INF721

2024/2

UFV

Deep Learning

L14: Recurrent Neural Networks (Part II)

Logistics

Announcements

- PA3 is due this Wednesday, 11:59pm

Last Lecture

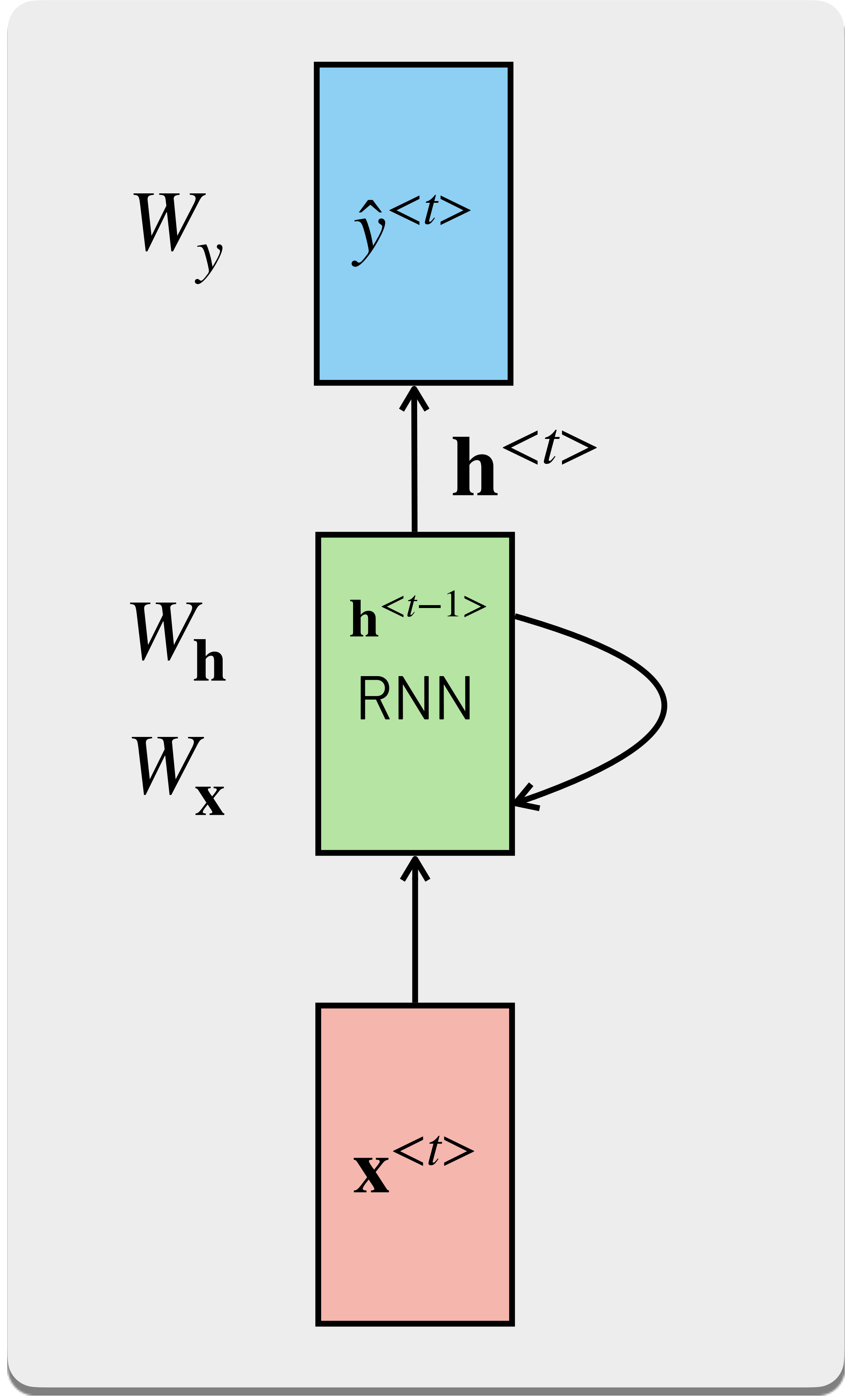
- Sequential Problems
- Recurrent Neural Networks
  - Hidden State
- Forward Pass
- Backward Pass (structure)

UFV

Lecture Outline

- Backpropagation
- Implementing RNNs
- Vanishing/Exploding Gradients
- LSTMs & GRUs
- Bidirectional RNN
- Deep RNNs

Recurrent Neural Networks (RNNs)



RNNs process each input element $\mathbf{x}^{<t>}$ at a time, keeping a state (vector) $\mathbf{h}^{<t>}$ that is updated at each time step t to produce the output $y^{<t>}$

$$\mathbf{h}^{<t>} = g_1(W_h \mathbf{h}^{<t-1>} + W_\mathbf{x} \mathbf{x}^{<t>} + \mathbf{b}_h)$$

$$\hat{y}^{<t>} = g_2(W_y \mathbf{h}^{<t>} + \mathbf{b}_y)$$

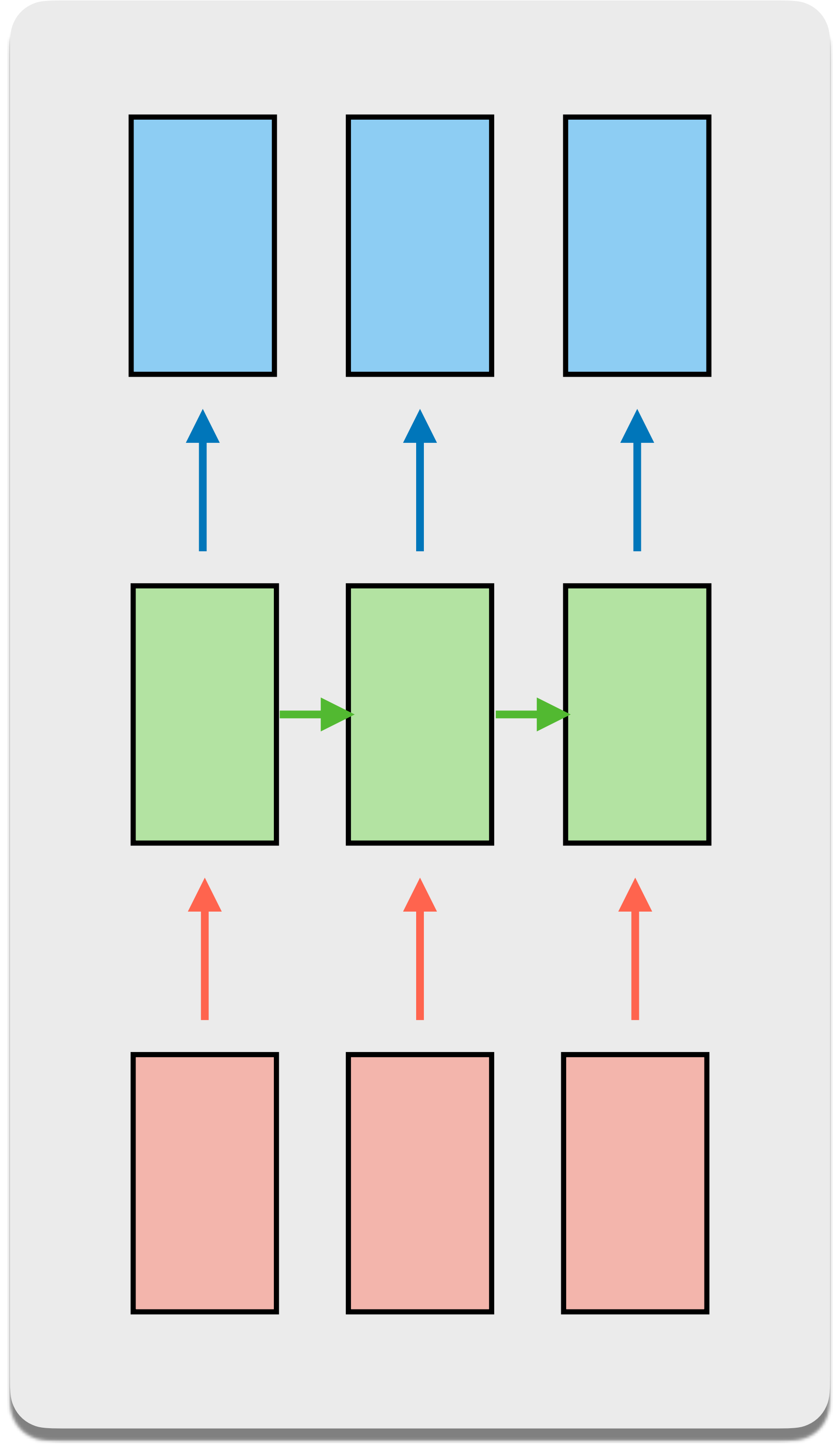
- g_1: hidden layer activation function (tanh/relu)
- g_2: output layer activation function (sigmoid/softmax)

UFV

Elman, J. L. (1990). Finding Structure in Time. Cognitive Science, 14(2), 179–211.

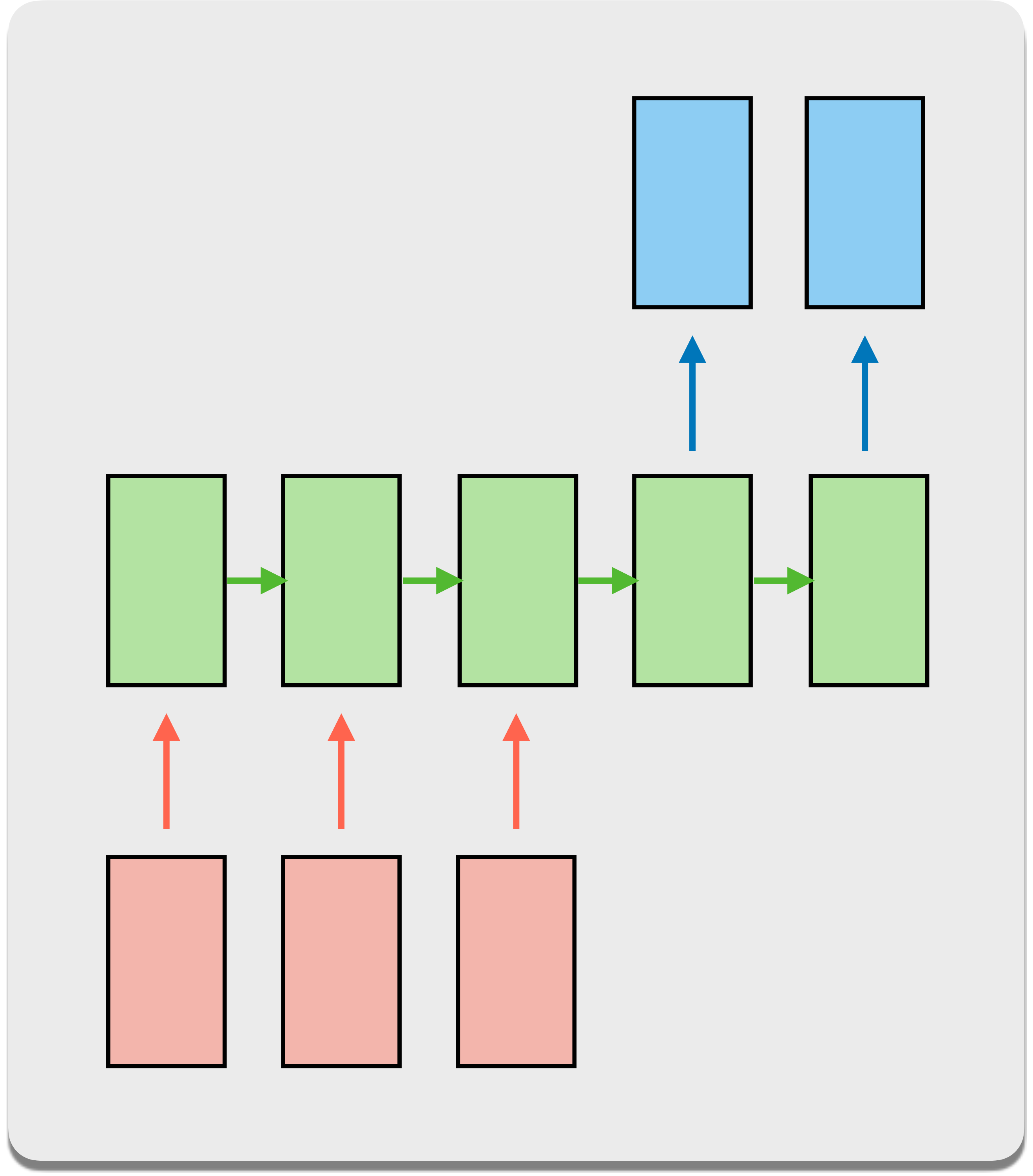
Types of RNNs

Many to **Many**



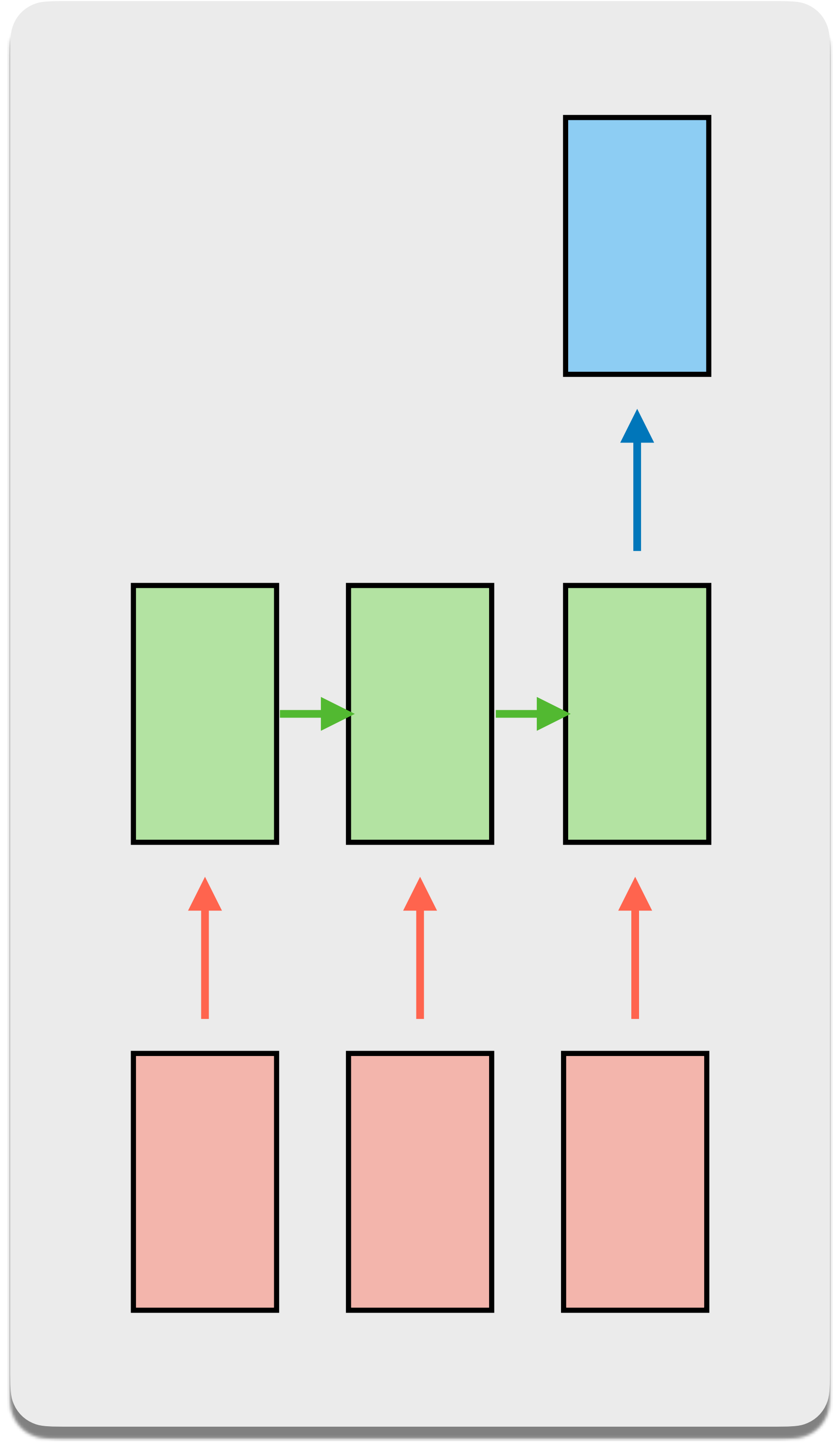
Example
Named Entity Recognition

Many to **Many (Seq2Seq)**



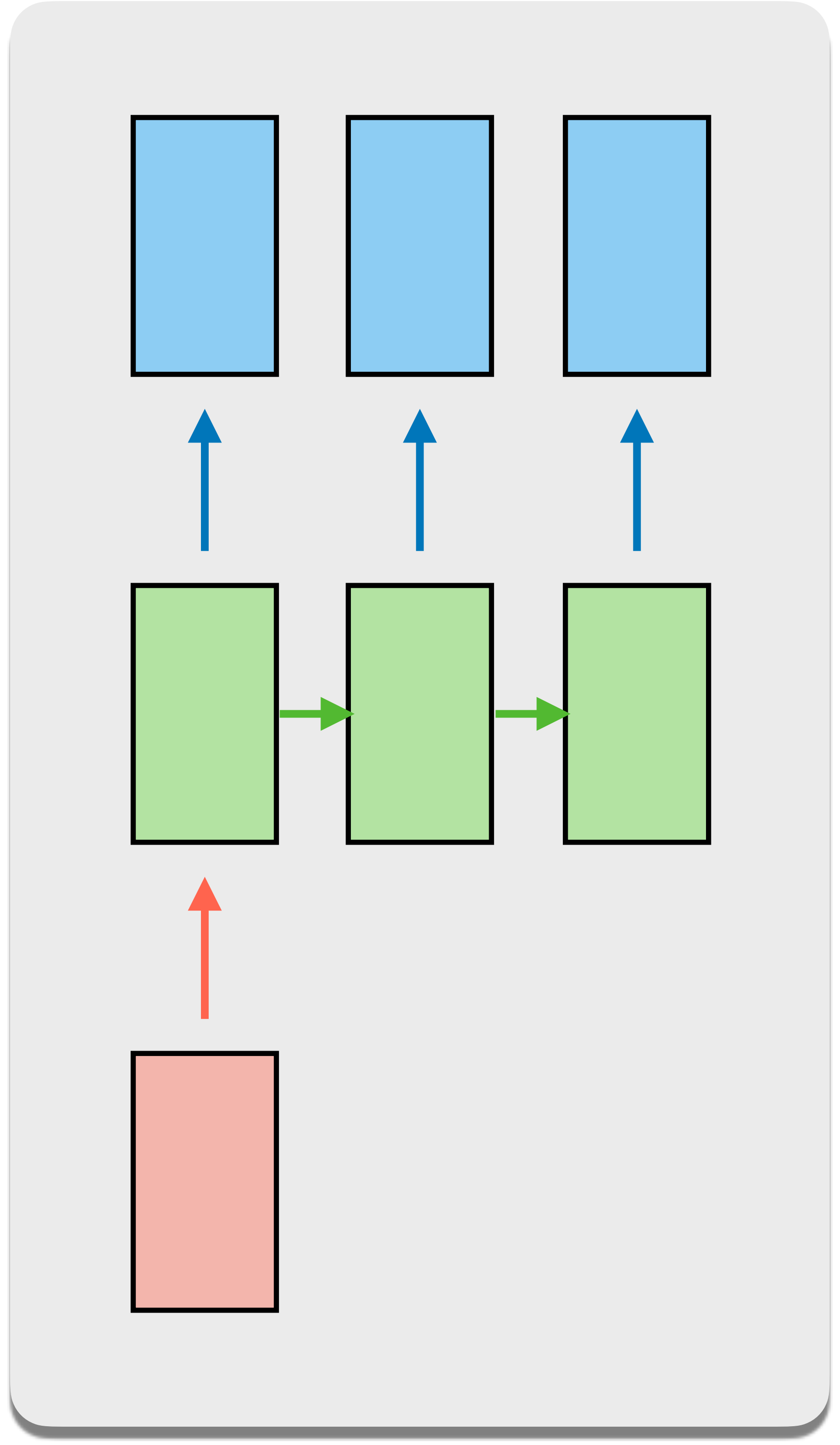
Example
Machine Translation

Many to **one**



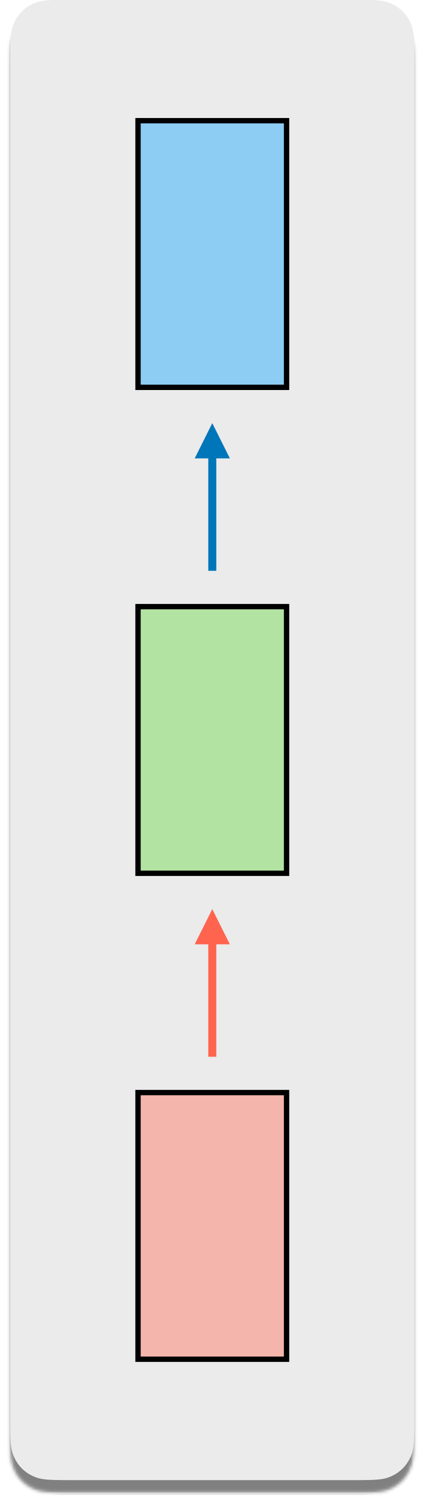
Example
Sentiment Analysis

One to **many**



Example
Image Description

one to **one**



MLP

Language Model

A **Language Model** predicts the next word (or character) from a textual context.

Language Modeling is a fundamental problem in Natural Language Processing!

This lecture is very

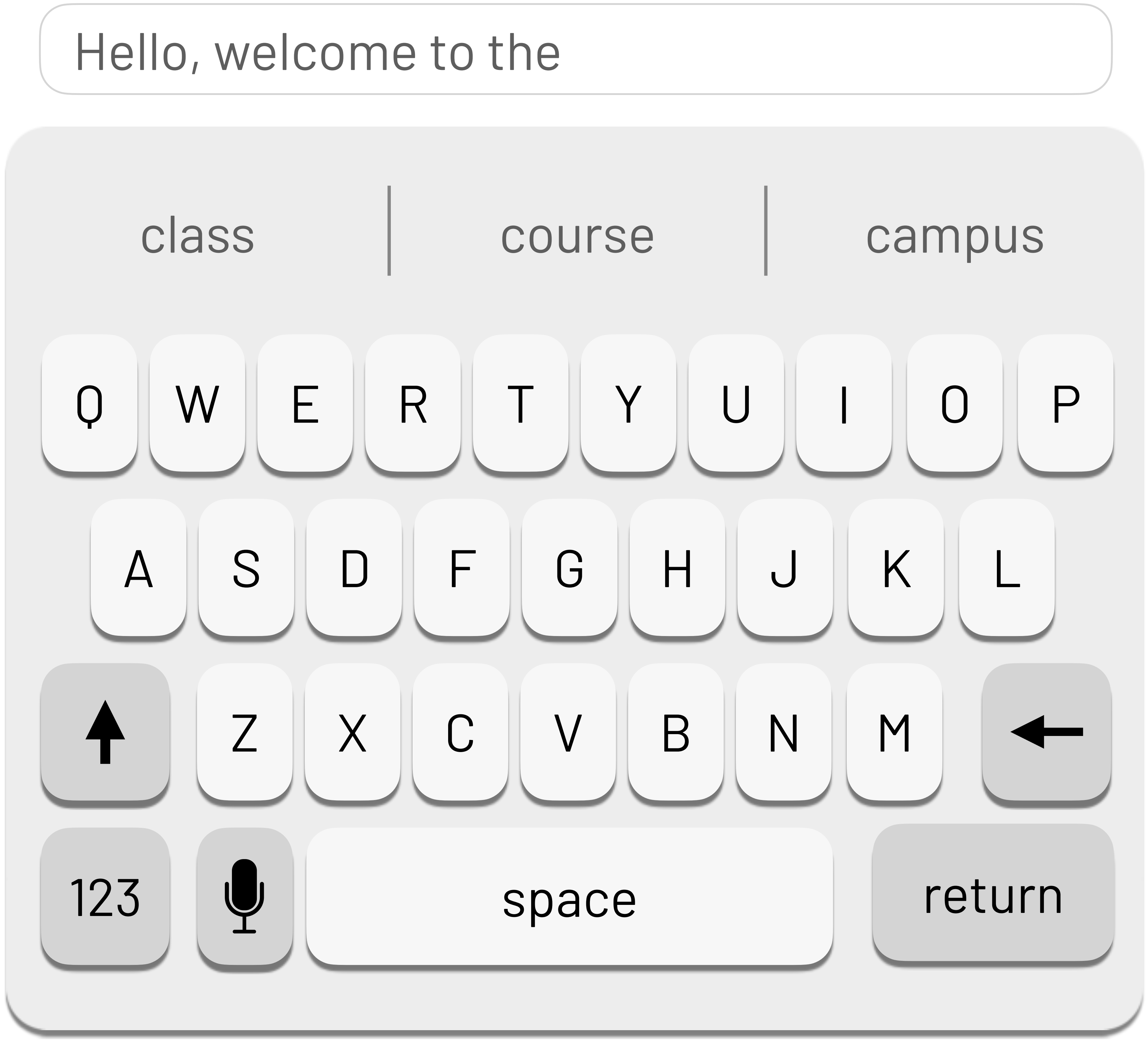
context $\{x_1, x_2, x_3, x_4\}$

$$P(x_5 \mid x_1, x_2, x_3, x_4)$$

0.31 cool
0.28 interesting
...
0.05 classroom
0.01 university

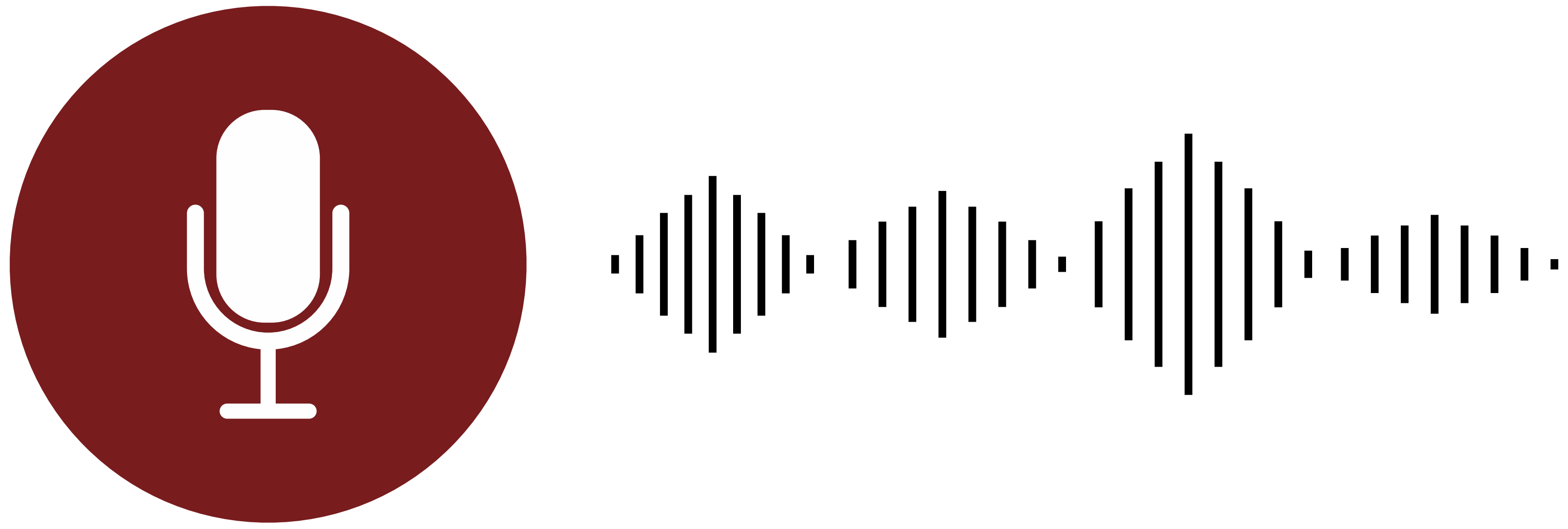
Applications of Language Models

We can use a Language Model to implement autocompletion in mobile keyboards:



Applications of Language Models

We can use a Language Model to compute the probability of a given sentence when decoding audio for speech recognition :



P("The apple and **pair** salad?") = 0.1

P("The apple and **pear** salad?") = 0.23

Default (GPT-3.5)

Explique em poucas palavras como você funciona.

Eu sou um modelo de linguagem desenvolvido pela OpenAI chamado GPT-3.5. Funciono processando texto ou linguagem escrita para gerar respostas ou textos coerentes e relevantes com base nas informações e no contexto fornecido. Isso é feito por meio do aprendizado de máquina e da análise de vastas quantidades de texto durante o treinamento. Posso ser usado para uma variedade de tarefas, como responder a perguntas, gerar texto, traduzir idiomas e muito mais. Minha capacidade de resposta depende das informações disponíveis até a minha data de corte em setembro de 2021.

Language Model: Dataset

https://pt.wikipedia.org/wiki/

Universidade Federal de Viçosa

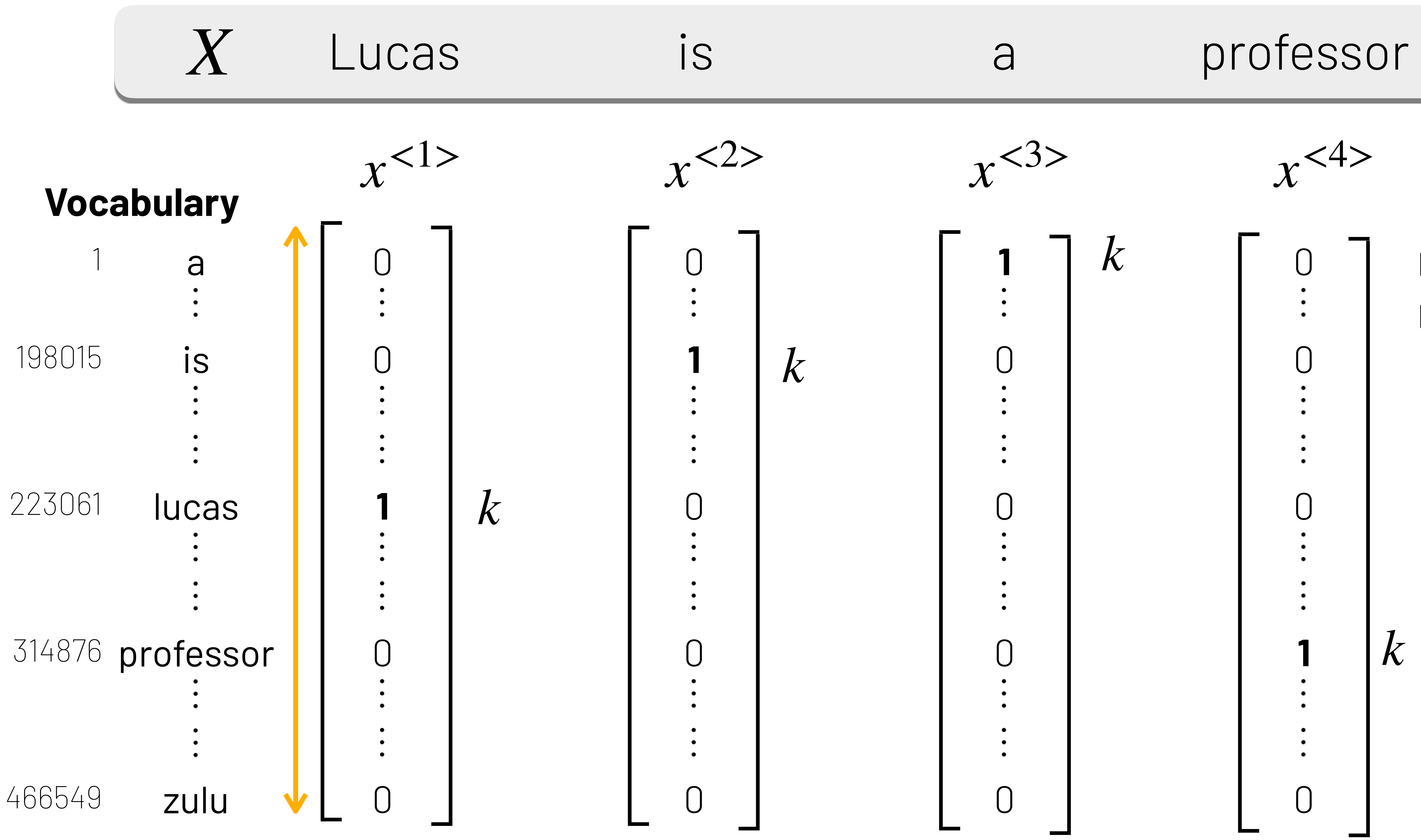
A Universidade Federal de Viçosa (UFV) é uma universidade pública brasileira, com sua sede localizada na cidade de Viçosa, no estado de Minas Gerais, possuindo campus também nas cidades de Rio Paranaíba e Florestal.

Collect a giant volume of text (e.g., wikipedia) and create examples $(\mathbf{x}, y)$ using a sliding window (e.g., size $j = 8$)

$x^{(1)}$	A	Universidade	Federal	de	Viçosa	(UFV)	é
$y^{(1)}$	Universidade	Federal	de	Viçosa	(UFV)	é	uma
$x^{(2)}$	universidade	pública	brasileira	,	com	sua	sede
$y^{(2)}$	pública	brasileira	,	com	sua	sede	localizada
$x^{(3)}$	na	cidade	de	Viçosa	,	no	estado
$y^{(3)}$	cidade	de	Viçosa	,	no	estado	de
$x^{(4)}$	Minas	Gerais	,	possuindo	campus	também	nas
$y^{(4)}$	Gerais	,	possuindo	campus	também	nas	Cidades
$x^{(5)}$	de	Rio	Paranaíba	e	Florestal	.	<PAD>
$y^{(5)}$	Rio	Paranaíba	e	Florestal	.	<PAD>	<PAD>

Representing words as vectors

One of the simplest strategies to represent works as vectors is the **one-hot encoding**:

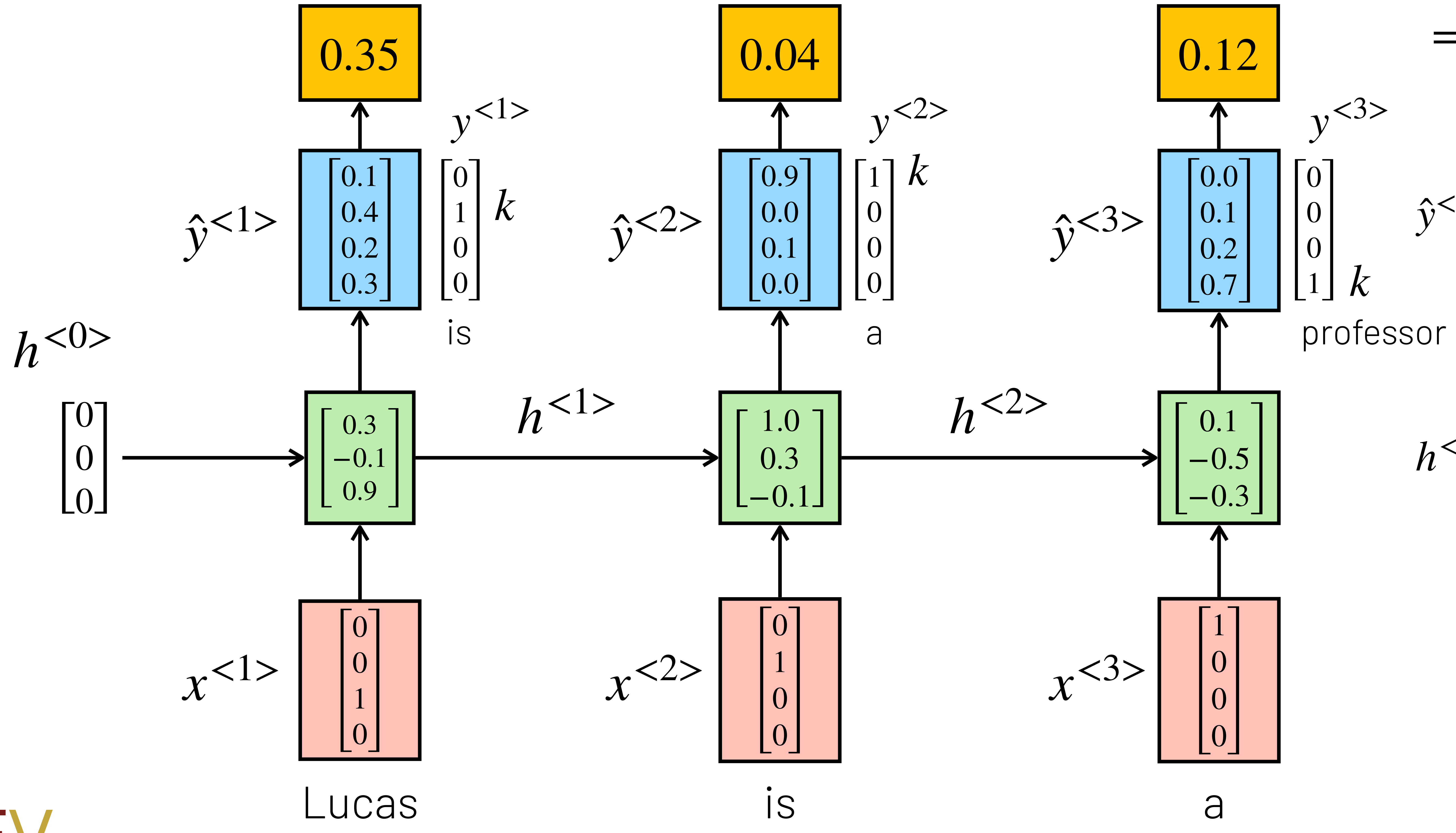


- Each word is assigned an index k
- Each word is represented by a feature vector where:
  - Only the k-th feature is 1;
  - All other features are 0

UFV

Language Model: Forward

Lucas is a professor



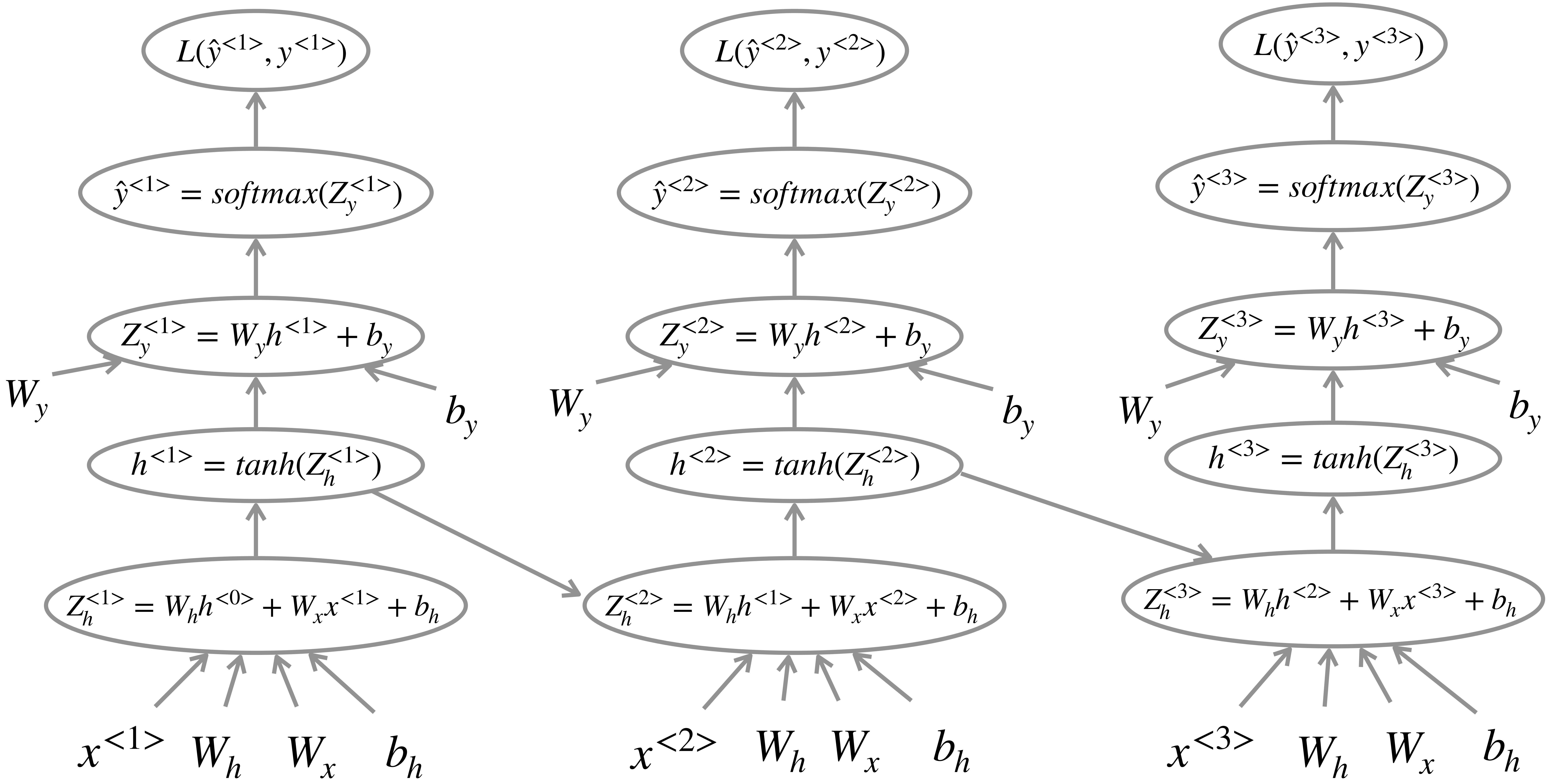
$$L^{<t>}(\hat{y}, y) = -\sum_{i=1}^{t}\sum_{j=1}^{d} y_j^{<i>} log\hat{y}_j^{<i>}$$

$$= -\sum_{i=1}^{t} y_k^{<i>} log\hat{y}_k^{<i>} = -\sum_{i=1}^{t} log\hat{y}_k^{<i>}$$

$$\hat{y}^{<t>} = softmax(W_y h^{<t>} + b_y)$$

$$h^{<t>} = tanh(W_h h^{<t-1>} + W_x x^{<t>} + b_h)$$

Language Model: Computational Graph



UFV

Backward timestep t

$$\frac{\partial L}{\partial \hat{y}^{<t>}} = -\frac{y^{<t>}}{\hat{y}^{<t>}}$$

$$\frac{\partial L}{\partial Z_y^{<t>}} = \hat{y}^{<t>} - y^{<t>}$$

$$\frac{\partial Z_y^{<t>}}{\partial W_y} = h^{<t>}, \frac{\partial Z_y^{<t>}}{\partial b_y} = 1, \frac{\partial Z_y^{<t>}}{\partial h^{<t>}} = W_y^T$$

$$\frac{\partial h^{<t>}}{\partial Z_h^{<t>}} = 1 - (h^{<t>})^2$$

$$\frac{\partial Z_h^{<t>}}{\partial W_h} = h^{<t-1>}, \frac{\partial Z_h^{<t>}}{\partial b_h} = 1, \frac{\partial Z_h^{<t>}}{\partial W_x} = x^{<t>}$$

$$L(\hat{y}^{<t>}, y^{<t>})$$

$$\hat{y}^{<t>} = softmax(Z_y^{<t>})$$

$$Z_y^{<t>} = W_y h^{<t>} + b_y$$

W_y b_y

$$h^{<t>} = tanh(Z_h^{<t>})$$

$$Z_h^{<t>} = W_h h^{<t-1>} + W_x x^{<t>} + b_h$$

$x^{<t>}$ W_h W_x b_h

1. $\frac{\partial L}{\partial L} = 1$
2. $\frac{\partial L}{\partial \hat{y}^{<t>}} = \frac{\partial L}{\partial \hat{y}^{<t>}} \cdot \frac{\partial L}{\partial L}$
3. $\frac{\partial L}{\partial Z_y^{<t>}} = \frac{\partial \hat{y}^{<t>}}{\partial Z_y^{<t>}} \cdot \frac{\partial L}{\partial \hat{y}^{<t>}}$
4. $\frac{\partial L}{\partial b_y} = \frac{\partial Z_y^{<t>}}{\partial b_y^{<t>}} \cdot \frac{\partial L}{\partial Z_y^{<t>}}$
5. $\frac{\partial L}{\partial W_y} = \frac{\partial Z_y^{<t>}}{\partial W_y^{<t>}} \cdot \frac{\partial L}{\partial Z_y^{<t>}}$
6. $\frac{\partial L}{\partial h^{<t>}} = \frac{\partial Z_y^{<t>}}{\partial h^{<t>}} \cdot \frac{\partial L}{\partial Z_y^{<t>}}$
7. $\frac{\partial L}{\partial Z_h^{<t>}} = \frac{\partial h^{<t>}}{\partial Z_h^{<t>}} \cdot \frac{\partial L}{\partial h^{<t>}}$
8. $\frac{\partial L}{\partial b_h^{<t>}} = \frac{\partial Z_h^{<t>}}{\partial b_h^{<t>}} \cdot \frac{\partial L}{\partial Z_h^{<t>}}$
9. $\frac{\partial L}{\partial W_x} = \frac{\partial Z_h^{<t>}}{\partial W_x} \cdot \frac{\partial L}{\partial Z_h^{<t>}}$
10. $\frac{\partial L}{\partial W_h} = \frac{\partial Z_h^{<t>}}{\partial W_h} \cdot \frac{\partial L}{\partial Z_h^{<t>}}$

UFV

Backward timestep t

$$L(\hat{y}^{<t>}, y^{<t>})$$

$$\hat{y}^{<t>} = softmax(Z_y^{<t>})$$

$$Z_y^{<t>} = W_y h^{<t>} + b_y$$

$$h^{<t>} = tanh(Z_h^{<t>})$$

$$Z_h^{<t>} = W_h h^{<t-1>} + W_x x^{<t>} + b_h$$

W_y, b_y, $x^{<t>}$, W_h, W_x, b_h

$$\frac{\partial L}{\partial \hat{y}^{<t>}} = -\frac{y^{<t>}}{\hat{y}^{<t>}}$$

$$\frac{\partial L}{\partial Z_y^{<t>}} = \hat{y}^{<t>} - y^{<t>}$$

$$\frac{\partial Z_y^{<t>}}{\partial W_y} = h^{<t>}, \quad \frac{\partial Z_y^{<t>}}{\partial b_y} = 1, \quad \frac{\partial Z_y^{<t>}}{\partial h^{<t>}} = W_y^T$$

$$\frac{\partial h^{<t>}}{\partial Z_h^{<t>}} = 1 - (h^{<t>})^2$$

$$\frac{\partial Z_h^{<t>}}{\partial W_h} = h^{<t-1>}, \quad \frac{\partial Z_h^{<t>}}{\partial b_h} = 1, \quad \frac{\partial Z_h^{<t>}}{\partial W_x} = x^{<t>}$$

1. $\dfrac{\partial L}{\partial L} = 1$

2. $\dfrac{\partial L}{\partial \hat{y}^{<t>}} = -\dfrac{1}{\hat{y}_r^{<t>}}$

3. $\dfrac{\partial L}{\partial Z_y^{<t>}} = \hat{y}_r^{<t>} - 1$

4. $\dfrac{\partial L}{\partial b_y} = \hat{y}_r^{<t>} - 1$

5. $\dfrac{\partial L}{\partial W_y} = \dfrac{\partial L}{\partial Z_y^{<t>}} \cdot (h^{<t>})^T$

6. $\dfrac{\partial L}{\partial h^{<t>}} = W_y^T \cdot \dfrac{\partial L}{\partial Z_y^{<t>}}$

7. $\dfrac{\partial L}{\partial Z_h^{<t>}} = (1 - (h^{<t>})^2) \odot \dfrac{\partial L}{\partial h^{<t>}}$

8. $\dfrac{\partial L}{\partial b_h} = \dfrac{\partial L}{\partial Z_h^{<t>}}$

9. $\dfrac{\partial L}{\partial W_x} = \dfrac{\partial L}{\partial Z_h^{<t>}} \cdot (x^{<t>})^T$

10. $\dfrac{\partial L}{\partial W_h} = \dfrac{\partial L}{\partial Z_h^{<t>}} \cdot (h^{<t-1>})^T$

UFV

Andrej Karpathy's Minimal Character Level RNN

```python
"""
Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
BSD License
"""
import numpy as np

# data I/O
data = open('input.txt', 'r').read() # should be simple plain text file
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print 'data has %d characters, %d unique.' % (data_size, vocab_size)
char_to_ix = { ch:i for i,ch in enumerate(chars) }
ix_to_char = { i:ch for i,ch in enumerate(chars) }

# hyperparameters
hidden_size = 100 # size of hidden layer of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias

def lossFun(inputs, targets, hprev):
  """
  inputs,targets are both list of integers.
  hprev is Hx1 array of initial hidden state
  returns the loss, gradients on model parameters, and last hidden state
  """
  xs, hs, ys, ps = {}, {}, {}, {}
  hs[-1] = np.copy(hprev)
  loss = 0
  # forward pass
  for t in xrange(len(inputs)):
    xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
    xs[t][inputs[t]] = 1
    hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
    ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
    ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
    loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
  # backward pass: compute gradients going backwards
  dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
  dbh, dby = np.zeros_like(bh), np.zeros_like(by)
  dhnext = np.zeros_like(hs[0])
  for t in reversed(xrange(len(inputs))):
    dy = np.copy(ps[t])
    dy[targets[t]] -= 1 # backprop into y. see http://cs231n.github.io/neural-networks-case-study/#grad if confused here
    dWhy += np.dot(dy, hs[t].T)
    dby += dy
    dh = np.dot(Why.T, dy) + dhnext # backprop into h
    dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
    dbh += dhraw
    dWxh += np.dot(dhraw, xs[t].T)
    dWhh += np.dot(dhraw, hs[t-1].T)
    dhnext = np.dot(Whh.T, dhraw)
  for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
    np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
  return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]
```

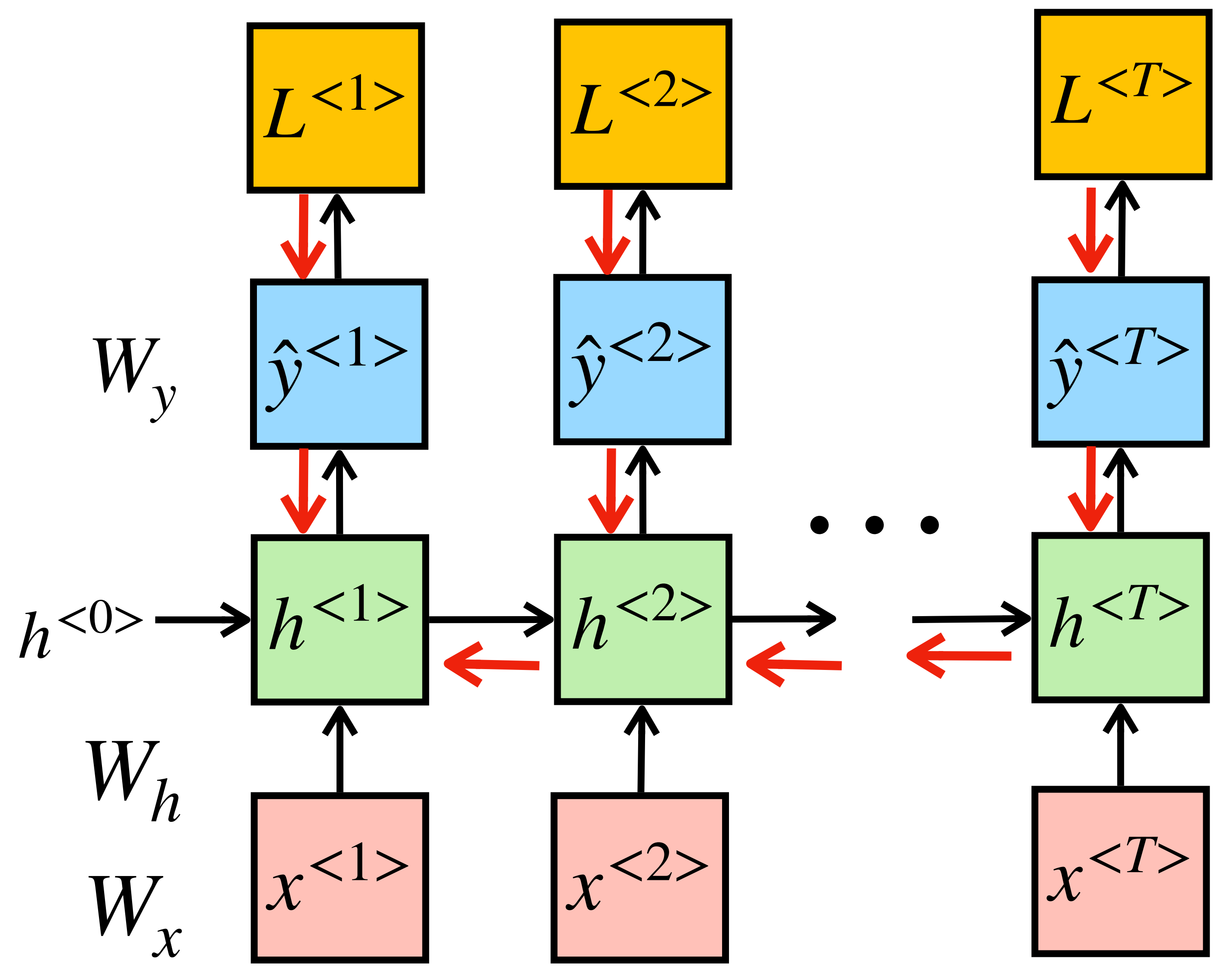
```python
def sample(h, seed_ix, n):
  """
  sample a sequence of integers from the model
  h is memory state, seed_ix is seed letter for first time step
  """
  x = np.zeros((vocab_size, 1))
  x[seed_ix] = 1
  ixes = []
  for t in xrange(n):
    h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
    y = np.dot(Why, h) + by
    p = np.exp(y) / np.sum(np.exp(y))
    ix = np.random.choice(range(vocab_size), p=p.ravel())
    x = np.zeros((vocab_size, 1))
    x[ix] = 1
    ixes.append(ix)
  return ixes

n, p = 0, 0
mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
while True:
  # prepare inputs (we're sweeping from left to right in steps seq_length long)
  if p+seq_length+1 >= len(data) or n == 0:
    hprev = np.zeros((hidden_size,1)) # reset RNN memory
    p = 0 # go from start of data
  inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
  targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]

  # sample from the model now and then
  if n % 100 == 0:
    sample_ix = sample(hprev, inputs[0], 200)
    txt = ''.join(ix_to_char[ix] for ix in sample_ix)
    print '----\n %s \n----' % (txt, )

  # forward seq_length characters through the net and fetch gradient
  loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
  smooth_loss = smooth_loss * 0.999 + loss * 0.001
  if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress

  # perform parameter update with Adagrad
  for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
                                [dWxh, dWhh, dWhy, dbh, dby],
                                [mWxh, mWhh, mWhy, mbh, mby]):
    mem += dparam * dparam
    param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

  p += seq_length # move data pointer
  n += 1 # iteration counter
```

https://gist.github.com/karpathy/d4dee566867f8291f086

Exploding/Vanishing Gradients

When processing large sequences, RNNs can suffer from exploding or vanishing gradients:

```
44     # backward pass: compute gradients going backwards
45     dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
46     dbh, dby = np.zeros_like(bh), np.zeros_like(by)
47     dhnext = np.zeros_like(hs[0])
48     for t in reversed(xrange(len(inputs))):
49       dy = np.copy(ps[t])
50       dy[targets[t]] -= 1
51       dWhy += np.dot(dy, hs[t].T)
52       dby += dy
53       dh = np.dot(Why.T, dy) + dhnext # backprop into h
54       dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
55       dbh += dhraw
56       dWxh += np.dot(dhraw, xs[t].T)
57       dWhh += np.dot(dhraw, hs[t-1].T)
58       dhnext = np.dot(Whh.T, dhraw)
```

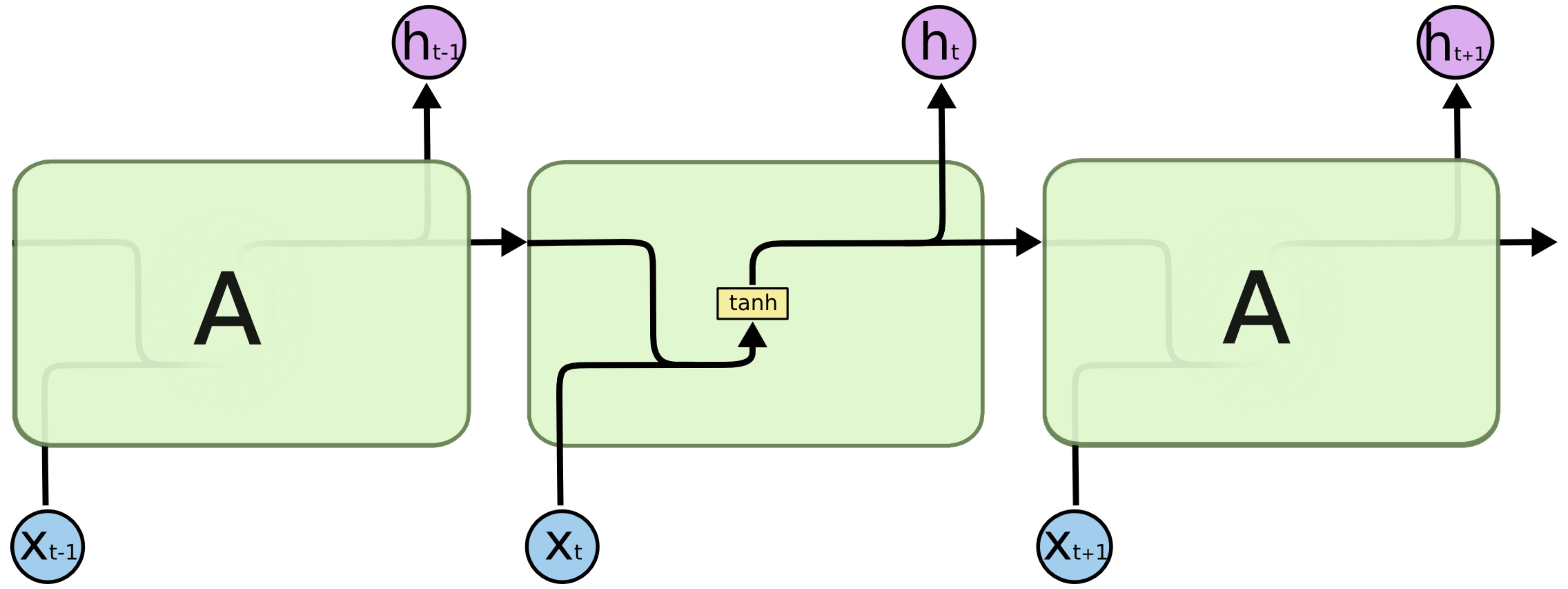


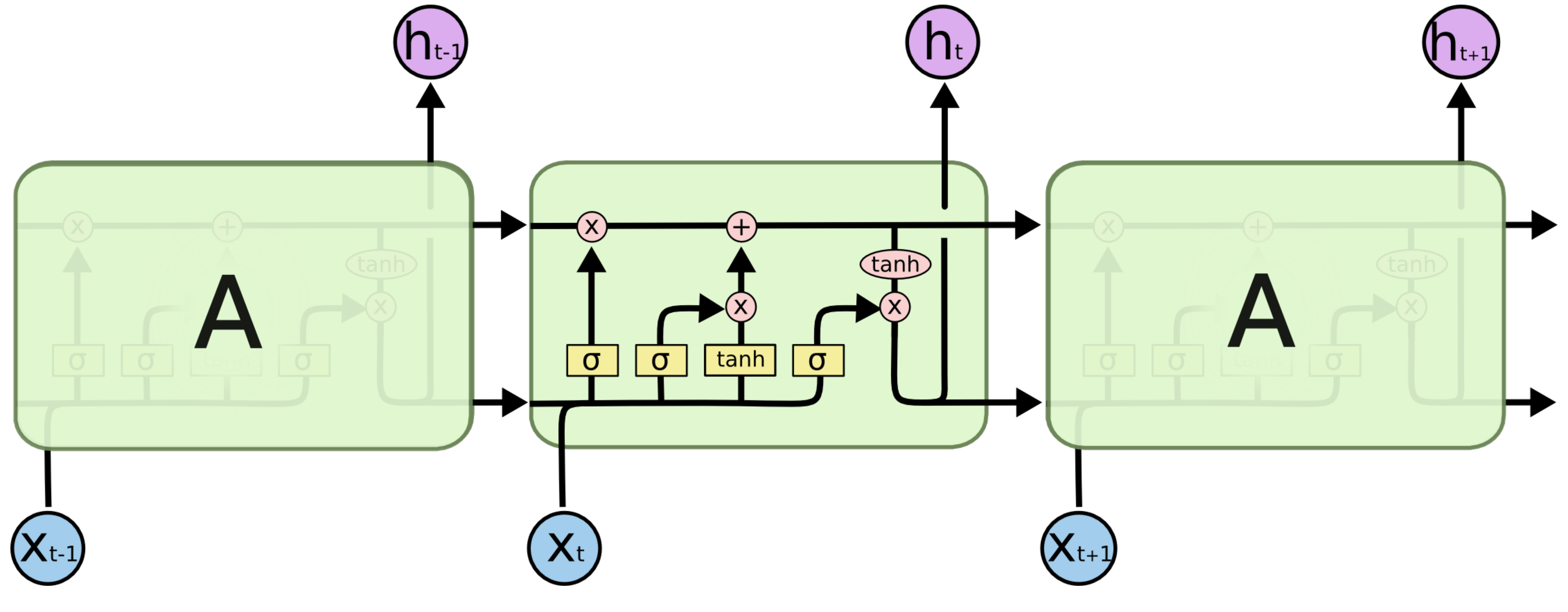
Backpropation thought time makes a series of multiplications of W_h by itself (line 58):

- If the weights of $W_h > 1 \rightarrow$ exploding gradients
- If the weights of $W_h < 1 \rightarrow$ vanishing grandients

UFV

Long-Short Term Memory (LSTM)





LSTM are complex RNNs to handle vanishing/exploding gradients

RNN Hidden Layer :

$$h^{<t>} = tanh(W_h h^{<t-1>} + W_x x^{<t>} + b_h)$$

LSTM Hidden Layer :

$$f^{<t>} = \sigma(W_{fh} h^{<t-1>} + W_{fx} x^{<t>} + b_f) \qquad \textit{Forget Gate}$$

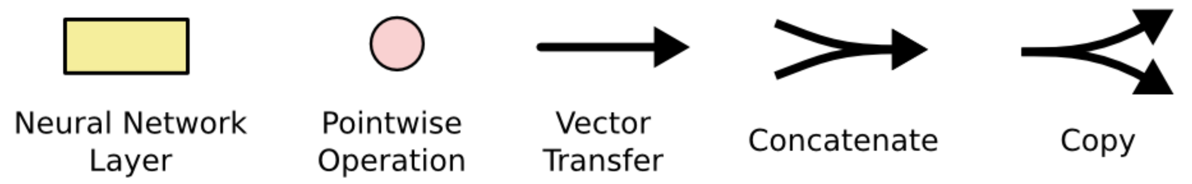
$$i^{<t>} = \sigma(W_{ih} h^{<t-1>} + W_{ix} x^{<t>} + b_i) \qquad \textit{Input Gate}$$

$$\tilde{C}^{<t>} = tanh(W_{ch} h^{<t-1>} + W_{cx} x^{<t>} + b_C)$$

$$C^{<t>} = f^{<t>} \odot C^{<t-1>} + i^{<t>} \odot \tilde{C}^{<t>} \qquad \textit{Cell State}$$

$$o^{<t>} = \sigma(W_{oh} h^{<t-1>} + W_{ox} x^{<t>} + b_o) \qquad \textit{Output Gate}$$

$$h^{<t>} = o_t \odot tanh(C^{<t>})$$

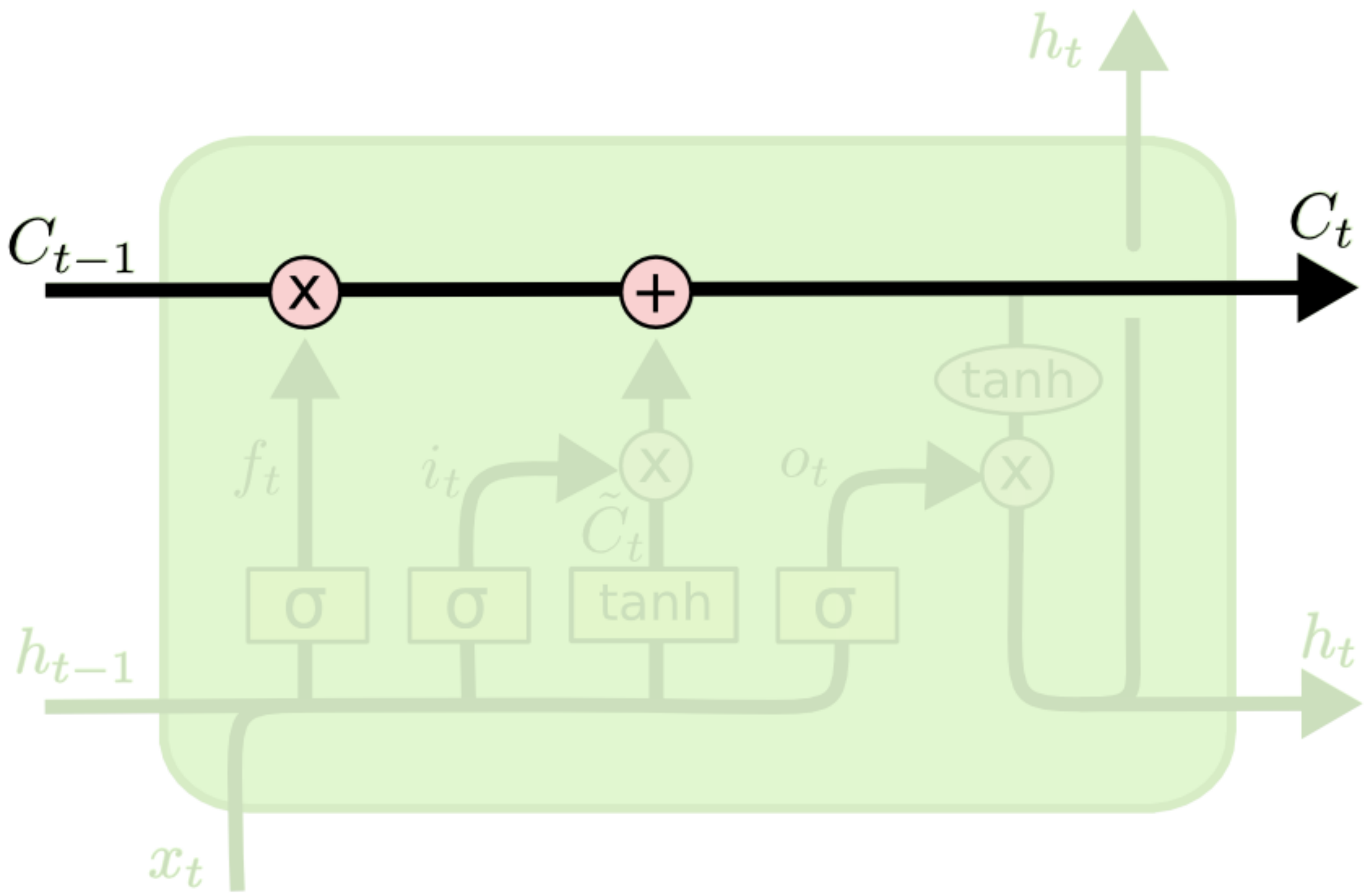


Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory." Neural Computation MIT-Press (1997).

LSTM: Cell State and Gates

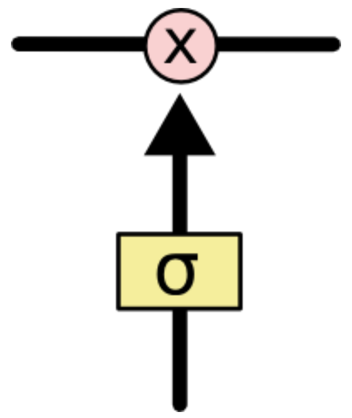
The key to LSTMs is the **cell state** $C^{<t>}$, which can be seen as another hidden state:

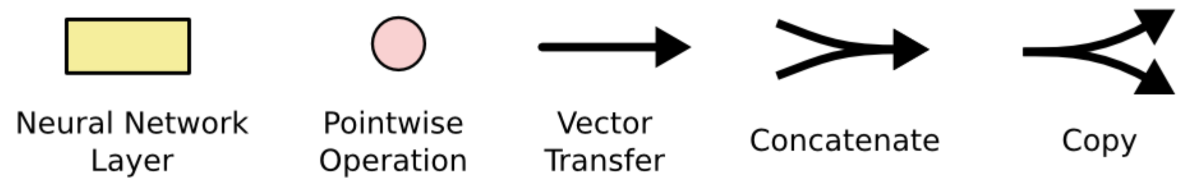
- It runs straight down the entire sequence, with only some minor linear interactions.
- It's very easy for information to just flow along it unchanged.



The **gates** control what information gets removed or added to the cell state:

- **Forget**, **Input**, and **Ouput** Gates
- Implemented as a sigmoid (σ) units and pointwise multiplication



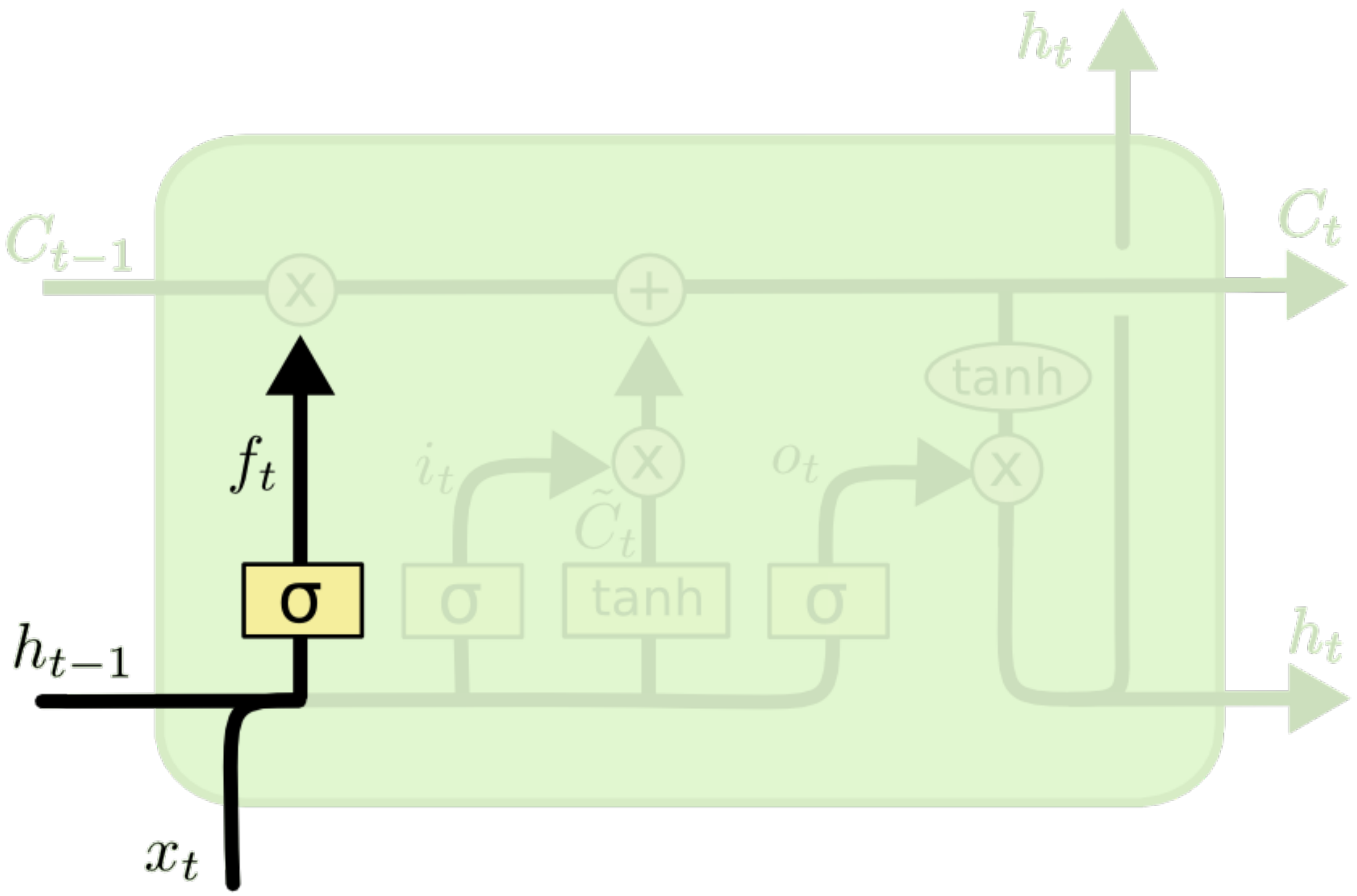


UFV

LSTM: The Forget Gate

The first step is to decide what information we're going to throw away from the cell state

- This decision is made by a sigmoid layer called the "forget gate layer."



LSTM Hidden Layer :

$$f^{<t>} = \sigma(W_{fh}h^{<t-1>} + W_{fx}x^{<t>} + b_f)$$

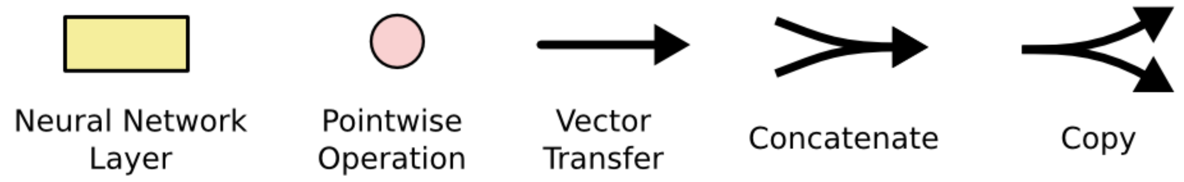
$$i^{<t>} = \sigma(W_{ih}h^{<t-1>} + W_{ix}x^{<t>} + b_i)$$

$$\tilde{C}^{<t>} = tanh(W_{ch}h^{<t-1>} + W_{cx}x^{<t>} + b_C)$$

$$C^{<t>} = f^{<t>} \odot C^{<t-1>} + i^{<t>} \odot \tilde{C}^{<t>}$$

$$o^{<t>} = \sigma(W_{oh}h^{<t-1>} + W_{ox}x^{<t>} + b_o)$$

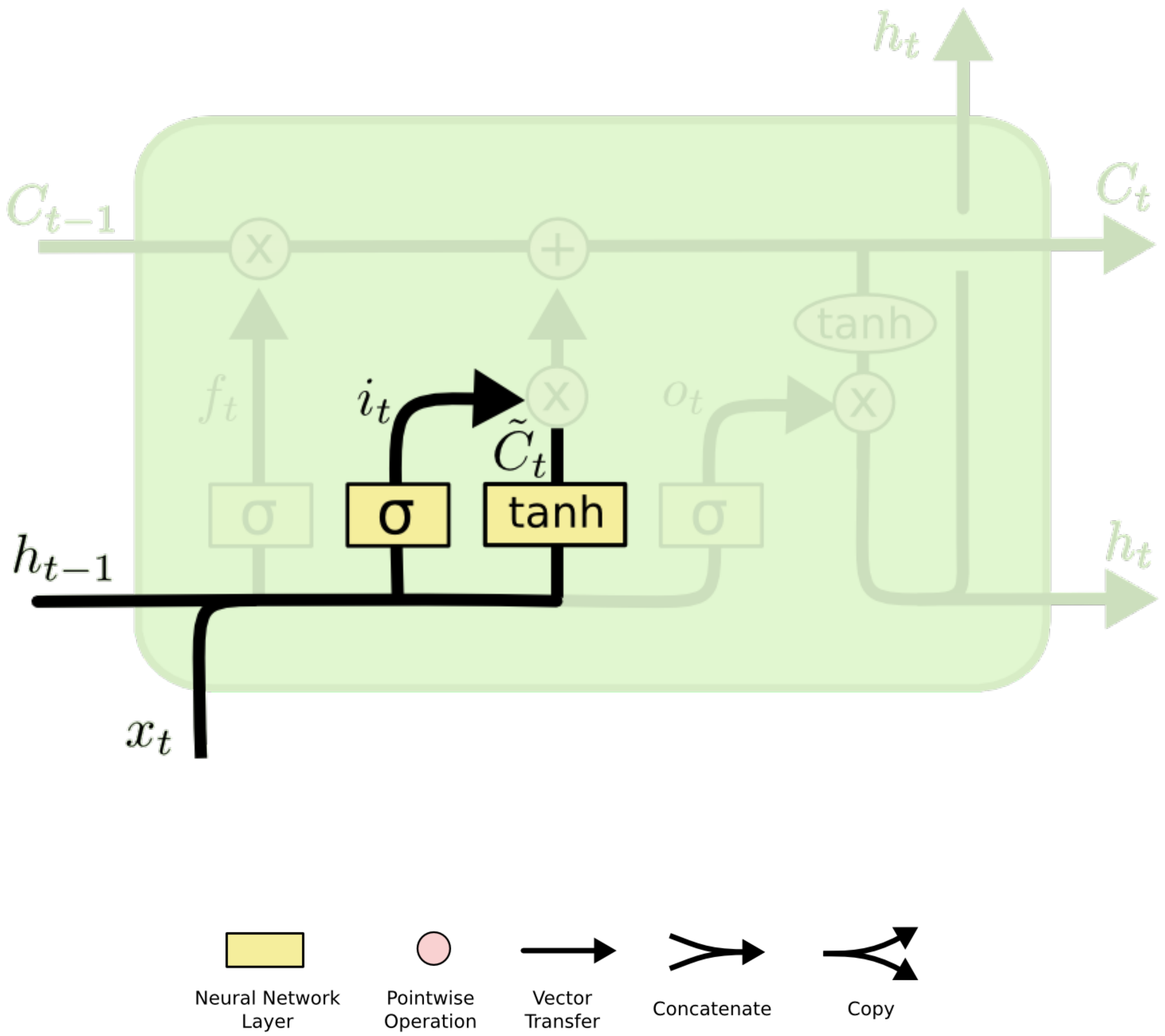
$$h^{<t>} = o_t \odot tanh(C^{<t>})$$



LSTM: The Input Gate

The next step is to decide what new information we're going to store in the cell state:

1. The input gate decides which values we'll update;
2. A $tanh$ unit creates a new candidate state $\tilde{C}^{<t>}$ that could be added to $C^{<t>}$



LSTM Hidden Layer :

$$f^{<t>} = \sigma(W_{fh}h^{<t-1>} + W_{fx}x^{<t>} + b_f)$$

$$i^{<t>} = \sigma(W_{ih}h^{<t-1>} + W_{ix}x^{<t>} + b_i)$$

$$\tilde{C}^{<t>} = tanh(W_{ch}h^{<t-1>} + W_{cx}x^{<t>} + b_C)$$

$$C^{<t>} = f^{<t>} \odot C^{<t-1>} + i^{<t>} \odot \tilde{C}^{<t>}$$

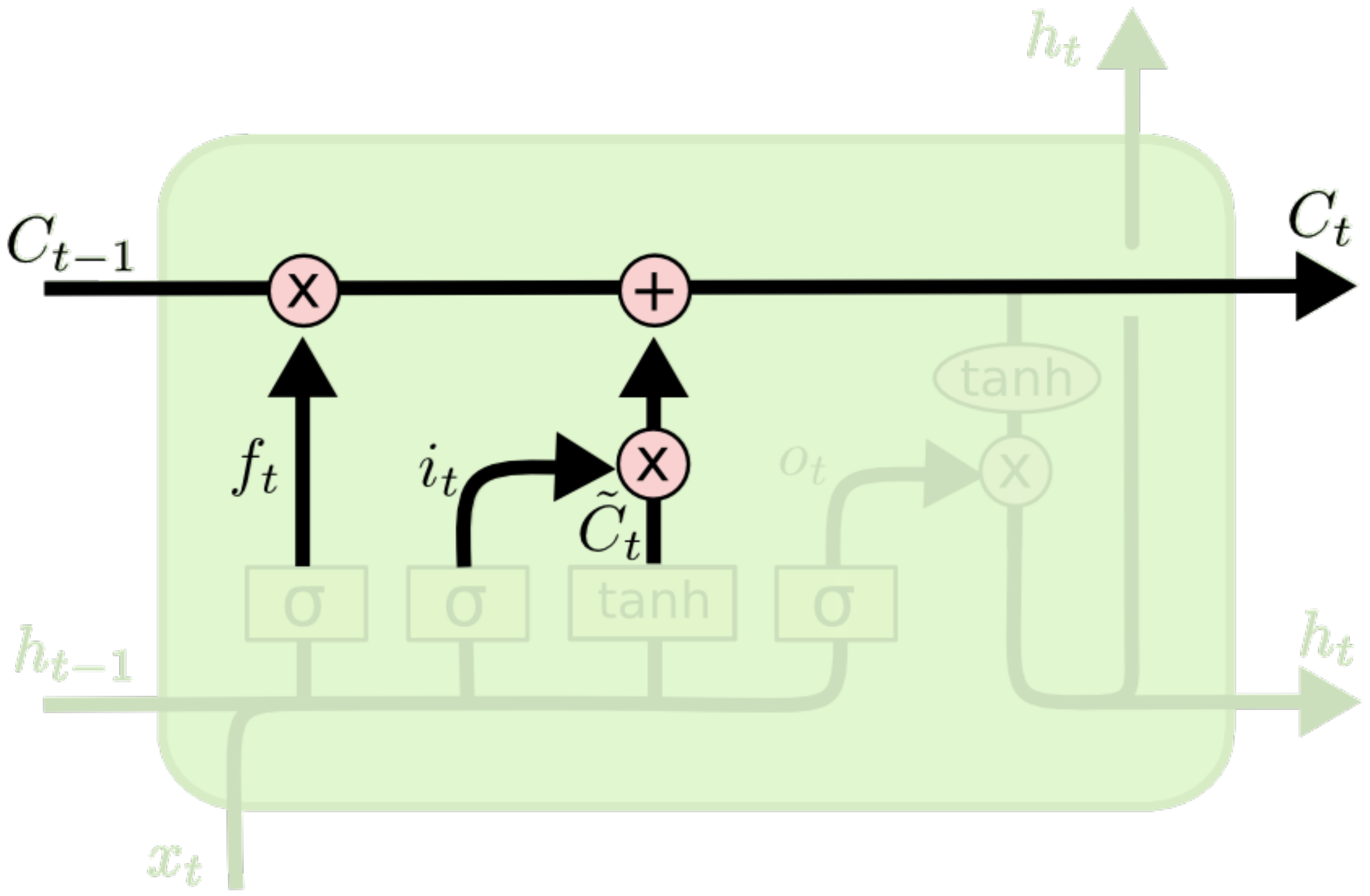
$$o^{<t>} = \sigma(W_{oh}h^{<t-1>} + W_{ox}x^{<t>} + b_o)$$

$$h^{<t>} = o_t \odot tanh(C^{<t>})$$

LSTM: Updating the Cell State

It's now time to update the old cell state:

1. Multiply the old state $C^{<t-1>}$ by $f^{<t>}$ to forget the information we decided to forget earlier
2. Sum $i^{<t>} \odot \tilde{C}^{<t>}$ to include the new information that is coming in



LSTM Hidden Layer :

$$f^{<t>} = \sigma(W_{fh}h^{<t-1>} + W_{fx}x^{<t>} + b_f)$$

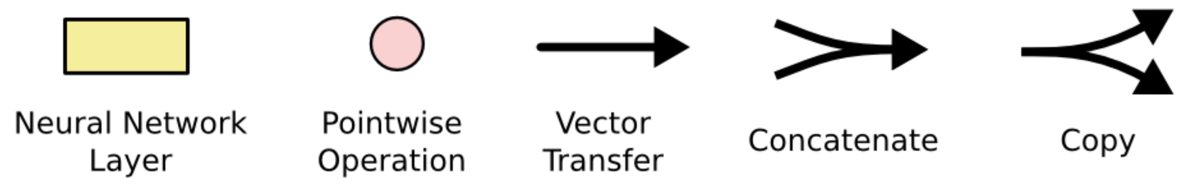
$$i^{<t>} = \sigma(W_{ih}h^{<t-1>} + W_{ix}x^{<t>} + b_i)$$

$$\tilde{C}^{<t>} = tanh(W_{ch}h^{<t-1>} + W_{cx}x^{<t>} + b_C)$$

$$C^{<t>} = f^{<t>} \odot C^{<t-1>} + i^{<t>} \odot \tilde{C}^{<t>}$$

$$o^{<t>} = \sigma(W_{oh}h^{<t-1>} + W_{ox}x^{<t>} + b_o)$$

$$h^{<t>} = o_t \odot tanh(C^{<t>})$$

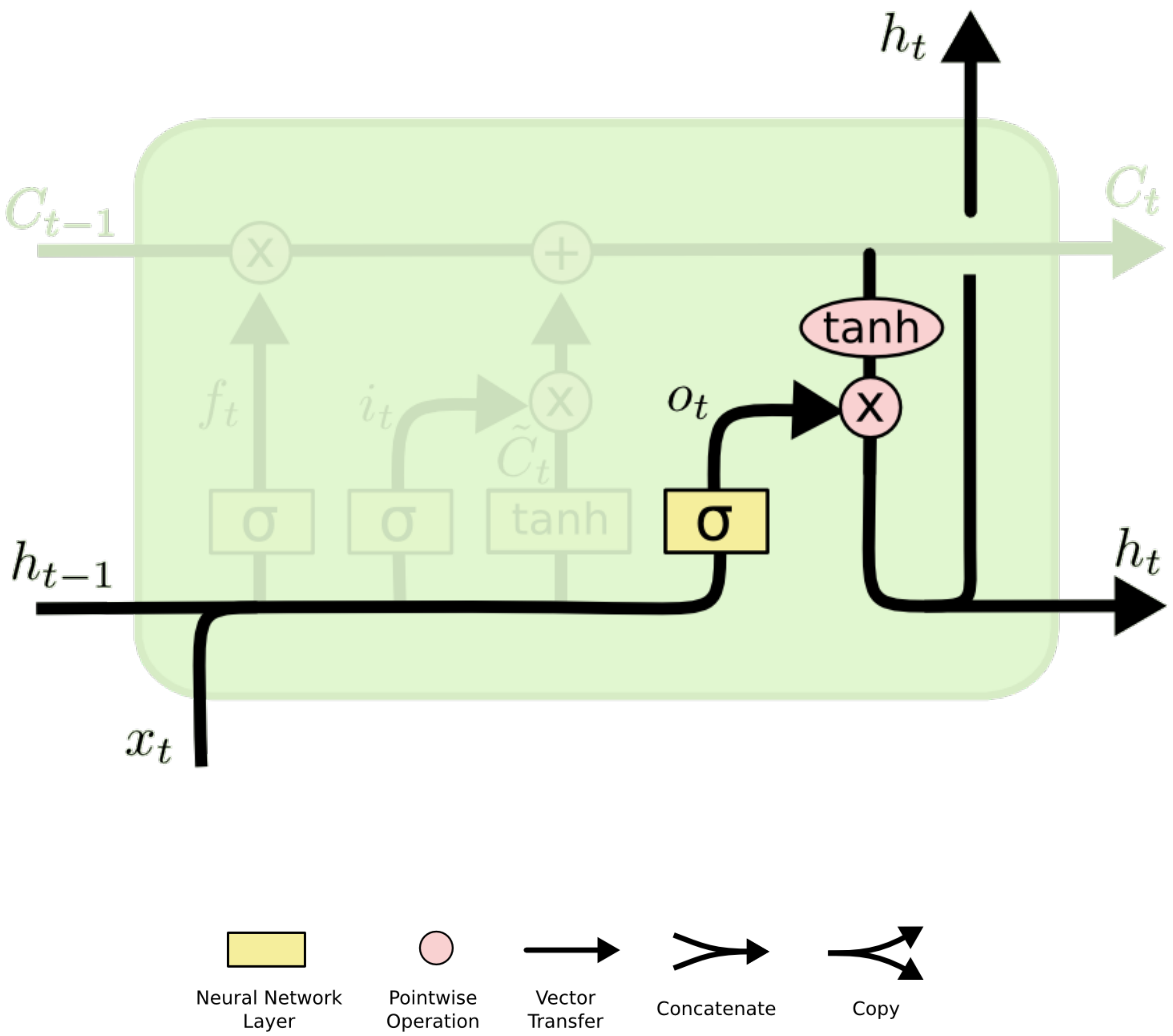


LSTM: The Ouput Gate

Finally, we need to decide what we're going to output:

1. The output gate decides what parts of the cell state we're going to output.

2. Pass updated $C^{<t>}$ state through $tanh$ and multiply it by the ouput gate, so we output only the parts we decided to.



LSTM Hidden Layer :

$$f^{<t>} = \sigma(W_{fh}h^{<t-1>} + W_{fx}x^{<t>} + b_f)$$

$$i^{<t>} = \sigma(W_{ih}h^{<t-1>} + W_{ix}x^{<t>} + b_i)$$

$$\tilde{C}^{<t>} = tanh(W_{ch}h^{<t-1>} + W_{cx}x^{<t>} + b_C)$$

$$C^{<t>} = f^{<t>} \odot C^{<t-1>} + i^{<t>} \odot \tilde{C}^{<t>}$$

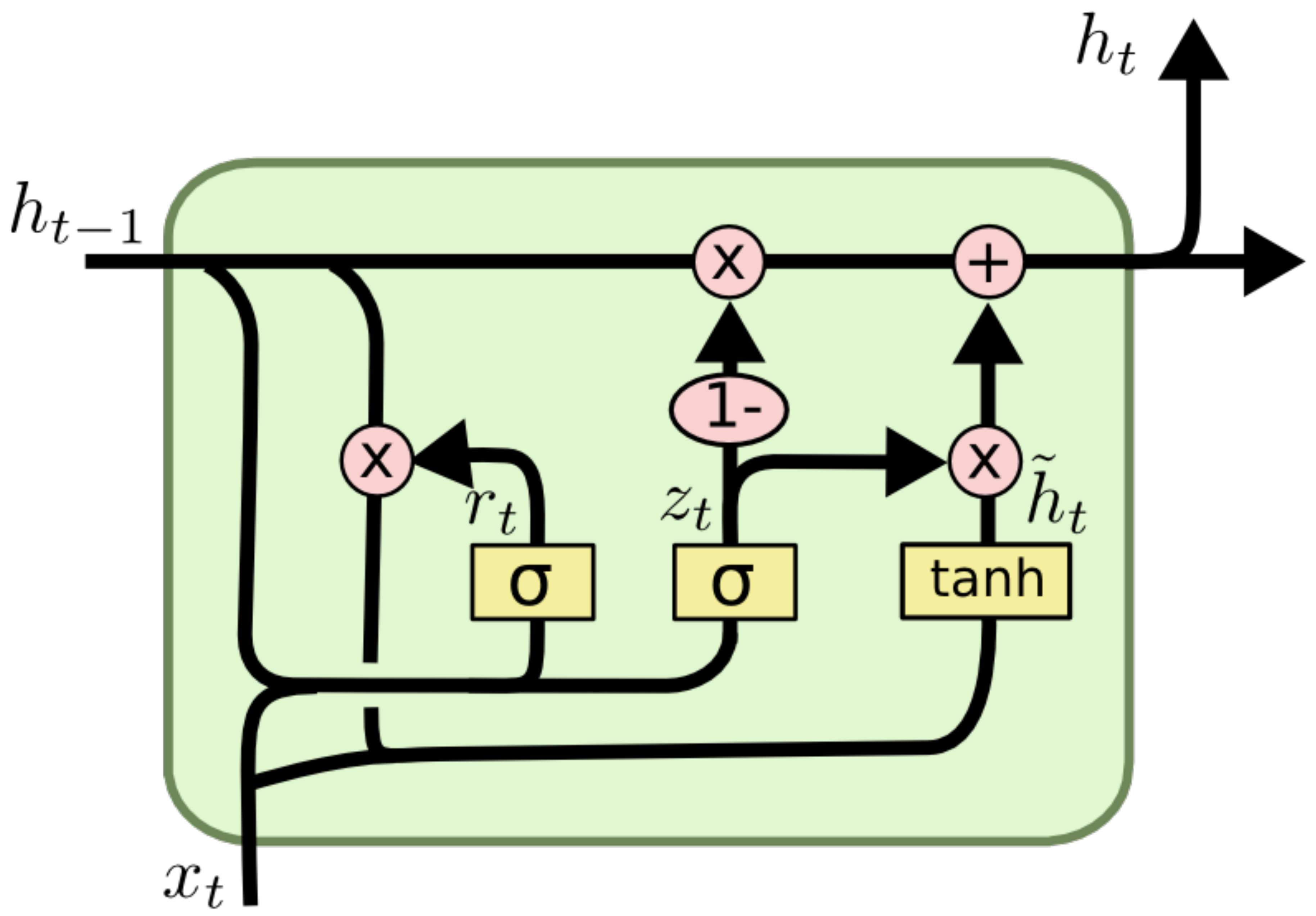
$$o^{<t>} = \sigma(W_{oh}h^{<t-1>} + W_{ox}x^{<t>} + b_o)$$

$$h^{<t>} = o_t \odot tanh(C^{<t>})$$

UFV

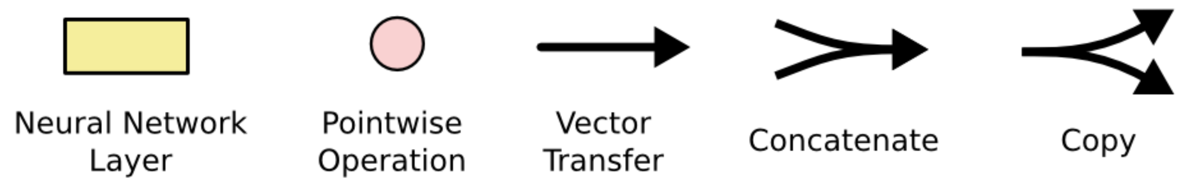
Gated Recurrent Unit (GRU)

The GRU combines the forget and input gates into a single "update gate" and merges the cell state with hidden state:



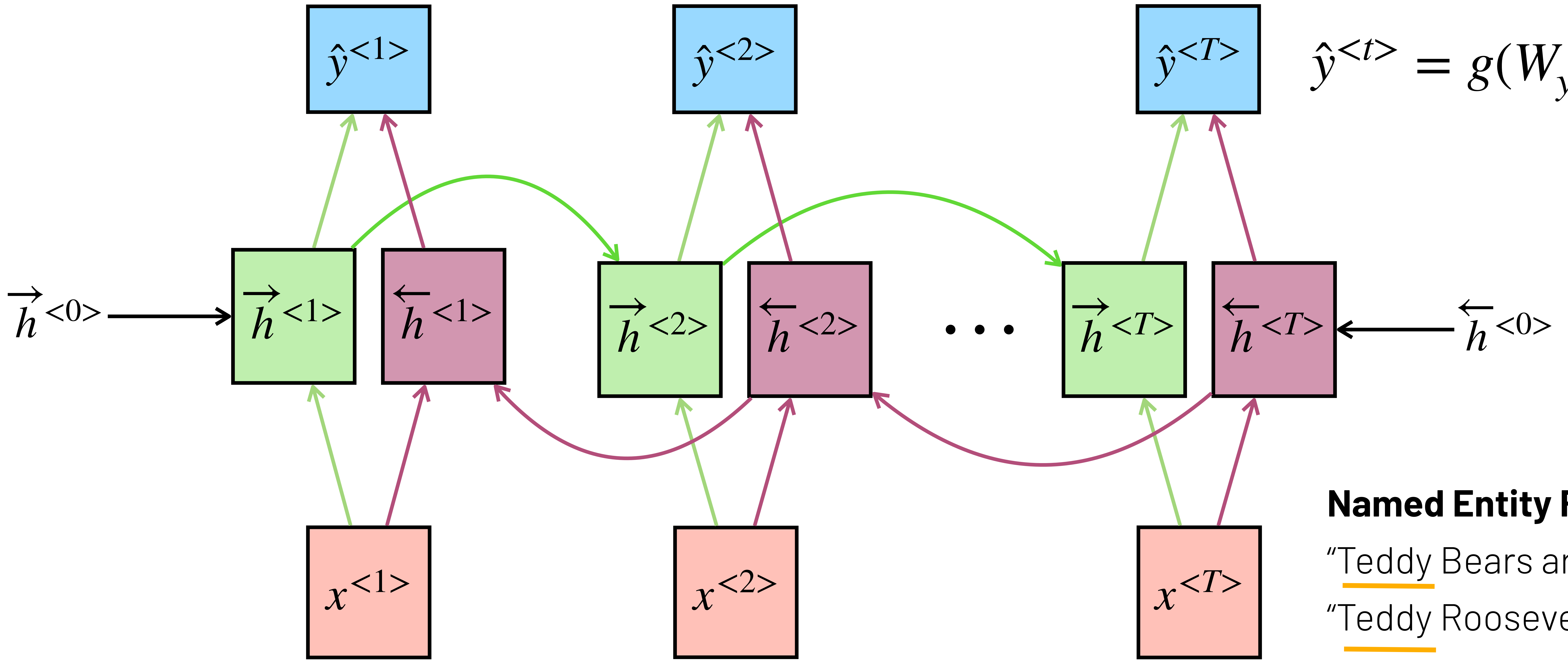
GRU Hidden Layer :

$$z^{<t>} = \sigma(W_{zh}h^{<t-1>} + W_{zx}x^{<t>} + b_z)$$
$$r^{<t>} = \sigma(W_{rh}h^{<t-1>} + W_{rx}x^{<t>} + b_r)$$
$$\tilde{h}^{<t>} = tanh(W_{hh}(r^{<t>} \odot h^{<t-1>}) + W_{hx}x^{<t>} + b_h)$$
$$h^{<t>} = (1 - z^{<t>}) \odot h^{<t-1>} + z^{<t>} \odot \tilde{h}^{<t>}$$



Bidirectional RNN

Bidirectional RNNs process sequences from **forward** and from **backward** to build a context in both directions.



$$\hat{y}^{<t>} = g(W_y[\overrightarrow{h}^{<1>}, \overleftarrow{h}^{<1>}] + b_y)$$

Hidden layer can have any recurrent structure:

- RNNs, LSTMs, GRUs ....

Named Entity Recognition

"Teddy Bears are on sale!"

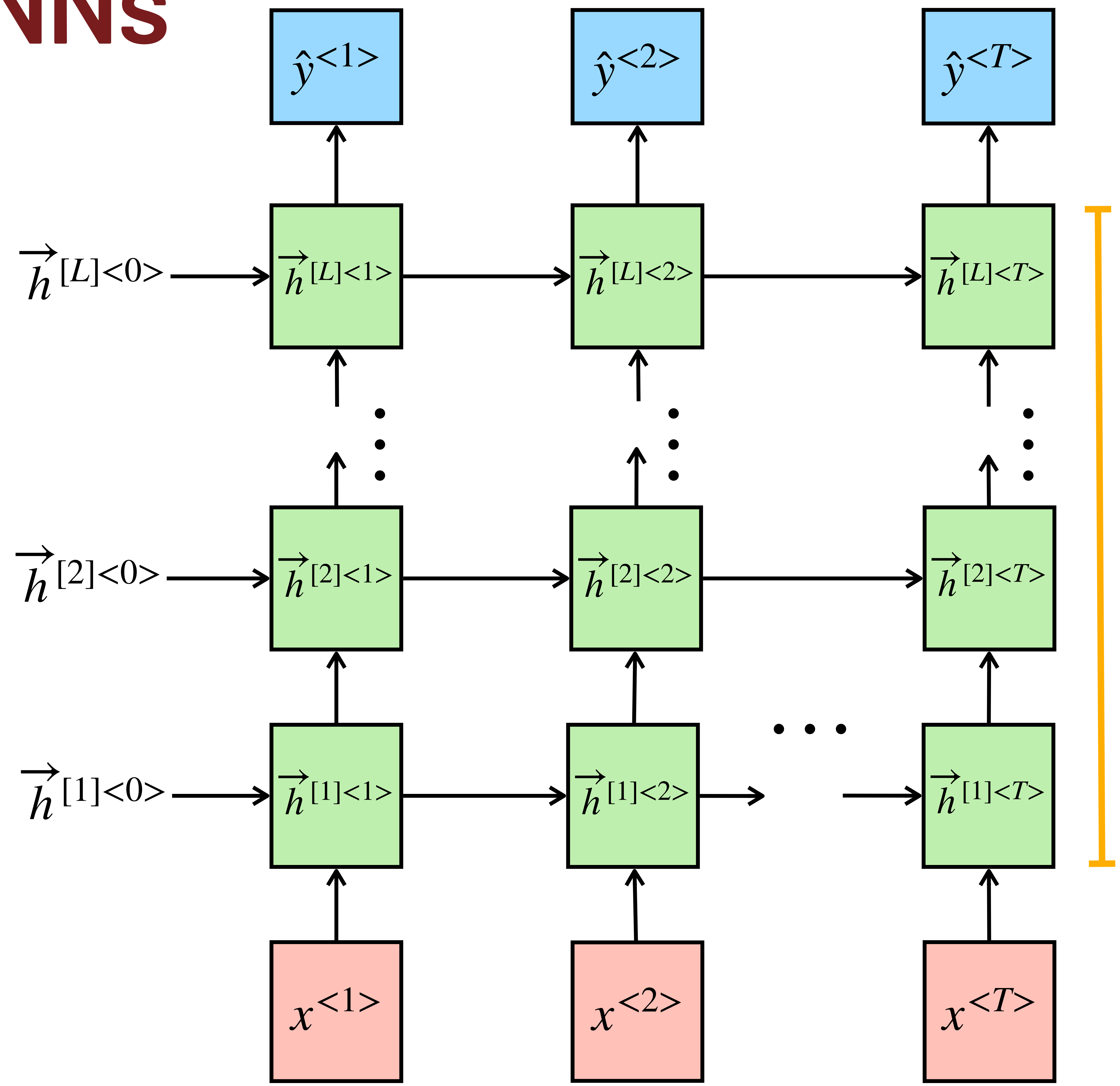
"Teddy Roosevelt was a great president!"

?

Schuster, Mike, and Kuldip K. Paliwal. "Bidirectional recurrent neural networks." IEEE transactions on Signal Processing 45.11 (1997): 2673-2681.

UFV

Deep RNNs



To create deeper RNNs, we can stack hidden layers on top of each other

Hidden layer can have any recurrent structure:

- RNNs, LSTMs, GRUs ....
- Biderectional

UFV

Next Lecture

L15: Word Embeddings

Learning vector representations for words