CS189 Summer 2018

Lecture 25: Recurrent Neural Networks

Josh Tobin

Where we are in the course

- Final covers up to **yesterday's** lecture
- Today and tomorrow: advanced neural networks
- Not covered: boosting (not on the final)
- Next week: enrichment lectures (not on the final)
- Also next week: Final

How to study for the final

- Start today!
- Do as many practice problems as possible
  - Midterm problems
  - HW problems
  - Old midterm / final problems
- Go through the derivations from class
- Make sure you can re-derive things on your own
- Go to office hours and ask questions
- (Hint): make sure you understand the geometric intuition for the methods we studied. E.g., make sure you can draw a picture of what's happening

Goal for this lecture

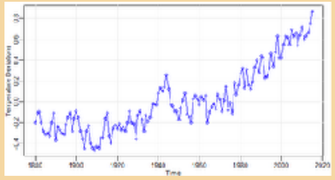
Describe a general-purpose architecture
for handling sequential data

Agenda

1. **Why recurrent neural networks (RNNs)?**
2. RNNs
3. Problem with RNNs: vanishing gradients
4. Dealing with vanishing gradients

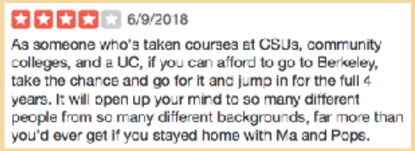
Sequence data is everywhere

Sequence

Application	Input data	Output data
Time series forecasting	Time series 	Predicted next value

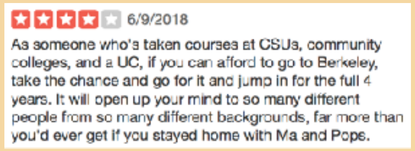
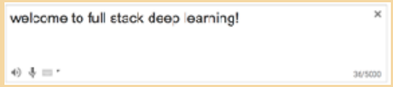
Sequence data is everywhere

Sequence

Application	Input data	Output data
Time series forecasting	Time series 	Predicted next value
Sentiment classification	Review text 	Predicted sentiment 

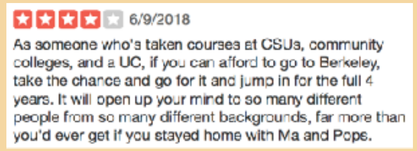
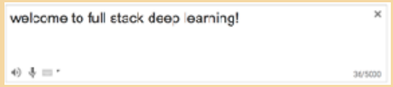
Sequence data is everywhere

Sequence

Application	Input data	Output data
Time series forecasting	Time series 	Predicted next value
Sentiment classification	Review text 	Predicted sentiment 
Machine translation	English text 	French text 

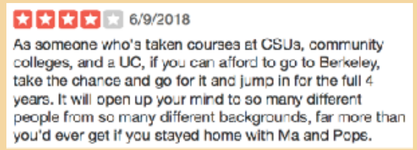
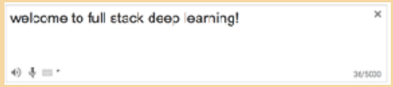
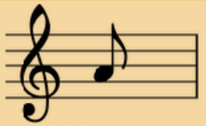
Sequence data is everywhere

Sequence

Application	Input data	Output data
Time series forecasting	Time series 	Predicted next value
Sentiment classification	Review text 	Predicted sentiment 
Machine translation	English text 	French text 
Speech recognition	Audio waveform 	Text

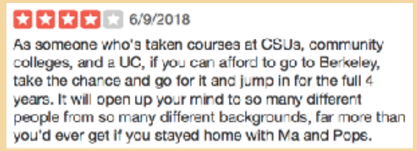
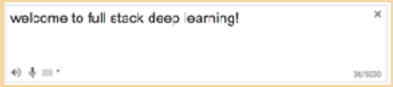
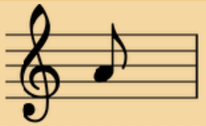
Sequence data is everywhere

Sequence

Application	Input data	Output data
Time series forecasting	Time series 	Predicted next value
Sentiment classification	Review text 	Predicted sentiment 
Machine translation	English text 	French text 
Speech recognition	Audio waveform 	Text
Music generation	Ø	

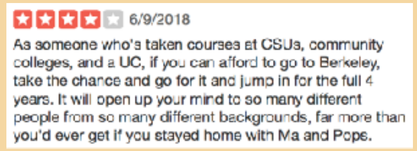
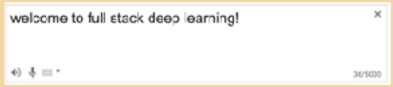
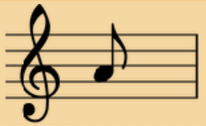
Sequence data is everywhere

Sequence

Application	Input data	Output data
Time series forecasting	Time series 	Predicted next value
Sentiment classification	Review text 	Predicted sentiment 
Machine translation	English text 	French text 
Speech recognition	Audio waveform 	Text
Music generation	Ø	
Image captioning	Image 	Description *“The quick brown fox jumped over the lazy dog”*

Sequence data is everywhere

Sequence

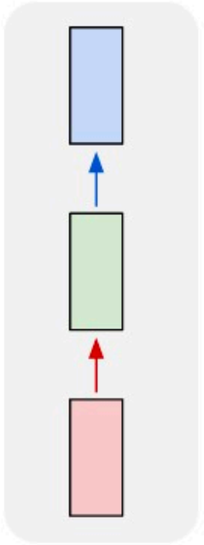
Application	Input data	Output data
Time series forecasting	Time series 	Predicted next value
Sentiment classification	Review text 	Predicted sentiment 
Machine translation	English text 	French text 
Speech recognition	Audio waveform 	Text
Music generation	Ø	
Image captioning	Image 	Description *“The quick brown fox jumped over the lazy dog”*

Sequence data is everywhere

- Lots of data is naturally sequential
- Sequences can form the input and/or output
- Three kinds of problems: many-to-one, one-to-many, many-to-many, none-to-many
- Even data that we don't think of temporally (like images) can be thought of as a sequence

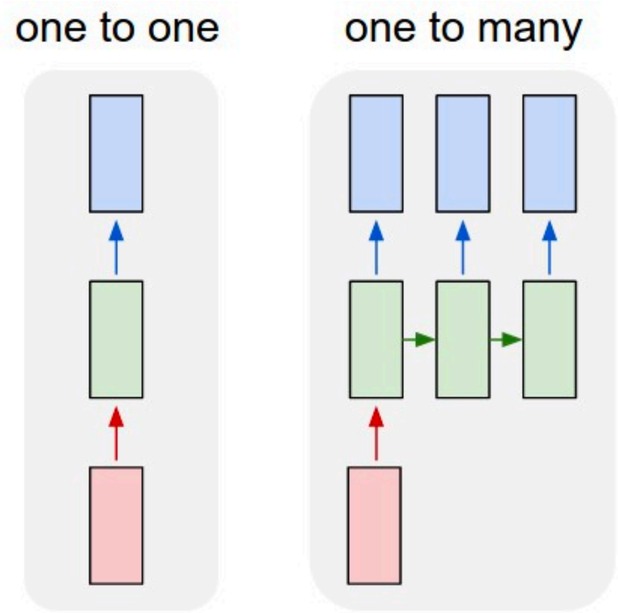
Types of sequence problems

one to one



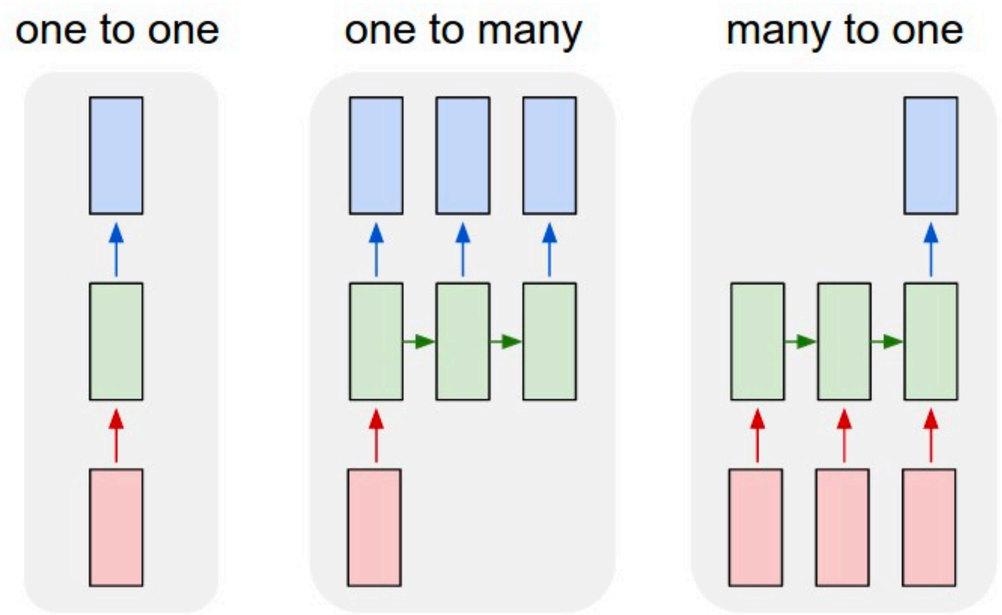
(From http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

Types of sequence problems



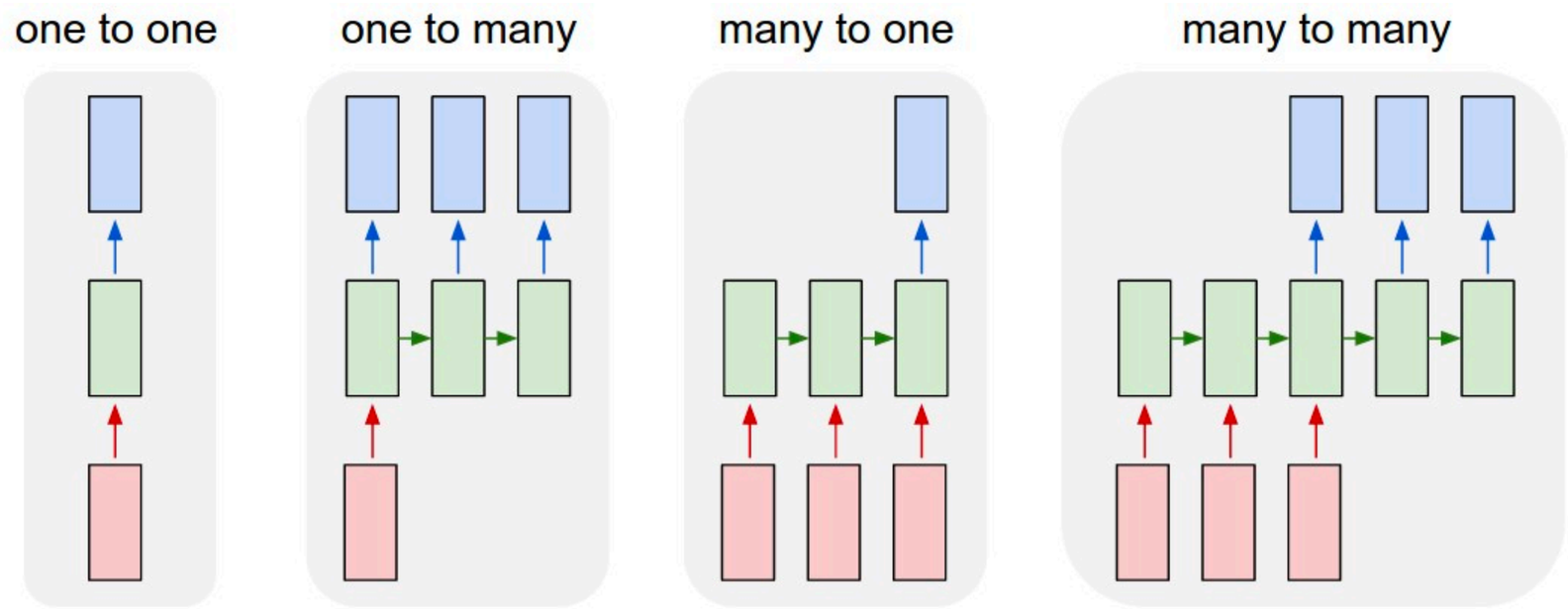
(From http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

Types of sequence problems



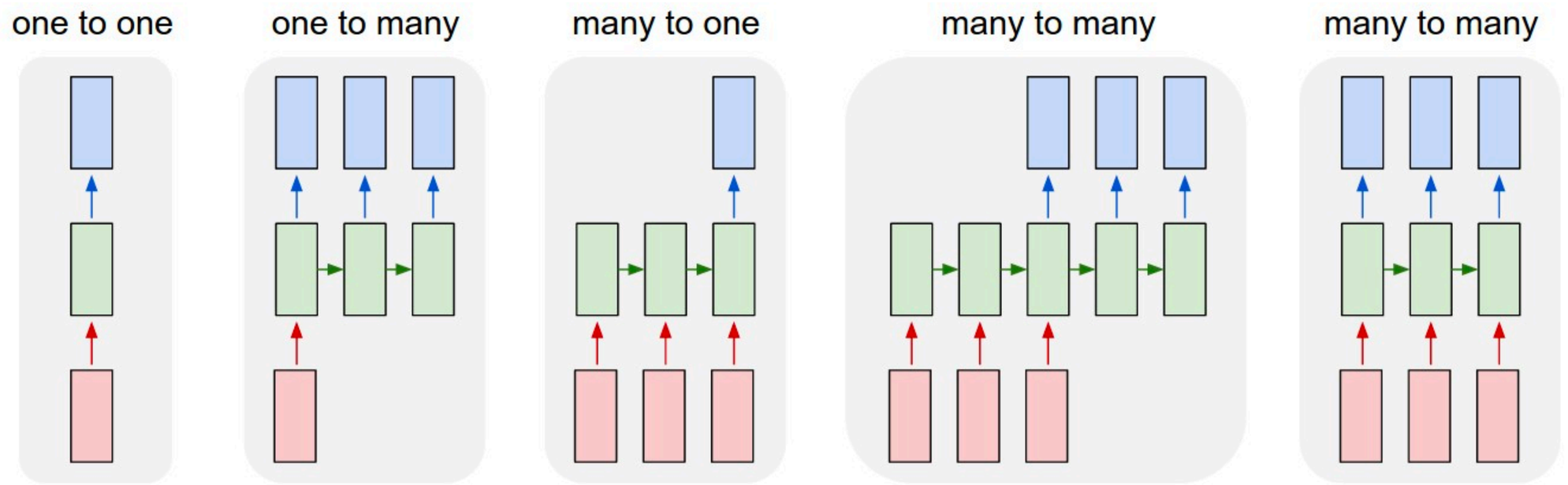
(From http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

Types of sequence problems



(From http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

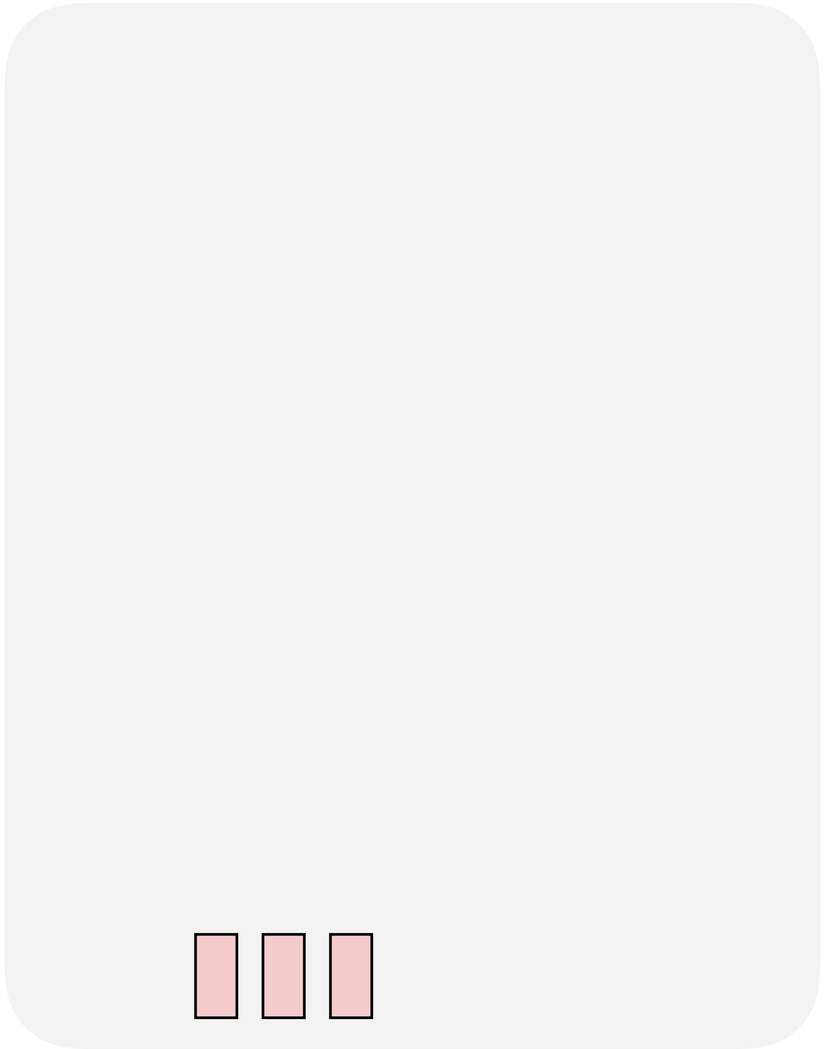
Types of sequence problems



(From http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

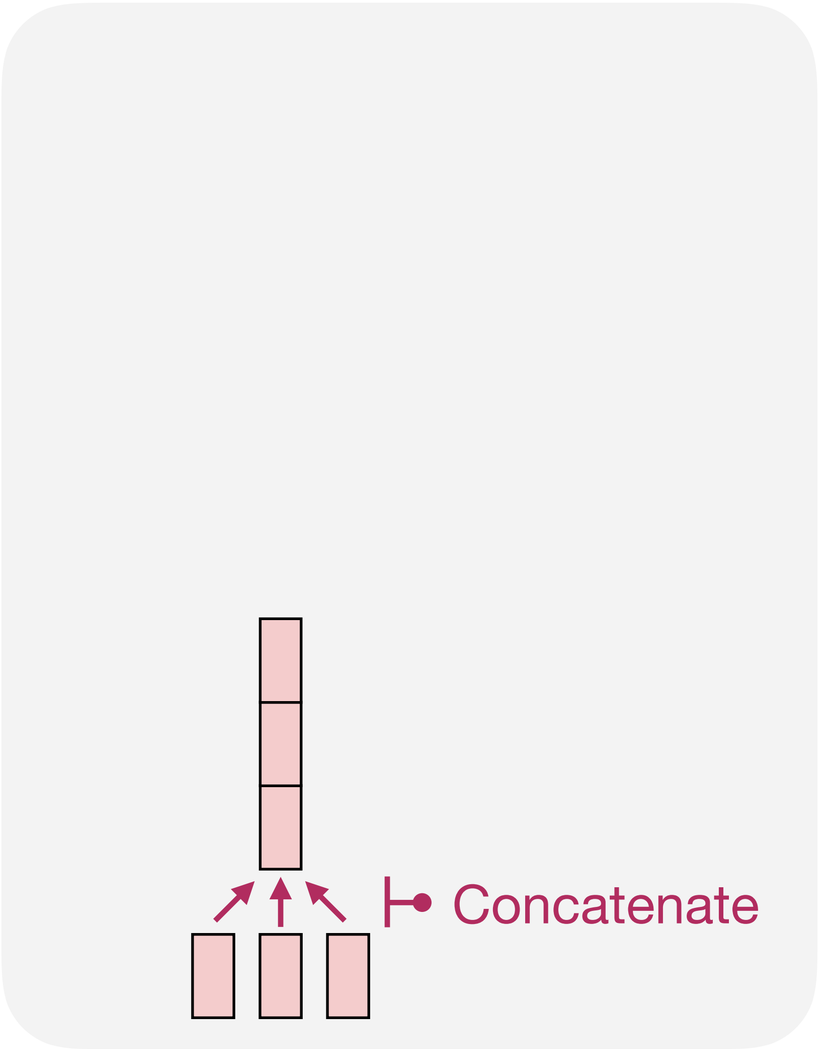
Why not use feedforward networks?

many to many

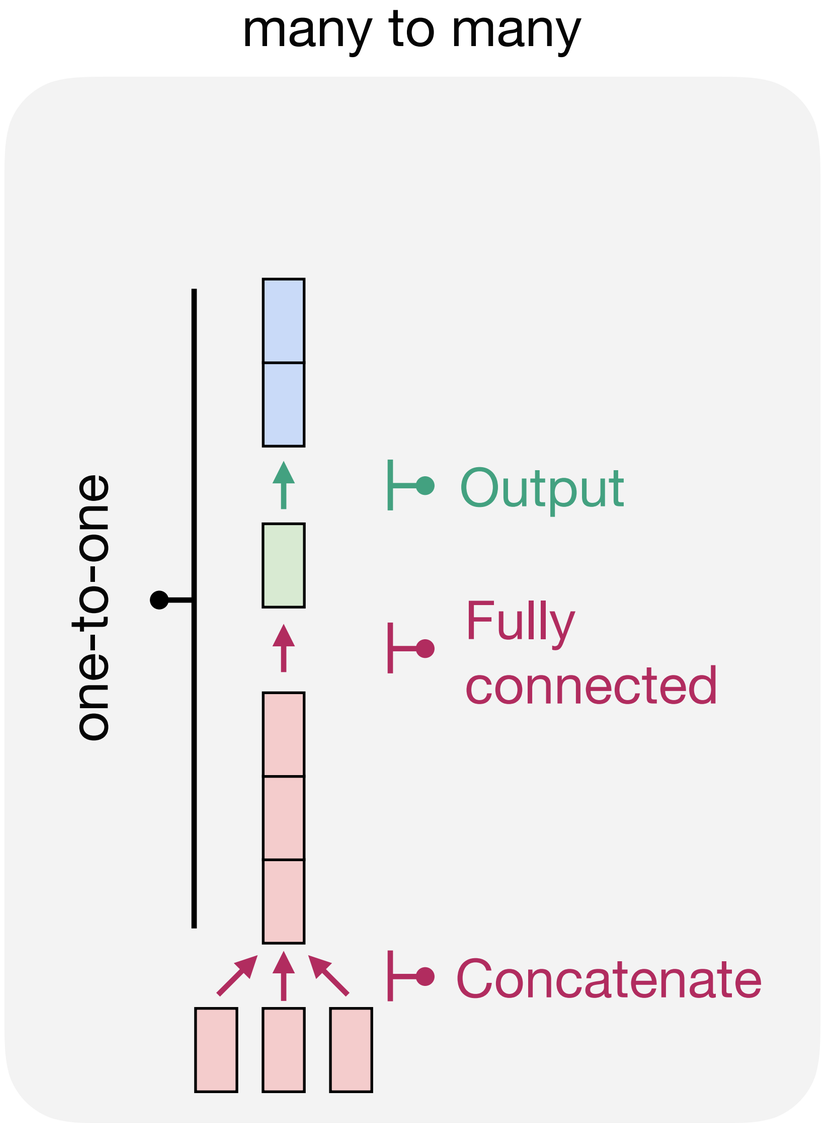


Why not use feedforward networks?

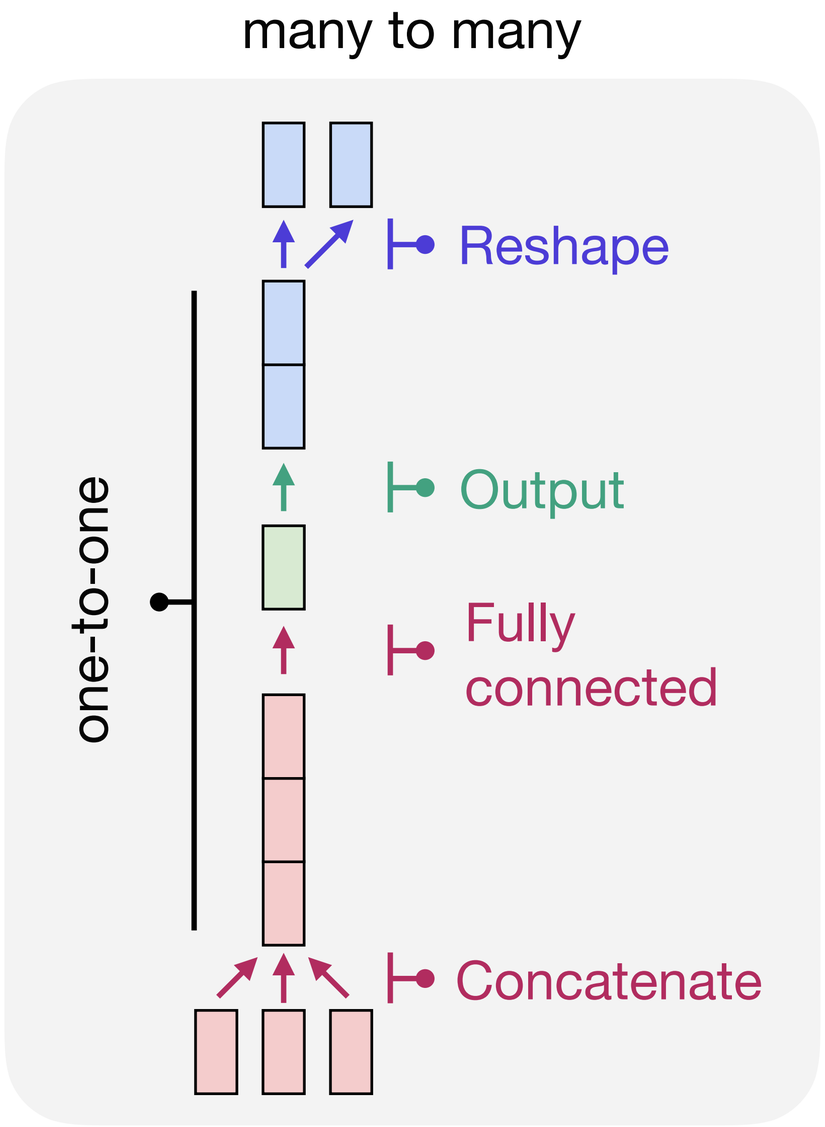
many to many



Why not use feedforward networks?

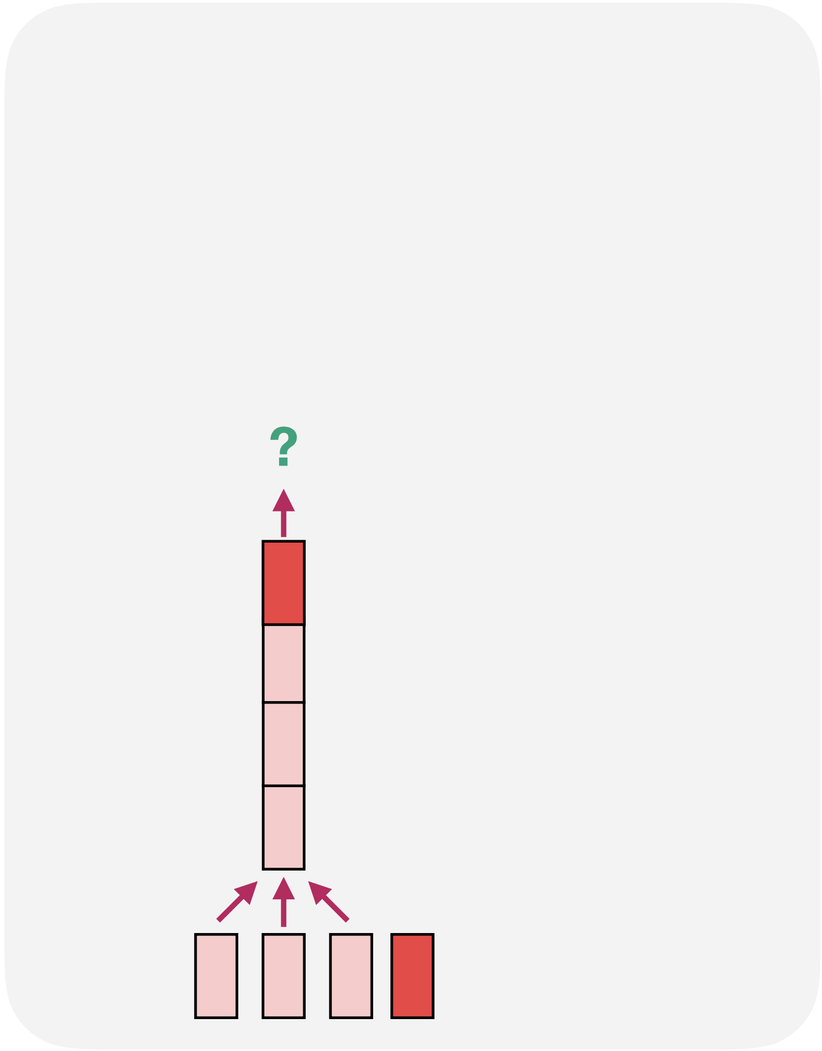


Why not use feedforward networks?



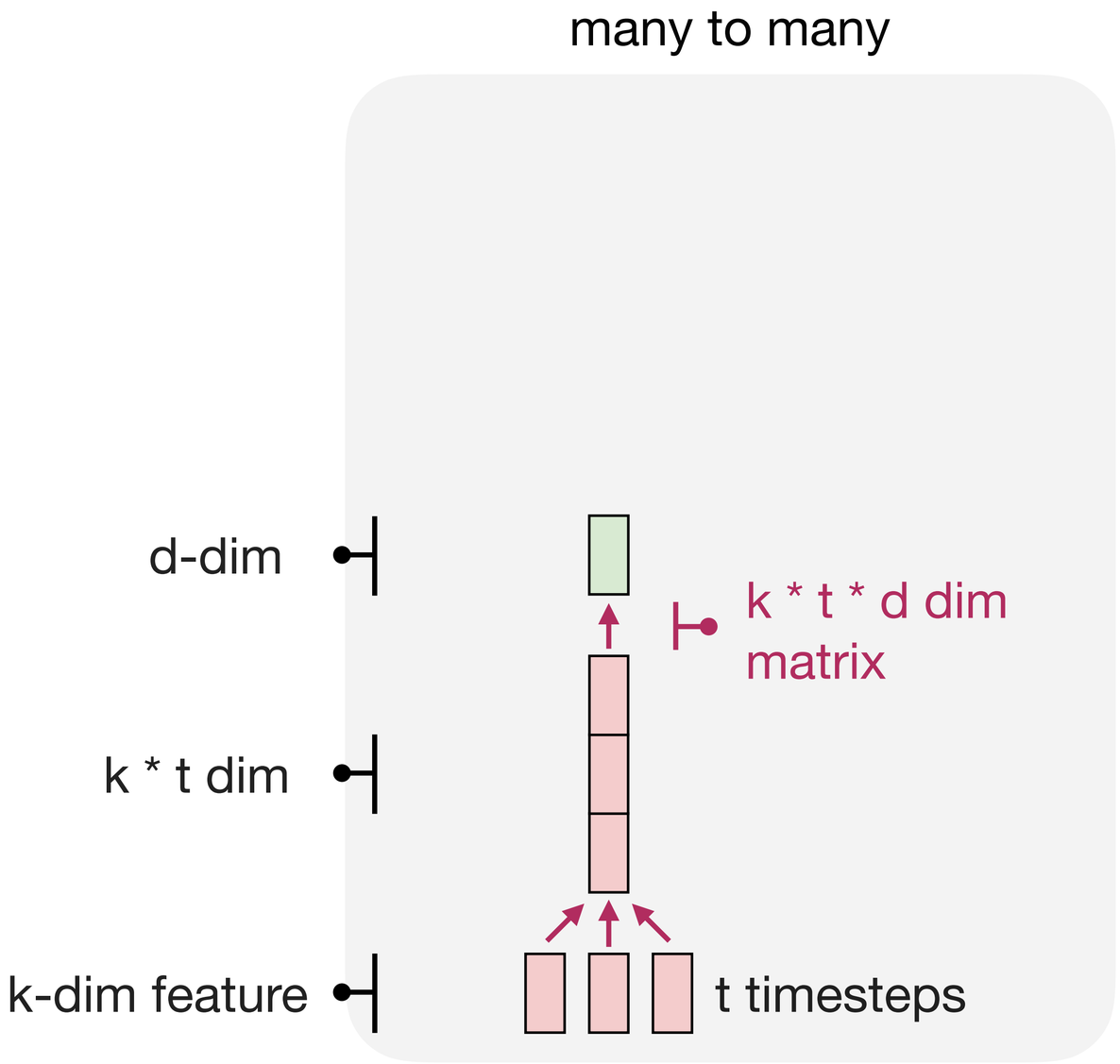
Problem 1: variable length inputs

many to many



Can deal with this by padding all sequences to the max length, but…

Problem 2: memory scaling



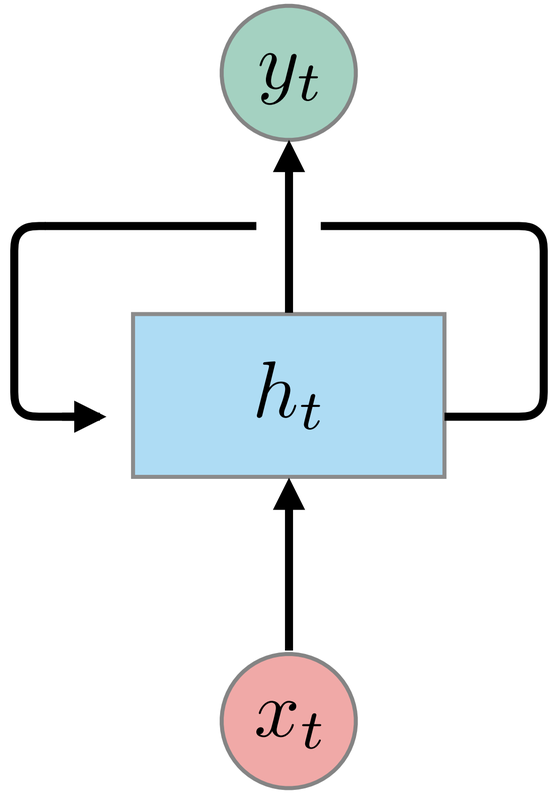
Memory requirement scales linearly in number of timesteps

Agenda

1. Why recurrent neural networks (RNNs)?
2. **RNNs**
3. Problem with RNNs: vanishing gradients
4. Dealing with vanishing gradients

Core idea of RNNs

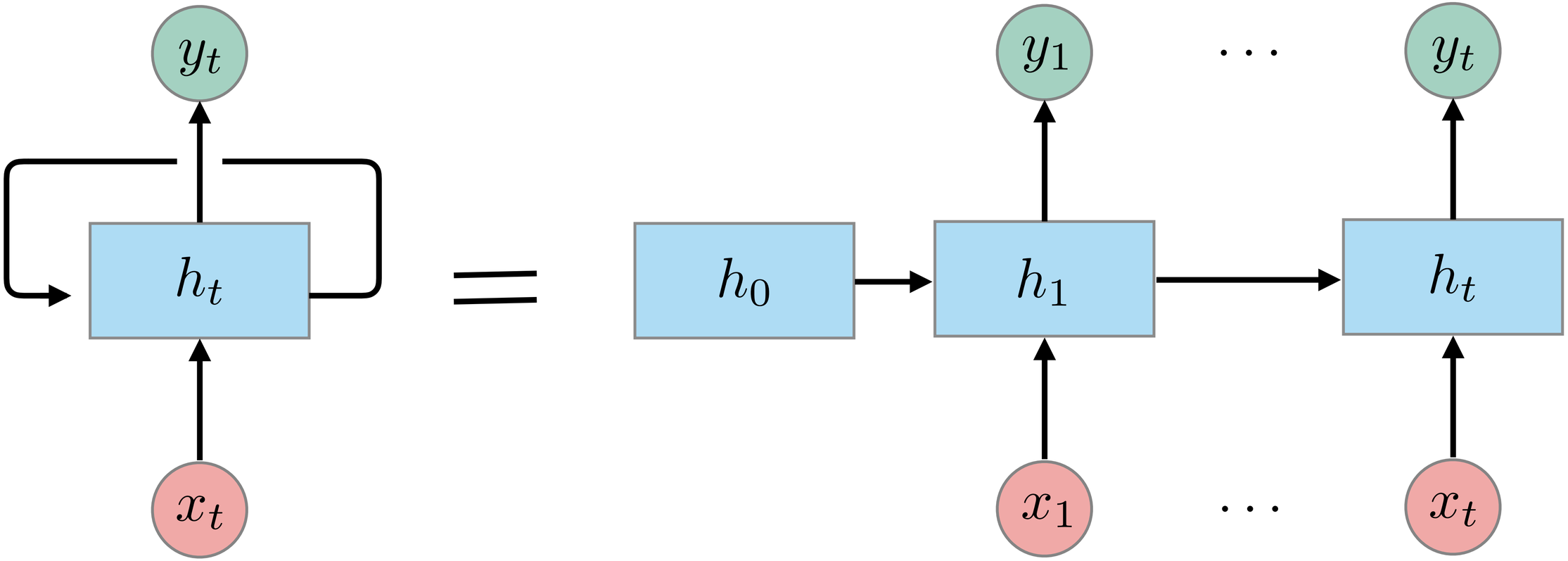
Stateful computation



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

Core idea of RNNs

Stateful computation



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

The RNN in code

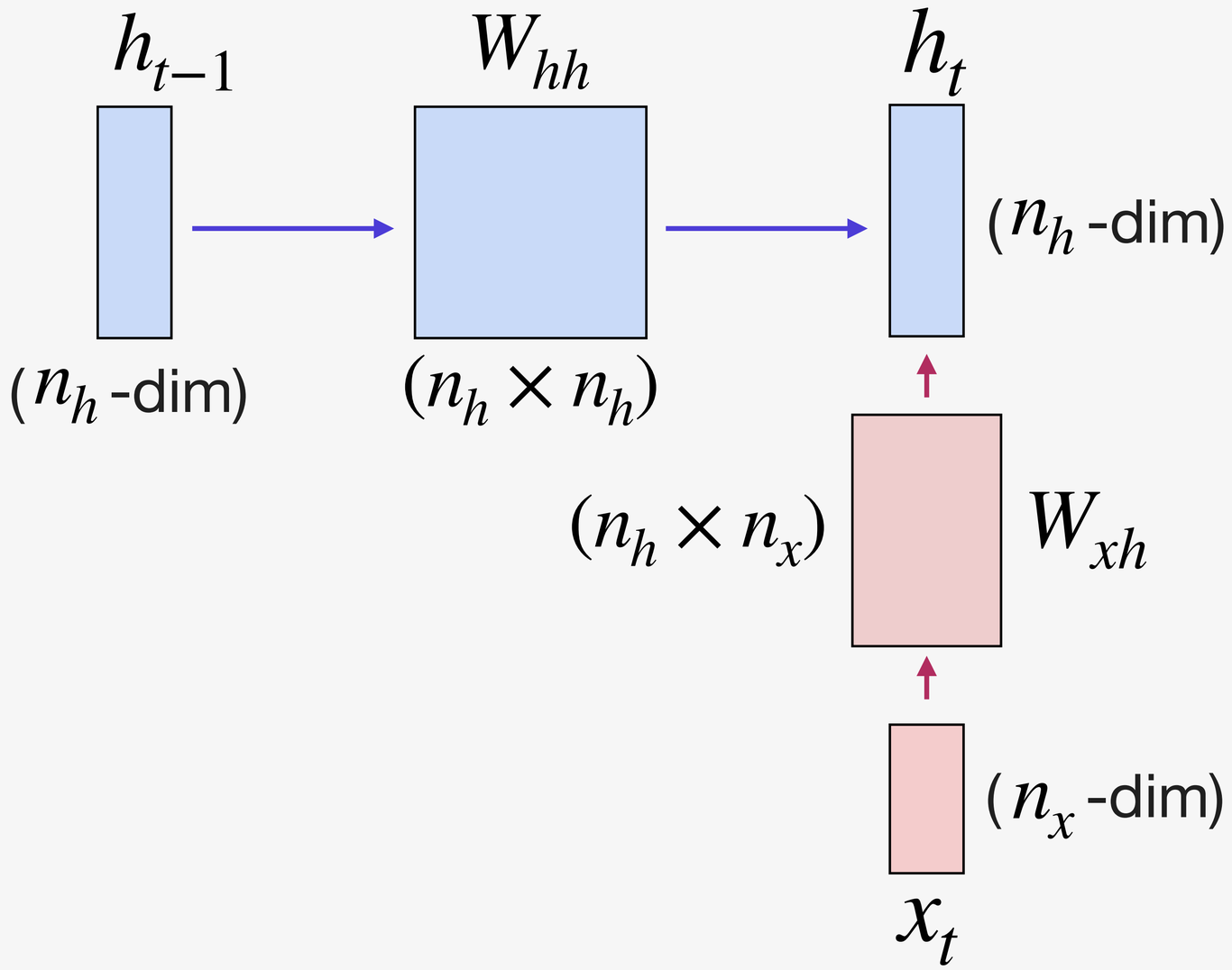
```
class RNN:
    # ...
    def compute_next_h(self, x):
        # Simplest hidden state computation. Will get fancier later.
        h = np.tanh(self.W_hh.dot(self.h) + self.W_xh.dot(x))
        return h

    def step(self, x):
        # Update the hidden state
        self.h = self.compute_next_h(x)
        # Compute the output vector
        y = self.W_hy.dot(self.h)
        return y
```

(*Adapted from* http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

A look at compute_next_h

```
def compute_next_h(self, x):
```



n_h = dim of the RNN

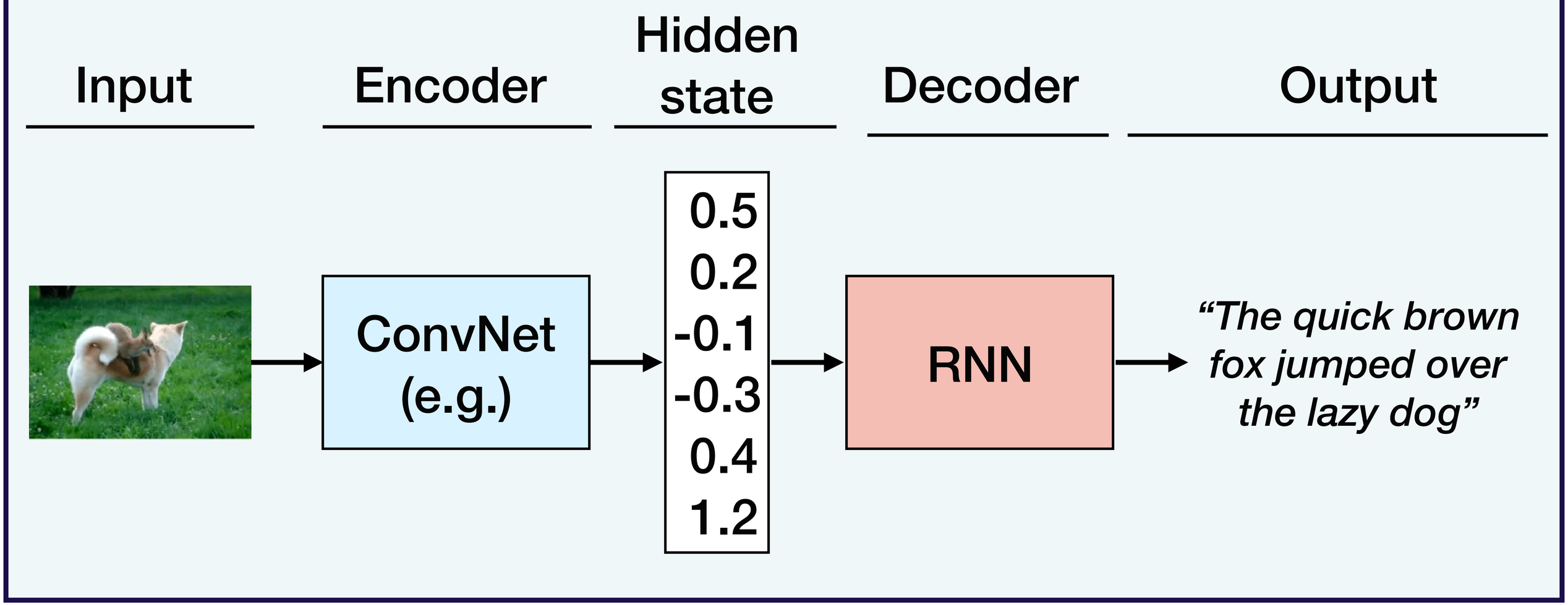
Using RNNs for many-to-one problems

- Prediction is based on the output of the RNN *last timestep*
- Reset the hidden state at the beginning of each sequence

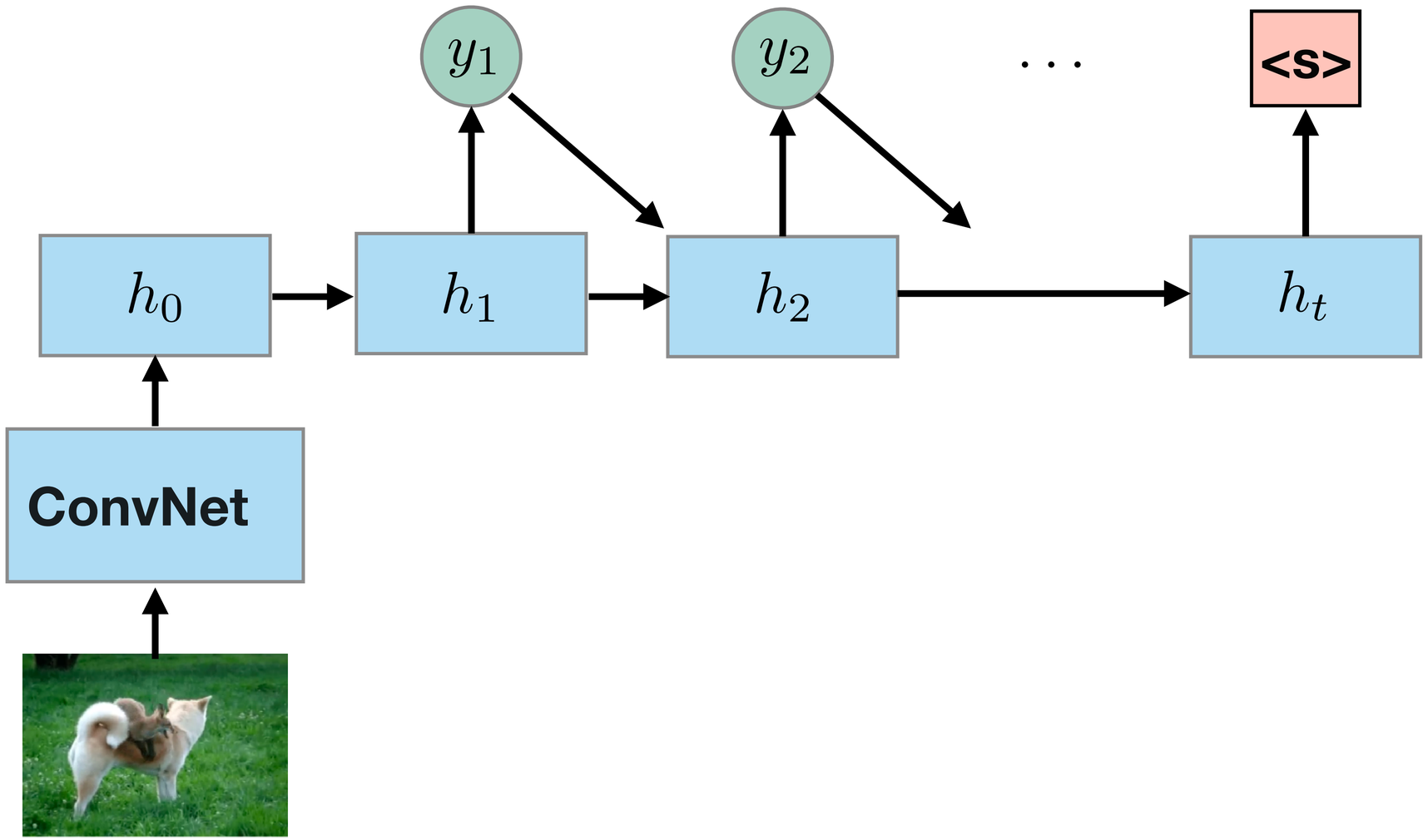
```
X, y = data # X.shape == (100, 7), y.shape == (1,)
T = X.shape[0]
rnn = RNN()
rnn.h.fill(0.0)
for i in range(T-1):
    RNN.step(x[i])
y_hat = RNN.step(x[-1])
```

Using RNNs for <one/many> to many problems

Encoder-decoder architectures

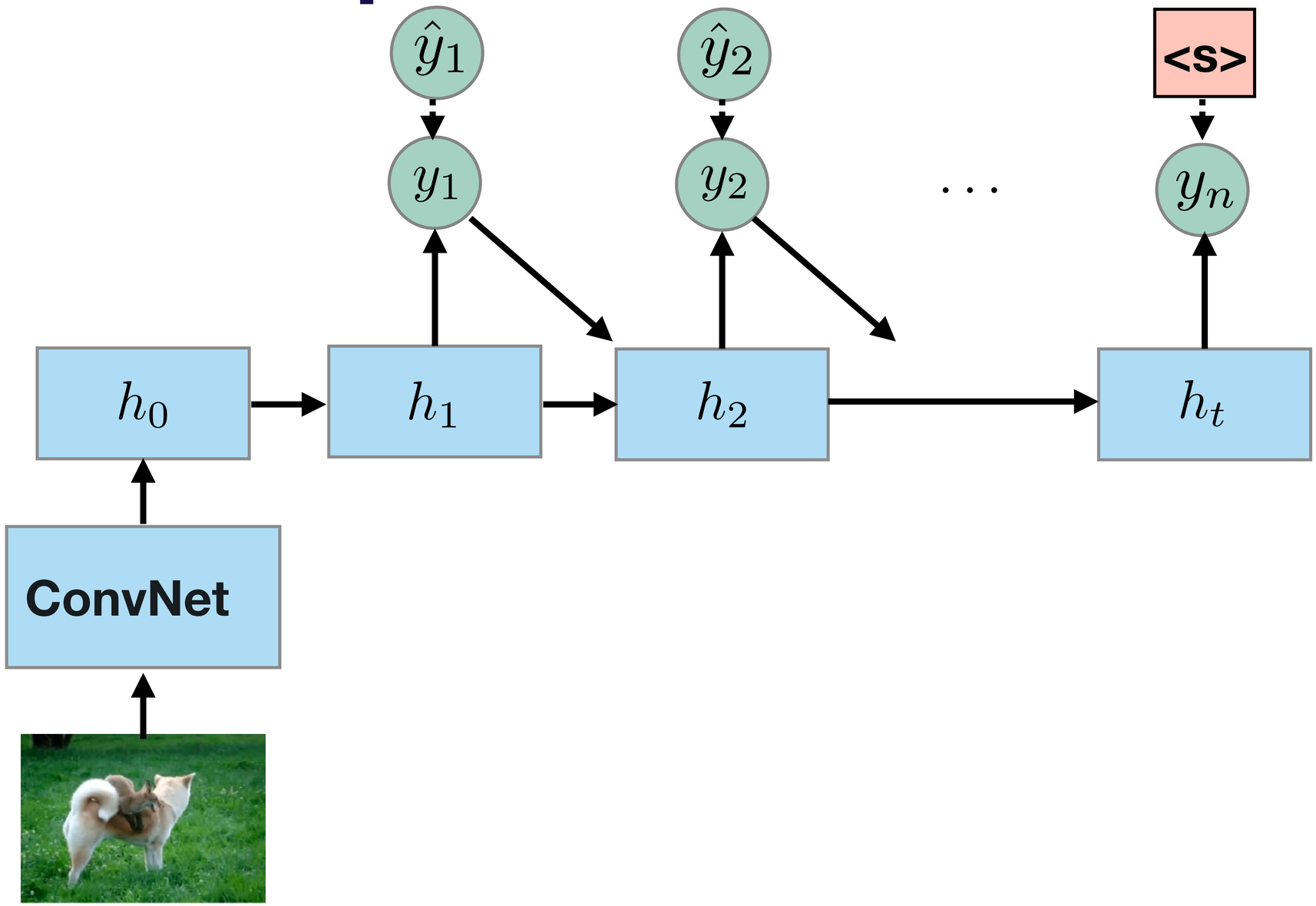


Stop characters



Special character that tells the network to stop generating

Sequence loss functions

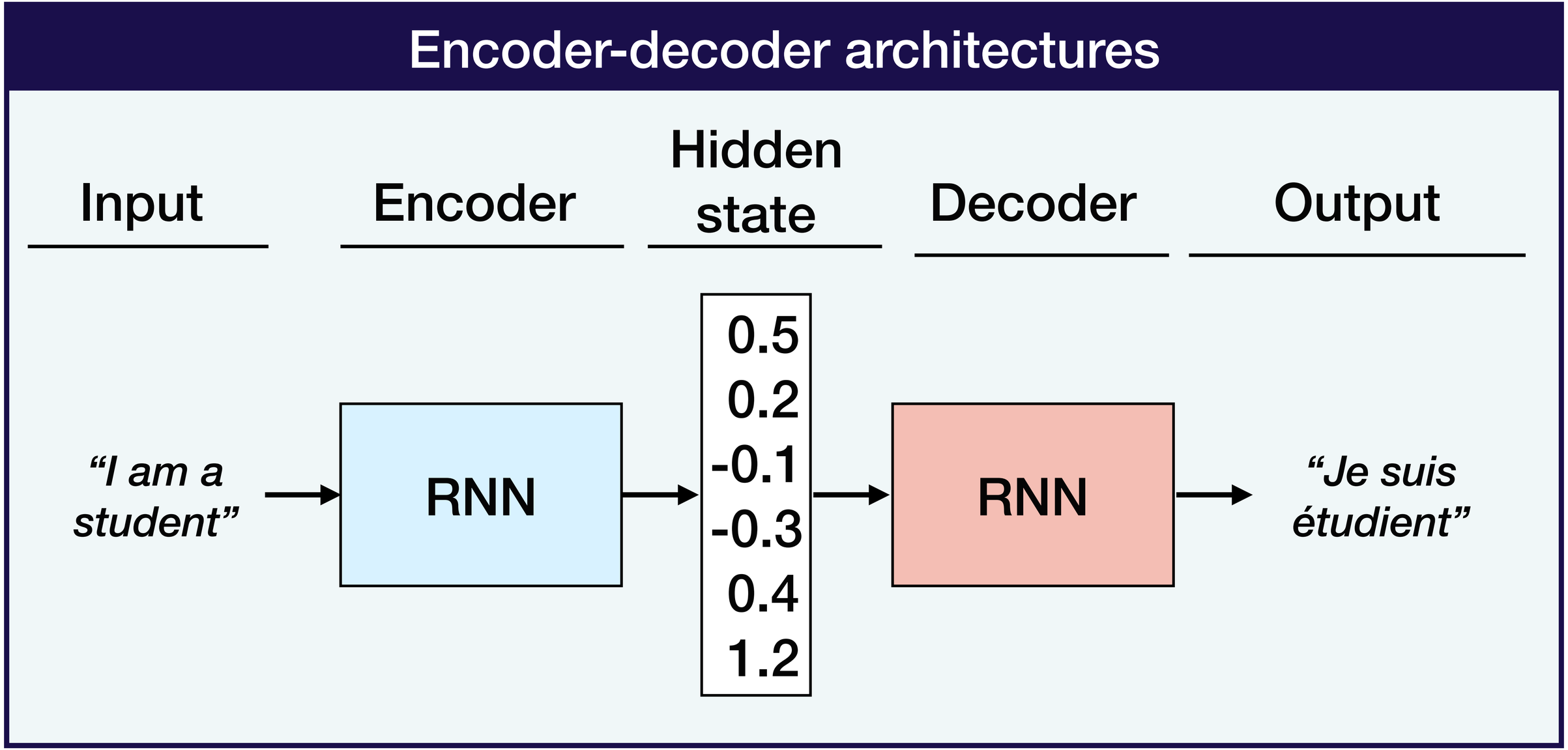


Sum of cross-entropy losses on *all* ground-truth terms from the sequence

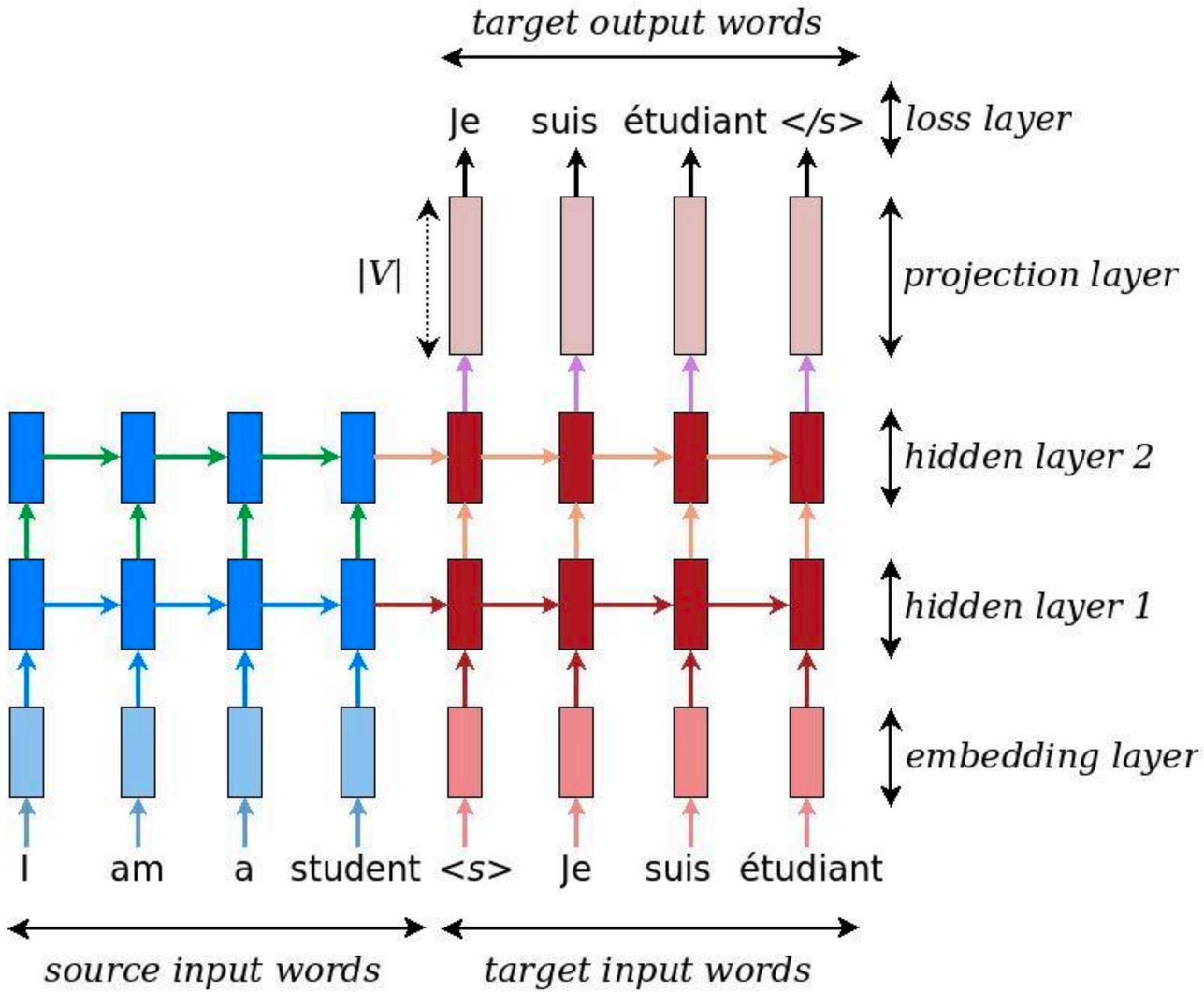
Using RNNs for one to many problems (in code)

```
output_dim = y.shape[0]
encoder = CNN(output_dim)
decoder = RNN()
hidden_state = encoder(image)
decoder.h = hidden_state
outputs = []
output = 0
while output != STOP_CHARACTER:
    output = decoder.step(output)
    outputs.append(output)
```

Using RNNs for <one/many> to many problems



Using RNNs for many->many problems

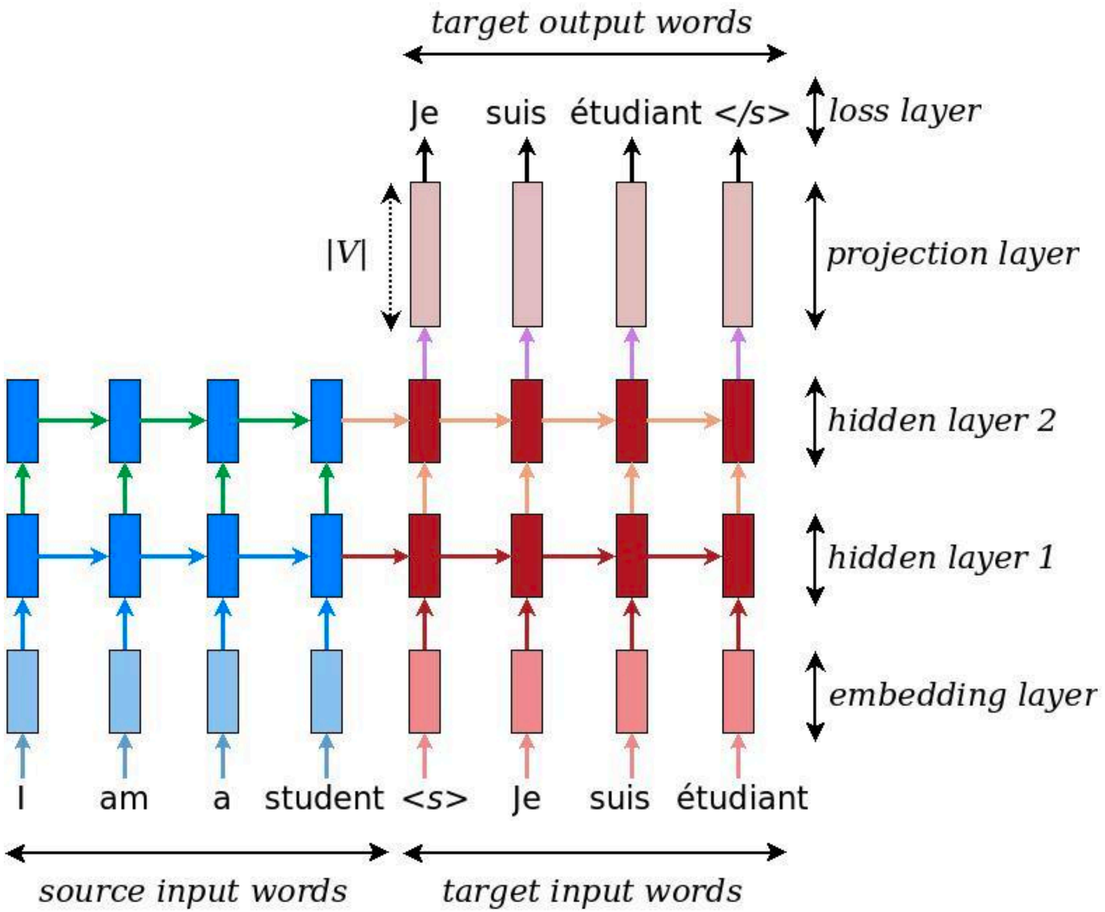


Using RNNs for many->many problems (in code)

```
encoder = RNN()
decoder = RNN()
output = 0
while output != STOP_CHARACTER:
    output = encoder.step(output)

decoder.h = encoder.h
outputs = []
output = 0
while output != STOP_CHARACTER:
    output = decoder.step(output)
    outputs.append(output)
```

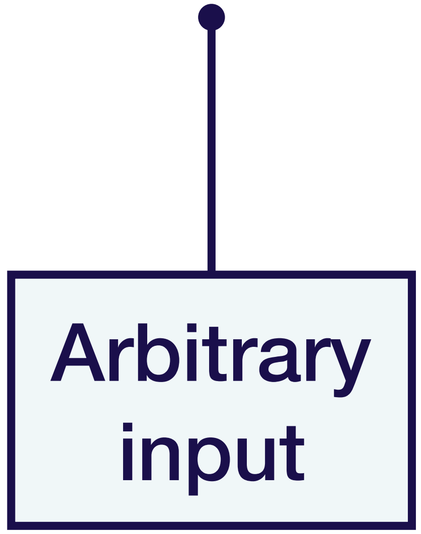
Deep learning with RNNs




```
rnn = RNN()
y_1 = rnn.step(x)
y_2 = rnn.step(y_1)
y_3 = rnn.step(y_2)
y = RNN.step(y_3)
```

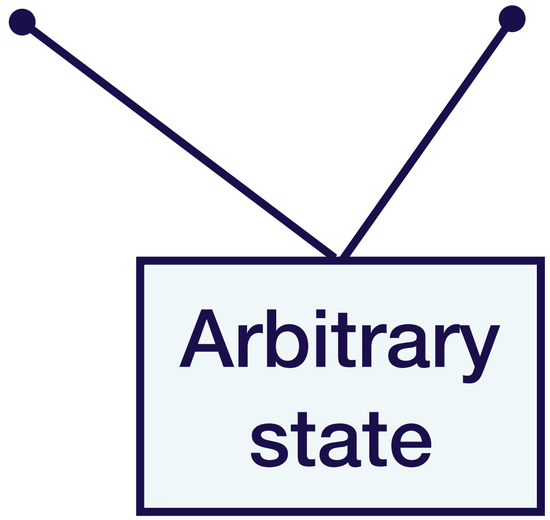
RNNs describe programs

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



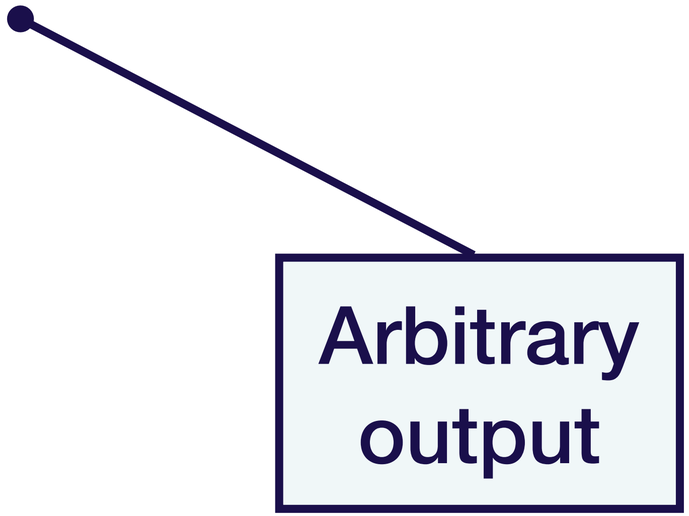
RNNs describe programs

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



RNNs describe programs

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



RNNs describe programs

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

Fact: RNNs are Turing complete
(i.e., you can model any program
with an arbitrarily large RNN)

http://binds.cs.umass.edu/papers/1995_Siegelmann_Science.pdf

Agenda

1. Why recurrent neural networks (RNNs)?
2. RNNs
3. **Problem with RNNs: vanishing gradients**
4. Dealing with vanishing gradients

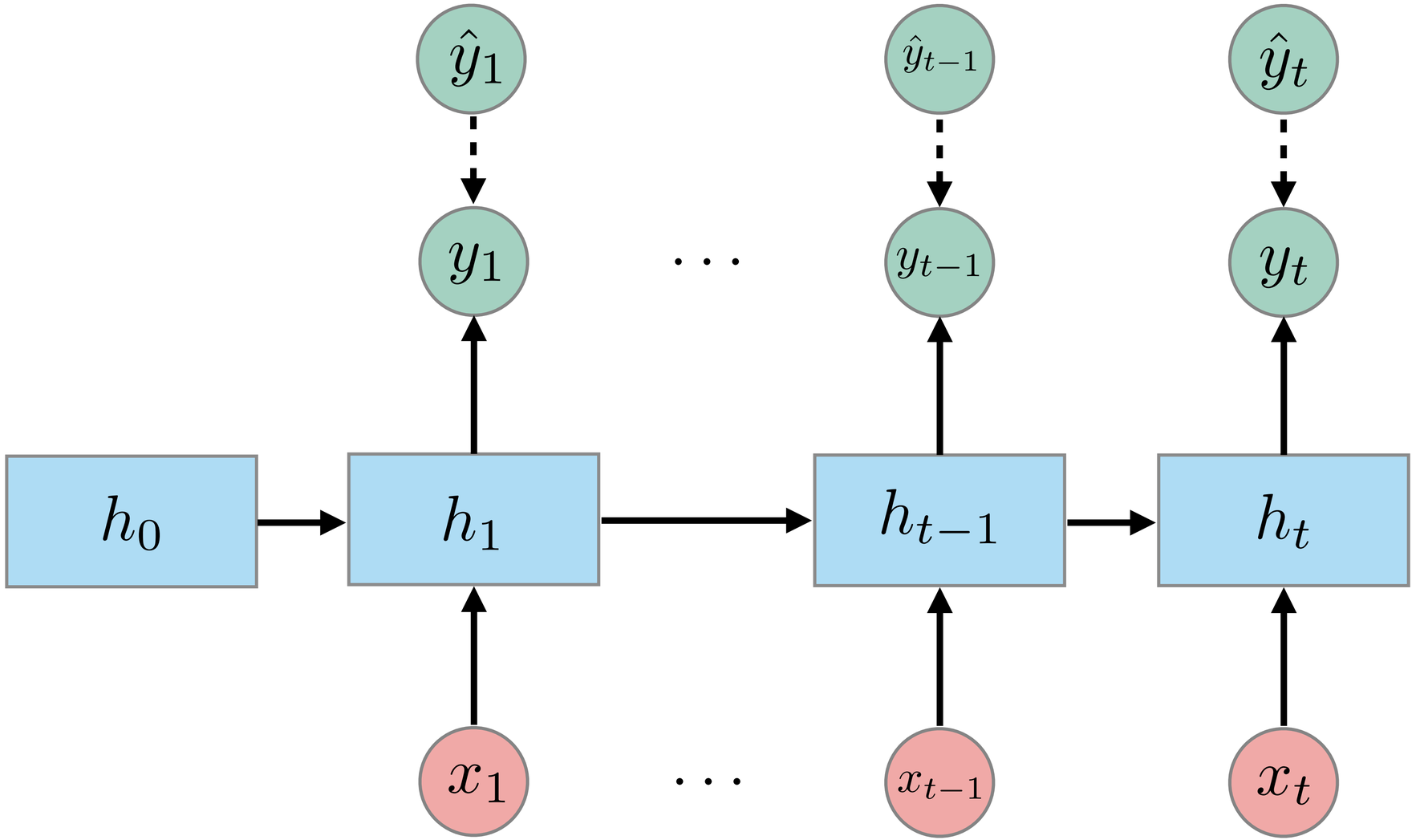
RNN Desiderata

- Goal: handle sequences
- In particular, long sequences (where linear scaling is an issue)
- Connect events from the past to outcomes in the future
  - i.e., *Long-term dependencies*
  - e.g., remember the name of a character from the first sentence

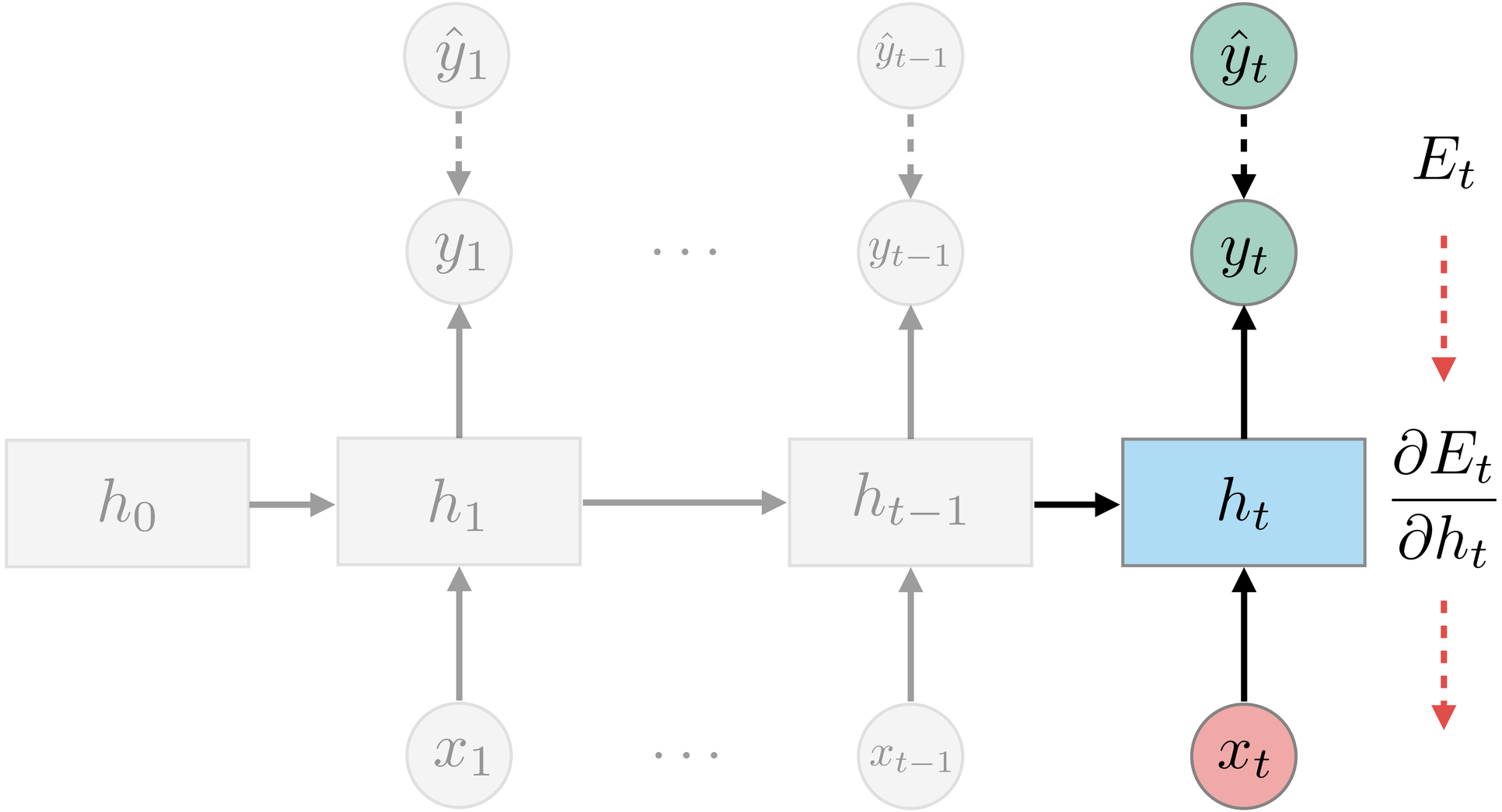
Vanilla RNNs: the reality

- Can’t handle more than 10-20 timesteps
- Longer-term dependencies get lost
- Why? ***Vanishing gradients***

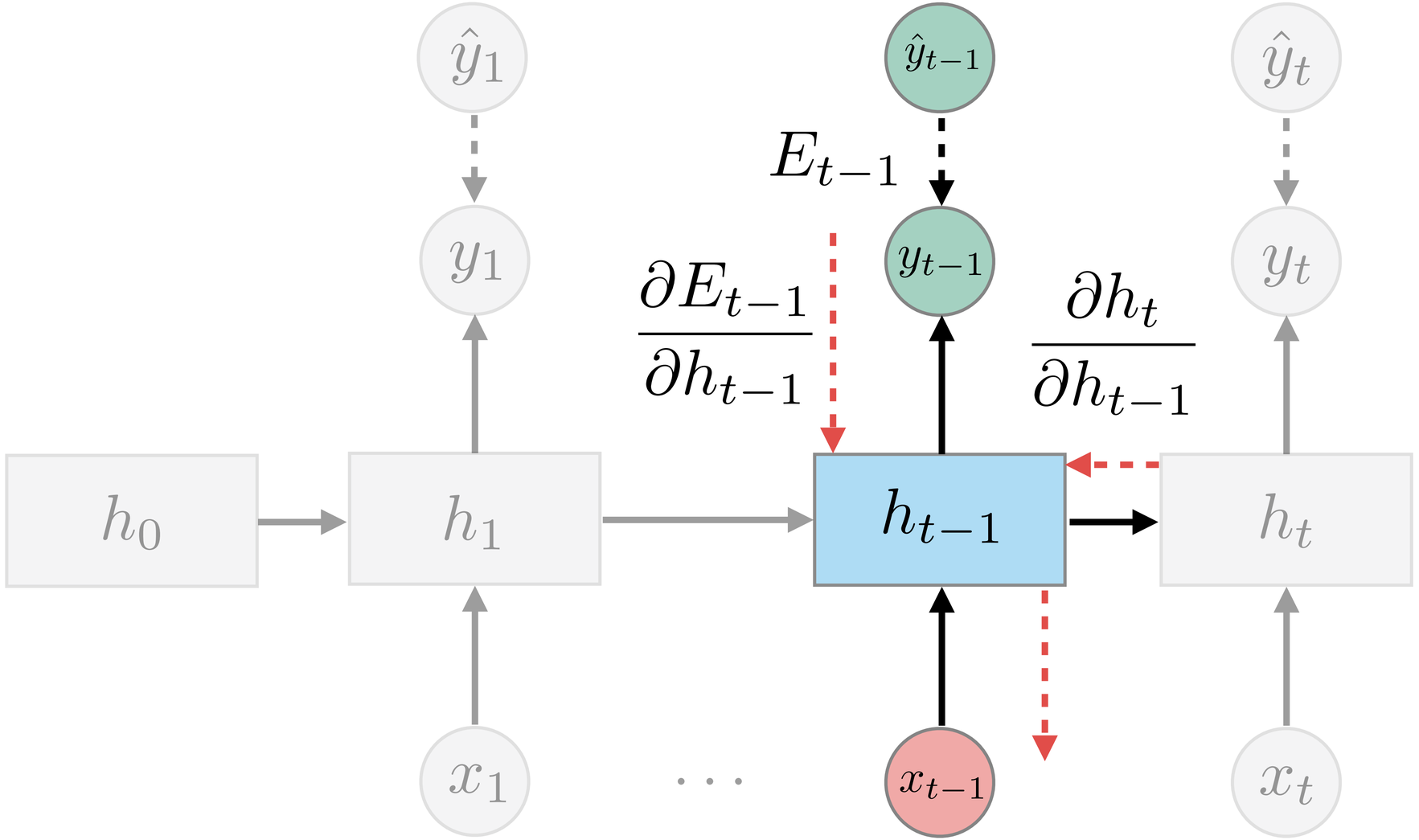
Gradients in RNNs: backpropogation through time



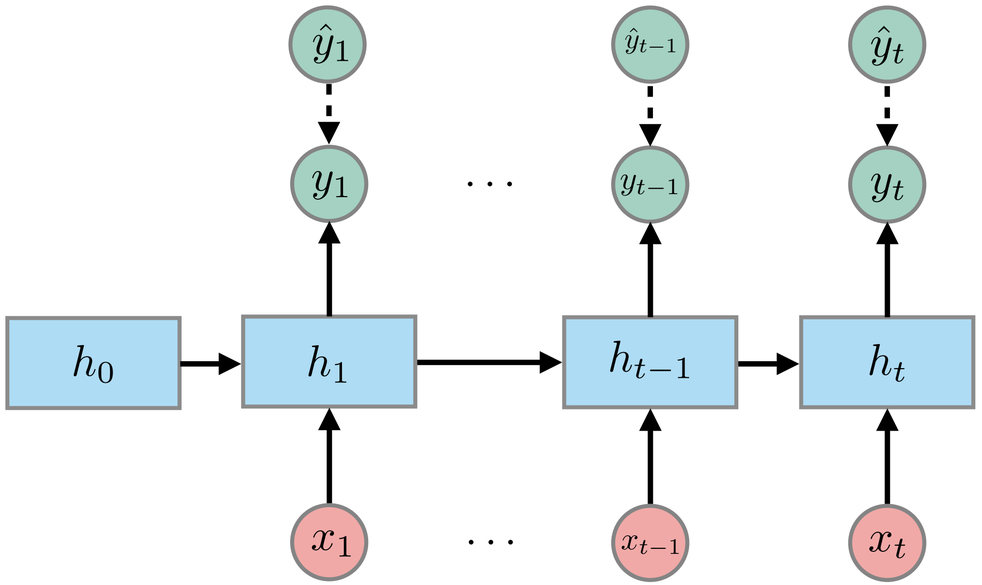
Gradients in RNNs: backpropogation through time



Gradients in RNNs: backpropogation through time



Gradients in RNNs: backpropogation through time



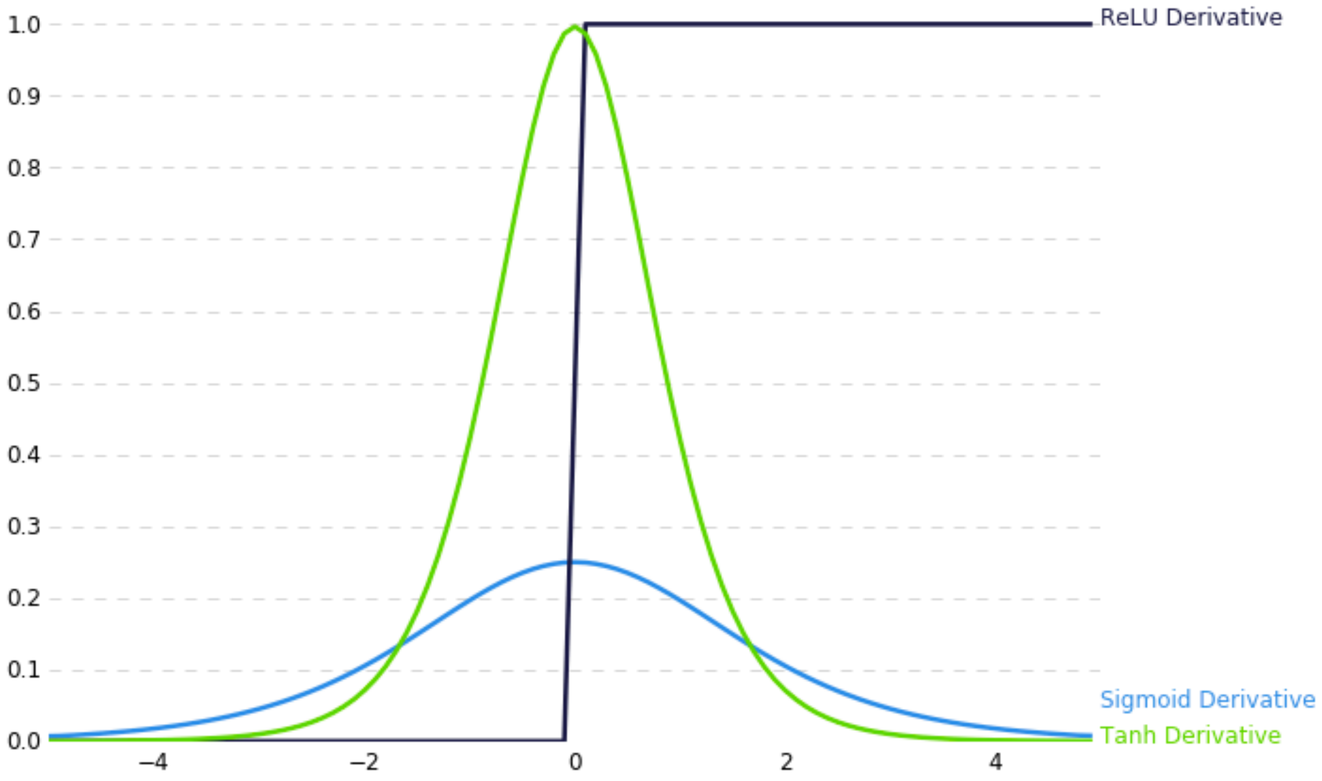
$\frac{\partial E_t}{\partial W_{hh}}$ depends on each $\frac{\partial h_j}{\partial h_{j-1}}$

Recall $h_j = \tanh(W_{hh}h_{j-1} + W_{xh}x_j)$

So each $\frac{\partial h_j}{\partial h_{j-1}}$ introduces a $\tanh'$

Back to vanishing gradients

Derivatives of common activation functions



- sigmoid and tanh have derivatives << 1 near saturation
- At each step magnitude of gradients tends to decrease
- After enough steps gradients are too small
- (ReLU RNNs often have the opposite problem - exploding gradients)

Agenda

1. Why recurrent neural networks (RNNs)?
2. RNNs
3. Problem with RNNs: vanishing gradients
4. **Dealing with vanishing gradients**

Intro to LSTMs

Recall vanilla RNNs

```
class RNN:
    # ...
    def compute_next_h(self, x):
        # Simplest hidden state computation. Will get fancier later.
        h = np.tanh(self.W_hh.dot(self.h) + self.W_xh.dot(x))
        return h

    def step(self, x):
        # ...
```

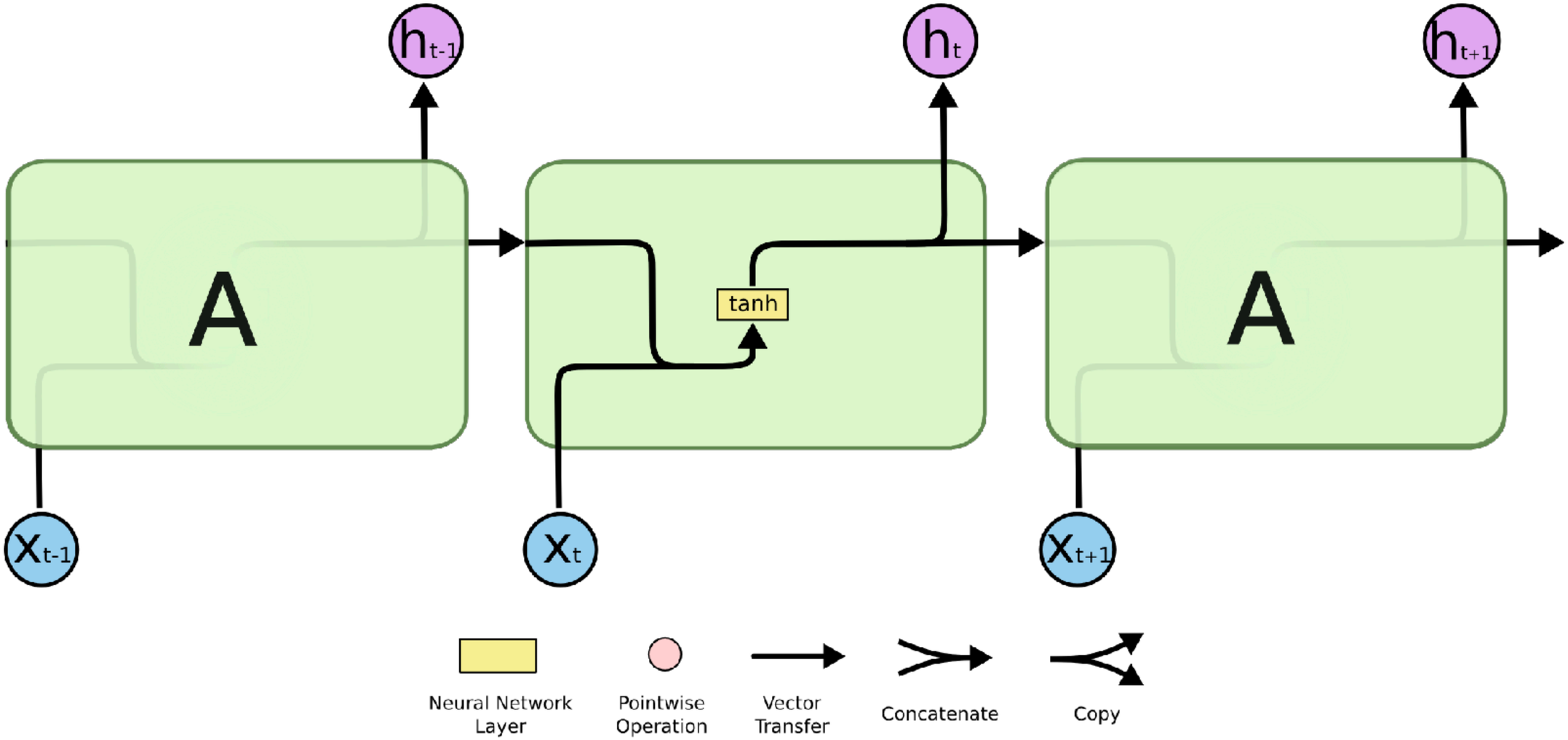
Intro to LSTMs

Idea: use a compute_next_h that preserves gradients

```
class LSTM(RNN):
    # ...
    def compute_next_h(self, x):
        h = lstm(x, self.h)
        # or gru(x, self.h), etc.
        return h
    # ...
```

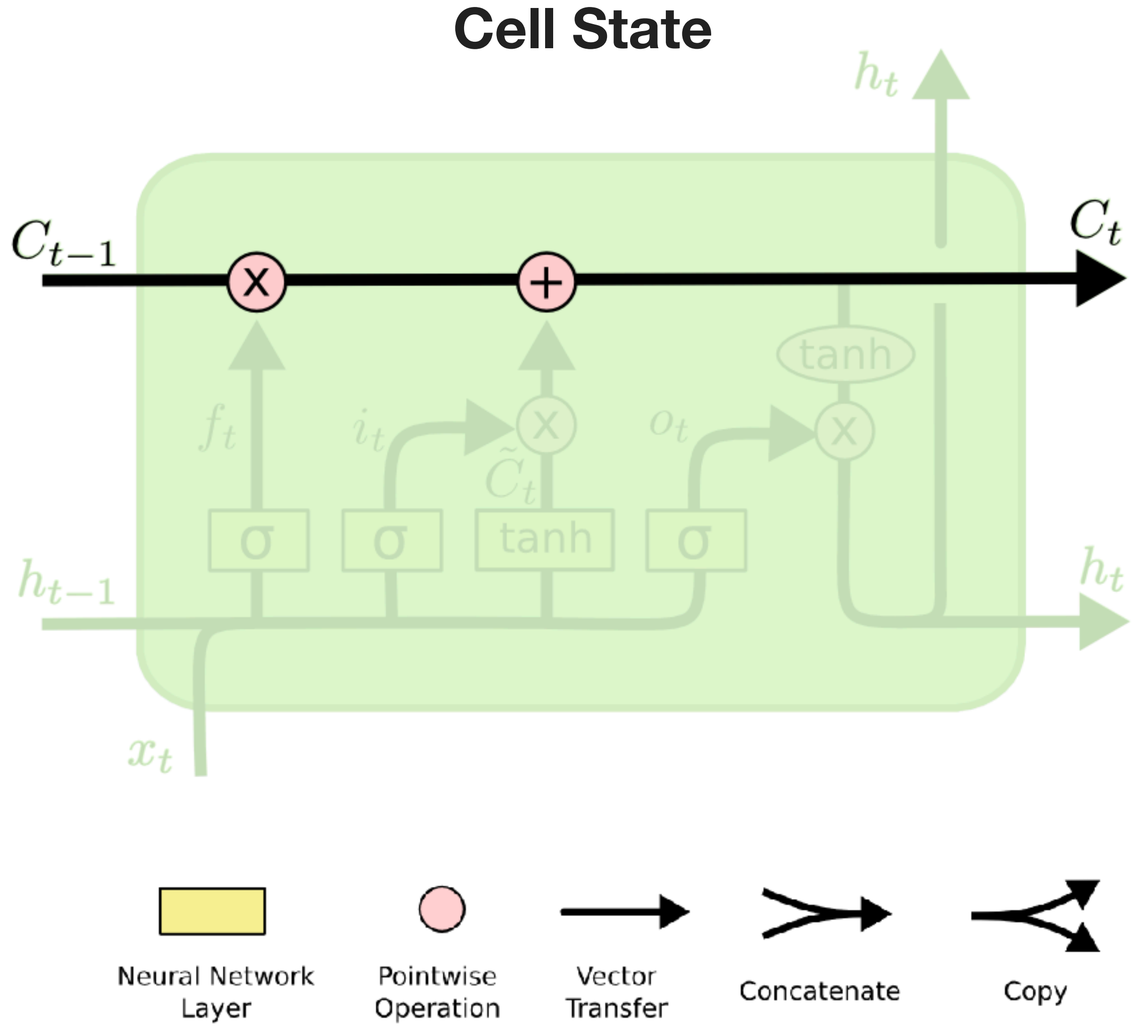
For more info, see http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Visualization of vanilla RNNs



From http://colah.github.io/posts/2015-08-Understanding-LSTMs/

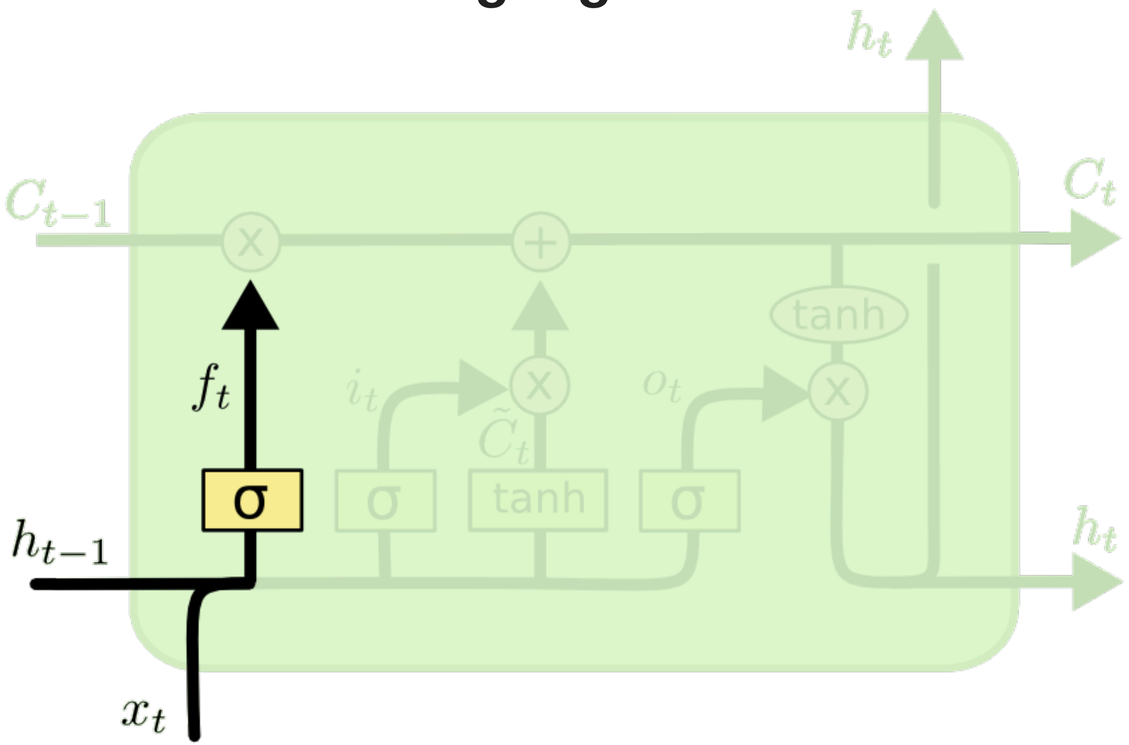
Building up to LSTMs



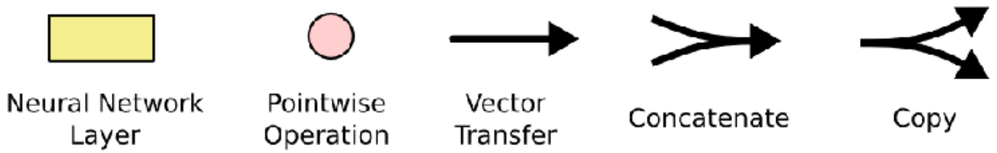
From http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Building up to LSTMs

Forget gate



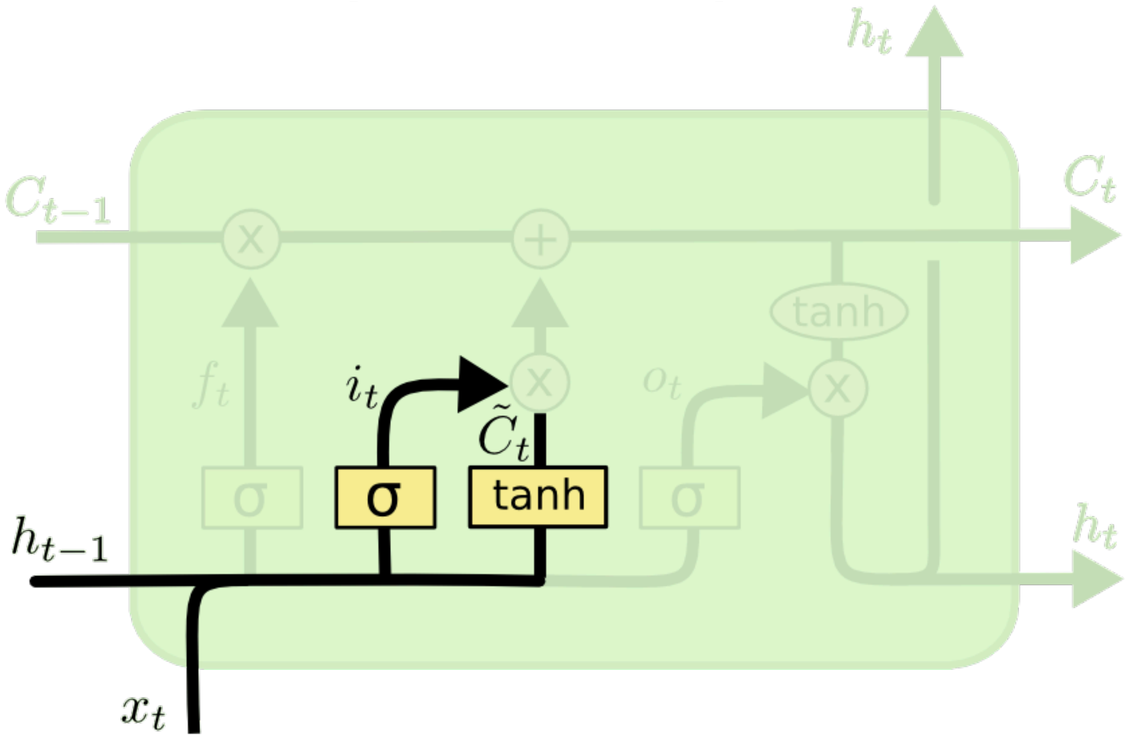
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$



From http://colah.github.io/posts/2015-08-Understanding-LSTMs/

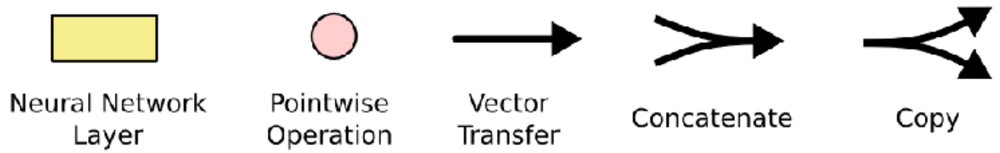
Building up to LSTMs

Proposed state update



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

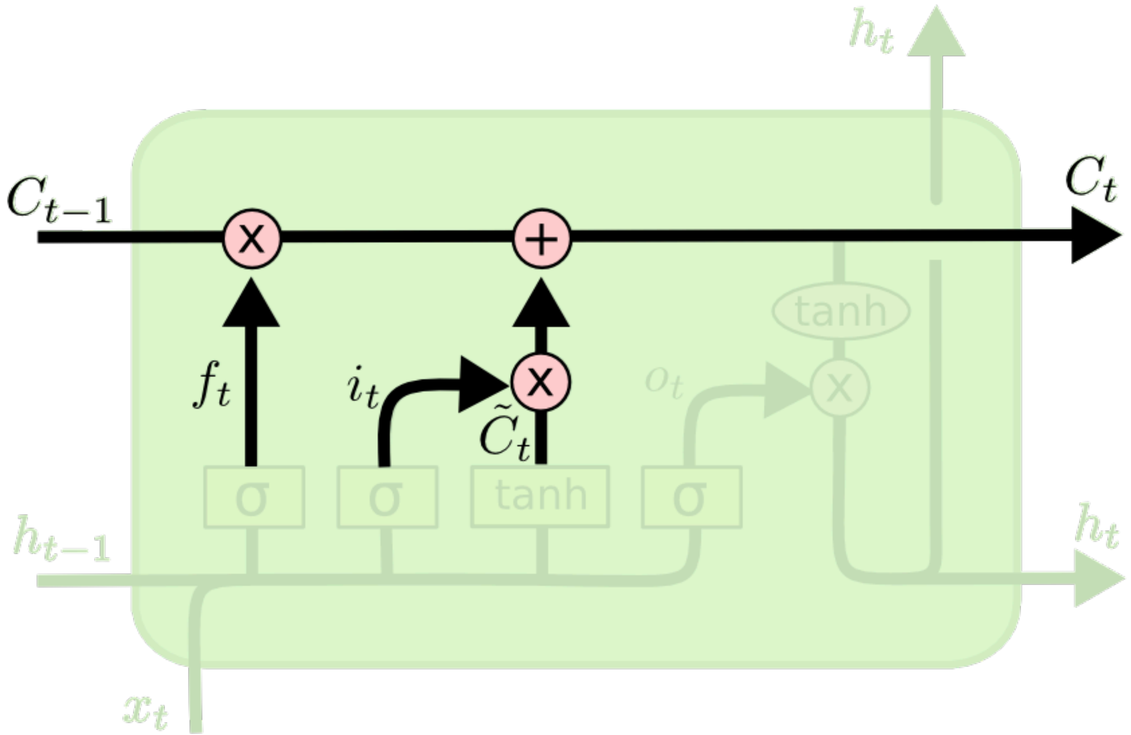
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$



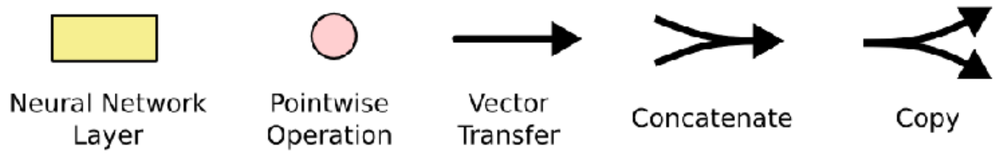
From http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Building up to LSTMs

Update the state



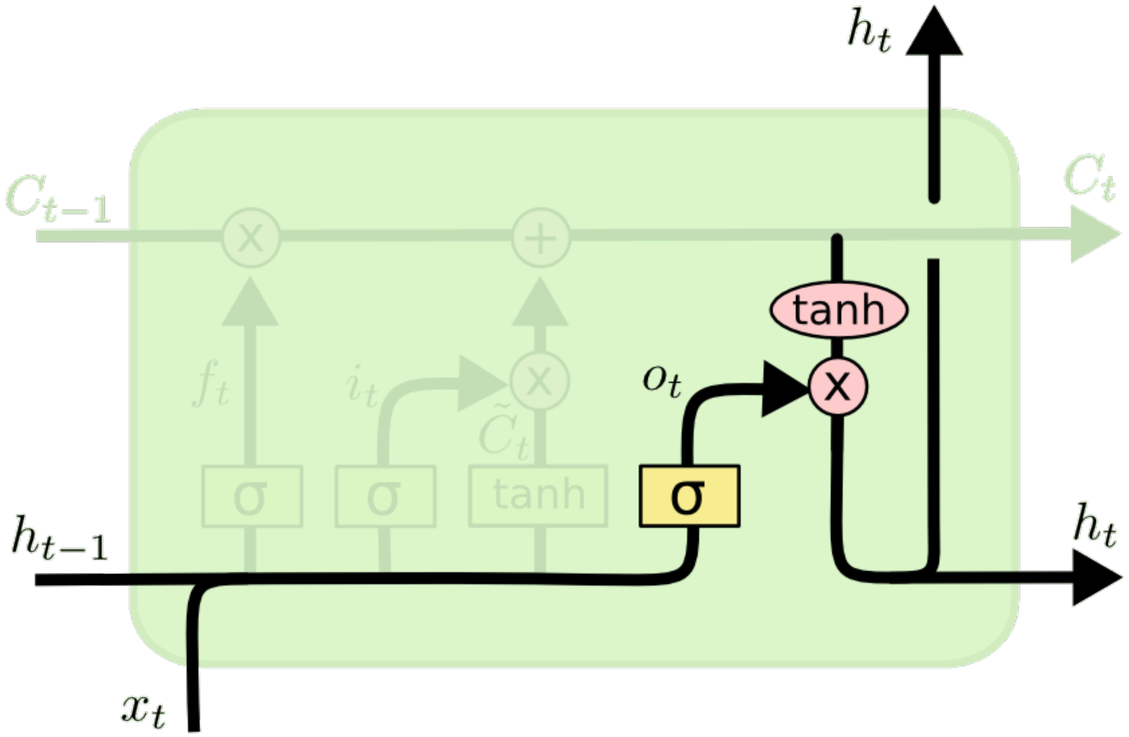
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$



From http://colah.github.io/posts/2015-08-Understanding-LSTMs/

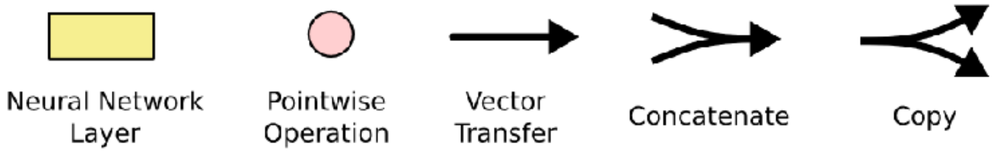
Building up to LSTMs

Decide what to output



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

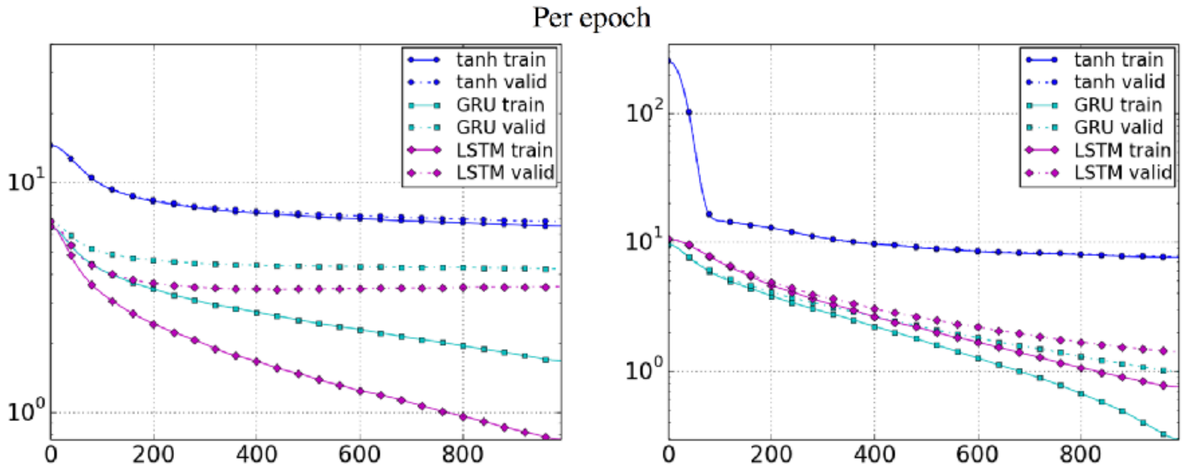
$$h_t = o_t * \tanh\left(C_t\right)$$



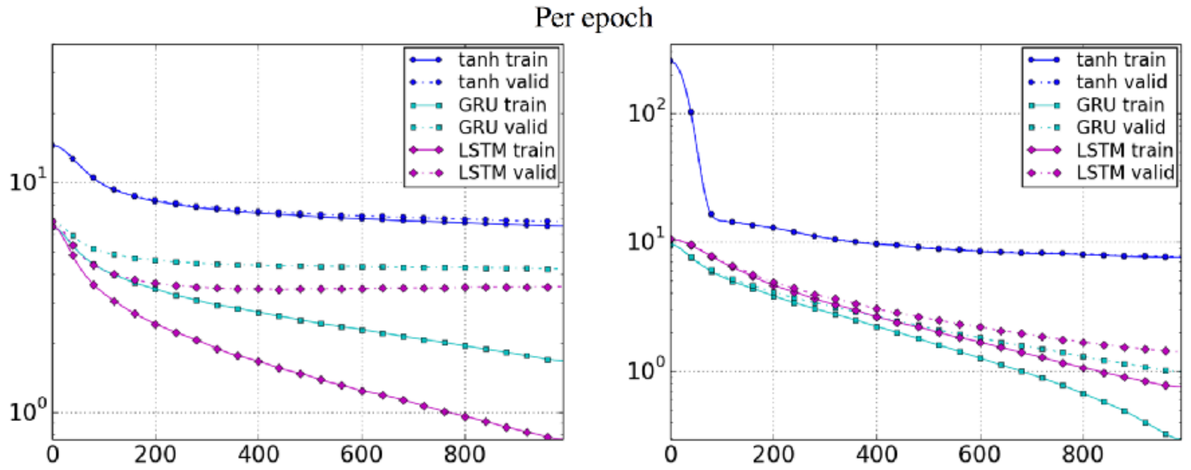
From http://colah.github.io/posts/2015-08-Understanding-LSTMs/

How well do LSTMs work?

Music modeling



Speech signal modeling



Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling (Chung et al.)

What about GRUs, etc?

LSTM: A Search Space Odyssey
Greff, et al.

- 9 LSTM variants (including GRU)
- 3 datasets
- **Conclusion:** Hard to beat regular LSTM

An Empirical Exploration of Recurrent Neural Network Architectures
(Jozefowicz, Zaremba, Sutskever)

- 10,000 architectures
- 3 datasets
- **Conclusion:** GRUs > LSTMs. Some architectures better than GRU (but only slightly)

What about GRUs, etc?

My advice

- LSTMs work well for most tasks
- May also try GRUs if LSTMs are not performing well
- Not usually worth considering other LSTM variants

Other ways of dealing with vanishing/exploding gradients

- Identity initialization & ReLU
- Gradient clipping
- Skip connections
- Leaky connections
- Attention
- Explicit memory networks
  (e.g., Neural Turing Machines)

Conclusion

- Encoder / decoder architectures can model arbitrary (one-to-many, many-to-one, and many-to-many) sequence problems
- Combined with LSTMs, great default option for tasks with sequential data

Where to learn more?

- Andrej Karpathy's blog post on RNNs:
  http://karpathy.github.io/2015/05/21/rnn-effectiveness/
- Chris Olah's blogpost on LSTMs:
  http://colah.github.io/posts/2015-08-Understanding-LSTMs/