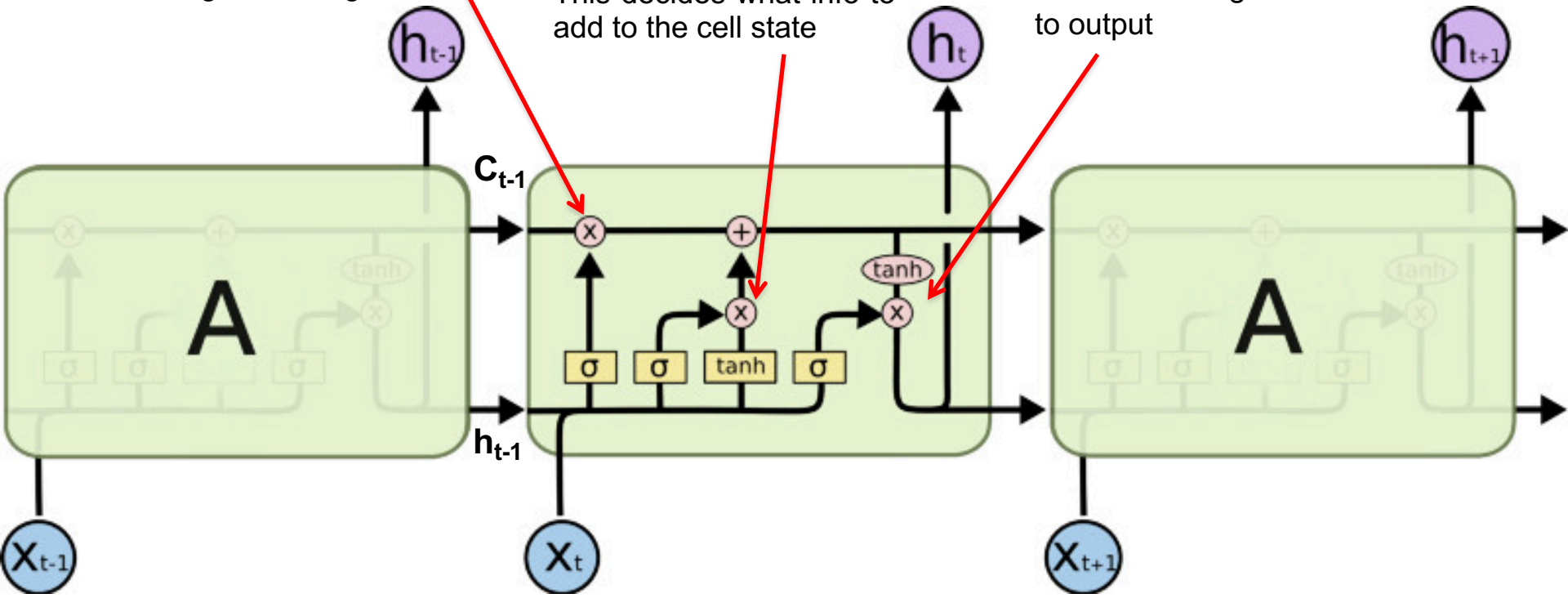
- The sigmoid layer outputs numbers between 0-1 which determines how much each component should be let through
- **x gate:** point-wise multiplication
- **+ gate:** point-wise addition

LSTM

This sigmoid gate determines how much information goes through

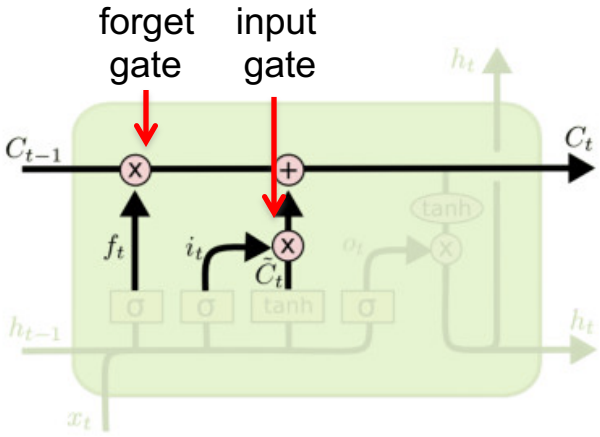
This decides what info to add to the cell state

The output gate controls what goes to output

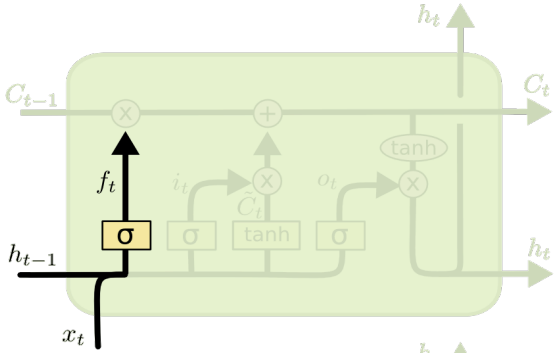


C_t: cell state (flows along unchanged) slowly, with only minor linear interactions

Long-term memory capability: stores and loads information of not necessarily immediately previous events

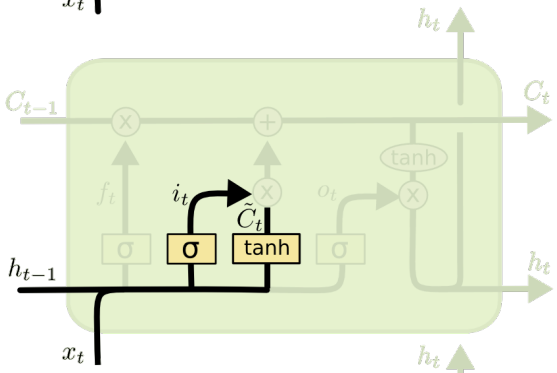


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$



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

f_t tries to estimate what features of the cell state should be forgotten

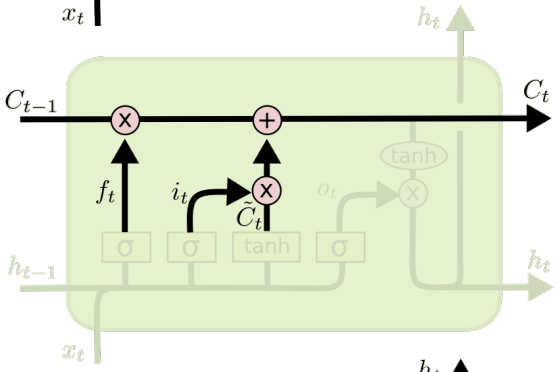


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

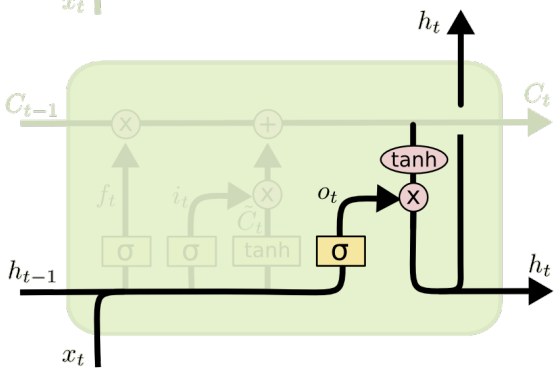
i_t decides what components of cell memory should be updated

$\tilde{C}_t$ determines the new input that may go to the long-term memory



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

updates the cell state

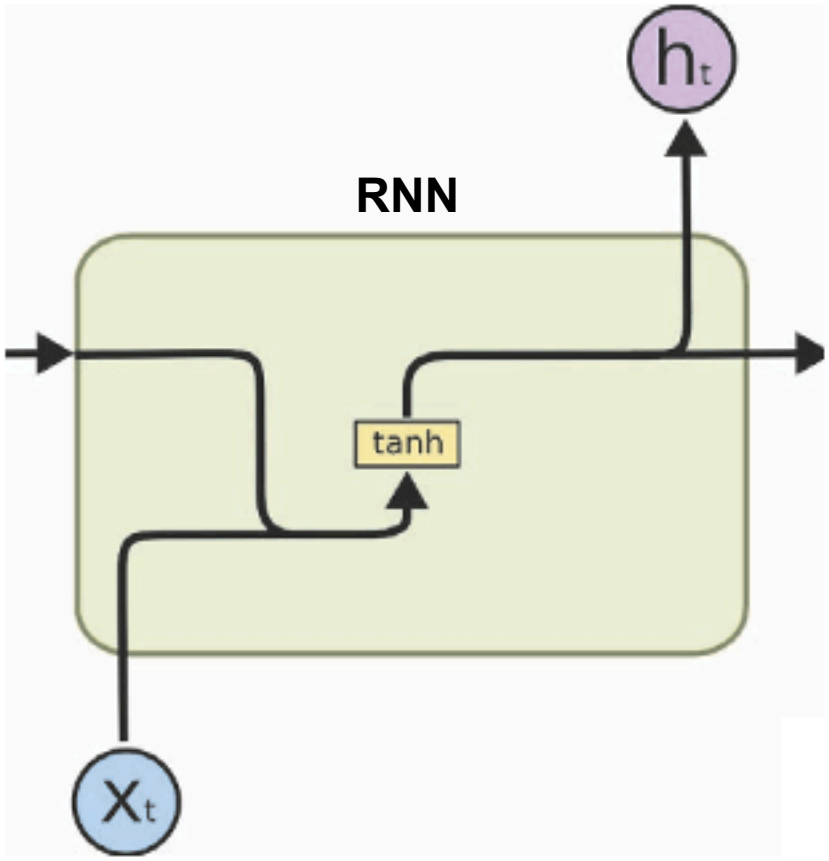


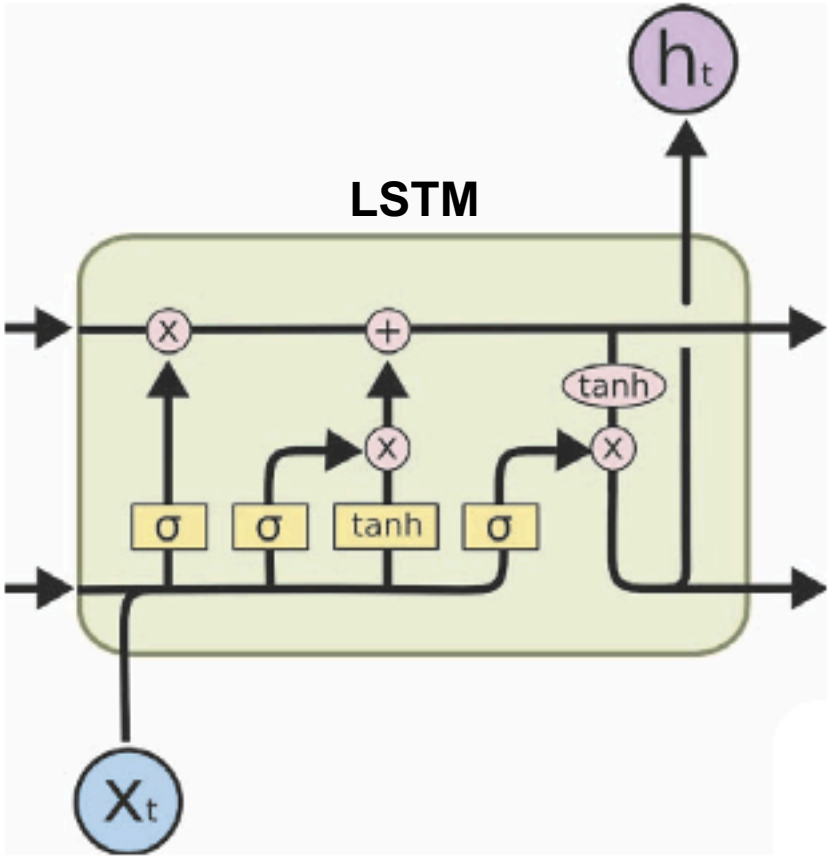
$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$

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

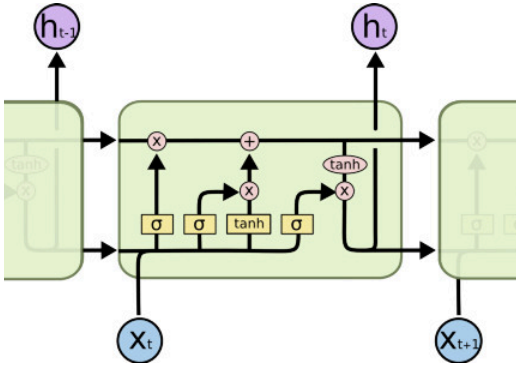
decides what part of the cell state goes to output

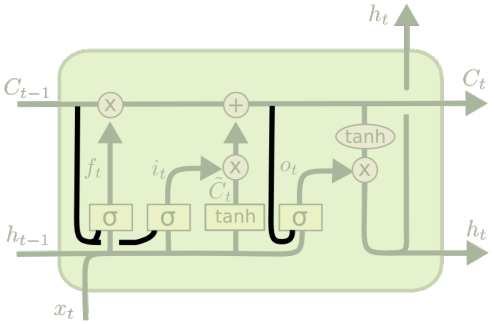
RNN vs LSTM





Peephole LSTM
[Gers & Schmidhuber 2000]

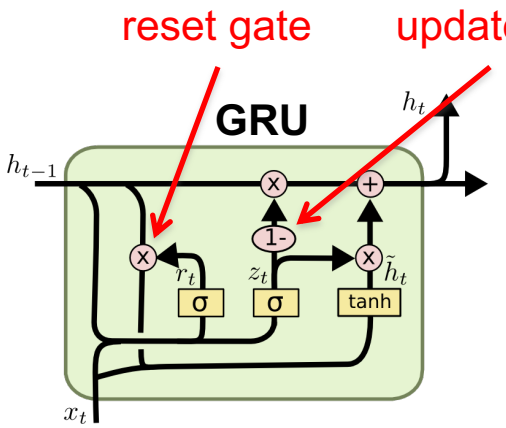




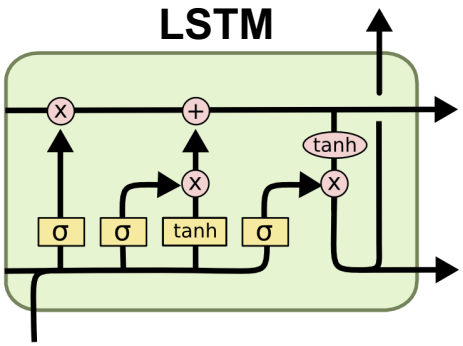
$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

- Allows "peeping" into the memory
- Gates can see the output from the previous time step, and can hence affect the construction of the output

Gated Recurrent Unit (GRU) [Cho et al. 2014]



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



- A single gating unit that combines the forget and input into a single update gate
- It also merges the cell state and hidden state
- This is simpler than LSTM
- There are many other variants too
- Has fewer parameters than an LSTM and has been shown to outperform LSTM on some tasks

LSTM vs GRU

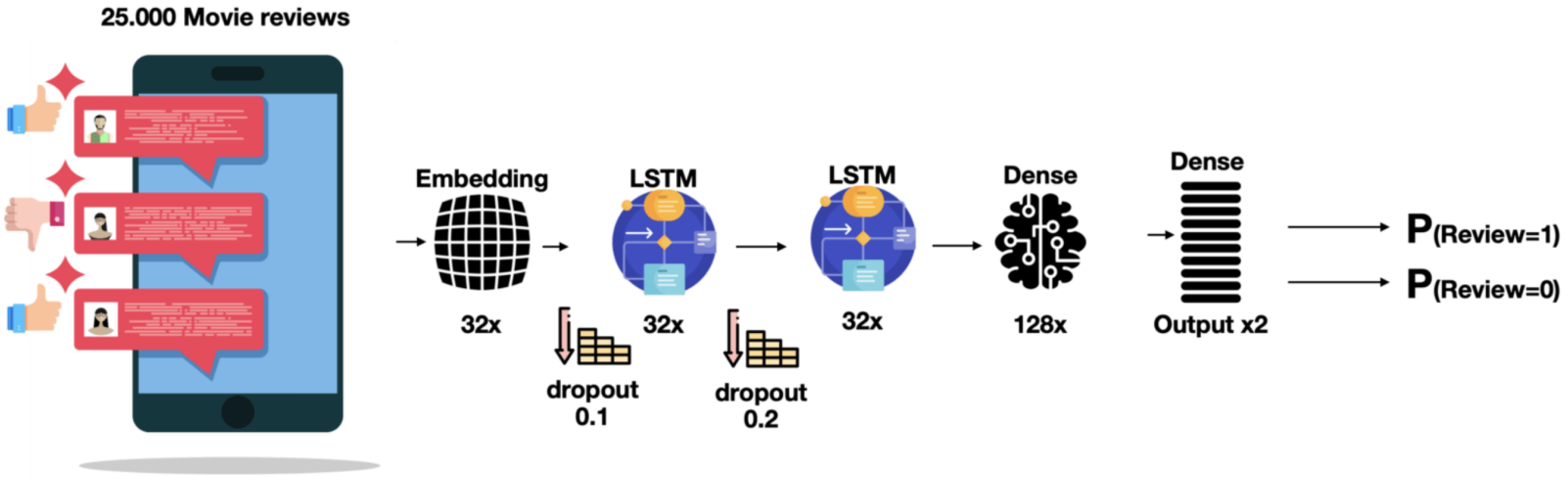
- GRU uses less training parameters and hence needs less memory
- GRU executes faster than LSTM whereas LSTM is more accurate on larger datasets and longer sequences
- One can choose LSTM when dealing with long sequences and accuracy is concerned
- GRU is used when we have less memory availability

Evolutionary Architecture Search

- A list of top-100 architectures so far is maintained, initialized with the LSTM and the GRU
- The GRU is considered as the baseline to beat
- New architectures are proposed and are retained based on their performance ratio with GRU
- All architectures are evaluated on **three** problems
  - **arithmetic:** compute digits of sum or difference of two numbers provided as inputs
  - **XML modeling:** predict next character in XML modeling
  - **Penn Tree-Bank Language Modeling:** predict distributions over words

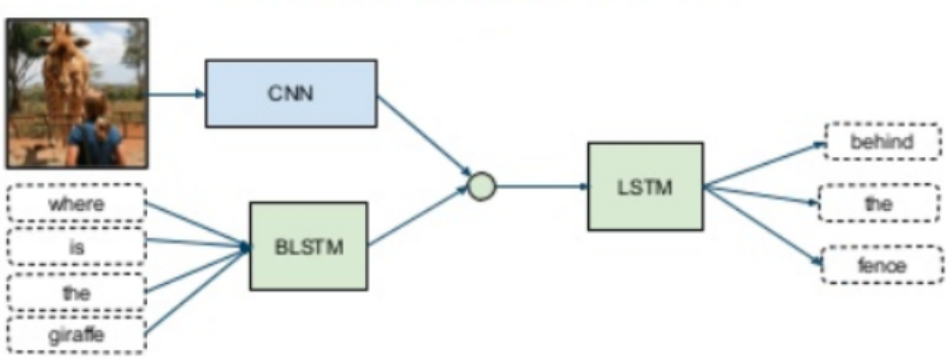
An Empirical Exploration of Recurrent Network Architectures, Jozefowicz *et al*., 2015

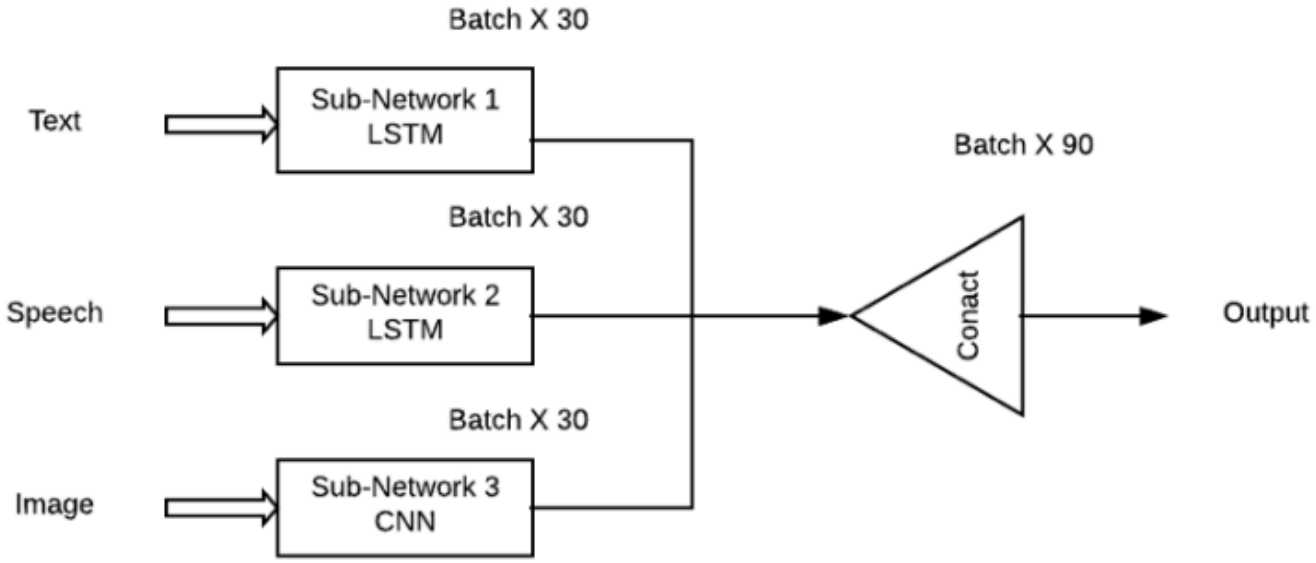
Sentiment Classification (revisited)



- **Input layer:** implementing vectorization through a 32-dimensional embedding, producing **32 features** for each word of the review
- **Second layer:** an **LSTM** layer with long memory
- **Third layer:** an **LSTM** with shorter memory
- **Fourth layer:** a fully-connected neural network with **128** nodes and an output probability that the review is **positive (1)** or **negative (0)**

Multimodal learning





Coming up next…

Jan 16	Introduction to machine learning
Jan 18	Regression analysis
Jan 19	Laboratory session 1: numpy and linear regression
Jan 23	Ensemble learning
Jan 25	Deep learning I: Training neural networks
Jan 26	Laboratory session 2: ML pipelines, ensemble learning
Jan 30	Deep learning II: Convolutional neural networks
Feb 1	Laboratory session 3: training NNs and tensorflow
Feb 6	Deep learning III: Recurrent neural networks
Feb 8	**Laboratory session 4: CNNs and RNs**
Feb 13	**Deep learning IV: Autoencoders, transformers, and attention**
Feb 20	Time series classification

TODOs

- **Reading:**
  - The Deep Learning Book: Chapter 10
- **Homework 2**
  - Out: today
  - Due: Feb 23