Gated Recurrent Models

Stephan Gouws & Richard Klein

Outline

Part 1: Intuition, Inference and Training

- Building intuitions: From Feedforward to Recurrent Models
- Inference in RNNs: Fprop
- Training in RNNs: Backpropagation-through-time (BPTT)

SHORT BREAK

Outline

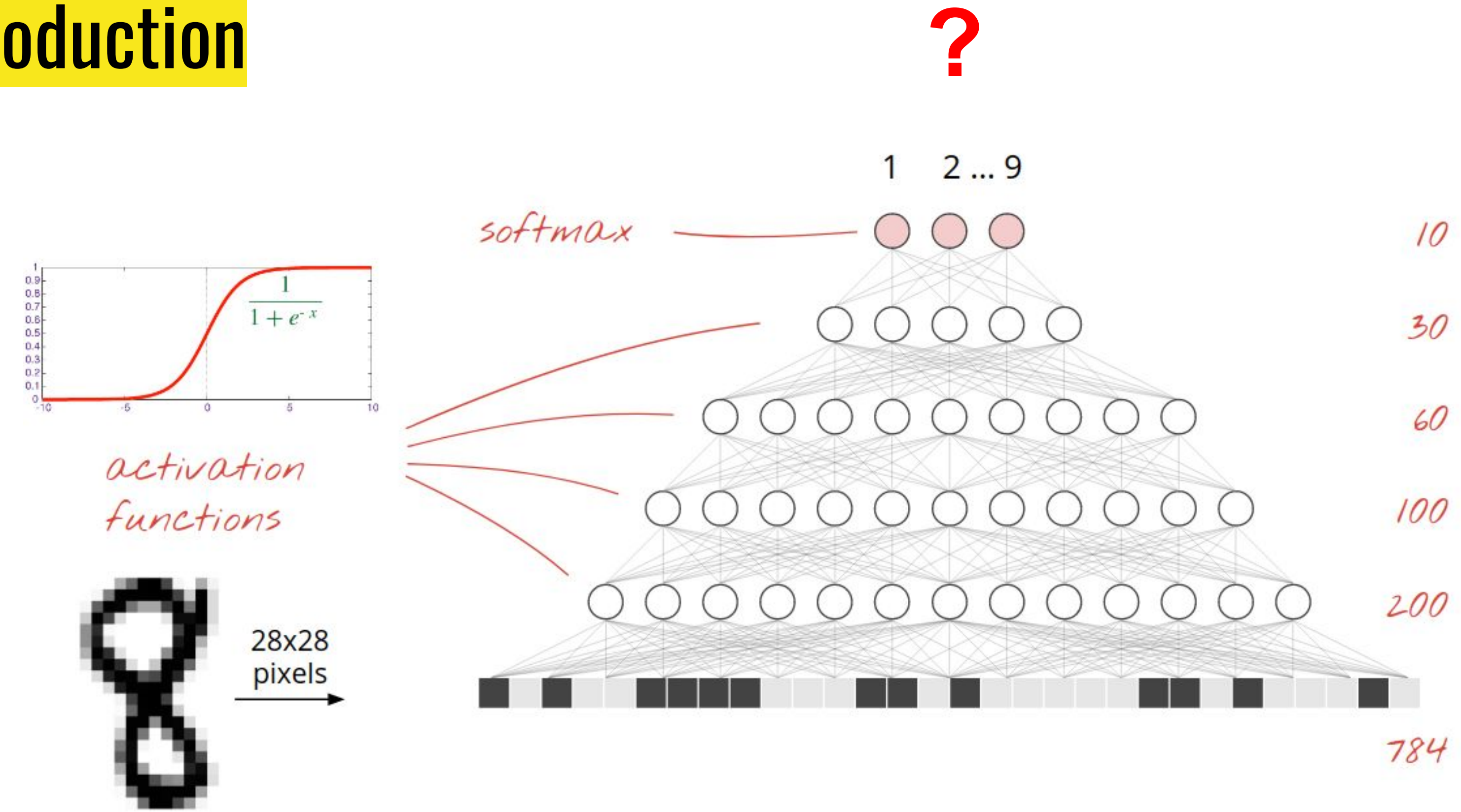
Part 2: Gated models & Applications

- Long Short-term Memory (LSTMs)
- Gated Recurrent Units (GRUs)
- Applications:
  - Image captioning
  - Sequence classification (Practical 4: MNIST)
  - Language modeling
  - Sequence-labeling (lots of NLP tasks, e.g. POS tagging, NER, ...)
  - Sequence-to-sequence learning (Machine translation, Dialogue modeling, ...)

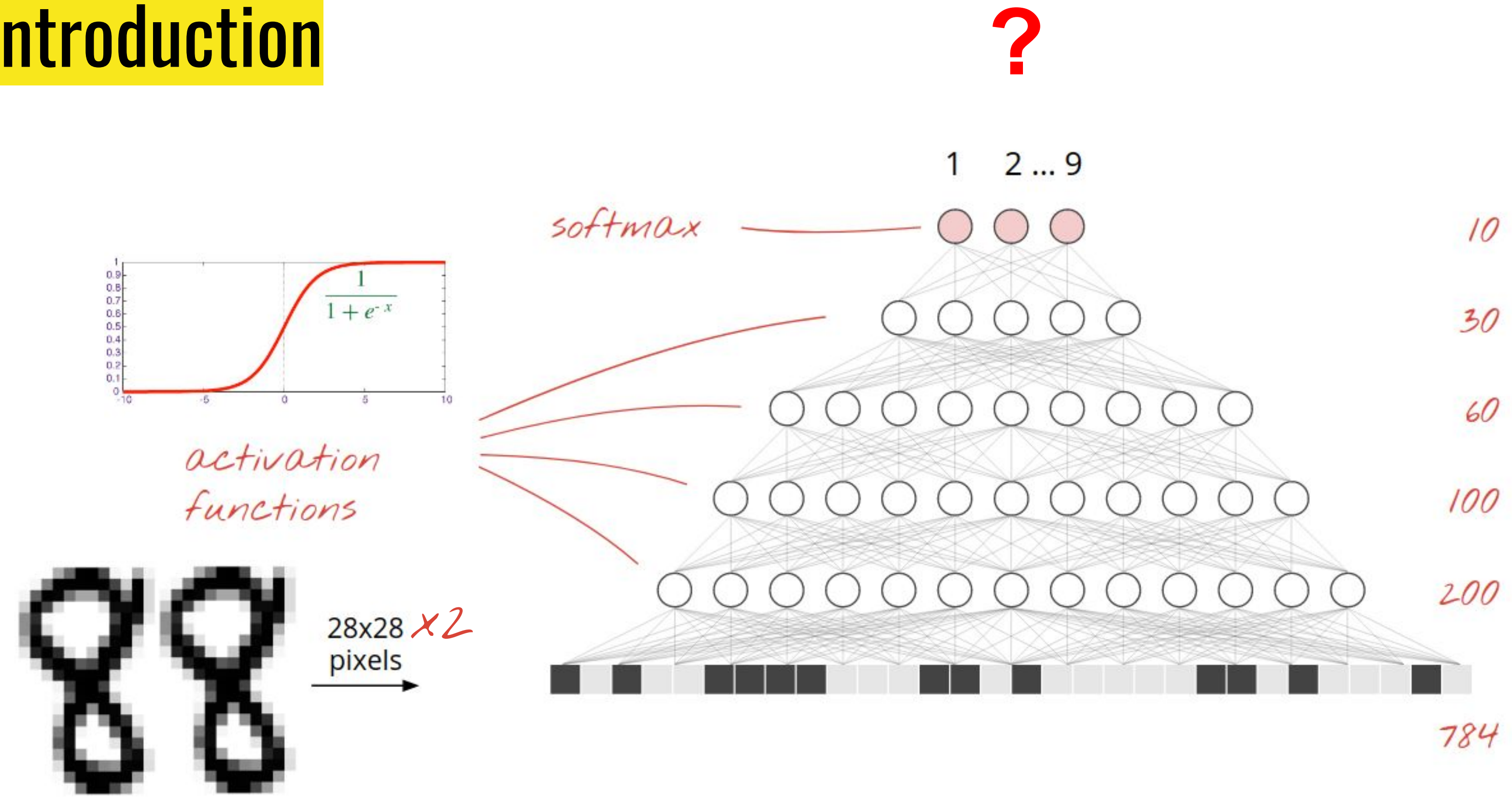
Recurrent Models

PART 1: Intuition, Inference and Training

Introduction

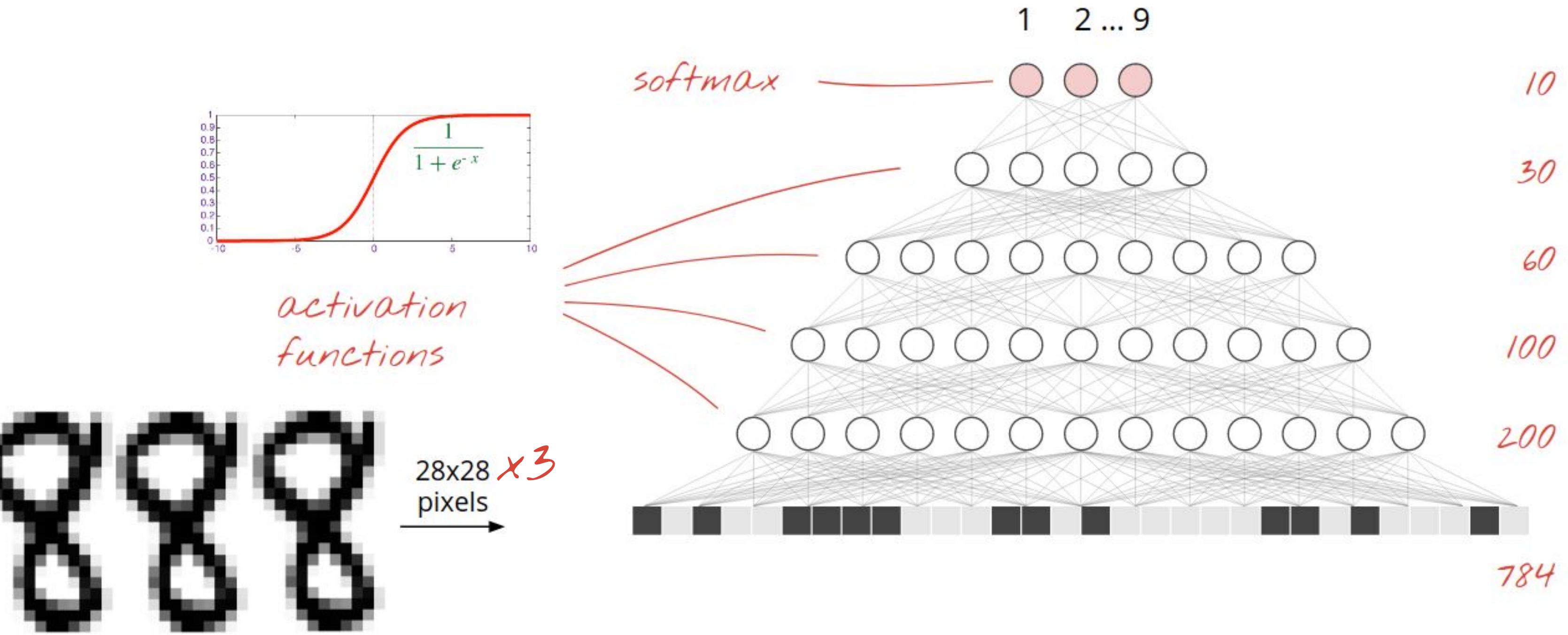


Introduction



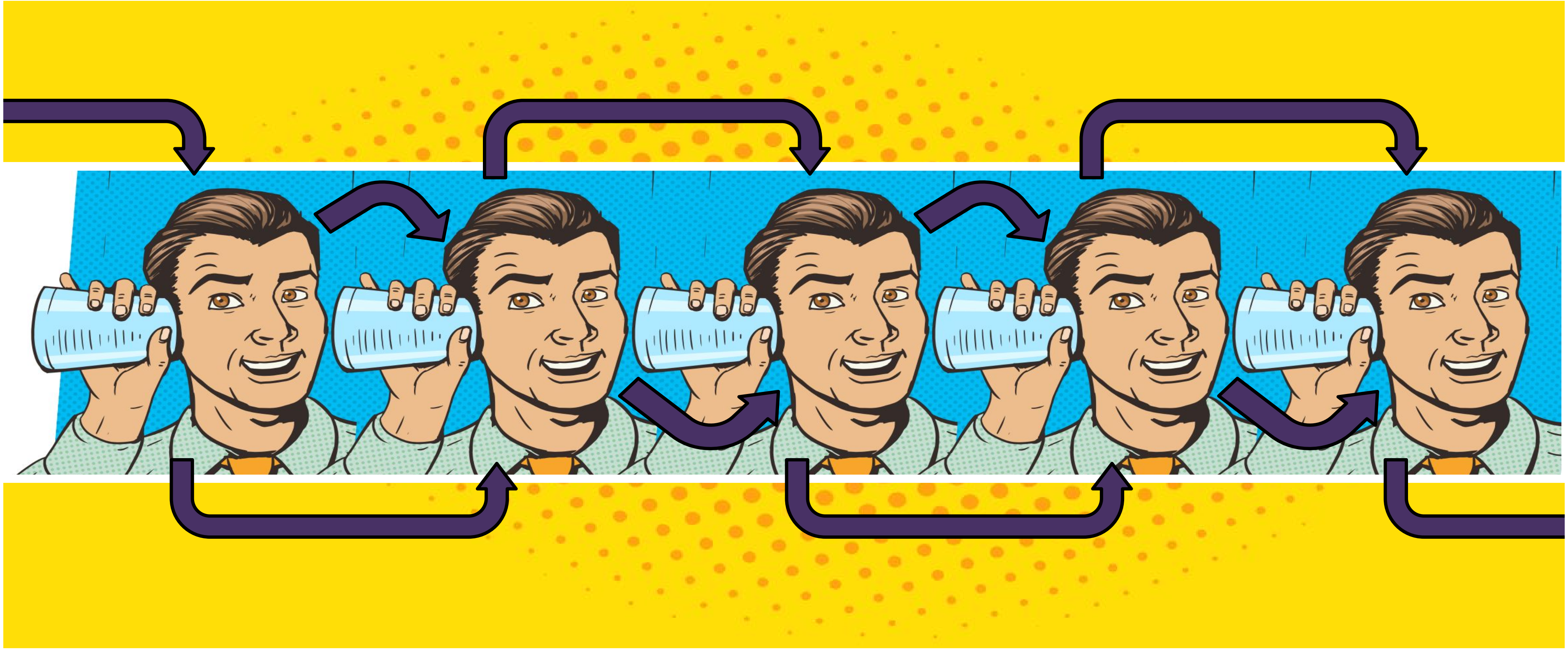
Introduction

?



We need to be able to remember information from previous time steps

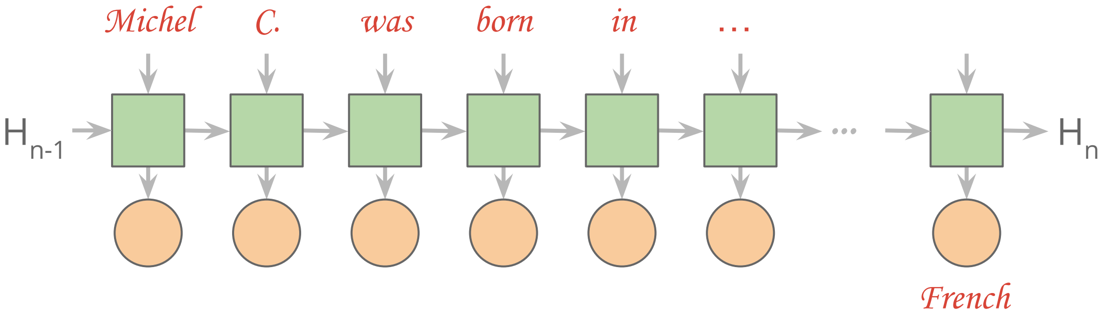
Recurrent neural networks: Intuition



Long-term dependencies: Why do they matter?

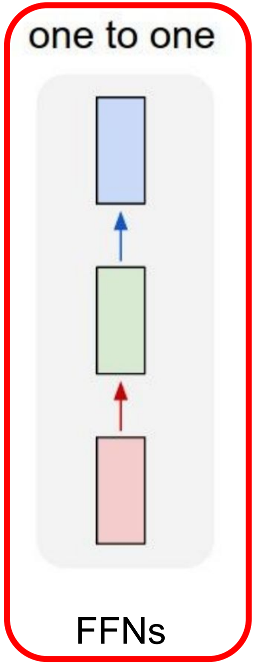
Michel C. was born in Paris, France. He is married and has three children. He received a M.S. in neurosciences from the University Pierre & Marie Curie and the Ecole Normale Supérieure in 1987, and and then spent most of his career in Switzerland, at the Ecole Polytechnique de Lausanne. He specialized in child and adolescent psychiatry and his first field of research was severe mood disorders in adolescent, topic of his PhD in neurosciences (2002). His mother tongue is ? ? ? ? ?

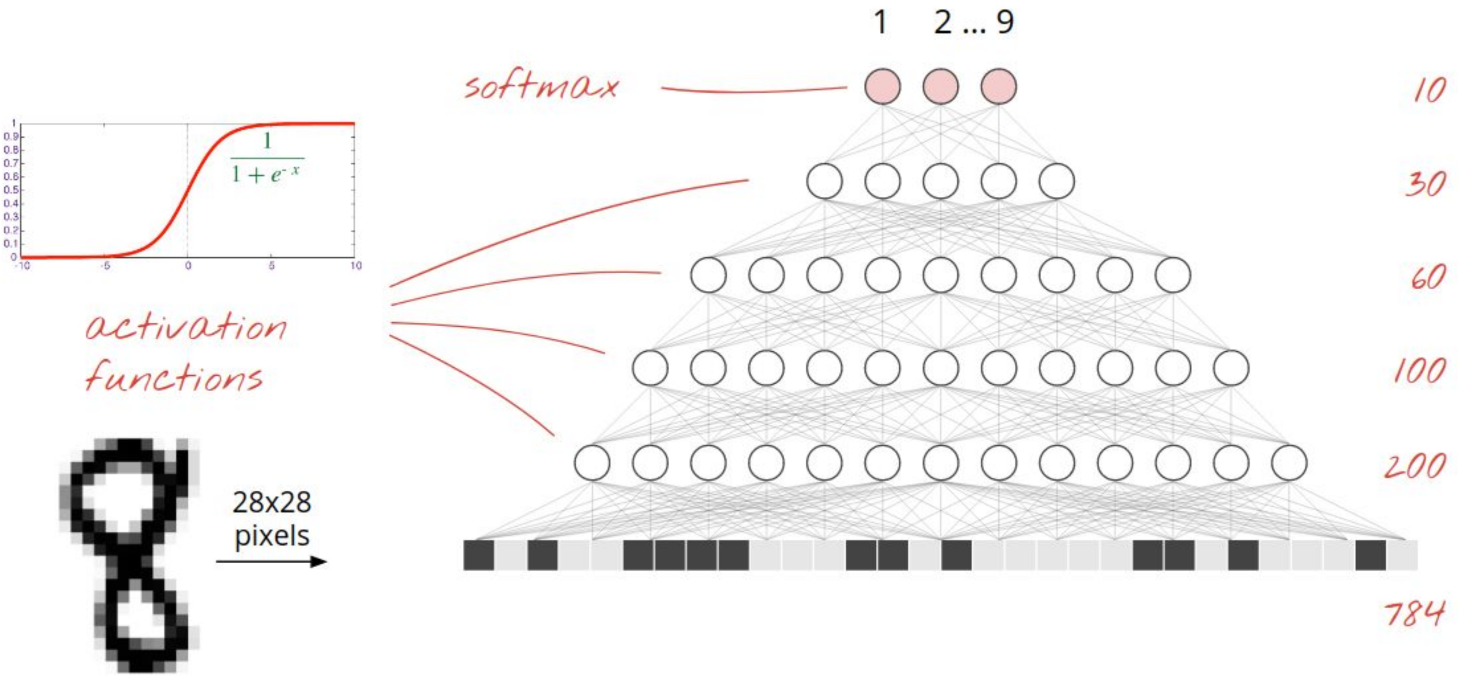
Short context → English, German, Russian, French ...



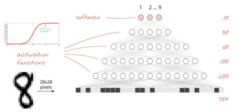
Long context → Problems...

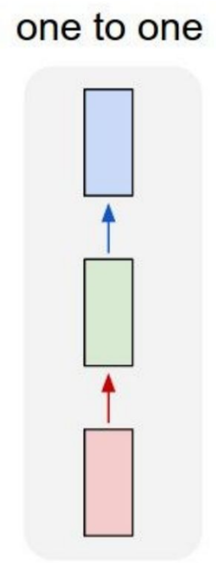
Types of Sequence Models





Types of Sequence Models





FFNs

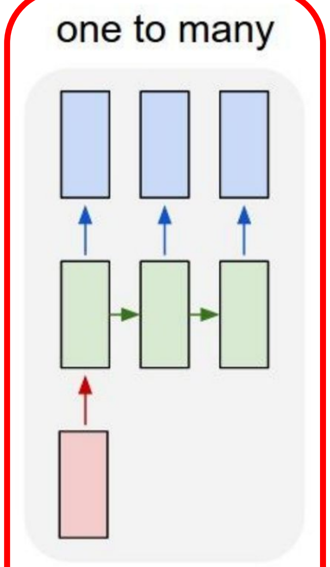
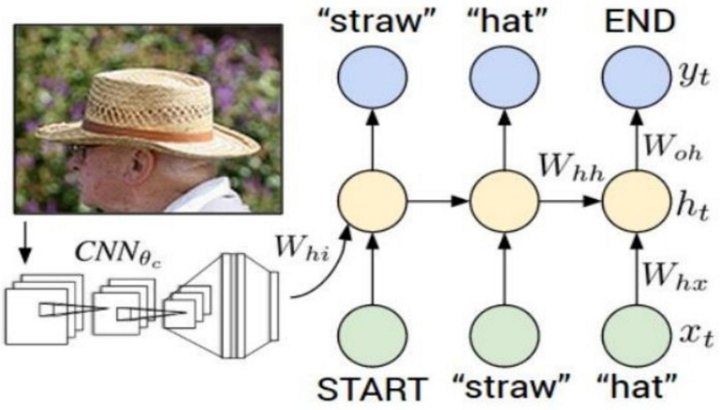
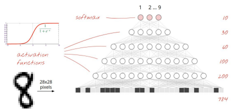
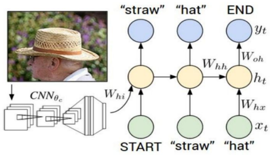


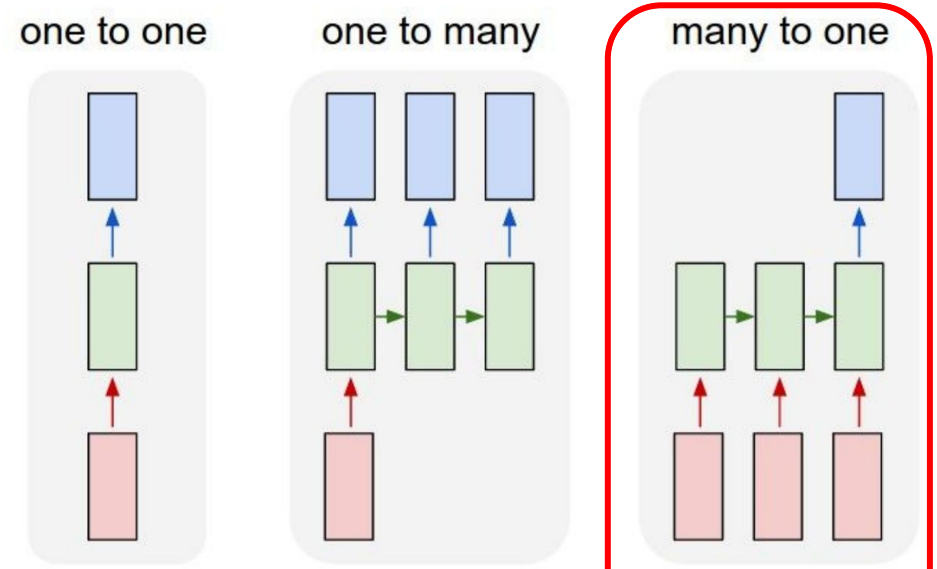
Image Captioning



Types of Sequence Models



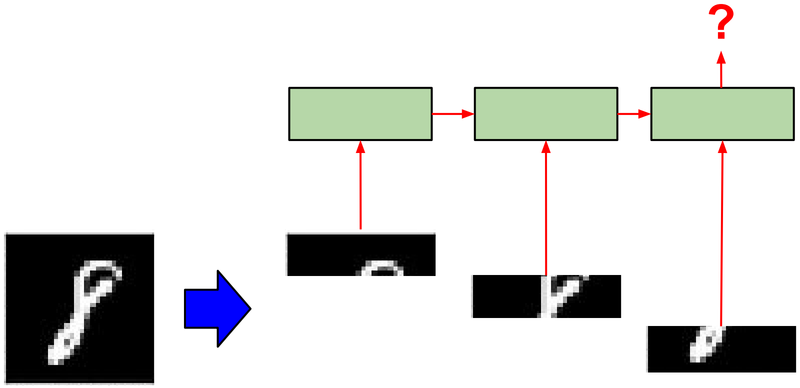




FFNs

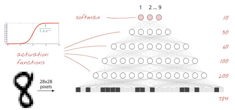
Image Captioning

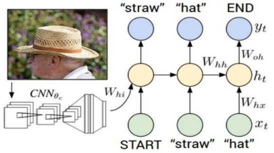
MNIST predictor

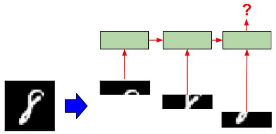


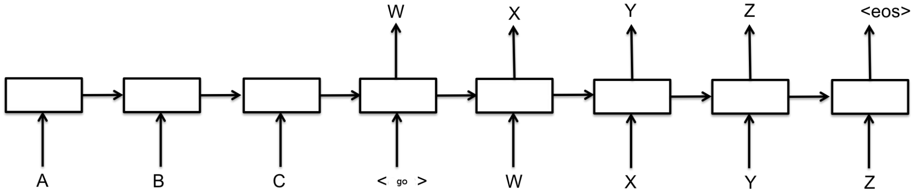
We'll talk about this a little later. We'll also implement this in today's practical!

Types of Sequence Models

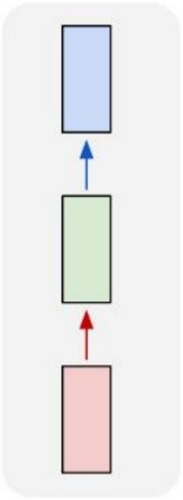








one to one



FFNs

one to many

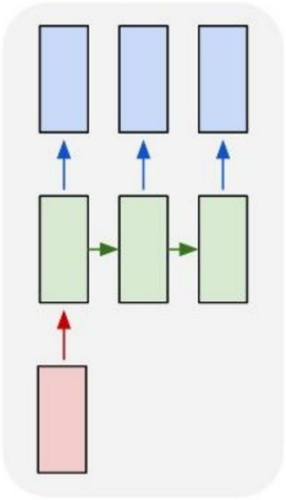
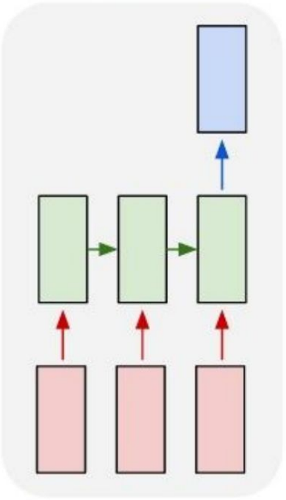


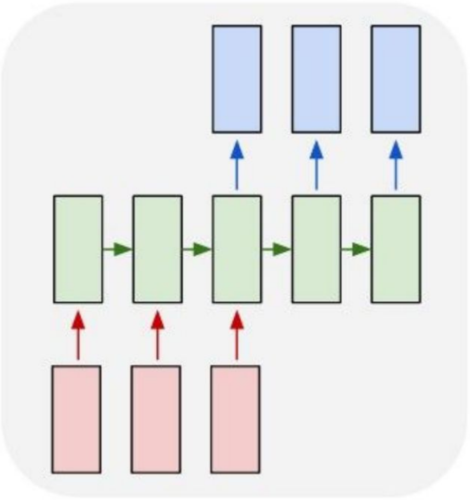
Image Captioning

many to one



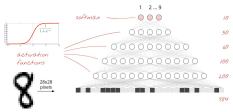
MNIST predictor

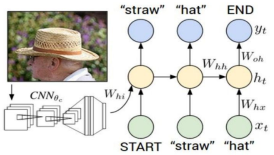
many to many

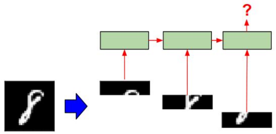


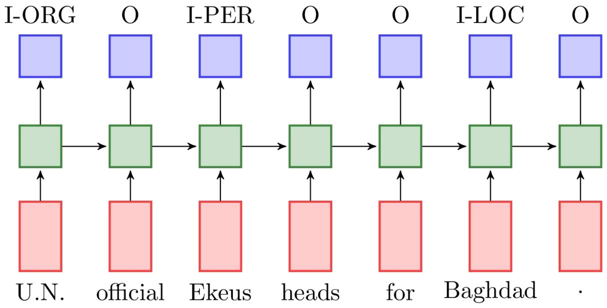
Seq2Seq

Types of Sequence Models

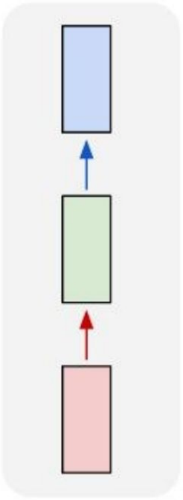








one to one



FFNs

one to many

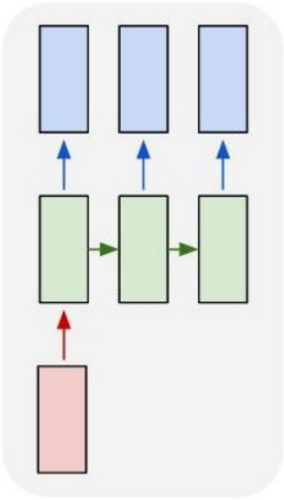
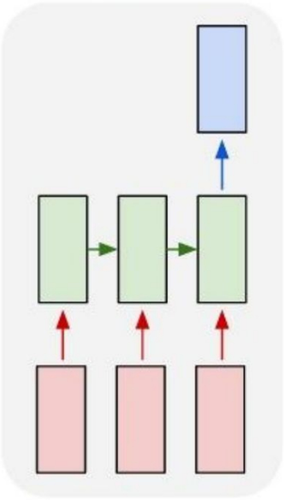


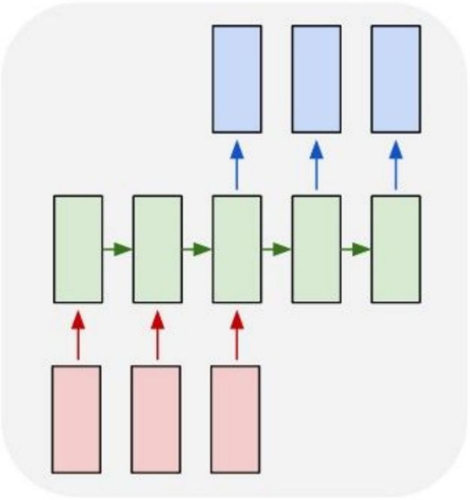
Image Captioning

many to one



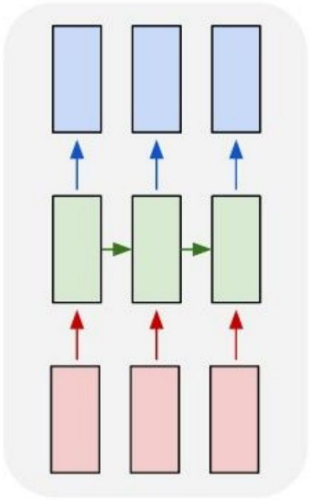
MNIST predictor

many to many



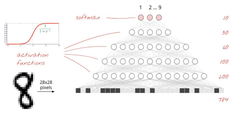
Seq2Seq

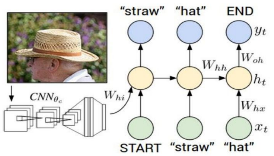
many to many

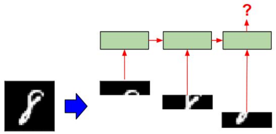


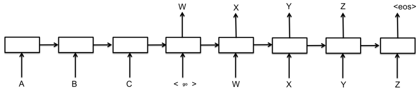
Sequence labeling (e.g. NER)

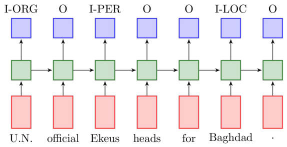
Types of Sequence Models

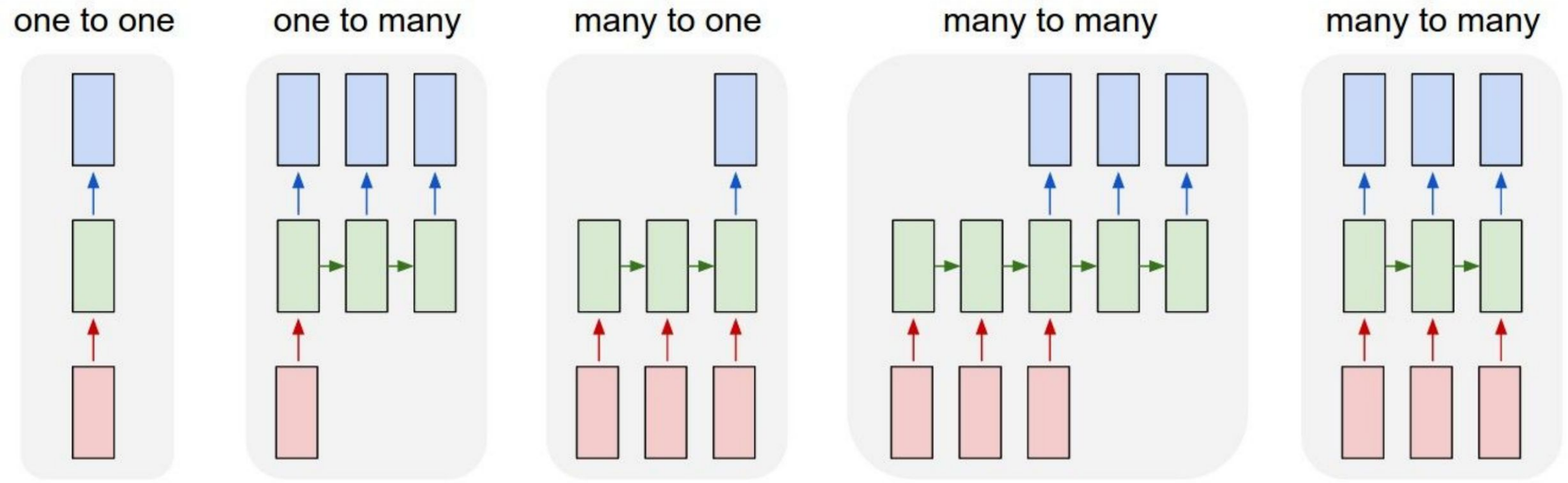












FFNs

Image Captioning

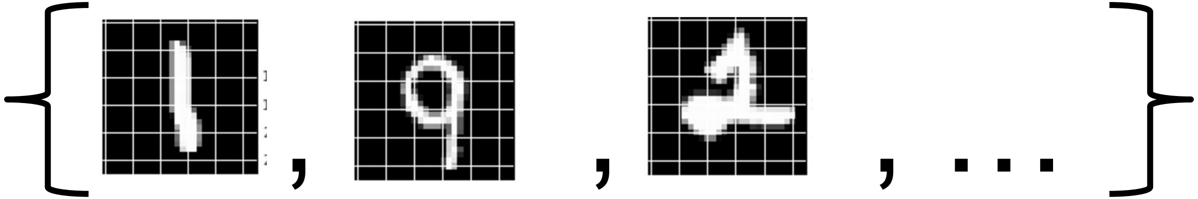
MNIST predictor

Seq2Seq

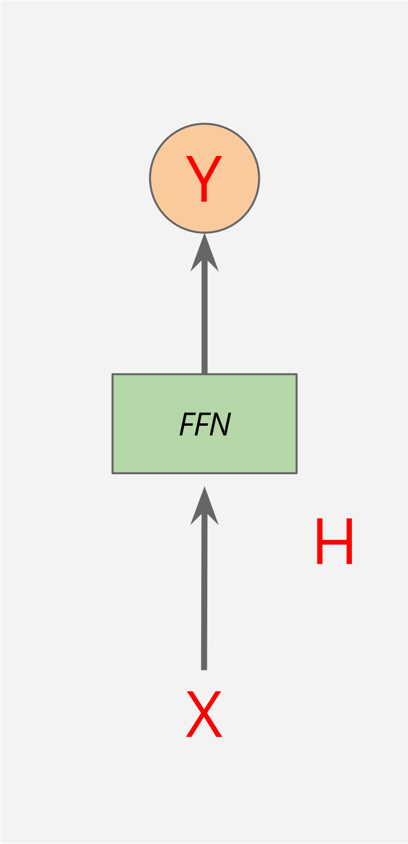
Sequence labeling (e.g. NER)

FFNs vs RNNs

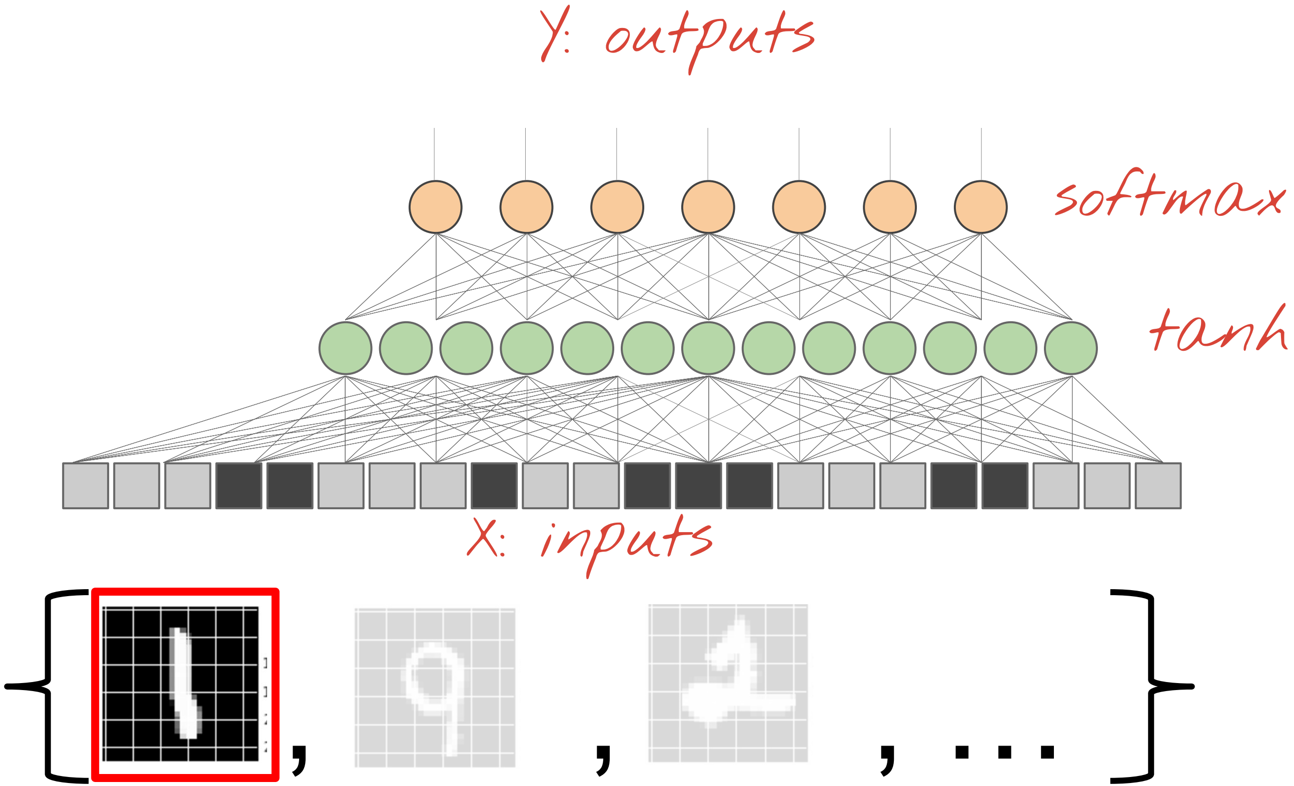
Classify following examples:



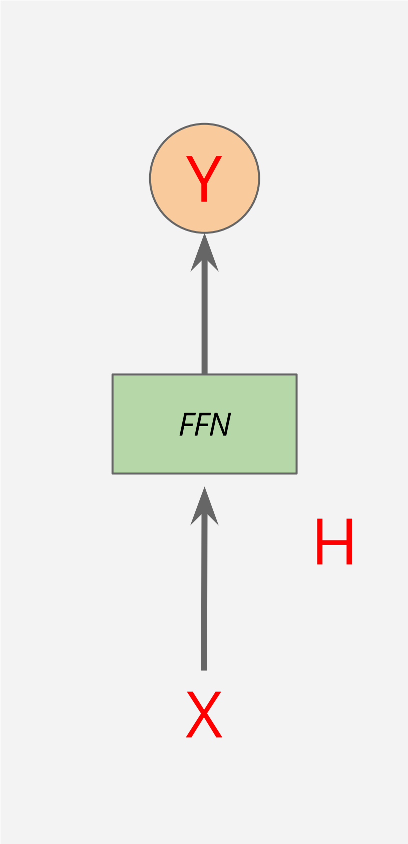
FFNs vs RNNs



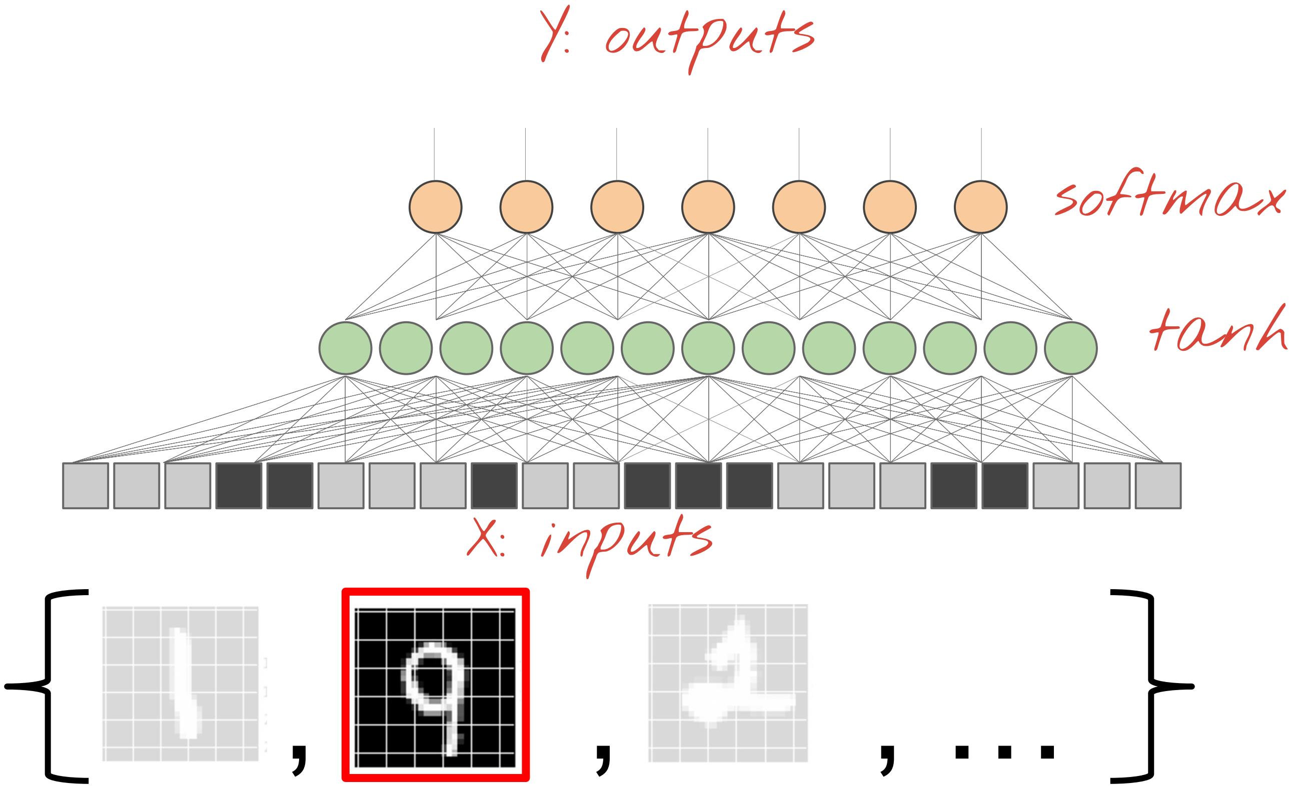
PREDICT: 1



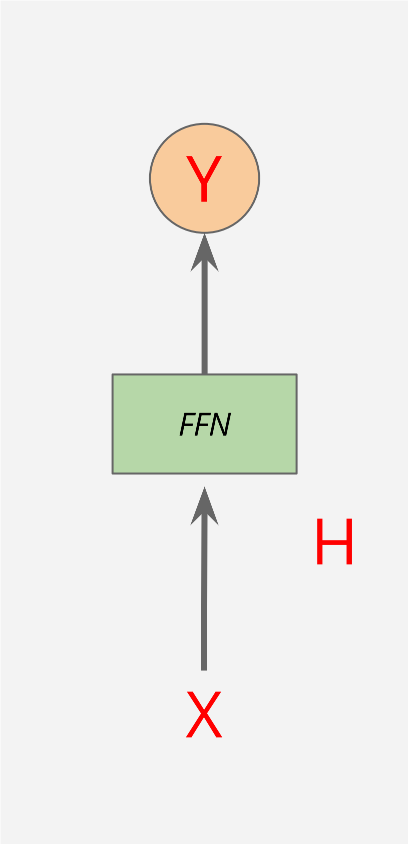
FFNs vs RNNs



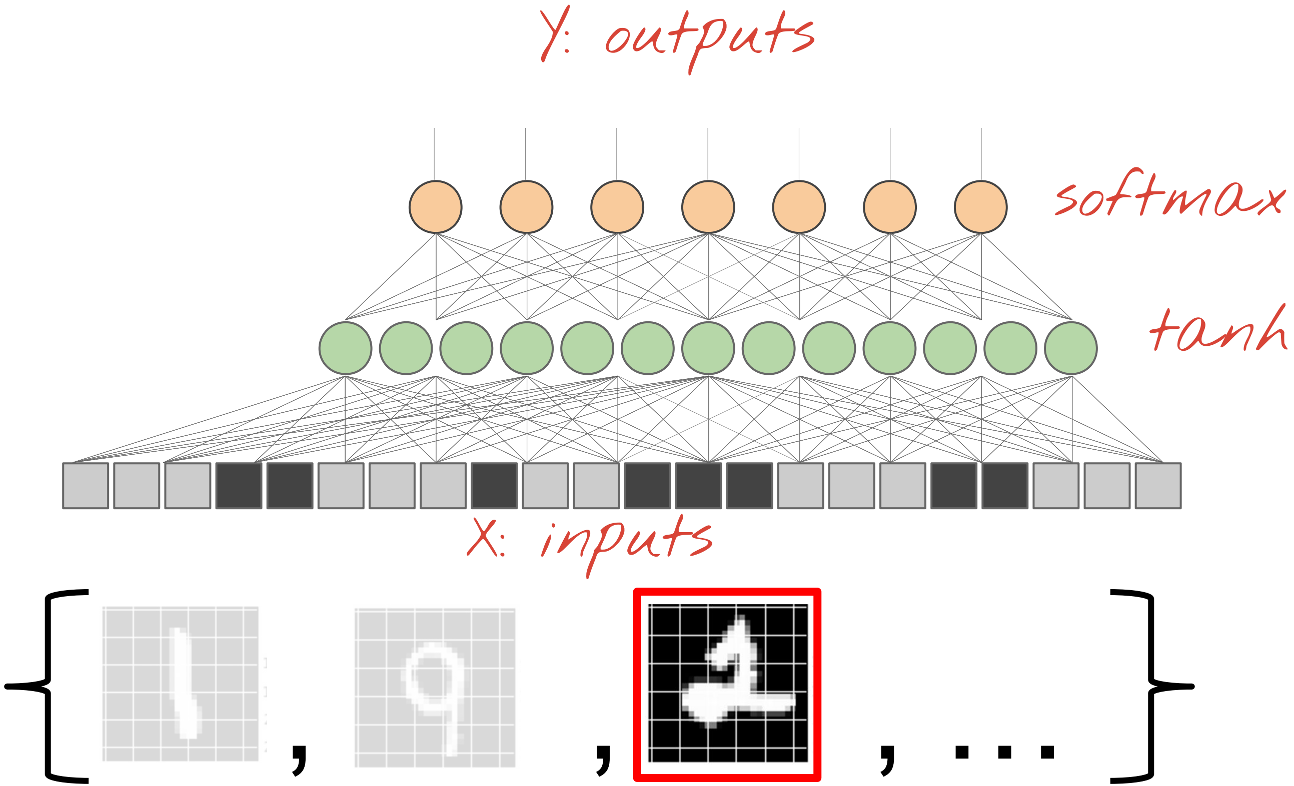
PREDICT: 9



FFNs vs RNNs

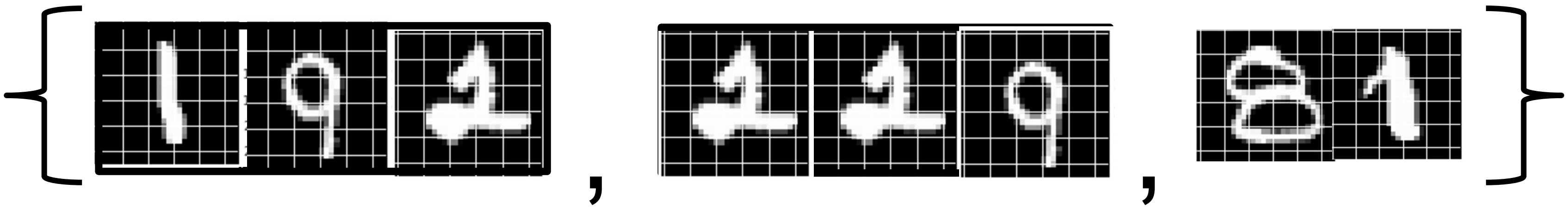


PREDICT: 2



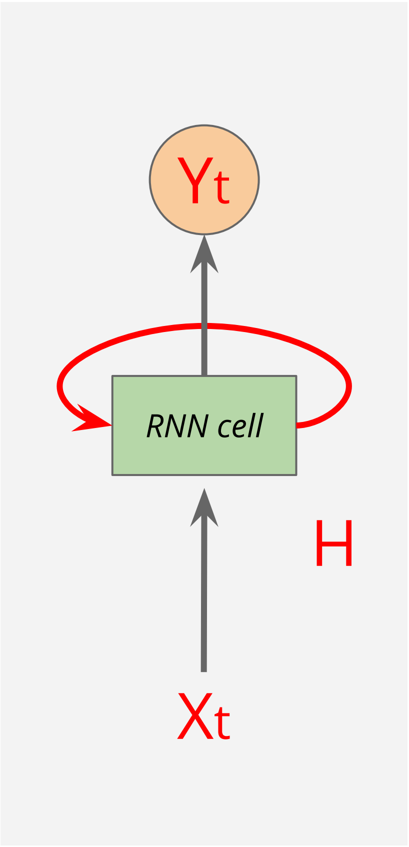
FFNs vs RNNs

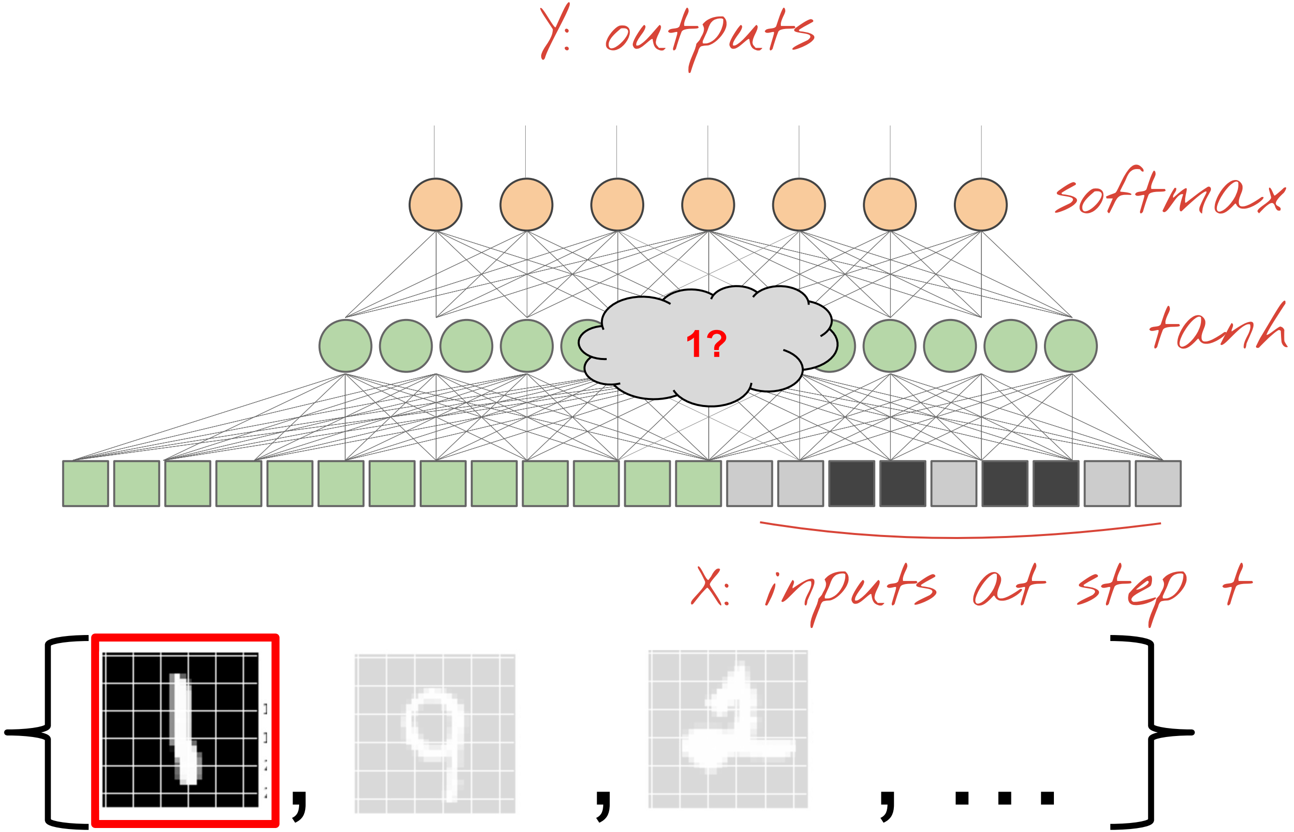
But what if these were not digits, but longer numbers?



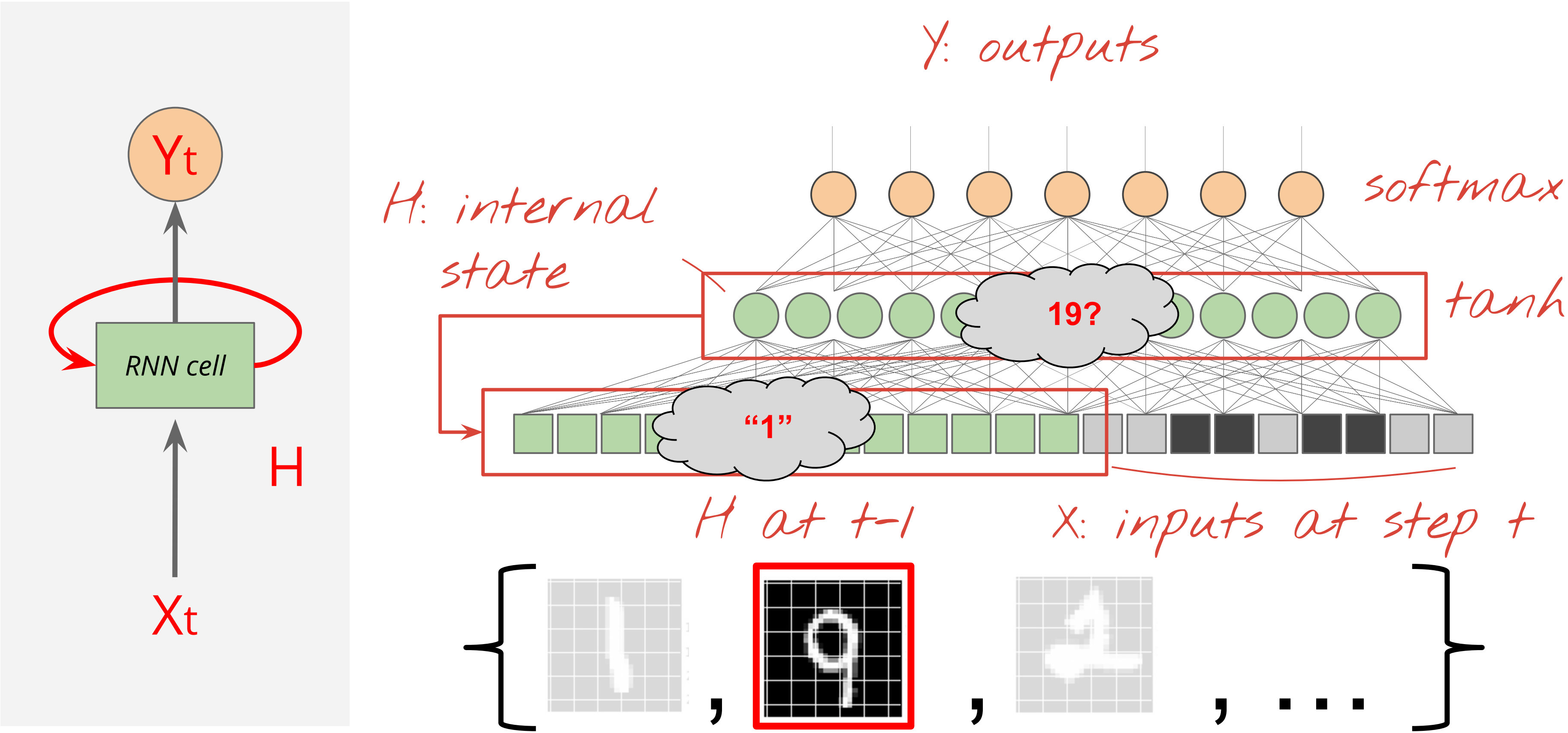
Problem? Variable length inputs.

FFNs vs RNNs



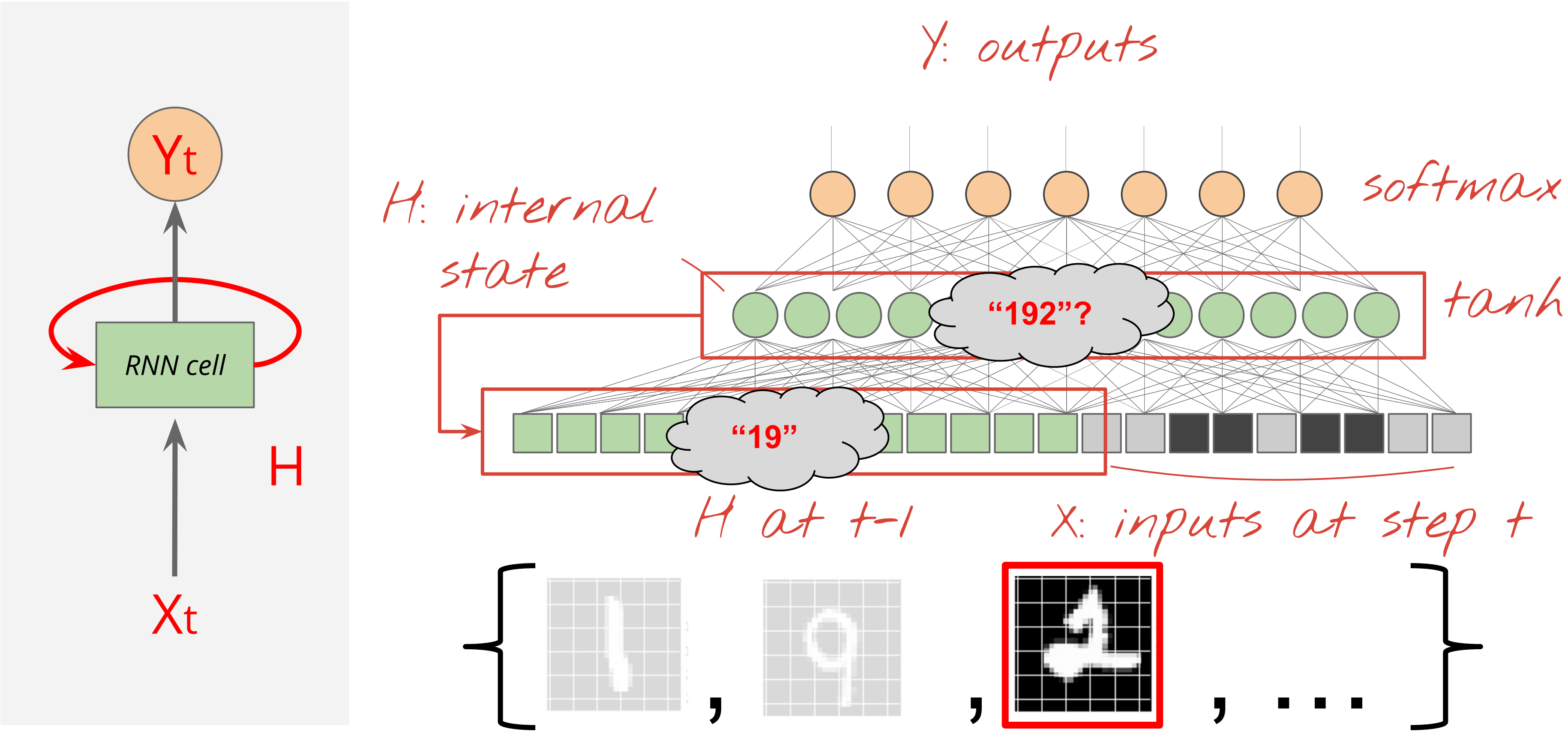


FFNs vs RNNs

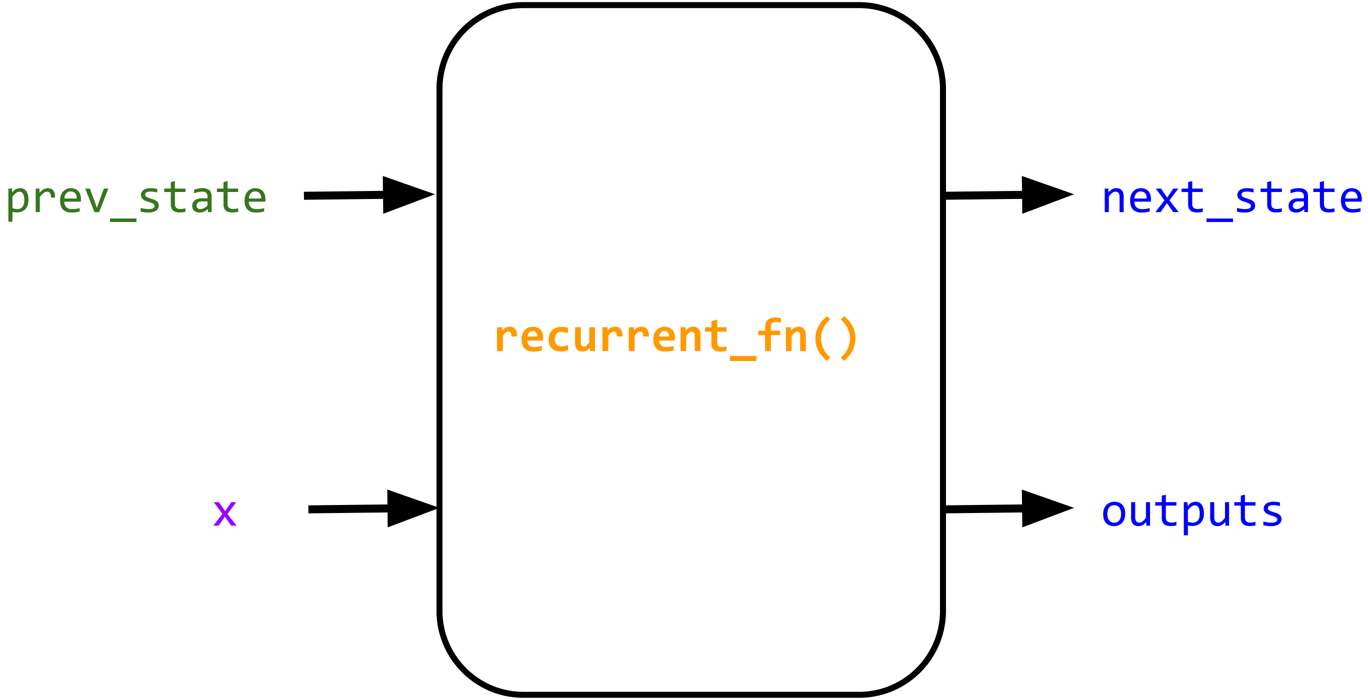

Yt
RNN cell
H
Xt
Y: outputs
softmax
H: internal state
tanh
19?
"1"
H at t-1
X: inputs at step t

FFNs vs RNNs

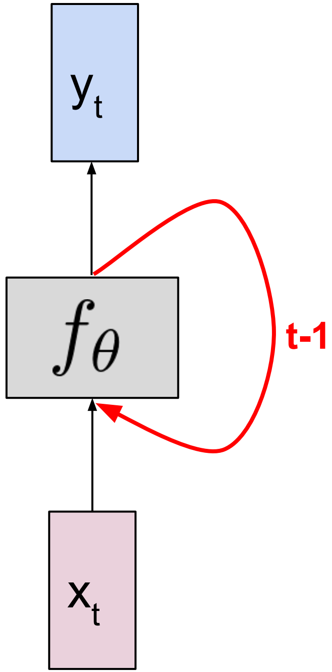
Maintains a state (memory) that carries information between inputs!



The RNN API

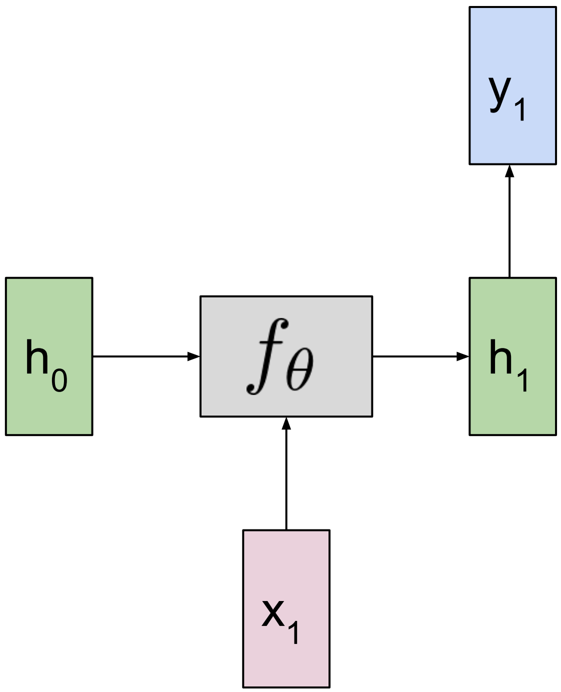


The RNN Computation Graph

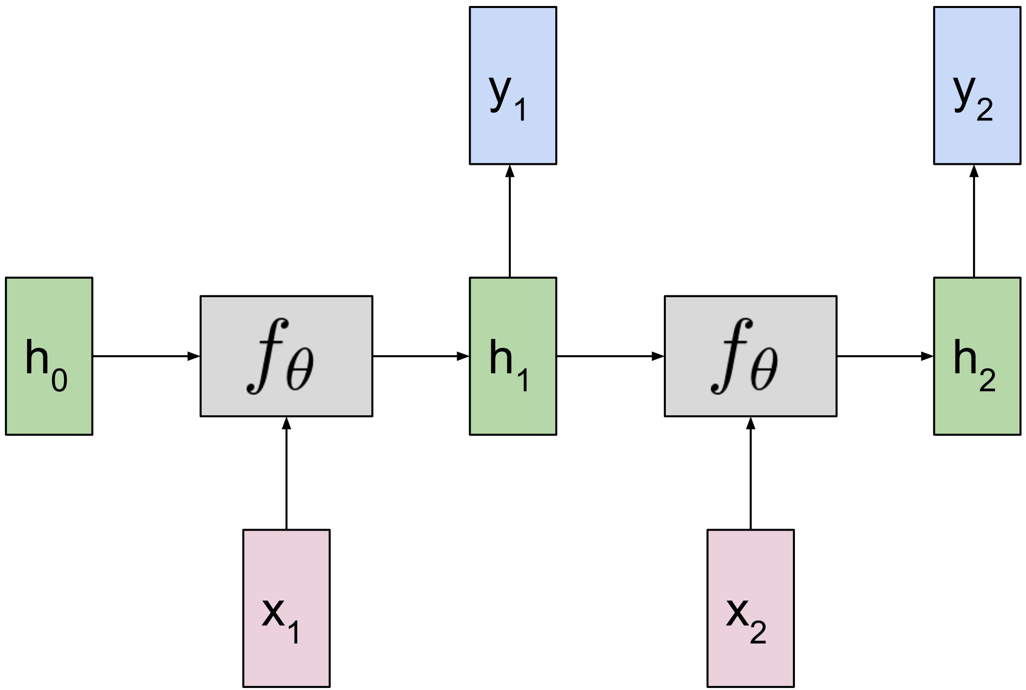


"Feedback loop" / state / memory / stack
(previous time-step)

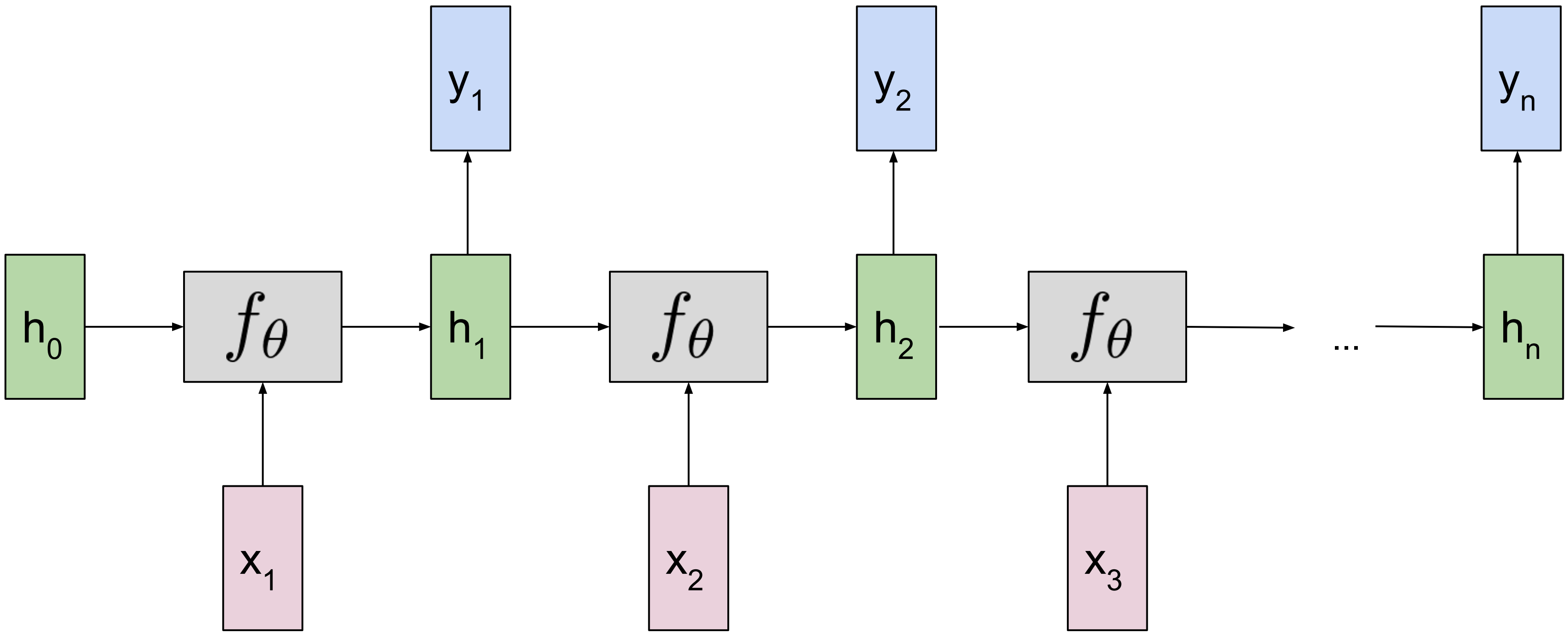
"Unrolling" the RNN Computation Graph



Unrolling the RNN Computation Graph

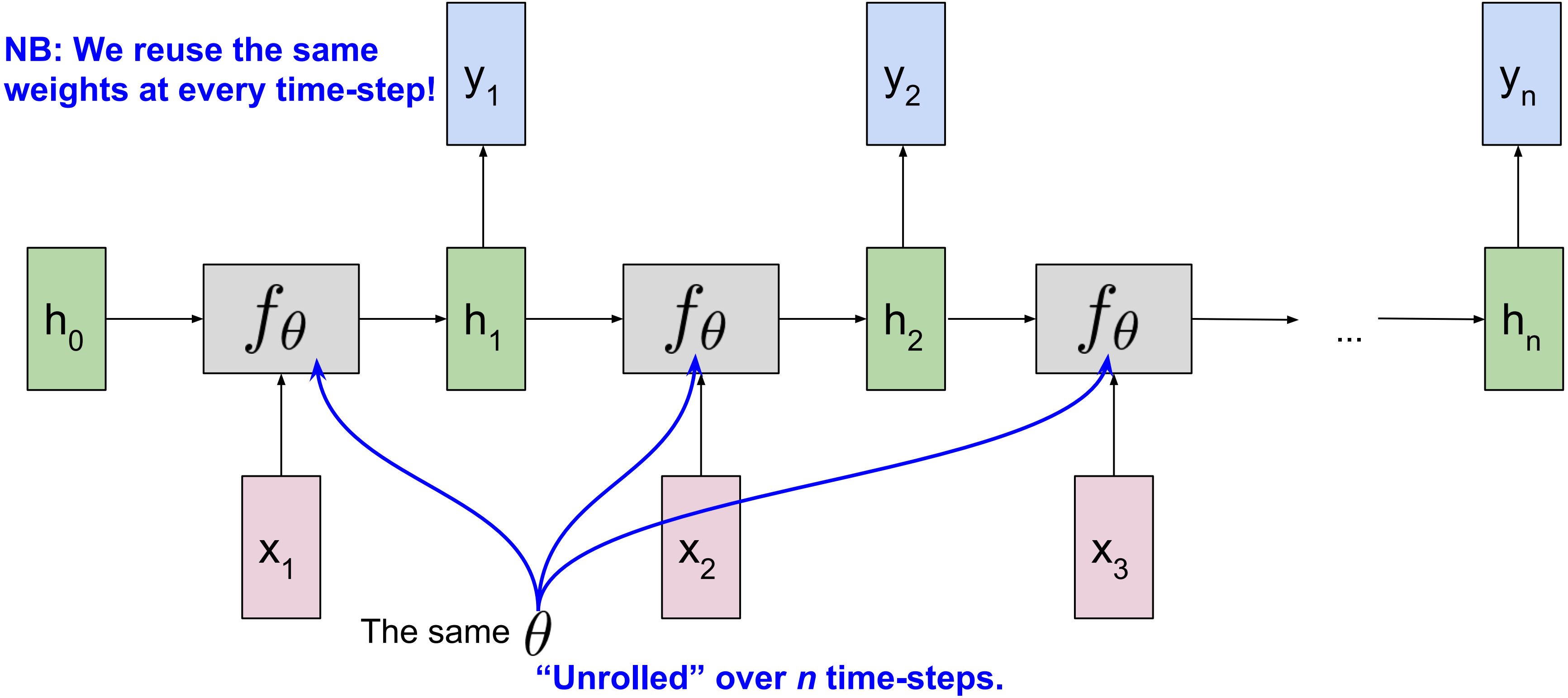


Unrolling the RNN Computation Graph

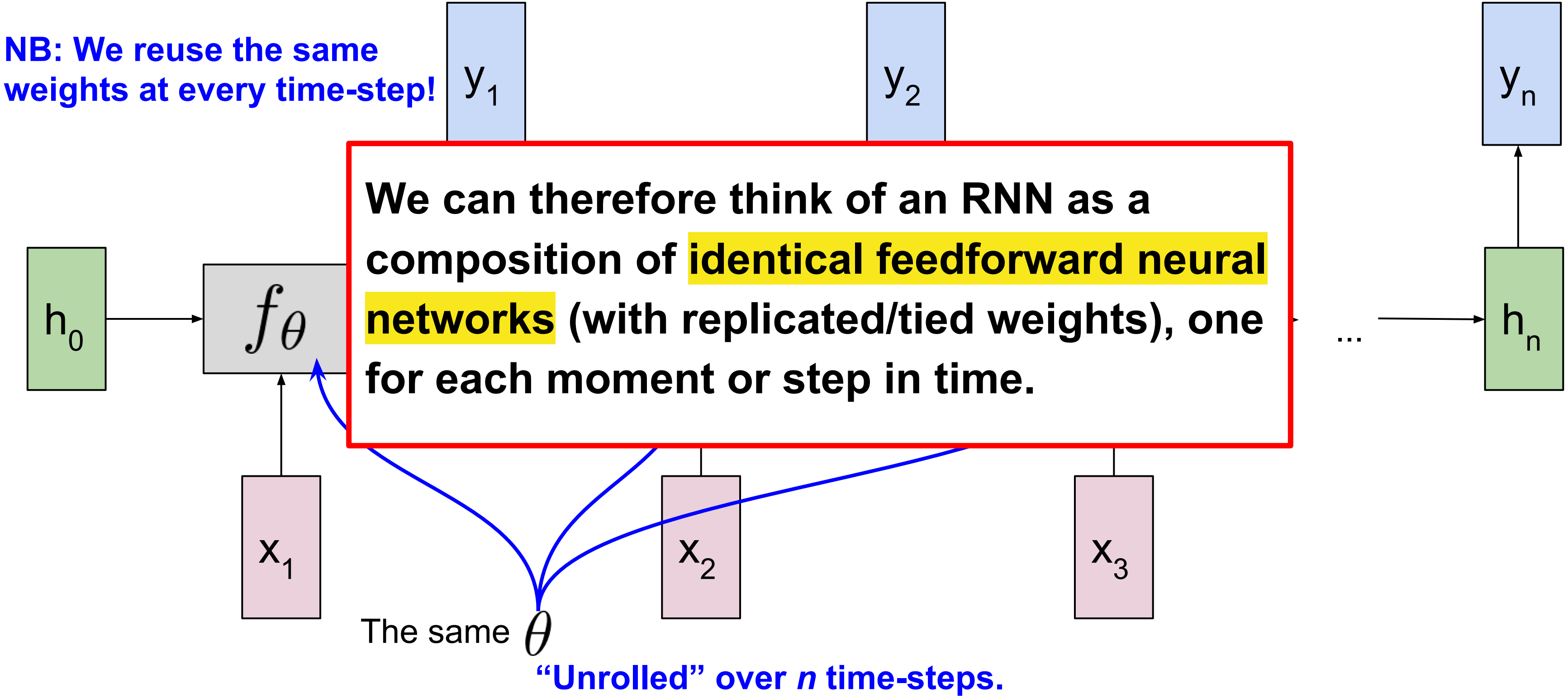


"Unrolled" over *n* time-steps.

Unrolling the RNN Computation Graph



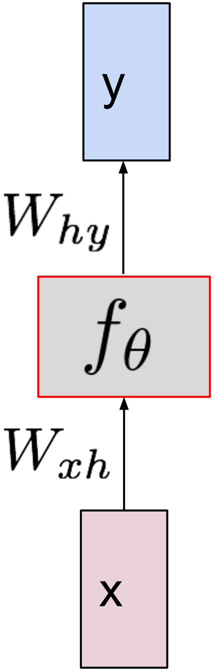
Unrolling the RNN Computation Graph



We can therefore think of an RNN as a composition of identical feedforward neural networks (with replicated/tied weights), one for each moment or step in time.

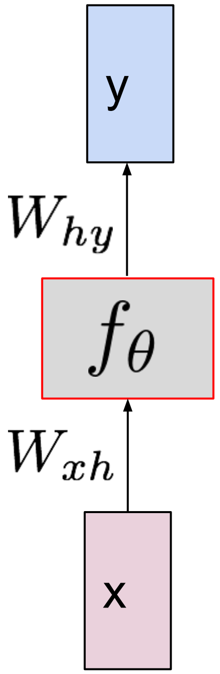
The FFN API

```
class FeedForwardModel():

  # ...

  def forward(self, x):
    # Compute activations on the hidden layer.
    hidden_layer = self.act_fn(np.dot(self.W_xh, x) + b)

    # Compute the (linear) output layer activations.
    y = np.dot(self.W_hy, hidden_layer)

    return y
```



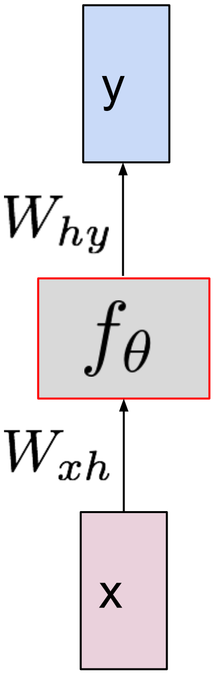
The FFN API

```
class FeedForwardModel():

  # ...

  def forward(self, x):
    # Compute activations on the hidden layer.
    hidden_layer = self.act_fn(np.dot(self.W_xh, x) + b)

    # Compute the (linear) output layer activations.
    y = np.dot(self.W_hy, hidden_layer)

    return y
```



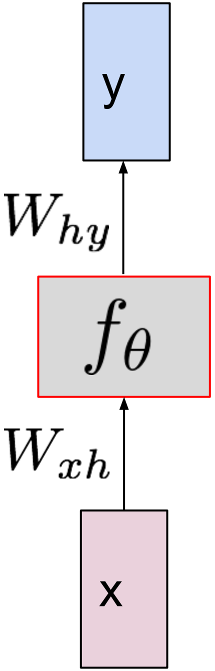
The FFN API

```
class FeedForwardModel():

  # ...

  def forward(self, x):
    # Compute activations on the hidden layer.
    hidden_layer = self.act_fn(np.dot(self.W_xh, x) + b)

    # Compute the (linear) output layer activations.
    y = np.dot(self.W_hy, hidden_layer)

    return y
```



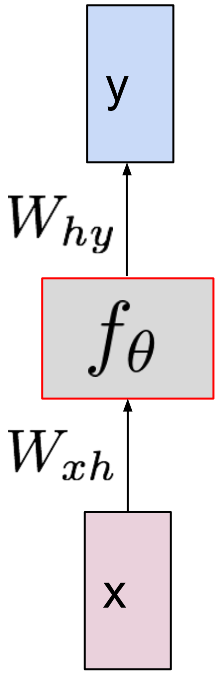
The FFN API

```
class FeedForwardModel():

  # ...

  def forward(self, x):
    # Compute activations on the hidden layer.
    hidden_layer = self.act_fn(np.dot(self.W_xh, x) + b)

    # Compute the (linear) output layer activations.
    y = np.dot(self.W_hy, hidden_layer)

    return y
```



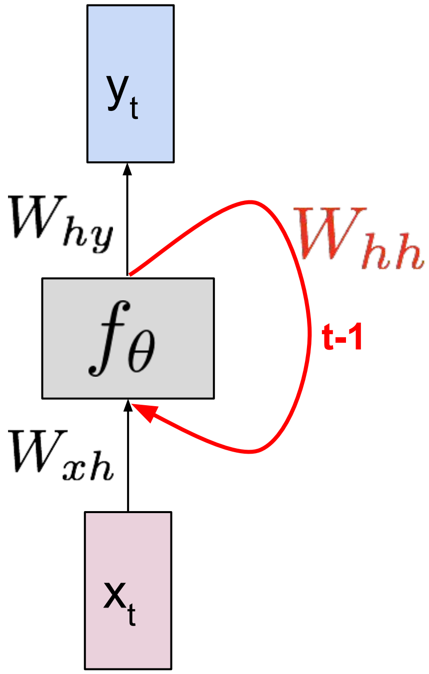
The FFN API

```
class FeedForwardModel():

  # ...

  def forward(self, x):
    # Compute activations on the hidden layer.
    hidden_layer = self.act_fn(np.dot(self.W_xh, x) + b)

    # Compute the (linear) output layer activations.
    y = np.dot(self.W_hy, hidden_layer)

    return y
```



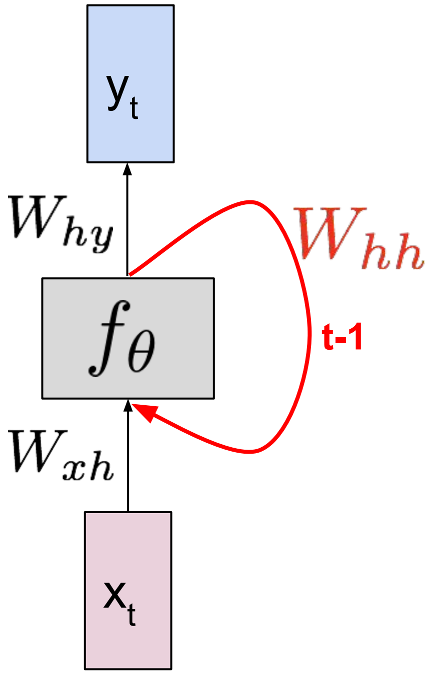
The RNN API

```
class RecurrentModel():

  # ...

  def recurrent_fn(self, x, prev_state):
    # Compute the new state based on the previous state and current input.
    new_state = self.act_fn(np.dot(self.W_xh, x) + np.dot(self.W_hh, prev_state) + b)

    # Compute the output vector.
    y = np.dot(self.W_hy, new_state)

    return new_state, y
```



The RNN API

```
class RecurrentModel():

  # ...

  def recurrent_fn(self, x, prev_state):
    # Compute the new state based on the previous state and current input.
    new_state = self.act_fn(np.dot(self.W_xh, x) + np.dot(self.W_hh, prev_state) + b)

    # Compute the output vector.
    y = np.dot(self.W_hy, new_state)

    return new_state, y
```



The RNN API

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

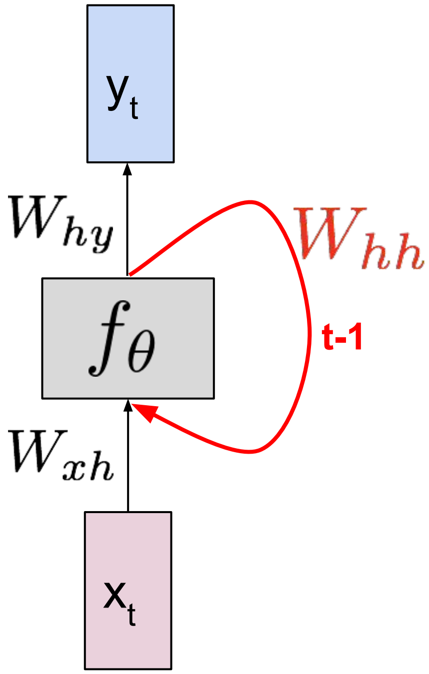
New **state**

Recurrent function

Input at current time-step

Previous **state**

```
class RecurrentModel():

  # ...

  def recurrent_fn(self, x, prev_state):
    # Compute the new state based on the previous state and current input.
    new_state = self.act_fn(np.dot(self.W_xh, x) + np.dot(self.W_hh, prev_state))

    # Compute the output vector.
    y = np.dot(self.W_hy, new_state)

    return new_state, y
```



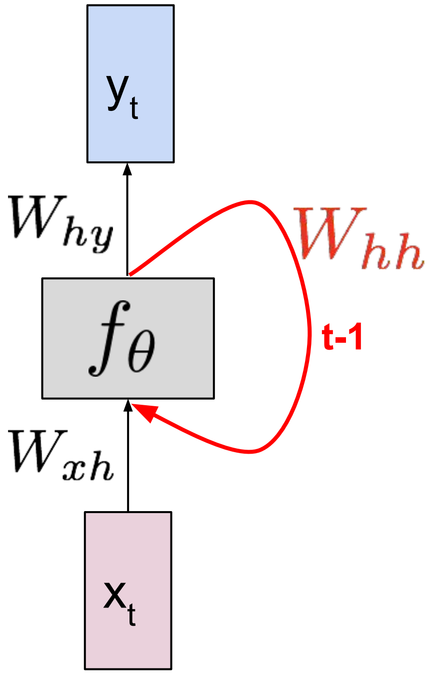
The RNN API

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

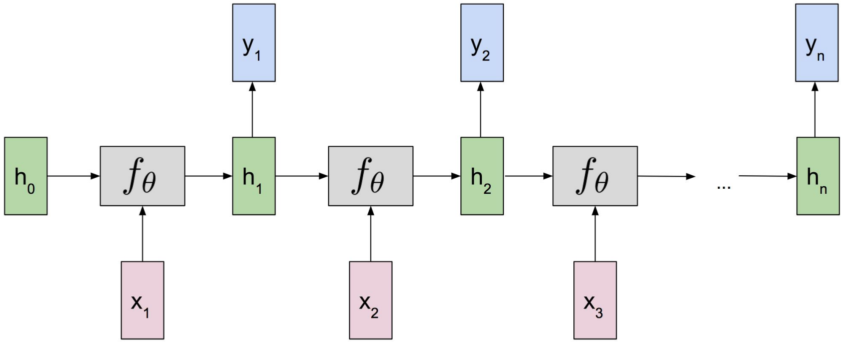
New **state** | Recurrent function | Input at current time-step | Previous **state**

$$y_t = W_{hy}h_t$$

```
class RecurrentModel():

  # ...

  def recurrent_fn(self, x, prev_state):
    # Compute the new state based on the previous state and current input.
    new_state = self.act_fn(np.dot(self.W_xh, x) + np.dot(self.W_hh, prev_state))

    # Compute the output vector.
    y = np.dot(self.W_hy, new_state)

    return new_state, y
```

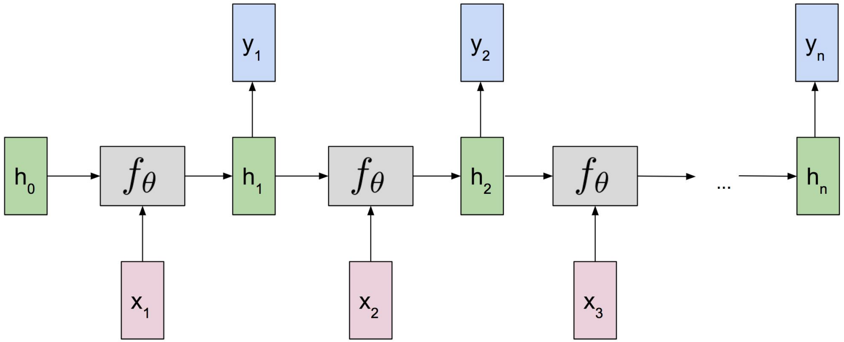


The RNN API



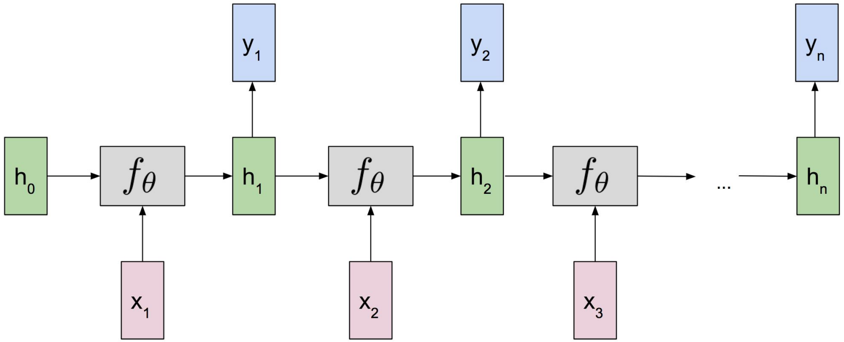

```
def forward(self, data_sequence, initial_state):
  state = initial_state
  all_states, all_ys = [state], []
  cache = []

  for x, y in data_sequence:
    new_state, y_pred = recurrent_fn(x, state)
    loss += cross_entropy(y_pred, y)

    cache.append((new_state, y_pred))
    state = new_state

  return loss, cache
```

The RNN API




```
def forward(self, data_sequence, initial_state):
  state = initial_state
  all_states, all_ys = [state], []
  cache = []

  for x, y in data_sequence:
    new_state, y_pred = recurrent_fn(x, state)
    loss += cross_entropy(y_pred, y)

    cache.append((new_state, y_pred))
    state = new_state

  return loss, cache
```

The RNN API

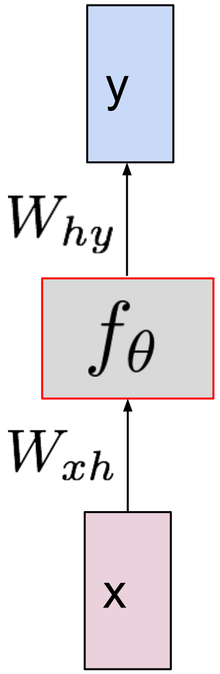



```
def forward(self, data_sequence, initial_state):
  state = initial_state
  all_states, all_ys = [state], []
  cache = []

  for x, y in data_sequence:
    new_state, y_pred = recurrent_fn(x, state)
    loss += cross_entropy(y_pred, y)

    cache.append((new_state, y_pred))
    state = new_state

  return loss, cache
```

Math: FFNs v RNNs

NOTATION: W_{xh} is a **matrix** that maps a **vector** x into a **vector** h.

$$h = f_\theta(W_{xh}x + b)$$

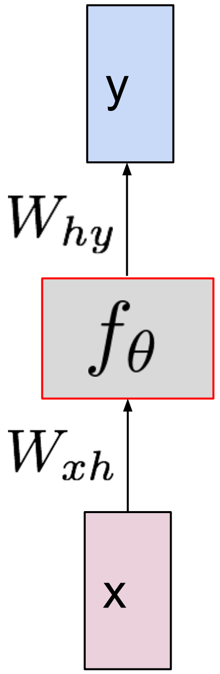


Math: FFNs v RNNs

NOTATION: W_{xh} is a **matrix** that maps a **vector** x into a **vector** h.

$$h = f_\theta(W_{xh}\boxed{x} + b)$$

Input



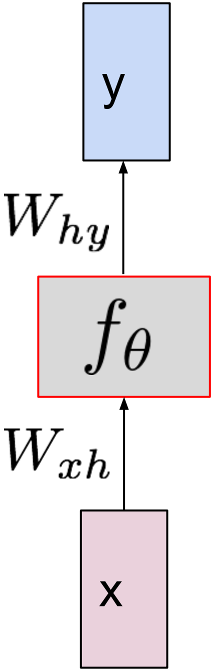
Math: FFNs v RNNs

NOTATION: W_{xh} is a **matrix** that maps a **vector** x into a **vector** h.

$$h = \boxed{f_\theta}(W_{xh}\boxed{x} + b)$$

Activation function

Input



Math: FFNs v RNNs

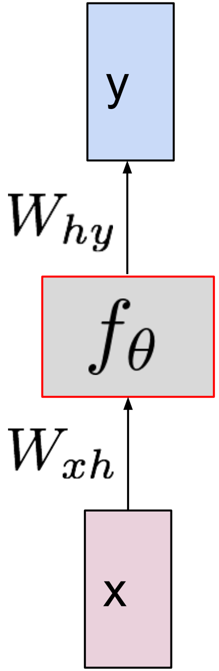
NOTATION: W_{xh} is a **matrix** that maps a **vector** x into a **vector** h.

Hidden layer

$$\boxed{h} = \boxed{f_\theta}(W_{xh}\boxed{x} + b)$$

Activation function

Input



Math: FFNs v RNNs

NOTATION: W_{xh} is a **matrix** that maps a **vector** x into a **vector** h.

Hidden layer

$$\boxed{h} = \boxed{f_\theta}(W_{xh}\boxed{x} + b)$$

Activation function

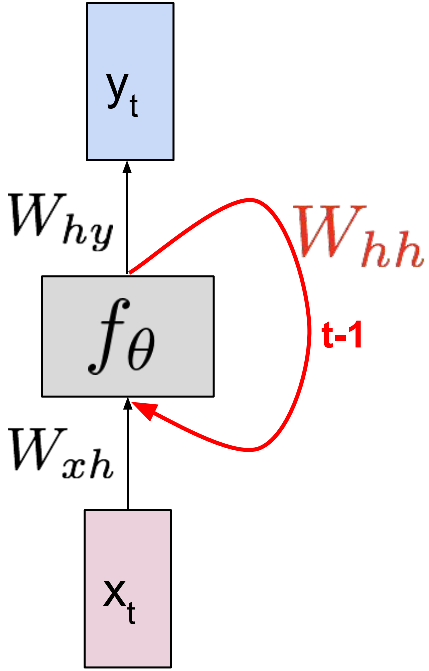
Input

$$\boxed{h_t} = \boxed{f_\theta}(W_{xh}\boxed{x_t} \quad \cdots \quad)$$

New **state**

Recurrent function

Input at current time-step



Math: FFNs v RNNs

NOTATION: W_{xh} is a **matrix** that maps a **vector** x into a **vector** h.

Hidden layer

$$\boxed{h} = \boxed{f_\theta}(W_{xh}\boxed{x} + b)$$

Activation function

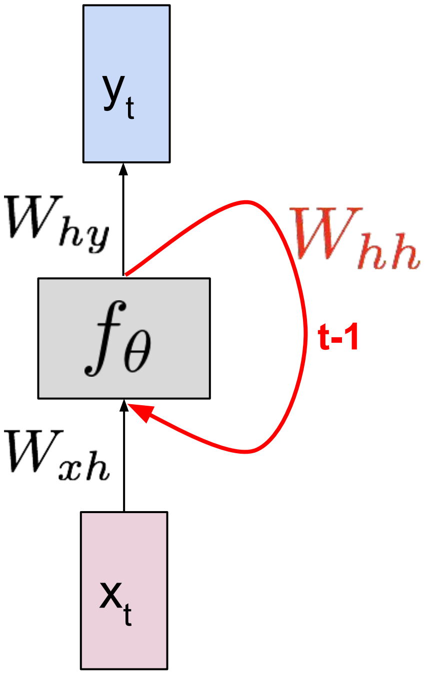
Input

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

New **state**

Recurrent function

Input at current time-step



Math: FFNs v RNNs

NOTATION: W_{xh} is a **matrix** that maps a **vector** x into a **vector** h.

Hidden layer

$$\boxed{h} = \boxed{f_\theta}(W_{xh}\boxed{x} + b)$$

Activation function

Input

“**Recurrent**” weights

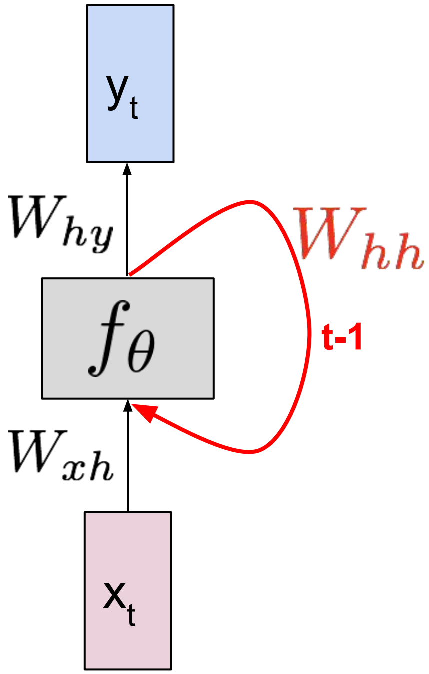
$$\boxed{h_t} = \boxed{f_\theta}(W_{xh}\boxed{x_t} + W_{hh}\boxed{h_{t-1}})$$

New **state**

Recurrent function

Input at current time-step

Previous **state**



Inference & Training

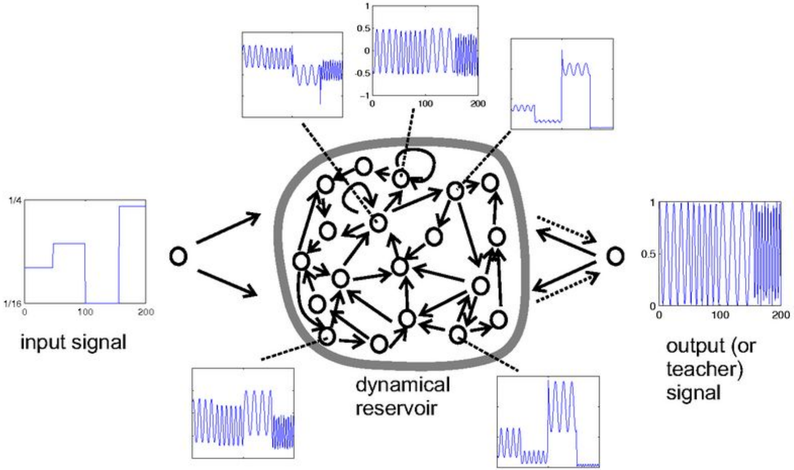
- How do we make predictions using RNNs?
  - Forward propagation: “Fprop”
  - Essentially a composition of functions: $a_2 = f_2(f_1(x))$.
  - We “unroll” the computational graph over time-steps.
- How do we train RNNs?
  - Backward propagation: “Backprop-through time”
  - We need to consider predictions over several time-steps!
  - Credit assignment over time.
  - We work backwards in time from the last state to the first.

Training: Ways to Train RNNs

- **Echo State Networks**: Initialize W_{xh}, W_{hh}, W_{ho}, carefully, then only train W_{ho}!
- **Backpropagation through time (BPTT)**: Propagate errors backwards through the unrolled graph.
- There are other options.

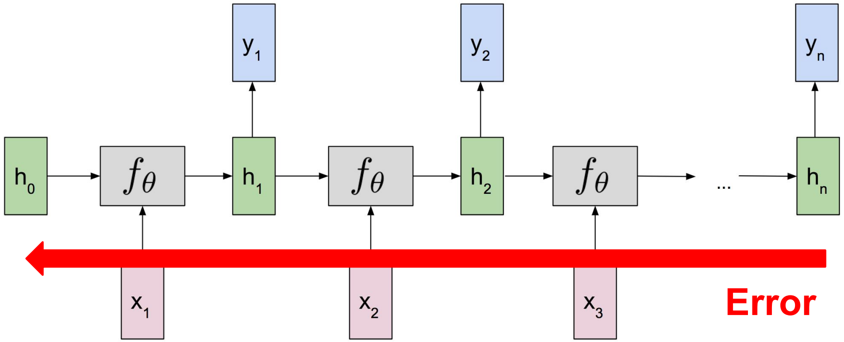
Training: ESNs

- Simple solution: don't train the recurrent weights (W_{hh} & W_{xh})!
- Initialization very important.
- Super simple. However, with recent improvements in initialization etc, BPTT does better!



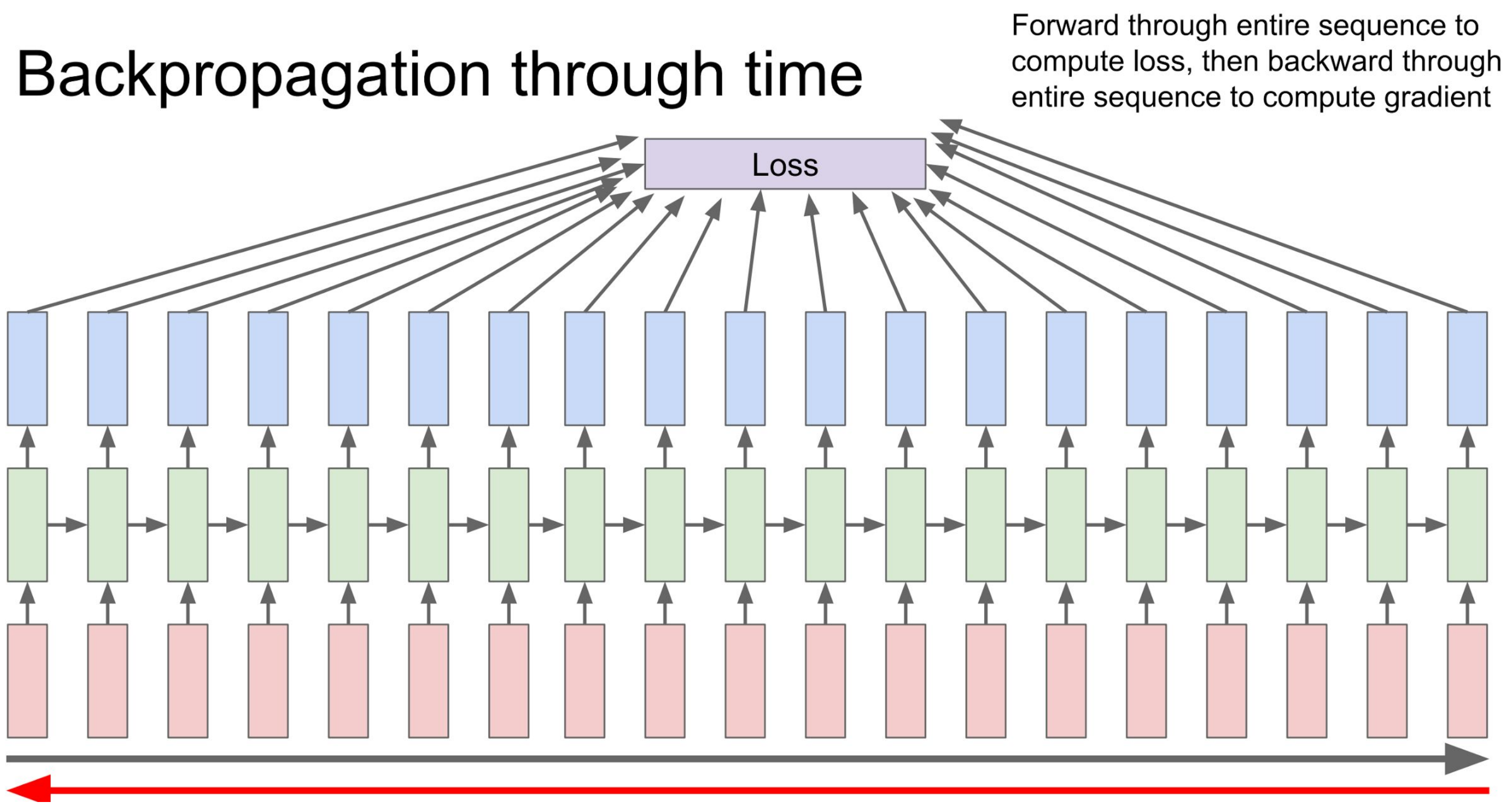
[Scholarpedia]

Inference & Training

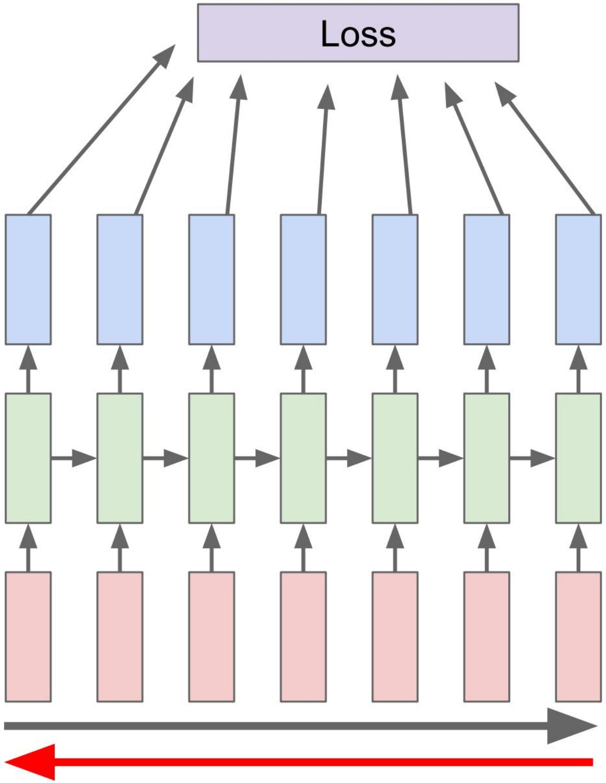


- How do we make predictions using RN
  - Forward propagation: “Fprop”
  - Essentially a composition of functions: $a_2 = f_2(f_1(x))$.
  - We “unroll” the computational graph over time-steps.
- How do we train RNNs?
  - Propagate errors backwards through unrolled graph: “Backprop-through time” (BPTT).
  - We need to consider predictions over several time-steps!
  - Credit assignment over time.
  - We work backwards in time from the last state to the first.

Training: BPTT Intuition

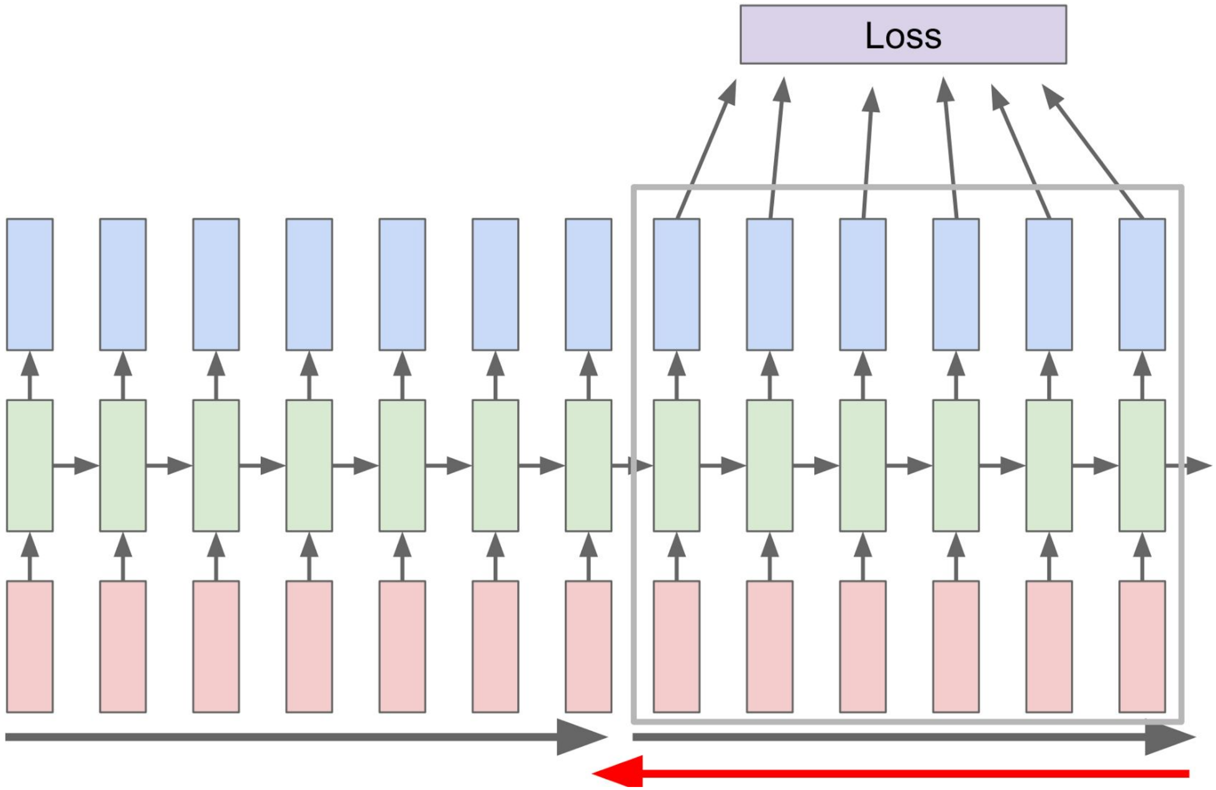


Training: Truncated BPTT



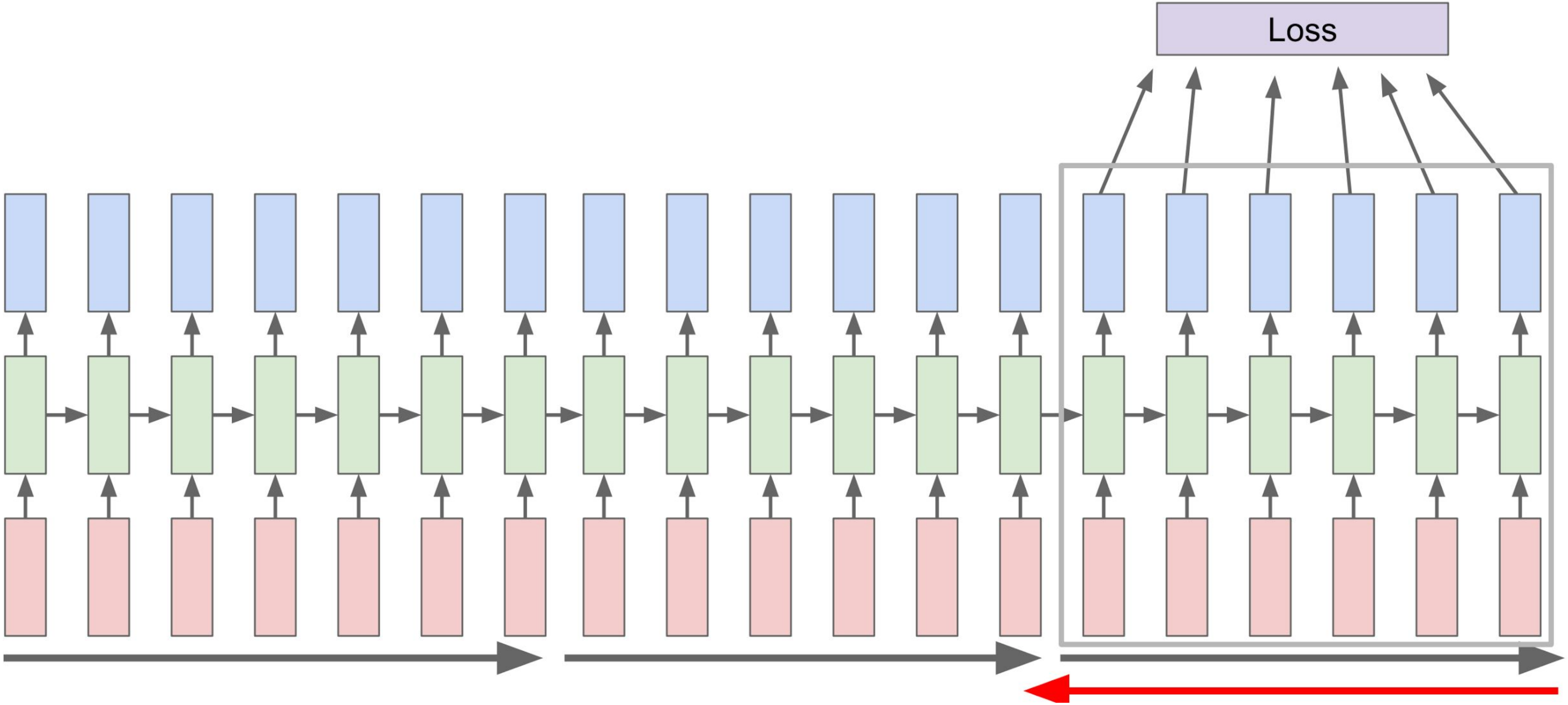
Run forward and backward through chunks of the sequence instead of whole sequence

Training: **Truncated** BPTT

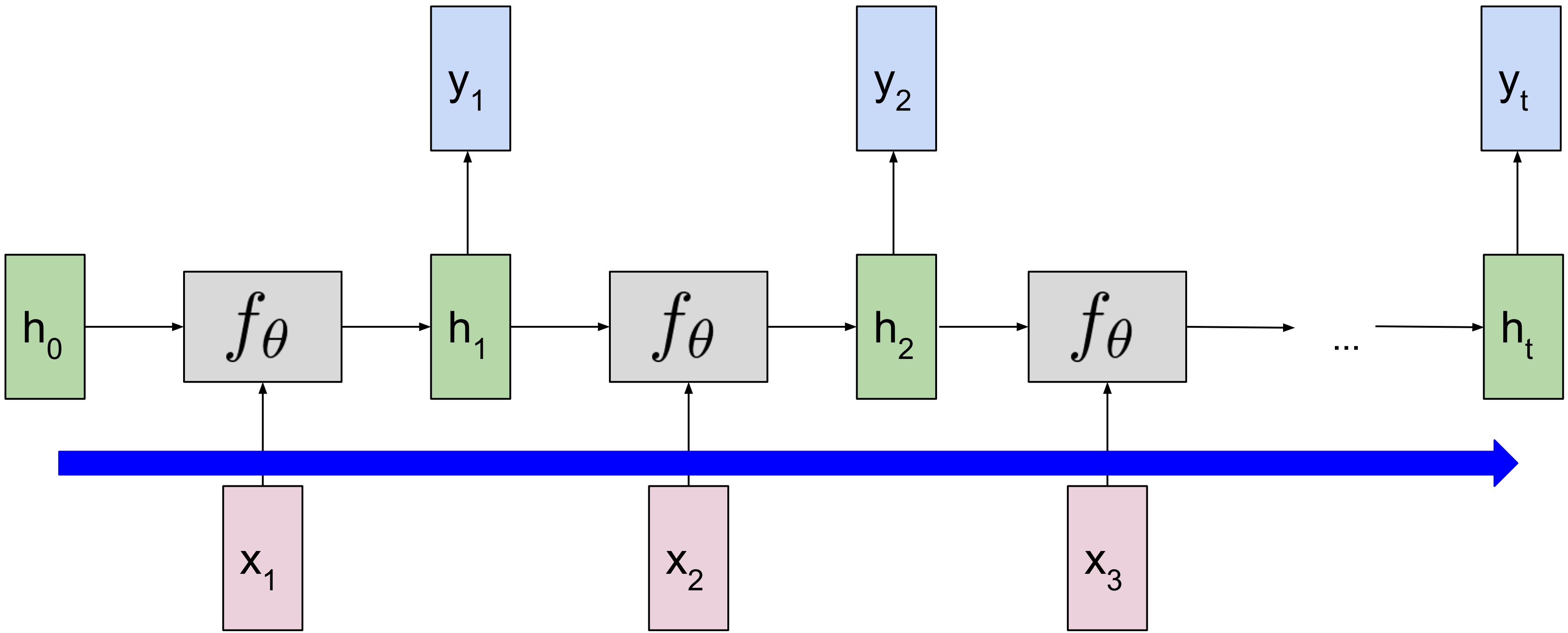


Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps

Training: **Truncated** BPTT

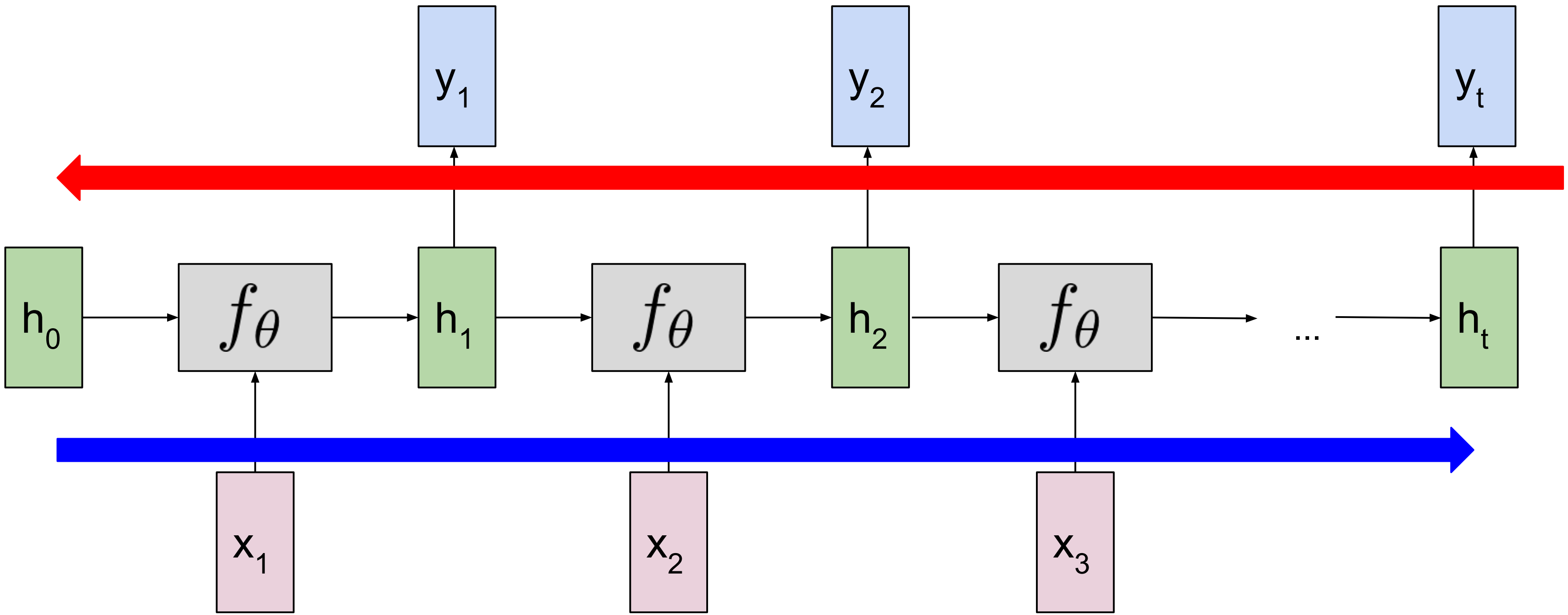


Unrolling the RNN Computation Graph



“Unrolled” over *n* time-steps.

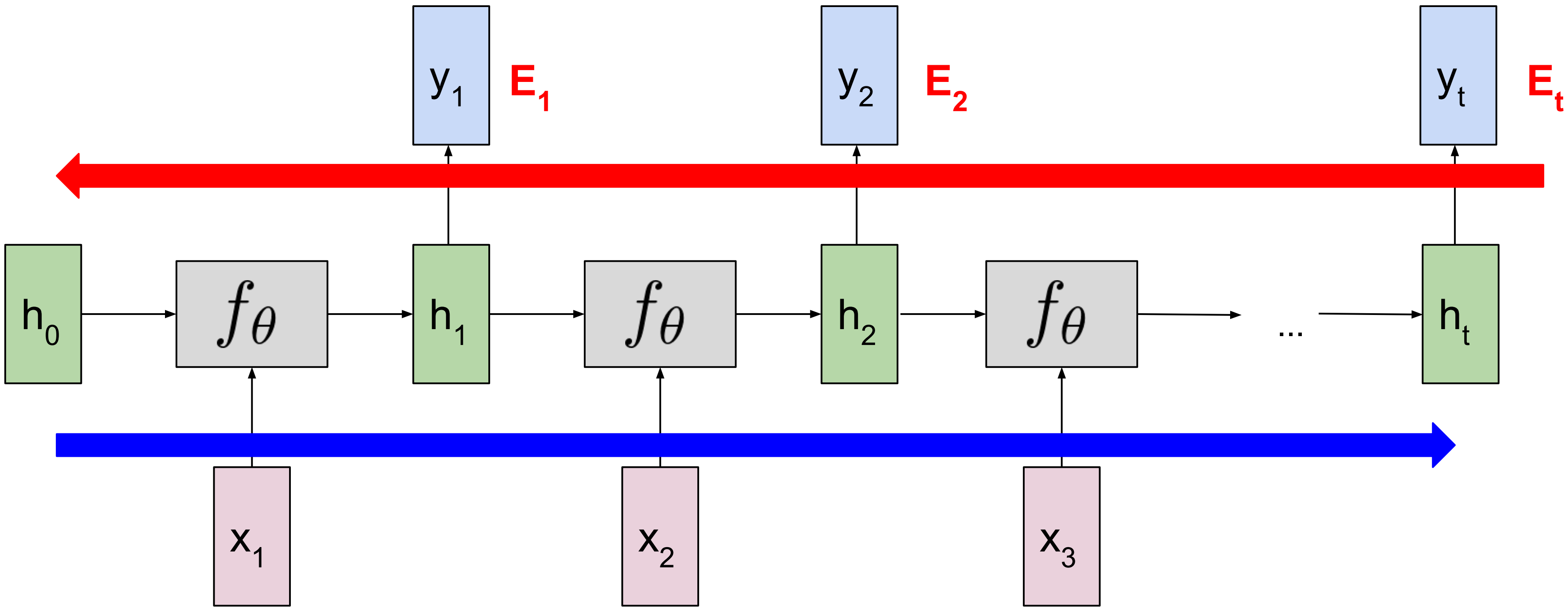
Unrolling the RNN Computation Graph



"Unrolled" over *n* time-steps.

Step 1: Compute all errors.

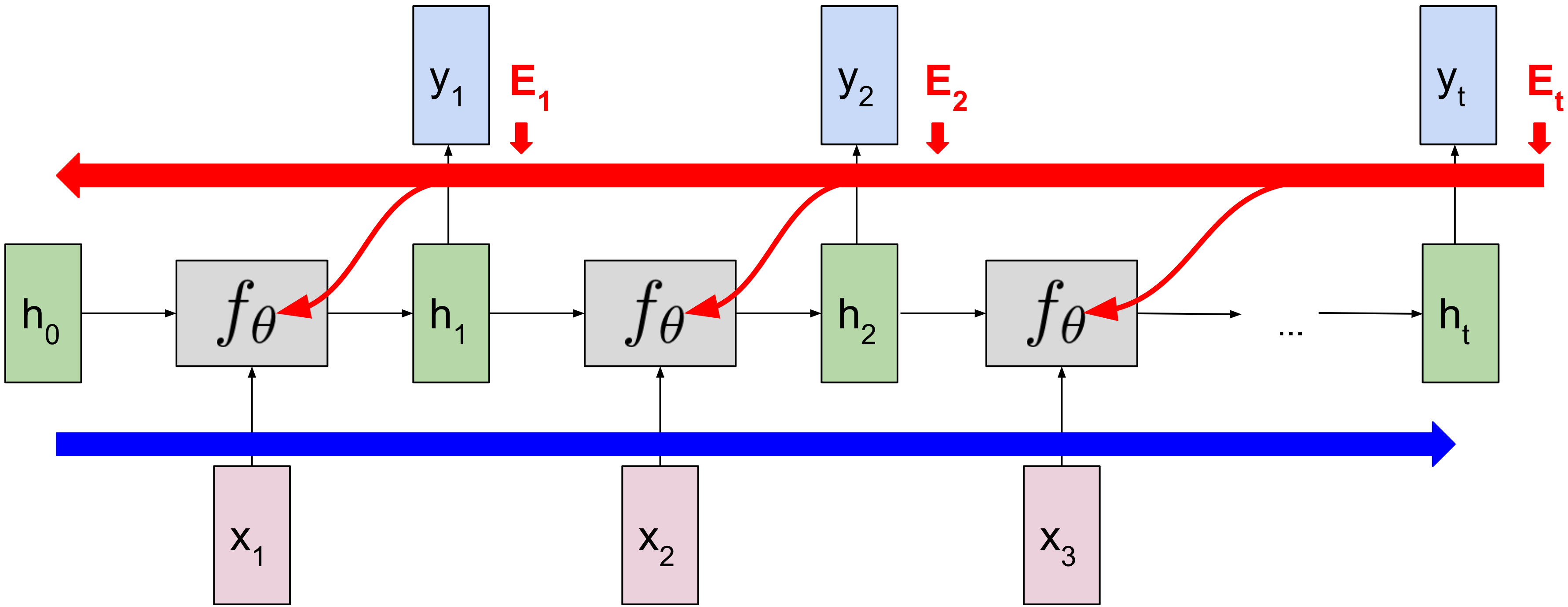
Unrolling the RNN Computation Graph



"Unrolled" over *n* time-steps.

Unrolling the RNN Computation Graph

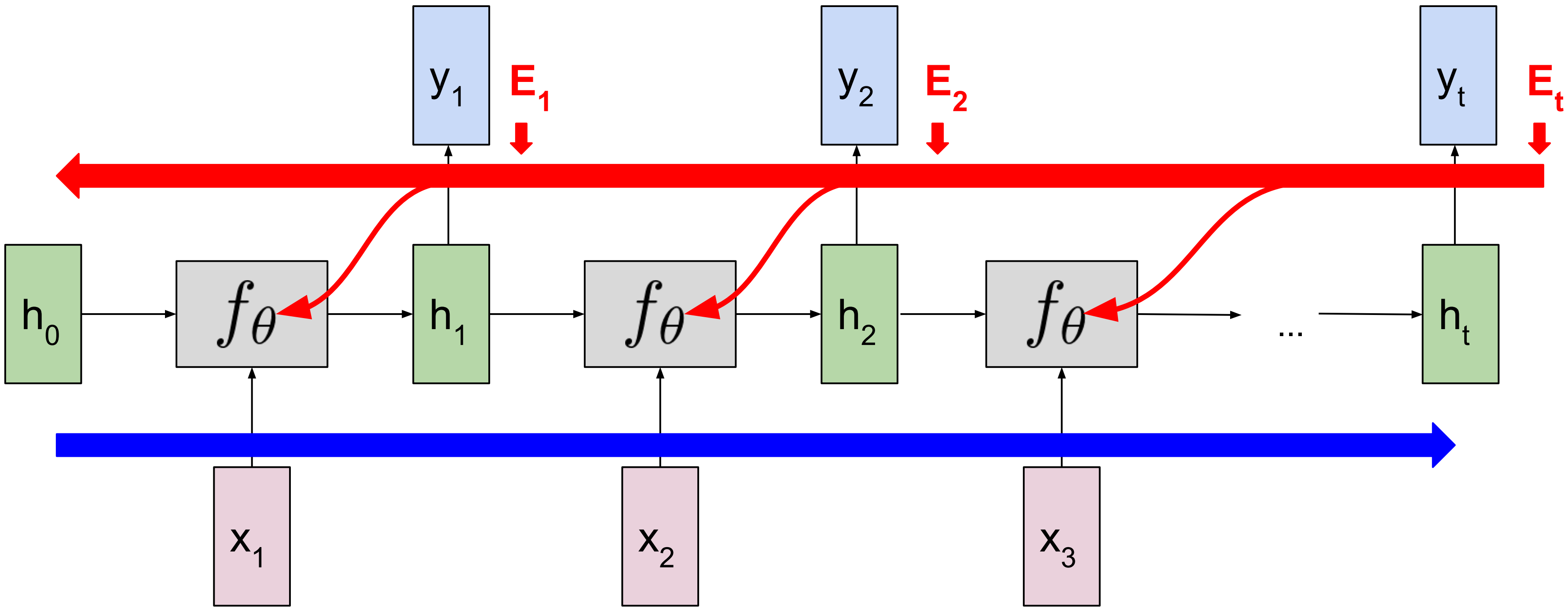
Step 1: Compute all errors.
Step 2: Pass error back for each time-step from n back to 1.



"Unrolled" over *n* time-steps.

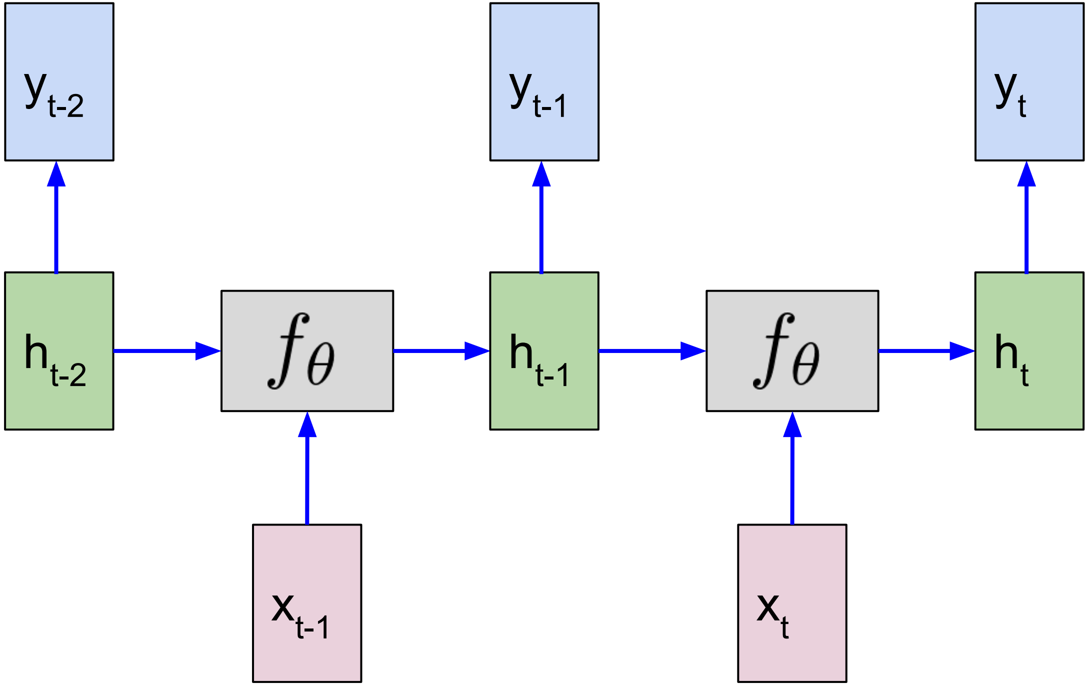
Unrolling the RNN Computation Graph

Step 1: Compute all errors.
Step 2: Pass error back for each time-step from n back to 1.
Step 3: Update weights.

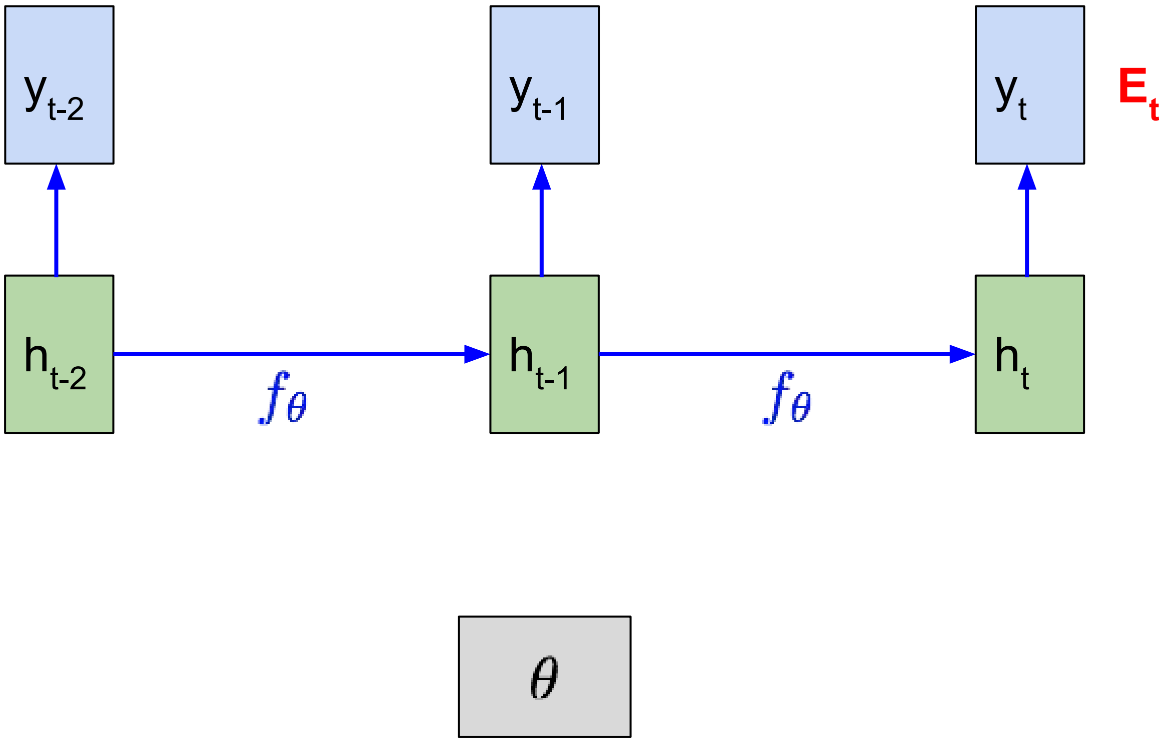


“Unrolled” over *n* time-steps.

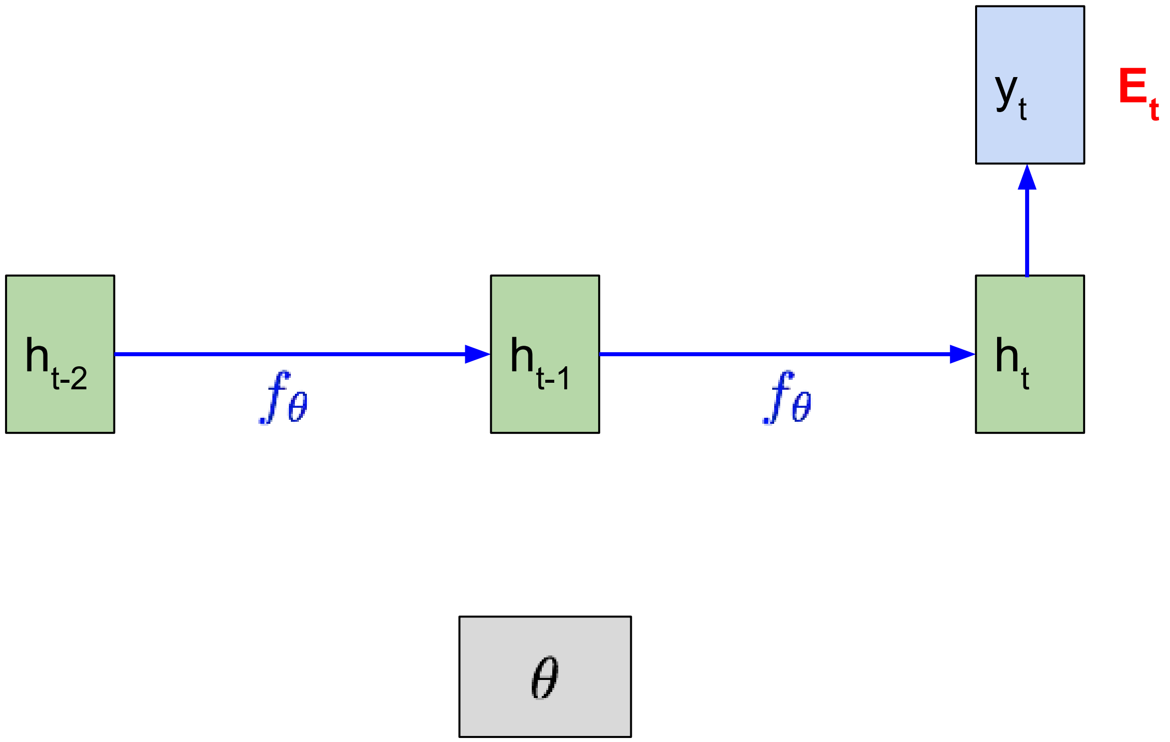
Unrolling the RNN Computation Graph



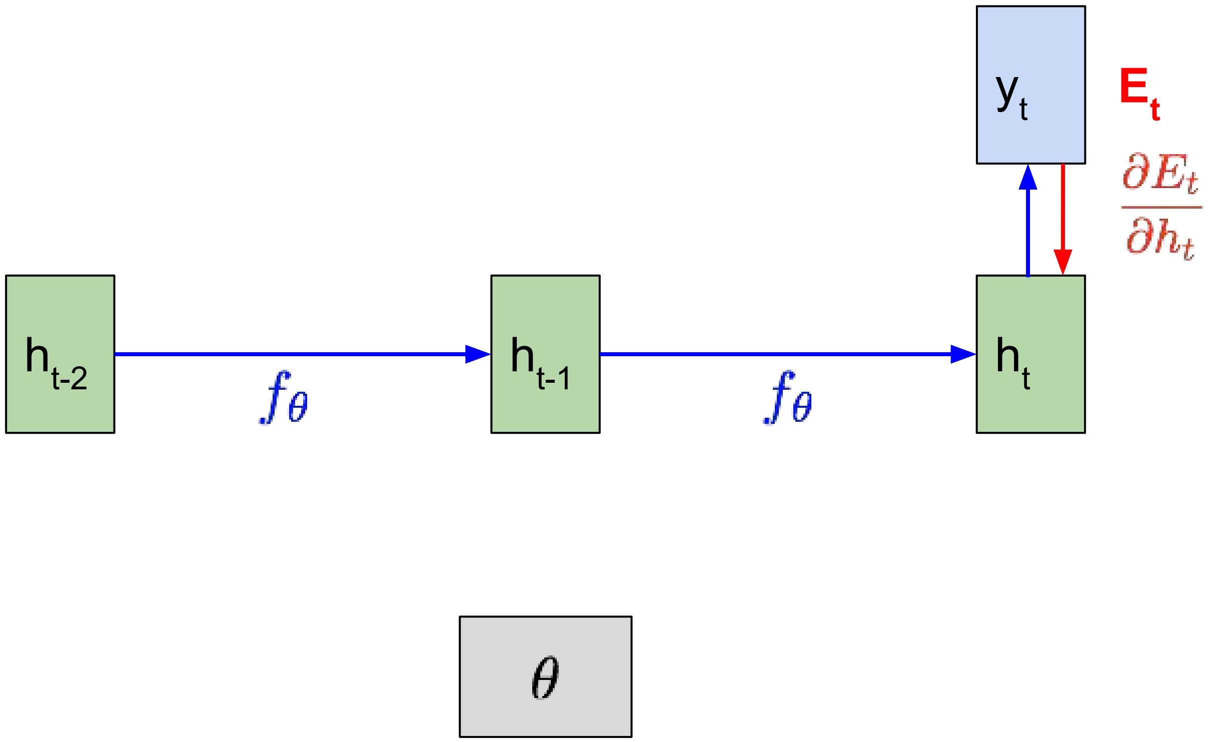
Unrolling the RNN Computation Graph



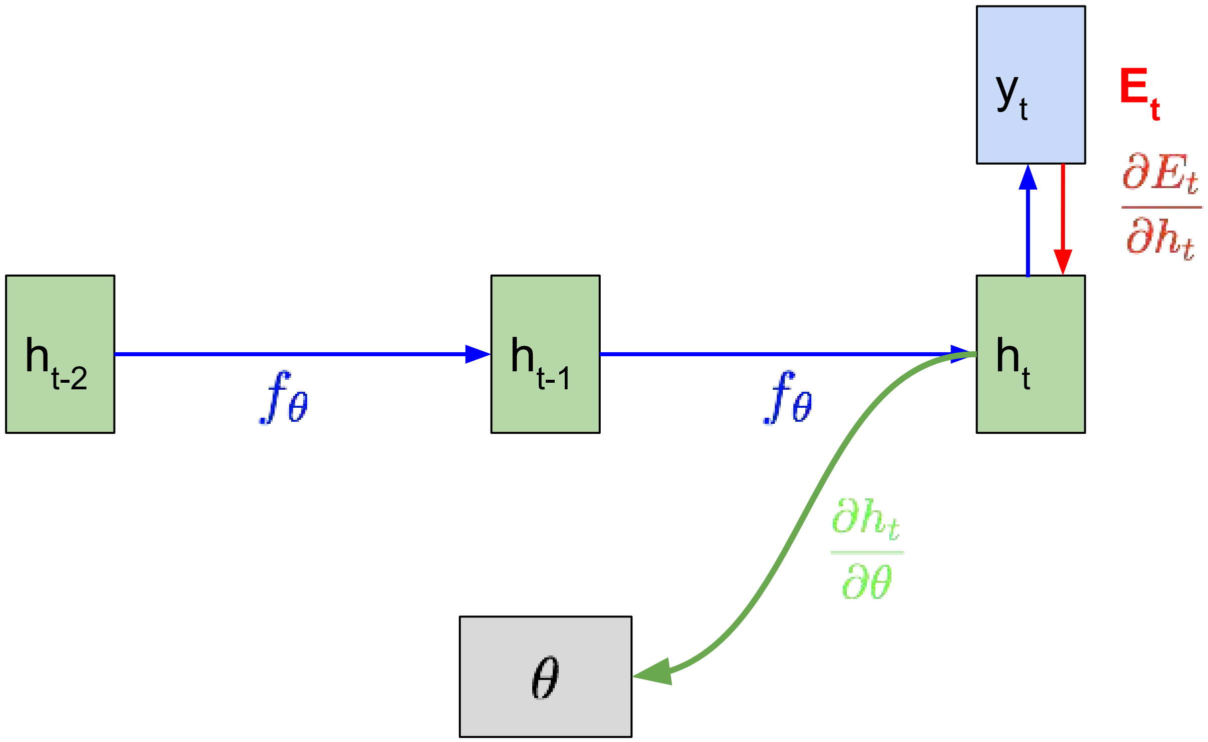
Unrolling the RNN Computation Graph



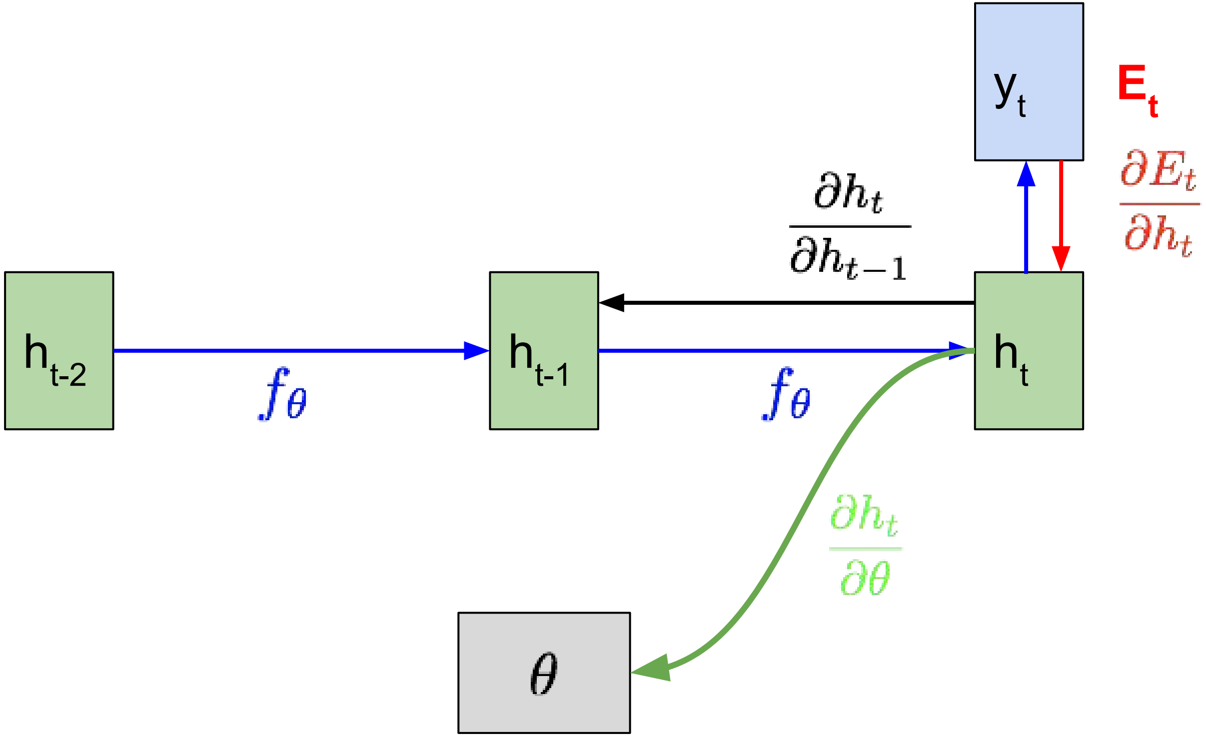
Unrolling the RNN Computation Graph



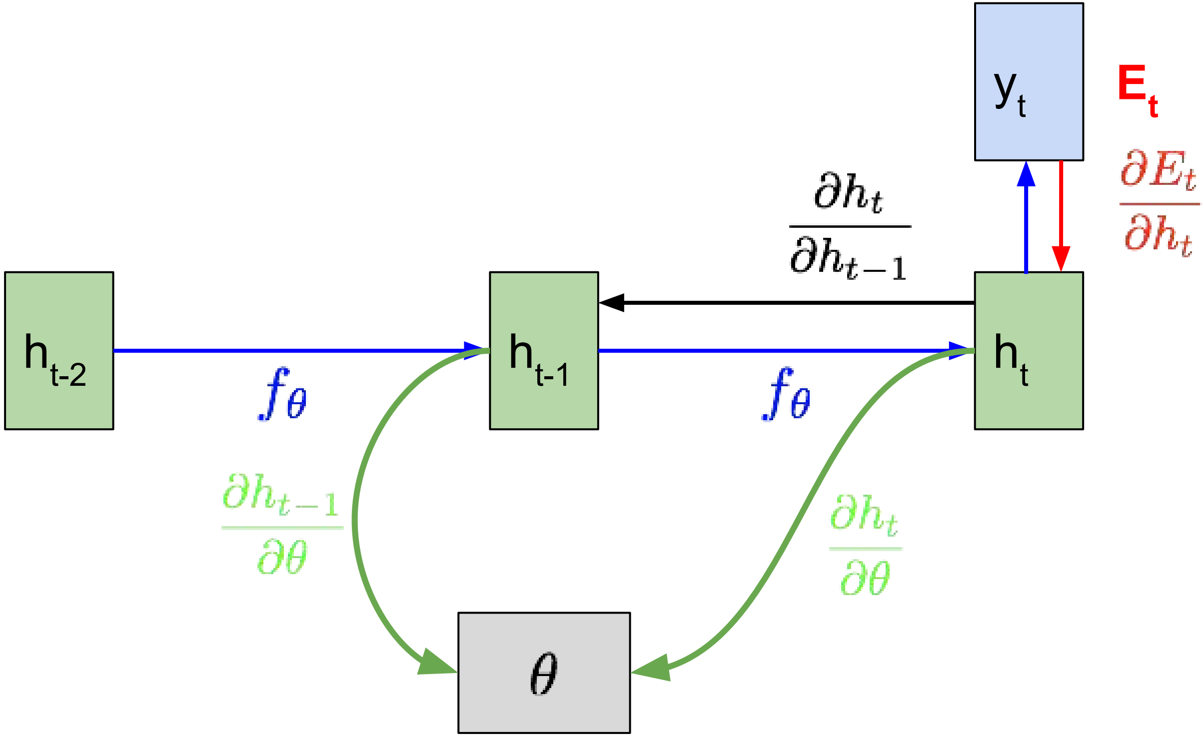
Unrolling the RNN Computation Graph



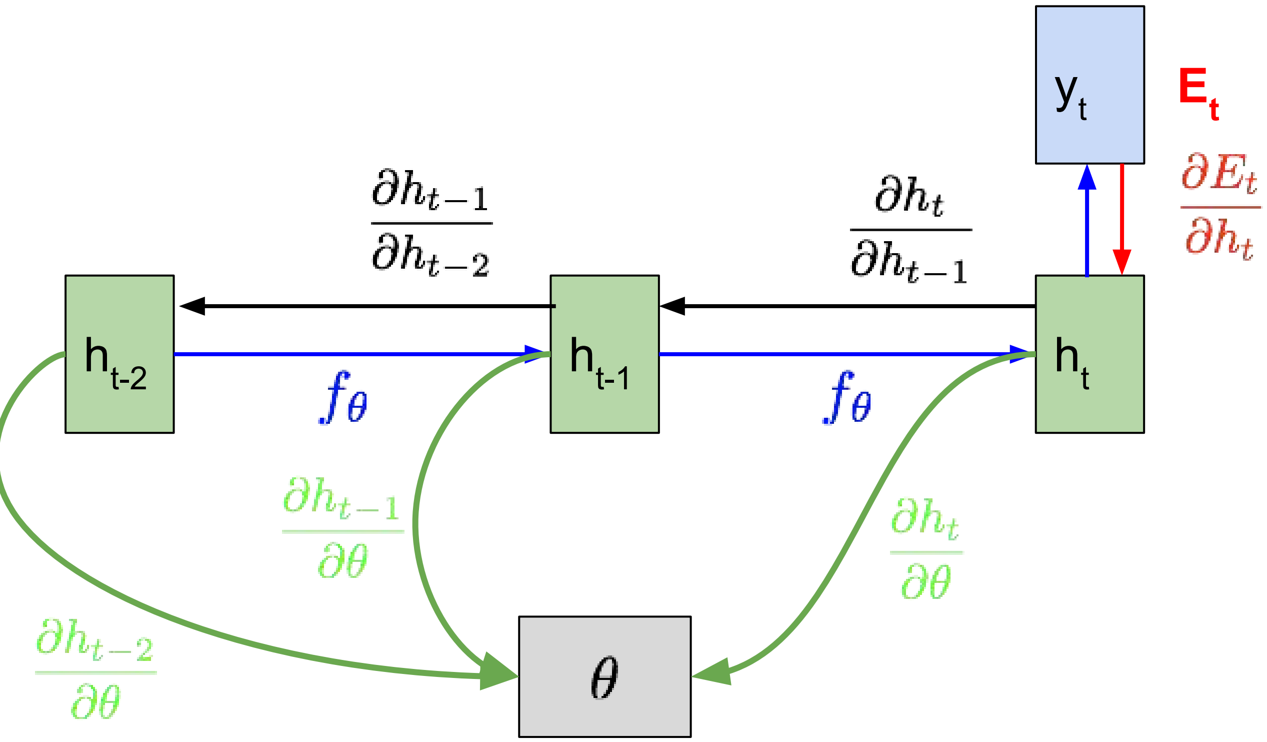
Unrolling the RNN Computation Graph



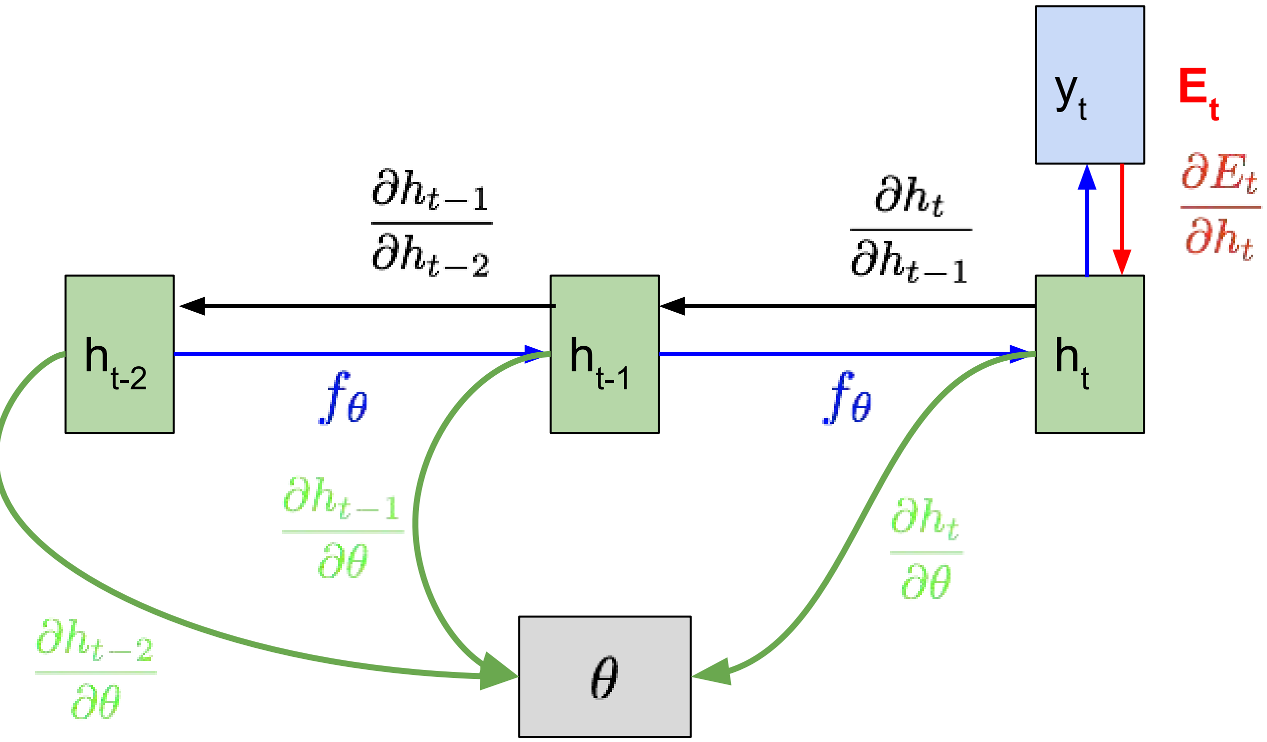
Unrolling the RNN Computation Graph



Unrolling the RNN Computation Graph



Unrolling the RNN Computation Graph

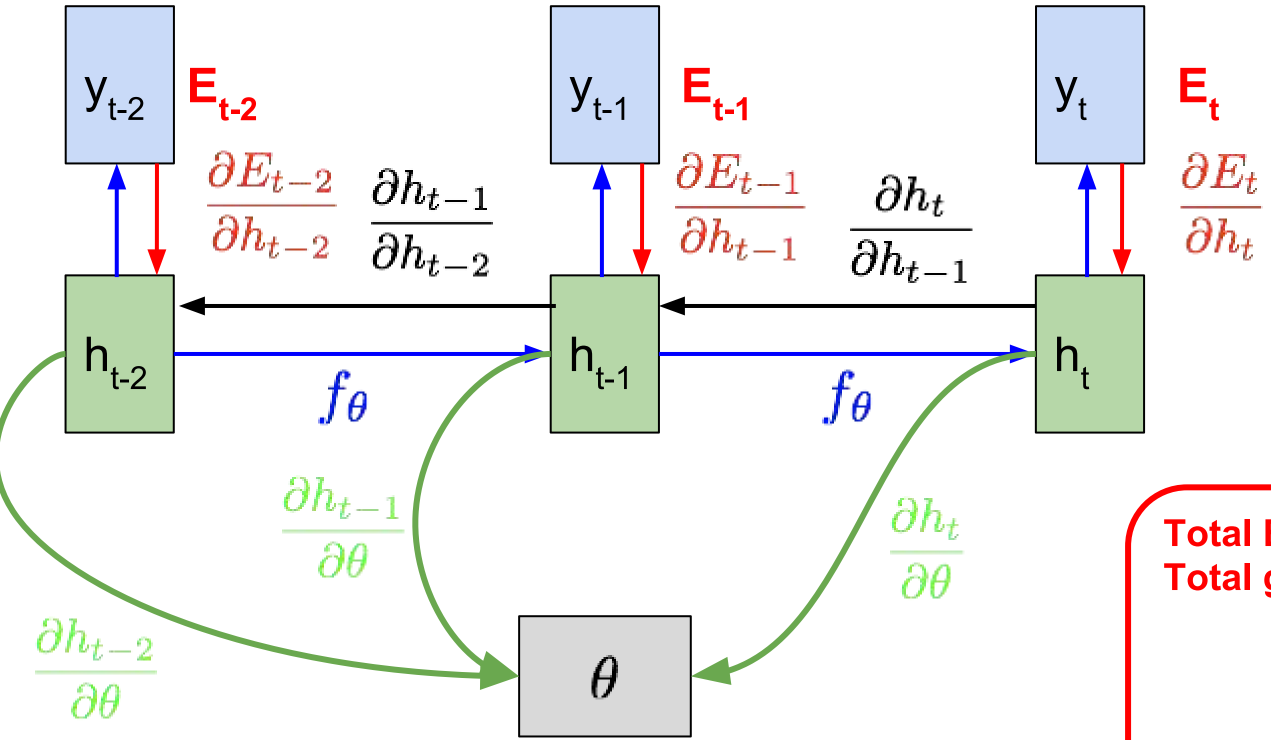


$$\frac{\partial E_t}{\partial \theta} = \sum_{t'=1}^{t} \frac{\partial E_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t'}} \frac{\partial h_{t'}}{\partial \theta}$$

where

$$\frac{\partial h_t}{\partial h_{t'}} = \Pi_{k=t'+1}^{t} \frac{\partial h_k}{\partial h_{k-1}}$$

Unrolling the RNN Computation Graph



$$\frac{\partial E_t}{\partial \theta} = \sum_{t'=1}^{t} \frac{\partial E_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t'}} \frac{\partial h_{t'}}{\partial \theta}$$

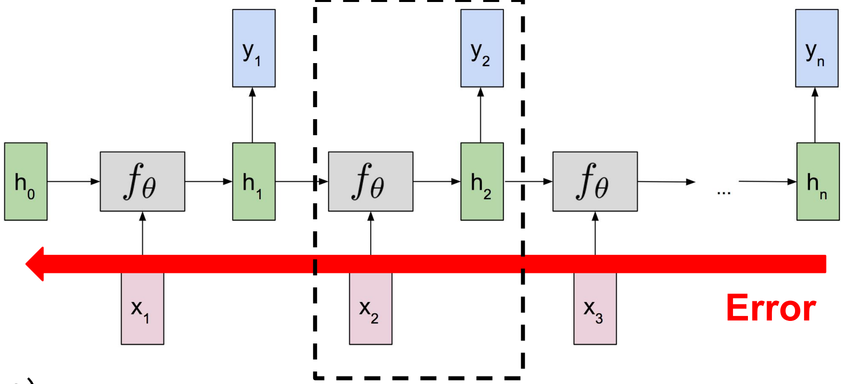
where

$$\frac{\partial h_t}{\partial h_{t'}} = \Pi_{k=t'+1}^{t} \frac{\partial h_k}{\partial h_{k-1}}$$

Total Error = E_1 + E_2 + … + E_t
Total gradient = sum of all $dE_t/d\theta$'s

$$\frac{\partial E_{TOTAL}}{\partial \theta} = \frac{\partial \sum_t E_t}{\partial \theta}$$
$$= \sum_t \frac{\partial E_t}{\partial \theta}$$

Training: Truncated BPTT Code

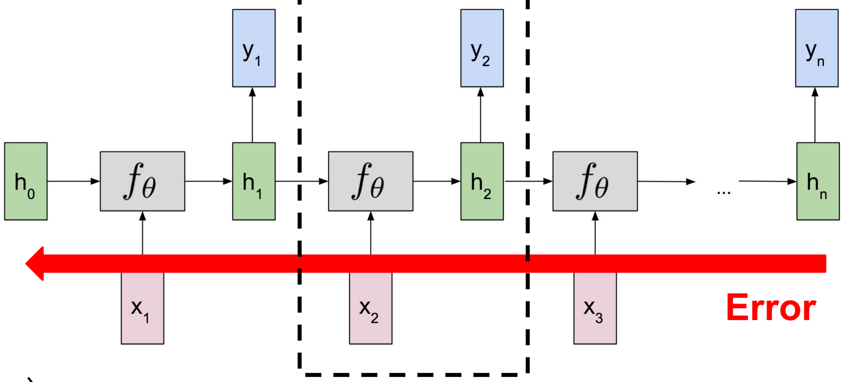


"Lego block"!

```
def bptt(model, X_train, y_train, initial_state):
  # Forward
  Loss, caches = forward(X_train, y_train, model, initial_state)
  avg_loss /= y_train.shape[0]
  # Backward
  dh_next = np.zeros((1, last_state.shape[0]))
  grads = {k: np.zeros_like(v) for k, v in model.items()}

  for t in reversed(range(len(X_train))):
    grad, dh_next = cell_fn_backward(ys[t], y_train[t], dh_next, caches[t])
    for k in grads.keys():
        grads[k] += grad[k]

  return grads, avg_loss
```

Training: Truncated BPTT Code

```
def bptt(model, X_train, y_train, initial_state):
  # Forward
  Loss, caches = forward(X_train, y_train, model, initial_state)
  avg_loss /= y_train.shape[0]
  # Backward
  dh_next = np.zeros((1, last_state.shape[0]))
  grads = {k: np.zeros_like(v) for k, v in model.items()}

  for t in reversed(range(len(X_train))):
    grad, dh_next = cell_fn_backward(ys[t], y_train[t], dh_next, caches[t])
    for k in grads.keys():
        grads[k] += grad[k]

  return grads, avg_loss
```

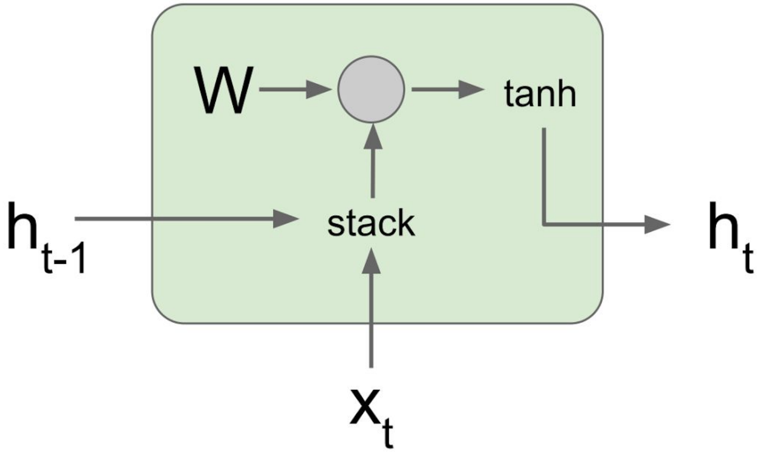


“Lego block”!

Total gradient = Sum of these lego-gradients over time!

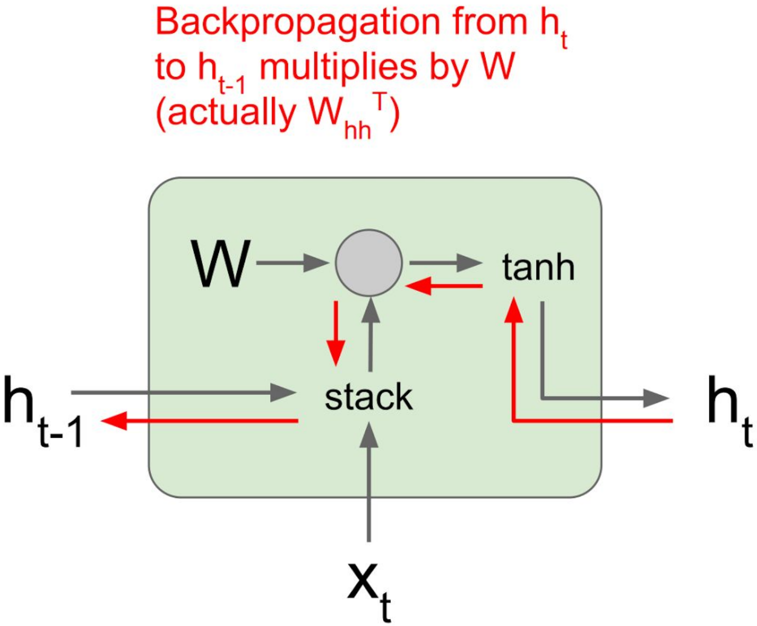
$$\frac{\partial E_{TOTAL}}{\partial \theta} = \frac{\partial \sum_t E_t}{\partial \theta} = \sum_t \frac{\partial E_t}{\partial \theta}$$

Vanilla RNN Gradient Flow



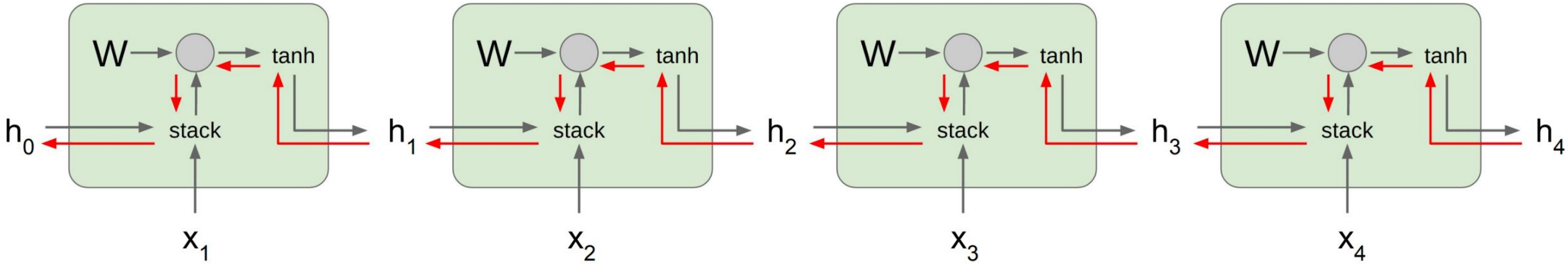
$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\
&= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)
\end{aligned}
$$

Vanilla RNN Gradient Flow



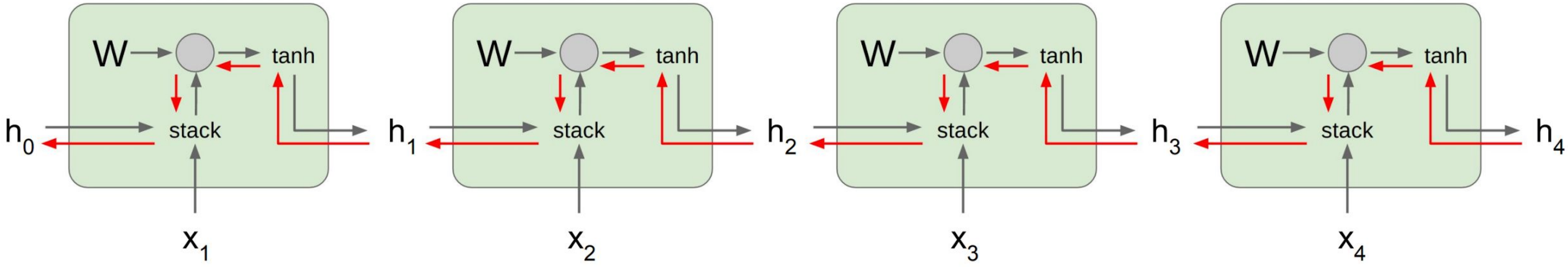
$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\
&= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)
\end{aligned}
$$

Vanilla RNN Gradient Flow



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

Vanilla RNN Gradient Flow

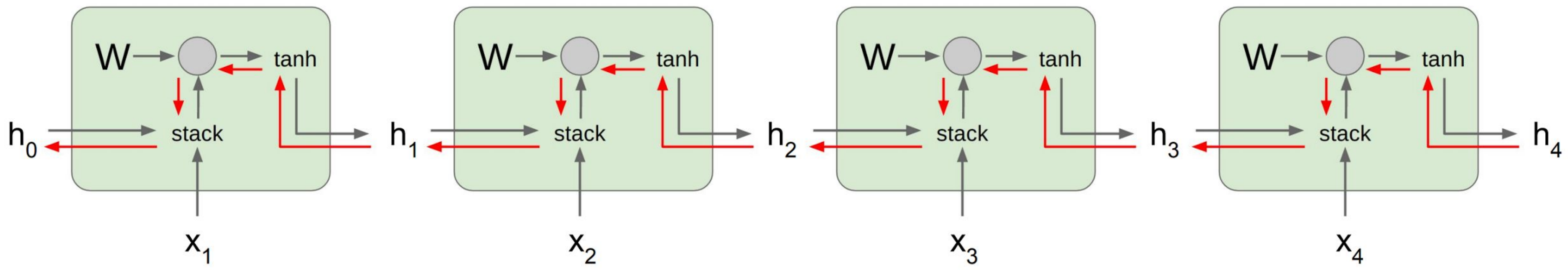


Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1:
Exploding gradients

Largest singular value < 1:
Vanishing gradients

Vanilla RNN Gradient Flow



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

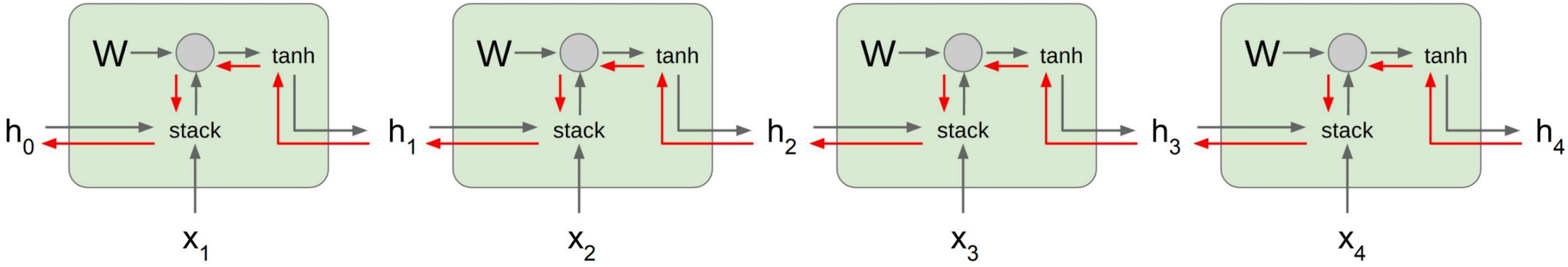
Largest singular value > 1: **Exploding gradients**

→ **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



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

Largest singular value > 1:
Exploding gradients

Largest singular value < 1:
Vanishing gradients

→ Change RNN architecture

Part 2!

Gated Recurrent Models

PART II: Gated Architectures & Applications

RECAP: The RNN API

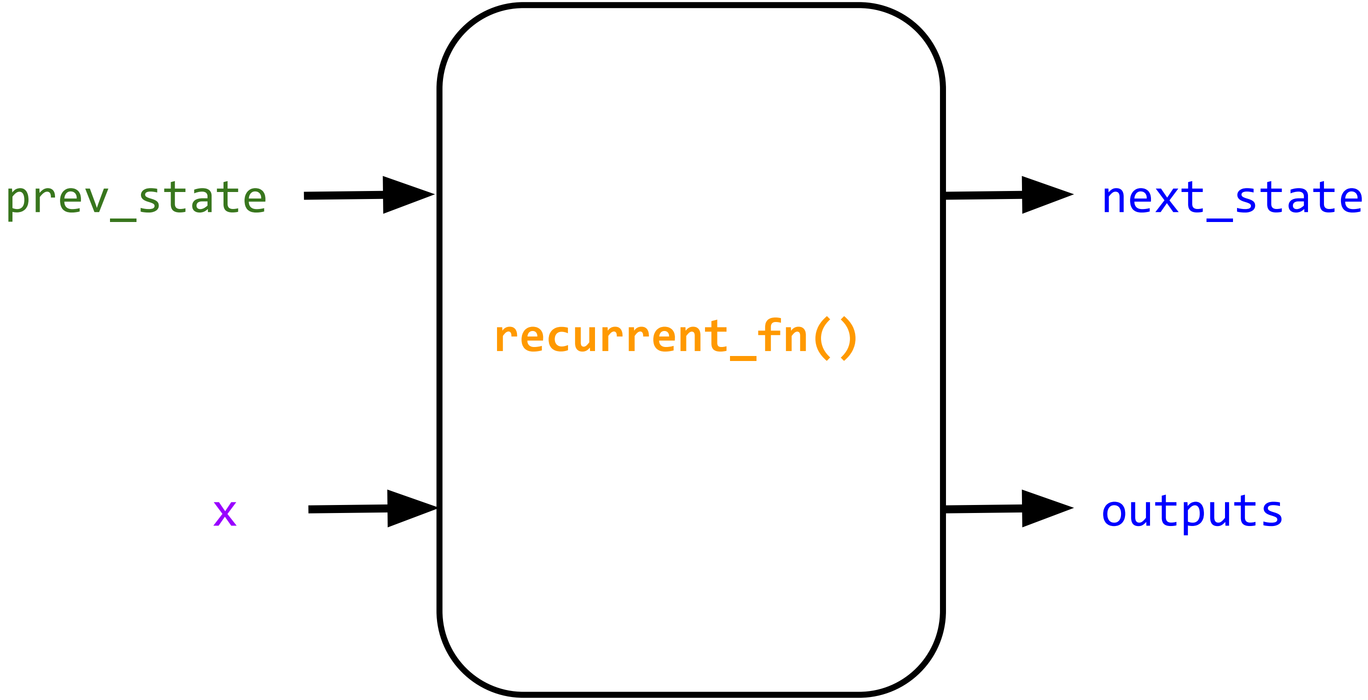
$$\boxed{h_t} = \boxed{f_\theta}(W_{xh}\boxed{x_t} + W_{hh}\boxed{h_{t-1}})$$

New **state**

Recurrent function

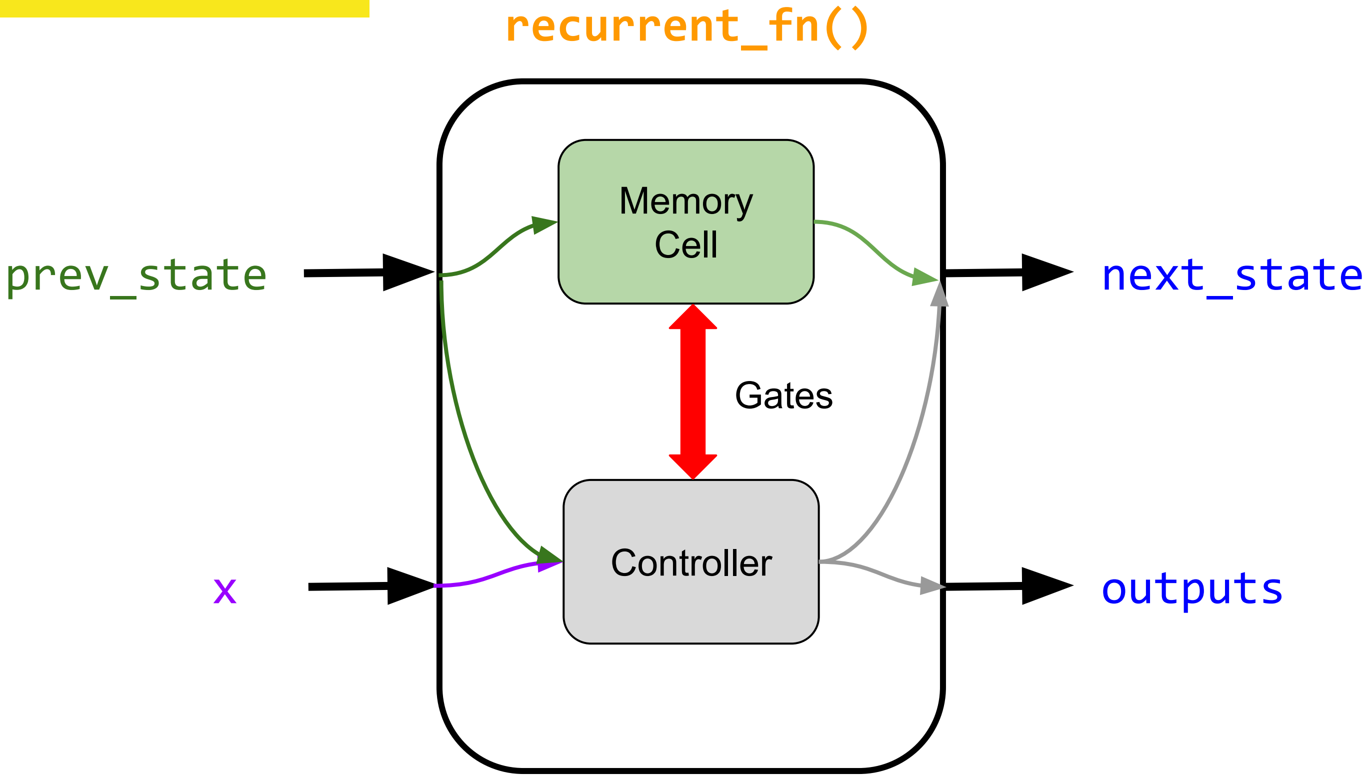
Input at current time-step

Previous **state**



The GATED RNN API

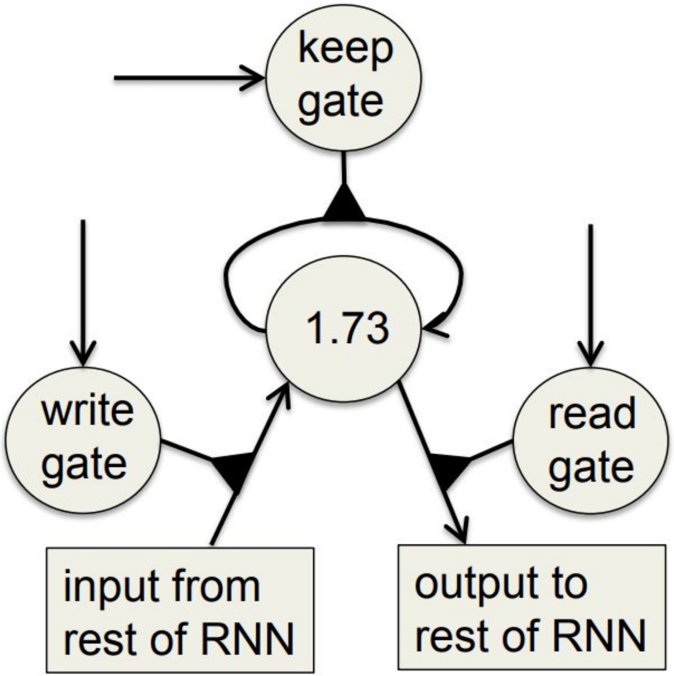
It is the same! Just a different way of computing the outputs.



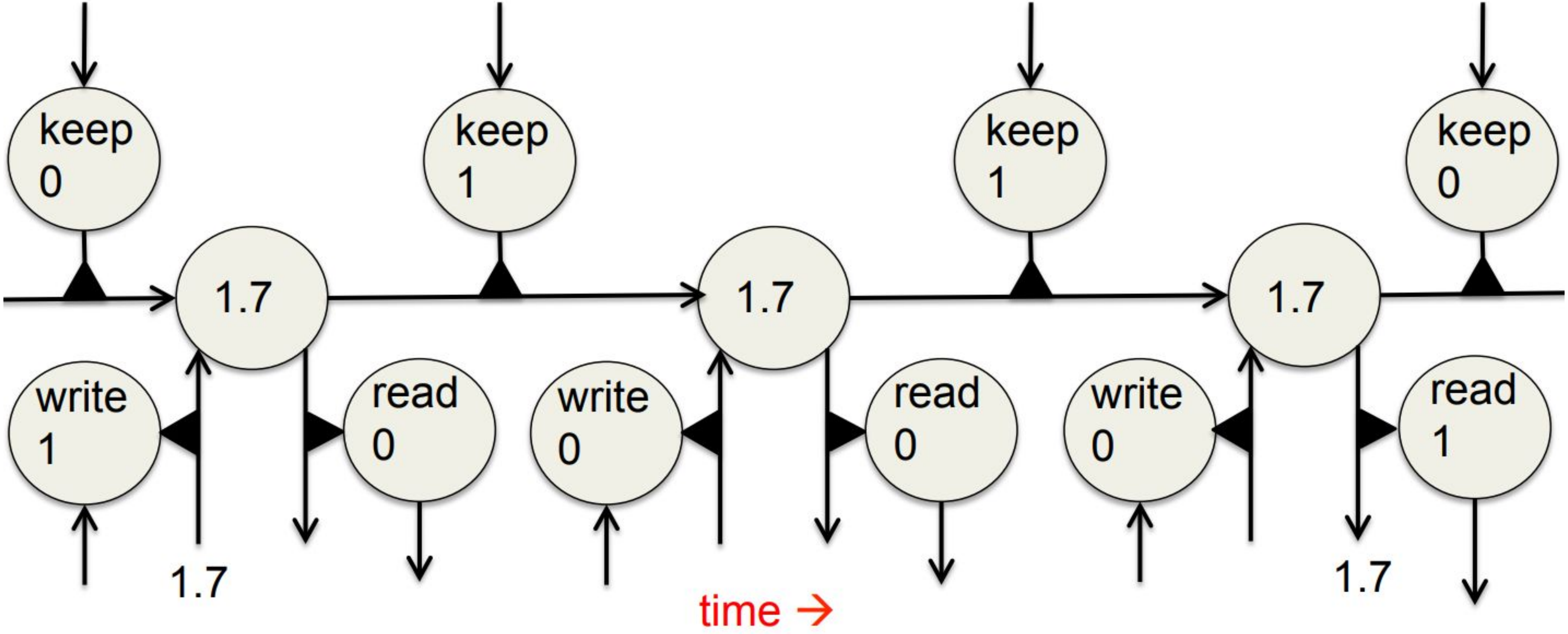
Implementing a memory cell in a neural network

To preserve information for a long time in the activities of an RNN, we use a circuit that implements an analog memory cell.

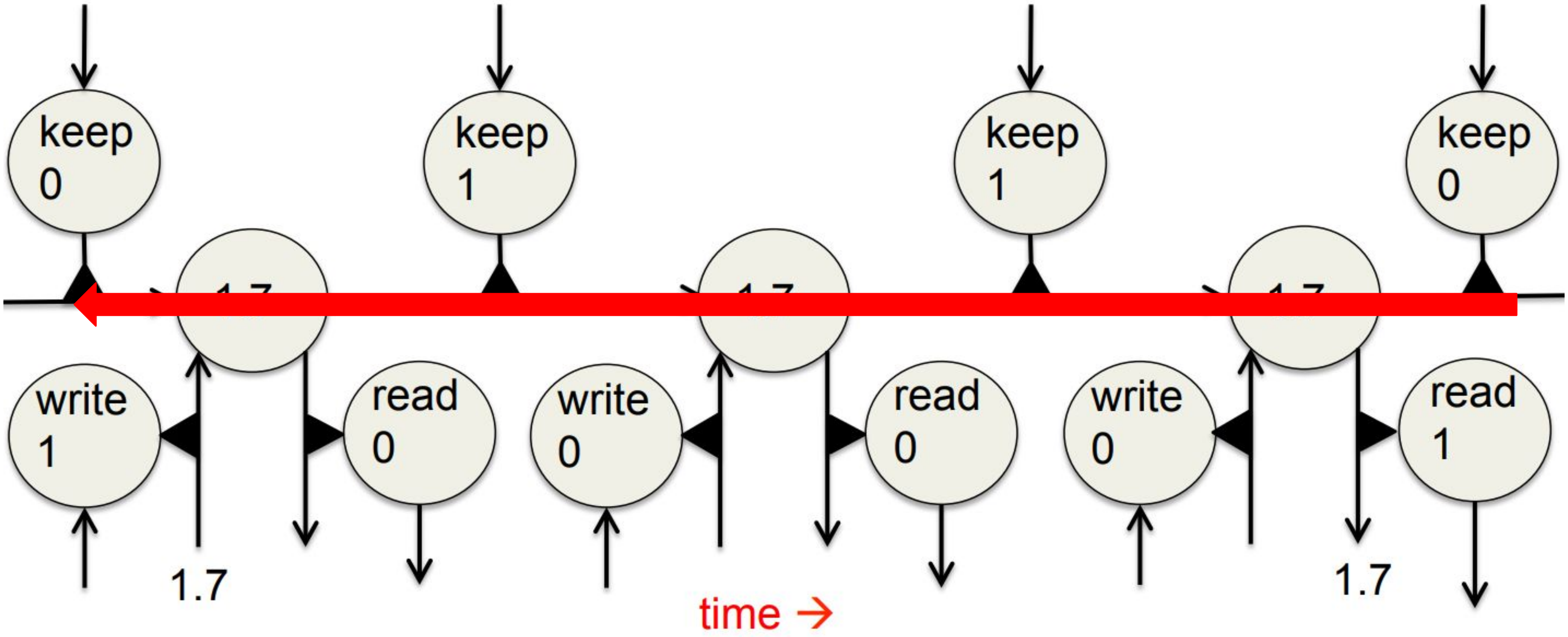
- A linear unit that has a self-link with a weight of 1 will maintain its state.
- Information is stored in the cell by activating its write gate.
- Information is retrieved by activating the read gate.
- We can backpropagate through this circuit because logistics are have nice derivatives.



Propagating through a memory cell



Backpropagating through a memory cell?



LSTM

LSTM = Long Short Term Memory

vector sizes

concatenate : $X = X_t \mid H_{t-1}$ $p+n$

forget gate : $f = \sigma(X.W_f + b_f)$ n

update gate : $u = \sigma(X.W_u + b_u)$ n

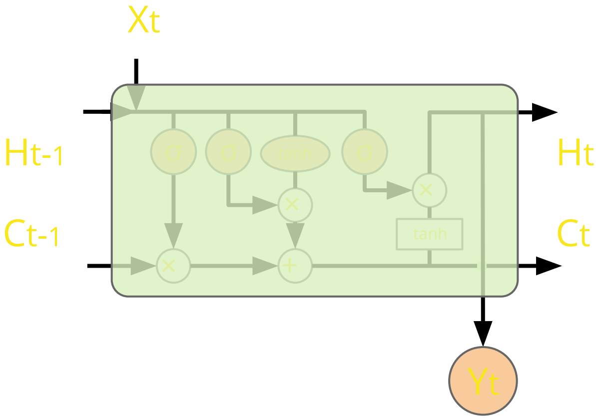
result gate : $r = \sigma(X.W_r + b_r)$ n

input : $X' = \tanh(X.W_c + b_c)$ n

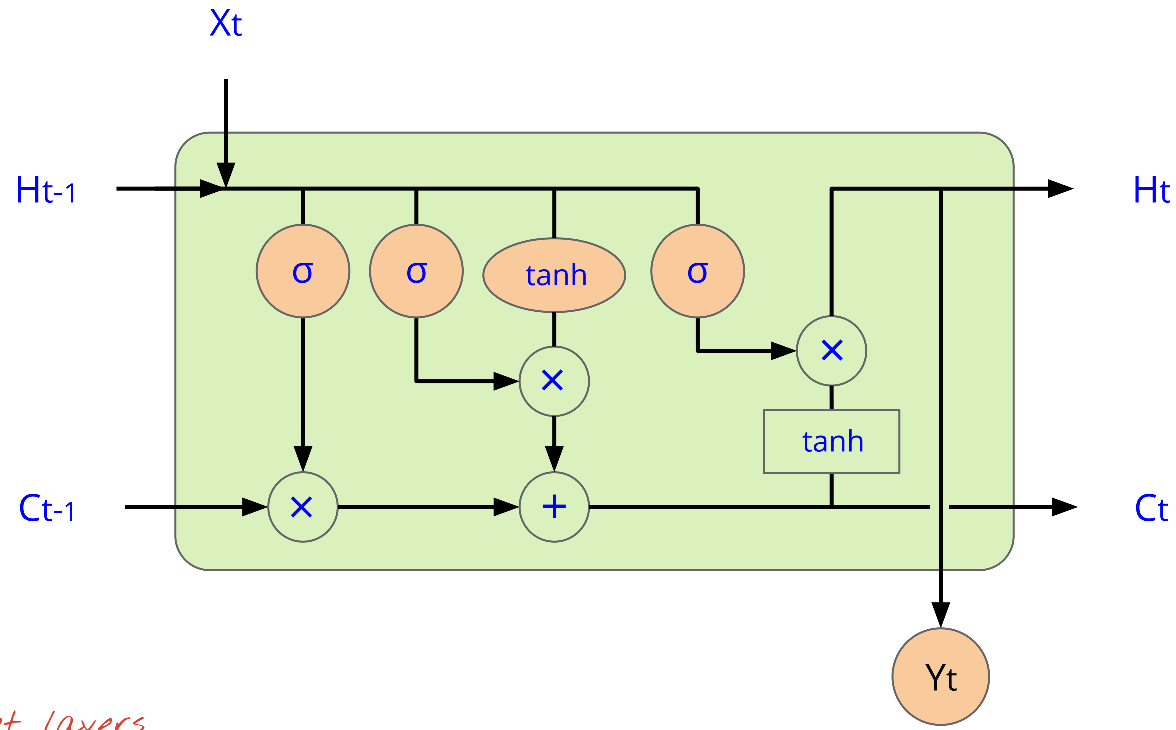
new C : $C_t = f * C_{t-1} + u * X'$ n

new H : $H_t = r * \tanh(C_t)$ n

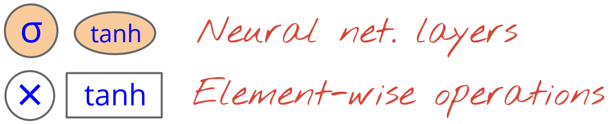
output : $Y_t = \text{softmax}(H_t.W + b)$ m



LSTM

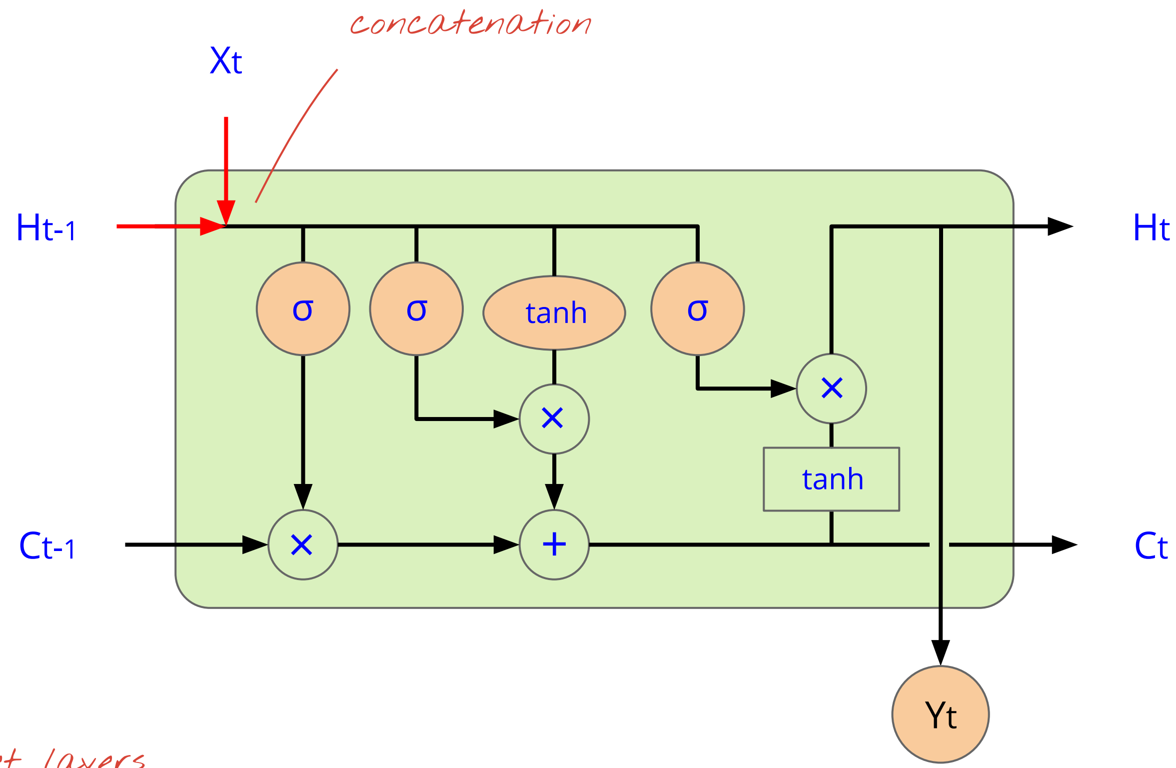


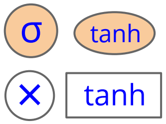
Xt
Ht-1
Ht
σ
σ
tanh
σ
×
×
tanh
Ct-1
×
+
Ct
Yt



σ
tanh
Neural net. layers
×
tanh
Element-wise operations

LSTM

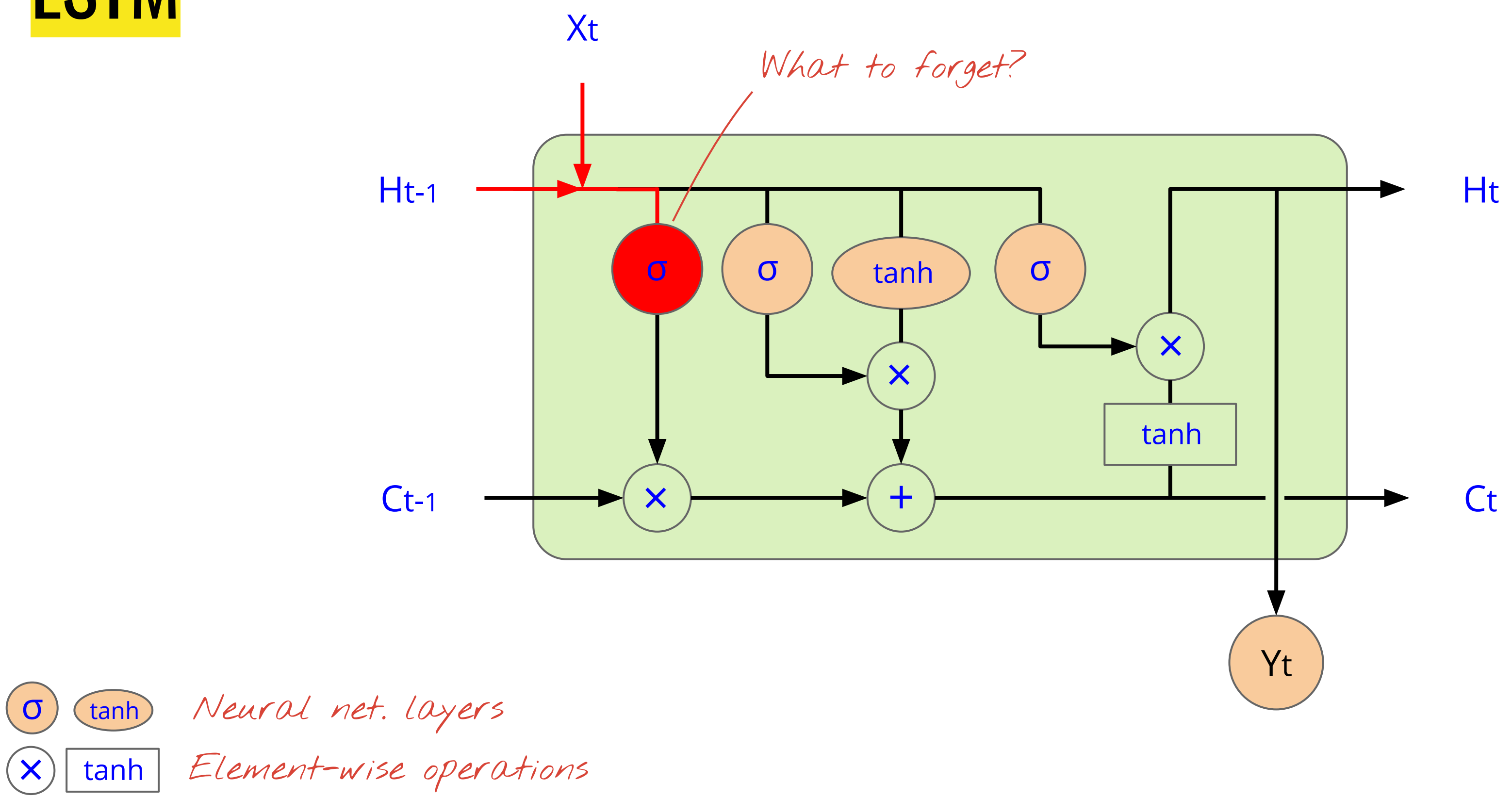




Neural net. layers

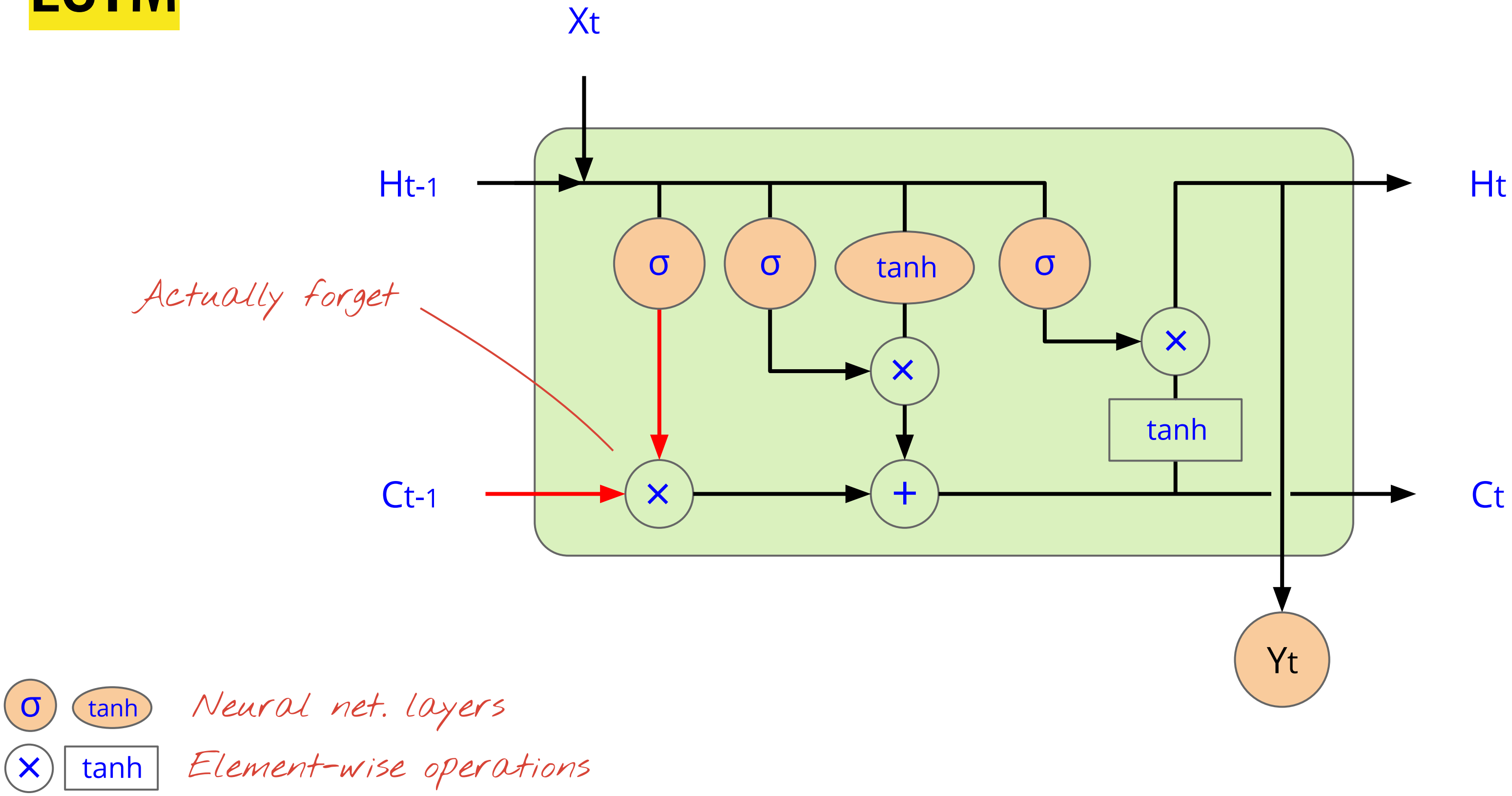
Element-wise operations

LSTM

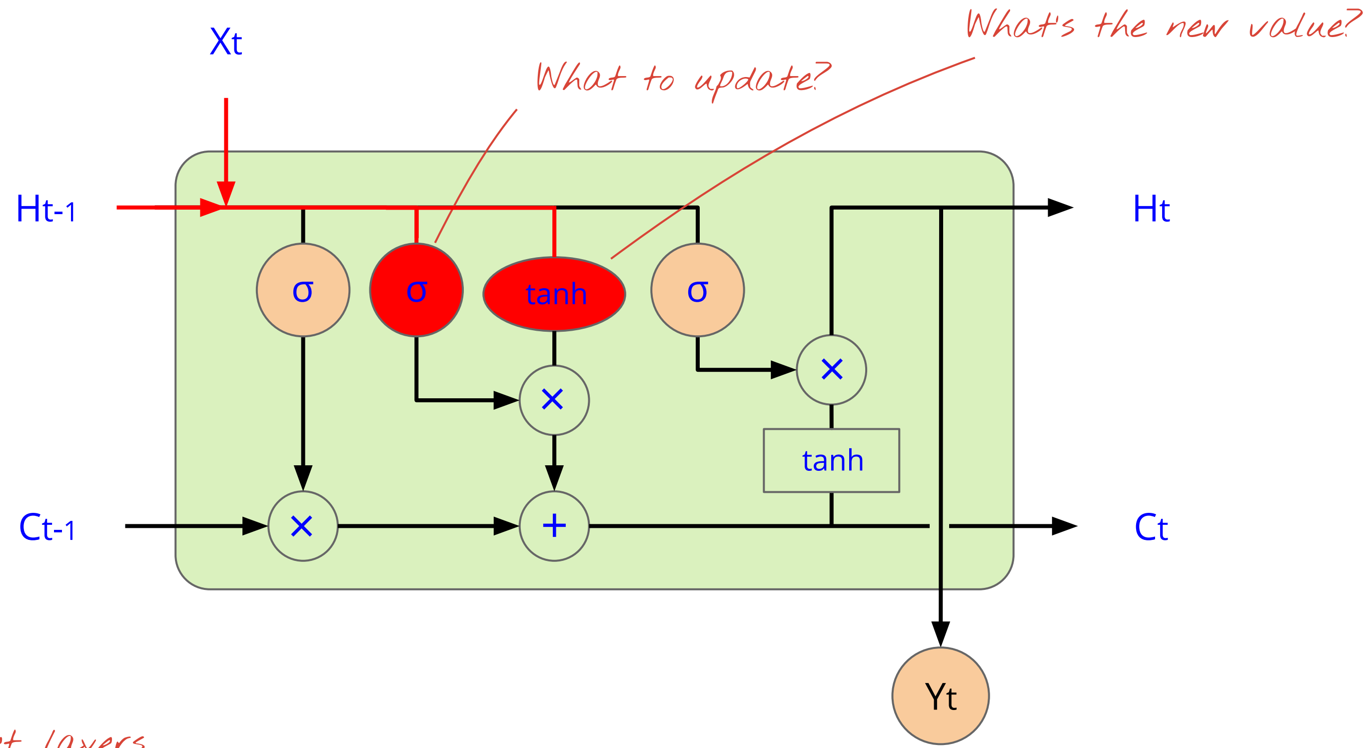


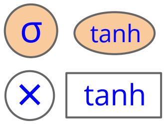
Xt
What to forget?
Ht-1
σ
σ
tanh
σ
×
×
tanh
Ht
Ct-1
×
+
Ct
Yt
σ
tanh
Neural net. layers
×
tanh
Element-wise operations

LSTM


Xt
Ht-1
Ht
Ct-1
Ct
Yt
σ
σ
tanh
σ
×
×
×
+
tanh
Actually forget
σ
tanh
Neural net. layers
×
tanh
Element-wise operations

LSTM



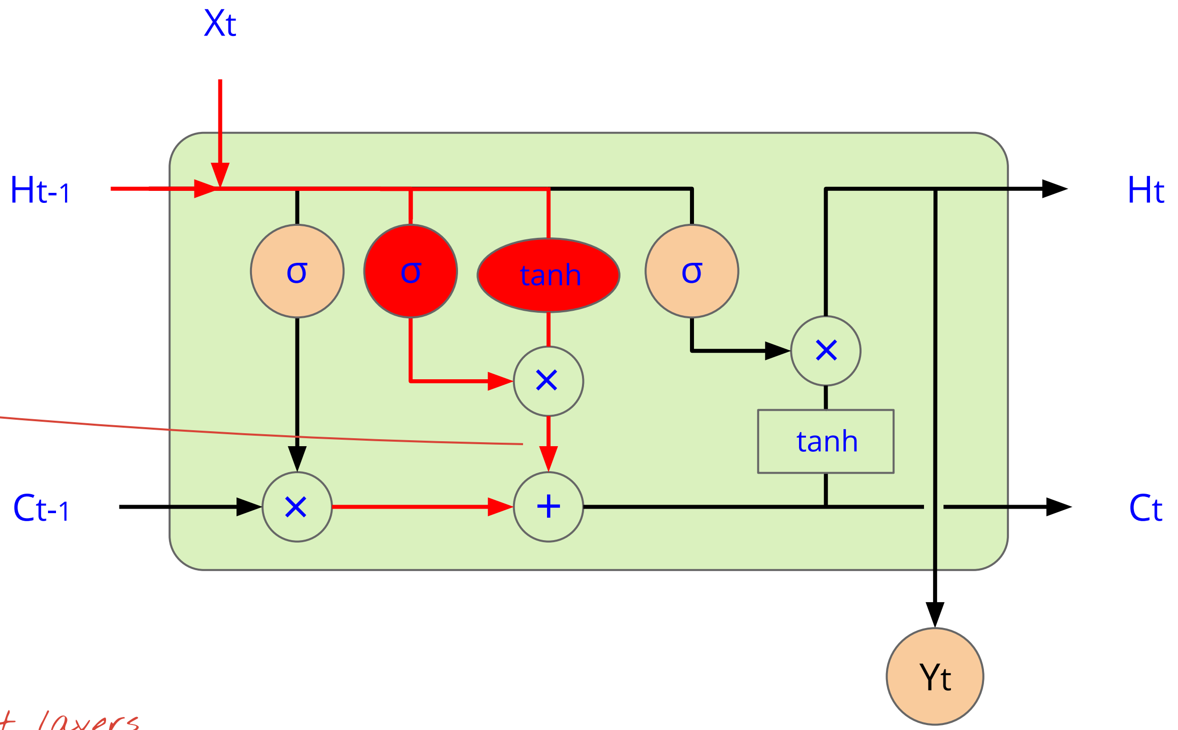


Neural net. layers

Element-wise operations

LSTM

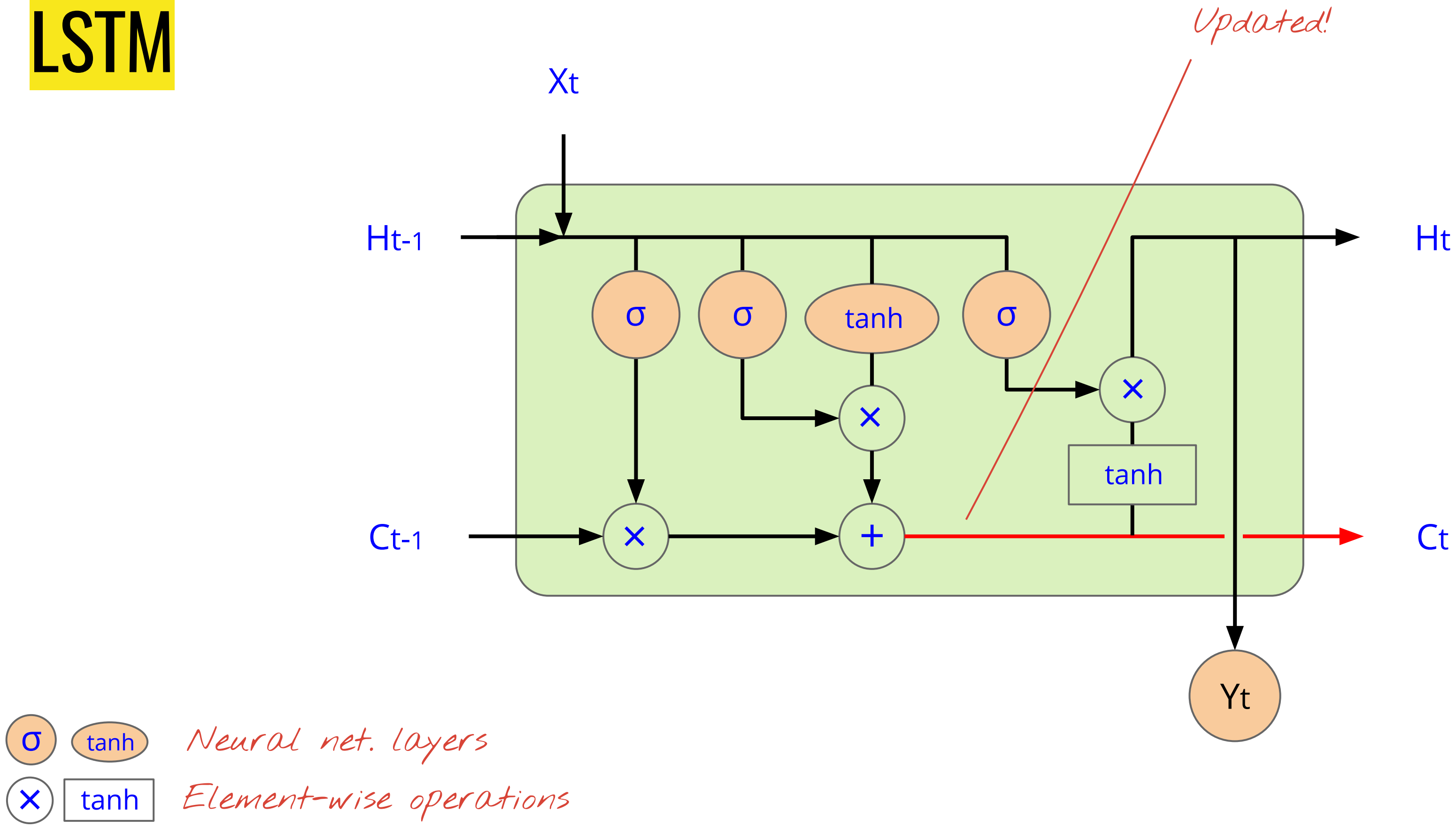
Actually update



σ tanh Neural net. layers

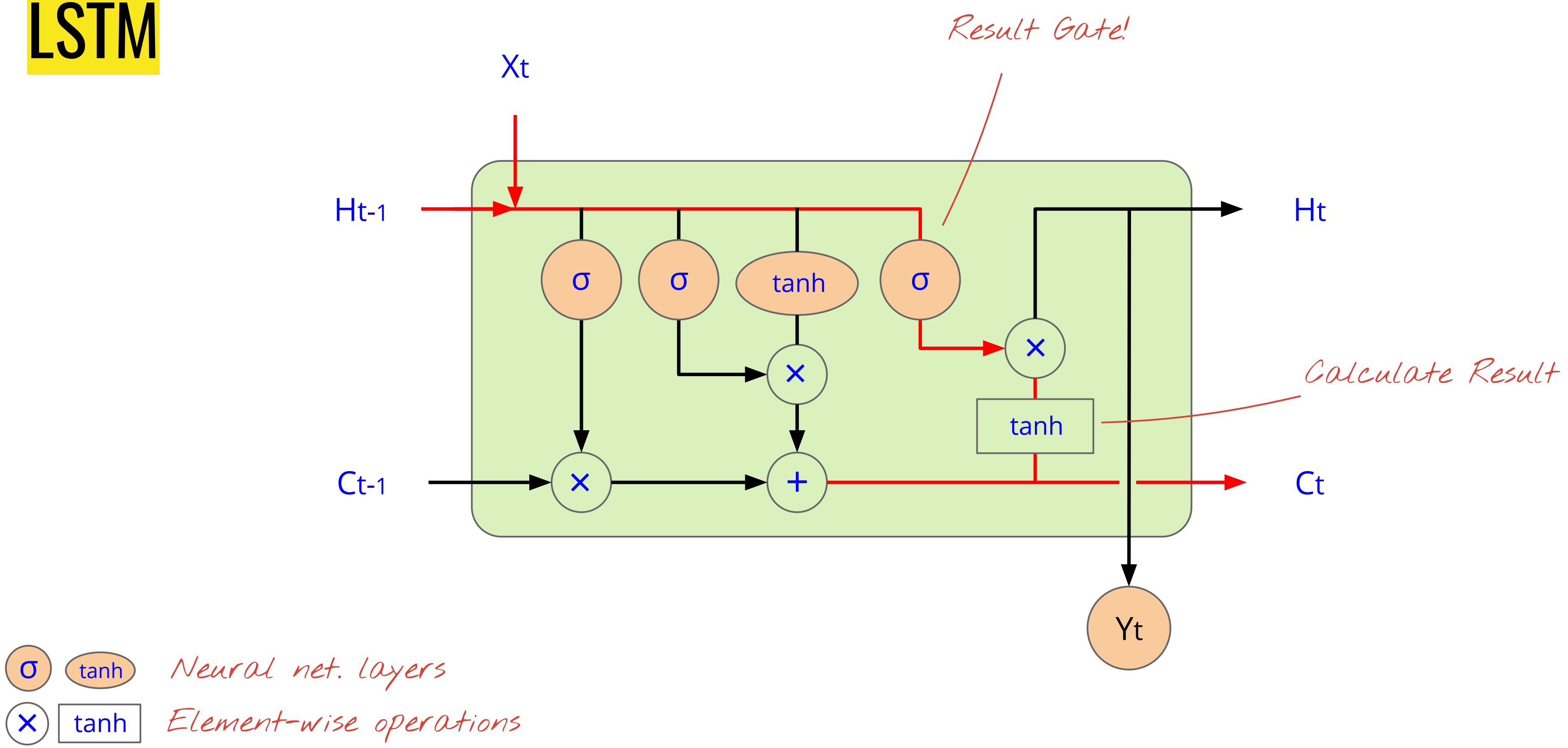
× tanh Element-wise operations

LSTM



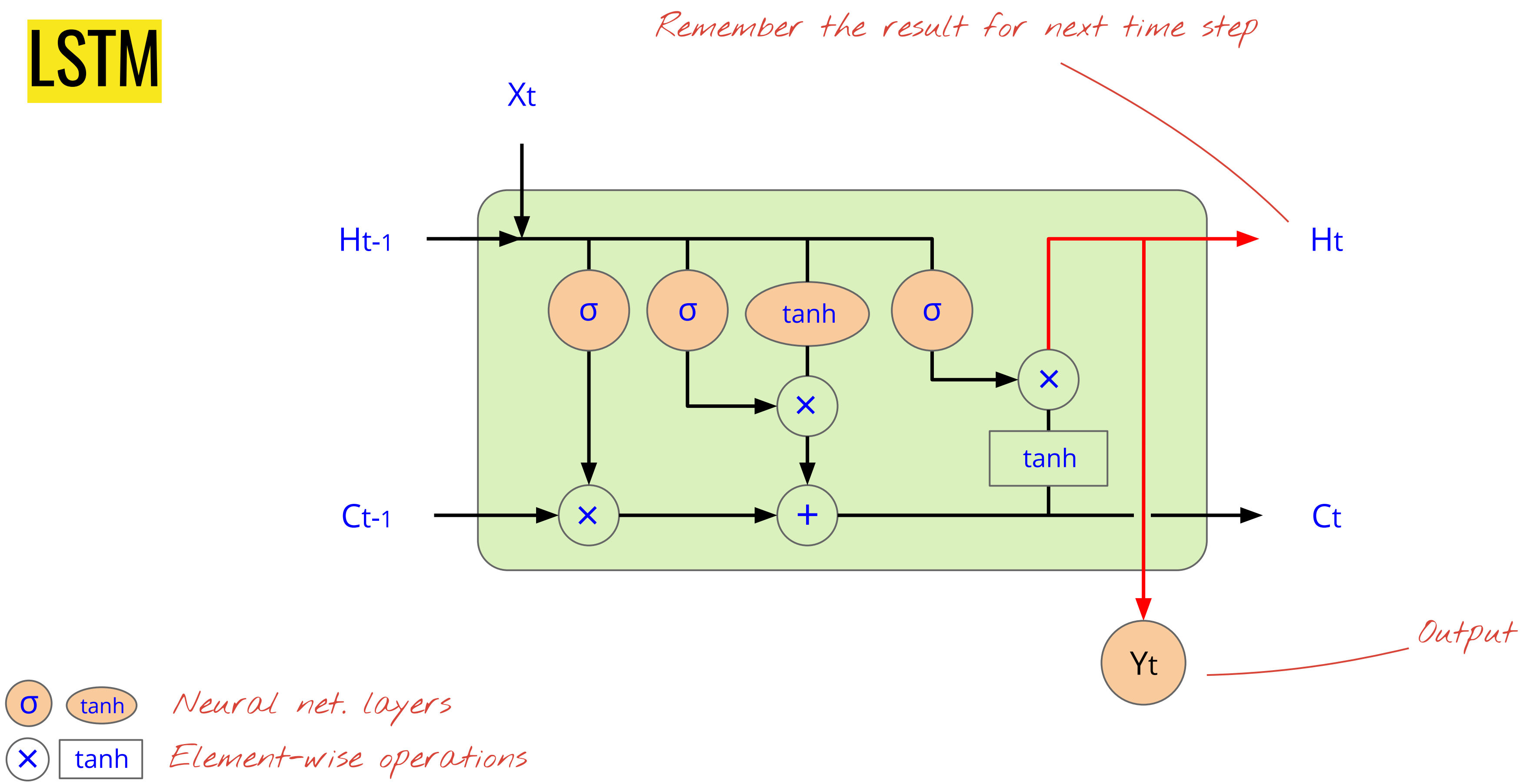
Updated!
Xt
Ht-1
Ht
Ct-1
Ct
σ
σ
tanh
σ
×
×
×
+
tanh
Yt
σ
tanh
Neural net. layers
×
tanh
Element-wise operations

LSTM



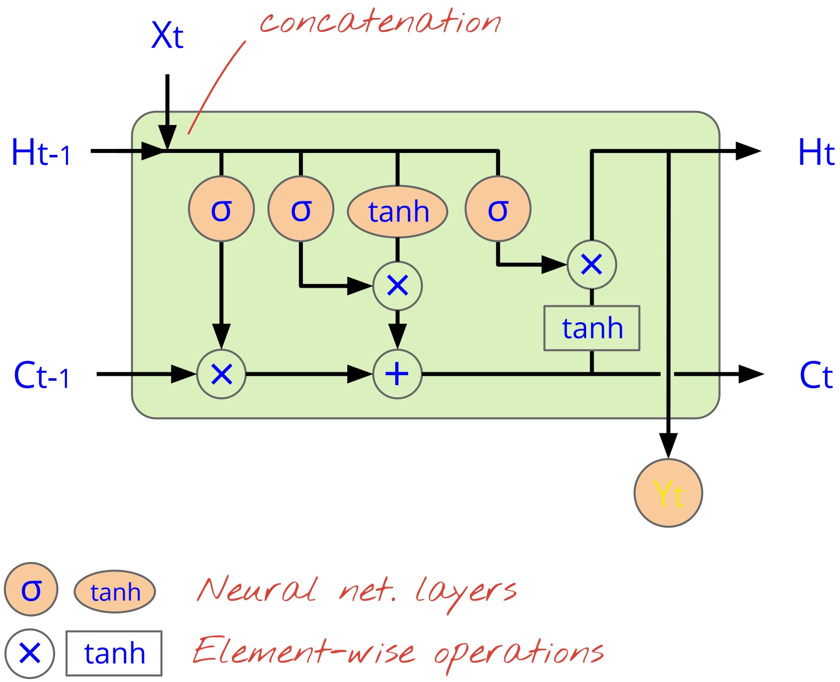
Result Gate!
Xt
Ht-1
σ
σ
tanh
σ
×
×
Calculate Result
tanh
Ht
Ct-1
×
+
Ct
Yt
σ
tanh
Neural net. layers
×
tanh
Element-wise operations

LSTM


Remember the result for next time step
Xt
Ht-1
Ht
σ
σ
tanh
σ
×
×
tanh
Ct-1
×
+
Ct
Yt
Output
σ
tanh
Neural net. layers
×
tanh
Element-wise operations

LSTM

LSTM = Long Short Term Memory



$$X = X_t \oplus H_{t-1}$$

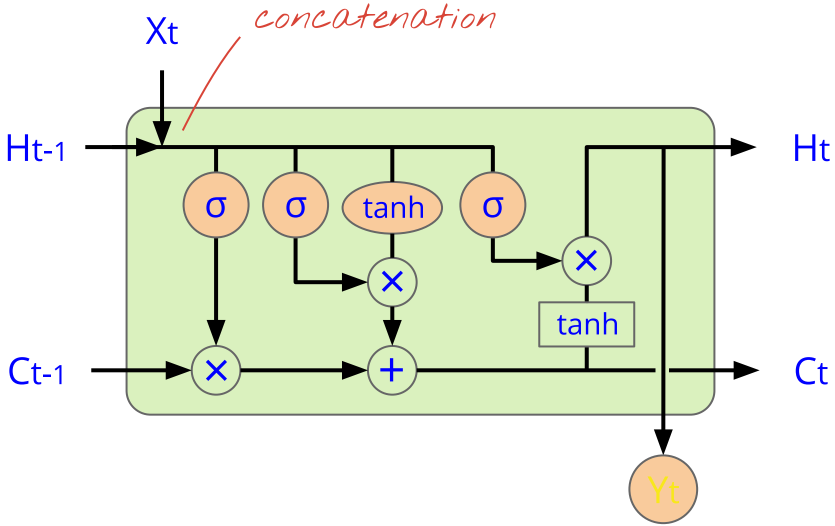
$$f = \sigma(X.W_f + b_f)$$
$$u = \sigma(X.W_u + b_u)$$
$$r = \sigma(X.W_r + b_r)$$
$$X' = \tanh(X.W_c + b_c)$$
$$C_t = f * C_{t-1} + u * X'$$
$$H_t = r * \tanh(C_t)$$

$$Y_t = \mathrm{softmax}(H_t.W + b)$$

LSTM

LSTM = Long Short Term Memory

$$X = X_t \oplus H_{t-1}$$



$$f = \sigma(X.W_f + b_f)$$

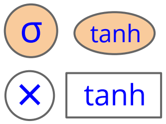
$$u = \sigma(X.W_u + b_u)$$

$$r = \sigma(X.W_r + b_r)$$

$$X' = \tanh(X.W_c + b_c)$$

$$C_t = f * C_{t-1} + u * X'$$

$$H_t = r * \tanh(C_t)$$



Neural net. layers

Element-wise operations

$$Y_t = \text{softmax}(H_t.W + b)$$


Gru !

Gated Recurrent Units (GRUs)

GRU = Gated Recurrent Unit

2 gates instead of 3 => cheaper

vector sizes

$$X = X_t \mid H_{t-1}$$ p+n

$$z = \sigma(X.W_z + b_z)$$ n

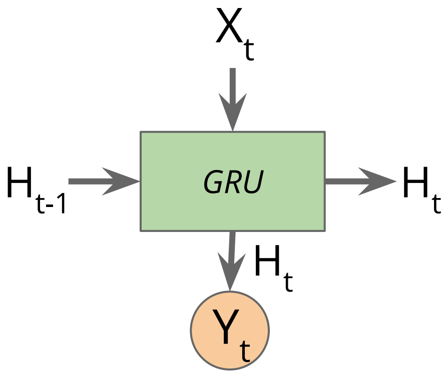
$$r = \sigma(X.W_r + b_r)$$ n

$$X' = X_t \mid r * H_{t-1}$$ p+n

$$X'' = \tanh(X'.W_c + b_c)$$ n

$$H_t = (1-z) * H_{t-1} + z * X''$$ n

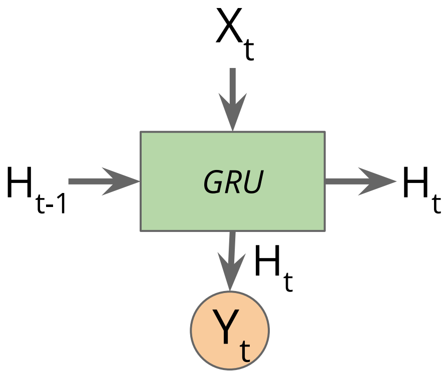
$$Y_t = \mathrm{softmax}(H_t.W + b)$$ m



Gated Recurrent Units (GRUs)

GRU = Gated Recurrent Unit

2 gates instead of 3 => cheaper



vector sizes

$X = X_t \,|\, H_{t-1}$ p+n

$z = \sigma(X.W_z + b_z)$ n

$r = \sigma(X.W_r + b_r)$ n

$X' = X_t \,|\, r * H_{t-1}$ p+n

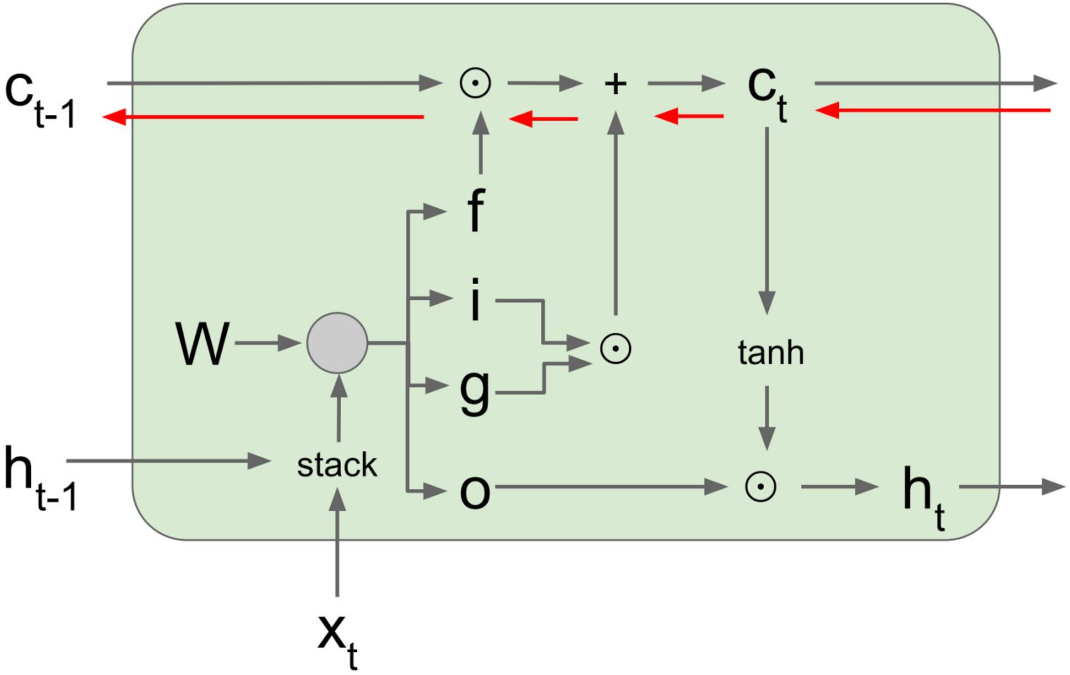
$X'' = \tanh(X'.W_c + b_c)$ n

$H_t = (1-z) * H_{t-1} + z * X''$ n

$Y_t = \text{softmax}(H_t.W + b)$ m

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

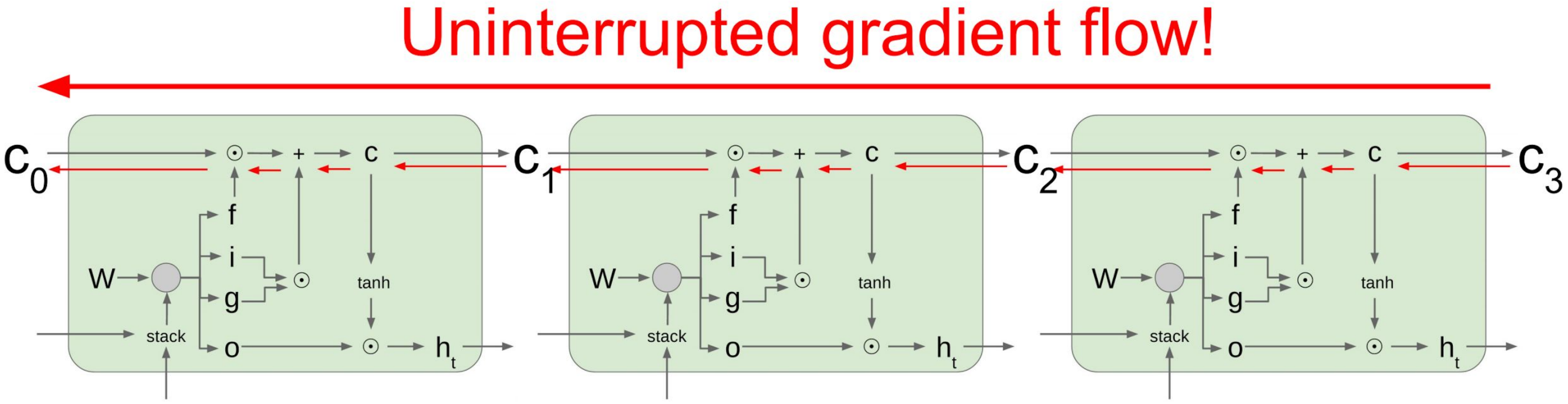
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

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

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

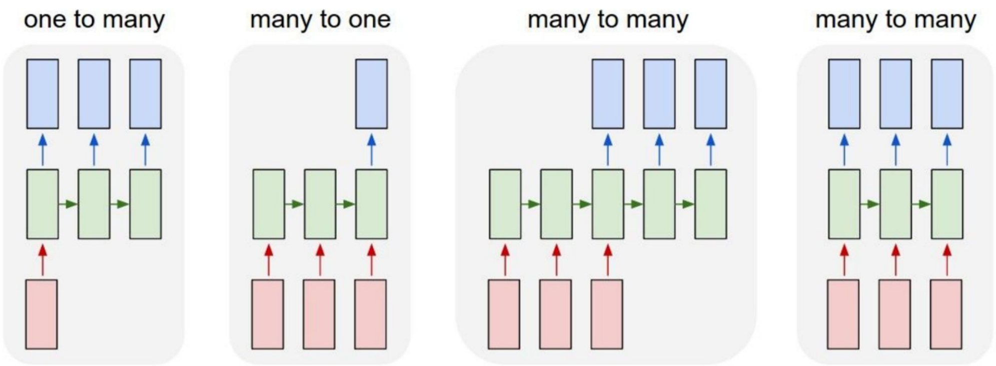
Long Short-term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]



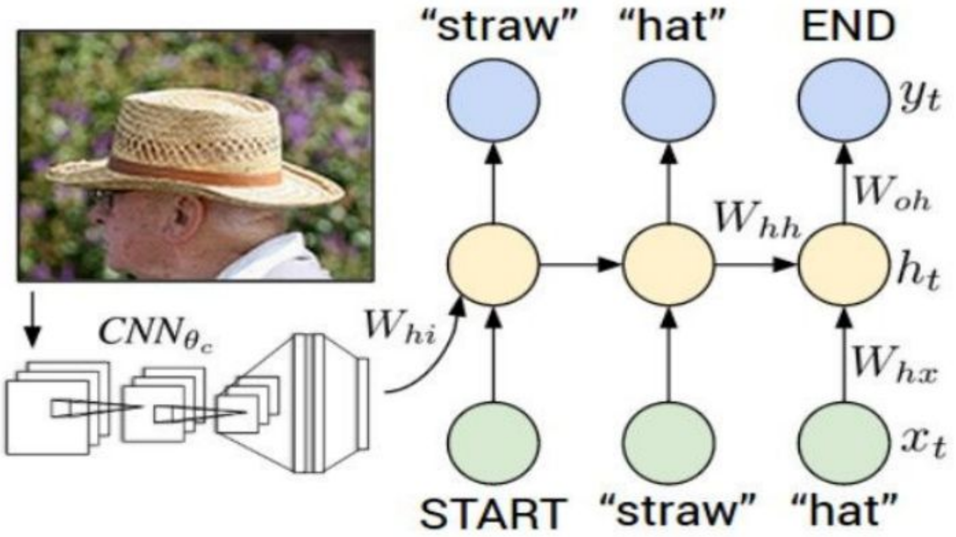
Applications/Tasks

- Image captioning
- Sequence classification (Practical 4: MNIST)
- Language modeling
- Sequence-labeling (lots of NLP tasks, e.g. POS tagging, NER, ...)
- Sequence-to-sequence learning (Machine translation, Summarization, ...)

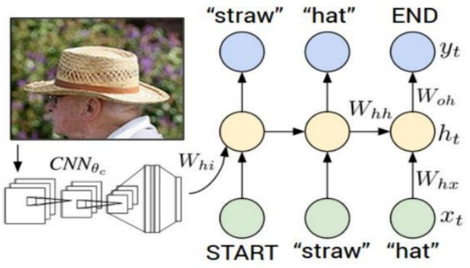


One-to-Many: Image captioning

GOAL: Given image, generate a sentence to describe its content.



One-to-Many: Image captioning



GOAL: Given image, generate a sentence to describe its content.

"man in black shirt is playing guitar."

"construction worker in orange safety vest is working on road."

"two young girls are playing with lego toy."

"boy is doing backflip on wakeboard."

Many-to-one: Sequence Classifier (Prac 4)

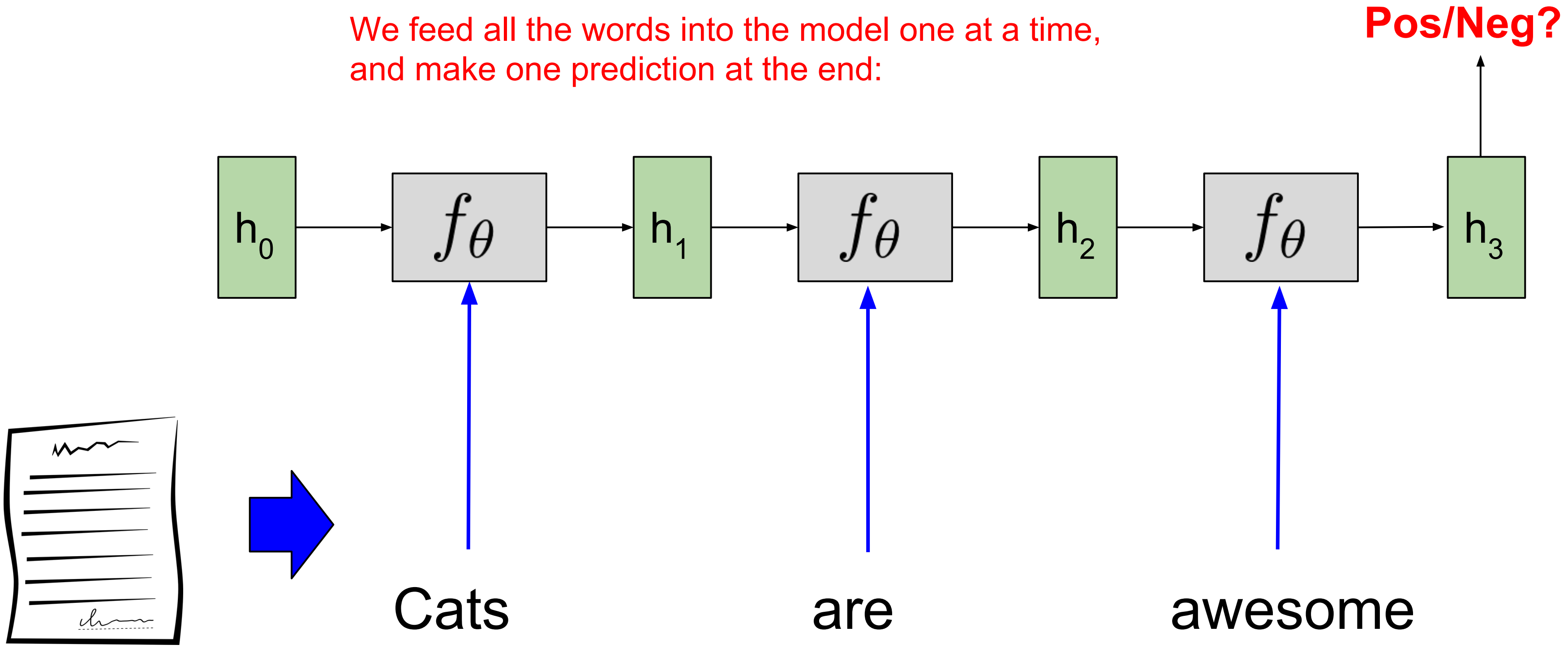
GOAL: Given a sequence of inputs, predict the label for the whole sequence.

Examples:

- Given a sentence, say if it is **{negative, neutral, positive}**
- Given the words in an email, predict if it is a **spam message**.
- Given "pieces" of an image, predict **what number** is in the image.

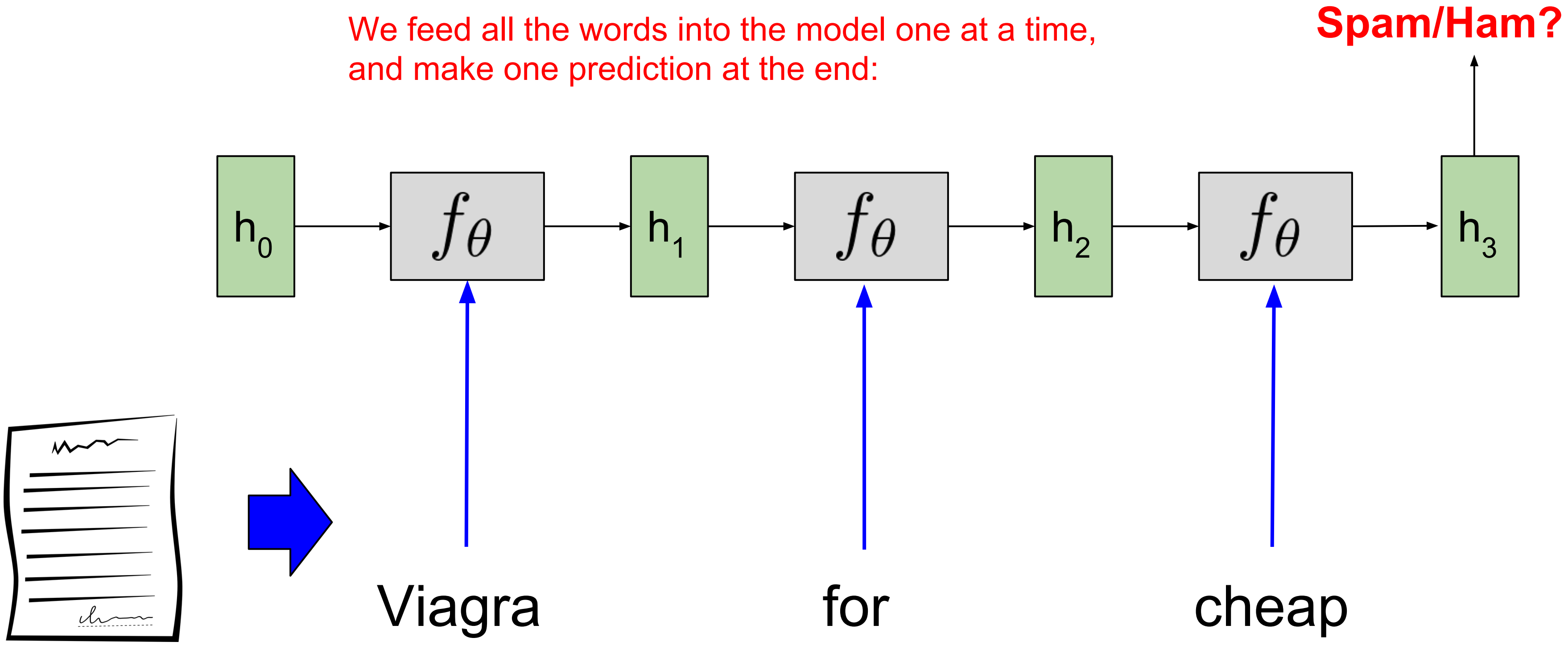
Many-to-1: Polarity/Sentiment Classifer

We feed all the words into the model one at a time,
and make one prediction at the end:



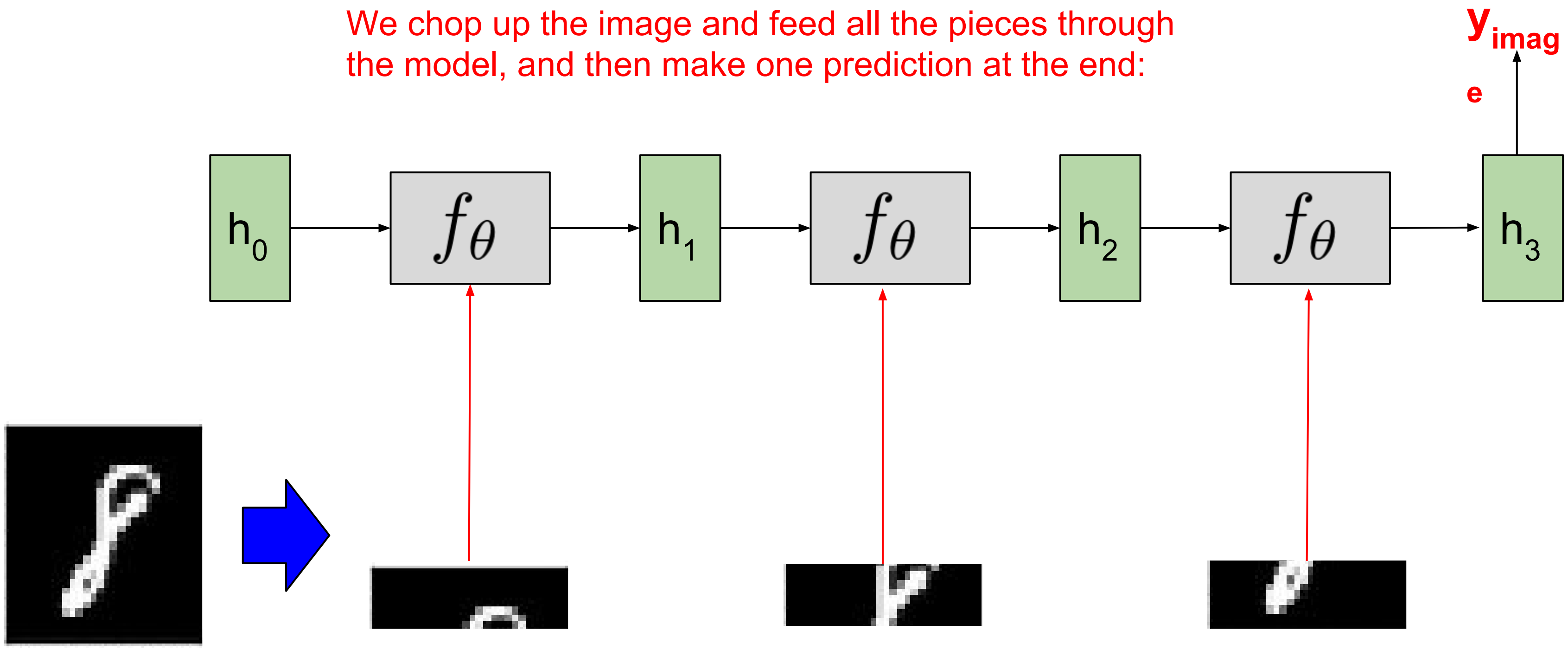
Many-to-1: Spam Classifer

We feed all the words into the model one at a time,
and make one prediction at the end:



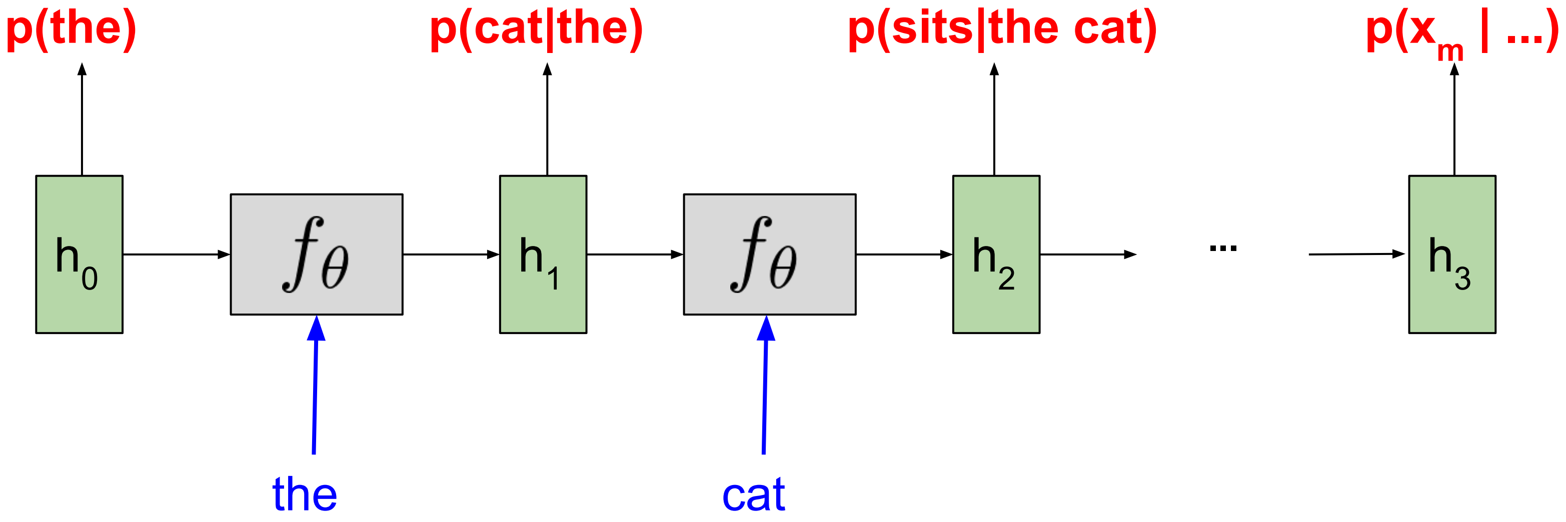
Many-to-1: Image Classifier (Prac 4)

We chop up the image and feed all the pieces through the model, and then make one prediction at the end:

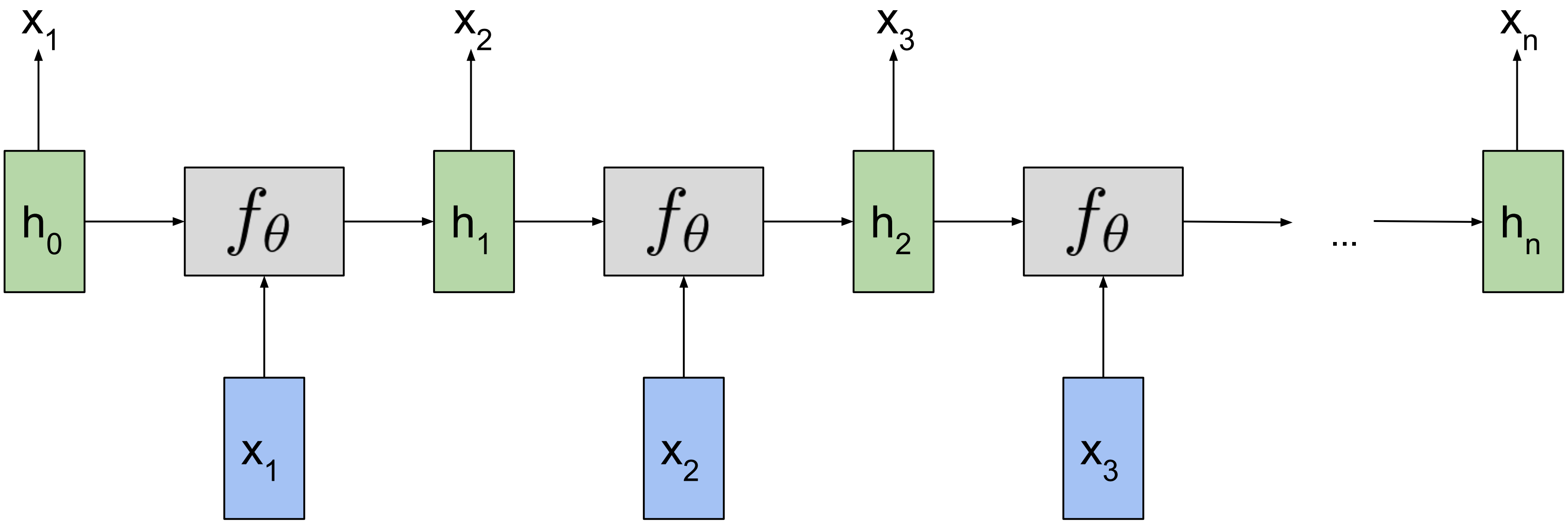


Next-token Prediction: Language modeling

1. Computing p(next word | previous words)
2. $$p(x_1, x_2, \ldots, x_m) = \prod_{i}^{m} p(x_i | x_1 \ldots x_{i-1})$$



Next-token Prediction: Language modeling

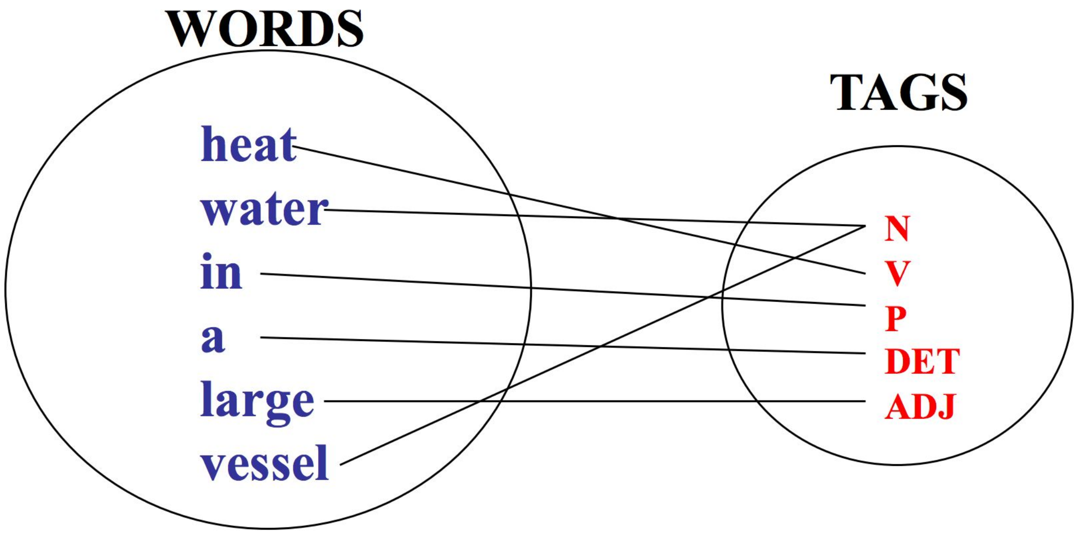


Many-to-many: Sequence labeling

- Mapping each input $x_1, x_2, \ldots, x_n$ to its own label $y_1, y_2, \ldots, y_n$
- (Notice: Same length m; each input has an output.)
- A lot of NLP Tasks fall in this category, e.g.:
  - **Part-of-speech tagging**: map words to their parts-of-speech (noun, verb, etc).
  - **Named-entity Recognition**: identify mentions of people, places, etc in text
  - **Semantic Role Labeling**: find the main **actions**, and **who** performs them on **whom/what**

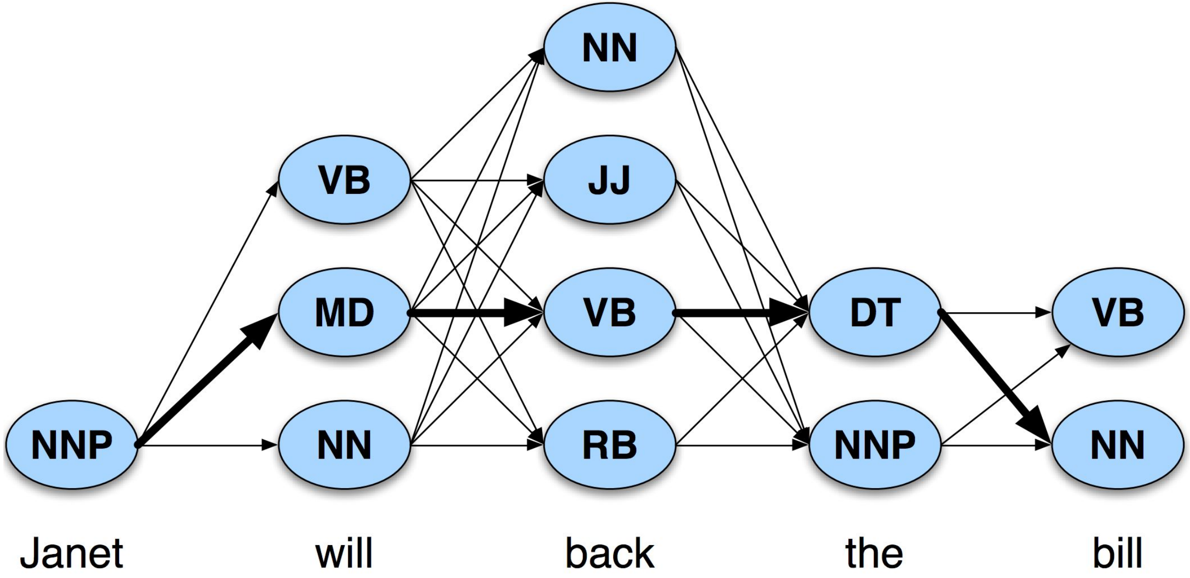
Many-to-many: Sequence labeling

- **Part-of-speech tagging**



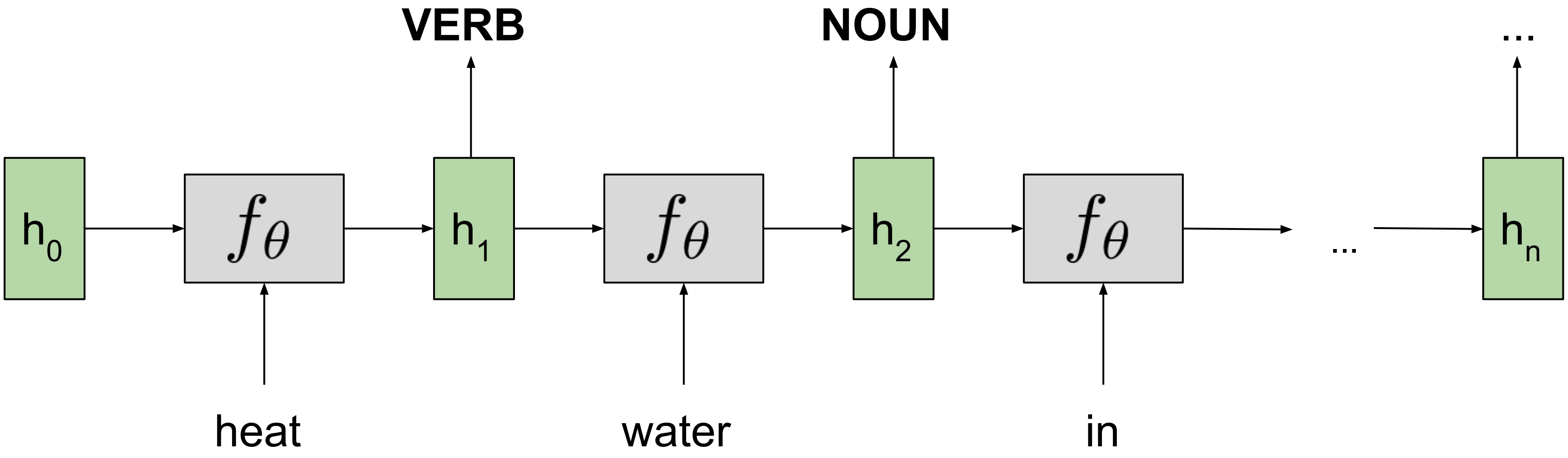
Many-to-many: Sequence labeling

- **Part-of-speech tagging**



Many-to-many: Sequence labeling

Heat water in a large vessel.

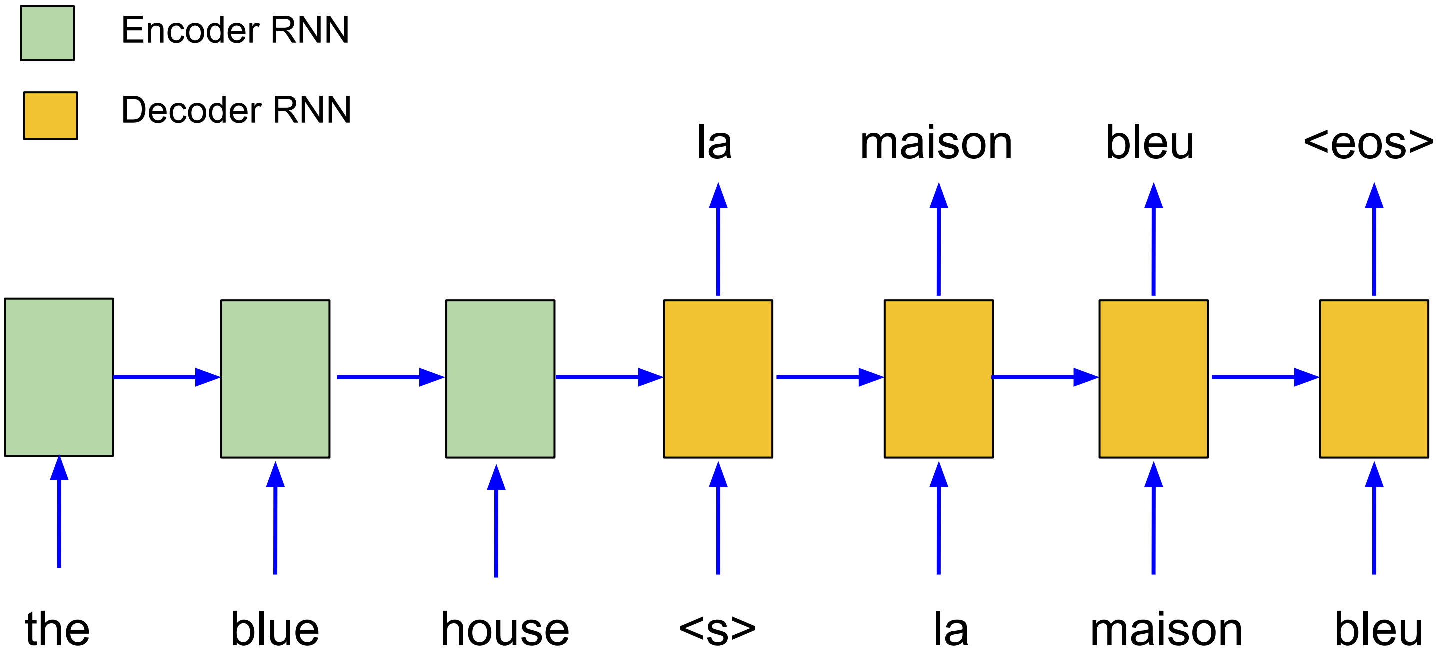


Many-to-many: Sequence-to-Sequence Modeling

- Mapping input sequences $\{x_1, x_2, \ldots, x_m\}$ to output sequences $\{y_1, y_2, \ldots, y_n\}$.
- Note: Arbitrary m and n.
- Many applications, most notably Machine Translation.
- Also: POS tagging, Parsing, Summarization, Dialog

Many-to-many: Sequence-to-Sequence Modeling

MACHINE TRANSLATION



Many-to-many: Sequence-to-Sequence Modeling

DIALOGUE MODELING

Prompt: You can't eat pasta every day. You can't live only on Italian food.

System: I don't eat Italian food, but I do eat a lot of Italian food.

Many-to-many: Sequence-to-Sequence Modeling

Prompt: What do you have planned tonight?

Baseline: I have no idea what I want to do with my life. I have no idea what I want to (Bad, Mediocre, Bad)

Ours: I was planning on doing a few of these, but I’m not sure what to choose. I need to find a place to camp, and I don’t have a lot of time to make a plan. (Excellent, Good, Mediocre)

Many-to-many: Sequence-to-Sequence Modeling

Prompt: Have you seen *Suicide Squad*?

Baseline: I have not, but I will check it out. (Good, Good, Good)

Ours: I saw it in a theater once and it was great. It was very… I don't know, a little dark. I like the psychological effects and the way it portrays the characters in the movie. (Excellent, Excellent, Excellent)

Key take-aways

- RNNs have memory/state that evolve over time.
- We unroll the graph over time to do forward propagation.
- Backprop-through-time (BPTT):
  - Perform Chain Rule over the unrolled graph efficiently by saving and reusing previous computations.
  - dE/dW is sum over all time-steps (b/c of tied weights)
- Vanishing/exploding gradients
- Gated architectures
  - State is *selectively* overwritten per time-step
  - Uninterrupted gradient flow through time: no vanishing/exploding gradients!
- Lots of cool applications!

Slide Credits

Thank-you to the following resources, from which some of these slides were drawn and adapted.

- Stanford CS231n
- TensorFlow without a PhD

The end.