



10-418/10-618 Machine Learning for Structured Data

Machine Learning Department
School of Computer Science
Carnegie Mellon University



















































Recurrent Neural Networks (RNNs) + Module-based Automatic Differentiation

Matt Gormley
Lecture 2
Aug. 31, 2022

RECURRENT NEURAL NETWORKS

Dataset for Supervised Part-of-Speech (POS) Tagging

Data: $\mathcal{D} = \{\mathbf{x}^{(n)}, \mathbf{y}^{(n)}\}_{n=1}^N$

Sample 1:							$\mathbf{y}^{(1)}$
							$\mathbf{x}^{(1)}$
Sample 2:							$\mathbf{y}^{(2)}$
							$\mathbf{x}^{(2)}$
Sample 3:							$\mathbf{y}^{(3)}$
							$\mathbf{x}^{(3)}$
Sample 4:							$\mathbf{y}^{(4)}$
							$\mathbf{x}^{(4)}$

Dataset for Supervised Handwriting Recognition

Data: $\mathcal{D} = \{x^{(n)}, y^{(n)}\}_{n=1}^N$



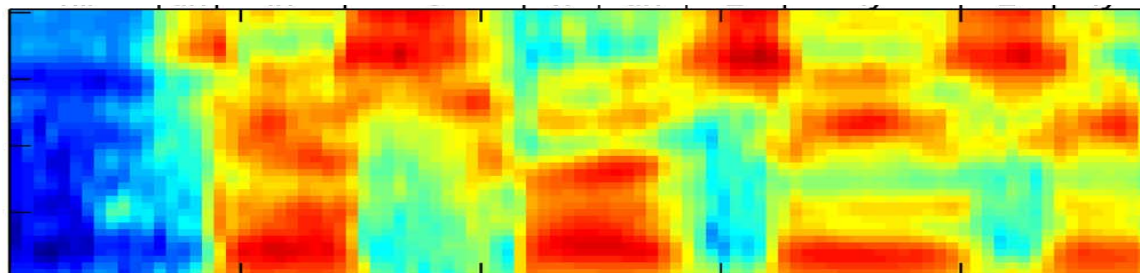
Dataset for Supervised Phoneme (Speech) Recognition

Data: $\mathcal{D} = \{\mathbf{x}^{(n)}, \mathbf{y}^{(n)}\}_{n=1}^N$

Sample 1:



} $\mathbf{y}^{(1)}$

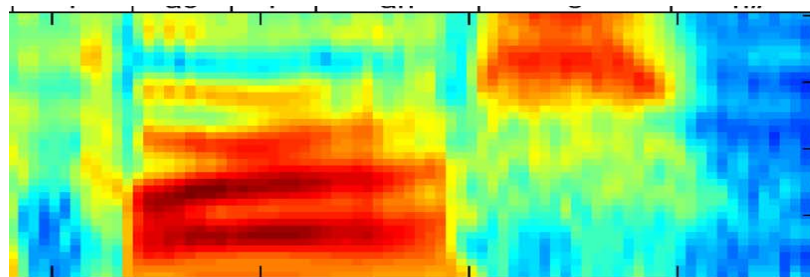


} $\mathbf{x}^{(1)}$

Sample 2:



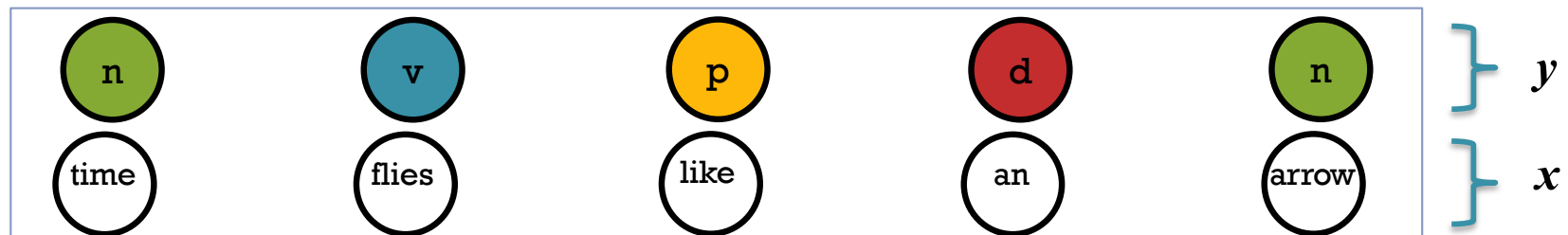
} $\mathbf{y}^{(2)}$



} $\mathbf{x}^{(2)}$

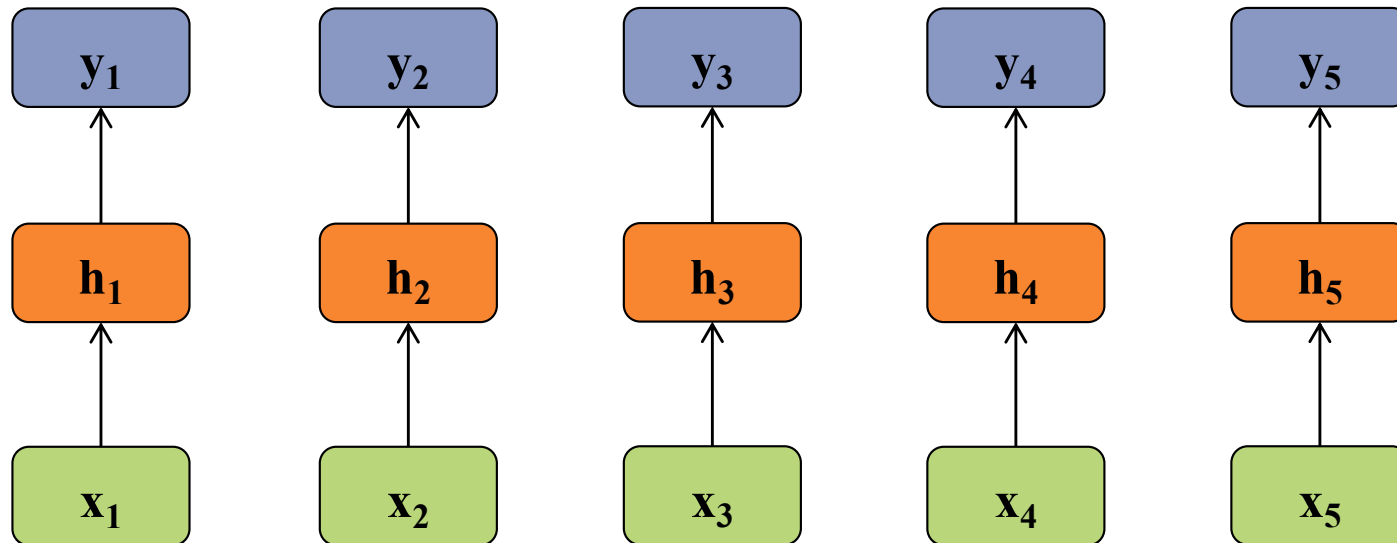
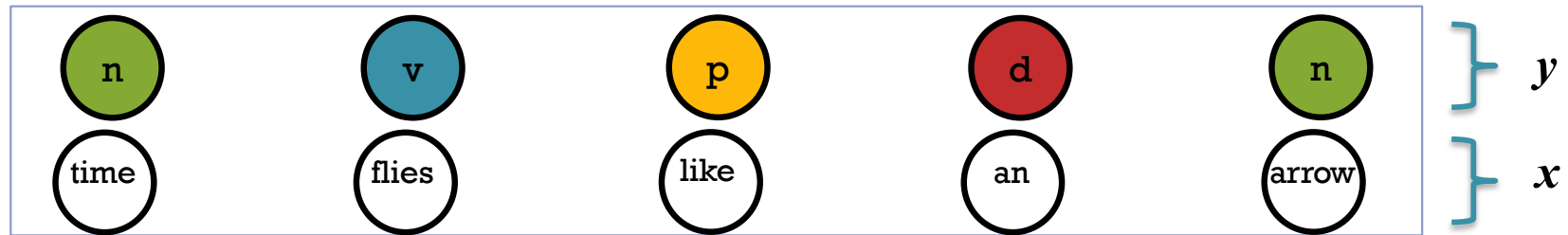
Time Series Data

Question 1: How could we apply the neural networks we've seen so far (which expect **fixed size input/output**) to a prediction task with **variable length input/output**?



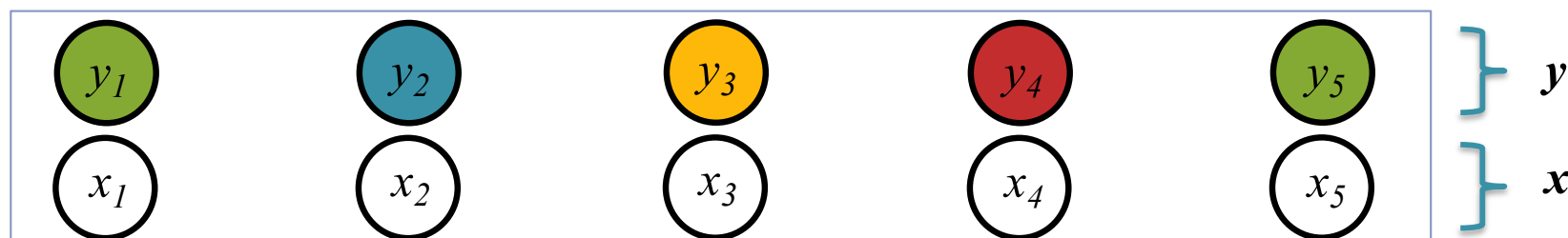
Time Series Data

Question 1: How could we apply the neural networks we've seen so far (which expect **fixed size input/output**) to a prediction task with **variable length input/output**?



Time Series Data

Question 2: How could we incorporate context (e.g. words to the left/right, or tags to the left/right) into our solution?



**Multiple
Choice:**

Working left-
to-right, use
features of...

	x_{i-1}	x_i	x_{i+1}	y_{i-1}	y_i	y_{i+1}
A	✓					
B				✓		
C	✓			✓		
D	✓			✓	✓	✓
E	✓	✓		✓	✓	✓
F	✓	✓	✓	✓		
G	✓	✓	✓	✓	✓	
H	✓	✓	✓	✓	✓	✓

Recurrent Neural Networks (RNNs)

inputs: $\mathbf{x} = (x_1, x_2, \dots, x_T), x_i \in \mathcal{R}^I$

hidden units: $\mathbf{h} = (h_1, h_2, \dots, h_T), h_i \in \mathcal{R}^J$

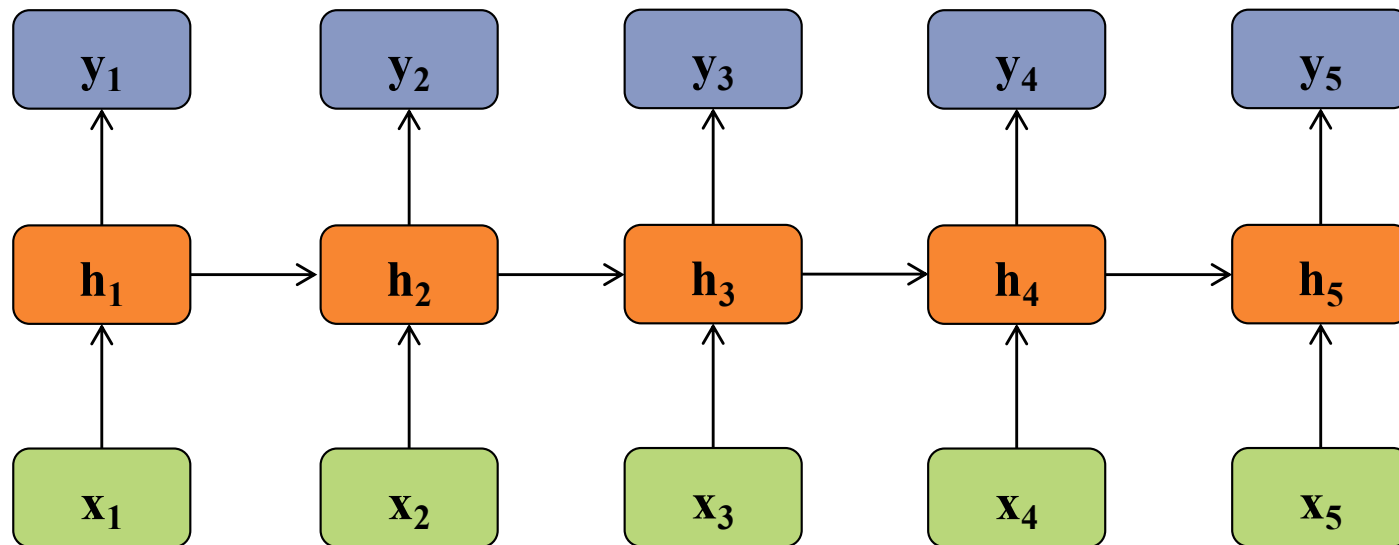
outputs: $\mathbf{y} = (y_1, y_2, \dots, y_T), y_i \in \mathcal{R}^K$

nonlinearity: \mathcal{H}

Definition of the RNN:

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

$$y_t = W_{hy}h_t + b_y$$



Recurrent Neural Networks (RNNs)

inputs: $\mathbf{x} = (x_1, x_2, \dots, x_T), x_i \in \mathcal{R}^I$

hidden units: $\mathbf{h} = (h_1, h_2, \dots, h_T), h_i \in \mathcal{R}^J$

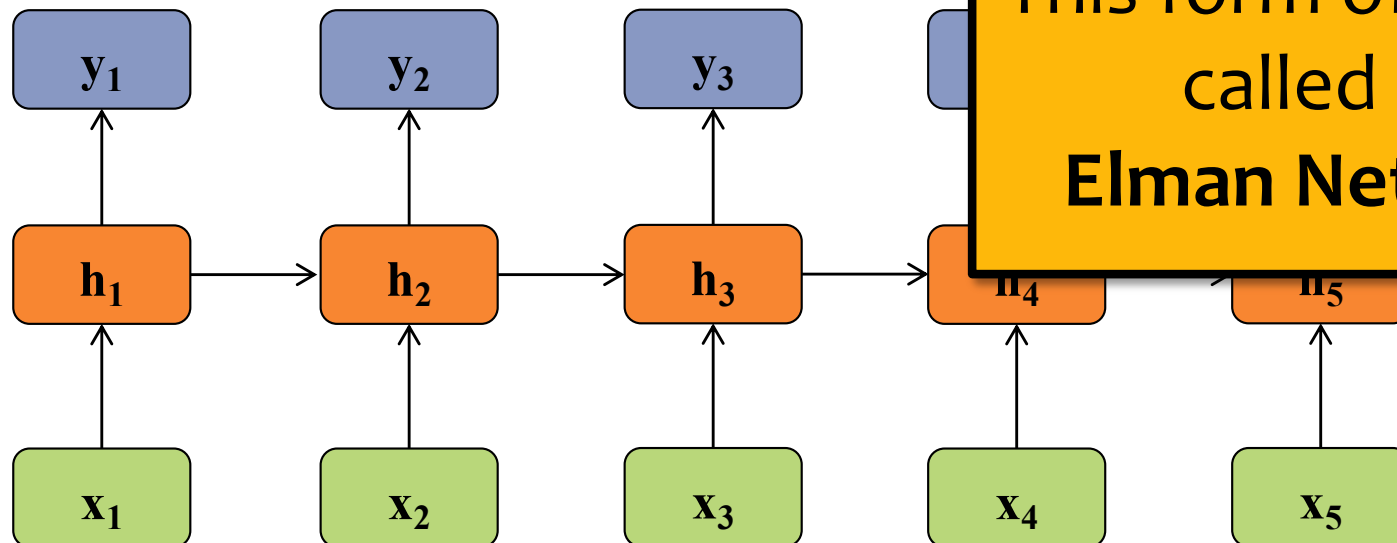
outputs: $\mathbf{y} = (y_1, y_2, \dots, y_T), y_i \in \mathcal{R}^K$

nonlinearity: \mathcal{H}

Definition of the RNN:

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

$$y_t = W_{hy}h_t + b_y$$



Recurrent Neural Networks (RNNs)

inputs: $\mathbf{x} = (x_1, x_2, \dots, x_T), x_i \in \mathcal{R}^I$

hidden units: $\mathbf{h} = (h_1, h_2, \dots, h_T), h_i \in \mathcal{R}^J$

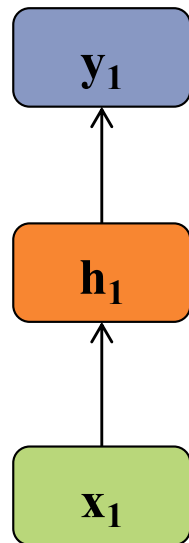
outputs: $\mathbf{y} = (y_1, y_2, \dots, y_T), y_i \in \mathcal{R}^K$

nonlinearity: \mathcal{H}

Definition of the RNN:

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

$$y_t = W_{hy}h_t + b_y$$



- If $T=1$, then we have a standard feed-forward **neural net with one hidden layer**
- All of the deep nets from last lecture required **fixed size inputs/outputs**

A Recipe for Machine Learning

1. Given training data:

$$\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$$

2. Choose each of these:

- Decision function

$$\hat{\mathbf{y}} = f_{\boldsymbol{\theta}}(\mathbf{x}_i)$$

- Loss function

$$\ell(\hat{\mathbf{y}}, \mathbf{y}_i) \in \mathbb{R}$$

3. Define goal:

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \sum_{i=1}^N \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{y}_i)$$

4. Train with SGD:

(take small steps
opposite the gradient)

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{y}_i)$$

A Recipe for Machine Learning

1. • Recurrent Neural Networks (RNNs) provide another form of **decision function**
• An RNN is just another differential function

2. CHOOSE EACH OF THESE:

– Decision function

$$\hat{y} = f_{\theta}(x_i)$$

4. Train with SGD:

(take small steps
opposite the gradient)

- We'll just need a method of computing the gradient efficiently
- Let's use Backpropagation Through Time...

$$-\eta_t \nabla \ell(f_{\theta}(x_i), y_i)$$

Recurrent Neural Networks (RNNs)

inputs: $\mathbf{x} = (x_1, x_2, \dots, x_T), x_i \in \mathcal{R}^I$

hidden units: $\mathbf{h} = (h_1, h_2, \dots, h_T), h_i \in \mathcal{R}^J$

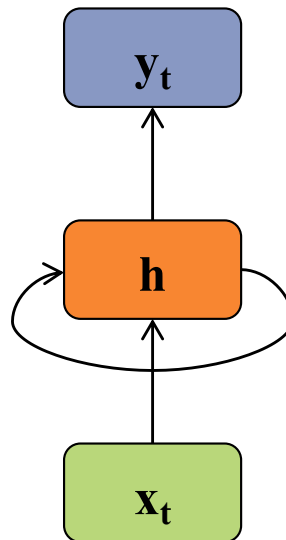
outputs: $\mathbf{y} = (y_1, y_2, \dots, y_T), y_i \in \mathcal{R}^K$

nonlinearity: \mathcal{H}

Definition of the RNN:

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

$$y_t = W_{hy}h_t + b_y$$



Recurrent Neural Networks (RNNs)

inputs: $\mathbf{x} = (x_1, x_2, \dots, x_T), x_i \in \mathcal{R}^I$

hidden units: $\mathbf{h} = (h_1, h_2, \dots, h_T), h_i \in \mathcal{R}^J$

outputs: $\mathbf{y} = (y_1, y_2, \dots, y_T), y_i \in \mathcal{R}^K$

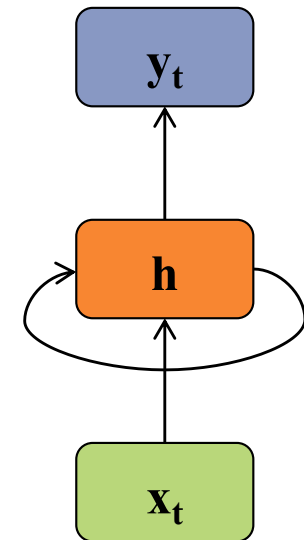
nonlinearity: \mathcal{H}

Definition of the RNN:

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

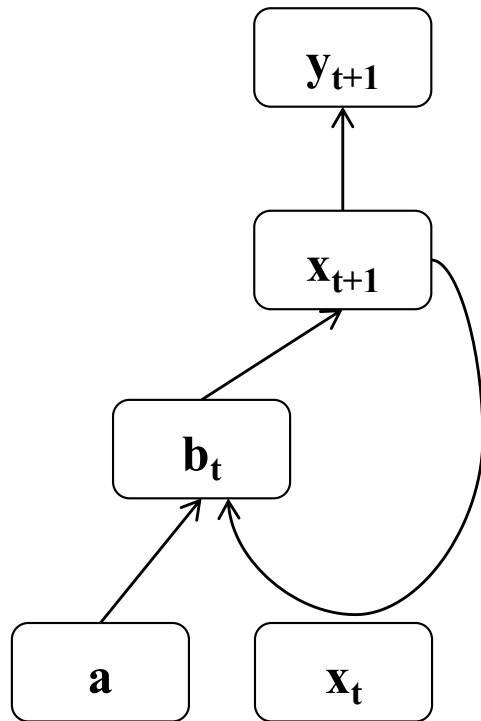
$$y_t = W_{hy}h_t + b_y$$

- By unrolling the RNN through time, we can **share parameters** and accommodate **arbitrary length** input/output pairs
- Applications: **time-series data** such as sentences, speech, stock-market, signal data, etc.



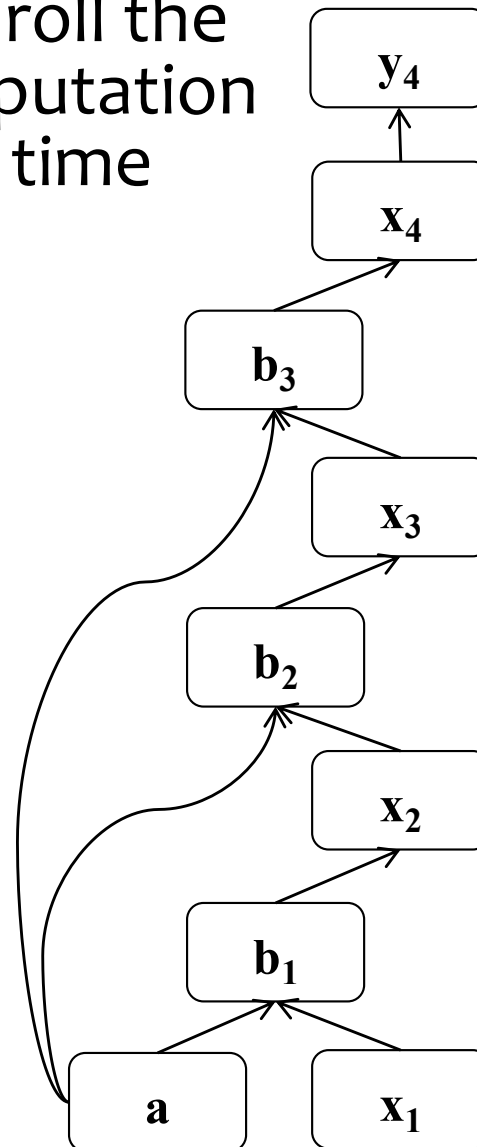
Background: Backprop through time

Recurrent neural network:



BPTT:

1. Unroll the computation over time



2. Run backprop through the resulting feed-forward network

(Robinson & Fallside, 1987)
(Werbos, 1988)
(Mozar, 1995)



Bidirectional RNN

inputs: $\mathbf{x} = (x_1, x_2, \dots, x_T), x_i \in \mathcal{R}^I$

hidden units: $\vec{\mathbf{h}}$ and $\overleftarrow{\mathbf{h}}$

outputs: $\mathbf{y} = (y_1, y_2, \dots, y_T), y_i \in \mathcal{R}^K$

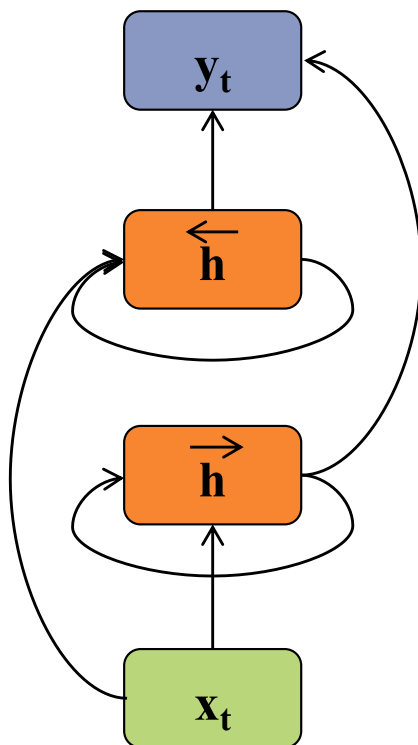
nonlinearity: \mathcal{H}

Recursive Definition:

$$\vec{h}_t = \mathcal{H} \left(W_{x\vec{h}} x_t + W_{\vec{h}\vec{h}} \vec{h}_{t-1} + b_{\vec{h}} \right)$$

$$\overleftarrow{h}_t = \mathcal{H} \left(W_{x\overleftarrow{h}} x_t + W_{\overleftarrow{h}\overleftarrow{h}} \overleftarrow{h}_{t+1} + b_{\overleftarrow{h}} \right)$$

$$y_t = W_{\vec{h}y} \vec{h}_t + W_{\overleftarrow{h}y} \overleftarrow{h}_t + b_y$$



Bidirectional RNN

inputs: $\mathbf{x} = (x_1, x_2, \dots, x_T), x_i \in \mathcal{R}^I$

hidden units: $\vec{\mathbf{h}}$ and $\overleftarrow{\mathbf{h}}$

outputs: $\mathbf{y} = (y_1, y_2, \dots, y_T), y_i \in \mathcal{R}^K$

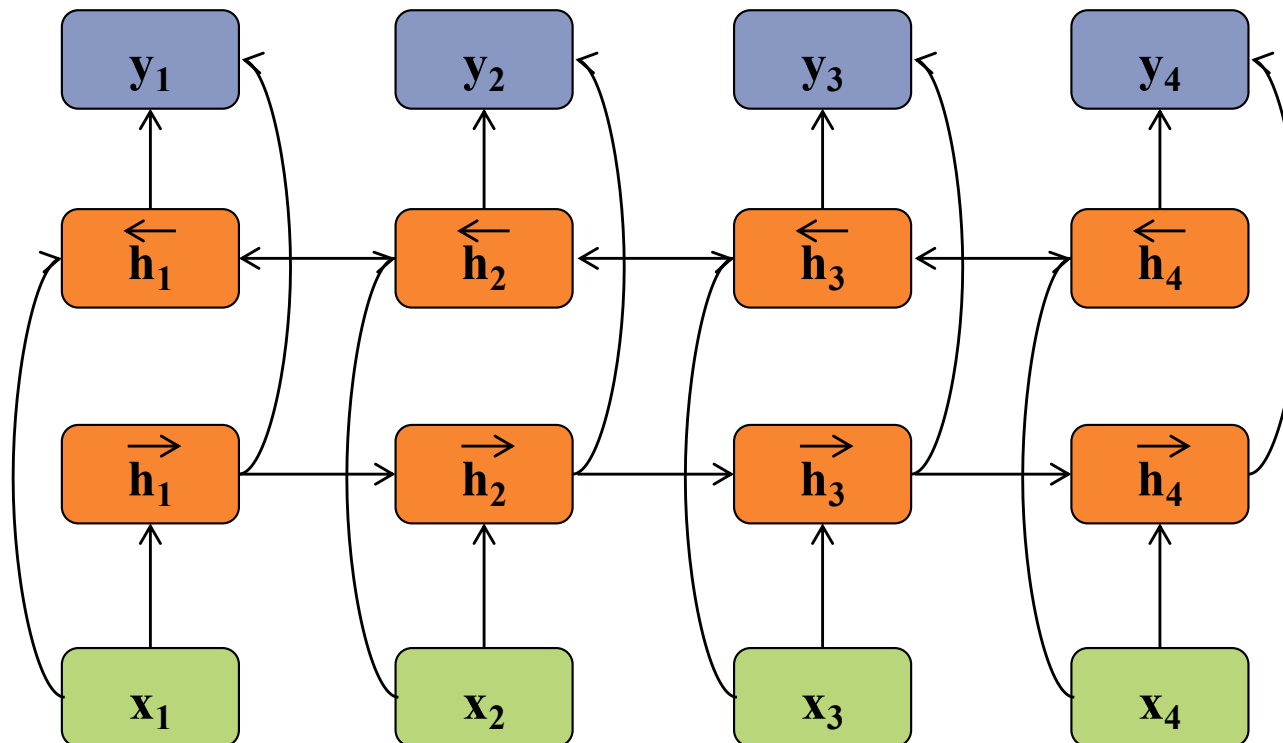
nonlinearity: \mathcal{H}

Recursive Definition:

$$\vec{h}_t = \mathcal{H} \left(W_{x\vec{h}} x_t + W_{\vec{h}\vec{h}} \vec{h}_{t-1} + b_{\vec{h}} \right)$$

$$\overleftarrow{h}_t = \mathcal{H} \left(W_{x\overleftarrow{h}} x_t + W_{\overleftarrow{h}\overleftarrow{h}} \overleftarrow{h}_{t+1} + b_{\overleftarrow{h}} \right)$$

$$y_t = W_{\vec{h}y} \vec{h}_t + W_{\overleftarrow{h}y} \overleftarrow{h}_t + b_y$$



Bidirectional RNN

inputs: $\mathbf{x} = (x_1, x_2, \dots, x_T), x_i \in \mathcal{R}^I$

hidden units: $\vec{\mathbf{h}}$ and $\overleftarrow{\mathbf{h}}$

outputs: $\mathbf{y} = (y_1, y_2, \dots, y_T), y_i \in \mathcal{R}^K$

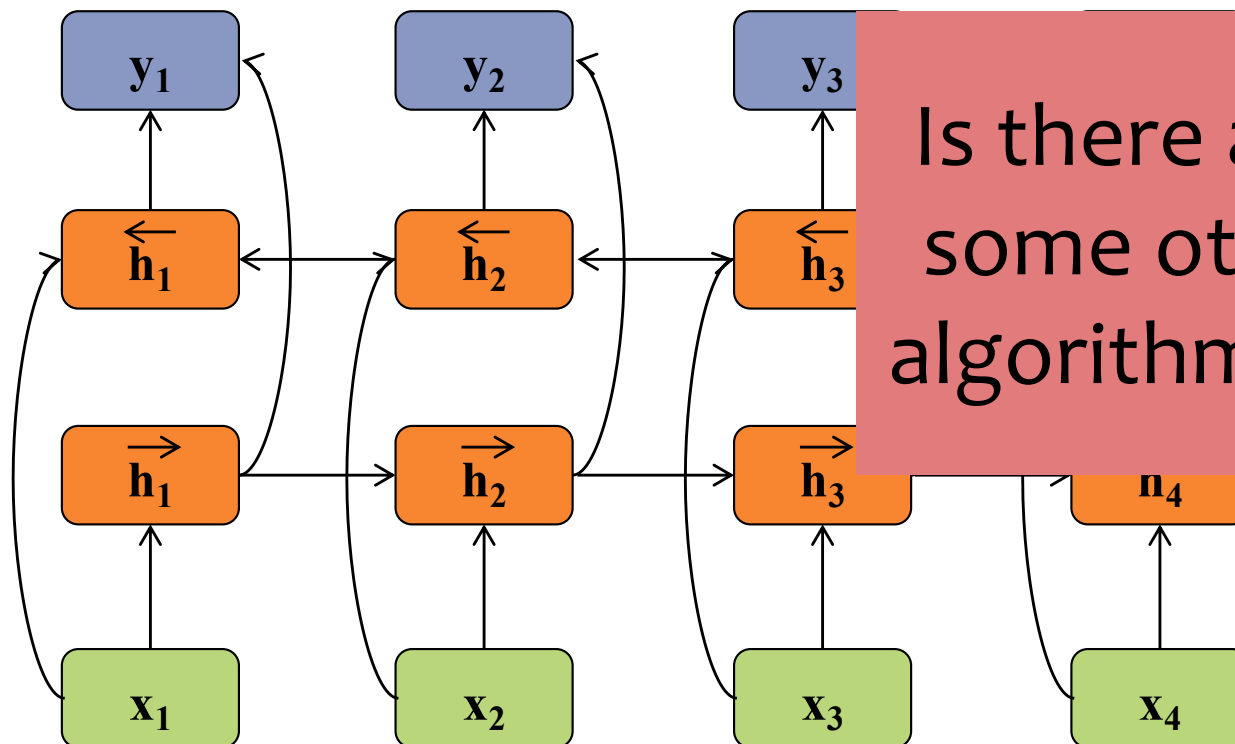
nonlinearity: \mathcal{H}

Recursive Definition:

$$\vec{h}_t = \mathcal{H} \left(W_{x\vec{h}} x_t + W_{\vec{h}\vec{h}} \vec{h}_{t-1} + b_{\vec{h}} \right)$$

$$\overleftarrow{h}_t = \mathcal{H} \left(W_{x\overleftarrow{h}} x_t + W_{\overleftarrow{h}\overleftarrow{h}} \overleftarrow{h}_{t+1} + b_{\overleftarrow{h}} \right)$$

$$y_t = W_{\vec{h}y} \vec{h}_t + W_{\overleftarrow{h}y} \overleftarrow{h}_t + b_y$$



Is there an analogy to some other recursive algorithm(s) we know?

Deep RNNs

inputs: $\mathbf{x} = (x_1, x_2, \dots, x_T), x_i \in \mathcal{R}^I$

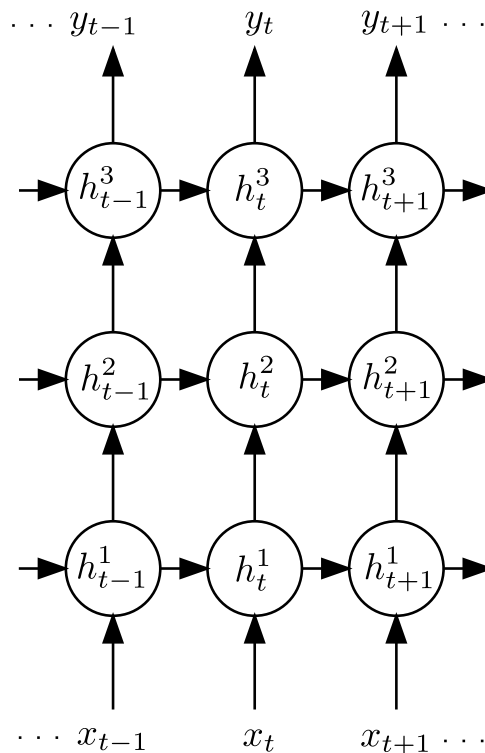
outputs: $\mathbf{y} = (y_1, y_2, \dots, y_T), y_i \in \mathcal{R}^K$

nonlinearity: \mathcal{H}

Recursive Definition:

$$h_t^n = \mathcal{H}(W_{h^{n-1}h^n}h_t^{n-1} + W_{h^n h^n}h_{t-1}^n + b_h^n)$$

$$y_t = W_{h^N y}h_t^N + b_y$$



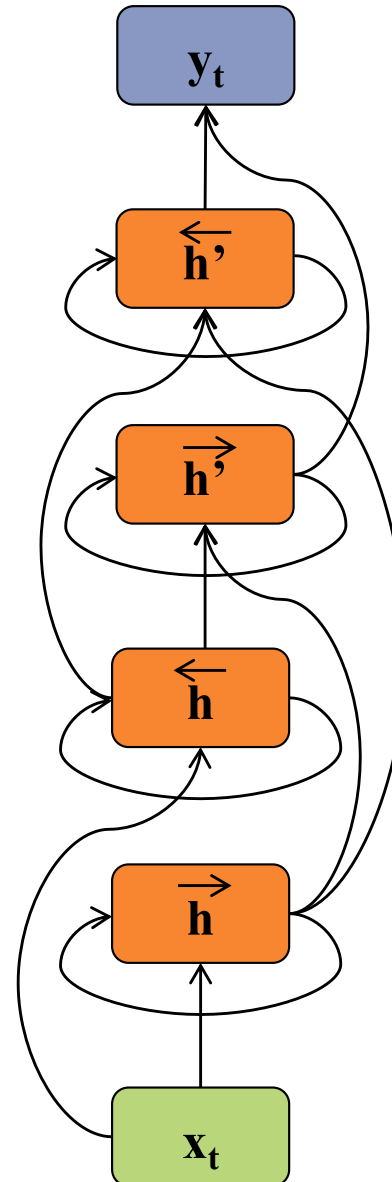
Deep Bidirectional RNNs

inputs: $\mathbf{x} = (x_1, x_2, \dots, x_T), x_i \in \mathcal{R}^I$

outputs: $\mathbf{y} = (y_1, y_2, \dots, y_T), y_i \in \mathcal{R}^K$

nonlinearity: \mathcal{H}

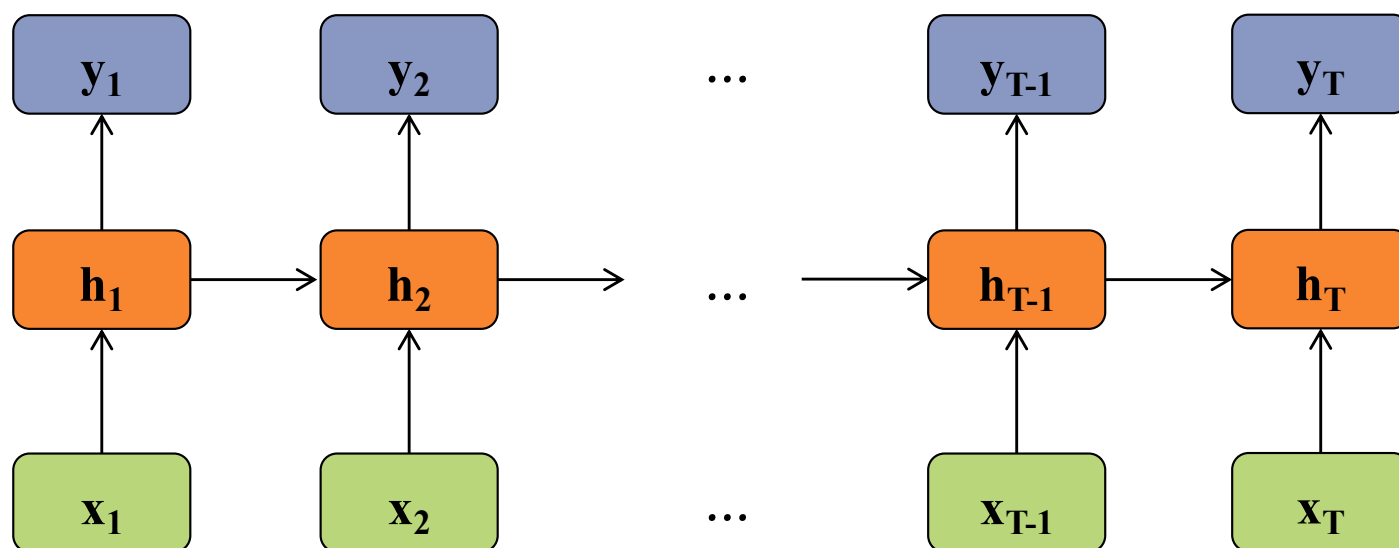
- Notice that the upper level hidden units have input from **two previous layers** (i.e. wider input)
- Likewise for the output layer
- What analogy can we draw to DNNs, DBNs, DBMs?



Long Short-Term Memory (LSTM)

Motivation:

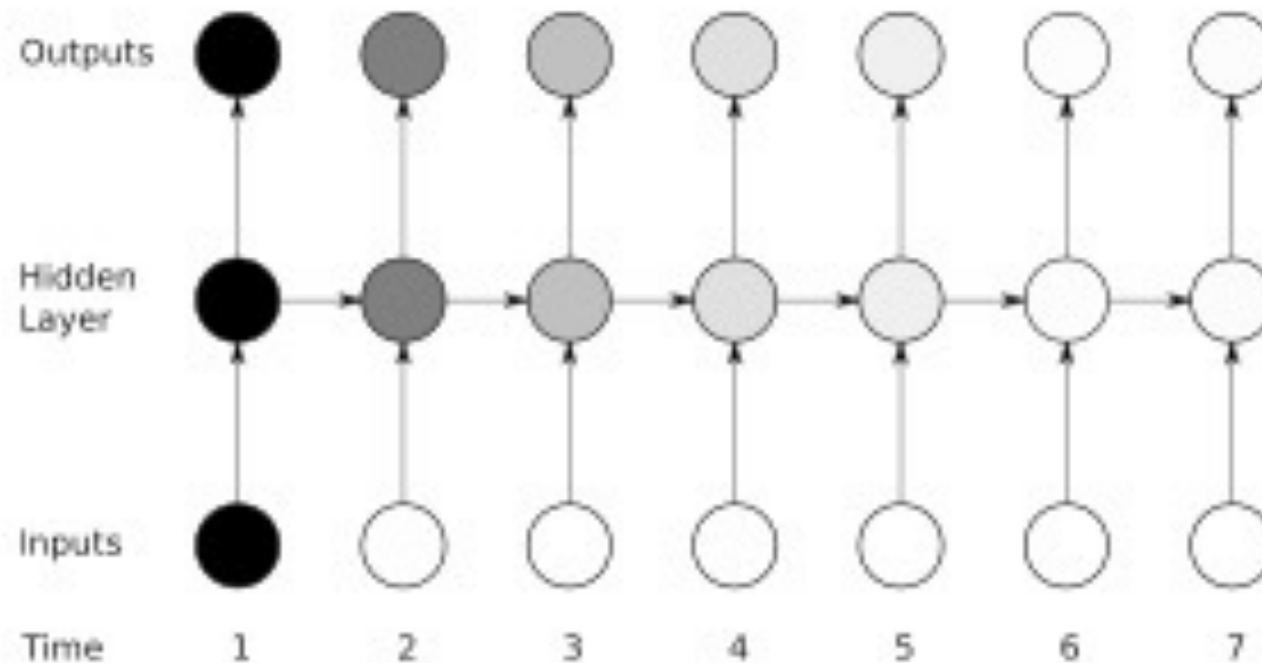
- Standard RNNs have trouble learning long distance dependencies
- LSTMs combat this issue



Long Short-Term Memory (LSTM)

Motivation:

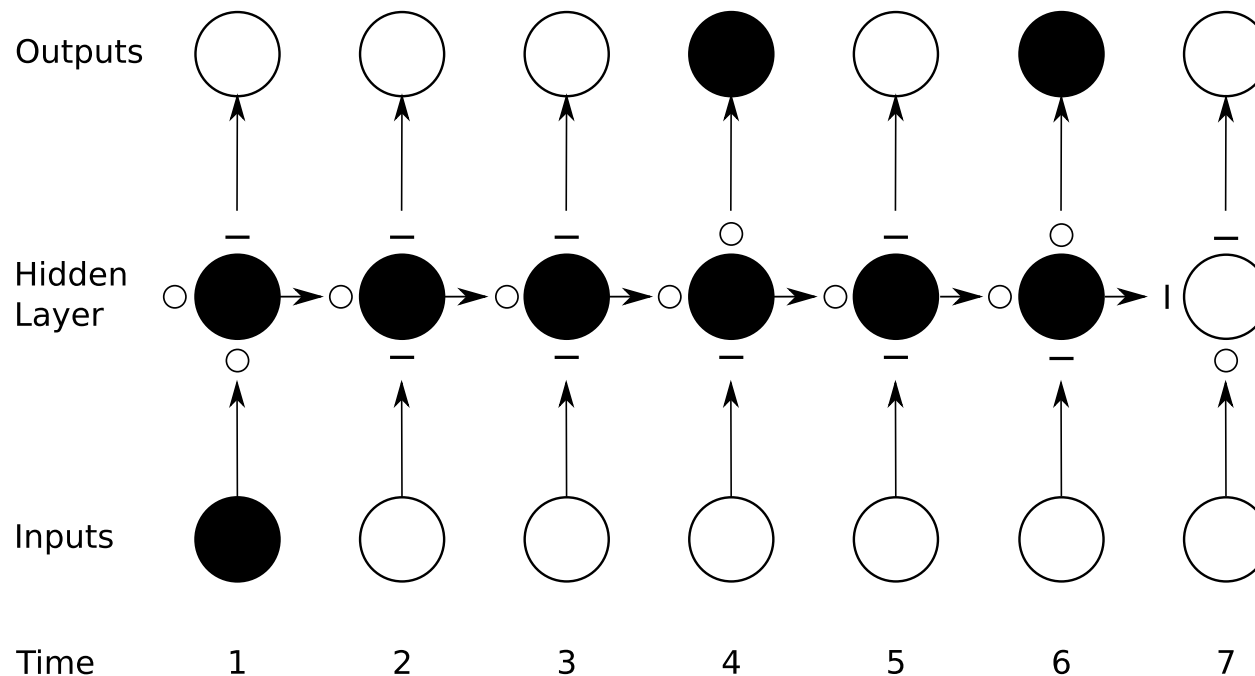
- Vanishing gradient problem for Standard RNNs
- Figure shows sensitivity (darker = more sensitive) to the input at time $t=1$



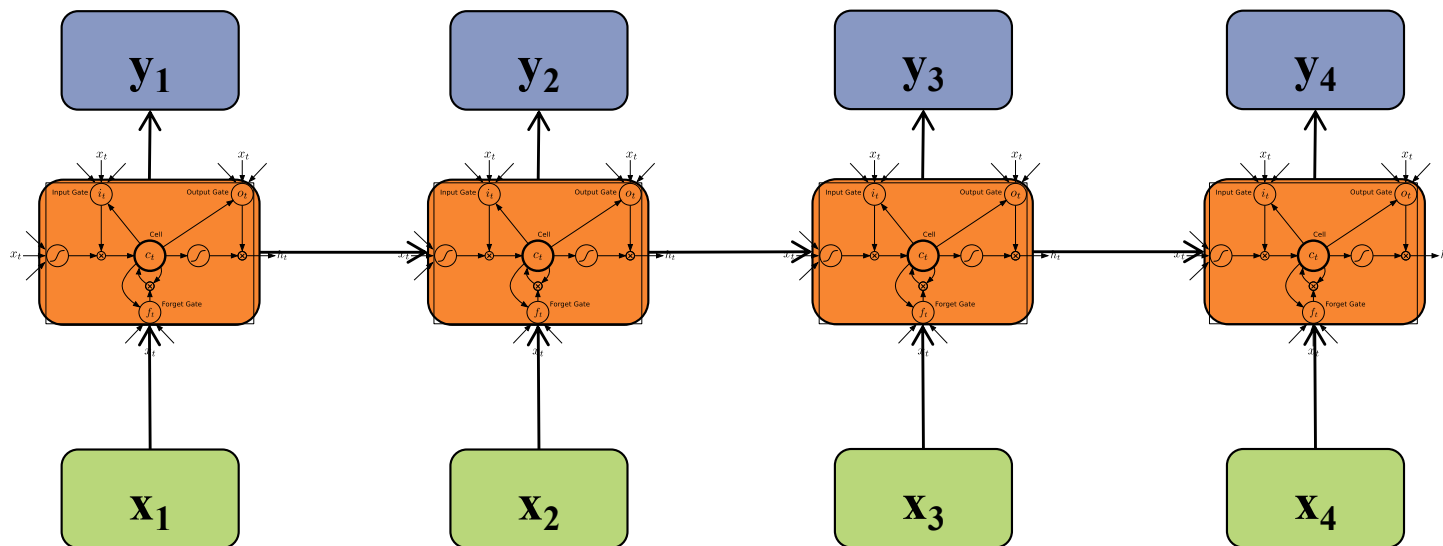
Long Short-Term Memory (LSTM)

Motivation:

- LSTM units have a rich internal structure
- The various “gates” determine the propagation of information and can choose to “remember” or “forget” information

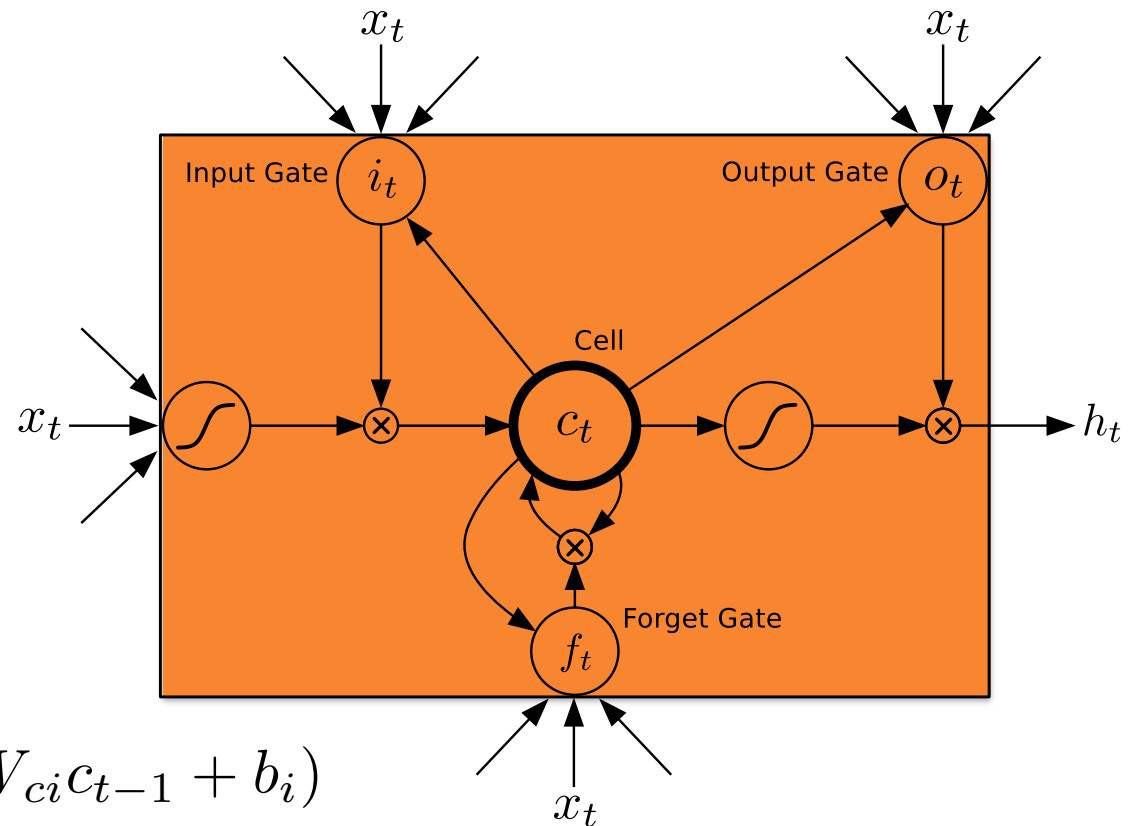


Long Short-Term Memory (LSTM)



Long Short-Term Memory (LSTM)

- **Input gate:** masks out the standard RNN inputs
- **Forget gate:** masks out the previous cell
- **Cell:** stores the input/forget mixture
- **Output gate:** masks out the values of the next hidden



$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i)$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f)$$

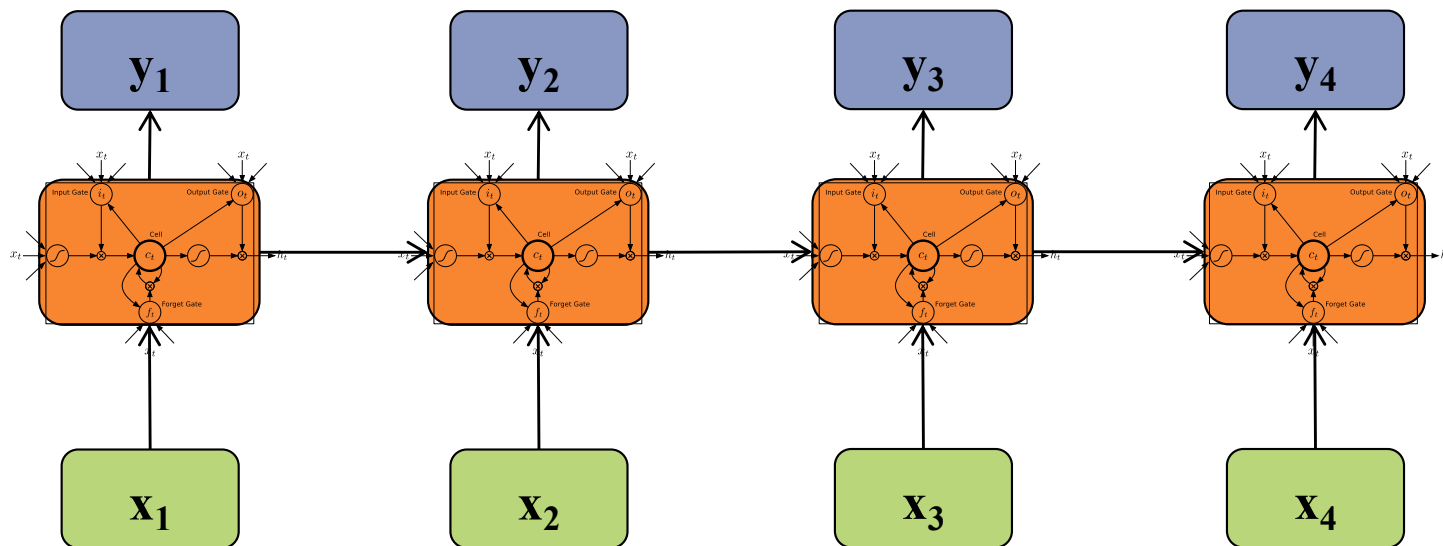
$$c_t = f_t c_{t-1} + i_t \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o)$$

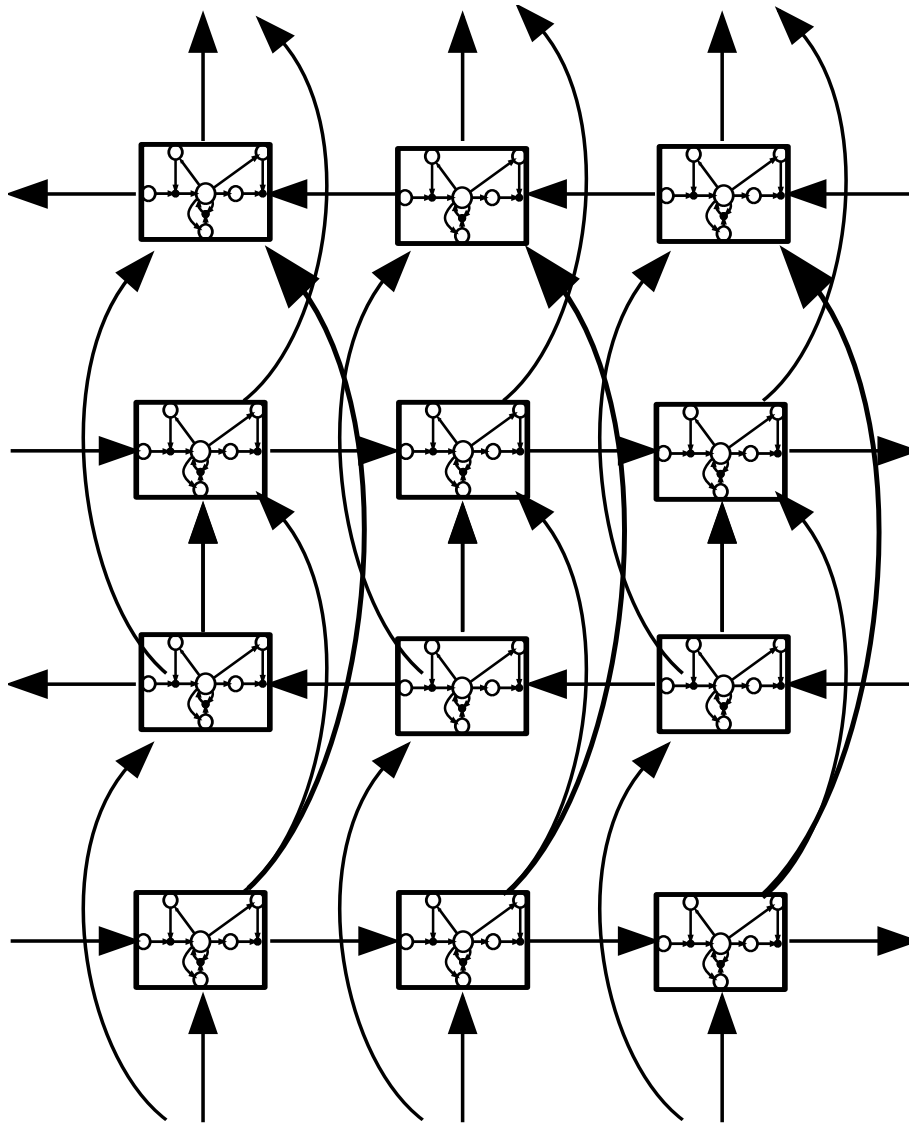
$$h_t = o_t \tanh(c_t)$$

Figure from (Graves et al., 2013)

Long Short-Term Memory (LSTM)

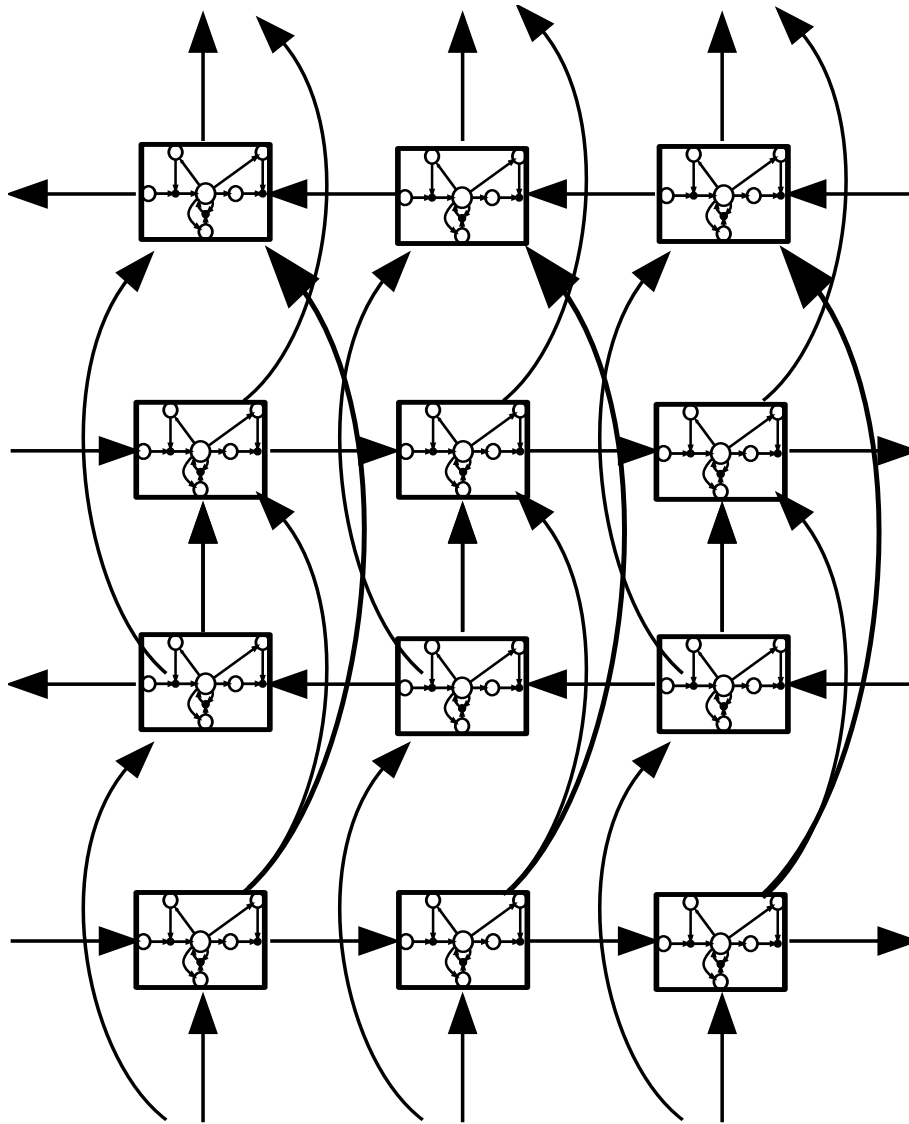


Deep Bidirectional LSTM (DBLSTM)



- Figure: input/output layers not shown
- **Same general topology** as a Deep Bidirectional RNN, but with **LSTM units** in the hidden layers
- No additional **representational power** over DBRNN, but **easier to learn** in practice

Deep Bidirectional LSTM (DBLSTM)



How important is this particular architecture?

Jozefowicz et al. (2015) **evaluated 10,000 different LSTM-like architectures** and found several variants that worked just as well on several tasks.

RNN Training Tricks

- Deep Learning models tend to consist largely of **matrix multiplications**
- Training tricks:
 - **mini-batching with masking**

	Metric	DyC++	DyPy	Chainer	DyC++ Seq	Theano	TF
RNNLM (MB=1)	words/sec	190	190	114	494	189	298
RNNLM (MB=4)	words/sec	830	825	295	1510	567	473
RNNLM (MB=16)	words/sec	1820	1880	794	2400	1100	606
RNNLM (MB=64)	words/sec	2440	2470	1340	2820	1260	636

- **sorting into buckets of similar-length sequences**, so that mini-batches have same length sentences
- **truncated BPTT**, when sequences are too long, divide sequences into chunks and use the final vector of the previous chunk as the initial vector for the next chunk (but don't backprop from next chunk to previous chunk)

RNN Summary

- **RNNs**
 - Applicable to tasks such as **sequence labeling**, speech recognition, machine translation, etc.
 - Able to **learn context features** for time series data
 - Vanishing gradients are still a problem – but **LSTM units** can help
- **Other Resources**
 - Christopher Olah's blog post on LSTMs
<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

MODULE-BASED AUTOMATIC DIFFERENTIATION

Backpropagation

Automatic Differentiation – Reverse Mode (aka. Backpropagation)

Forward Computation

1. Write an **algorithm** for evaluating the function $y = f(\mathbf{x})$. The algorithm defines a **directed acyclic graph**, where each variable is a node (i.e. the “**computation graph**”)
2. Visit each node in **topological order**.
For variable u_i with inputs v_1, \dots, v_N
 - a. Compute $u_i = g_i(v_1, \dots, v_N)$
 - b. Store the result at the node

Backward Computation (Version A)

1. **Initialize** $dy/dy = 1$.
2. Visit each node v_j in **reverse topological order**.
Let u_1, \dots, u_M denote all the nodes with v_j as an input
Assuming that $y = h(\mathbf{u}) = h(u_1, \dots, u_M)$
and $\mathbf{u} = g(\mathbf{v})$ or equivalently $u_i = g_i(v_1, \dots, v_j, \dots, v_N)$ for all i
 - a. We already know dy/du_i for all i
 - b. Compute dy/dv_j as below (Choice of algorithm ensures computing (du_i/dv_j) is easy)

$$\frac{dy}{dv_j} = \sum_{i=1}^M \frac{dy}{du_i} \frac{du_i}{dv_j}$$

Return partial derivatives dy/du_i for all variables

Backpropagation

Automatic Differentiation – Reverse Mode (aka. Backpropagation)

Forward Computation

1. Write an **algorithm** for evaluating the function $y = f(\mathbf{x})$. The algorithm defines a **directed acyclic graph**, where each variable is a node (i.e. the “**computation graph**”)
2. Visit each node in **topological order**.
For variable u_i with inputs v_1, \dots, v_N
 - a. Compute $u_i = g_i(v_1, \dots, v_N)$
 - b. Store the result at the node

Backward Computation (Version B)

1. **Initialize** all partial derivatives dy/du_j to 0 and $dy/dy = 1$.
2. Visit each node in **reverse topological order**.
For variable $u_i = g_i(v_1, \dots, v_N)$
 - a. We already know dy/du_i
 - b. Increment dy/dv_j by $(dy/du_i)(du_i/dv_j)$
(Choice of algorithm ensures computing (du_i/dv_j) is easy)

Return partial derivatives dy/du_i for all variables

Training Backpropagation

Why is the backpropagation algorithm efficient?

1. Reuses **computation from the forward pass** in the backward pass
2. Reuses **partial derivatives** throughout the backward pass (*but only if the algorithm reuses shared computation in the forward pass*)

(Key idea: partial derivatives in the backward pass should be thought of as variables stored for reuse)

A Recipe for Background Machine Gradients

1. Given training data

$$\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$$

2. Choose each of the

– Decision function

$$\hat{\mathbf{y}} = f_{\boldsymbol{\theta}}(\mathbf{x}_i)$$


– Loss function

$$\ell(\hat{\mathbf{y}}, \mathbf{y}_i) \in \mathbb{R}$$

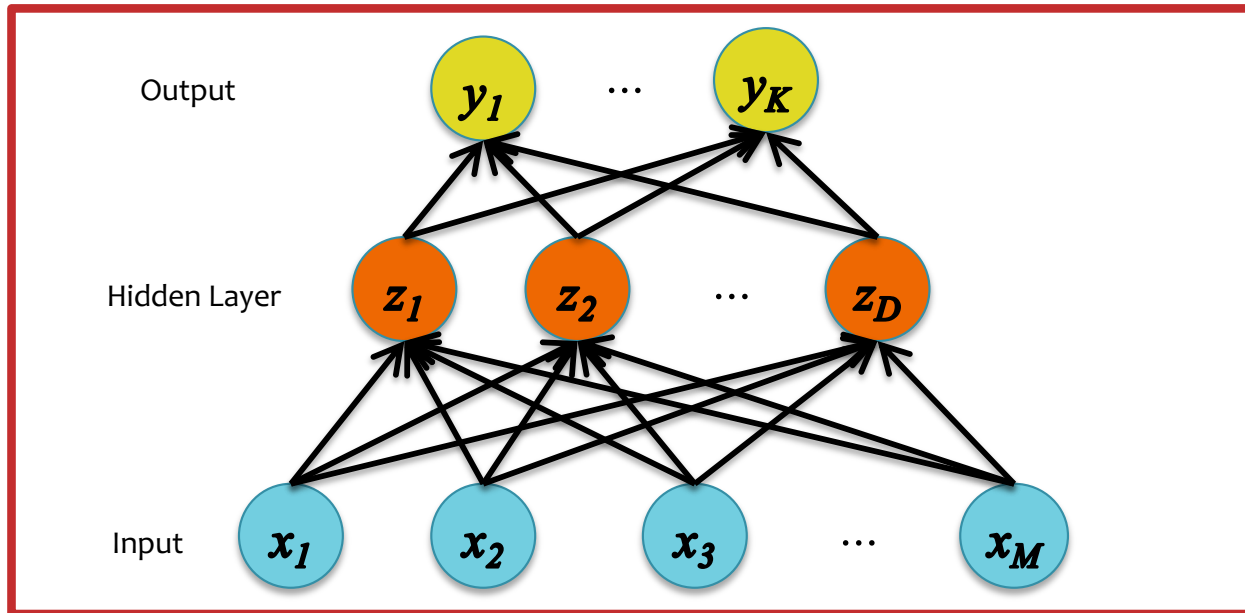
Backpropagation can compute this gradient!

And it's a **special case of a more general algorithm** called reverse-mode automatic differentiation that can compute the gradient of any differentiable function efficiently!

(opposite the gradient)


$$\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{y}_i)$$

Backpropagation: Abstract Picture



Forward

5. $J = -\mathbf{y}^T \log \hat{\mathbf{y}}$
4. $\hat{\mathbf{y}} = \text{softmax}(\mathbf{b})$
3. $\mathbf{b} = \beta \mathbf{z}$
2. $\mathbf{z} = \sigma(\mathbf{a})$
1. $\mathbf{a} = \alpha \mathbf{x}$

Backward

6. $\mathbf{g}_{\hat{\mathbf{y}}} = -\mathbf{y} \div \hat{\mathbf{y}}$
7. $\mathbf{g}_{\mathbf{b}} = \mathbf{g}_{\hat{\mathbf{y}}}^T (\text{diag}(\hat{\mathbf{y}}) - \hat{\mathbf{y}} \hat{\mathbf{y}}^T)$
8. $\mathbf{g}_{\beta} = \mathbf{g}_{\mathbf{b}}^T \mathbf{z}^T$
 $\mathbf{g}_{\mathbf{z}} = \beta^T \mathbf{g}_{\mathbf{b}}$
10. $\mathbf{g}_{\mathbf{a}} = \mathbf{g}_{\mathbf{z}} \odot \mathbf{z} \odot (1 - \mathbf{z})$
11. $\mathbf{g}_{\alpha} = \mathbf{g}_{\mathbf{a}} \mathbf{x}^T$

(F) Loss

$$J = \sum_{k=1}^K y_k^* \log(y_k)$$

(E) Output (softmax)

$$y_k = \frac{\exp(b_k)}{\sum_{l=1}^K \exp(b_l)}$$

(D) Output (linear)

$$b_k = \sum_{j=0}^D \beta_{kj} z_j \quad \forall k$$

(C) Hidden (nonlinear)

$$z_j = \sigma(a_j), \quad \forall j$$

(B) Hidden (linear)

$$a_j = \sum_{i=0}^M \alpha_{ji} x_i, \quad \forall j$$

(A) Input

Given $x_i, \quad \forall i$

Backpropagation: Procedural Method

Algorithm 1 Forward Computation

```
1: procedure NNFORWARD(Training example  $(\mathbf{x}, \mathbf{y})$ , Params  $\alpha, \beta$ )
2:    $\mathbf{a} = \alpha \mathbf{x}$ 
3:    $\mathbf{z} = \sigma(\mathbf{a})$ 
4:    $\mathbf{b} = \beta \mathbf{z}$ 
5:    $\hat{\mathbf{y}} = \text{softmax}(\mathbf{b})$ 
6:    $J = -\mathbf{y}^T \log \hat{\mathbf{y}}$ 
7:    $\mathbf{o} = \text{object}(\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J)$ 
8:   return intermediate quantities  $\mathbf{o}$ 
```

Algorithm 2 Backpropagation

```
1: procedure NNBACKWARD(Training example  $(\mathbf{x}, \mathbf{y})$ , Params  $\alpha, \beta$ ,  
   Intermediates  $\mathbf{o}$ )
2:   Place intermediate quantities  $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$  in  $\mathbf{o}$  in scope
3:    $\mathbf{g}_{\hat{\mathbf{y}}} = -\mathbf{y} \div \hat{\mathbf{y}}$ 
4:    $\mathbf{g}_{\mathbf{b}} = \mathbf{g}_{\hat{\mathbf{y}}}^T (\text{diag}(\hat{\mathbf{y}}) - \hat{\mathbf{y}}\hat{\mathbf{y}}^T)$ 
5:    $\mathbf{g}_{\beta} = \mathbf{g}_{\mathbf{b}}^T \mathbf{z}^T$ 
6:    $\mathbf{g}_{\mathbf{z}} = \beta^T \mathbf{g}_{\mathbf{b}}^T$ 
7:    $\mathbf{g}_{\mathbf{a}} = \mathbf{g}_{\mathbf{z}} \odot \mathbf{z} \odot (1 - \mathbf{z})$ 
8:    $\mathbf{g}_{\alpha} = \mathbf{g}_{\mathbf{a}} \mathbf{x}^T$ 
9:   return parameter gradients  $\mathbf{g}_{\alpha}, \mathbf{g}_{\beta}$ 
```

Drawbacks of Procedural Method

1. Hard to reuse / adapt for other models
2. (Possibly) harder to make individual steps more efficient
3. Hard to find source of error if finite-difference check reports an error (since it tells you only that there is an error somewhere in those 17 lines of code)

Module-based AutoDiff

Module-based automatic differentiation (AD / Autodiff) is a technique that has long been used to develop libraries for deep learning

- **Dynamic neural network packages** allow a specification of the computation graph dynamically at runtime
 - PyTorch <http://pytorch.org>
 - Torch <http://torch.ch>
 - DyNet <https://dynet.readthedocs.io>
- **Static neural network packages** require a static specification of a computation graph which is subsequently compiled into code
 - TensorFlow <https://www.tensorflow.org>
 - Aesara (and Theano) <https://aesara.readthedocs.io>
 - *(These libraries are also module-based, but herein by “module-based AD” we mean the dynamic approach)*

Module-based AutoDiff

- **Key Idea:**

- componentize the computation of the neural-network into layers
- each layer consolidates multiple **real-valued nodes** in the computation graph (a subset of them) into one **vector-valued node** (aka. a **module**)

- Each **module** is capable of two actions:

1. Forward computation of output $\mathbf{b} = [b_1, \dots, b_B]$ given input $\mathbf{a} = [a_1, \dots, a_A]$ via some differentiable function f . That is $\mathbf{b} = f(\mathbf{a})$.
2. Backward computation of the gradient of the input $\mathbf{g}_\mathbf{a} = \nabla_\mathbf{a} J = [\frac{dJ}{da_1}, \dots, \frac{dJ}{da_A}]$ given the gradient of output $\mathbf{g}_\mathbf{b} = \nabla_\mathbf{b} J = [\frac{dJ}{db_1}, \dots, \frac{dJ}{db_B}]$, where J is the final real-valued output of the entire computation graph. This is done via the chain rule $\frac{dJ}{da_i} = \sum_{j=1}^J \frac{dJ}{db_j} \frac{db_j}{da_i}$ for all $i \in \{1, \dots, A\}$.

Module-based AutoDiff

Dimensions: input $\mathbf{a} \in \mathbb{R}^A$, output $\mathbf{b} \in \mathbb{R}^B$, gradient of output $\mathbf{g}_a \triangleq \nabla_{\mathbf{a}} J \in \mathbb{R}^A$, and gradient of input $\mathbf{g}_b \triangleq \nabla_{\mathbf{b}} J \in \mathbb{R}^B$.

Sigmoid Module The sigmoid layer has only one input vector \mathbf{a} . Below σ is the sigmoid applied element-wise, and \odot is element-wise multiplication s.t. $\mathbf{u} \odot \mathbf{v} = [u_1 v_1, \dots, u_M v_M]$.

```
1: procedure SIGMOIDFORWARD( $\mathbf{a}$ )
2:    $\mathbf{b} = \sigma(\mathbf{a})$ 
3:   return  $\mathbf{b}$ 
4: procedure SIGMOIDBACKWARD( $\mathbf{a}, \mathbf{b}, \mathbf{g}_b$ )
5:    $\mathbf{g}_a = \mathbf{g}_b \odot \mathbf{b} \odot (1 - \mathbf{b})$ 
6:   return  $\mathbf{g}_a$ 
```

Softmax Module The softmax layer has only one input vector \mathbf{a} . For any vector $\mathbf{v} \in \mathbb{R}^D$, we have that $\text{diag}(\mathbf{v})$ returns a $D \times D$ diagonal matrix whose diagonal entries are v_1, v_2, \dots, v_D and whose non-diagonal entries are zero.

```
1: procedure SOFTMAXFORWARD( $\mathbf{a}$ )
2:    $\mathbf{b} = \text{softmax}(\mathbf{a})$ 
3:   return  $\mathbf{b}$ 
4: procedure SOFTMAXBACKWARD( $\mathbf{a}, \mathbf{b}, \mathbf{g}_b$ )
5:    $\mathbf{g}_a = \mathbf{g}_b^T (\text{diag}(\mathbf{b}) - \mathbf{b}\mathbf{b}^T)$ 
6:   return  $\mathbf{g}_a$ 
```

Linear Module The linear layer has two inputs: a vector \mathbf{a} and parameters $\omega \in \mathbb{R}^{B \times A}$. The output \mathbf{b} is not used by LINEARBACKWARD, but we pass it in for consistency of form.

```
1: procedure LINEARFORWARD( $\mathbf{a}, \omega$ )
2:    $\mathbf{b} = \omega \mathbf{a}$ 
3:   return  $\mathbf{b}$ 
4: procedure LINEARBACKWARD( $\mathbf{a}, \omega, \mathbf{b}, \mathbf{g}_b$ )
5:    $\mathbf{g}_\omega = \mathbf{g}_b \mathbf{a}^T$ 
6:    $\mathbf{g}_a = \omega^T \mathbf{g}_b$ 
7:   return  $\mathbf{g}_\omega, \mathbf{g}_a$ 
```

Cross-Entropy Module The cross-entropy layer has two inputs: a gold one-hot vector \mathbf{a} and a predicted probability distribution $\hat{\mathbf{a}}$. Its output $b \in \mathbb{R}$ is a scalar. Below \div is element-wise division. The output b is not used by CROSSENTROPYBACKWARD, but we pass it in for consistency of form.

```
1: procedure CROSSENTROPYFORWARD( $\mathbf{a}, \hat{\mathbf{a}}$ )
2:    $b = -\mathbf{a}^T \log \hat{\mathbf{a}}$ 
3:   return  $b$ 
4: procedure CROSSENTROPYBACKWARD( $\mathbf{a}, \hat{\mathbf{a}}, b, \mathbf{g}_b$ )
5:    $\mathbf{g}_{\hat{\mathbf{a}}} = -\mathbf{g}_b (\mathbf{a} \div \hat{\mathbf{a}})$ 
6:   return  $\mathbf{g}_{\hat{\mathbf{a}}}$ 
```

Module-based AutoDiff

Algorithm 1 Forward Computation

```
1: procedure NNFORWARD(Training example  $(\mathbf{x}, \mathbf{y})$ , Parameters  $\alpha, \beta$ )
2:    $\mathbf{a} = \text{LINEARFORWARD}(\mathbf{x}, \alpha)$ 
3:    $\mathbf{z} = \text{SIGMOIDFORWARD}(\mathbf{a})$ 
4:    $\mathbf{b} = \text{LINEARFORWARD}(\mathbf{z}, \beta)$ 
5:    $\hat{\mathbf{y}} = \text{SOFTMAXFORWARD}(\mathbf{b})$ 
6:    $J = \text{CROSSENTROPYFORWARD}(\mathbf{y}, \hat{\mathbf{y}})$ 
7:    $\mathbf{o} = \text{object}(\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J)$ 
8:   return intermediate quantities  $\mathbf{o}$ 
```

Algorithm 2 Backpropagation

```
1: procedure NNBACKWARD(Training example  $(\mathbf{x}, \mathbf{y})$ , Parameters  $\alpha, \beta$ , Intermediates  $\mathbf{o}$ )
2:   Place intermediate quantities  $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$  in  $\mathbf{o}$  in scope
3:    $g_J = \frac{dJ}{dJ} = 1$  ▷ Base case
4:    $\mathbf{g}_{\hat{\mathbf{y}}} = \text{CROSSENTROPYBACKWARD}(\mathbf{y}, \hat{\mathbf{y}}, J, g_J)$ 
5:    $\mathbf{g}_{\mathbf{b}} = \text{SOFTMAXBACKWARD}(\mathbf{b}, \hat{\mathbf{y}}, \mathbf{g}_{\hat{\mathbf{y}}})$ 
6:    $\mathbf{g}_{\beta}, \mathbf{g}_{\mathbf{z}} = \text{LINEARBACKWARD}(\mathbf{z}, \mathbf{b}, \mathbf{g}_{\mathbf{b}})$ 
7:    $\mathbf{g}_{\mathbf{a}} = \text{SIGMOIDBACKWARD}(\mathbf{a}, \mathbf{z}, \mathbf{g}_{\mathbf{z}})$ 
8:    $\mathbf{g}_{\alpha}, \mathbf{g}_{\mathbf{x}} = \text{LINEARBACKWARD}(\mathbf{x}, \mathbf{a}, \mathbf{g}_{\mathbf{a}})$  ▷ We discard  $\mathbf{g}_{\mathbf{x}}$ 
9:   return parameter gradients  $\mathbf{g}_{\alpha}, \mathbf{g}_{\beta}$ 
```

Advantages of Module-based AutoDiff

1. Easy to reuse / adapt for other models
2. Encapsulated layers are easier to optimize (e.g. implement in C++ or CUDA)
3. Easier to find bugs because we can run a finite-difference check on each layer separately

Module-based AutoDiff (OOP Version)

Object-Oriented Implementation:

- Let each module be an **object**
- Then allow the **control flow** dictate the creation of the **computation graph**
- No longer need to implement `NNBackward(·)`, just follow the computation graph in **reverse topological order**

```
1 class Sigmoid(Module)
2     method forward(a)
3          $b = \sigma(a)$ 
4         return b
5     method backward(a, b, gb)
6          $g_a = g_b \odot b \odot (1 - b)$ 
7         return ga
```

```
1 class Softmax(Module)
2     method forward(a)
3          $b = \text{softmax}(a)$ 
4         return b
5     method backward(a, b, gb)
6          $g_a = g_b^T (\text{diag}(b) - bb^T)$ 
7         return ga
```

```
1 class Linear(Module)
2     method forward(a,  $\omega$ )
3          $b = \omega a$ 
4         return b
5     method backward(a,  $\omega$ , b, gb)
6          $g_\omega = g_b a^T$ 
7          $g_a = \omega^T g_b$ 
8         return g $_\omega$ , ga
```

```
1 class CrossEntropy(Module)
2     method forward(a,  $\hat{a}$ )
3          $b = -a^T \log \hat{a}$ 
4         return b
5     method backward(a,  $\hat{a}$ , b, gb)
6          $g_{\hat{a}} = -g_b (a \div \hat{a})$ 
7         return ga
```

Module-based AutoDiff (OOP Version)

```
1  class NeuralNetwork(Module):
2
3      method init()
4          lin1_layer = Linear()
5          sig_layer = Sigmoid()
6          lin2_layer = Linear()
7          soft_layer = Softmax()
8          ce_layer = CrossEntropy()
9
10     method forward(Tensor  $\mathbf{x}$  , Tensor  $\mathbf{y}$  , Tensor  $\boldsymbol{\alpha}$  , Tensor  $\boldsymbol{\beta}$ )
11          $\mathbf{a}$  = lin1_layer.apply_fwd( $\mathbf{x}$ ,  $\boldsymbol{\alpha}$ )
12          $\mathbf{z}$  = sig_layer.apply_fwd( $\mathbf{a}$ )
13          $\mathbf{b}$  = lin2_layer.apply_fwd( $\mathbf{z}$ ,  $\boldsymbol{\beta}$ )
14          $\hat{\mathbf{y}}$  = soft_layer.apply_fwd( $\mathbf{b}$ )
15          $J$  = ce_layer.apply_fwd( $\mathbf{y}$ ,  $\hat{\mathbf{y}}$ )
16         return  $J.out_{tensor}$ 
17
18     method backward(Tensor  $\mathbf{x}$  , Tensor  $\mathbf{y}$  , Tensor  $\boldsymbol{\alpha}$  , Tensor  $\boldsymbol{\beta}$ )
19         tape_bwd()
20         return lin1_layer.in_gradients[1] , lin2_layer.in_gradients[1]
```

Module-based AutoDiff (OOP Version)

```
1  global tape = stack()
2
3  class Module:
4
5      method init()
6          out_tensor = null
7          out_gradient = 1
8
9      method apply_fwd(List in_modules)
10         in_tensors = [x.out_tensor for x in in_modules]
11         out_tensor = forward(in_tensors)
12         tape.push(self)
13         return self
14
15     method apply_bwd():
16         in_gradients = backward(in_tensors, out_tensor, out_gradient)
17         for i in 1,..., len(in_modules):
18             in_modules[i].out_gradient += in_gradients[i]
19         return self
20
21 function tape_bwd():
22     while len(tape) > 0
23         m = tape.pop()
24         m.apply_bwd()
```

PyTorch

The same simple neural network we defined in pseudocode can also be defined in PyTorch.

```
1 # Define model
2 class NeuralNetwork(nn.Module):
3     def __init__(self):
4         super(NeuralNetwork, self).__init__()
5         self.flatten = nn.Flatten()
6         self.linear1 = nn.Linear(28*28, 512)
7         self.sigmoid = nn.Sigmoid()
8         self.linear2 = nn.Linear(512, 512)
9
10    def forward(self, x):
11        x = self.flatten(x)
12        a = self.linear1(x)
13        z = self.sigmoid(a)
14        b = self.linear2(z)
15        return b
16
17 # Take one step of SGD
18 def one_step_of_sgd(X, y):
19     loss_fn = nn.CrossEntropyLoss()
20     optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
21
22     # Compute prediction error
23     pred = model(X)
24     loss = loss_fn(pred, y)
25
26     # Backpropagation
27     optimizer.zero_grad()
28     loss.backward()
29     optimizer.step()
```

PyTorch

Q: Why don't we call `linear.forward()` in PyTorch?

A: This is just syntactic sugar. There's a special method in Python `__call__` that allows you to define what happens when you treat an object as if it were a function.

In other words, running the following:

```
linear(x)
```

is equivalent to running:

```
linear.__call__(x)
```

which in PyTorch is (nearly) the same as running:

```
linear.forward(x)
```

This is because PyTorch defines every Module's `__call__` method to be something like this:

```
def __call__(self):  
    self.forward()
```

PyTorch

Q: Why don't we pass in the parameters to a PyTorch Module?

A: This just makes your code cleaner.

In PyTorch, you store the parameters inside the Module and “mark” them as parameters that should contribute to the eventual gradient used by an optimizer

Q&A