Machine Learning

CS 539

Worcester Polytechnic Institute
Department of Computer Science
Instructor: Prof. Kyumin Lee

Upcoming Schedule

- HW 3
  - https://canvas.wpi.edu/courses/58900/assignments/356655
  - Due date is July 2

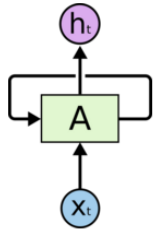
Project Website & Presentation

Item		4	3	2	1	0
Effective use of visual aids	Clear and interesting?					
Clarity of voice and pronunciation	Clear and loud? Good use of vocabulary?					
Completeness	Did they complete what they proposed?					
Demo/Evaluation						
	Total					

Recurrent Neural Network (RNN)

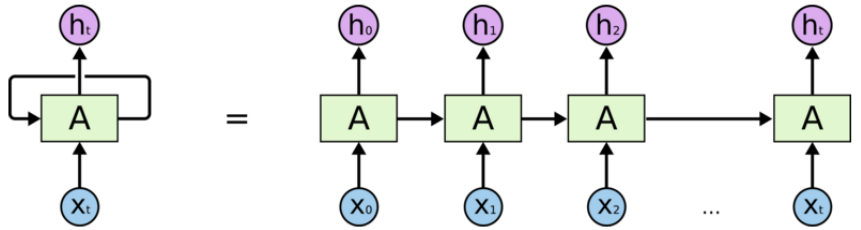
Recurrent Neural Networks

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



Recurrent Neural Networks have loops.

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

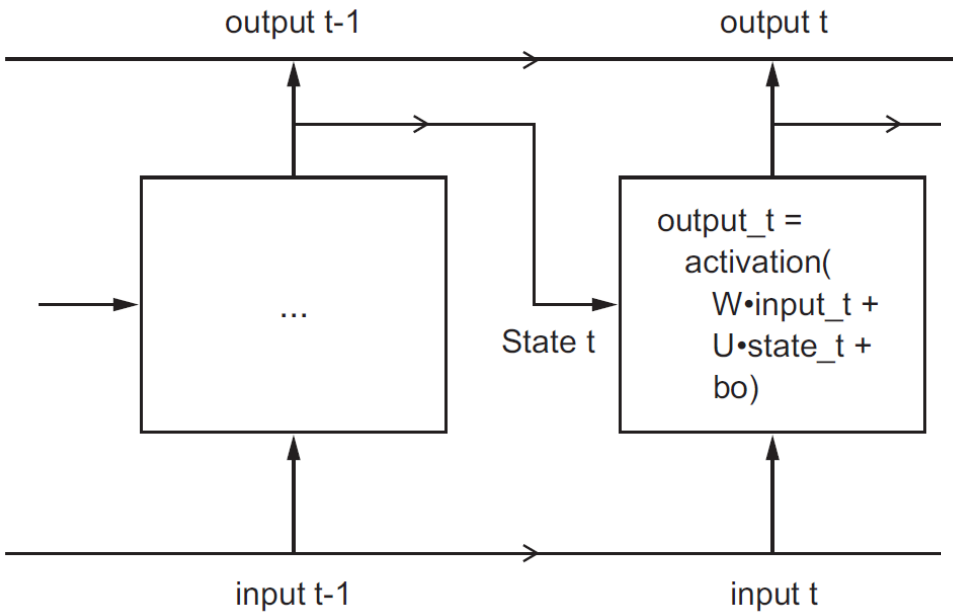


An unrolled recurrent neural network.

A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.
The diagram above shows what happens if we unroll the loop.

Recurrent Neural Networks

- Intuition of Recurrent Neural Networks
  - Human thoughts have persistence; humans don't start their thinking from scratch every second.
    - As you read this sentence, you understand each word based on your understanding of previous words.
  - One of the appeals of RNNs is the idea that they are able to connect previous information to the present task
    - E.g., using previous video frames to inform the understanding of the present frame.
    - E.g., a language model tries to predict the next word based on the previous ones.



A simple RNN, unrolled over time

Listing 6.21 Numpy implementation of a simple RNN

```
import numpy as np

timesteps = 100
input_features = 32
output_features = 64

inputs = np.random.random((timesteps, input_features))

state_t = np.zeros((output_features,))

W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features))
b = np.random.random((output_features,))

successive_outputs = []
for input_t in inputs:
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)

    successive_outputs.append(output_t)

    state_t = output_t

final_output_sequence = np.concatenate(successive_outputs, axis=0)
```

Number of timesteps in the input sequence

Dimensionality of the input feature space

Dimensionality of the output feature space

Input data: random noise for the sake of the example

Initial state: an all-zero vector

Creates random weight matrices

input_t is a vector of shape (input_features,).

Stores this output in a list

Combines the input with the current state (the previous output) to obtain the current output

Updates the state of the network for the next timestep

The final output is a 2D tensor of shape (timesteps, output_features).

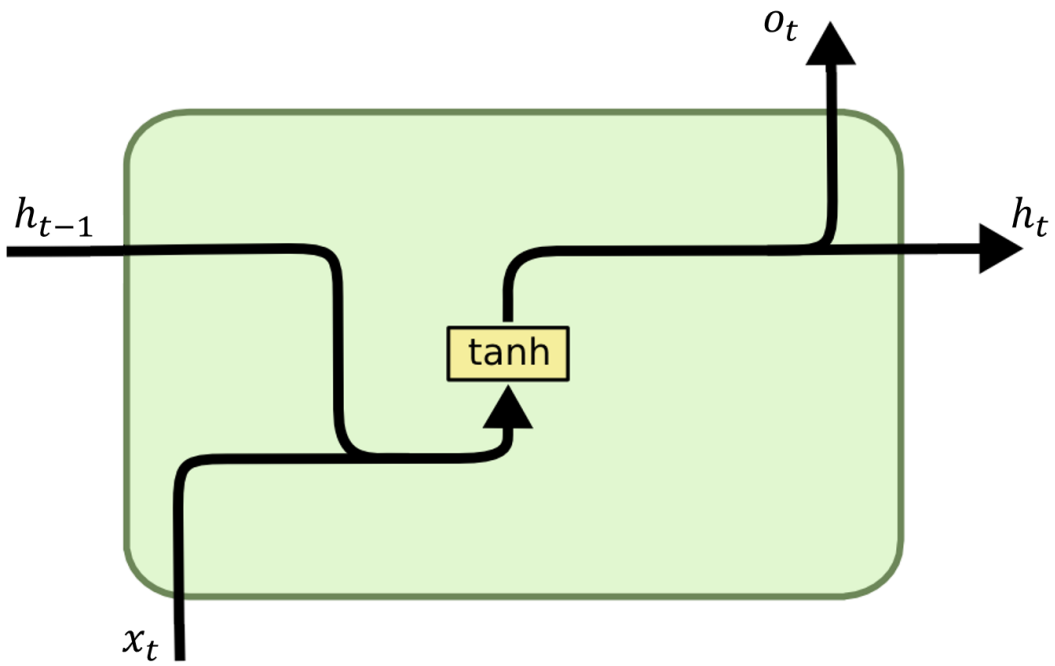
Figures are adapted from “Deep Learning with Python” writtend by F. Chollet
Refer to Section 6.2 of the book

RNN/SimpleRNN in PyTorch

CLASS torch.nn.RNN(*self, input_size, hidden_size, num_layers=1, nonlinearity='tanh', bias=True, batch_first=False, dropout=0.0, bidirectional=False, device=None, dtype=None*) [SOURCE]

Apply a multi-layer Elman RNN with $\tanh$ or ReLU non-linearity to an input sequence. For each element in the input sequence, each layer computes the following function:

$$h_t = \tanh(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh})$$




```
class RNNTagger(torch.nn.Module):

    def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
        super(RNNTagger, self).__init__()
        self.hidden_dim = hidden_dim

        self.word_embeddings = torch.nn.Embedding(vocab_size, embedding_dim)

        # The RNN takes word embeddings as inputs, and outputs hidden states
        # with dimensionality hidden_dim.
        self.rnn = torch.nn.RNN(embedding_dim, hidden_dim)

        # The linear layer that maps from hidden state space to tag space
        self.hidden2tag = torch.nn.Linear(hidden_dim, tagset_size)

    def forward(self, sentence):
        embeds = self.word_embeddings(sentence)
        rnn_out, _ = self.rnn(embeds.view(len(sentence), 1, -1))
        tag_space = self.hidden2tag(rnn_out.view(len(sentence), -1))
        tag_scores = F.log_softmax(tag_space, dim=1)
        return tag_scores
```

Pytorch reference: https://pytorch.org/docs/stable/generated/torch.nn.RNN.html

RNN/SimpleRNN in PyTorch

```python
class RNNTagger(torch.nn.Module):

    def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
        super(RNNTagger, self).__init__()
        self.hidden_dim = hidden_dim

        self.word_embeddings = torch.nn.Embedding(vocab_size, embedding_dim)

        # The RNN takes word embeddings as inputs, and outputs hidden states
        # with dimensionality hidden_dim.
        self.rnn = torch.nn.RNN(embedding_dim, hidden_dim)

        # The linear layer that maps from hidden state space to tag space
        self.hidden2tag = torch.nn.Linear(hidden_dim, tagset_size)

    def forward(self, sentence):
        embeds = self.word_embeddings(sentence)
        rnn_out, _ = self.rnn(embeds.view(len(sentence), 1, -1))
        tag_space = self.hidden2tag(rnn_out.view(len(sentence), -1))
        tag_scores = F.log_softmax(tag_space, dim=1)
        return tag_scores
```

vocab_size: # of words/tokens in the vocabulary/dictionary

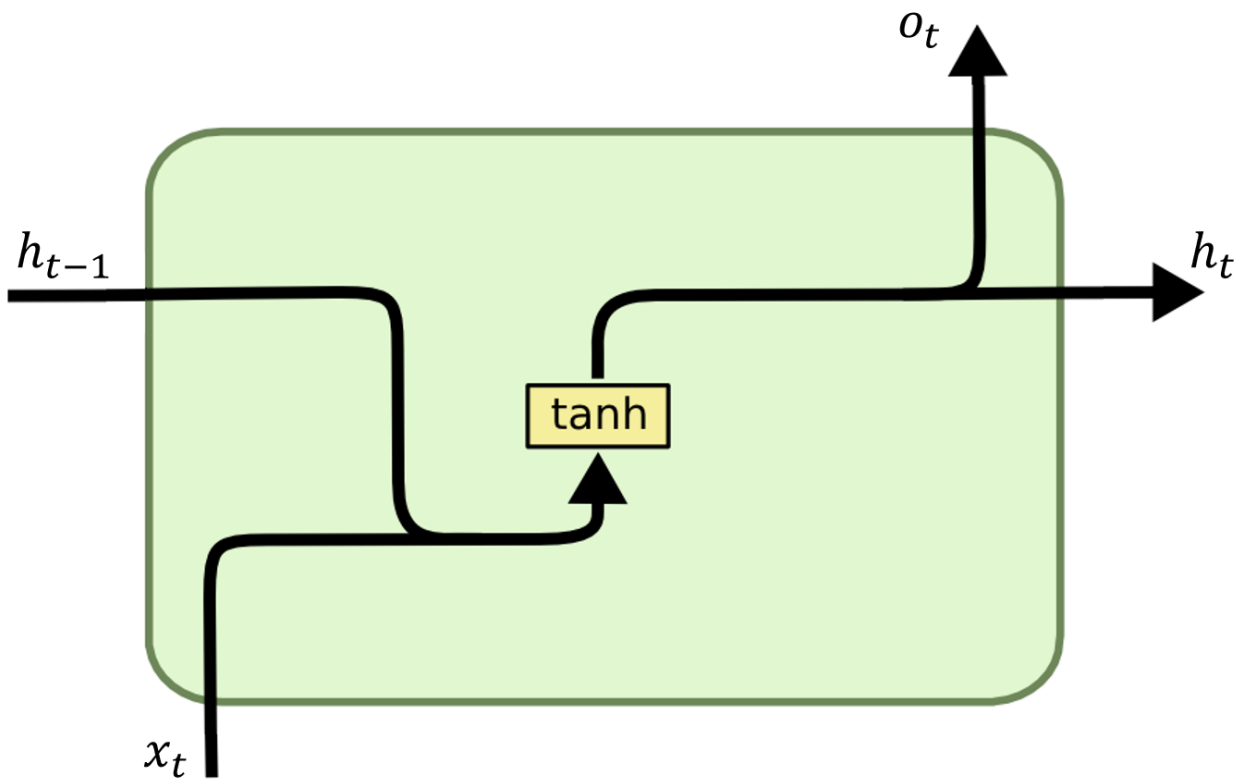
embedding_dim: embedding size of each word/token

torch.nn.Embedding:
A simple lookup table that stores embeddings of a fixed dictionary and size.

This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

Turns positive integers (indexes) into dense vectors of fixed size.
e.g. [4, 20] -> [[0.25, 0.1], [0.6, -0.2]]

RNN/SimpleRNN in Keras




```
from keras.layers import Dense

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

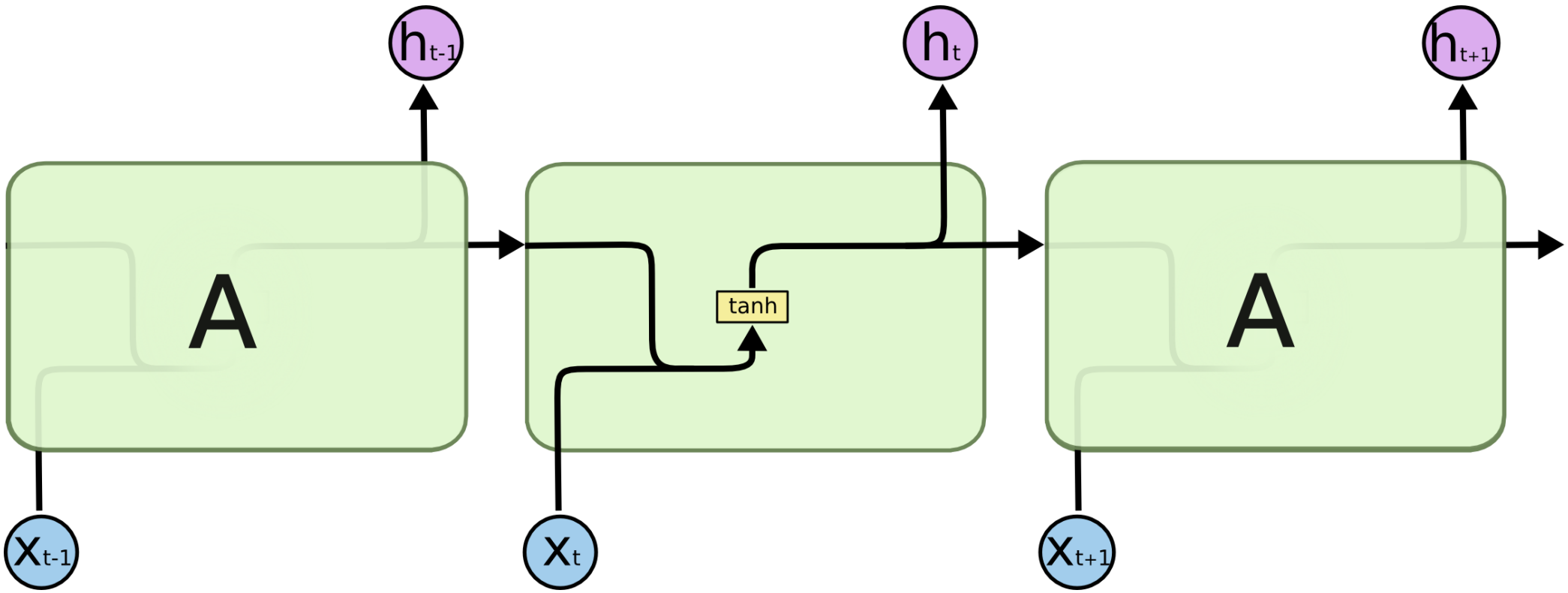
max_features = 10000 in this example

number of words to consider as featur

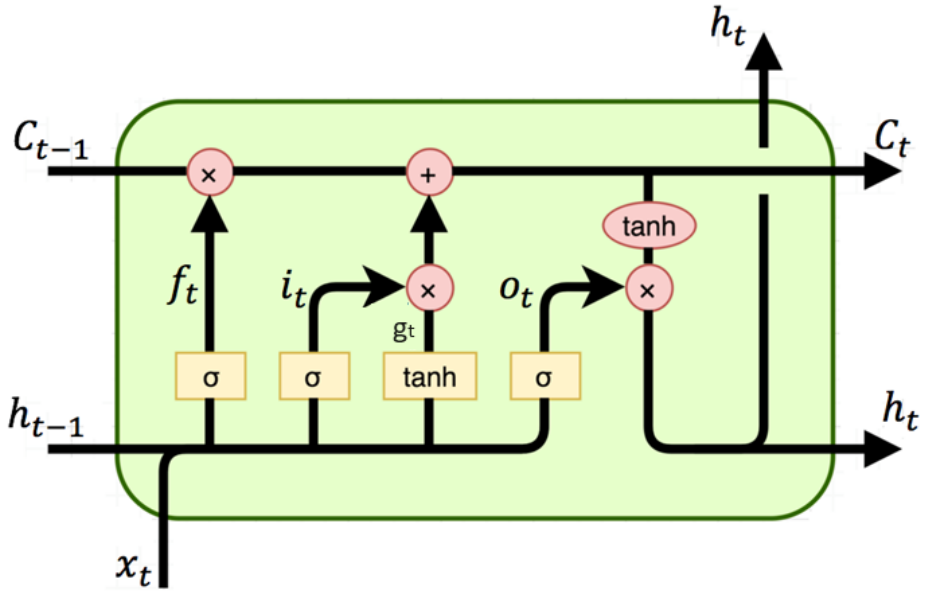
←dimensionality of the output space

```
Model: "sequential_11"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_11 (Embedding)     (None, None, 32)          320000
_________________________________________________________________
simple_rnn_12 (SimpleRNN)    (None, 32)                2080
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 33
=================================================================
Total params: 322,113
Trainable params: 322,113
Non-trainable params: 0
_________________________________________________________________
```

RNN/SimpleRNN



Long-Short Term Memory (LSTM) in PyTorch

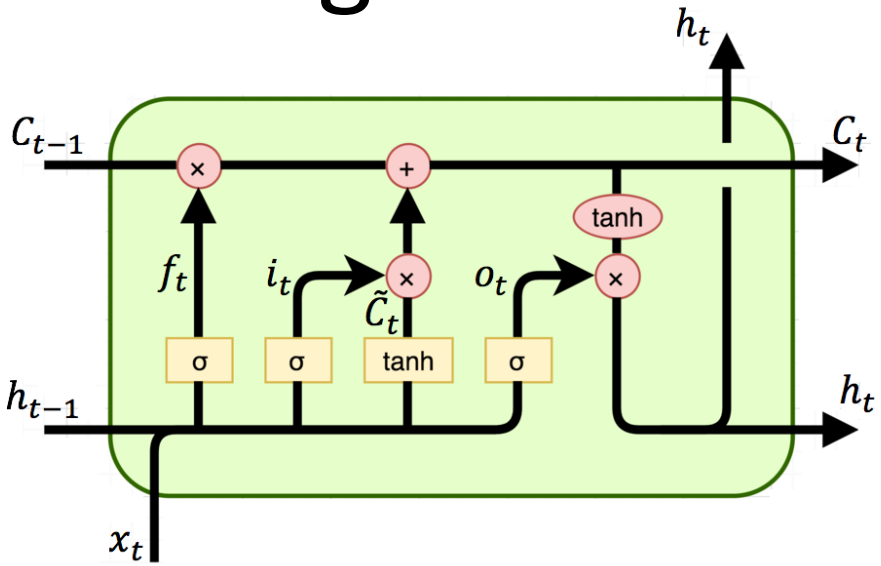


$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi})$$
$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf})$$
$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg})$$
$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho})$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$
$$h_t = o_t \odot \tanh(c_t)$$

```python
class LSTMTagger(torch.nn.Module):

    def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
        super(LSTMTagger, self).__init__()
        self.hidden_dim = hidden_dim

        self.word_embeddings = torch.nn.Embedding(vocab_size, embedding_dim)

        # The LSTM takes word embeddings as inputs, and outputs hidden states
        # with dimensionality hidden_dim.
        self.lstm = torch.nn.LSTM(embedding_dim, hidden_dim)

        # The linear layer that maps from hidden state space to tag space
        self.hidden2tag = torch.nn.Linear(hidden_dim, tagset_size)

    def forward(self, sentence):
        embeds = self.word_embeddings(sentence)
        lstm_out, _ = self.lstm(embeds.view(len(sentence), 1, -1))
        tag_space = self.hidden2tag(lstm_out.view(len(sentence), -1))
        tag_scores = F.log_softmax(tag_space, dim=1)
        return tag_scores
```

Pytorch reference: https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html
Example: https://pytorch.org/tutorials/beginner/nlp/sequence_models_tutorial.html

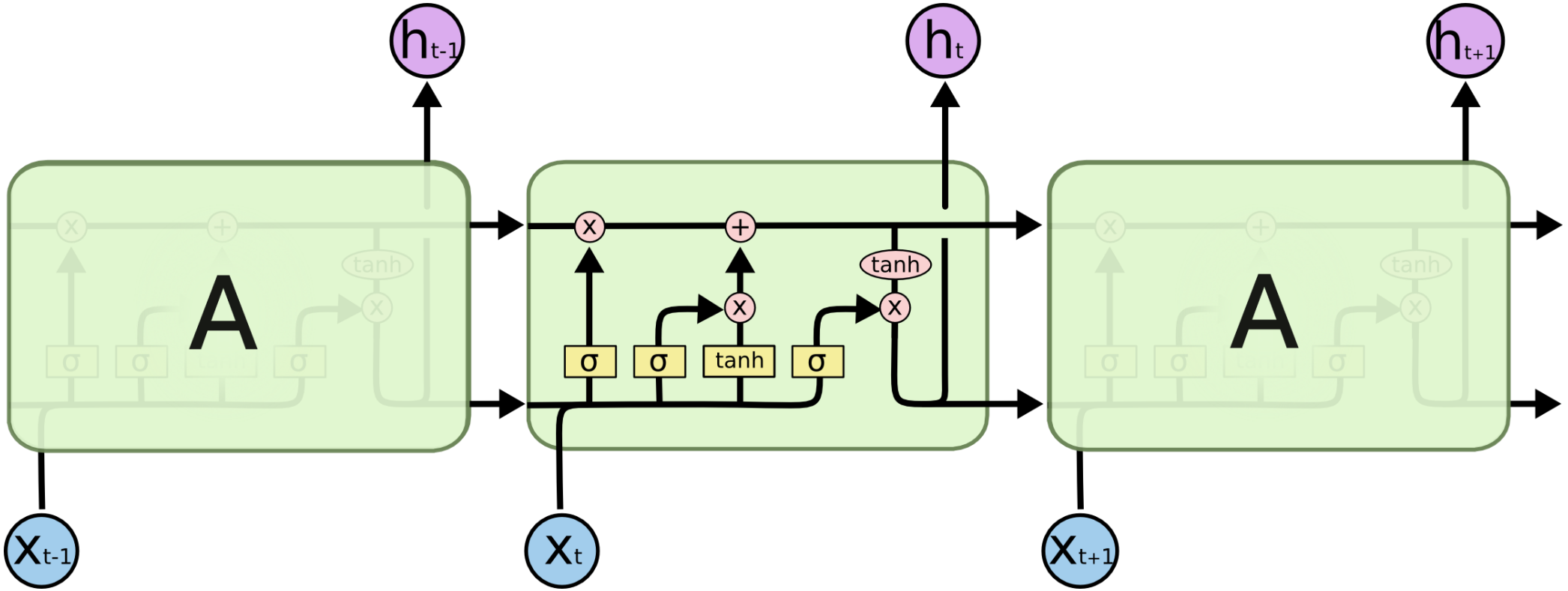
Long-Short Term Memory (LSTM) in Keras




```
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

```
Model: "sequential_13"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_13 (Embedding)     (None, None, 32)          320000
_________________________________________________________________
lstm_4 (LSTM)                (None, 32)                8320
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 33
=================================================================
Total params: 328,353
Trainable params: 328,353
Non-trainable params: 0
_________________________________________________________________
```

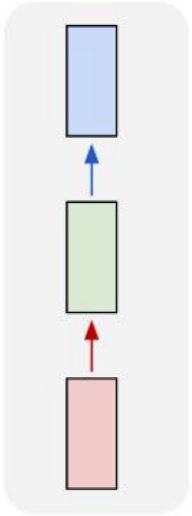
Long-Short Term Memory (LSTM)



Recurrent Neural Network

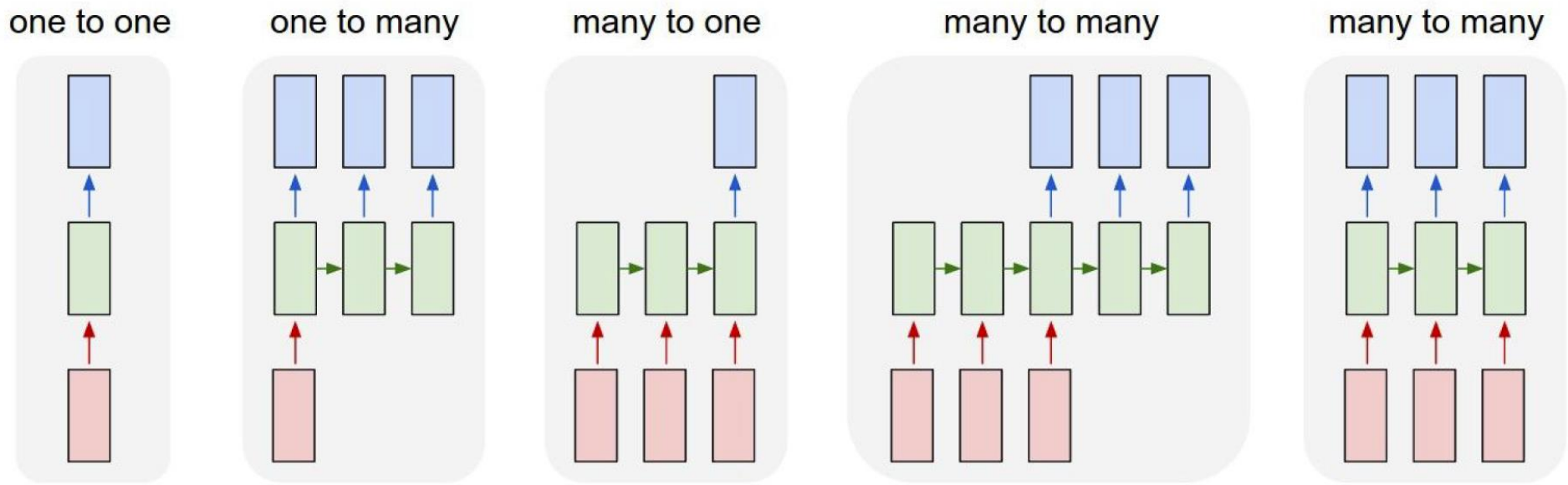
“Vanilla” Neural Network

one to one



Vanilla Neural Networks

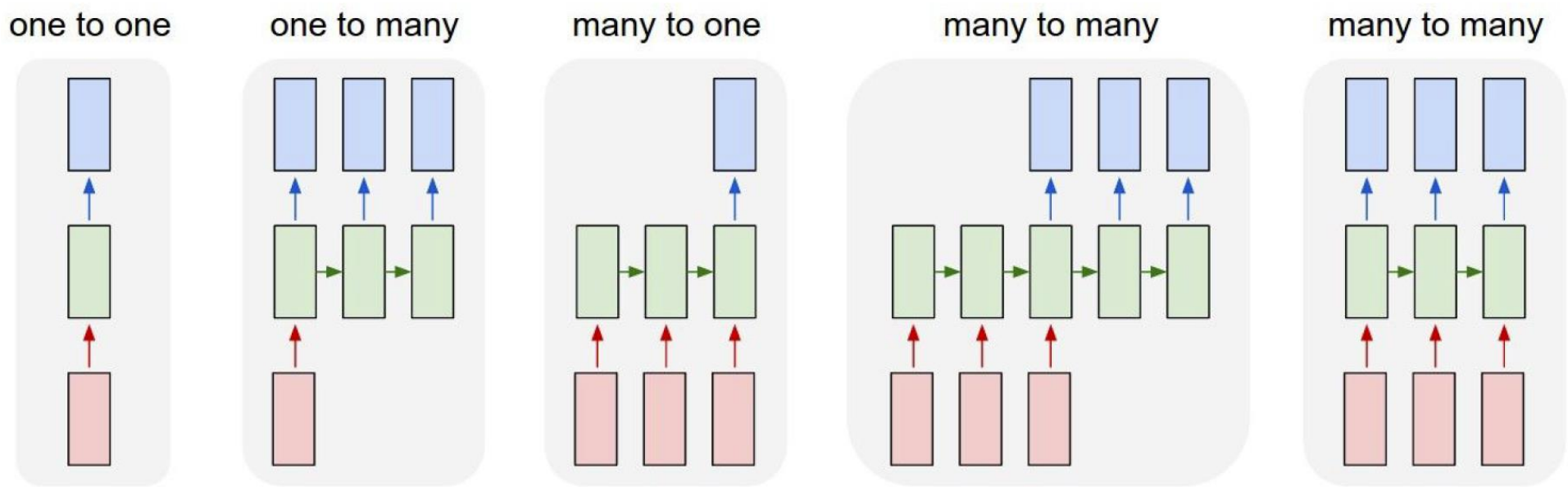
Recurrent Neural Networks: Process Sequences



e.g. **Image Captioning**
image -> sequence of words

➔ A person riding a motorbike on dirt road

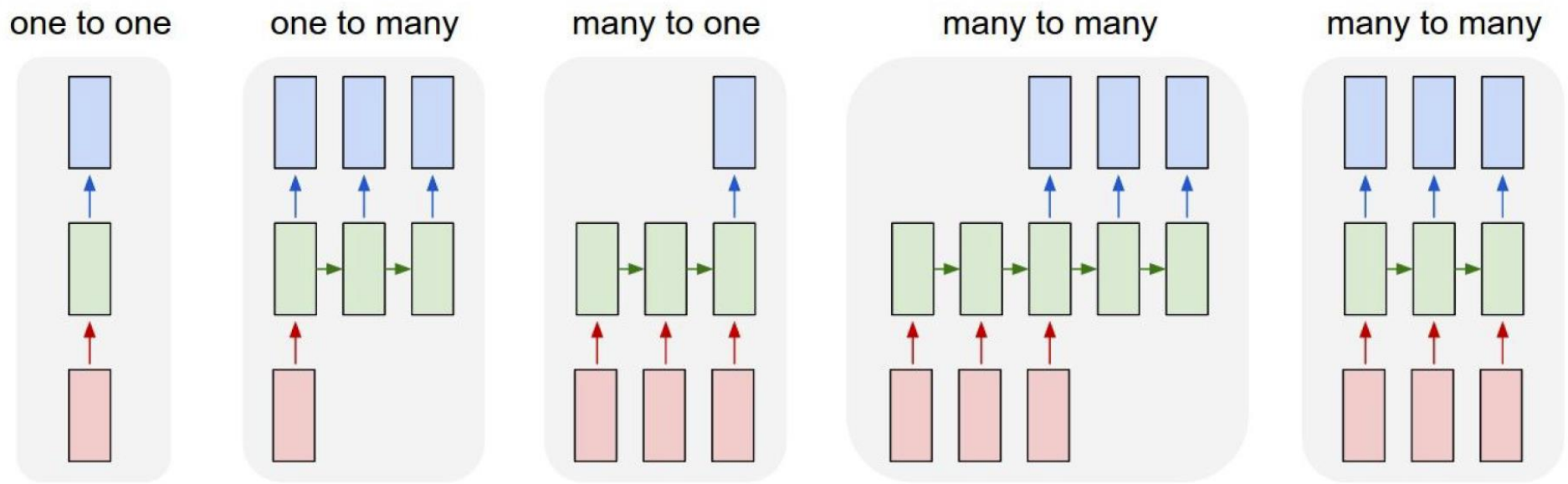
Recurrent Neural Networks: Process Sequences



e.g. **Sentiment Classification**
sequence of words -> sentiment

Awesome lecture! ➔ Positive

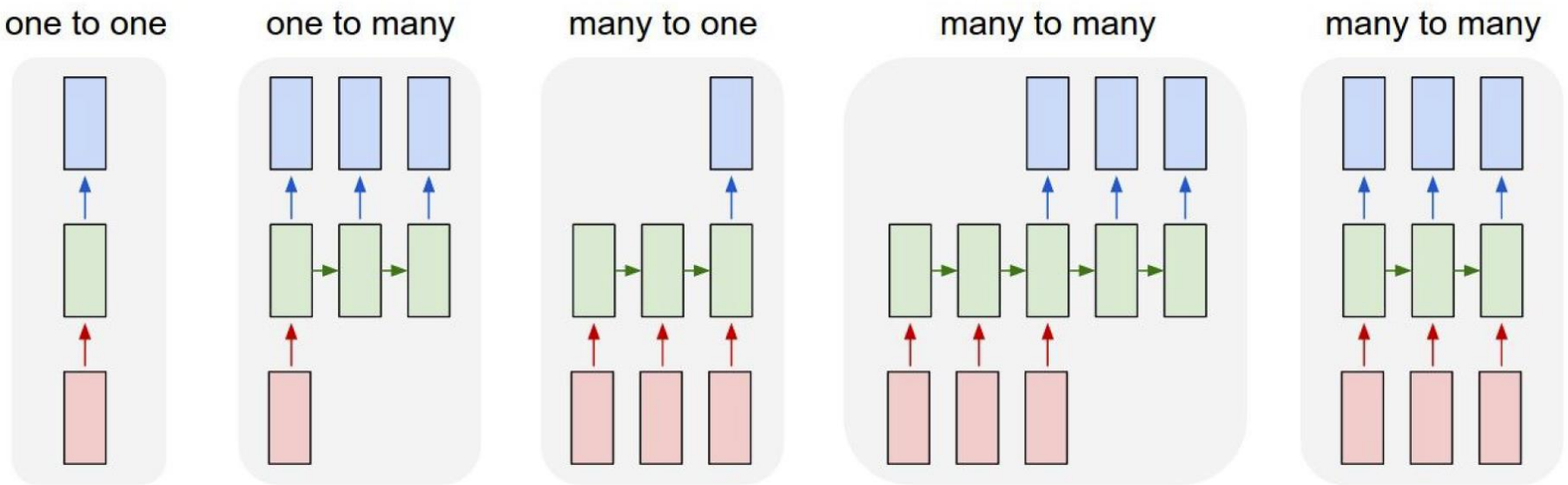
Recurrent Neural Networks: Process Sequences



e.g. **Machine Translation**
seq of words -> seq of words

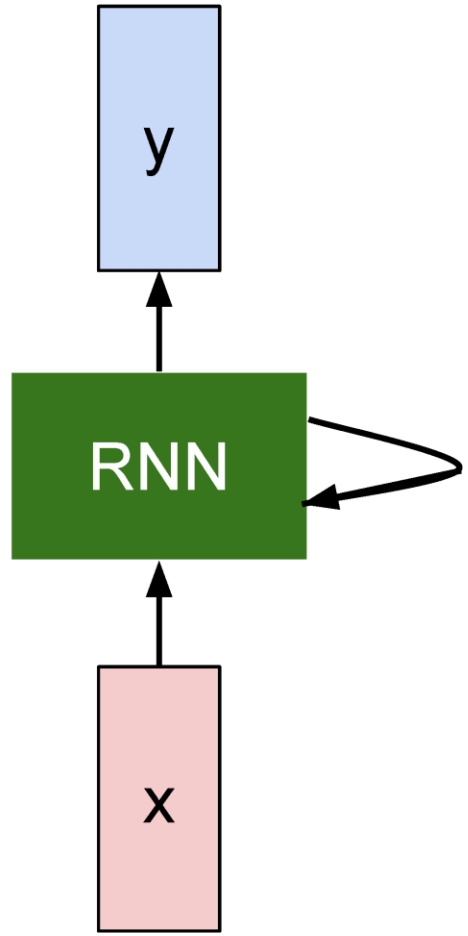
happy thanksgiving ➔행복한 추수 감사절

Recurrent Neural Networks: Process Sequences

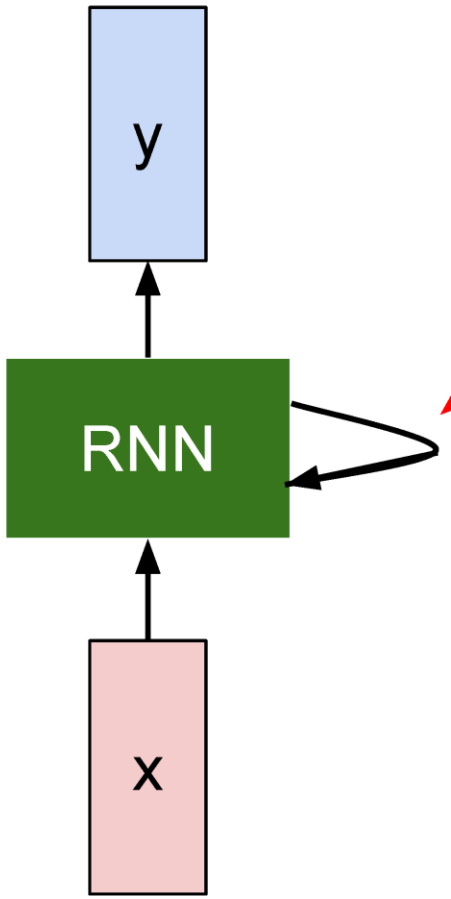


e.g. **Video classification on frame level**

Recurrent Neural Network

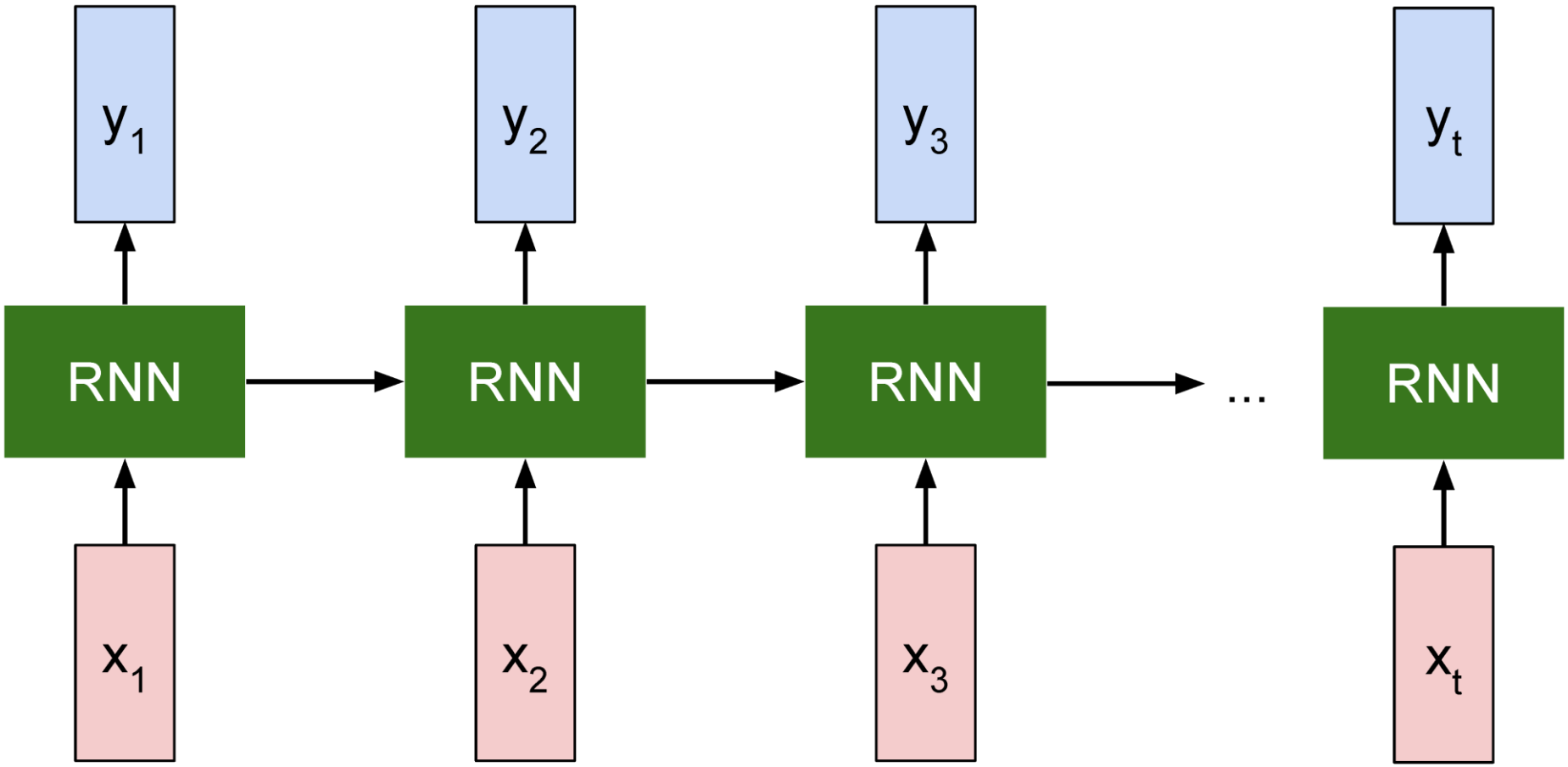


Recurrent Neural Network



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

Unrolled RNN



RNN hidden state update

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

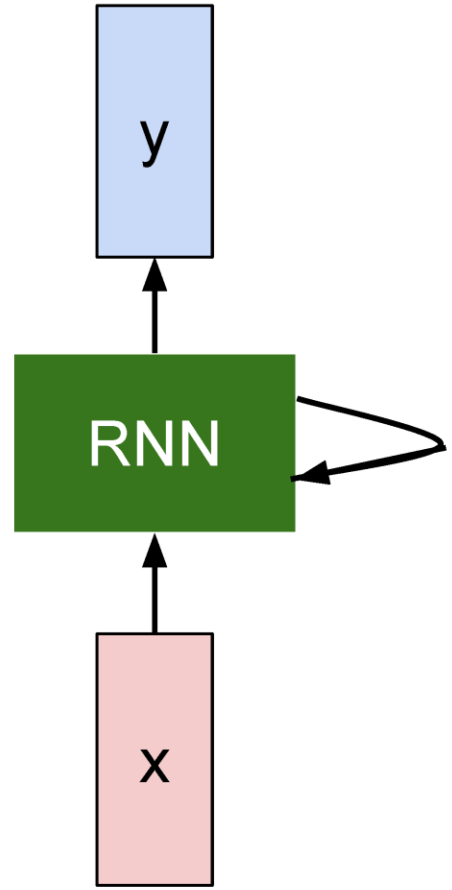
$$h_t = f_W(h_{t-1}, x_t)$$

new state

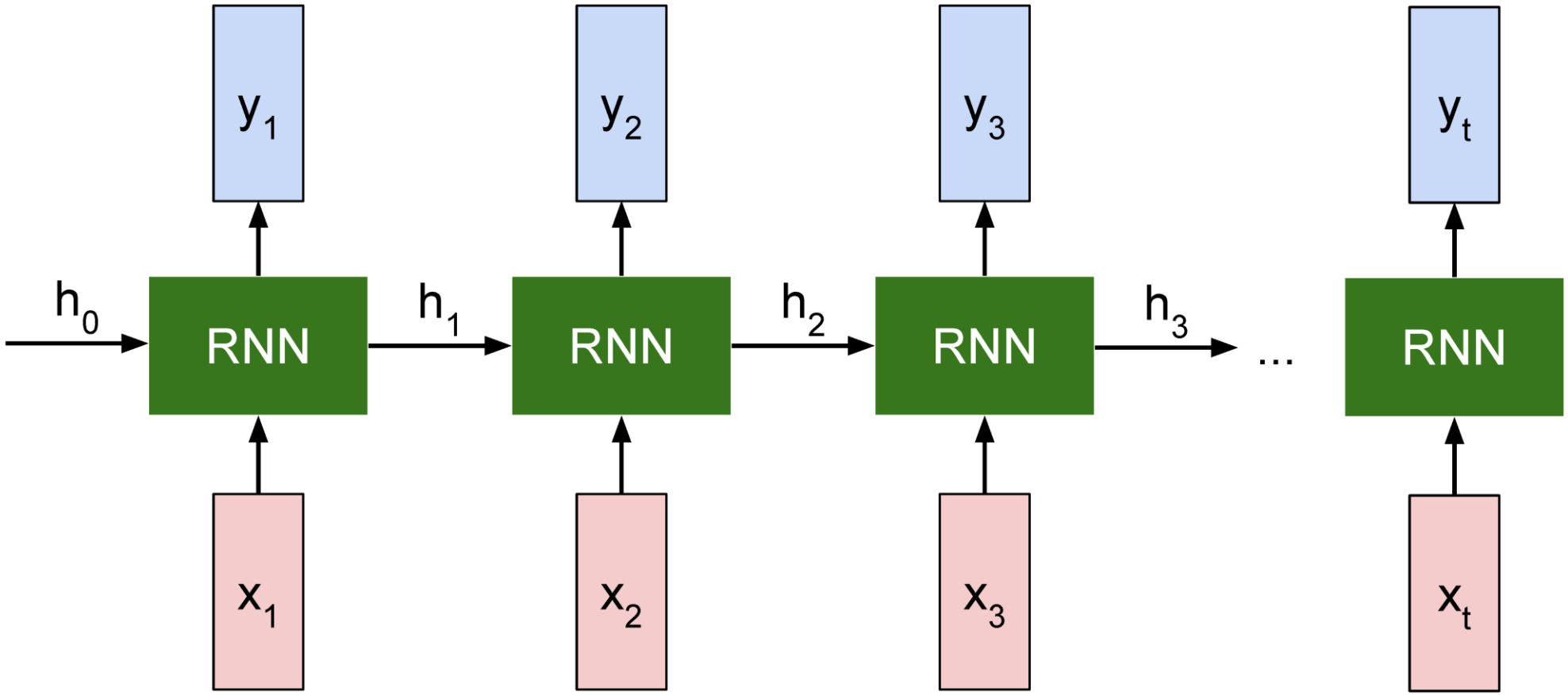
some function with parameters W

old state

input vector at some time step



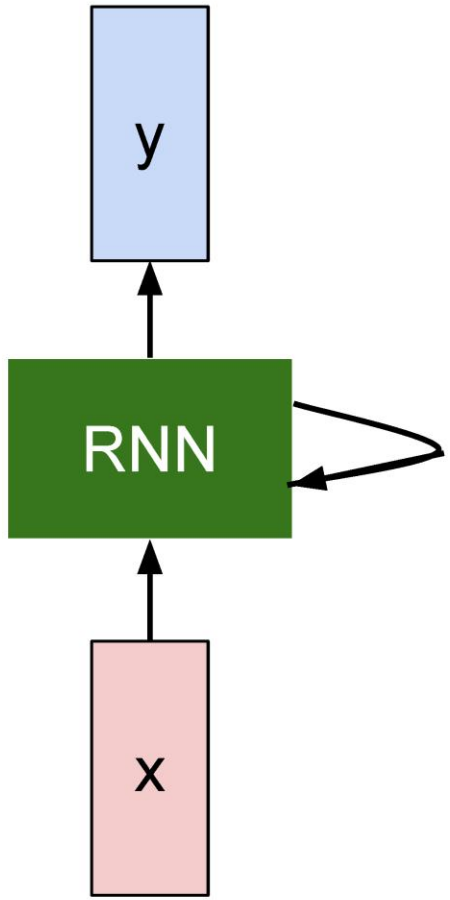
Recurrent Neural Network



Recurrent Neural Network

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

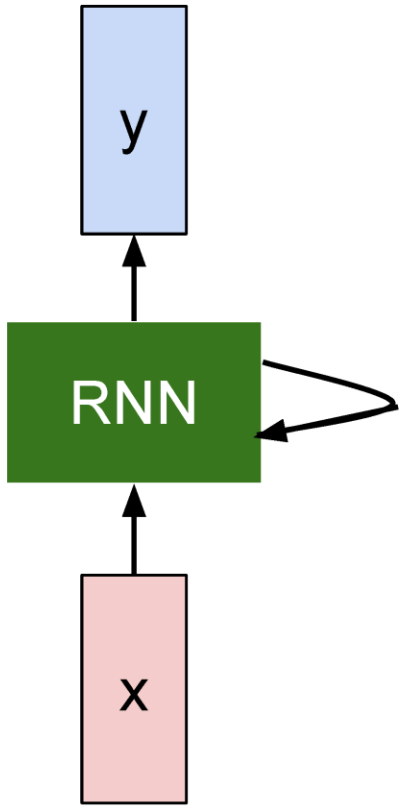
$$h_t = f_W(h_{t-1}, x_t)$$



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

(Vanilla) Recurrent Neural Network

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



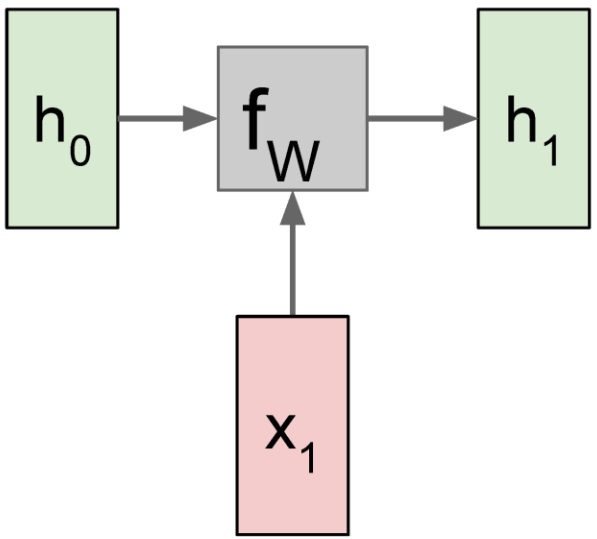
$$h_t = f_W(h_{t-1}, x_t)$$

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

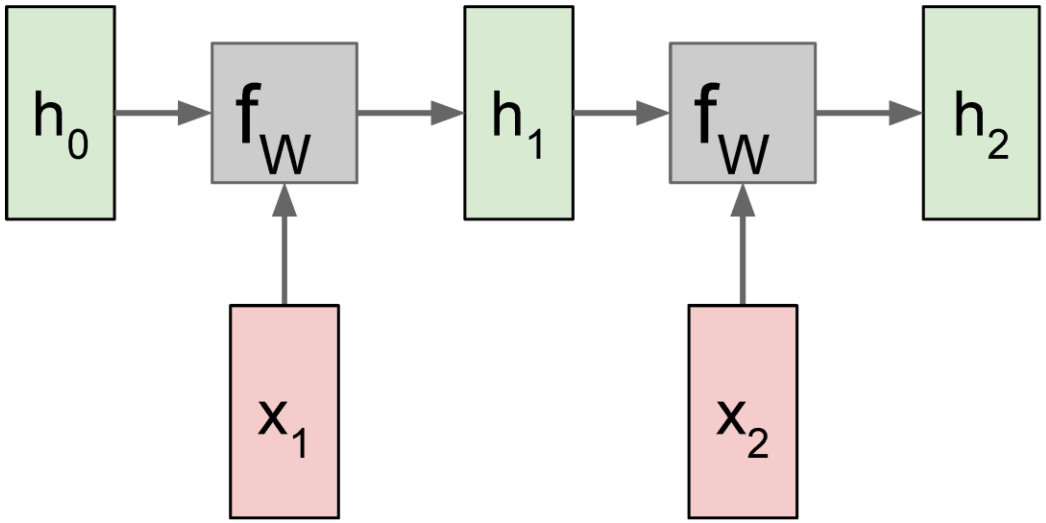
$$y_t = W_{hy}h_t$$

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

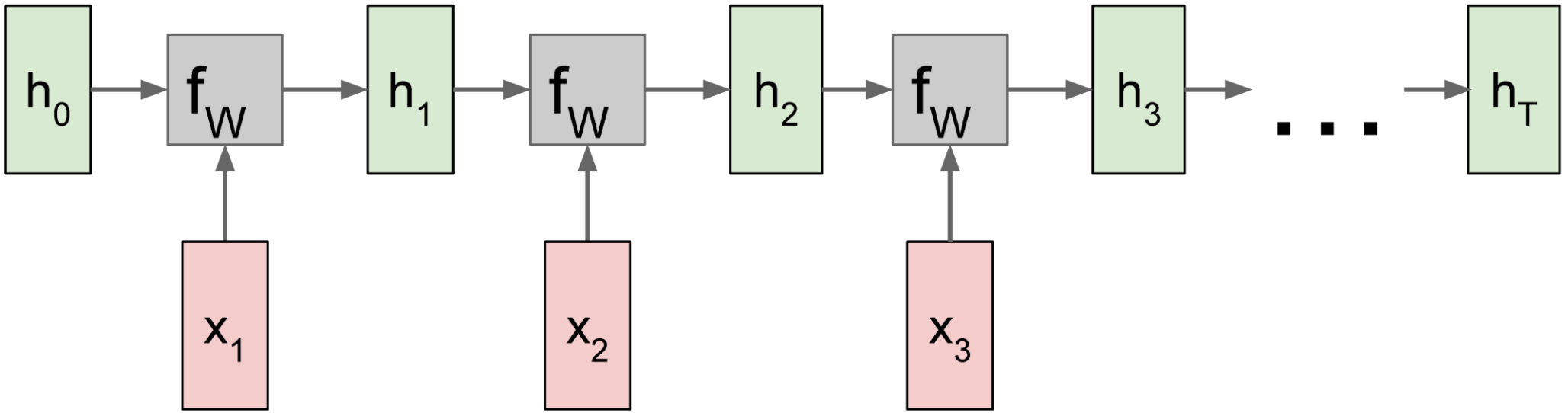
RNN: Computational Graph



RNN: Computational Graph

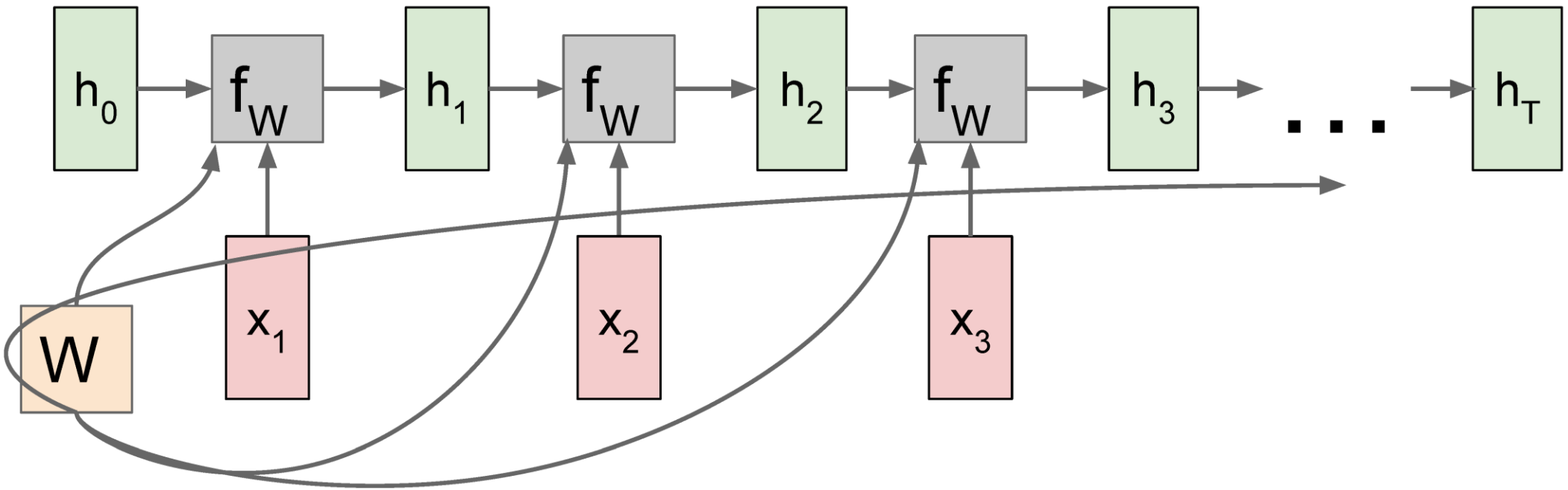


RNN: Computational Graph

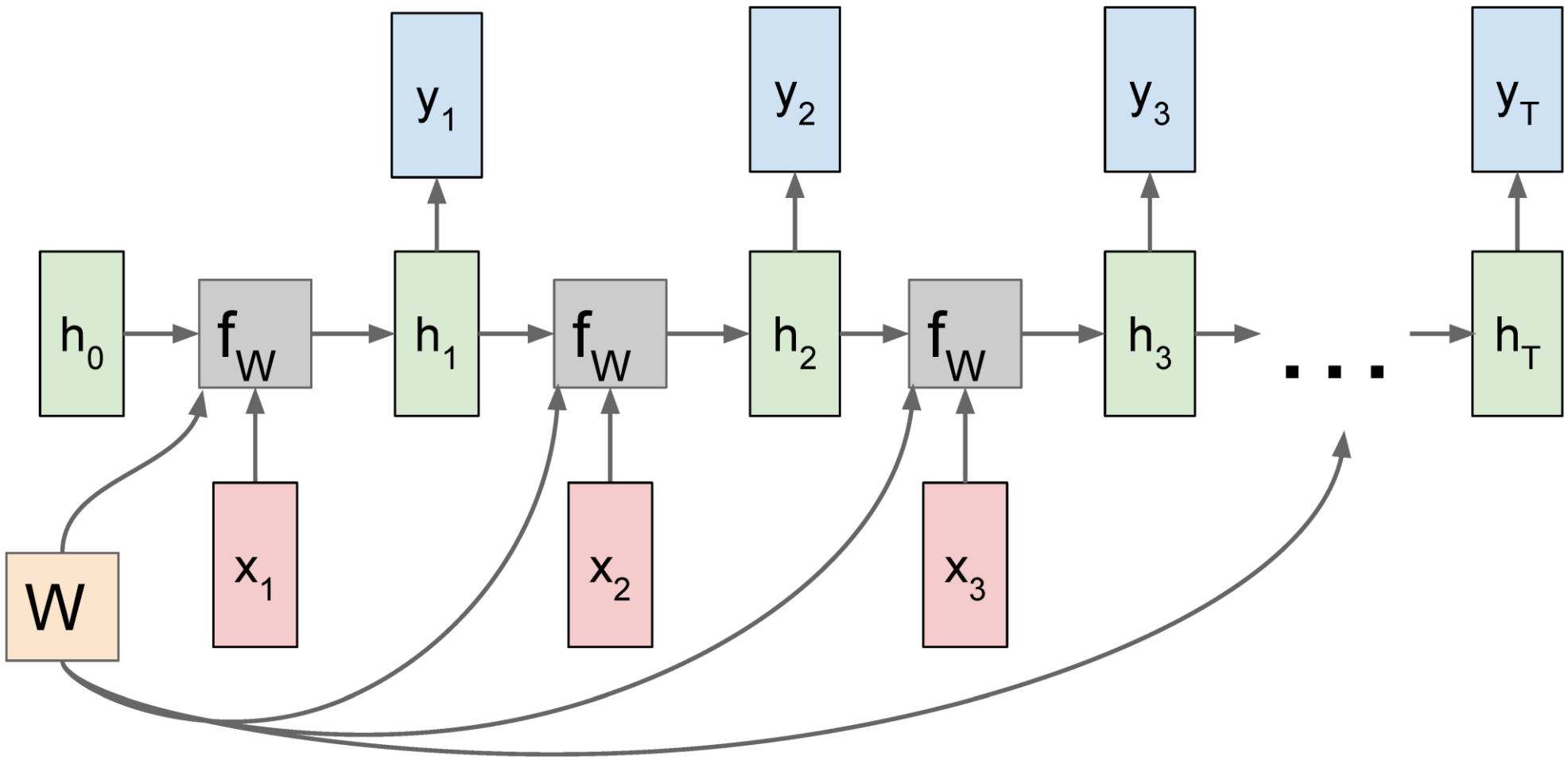


RNN: Computational Graph

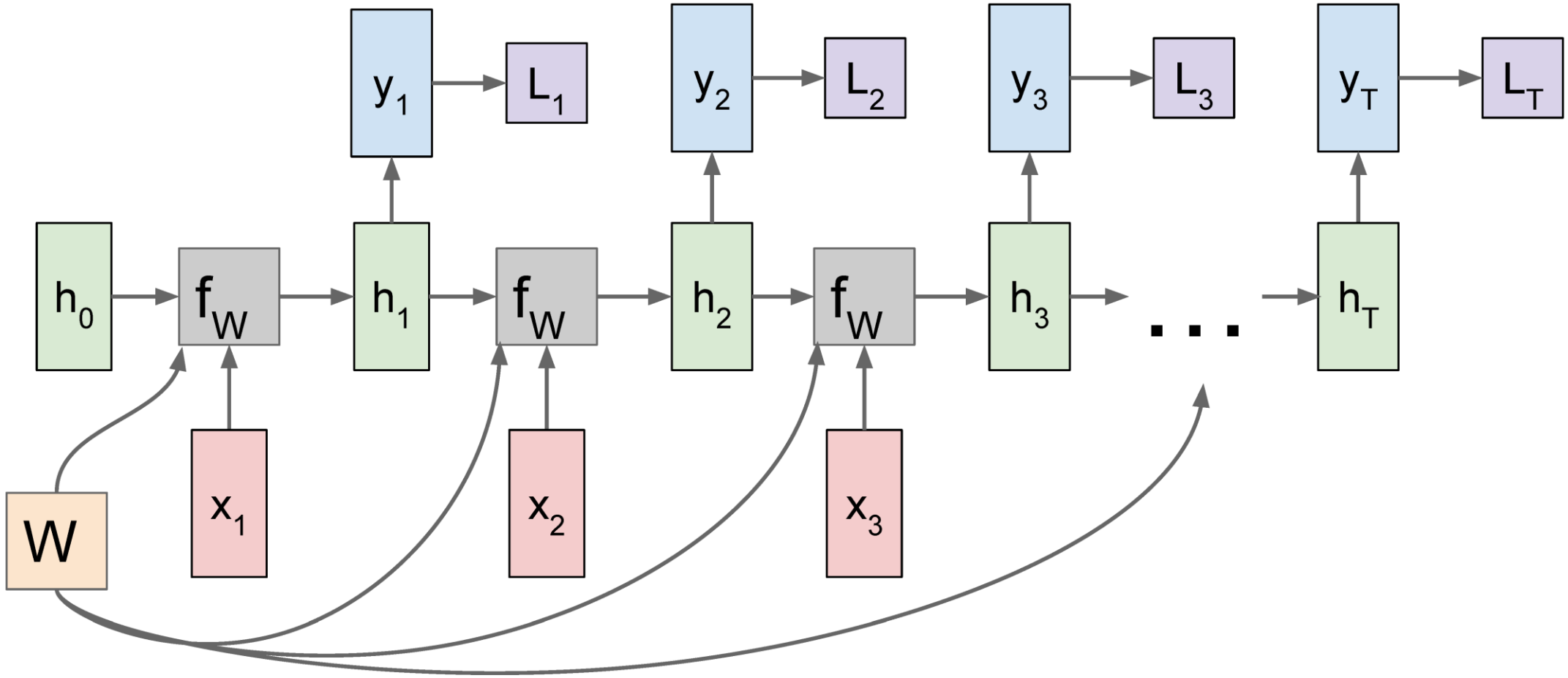
Re-use the same weight matrix at every time-step



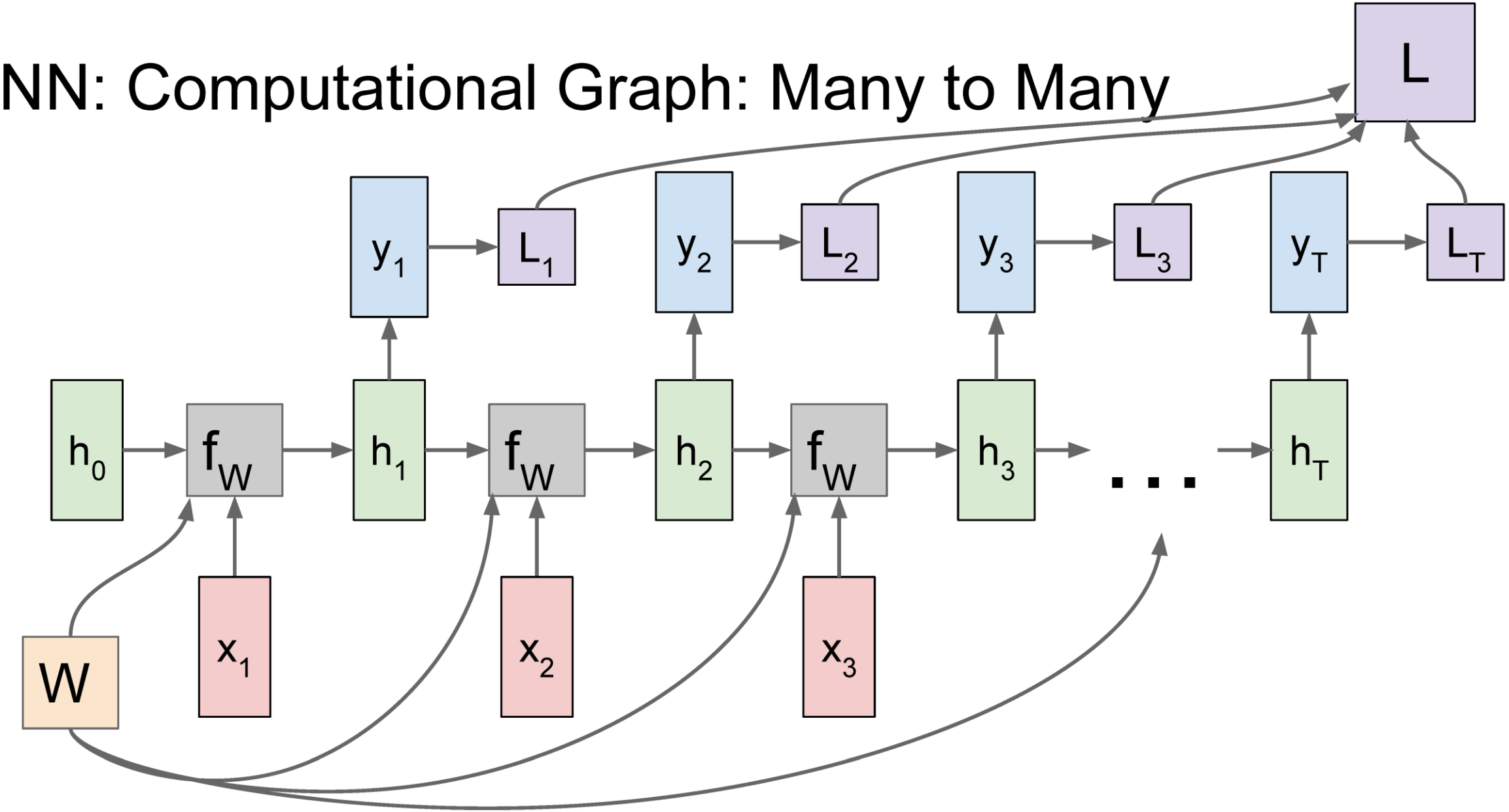
RNN: Computational Graph: Many to Many



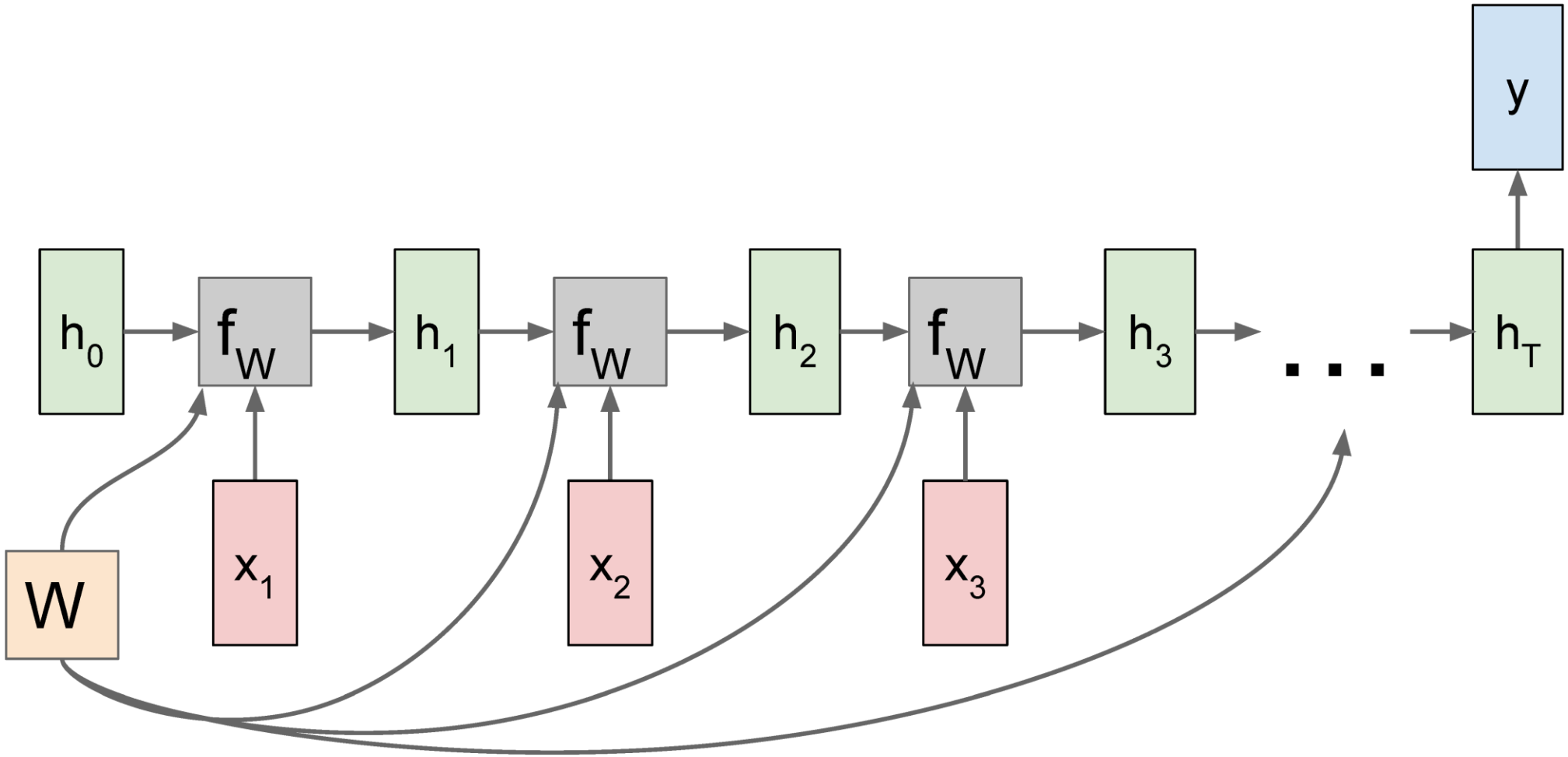
RNN: Computational Graph: Many to Many



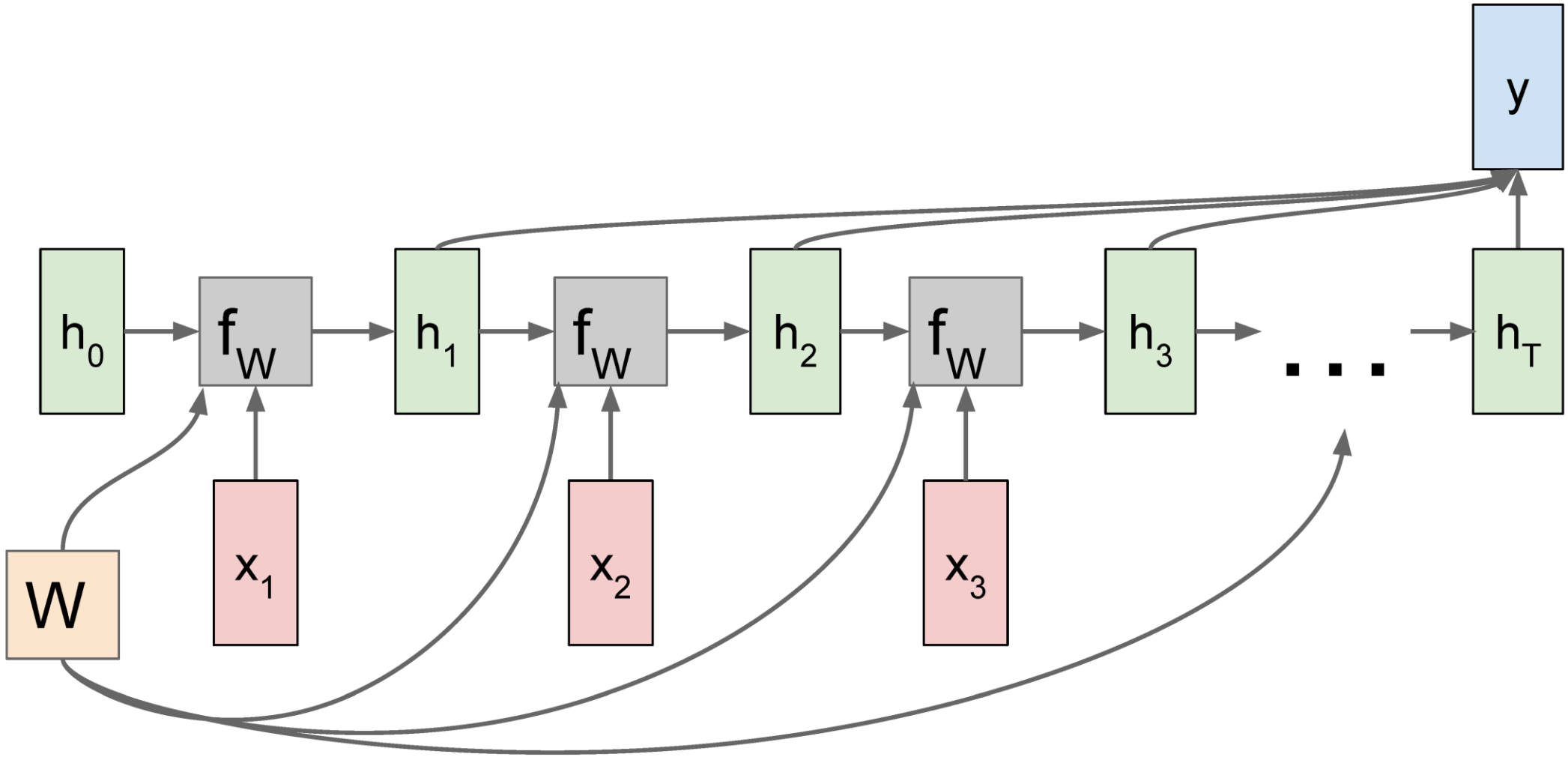
RNN: Computational Graph: Many to Many



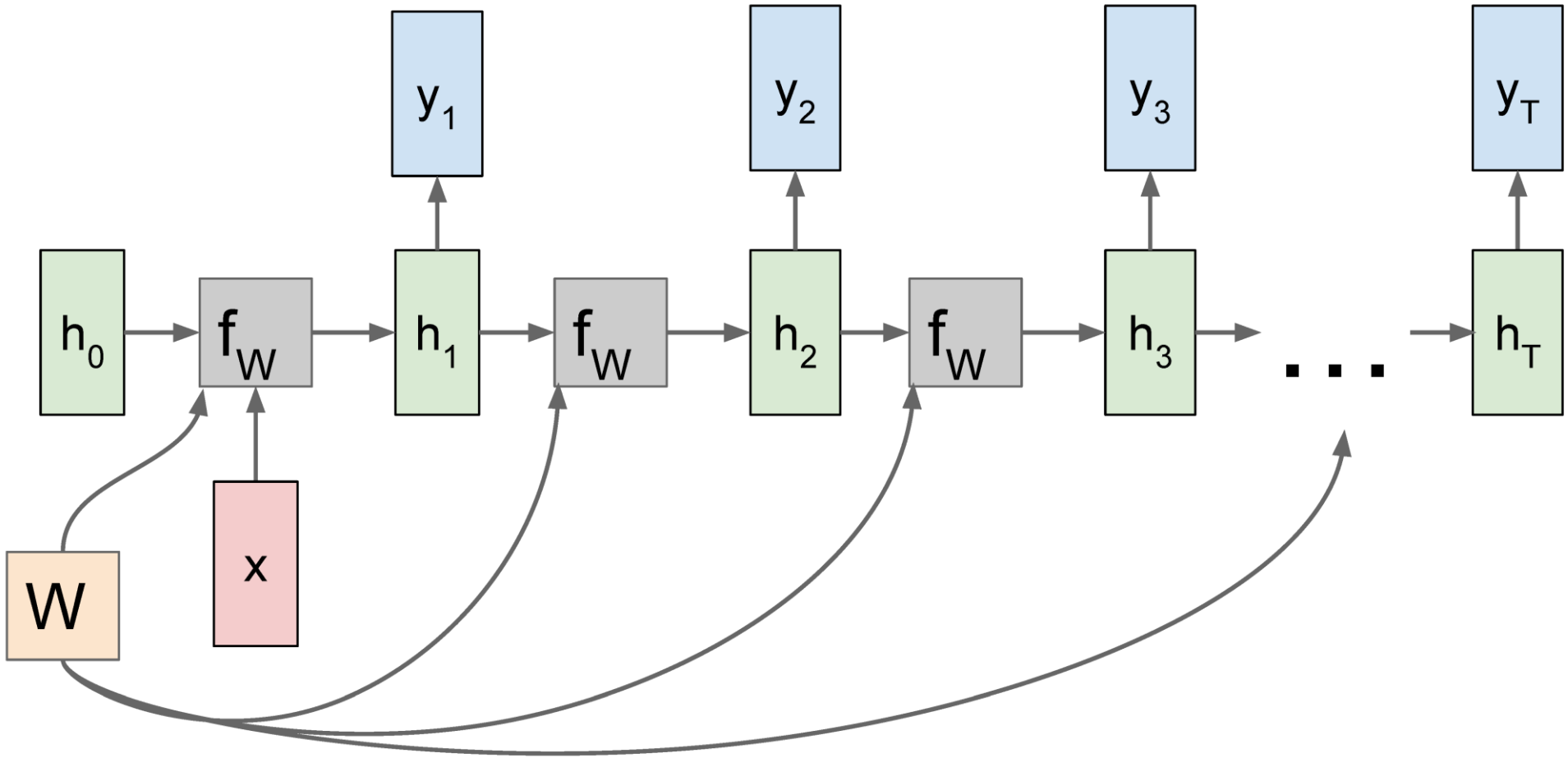
RNN: Computational Graph: Many to One



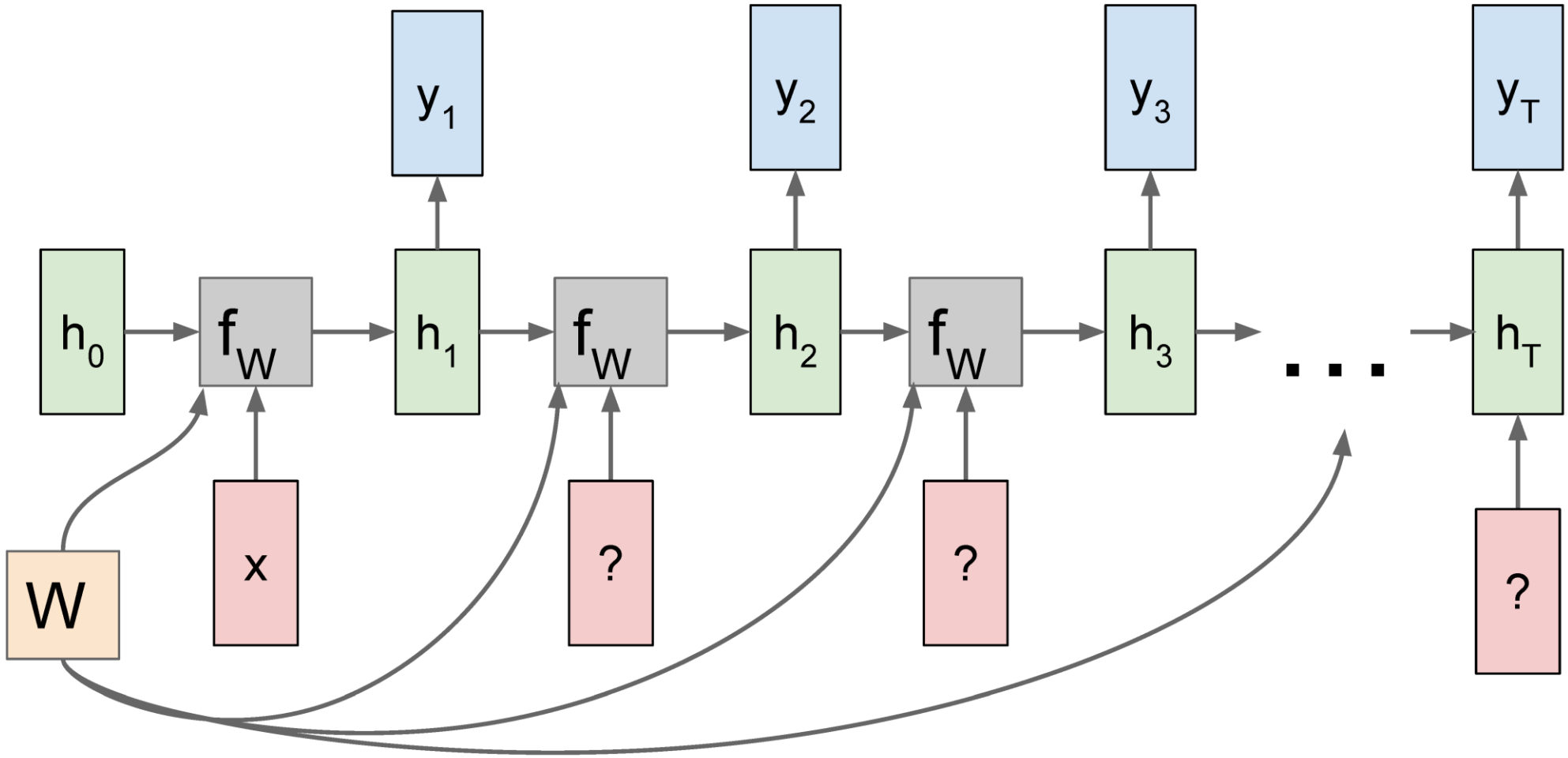
RNN: Computational Graph: Many to One



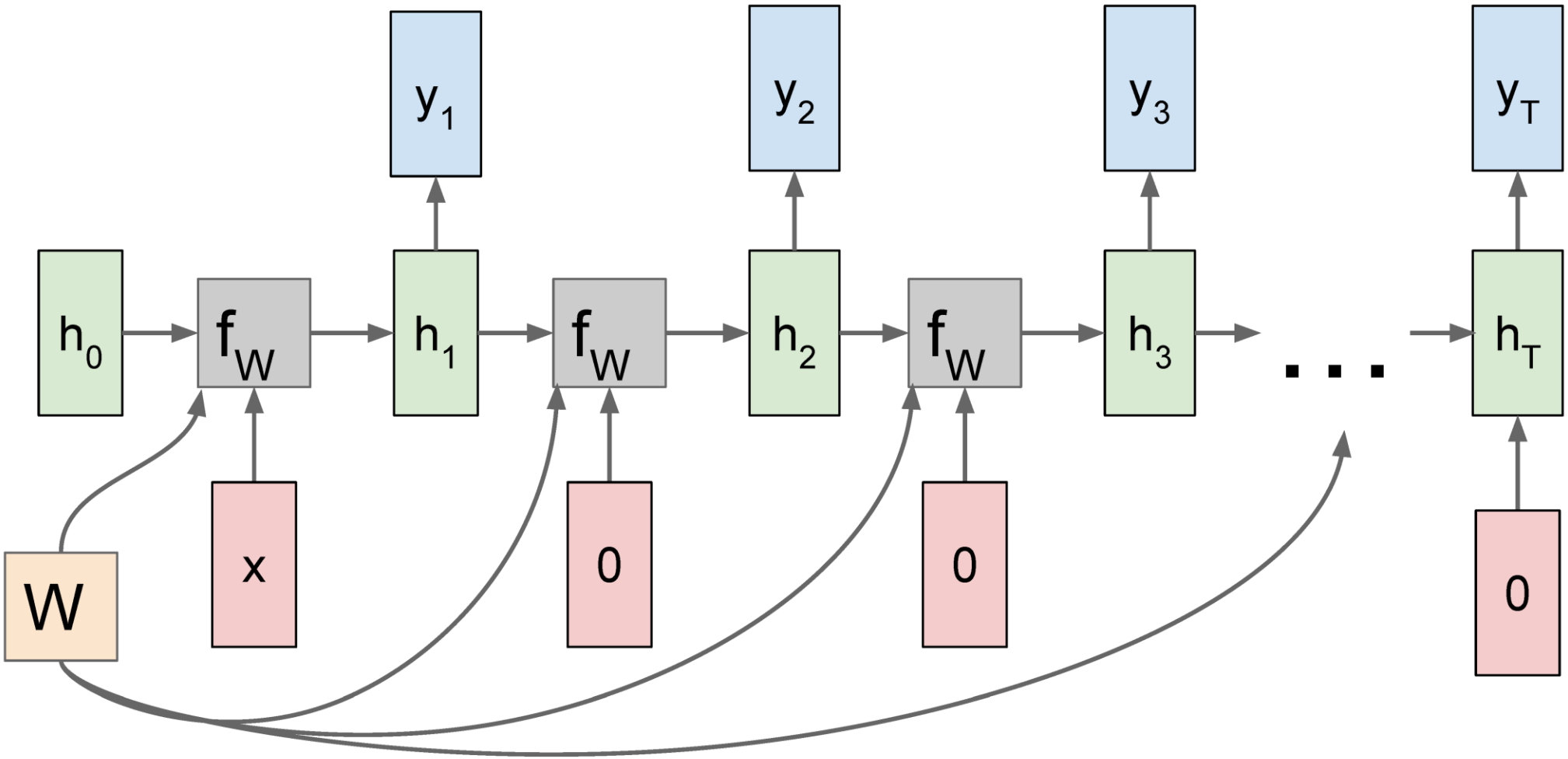
RNN: Computational Graph: One to Many



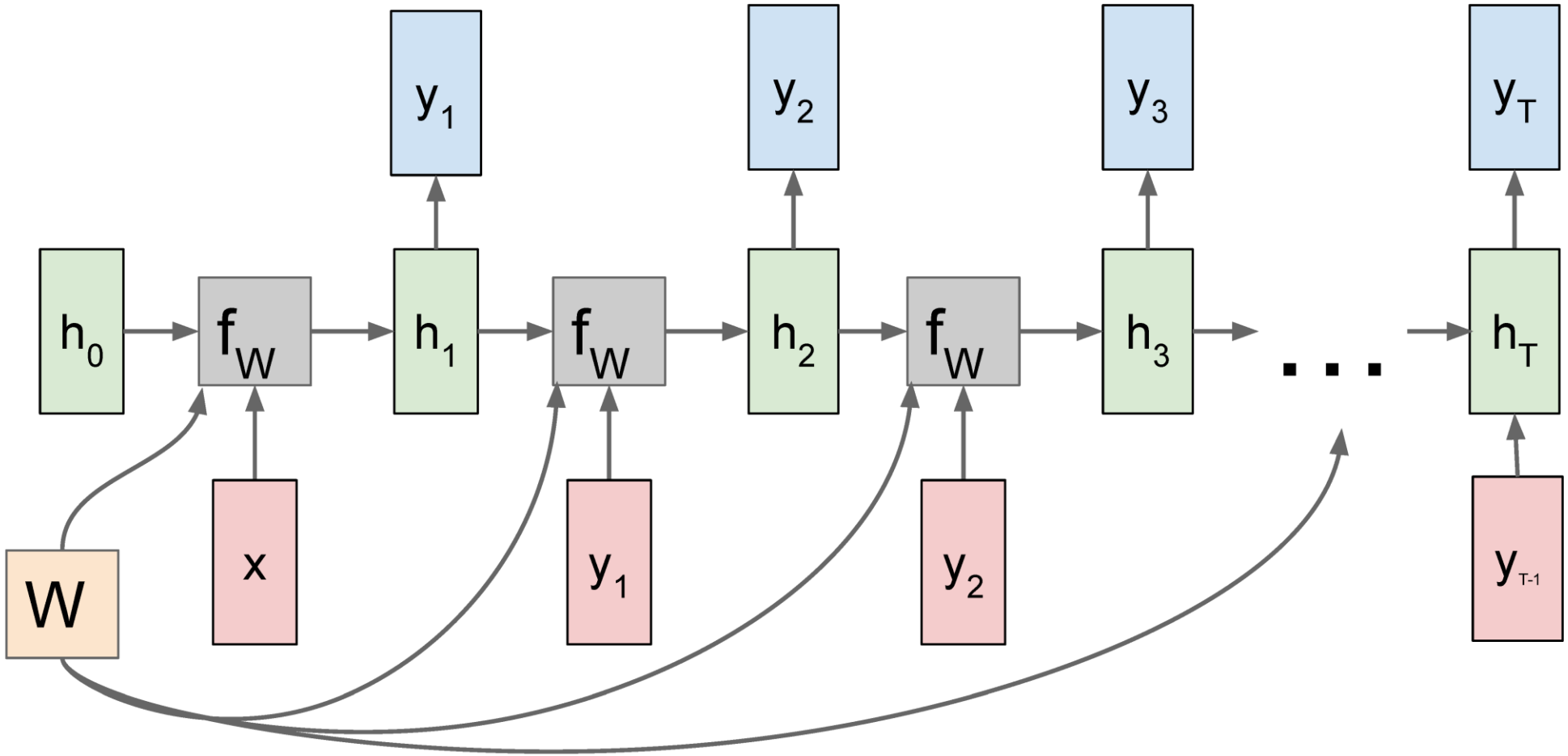
RNN: Computational Graph: One to Many



RNN: Computational Graph: One to Many

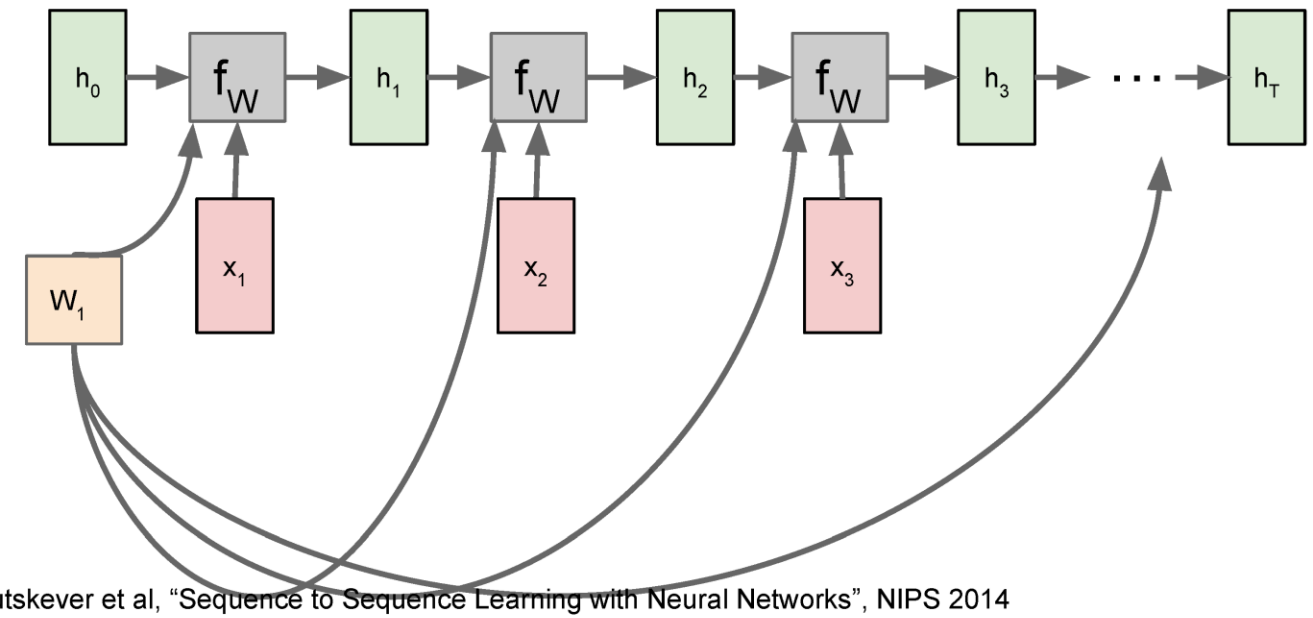


RNN: Computational Graph: One to Many



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

Many to one: Encode input sequence in a single vector

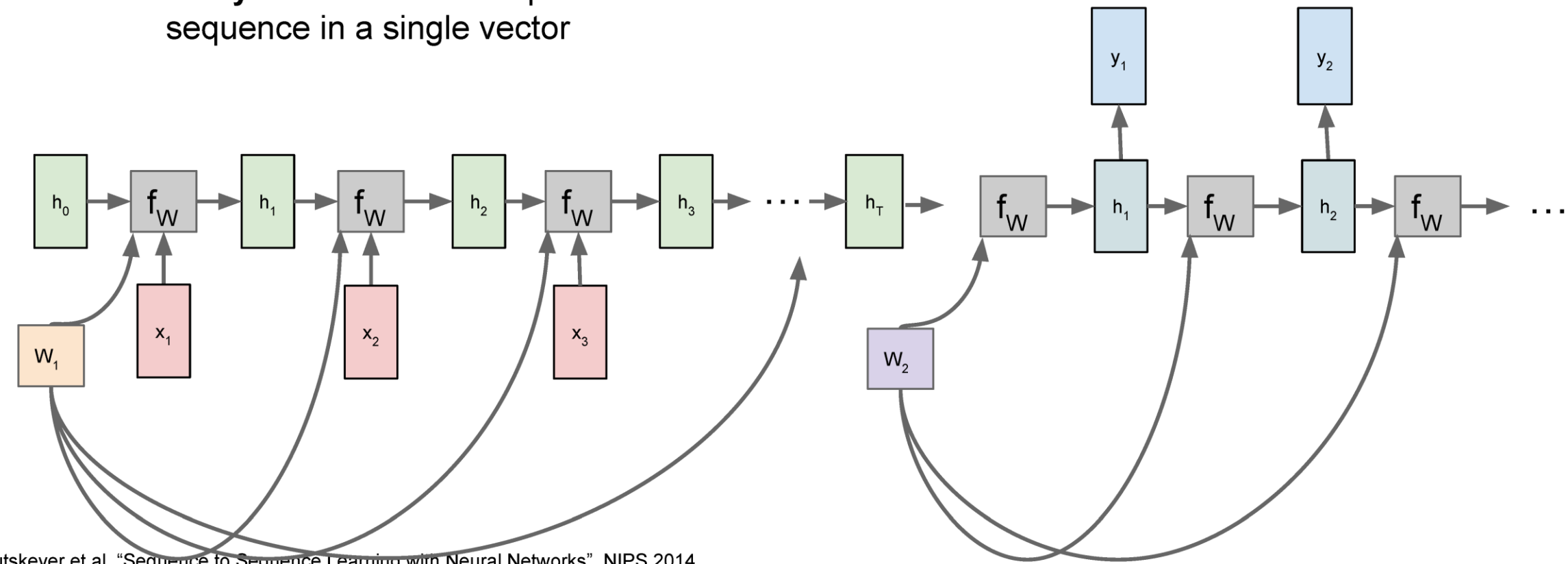


Sutskever et al, “Sequence to Sequence Learning with Neural Networks”, NIPS 2014

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

Many to one: Encode input sequence in a single vector

One to many: Produce output sequence from single input vector

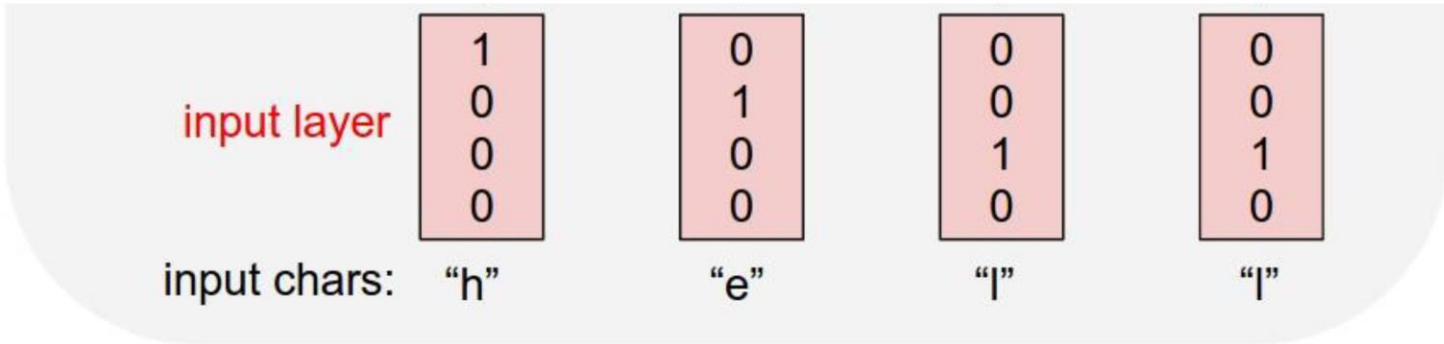


Sutskever et al, “Sequence to Sequence Learning with Neural Networks”, NIPS 2014

Example:
Character-level
Language Model

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”

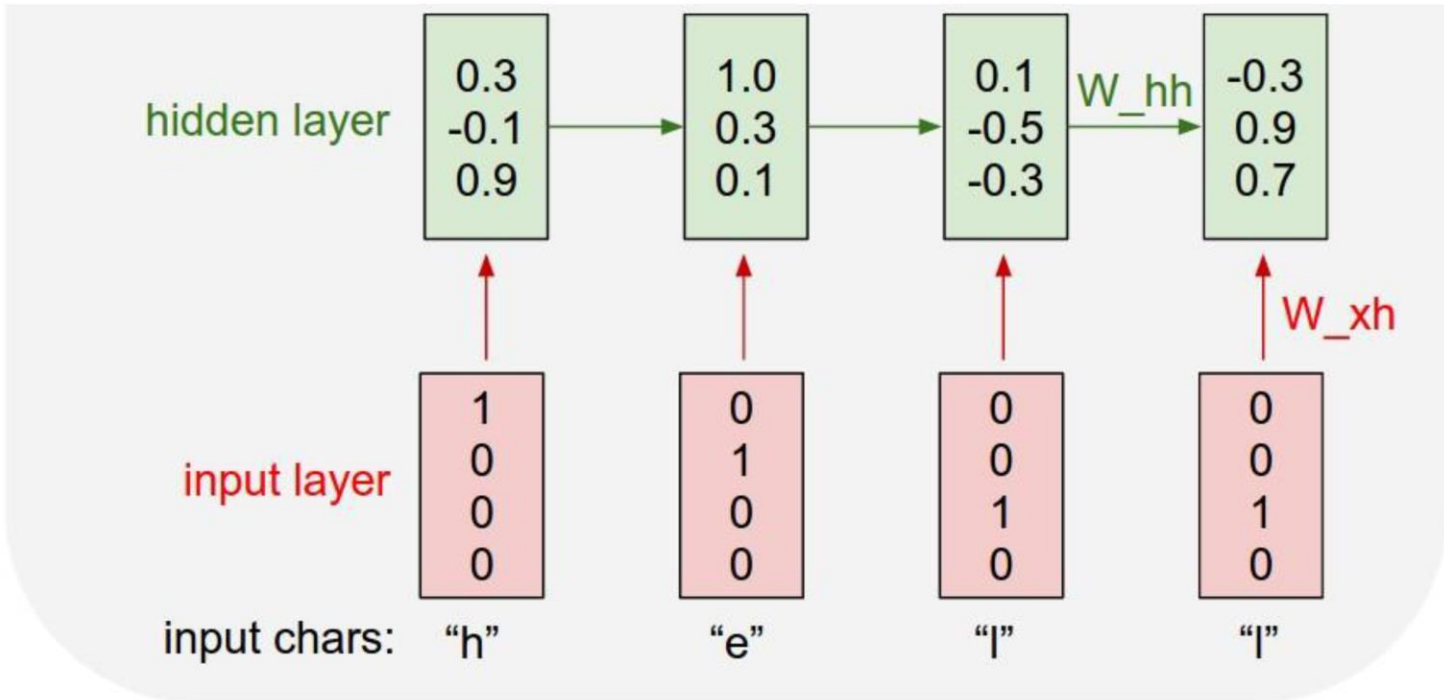


Example: Character-level Language Model

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

Vocabulary:
[h,e,l,o]

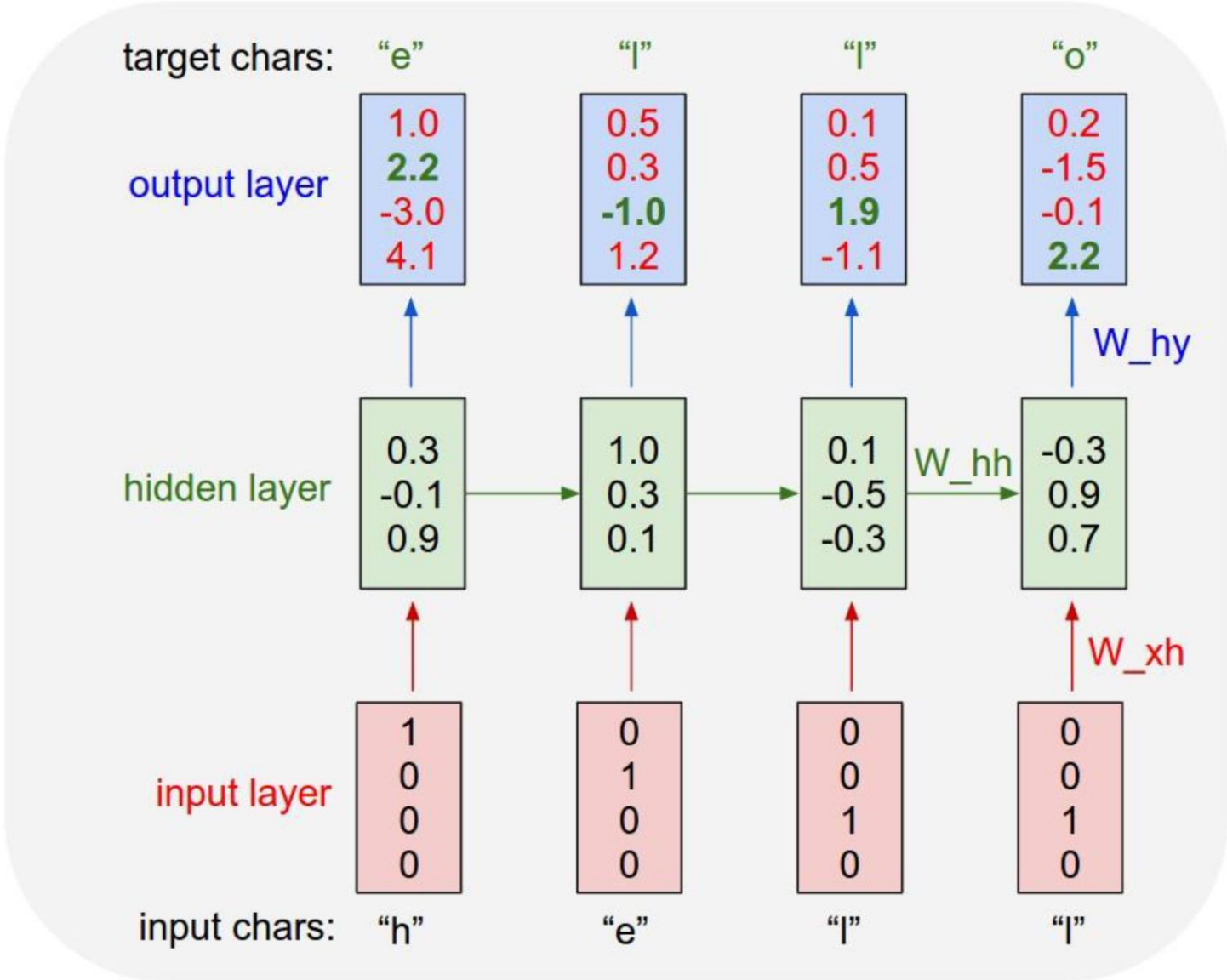
Example training sequence:
“hello”



Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

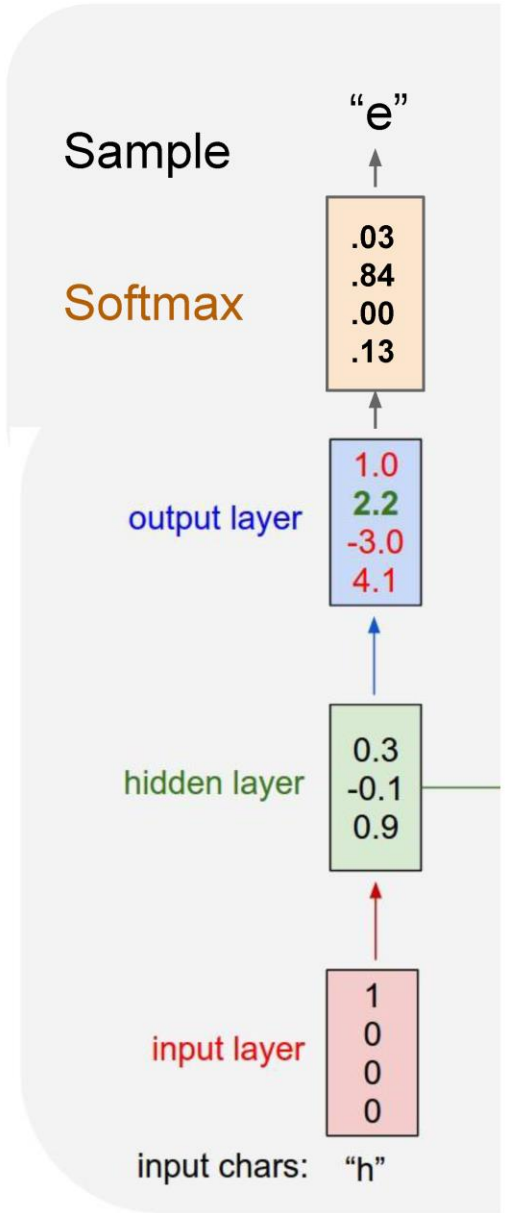
Example training sequence:
“hello”



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

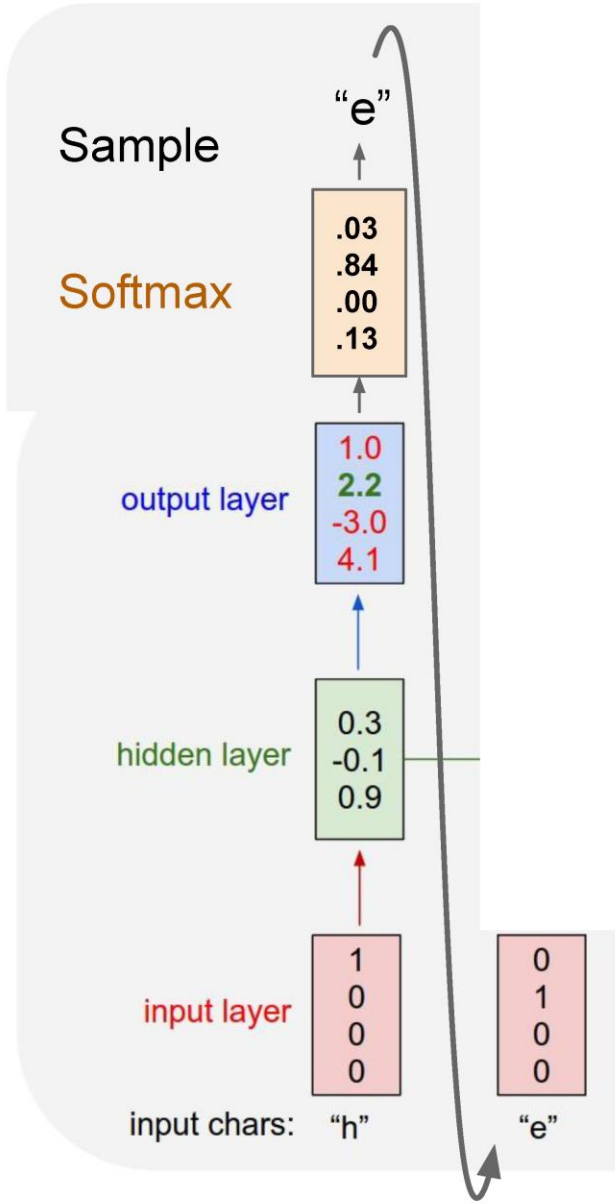
At test-time sample characters one at a time, feed back to model



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

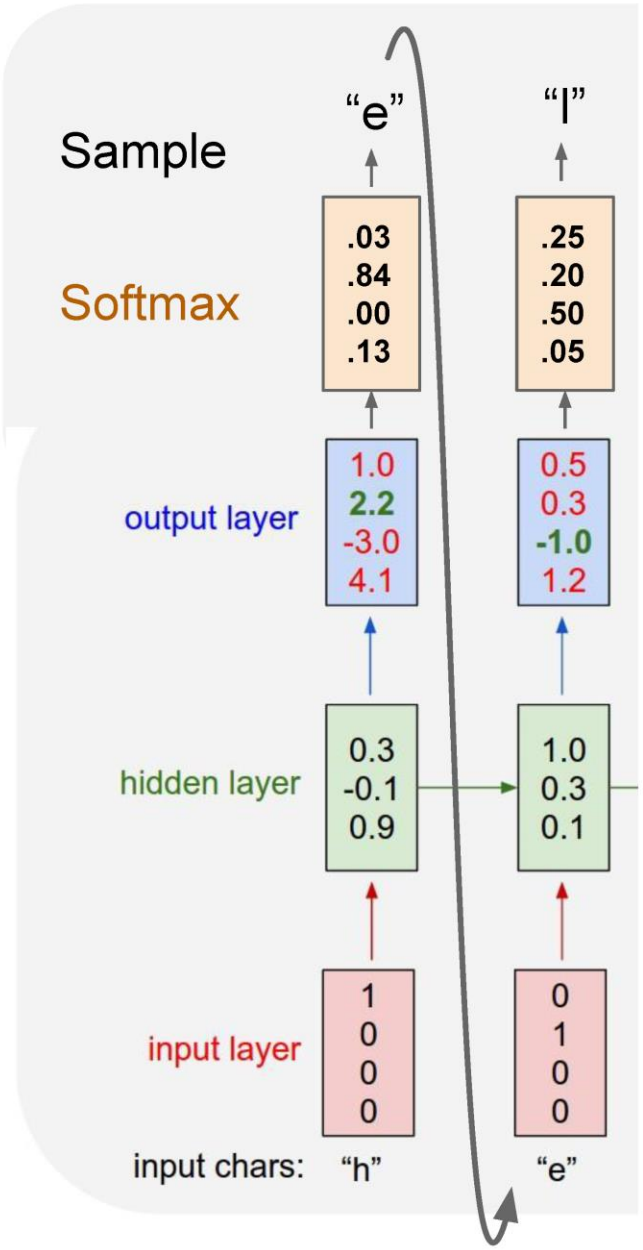
At test-time sample characters one at a time, feed back to model



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

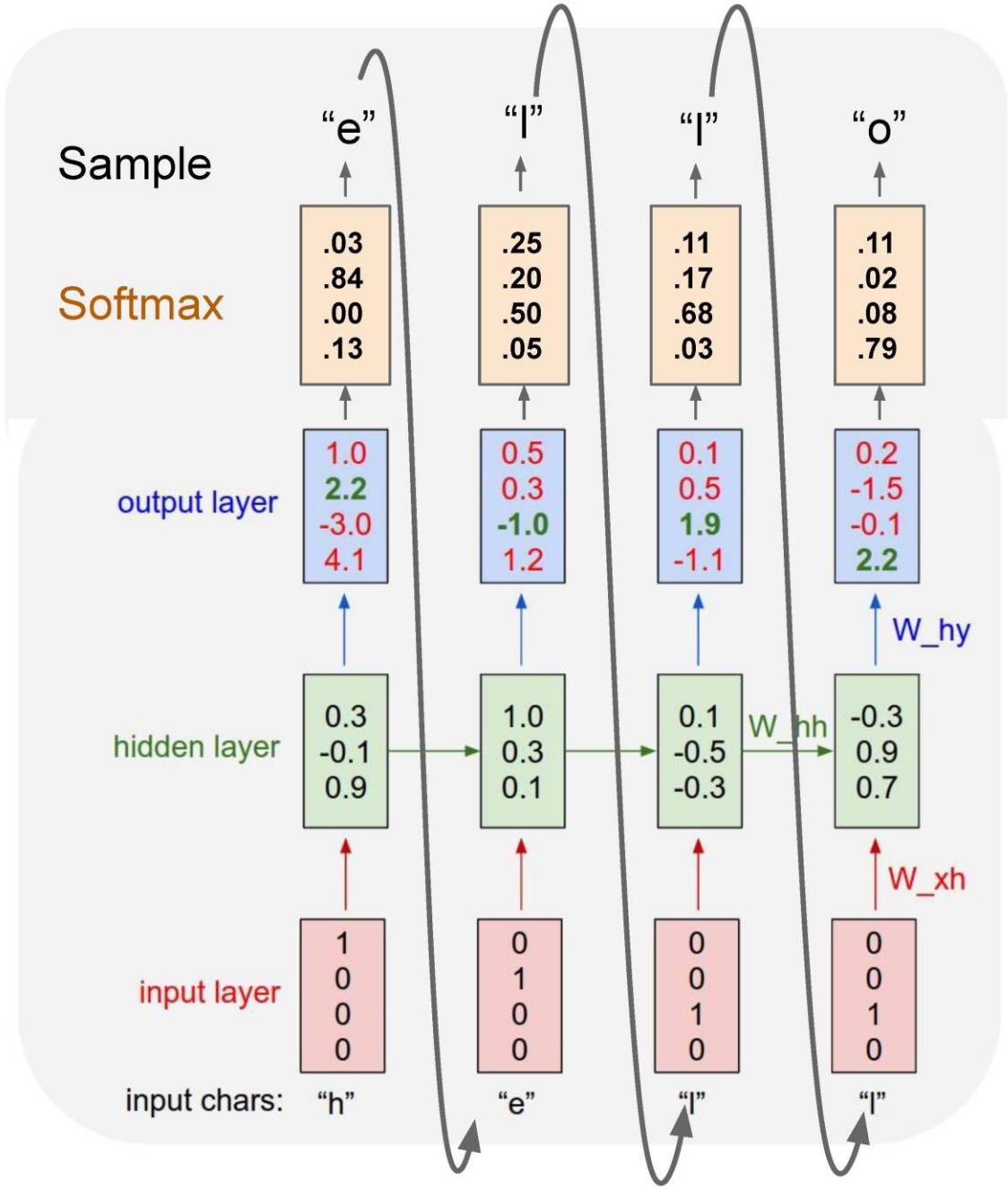
At test-time sample characters one at a time, feed back to model



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

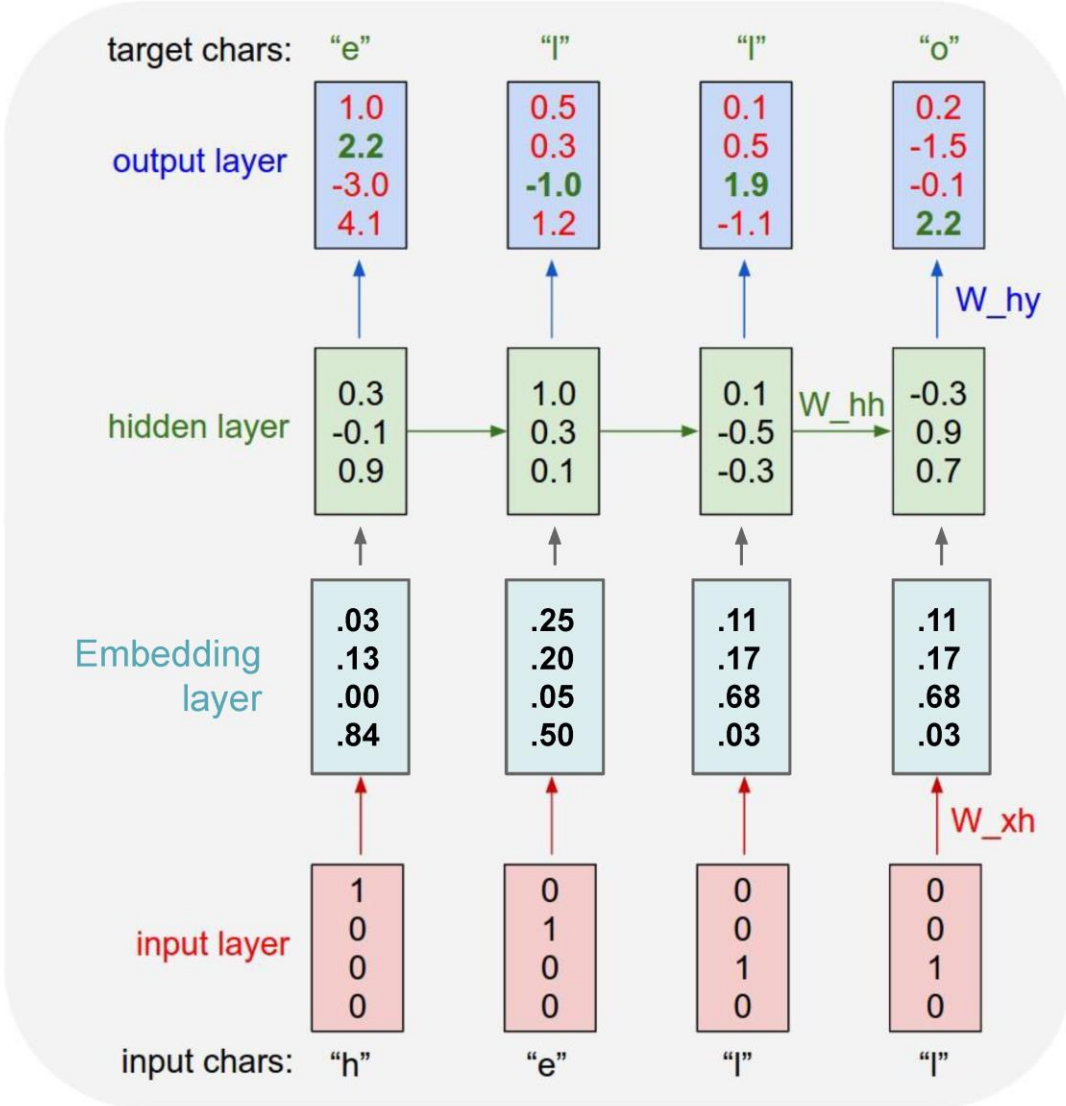
At test-time sample characters one at a time, feed back to model



Example: Character-level Language Model Sampling

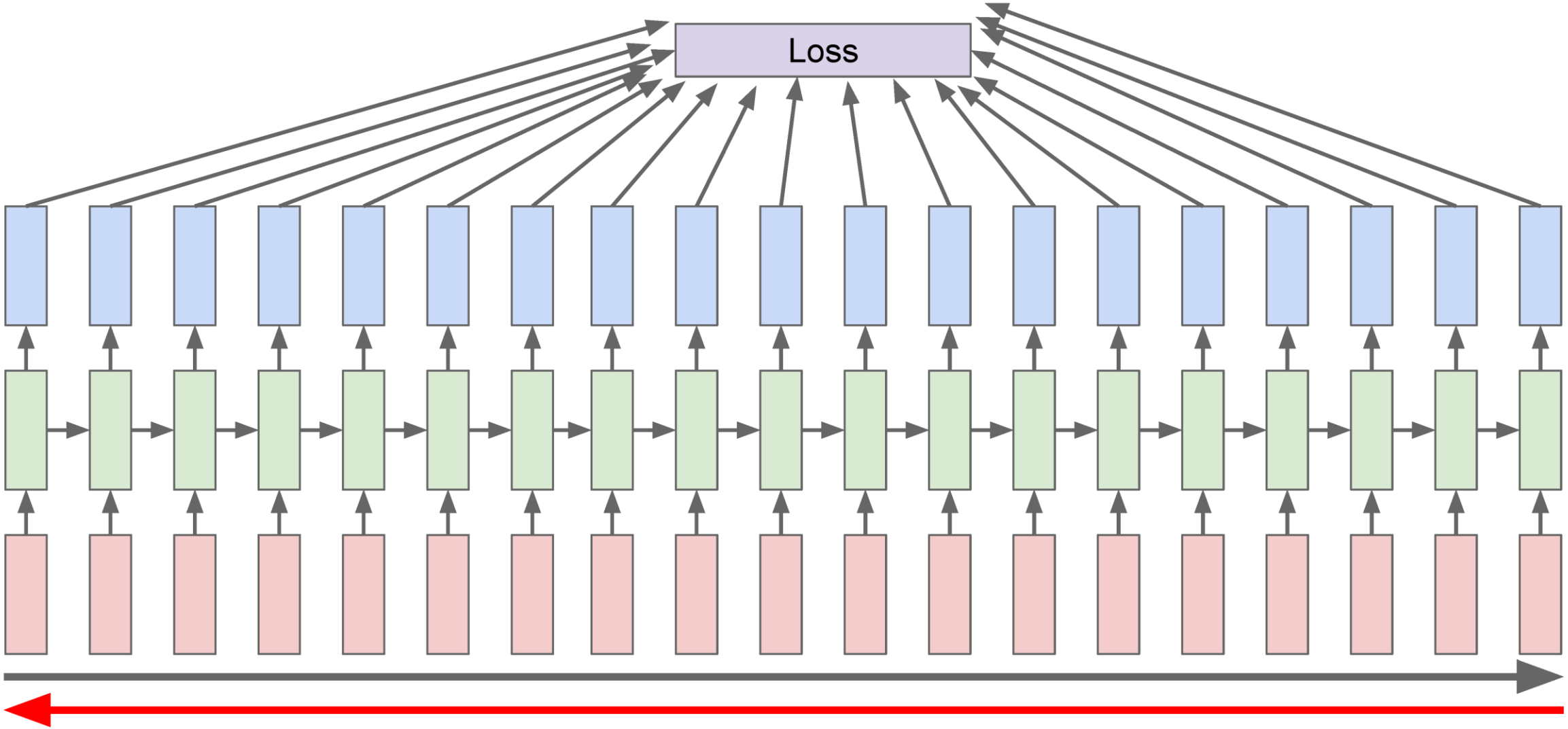
$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{14} \\ w_{31} & w_{32} & w_{33} & w_{14} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} w_{11} \\ w_{21} \\ w_{31} \end{bmatrix}$$

Matrix multiply with a one-hot vector just extracts a column from the weight matrix. We often put a separate **embedding** layer between input and hidden layers.

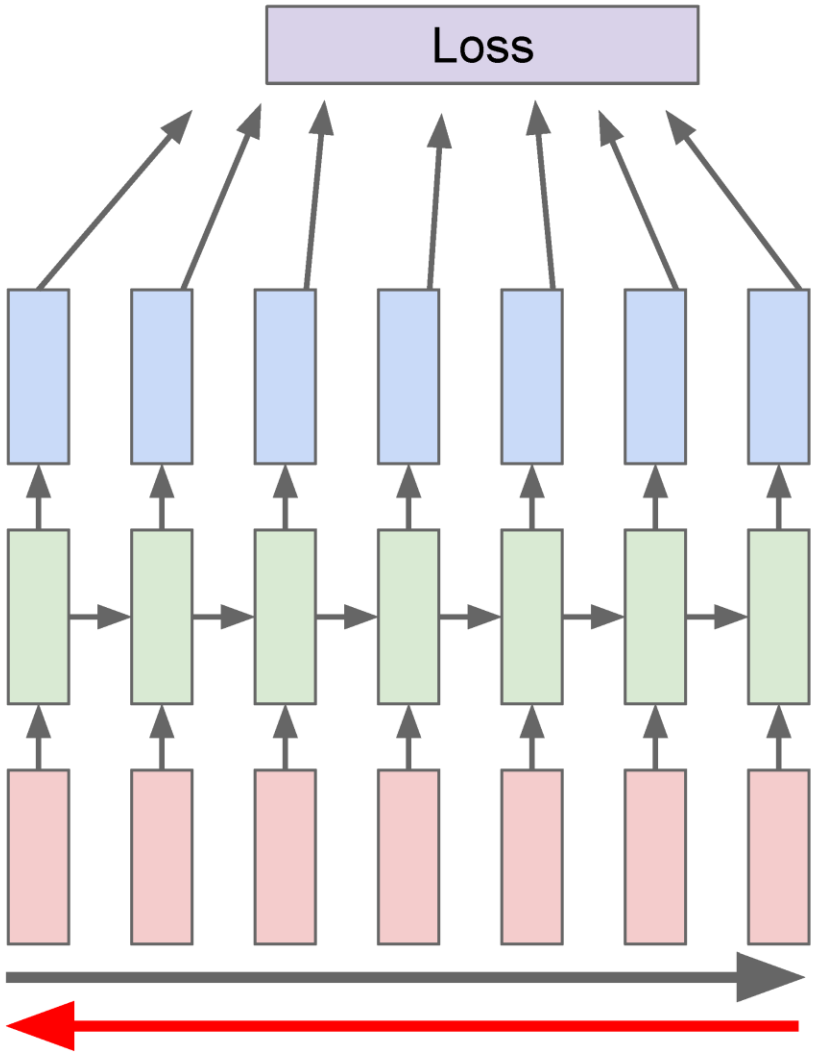


Backpropagation through time

Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient

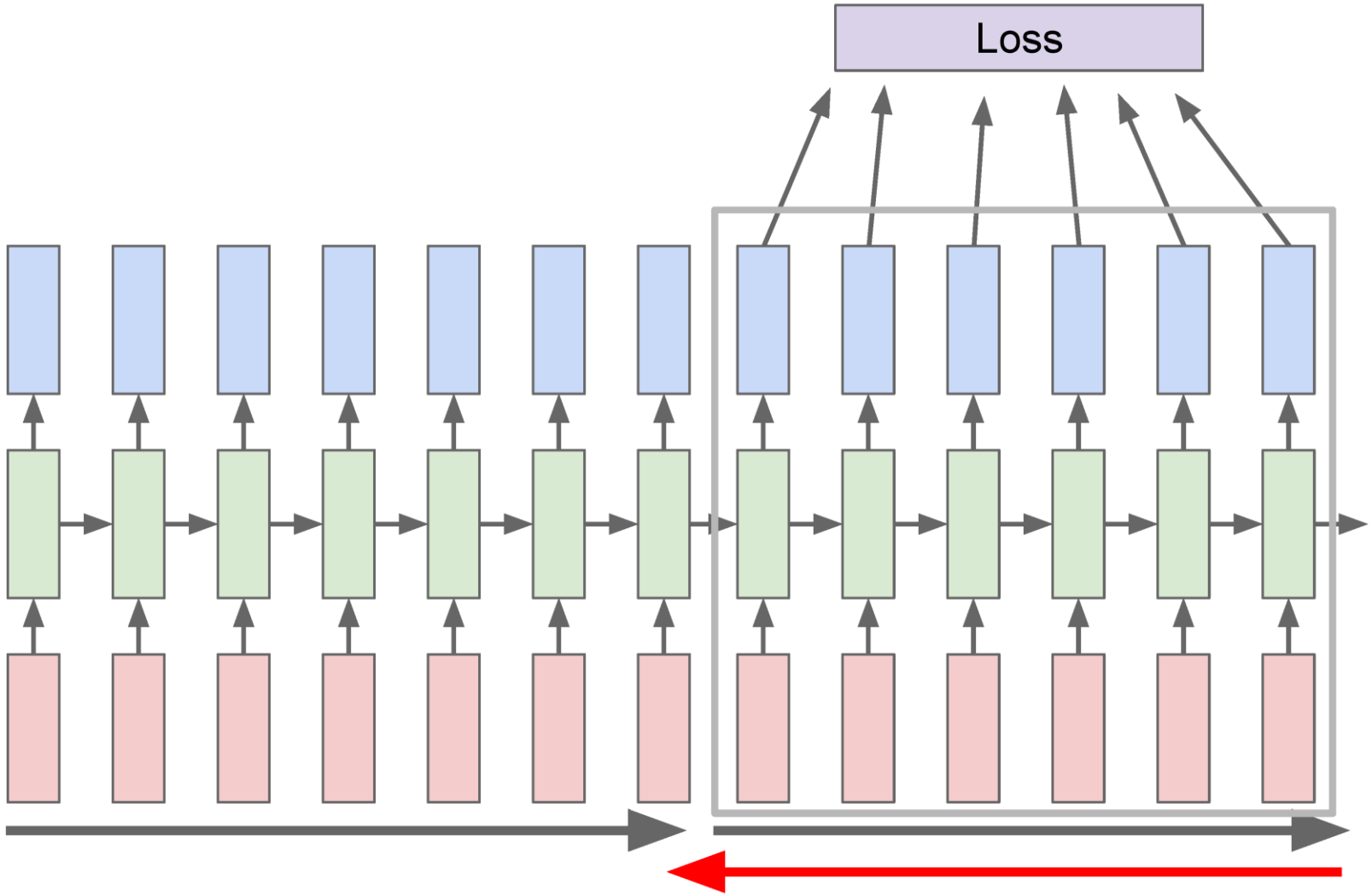


Truncated Backpropagation through time



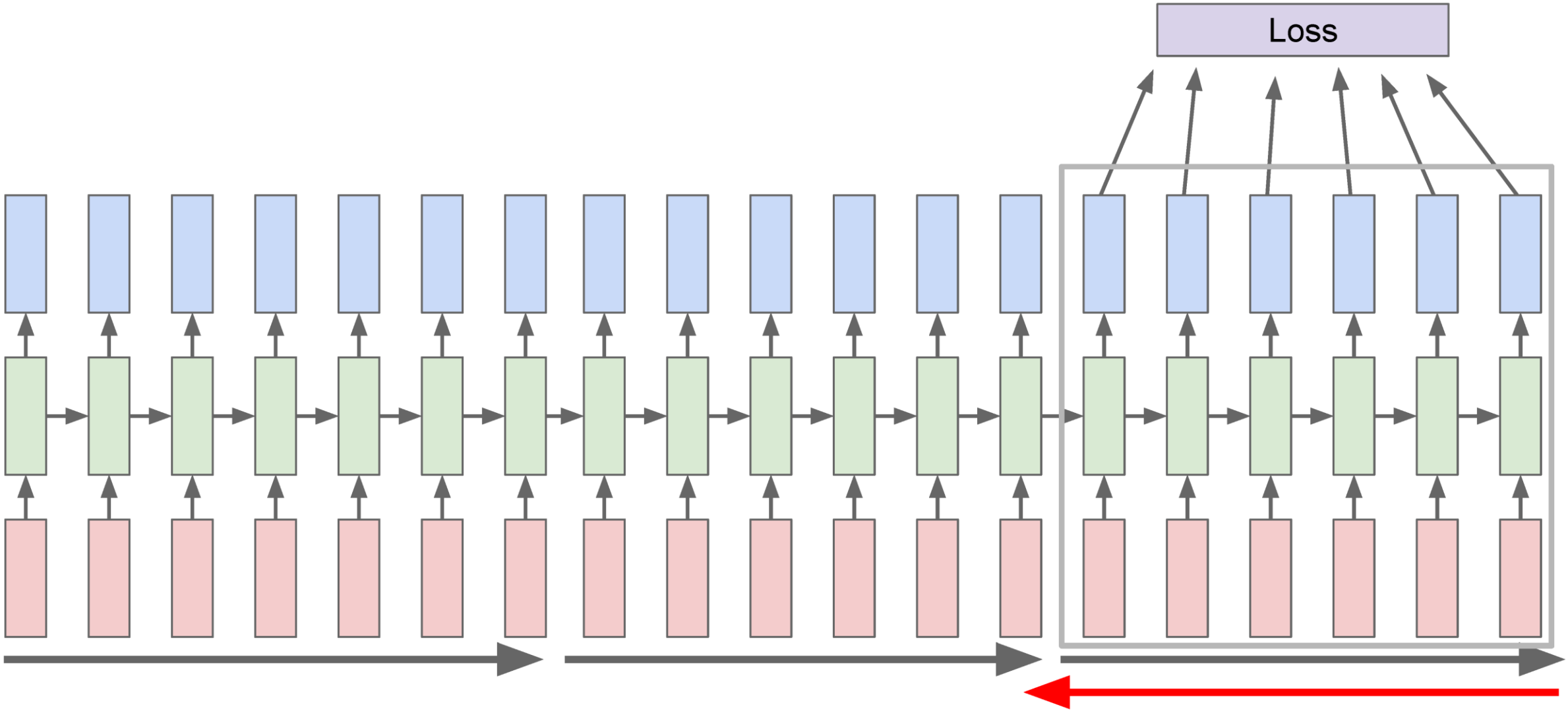
Run forward and backward through chunks of the sequence instead of whole sequence

Truncated Backpropagation through time



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

Truncated Backpropagation through time



Minimal character-level language model with a Vanilla Recurr

min-char-rnn.py

```python
"""
Minimal character-level Vanilla RNN model. Writt
BSD License
"""
import numpy as np

# data I/O
data = open('input.txt', 'r').read() # should be
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print 'data has %d characters, %d unique.' % (da
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 neur
seq_length = 25 # number of steps to unroll the
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0
Whh = np.random.randn(hidden_size, hidden_size)*
Why = np.random.randn(vocab_size, hidden_size)*0
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 parameter
  """
  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
    xs[t][inputs[t]] = 1
    hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[
    ys[t] = np.dot(Why, hs[t]) + by # unnormalized log p
    ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # prob
    loss += -np.log(ps[t][targets[t],0]) # softmax (cros
  # backward pass: compute gradients going backwards
  dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(W
  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://cs
    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
    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 mitigat
  return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs

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 ti
  """
  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

Example:
Image Captioning

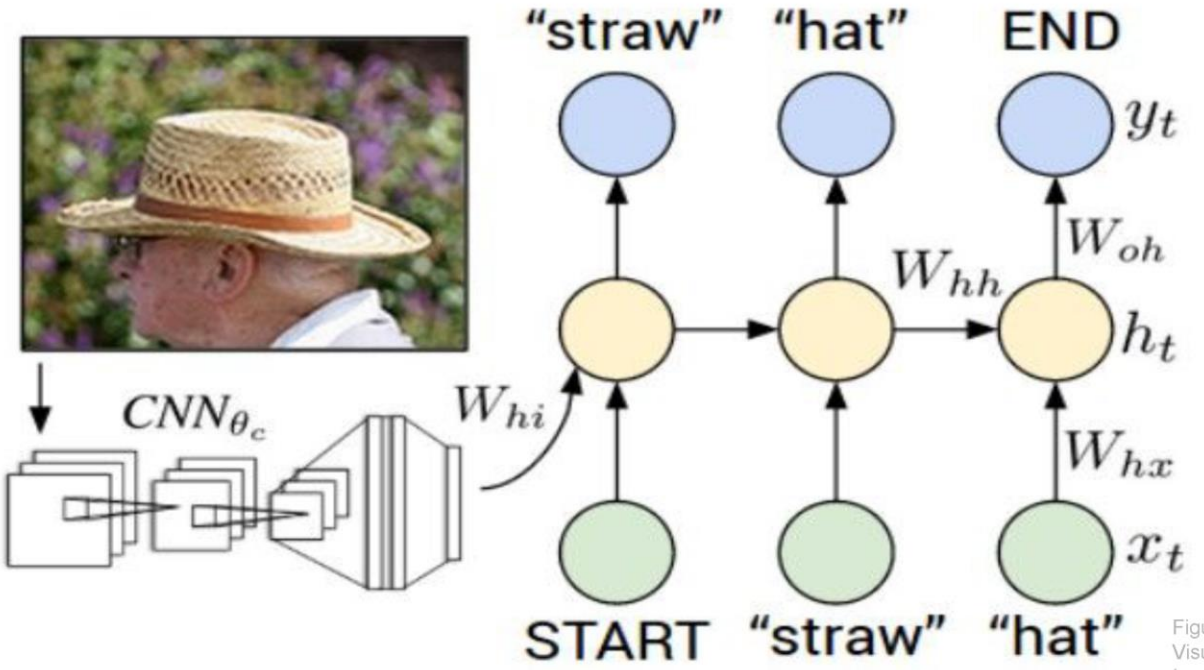
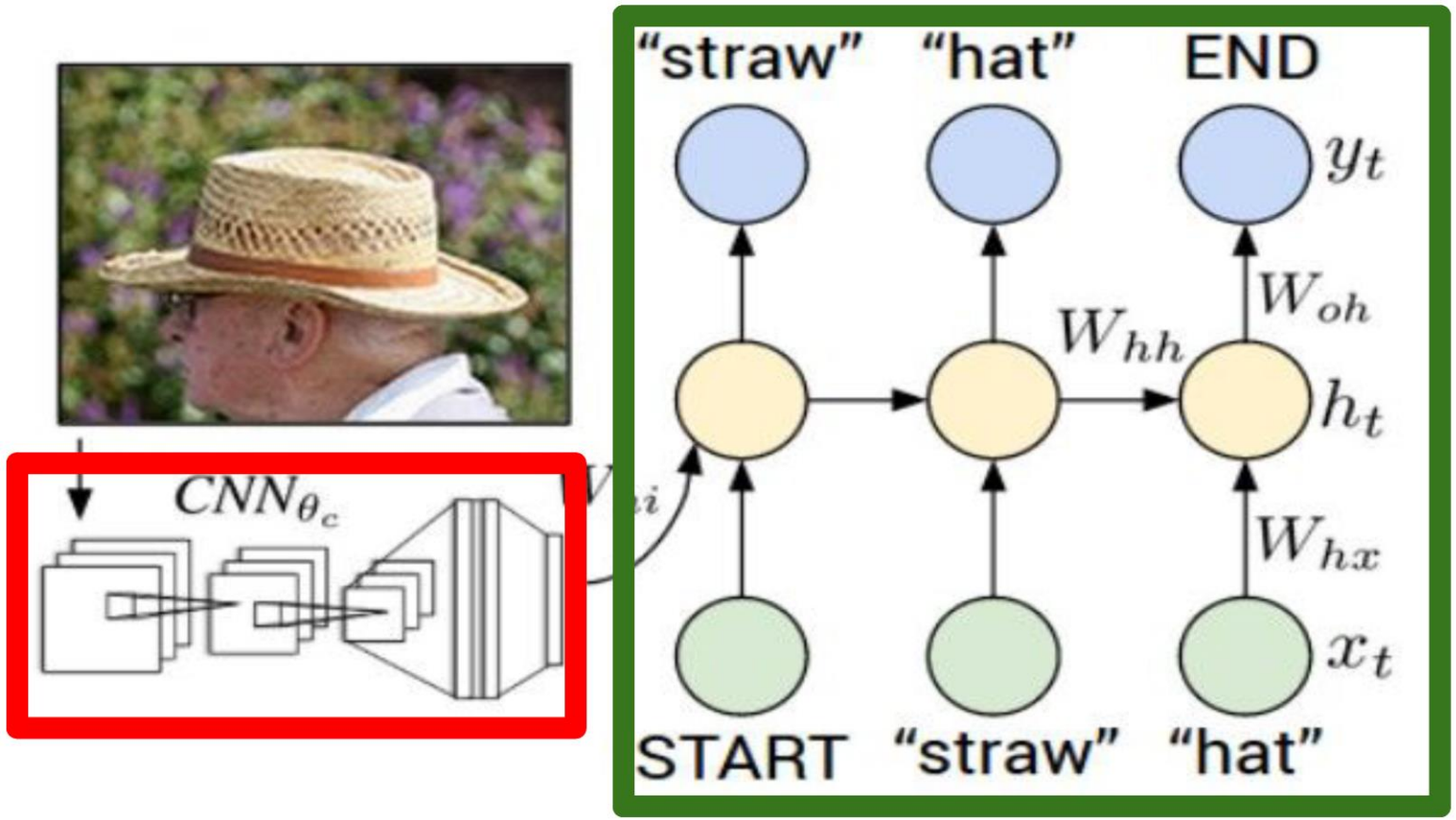


Figure from Karpathy et a, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015; figure copyright IEEE, 2015.
Reproduced for educational purposes.

Deep Visual-Semantic Alignments for Generating Image Descriptions, CVPR 2015

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

FC-1000

softmax

X

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

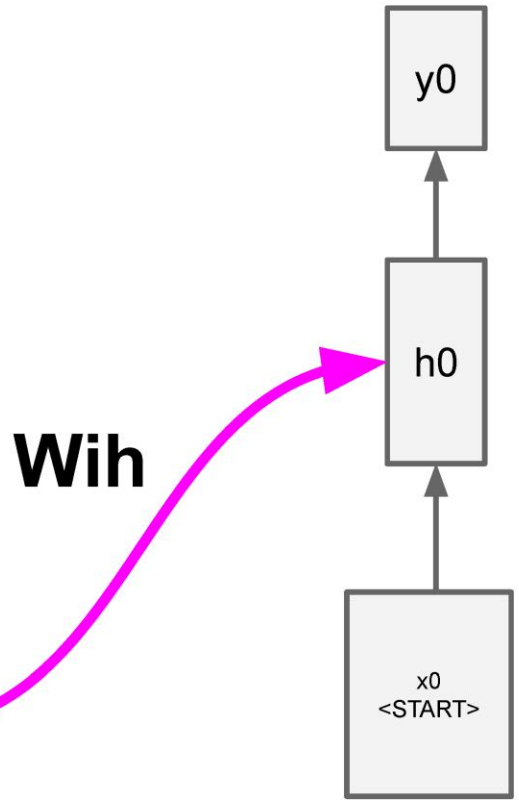
FC-4096

FC-4096

test image

x0
<START>

test image



before:

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

now:

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

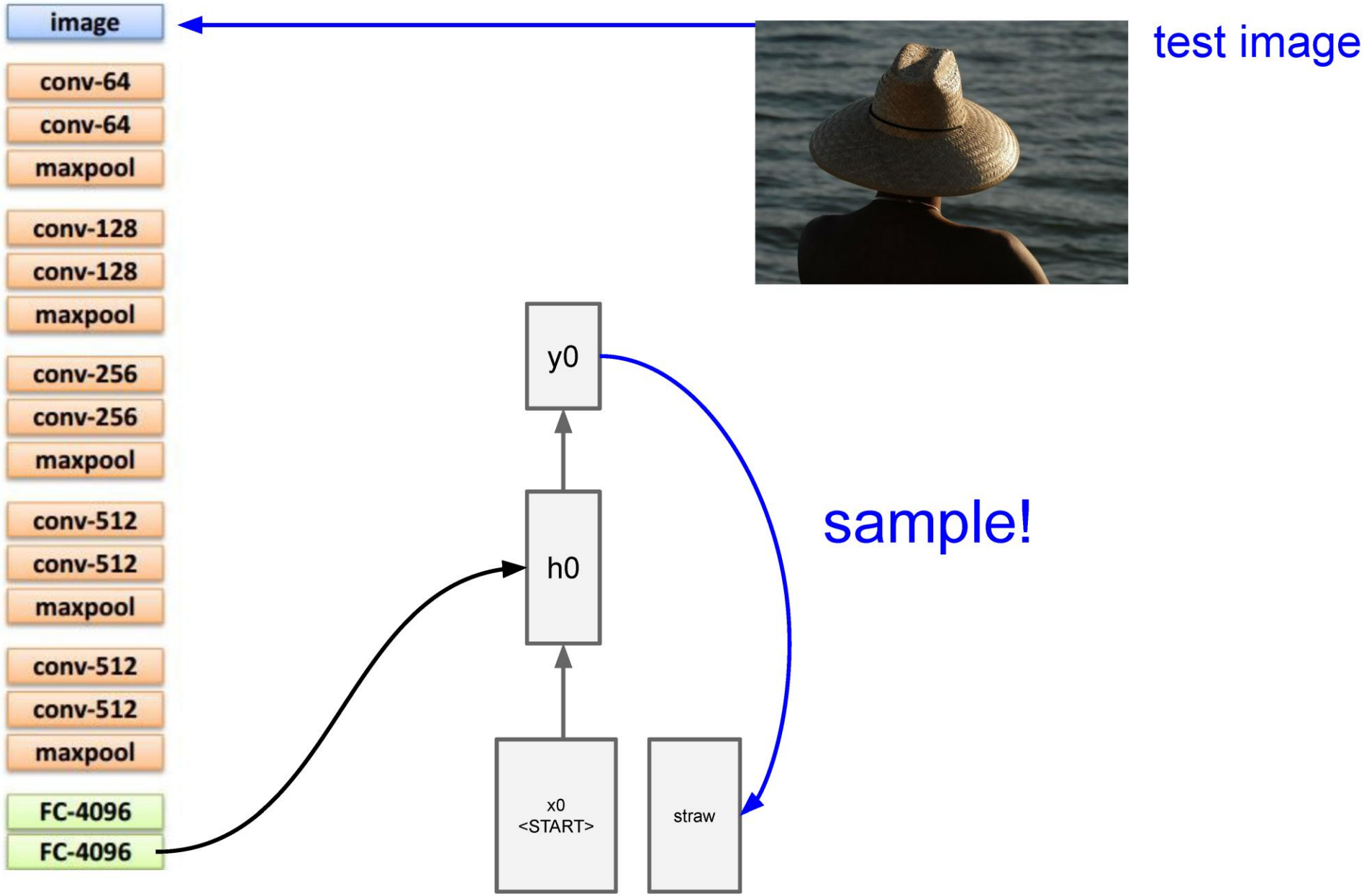


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
y0
h0
x0
<START>
straw
sample!

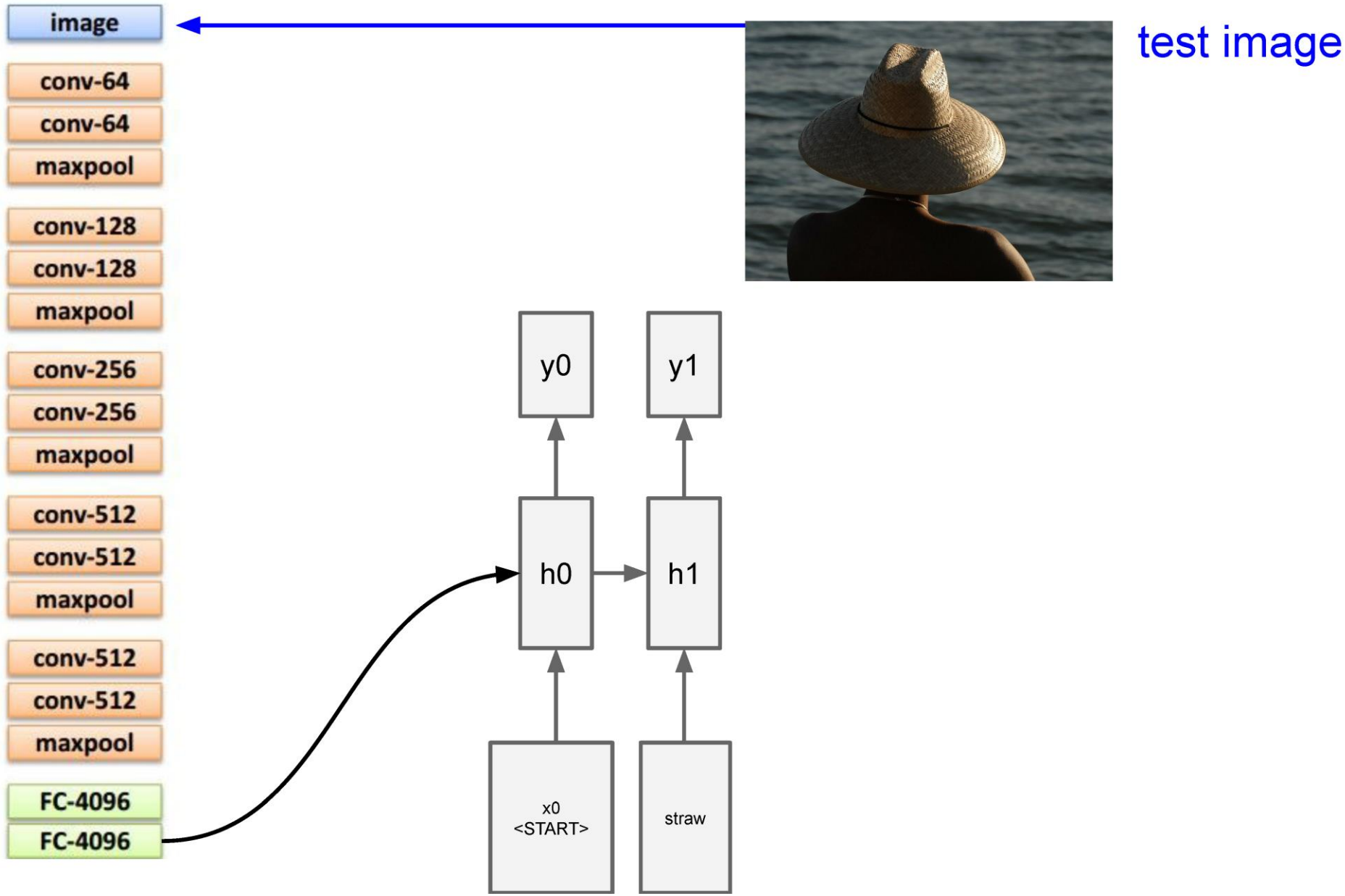


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
y0
y1
h0
h1
x0
<START>
straw

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

x0
<START>

straw

hat

sample!

image

conv-64
conv-64
maxpool

conv-128
conv-128
maxpool

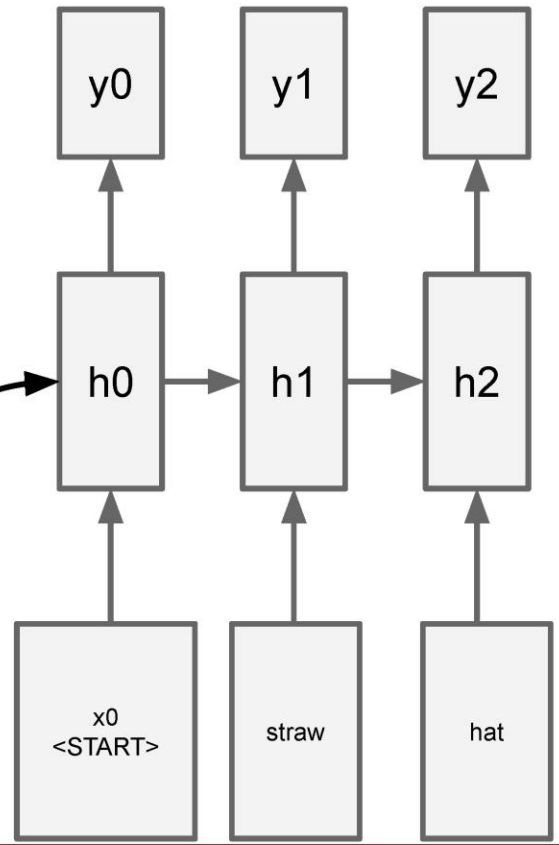
conv-256
conv-256
maxpool

conv-512
conv-512
maxpool

conv-512
conv-512
maxpool

FC-4096
FC-4096

test image



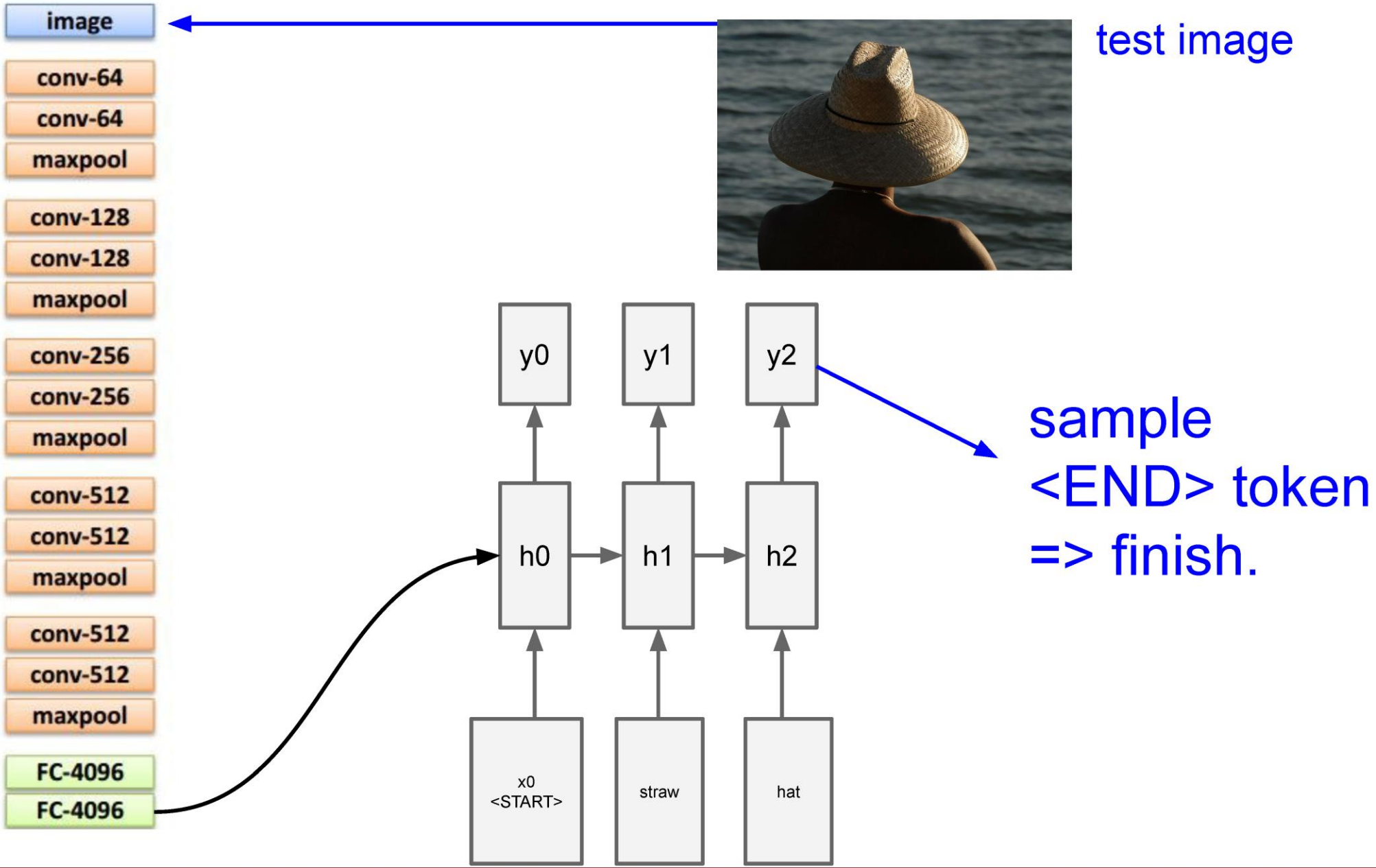


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
y0
y1
y2
h0
h1
h2
x0
<START>
straw
hat
sample
<END> token
=> finish.

Image Captioning: Example Results

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

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

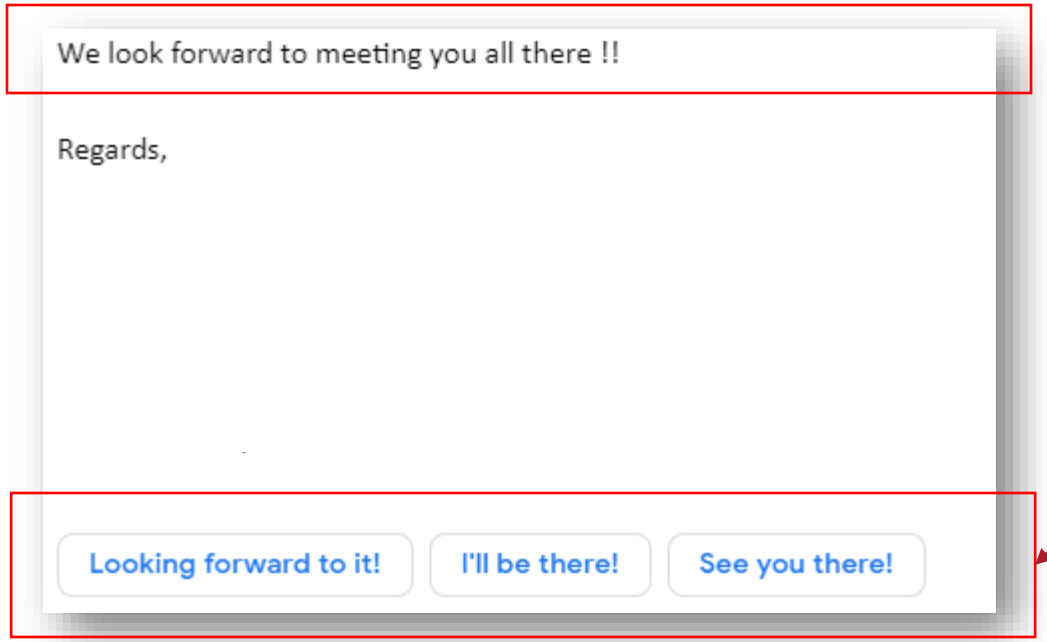
A tennis player in action on the court

Two giraffes standing in a grassy field

A man riding a dirt bike on a dirt track

Example: Generating Responses to Support Fact-checkers

- Text Generation is very popular.



Responses generated by Gmail

Users can select a response and edit it.
- Faster typing
- Increase users' engagement

Learning from Fact-checkers: Analysis and Generation of Fact-checking Language, SIGIR 2019

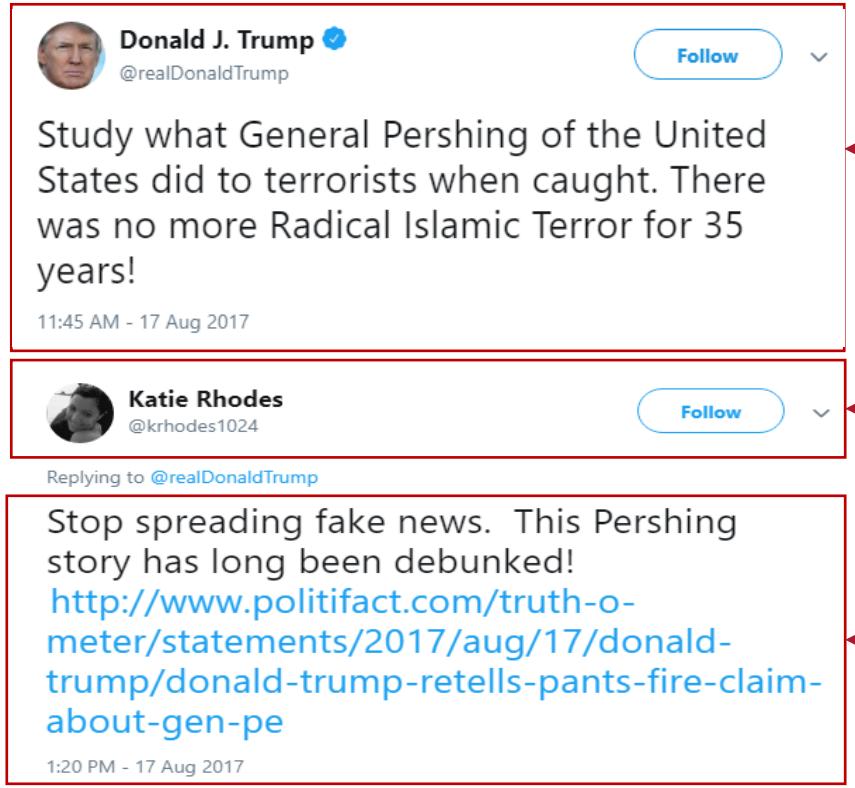
Generating Responses to Support Fact-checkers

- Goal:
  - Given content of an original tweet:
    - Generating responses with **fact-checking intention**
  - Fact-checking intention:
    - Refuting
    - Supporting

Refuting

Generating Responses to Support Fact-checkers

Recap: An example of fact-checking activity



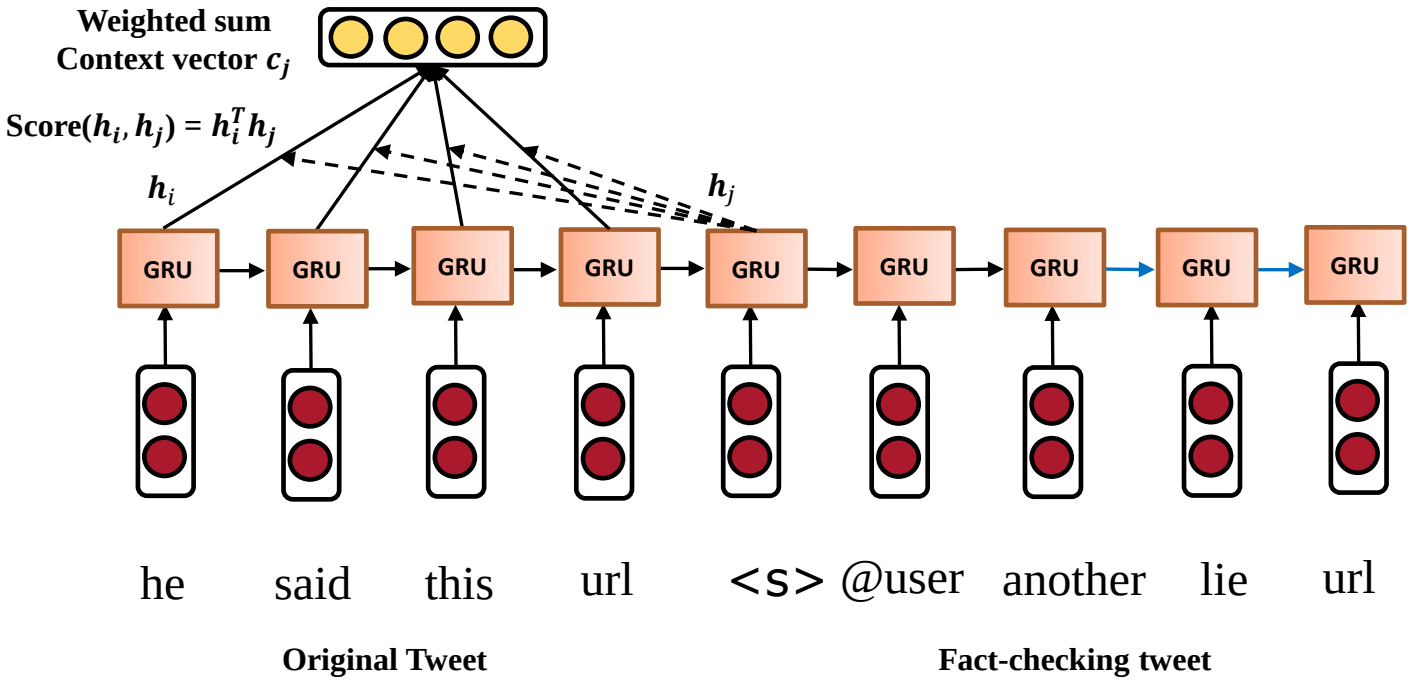
Original tweet

An online fact-checker

A fact-checking tweet (FC-tweet)

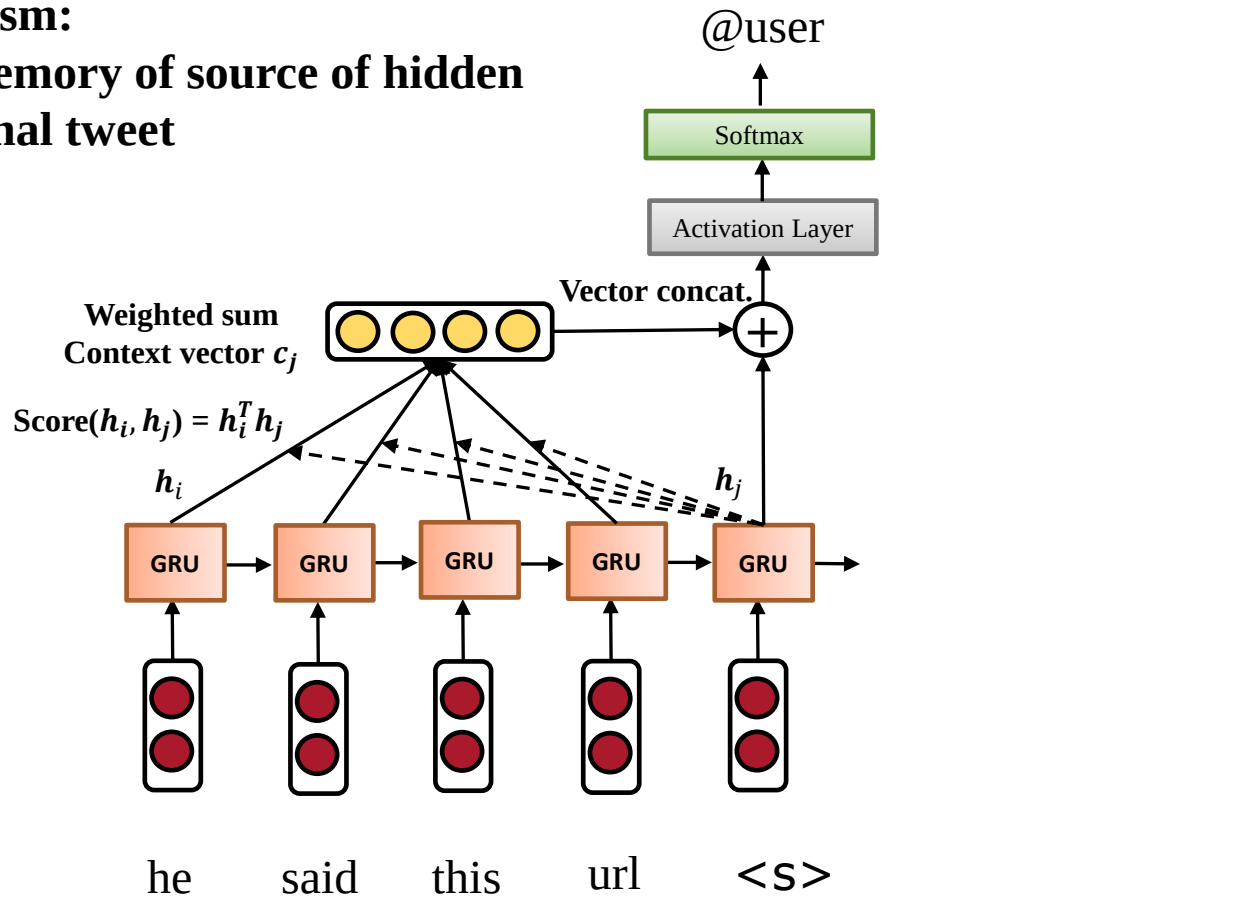
Generating Responses to Support Fact-checkers

- **The framework is based on Sequence to Sequence model with attention mechanism**
  - **Input: Original tweet**
  - **Output: Fact-checking tweet**
- **Attention mechanism:**
  - **Maintain a memory of source of hidden states from original tweet**



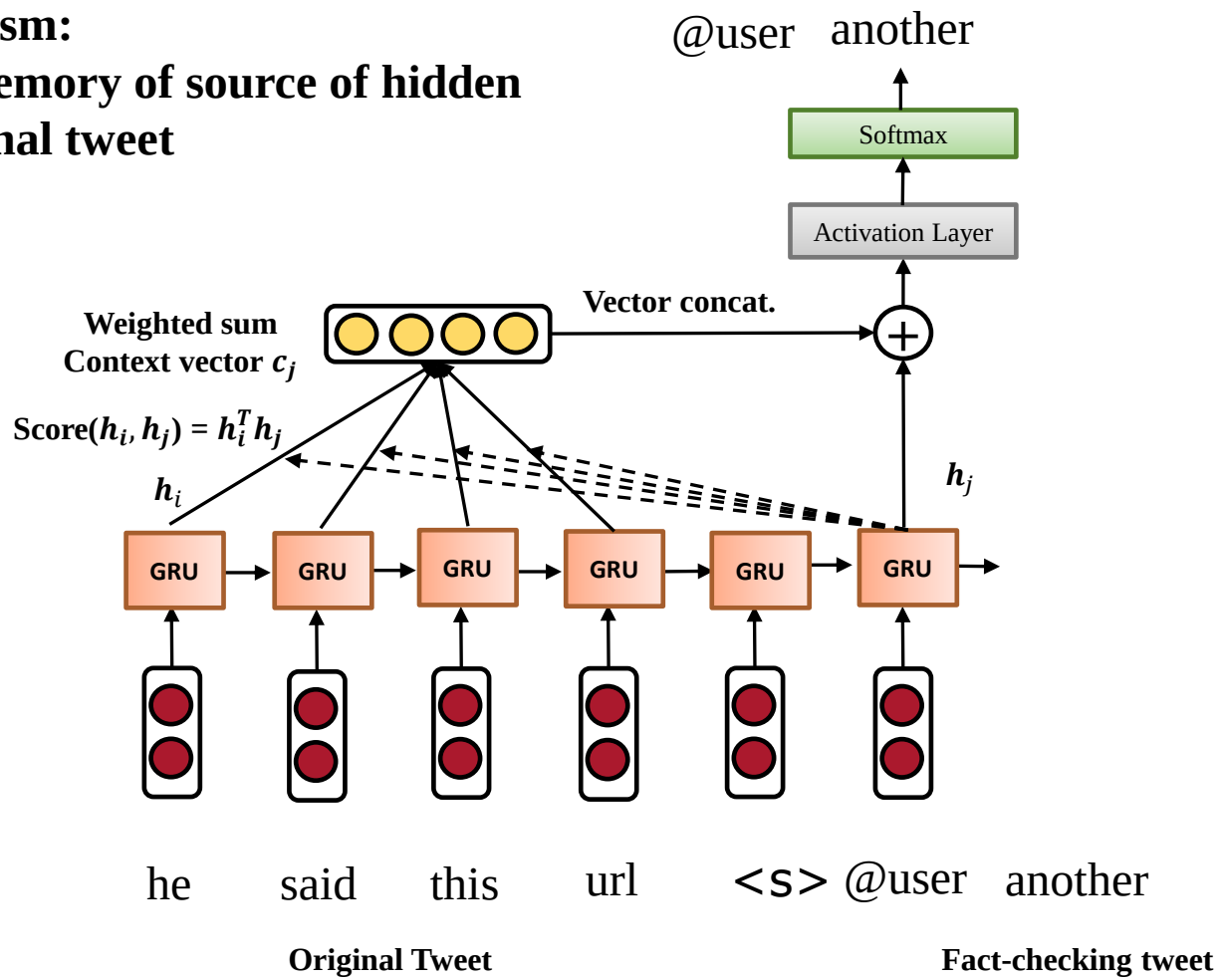
Generating Responses to Support Fact-checkers

- **Attention mechanism:**
  - **Maintain a memory of source of hidden states in original tweet**



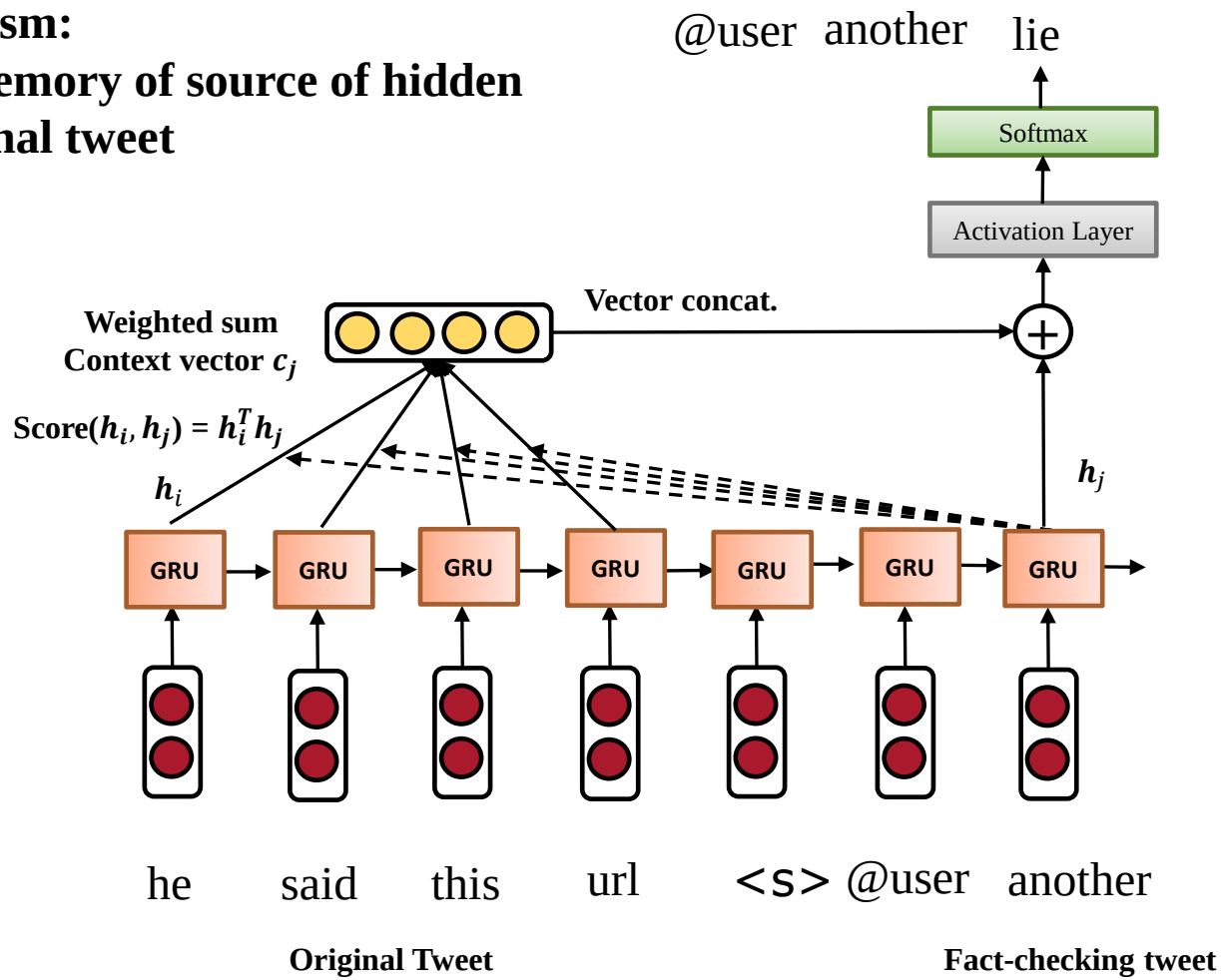
Generating Responses to Support Fact-checkers

- **Attention mechanism:**
  - **Maintain a memory of source of hidden states in original tweet**



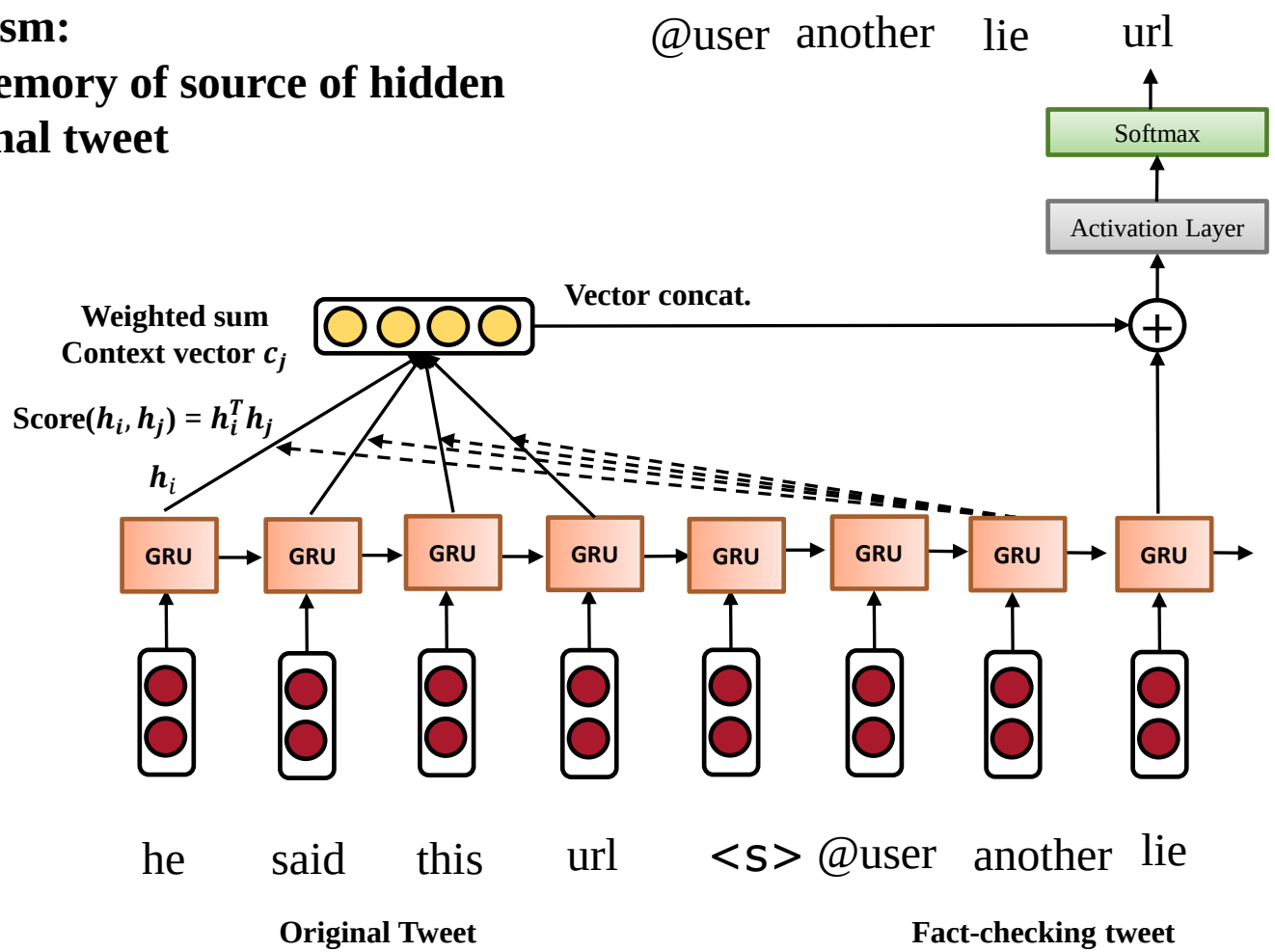
Generating Responses to Support Fact-checkers

- **Attention mechanism:**
  - **Maintain a memory of source of hidden states in original tweet**



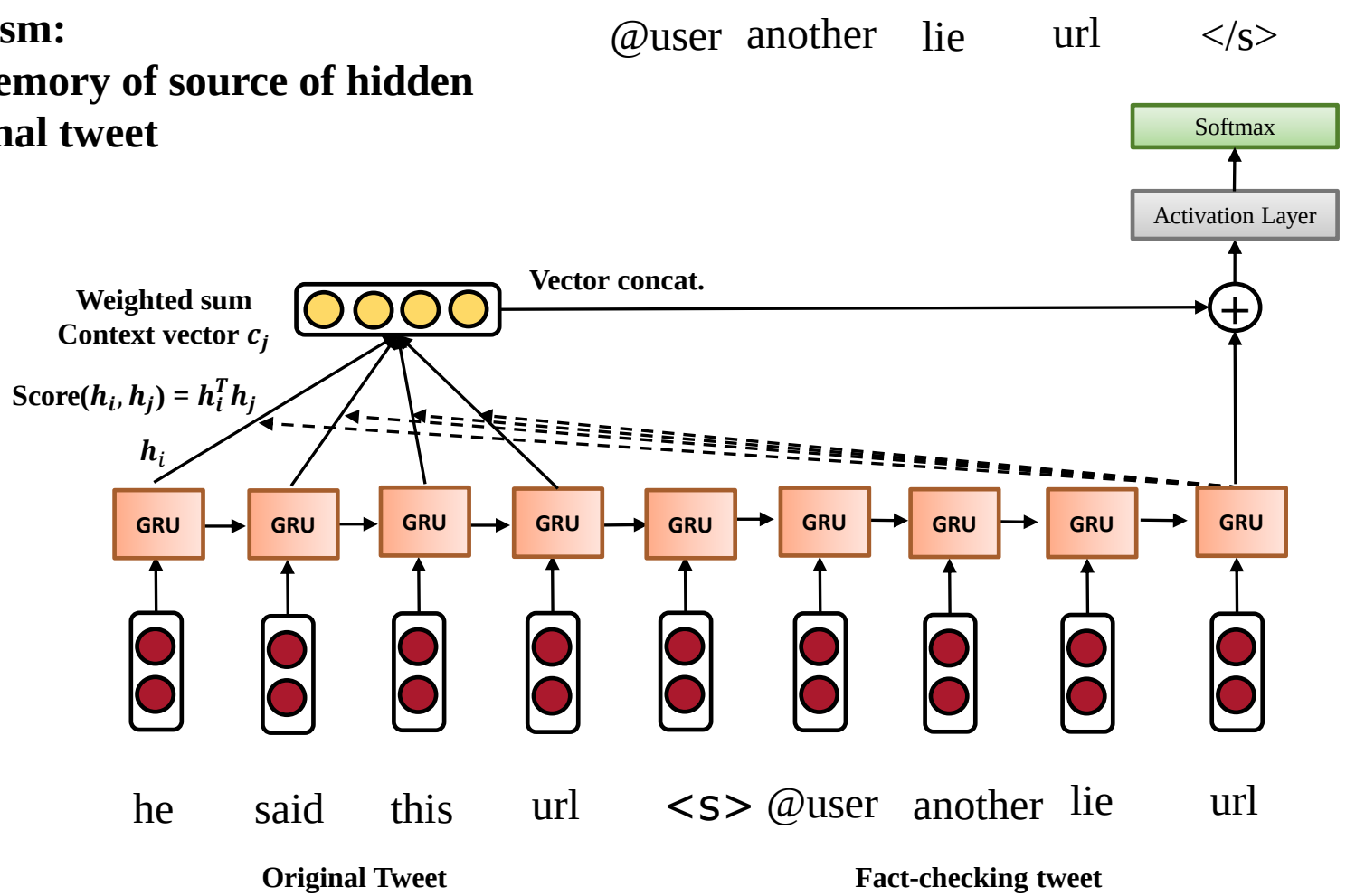
Generating Responses to Support Fact-checkers

- **Attention mechanism:**
  - **Maintain a memory of source of hidden states in original tweet**

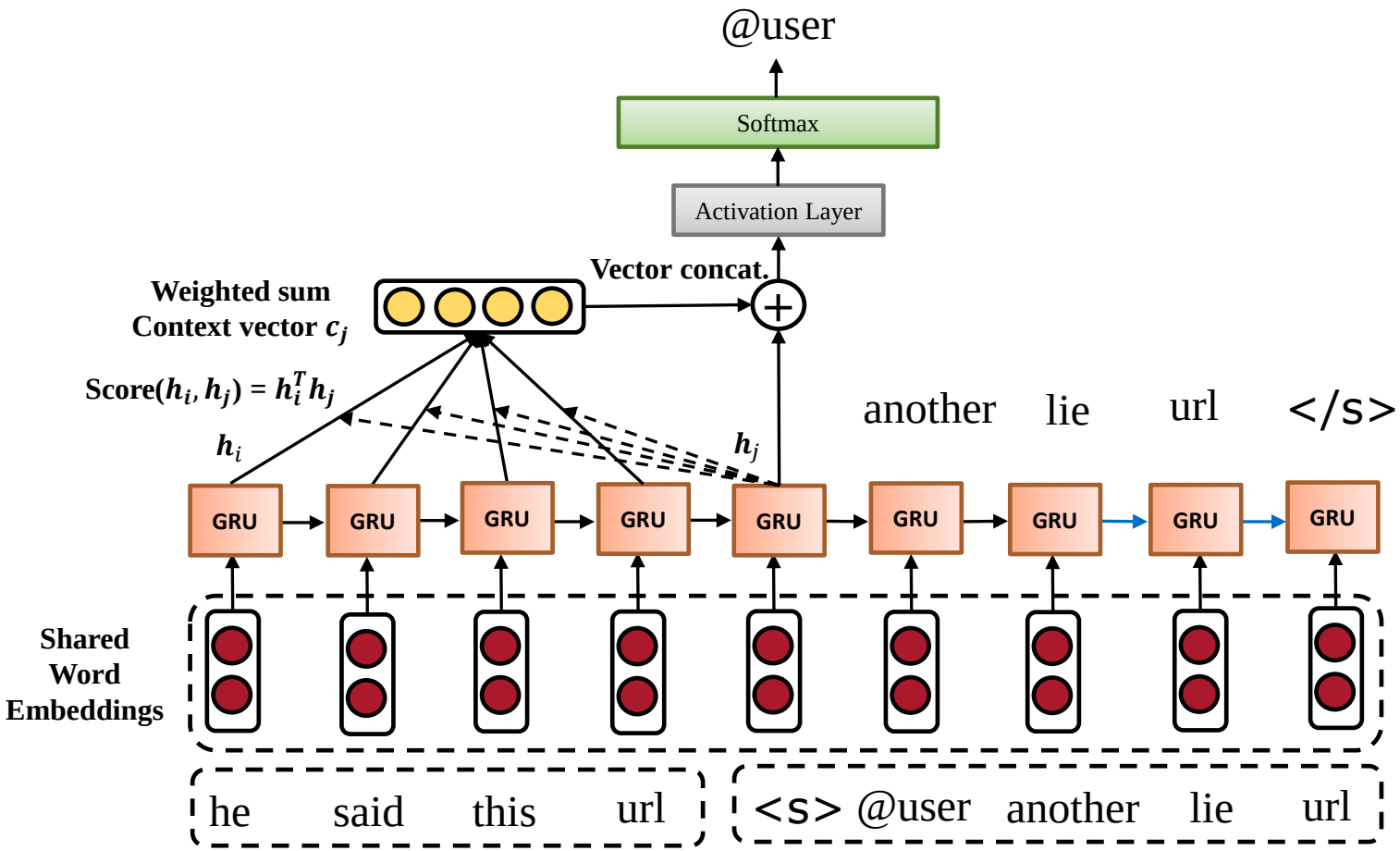


Generating Responses to Support Fact-checkers

- **Attention mechanism:**
  - **Maintain a memory of source of hidden states in original tweet**



Generating Responses to Support Fact-checkers



$$\min L = \min_{\theta} -\log\big(p(y|x)\big) = -\sum_{j=1}^{M} log\ p\big(y_j|y_{j-1}y_{j-2}\ \ldots, y_1, x\big)$$

What's next?

- Study Deep Learning
  - Deep Learning by Yoshua Bengio, Ian Goodfellow, and Aaron Courville. (Available online)
  - Deep Learning with Python by Francois Chollet
  - http://neuralnetworksanddeeplearning.com/index.html
  - https://keras.io/examples/
  - https://www.tensorflow.org/tutorials
  - https://pytorch.org/tutorials/

- Take Deep Learning course
  - CS 541. DEEP LEARNING
  - Online course: https://www.coursera.org/specializations/deep-learning