RNNs, LSTMs, GRUs, Transformer

Lecture 8

Parijat Dube

Agenda

- Introduction to Recurrent Neural Networks (RNNs)
- Vanilla RNN architecture
- Gradient flow in RNNs
- Long Short-Term Memory (LSTM) networks
- LSTM variants
- Sequence learning with RNNs
- Word embeddings
- Attention mechanism
- Transformer architecture

Recurrent Neural Networks

- Human brain deals with information streams. Most data is obtained, processed, and generated sequentially.
  - E.g., listening: soundwaves → vocabularies/sentences
  - E.g., action: brain signals/instructions → sequential muscle movements
- Human thoughts have persistence; humans don't start their thinking from scratch every second.
  - As you read this sentence, you understand each word based on your prior knowledge.
- The applications of standard Artificial Neural Networks (and also Convolutional Networks) are limited due to:
  - They only accepted a fixed-size vector as input (e.g., an image) and produce a fixed-size vector as output (e.g., probabilities of different classes).
  - These models use a fixed amount of computational steps (e.g. the number of layers in the model).
- Recurrent Neural Networks (RNNs) are a family of neural networks introduced to **learn sequential data**.
  - Inspired by the temporal-dependent and persistent human thoughts

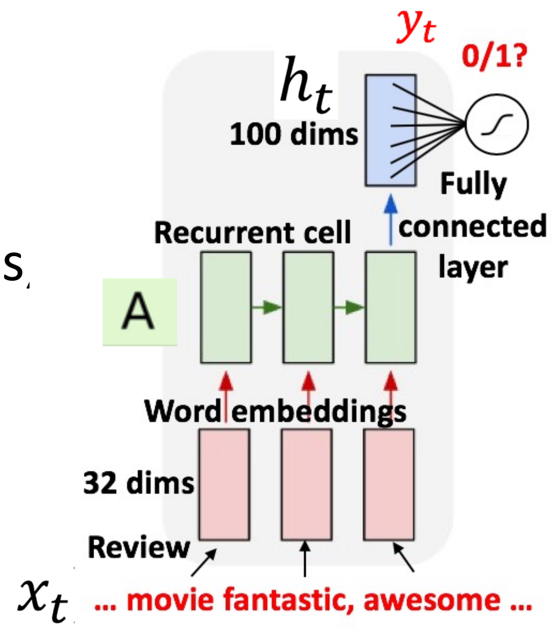
Real-life Sequence Learning Applications

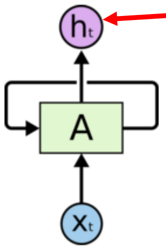
- RNNs can be applied to various type of sequential data to learn the temporal patterns.
  - Time-series data (e.g., stock price) → Prediction, regression
  - Raw sensor data (e.g., signal, voice, handwriting) → Labels or text sequences
  - Text → Label (e.g., sentiment) or text sequence (e.g., translation, summary, answer)
  - Image and video → Text description (e.g., captions, scene interpretation)

Task	**Input**	**Output**
Activity Recognition (Zhu et al. 2018)	Sensor Signals	Activity Labels
Machine translation (Sutskever et al. 2014)	English text	French text
Question answering (Bordes et al. 2014)	Question	Answer
Speech recognition (Graves et al. 2013)	Voice	Text
Handwriting prediction (Graves 2013)	Handwriting	Text
Opinion mining (Irsoy et al. 2014)	Text	Opinion expression

Recurrent Neural Networks

- Recurrent Neural Networks are networks with loops, allowing information to persist.





Output is to predict a vector h_t, where $output\ y_t = \varphi(h_t)$ at some time steps (t)

Recurrent Neural Networks have loops.

In the above diagram, a chunk of neural network, ***A* = *f***

In the above diagram, a chunk of neural network, $A = f_W$, looks at some input x_t and outputs a value h_t. A loop allows information to be passed from one step of the network to the next.

new state

old state

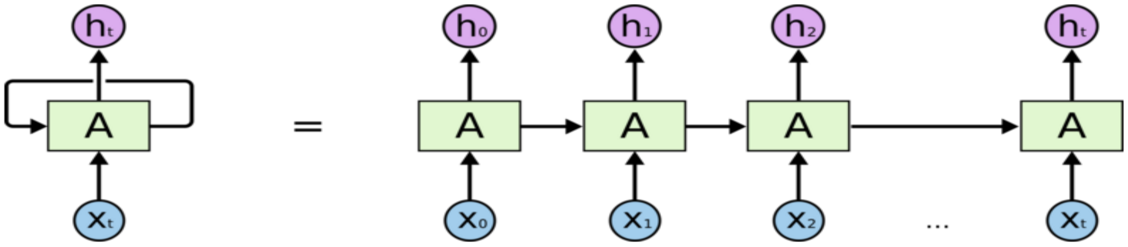
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

function with parameter W

Input vector at some time step

Recurrent Neural Networks

- Unrolling RNN

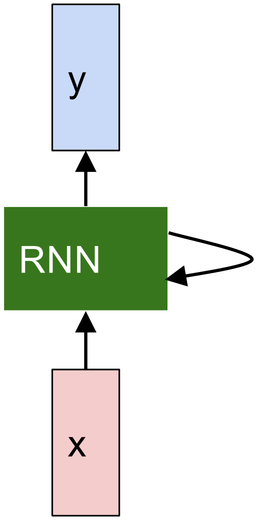


An unrolled recurrent neural network.

- A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. The diagram above shows what happens if we **unroll the loop**.
- RNNs combine the input vector with their state vector with a fixed (but learned) function to produce a new state vector.

(Vanilla) Recurrent Neural Network

The state consists of a single *“hidden”* vector **h**:

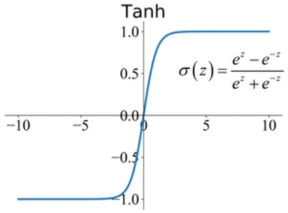


$$y_t = W_{hy}h_t + b_y$$

$$h_t = f_W(h_{t-1}, x_t)$$

↓

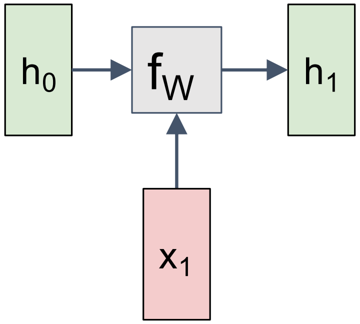
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$



Sometimes called a “Vanilla RNN” or an “Elman RNN” after Prof. Jeffrey Elman

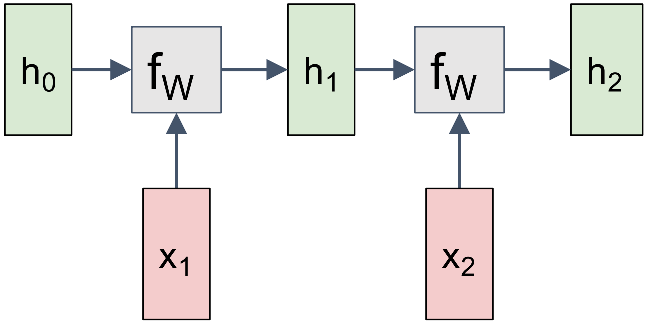
Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

RNN: Computational Graph



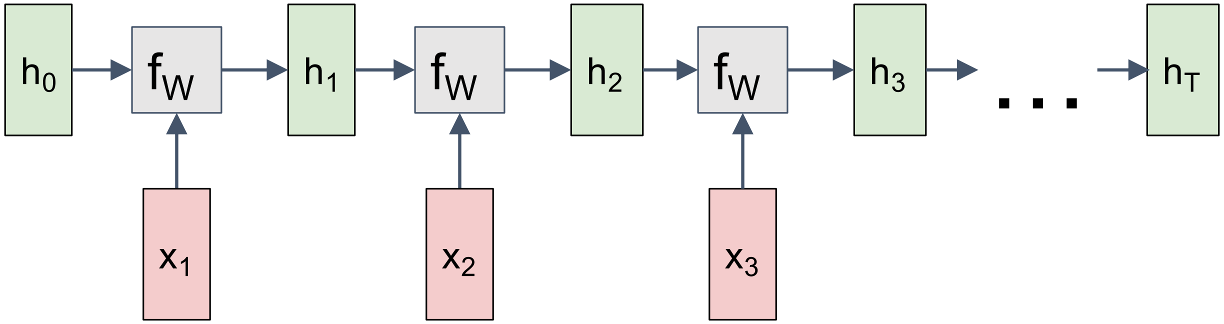
Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

RNN: Computational Graph



Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

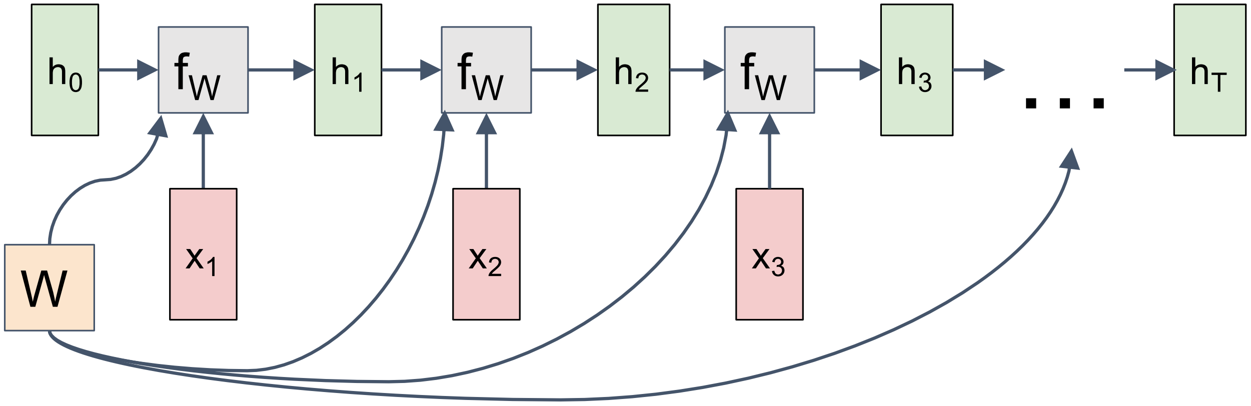
RNN: Computational Graph



Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

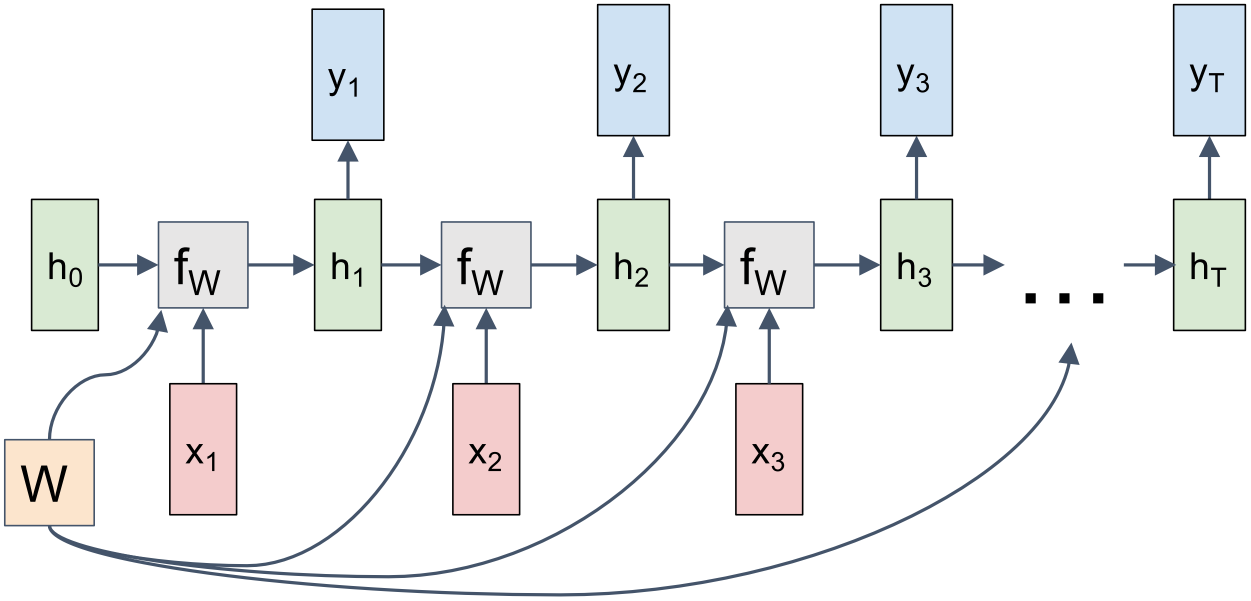
RNN: Computational Graph

Re-use the same weight matrix at every time-step



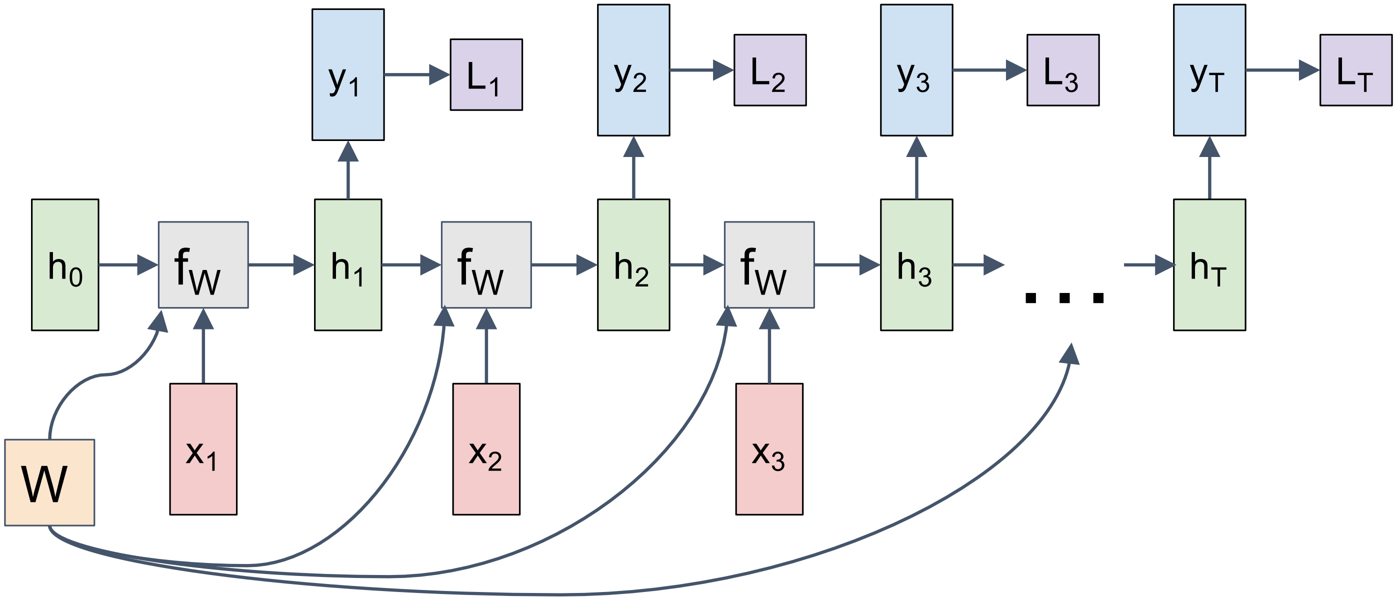
Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

RNN: Computational Graph: Many to Many



Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

RNN: Computational Graph: Many to Many

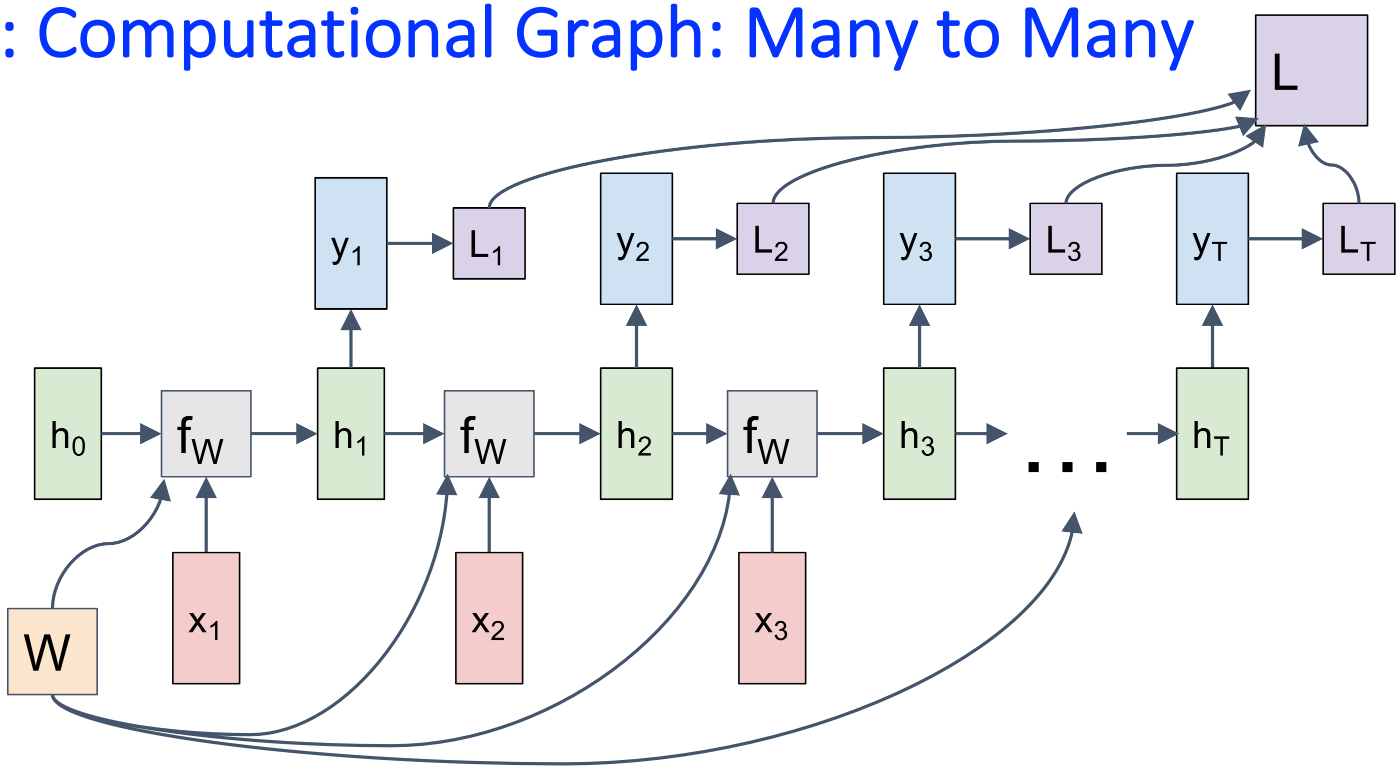


Many-to-Many sequence learning can be used for machine translation where the input sequence is in some language, and the output sequence is in some other language.

It can be used for Video Classification as well, where the input sequence is the feature representation of each frame of the video at different time steps.

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

RNN: Computational Graph: Many to Many



Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Recurrent Neural Networks Features

- The recurrent structure of RNNs enables the following characteristics:
  - Specialized for processing a sequence of values $x^{(1)}, \dots, x^{(\tau)}$
    - Each value $x^{(i)}$ is processed with the **same network *A* that preserves past information**

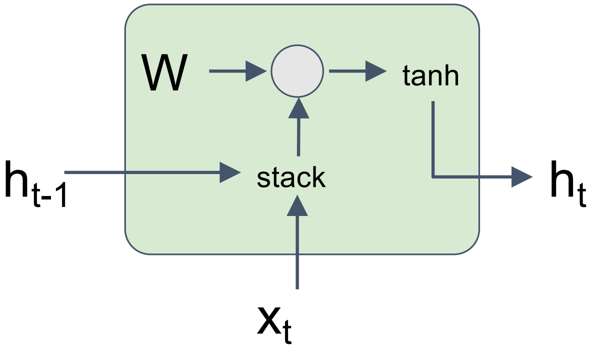
  - Can scale to much **longer sequences** than would be practical for networks without a recurrent structure
    - Reusing network ***A*** reduces the required amount of parameters in the network

  - Can process **variable-length sequences**
    - The network complexity does not vary when the input length change

  - However, vanilla RNNs suffer from the training difficulty due to **exploding and vanishing gradients**.

Vanilla RNN Gradient Flow

Bengio et al, “Learning long-term dependencies with gradient descent is difficult”, IEEE Transactions on Neural Networks, 1994
Pascanu et al, “On the difficulty of training recurrent neural networks”, ICML 2013



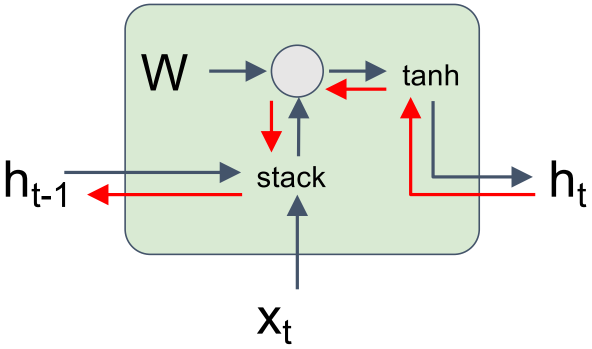
$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\
&= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)
\end{aligned}
$$

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Vanilla RNN Gradient Flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

Backpropagation from h_t to h_{t-1} multiplies by W (actually W_{hh})

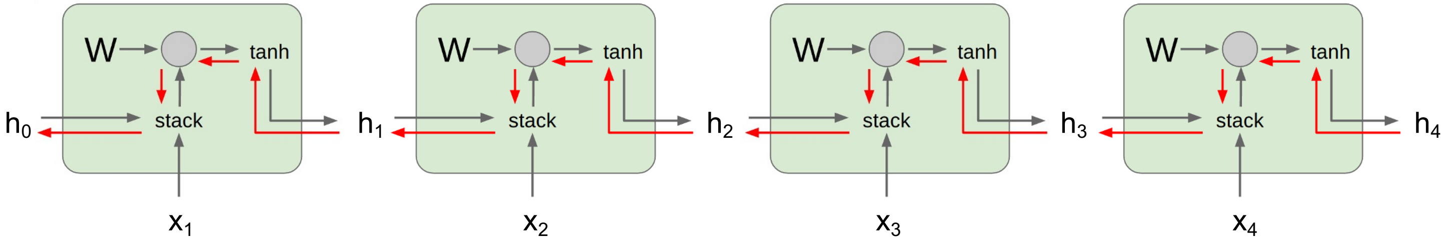


$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\
&= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)
\end{aligned}
$$

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Vanilla RNN Gradient Flow

Bengio et al, “Learning long-term dependencies with gradient descent is difficult”, IEEE Transactions on Neural Networks, 1994
Pascanu et al, “On the difficulty of training recurrent neural networks”, ICML 2013

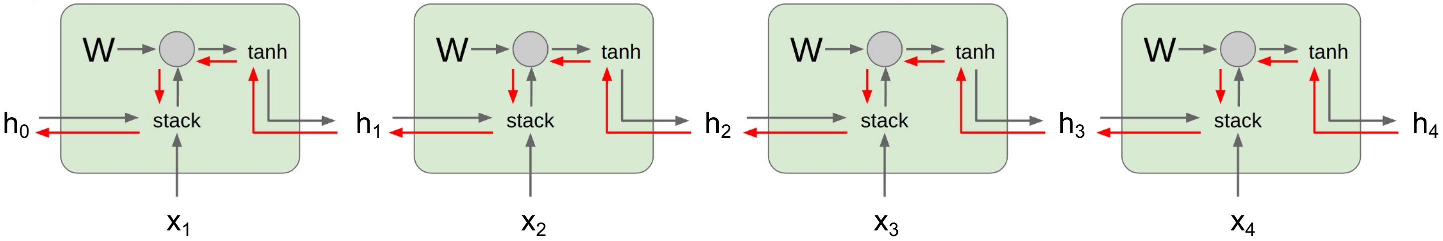


Computing gradient of h_0 involves many factors of W
(and repeated tanh)

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Vanilla RNN Gradient Flow

Bengio et al, “Learning long-term dependencies with gradient descent is difficult”, IEEE Transactions on Neural Networks, 1994
Pascanu et al, “On the difficulty of training recurrent neural networks”, ICML 2013



Computing gradient of h_0 involves many factors of W (and repeated tanh)

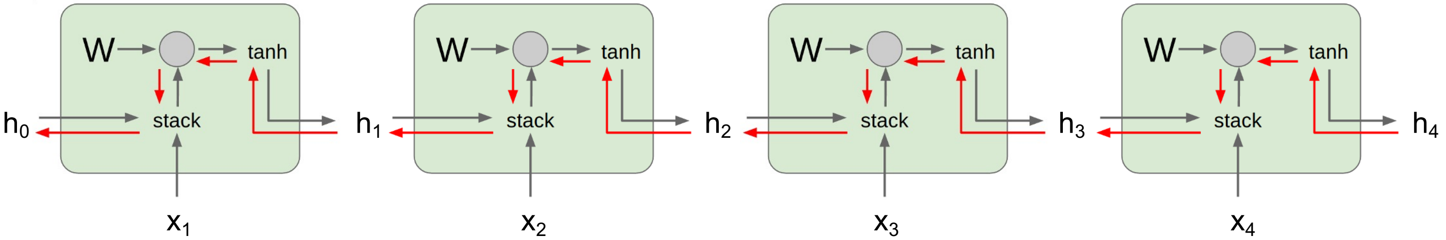
Largest singular value > 1:
Exploding gradients

Largest singular value < 1:
Vanishing gradients

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Vanilla RNN Gradient Flow

Bengio et al, “Learning long-term dependencies with gradient descent is difficult”, IEEE Transactions on Neural Networks, 1994
Pascanu et al, “On the difficulty of training recurrent neural networks”, ICML 2013

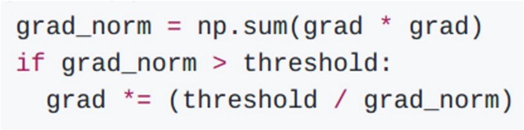


Computing gradient of h_0 involves many factors of W
(and repeated tanh)

Largest singular value > 1:
Exploding gradients

Largest singular value < 1:
Vanishing gradients

→ **Gradient clipping**: Scale gradient if its Euclidian norm

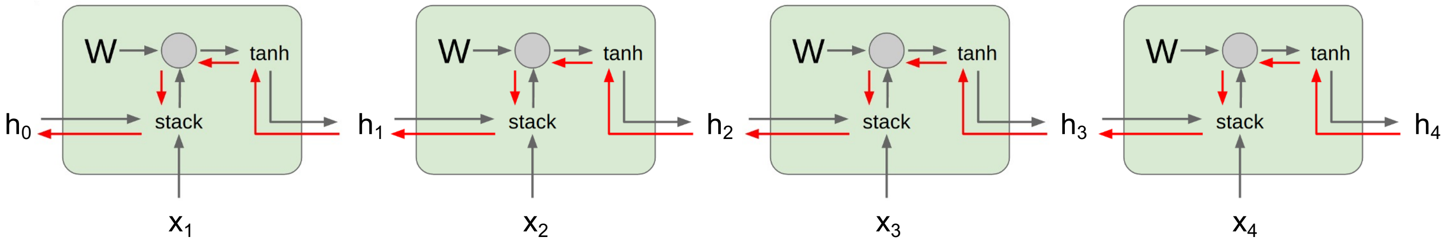


```
grad_norm = np.sum(grad * grad)
if grad_norm > threshold:
  grad *= (threshold / grad_norm)
```

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Vanilla RNN Gradient Flow

Bengio et al, “Learning long-term dependencies with gradient descent is difficult”, IEEE Transactions on Neural Networks, 1994
Pascanu et al, “On the difficulty of training recurrent neural networks”, ICML 2013



Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1:
Exploding gradients

Largest singular value < 1:
Vanishing gradients

→ Change RNN architecture

- **Exploding gradients**: Slight error in the late time steps causes drastic updates in the early time steps → Unstable learning
- **Vanishing gradients:** Gradients passed to the early time steps is close to 0. → Uninformed correction

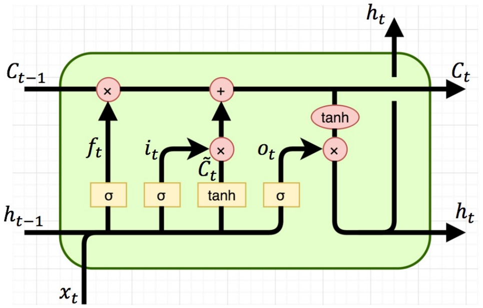
Pros and Cons of using RNNs

- Pros:
  - Possibility of processing input of any length
  - Model size not increasing with size of input
  - Computation takes into account historical information
  - Weights are shared across time
- Cons:
  - Computation being slow
  - Difficulty of accessing information from a long time ago
  - Cannot consider any future input for the current state
  - Suffers from vanishing and exploding gradient problem and hence training is unstable

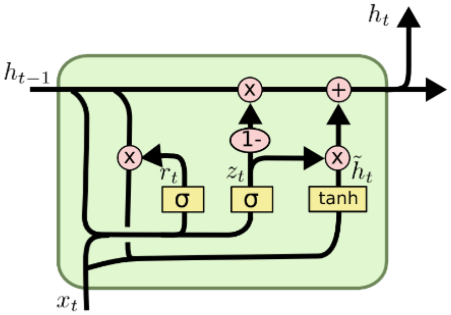
Networks with Memory

- Vanilla RNN operates in a “multiplicative” way (repeated tanh).
- Two recurrent cell designs were proposed and widely adopted:
  - **Long Short-Term Memory (LSTM)** (Hochreiter and Schmidhuber, 1997)
  - Gated Recurrent Unit (GRU) (Cho et al. 2014)
- Both designs process information in an “additive” way with gates to control information flow.
  - **Sigmoid gate outputs numbers between 0 and 1, describing how much of each component should be let through.**

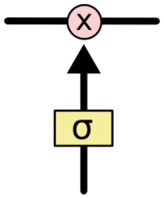
E.g. $f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \;+\; b_f\right) = Sigmoid\,(\,W_f x_t + U_t h_{t\text{-}1} + b_f\,)$



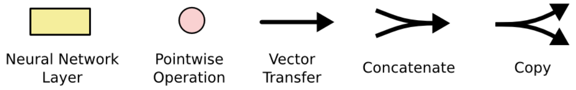
Standard LSTM Cell



GRU Cell



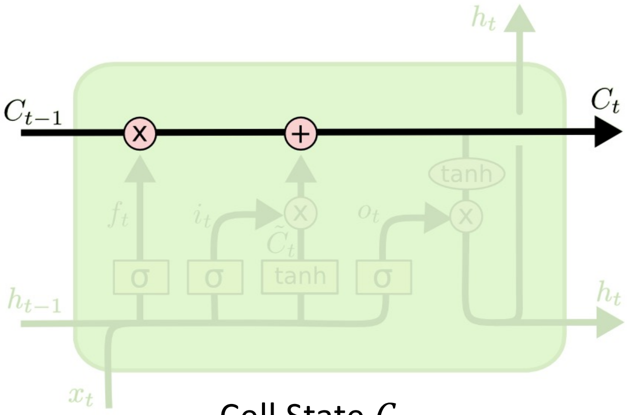
A Sigmoid Gate



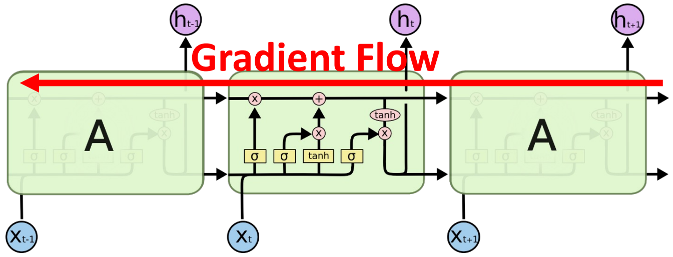
Long Short-Term Memory (LSTM)

- The key to LSTMs is the **cell state**.
  - **Stores information of the past → long-term memory**
  - **Passes along time steps with minor linear interactions → "additive"**
  - Results in an **uninterrupted gradient flow** → errors in the past pertain and impact learning in the future

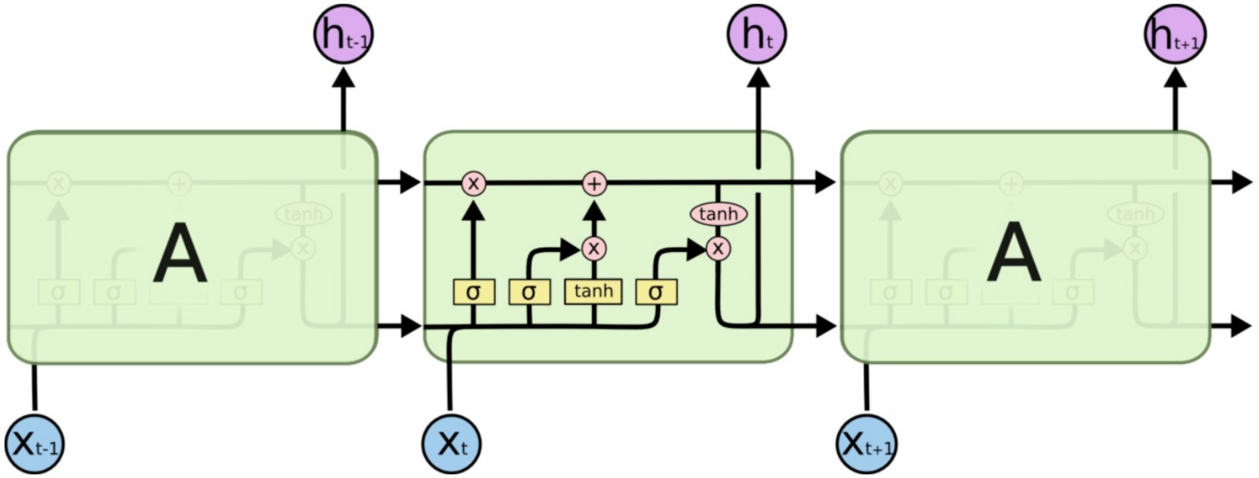
- The LSTM cell manipulates input information with three gates.
  - **Input gate** → controls the intake of new information
  - **Forget gate** → determines what part of the cell state to be updated
  - **Output gate** → determines what part of the cell state to output



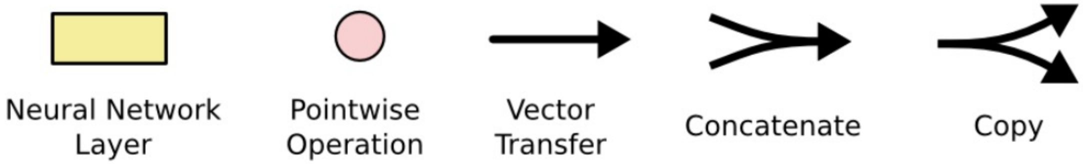
Cell State C_t



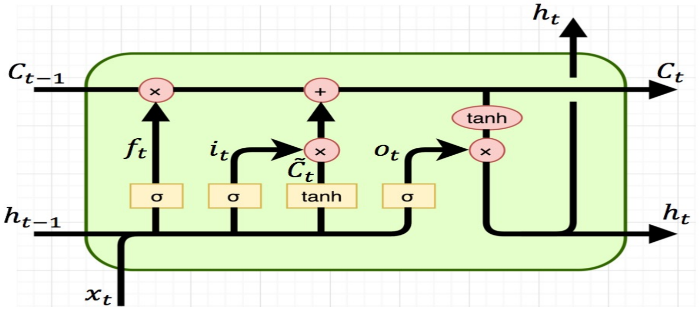
Long Short-Term Memory Networks (LSTM)



4 gates
4 fully connected layers
4 activation functions
4 math operations



LSTM: Components & Flow



- LSM unit output

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

- ***Output gate*** units

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

- Transformed memory cell contents

$$\tanh\left(C_t\right)$$

- Gated update to memory cell units

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

- ***Forget gate*** units

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

- ***Input gate*** units

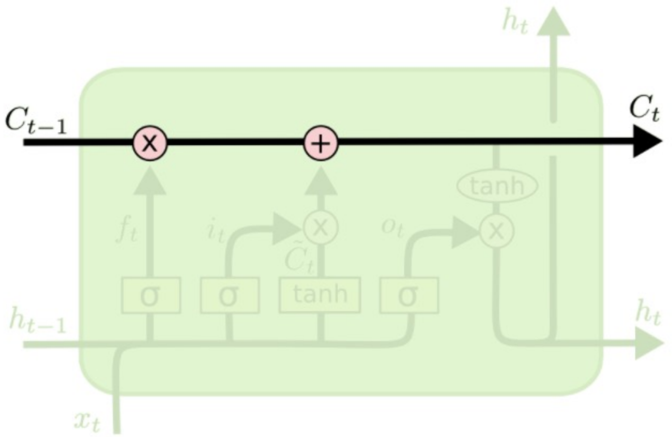
$$i_t = \sigma\left(\underline{W_i \cdot \left[h_{t-1}, x_t\right]\ +\ b_i}\right)$$

- Potential *input* to memory cell

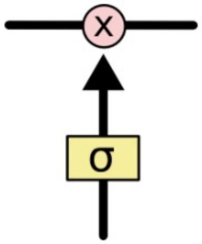
$$\tilde{C}_t = \tanh(\underline{W_C \cdot \left[h_{t-1}, x_t\right]\ +\ b_C})$$

Cell state and Gates in LSTMs

Cell state carries cumulative information of the sequence data from one time step to the next time step till the end of the sequence.



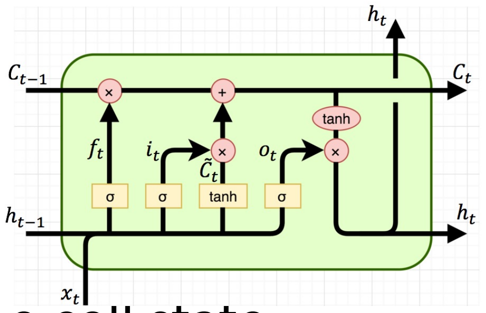
- The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.
- Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.



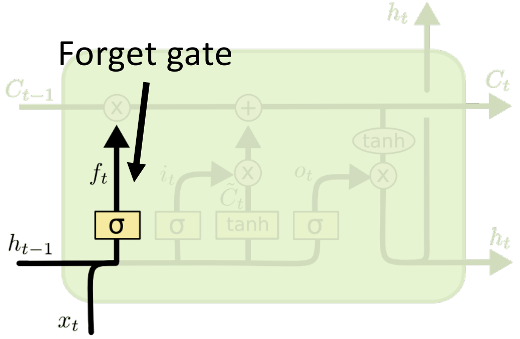
Cell state vs hidden state

- Hidden state feature vector will pass through the Neural network and gets to interact with weight matrices on the way. This causes vanishing or exploding gradient issues.
- Cell state acts as a highway without touching any fully connected layers on the way, it is literally similar to the residual block in a resent architecture.
- Cell state is a feature vector, which takes updates from input at every time step, and gives out the hidden state output at every time step. This hidden state branches out as output of the current time step and hidden state to the next time step.

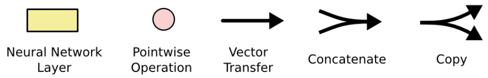
Step-by-step LSTM Walk Through



- **Step 1**: Decide what information to throw away from the cell state (memory) ➔ $f_t * C_{t-1}$
  - The output of the previous state h_{t-1} and the new information x_t jointly determine what to forget
    - h_{t-1} contains selected features from the memory C_{t-1}
  - Forget gate f_t ranges between $[0, 1]$

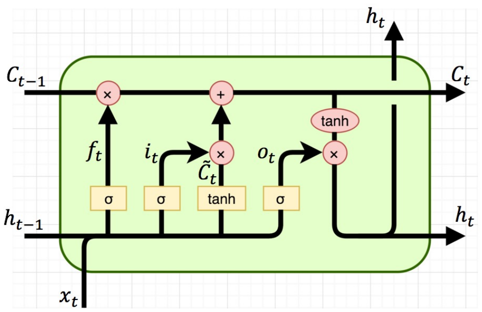

Forget gate

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

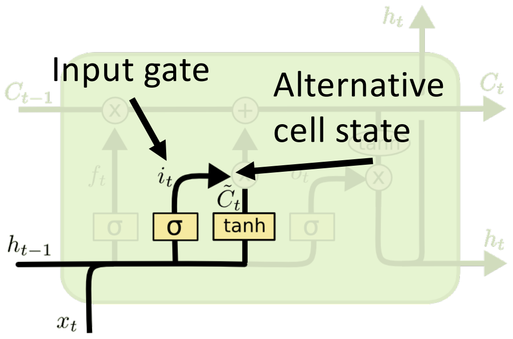


Text processing example:
Cell state may include the gender of the current subject (h_{t-1}). When the model observes a new subject (x_t), it may want to forget ($f_t \to 0$) the old subject in the memory (C_{t-1}).

Step-by-step LSTM Walk Through



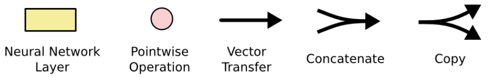
- **Step 2**: Prepare the updates for the cell state from input ➔ $i_t * \tilde{C}_t$
  - An alternative cell state $\widetilde{C_t}$ is created from the new information x_t with the guidance of h_{t-1}.
  - Input gate i_t ranges between $[0, 1]$


Input gate

Alternative cell state

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

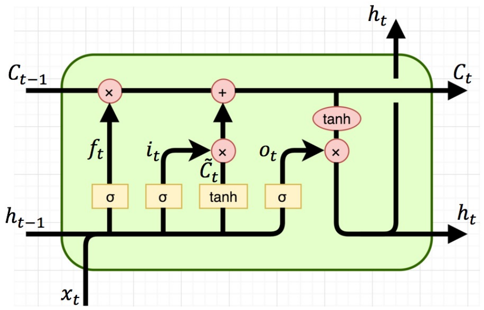
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$



Example:
The model may <u>want to add</u> ($i_t \rightarrow 1$) the <u>gender of new subject</u> ($\widetilde{C_t}$) to the cell state to replace the old one it is forgetting.

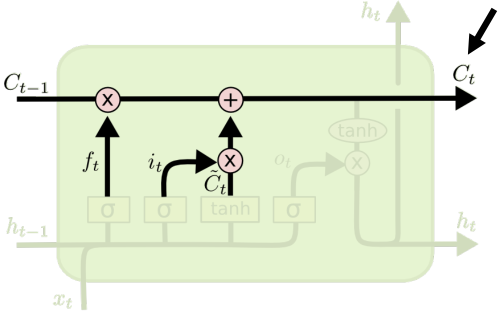
Input gate branch and the cell state branch will ensure the cell state is updated with the feature weights of current input without forgetting the important features in the cell state.

Step-by-step LSTM Walk Through

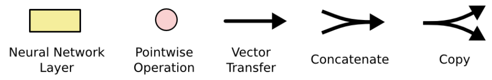


- **Step 3**: Update the cell state ➔ $f_t * C_{t-1} + i_t * \tilde{C}_t$
  - The new cell state C_t is comprised of information from the past $f_t * C_{t-1}$ and valuable new information $i_t * \tilde{C}_t$
  - $*$ denotes elementwise multiplication

New cell state

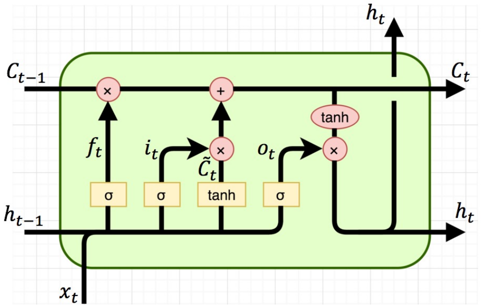


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

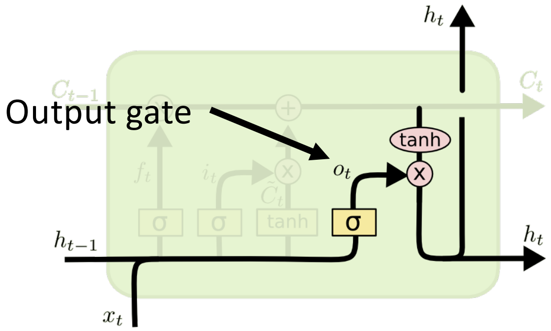


Example:
The model drops the old gender information ($f_t * C_{t-1}$) and adds new gender information ($i_t * \tilde{C}_t$) to form the new cell state (C_t).

Step-by-step LSTM Walk Through

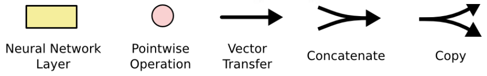


- **Step 4**: Decide the filtered output from the new cell state ➔ $o_t * \tanh(C_t)$
  - tanh function filters the new cell state to characterize stored information
    - Significant information in C_t → ± 1
    - Minor details → 0
  - Output gate o_t ranges between $[0, 1]$
  - h_t serves as a control signal for the next time step



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

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

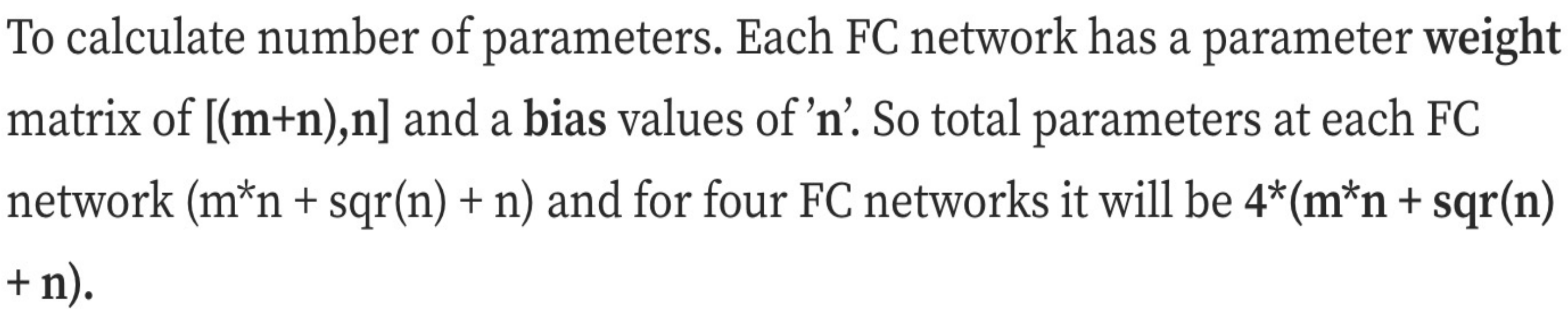


Example:
Since the model just saw a new subject (x_t), it might <u>want to output</u> ($o_t \to 1$) information relevant to a verb ($\tanh(C_t)$), e.g., singular/plural, in case a verb comes next.

Parameter calculation

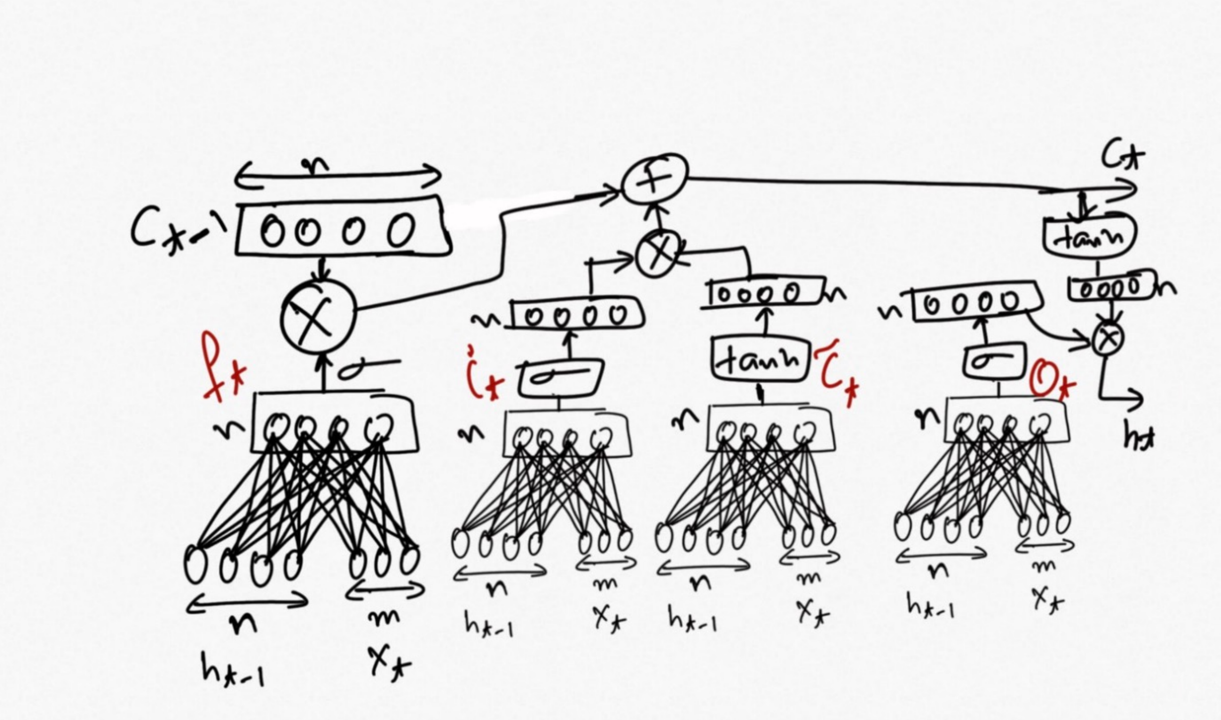
- Input to all FC networks is a concatenated vector of current input and hidden state vector

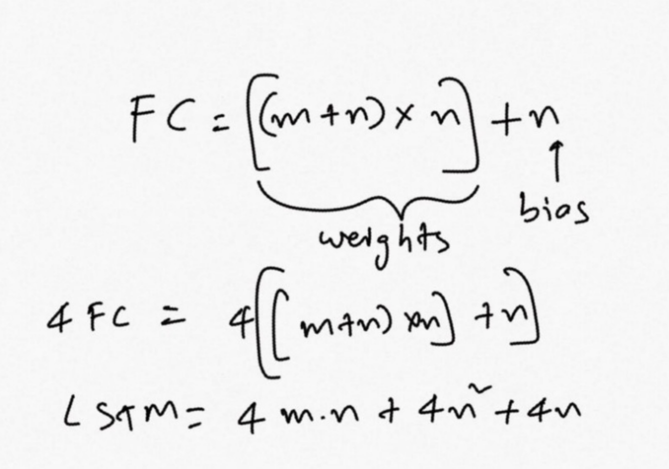
$$[h_{t-1}, x_t]$$



To calculate number of parameters. Each FC network has a parameter **weight** matrix of **[(m+n),n]** and a **bias** values of ’**n**’. So total parameters at each FC network (m*n + sqr(n) + n) and for four FC networks it will be **4*(m*n + sqr(n) + n).**

Parameter calculation





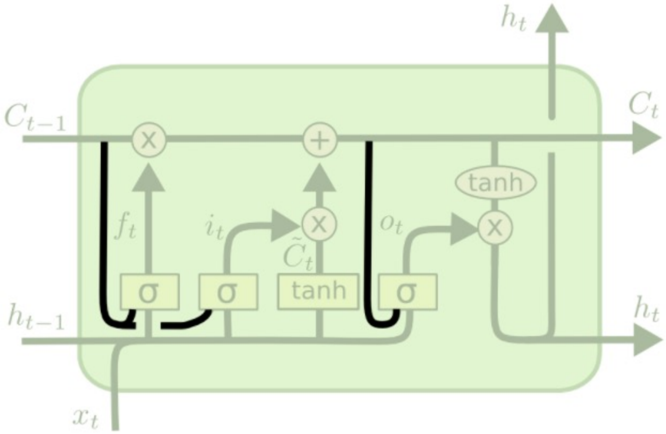
$$FC = \underbrace{\left[(m+n) \times n\right]}_{\text{weights}} + \overset{\text{bias}}{n}$$

$$4FC = 4\left(\left[m+n) \times n\right] + n\right)$$

$$LSTM = 4\,m \cdot n + 4n^2 + 4n$$

Variants of LSTMs-1: Peephole connections

- Adding peephole connections

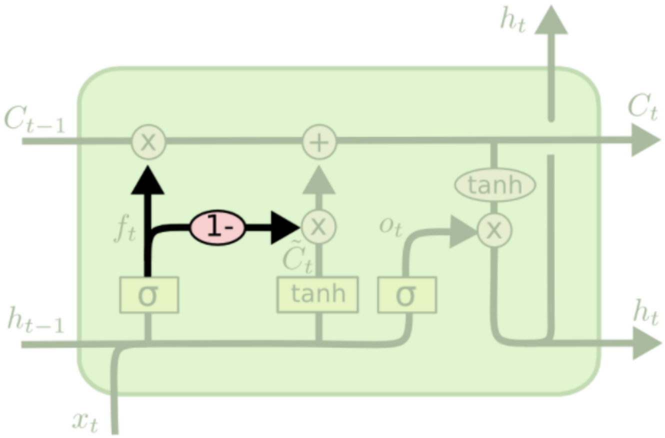


$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

We let the gate layers look at the cell state.

Variants of LSTMs-2: Coupled gates

- Coupled forget and input gates



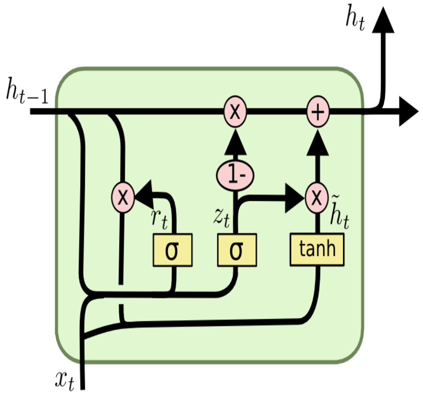
$$C_t = f_t * C_{t-1} + (1 - \boldsymbol{f_t}) * \tilde{C}_t$$

We only forget when we're going to input something in its place.

Variants of LSTMs-3: Gated Recurrent Unit (GRU)

- GRU is a variation of LSTM that also adopts the gated design.
- Differences:
  - GRU uses an **update gate $\boldsymbol{z}$** to substitute the input and forget gates i_t and f_t
  - Combined the cell state C_t and hidden state h_t in LSTM as a single cell state h_t

- GRU obtains similar performance compared to LSTM with fewer parameters and faster convergence. (Cho et al. 2014)



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

Update gate: controls the composition of the new state

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

Reset gate: determines how much old information is needed in the alternative state $\widetilde{h}_t$

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

Alternative state: contains new information

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

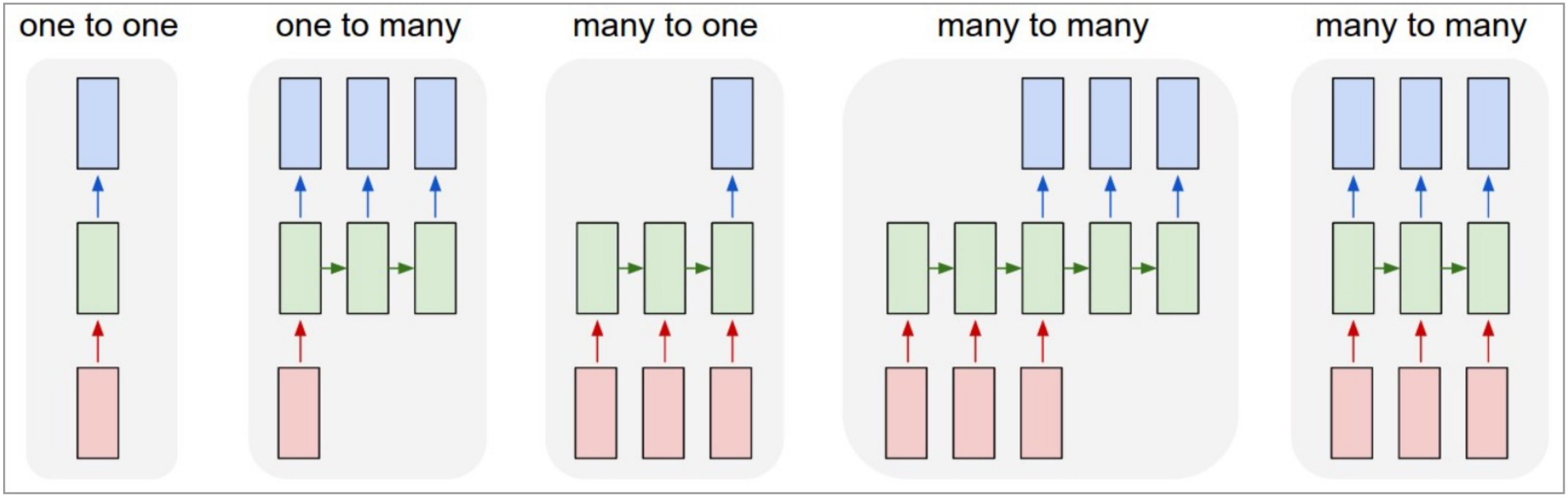
New state: replace selected old information with new information in the new state

Sequence Learning Architectures

- Learning on RNN is more robust when the vanishing/exploding gradient problem is resolved.
  - RNNs can now be applied to different Sequence Learning tasks.

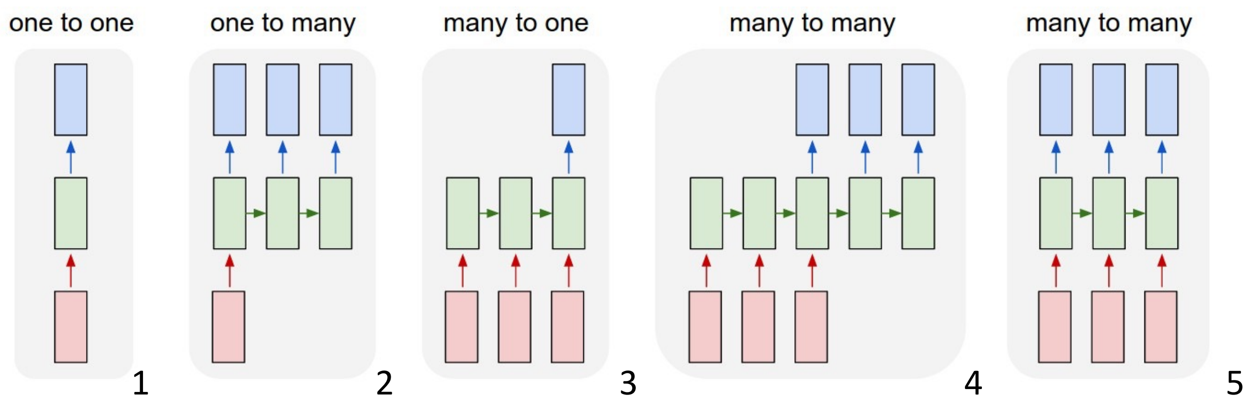
- Recurrent NN architecture is flexible to operate over various sequences of vectors.
  - Sequence in the input, the output, or in the most general case both
  - Architecture with one or more RNN layers

RNN use-cases



no pre-specified constraints on the lengths sequences because the recurrent transformation (green) is fixed and can be applied as many times as needed

Sequence Learning with One RNN Layer

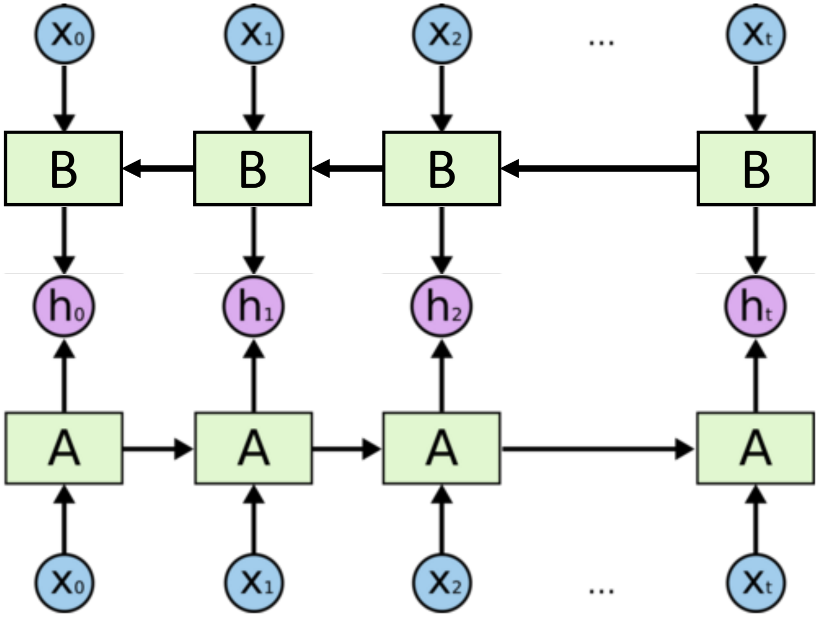


- Each rectangle is a vector and arrows represent functions (e.g. matrix multiply).
- Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state

(1) Standard NN mode without recurrent structure (e.g. **image classification, one label for one image**).
(2) Sequence output (e.g. **image captioning, takes an image and outputs a sentence of words**).
(3) Sequence input (e.g. **sentiment analysis, a sentence is classified as expressing positive or negative sentiment**).
(4) Sequence input and sequence output (e.g. **machine translation, a sentence in English is translated into a sentence in French**).
(5) Synced sequence input and output (e.g. **video classification, label each frame of the video**).

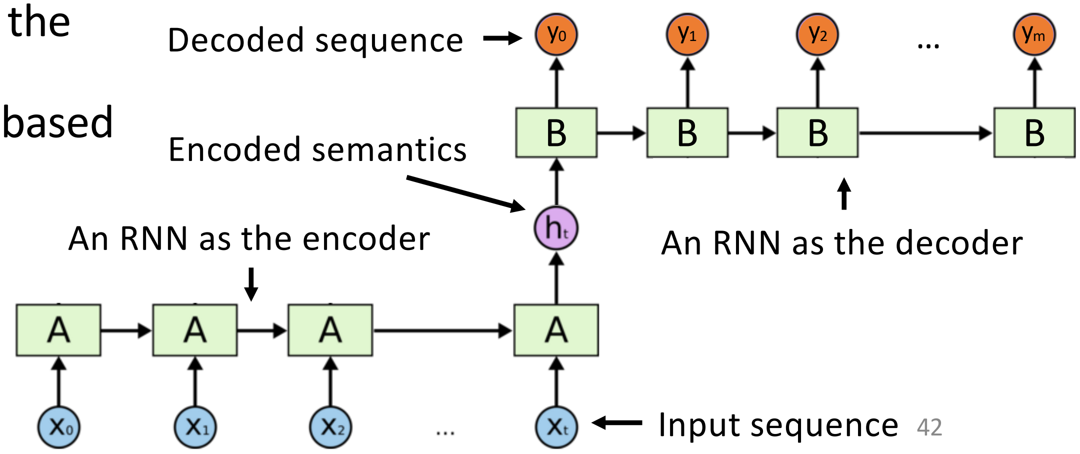
Sequence Learning with Multiple RNN Layers

- Bidirectional RNN
  - Connects two recurrent units (synced many-to-many model) of opposite directions to the same output.
  - Captures forward and backward information from the input sequence
  - Apply to data whose current state (e.g., h_0) can be better determined when given future information (e.g., $x_1, x_2, \ldots, x_t$)
    - E.g., in the sentence “the bank is robbed,” the semantics of “bank” can be determined given the verb “robbed.”

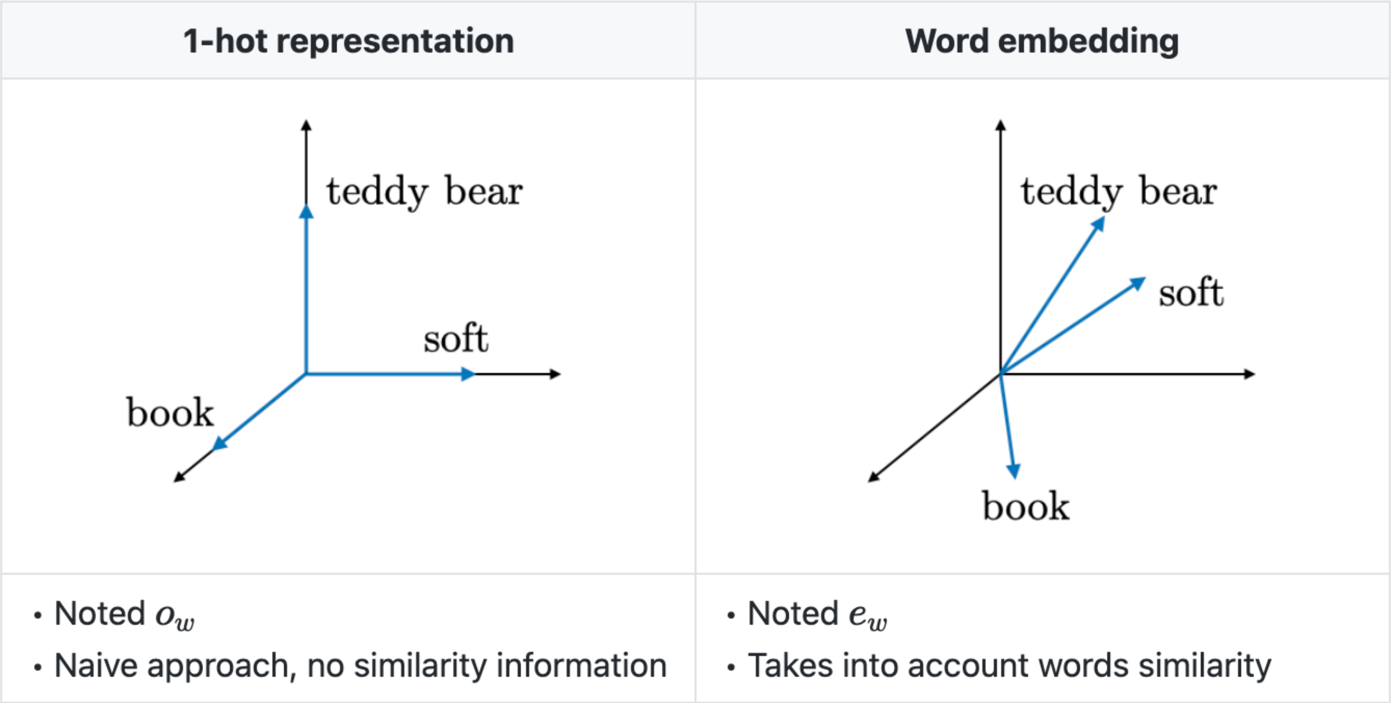


Sequence Learning with Multiple RNN Layers

- **Sequence-to-Sequence (Seq2Seq) model**
  - Developed by Google in 2018 for use in machine translation.
  - Seq2seq turns one sequence into another sequence. It does so by use of a recurrent neural network (RNN) or more often LSTM or GRU to avoid the problem of vanishing gradient.
  - The primary components are one Encoder and one Decoder network. The encoder turns each item into a corresponding hidden vector containing the item and its context. The decoder reverses the process, turning the vector into an output item, using the previous output as the input context.
  - **Encoder RNN**: extract and compress the semantics from the input sequence
  - **Decoder RNN**: generate a sequence based on the input semantics
  - Apply to tasks such as machine translation
    - Similar underlying semantics
    - E.g., "I love you." to "Je t'aime."

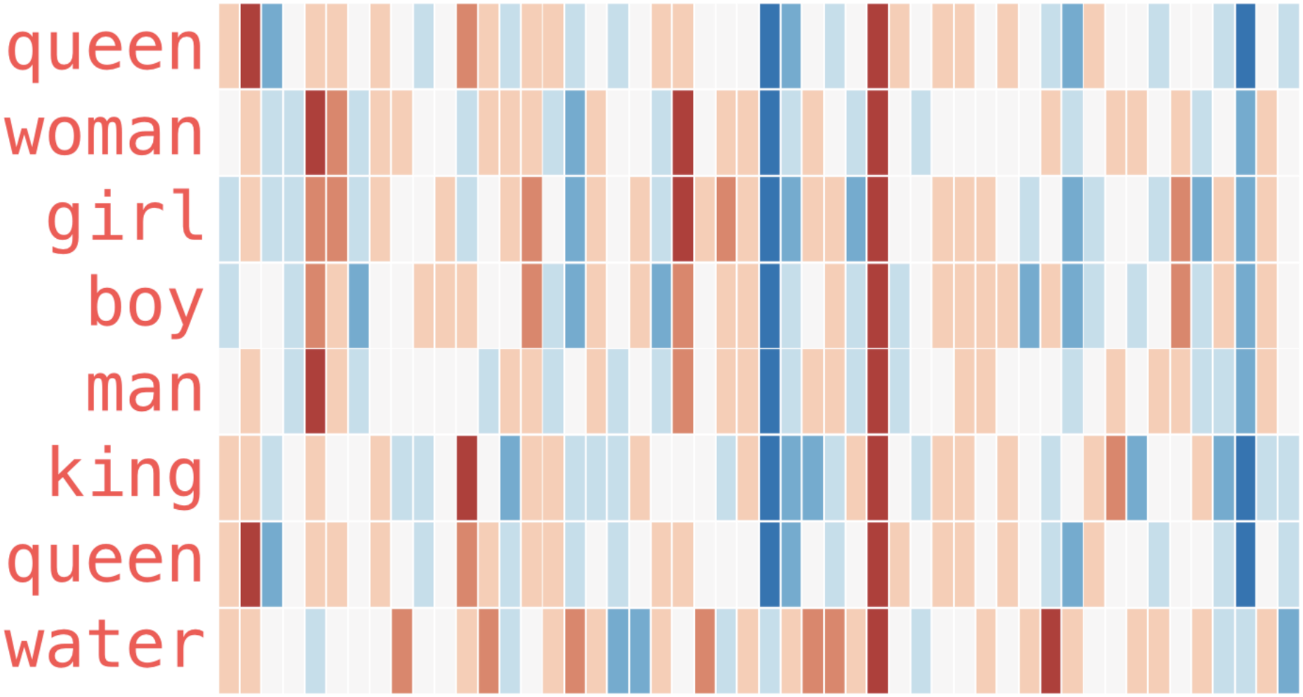


Word Embeddings


teddy bear
soft
book
teddy bear
soft
book

1-hot representation	Word embedding
• Noted o_w • Naive approach, no similarity information	• Noted e_w • Takes into account words similarity

Word2Vec



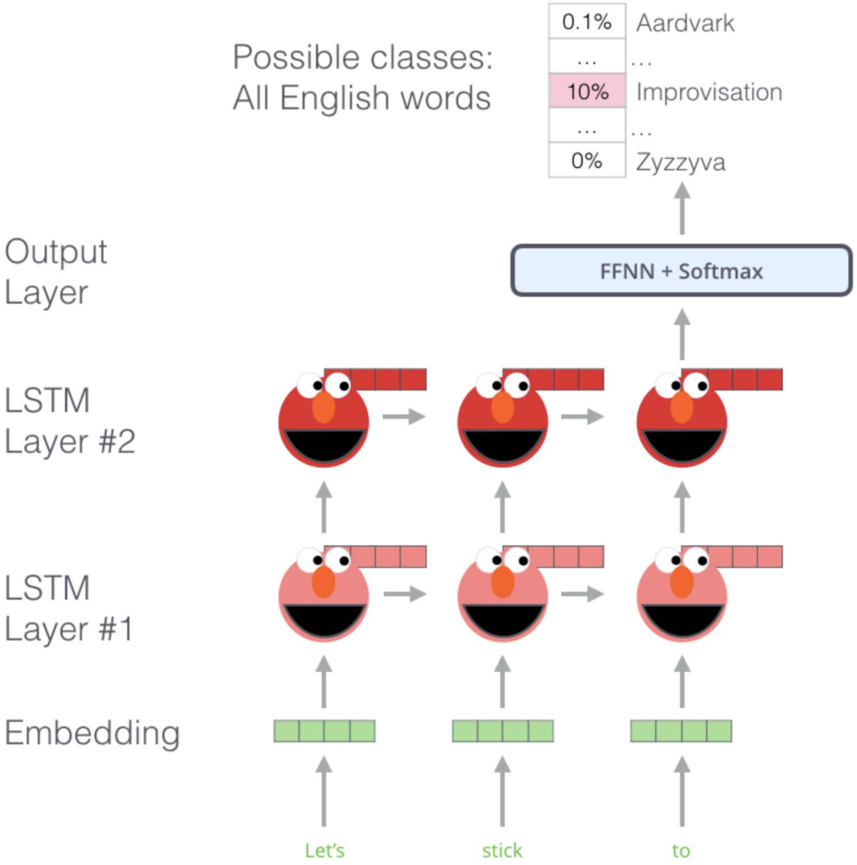
[The Illustrated Word2vec](https://jalammar.github.io/illustrated-word2vec/)

ELMo: Context Matters

- A word can have different meanings depending on the context
  - Give me the stick
  - Let’s stick to improving our work
- ELMo generates *contextualized* embeddings for a word
- Contextualized word-embeddings can give words different embeddings based on the meaning they carry in the context of the sentence
- LSTM based architecture

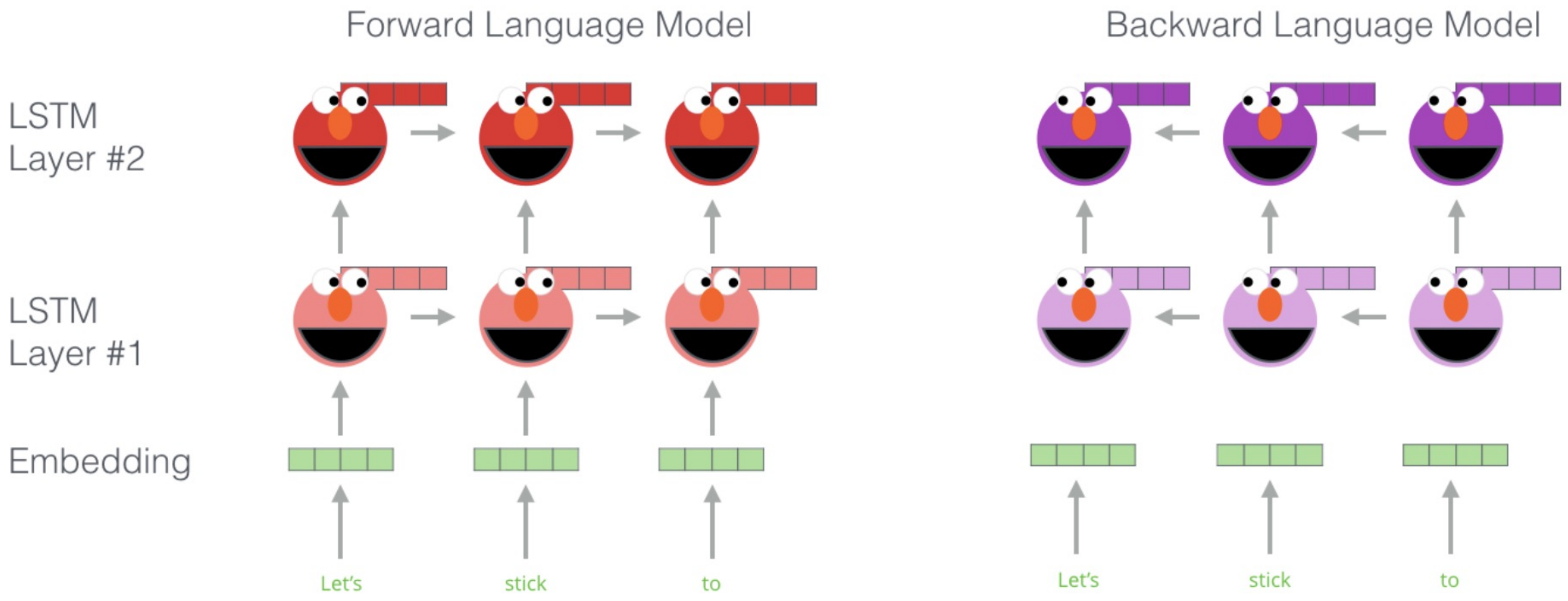
ELMo Training

- Predict next word in a sequence of words
- Self-supervised learning, without any need for labels



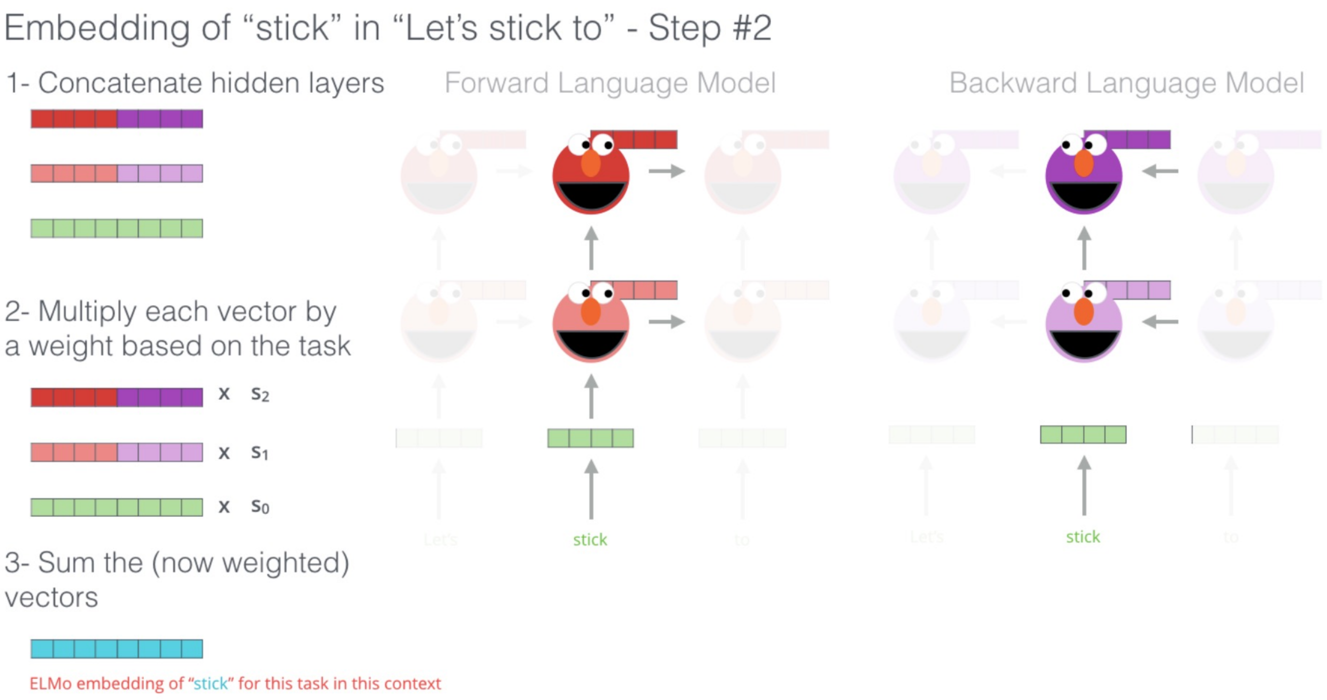
Embedding generation in ELMo: Step 1

Embedding of “stick” in “Let’s stick to” - Step #1



Embedding generation in ELMo: Step 2

- Contextualized embedding through grouping together the hidden states (and initial embedding)



Embedding of "stick" in "Let's stick to" - Step #2

1- Concatenate hidden layers

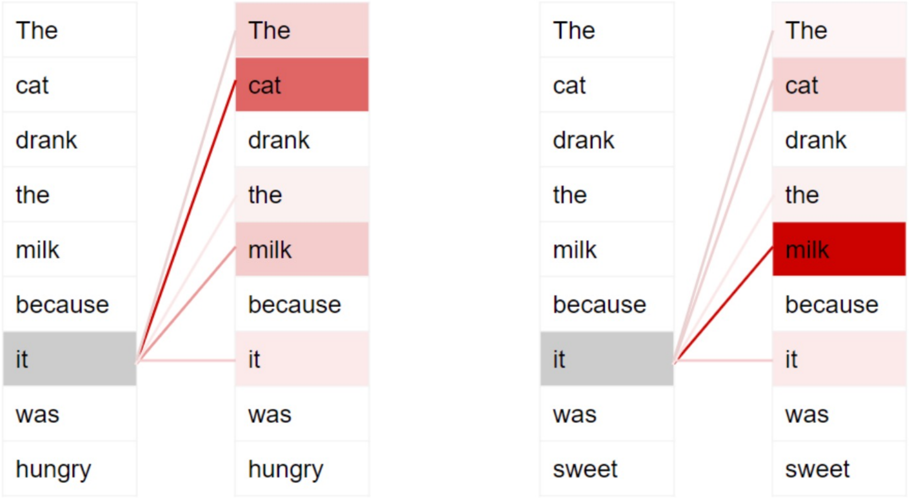
2- Multiply each vector by a weight based on the task

3- Sum the (now weighted) vectors

ELMo embedding of "stick" for this task in this context

Attention

- Capture long-range dependence in context
- Example:
  1. The *cat* drank the milk because **it** was hungry.
  2. The cat drank the *milk* because **it** was sweet.

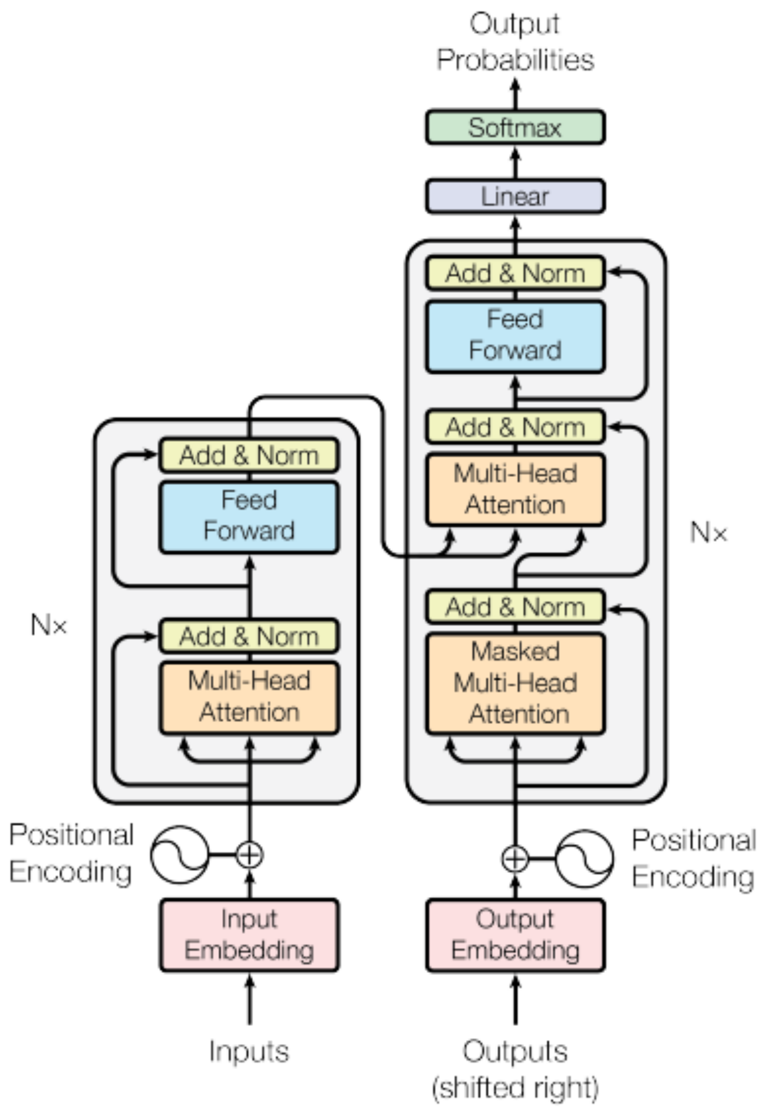


Deep Dive into Attention Mechanism

- Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)

Transformer

- Attention is all you need
- Encoder—Decoder architecture
- The Illustrated Transformer



Multi-Head Attention

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

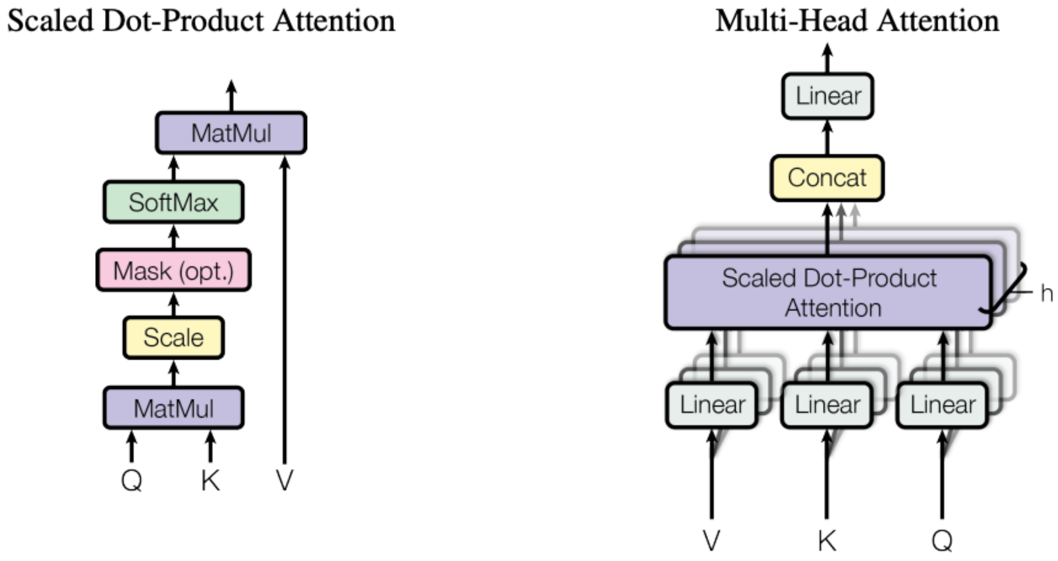


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

Attention in Transformer models

- The Transformer uses multi-head attention in three different ways:
- In "encoder-decoder attention" layers, the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence. This mimics the typical encoder-decoder attention mechanisms in sequence-to-sequence models.
- The encoder contains self-attention layers. In a self-attention layer all of the keys, values and queries come from the same place, in this case, the output of the previous layer in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder.
- Similarly, self-attention layers in the decoder allow each position in the decoder to attend to all positions in the decoder up to and including that position. We need to prevent leftward information flow in the decoder to preserve the auto-regressive property. We implement this inside of scaled dot-product attention by masking out (setting to $-\infty$) all values in the input of the softmax which correspond to illegal connections.

Deep Dive into Transformer

- The Illustrated Transformer – Part2