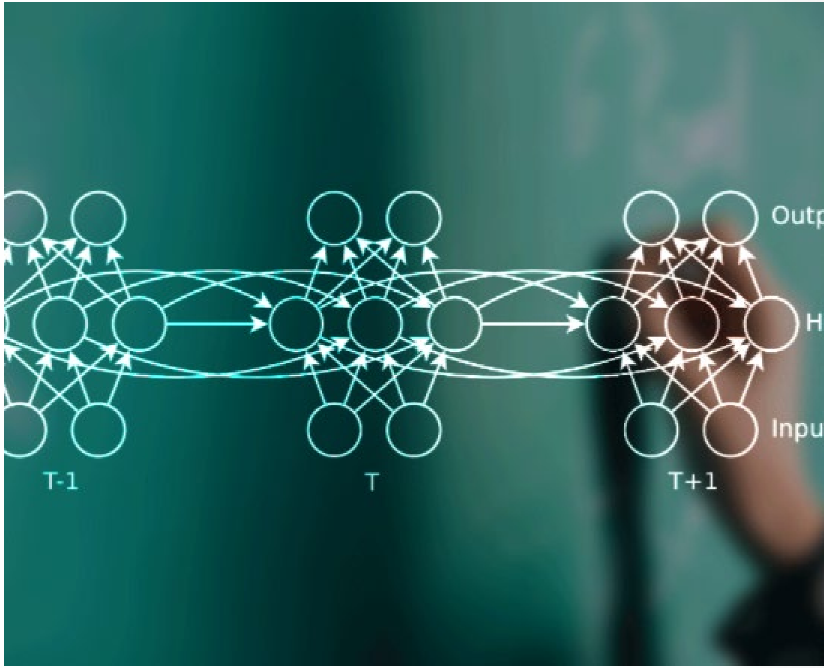
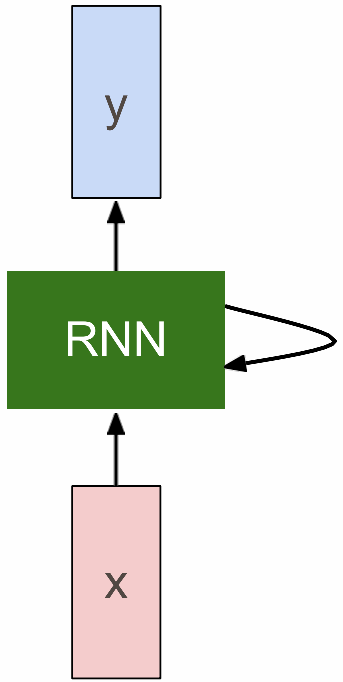
# RECURRENT NEURAL NETWORKS



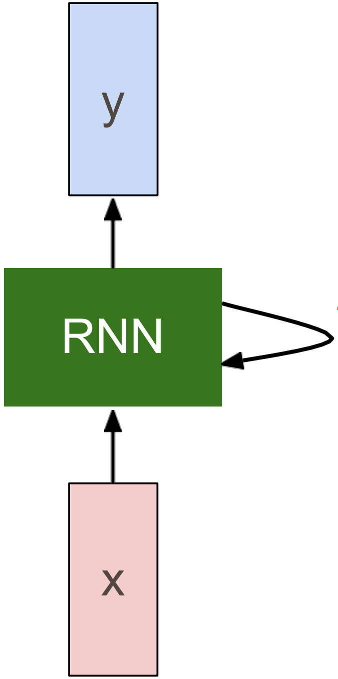

Chih-Chung Hsu (許志仲)
Institute of Data Science
National Cheng Kung University
https://cchsu.info

# Recurrent Neural Network



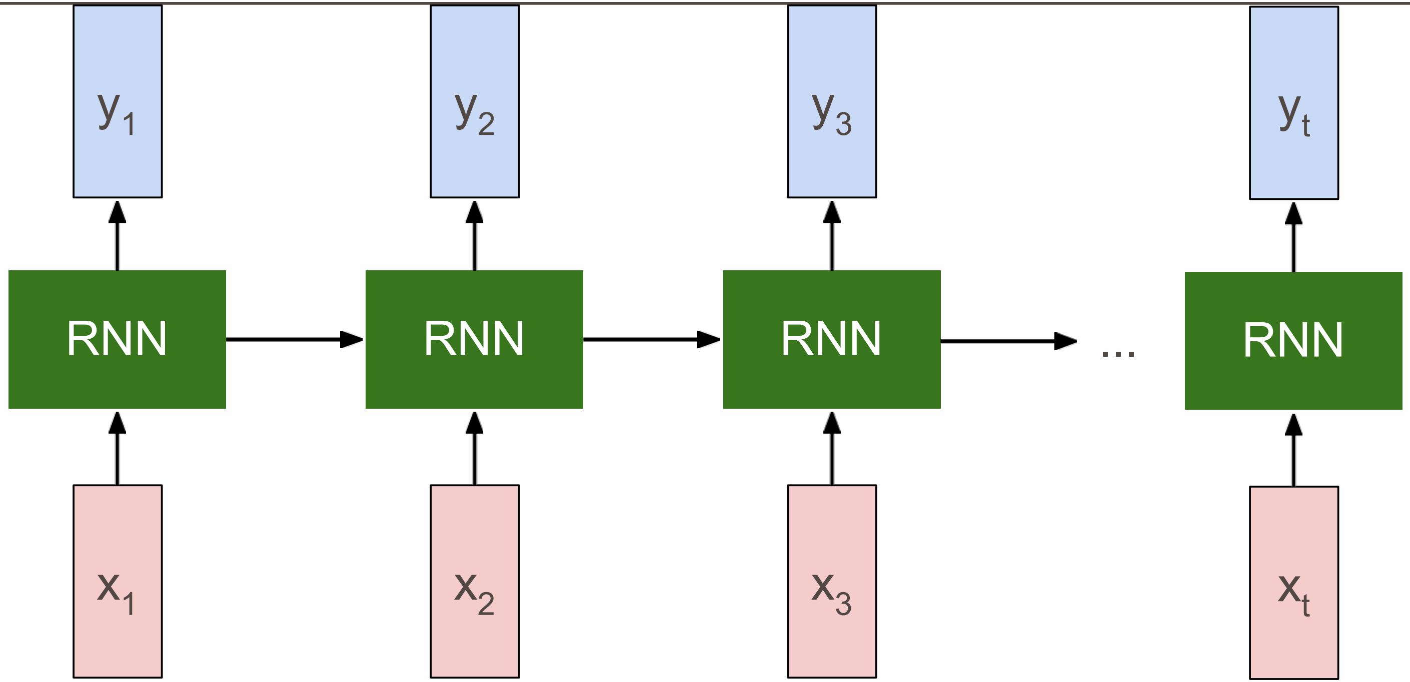

partial credit by CS311n

# Recurrent Neural Network




Key idea: RNNs have an "internal state" that is updated as a sequence is processed

# Recurrent Neural Network




partial credit by CS311n

# Recurrent Neural Network

We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

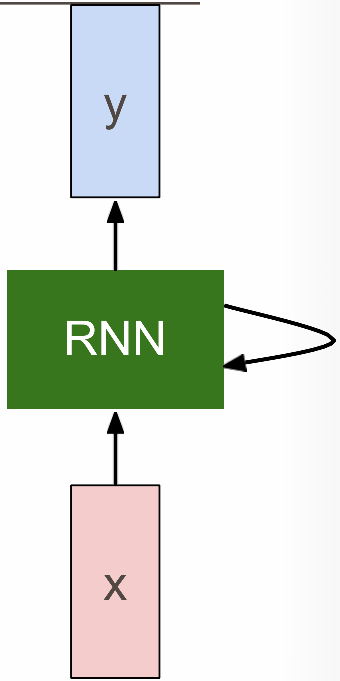
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state

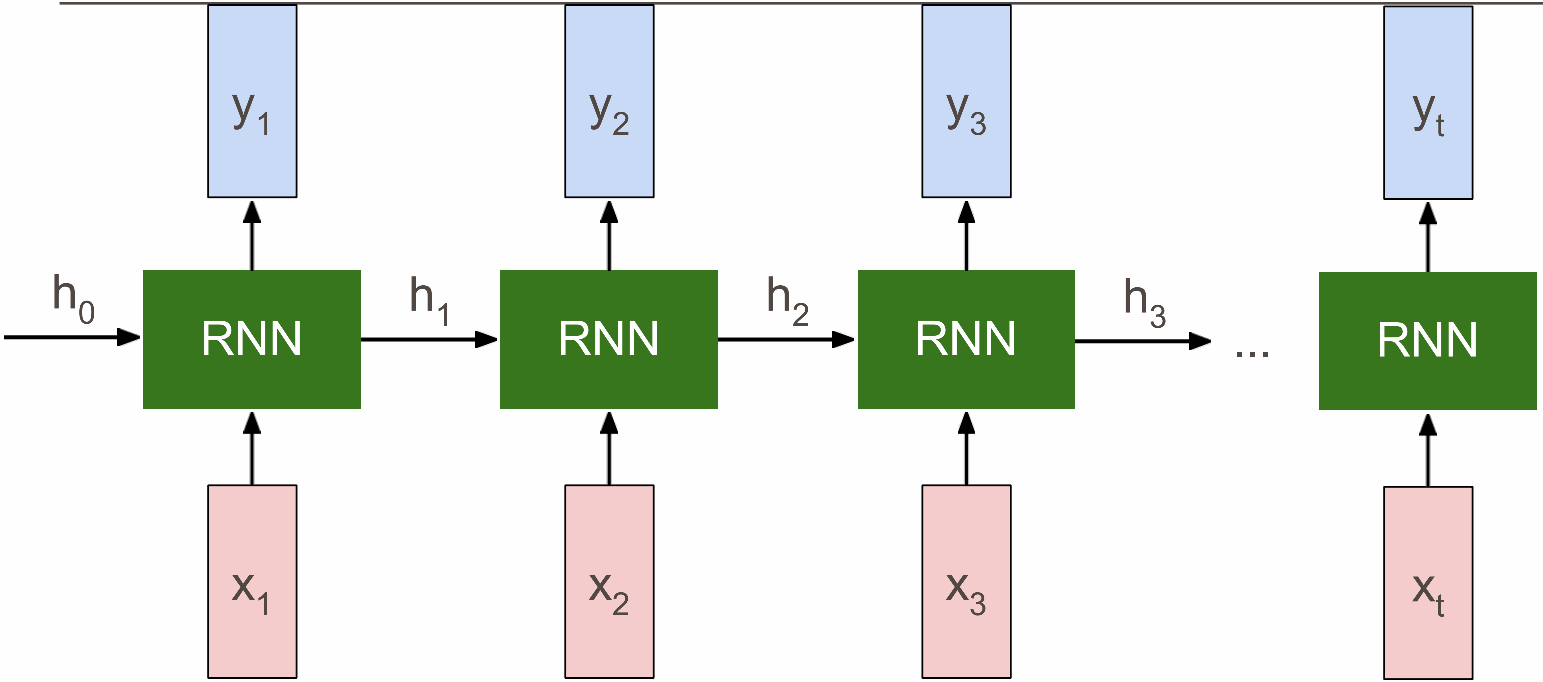
old state

input vector at some time step

some function with parameters W



# Recurrent Neural Network

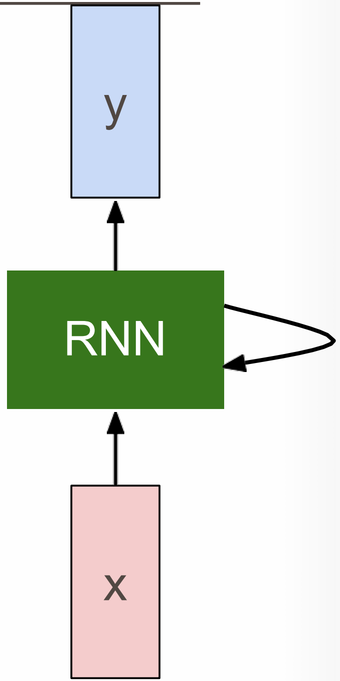



partial credit by CS311n

# Recurrent Neural Network

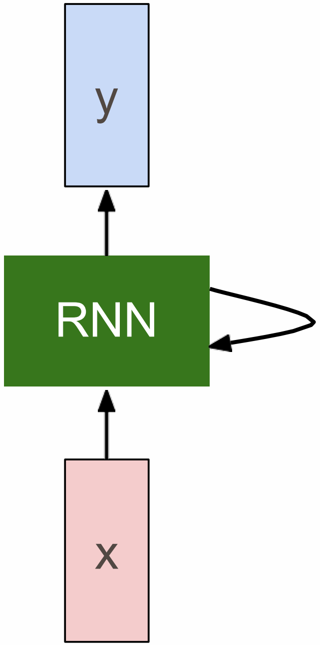
We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

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




Notice: the same function and the same set of parameters are used at every time step.

# (Simple) Recurrent Neural Network




The state consists of a single "hidden" vector h:

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

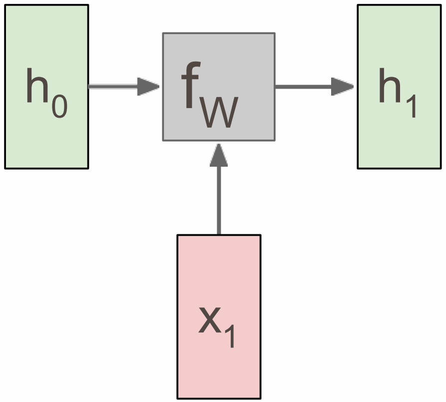
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

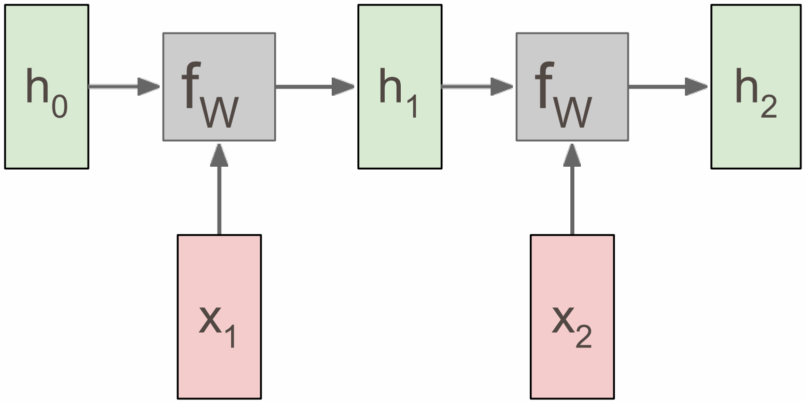
Sometimes called a "Vanilla RNN" or an "Elman RNN" after Prof. Jeffrey Elman

partial credit by CS311n

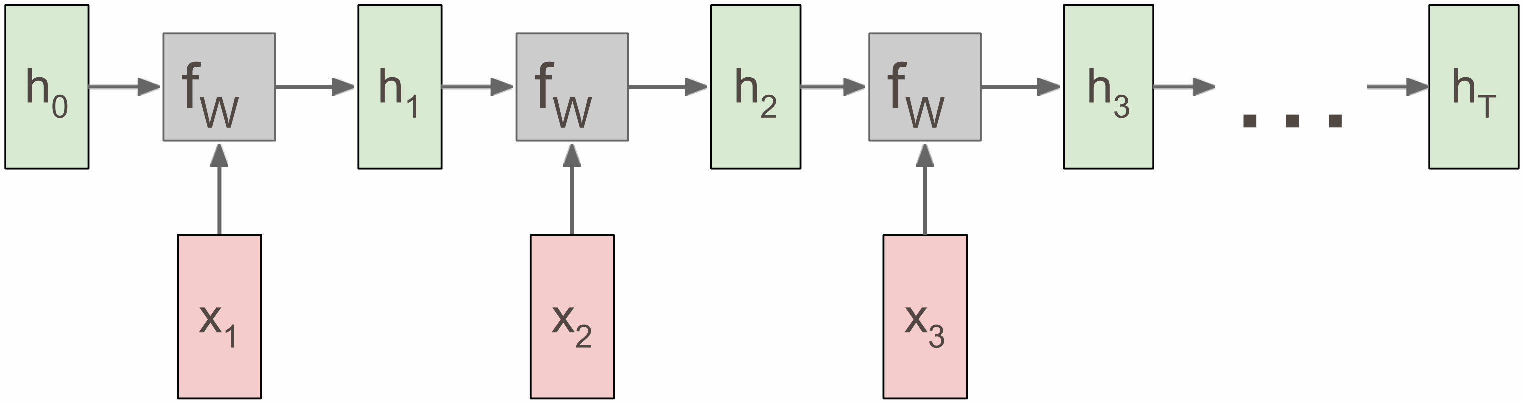
# RNN: Computational Graph



# RNN: Computational Graph

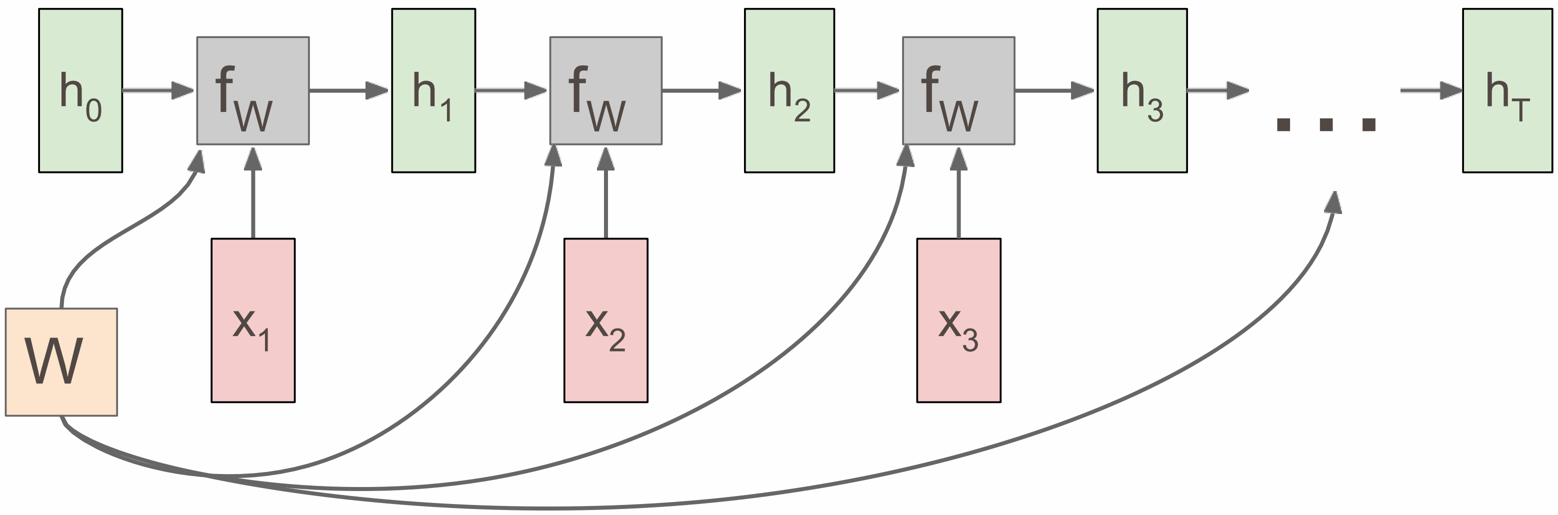


# RNN: Computational Graph

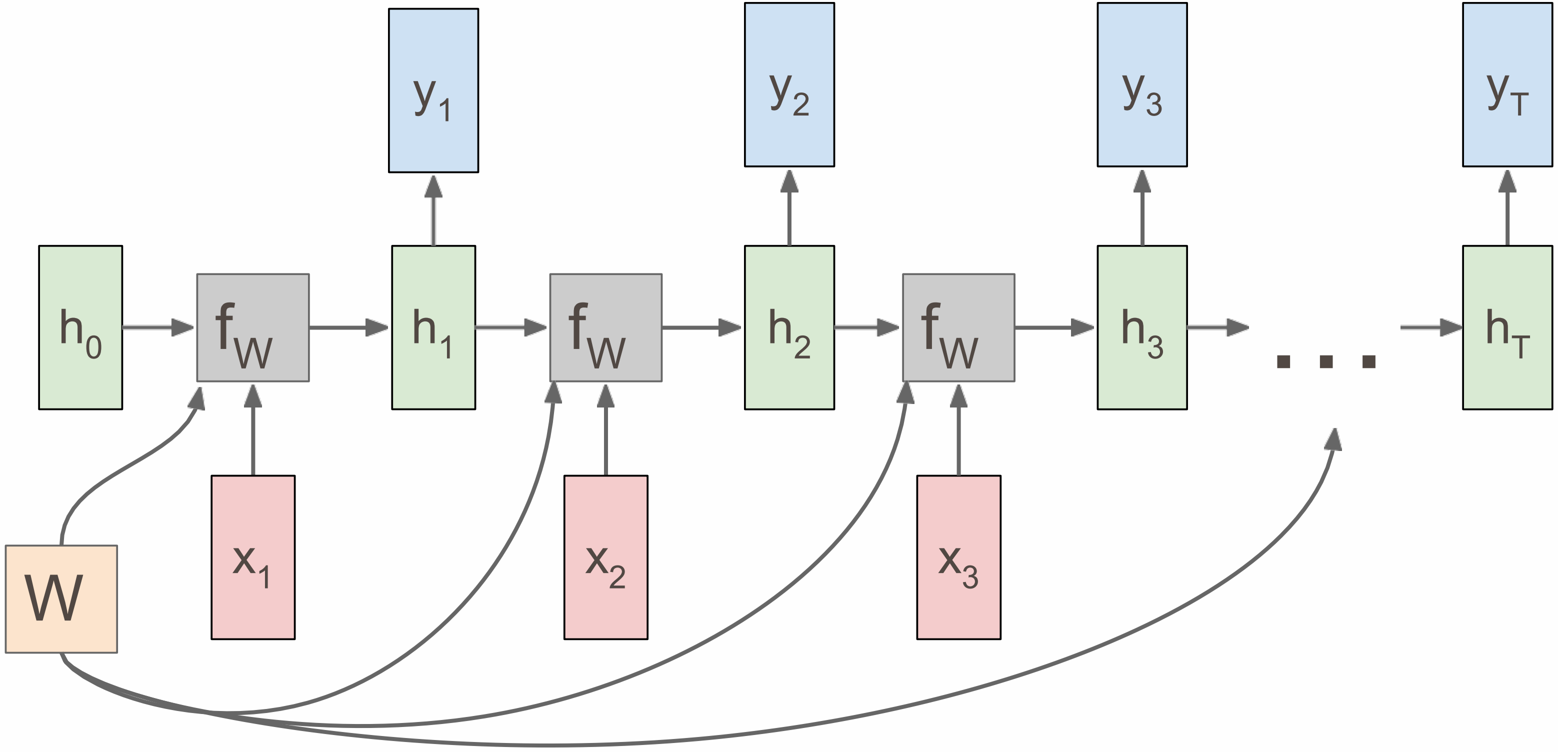


# RNN: Computational Graph

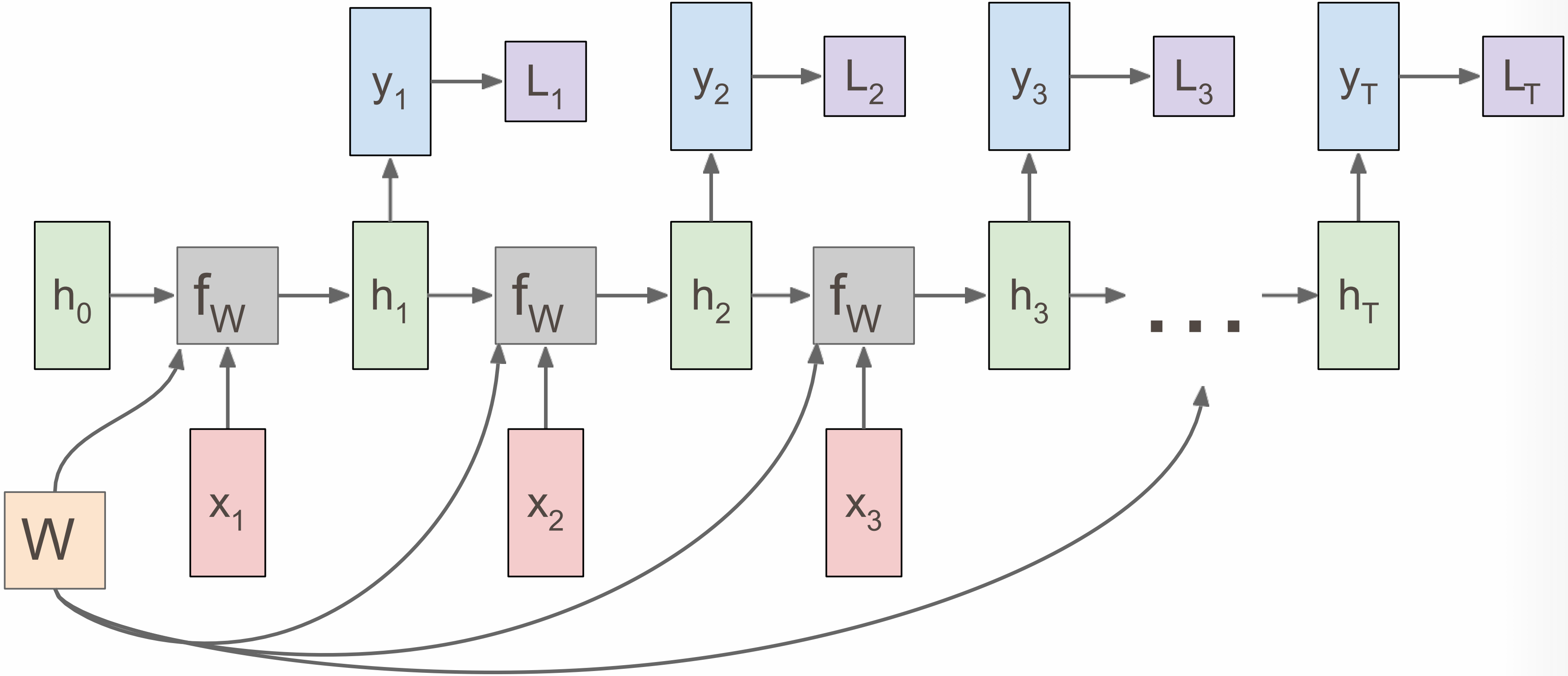
Re-use the same weight matrix at every time-step



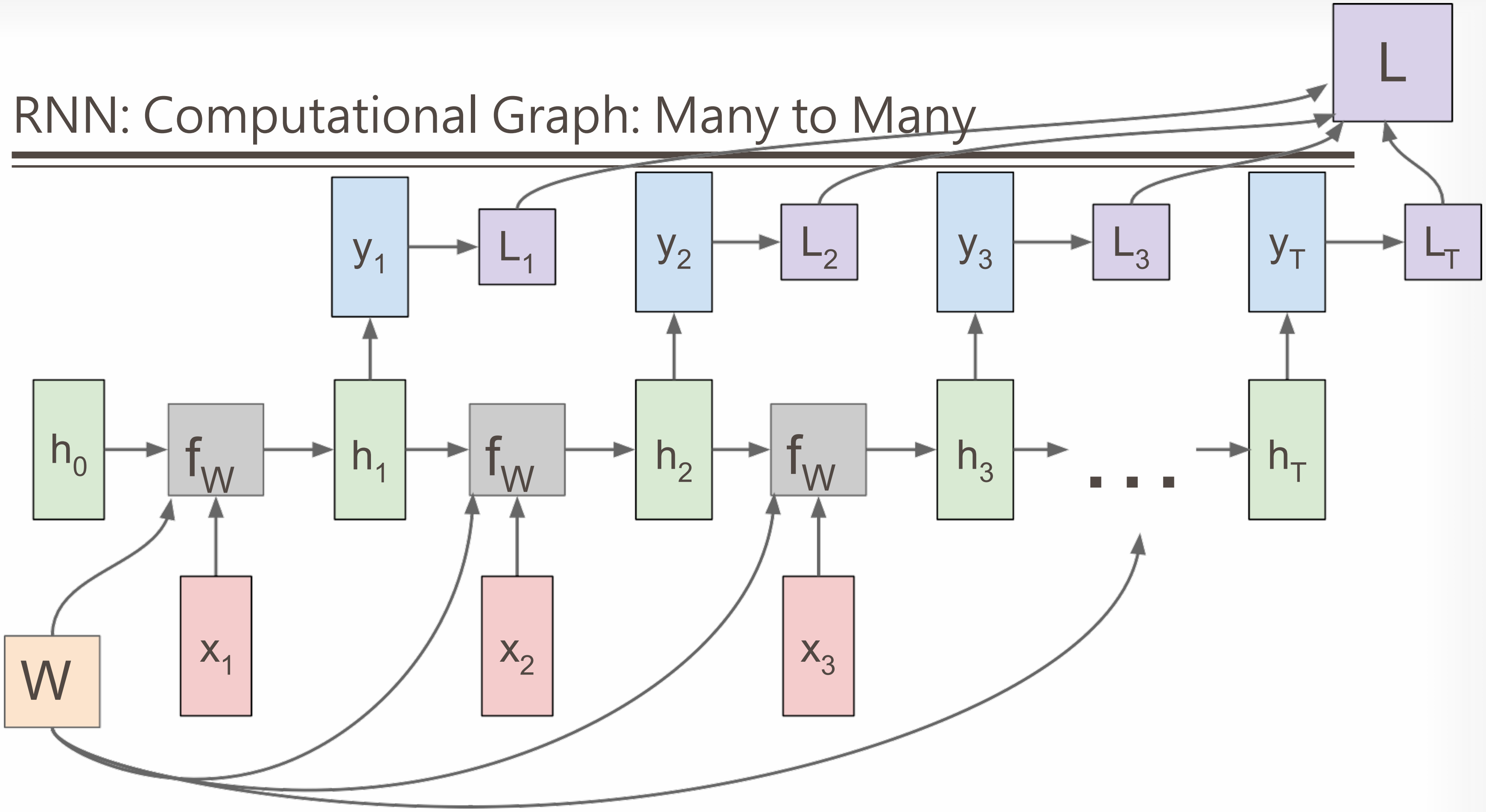
# RNN: Computational Graph: Many to Many



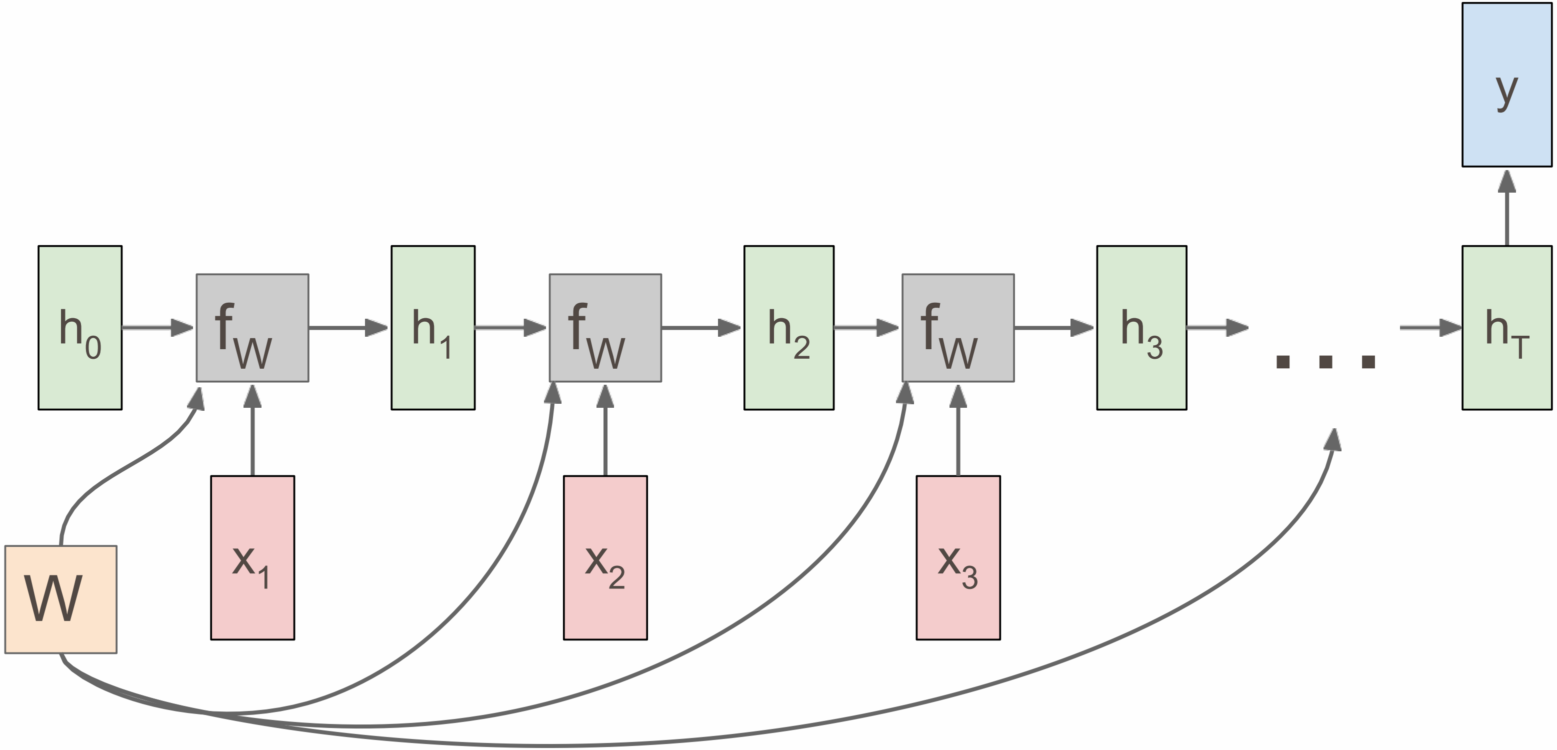
# RNN: Computational Graph: Many to Many



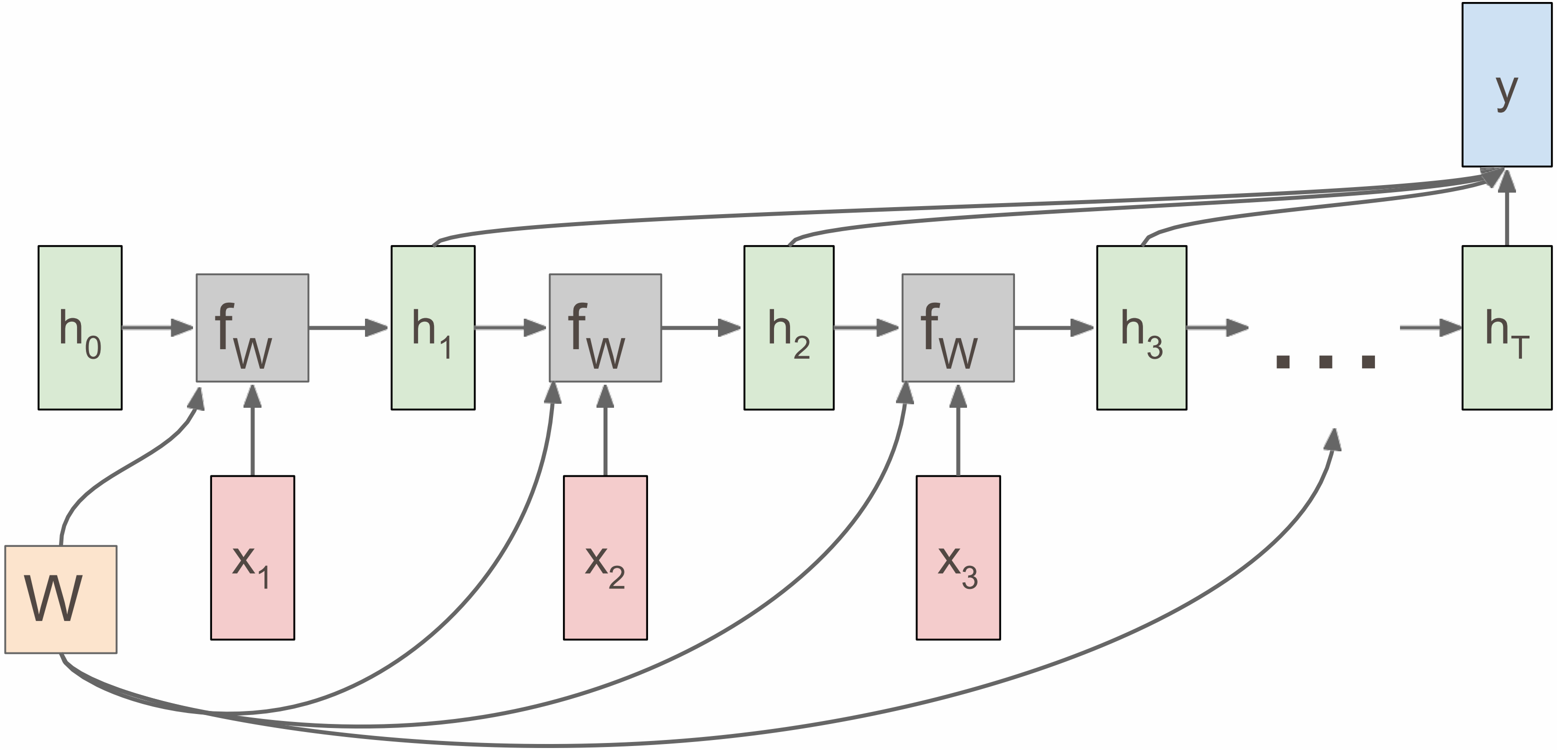
# RNN: Computational Graph: Many to Many



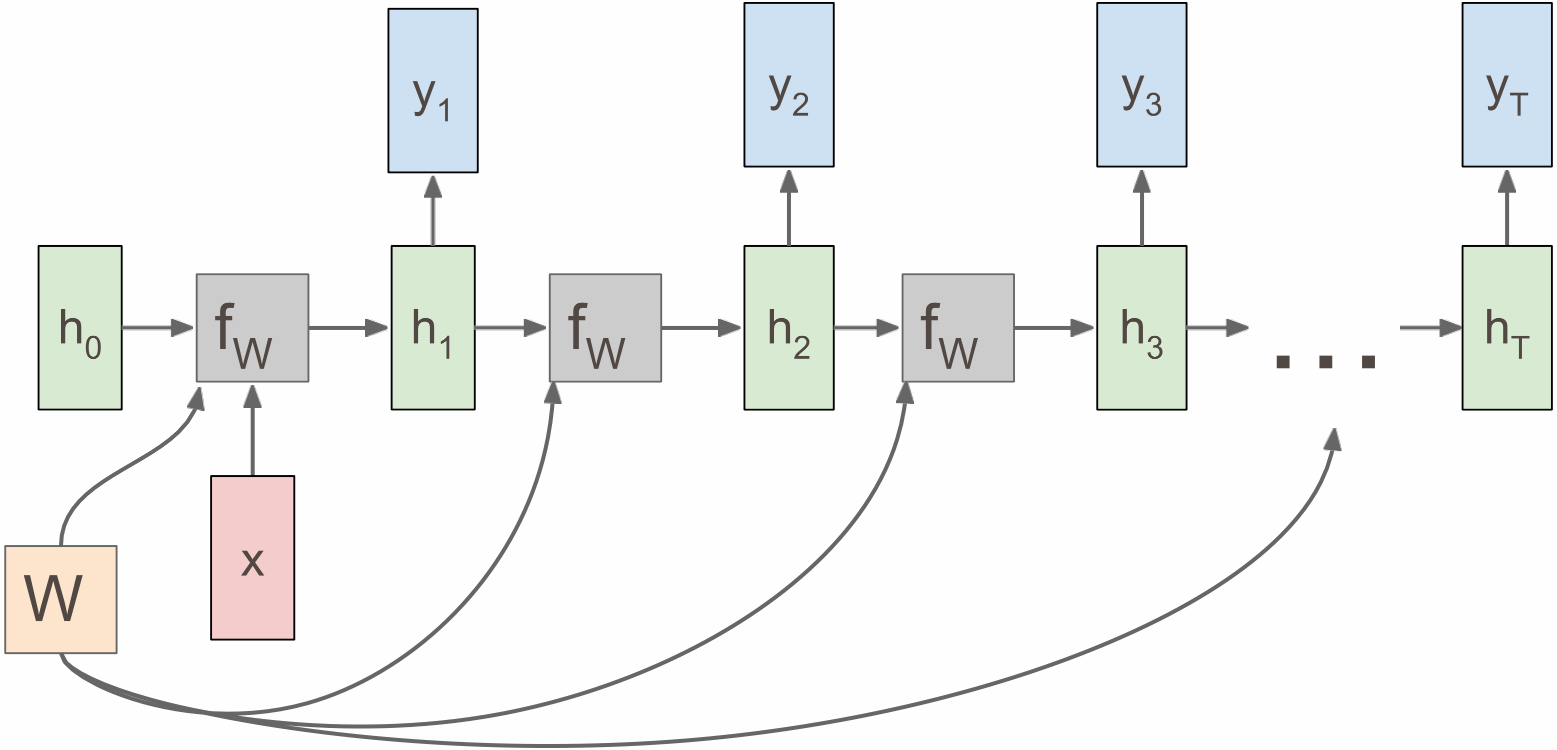
# RNN: Computational Graph: Many to One



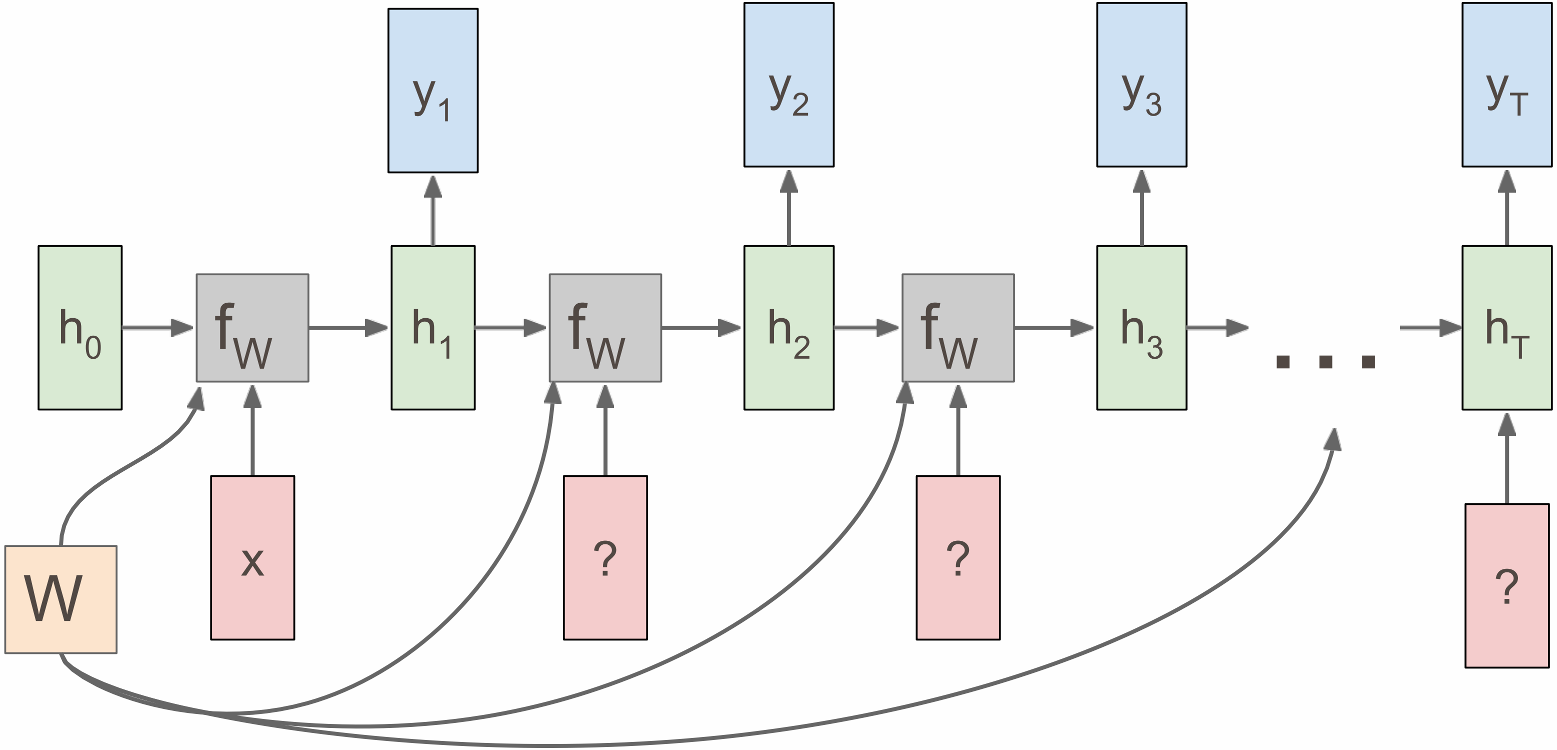
# RNN: Computational Graph: Many to One



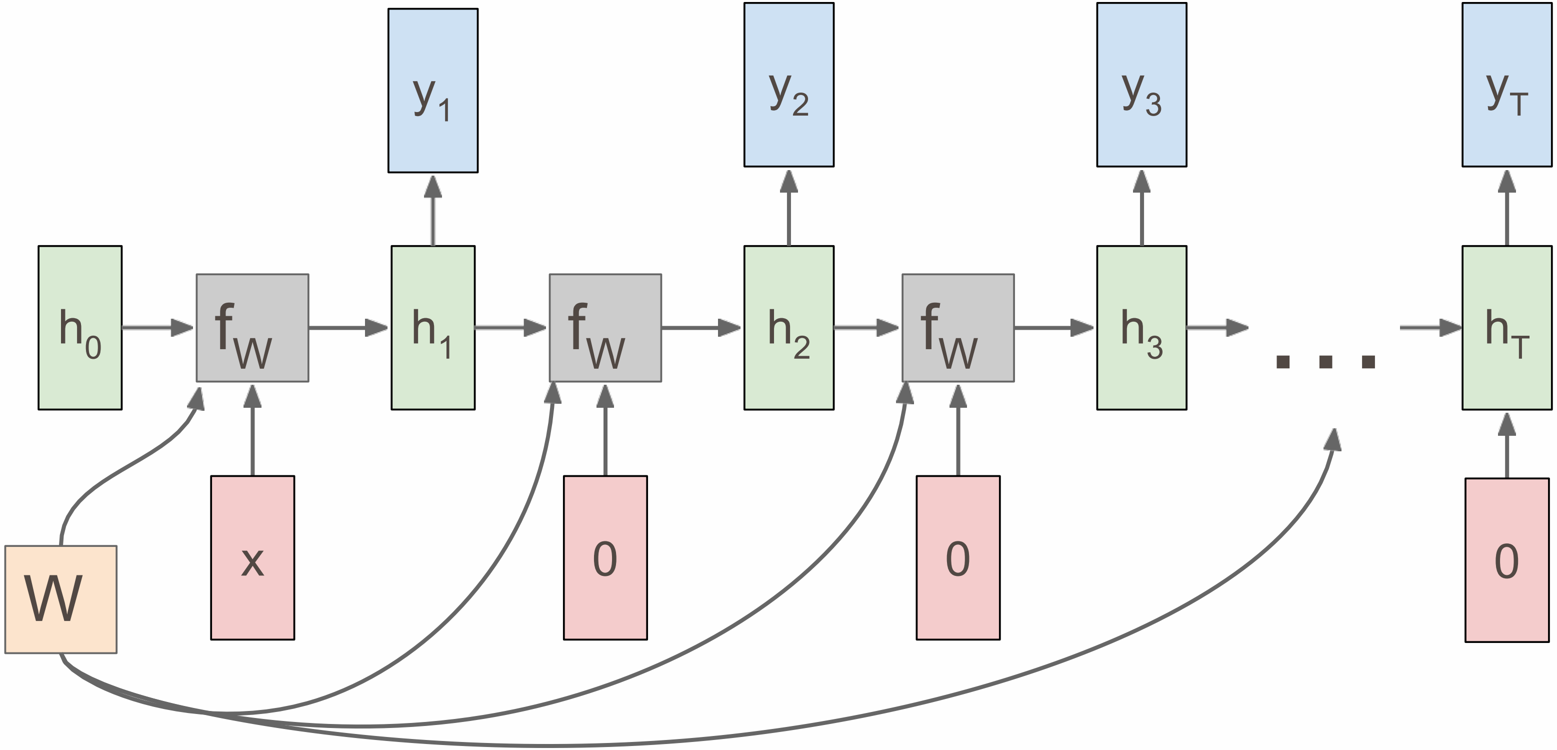
# RNN: Computational Graph: One to Many



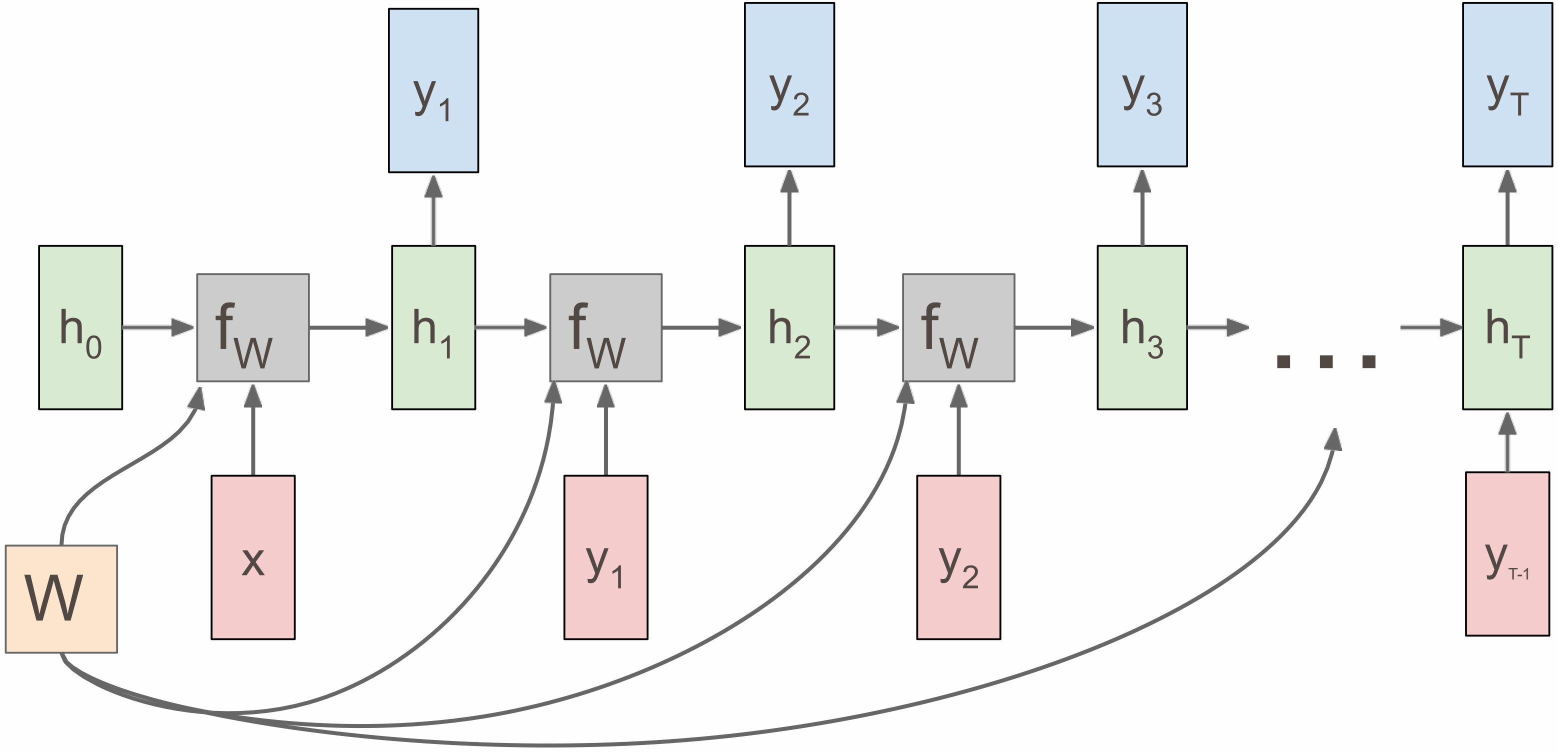
# RNN: Computational Graph: One to Many



# RNN: Computational Graph: One to Many

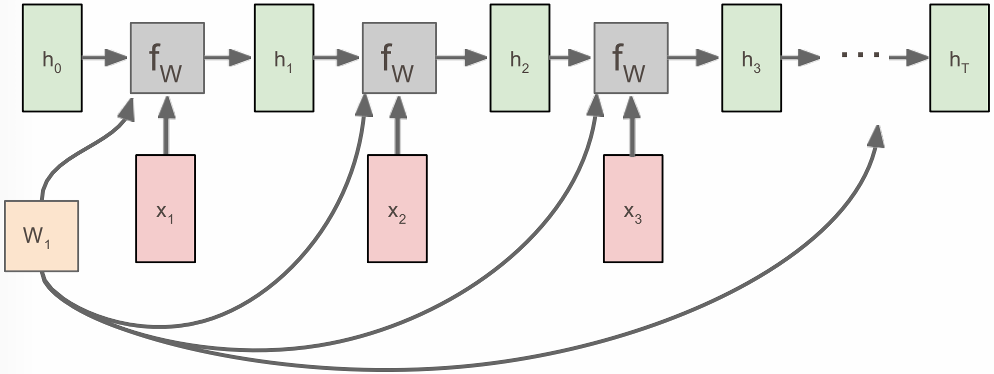


# RNN: Computational Graph: One to Many



# Sequence to Sequence: Many-to-one + one-to-many

**Many to one**: Encode input sequence in a single vector

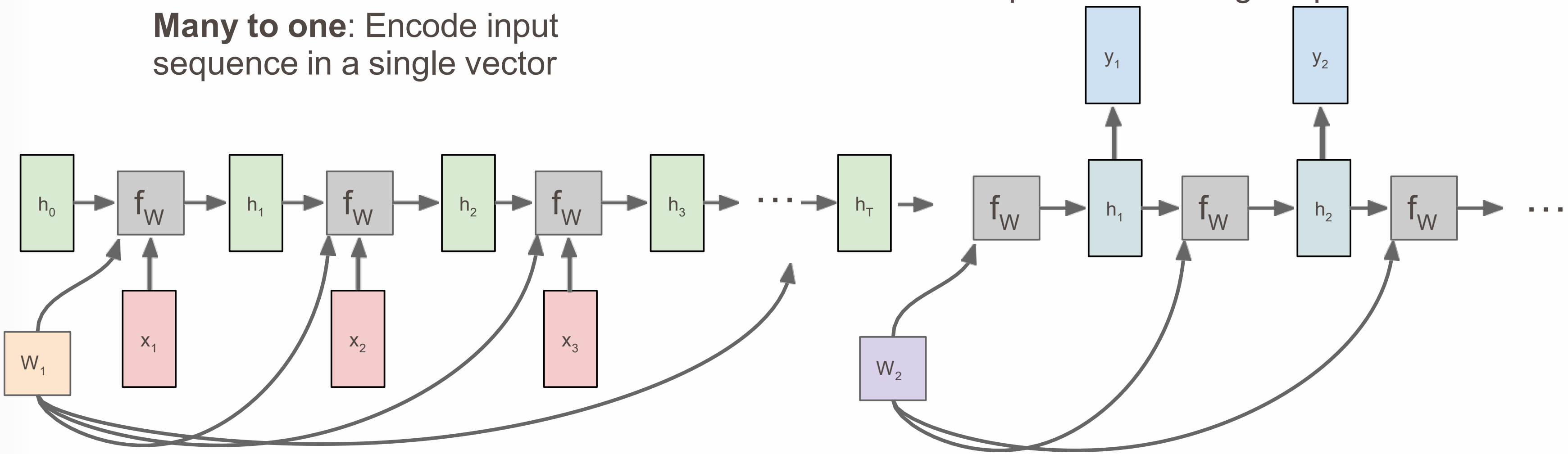



Sutskever et al, “Sequence to Sequence Learning with Neural Networks”, NIPS 2014

# Sequence to Sequence: Many-to-one + one-to-many

**One to many**: Produce output sequence from single input vector

**Many to one**: Encode input sequence in a single vector

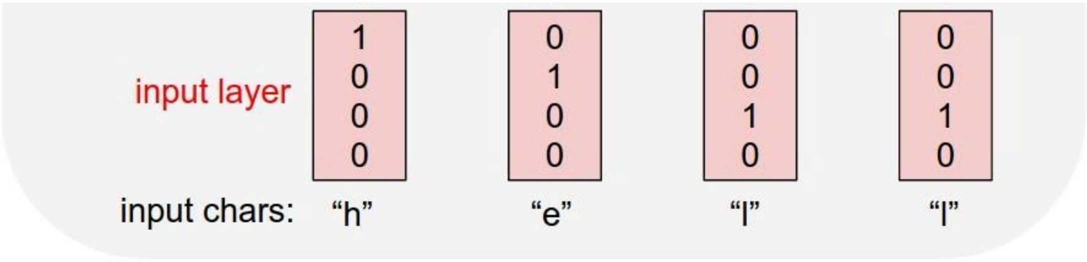



Sutskever et al, "Sequence to Sequence Learning with Neural Networks", NIPS 2014

## Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

Example training sequence:
**“hello”**

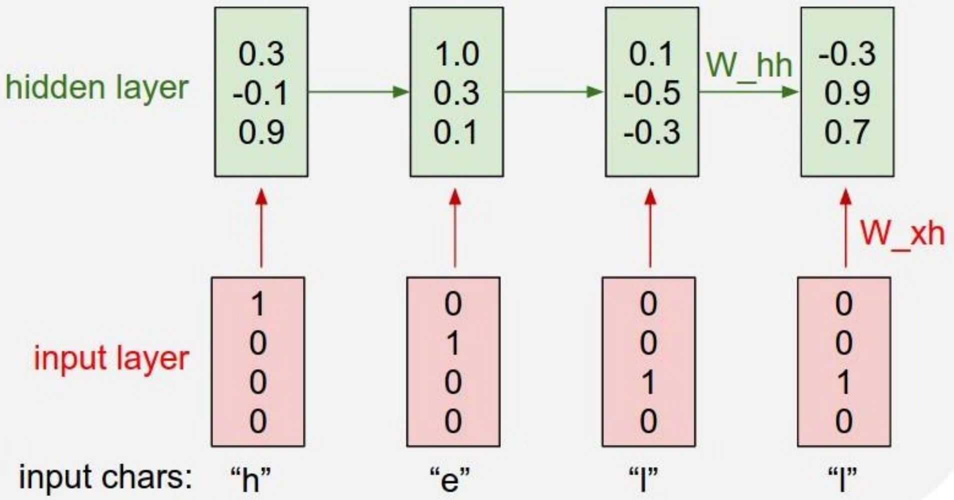


Example: Character-level Language Model

Vocabulary: [h,e,l,o]

Example training sequence: **“hello”**

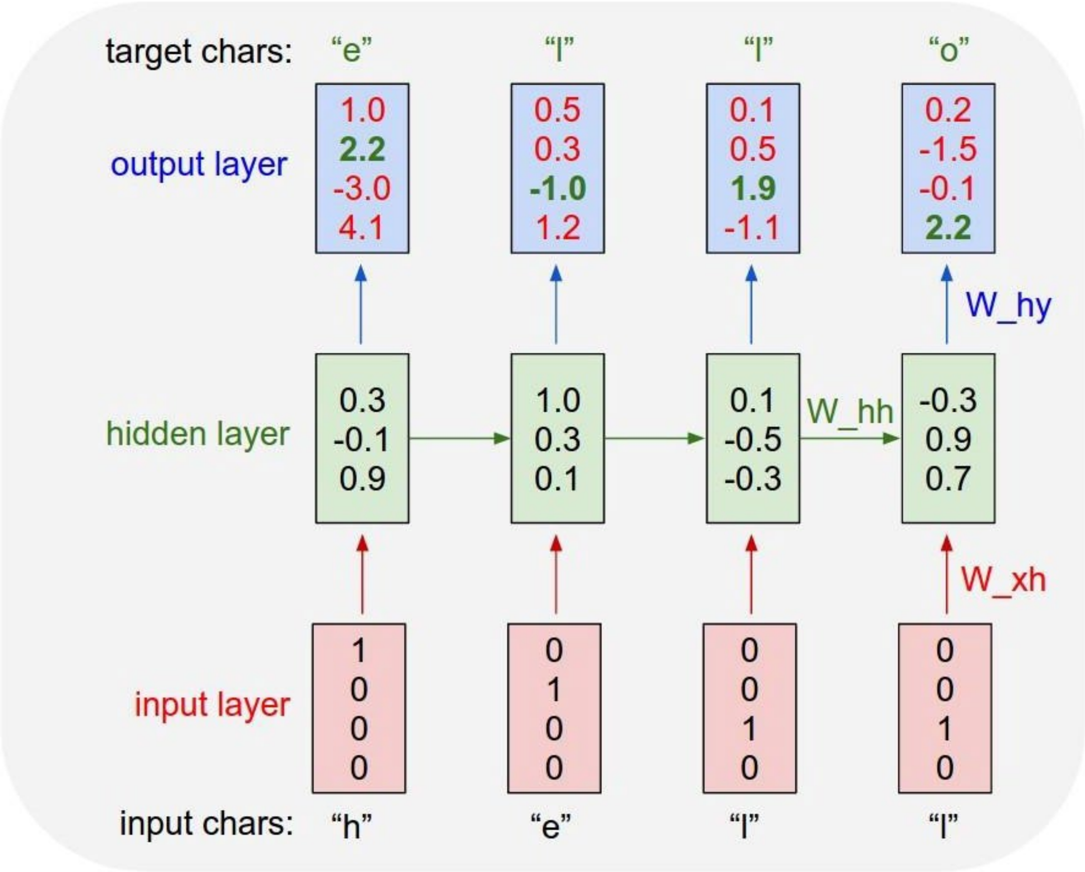
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$



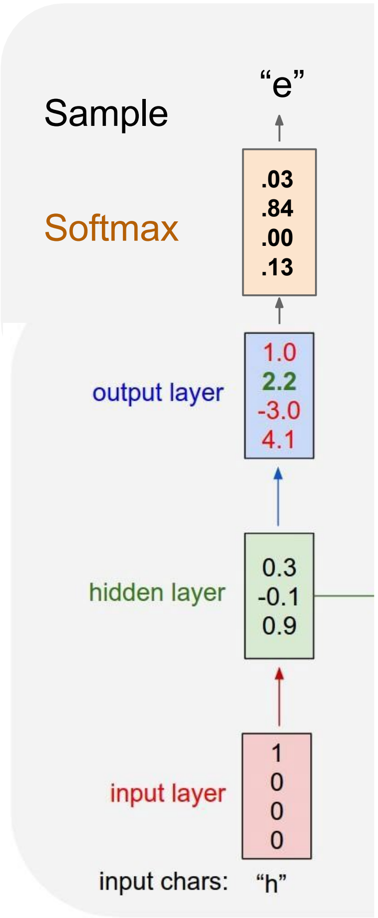
Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

Example training sequence:
**“hello”**



# Example: Character-level Language Model



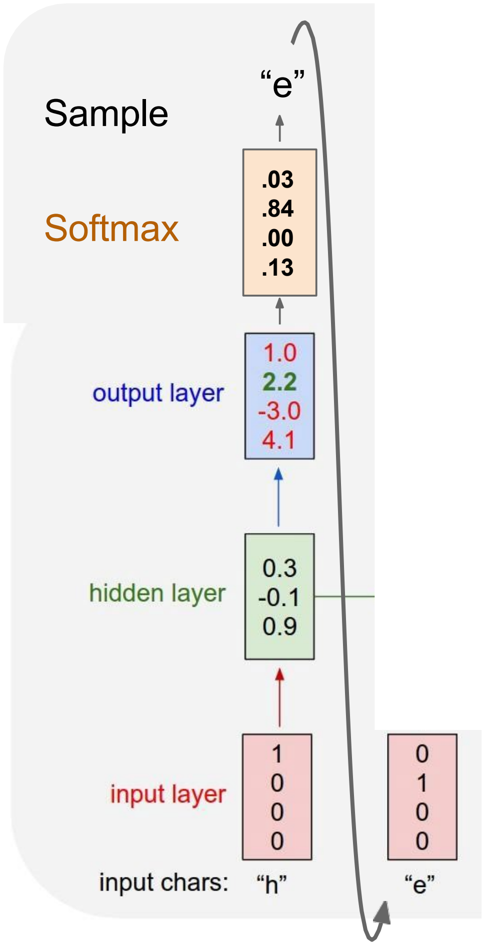

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model

# Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

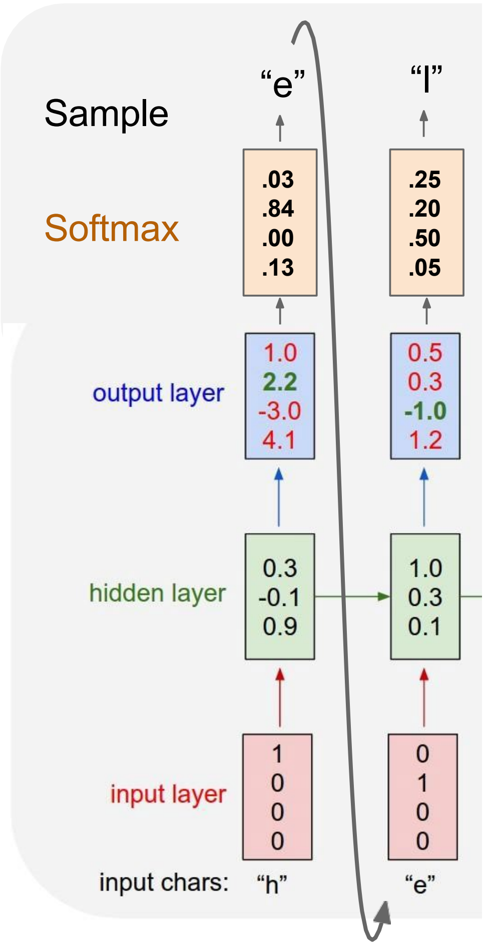
At test-time sample characters one at a time, feed back to model



# Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

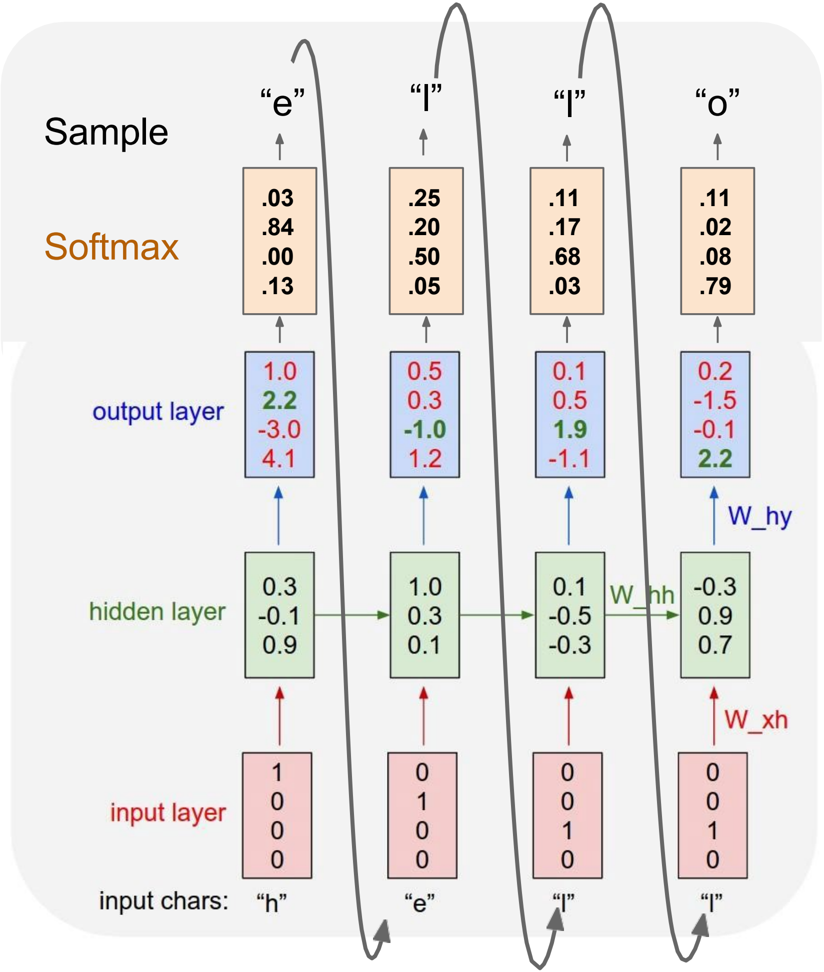
At test-time sample characters one at a time, feed back to model



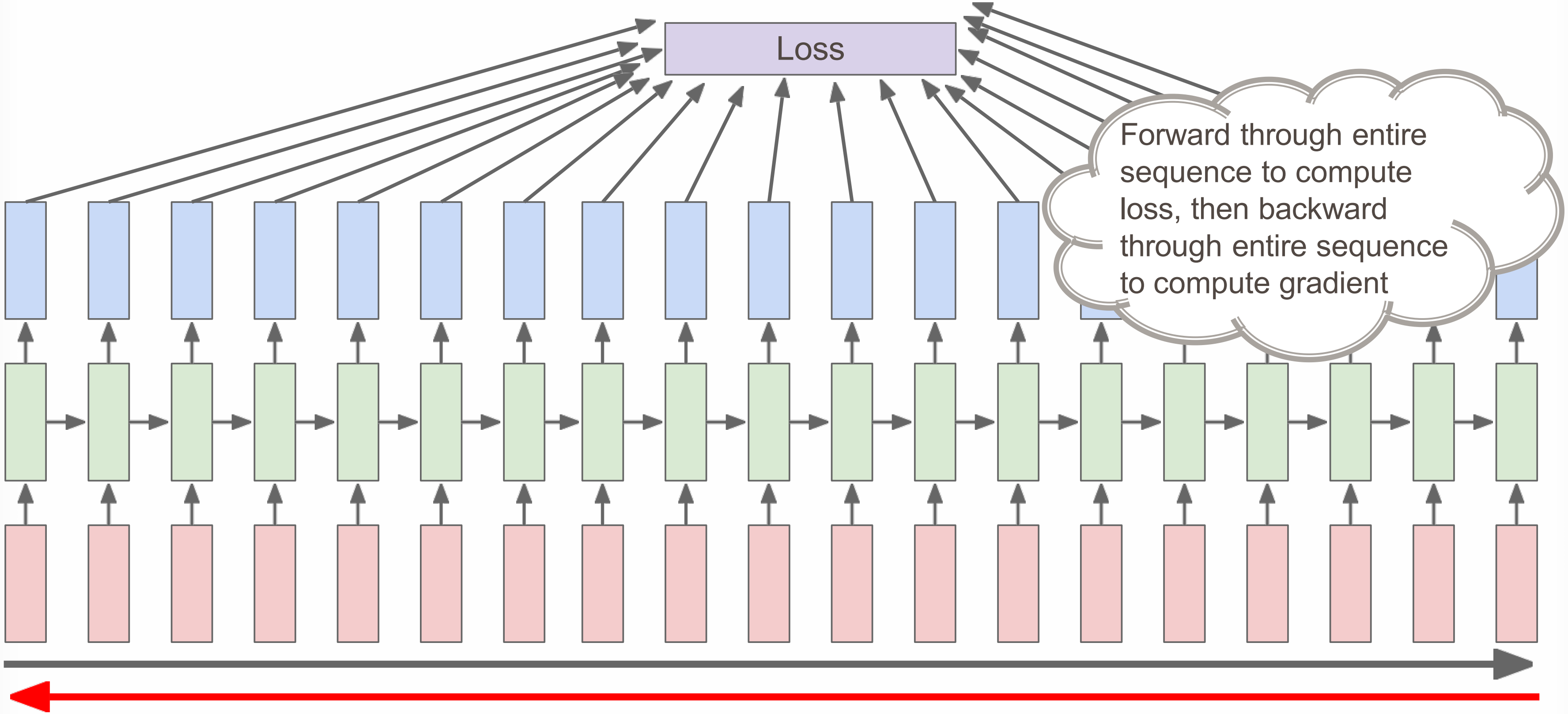
Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

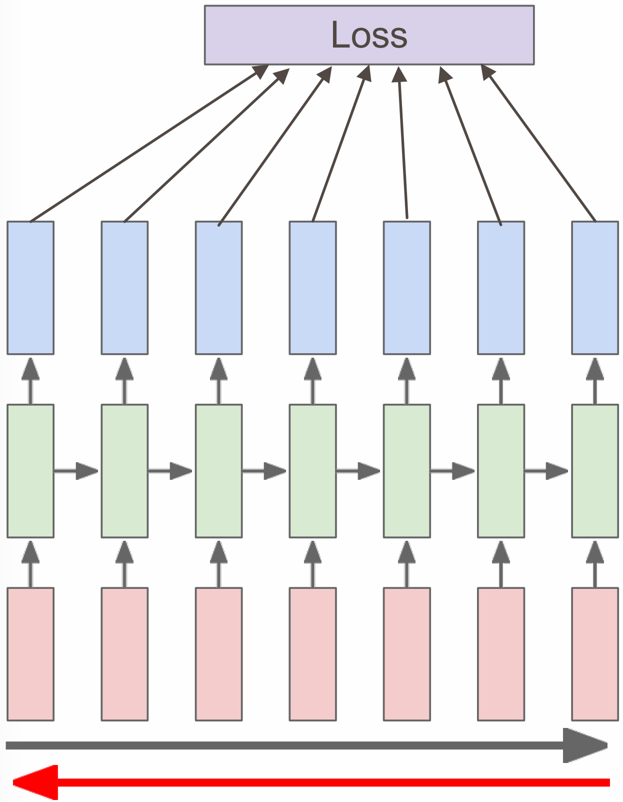
At test-time sample characters one at a time, feed back to model



# Backpropagation through time

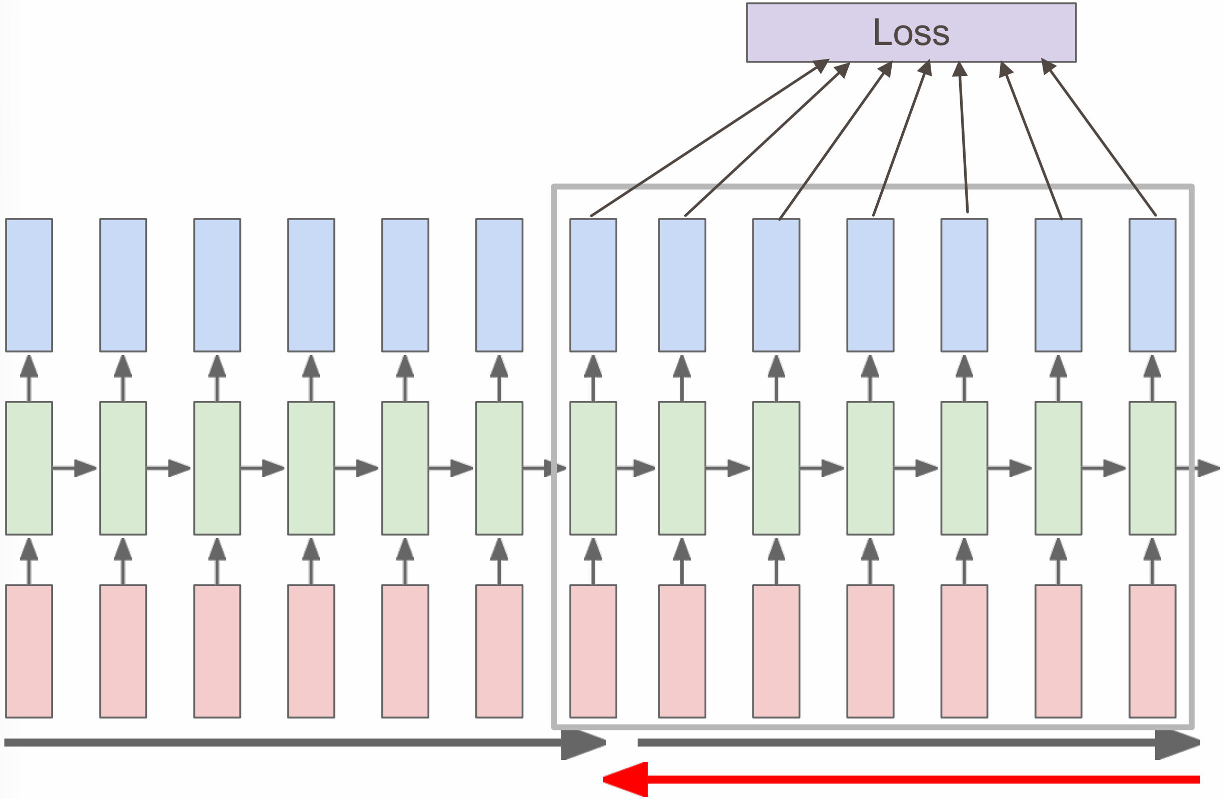


# Truncated Backpropagation through time



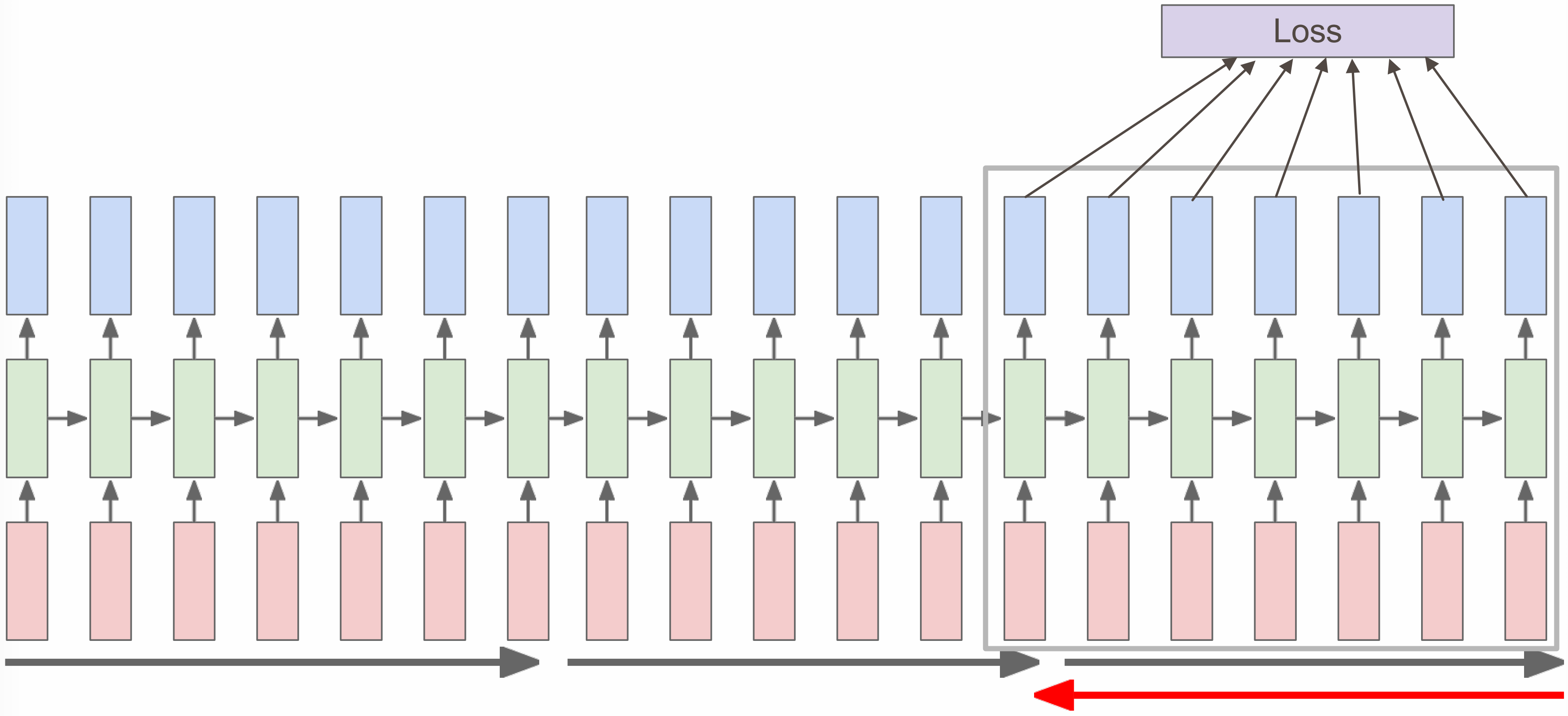

Run forward and backward through chunks of the sequence instead of whole sequence

# Truncated Backpropagation through time




Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps

# Truncated Backpropagation through time




partial credit by CS311n

```python
import numpy as np

# data I/O
!wget https://www.w3.org/TR/PNG/iso_8859-1.txt
data = open('iso_8859-1.txt', 'r').read() # should be simple plain text
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 sta
    """
    xs, hs, ys, ps = {}, {}, {}, {}
    hs[-1] = np.copy(hprev)
    loss = 0
    # forward pass
    for t in range(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 stat
        ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next ch
        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(range(len(inputs))):
        dy = np.copy(ps[t])
        dy[targets[t]] -= 1
        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]

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 range(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
```

```python
    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
```

```
---
ter 17400, loss: 9.990117
---
H ACUTA
4D A1 (STHPYavCrB1  C7BCtIVE                 HENDISN
7  SMALL LETTER D                            ED  RIGIT SIGN
5  CAPITAL LETTER Y              B6  TMACINWTRRAC CAPITAL LETAE              PE
---
```

https://colab.research.google.com/drive/1VqYQIGmDtIZT5fC7TfpWNIXYId4-Pr00?usp=sharing

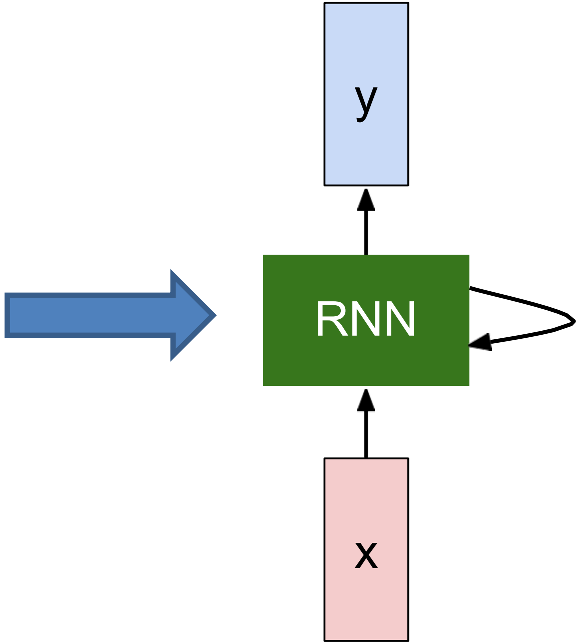
/

THE BOY WHO LIVED

Mr. and Mrs. Dursley, of number four, Privet Drive, were proud to say that they were perfectly normal, thank you very much. They were the last people you'd expect to be involved in anything strange or mysterious, because they just didn't hold with such nonsense.

Mr. Dursley was the director of a firm called Grunnings, which made drills. He was a big, beefy man with hardly any neck, although he did have a very large mustache. Mrs. Dursley was thin and blonde and had nearly twice the usual amount of neck, which came in very useful as she spent so much of her time craning over garden fences, spying on the neighbors. The Dursley s had a small son called Dudley and in their opinion there was no finer boy anywhere.

The Dursleys had everything they wanted, but they also had a secret, and their greatest fear was that somebody would discover it. They didn't think they could bear it if anyone found out about the Potters. Mrs. Potter was Mrs. Dursley's sister, but they hadn't



at first:

After some iterations

inn. yoomnsa
ihae— aci iu— —e—hic—sihivt attcelu —e—’ r ap—s—eaairee
tm— scuyroahufher o — hni eat —nrsitnoe gcaats— f nneitthmgoeen
—, —Drise—srg—e o, tes icu veruwi—ehuigmah eeitgpit, ilso. 1—mMeah—e t

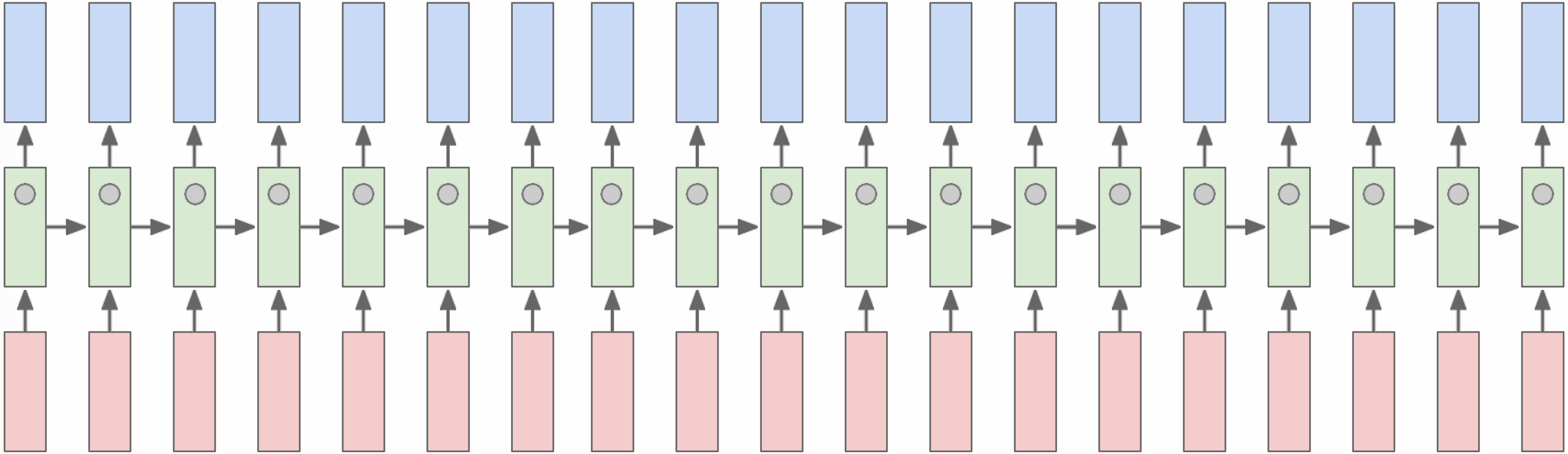
Ducbing ou, dy |ud the ale dos hat bey at gorill s othee
Dndt sun che s ak pakr tteme this aar ve Vind at fhate. Hedisn Dant the hed” dY okes yrecse
ove thou Dud ittthat

Potter
pars turned to toubbut jumbleangoching a crays and was
tringers Sthione iw fars. I’ m to chamy nirmp’ t Cuarlis sacd bask alay, ”

“It’ s tree at it
smoutture and fore\”

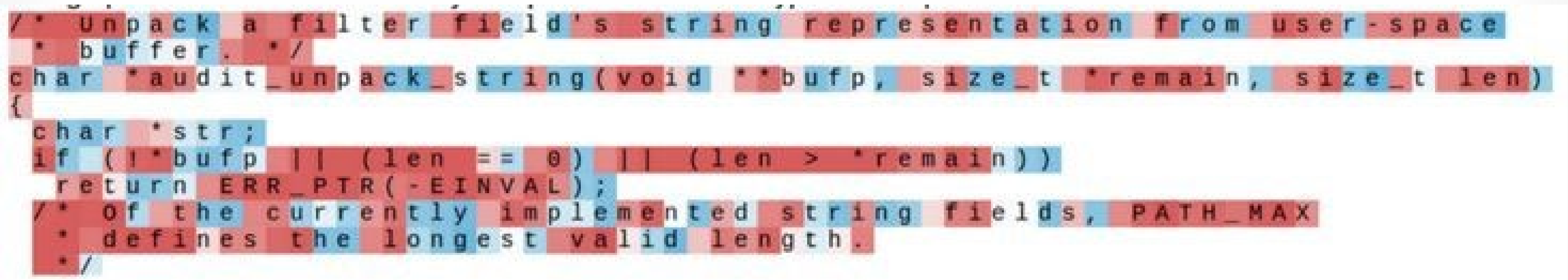
Harry spine a way to leep, said the little supposk a daricing the

# Searching for interpretable cells



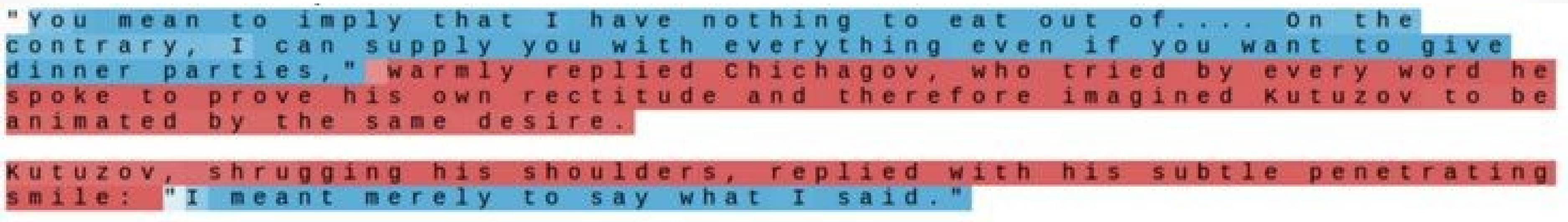
Karpathy, Johnson, and Fei-Fei: Visualizing and Understanding Recurrent Networks, ICLR Workshop 2016

# Searching for interpretable cells




Karpathy, Johnson, and Fei-Fei: Visualizing and Understanding Recurrent Networks, ICLR Workshop 2016

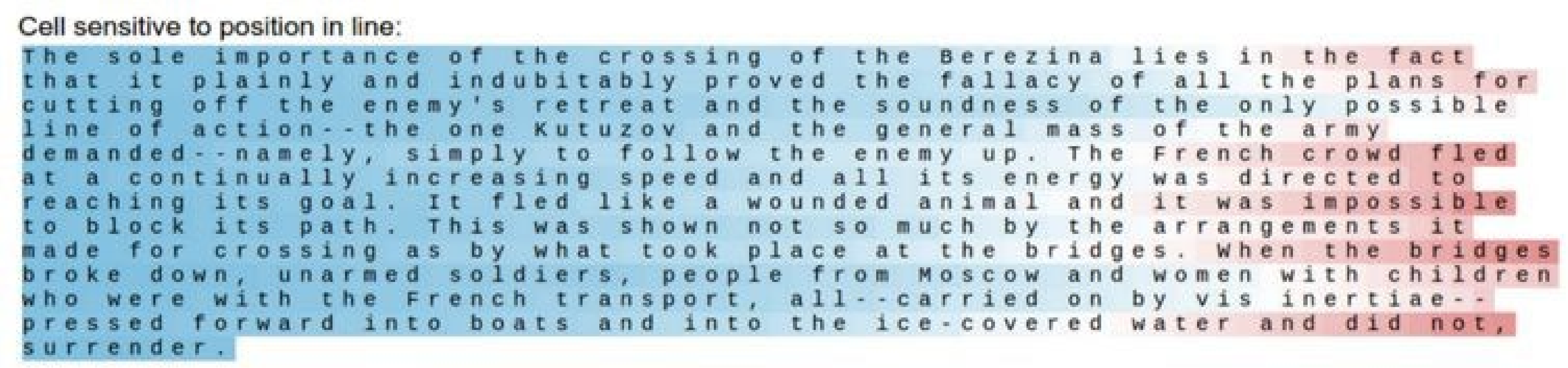
# Searching for interpretable cells




quote detection cell

Karpathy, Johnson, and Fei-Fei: Visualizing and Understanding Recurrent Networks, ICLR Workshop 2016

# Searching for interpretable cells




line length tracking cell

Karpathy, Johnson, and Fei-Fei: Visualizing and Understanding Recurrent Networks, ICLR Workshop 2016

# Searching for interpretable cells


if statement cell

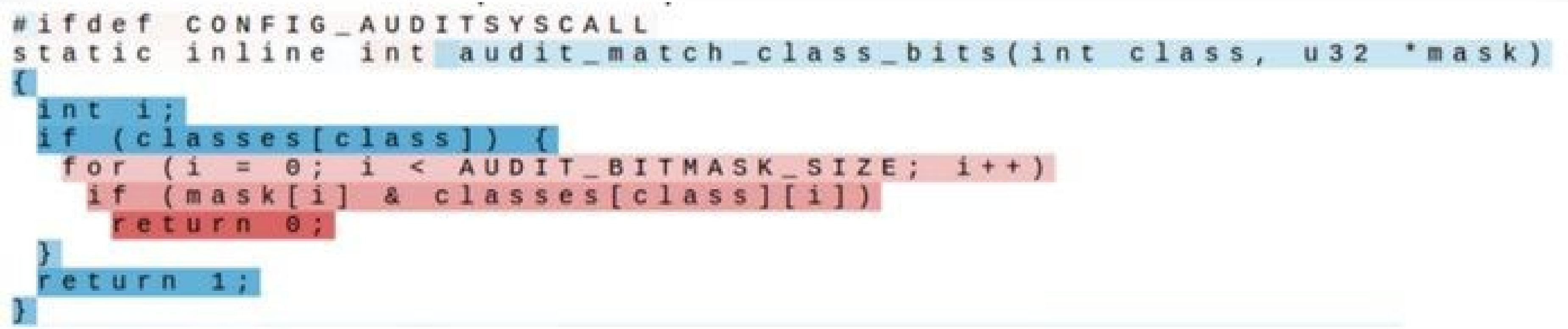
Karpathy, Johnson, and Fei-Fei: Visualizing and Understanding Recurrent Networks, ICLR Workshop 2016

# Searching for interpretable cells


quote/comment cell

Karpathy, Johnson, and Fei-Fei: Visualizing and Understanding Recurrent Networks, ICLR Workshop 2016

# Searching for interpretable cells




code depth cell

Karpathy, Johnson, and Fei-Fei: Visualizing and Understanding Recurrent Networks, ICLR Workshop 2016

# RNN tradeoffs

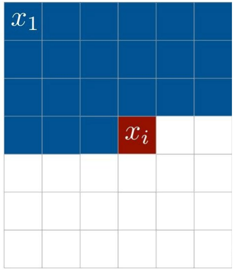
- RNN Advantages:
  - Can process any length input
  - Computation for step t can (in theory) use information from many steps back
  - Model size doesn’t increase for longer input
  - Same weights applied on every timestep, so there is symmetry in how inputs are processed.
- RNN Disadvantages:
  - Recurrent computation is slow
  - In practice, difficult to access information from many steps back

# IN IMAGE: PIXEL-RNN

# Fully visible belief network (FVBN)

- Explicit density model
- Use chain rule to decompose likelihood of an image x into product of 1-d distributions:

$$p(x) = \prod_{i=1}^{n} p(x_i | x_1, ..., x_{i-1})$$



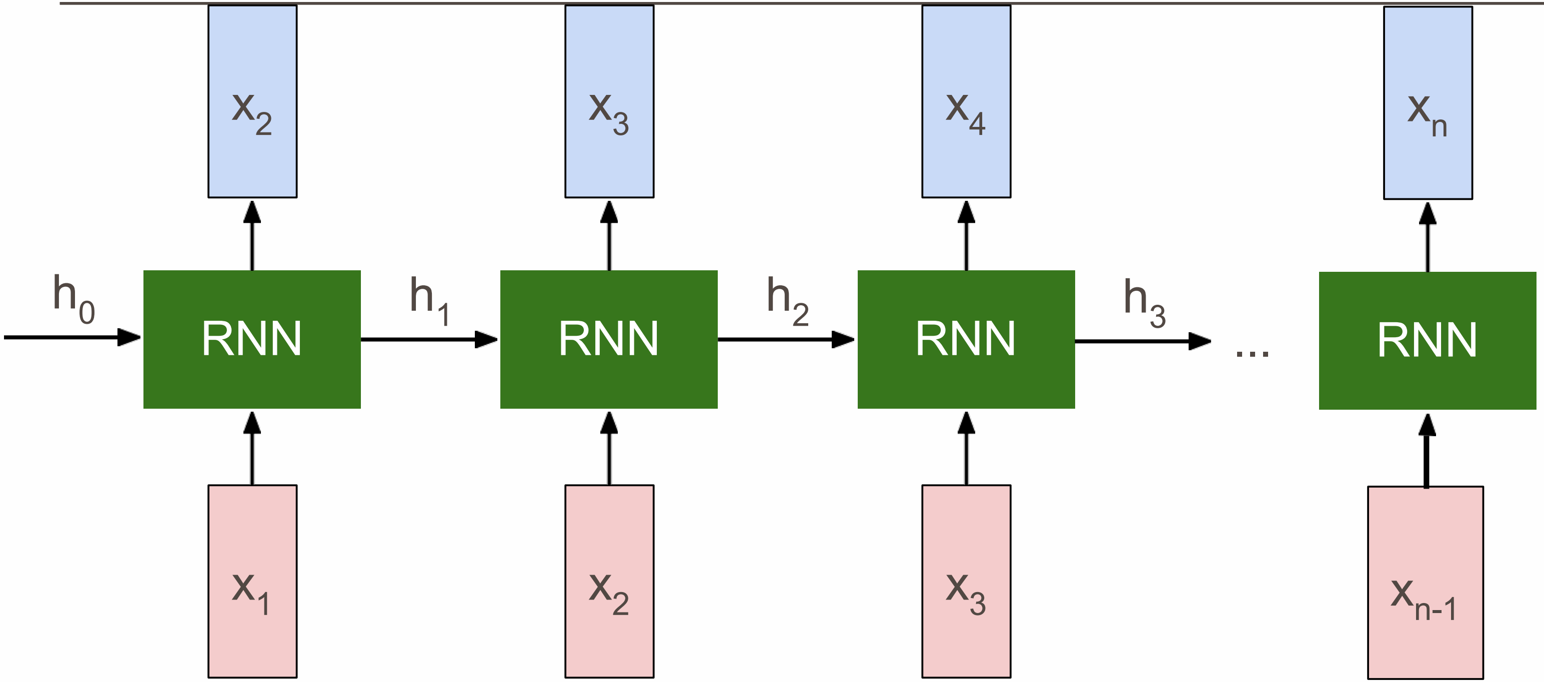

Likelihood of image x

Probability of i'th pixel value given all previous pixels

- Then maximize likelihood of training data

Complex distribution over pixel values => Express using a neural network!

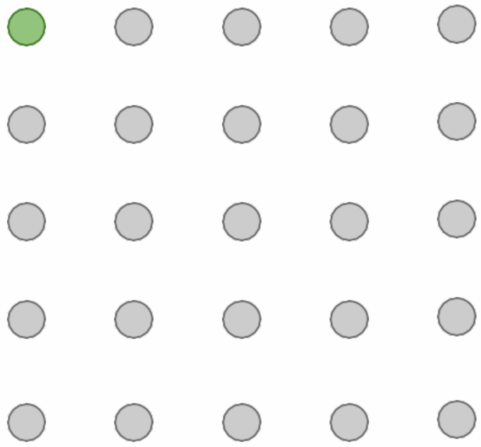
# Recurrent Neural Network



# PixelRNN *[van der Oord et al. 2016]*

Generate image pixels starting from corner

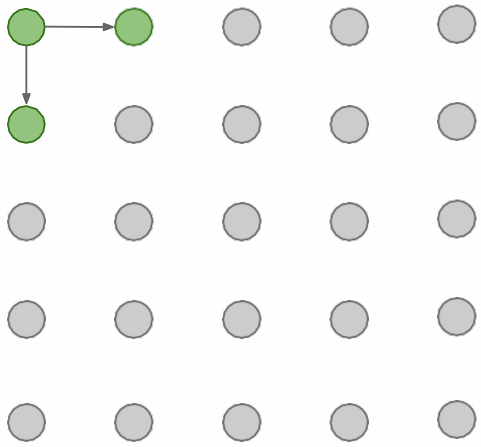
Dependency on previous pixels modeled using an RNN (LSTM)



# PixelRNN *[van der Oord et al. 2016]*

Generate image pixels starting from corner

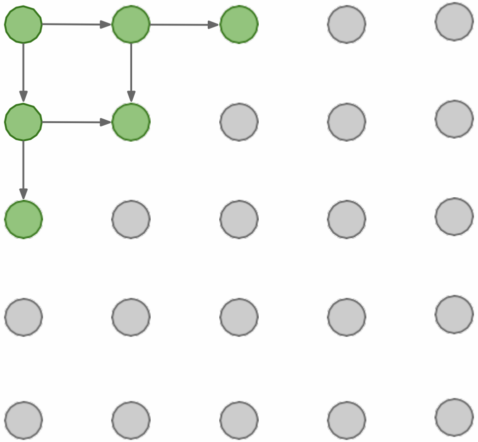
Dependency on previous pixels modeled using an RNN (LSTM)



# PixelRNN *[van der Oord et al. 2016]*

Generate image pixels starting from corner

Dependency on previous pixels modeled using an RNN (LSTM)

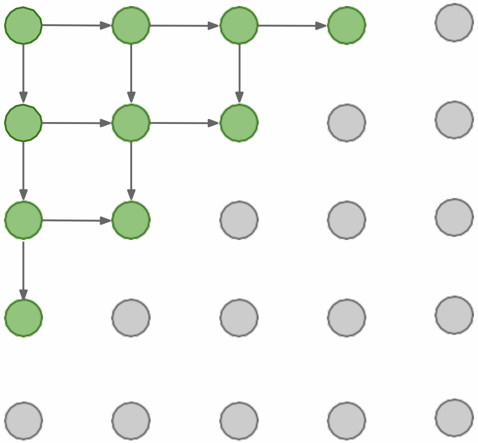


# PixelRNN *[van der Oord et al. 2016]*

Generate image pixels starting from corner

Dependency on previous pixels modeled using an RNN (LSTM)

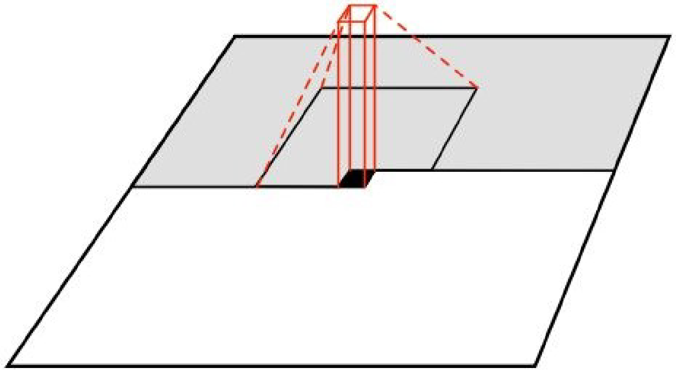
Drawback: sequential generation is slow in both training and inference!



# PixelCNN *[van der Oord et al. 2016]*

Still generate image pixels starting from corner

Dependency on previous pixels now modeled using a CNN over context region (masked convolution)



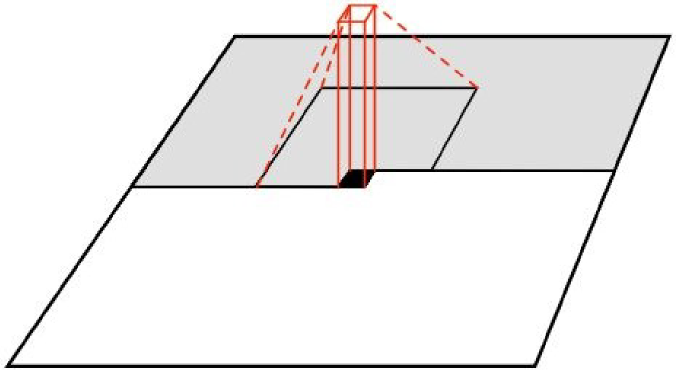
# PixelCNN *[van der Oord et al. 2016]*

Still generate image pixels starting from corner

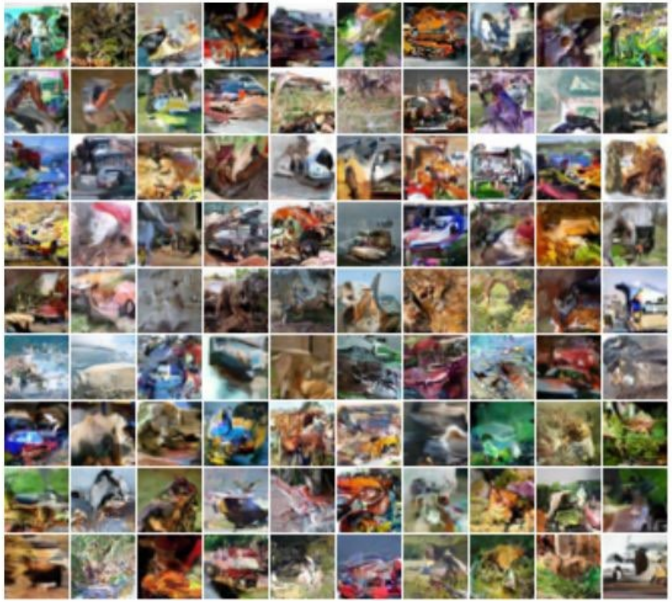
Dependency on previous pixels now modeled using a CNN over context region (masked convolution)

Training is faster than PixelRNN
(can parallelize convolutions since context region values known from training images)

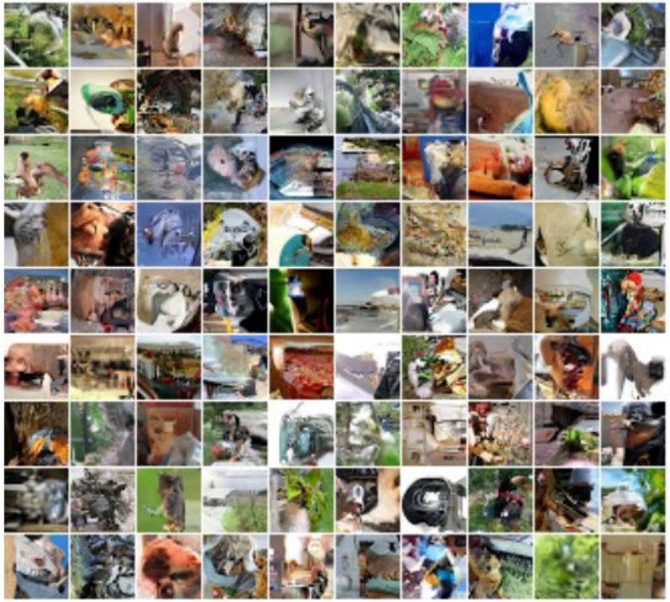
Generation is still slow:
For a 32x32 image, we need to do forward passes of the network 1024 times for a single image



# Generation Samples



32x32 CIFAR-10



32x32 ImageNet

# Image Captioning

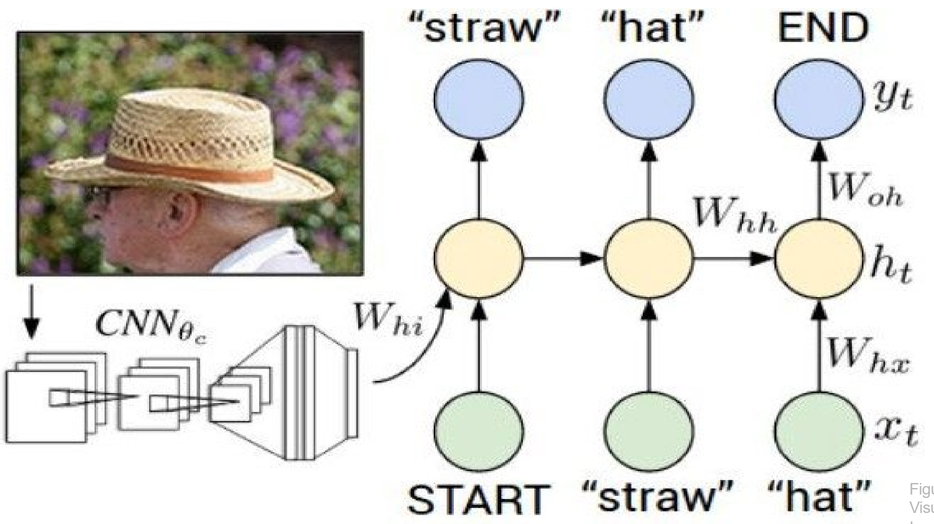
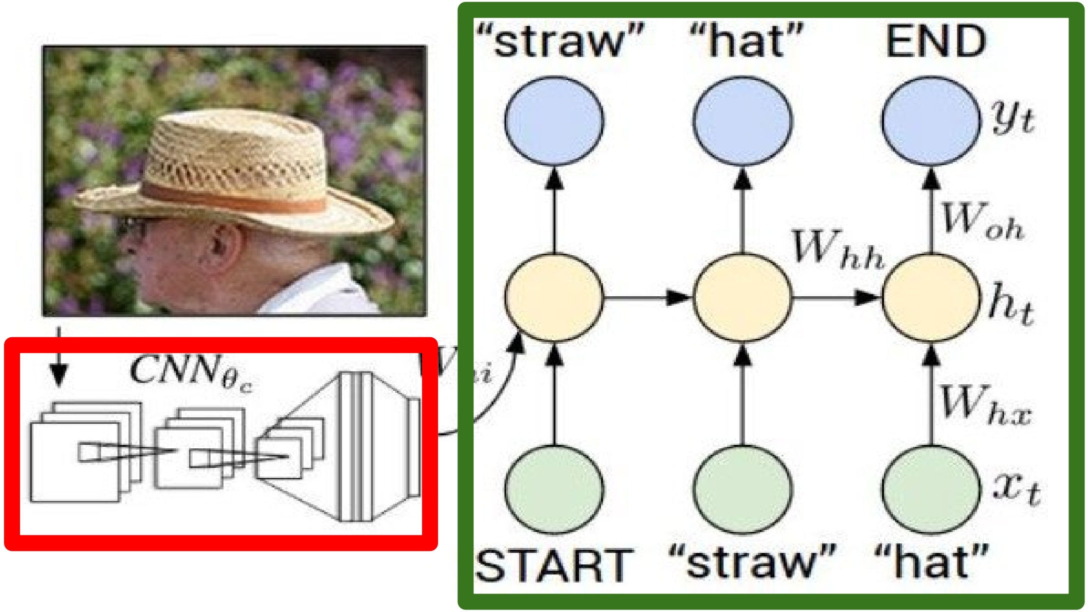



Figure from Karpathy et a, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015; figure copyright IEEE, 2015.
Reproduced for educational purposes.

Explain Images with Multimodal Recurrent Neural Networks, Mao et al.
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei
Show and Tell: A Neural Image Caption Generator, Vinyals et al.
Long-term Recurrent Convolutional Networks for Visual Recognition and Description, Donahue et al.
Learning a Recurrent Visual Representation for Image Caption Generation, Chen and Zitnick

Recurrent Neural Network




Convolutional Neural Network

# test image

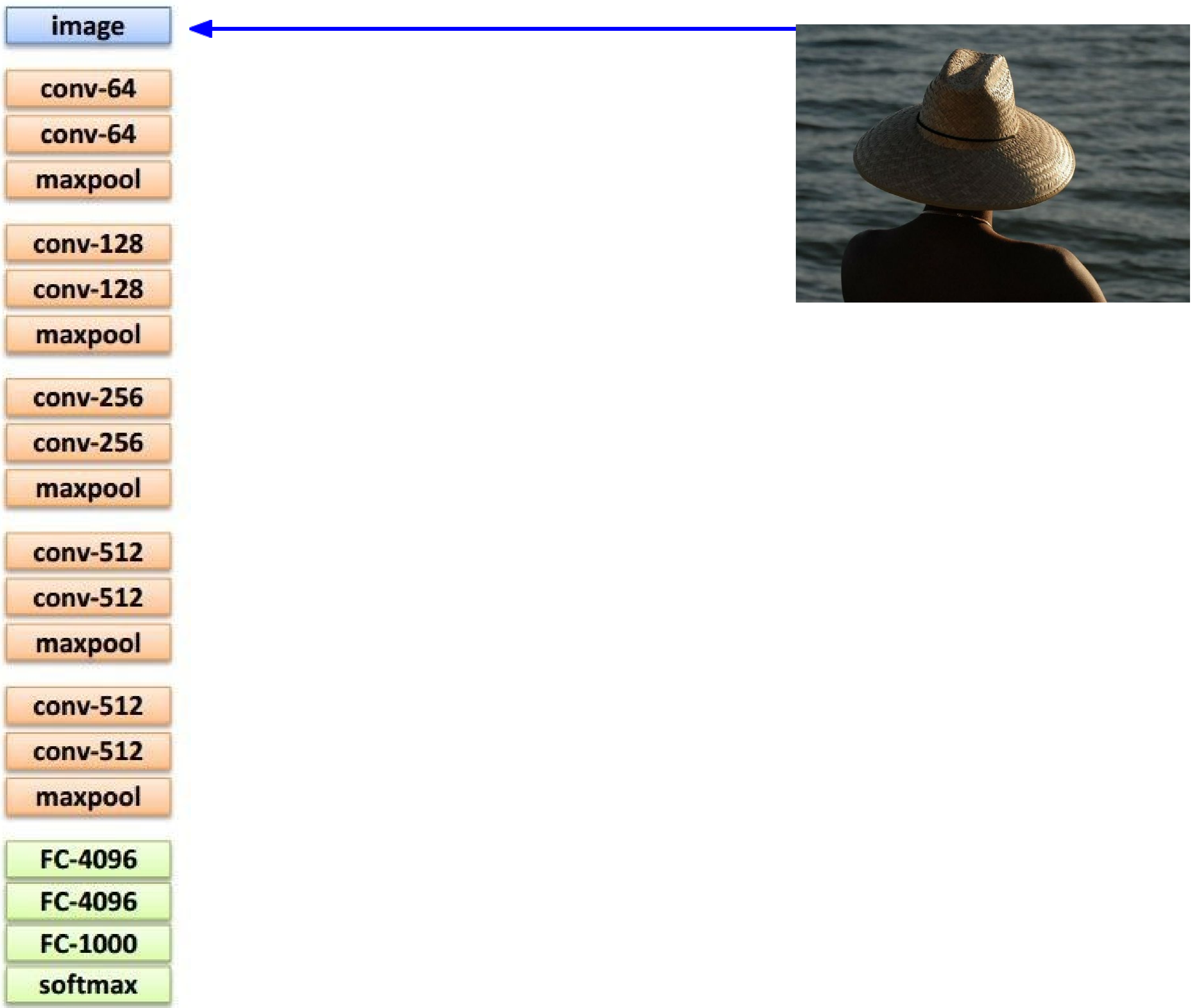


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
FC-1000
softmax

test image

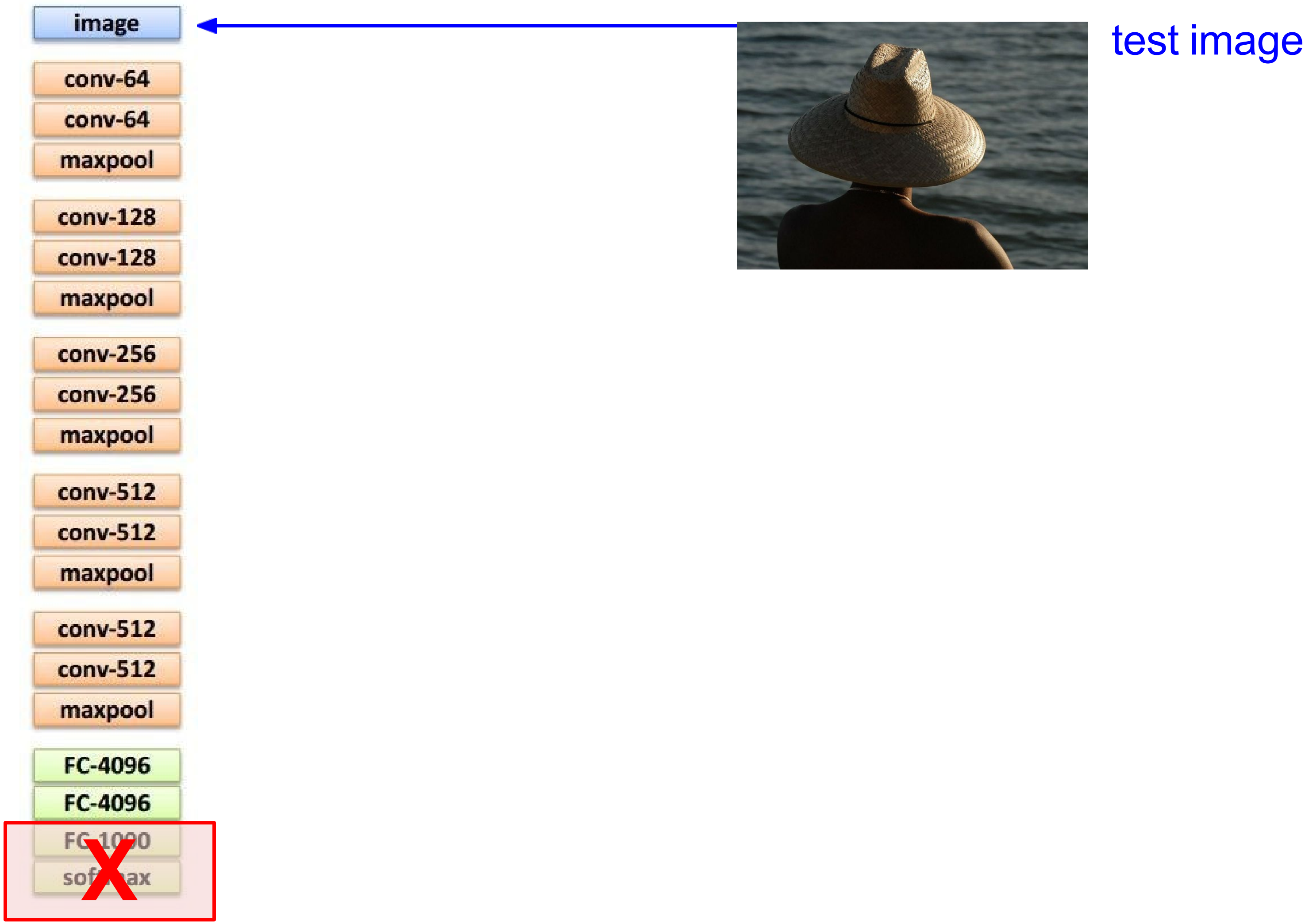


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image

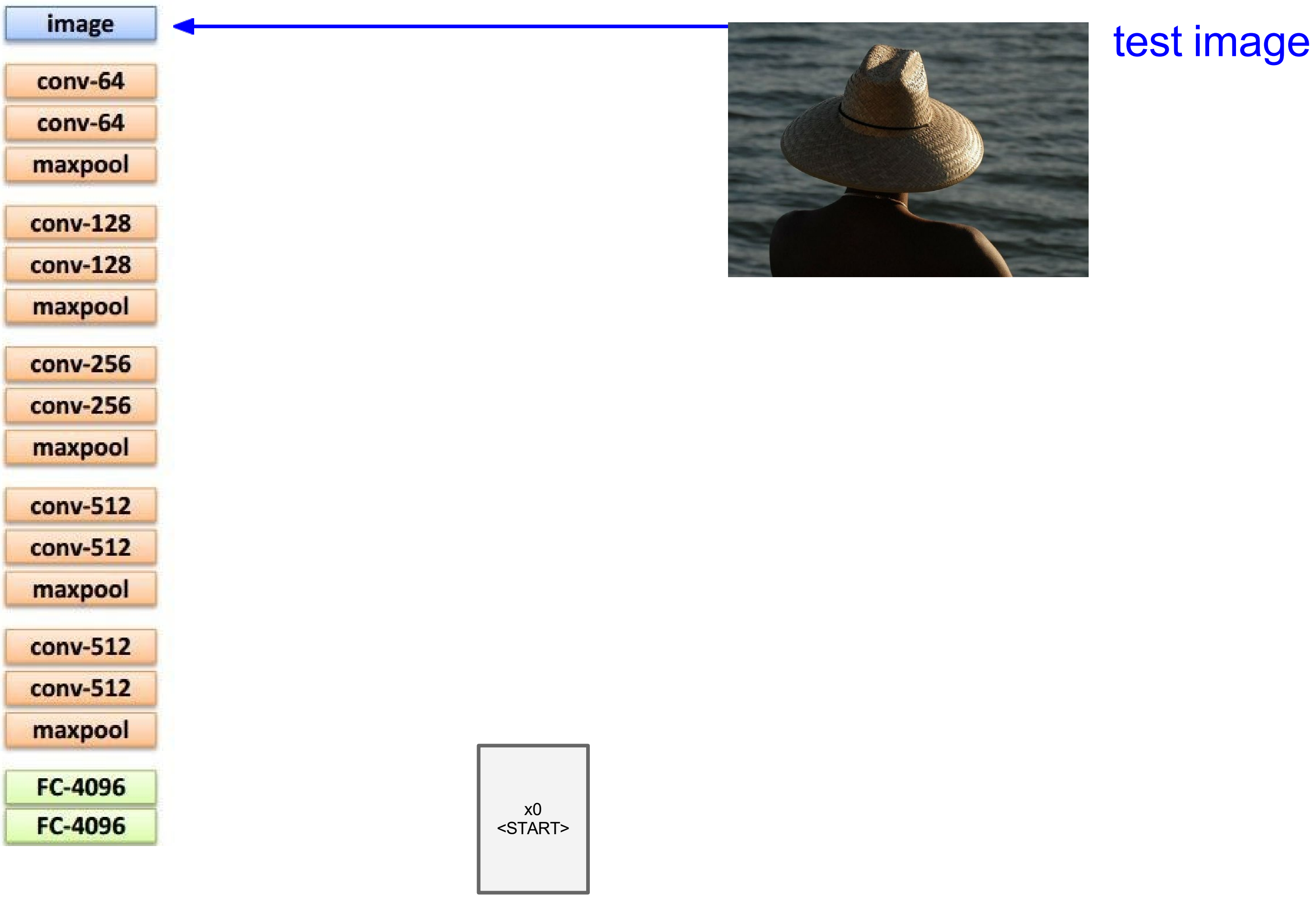
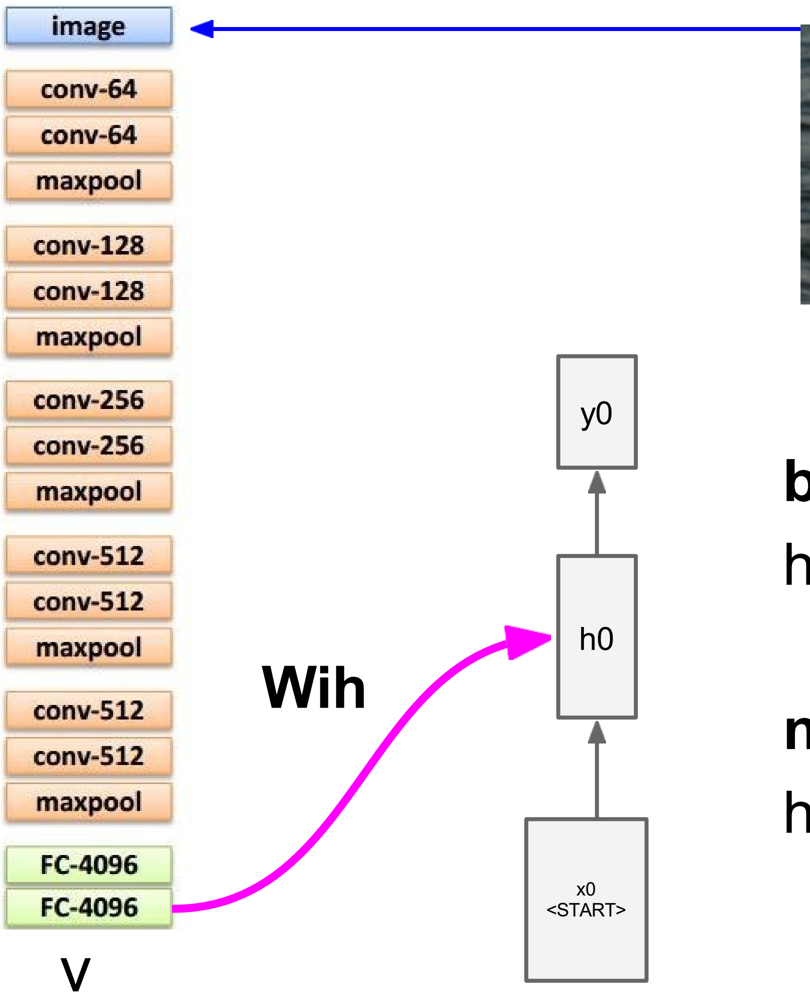


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
x0
<START>




test image

**before:**

$$h = \tanh(W_{xh} * x + W_{hh} * h)$$

**now:**

$$h = \tanh(W_{xh} * x + W_{hh} * h + W_{ih} * v)$$

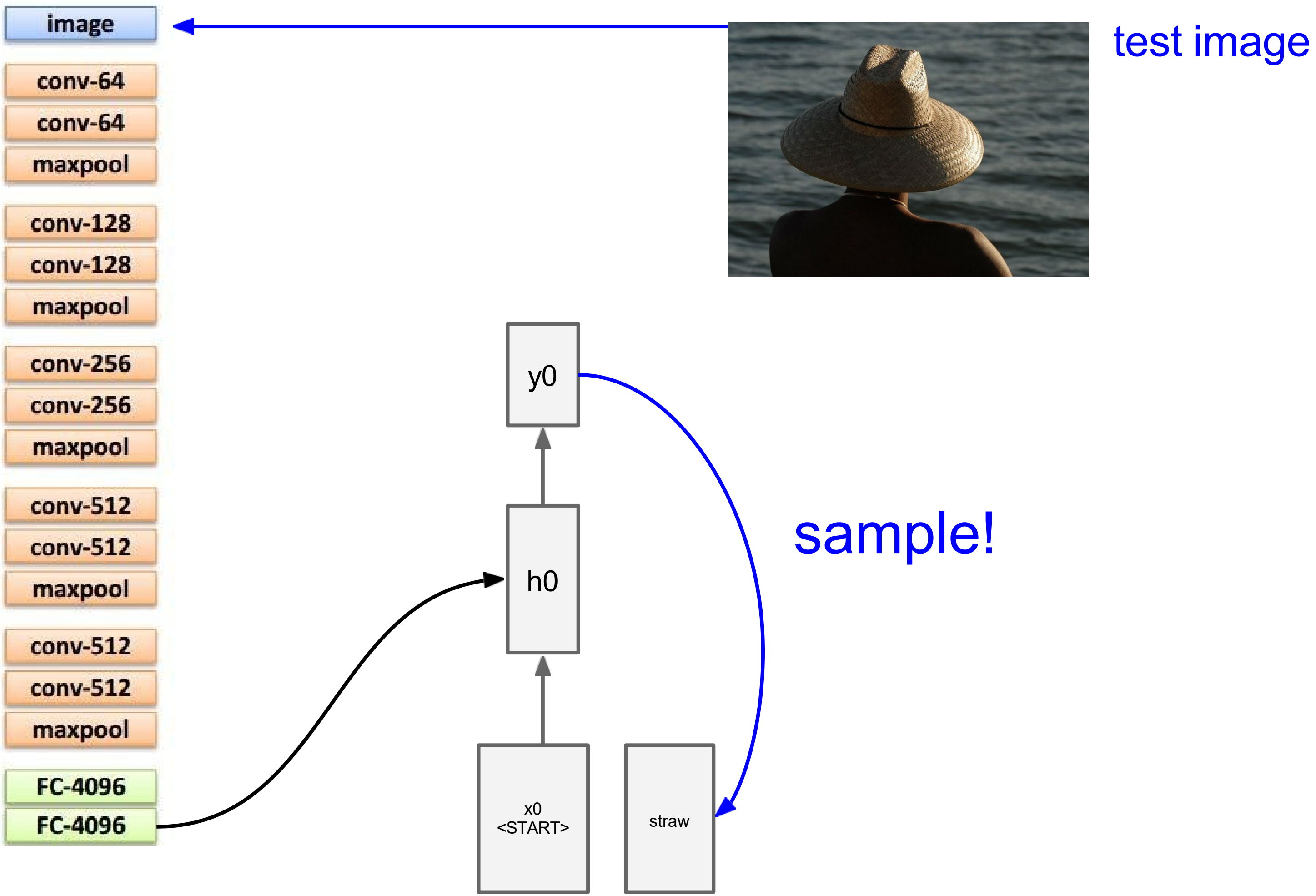


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
y0
h0
x0
<START>
straw
sample!

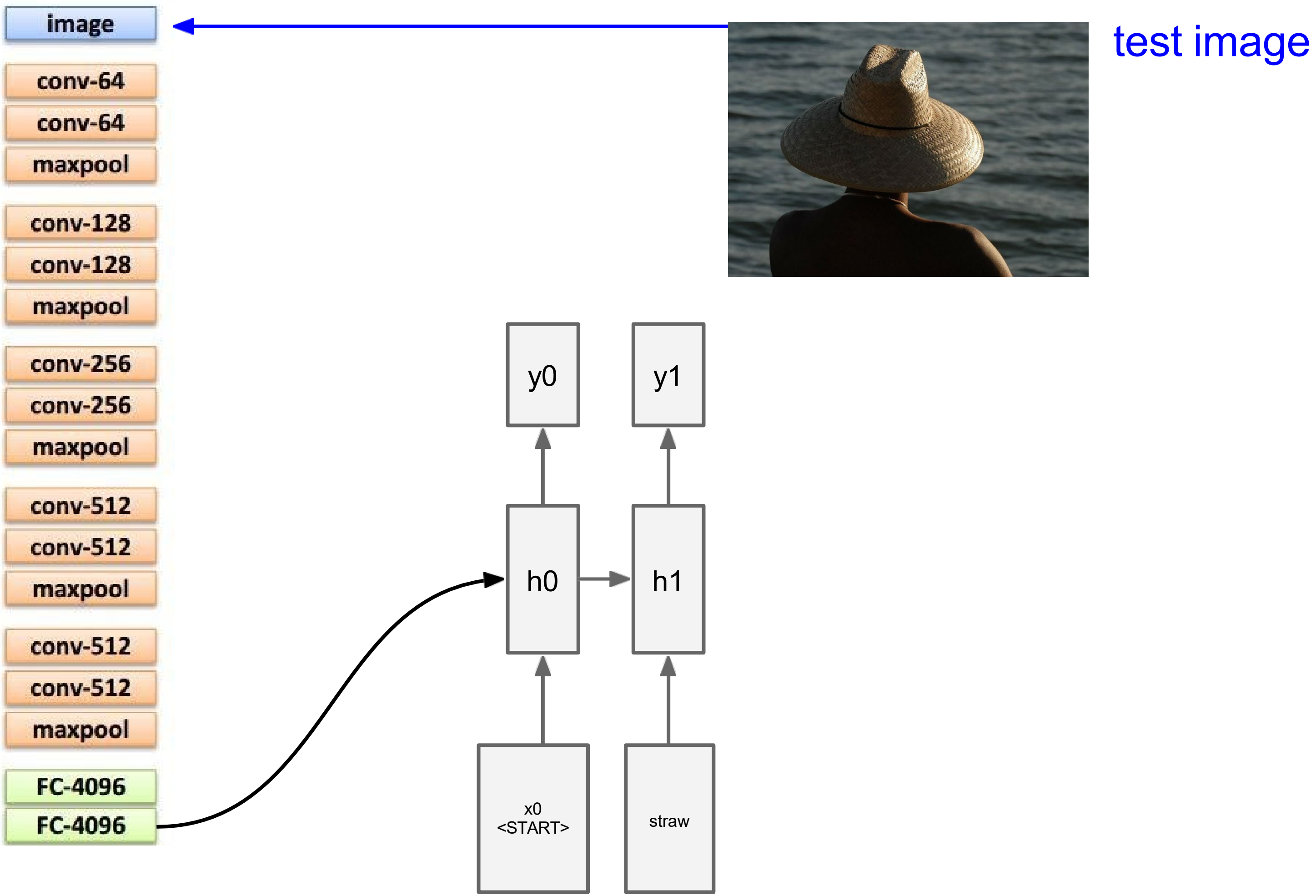


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
y0
y1
h0
h1
x0
<START>
straw

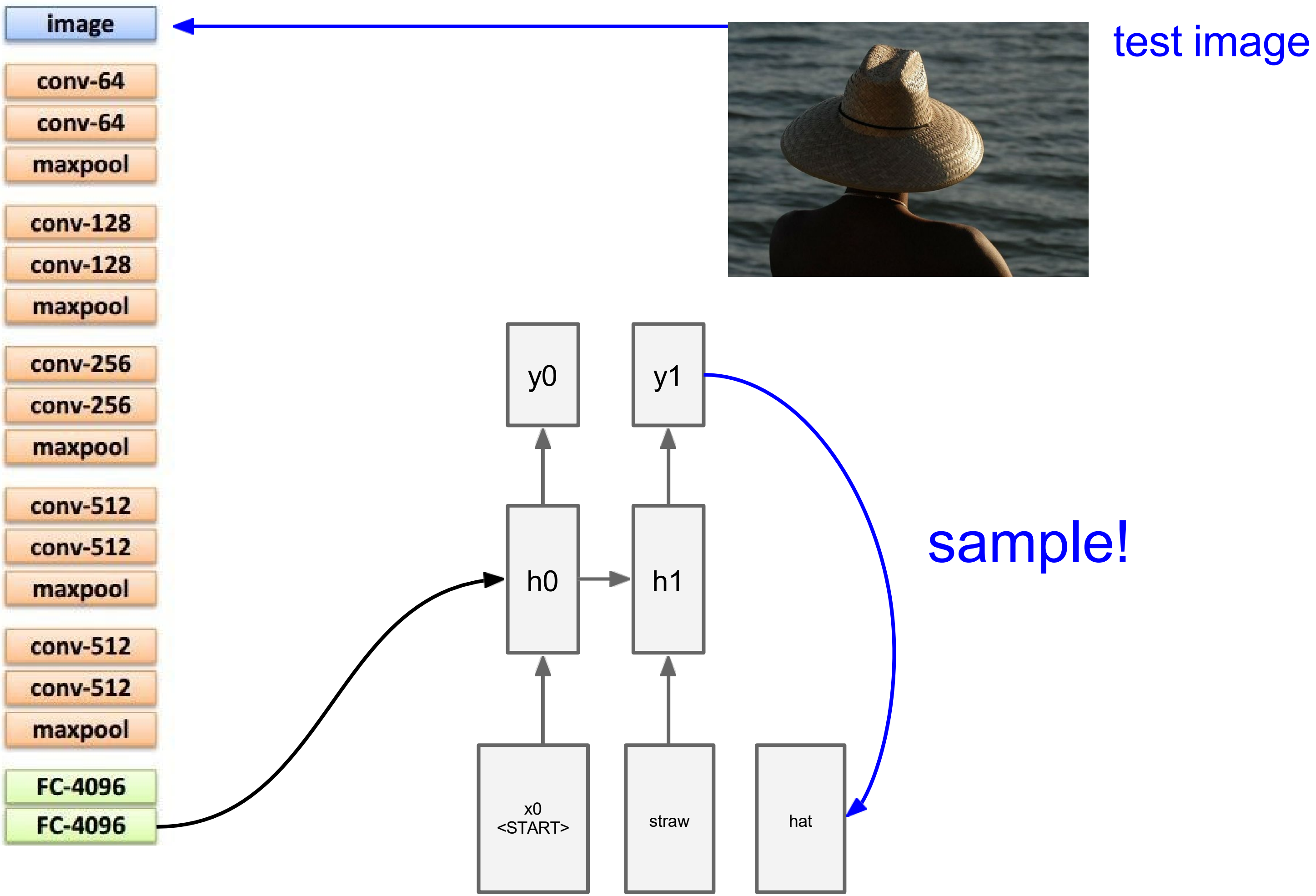


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
y0
y1
h0
h1
sample!
x0
<START>
straw
hat

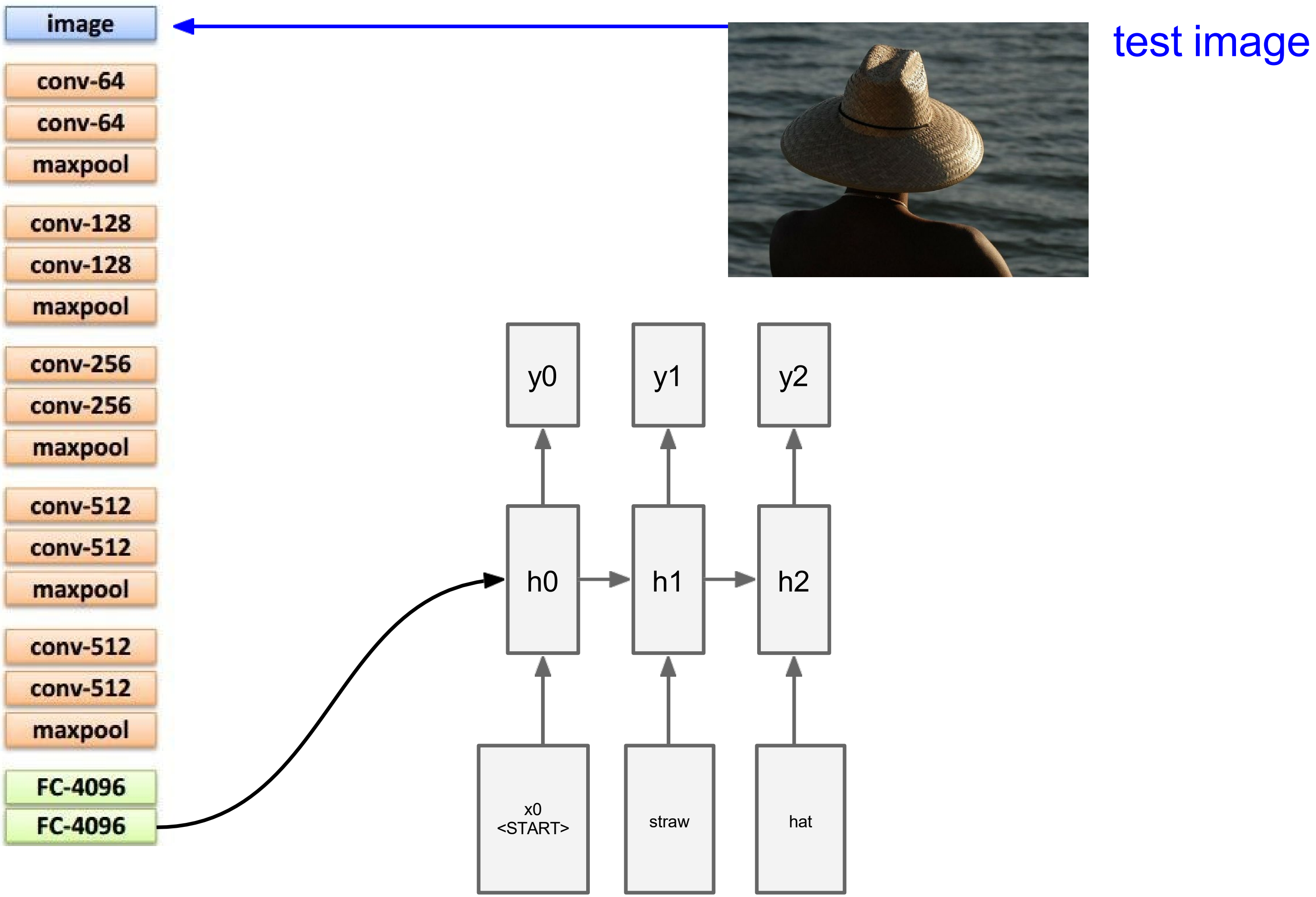


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
y0
y1
y2
h0
h1
h2
x0
<START>
straw
hat

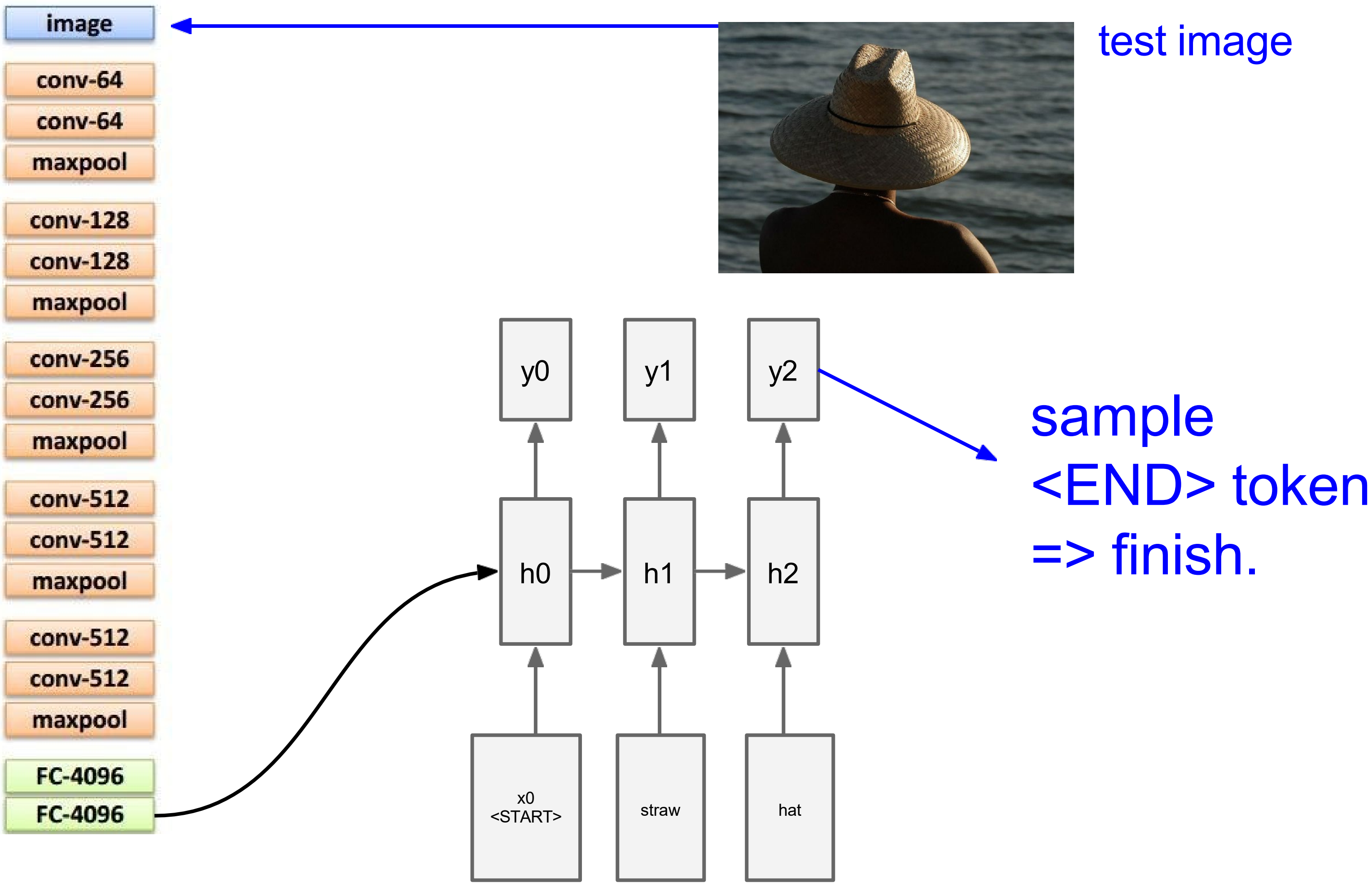


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
y0
y1
y2
h0
h1
h2
x0
<START>
straw
hat
sample
<END> token
=> finish.

Captions generated using neuraltalk2
All images are CC0 Public domain: cat suitcase, cat tree, dog, bear, surfers, tennis, giraffe, motorcycle

# Image Captioning: Example Results

*A cat sitting on a suitcase on the floor*

*A cat is sitting on a tree branch*

*A dog is running in the grass with a frisbee*

*A white teddy bear sitting in the grass*

*Two people walking on the beach with surfboards*


*A tennis player in action on the court*

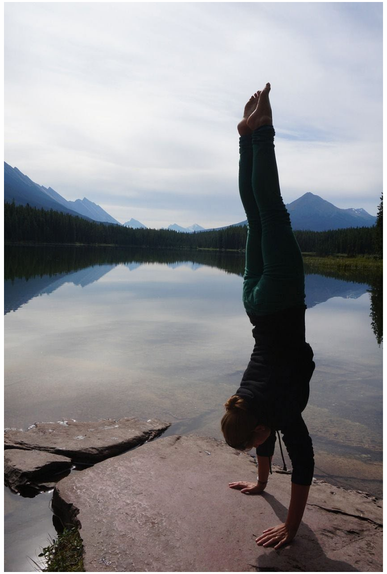
*Two giraffes standing in a grassy field*

*A man riding a dirt bike on a dirt track*

# Image Captioning: Failure Cases

*A woman is holding a cat in her hand*

*A person holding a computer mouse on a desk*



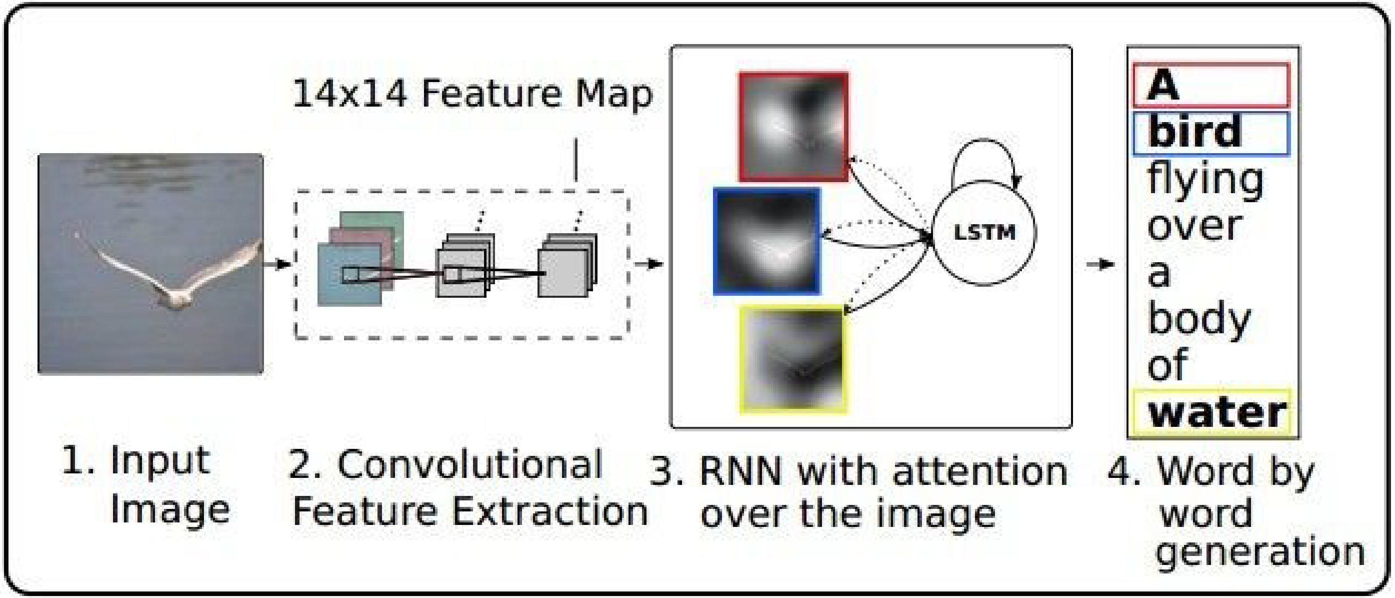
*A woman standing on a beach holding a surfboard*

*A bird is perched on a tree branch*

*A man in a baseball uniform throwing a ball*

# Image Captioning with Attention

RNN focuses its attention at a different spatial location when generating each word




Xu et al, “Show, Attend, and Tell: Neural Image Caption Generation with Visual Attention”, ICML 2015

# Image Captioning with Attention

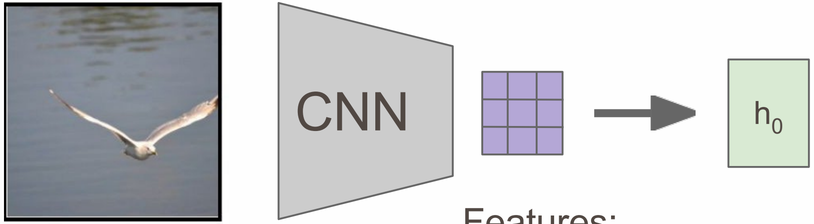
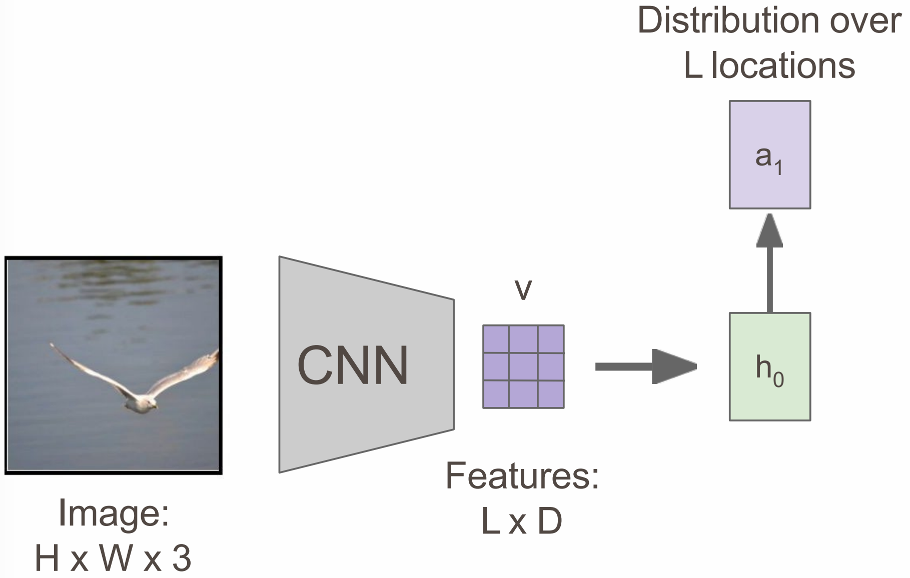



Image:
H x W x 3

Features:
L x D
Where L = W x H

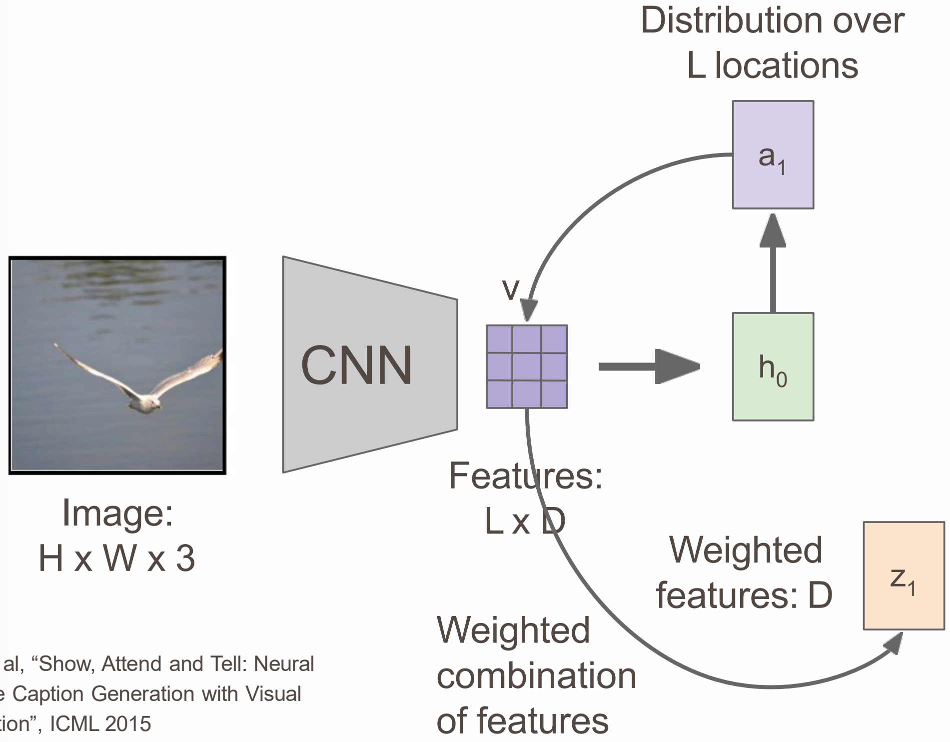
Xu et al, “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention”, ICML 2015

# Image Captioning with Attention




Xu et al, “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention”, ICML 2015

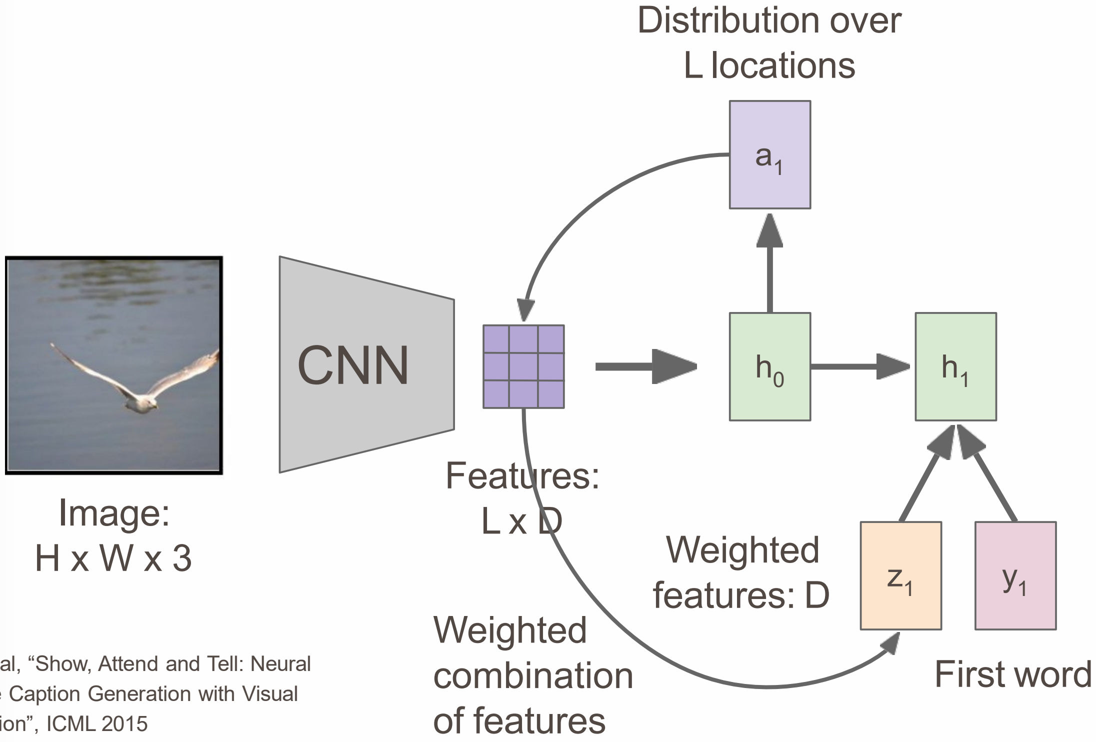
# Image Captioning with Attention




$$z_1 = \sum_{i=1}^{L} a_i v_i$$

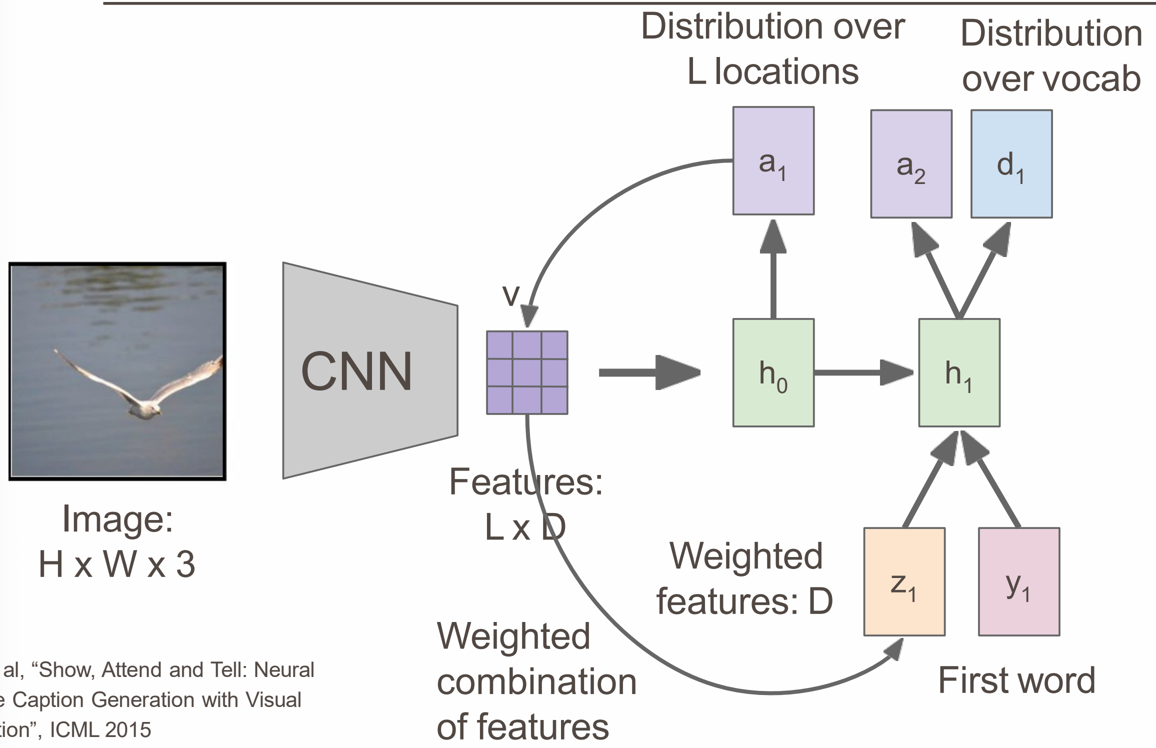
Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

# Image Captioning with Attention



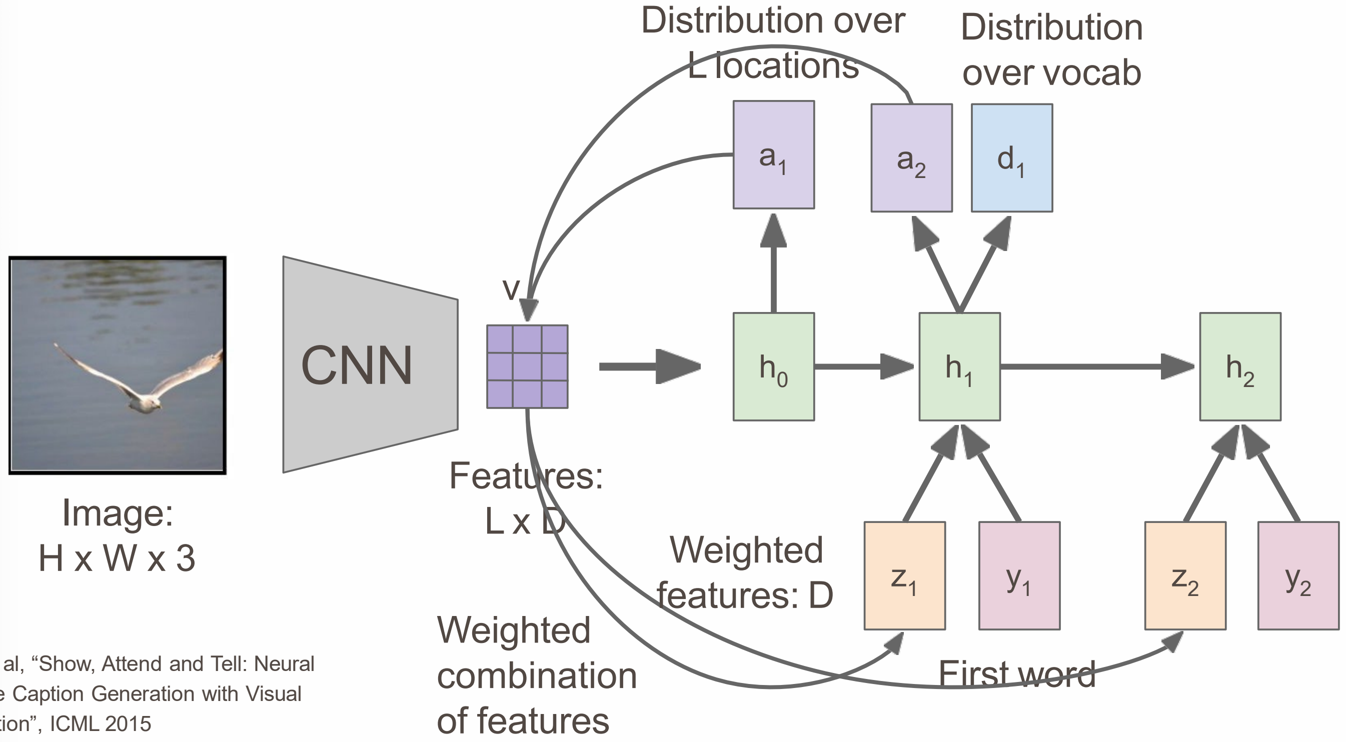

Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

# Image Captioning with Attention



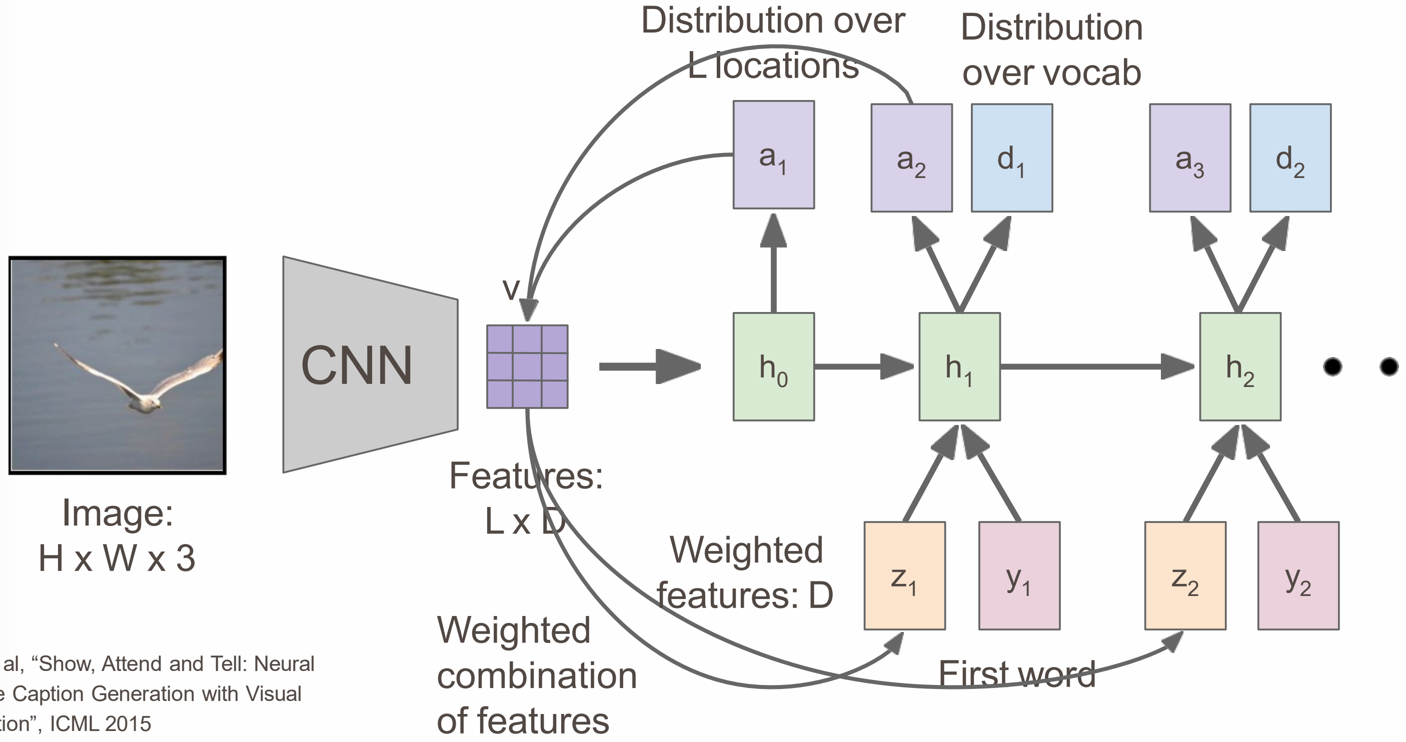

Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

# Image Captioning with Attention



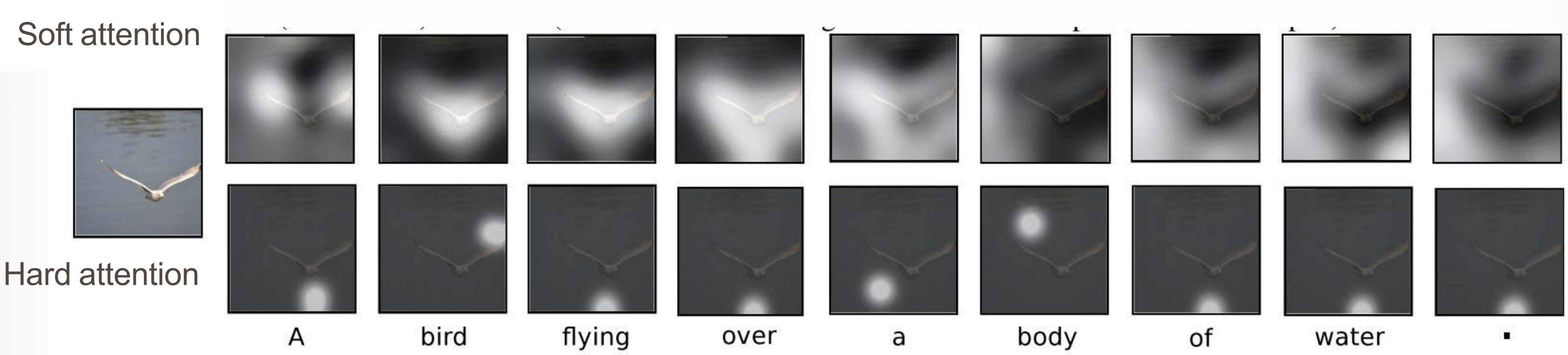

Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

# Image Captioning with Attention



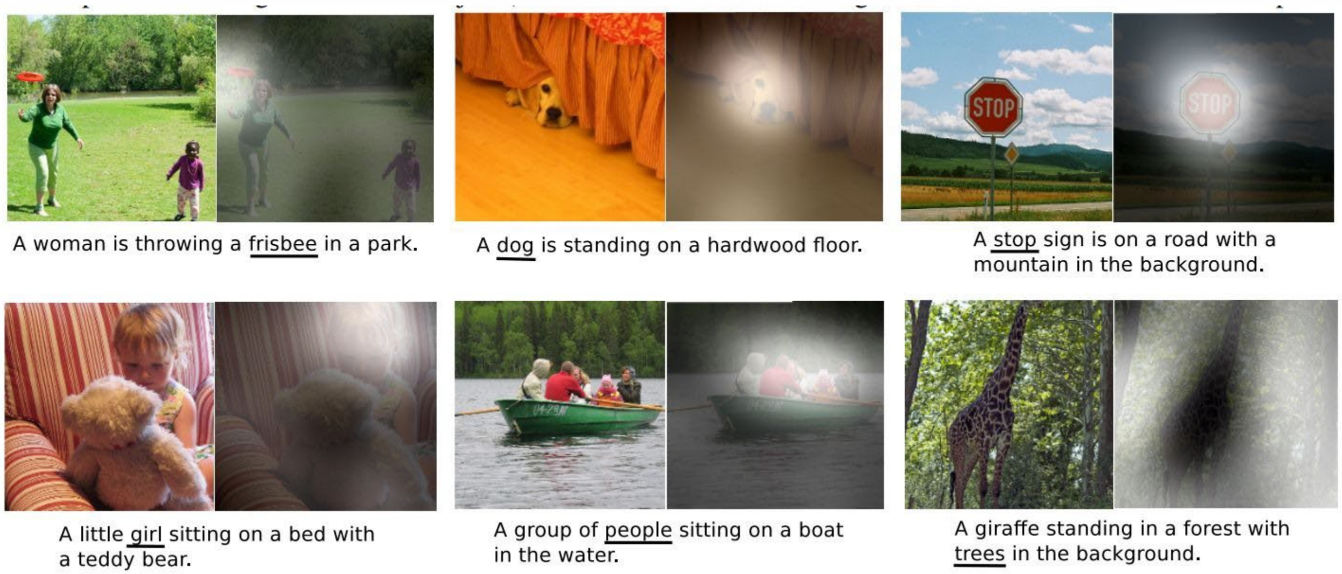

Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

# Image Captioning with Attention




Xu et al, “Show, Attend, and Tell: Neural Image Caption Generation with Visual Attention”, ICML 2015

# Image Captioning with Attention




Xu et al, “Show, Attend, and Tell: Neural Image Caption Generation with Visual Attention”, ICML 2015

# Visual Question Answering (VQA)


Q: What endangered animal is featured on the truck?

A: A bald eagle.
A: A sparrow.
A: A humming bird.
A: A raven.


Q: Where will the driver go if turning right?

A: Onto 24 ¾ Rd.
A: Onto 25 ¾ Rd.
A: Onto 23 ¾ Rd.
A: Onto Main Street.

Q: When was the picture taken?

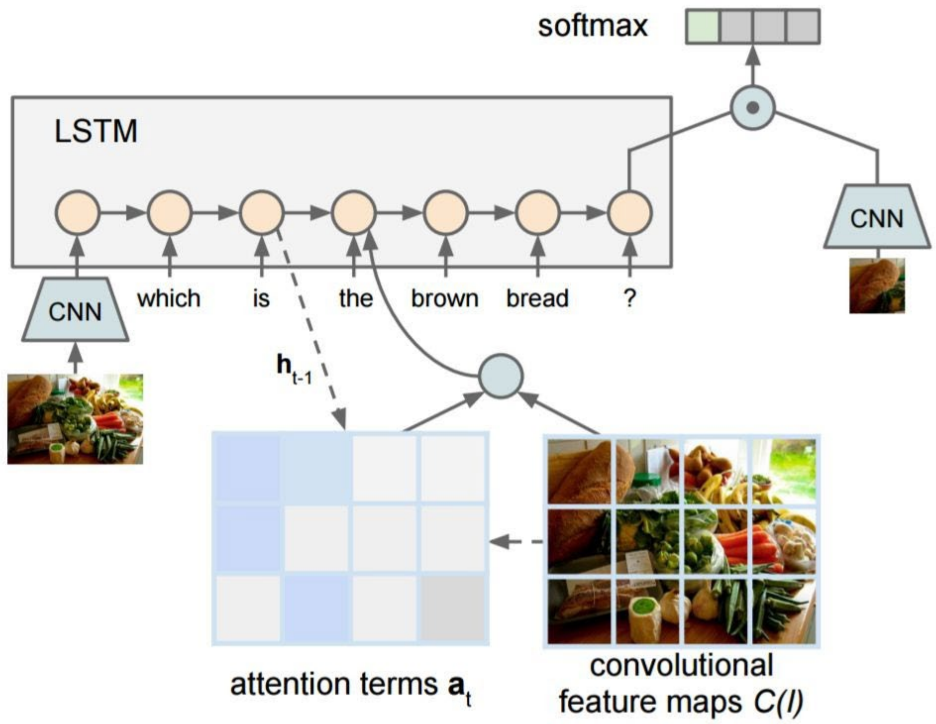
A: During a wedding.
A: During a bar mitzvah.
A: During a funeral.
A: During a Sunday church service

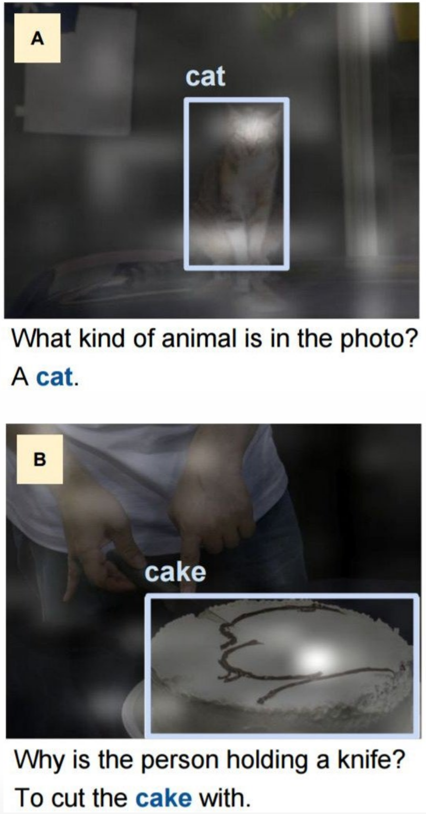
Q: Who is under the umbrella?

A: Two women.
A: A child.
A: An old man.
A: A husband and a wife.

Agrawal et al, “VQA: Visual Question Answering”, ICCV 2015
Zhu et al, “Visual 7W: Grounded Question Answering in Images”, CVPR 2016

# Visual Question Answering: RNNs with Attention




Zhu et al, "Visual 7W: Grounded Question Answering in Images", CVPR 2016



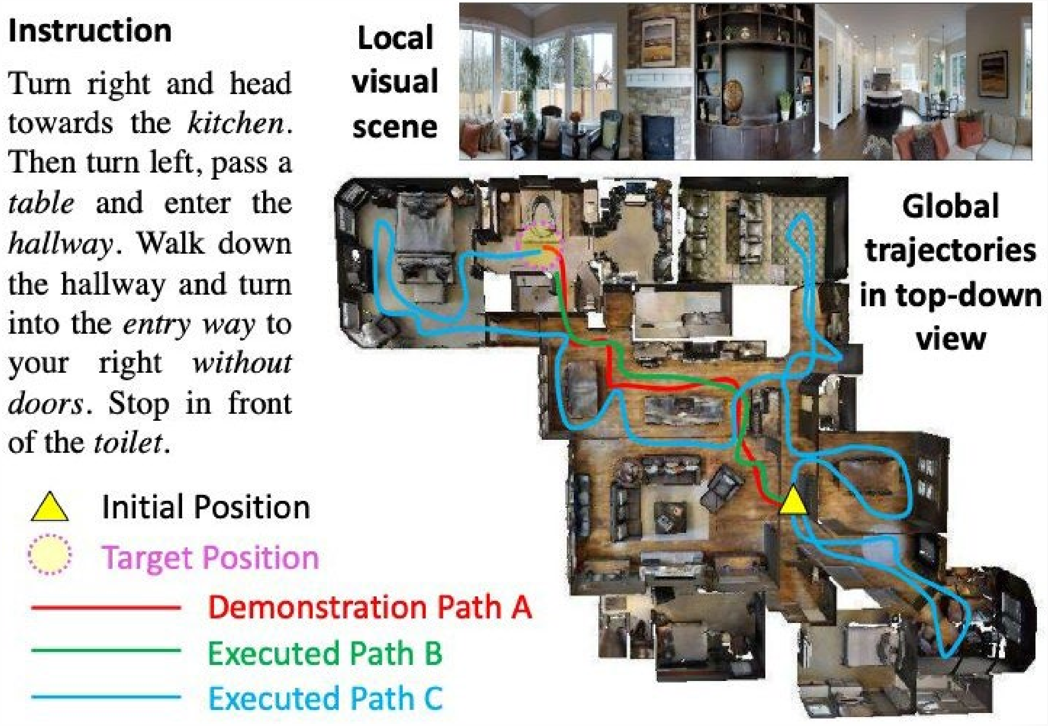
partial credit by CS311n

# Visual Dialog: Conversations about images


Das et al, "Visual Dialog", CVPR 2017

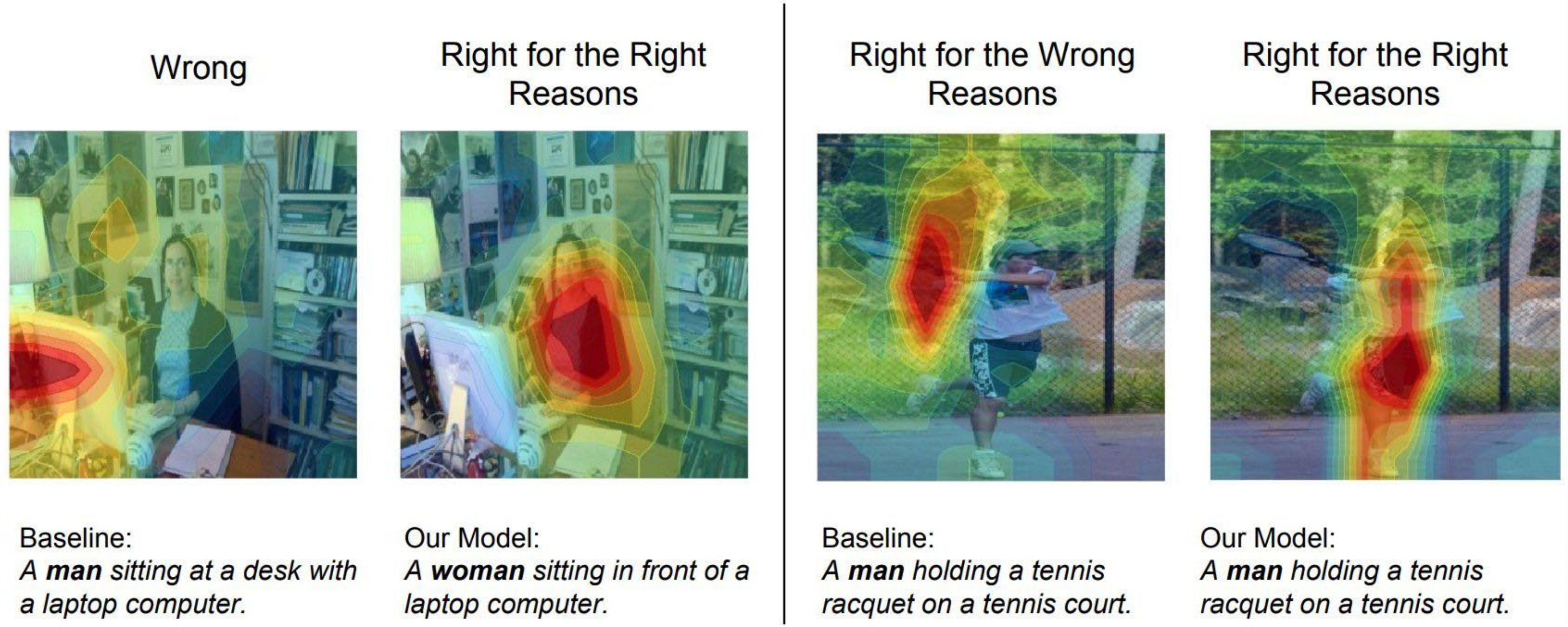
# Visual Language Navigation: Go to the living room

Agent encodes instructions in language and uses an RNN to generate a series of movements as the visual input changes after each move.




Wang et al, "Reinforced Cross-Modal Matching and Self-Supervised Imitation Learning for Vision-Language Navigation", CVPR 2018

# Image Captioning: Gender Bias

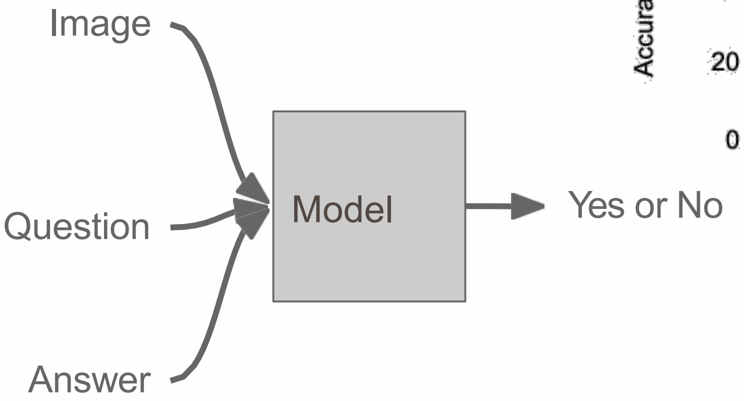


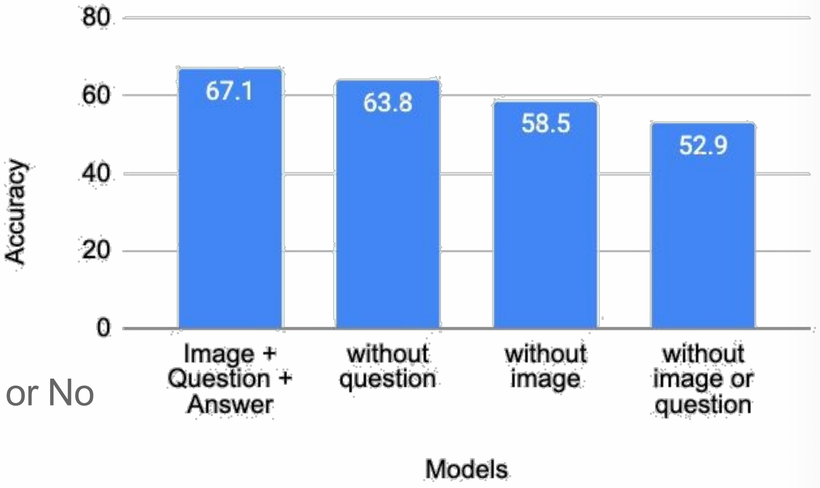

Burns et al. "Women also Snowboard: Overcoming Bias in Captioning Models" ECCV 2018

# Visual Question Answering: Dataset Bias

What is the dog playing with?

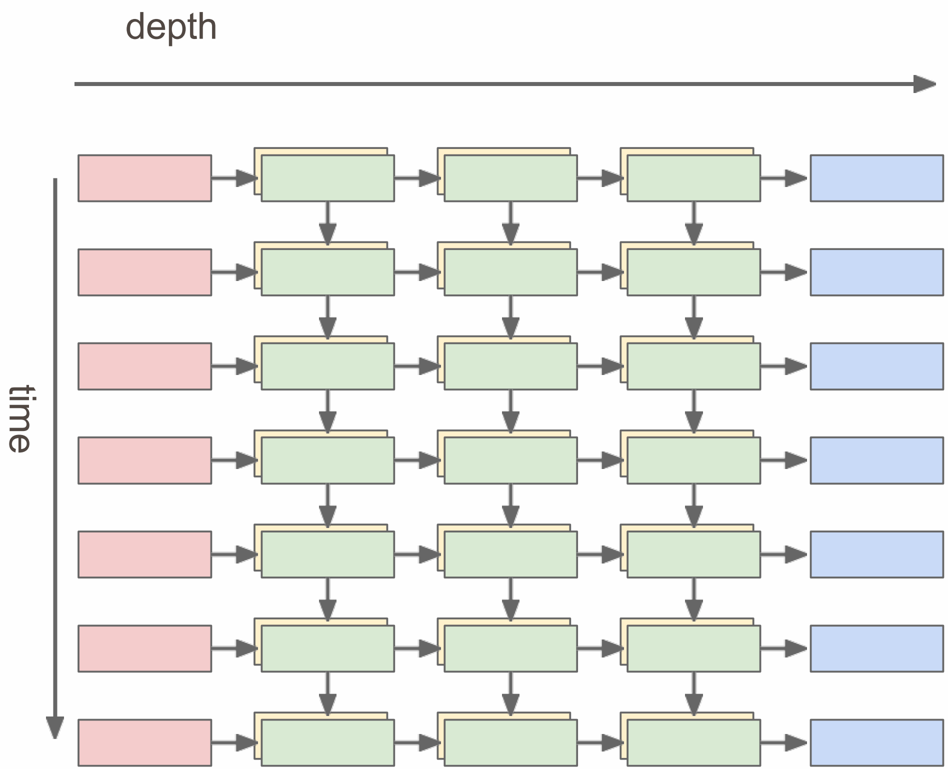
Frisbee




Jabri et al. "Revisiting Visual Question Answering Baselines" ECCV 2016

# Multilayer RNNs






partial credit by CS311n

# Long Short Term Memory (LSTM)

## Vanilla RNN

$$h_t = \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

## LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

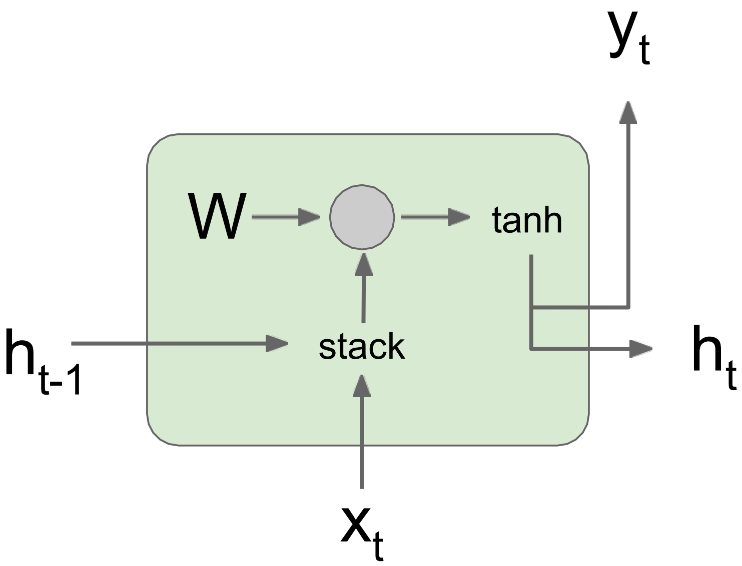
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation 1997

# Vanilla RNN Gradient Flow

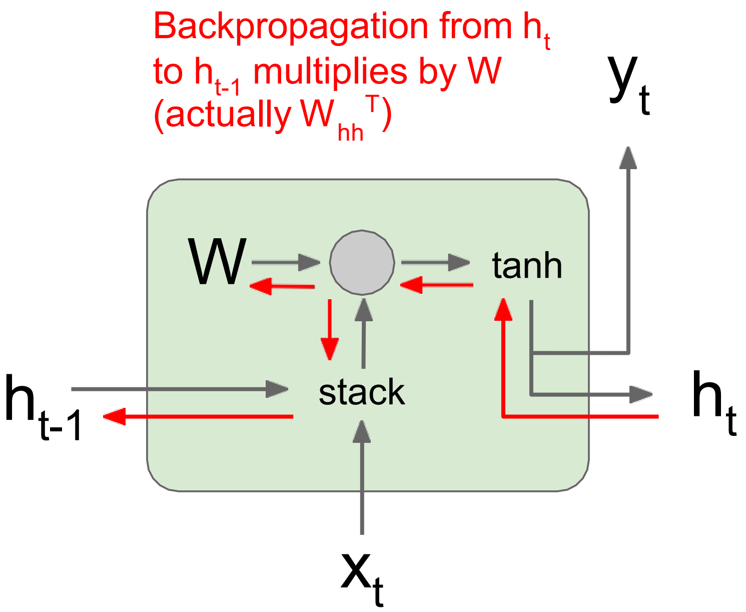
Bengio et al, “Learning long-term dependencies with gradient descent is difficult”, IEEE Transactions on Neural Networks, 1994
Pascanu et al, “On the difficulty of training recurrent neural networks”, ICML 2013




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

# Vanilla RNN Gradient Flow

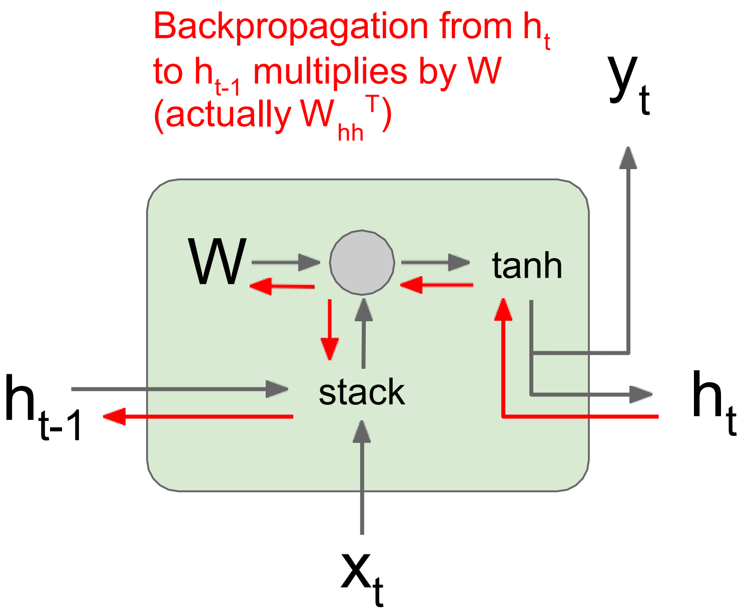
Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013




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

# 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

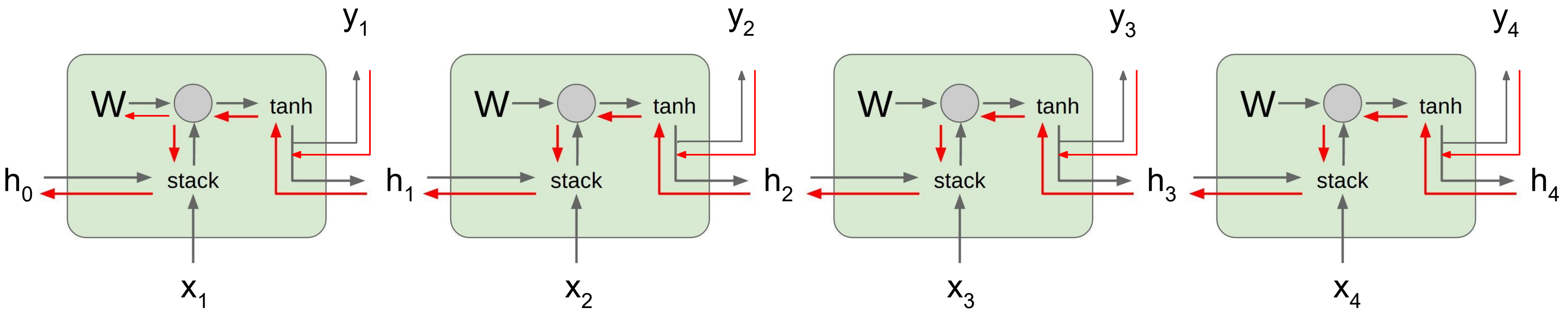



$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$
$$= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$
$$= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

$$\frac{\partial h_t}{\partial h_{t-1}} = tanh'(W_{hh}h_{t-1} + W_{xh}x_t)W_{hh}$$

# 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

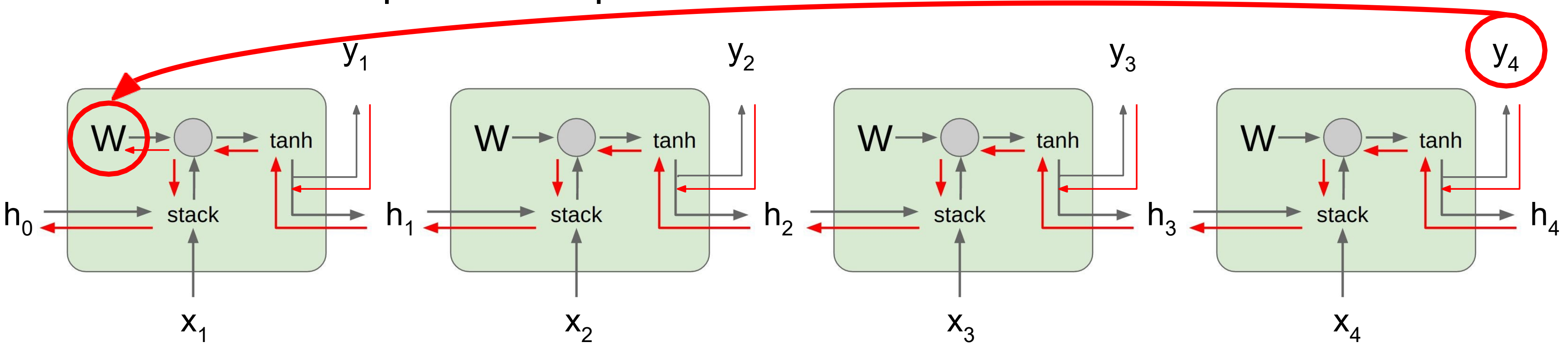



$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

# 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

Gradients over multiple time steps:



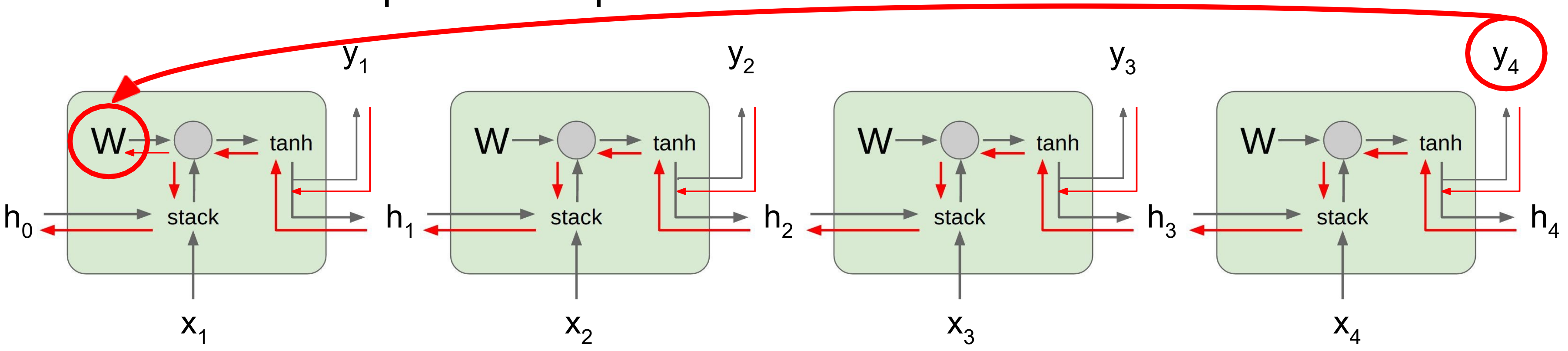

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial W}$$

# 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

Gradients over multiple time steps:



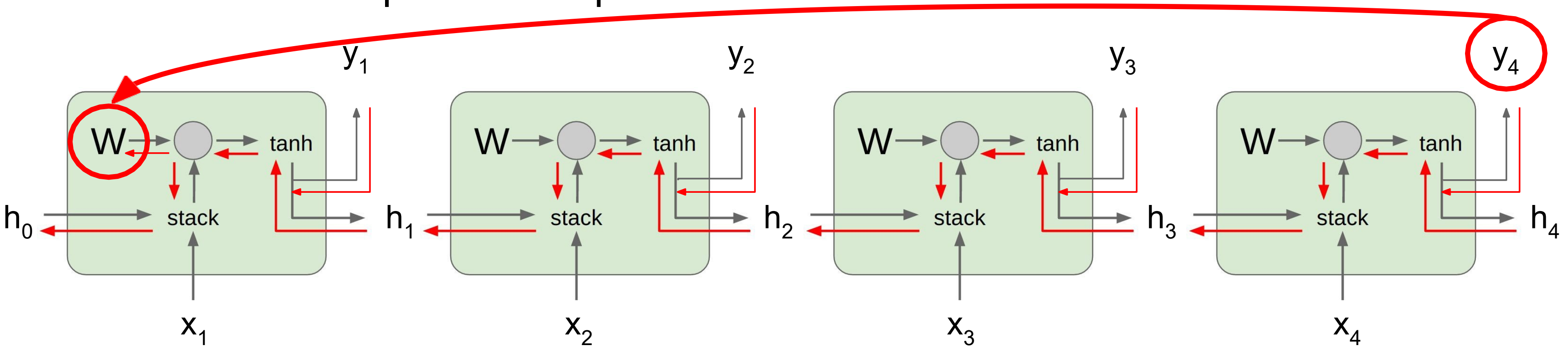

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial W} = \frac{\partial L_T}{\partial h_T} \left(\prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}}\right) \frac{\partial h_1}{\partial W}$$

# Vanilla RNN Gradient Flow

Gradients over multiple time steps:

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



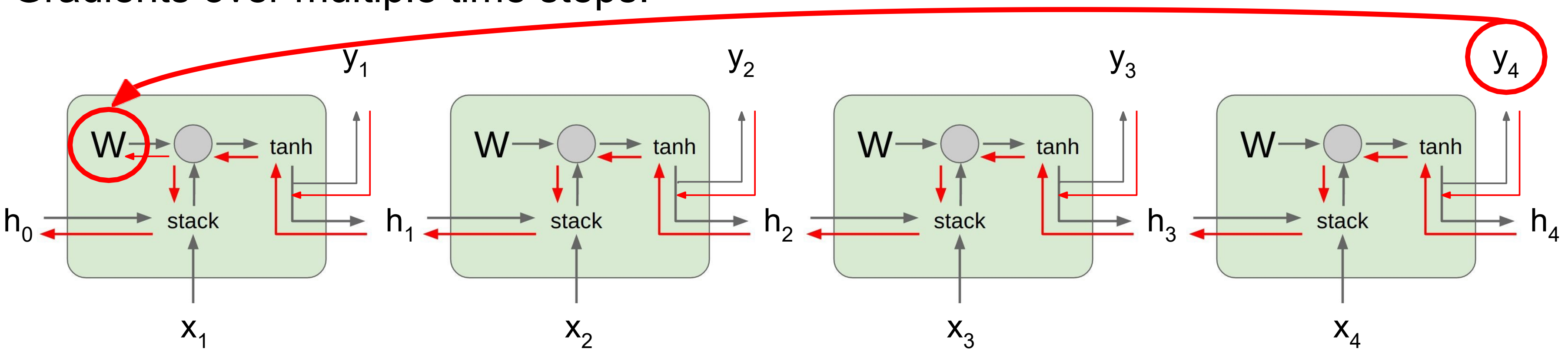

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W} \qquad \boxed{\frac{\partial h_t}{\partial h_{t-1}} = tanh'(W_{hh}h_{t-1} + W_{xh}x_t)W_{hh}}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T}\frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial W} = \frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T} \boxed{\frac{\partial h_t}{\partial h_{t-1}}}\right)\frac{\partial h_1}{\partial W}$$

# 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

Gradients over multiple time steps:




$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

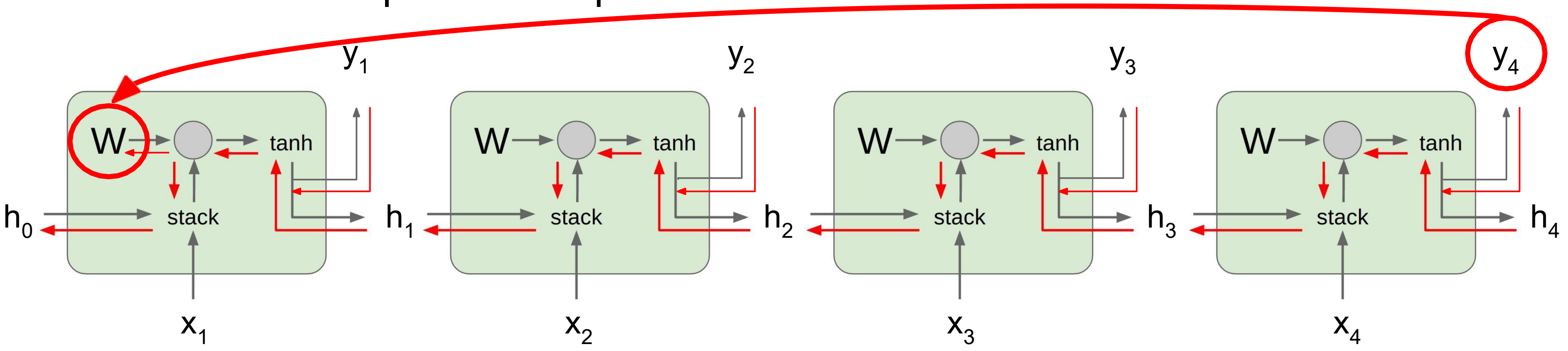
Almost always < 1
**Vanishing gradients**

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T} \boxed{tanh'(W_{hh}h_{t-1} + W_{xh}x_t)}\right)W_{hh}^{T-1}\frac{\partial h_1}{\partial W}$$

# 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

Gradients over multiple time steps:



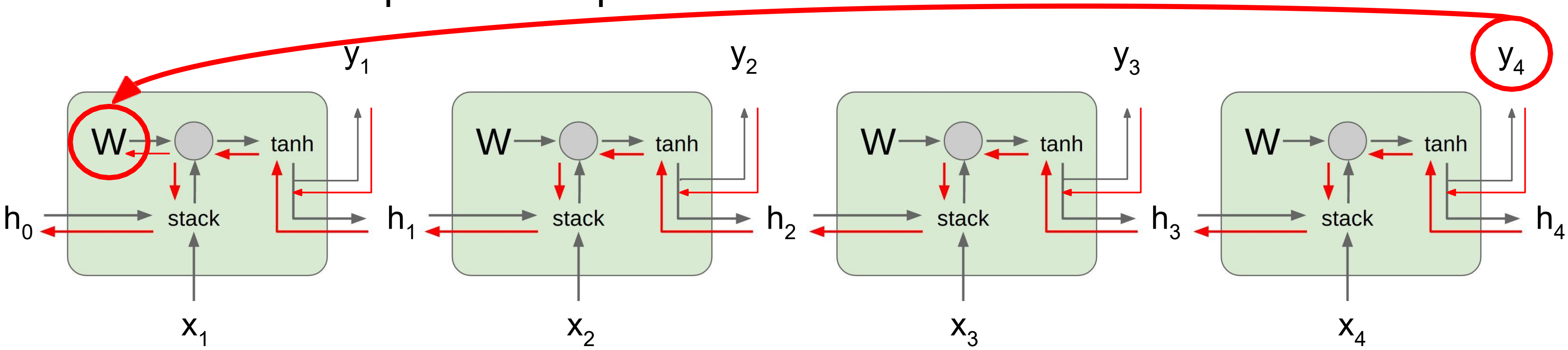

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

What if we assumed no non-linearity?

# 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

Gradients over multiple time steps:




What if we assumed no non-linearity?

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \boxed{W_{in}^{T-1}} \frac{\partial h_1}{\partial W}$$

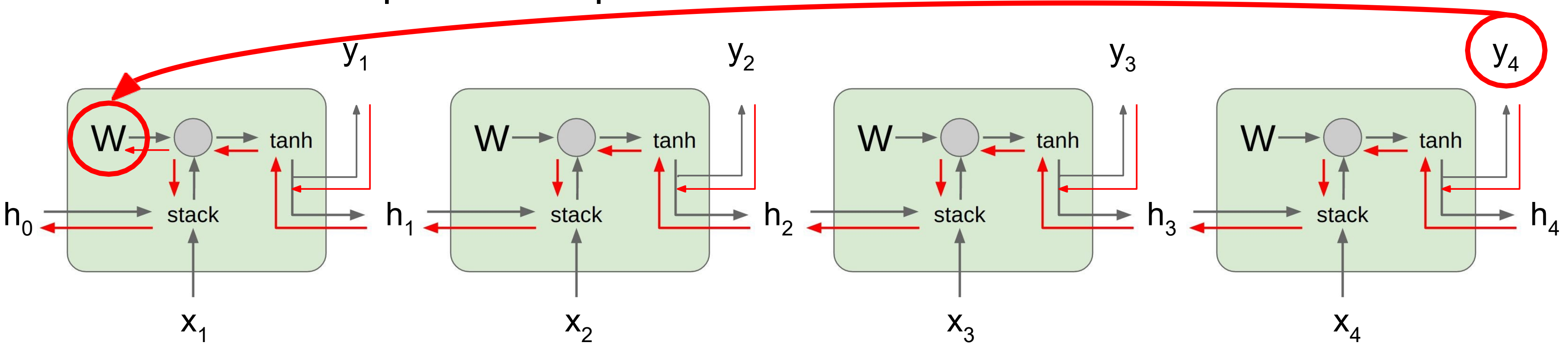
Largest singular value > 1:
**Exploding gradients**

Largest singular value < 1:
**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

Gradients over multiple time steps:




What if we assumed no non-linearity?

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \boxed{W_{hh}^{T-1}} \frac{\partial h_1}{\partial W}$$

Largest singular value > 1:
**Exploding gradients**

→ **Gradient clipping**: Scale gradient if its norm is too big

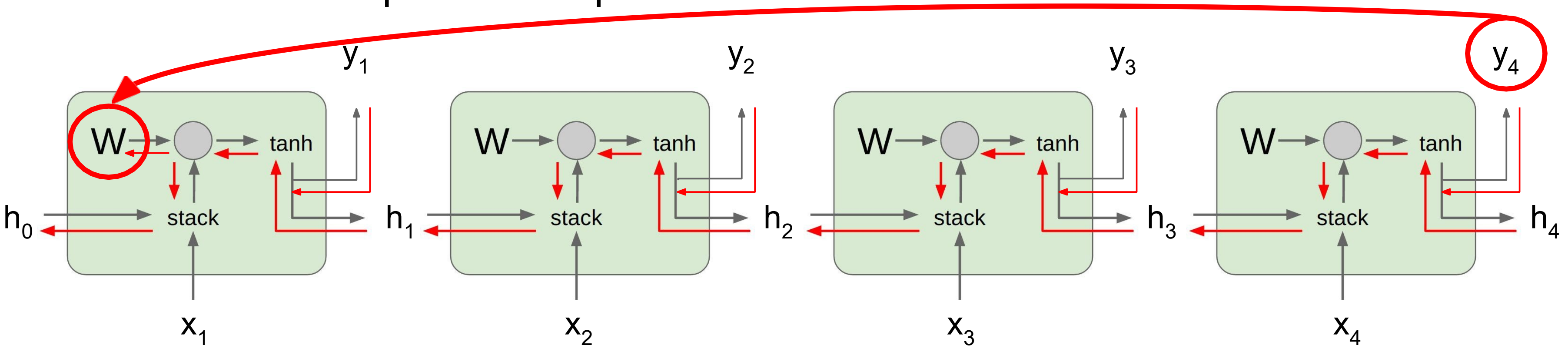
```
grad_norm = np.sum(grad * grad)
if grad_norm > threshold:
  grad *= (threshold / grad_norm)
```

Largest singular value < 1:
**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

Gradients over multiple time steps:




What if we assumed no non-linearity?

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \boxed{W_{in}^{T-1}} \frac{\partial h_1}{\partial W}$$

Largest singular value > 1:
**Exploding gradients**

Largest singular value < 1:
**Vanishing gradients**

→ Change RNN architecture

# Long Short Term Memory (LSTM)

## Vanilla RNN

$$h_t = \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

## LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

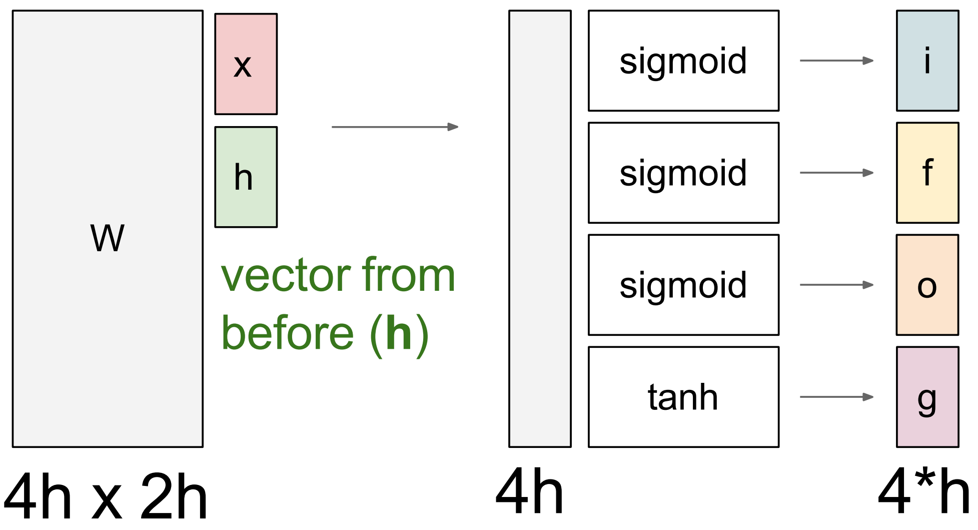
$$h_t = o \odot \tanh(c_t)$$

Hochreiter and Schmidhuber, “Long Short Term Memory”, Neural Computation 1997

# Long Short Term Memory (LSTM)

*[Hochreiter et al., 1997]*

**i**: Input gate, whether to write to cell
**f**: Forget gate, Whether to erase cell
**o**: Output gate, How much to reveal cell
**g**: Gate gate (?), How much to write to cell



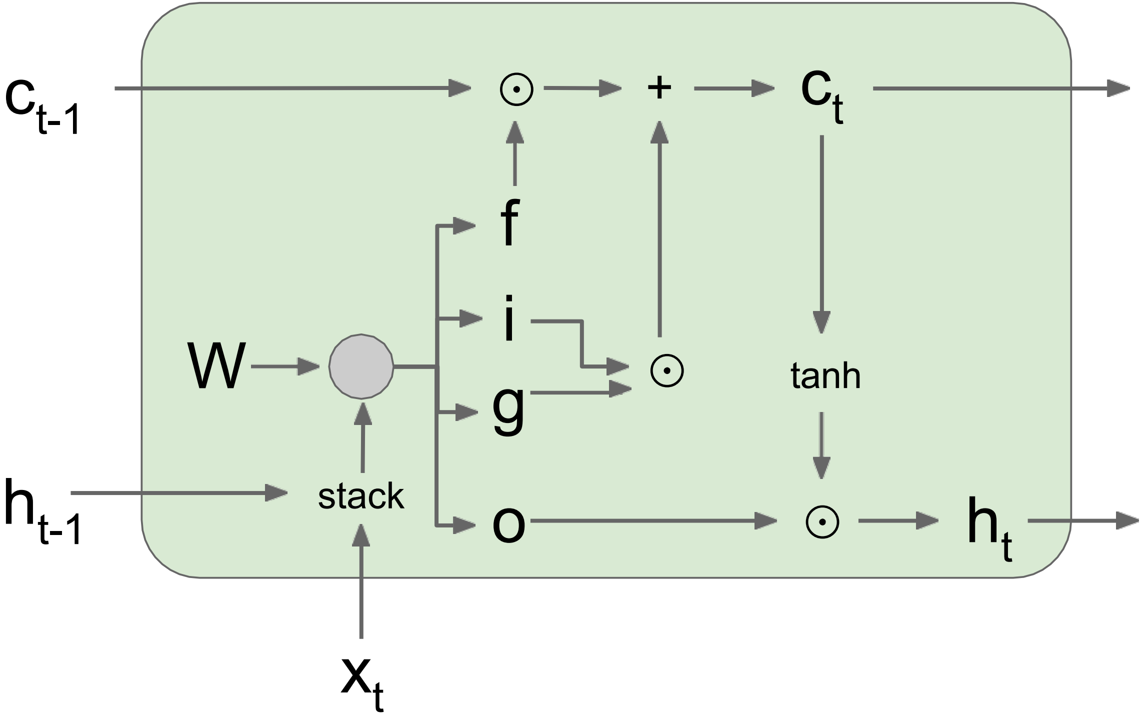

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

# Long Short Term Memory (LSTM)

*[Hochreiter et al., 1997]*



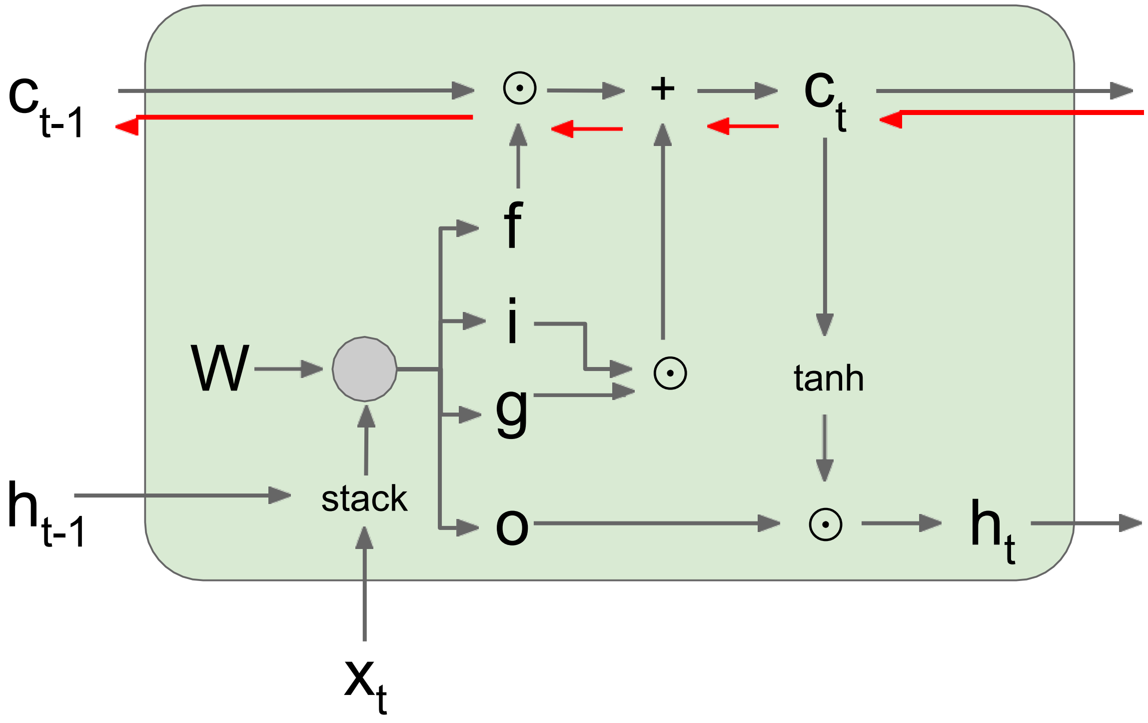

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

# Long Short Term Memory (LSTM): Gradient Flow

*[Hochreiter et al., 1997]*




Backpropagation from $c_t$ to $c_{t-1}$ only elementwise multiplication by f, no matrix multiply by W

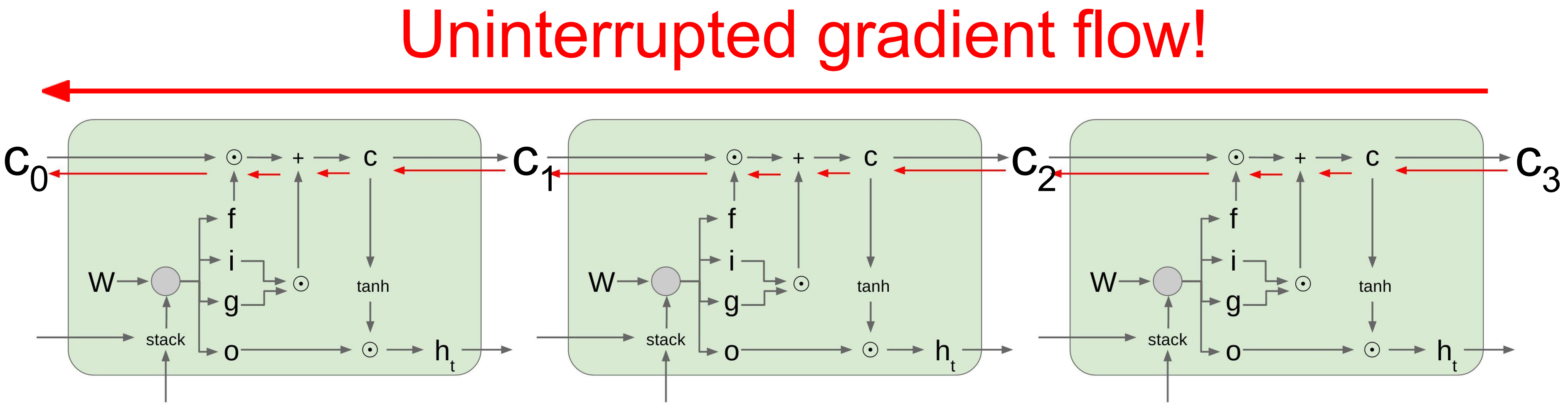
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

# Long Short Term Memory (LSTM): Gradient Flow

*[Hochreiter et al., 1997]*




Notice that the gradient contains the **f** gate's vector of activations

- allows better control of gradients values, using suitable parameter updates of the forget gate.

Also notice that are added through the **f, i, g,** and **o** gates

- better balancing of gradient values

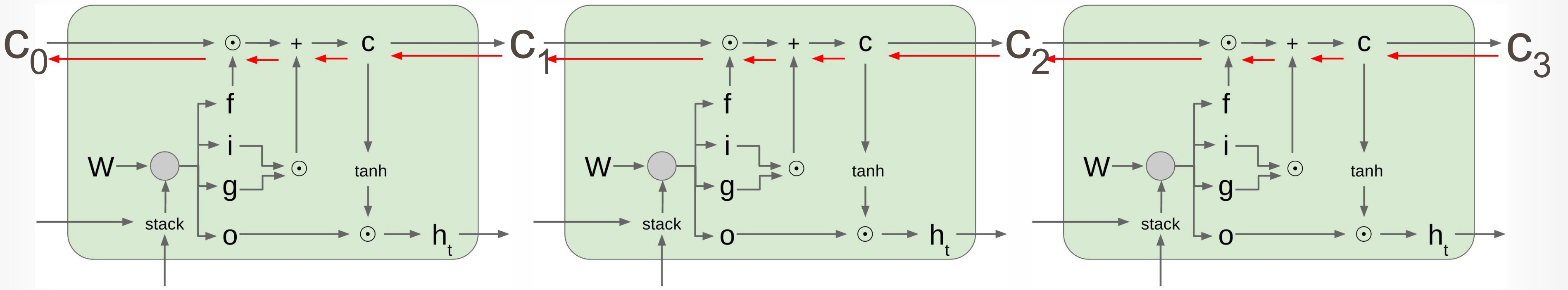
# Do LSTMs solve the vanishing gradient problem?

- The LSTM architecture makes it easier for the RNN to preserve information over many timesteps
  - e.g. if the f = 1 and the i = 0, then the information of that cell is preserved indefinitely.
  - By contrast, it's harder for vanilla RNN to learn a recurrent weight matrix Wh that preserves info in hidden state •

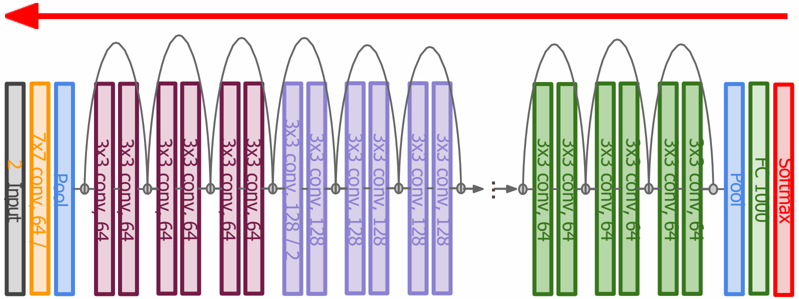
- LSTM doesn't guarantee that there is no vanishing/exploding gradient, but it does provide an easier way for the model to learn long-distance dependencies

# Long Short Term Memory (LSTM): Gradient Flow

Uninterrupted gradient flow!




Similar to ResNet!




In between:

**Highway Networks**

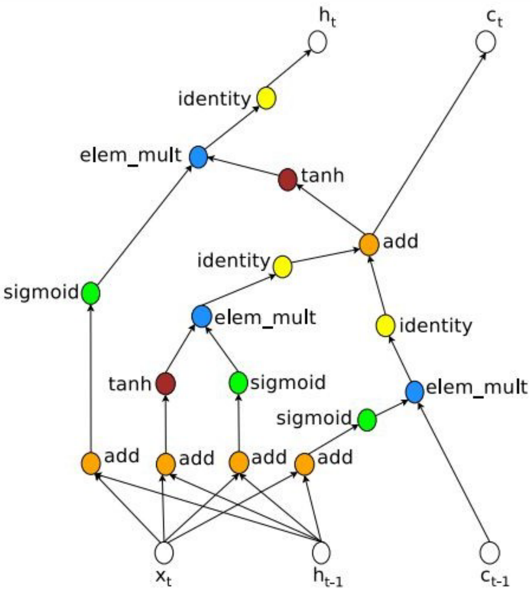
$$g = T(x, W_T)$$

$$y = g \odot H(x, W_H) + (1 - g) \odot x$$

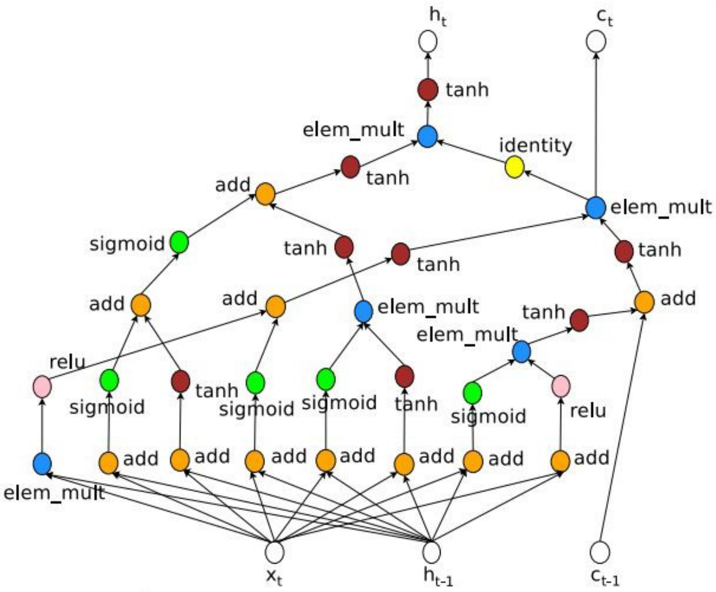
Srivastava et al, “Highway Networks”, ICML DL Workshop 2015

[Hochreiter et al., 1997]

# Neural Architecture Search for RNN architectures




LSTM cell




Cell they found

Zoph et Le, “Neural Architecture Search with Reinforcement Learning”, ICLR 2017

# Other RNN Variants

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

**GRU** [*Learning phrase representations using rnn encoder-decoder for statistical machine translation*, Cho et al. 2014]

$$
\begin{aligned}
r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \\
z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \\
\tilde{h}_t &= \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h) \\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t
\end{aligned}
$$

[*LSTM: A Search Space Odyssey*, Greff et al., 2015]

MUT1:

$$
\begin{aligned}
z &= \mathrm{sigm}(W_{\mathrm{xz}}x_t + b_{\mathrm{z}}) \\
r &= \mathrm{sigm}(W_{\mathrm{xr}}x_t + W_{\mathrm{hr}}h_t + b_{\mathrm{r}}) \\
h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + \tanh(x_t) + b_{\mathrm{h}}) \odot z \\
&+ h_t \odot (1 - z)
\end{aligned}
$$

MUT2:

$$
\begin{aligned}
z &= \mathrm{sigm}(W_{\mathrm{xz}}x_t + W_{\mathrm{hz}}h_t + b_{\mathrm{z}}) \\
r &= \mathrm{sigm}(x_t + W_{\mathrm{hr}}h_t + b_{\mathrm{r}}) \\
h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + W_{xh}x_t + b_{\mathrm{h}}) \odot z \\
&+ h_t \odot (1 - z)
\end{aligned}
$$

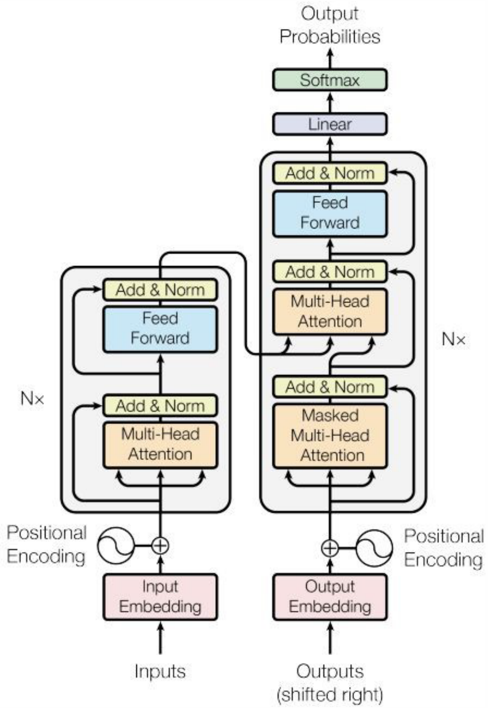
MUT3:

$$
\begin{aligned}
z &= \mathrm{sigm}(W_{\mathrm{xz}}x_t + W_{\mathrm{hz}}\tanh(h_t) + b_{\mathrm{z}}) \\
r &= \mathrm{sigm}(W_{\mathrm{xr}}x_t + W_{\mathrm{hr}}h_t + b_{\mathrm{r}}) \\
h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + W_{xh}x_t + b_{\mathrm{h}}) \odot z \\
&+ h_t \odot (1 - z)
\end{aligned}
$$

# Recently in Natural Language Processing

**New paradigms for reasoning over sequences**

- ["Attention is all you need", Vaswani et al., 2018]
- New "Transformer" architecture no longer processes inputs sequentially; instead it can operate over inputs in a sequence in parallel through an attention mechanism
- Has led to many state-of-the-art results and pre-training in NLP, for more interest see e.g.
- "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", Devlin et al., 2018
- OpenAI GPT-2, Radford et al., 2018



# Transformers for Vision

- LSTM is a good default choice
- Use variants like GRU if you want faster compute and less  parameters
- Use transformers (not covered in this lecture) as they are
- dominating NLP models
  - We need more work studying vision models in tandem with transformers

Su et al. "Vl-bert: Pre-training of generic visual-linguistic representations." ICLR 2020
Lu et al. "Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks." NeurIPS 2019
Li et al. "Visualbert: A simple and performant baseline for vision and language." *arXiv* 2019

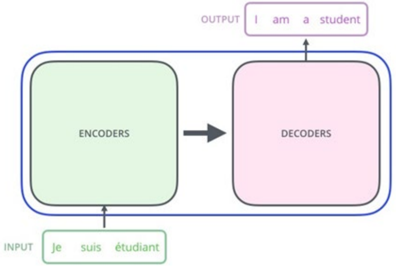
# Summary

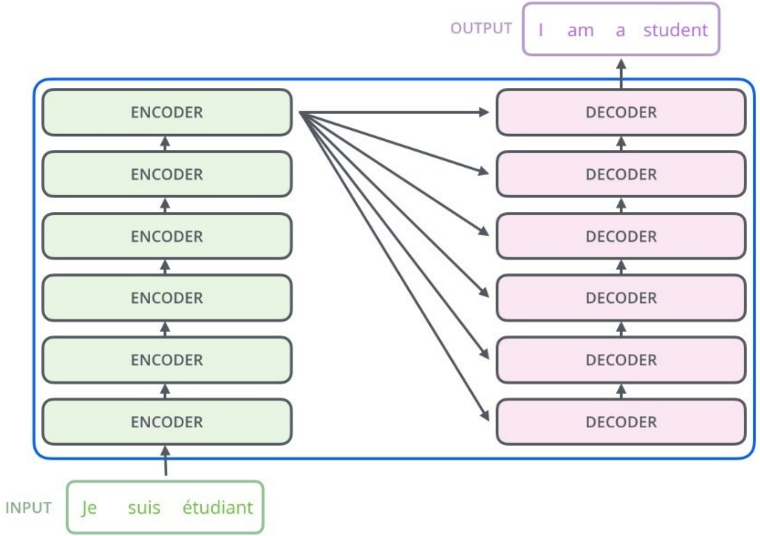
- RNNs allow a lot of flexibility in architecture design
- Vanilla RNNs are simple but don't work very well
- Common to use LSTM or GRU: their additive interactions  improve gradient flow
- Backward flow of gradients in RNN can explode or vanish.
- Exploding is controlled with gradient clipping. Vanishing is  controlled with additive interactions (LSTM)
- Better/simpler architectures are a hot topic of current research,
- as well as new paradigms for reasoning over sequences
- Better understanding (both theoretical and empirical) is needed.

# BEYOND RNNS

# Transformer

- In NLP, transformer is the first try without RNN
  - RNN relies on “ordered-data”
  - RNN needs to computing sequentially
    - Slow




[1] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need. NPIS. 2017: 5998-6008.

# Sequence-to-sequence is everywhere!

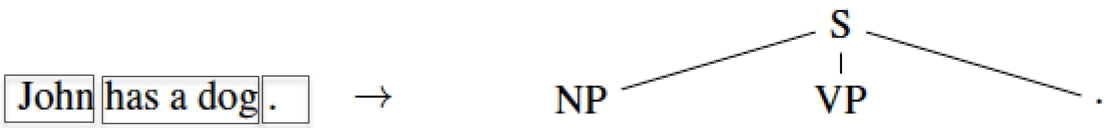
- Sequence-to-sequence is useful for *more than just MT*
- Many NLP tasks can be phrased as sequence-to-sequence:
  - Summarization (long text → short text)
  - Dialogue (previous utterances → next utterance)
  - Parsing (input text → output parse as sequence)
  - Code generation (natural language → Python code)

# Grammar as a Foreign Language

- Vinyals et al., 2015

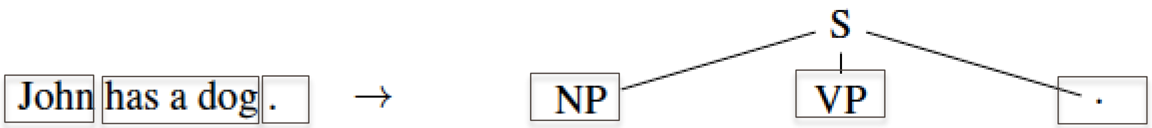
# Grammar as a Foreign Language

Parsing tree



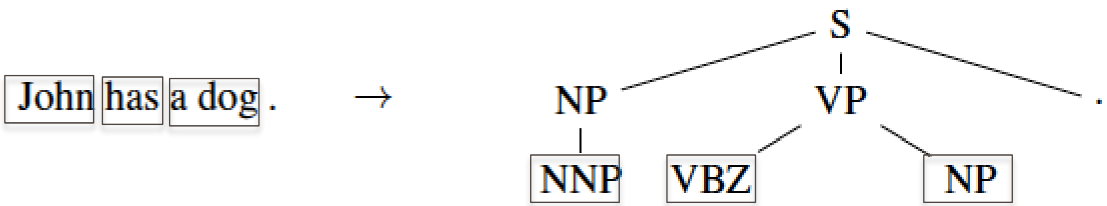
# Grammar as a Foreign Language

Parsing tree



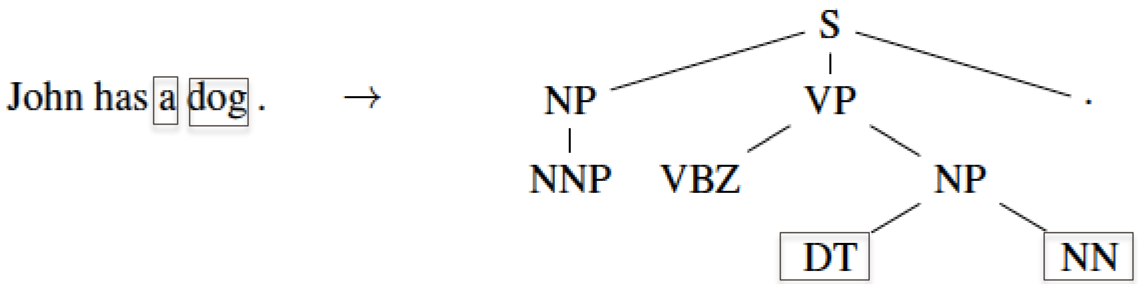
# Grammar as a Foreign Language

Parsing tree



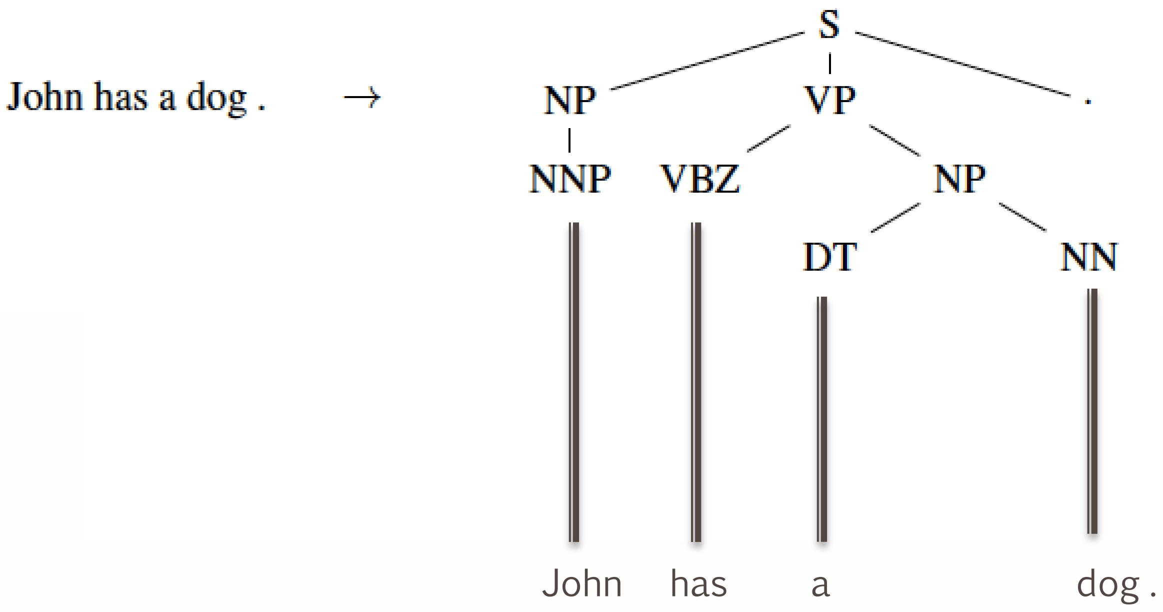
# Grammar as a Foreign Language

Parsing tree



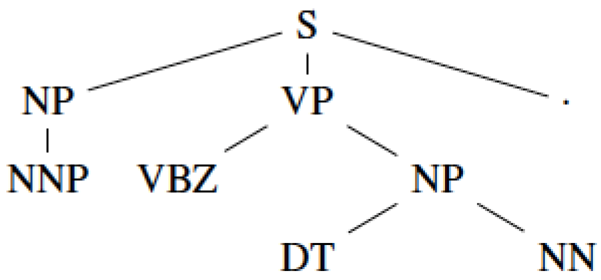
# Grammar as a Foreign Language

Parsing tree



# Grammar as a Foreign Language

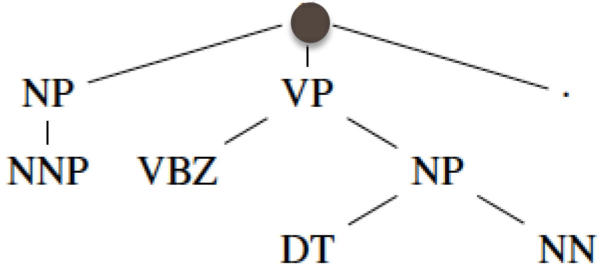
## Converting tree to sequence




$(S\ (NP\ NNP\ )_{NP}\ (VP\ VBZ\ (NP\ DT\ NN\ )_{NP}\ )_{VP}\ .\ )_{S}$

# Grammar as a Foreign Language

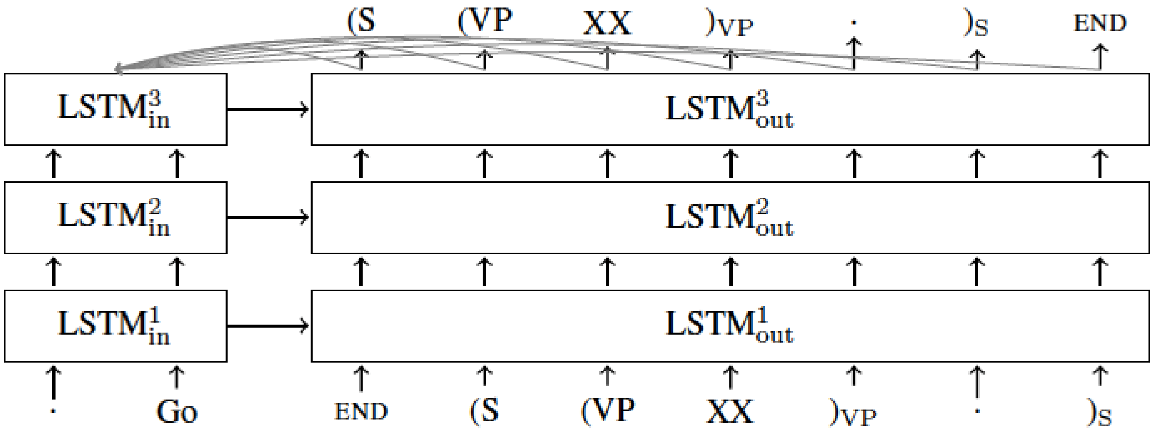
## Converting tree to sequence




( (NP NNP $)_{NP}$ (VP VBZ (NP DT NN $)_{NP}$ $)_{VP}$ . $)_{S}$

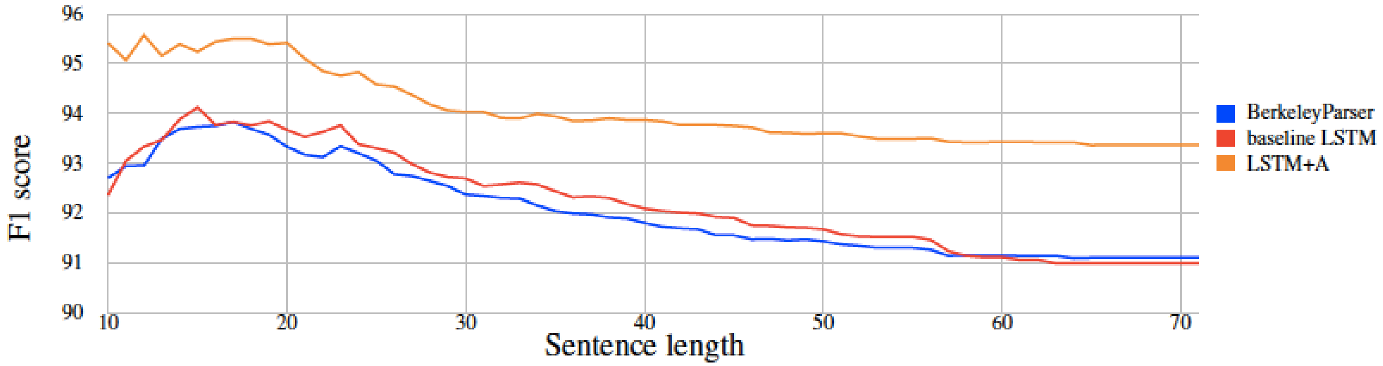
# Grammar as a Foreign Language

Model



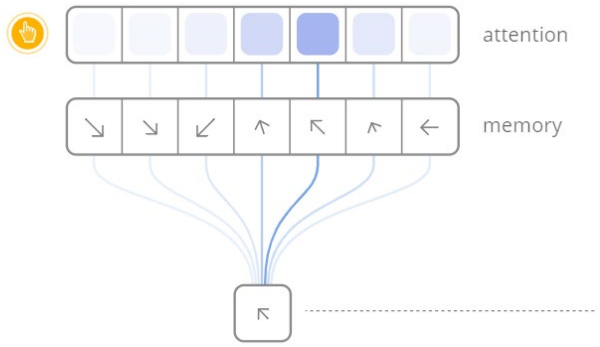
# Grammar as a Foreign Language

## Results



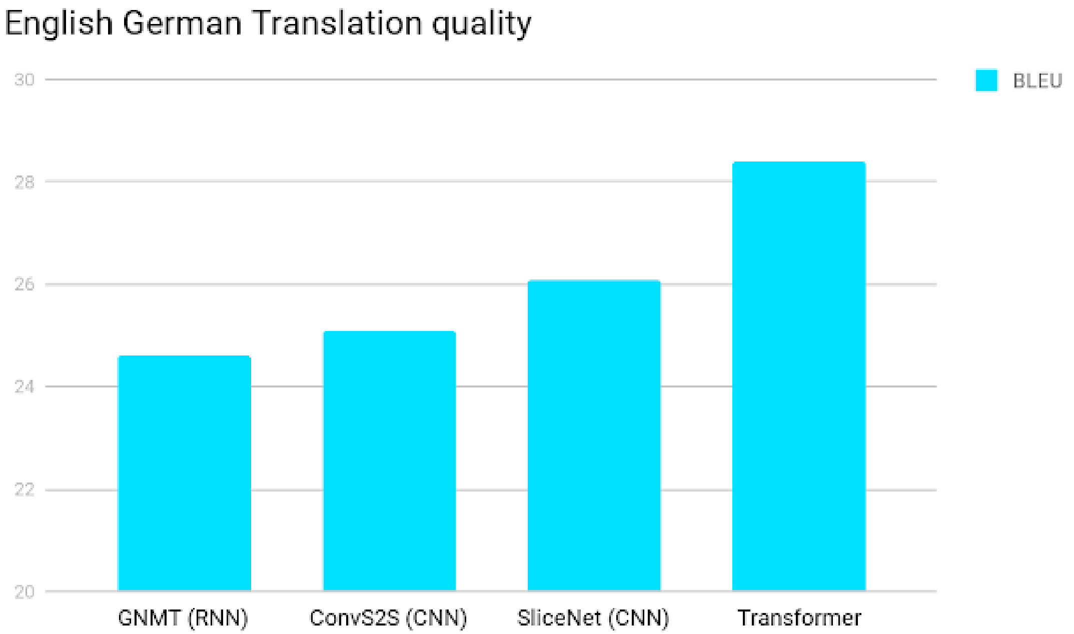
# Attention is a *general* Deep Learning technique

- More general definition of attention
  - Given a set of vector *values*, and a vector *query*, **attention** is a technique to compute a weighted sum of the values, dependent on the query.
  - We sometimes say that the query *attends to* the values
  - For example, in the seq2seq + attention model, each decoder hidden state *attends to* the encoder hidden states




https://distill.pub/2016/augmented-rnns/

# Attention is all you need? – Transformer Networks




https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

partial credit by CS311n

# Problems with RNNs = Motivation for Transformers

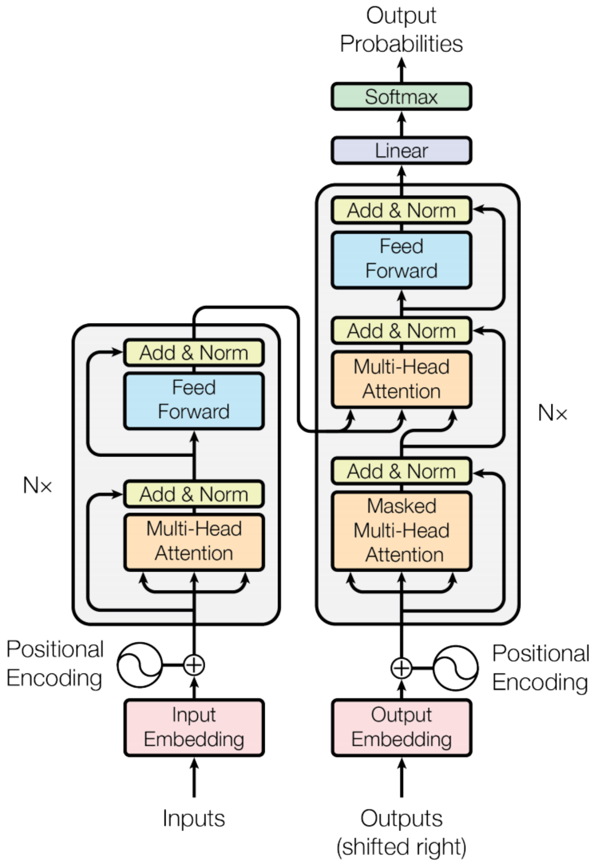
- Recurrent models typically factor computation along the symbol positions of the input and output sequences
  - Sequential computation prevents parallelization
  - Critical at longer sequence lengths, as memory constraints limit batching across examples

- Despite advanced RNNs like LSTMs, RNNs still need attention mechanism to deal with long range dependencies – path length for codependent computation between states grows with sequence

- But if attention gives us access to any state… maybe we don't need the RNN?

# Transformer Overview

- Sequence-to-sequence
- Encoder-Decoder
- Task: machine translation with parallel corpus
- Predict each translated word
- Final cost/error function is standard cross-entropy error on top of a softmax classifier




https://arxiv.org/pdf/1706.03762.pdf

# Transformer Basics

- Let's define the basic building blocks of transformer networks first: new attention layers!

# Dot-Product Attention (Extending our previous def.)

- Inputs: a query q and a set of key-value (k-v) pairs to an output
  - Query, keys, values, and output are all vectors
- Output is weighted sum of values, where
  - Weight of each value is computed by an inner product of query and corresponding key
  - Queries and keys have same dimensionality

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

# Dot-Product Attention – Matrix notation

- When we have multiple queries q, we stack them in a matrix Q:

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

- Becomes:

$$A(Q, K, V) = softmax(QK^T)V$$

$[|Q| \times d_k] \times [d_k \times |K|] \times [|K| \times d_v]$

softmax
row-wise

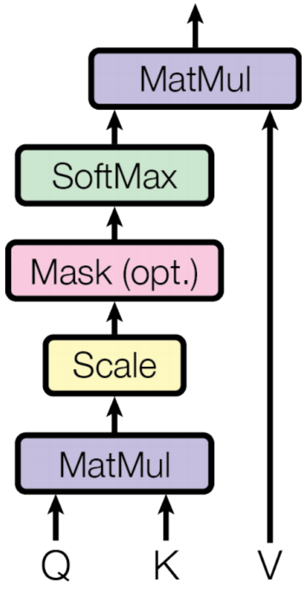
$= [|Q| \times d_v]$

# Scaled Dot-Product Attention

- Problem: As $d_k$ gets large, the variance of $q^T k$ increases → some values inside the softmax get large → the softmax gets very peaked → hence its gradient gets smaller.

- Solution: Scale by length of query/key vectors:

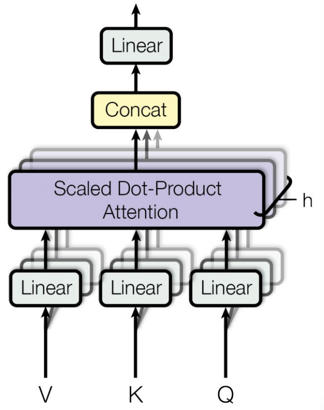
$$A(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$



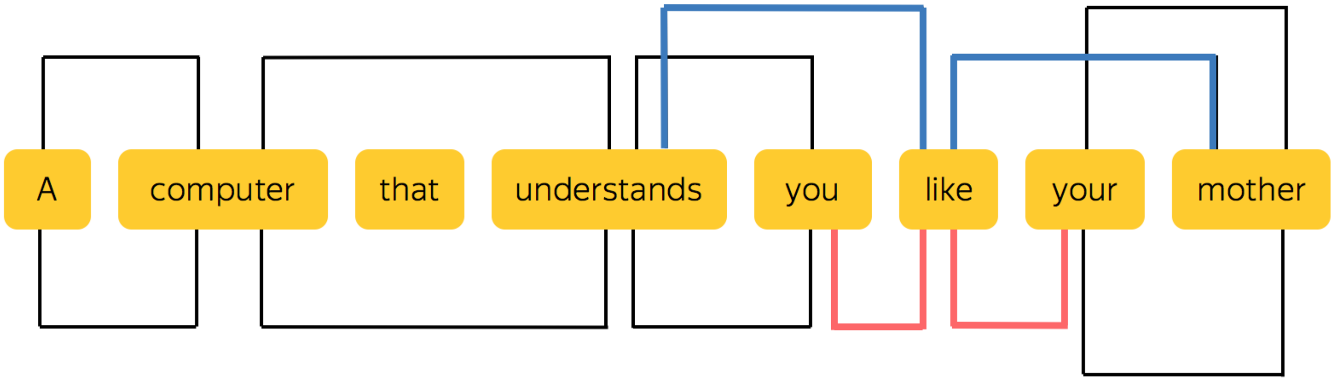
# Self-attention and Multi-head attention

- The input word vectors could be the queries, keys and values
  - In other words: the word vectors themselves select each other
  - Word vector stack = Q = K = V
- Problem: Only one way for words to interact with one-another
- Solution: Multi-head attention
  - First map Q, K, V into h many lower dimensional spaces via W matrices
  - Then apply attention, then concatenate outputs and pipe through linear layer

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head_1}, ..., \mathrm{head_h})W^O$$
$$\text{where } \mathrm{head_i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

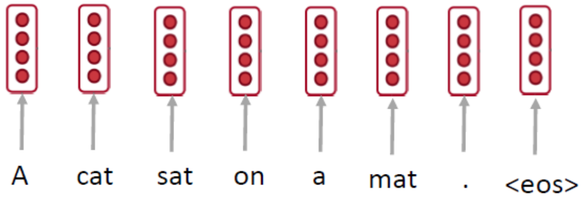


# Self-Attention



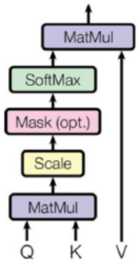

(example and picture from David Talbot)

# Self-attention: A Running Example

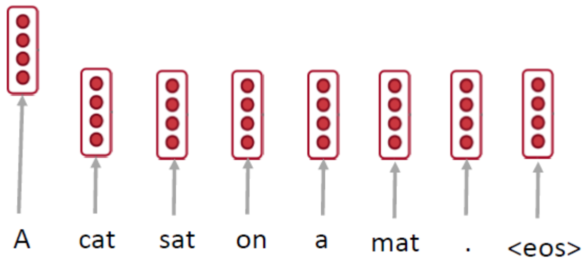



$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Scaled Dot-Product Attention

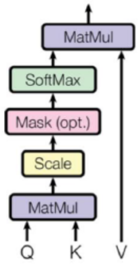


# Self-attention: A Running Example

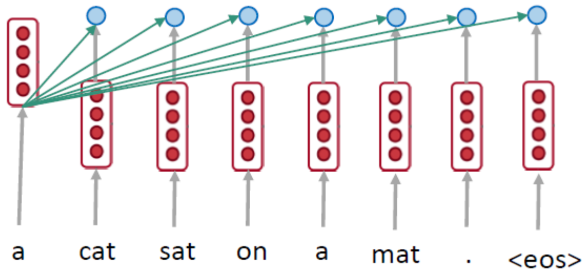



$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Scaled Dot-Product Attention

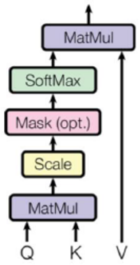


# Self-attention: A Running Example

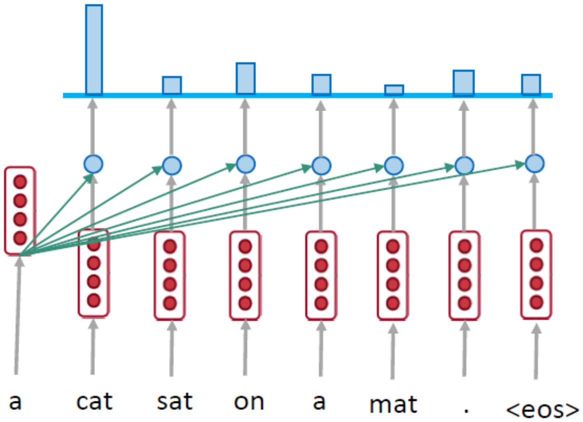



$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Scaled Dot-Product Attention

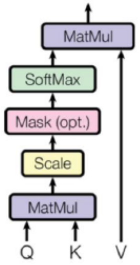


# Self-attention: A Running Example

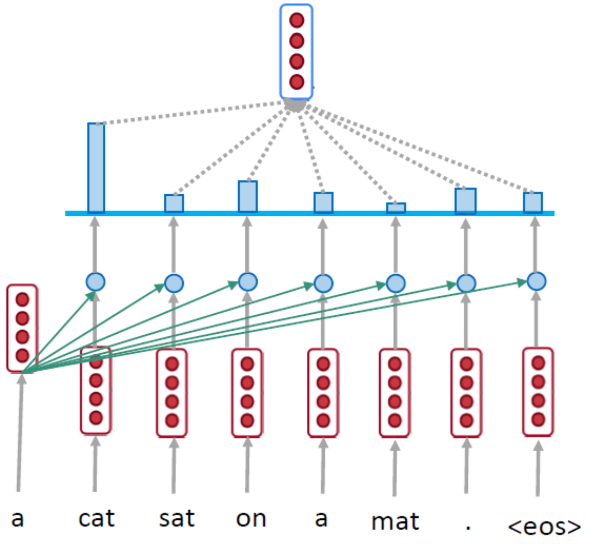



$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Scaled Dot-Product Attention

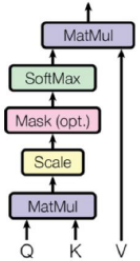


# Self-attention: A Running Example



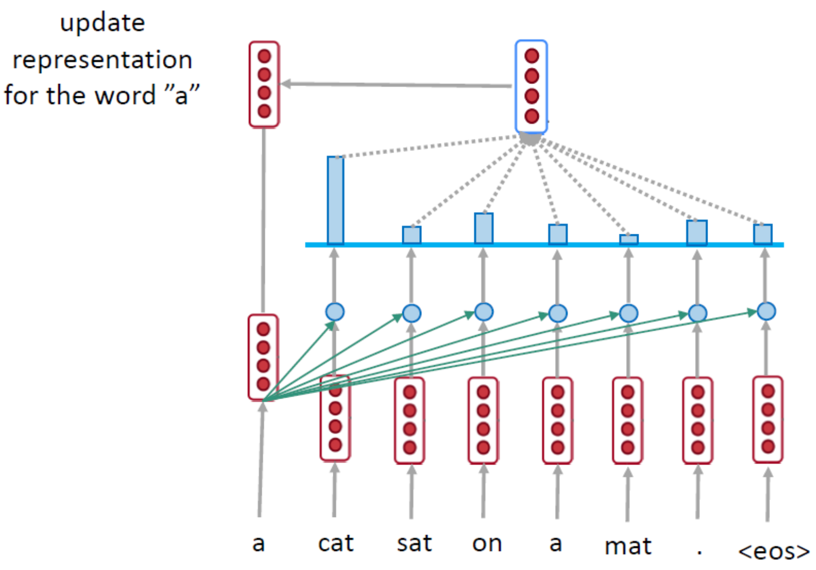

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Scaled Dot-Product Attention



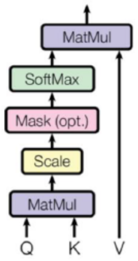

partial credit by CS311n

# Self-attention: A Running Example



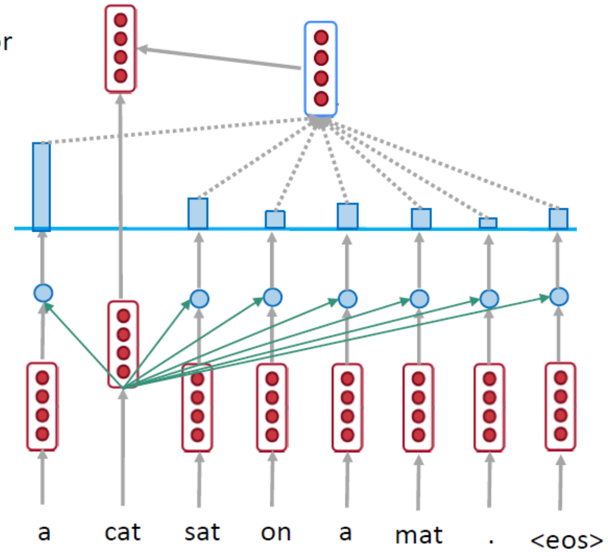

$$\mathrm{Attention}(Q,K,V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Scaled Dot-Product Attention



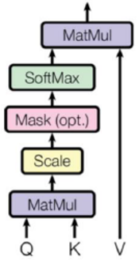
# Self-attention: A Running Example

update
representation for
the word "cat"

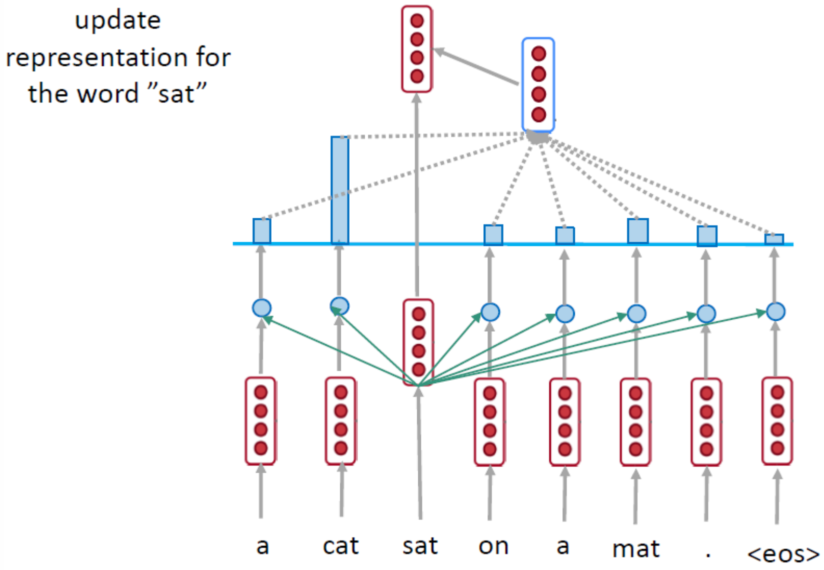



$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Scaled Dot-Product Attention

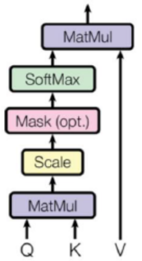


# Self-attention: A Running Example

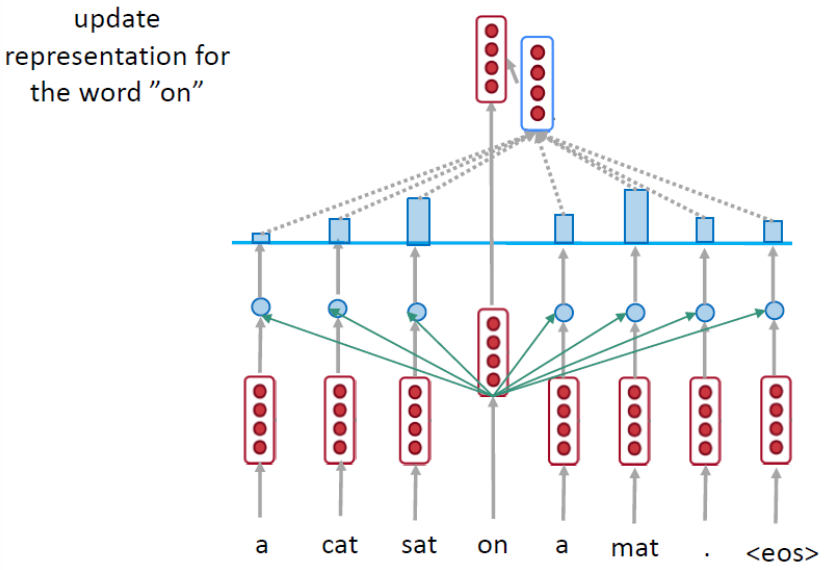



$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Scaled Dot-Product Attention

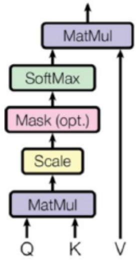


# Self-attention: A Running Example



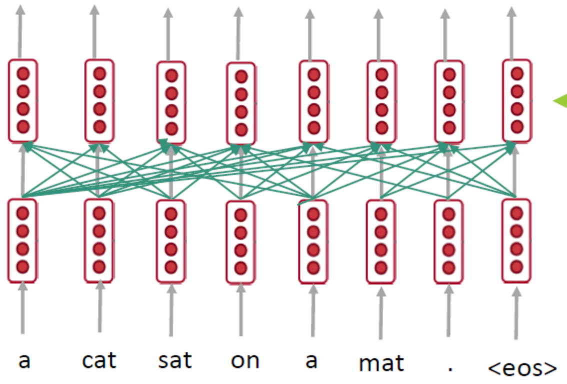

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Scaled Dot-Product Attention



# Self-attention: A Running Example

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

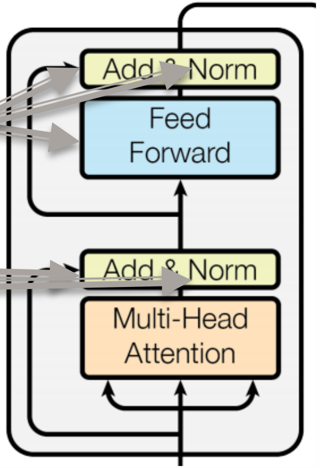



Updated representations
for each word
(one layer)

# Complete transformer block

- Each block has two "sublayers"
  - 1. Multihead attention
  - 2. 2 layer feed-forward Nnet (with relu)

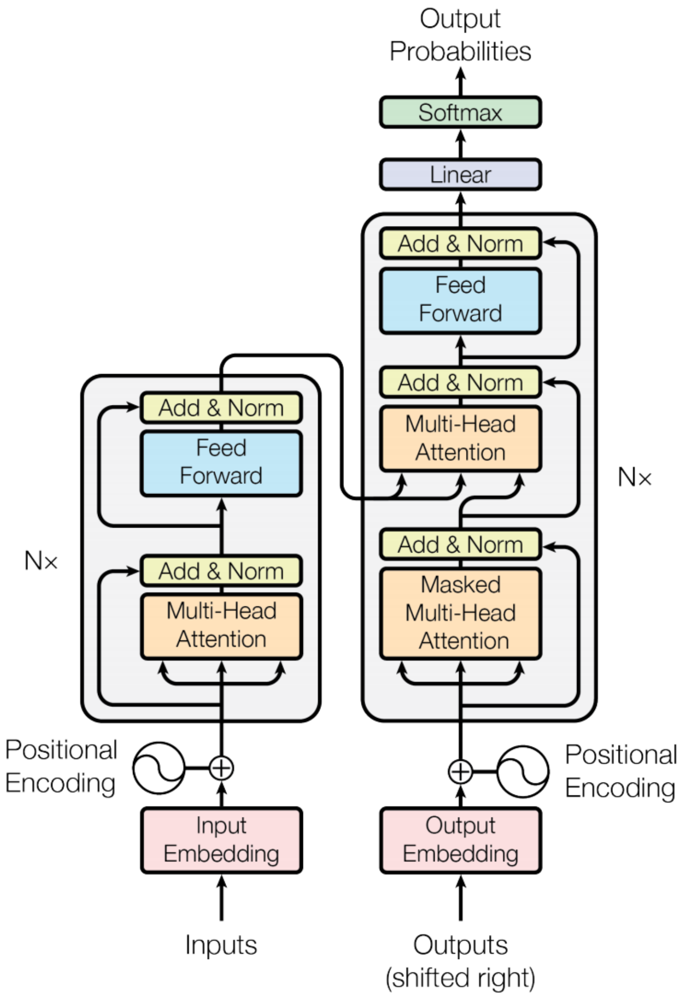
- Each of these two steps also has:
  - Residual (short-circuit) connection and LayerNorm:
  - LayerNorm(x + Sublayer(x))
    - Layernorm changes input to have mean 0 and variance 1, per layer and per training point (and adds two more parameters)




$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2} \qquad h_i = f\left(\frac{g_i}{\sigma_i}(a_i - \mu_i) + b_i\right)$$

Layer Normalization by Ba, Kiros and Hinton, https://arxiv.org/pdf/1607.06450.pdf

Blocks are repeated N=6 times

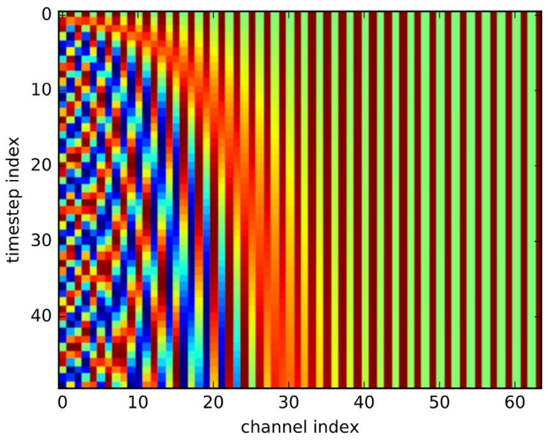


# Encoder Input

- Actual word representations are byte-pair encodings
  - Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural Machine Translation of Rare Words with Subword Units. ACL 2016.
- Added is a positional encoding so same words at different locations have different overall representations:

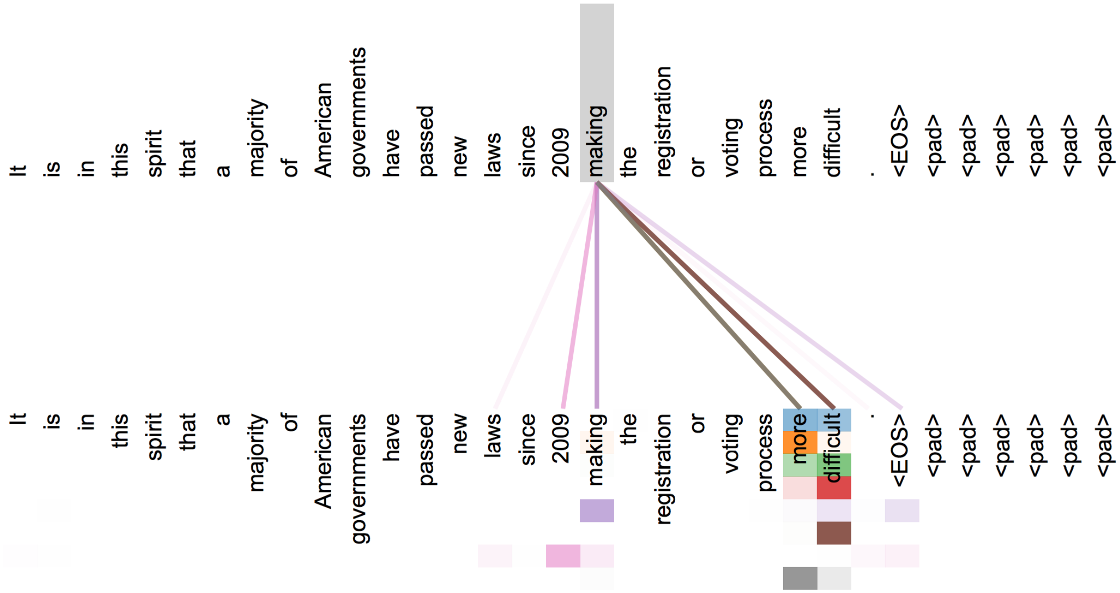
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

*pos* is the position of a word
*i* is the dimension index



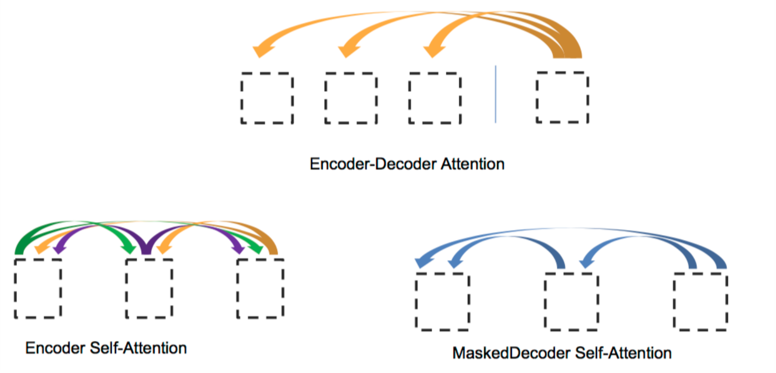
# Self Attention Visualization in Layer 5

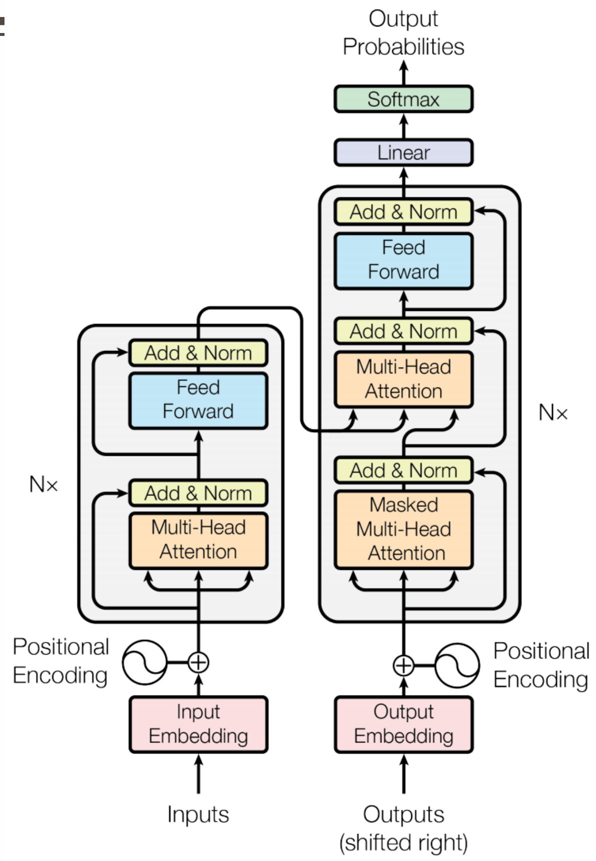
- Words start to pay attention to other words in sensible ways



# Transformer Decoder

- 2 sublayer changes in decoder
- Masked decoder self-attention
  - Only depends on previous words
- Encoder-Decoder Attention
  - Queries come from previous decoder layer and keys and values come from output of encoder

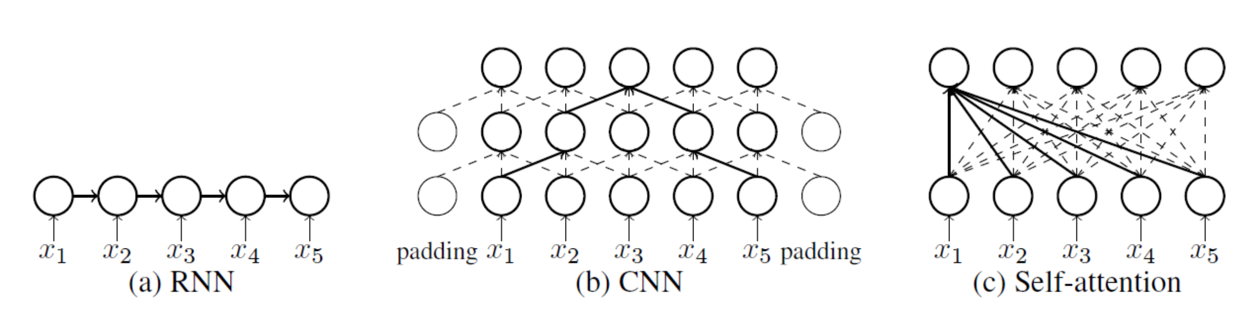


# Advantages

- No recurrence: parallel encoding
- Fast training: both encoder and decoder are parallel
- No long range problem: O(1) for all tokens direct connections
- Three attentions: the model does not have to remember too much
- Multi-head attention allows to pay attention to different aspects

- Why se ion




A comparison of RNN, CNN, and self-attention
http://aclweb.org/anthology/D18-1458

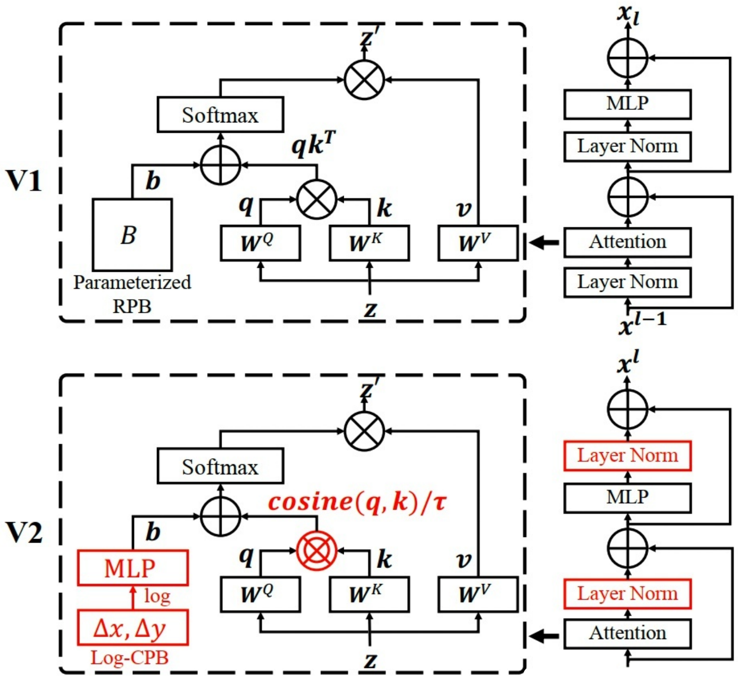
# Summary

- RNN:
  - Arbitrary length of the input sequences
  - Ordered information captured by RNN itself
  - First-order dependency considered
  - Slow (both training and inference)
- Transformer (Fully-connected layer actually):
  - Long-ranged dependency considered
  - Unordered information
    - Positional encoding required
  - High space complexity but fast

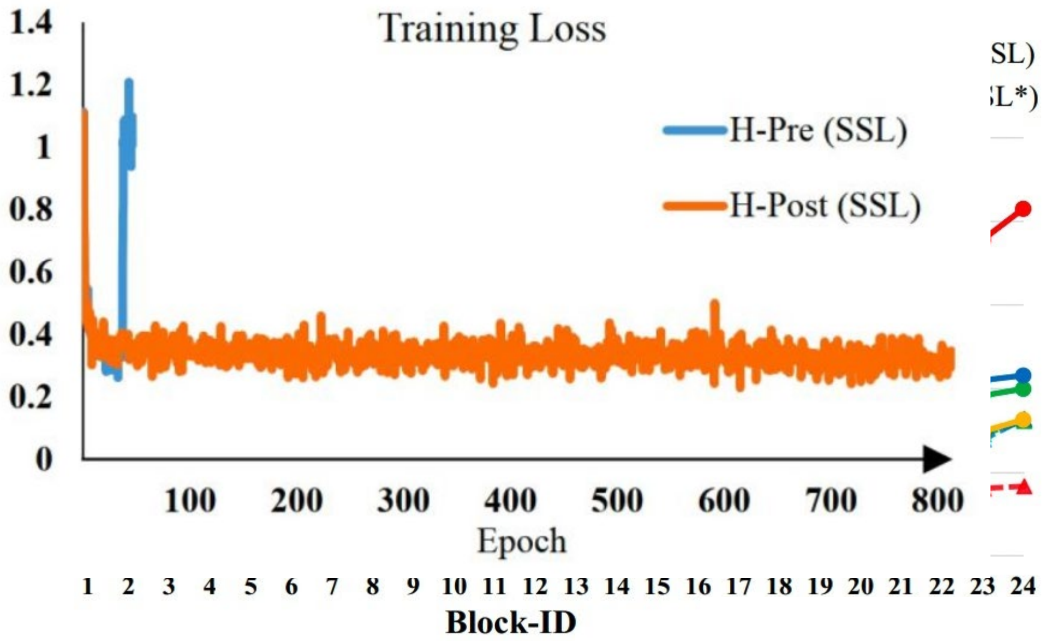
# SWIN-TRANSFORMER V2 [2022.4]

# Efficient Attention Block

- It is hard to scale up (unstable in the training phase)
- Ineffective for different scales setting
- GPU memory cost is large
- Solution:
    - Post normalization
    - Scaled cosine attention approach
    - Log-spaced continuous position bias (Log-CPB)



# Post normalization



# Scaled cosine attention

- Since post-normalization results in some dominations in some blocks/head, attention should be improved

$$\mathrm{Sim}(\mathbf{q}_i, \mathbf{k}_j) = \cos(\mathbf{q}_i, \mathbf{k}_j)/\tau + B_{ij},$$

- Rescale by the factors of element norm B
  - Gamma is learnable parameter

# Log spaced CPB

| method | ImageNet* | ImageNet† | | | | COCO | | ADE20k | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | W8, I256 top-1 acc | W12, I384 top-1 acc | W16, I512 top-1 acc | W20, I640 top-1 acc | W24, I768 top-1 acc | W16 $AP^{box}$ | W32 $AP^{box}$ | W16 mIoU | W20 mIoU | W32 mIoU |
| Parameterized position bias [35] | 81.7 | 79.4/82.7 | 77.2/83.0 | 73.2/83.2 | 68.7/83.2 | 50.8 | 50.9 | 45.5 | 45.8 | 44.5 |
| Linear-Spaced CPB | 81.7 (+0.0) | 82.0/82.9 (+2.6/+0.2) | 81.2/83.3 (+4.0/+0.3) | 79.8/83.6 (+6.6/+0.4) | 77.6/83.7 (+8.9/+0.5) | 50.9 (+0.1) | 51.7 (+0.8) | 47.0 (+1.5) | 47.4 (+1.6) | 47.2 (+2.7) |
| Log-Spaced CPB | 81.8 (+0.1) | 82.4/83.2 (+3.0/+0.5) | 81.7/83.8 (+4.5/+0.8) | 80.4/84.0 (+7.2/+0.8) | 79.1/84.2 (+10.4/+1.0) | 51.1 (+0.3) | 51.8 (+0.9) | 47.0 (+1.5) | 47.7 (+1.9) | 47.8 (+3.3) |

- Solution
  - Learning a “position” predictor by network (G, e.g. , MLP)

$$B(\Delta x, \Delta y) = \mathcal{G}(\Delta x, \Delta y), \qquad (3)$$

# Log spaced CPB

$$\widehat{\Delta x} = \text{sign}(x) \cdot \log(1 + |\Delta x|),$$
$$\widehat{\Delta y} = \text{sign}(y) \cdot \log(1 + |\Delta y|), \quad (4)$$

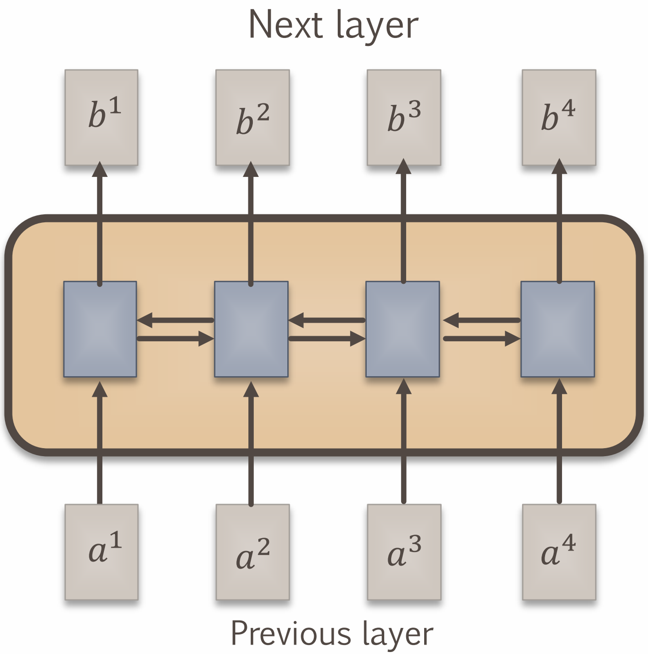
- Solving the mismatching between different window sizes

| method | ImageNet* | ImageNet† | | | | COCO | | ADE20k | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | W8, I256 top-1 acc | W12, I384 top-1 acc | W16, I512 top-1 acc | W20, I640 top-1 acc | W24, I768 top-1 acc | W16 $AP^{box}$ | W32 $AP^{box}$ | W16 mIoU | W20 mIoU | W32 mIoU |
| Parameterized position bias [35] | 81.7 | 79.4/82.7 | 77.2/83.0 | 73.2/83.2 | 68.7/83.2 | 50.8 | 50.9 | 45.5 | 45.8 | 44.5 |
| Linear-Spaced CPB | 81.7 (+0.0) | 82.0/82.9 (+2.6/+0.2) | 81.2/83.3 (+4.0/+0.3) | 79.8/83.6 (+6.6/+0.4) | 77.6/83.7 (+8.9/+0.5) | 50.9 (+0.1) | 51.7 (+0.8) | 47.0 (+1.5) | 47.4 (+1.6) | 47.2 (+2.7) |
| Log-Spaced CPB | 81.8 (+0.1) | 82.4/83.2 (+3.0/+0.5) | 81.7/83.8 (+4.5/+0.8) | 80.4/84.0 (+7.2/+0.8) | 79.1/84.2 (+10.4/+1.0) | 51.1 (+0.3) | 51.8 (+0.9) | 47.0 (+1.5) | 47.7 (+1.9) | 47.8 (+3.3) |

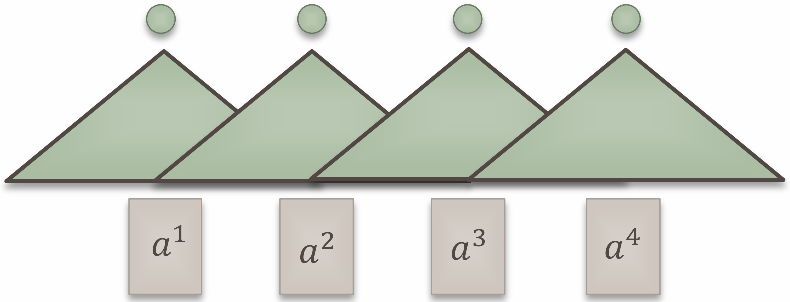
# APPENDIX

Multi-head self-attention

# Sequence



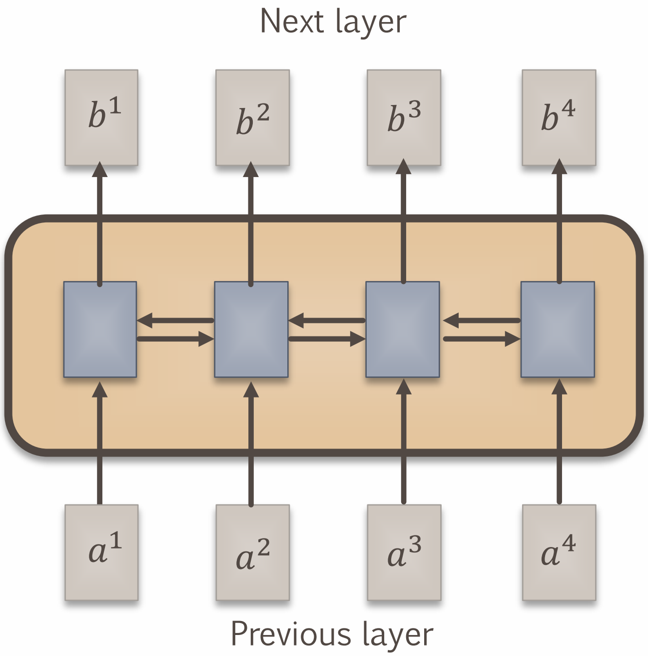

Hard to parallel !



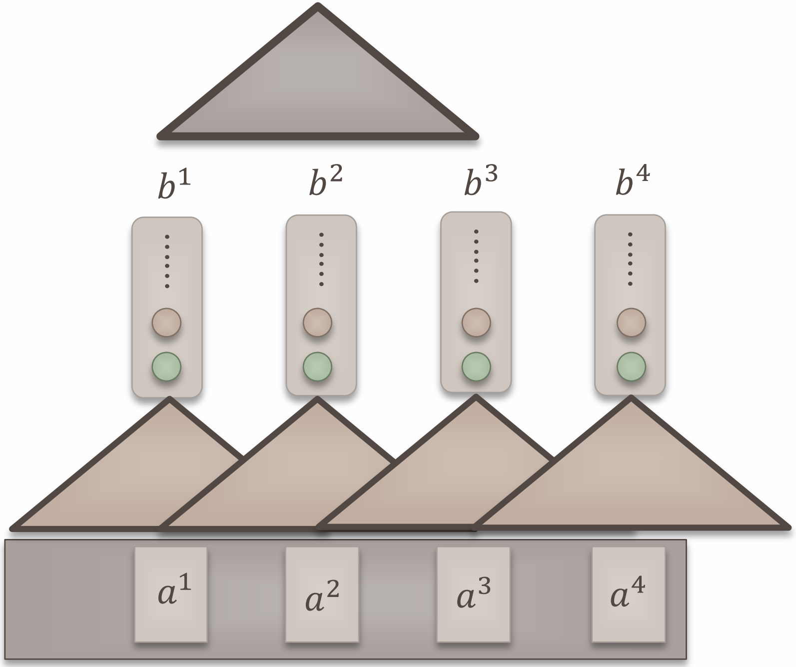

Using CNN to replace RNN

# Sequence

Filters in higher layer can consider longer sequence




Hard to parallel !

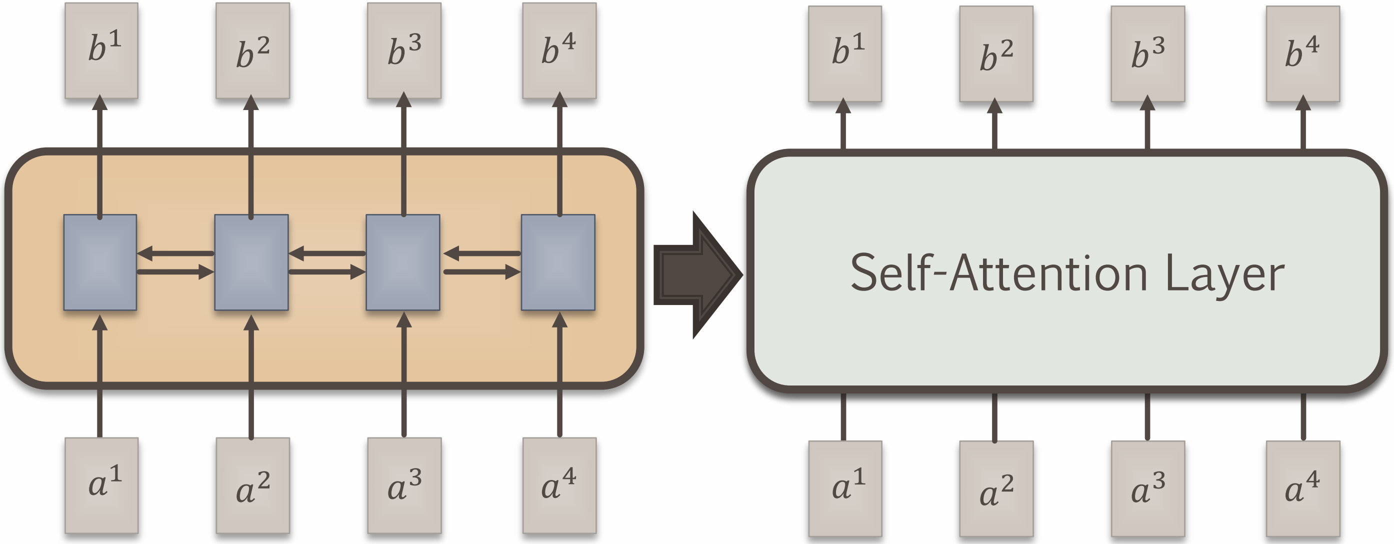



Using CNN to replace RNN
(CNN can parallel)

# Self-Attention

$b^1, b^2, b^3, b^4$ can be parallelly computed.

$b^i$ is obtained based on the whole input sequence.



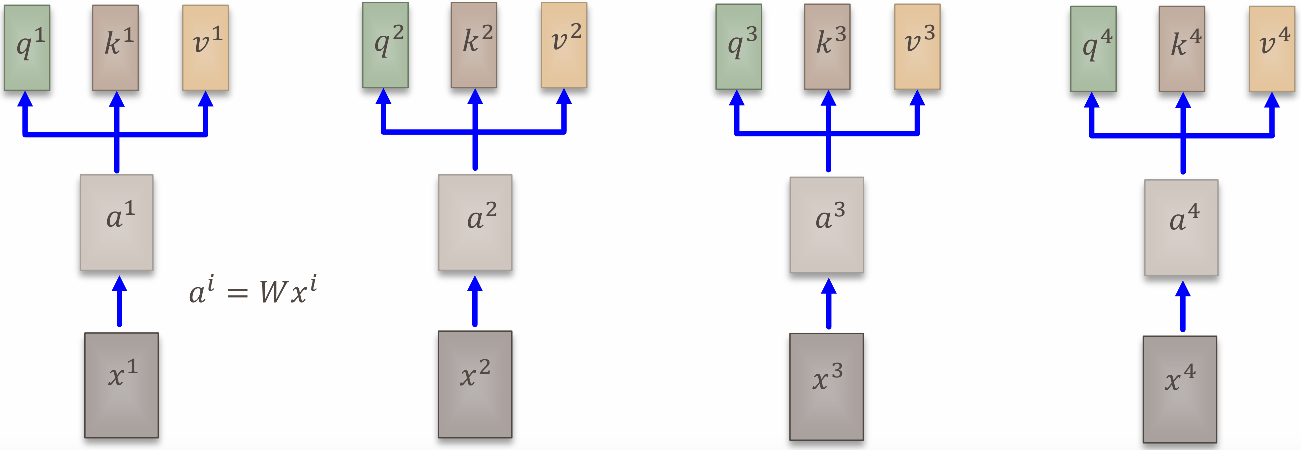

You can try to replace any thing that has been done by RNN with self-attention.

# Self-attention

$q$: query (to match others) $\qquad q^i = W^q a^i$

$k$: key (to be matched) $\qquad k^i = W^k a^i$

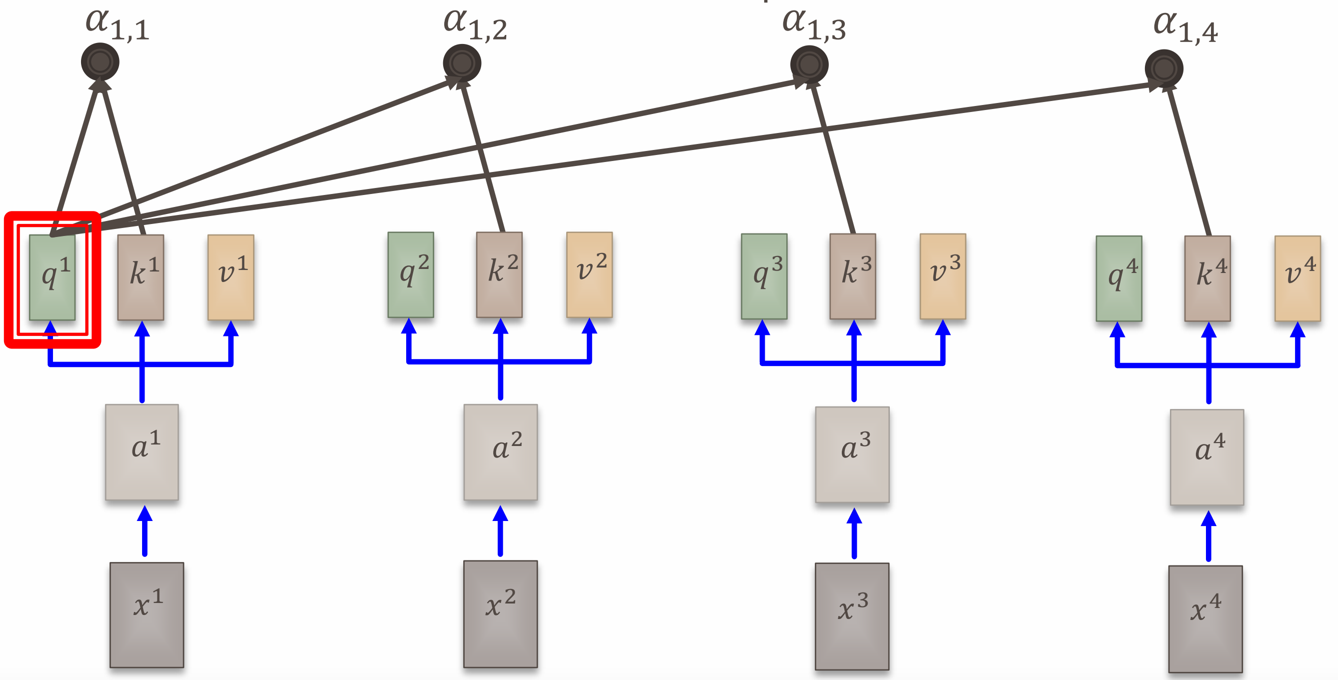
$v$: information to be extracted $\qquad v^i = W^v a^i$




https://arxiv.org/abs/1706.03762

# Self-attention

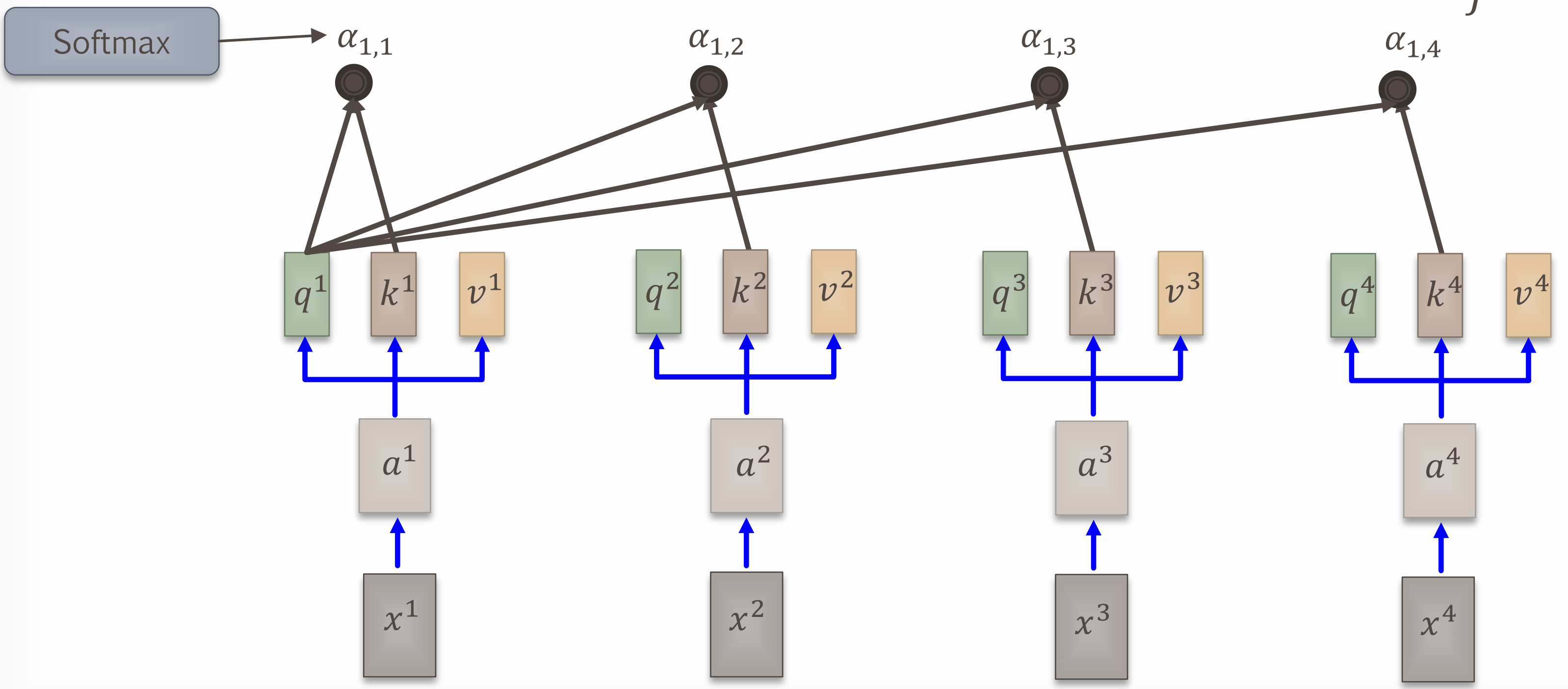
Scaled Dot-Product Attention: $\alpha_{1,i} = \underbrace{q^1 \cdot k^i}_{\text{dot product}} / \sqrt{d}$ $\quad d$ is the dim of $q$ and $k$




partial credit by CS311n

# Self-attention

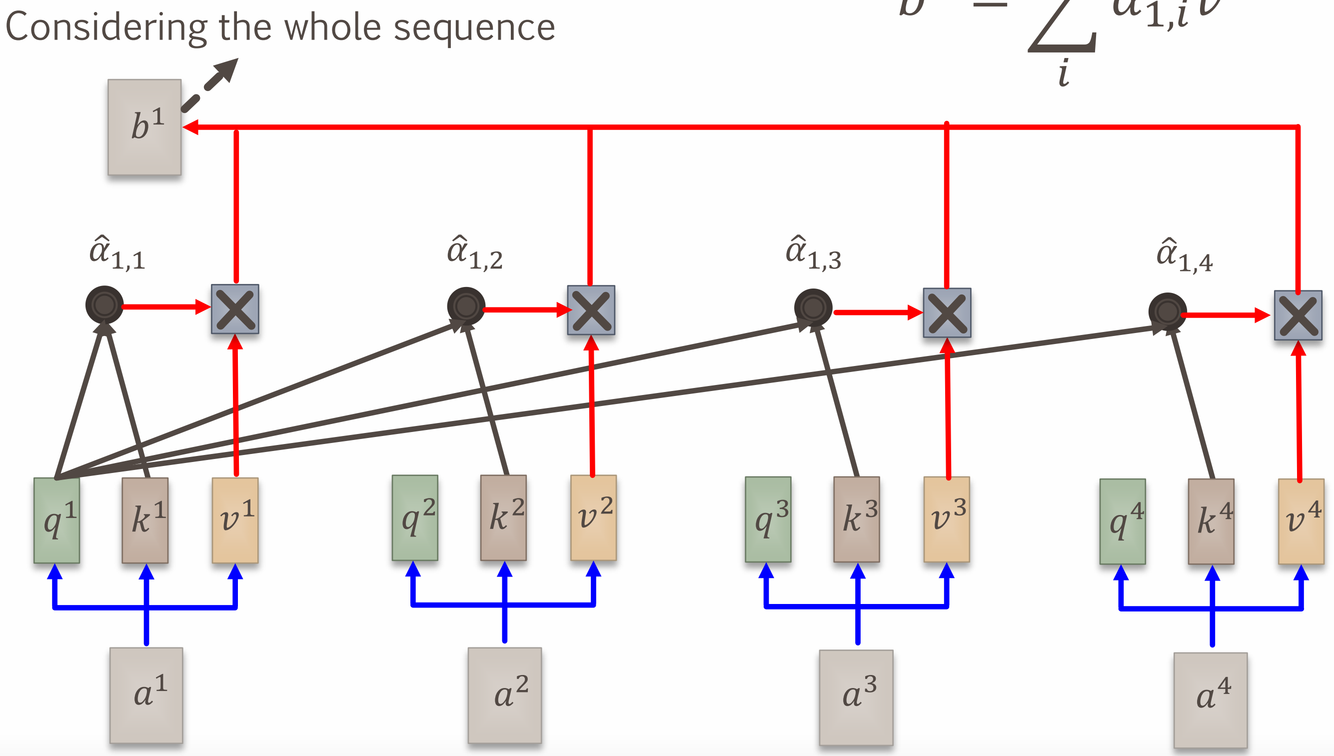
$$\hat{\alpha}_{1,i} = exp(\alpha_{1,i})/\sum_j exp(\alpha_{1,j})$$




partial credit by CS311n

# Self-attention

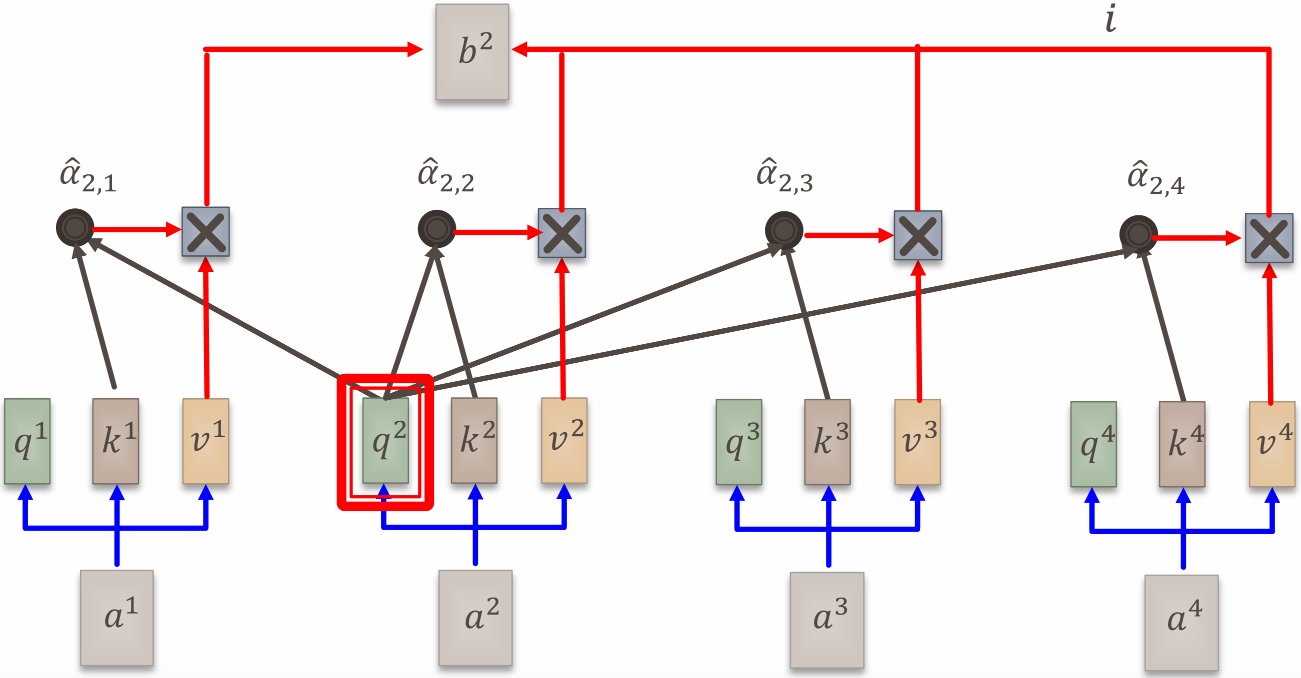
$$b^1 = \sum_i \hat{\alpha}_{1,i} v^i$$




partial credit by CS311n

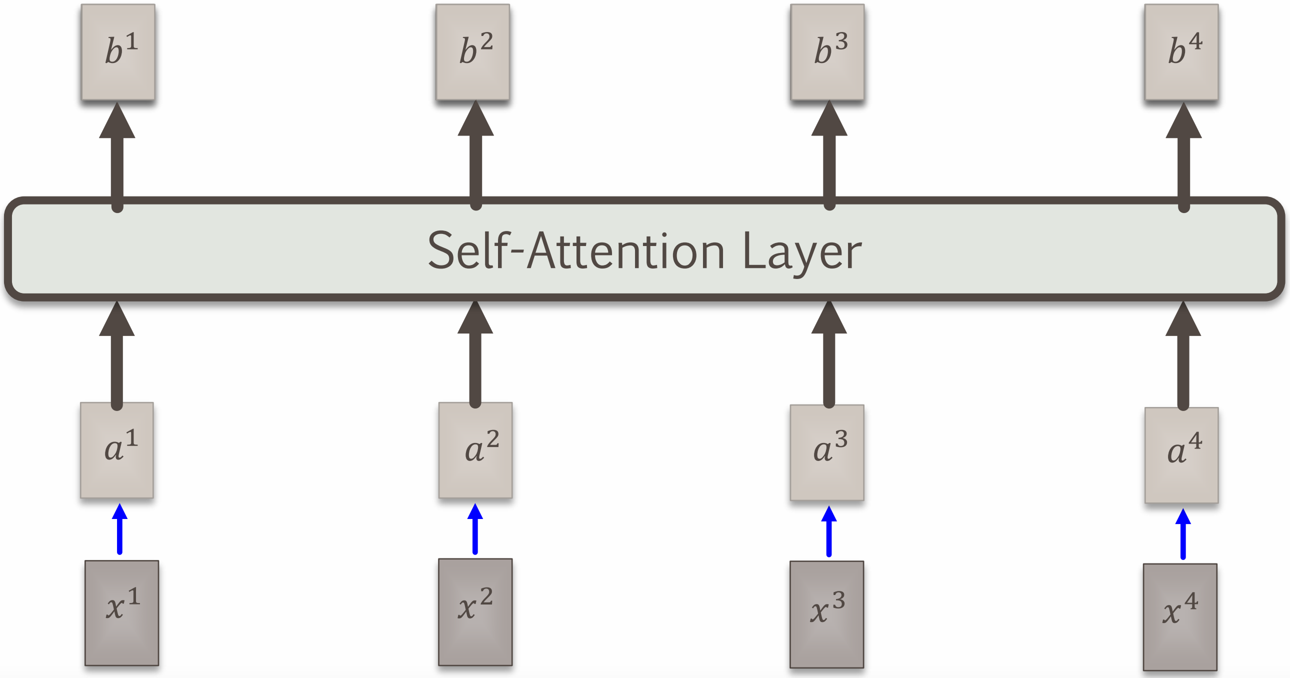
# Self-attention

$$b^2 = \sum_i \hat{\alpha}_{2,i} v^i$$



# Self-attention

$b^1, b^2, b^3, b^4$ can be parallelly computed.



# Self-attention: Matrix operation

$$q^i = W^q a^i$$

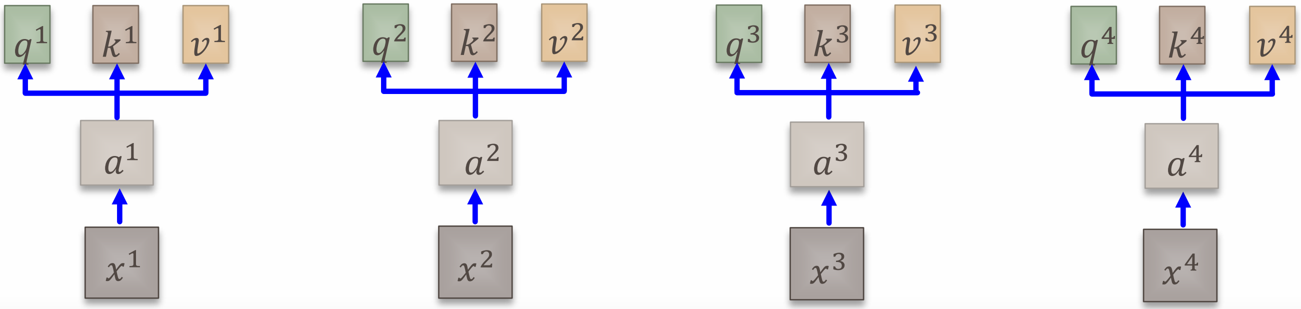
$$k^i = W^k a^i$$

$$v^i = W^v a^i$$

$$\underbrace{\begin{bmatrix} q^1 & q^2 & q^3 & q^4 \end{bmatrix}}_{Q} = W^q \underbrace{\begin{bmatrix} a^1 & a^2 & a^3 & a^4 \end{bmatrix}}_{I}$$

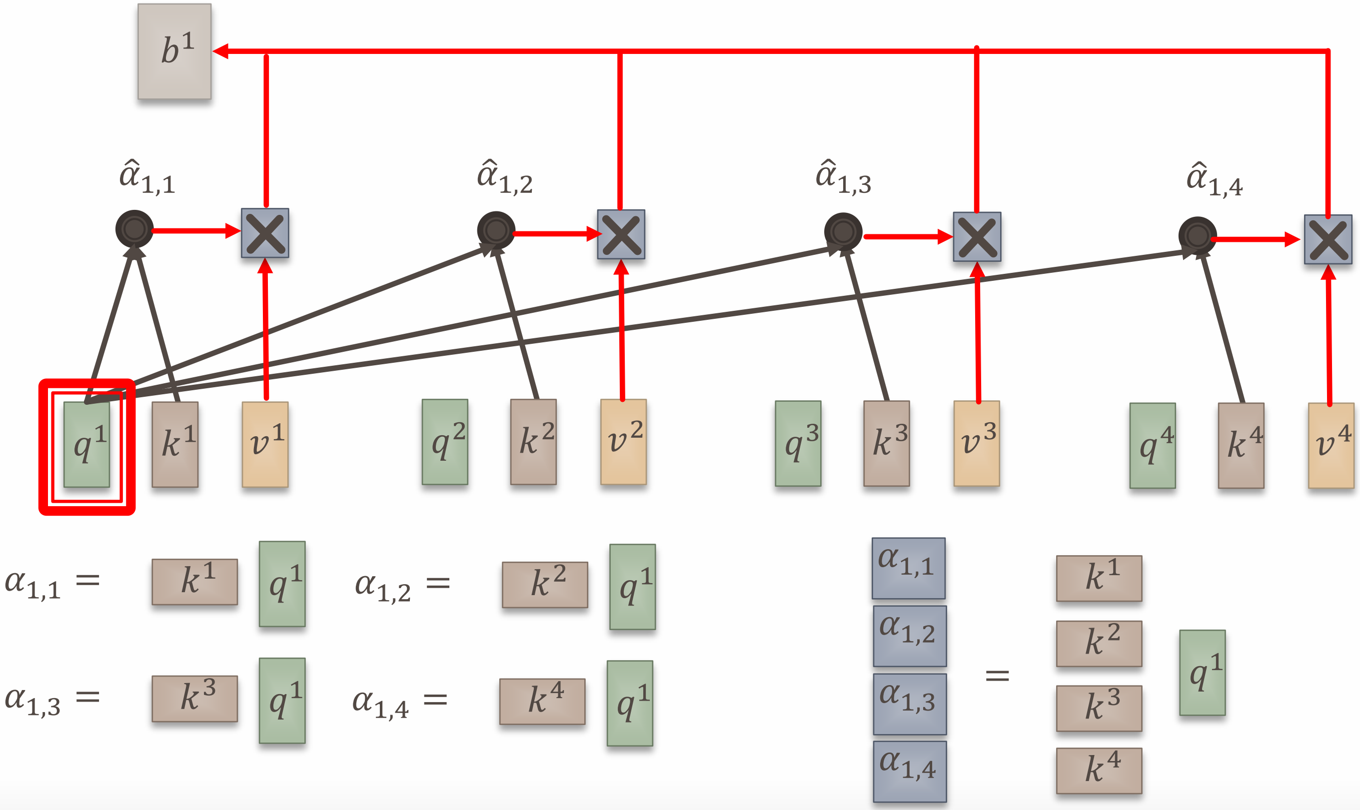
$$\underbrace{\begin{bmatrix} k^1 & k^2 & k^3 & k^4 \end{bmatrix}}_{K} = W^k \underbrace{\begin{bmatrix} a^1 & a^2 & a^3 & a^4 \end{bmatrix}}_{I}$$

$$\underbrace{\begin{bmatrix} v^1 & v^2 & v^3 & v^4 \end{bmatrix}}_{V} = W^v \underbrace{\begin{bmatrix} a^1 & a^2 & a^3 & a^4 \end{bmatrix}}_{I}$$



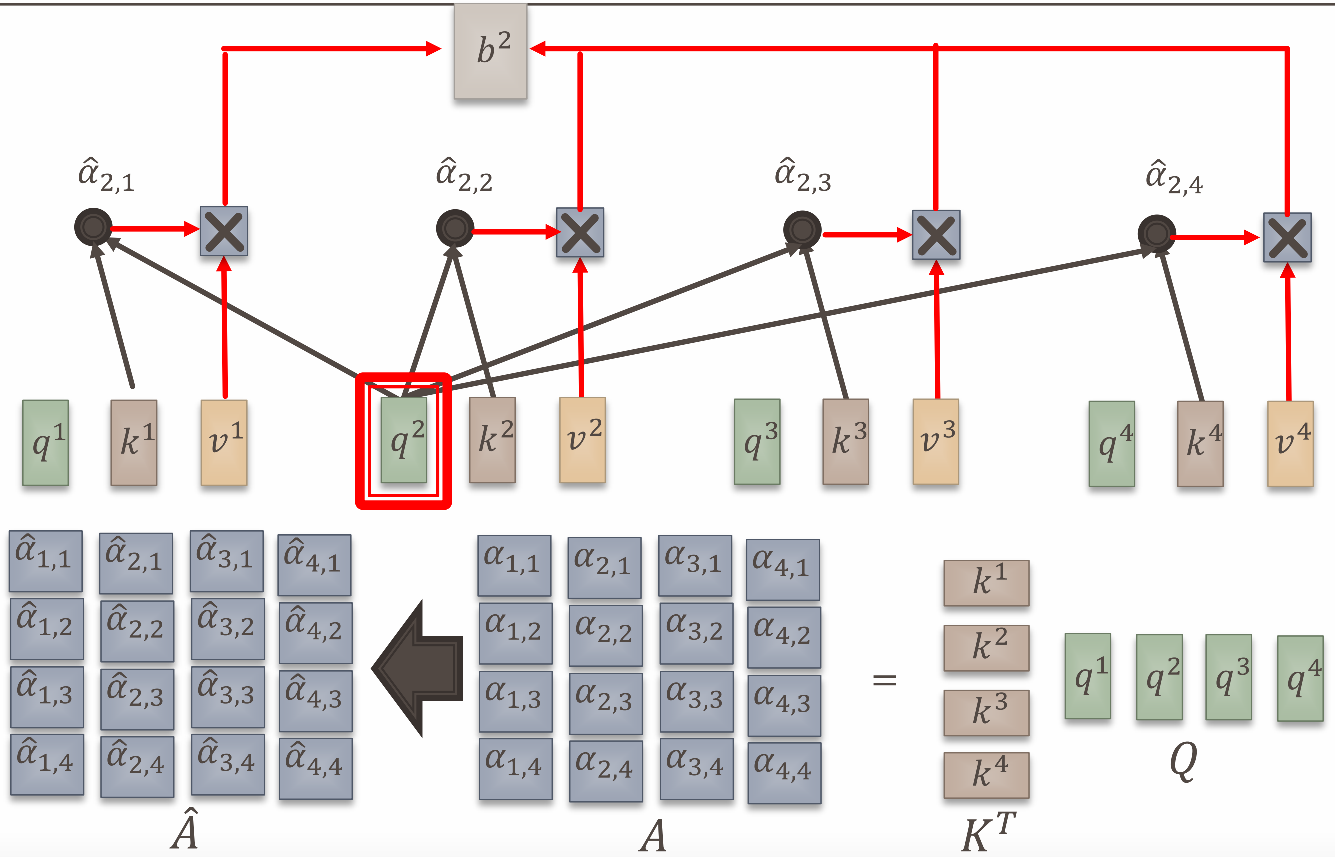
# Self-attention

(ignore $\sqrt{d}$ for simplicity)



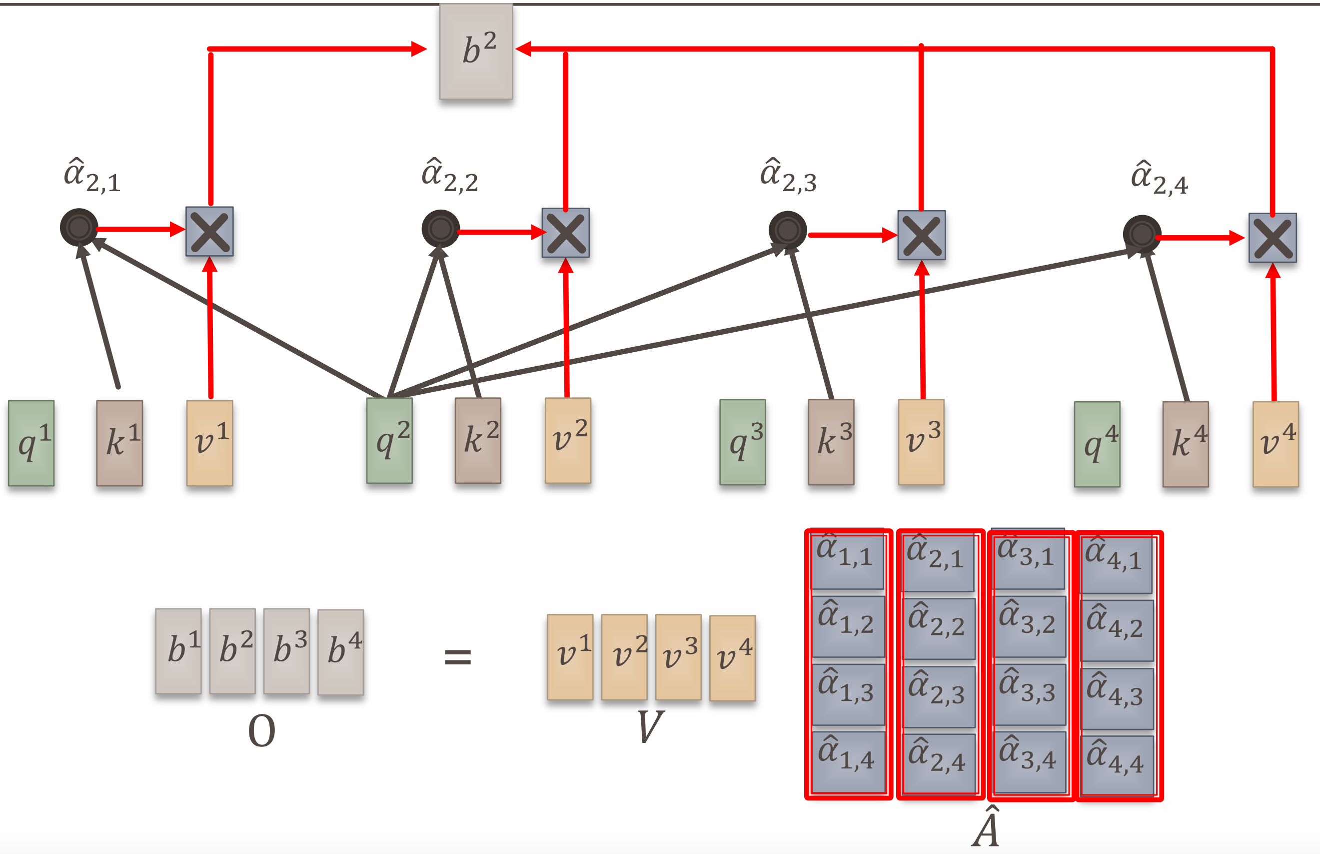
# Self-attention

$$b^2 = \sum_i \hat{\alpha}_{2,i} v^i$$

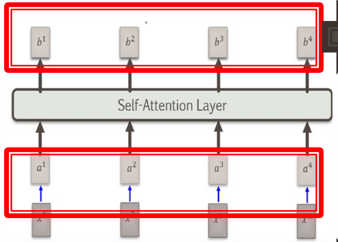


$$b^2 = \sum_i \hat{\alpha}_{2,i} v^i$$

# Self-attention



# Self-attention




O

I

$$Q = W^q I$$

$$K = W^k I$$

$$V = W^v I$$

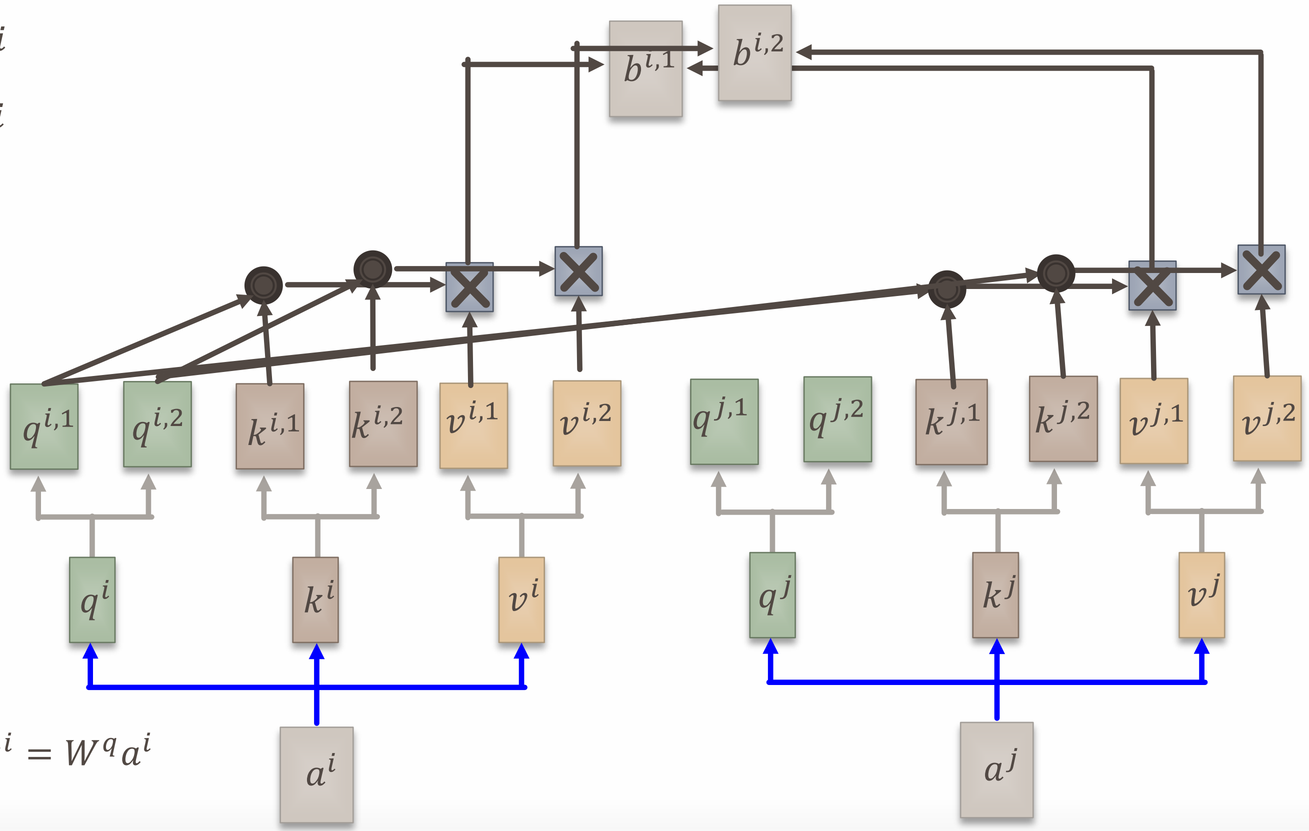
$$\hat{A} \Leftarrow A = K^T Q$$

$$O = V \hat{A}$$

# Multi-head Self-attention: 2 heads as example

$$q^{i,1} = W^{q,1} q^i$$

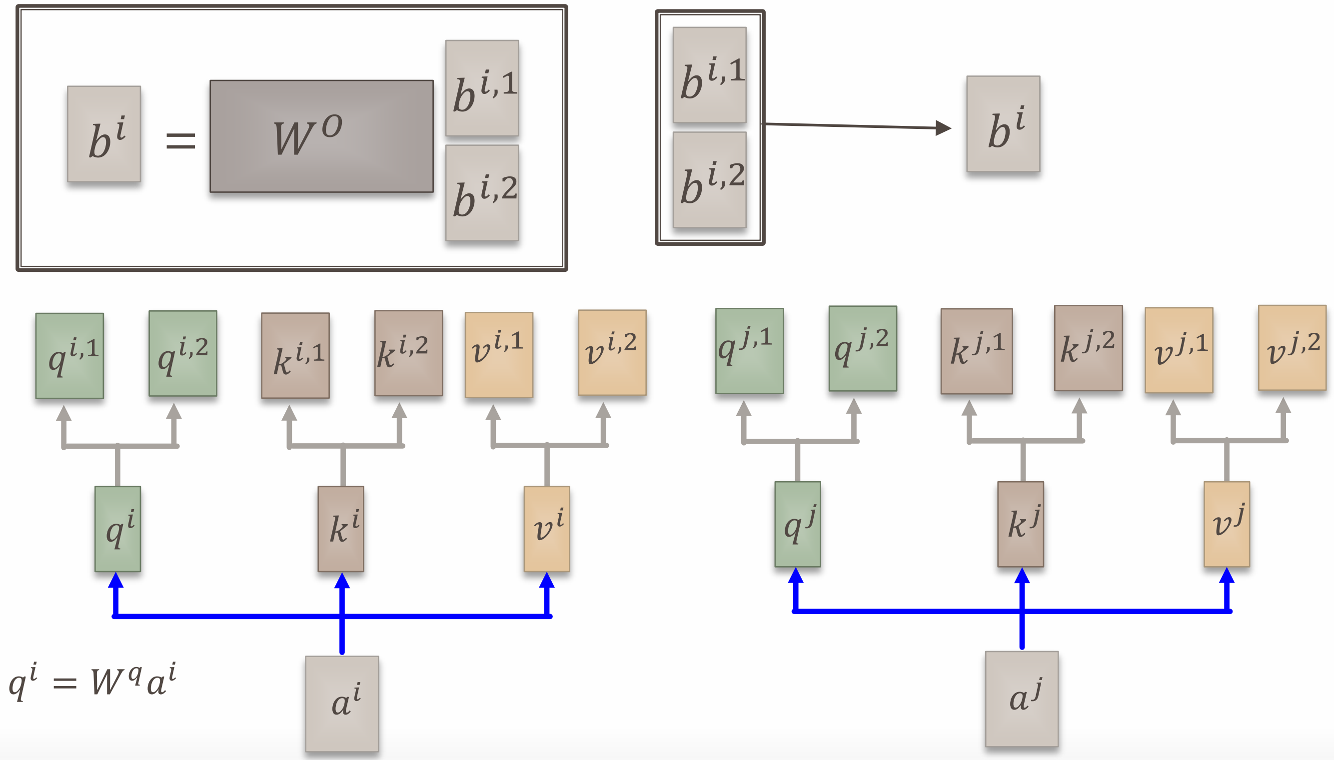
$$q^{i,2} = W^{q,2} q^i$$




$$q^i = W^q a^i$$

partial credit by CS311n

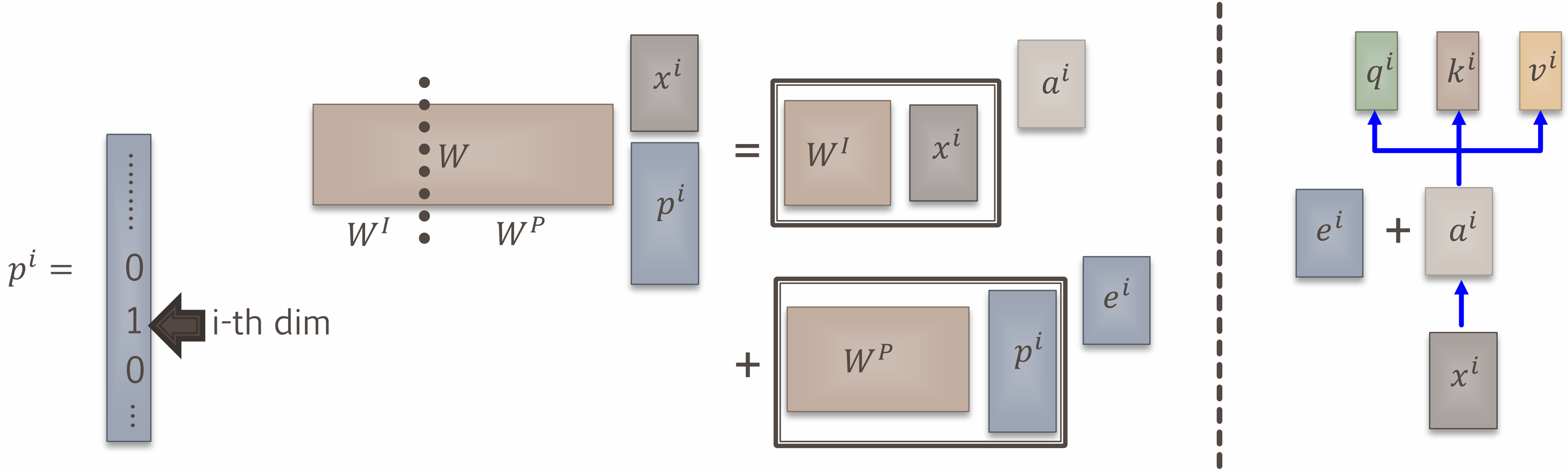
# Multi-head Self-attention: 2 heads as example




partial credit by CS311n

# Positional Encoding

- No position/order information in self-attention.
- Original paper: each position has a unique positional vector $e^i$ (not learned from data)
- In other words: each $x^i$ appends a one-hot vector $p^i$



# NEXT: EFFICIENT ATTENTION AND DOWNSTREAM TASKS

Flash Attention, SSM, Mamba