CS 291A: Deep Learning for NLP

Neural Networks: LSTMs and GRUs

William Wang
UCSB Computer Science
william@cs.ucsb.edu

Slides adapted from Y. V. Chen and B. Ramsundar.

Any questions about HW1 or project?

Agenda Today

1. A quick intro to TensorFlow and Keras.

2. Advanced RNNs --- LSTMs and GRUs.

3. Four paper presentations today (Time keeping will be strictly enforced: 12 mins presentation and 3 mins QA).

Deep Learning Toolkit

(Py)Torch

Tensorflow (Keras)

Caffe

Theano (Keras, Lasagne)

CNTK

CuDNN

Mxnet

DyNet

etc.

Tool Design

Model specification

- Configuration file
  - caffe, CNTK, etc
- Programmatic generation
  - Torch, Theano, TensorFlow

High-level language

- Lua
  - Torch
- Python
  - Theano, Torch, TensorFlow

Introduction

TensorFlow is an open source software library for machine intelligence developed by Google

- Provides primitives for *defining functions on tensors* and *automatically computing their derivatives*

Prerequisite: Python 2.7/3.3+ & numpy

What is a Tensor?

Definition

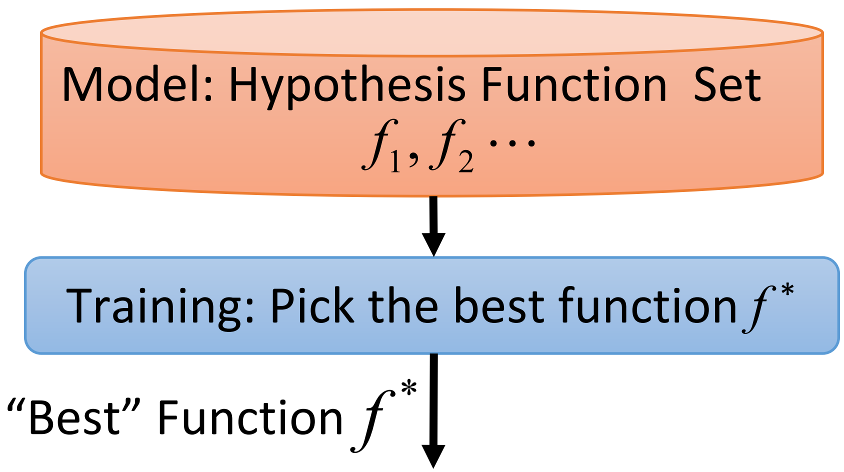
- Tensors are multilinear maps from vector spaces to the real numbers → n-dimensional arrays

Example

- Scalar $f : \mathbb{R} \to \mathbb{R}, f(e_1) = c$
- Vector $f : \mathbb{R}^n \to \mathbb{R}, f(e_i) = v_i$
- Matrix $f : \mathbb{R}^n \to \mathbb{R}^m, f(e_i, e_j) = M_{ij}$

Deep learning process is flows of tensors → a sequence of tensor operations

Deep Learning Framework



Q1. What is the model? **Model Architecture**

Q2. What does a "good" function mean? **Loss Function Design**

Q3. How do we pick the "best" function? **Optimization**

After defining the model and loss function, TensorFlow automatically computes the gradient for optimization

Sample TensorFlow Program

```
import tensorflow as tf
import numpy as np
```

Import the APIs

```
x_data = np.random.rand(100).astype(np.float32)
y_data = x_data * 0.1 + 0.3
```

Create 100 phony x, y data points in NumPy, y = x * 0.1 + 0.3

```
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b
```

Try to find values for W and b that compute y_data = W * x_data + b (W should be 0.1 and b 0.3)

```
loss = tf.reduce_mean(tf.square(y - y_data))
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)
```

Minimize the mean squared errors.

```
init = tf.initialize_all_variables()
```

Initialize the variables.

```
sess = tf.Session()
sess.run(init)
```

Launch the graph.

```
for step in range(201):
    sess.run(train)
    if step % 20 == 0:
        print(step, sess.run(W), sess.run(b))
```

Fit the line.

→ Learns best fit is W: [0.1], b: [0.3]

Basic Usage

Represents computations as graphs

Executes graphs in the context of `Sessions`

Represents data as tensors

Maintains state with `Variables`

Uses feeds and fetches to get data into and out of any operations

TensorFlow programs are usually structured into a *construction phase*, that assembles a graph, and an *execution phase* that uses a session to execute ops in the graph

Dual CNN-BiLSTM Network: Sample Keras Program

```
branch1 = Sequential()
branch1.add(Embedding(max_features, 128, input_length=maxlen))
branch1.add(Dropout(0.5))
branch1.add(Conv1D(filters,
            kernel_size,
            padding='valid',
            activation='relu',
            strides=1))
branch1.add(MaxPooling1D(pool_size=pool_size))
branch1.add(Bidirectional(LSTM(64)))

branch2 = Sequential()
branch2.add(Embedding(max_features, 128, input_length=maxlen))
branch2.add(Dropout(0.5))
branch2.add(Conv1D(filters,
            kernel_size,
            padding='valid',
            activation='relu',
            strides=1))
branch2.add(MaxPooling1D(pool_size=pool_size))
branch2.add(Bidirectional(LSTM(64)))

model = Sequential()
model.add(Merge([branch1, branch2], mode = 'mul'))
model.add(Dense(6, activation='softmax'))
```

Define branch 1
Embedding layer

Dropout
ConvNet layer
Max-pooling
BiLSTM

Define branch 2

Dropout
ConvNet layer
Max-pooling
BiLSTM

Merge two branches
Dense layer – softmax prediction

Vanishing Gradients

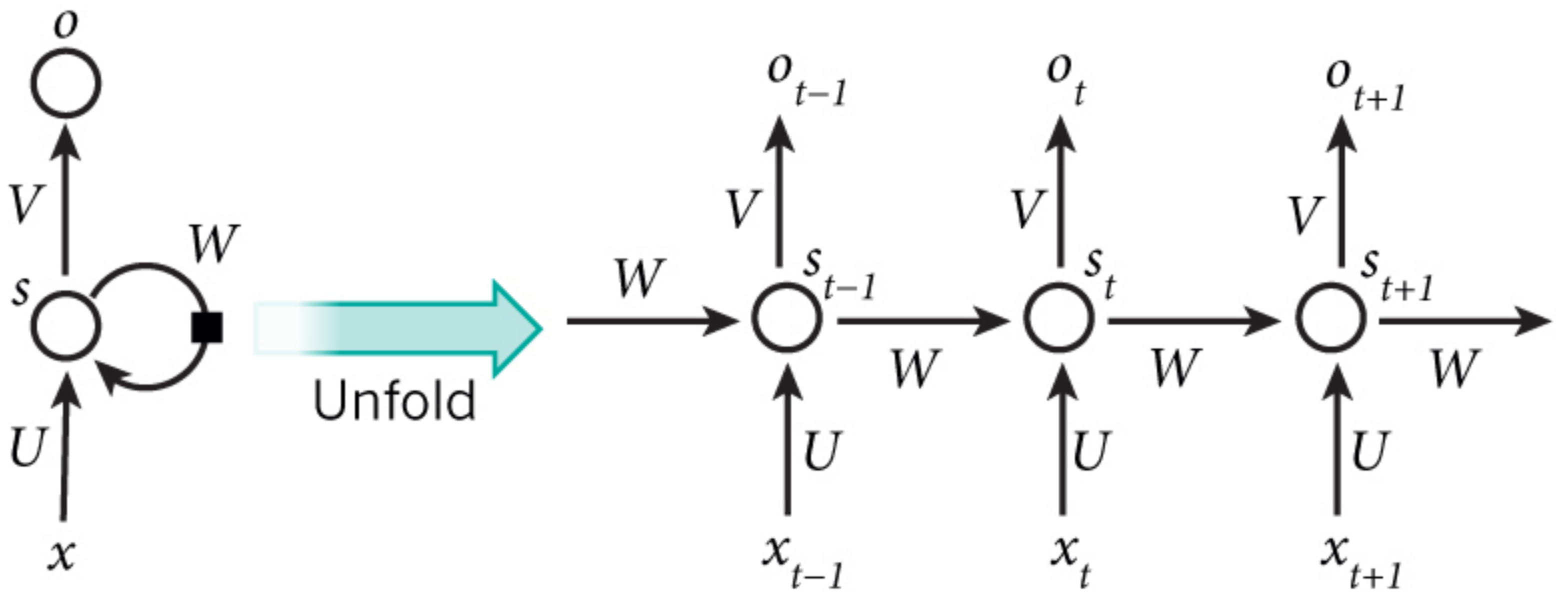
Vanishing Gradient Problem

Recurrent Neural Network Definition

$$s_t = \sigma(W s_{t-1} + U x_t)$$

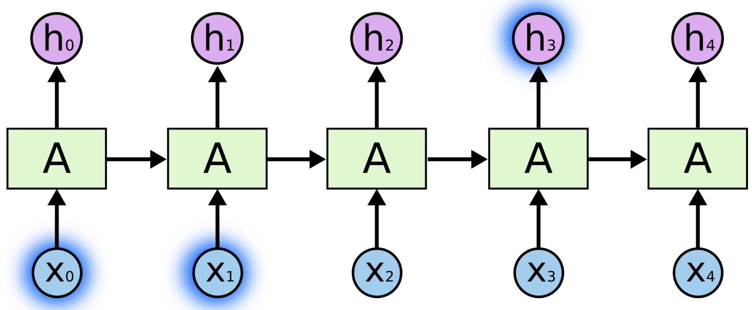
$$o_t = \mathrm{softmax}(V s_t)$$

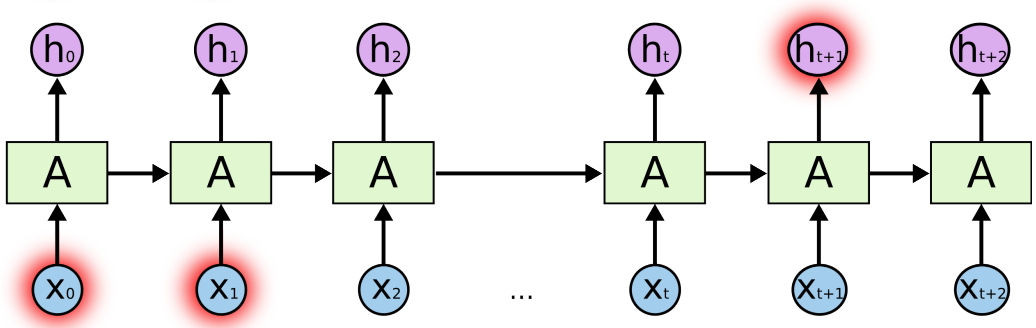
$\sigma(\cdot)$: tanh, ReLU



Vanishing Gradient: Gating Mechanism

RNN: keeps temporal sequence information





"I grew up in France…
I speak fluent *French*."

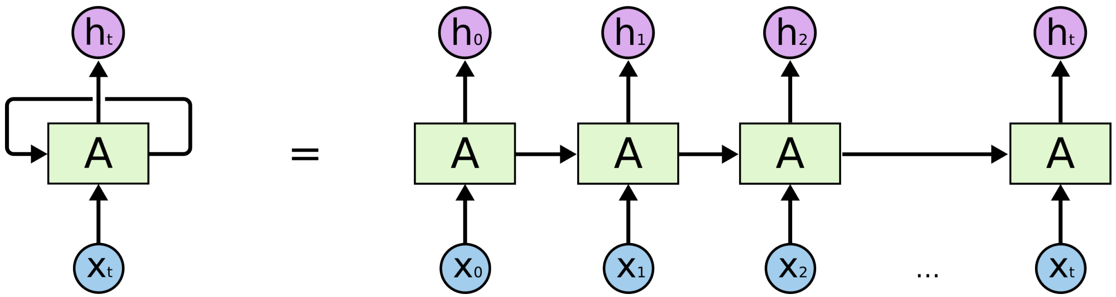
Issue: in theory, RNNs can handle such "long-term dependencies," but they cannot in practice
→ use gates to directly encode the long-distance information

Long Short-Term Memory

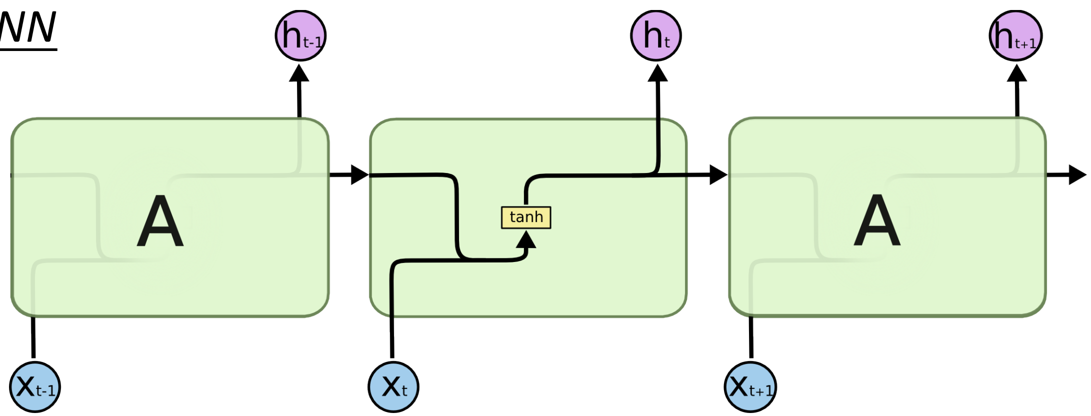
Addressing Vanishing Gradient Problem

Long Short-Term Memory (LSTM)

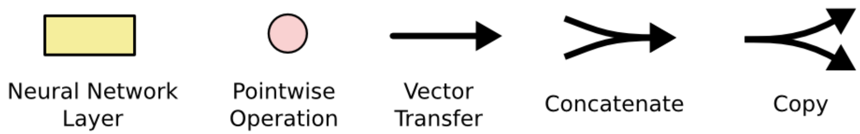
LSTMs are explicitly designed to avoid the long-term dependency problem



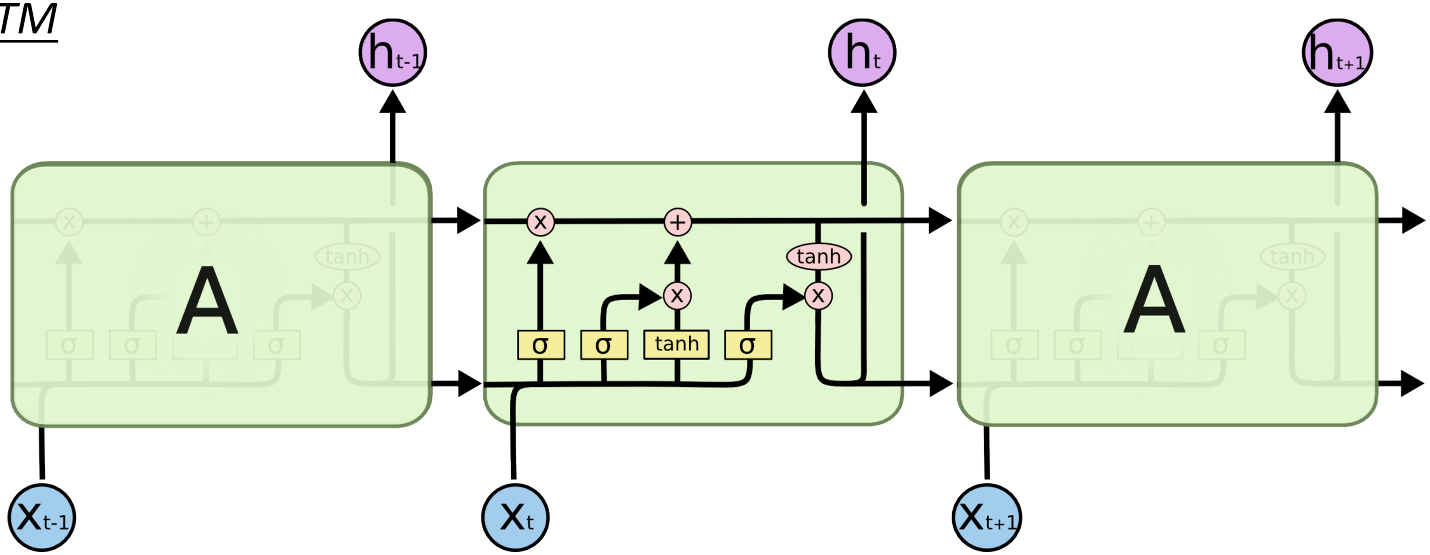
Vanilla RNN



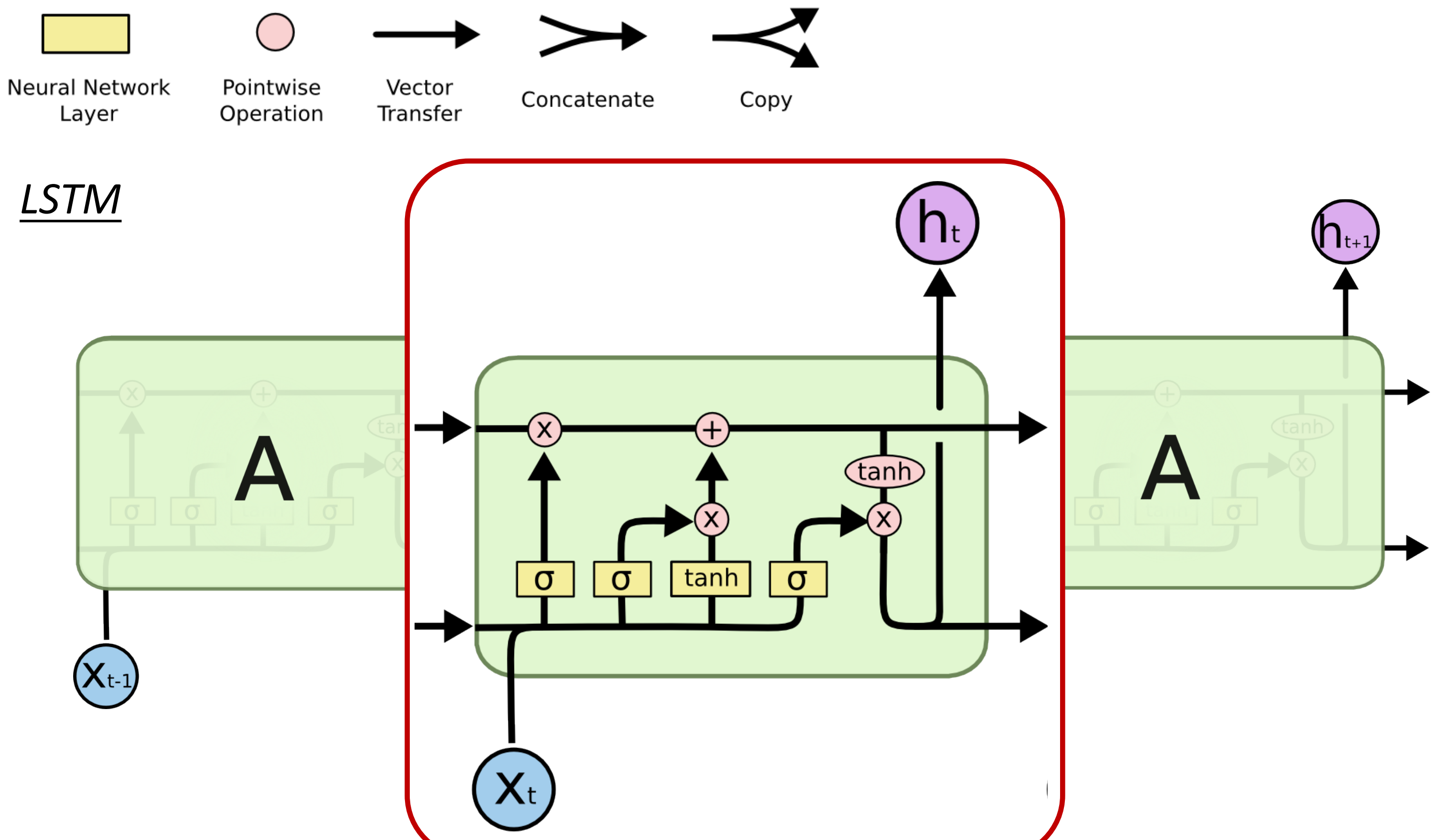
Long Short-Term Memory (LSTM)



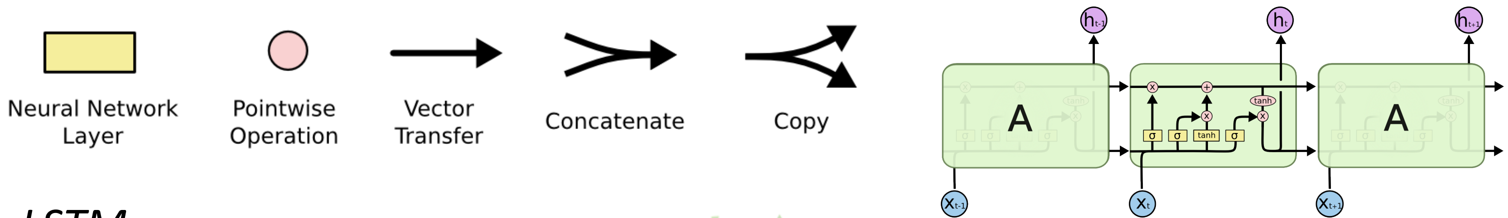
LSTM



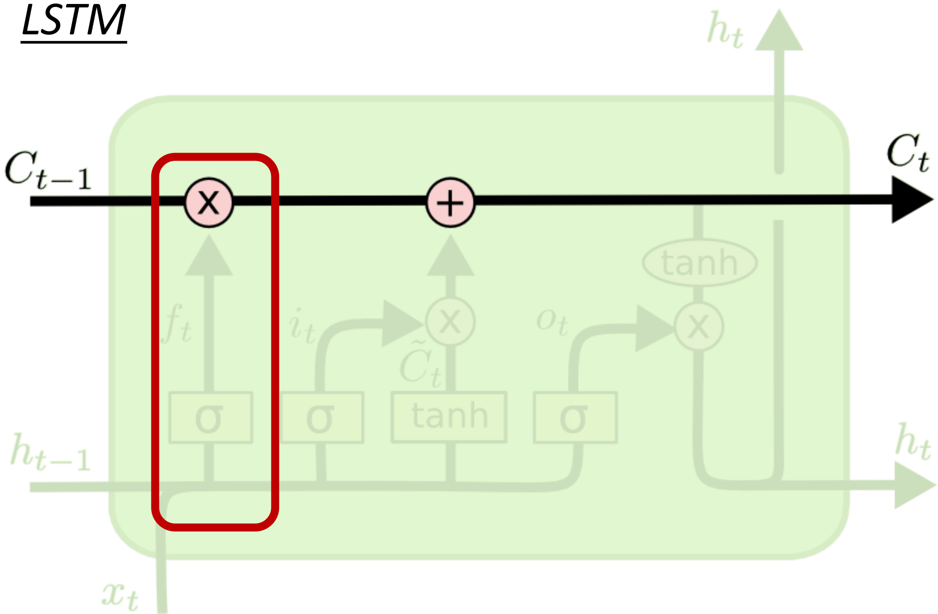
Long Short-Term Memory (LSTM)



Piazza Poll: if you are the designer of LSTM, which non-linear function would you choose for gates?



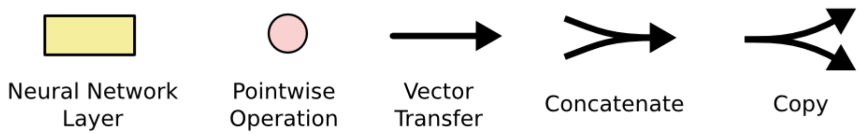
LSTM

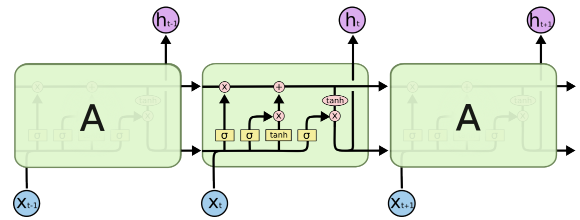


runs straight down the chain with minor linear interactions
→ easy for information to flow along it unchanged

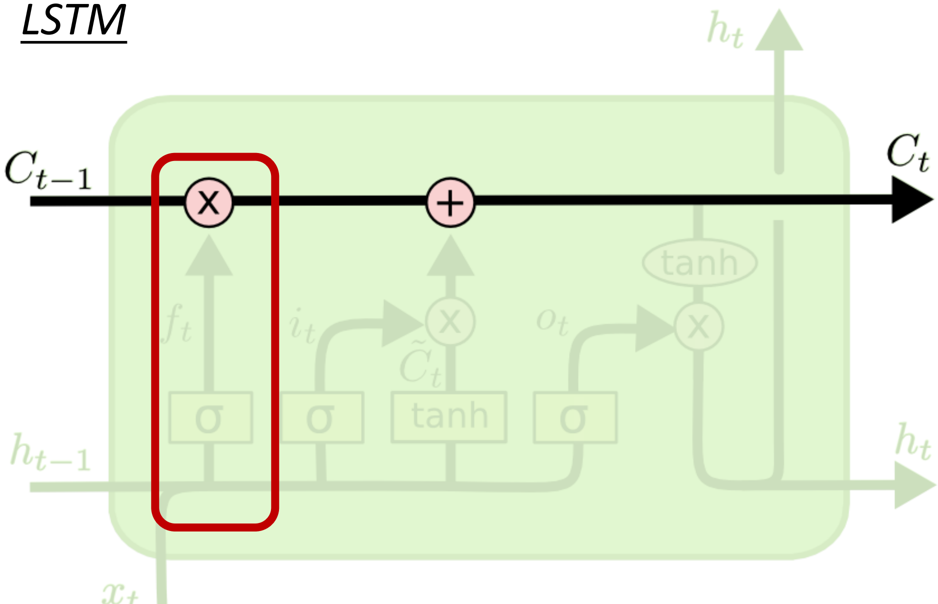
Gates are a way to optionally let information through

Long Short-Term Memory (LSTM)





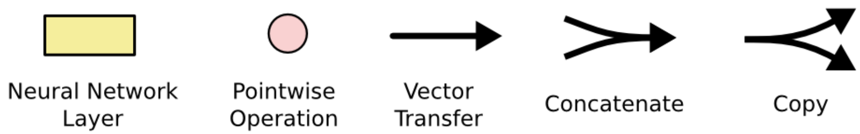
LSTM

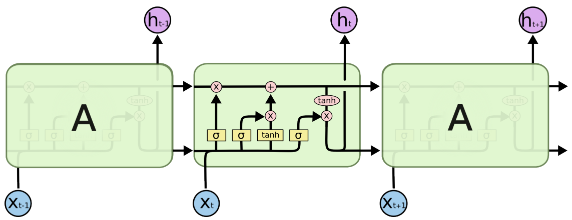


runs straight down the chain with minor linear interactions
→ easy for information to flow along it unchanged

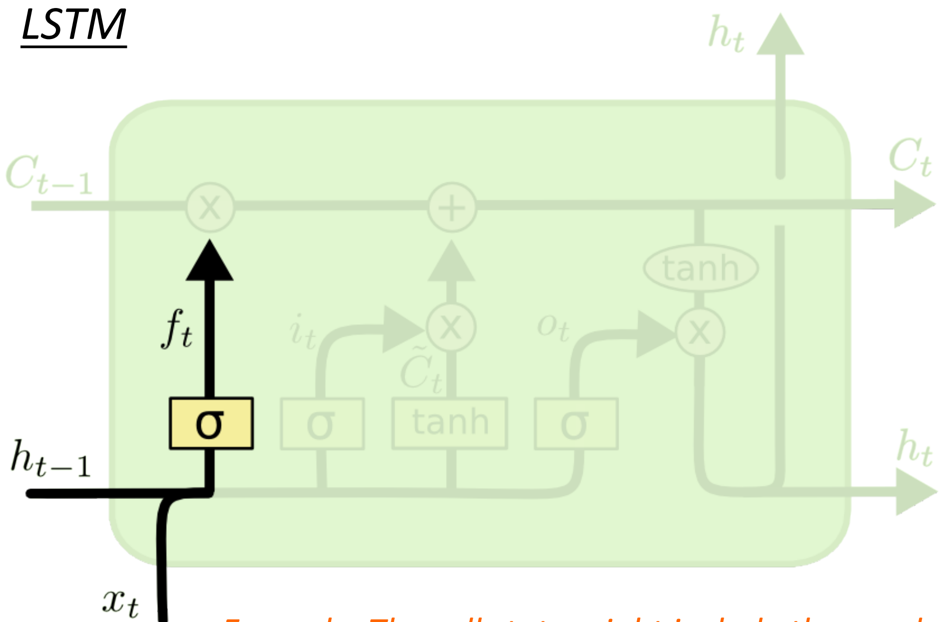
Gates are a way to optionally let information through
→ composed of a sigmoid and a pointwise multiplication operation

Long Short-Term Memory (LSTM)





LSTM



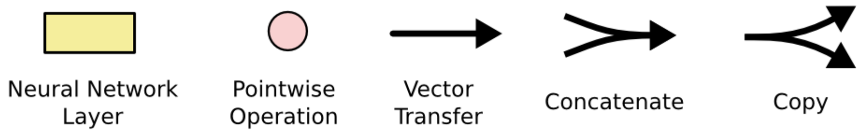
forget gate (a sigmoid layer): decides what information we're going to throw away from the cell state

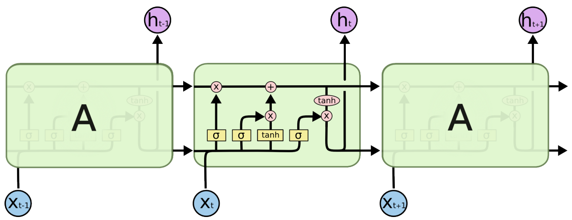
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

- 1: "completely keep this"
- 0: "completely get rid of this"

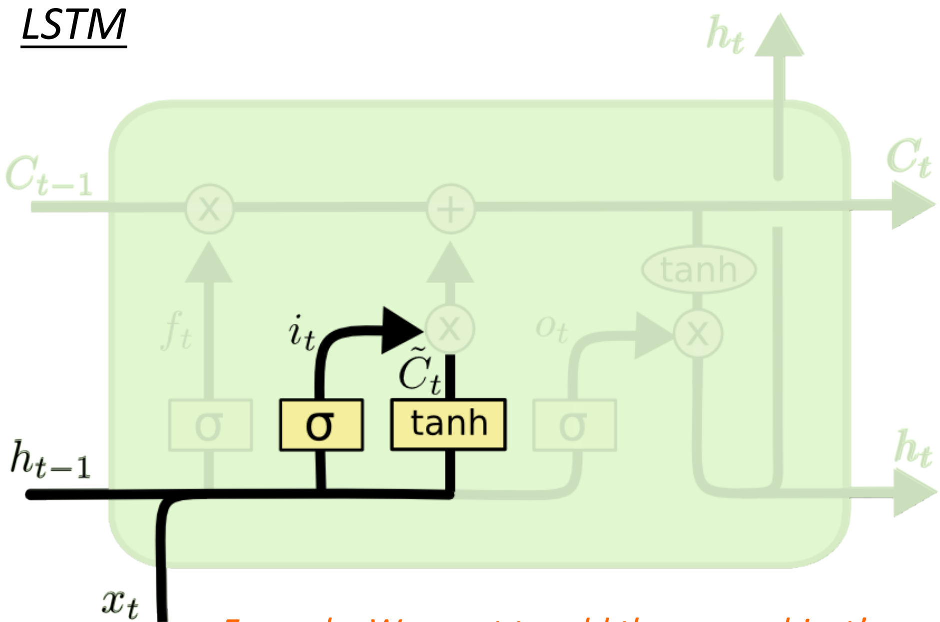
Example: The cell state might include the gender of the present subject, so that the correct pronouns can be used. When seeing a new subject, we want to forget the old subject's gender.

Long Short-Term Memory (LSTM)





LSTM



input gate (a sigmoid layer): decides what new information we're going to store in the cell state

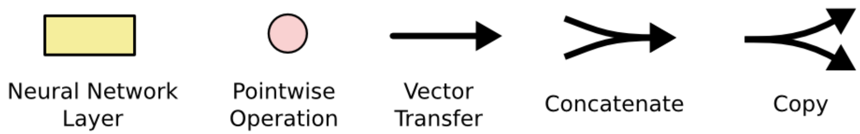
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

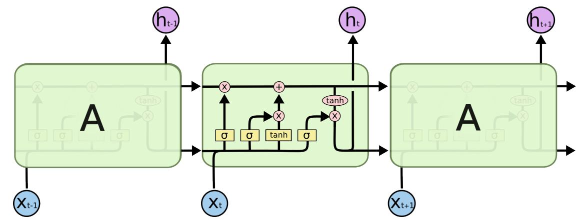
$$\boxed{\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)}$$

Vanilla RNN

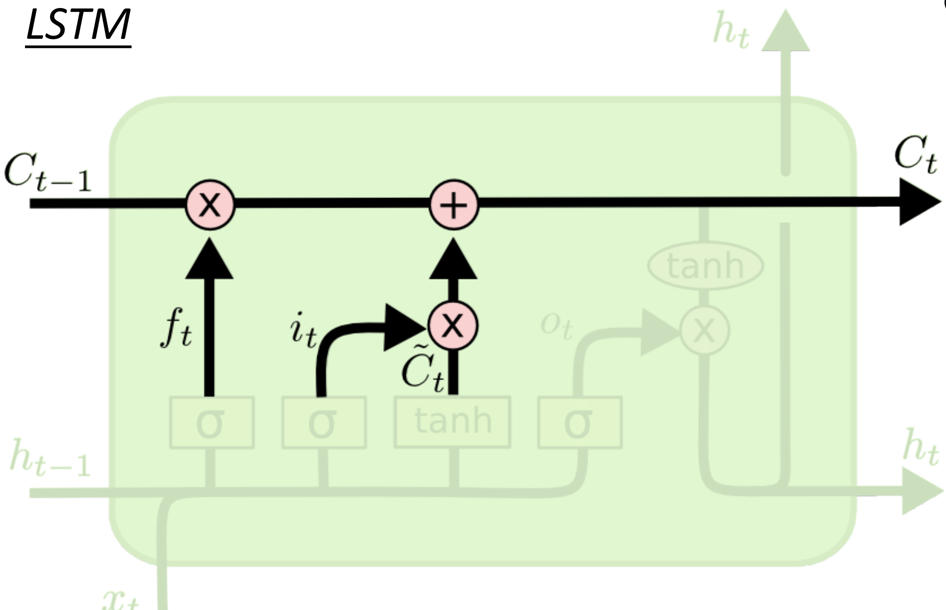
Example: We want to add the new subject's gender to the cell state for replacing the old one.

Long Short-Term Memory (LSTM)





LSTM



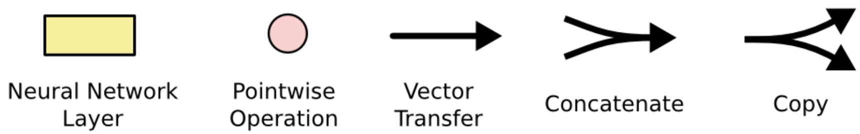
where we actually drop the information about the old subject's gender and add the new information

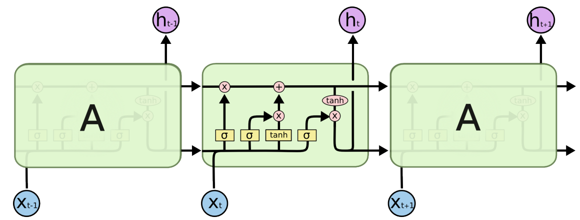
cell state update: forgets the things we decided to forget earlier and add the new candidate values, scaled by how much we decided to update each state value

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

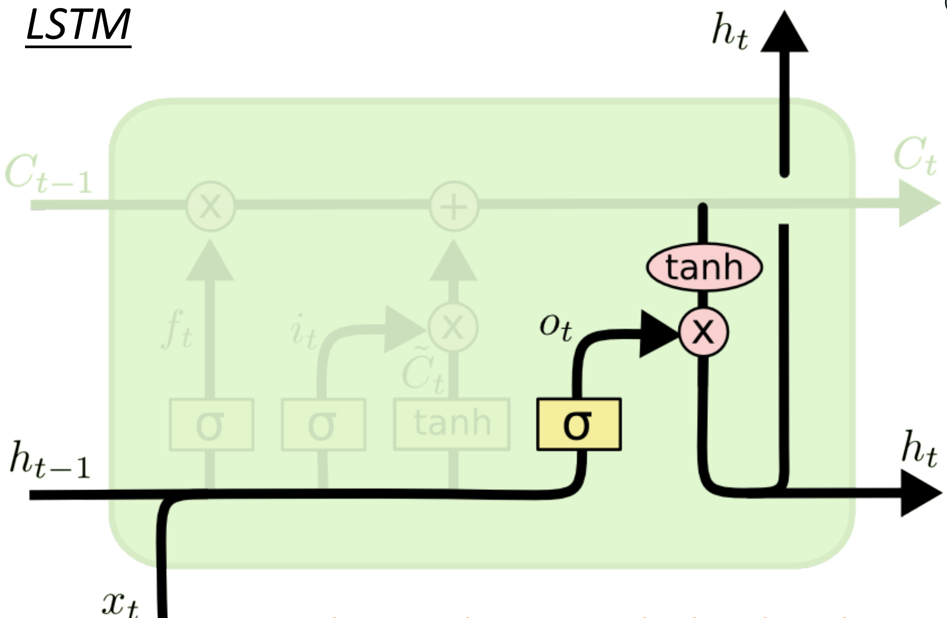
- f_t: decides which to forget
- i_t: decide which to update

Long Short-Term Memory (LSTM)





LSTM



output gate (a sigmoid layer): decides what new information we're going to output

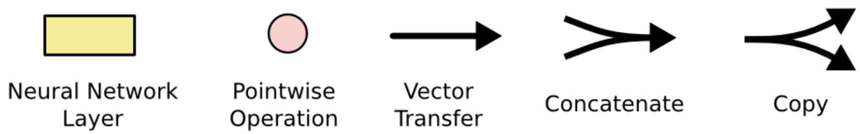
$$o_t = \sigma\left(W_o \left[h_{t-1}, x_t\right] + b_o\right)$$

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

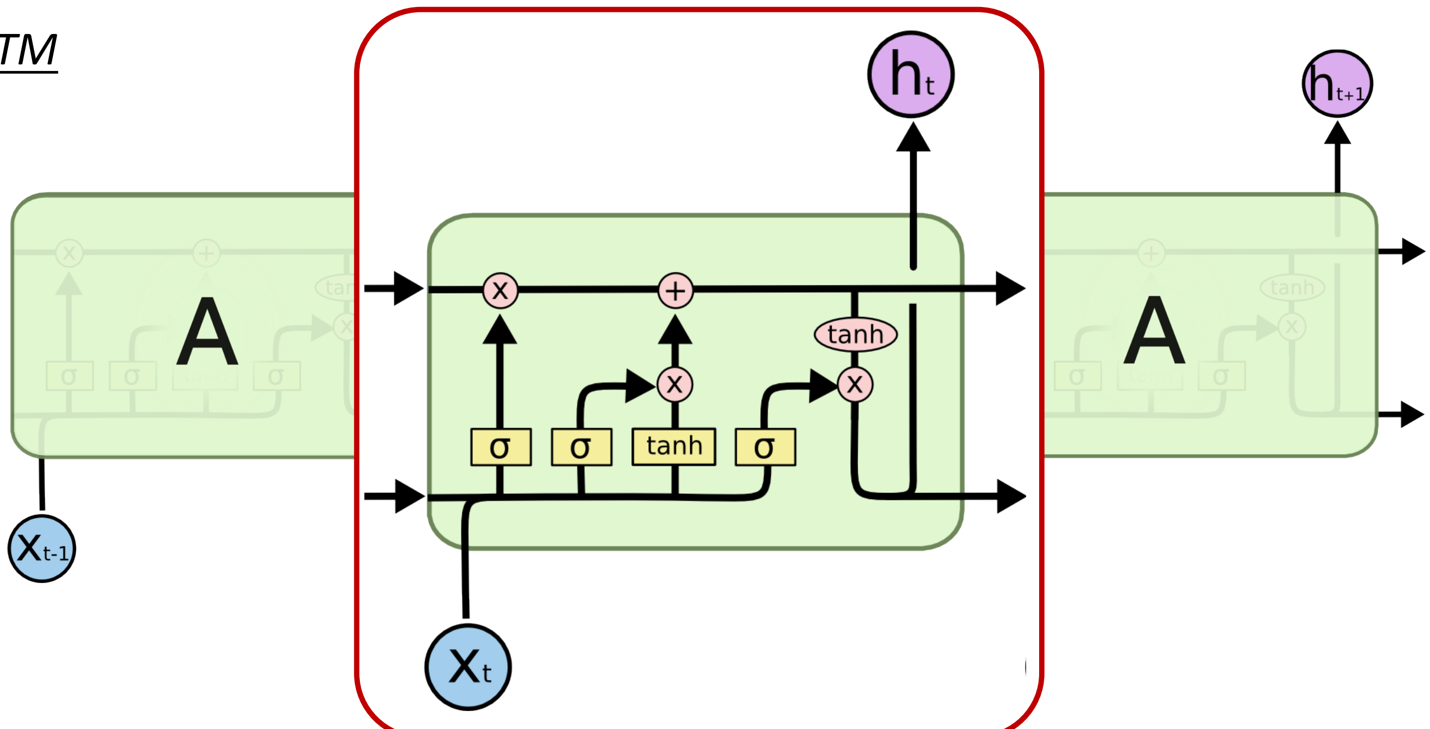
Example: It might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.

Hochreiter and Schmidhuber, "Long short-term memory," in Neural Computation, 1997. [link]

Question: Why can LSTMs prevent gradient vanishing / exploding?



LSTM



Hochreiter and Schmidhuber, "Long short-term memory," in Neural Computation, 1997. [link]

Why can LSTMs prevent the gradient vanishing / exploding issues?

1. Memory cells and gating units allow information to be stored for long periods of time.
2. Memory cells are additive in time
   1. Gradients also additive in time which alleviates vanishing gradient

Standard RNN

$$h_t = \sigma(w h_{t-1}).$$

$$\frac{\partial h_{t'}}{\partial h_t} = \prod_{k=1}^{t'-t} w\sigma'(w h_{t'-k})$$

$$= \underbrace{w^{t'-t}}_{!!!} \prod_{k=1}^{t'-t} \sigma'(w h_{t'-k})$$

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

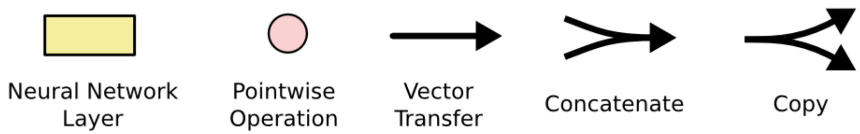
In LSTMs, if you take the partial derivation of $\frac{\partial C_t}{\partial C_{t-1}}$

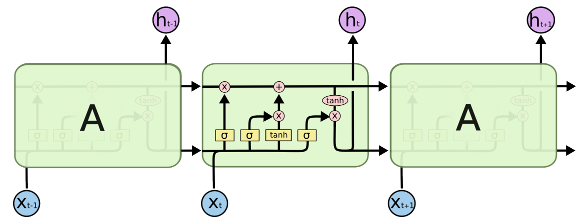
There's no w outside.

Variants on LSTM

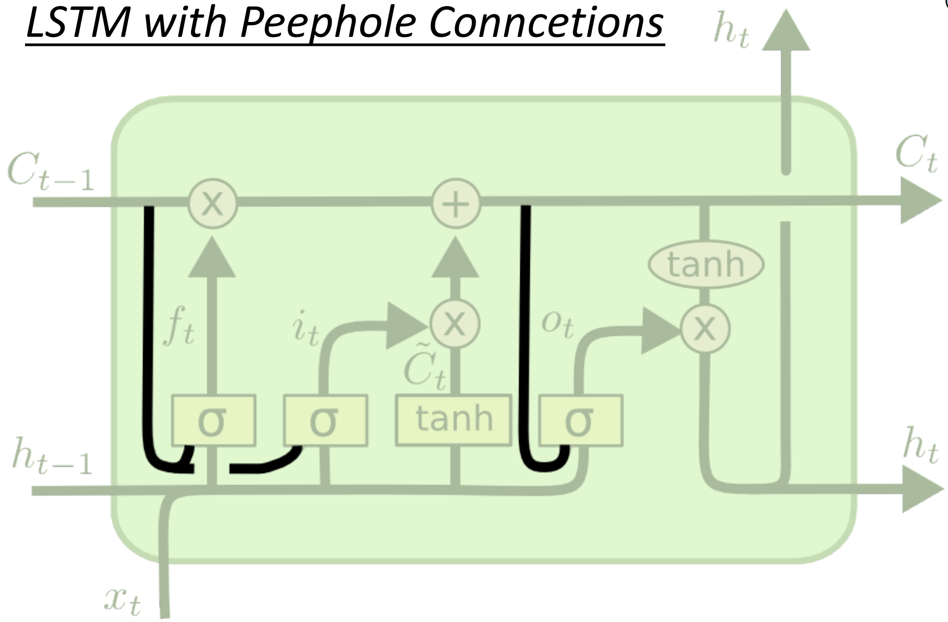
Addressing Vanishing Gradient Problem

LSTM with Peephole Connections





LSTM with Peephole Conncetions



Idea: allow gate layers to look at the cell state

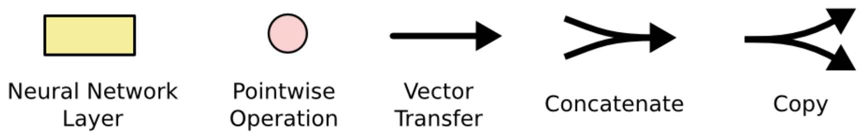
$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \ + \ b_f\right)$$

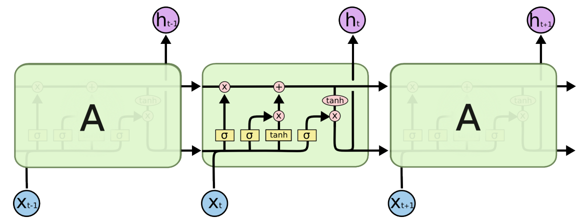
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \ + \ b_i\right)$$

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

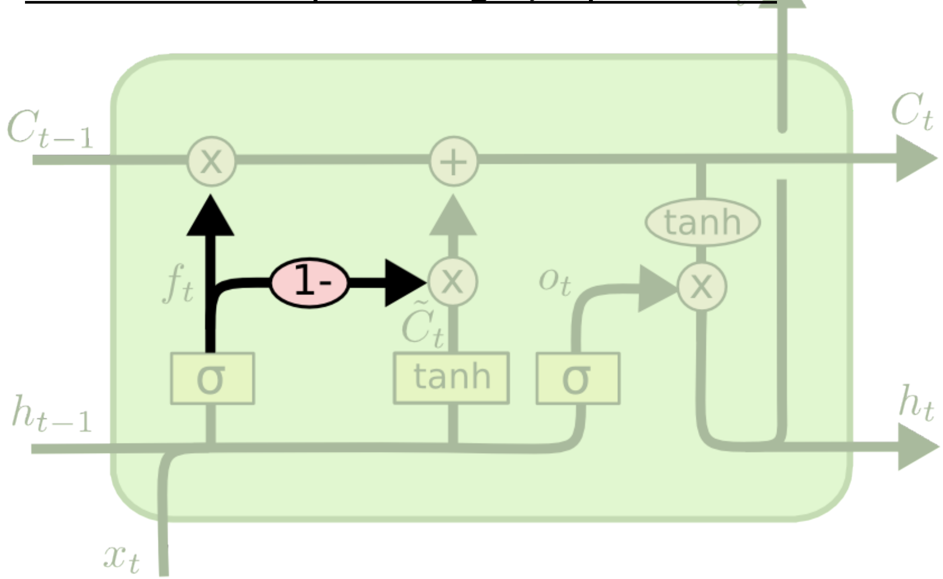
Gers and Schmidhuber, "Recurrent nets that time and count," in IJCNN, 2000. [link]

LSTM with Coupled Forget/Input Gates





LSTM with Coupled Forget/Input Gates



Idea: instead of separately deciding what to forget and what we should add new information to, we make those decisions together

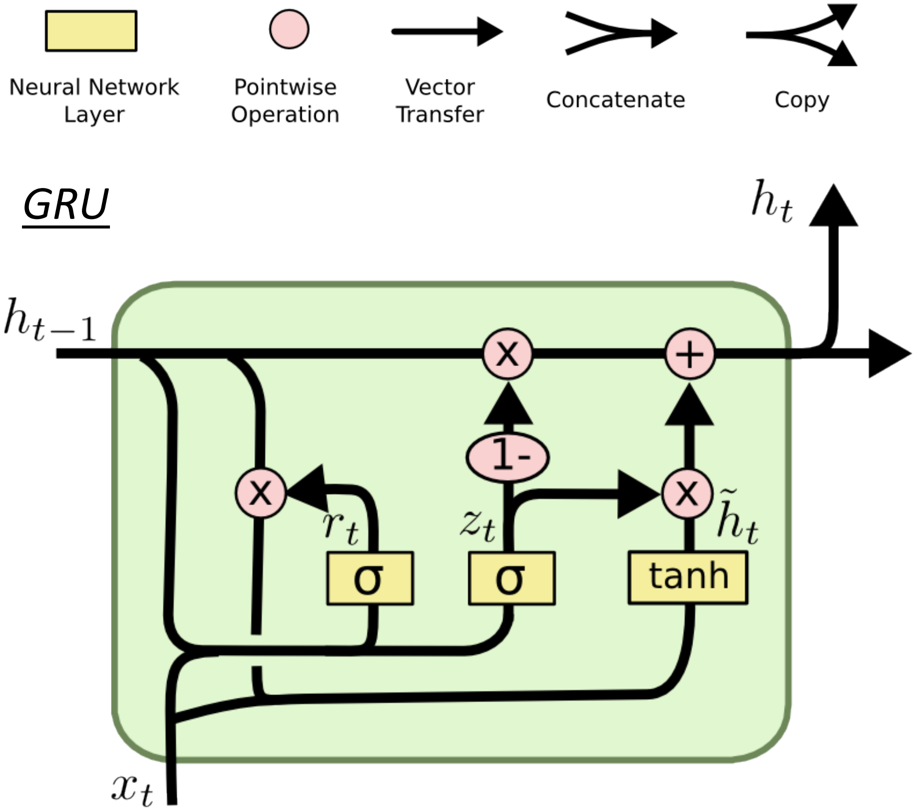
$$C_t = f_t * C_{t-1} + (1 - f_t) * \tilde{C}_t$$

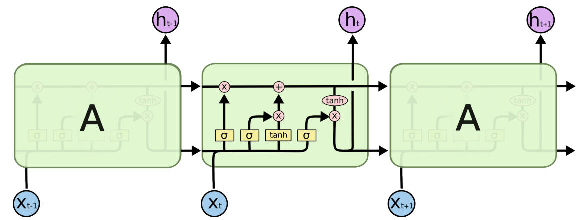
We only forget when we're going to input something in its place, and vice versa.

Gated Recurrent Unit

Addressing Vanishing Gradient Problem

Gated Recurrent Unit (GRU)





Idea: combine the forget and input gates into a single "update gate"; merge the cell state and hidden state

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

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

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

$r_t=0$: ignore previous memory and only stores the new word information

GRU is simpler and has less parameters than LSTM

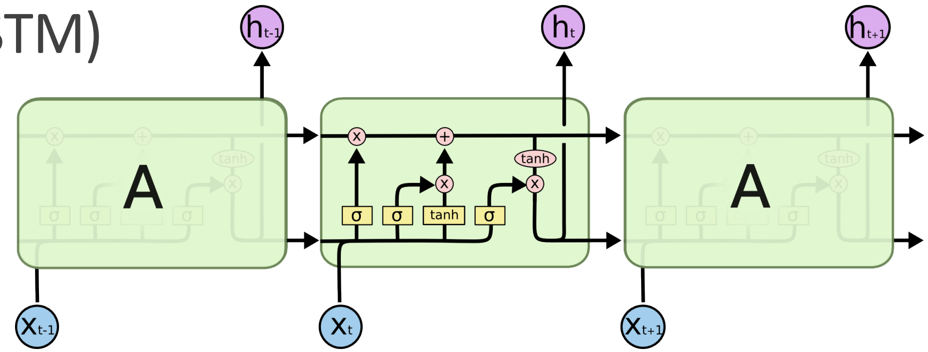
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

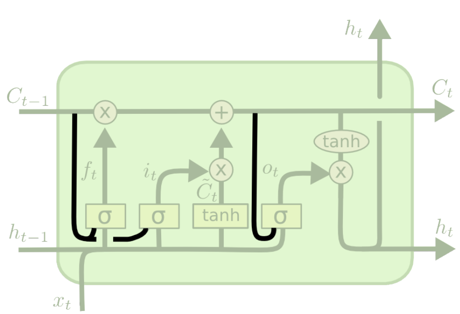
Concluding Remarks

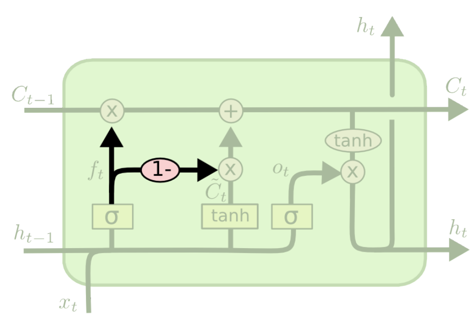
Gating mechanism for vanishing gradient problem

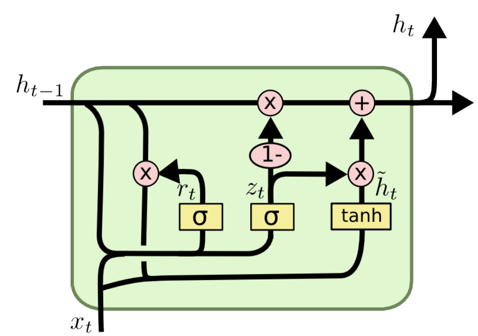
Gated RNN

- Long Short-Term Memory (LSTM)
  - Peephole Connections
  - Coupled Forget/Input Gates
- Gated Recurrent Unit (GRU)







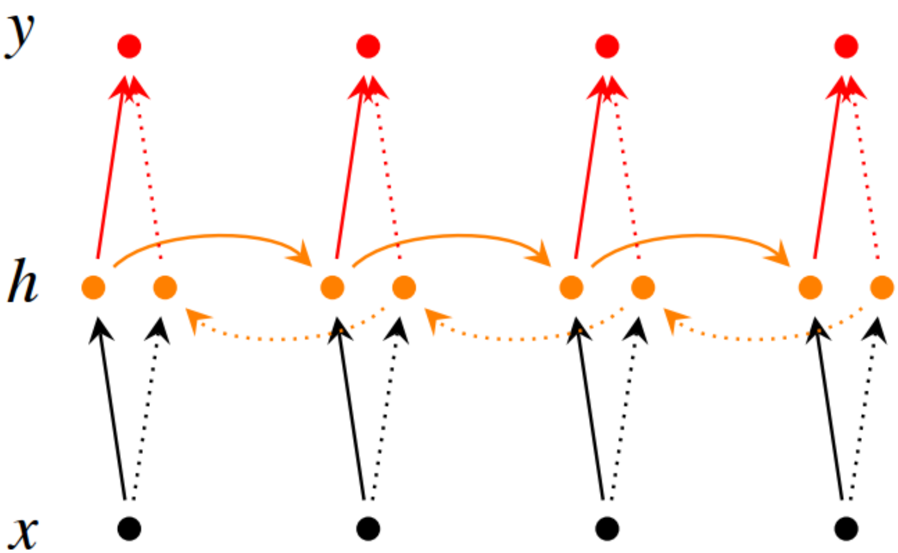


What are some issues with LSTMs?

Extension

Recurrent Neural Network

Bidirectional RNN



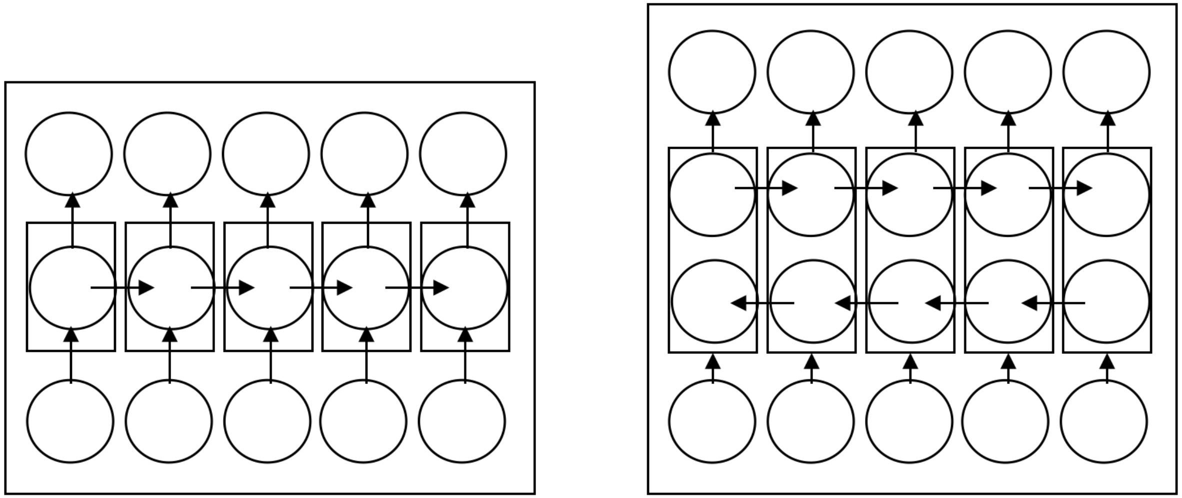
$$\overrightarrow{h}_t = f(\overrightarrow{W} x_t + \overrightarrow{V}\overrightarrow{h}_{t-1} + \overrightarrow{b})$$

$$\overleftarrow{h}_t = f(\overleftarrow{W} x_t + \overleftarrow{V}\overleftarrow{h}_{t+1} + \overleftarrow{b})$$

$$y_t = g(U[\overrightarrow{h}_t;\overleftarrow{h}_t] + c)$$

$h = [\overrightarrow{h}; \overleftarrow{h}]$ represents (summarizes) the past and future around a single token

How to train a bi-directional RNN model?

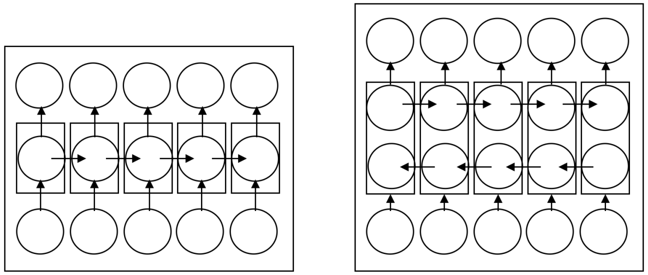


(a) (b)

Structure overview

(a) unidirectional RNN
(b) bidirectional RNN

How to train a bi-directional RNN model?

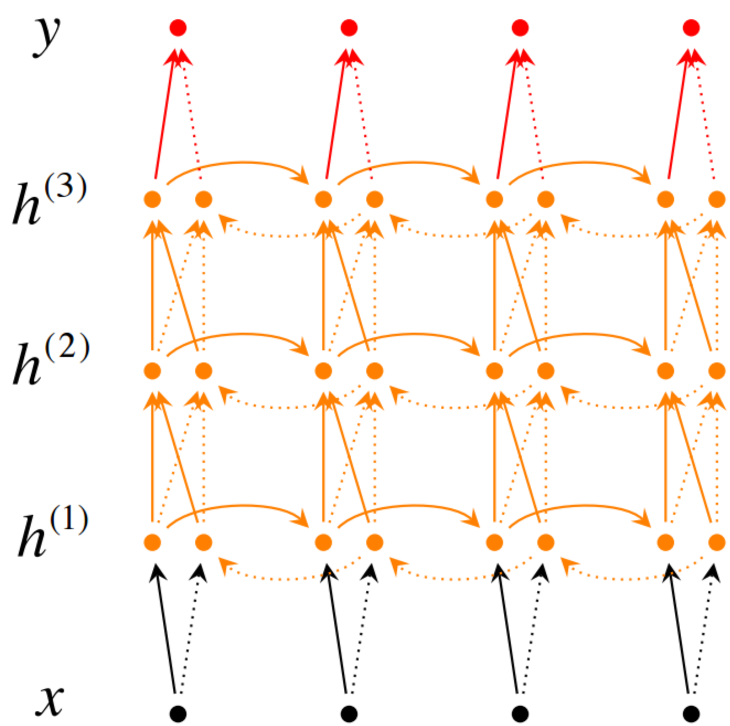


(a) (b)

Structure overview
(a) unidirectional RNN
(b) bidirectional RNN

- For forward pass, forward states and backward states are passed first, then output neurons are passed.
- For backward pass, output neurons are passed first, then forward states and backward states are passed next. After forward and backward passes are done, the weights are updated.

Deep Bidirectional RNN



$$\overrightarrow{h}_t^{(i)} = f(\overrightarrow{W}^{(i)} h_t^{(i-1)} + \overrightarrow{V}^{(i)} \overrightarrow{h}_{t-1}^{(i)} + \overrightarrow{b}^{(i)})$$

$$\overleftarrow{h}_t^{(i)} = f(\overleftarrow{W}^{(i)} h_t^{(i-1)} + \overleftarrow{V}^{(i)} \overleftarrow{h}_{t+1}^{(i)} + \overleftarrow{b}^{(i)})$$

$$y_t = g(U[\overrightarrow{h}_t^{(L)}; \overleftarrow{h}_t^{(L)}] + c)$$

Each memory layer passes an intermediate representation to the next

Concluding discussion

Recursive vs. recurrent neural network models:

- When should we use one vs. the other?