NEURAL NETWORKS

Lecture 14

April 13, 2021

ANNOUNCEMENTS

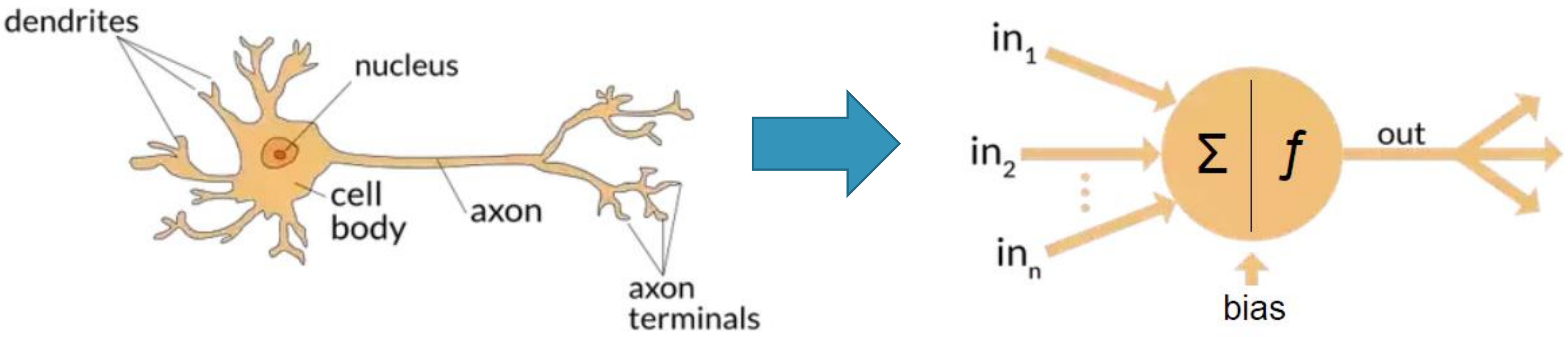
- Last Lecture Topics
  - Reinforcement Learning
  - Nonlinear dimension reduction (tSNE)
  - NLP (deep learning methods (RNN, LSTM))
- Last Class: April 20
  - Exam 3 Review? Cancel?
- HW 10: Group
- Code 6 is up; Code 7 & 8 this week
- Grade Questions due by Friday April 16

TODAY'S LECTURE

- NN applied to classification
  - Language Model: assigning probabilities to word sequences & predicting upcoming words
- Feedforward network: computation proceeds iteratively from one layer of units to next
- Deep learning: modern networks have many layers (~ deep)

HISTORICALLY SPEAKING

- Fundamental tool for language processing
- Derived from McCulloch-Pitts neuron (1943)
  - Human neuron → propositional logic computing unit
- Modern NN
  - Network of small computing units
  - Input: vector of values → output: one single value



NN VS. LR

- Share similar mathematics with logistic regression
  - More powerful as classifiers
  - Basic NN can learn almost any function
- Different classification approaches
  - LR classifier used on many tasks by developing features based on domain knowledge
  - NN usually take raw words as input & learn features as part of classification process
    - Deep NNs are great for large scale problems with enough training data to automatically learn features

NN UNITS

- NN building block = single computational unit
- Real valued numbers → do some computations → output
- Computation: weighted sum (z) of input + bias

$\mathbf{x_1..x_n}$
$\mathbf{w_1..w_n}$

$$z = b + \sum_i w_i x_i$$

NN UNITS

- NN building block = single computational unit
- Real valued numbers → do some computations → output
- Computation: weighted sum (z) of input + bias

$$\boxed{\begin{matrix} \mathbf{x_1..x_n} \\ \mathbf{w_1..w_n} \end{matrix}} \Rightarrow z = b + \sum_i w_i x_i \Rightarrow z = w \cdot x + b$$

- Usually expressed as vector notation

ACTIVATION

- Instead of using *z* (linear function of x) NN units apply non-linear function *f* to *z*
- Output of this function == activation value for NN unit *a*
  - If we just model 1 single unit, then the activation for that node/unit *is* the final output of NN: $y = a = f(z)$

ACTIVATION

- Instead of using *z* (linear function of x) NN units apply non-linear function *f* to *z*
- Output of this function == activation value for NN unit *a*
  - If we just model 1 single unit, then the activation for that node/unit *is* the final output of NN: $y = a = f(z)$
- 3 popular non-linear (activation) functions *f( )*
  - Sigmoid (again!)
  - tanh
  - Rectified linear (ReLU)

SIGMOID

- Maps output into the range [0, 1]
  - Good for handling outliers
- Differentiable
  - Good for learning

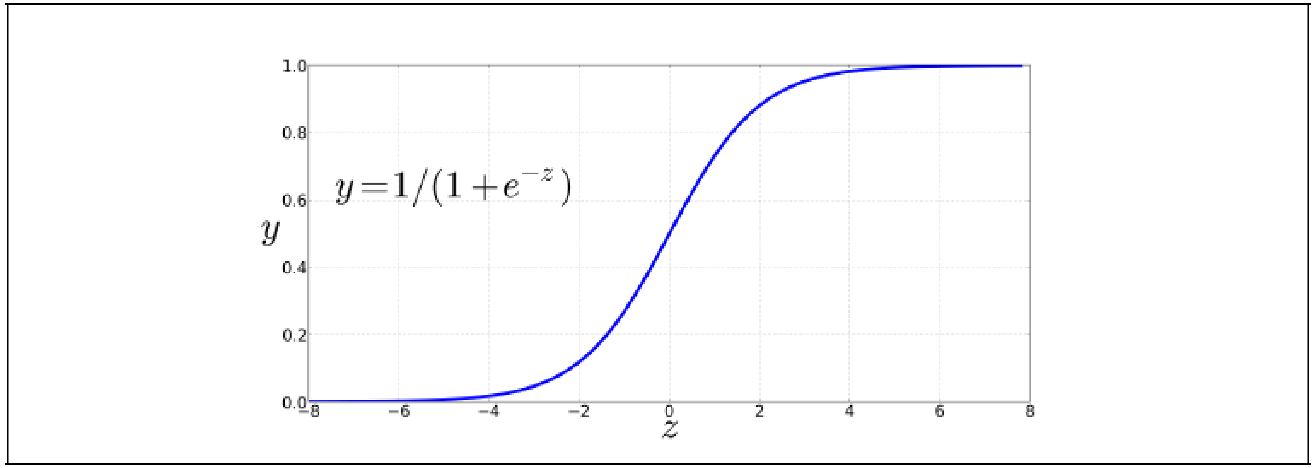


Figure 7.1 The sigmoid function takes a real value and maps it to the range $[0, 1]$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

SIGMOID

- Maps output into the range [0, 1]
  - Good for handling outliers
- Differentiable
  - Good for learning

Output of neural unit

$$y = \sigma(w \cdot x + b) = \frac{1}{1 + \exp(-(w \cdot x + b))}$$

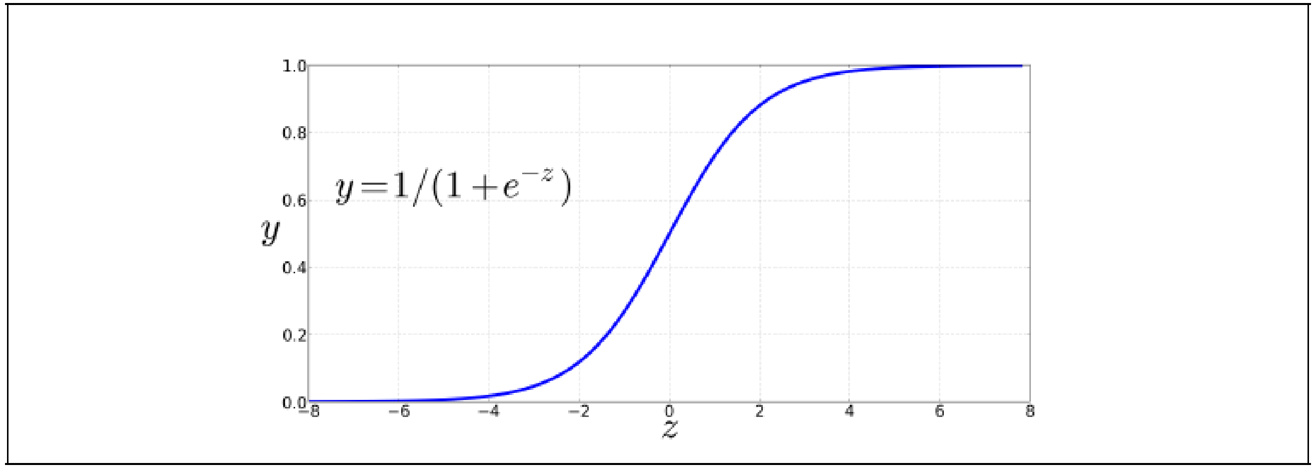
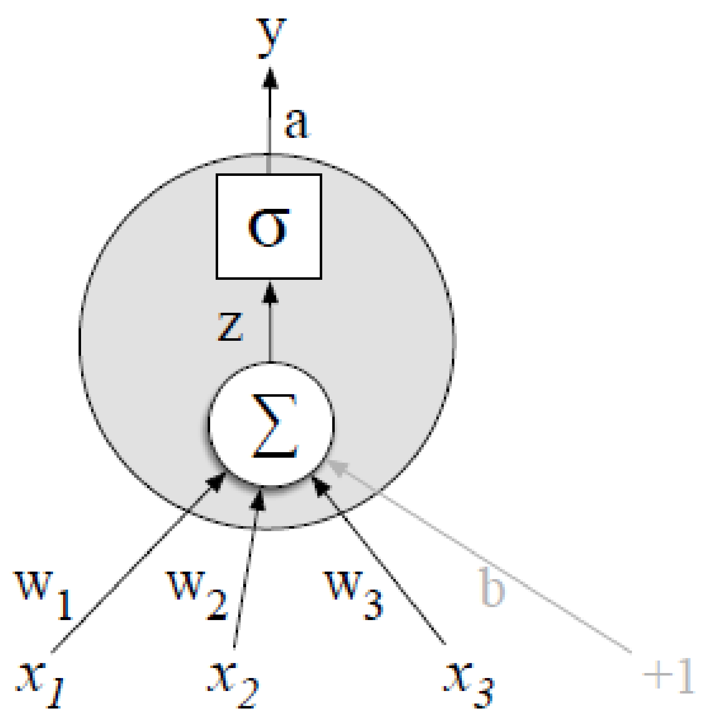


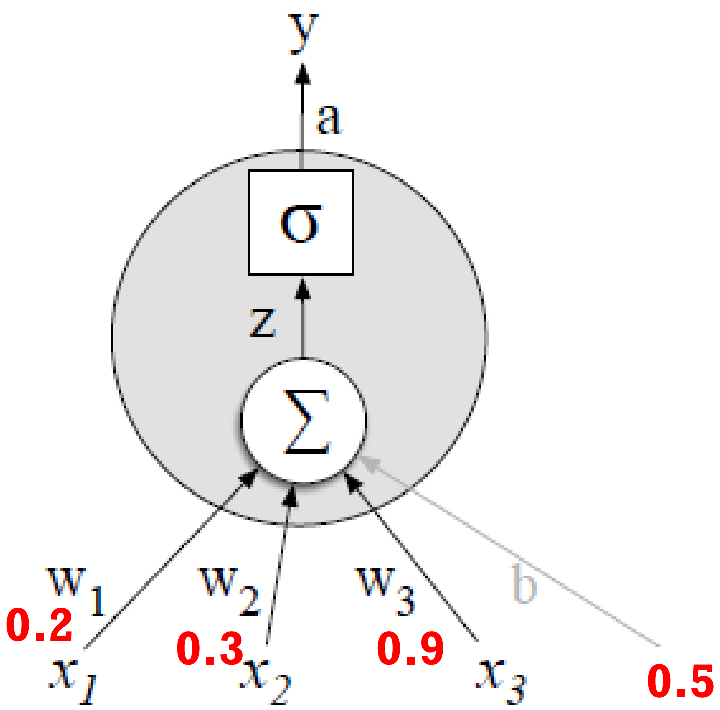
Figure 7.1 The sigmoid function takes a real value and maps it to the range [0, 1]. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

SIGMOID

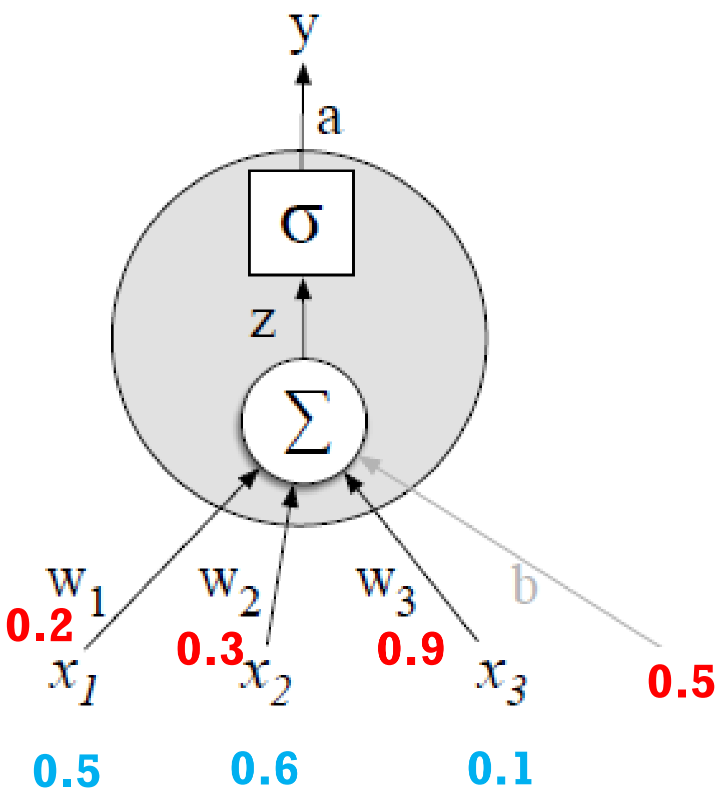
- Input: x_1, x_2, x_3
- Computation
  - Weighted sum $(x_1 w_1 + x_2 w_2 + x_3 w_3)$
  - Adds bias term *b*
  - Passes sum through sigmoid function
- Output: # between 0..1



SIGMOID

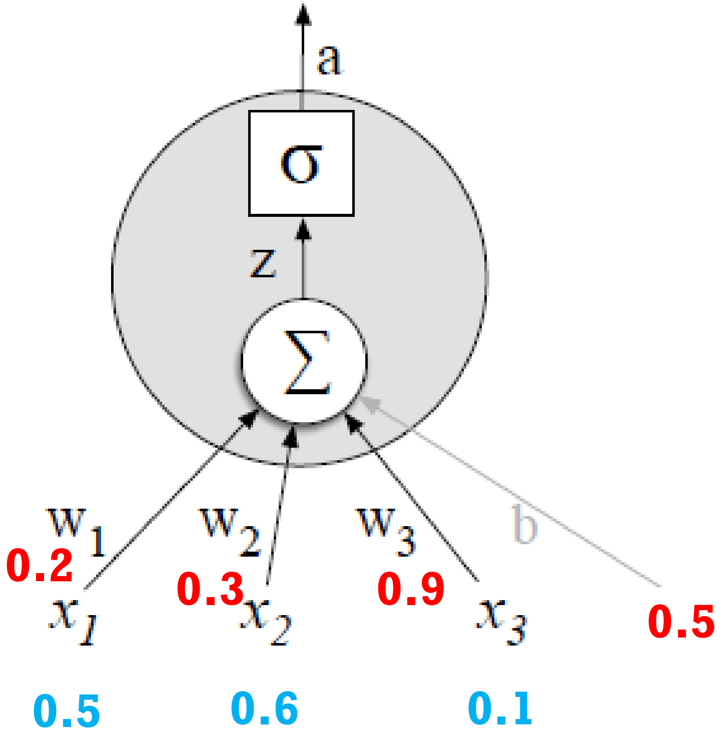


SIGMOID



SIGMOID

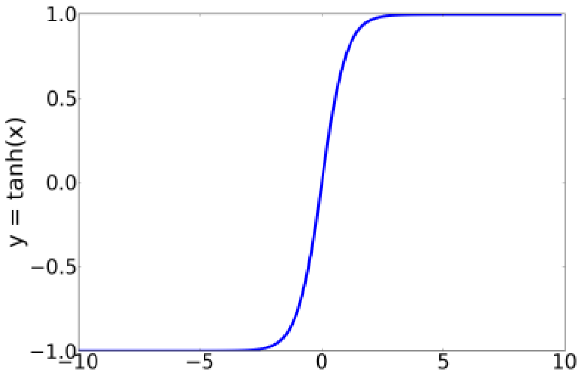
$$y = \sigma(w \cdot x + b) = \frac{1}{1+e^{-(w \cdot x+b)}} = \frac{1}{1+e^{-(.5*.2+.6*.3+.1*.9+.5)}} = e^{-0.87} = .70$$



ACTIVATION FUNCTIONS

tanh $$y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

- Very similar to sigmoid, but usually better
- Sigmoid variant
- Range: [-1, 1]

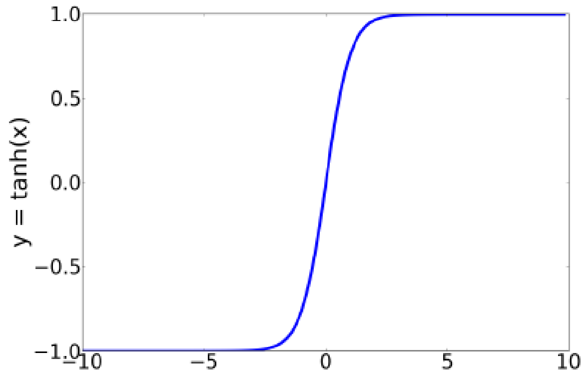


ACTIVATION FUNCTIONS

tanh

$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

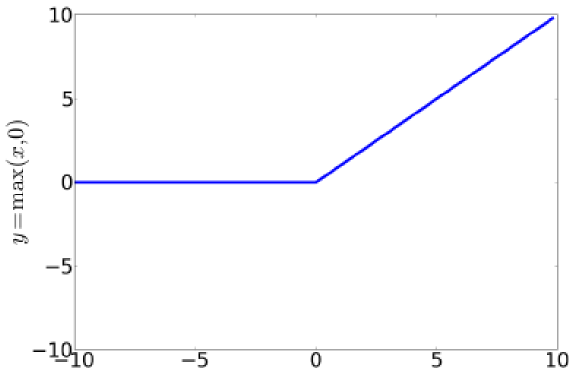
- Very similar to sigmoid, but usually better
- Sigmoid variant
- Range: [-1, 1]



ReLU

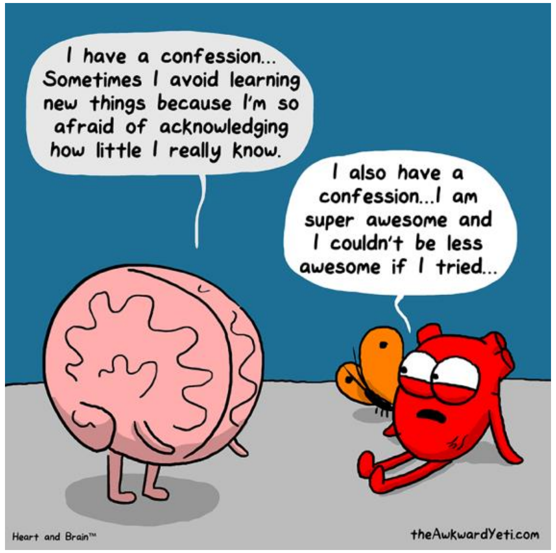
$$y = max(x, 0)$$

- Rectified Linear Unit
- Simplest
- Most commonly used
- Avoids saturation problem



WHY NEURAL NETWORKS?

- Combine neural units into larger & larger networks just like biological neurons
- Minsky & Papert (1969): single neural unit cannot compute simple functions of its input (so need layers)



XOR PROBLEM

AND		
X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR		
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

XOR PROBLEM

AND		
X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR		
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

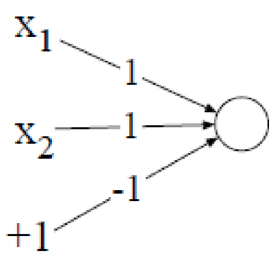
- M & P used perceptron
- Simple neural unit
- Binary output $y = 0$ or 1
- No non-linear activation function

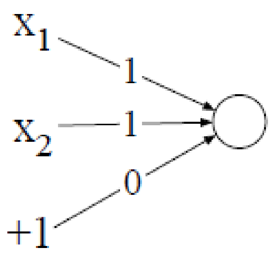
$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$

XOR PROBLEM

AND		
X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR		
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1



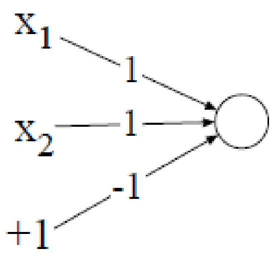


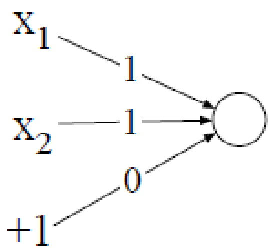
XOR PROBLEM

AND		
X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR		
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0



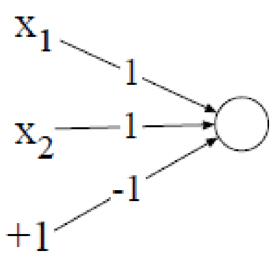


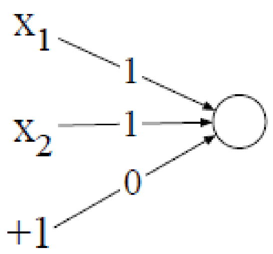
XOR PROBLEM

AND		
X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR		
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

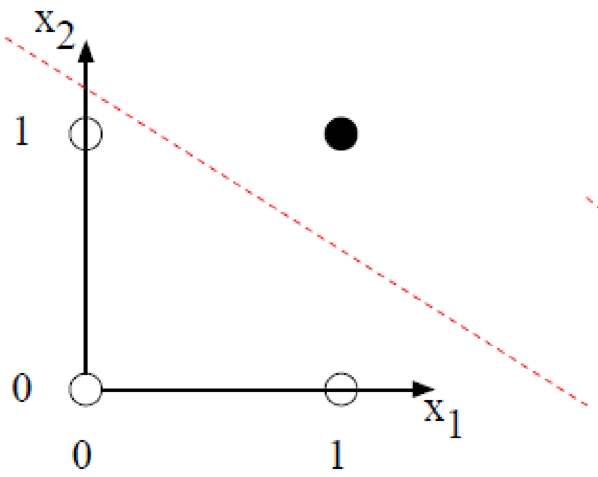




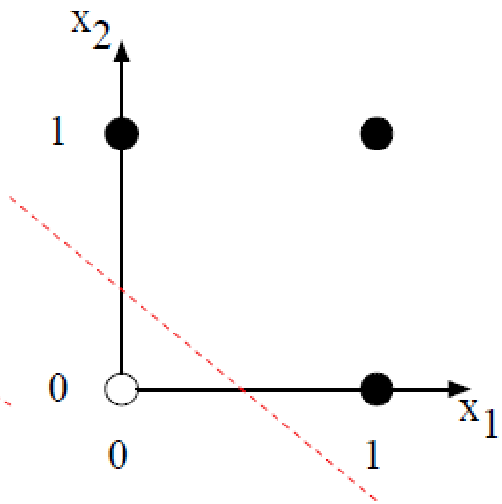
- Perceptron for XOR?
- Linear classifier
  - For 2D input x_1 & x_2
    $w_1x_1 + w_2x_2 + b = 0$
  - It's a line → decision boundary

XOR PROBLEM

AND		
X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1



a) x_1 AND x_2

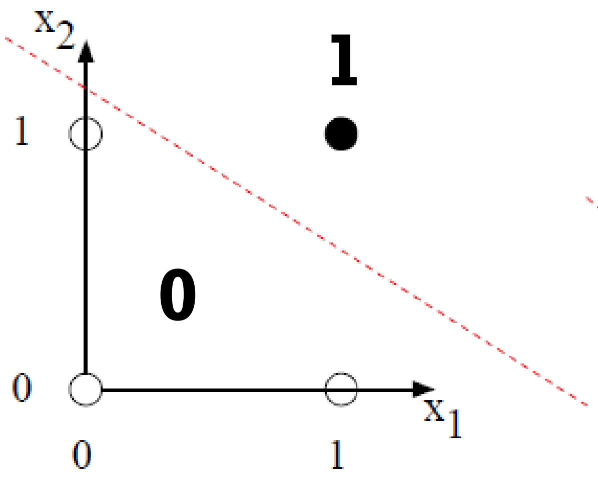


b) x_1 OR x_2

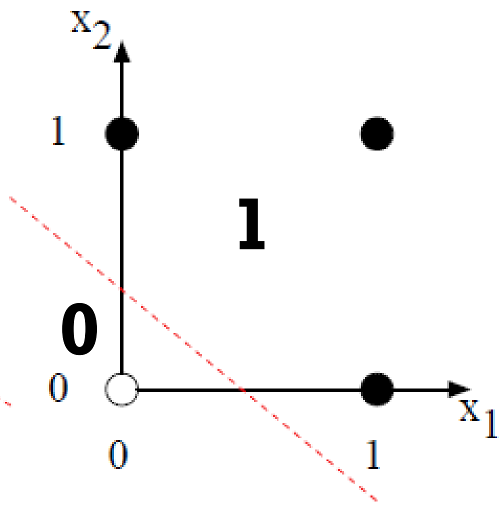
OR		
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR PROBLEM

AND		
X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1



a) x_1 AND x_2

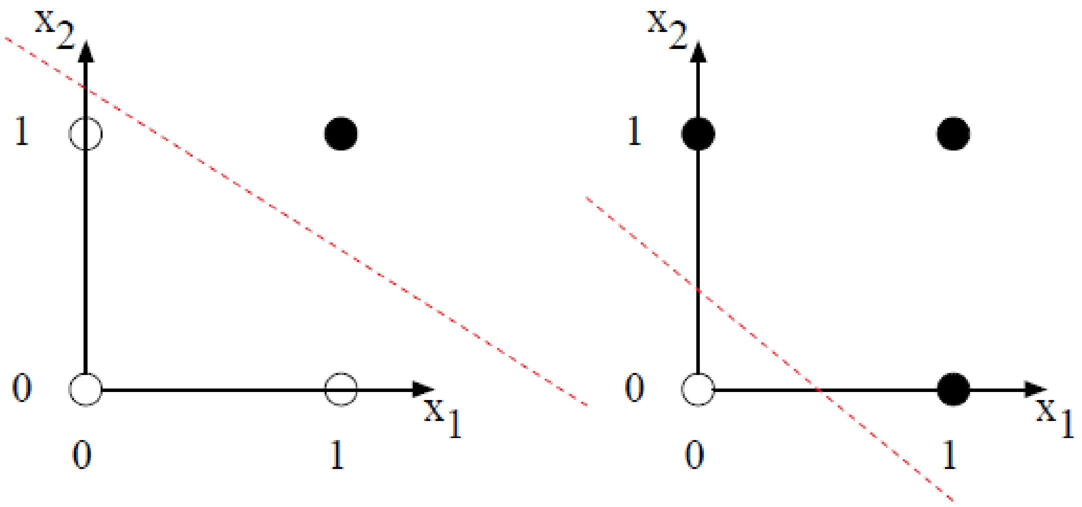


b) x_1 OR x_2

OR		
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR PROBLEM

AND		
X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1



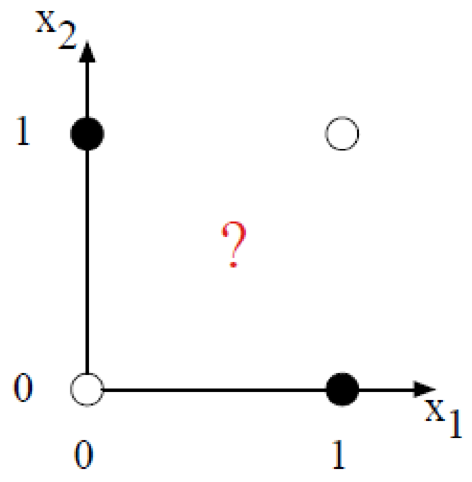
a) x_1 AND x_2 b) x_1 OR x_2

OR		
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1

Linearly Separable

XOR PROBLEM

XOR		
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

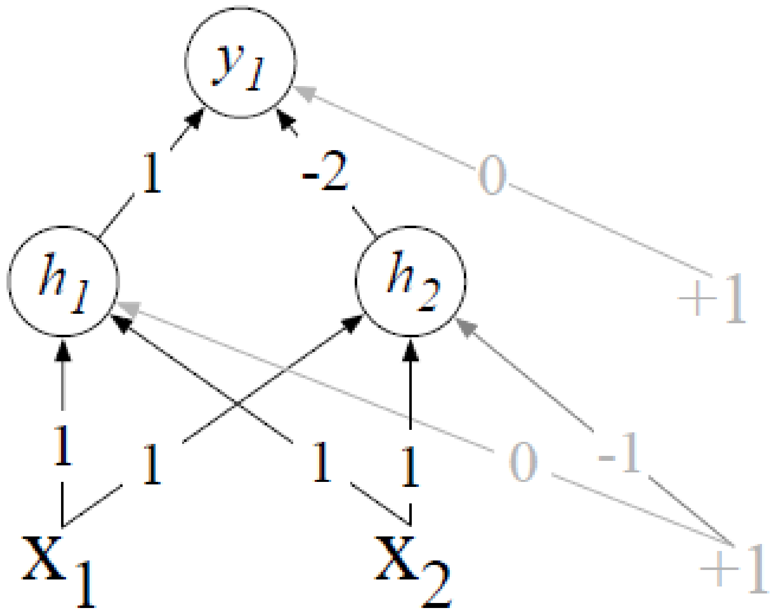


c) x_1 XOR x_2

Not Linearly Separable

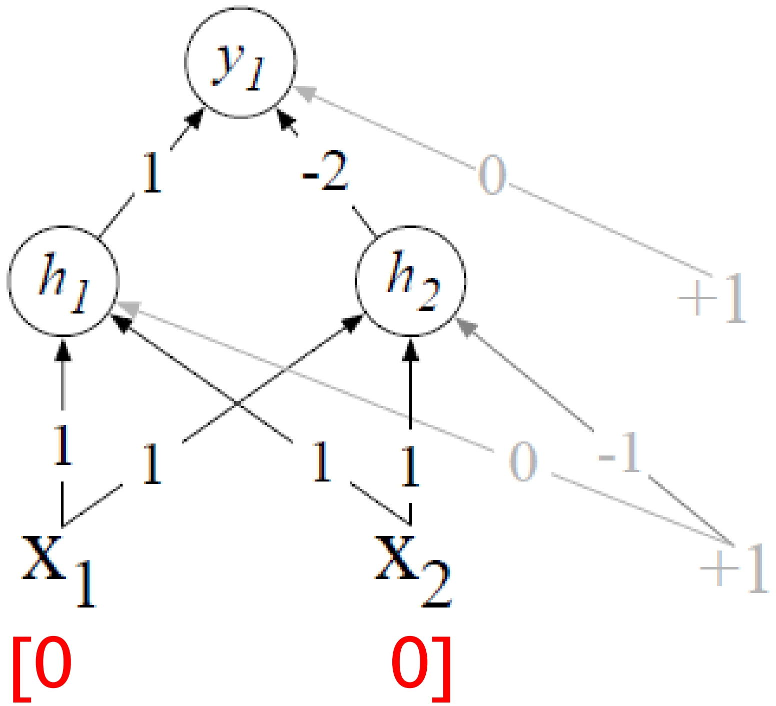
SOLUTION: NEURAL NETWORKS!

- Calculate XOR with a layered network of units
- Goodfellow et al. solution: use 2 layers of ReLU-based units



SOLUTION: NEURAL NETWORKS!

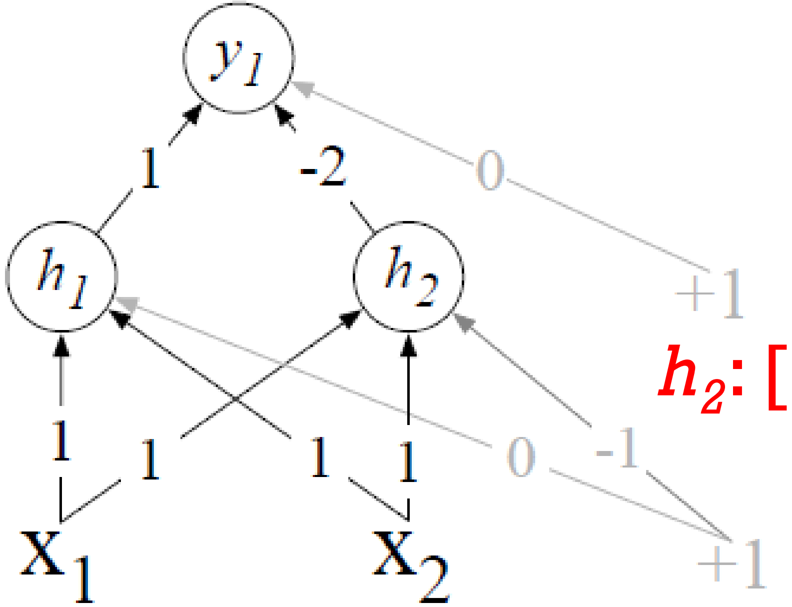
- Calculate XOR with a layered network of units
- Goodfellow et al. solution: use 2 layers of ReLU-based units



SOLUTION: NEURAL NETWORKS!

- Calculate XOR with a layered network of units
- Goodfellow et al. solution: use 2 layers of ReLU-based units

h_1: [(0*1 + 0*1) + 0]

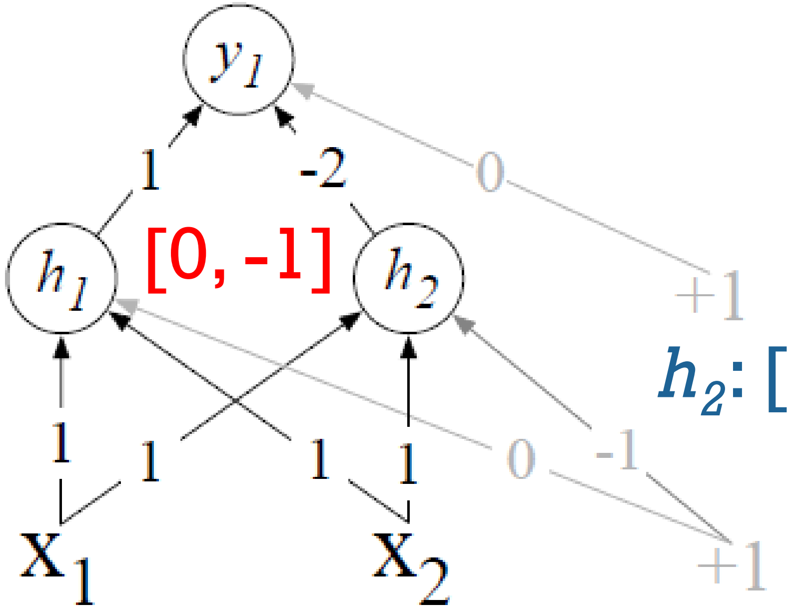


h_2: [(0*1 + 0*1) + -1]

[0 0]

SOLUTION: NEURAL NETWORKS!

- Calculate XOR with a layered network of units
- Goodfellow et al. solution: use 2 layers of ReLU-based units



h_1: [(0*1 + 0*1) + 0]

h_2: [(0*1 + 0*1) + -1]

[0 0]

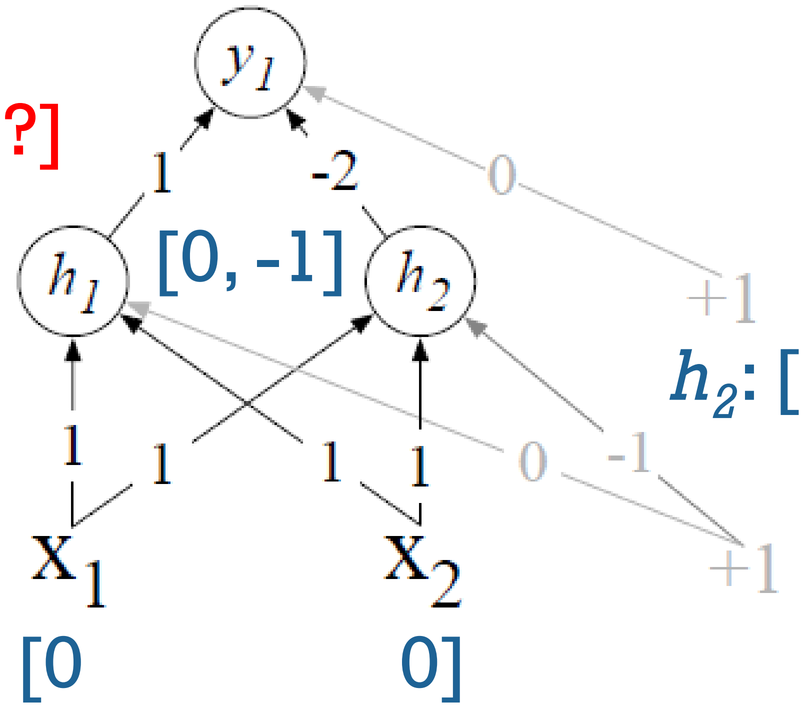
SOLUTION: NEURAL NETWORKS!

- Calculate XOR with a layered network of units
- Goodfellow et al. solution: use 2 layers of ReLU-based units

ReLU on [0, -1] = [?, ?]

h_1: [(0*1 + 0*1) + 0]

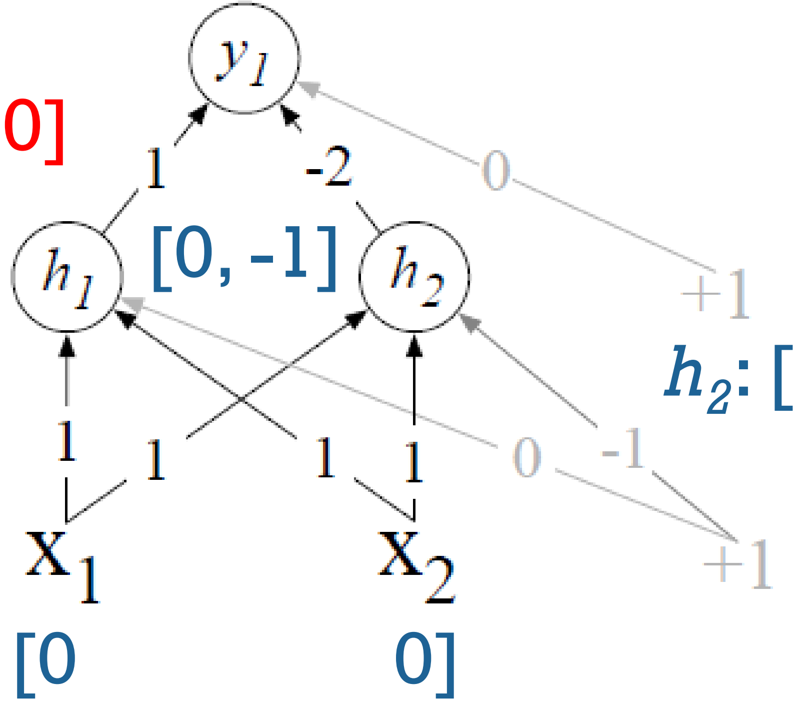
h_2: [(0*1 + 0*1) + -1]



SOLUTION: NEURAL NETWORKS!

- Calculate XOR with a layered network of units
- Goodfellow et al. solution: use 2 layers of ReLU-based units

ReLU on [0, -1] = [0, 0]



h_1: [(0*1 + 0*1) + 0]

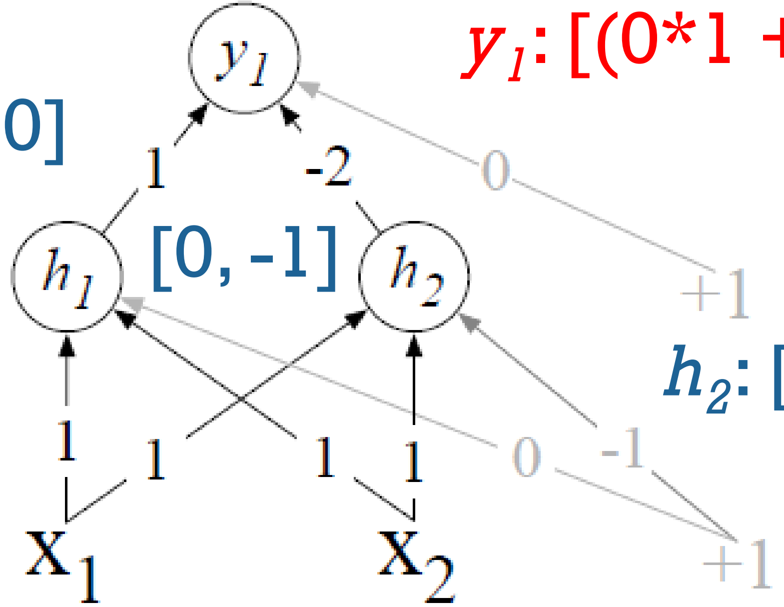
h_2: [(0*1 + 0*1) + -1]

SOLUTION: NEURAL NETWORKS!

- Calculate XOR with a layered network of units
- Goodfellow et al. solution: use 2 layers of ReLU-based units

y_1: [(0*1 + 0*-2) + 0] = 0

ReLU on [0, -1] = [0, 0]



[0, -1]

h_1: [(0*1 + 0*1) + 0]

h_2: [(0*1 + 0*1) + -1]

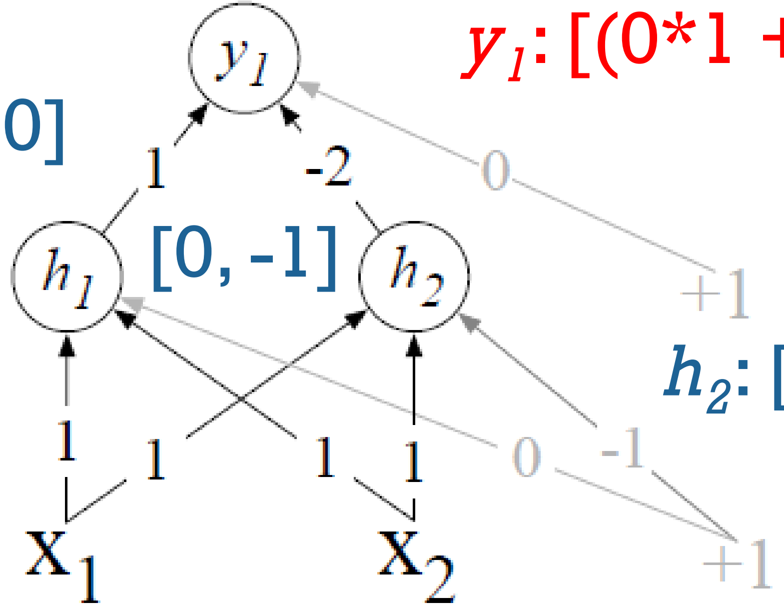
[0 0]

SOLUTION: NEURAL NETWORKS!

- Calculate XOR with a layered network of units
- Goodfellow et al. solution: use 2 layers of ReLU-based units

y_1: [(0*1 + 0*-2) + 0] = 0

ReLU on [0, -1] = [0, 0]



[0 0]

h_2: [(0*1 + 0*1) + -1]

Here we gave the NN the weights but NNs auto-learn their weights through error backpropagation → so NNs can automatically learn useful representations of input

FEED FORWARD NEURAL NETWORKS

- Multi-layer network of connected units with no cycles
  - No cycles = unit outputs from each layer are sent to the next higher layer & no output is passed back to lower layers
- Multi-layer feed forward network {=, ≠} multi-layer perceptron (MLP)???

FEED FORWARD NEURAL NETWORKS

- 3 kinds of nodes
  - Input units
  - Hidden units
  - Output units

FEED FORWARD NEURAL NETWORKS

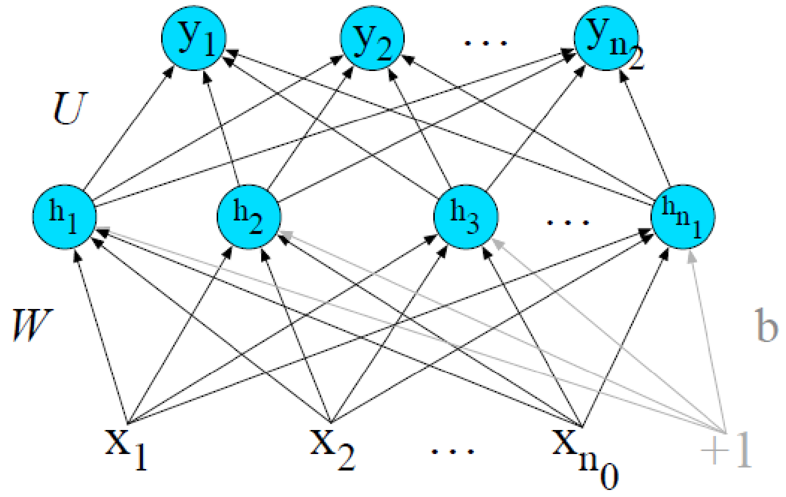
- 3 kinds of nodes
  - Input units: scalar values
  - Hidden units
  - Output units

FEED FORWARD NEURAL NETWORKS

- 3 kinds of nodes
  - Input units: scalar values
  - Hidden units: take weighted sum of inputs & apply activation function (non-linear function)
  - Output units

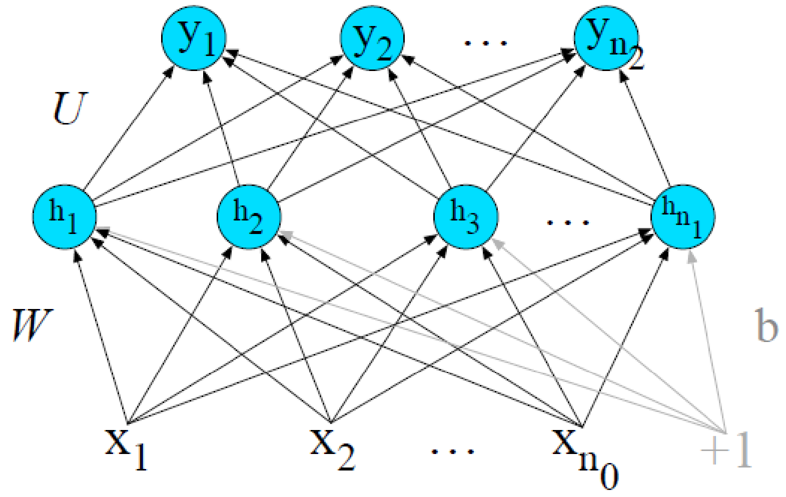
FEED FORWARD NEURAL NETWORKS

- 3 kinds of nodes
  - Input units: scalar values
  - Hidden units: take weighted sum of inputs & apply activation function (non-linear function)
  - Output units: answer!



FEED FORWARD NEURAL NETWORKS

- Hidden layer: core of the NN, composed of hidden units
- Fully-connected
  - Input of each unit in each layer is output from all units in the previous layer
  - Link between every pair of units from 2 adjacent layers



HIDDEN LAYER

- Hidden unit parameters: vector for w & b scalar
- Hidden layer parameters: W matrix & $\boldsymbol{b}$ vector
  - Parameters for entire hidden layer (all hidden units)
  - W combines weight vector w_i for each hidden unit h_i
  - W_{ij} = weight of connection from i^{th} input x_i to j^{th} hidden unit h_j
  - $\mathbf{b}$ combines bias b_i for each hidden unit h_i
- Advantage of using a single matrix W to represent weights for entire layer?

HIDDEN LAYER

- Efficient hidden layer computation using simple matrix operations
  - 1. Multiply weight matrix W by input vector $\mathbf{x}$
  - 2. Add bias vector $\mathbf{b}$
  - 3. Apply activation function g (element-wise)

HIDDEN LAYER

- Efficient hidden layer computation using simple matrix operations
  - 1. Multiply weight matrix W by input vector **x**
  - 2. Add bias vector **b**
  - 3. Apply activation function g (element-wise)

$$h = \sigma(Wx + b)$$

HOW DO WE GET Y?

- Hidden layer forms a representation *h* of input
- Output layer takes *h* & computes final output value

HOW DO WE GET Y?

- Real valued #
- If classification is NN's goal:
  - Binary task (e.g., sentiment classification)
    → y = probability of positive vs. negative sentiment
  - Multinomial classification (e.g., part-of-speech (POS) tagging)
    → 1 output node for each POS where value = probability of that POS
    → all values of output nodes must sum to 1
    → output layer ≈ probability distribution across output nodes

HOW DO WE GET Y?

- Output layer has a weight matrix U (bias vector optional)
- Intermediate output z = weight matrix U * input vector h
- At this point, what's wrong with z?

$$z = Uh$$

HOW DO WE GET Y?

- z is a vector of real-valued #s
- For classification need a vector of probabilities

HOW DO WE GET Y?

- z is a vector of real-valued #s
- For classification need a vector of probabilities
- Normalize!
- **Softmax** function "normalizes" a vector of real values by converting it into one that encodes a probability distribution (all #s are between 0..1 & sum to 1)

SOFTMAX

- Just plug in real values from *z*

$$\mathrm{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_j}} \quad 1 \leq i \leq d$$

- LR uses to create a probability distribution from sum of weights * features

NN VS. LR

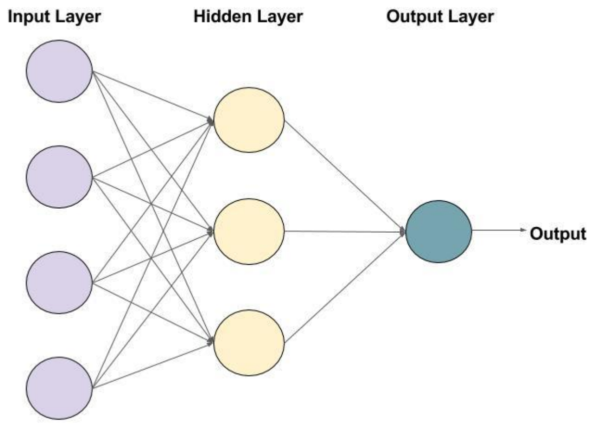
- Think of NN as classifier with 1 hidden layer
  - Build input representation (hidden) vector *h*
  - Run standard LR on features the NN developed in *h*

NN VS. LR

- Think of NN as classifier with 1 hidden layer
  - Build input representation (hidden) vector *h*
  - Run standard LR on features the NN developed in *h*
- How NN differs from LR
  - Many layers (deep NN ≈ layer on layer of LR classifiers)
  - Instead of using feature templates/engineering
    use previous layers to induce feature representations

FEEDFORWARD NN

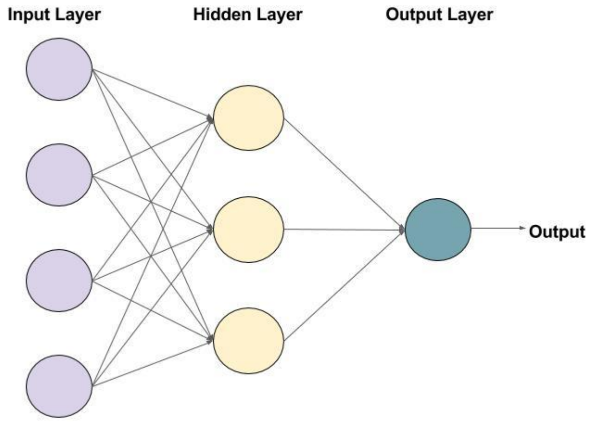
$$h = \sigma(Wx + b)$$
$$z = Uh$$
$$y = softmax(z)$$



FEEDFORWARD NN

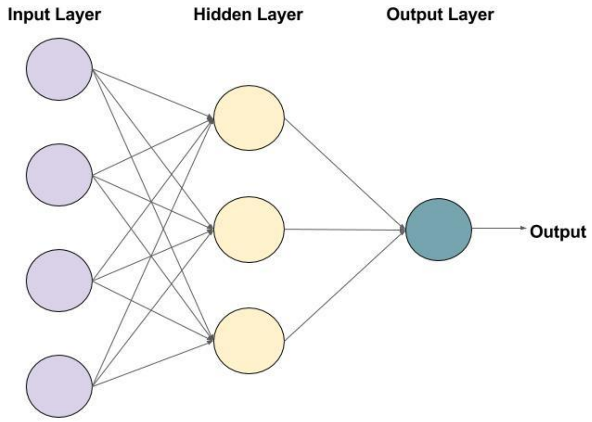
$$h = \sigma(Wx + b)$$
$$z = Uh$$
$$y = softmax(z)$$

$$\begin{aligned} z^{[1]} &= W^{[1]}a^{[0]} + b^{[1]} \\ a^{[1]} &= g^{[1]}(z^{[1]}) \\ z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} &= g^{[2]}(z^{[2]}) \\ \hat{y} &= a^{[2]} \end{aligned}$$



FEEDFORWARD NN

$$h = \sigma(Wx + b)$$
$$z = Uh$$
$$y = softmax(z)$$



$$\begin{aligned} z^{[1]} &= W^{[1]}a^{[0]} + b^{[1]} \\ a^{[1]} &= g^{[1]}(z^{[1]}) \\ z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} &= g^{[2]}(z^{[2]}) \\ \hat{y} &= a^{[2]} \end{aligned}$$

for i **in 1..n**

$$z^{[i]} = W^{[i]}\, a^{[i-1]} + b^{[i]}$$
$$a^{[i]} = g^{[i]}(z^{[i]})$$
$$\hat{y} = a^{[n]}$$

TRAINING NN

- Feedforward NN is instance of supervised ML
- We know the correct output *y* for each observation *x*
- NN produces *y** (prediction; estimate of true *y*)

TRAINING NN

- Goal of training: learn parameters
  - Learn $W^{[i]}$ & $b^{[i]}$ for each layer i that makes y^* for each training observation as close as possible to true y

TRAINING NN

- Goal of training: learn parameters
  - Learn $W^{[i]}$ & $b^{[i]}$ for each layer i that makes y^* for each training observation as close as possible to true y
- Follow same steps as LR
  - Cross-entropy loss: to model distance between y^* & y
  - Gradient descent: to find parameters to minimize loss
  - Optimization is tricky now!

TRAINING NN

- Optimization: GD requires knowing gradient of loss function
  - Vector contains partial derivative of loss function w.r.t. each parameter
- In LR: directly compute derivative
- In NN: how can we compute the partial derivative of some weight in layer 1 when loss is attached to later layer?

TRAINING NN

- Optimization: GD requires knowing gradient of loss function
  - Vector contains partial derivative of loss function w.r.t. each parameter
- In LR: directly compute derivative
- In NN: how can we compute the partial derivative of some weight in layer 1 when loss is attached to later layer?
  - Error backpropagation or reverse differentiation

TRAINING NN: CROSS-ENTROPY LOSS

- Binary classification (with sigmoid on final output layer) == LR loss equation

$$L_{CE}(\hat{y},y) = -\log p(y|x) \;=\; -[y\log\hat{y} + (1-y)\log(1-\hat{y})]$$

TRAINING NN: CROSS-ENTROPY LOSS

- Binary classification (with sigmoid on final output layer) == LR loss equation

$$L_{CE}(\hat{y}, y) = -\log p(y|x) \; = \; -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$

- Multinomial classification $$L_{CE}(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log \hat{y}_i$$

TRAINING NN: CROSS-ENTROPY LOSS

- Binary classification (with sigmoid on final output layer) == LR loss equation

$$L_{CE}(\hat{y},y) = -\log p(y|x) \;=\; -[y\log\hat{y} + (1-y)\log(1-\hat{y})]$$

- Multinomial classification $L_{CE}(\hat{y},y) = -\sum_{i=1}^{C} y_i \log\hat{y}_i$

- Hard classification (only 1 class is correct)

$$L_{CE}(\hat{y},y) \;=\; -\log\hat{y}_i$$

negative log likelihood loss

TRAINING NN: CROSS-ENTROPY LOSS

- Binary classification (with sigmoid on final output layer) == LR loss equation

$$L_{CE}(\hat{y}, y) = -\log p(y|x) \;=\; -[y\log\hat{y} + (1-y)\log(1-\hat{y})]$$

- Multinomial classification $L_{CE}(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log \hat{y}_i$

- Hard classification (only 1 class is correct)

$$L_{CE}(\hat{y}, y) \;=\; -\log \hat{y}_i$$

negative log likelihood loss

softmax
K classes

$$L_{CE}(\hat{y}, y) \;=\; -\log \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

TRAINING NN: COMPUTE GRADIENT

- Gradient = partial derivative of loss function w.r.t. each parameter
- Simple cases
  - 1 weight layer & sigmoid output → derivative of LR loss
  - 1 hidden layer & softmax output → derivative of softmax

TRAINING NN: COMPUTE GRADIENT

- Gradient = partial derivative of loss function w.r.t. each parameter
- Simple cases
  - 1 weight layer & sigmoid output → derivative of LR loss
  - 1 hidden layer & softmax output → derivative of softmax
- Problems
  - Only get correct update for last layer
  - Deep: compute derivative w.r.t. weight parameters that appear in the early layers but loss only computed at end
- Solution: error backpropagation (backprop)

BACKGROUND: COMPUTATION GRAPHS

- Represents process of computing mathematical expression broken down into separate operations
- Each operation is a node in the graph

COMPUTATION GRAPHS

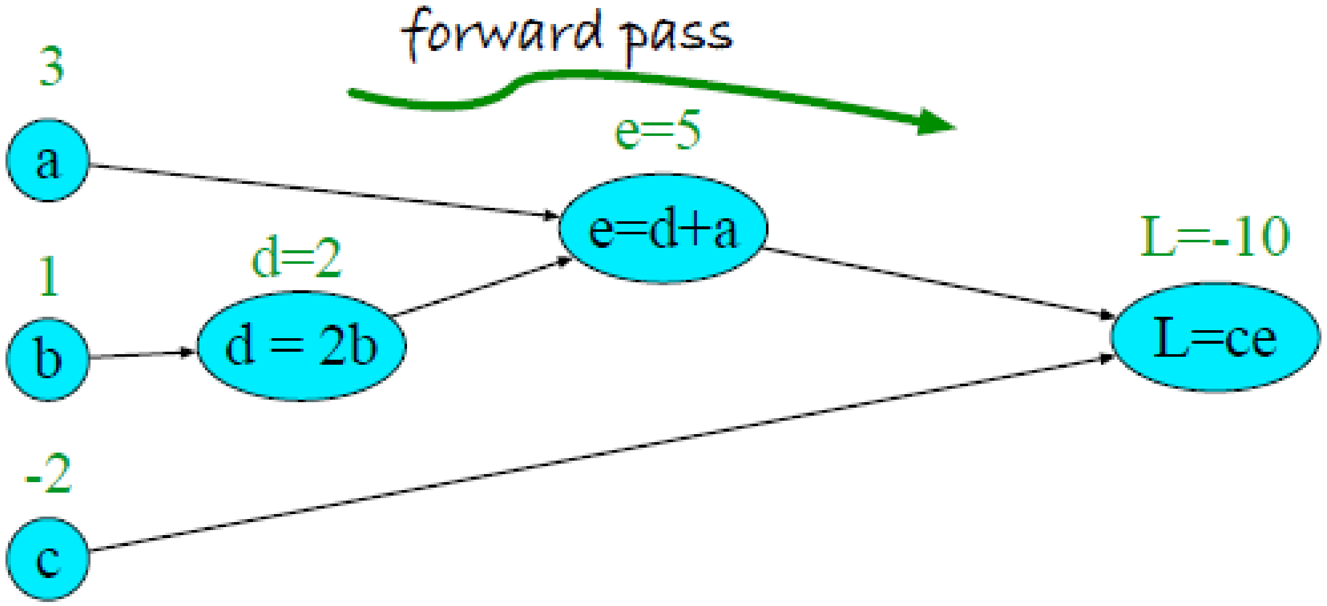
- Example: $L(a,b,c) = c(a + 2b)$
  - $d = 2 * b$
  - $e = a + d$
  - $L = c * e$

COMPUTATION GRAPHS

- Example: $L(a,b,c) = c(a + 2b)$
  - $d = 2 * b$
  - $e = a + d$
  - $L = c * e$
- Represent steps as graph
  - Nodes = operations
  - Directed edges = show output from operation as input to next
- Forward pass: apply each operation left to right, passing outputs forward to next node

COMPUTATION GRAPHS

- Example: $L(a=3, b=1, c=-2) = c(a + 2b)$
  - $d = 2 * b$
  - $e = a + d$
  - $L = c * e$



BACKWARD DIFFERENTIATION ON COMPUTATION GRAPHS

- Backward pass: used to compute derivatives for weight update
- Example: compute derivative of output function L w.r.t. each input variable

BACKWARD DIFFERENTIATION ON COMPUTATION GRAPHS

- **Backward pass**: used to compute derivatives for weight update
- Example: compute derivative of output function L w.r.t. each input variable
  - Derivatives tell how much small change in variable affects L
  - $\frac{\partial L}{\partial a} \; \frac{\partial L}{\partial b} \; \frac{\partial L}{\partial c}$

BACKWARD DIFFERENTIATION ON COMPUTATION GRAPHS

- Backward pass: used to compute derivatives for weight update
- Example: compute derivative of output function L w.r.t. each input variable
  - Derivatives tell how much small change in variable affects L
  - $\frac{\partial L}{\partial a}$ $\frac{\partial L}{\partial b}$ $\frac{\partial L}{\partial c}$
  - Using $L = ce$ & chain rule: $\frac{\partial L}{\partial c} = e, \frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}, \frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$

BACKWARD DIFFERENTIATION ON COMPUTATION GRAPHS

- Backward pass: used to compute derivatives for weight update
- Example: compute derivative of output function L w.r.t. each input variable
  - Derivatives tell how much small change in variable affects L
  - $\frac{\partial L}{\partial a}$ $\frac{\partial L}{\partial b}$ $\frac{\partial L}{\partial c}$
  - Using $L = ce$ & chain rule: $\frac{\partial L}{\partial c} = \mathrm{e}$, $\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$, $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$

BACKWARD DIFFERENTIATION ON COMPUTATION GRAPHS

- Using $L = ce$ & chain rule: $\frac{\partial L}{\partial c} = \text{e}, \frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}, \frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$

$$
\begin{aligned}
L = ce &: \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e \\
e = a + d &: \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1 \\
d = 2b &: \quad \frac{\partial d}{\partial b} = 2
\end{aligned}
$$

BACKWARD DIFFERENTIATION ON COMPUTATION GRAPHS

- Using $L = ce$ & chain rule: $\frac{\partial L}{\partial c} = \mathrm{e}, \frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}, \frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$

$$
\begin{aligned}
L = ce &: \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e \\
e = a + d &: \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1 \\
d = 2b &: \quad \frac{\partial d}{\partial b} = 2
\end{aligned}
$$

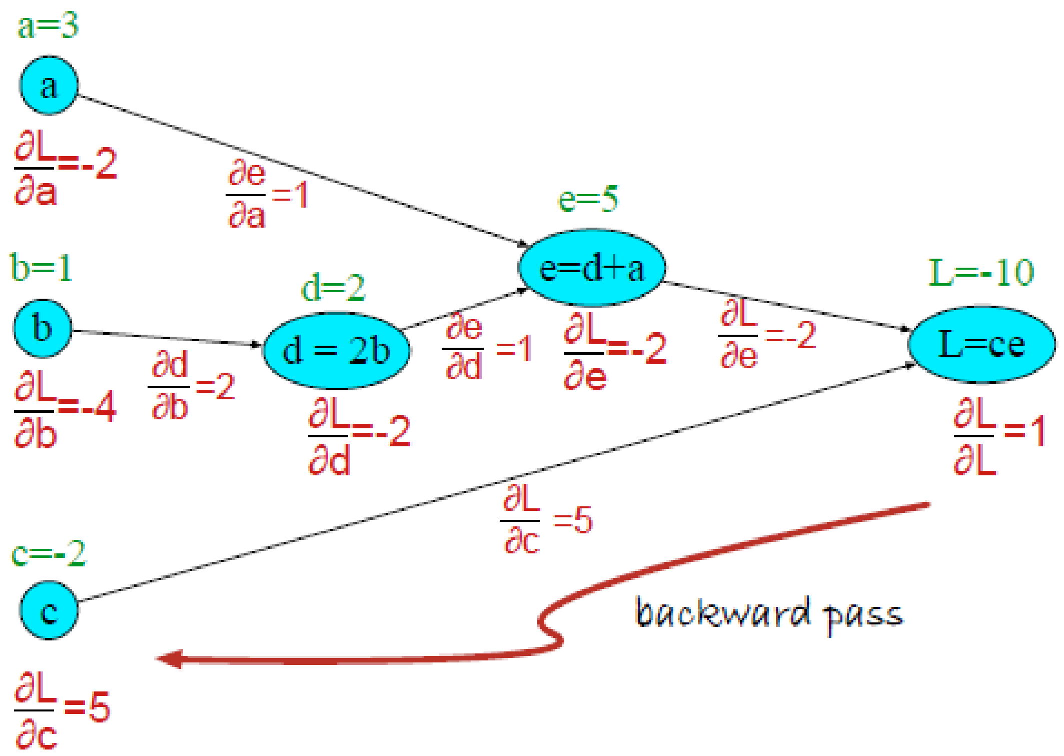
- To compute backward pass
  - Compute each partial along each edge of CG from right to left
  - Multiply necessary partials to get final derivative needed

BACKWARD DIFFERENTIATION ON COMPUTATION GRAPHS

$$L = ce \quad : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

$$e = a + d \quad : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

$$d = 2b \quad : \quad \frac{\partial d}{\partial b} = 2$$

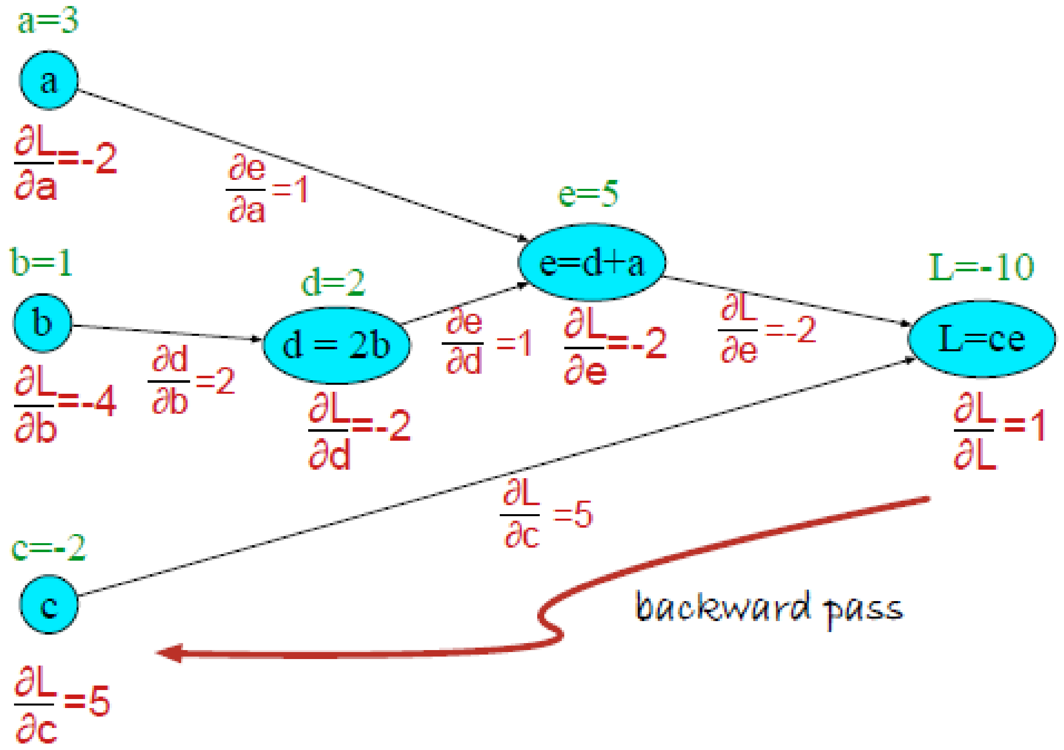


BACKWARD DIFFERENTIATION ON COMPUTATION GRAPHS

$$L = ce \; : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

$$e = a + d \; : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

$$d = 2b \; : \quad \frac{\partial d}{\partial b} = 2$$



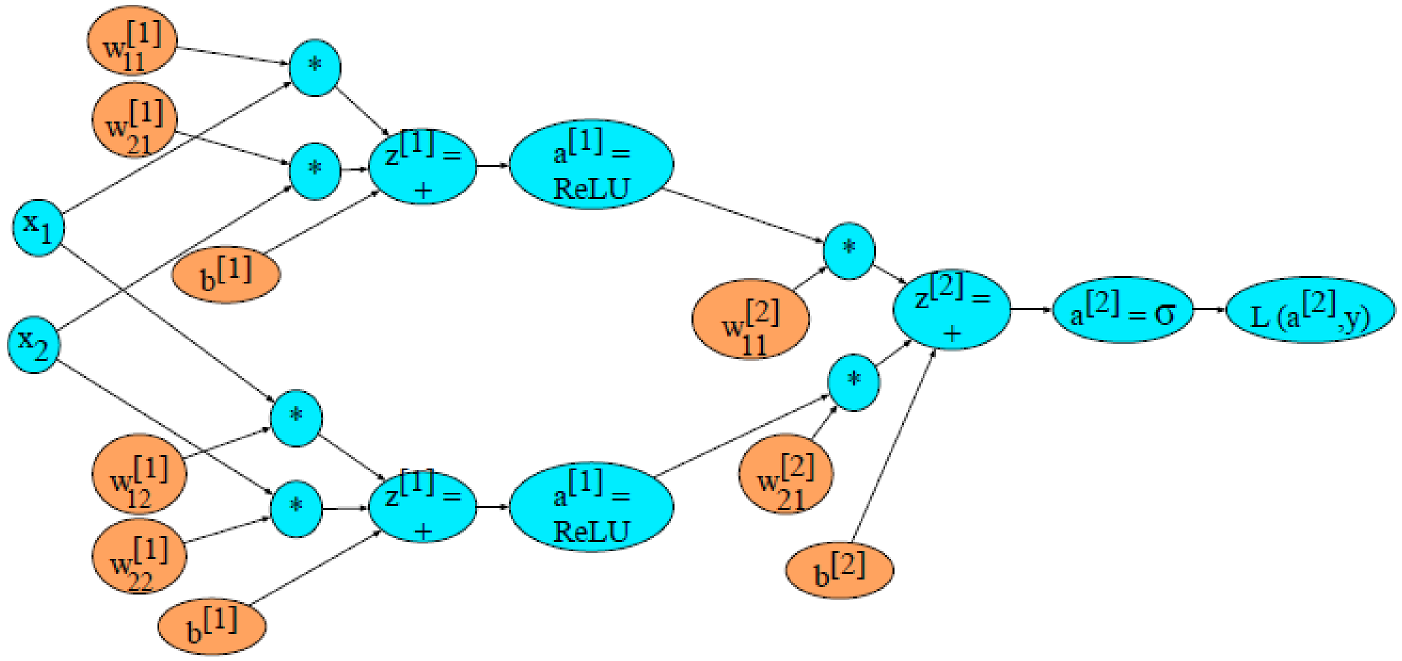
1. Compute local partial derivate w.r.t. parent
2. Multiply it by partial derivative passed down from parent
3. Pass value to child

BACKWARD DIFFERENTIATION FOR NN

BACKWARD DIFFERENTIATION FOR NN

- Binary classification (sigmoid)

$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \mathrm{RELu}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$



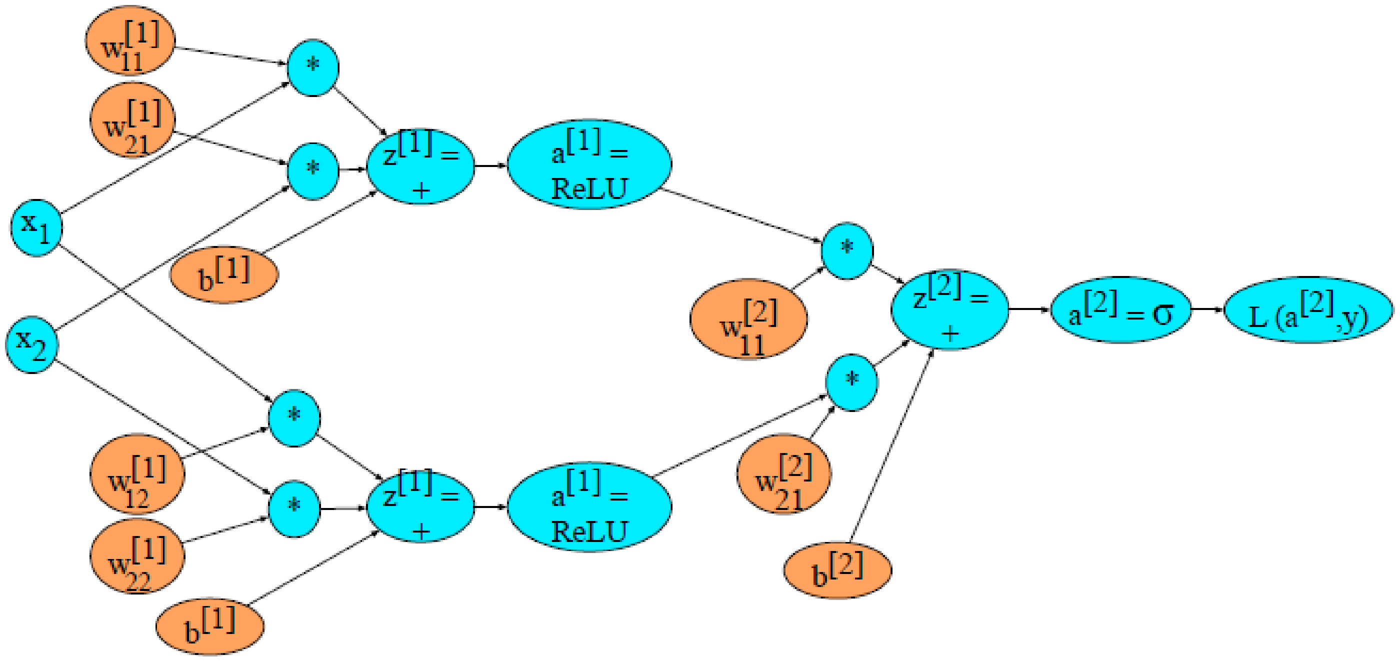
BACKWARD DIFFERENTIATION FOR NN

- Activation function derivatives

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$$

$$\frac{d\tanh(z)}{dz} = 1-\tanh^2(z)$$

$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for\ \ x<0 \\ 1 & for\ \ x\geq 0 \end{cases}$$



NN LEARNING

- NN Optimization is non-convex & more complex than LR
- Initialize weights with small random numbers
- Normalize input values to have 0 mean & unit variance
- Use dropout regularization to help avoid overfitting
  - Randomly drop some units & their connections from the network during training
- Tune hyperparameters on devset
  - NN parameters = W & b learned by GD
  - Learning rate η, mini-batch size, architecture, regularization …
    - # of layers, # of hidden nodes per layer, activation functions

- Neural Units
  - Biological neuron → artificial neuron
  - Inputs (x,w,b)
  - Activation functions
    - Sigmoid, tanh, ReLU
- XOR Problem
  - Unit → networks
- Feed forward NN
  - No cycles
  - Hidden layer & matrix computation
  - Output calculation
- Training NN
  - Cross-entropy loss, gradient descent, back prop

SUMMARY

Units

XOR Problem

Feed Forward Neural Network

Training NN

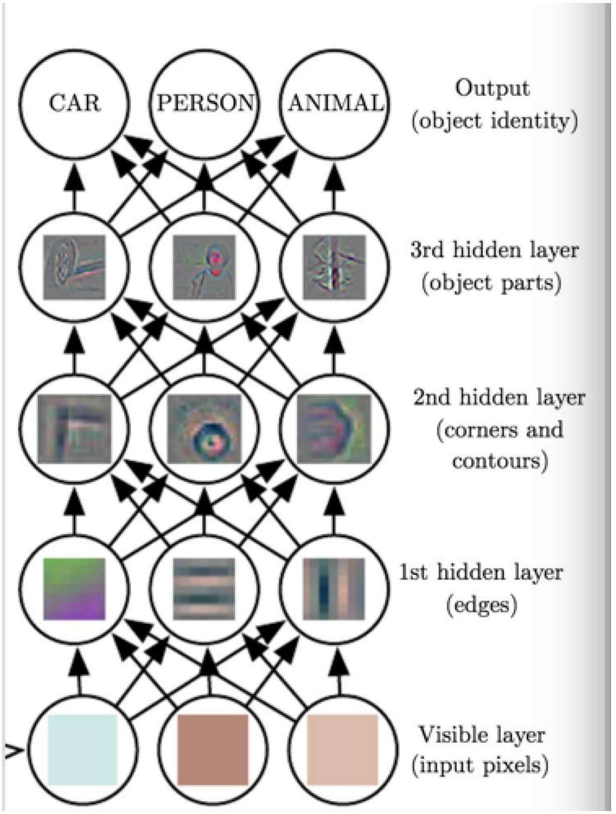
RNN & LSTM

NN

SIMPLE RNN

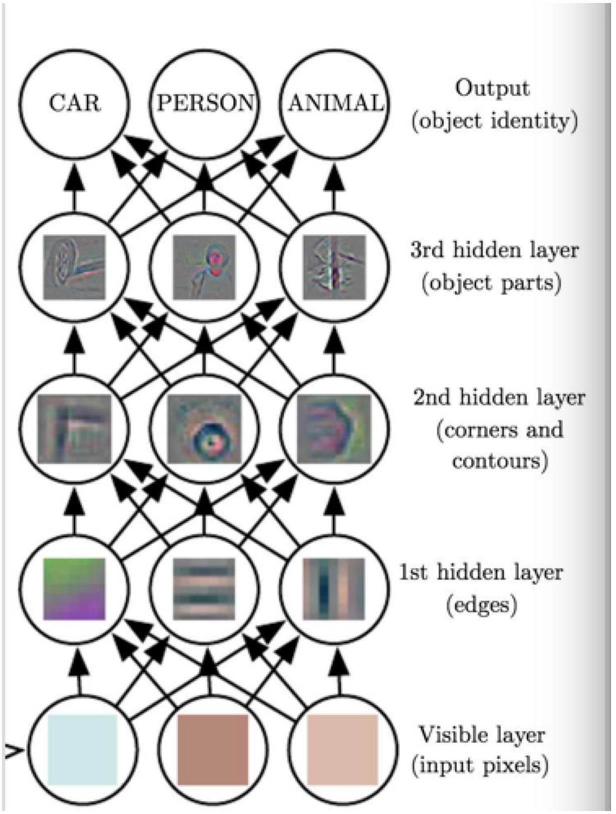
- RNN: any network that contains a **cycle** within its network connections
- Cycle: output value of unit/node is in/directly dependent on earlier outputs (as its input)
- Elman (1990) or simple recurrent networks
  - Effective for spoken & written language
  - Base for Encoder-Decoder models & QA models

FEEDFORWARD RECAP



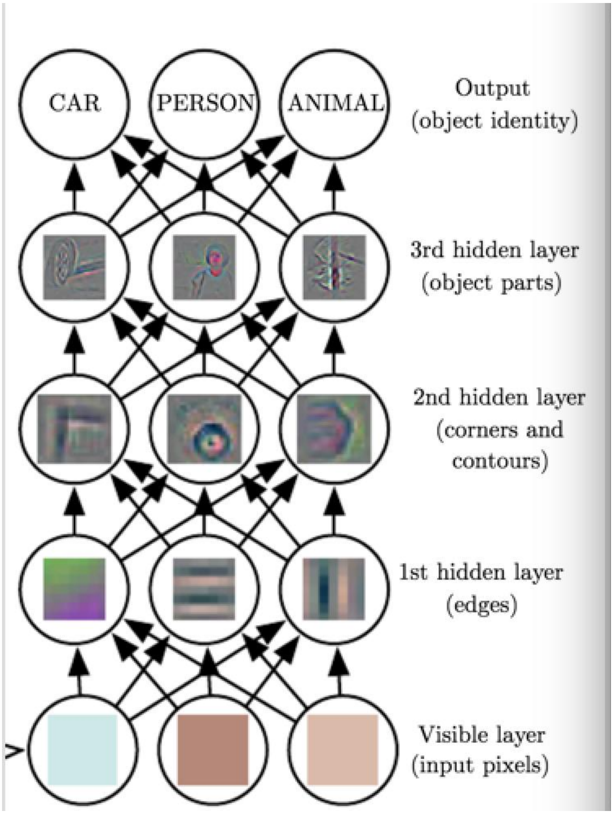
FEEDFORWARD RECAP

- Training
  - Input units represent info
  - Multiply by weights
  - If the sum of weights > threshold (activates) triggers next units

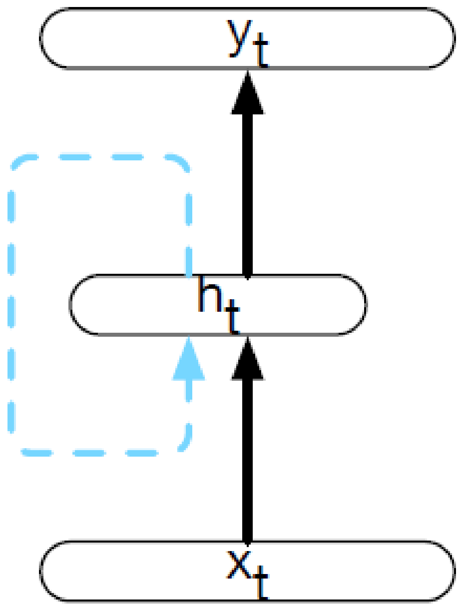


FEEDFORWARD RECAP

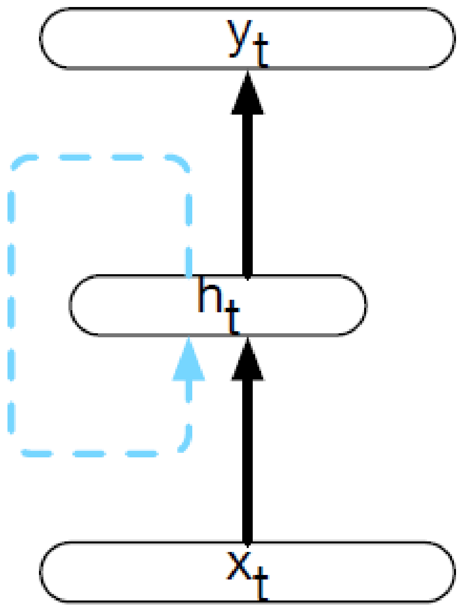
- Training
  - Input units represent info
  - Multiply by weights
  - If the sum of weights > threshold (activates) triggers next units
- Learning (Backprop)
  - Compare the output network produces ($y*$) with output should have produced (y)
  - Use difference between them to modify weights (work backwards)



SIMPLE RNN



SIMPLE RNN

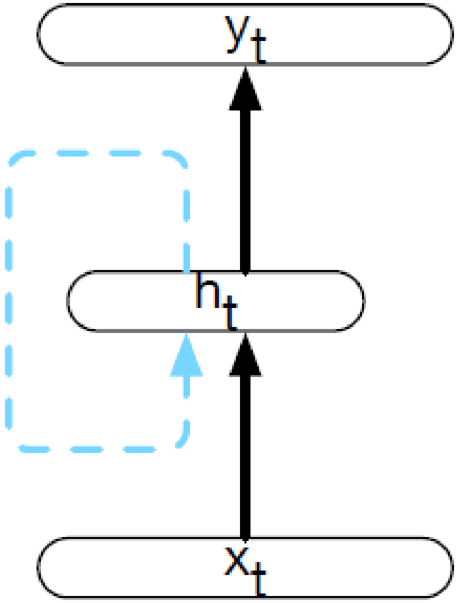


current input's features * weight matrix

SIMPLE RNN

pass through activation function to
compute activation value *a* for *h*

current input's features * weight matrix

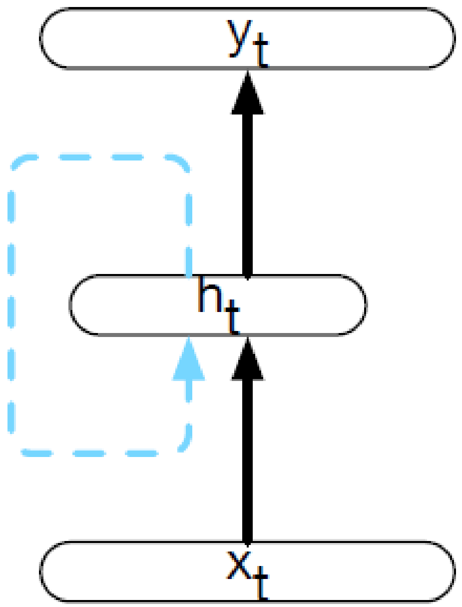


SIMPLE RNN

h calculates output value

pass through activation function to compute activation value *a* for *h*

current input's features * weight matrix

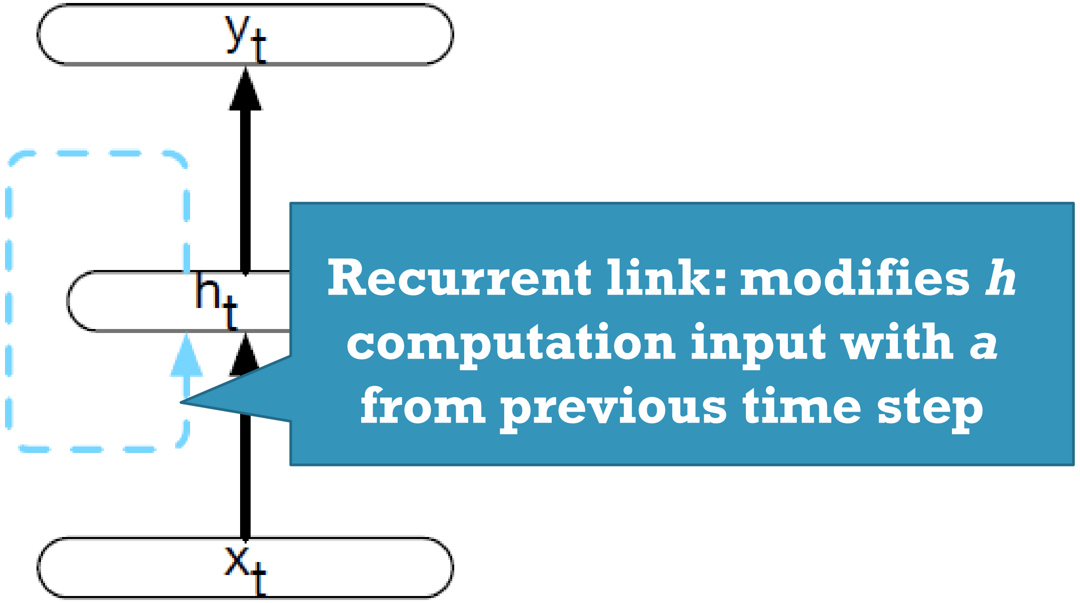


SIMPLE RNN

h calculates output value

pass through activation function to compute activation value *a* for *h*

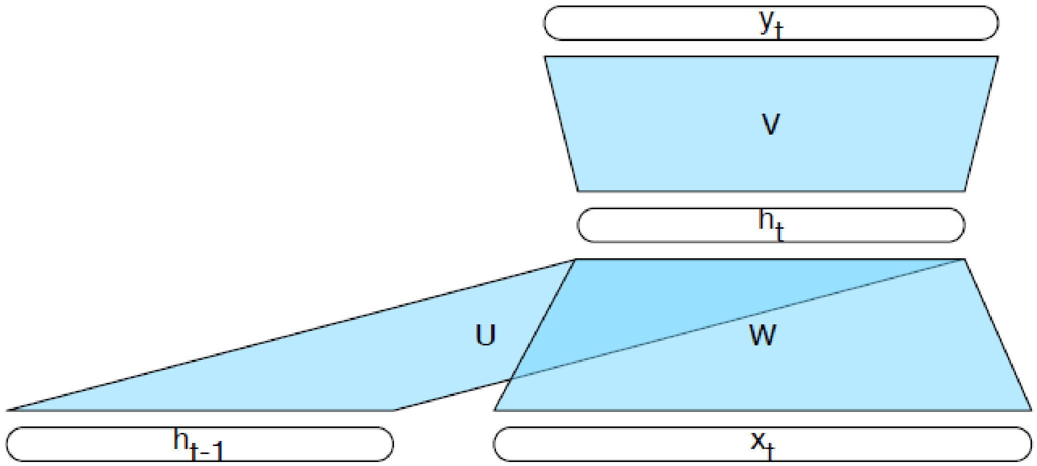
current input's features * weight matrix



RECURRENT LINK

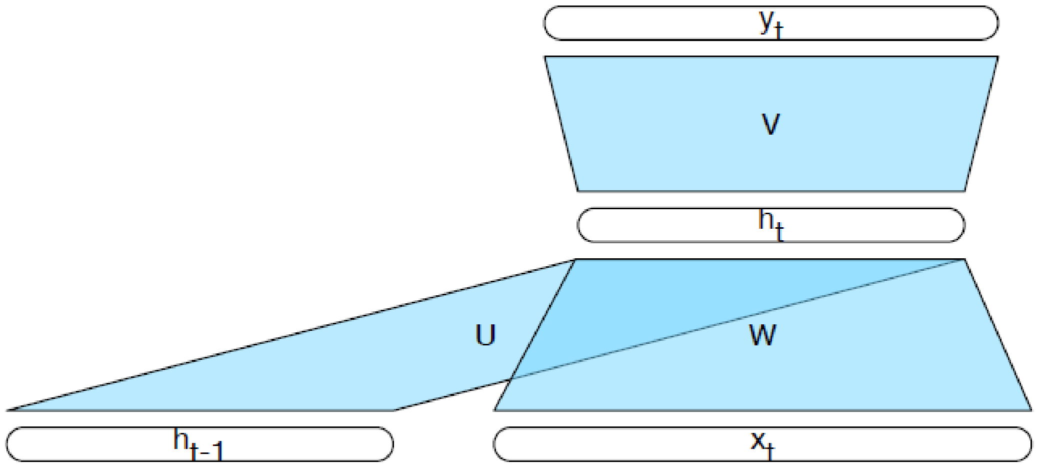
- Previous timestep's hidden layer ≈ memory, context
- Encodes earlier processing steps
- Helps make future decisions
- Key: doesn't limit length of prior context
  - Context "remembered" in the previous hidden layer includes info all the way back to the beginning of the sequence

SIMPLE RNN



- Given
  - Input vector (x_t)
  - Values for h from previous time step
- Still standard feedforward

SIMPLE RNN



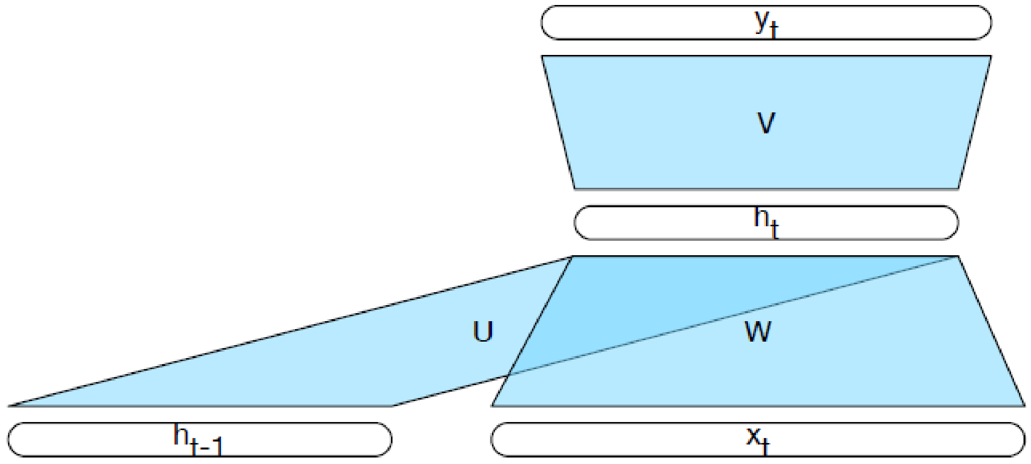
- Given
  - Input vector (x_t)
  - Values for *h* from previous time step
- Still standard feedforward
- Key change: new set of weights *U*
  - Connects previous timestep hidden layer (h_{t-1}) to current hidden layer (h_t)
  - Weights *U* determine how network should use past context to calculate output for current input

INFERENCE

- Forward inference: map sequence of inputs to sequence of outputs
- Similar to feedforward network
- Output y_t = input x_t + activation value for hidden layer h_t

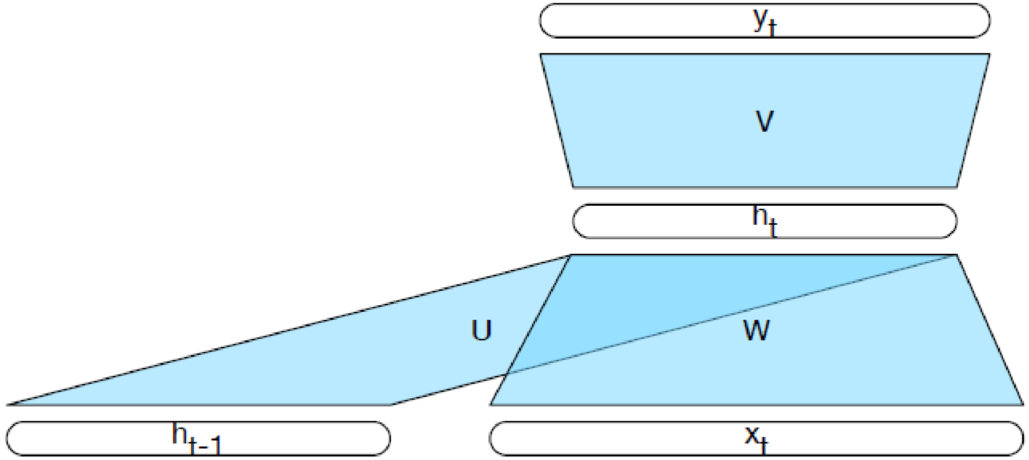
INFERENCE

- $h_t = g\{(x_t * W) + (h_{t-1} * U)\}$
- $y_t = softmax(h_t * V)$



INFERENCE

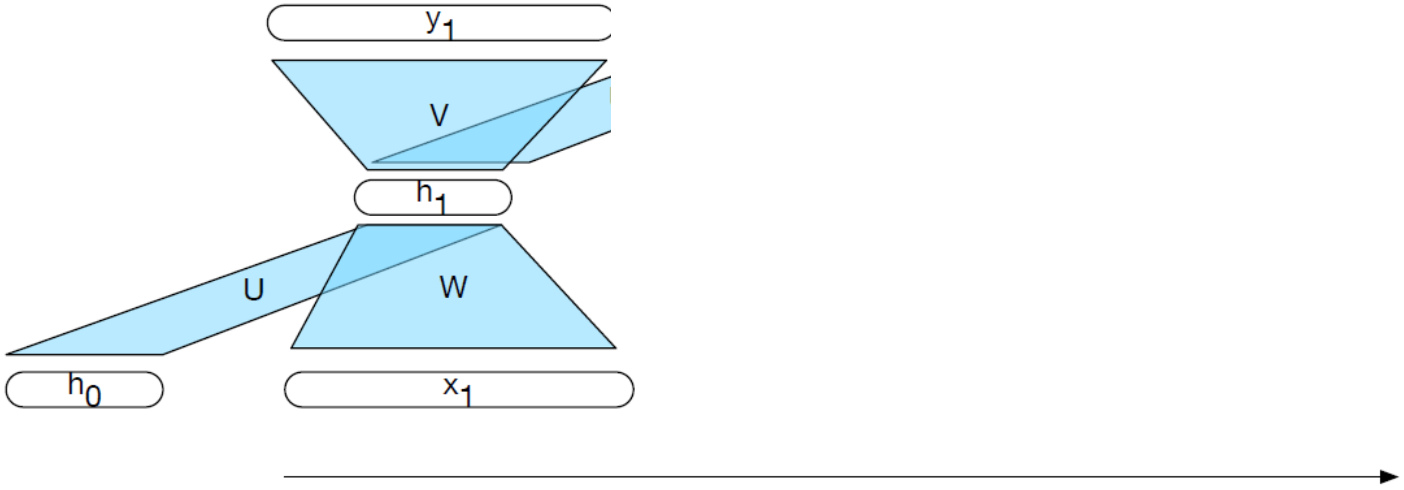
- $h_t = g\{(x_t * W) + (h_{t-1} * U)\}$
- $y_t = softmax(h_t * V)$



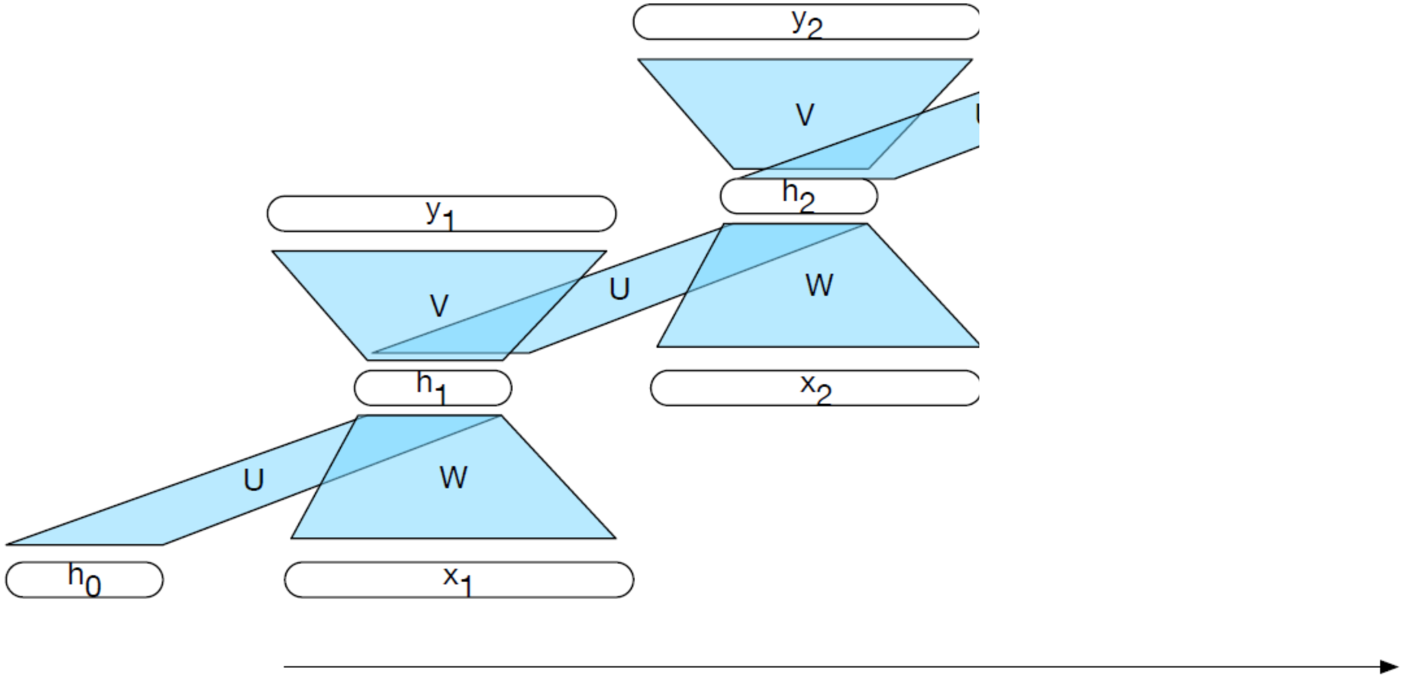
function FORWARDRNN(x, *network*) **returns** output sequence y

$h_0 \leftarrow 0$
for $i \leftarrow 1$ **to** LENGTH(x) **do**
  $h_i \leftarrow g(U\ h_{i-1} + W\ x_i)$
  $y_i \leftarrow f(V\ \ h_i)$
return y

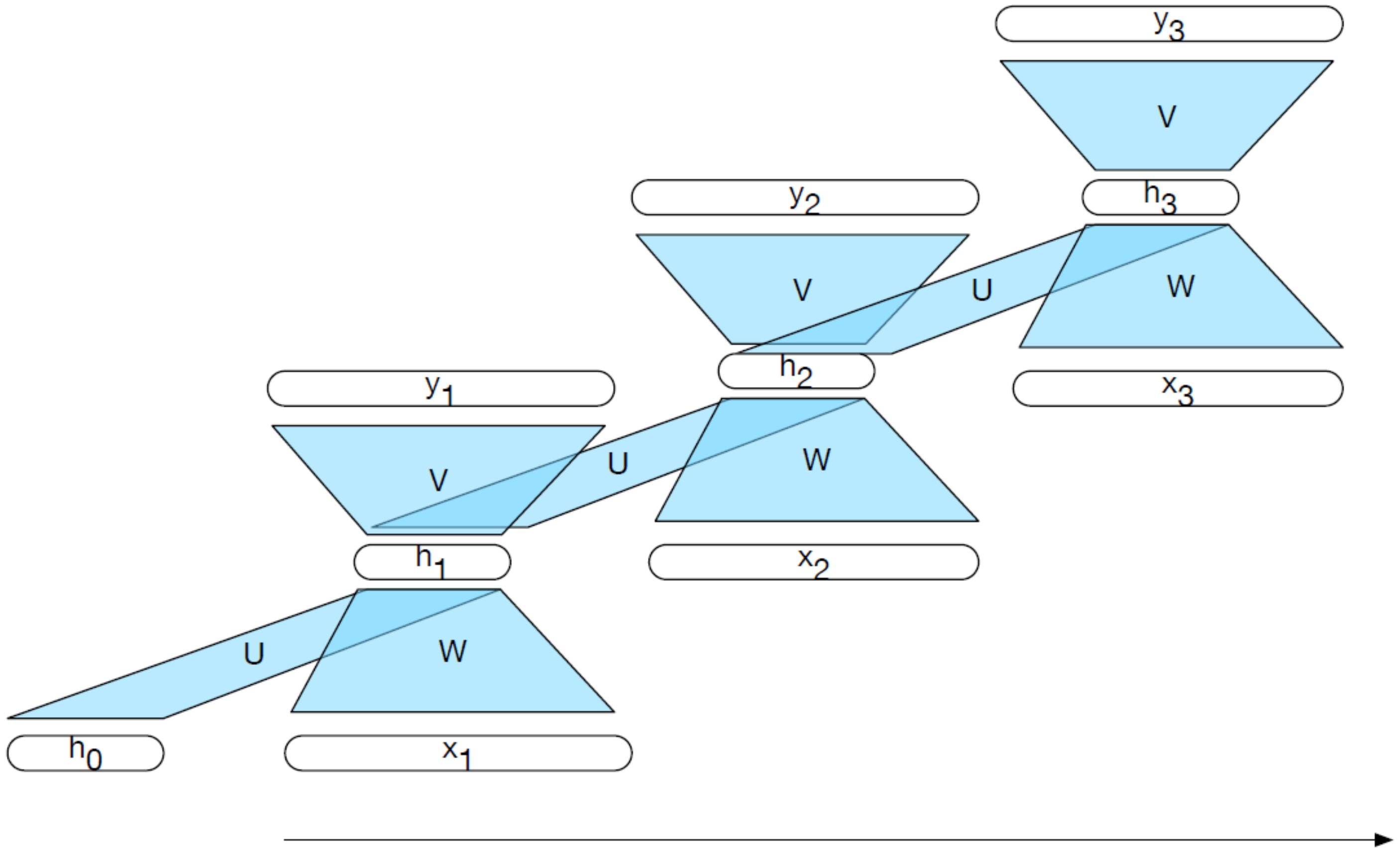
INFERENCE



INFERENCE



INFERENCE



TRAINING

- Just like feedforward networks
  - Training set
  - Loss function
  - Backprop

TRAINING

- Just like feedforward networks
  - Training set
  - Loss function
  - Backprop
- Weights
  - W: from input layer to hidden layer
  - U: from previous hidden layer to current hidden layer
  - V: from hidden layer to output layer

BACKPROP

- 2 new concerns
  - (1) To calculate loss for output at time *t* we need hidden layer from *t-1*

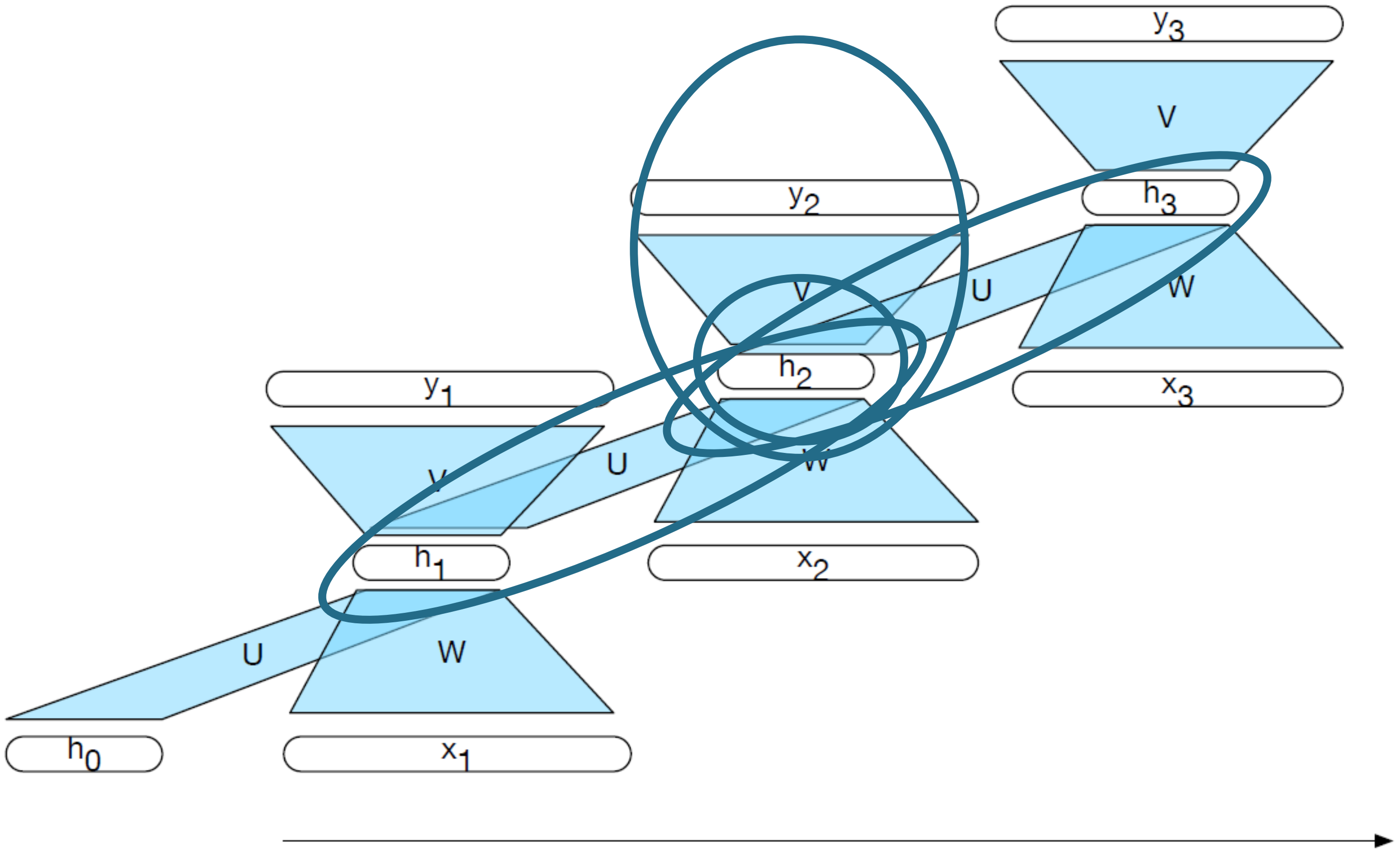
BACKPROP

- 2 new concerns
  - (1) To calculate loss for output at time *t* we need hidden layer from *t-1*
  - (2) Hidden layer at time *t* influences output at time *t* & hidden layer at time *t+1*

BACKPROP

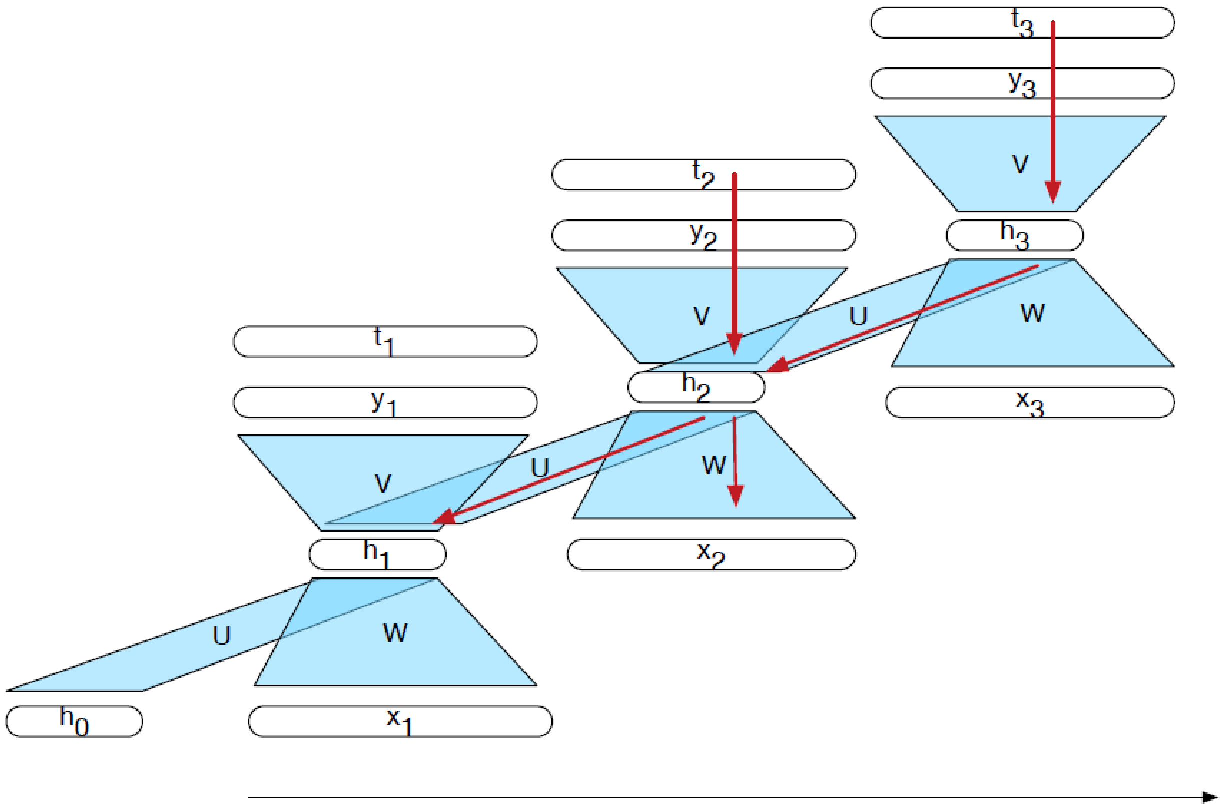
- 2 new concerns
  - (1) To calculate loss for output at time t we need hidden layer from $t-1$
  - (2) Hidden layer at time t influences output at time t & hidden layer at time $t+1$
    - So it also influences loss at $t+1$
    - To calculate error accruing in h_t we need to know influence on current output & those that follow

BACKPROP



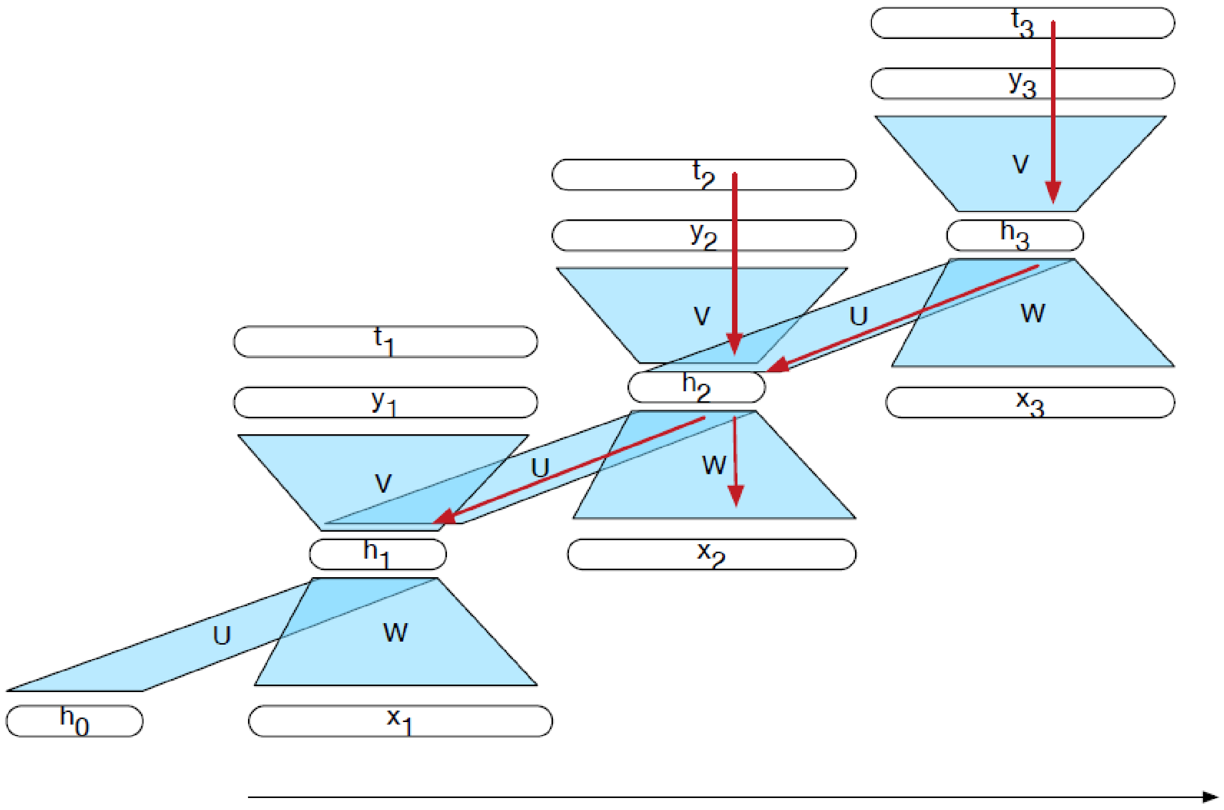
BACKPROP THROUGH TIME

- Input/output example pair at $t = 2$
- What do we need to compute gradients to update U, V, & W?



V UPDATE

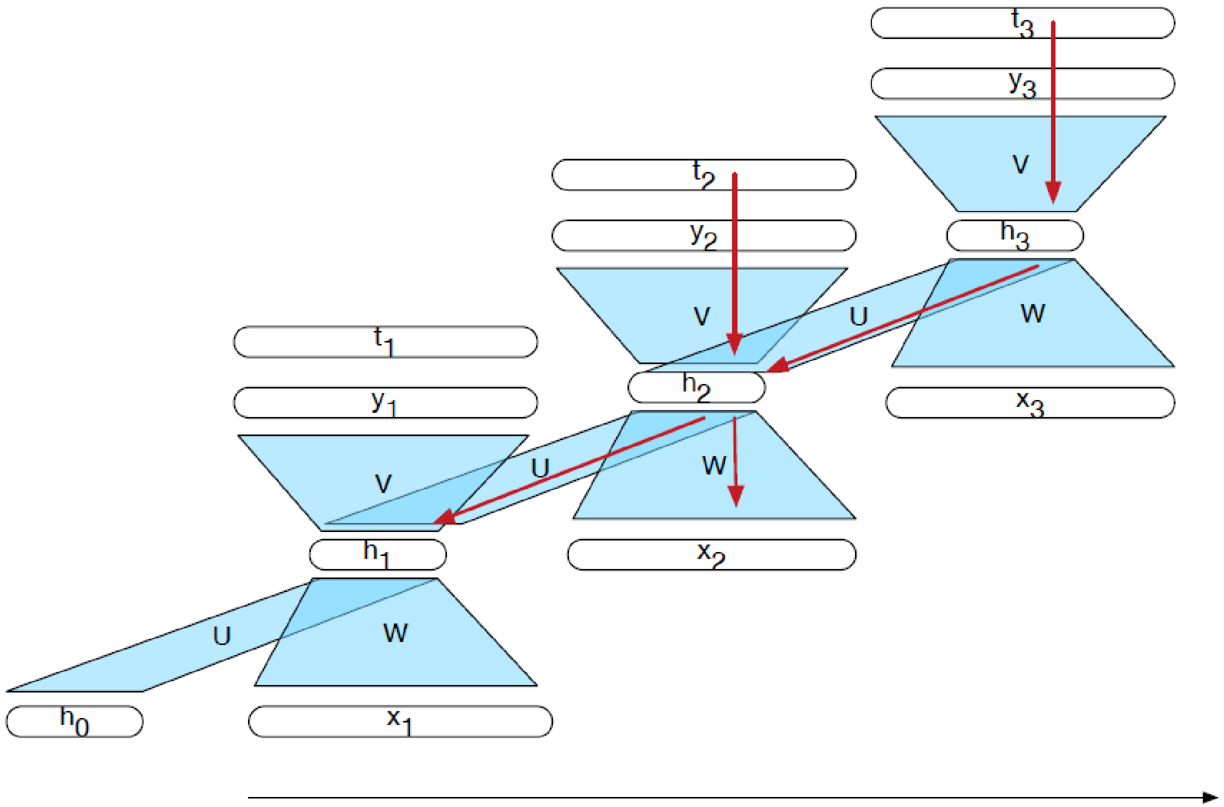
- Gradients to update V
- Same as in FFN
- Derivative of loss w.r.t. weights V



V UPDATE

- Gradients to update V
- Same as in FFN
- Derivative of loss w.r.t. weights *V*

$$\boxed{\frac{\partial L}{\partial V}} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial V}$$

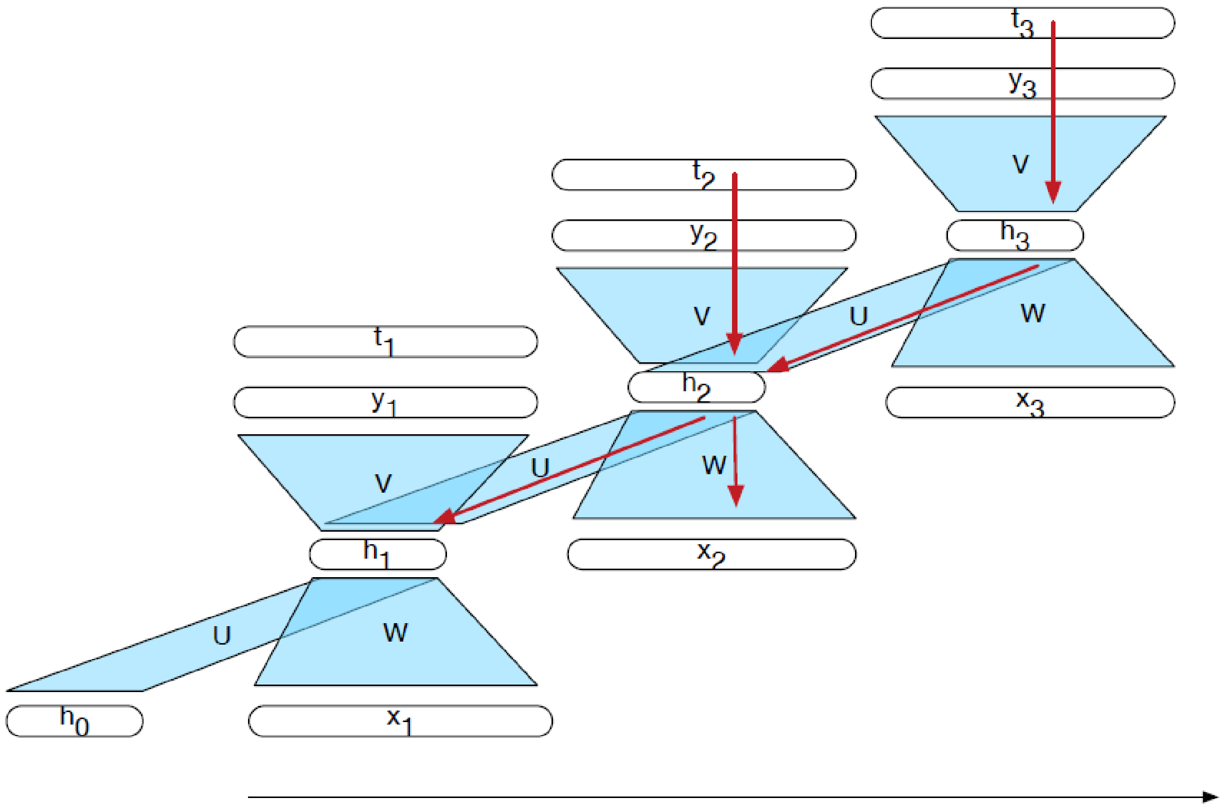


V UPDATE

- Gradients to update V
- Same as in FFN
- Derivative of loss w.r.t. weights *V*

$$\frac{\partial L}{\partial V} = \boxed{\frac{\partial L}{\partial a}} \frac{\partial a}{\partial z} \frac{\partial z}{\partial V}$$

Derivative of loss function w.r.t network output *a*

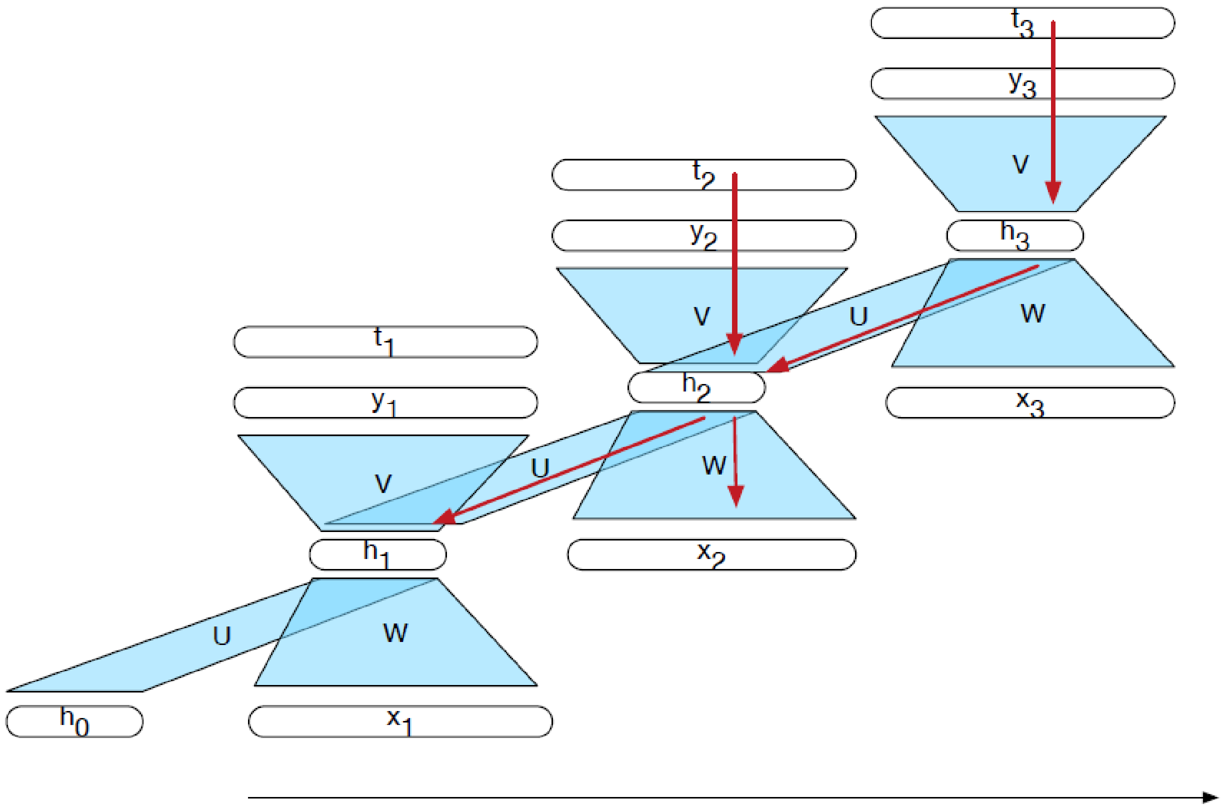


V UPDATE

- Gradients to update V
- Same as in FFN
- Derivative of loss w.r.t. weights V

$$\frac{\partial L}{\partial V} = \frac{\partial L}{\partial a}\boxed{\frac{\partial a}{\partial z}}\frac{\partial z}{\partial V}$$

Derivative of network output *a* w.r.t. intermediate activation *z*

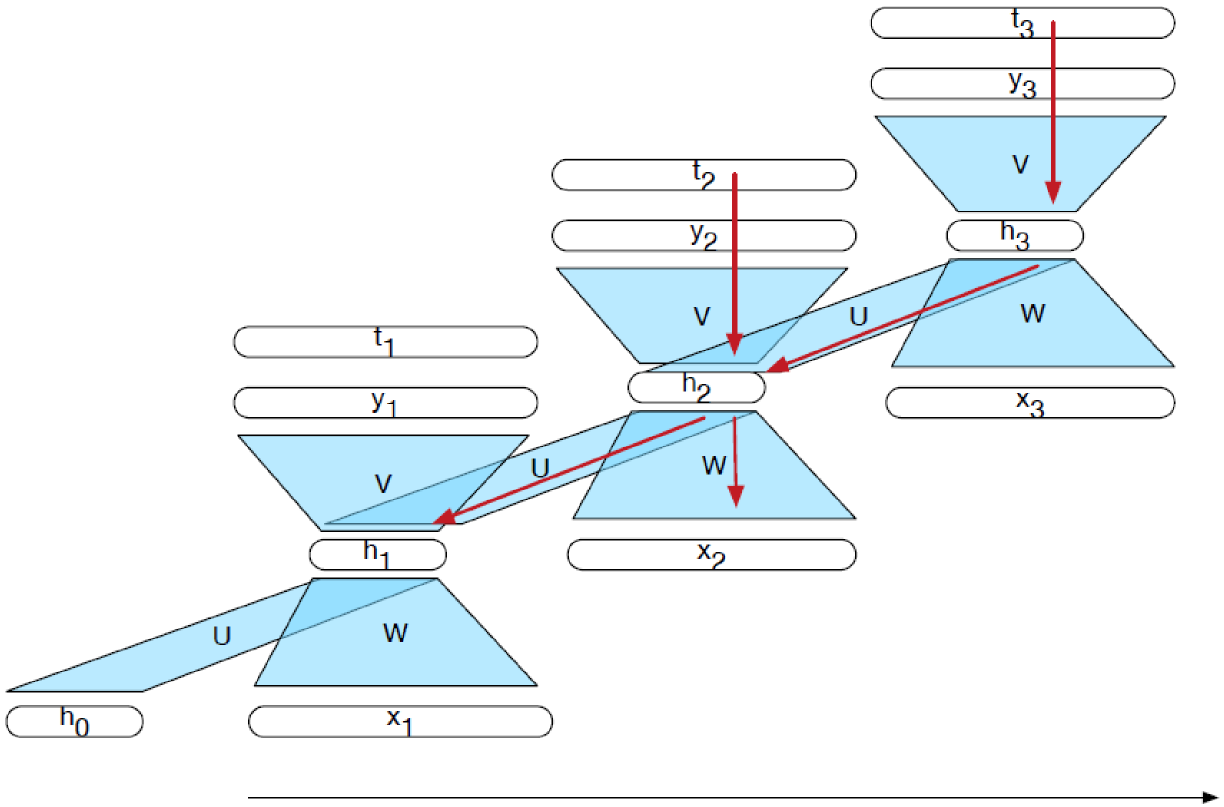


V UPDATE

- Gradients to update V
- Same as in FFN
- Derivative of loss w.r.t. weights V

$$\frac{\partial L}{\partial V} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\boxed{\frac{\partial z}{\partial V}}$$

Derivative of intermediate activation w.r.t weights V



V UPDATE

- δ_{out}: error term that represents how much loss is associated with each of the units in output layer

$$\frac{\partial L}{\partial V} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial V}$$

- Final gradient needed to update *V*

$$\frac{\partial L}{\partial V} = \delta_{out} h_t$$

U & W UPDATE

- Difference from FFN: computing *W* & *U*
- h_t contributes to output & error at time *t* & *t+1*
- So δ_h must include error from both timesteps

$$\delta_h \;=\; g'(z)V\,\delta_{out} + \delta_{next}$$

U & W UPDATE

- Difference from FFN: computing *W* & *U*
- h_t contributes to output & error at time *t* & *t+1*
- So δ_h must include error from both timesteps

$$\frac{\partial L}{\partial W} = \delta_h x_t$$

$$\frac{\partial L}{\partial U} = \delta_h h_{t-1}$$

U & W UPDATE

- Difference from FFN: computing W & U
- h_t contributes to output & error at time t & $t+1$
- So δ_h must include error from both timesteps

$$\frac{\partial L}{\partial W} = \delta_h x_t$$

$$\frac{\partial L}{\partial U} = \delta_h h_{t-1}$$

- Compute error: "assign proportional blame"
  - Backprop δ_h to previous h_{t-1}
  - Proportional based on U

$$\delta_{next} = g'(z) U \delta_h$$

TRAINING SUMMARY

- Backpropagation Through Time
- First pass
  - Do forward inference: compute h_t & y_t
  - Accumulate loss at each step
  - Save value of h_t at each step to use in next timestep

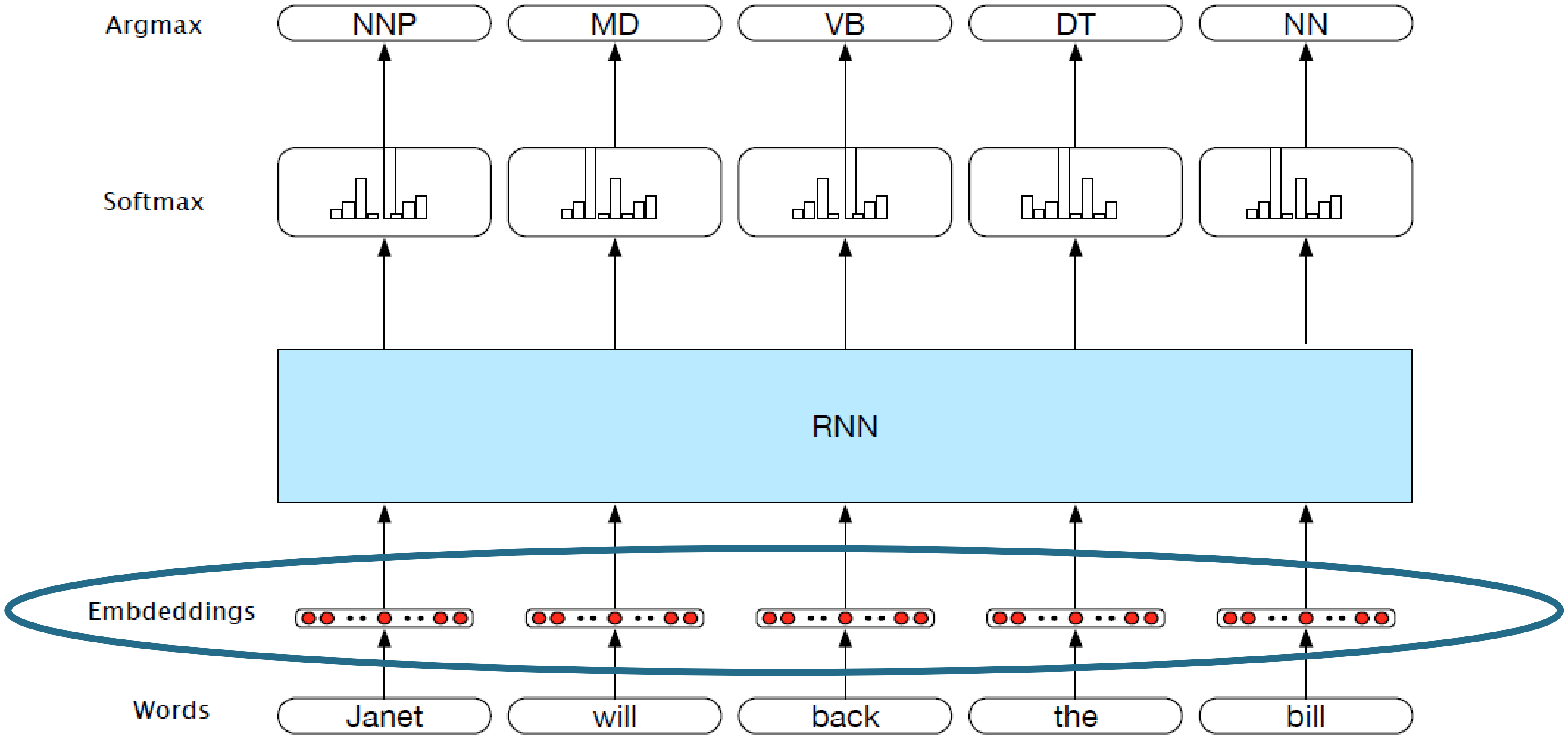
TRAINING SUMMARY

- Backpropagation Through Time
- First pass
  - Do forward inference: compute h_t & y_t
  - Accumulate loss at each step in time
  - Save value of h_t at each step to use in next timestep
- Second pass
  - Process sequence in reverse
  - Compute required error term gradients
  - Compute & save error term for use in hidden layer for each backward step in time

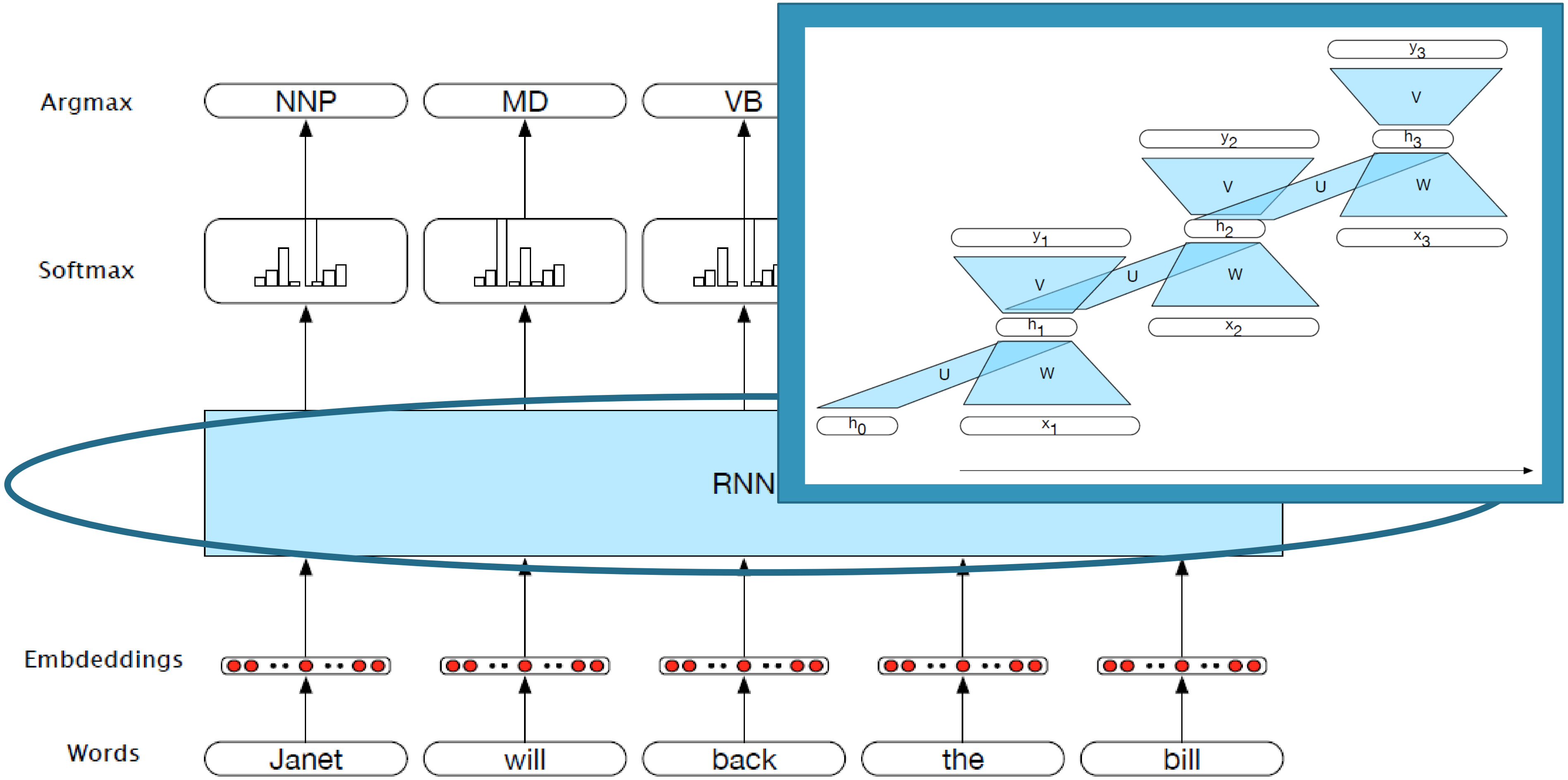
RNN APPLICATIONS

- Effective for
  - Language modeling
  - Sequence labeling tasks (e.g., POS tagging)
  - Sequence classification tasks (sentiment analysis, topic classification)
- Basis for sequence-to-sequence approaches
  - Summarization
  - Machine Translation
  - Question Answering

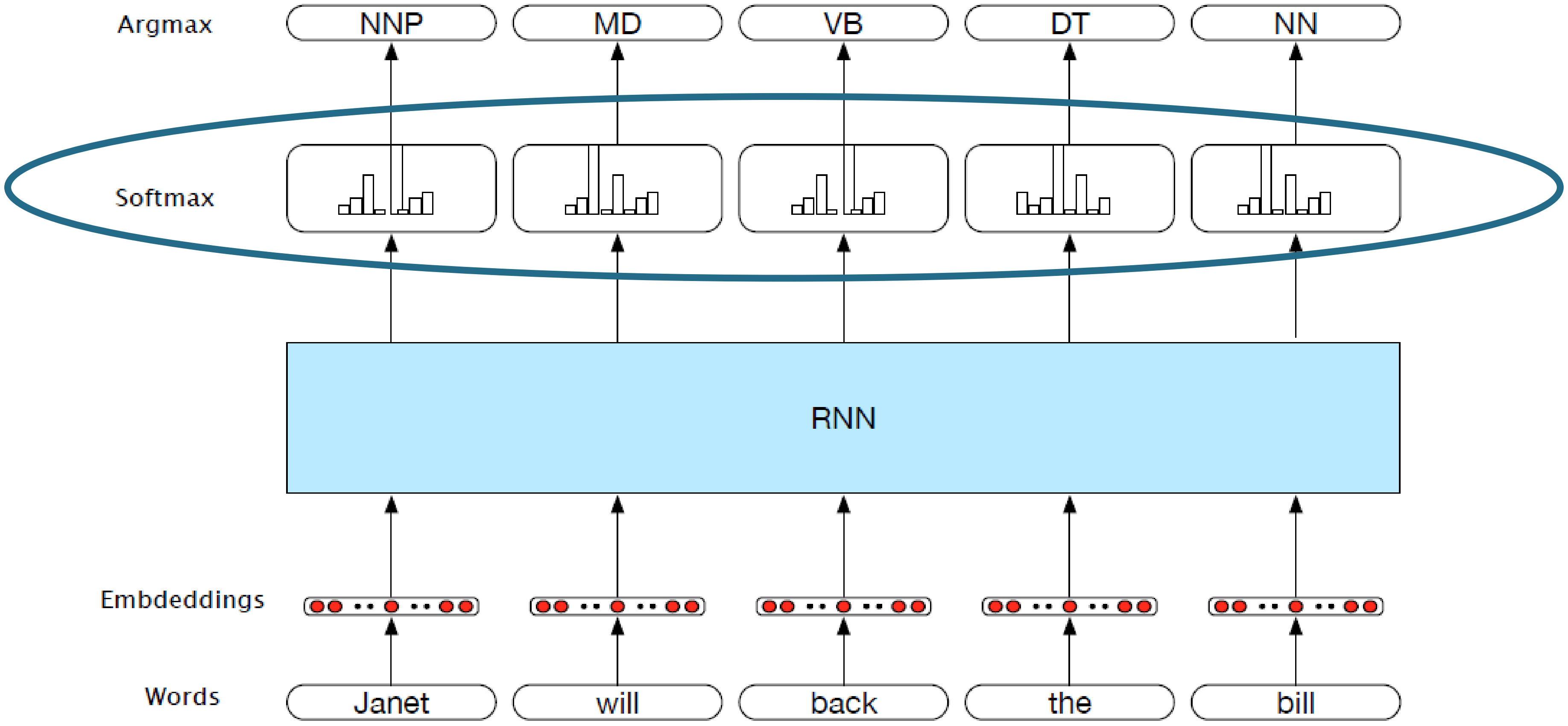
SEQUENCE LABELING



SEQUENCE LABELING

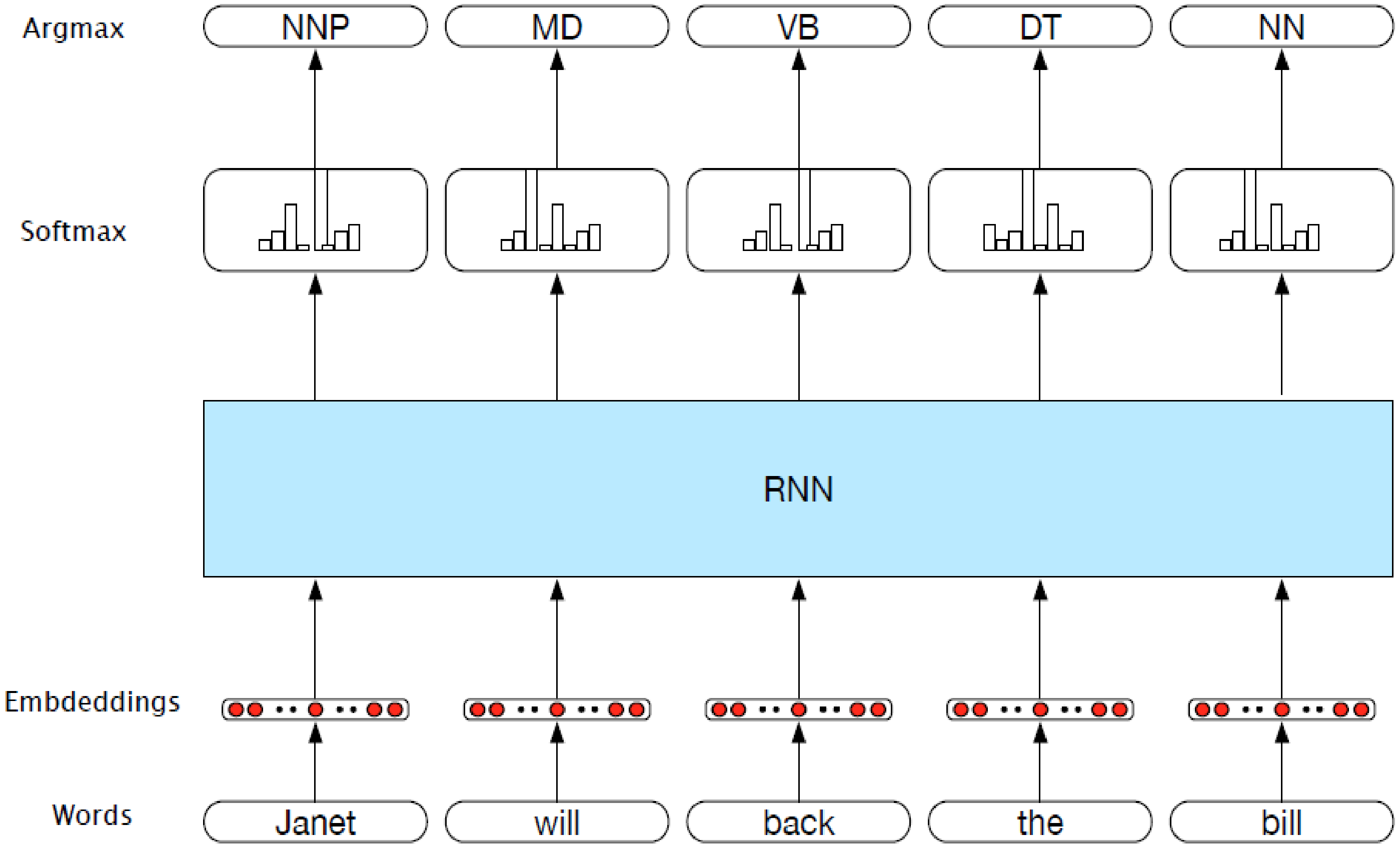


SEQUENCE LABELING



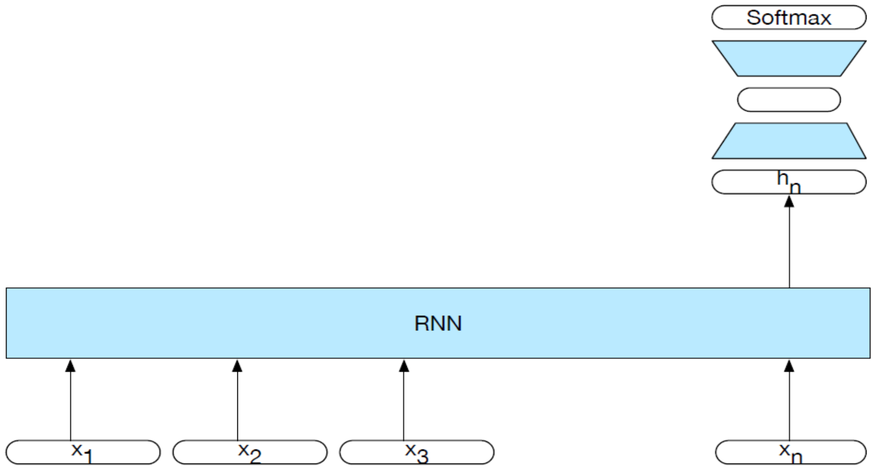
SEQUENCE LABELING

- To generate a tag sequence for a given input
  - Run forward inference over input sequence
  - Select most likely tag from *softmax* at each step



DEEP NEURAL NETWORK

- Deep NN = simple RNN with feedforward classifier
  - Stacked RNNs
  - Bidirectional RNNs
- End-to-end training: uses loss from downstream apps to adjust weights all the way throughout the network

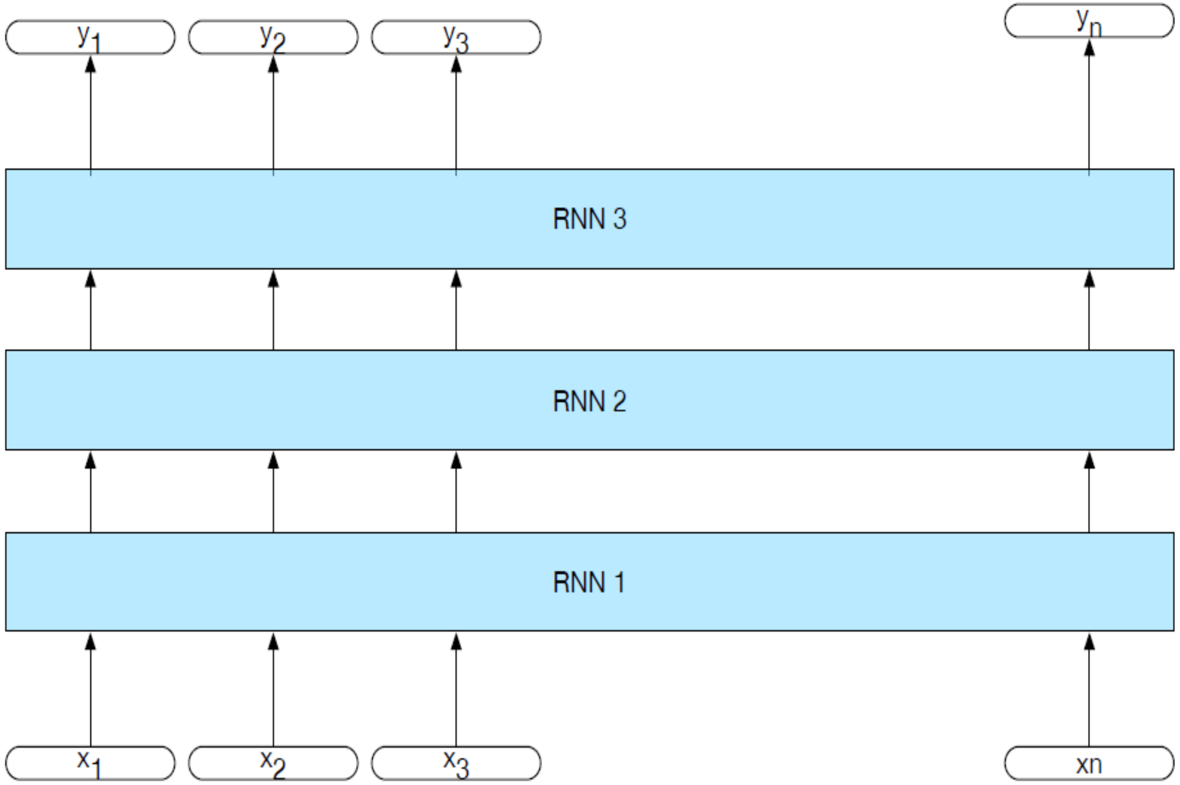


DEEP NEURAL NETWORK

- No intermediate outputs for words in sequence preceding last element → no loss terms associated with those
- Loss function based entirely on final text classification task
  - *Softmax* output (from FFN) + cross-entropy loss → training
  - Classification error is backpropagated through all aspects of FFN: weights in FF classifier → input → RNN 3 matrices (U, V, W)

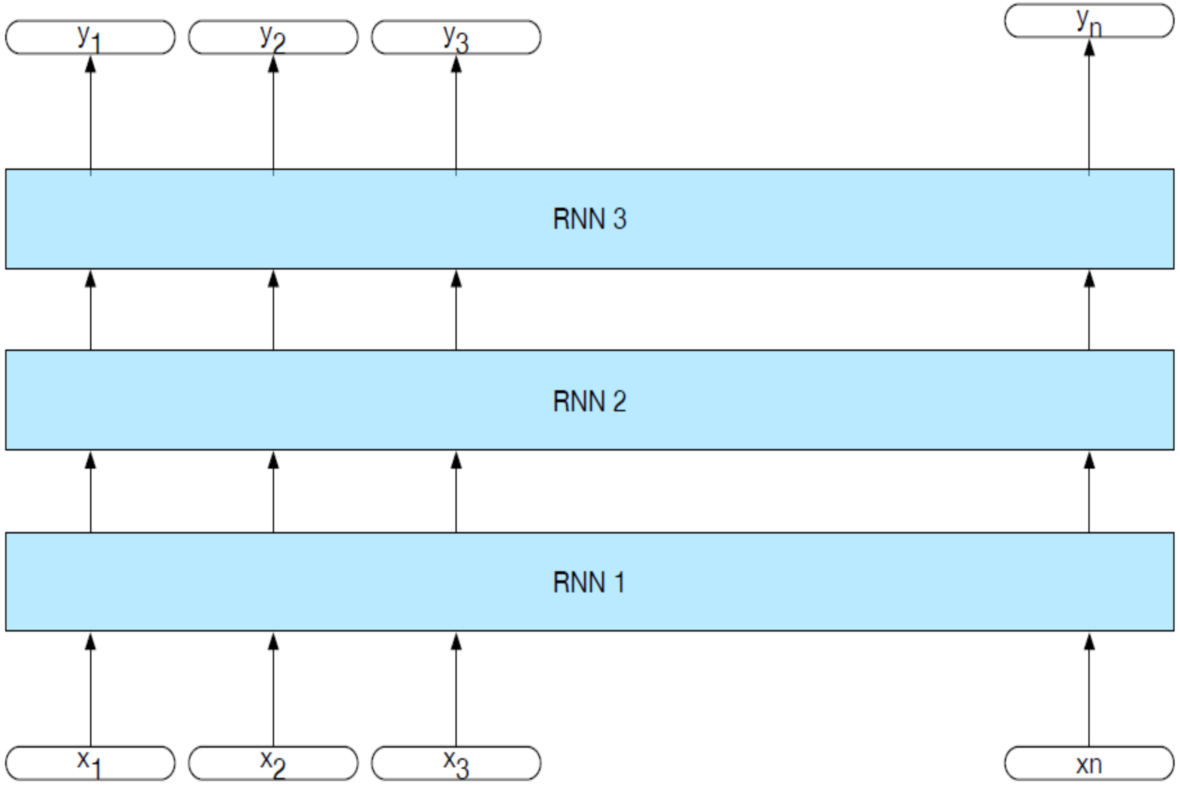
STACKED RNN

- Up until now:
  - RNN input = word sequences or embeddings (vectors)
  - RNN output = vectors for predicting words, tags, sequence labels
- Why not use entire sequence of outputs from 1 RNN as input sequence to another?



STACKED RNN

- Up until now:
  - RNN input = word sequences or embeddings (vectors)
  - RNN output = vectors for predicting words, tags, sequence labels
- Why not use entire sequence of outputs from 1 RNN as input sequence to another?
- Stacked RNN: multiple networks where output layer of 1 layer serves as input to subsequent layer
  - # of stacks task & training set specific
  - # of stacks rises, training costs rises
  - Induces representations at differing levels of abstractions across layers



BIDIRECTIONAL RNN (BI-RNN)

- In simple RNN hidden state at time *t* represents everything network knows about sequence up to that point
- Think of it as context to the **left** of the current time

$$h_t^f = RNN_{forward}(x_1^t)$$

BIDIRECTIONAL RNN (BI-RNN)

- In simple RNN hidden state at time *t* represents everything network knows about sequence up to that point
- Think of it as context to the **left** of the current time

$$h_t^f = RNN_{forward}(x_1^t)$$

- If we have access to entire input sequence at once, use context to the right too
- To grab it, we train an RNN on input sequence *in reverse*

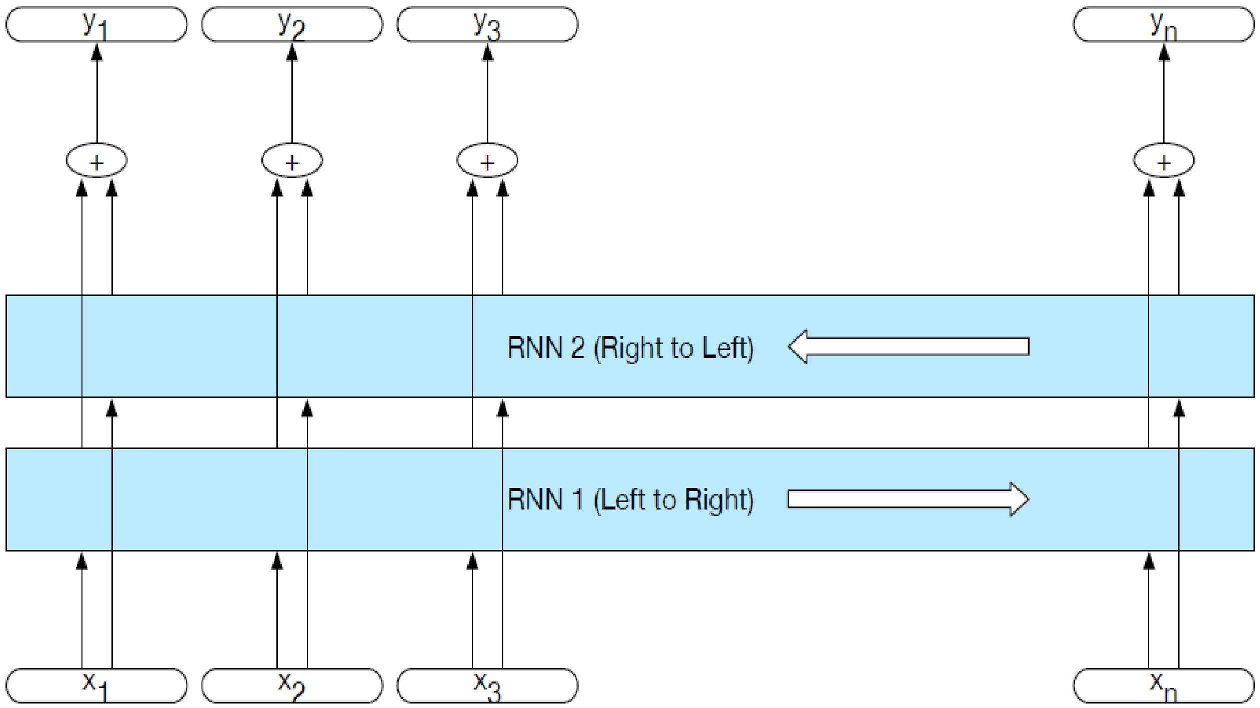
$$h_t^b = RNN_{backward}(x_t^n)$$

BI-RNN

- Bidirectional RNN = forward information + backward
- 2 independent RNNs
  - Input processed start to end
  - Input processed end to start
- Output combined into single representation that captures left & right contexts of input at each point in time

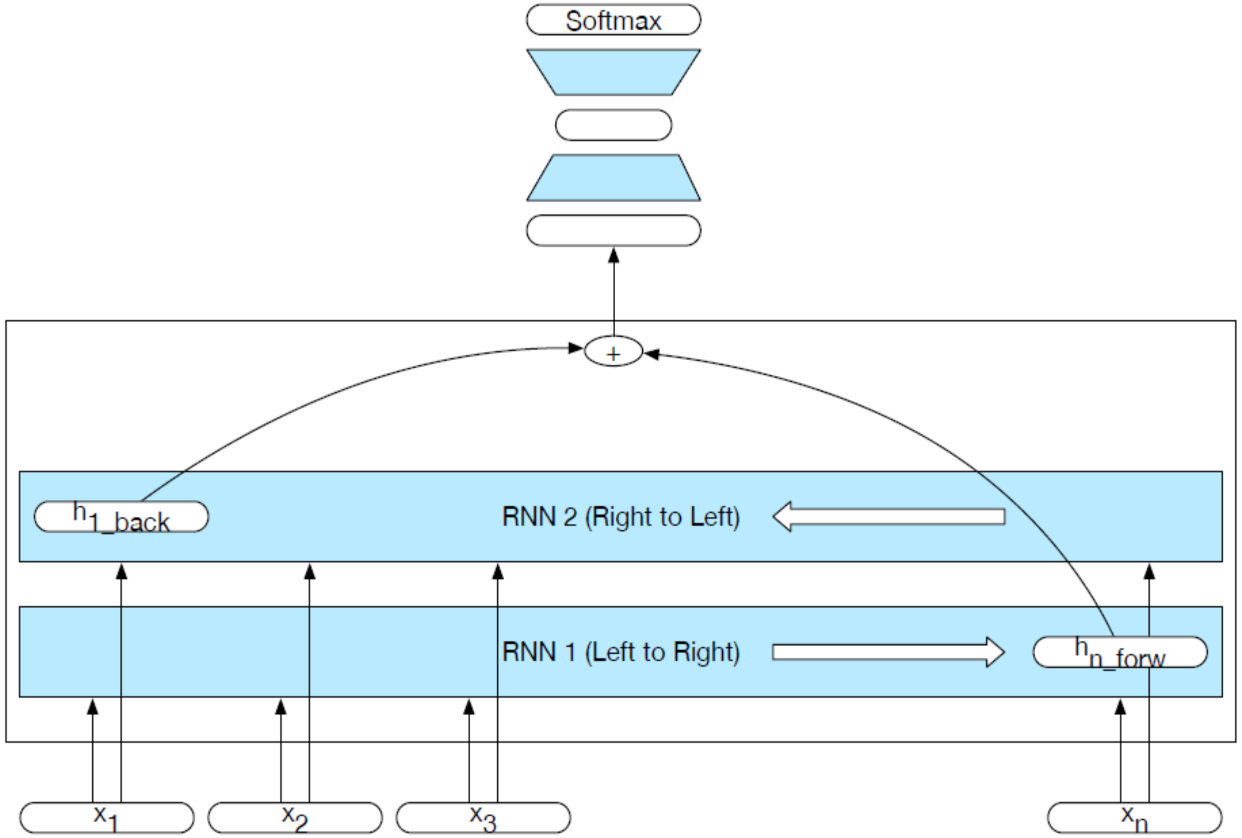
BI-RNN

- Outputs of forward & backward pass are concatenated
- Output at each time step captures info to the left & right of current input
- Example: use concatenated output as basis for local labeling decision in sequence labeling apps



BI-RNN

- Highly effective for sequence classification
- Final state naturally reflects more info about end of sentence than its beginning
  - Previous attempt: input final RNN's hidden sate to FF classifier
- Bi-RNN solution: combine final forward & backward hidden states & use as input



MANAGING CONTEXT

- Difficult to train RNNs for tasks that need information far away from current position in processing
- Have access to entire preceding sequence
- But info encoded in hidden states is usually *local*
  i.e., more relevant to most recent parts of input sequence & recent decisions

MANAGING CONTEXT

- Usually though distant information is important for NLP applications
- LM example: *The flights the airline was cancelling were full.*
  - P(was|…airline): makes sense because verb matches
  - P(were|…airline): trickier because *flights* further away & singular *airline* is closer
- Ideally network should be able to retain distant info until needed while processing intermediate parts of sequence correctly

MANAGING CONTEXT

- RNNs have trouble carrying distant information forward
  - Hidden layers (& weights that determine their value) are asked to handle 2 tasks simultaneously
    - Provide useful info for current decision
    - Update & carry forward info for future decisions

MANAGING CONTEXT

- RNNs have trouble carrying distant information forward
  - Hidden layers (& weights that determine their value) are asked to handle 2 tasks simultaneously
    - Provide useful info for current decision
    - Update & carry forward info for future decisions
  - Need to backprop error through time
    - h_t contributes to loss at $t+1$
    - During backward pass, hidden layers are multiplied repeatedly
    - Result: vanishing gradient problem

MANAGING CONTEXT

- RNNs have trouble carrying distant information forward
  - Hidden layers (& weights that determine their value) are asked to handle 2 tasks simultaneously
    - Provide useful info for current decision
    - Update & carry forward info for future decisions
  - Need to backprop error through time
    - h_t contributes to loss at $t+1$
    - During backward pass, hidden layers are multiplied repeatedly
    - Result: vanishing gradient problem
- Need a way to forget info we don't need anymore & remember info we'll need in the future

LONG SHORT-TERM MEMORY (LSTM)

- Divide context management problem into 2 subproblems
  - Remove info that's no longer needed
  - Add info likely to be needed for later decision making
- Key to solving both: learn how to manage context instead of hard-coding it into architecture

LONG SHORT-TERM MEMORY (LSTM)

- (1) Add explicit context layer to architecture
- (2) Use specialized neural units with gates to control info flow through the units in layers
  - Implemented through additional weights that operate sequentially on input & previous hidden & context layers

LSTM GATES

- Design
  - Feedforward layer
  - Sigmoid activation function
  - Pointwise multiplication with layer being gated
- Forget gate
- Add gate
- Output gate

LSTM GATES

- Design
  - Feedforward layer
  - Sigmoid activation function
  - Pointwise multiplication with layer being gated
- Forget gate
- Add gate
- Output gate

Sigmoid because it pushes output to 0 or 1

LSTM GATES

- Design
  - Feedforward layer
  - Sigmoid activation function
  - Pointwise multiplication with layer being gated
- Forget gate
- Add gate
- Output gate

Sigmoid + PM ≈ binary mask
Values align near 1 pass; lower erased

LSTM GATES

- Forget gate: Deletes info from context that's no longer needed
  - Computes weighted sum of h_{t-1} + current input
  - Passes that value through sigmoid → mask
  - Multiply that mask by context vector → removes info
- Computation step
- Add gate
- Output gate

LSTM GATES

- Forget gate: Deletes info from context that's no longer needed
- Computation step: computes info needed from previous hidden state & current inputs using *tanh*
- Add gate
- Output gate

LSTM GATES

- Forget gate: Deletes info from context that's no longer needed
- Computation step: computes info needed
- Add gate: selects information to add to current context
  - Computes weighted sum of previous hidden layer & current input
  - Passes that value through sigmoid → mask
  - Adds mask to context vector → new context vector with new info
- Output gate

LSTM GATES

- Forget gate: Deletes info from context that's no longer needed
- Computation step: computes info needed
- Add gate: selects information to add to current context
- Output gate: decides what info is needed for current hidden state

GATED RECURRENT UNIT (GRU)

- LSTM requires learning 8 weights
  - U & W for each of the 4 gates within each unit
- GRUs
  - Drop separate context vector
  - Reduce number of gates to 2
    - Reset gate (r)
    - Update gate (z)

GRU GATES

- Like LSTM: uses sigmoid to create binary-like mask
  - Blocks info if values near zero
  - Allows info to pass through unchanged if values near 1

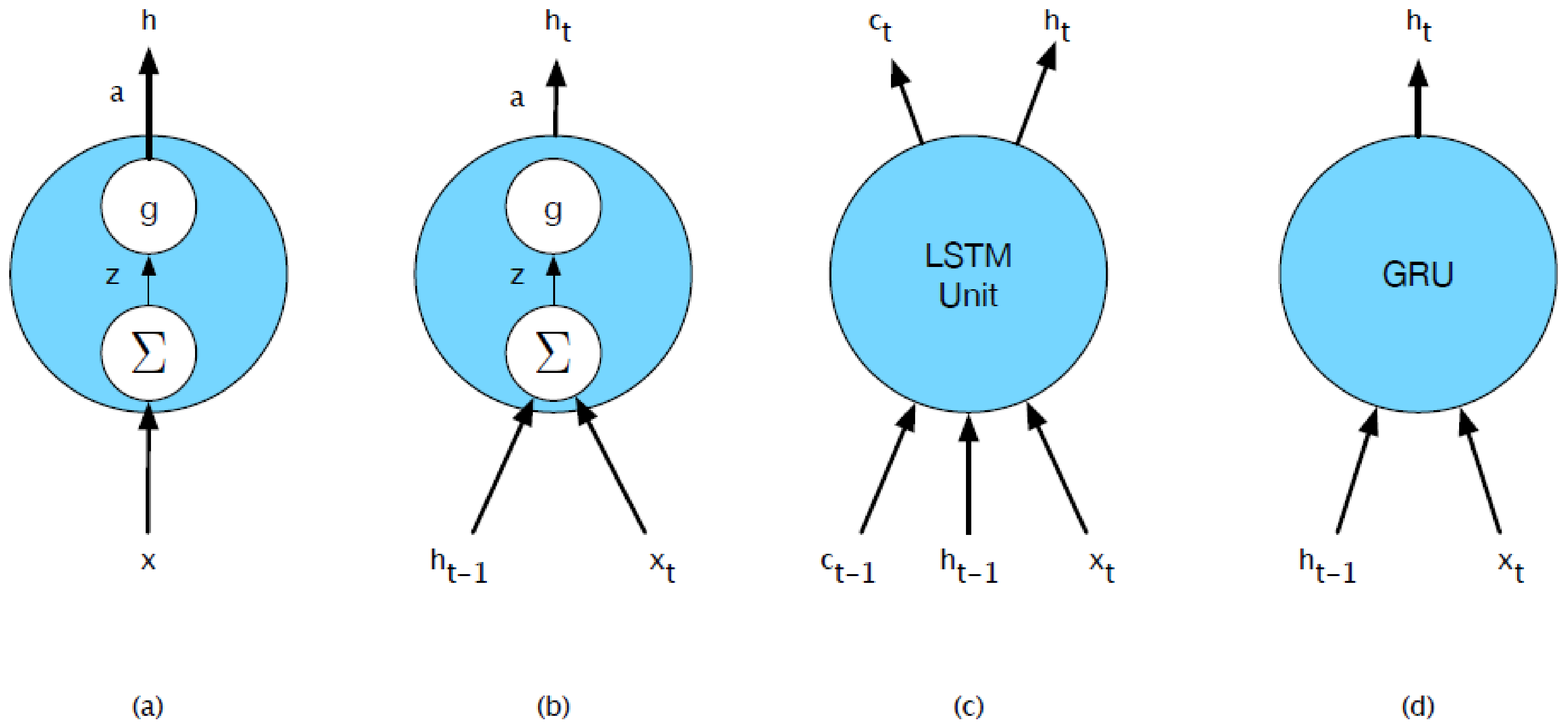
GRU GATES

- Like LSTM: uses sigmoid to create binary-like mask
  - Blocks info if values near zero
  - Allows info to pass through unchanged if values near 1
- Reset gate
  - Decides which aspects of previous hidden state are relevant to current context (or should be ignored)
  - Computes mask to get intermediate new hidden state

GRU GATES

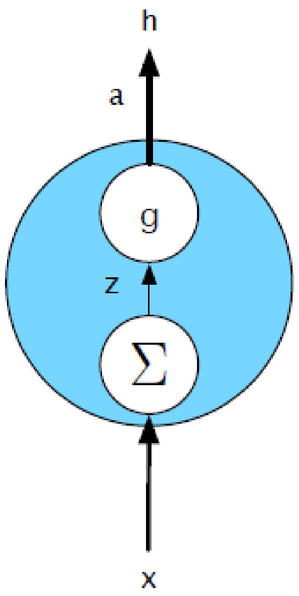
- Like LSTM: uses sigmoid to create binary-like mask
  - Blocks info if values near zero
  - Allows info to pass through unchanged if values near 1
- Reset gate
  - Decides which aspects of previous hidden state are relevant to current context (or should be ignored)
  - Computes mask to get intermediate new hidden state
- Update gate
  - Decides which aspects of new state will be used directly in new hidden state
  - Decides which aspects of previous state are preserved for future use

MODULARITY



Feedforward Simple RNN

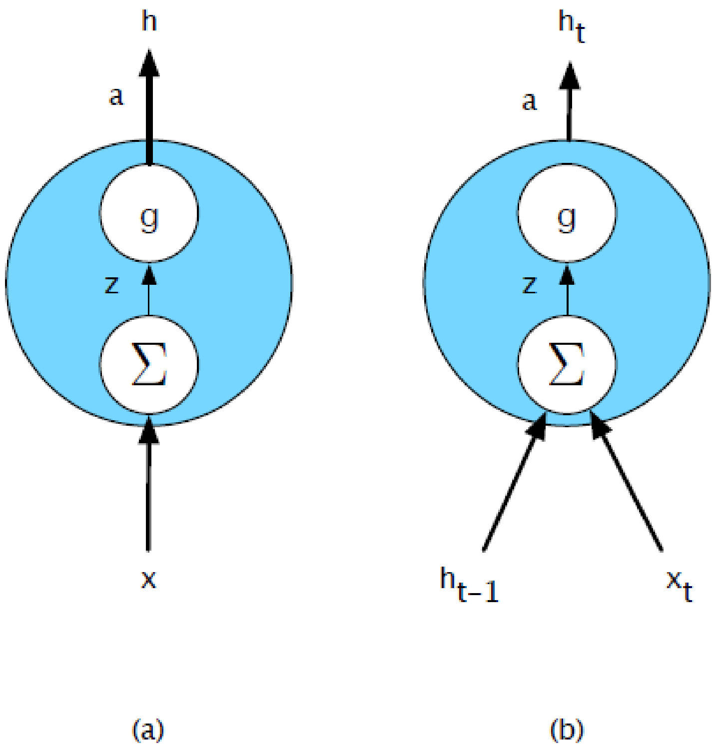
MODULARITY



(a)

Feedforward

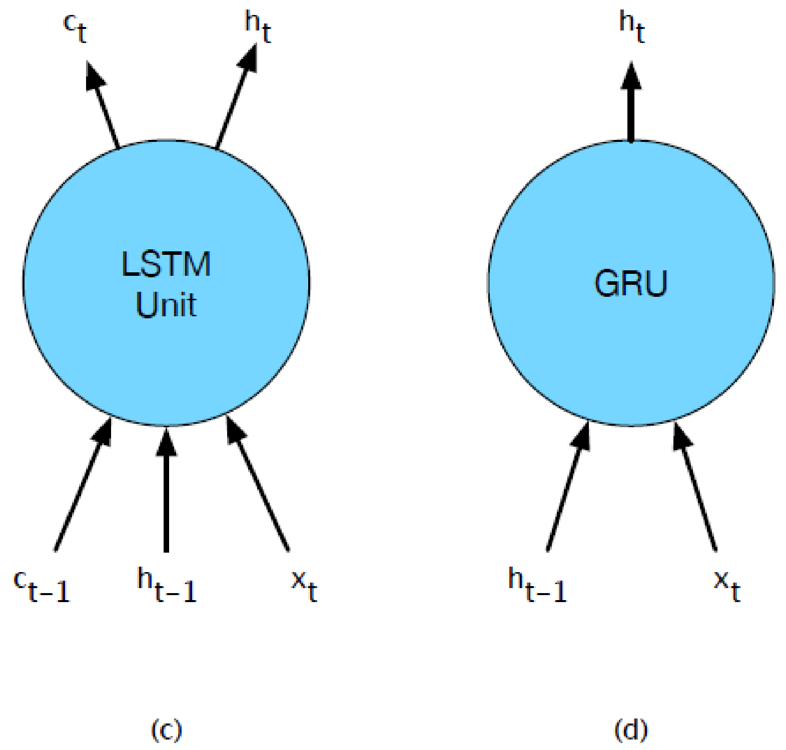
MODULARITY



Feedforward Simple RNN

MODULARITY

- Complexity encapsulated within neural unit
- Modularity allows LSTM & GRU units to be used in other network architectures
- Multi-layer networks using gated units can be unrolled into deep feedforward networks



SUMMARY

- Simple RNN
  - Inference
  - Training
- Applications
  - RNLM
  - Sequence labeling
    - POS tagging
    - NER
  - Sequence Classification
- Deep Networks
  - Stacked RNNs
  - Bidirectional RNNs
- Managing Context
  - LSTMs
  - GRUs