COMP 432 Machine Learning

Recurrent Networks

Computer Science & Software Engineering
Concordia University, Fall 2021

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*"

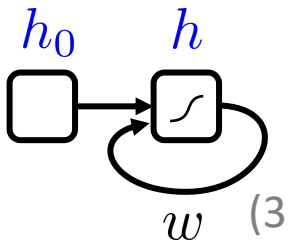
"initial hidden value(s)" h_0

"layer 1 hidden value(s) at step 1" $h_1 = f(wh_0 + b)$

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

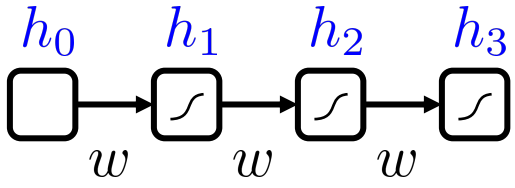
"layer 1 hidden value(s) at step 3" $h_3 = f(wh_2 + b)$

as loop



(3 times)

as unrolled loop



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

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

```
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

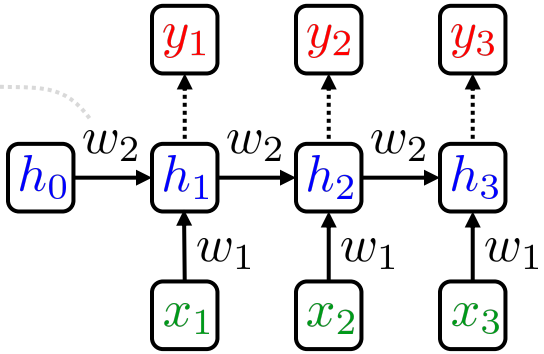
"layer 1 hidden value(s)" (steps 1,2,3)

$$h_1 = f(w_1 x_1 + w_2 h_0 + b)$$
$$h_2 = f(w_1 x_2 + w_2 h_1 + b)$$
$$h_3 = f(w_1 x_3 + w_2 h_2 + b)$$

$$y_1 = h_1$$
$$y_2 = h_2$$
$$y_3 = h_3$$

"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/ 2-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)"

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

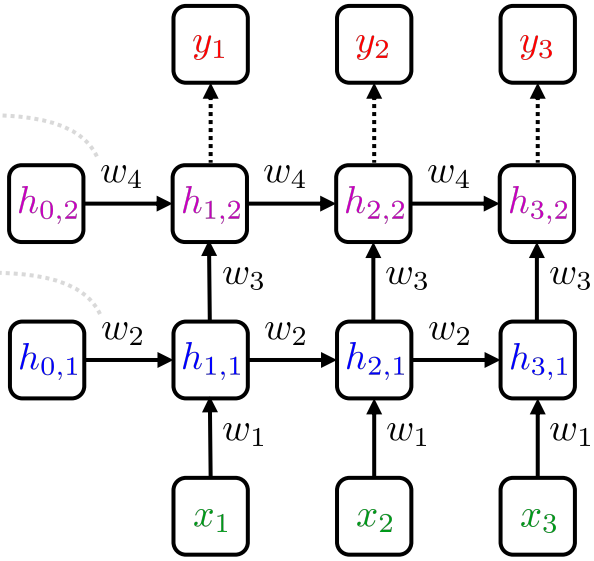
"layer 2 hidden value(s)"

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

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;
```

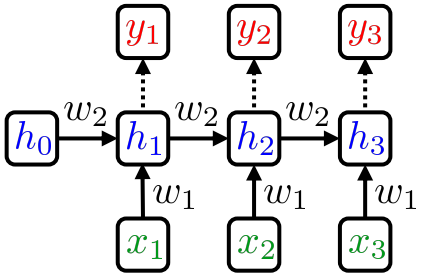
Pure Python RNN example (1-layer, 1-hidden, 3-steps, ReLU, no bias)

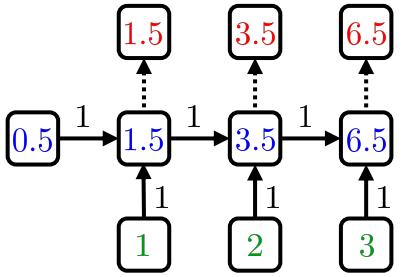
```
def f(x):
    return max(0, x)    # relu
```

```
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
```

```
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]
```

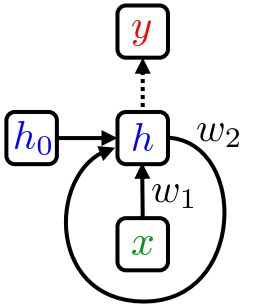
```
[1.5, 3.5, 6.5]
```





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 ReLU non-linearity to an input sequence.

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

$$\mathrm{h}_\mathrm{t} = \tanh(\mathrm{W}_{\mathrm{ih}}\mathrm{x}_\mathrm{t} + \mathrm{b}_{\mathrm{ih}} + \mathrm{W}_{\mathrm{hh}}\mathrm{h}_{(\mathrm{t}-1)} + \mathrm{b}_{\mathrm{hh}})$$

where h_t is the hidden state at time *t*, x_t is the input at time *t*, and $\mathrm{h}_{(\mathrm{t}-1)}$ 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 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`

```
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
```

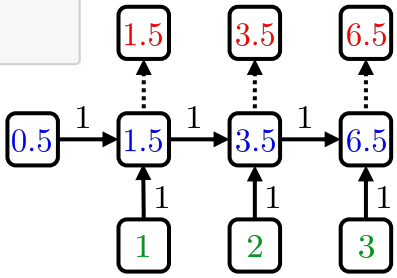
```
w_ih: tensor([[-0.3852]])
w_hh: tensor([[0.2682]])
b_ih: tensor([-0.0198])
b_hh: tensor([0.7929])
```

← random initial weights … let's replace these

```
# 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.);
```

```
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>)]
```

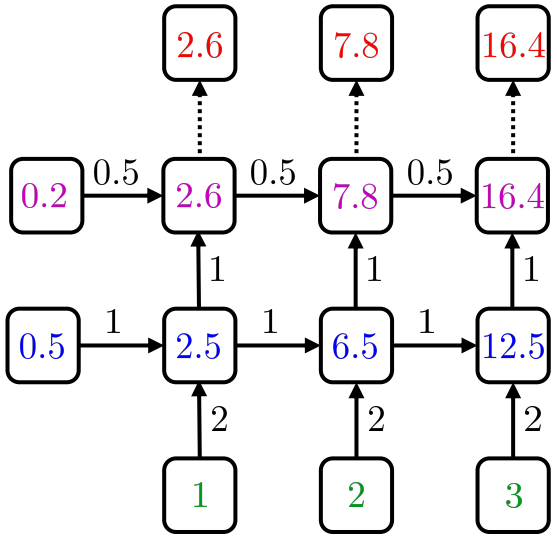
Or, PyTorch RNNs also accept *PackedSequence* objects as input, which omits the padding.
See the PyTorch docs.

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)

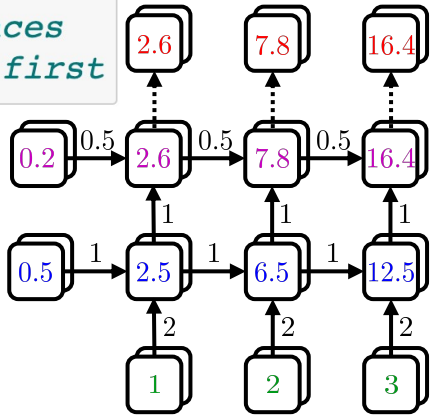
```
rnn = torch.nn.RNN(input_size=1, hidden_size=1,
                   num_layers=2, nonlinearity='relu')
```

```
# 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.);
```

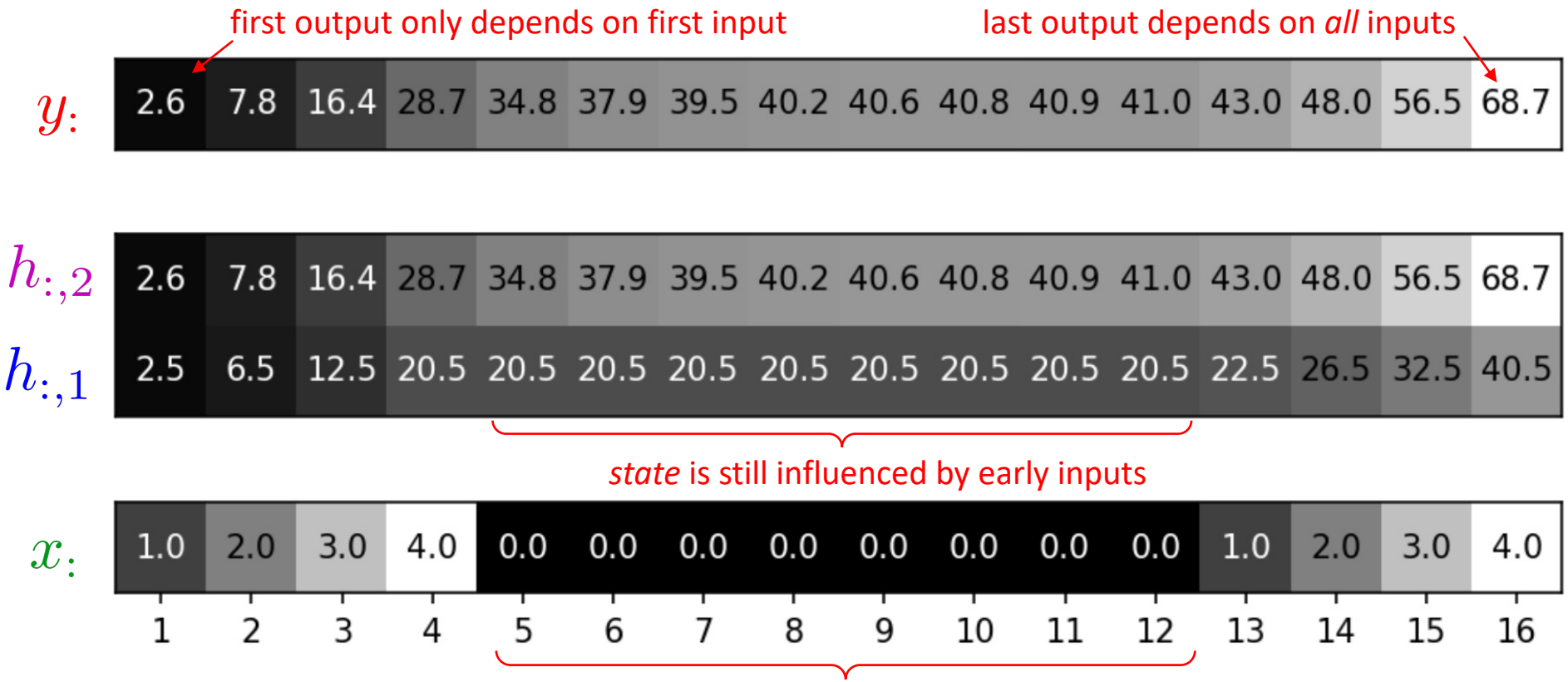
```
X = torch.tensor([[[1.], [10.]],      # (L,N,D) format => shape (3,2,1)
                  [[2.], [20.]],
                  [[3.], [30.]]])

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
```

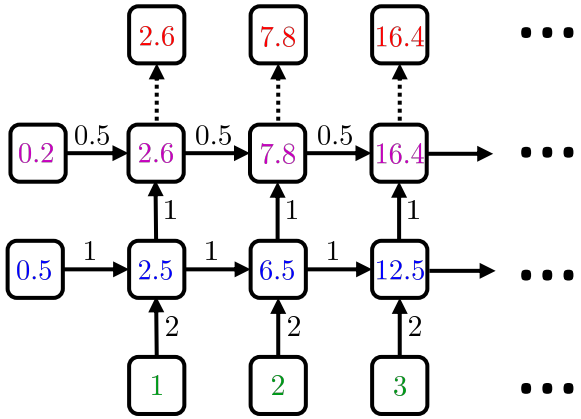
```
tensor([[[  2.6000],
         [ 26.0000]],

        [[  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)

Vanishing/exploding gradients

- During backprop, all steps "contribute" to the gradient of the shared RNN parameters.
- *Vanishing gradients:* the gradient contribution of earlier steps becomes vanishingly small; RNN never learns to recognize patterns in early sequence.
- *Exploding gradients:* RNN loss surfaces often have steep 'walls'; steepness implies a huge gradient that suddenly destabilizes training

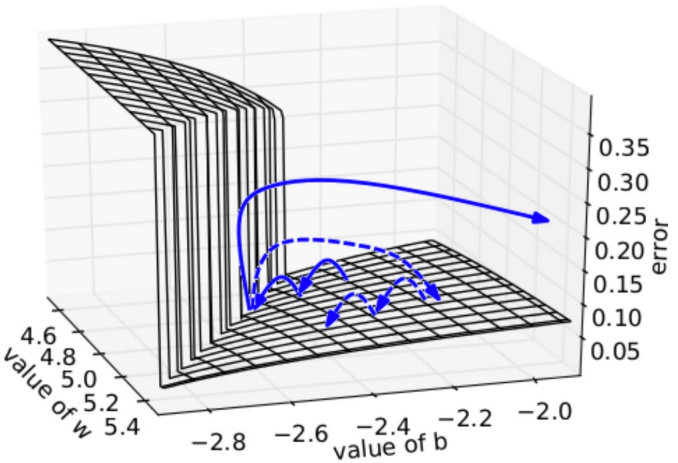
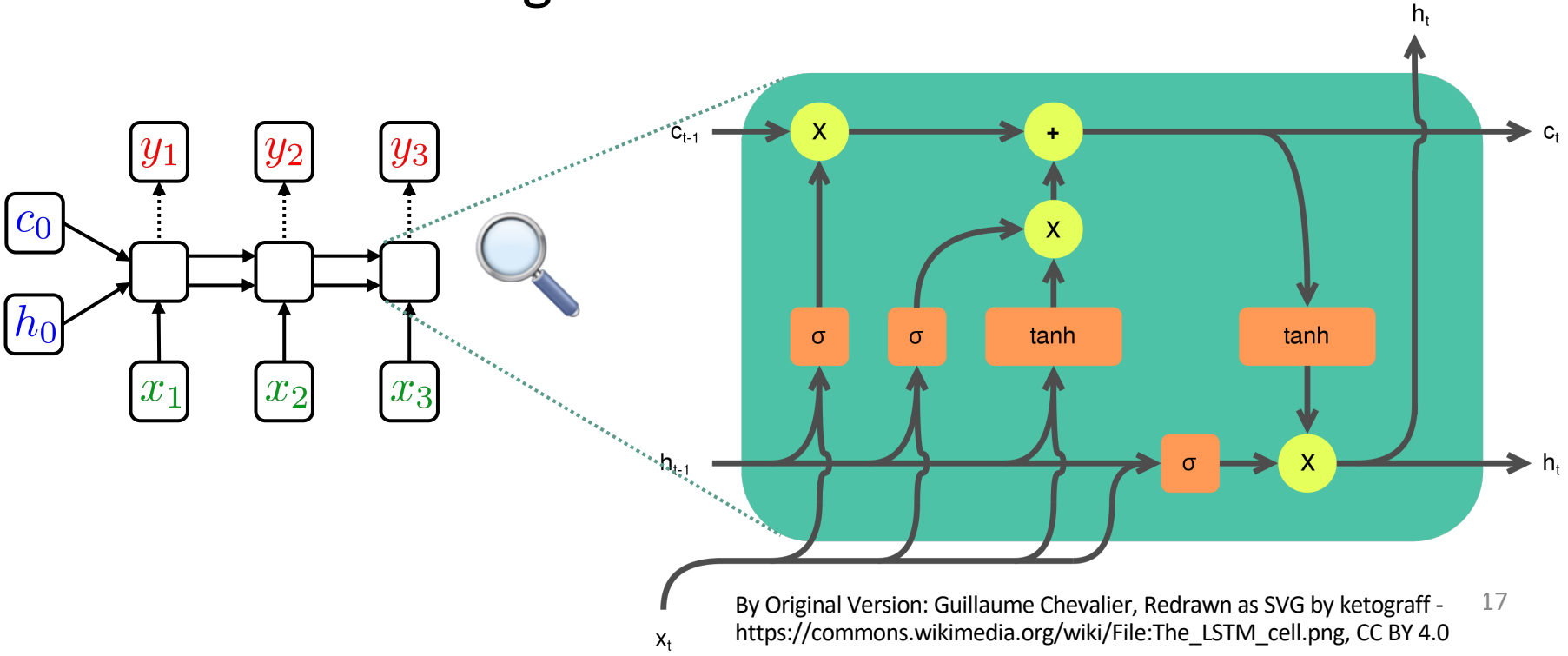


Image credit: Pascanu *et al.* 2013
https://arxiv.org/abs/1211.5063

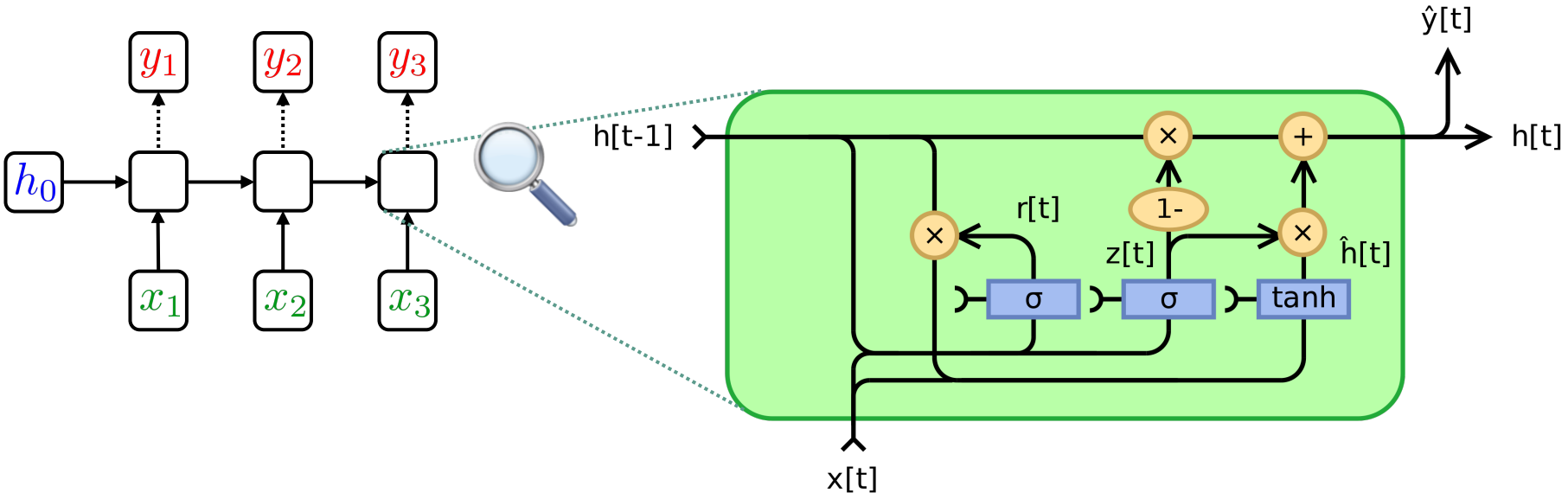
Long Short-Term Memory (LSTM)

- *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



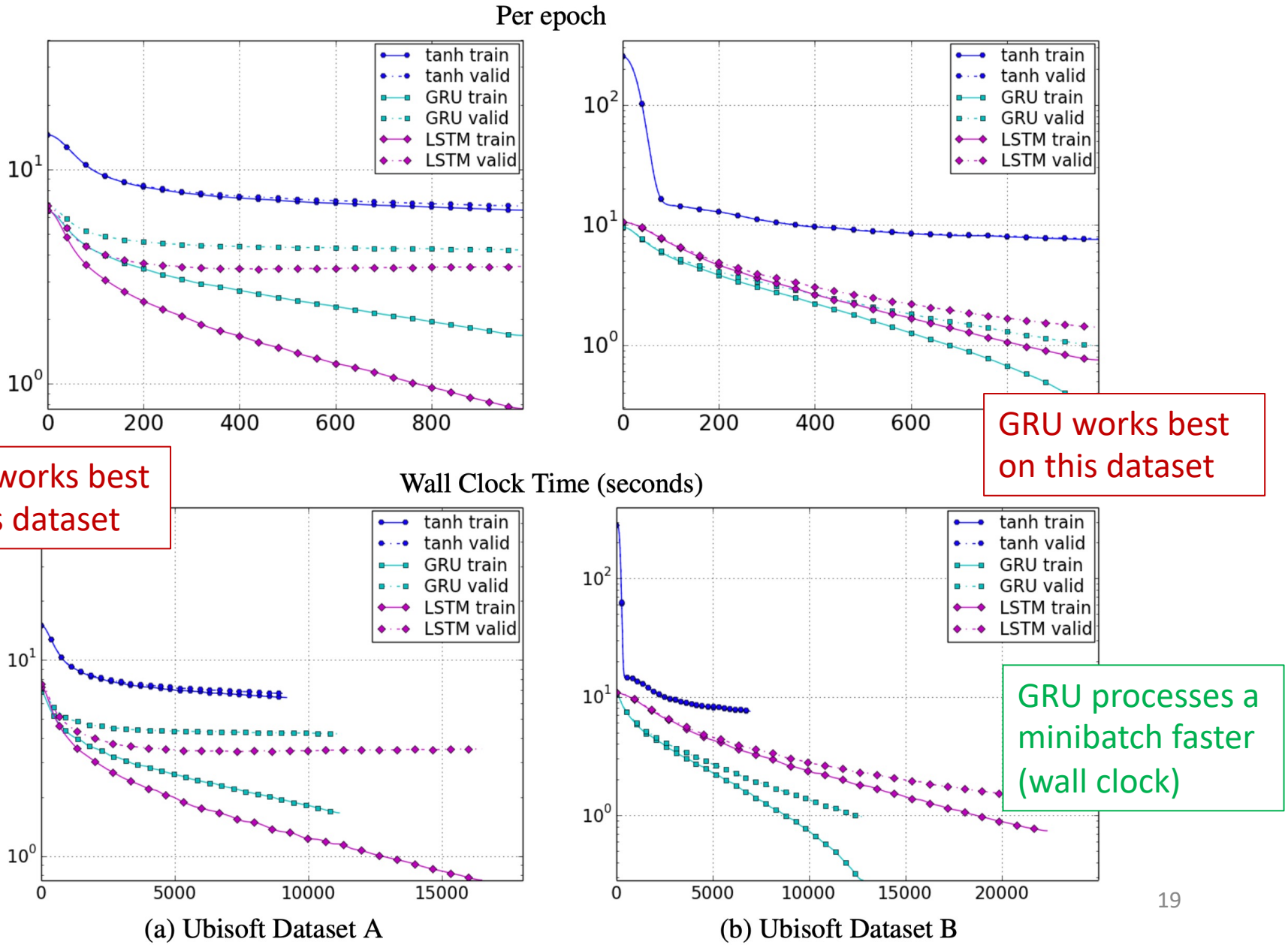
Gated Recurrent Units (GRUs)

- *Question:* Can we simplify LSTM "cell" architecture, yet still be trainable by gradient descent?
- GRUs are one early attempt to answer that question
  - "Reset gate" r[t] and "update gate" z[t] are in range [0,1]
  - They control how much of h[t-1] passes directly to h[t]
  - As training progresses, h[t] can differ strongly from h[t-1]



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

Figure from “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling” by Chung et al 2014, https://arxiv.org/abs/1412.3555

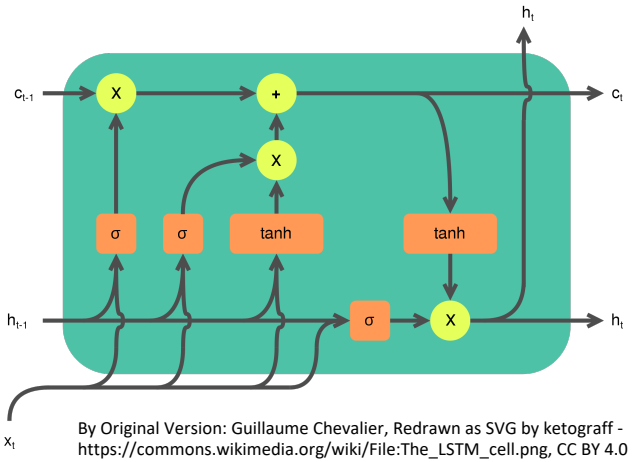


`torch.nn.LSTM(*args, **kwargs)` [SOURCE]

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

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

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



By Original Version: Guillaume Chevalier, Redrawn as SVG by ketograff - https://commons.wikimedia.org/wiki/File:The_LSTM_cell.png, CC BY 4.0

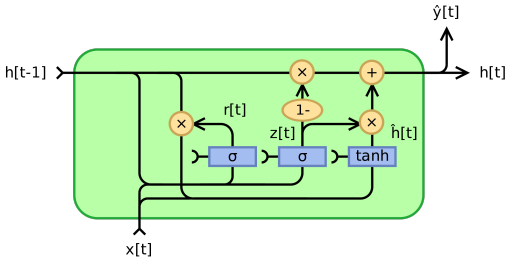
where h_t is the hidden state at time *t*, c_t is the cell state at time *t*, x_t is the input at time *t*, h_{t-1} is the hidden state of the layer at time *t-1* or the initial hidden state at time *0*, and i_t , f_t , g_t , o_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function, and $\odot$ is the Hadamard product.

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

Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.

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

$$
\begin{aligned}
r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \\
z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \\
n_t &= \tanh(W_{in}x_t + b_{in} + r_t * (W_{hn}h_{(t-1)} + b_{hn})) \\
h_t &= (1 - z_t) * n_t + z_t * h_{(t-1)}
\end{aligned}
$$



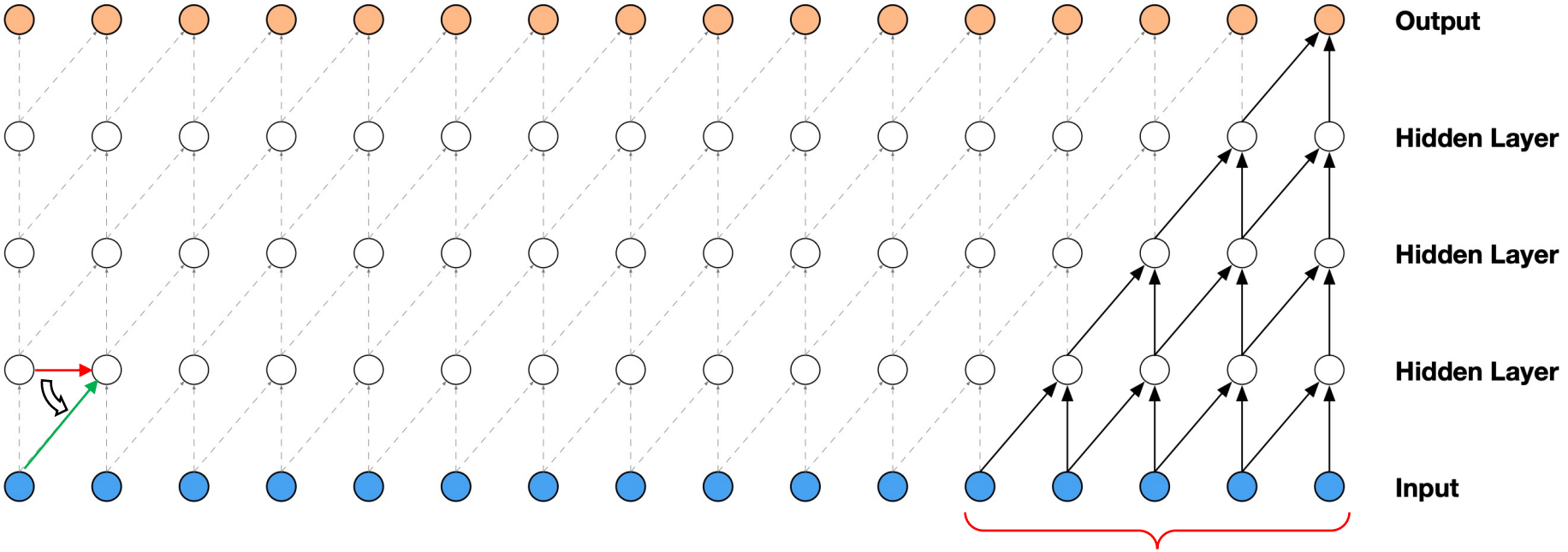
https://commons.wikimedia.org/wiki/File:Gated_Recurrent_Unit,_base_type.svg

where h_t is the hidden state at time *t*, x_t is the input at time *t*, $h_{(t-1)}$ is the hidden state of the layer at time *t-1* or the initial hidden state at time *0*, and r_t , z_t , n_t are the reset, update, and new gates, respectively. σ is the sigmoid function, and $*$ is the Hadamard product.

Other extensions

- Chris Olah's intro to LSTMs is excellent guide
- Bi-directional RNNs / LSTMs / GRUs
  - *Idea:* Each output should accumulate state from both directions in the sequence
    - Each layer comprises two RNNs, running in opposite directions
    - Each direction has its own parameters
  - Can still be applied even when predicting future from past, so long as features from "future" not used as input.
  - Can be better for many kinds of sequence data, such as text, computer vision, DNA sequences, …
- Convolutional RNNs
  - *Idea:* stack the RNN layers on top of convolutions, so the input sequence is transformed before RNN sees it.
  - Convolution weights and RNN weights still trained *jointly*

Convolutions versus RNNs

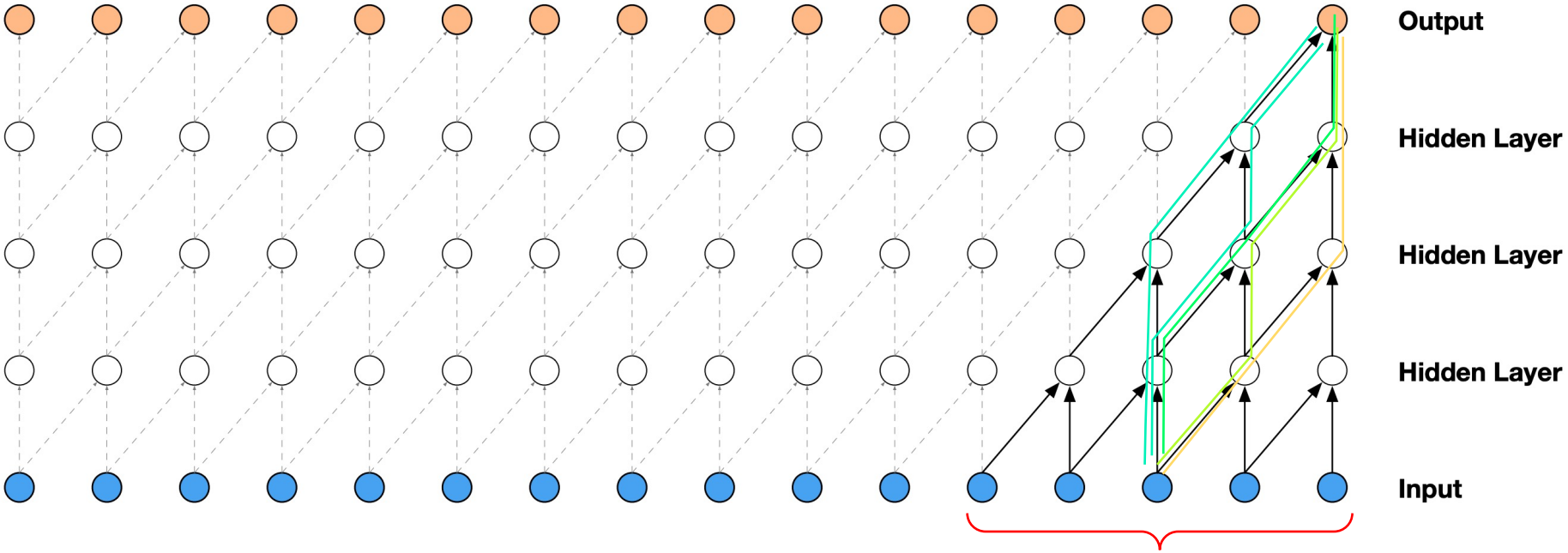


- Stacking convolutions is like moving the RNN's hidden-to-hidden weights to be "previous-input-to-hidden" weights
- Larger convolution filter sizes mean looking "farther backward" among the inputs.

However, the last output only depends on a small "receptive field", and is independent of all previous inputs

Figure from WaveNet paper: https://arxiv.org/abs/1609.03499

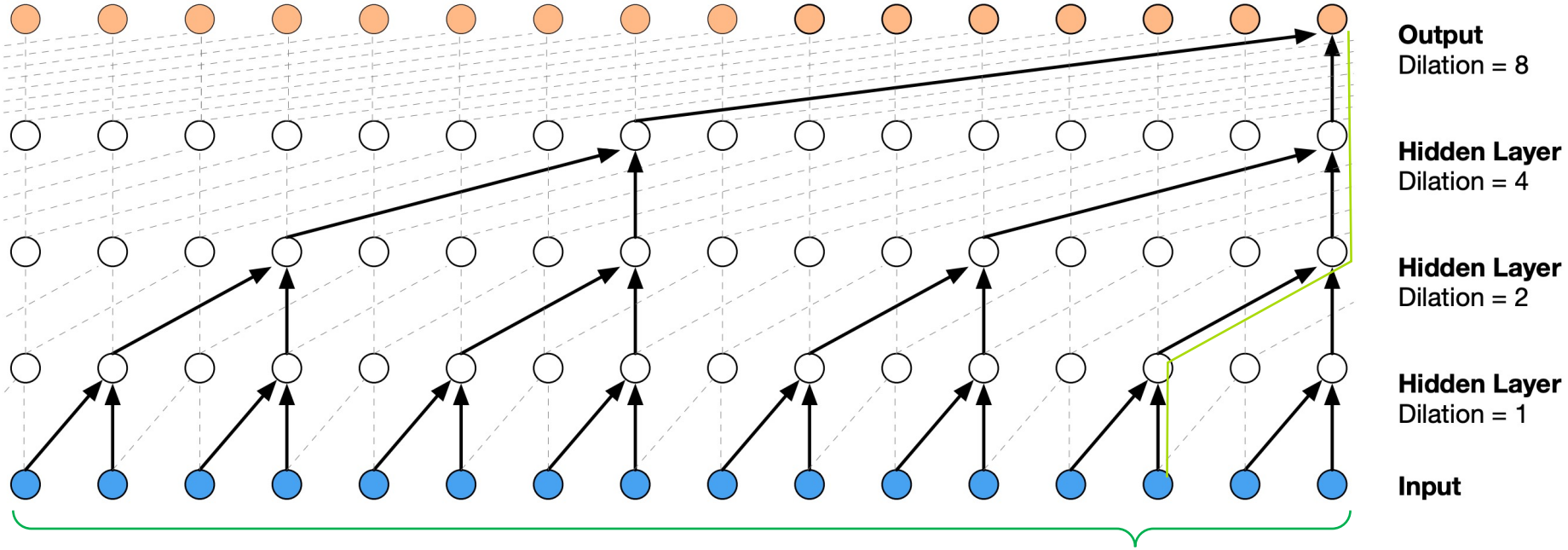
Standard convolutions re-use features and hidden values many times



- Notice that an input can influence the output along many paths
- Can we exchange these overlapping paths for a larger receptive field?

Figure from WaveNet paper: https://arxiv.org/abs/1609.03499

“Dilated” convolutions expand the receptive field by skipping positions



- By skipping steps, same filter size can provide much larger receptive field.
- With full dilation (as shown above), each output step depends on each input step through only *one path*.

  However, realize that each circle in this figure could represent a vector of inputs / hidden values, so within an input step the individual features can still have multiple ‘paths’ to an output value.
- Dilation allows longer dependencies to be learned, and is easier to train than RNNs, but only works up to limit of receptive field. Can be *combined* with RNNs.

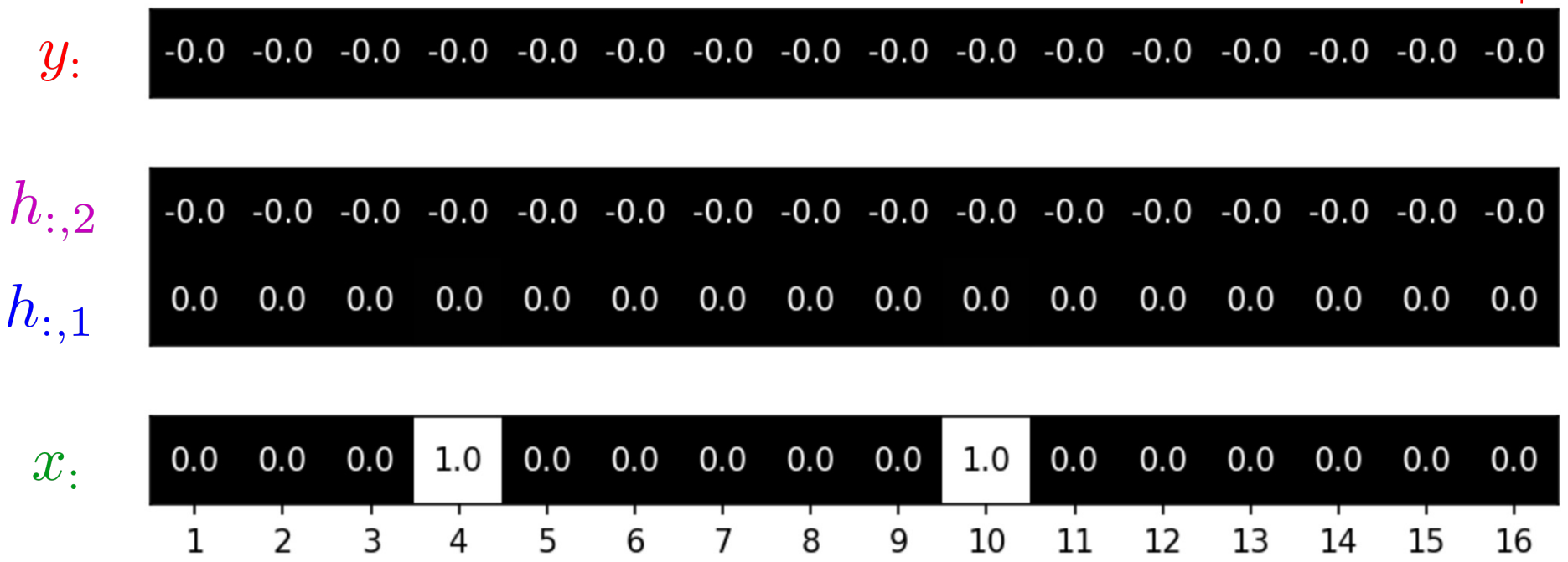
Figure from WaveNet paper: https://arxiv.org/abs/1609.03499

RNN before training

target 1.0

output z 0.5

Synthetic task: predict *1.0* if sequence contains any "+1"
In this task, loss is computed from final output only

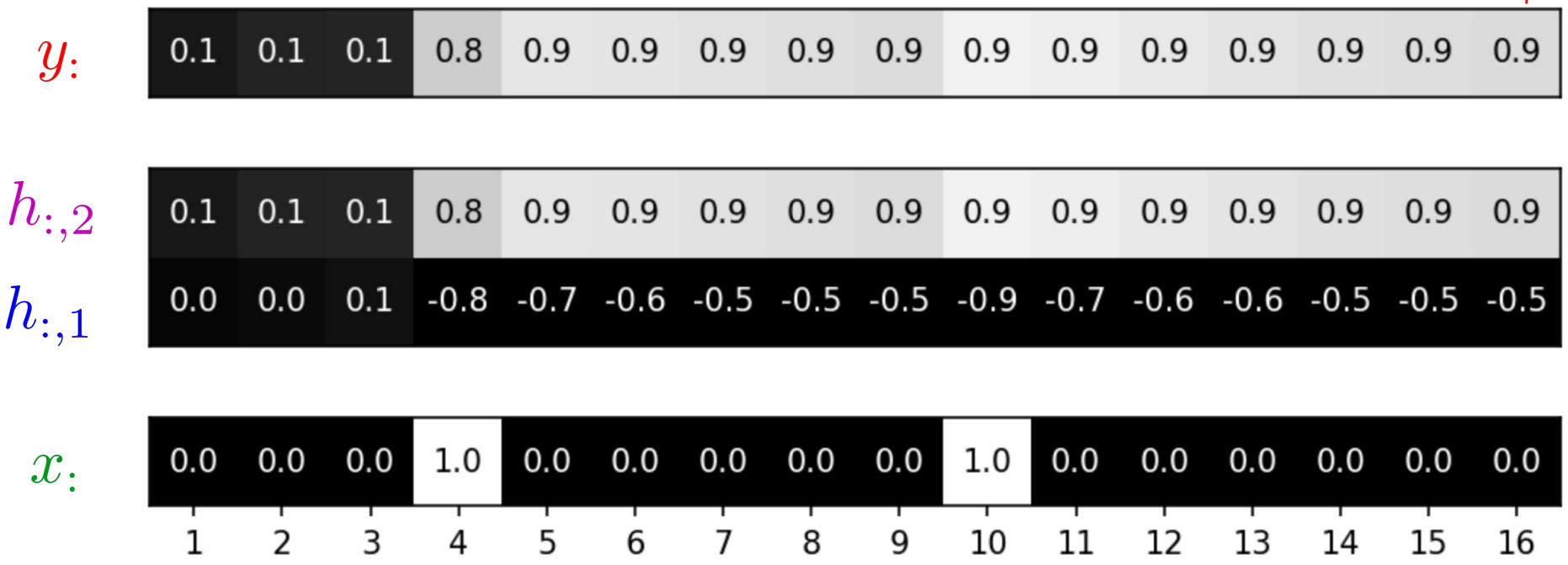


RNN **after** training

target 1.0

output z 1.0

Synthetic task: predict *1.0* if sequence contains any "+1"
In this task, loss is computed from final output only



How the synthetic training set for the synthetic task on previous slide was generated... (for completeness)

```
L = 16     # Sequence length 16
N = 500    # 500 examples
D = 1      # One input feature
X = torch.bernoulli(torch.full((L, N, D), 0.05)).type(torch.float32)   # Mostly 0s, some 1s
y = torch.any(X >= 1.0, dim=0).type(torch.float32)                     # Positive if has 1
```

```
print(X[:,:5,0])  # Print first few sequences as columns
print(y[:5,0])    # Print the targets too, for reference
```

```
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [1., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 1.],
        [1., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])
tensor([1., 0., 1., 1., 1.])
```

The first column is the example input sequence [0,0,0,1,0,...] that was used on previous slide.

Other columns represent other training sequences.

These are the synthetic targets for the 5 sequences shown.

(…cont'd)

Define 2-layer tanh RNN that returns its final output (x10 as a hack) as its prediction

(Why the times 10 hack in this example? Because tanh in range [-1,1] but when we feed that value as a 'logit' of a logistic sigmoid [-10,+10] allows sigmoid to predict 0.0 or 1.0, but range [-1,+1] doesn't!)

```
class ExampleRNN(torch.nn.Module):
    def __init__(self):
        super(ExampleRNN, self).__init__()
        self.rnn = torch.nn.RNN(input_size=D, hidden_size=1, num_layers=2)

        for p in self.rnn.parameters():  # Use smaller initial values
            p.data *= 0.01

    def forward(self, X):
        Z, _ = self.rnn(X)  # Run the standard RNN
        z = 10*Z[-1:,:,0]   # Extract and scale final output from each exa
        return z.T          # Shape (1,N) -> (N,1) to match targets

rnn = ExampleRNN()
```

Run the training loop using binary cross-entropy (BCE) loss, where we feed t

```
loss = torch.nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(rnn.parameters(), lr=0.001)
batch_size = 100
num_epoch = 500
for epoch in range(num_epoch):
    for i in range(0, N, batch_size):
        Xi = X[:,i:i+batch_size,:]
        yi = y[i:i+batch_size,:]
        z = rnn(Xi)
        l = loss(z, yi)
        rnn.zero_grad()
        l.backward()
        optimizer.step()

    if (epoch+1) % 50 == 0:
        print("%03d: %.4f" % (epoch+1, loss(rnn(X_trn), y_trn).item()))
```

```
050: 0.6315
100: 0.0121
150: 0.0065
200: 0.0044
250: 0.0032
300: 0.0024
350: 0.0018
400: 0.0015
450: 0.0012
500: 0.0010
```

This is all the code that trained the toy RNN we used in the slide.