# COMP 6321 Machine Learning

# Recurrent Networks

Computer Science & Software Engineering
Concordia University, Fall 2020

# Recurrence

- ‘Recurrent’ means “*happening again many times*.”
- In programming:
  - A for-loop that repeatedly applies the same code to the current program state, updating that state
  - A recursive function that repeatedly applies its code to different inputs
- In *recurrent neural networks* (RNNs):
  - Repeatedly apply a network to its own output, in a *loop*
  - Or, can view as “unrolled” network with *weight-sharing*
  - Can be applied to variable-length inputs (just loop!)
  - Often applied when *state* must be *accumulated* over *time* and/or *space* to make good prediction
    - e.g. reading a sentence, predicting future based on past, etc.

# Unrolling a loop

## A loop

```
h = h0
for i in range(3):
    h = w*h
```

## An unrolled loop

```
h = h0
h = w*h
h = w*h
h = w*h
```

## An unrolled loop with unique variables

```
h1 = w*h0
h2 = w*h1
h3 = w*h2
```

# Simplest RNN (hidden state only)

“activation *function*”

“hidden layer *activation*”

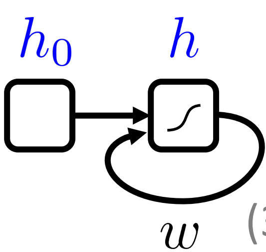
“initial hidden value(s)” $h_0$

“layer 1 hidden value(s) at step 1” $h_1 = f(\boxed{wh_0 + b})$

“layer 1 hidden value(s) at step 2” $h_2 = f(wh_1 + b)$

“layer 1 hidden value(s) at step 3” $h_3 = f(wh_2 + b)$

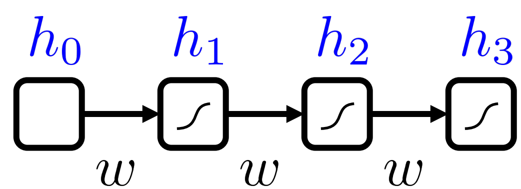
as loop




(3 times)

```
h = h0
for i in range(3):
    h = f(w*h + b)
```

as unrolled loop




weight is *shared* across steps
(like “convolution through time”!)

```
h1 = f(w*h0 + b)
h2 = f(w*h1 + b)
h3 = f(w*h2 + b)
```

# RNN w/ 1-layer, 1-hidden, 3-steps

"input sequence" $x_1, x_2, x_3$

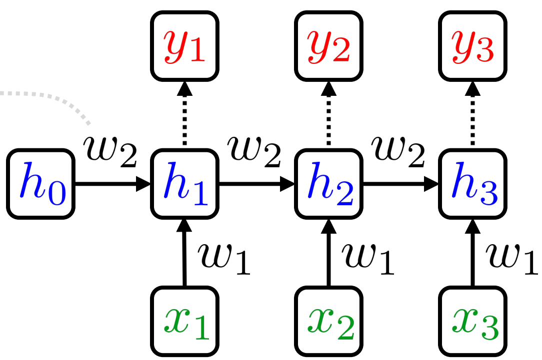
"initial hidden value(s)" $h_0$

"layer 1 hidden value(s)" (steps 1,2,3)

$$h_1 = f(w_1 x_1 + w_2 h_0 + b) \qquad y_1 = h_1$$
$$h_2 = f(w_1 x_2 + w_2 h_1 + b) \qquad y_2 = h_2$$
$$h_3 = f(w_1 x_3 + w_2 h_2 + b) \qquad y_3 = h_3$$

"output sequence"

"*hidden-to-hidden* weight(s)"




"*input-to-hidden* weight(s)"

```
h1 = f(w1*x1 + w2*h0 + b);  y1 = h1;
h2 = f(w1*x2 + w2*h1 + b);  y2 = h2;
h3 = f(w1*x3 + w2*h2 + b);  y3 = h3;
```

# RNN w/ 1-layer, 1-hidden, 3-steps

$$h_{1,1} = f(w_1 x_1 + w_2 h_{0,1} + b_1)$$
$$h_{2,1} = f(w_1 x_2 + w_2 h_{1,1} + b_1)$$
$$h_{3,1} = f(w_1 x_3 + w_2 h_{2,1} + b_1)$$

"layer 1 hidden value(s)"

$$h_{1,2} = f(w_3 h_{1,1} + w_4 h_{0,2} + b_2)$$
$$h_{2,2} = f(w_3 h_{2,1} + w_4 h_{1,2} + b_2)$$
$$h_{3,2} = f(w_3 h_{3,1} + w_4 h_{2,2} + b_2)$$

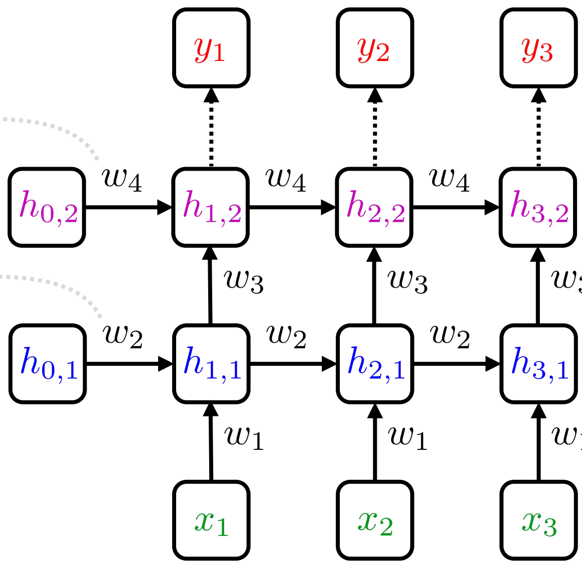
"layer 2 hidden value(s)"

$$y_1 = h_{1,2}$$
$$y_2 = h_{2,2}$$
$$y_3 = h_{3,2}$$

output sequence is copy of last final layer hidden values

"layer 2 *hidden-to-hidden* weight(s)"

"layer 1 *hidden-to-hidden* weight(s)"




"layer 2 *input-to-hidden* weight(s)"

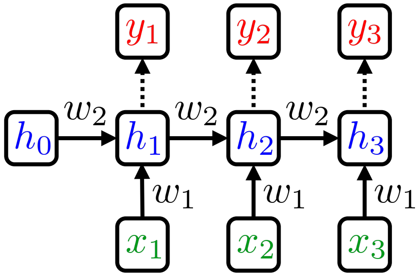
"layer 1 *input-to-hidden* weight(s)"

```
h11 = f(w1*x1 + w2*h01 + b1);  h12 = f(w3*h11 + w4*h02 + b2);  y1 = h12;
h21 = f(w1*x2 + w2*h11 + b1);  h22 = f(w3*h21 + w4*h12 + b2);  y2 = h22;
h31 = f(w1*x3 + w2*h21 + b1);  h32 = f(w3*h31 + w4*h22 + b2);  y3 = h32;
```

# Pure Python RNN example
# (1-layer, 1-hidden, 3-steps, ReLU, no bias)

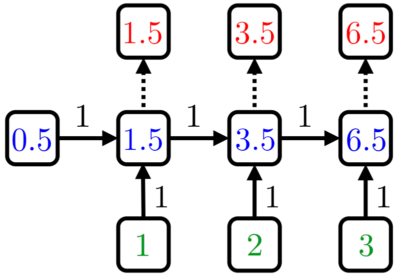
```python
def f(x):
    return max(0, x)    # relu
```

```python
def rnn(x1, x2, x3, h0, w1, w2):
    h1 = f(w1*x1 + w2*h0);  y1 = h1;
    h2 = f(w1*x2 + w2*h1);  y2 = h2;
    h3 = f(w1*x3 + w2*h2);  y3 = h3;
    return y1, y2, y3
```



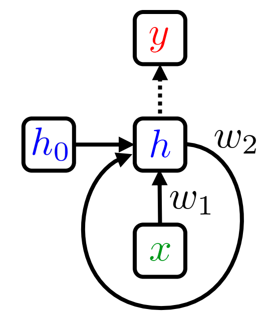

```python
h0 = 0.5
x1, x2, x3 = [1., 2., 3.]
w1, w2 = 1., 1.

y1, y2, y3 = rnn(x1, x2, x3, h0, w1, w2)
[y1, y2, y3]
```

```
[1.5, 3.5, 6.5]
```



# Pure Python RNN example (same, but variable number of steps)

```
def rnn(x, h0, w1, w2):
    y = []
    h = h0
    for xi in x:
        h = f(w1*xi + w2*h);
        y.append(h);
    return y
```




```
h0 = 0.5
x = [1., 2., 3.]
y = rnn(x, h0, w1, w2)      # length 3
y
```

```
[1.5, 3.5, 6.5]
```

```
rnn(x[:-1], h0, w1, w2)     # length 2
```

```
[1.5, 3.5]
```

`torch.nn.RNN(*args, **kwargs)` [SOURCE]

Applies a multi-layer Elman RNN with $\tanh$ or $\text{ReLU}$ non-linearity to an input sequence.

For each element in the input sequence, each layer computes the following function:

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

where $h_t$ is the hidden state at time *t*, $x_t$ is the input at time *t*, and $h_{(t-1)}$ is the hidden state of the previous layer at time *t-1* or the initial hidden state at time *0*. If `nonlinearity` is `'relu'`, then $\text{ReLU}$ is used instead of $\tanh$.

Parameters

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2`

```python
rnn = torch.nn.RNN(input_size=1, hidden_size=1,       # Create RNN object
                   num_layers=1, nonlinearity='relu')

print("w_ih:", rnn.weight_ih_l0.data)  # input-to-hidden weights
print("w_hh:", rnn.weight_hh_l0.data)  # hidden-to-hidden weights
print("b_ih:", rnn.bias_ih_l0.data)    # input-to-hidden bias
print("b_hh:", rnn.bias_hh_l0.data)    # hidden-to-hidden bias
```

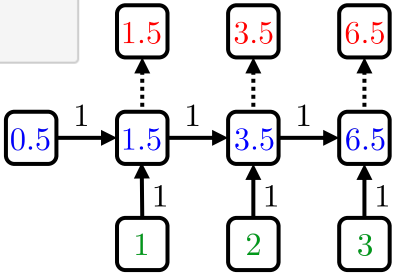
```
w_ih: tensor([[-0.3852]])
w_hh: tensor([[0.2682]])
b_ih: tensor([-0.0198])
b_hh: tensor([0.7929])
```

random initial weights … let's replace these

```python
# Set weighs to 1, biases to zero.
rnn.weight_ih_l0.data.fill_(1.);  rnn.bias_ih_l0.data.fill_(0.);
rnn.weight_hh_l0.data.fill_(1.);  rnn.bias_hh_l0.data.fill_(0.);
```

```python
X = torch.tensor([[[1.]],          # (L,N,D) format => shape (3,1,1)
                  [[2.]],          # L=seq_len, N=batch_size, D=num_inputs
                  [[3.]]])

H0 = torch.tensor([[[0.5]]])       # (S,N,M) format => shape (1,1,1)
                                   # S=num_layers, N=batch_size, M=num_hidden

Y, H3 = rnn(X, H0)                 # Run RNN on sequence X from initial H0
Y
```

```
tensor([[[1.5000]],

        [[3.5000]],

        [[6.5000]]], grad_fn=<StackBackward>)
```



“input dimension” means the number of features at *each* step in the sequence.

# Batching for PyTorch RNNs

PyTorch RNNs expect input to be in ($L$,$N$,$D$) format:

$L$ = sequence length, $N$ = batch size, $D$ = input dimension

but supports ($N$,$L$,$D$) format if explicitly requested

```
X = torch.tensor([[[1.], [10.]],    # (L,N,D) format => shape (3,2,1)
                  [[2.], [20.]],
                  [[3.], [30.]]])

H0 = torch.tensor([[[0.5],          # (S,N,M) format => shape (1,2,1)
                    [5.0]]])

Y, H3 = rnn(X, H0)
Y
```

```
tensor([[[ 1.5000],
         [15.0000]],

        [[ 3.5000],
         [35.0000]],

        [[ 6.5000],
         [65.0000]]], grad_fn=<StackBackward>)
```

# Variable-length sequences

Handled by *padding*. (Yes, redundant computations.)

```
X_list = [ torch.tensor([[ 1.],        # length 3 sequence
                         [ 2.],
                         [ 3.]]),
           torch.tensor([[10.],        # length 2 sequence
                         [20.]]) ]
X = torch.nn.utils.rnn.pad_sequence(X_list)
X
```

```
tensor([[[ 1.],
         [10.]],

        [[ 2.],
         [20.]],

        [[ 3.],
         [ 0.]]])
```

shorter sequences get padded with a default value

```
Y, _ = rnn(X, H0)    # Run the RNN on batch of sequences
Y
```

```
tensor([[[ 1.5000],
         [15.0000]],

        [[ 3.5000],
         [35.0000]],

        [[ 6.5000],
         [35.0000]]],
```

*it's your* job to ignore padded outputs

```
Y_list = [Y[:len(x),i] for i, x in enumerate(X_list)]
Y_list    # Padding has been stripped
```

```
[tensor([[1.5000],
         [3.5000],
         [6.5000]], grad_fn=<SelectBackward>),
 tensor([[15.],
         [35.]], grad_fn=<SelectBackward>)]
```

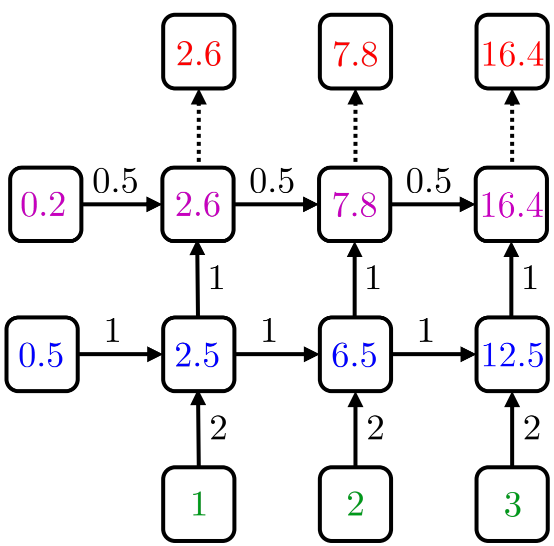
Or, PyTorch RNNs also accept *PackedSequence* objects as input, which omits the padding.
See the PyTorch docs.

# Pure Python RNN example (2-layer, 1-hidden, 3-steps, ReLU, no bias)

```
def rnn(x1, x2, x3, h01, h02, w1, w2, w3, w4):
    h11 = f(w1*x1 + w2*h01);  h12 = f(w3*h11 + w4*h02);  y1 = h12;
    h21 = f(w1*x2 + w2*h11);  h22 = f(w3*h21 + w4*h12);  y2 = h22;
    h31 = f(w1*x3 + w2*h21);  h32 = f(w3*h31 + w4*h22);  y3 = h32;
    return y1, y2, y3
```

```
h01, h02 = 0.5, 0.2
x1, x2, x3 = [1., 2., 3.]
w1, w2, w3, w4 = 2., 1., 1., .5

y1, y2, y3 = rnn(x1, x2, x3, h01, h02, w1, w2, w3, w4)
[y1, y2, y3]
```

```
[2.6, 7.8, 16.4]
```



# PyTorch version of previous slide
(but also batched)

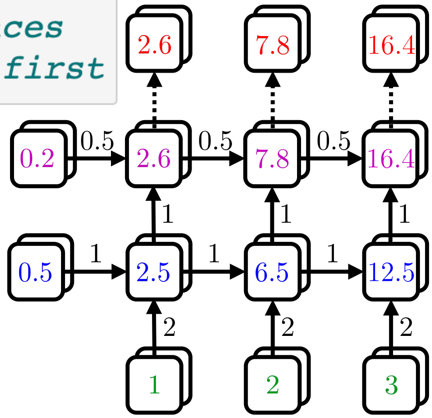
```
rnn = torch.nn.RNN(input_size=1, hidden_size=1,
                   num_layers=2, nonlinearity='relu')
```

```
# Layer 1 weights and biases
rnn.weight_ih_l0.data.fill_(2.);  rnn.bias_ih_l0.data.fill_(0.);
rnn.weight_hh_l0.data.fill_(1.);  rnn.bias_hh_l0.data.fill_(0.);

# Layer 2 weights and biases
rnn.weight_ih_l1.data.fill_(1.);  rnn.bias_ih_l1.data.fill_(0.);
rnn.weight_hh_l1.data.fill_(.5);  rnn.bias_hh_l1.data.fill_(0.);
```

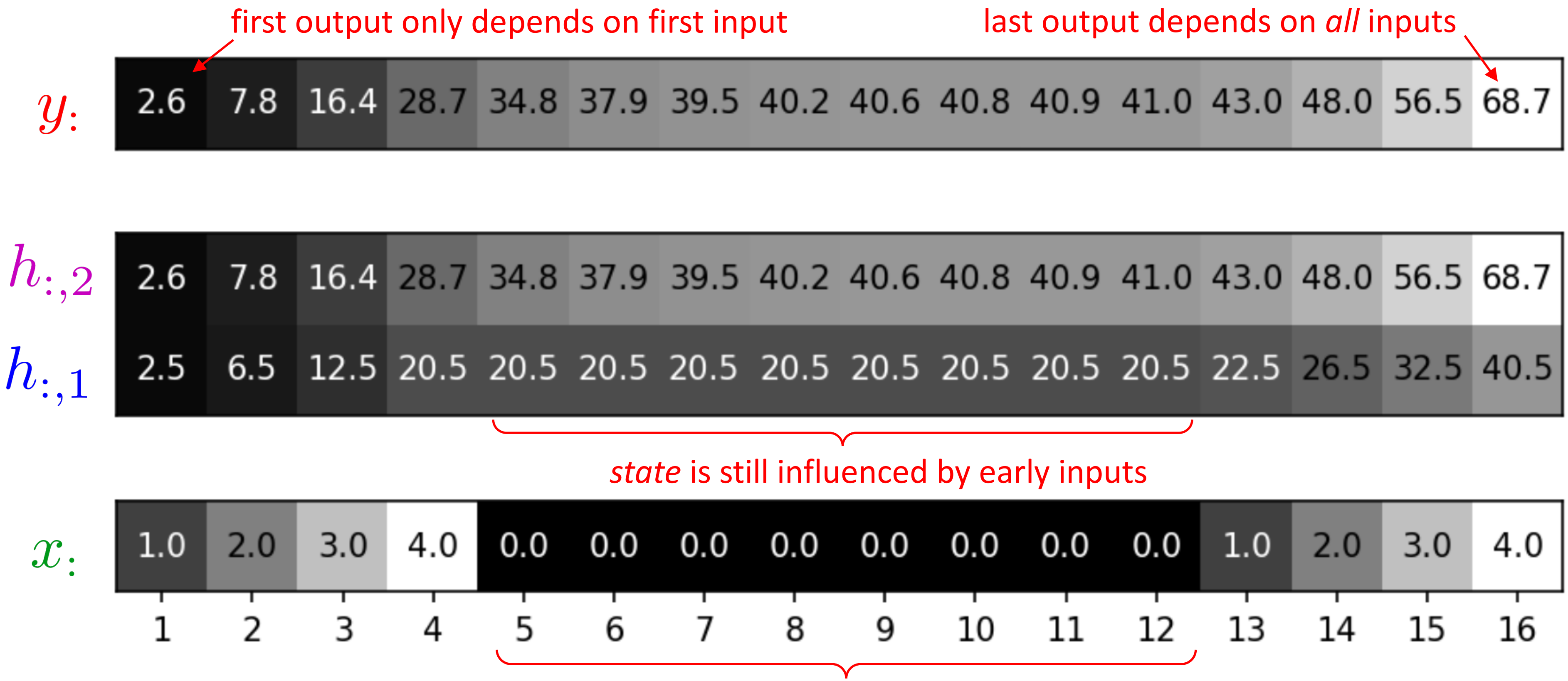
```
X = torch.tensor([[[1.], [10.]],      # (L,N,D) format => shape (3,2,1)
                  [[2.], [20.]],
                  [[3.], [30.]]])

H0 = torch.tensor([[[0.5], [5.0]],    # (S,N,M) format => shape (2,2,1)
                   [[0.2], [2.0]]])

Y, H3 = rnn(X, H0)                    # Run RNN on batch of sequences
Y                                     # where second sequence 10x first
```

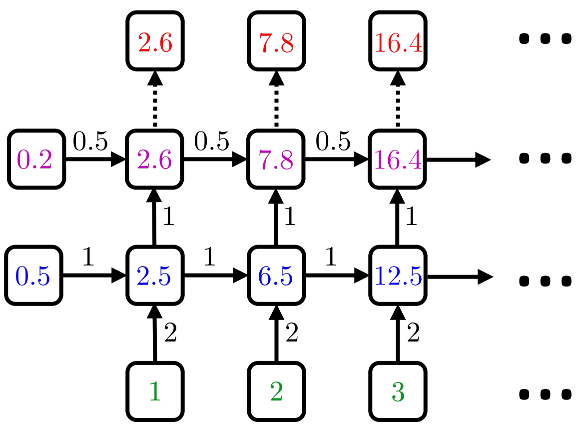
```
tensor([[[  2.6000],
         [ 26.0000]],

        [[  7.8000],
         [ 78.0000]],

        [[ 16.4000],
         [164.0000]]], grad_fn=<StackBackward>)
```

to demonstrate batching, 2nd sequence is 10x the 1st



# RNNs "remember" state across steps




In principle, RNNs can learn long-range dependencies.

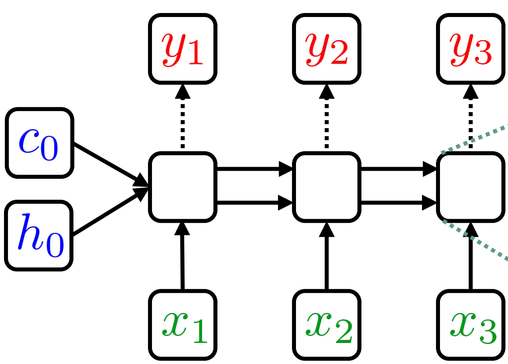
In practice, training RNNs can be hard:

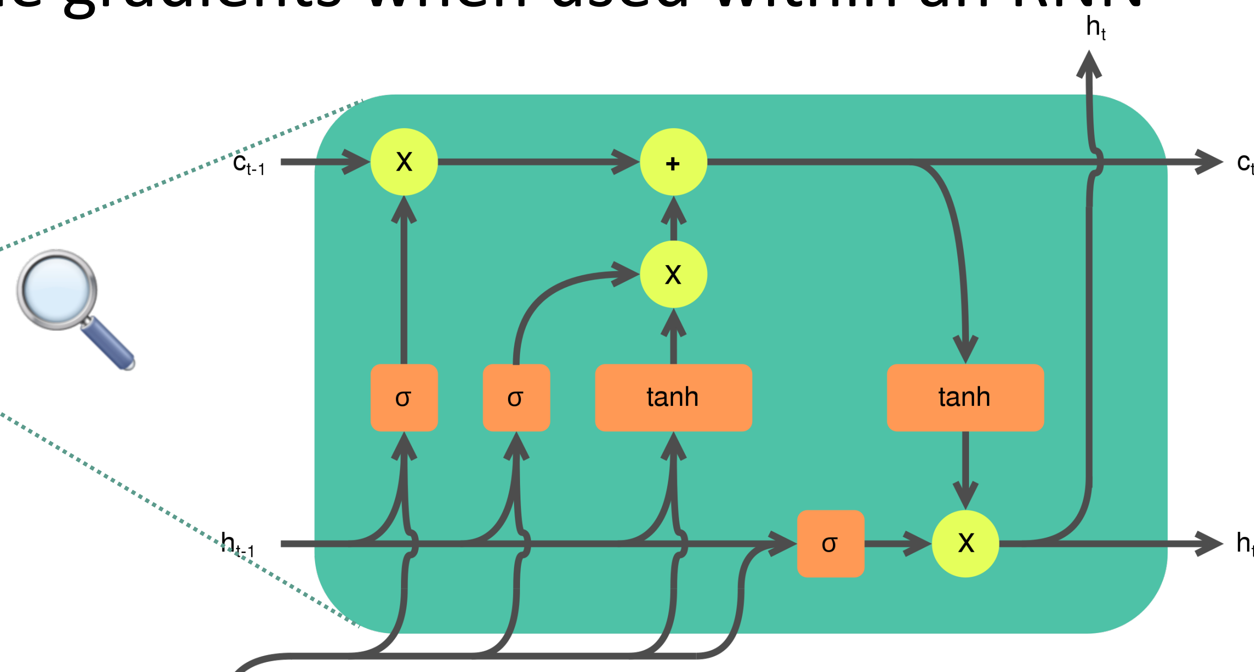
- Gradients can "vanish" (lose numerical precision)
- Gradients can "explode" (blow up to +/- infinity)

# Long Short-Term Memory

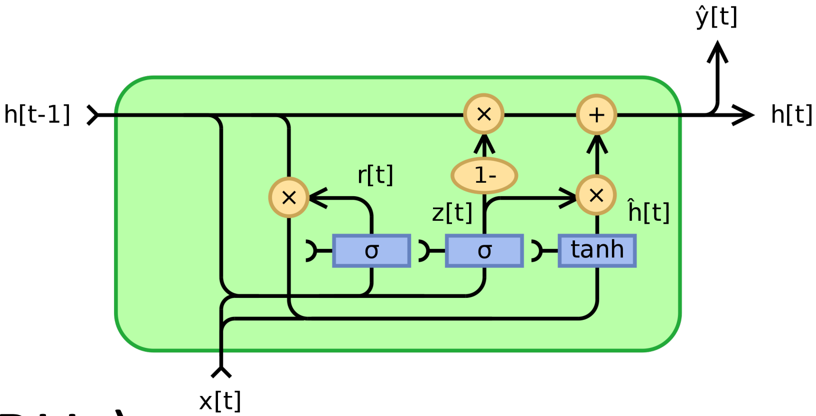
- *Problem:* simple "neurons" from neural networks, when used recurrently, are hard to train with gradient descent (vanishing / exploding gradients)

- *Idea:* replace "neuron" with a "cell" that is designed to have stable gradients when used within an RNN



https://commons.wikimedia.org/wiki/File:Gated_Recurrent_Unit,_base_type.svg




# Other extensions

- Gated Recurrent Networks (GRUs)
  - Basically a modern attempt to simplify LSTM architecture as much possible, while retaining benefits
- Bi-directional RNNs / LSTMs / GRUs
  - Accumulate state from *both directions* in the sequence, where each direction gets its *own* parameters
  - Not appropriate when predicting future from past
  - Much better for other kinds of sequence data, such as natural language processing or computer vision