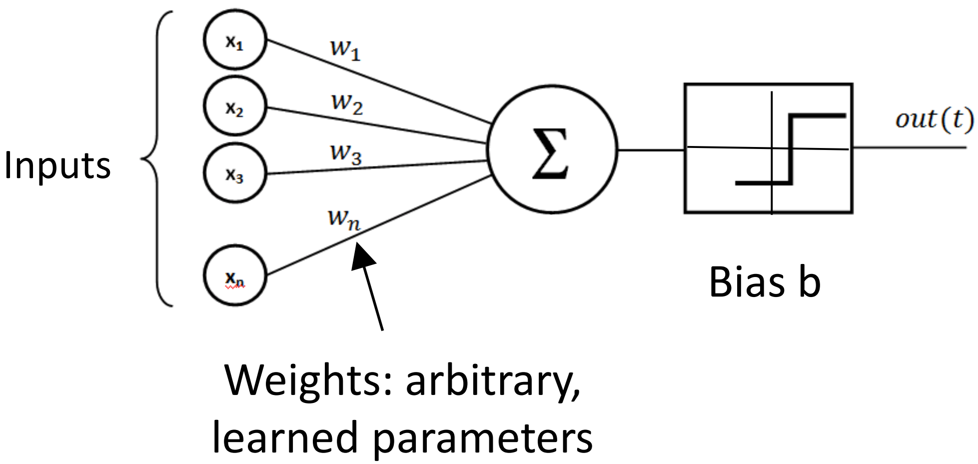
Neural Network

Sudeshna Sarkar

8 AUG 2019

Simple Artificial Neuron Model
(Linear Threshold Unit)



$$net_j = \sum_i w_{ji} o_i$$

$$o_j = \begin{cases} 0 \text{ if } net_j < T_j \\ 1 \text{ if } net_i \geq T_j \end{cases}$$

(T_j is threshold for unit j)

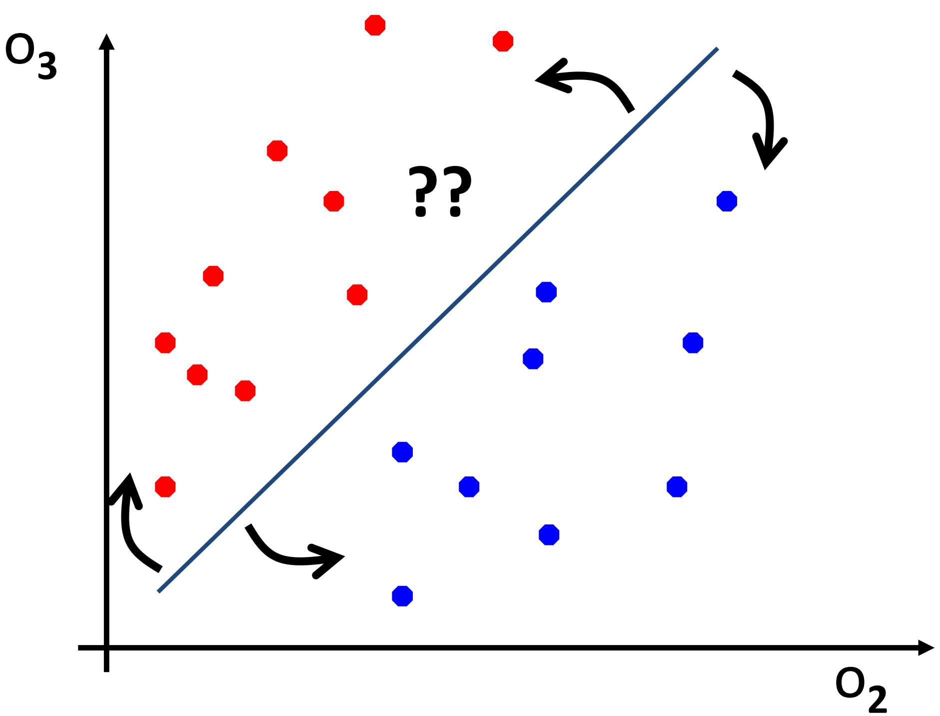
Update weights by: $w_{ji} = w_{ji} + \eta(t_j - o_j)o_i$

where η is the "learning rate"

Adjust threshold $T_j = T_j - \eta(t_j - o_j)$

Perceptron as a Linear Separator

- Since perceptron uses linear threshold function, it is searching for a linear separator that discriminates the classes.



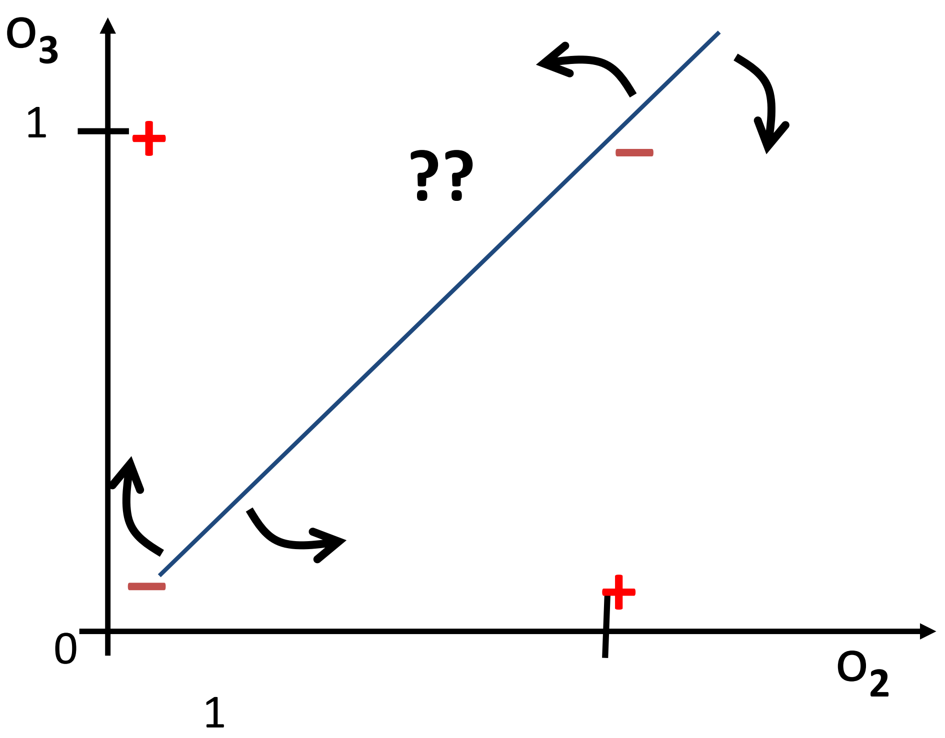
$$w_{12}o_2 + w_{13}o_3 > T_1$$

$$o_3 > -\frac{w_{12}}{w_{13}} o_2 + \frac{T_1}{w_{13}}$$

Or hyperplane in n-dimensional space

Concept Perceptron Cannot Learn

- Cannot learn exclusive-or, or parity function in general.

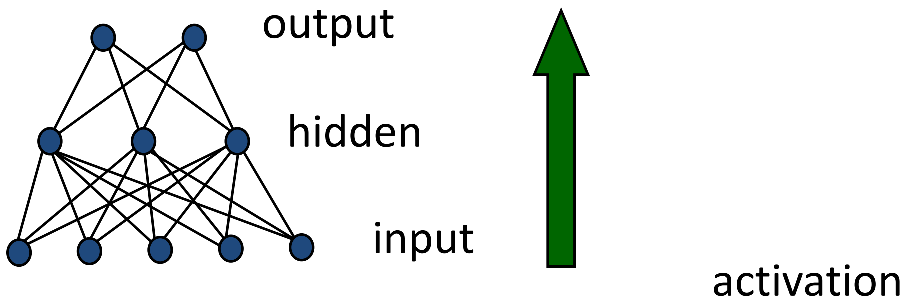


Perceptron Convergence

- **Perceptron convergence theorem**: If the data is linearly separable and therefore a set of weights exist that are consistent with the data, then the Perceptron algorithm will eventually converge to a consistent set of weights.

Multi-Layer Feed-Forward Networks

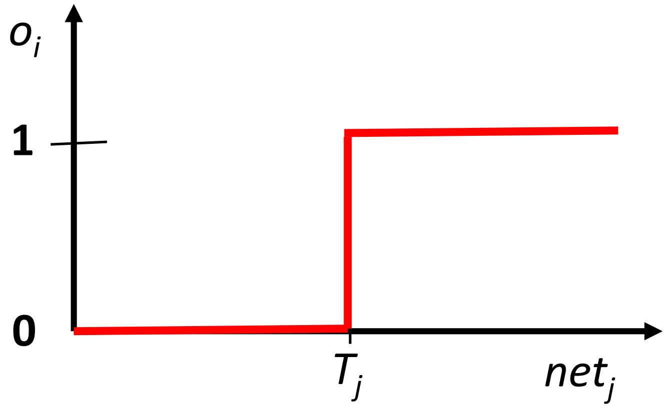
- Multi-layer networks can represent arbitrary functions.
- A typical multi-layer network consists of an input, hidden and output layer, each fully connected to the next, with activation feeding forward.



- The weights determine the function computed. Given an arbitrary number of hidden units, any boolean function can be computed with a single hidden layer.

Multi-Layer Nets

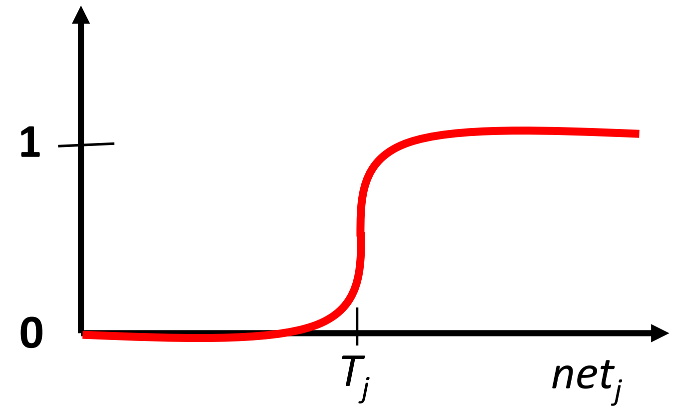
- To do gradient descent, we need the output of a unit to be a differentiable function of its input and weights.
- Standard linear threshold function is not differentiable at the threshold.



Differentiable Output Function

- Need non-linear output function to move beyond linear functions.
  - A multi-layer linear network is still linear.
- Standard solution is to use the non-linear, differentiable sigmoidal “logistic” function:

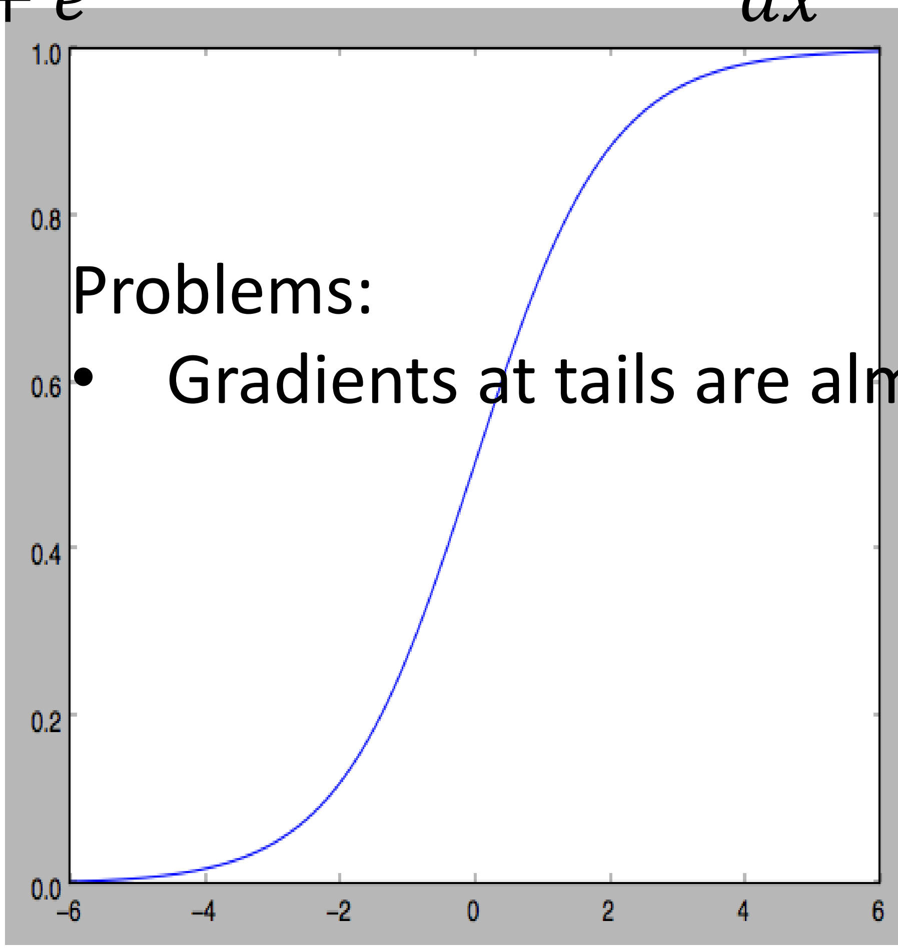
$$o_j = \frac{1}{1+e^{-(net_j - T_j)}}$$



Can also use tanh or Gaussian output function

Activation Functions: Sigmoid / Logistic

$$f(x) = \frac{1}{1+e^{-x}} \qquad \frac{df}{dx} = f(x)(1-f(x))$$

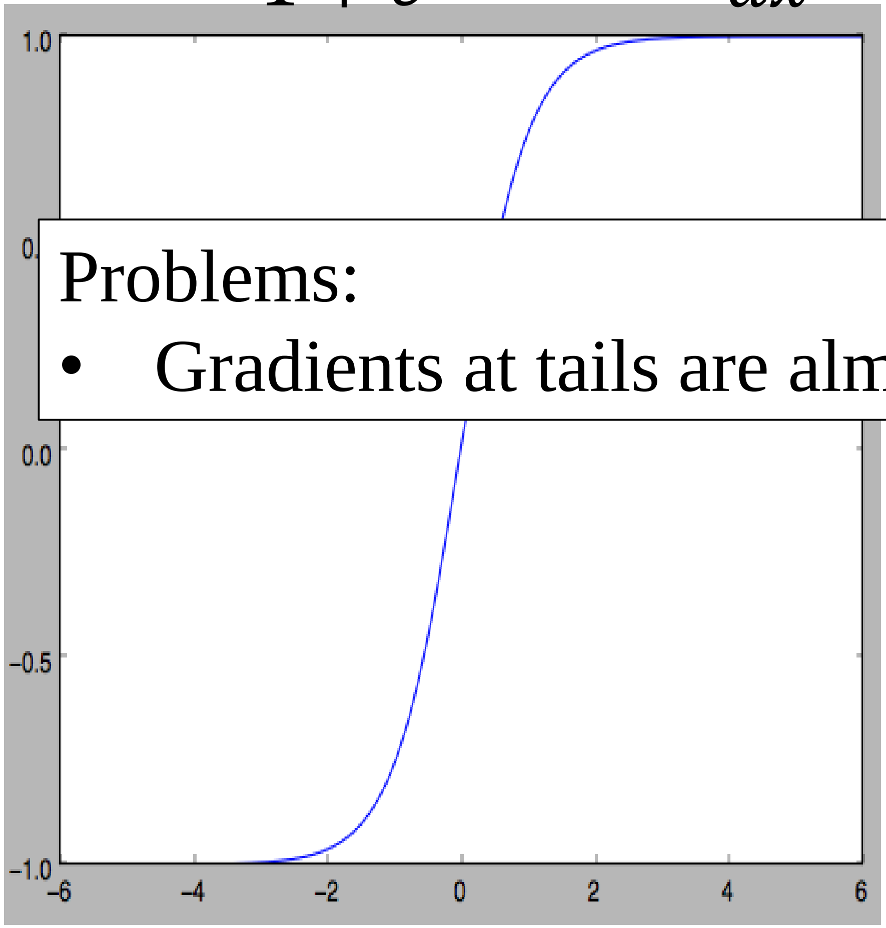


Problems:

- Gradients at tails are almost zero

Activation Functions: Tanh

$$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1 \quad \frac{df}{dx} = 1 - f(x)^2$$



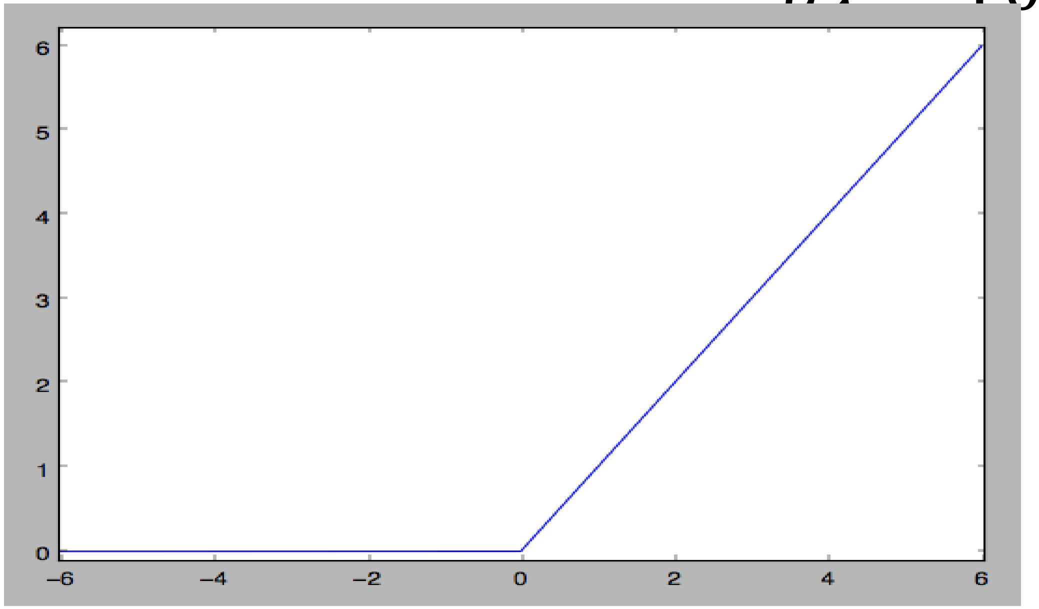
Problems:

- Gradients at tails are almost zero

Activation Functions:
ReLU (Rectified Linear Unit)

$$f(x) = \max(x, 0)$$

$$\frac{df}{dx} = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x < 0 \end{cases}$$



Universal Approximation Theorem

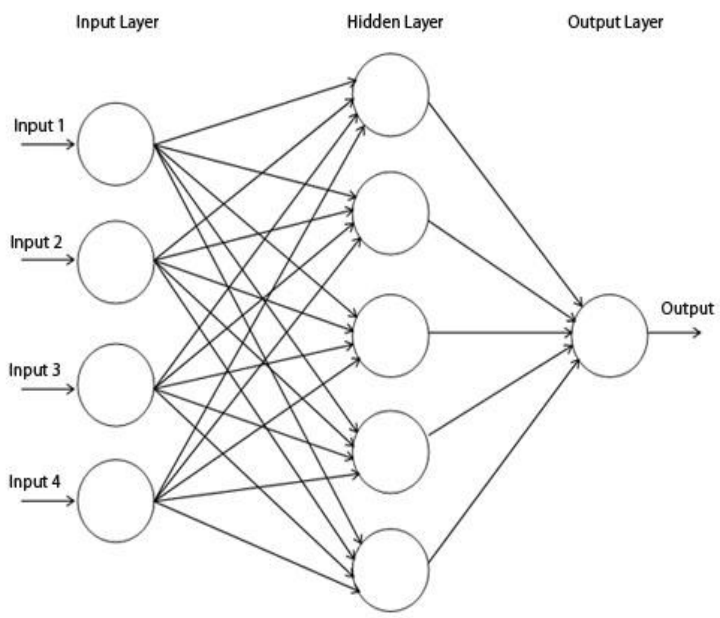
- Multilayer perceptron with a single hidden layer and linear output layer can approximate any continuous function on a compact subset of $\mathbb{R}^n$ to within any desired degree of accuracy.
- Assumes activation function is bounded, non-constant, monotonically increasing.
- Also applies for ReLU activation function.

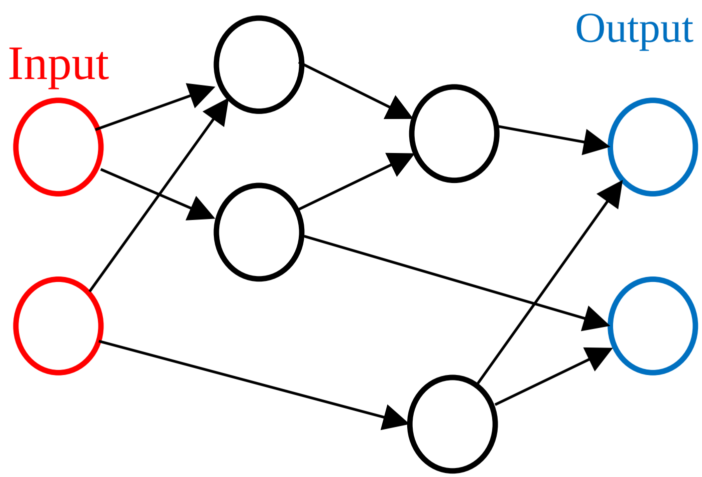
Why Use Deep Networks?

- Functions representable with a deep rectifier network can require an exponential number of hidden units with a shallow (one hidden layer) network
- Piecewise linear networks (e.g. using ReLU) can represent functions that have a number of regions exponential in depth of network.
  - Can capture repeating / mirroring / symmetric patterns in data.
  - Empirically, greater depth often results in better generalization.

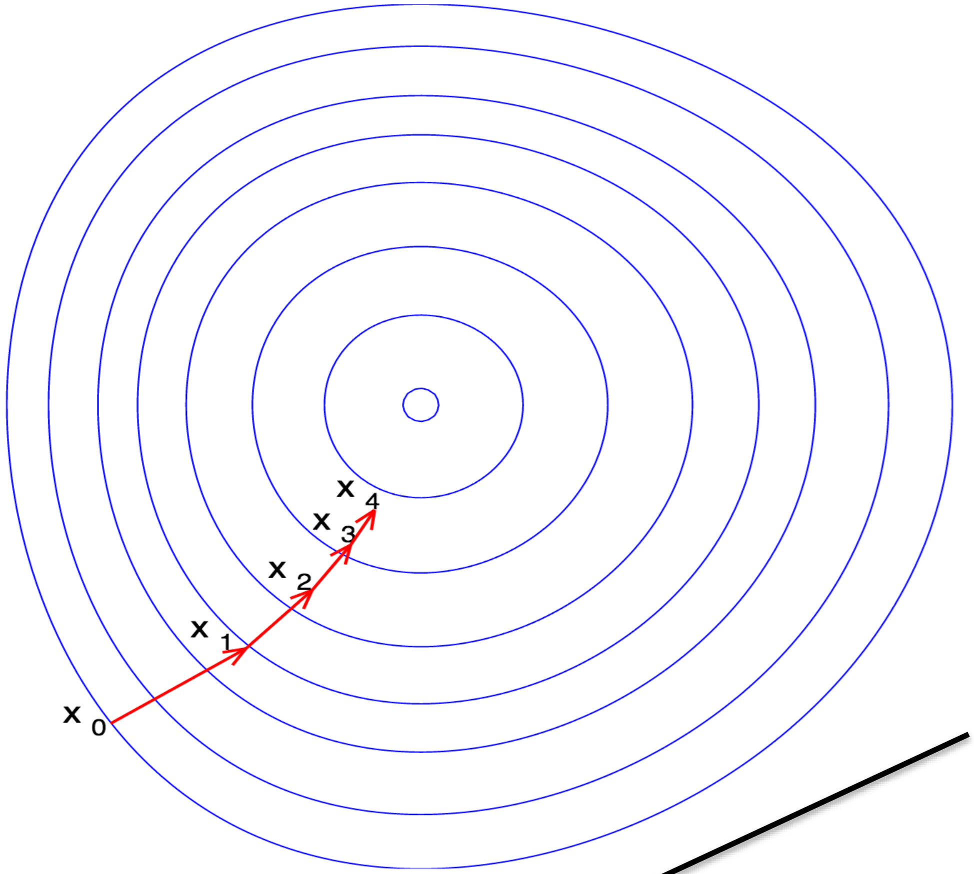
Neural Network Architecture

- **Fully connected:** A common connectivity pattern for multilayer perceptrons. All possible connections made between layers *i*-1 and *i*.





Gradient Descent



Step size
or
Learning rate

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n)$$

Gradient Descent

- Define objective to minimize error:

$$E(W) = \sum_{d \in D} \sum_{k \in K} (t_{kd} - o_{kd})^2$$

where D is the set of training examples, K is the set of output units, t_{kd} and o_{kd} are, respectively, the teacher and current output for unit k for example d.

- The derivative of a sigmoid unit with respect to net input is:

$$\frac{\partial o_j}{\partial net_j} = o_j(1 - o_j)$$

- Learning rule to change weights to minimize error is:

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}}$$

Stochastic gradient descent

- Gradient of sum of n terms where n is large
- Sample rather than computing the full sum
  - Sample size *s* is “mini-batch size”
  - Could be 1 (very noisy gradient estimate)
  - Could be 100 (collect photos 100 at a time to find each noisy “next” estimate for the gradient)
- Use same step as in gradient descent to the estimated gradient

Problem Statement

- Take the gradient of an arbitrary program or model (e.g. a neural network) with respect to the parameters in the model (e.g. weights).

Review: Chain Rule in One Dimension

- Suppose $f: \mathbb{R} \to \mathbb{R}$ and $g: \mathbb{R} \to \mathbb{R}$
- Define

$$h(x) = f(g(x))$$

- Then what is $h'(x) = dh/dx$?

$$h'(x) = f'\big(g(x)\big)g'(x)$$

Chain Rule in Multiple Dimensions

- Suppose $f: \mathbb{R}^m \to \mathbb{R}$ and $g: \mathbb{R}^n \to \mathbb{R}^m$, and $\mathbf{x} \in \mathbb{R}^n$
- Define

$$h(\mathbf{x}) = f(g_1(\mathbf{x}), \dots, g_m(\mathbf{x}))$$

- Then we can define partial derivatives using the multidimensional chain rule:

$$\frac{\partial f}{\partial x_i} = \sum_{l=1}^{m} \frac{\partial f}{\partial g_j} \frac{\partial g_j}{\partial x_i}$$

Solution for Simplified Chain of Dependencies

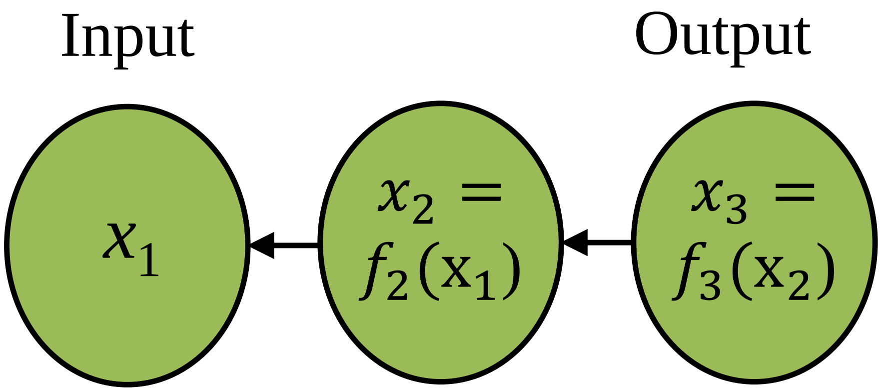
- Suppose $\pi(i) = i - 1$
- The computation:

  For $i = n + 1, \ldots, N$

  $$x_i = f_i(\mathrm{x}_{i-1})$$

- What is $\dfrac{dx_N}{dx_N}$?

For example:



$$\frac{dx_N}{dx_N} = 1$$

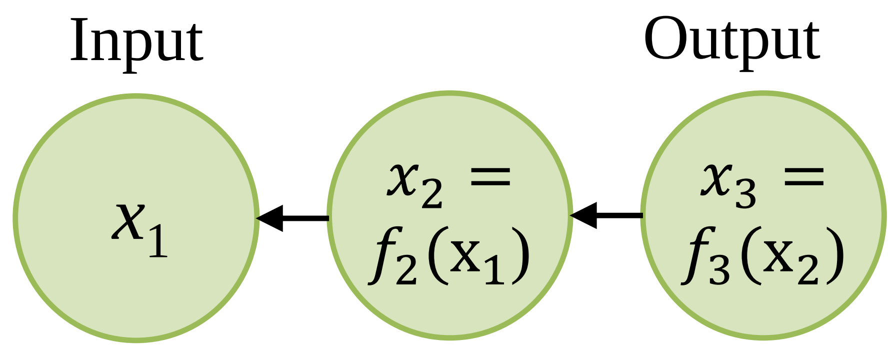
Solution for Simplified Chain of Dependencies

- Suppose $\pi(i) = i - 1$
- The computation:

For $i = n + 1, \ldots, N$

$$x_i = f_i(\mathrm{x}_{i-1})$$

For example:



What is $\dfrac{dx_N}{dx_i}$ in terms of $\dfrac{dx_N}{dx_{i+1}}$?

$$\frac{dx_N}{dx_i} = \frac{dx_N}{dx_{i+1}}\left(\frac{\partial x_{i+1}}{\partial x_i}\right)$$

$$dx_{i+1} = dx_i\left(\frac{\partial x_{i+1}}{\partial x_i}\right) \Rightarrow \frac{dx_{i+1}}{dx_N} = \frac{dx_i}{dx_N}\left(\frac{\partial x_{i+1}}{\partial x_i}\right)$$

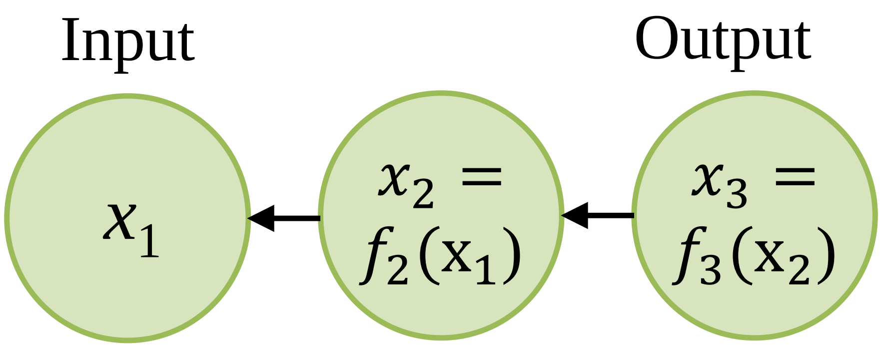
Solution for Simplified Chain of Dependencies

- Suppose $\pi(i) = i - 1$
- The computation:

For $i = n + 1, \ldots, N$

$$x_i = f_i(\mathrm{x}_{i-1})$$

For example:



What is $\dfrac{dx_N}{dx_i}$ in terms of $\dfrac{dx_N}{dx_{i+1}}$?

$$\frac{dx_N}{dx_i} = \frac{dx_N}{dx_{i+1}} \left(\frac{\partial x_{i+1}}{\partial x_i} \right)$$

Conclusion: run the computation forwards. Then initialize $\dfrac{dx_N}{dx_N} = 1$ and work **backwards** through the computation to find $\dfrac{dx_N}{dx_i}$ for each *I* from $\dfrac{dx_N}{dx_{i+1}}$.

Backpropagation Learning Rule

- Each weight changed by:

$$\Delta w_{ji} = \eta \delta_j o_i$$

$$\delta_j = o_j(1 - o_j)(t_j - o_j) \qquad \text{if } j \text{ is an output unit}$$

$$\delta_j = o_j(1 - o_j)\sum_k \delta_k w_{kj} \qquad \text{if } j \text{ is a hidden unit}$$

where η is a constant called the learning rate

t_j is the correct teacher output for unit *j*

δ_j is the error measure for unit *j*

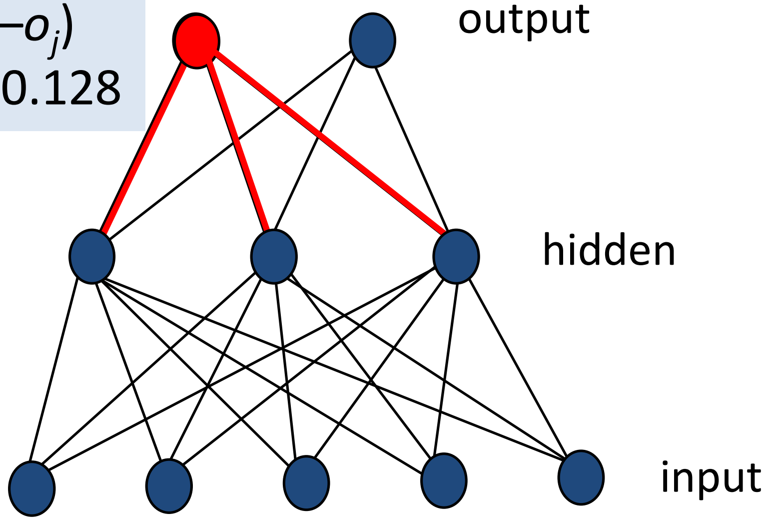
Error Backpropagation

- First calculate error of output units and use this to change the top layer of weights.

Current output: o_j=0.2
Correct output: t_j=1.0
Error $\delta_j = o_j(1 – o_j)(t_j – o_j)$
0.2(1–0.2)(1–0.2)=0.128

Update weights into j

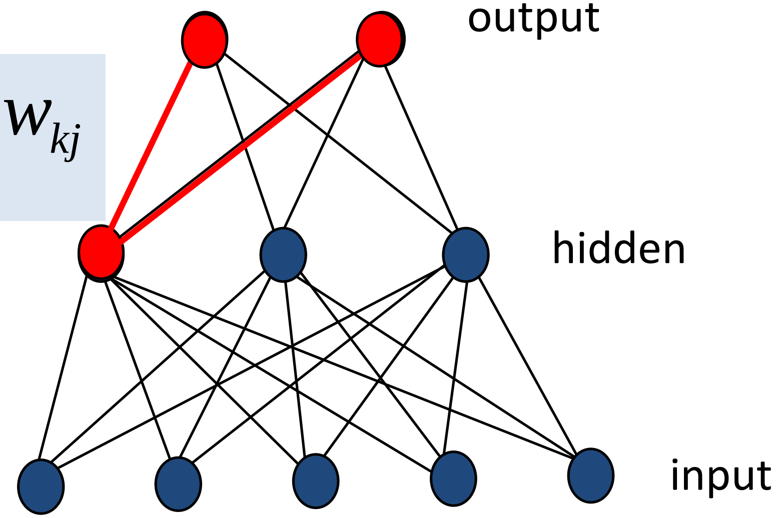
$$\Delta w_{ji} = \eta \delta_j o_i$$



Error Backpropagation

- Next calculate error for hidden units based on errors on the output units it feeds into.

$$\delta_j = o_j(1-o_j)\sum_k \delta_k w_{kj}$$



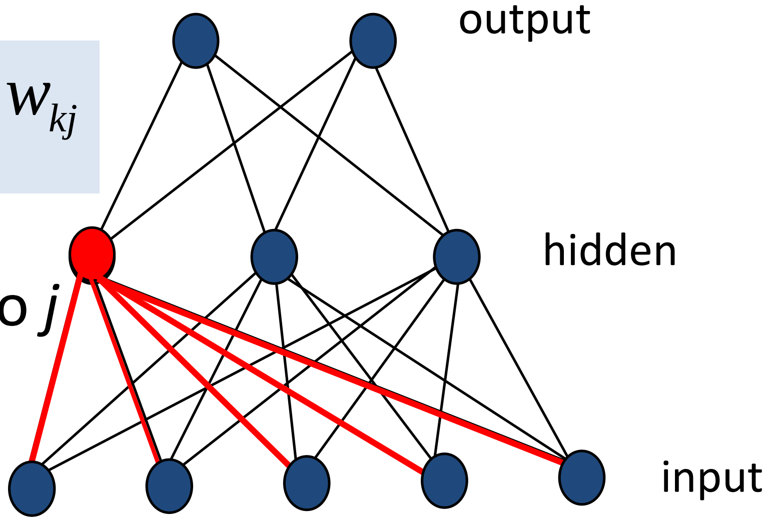
Error Backpropagation

- Finally update bottom layer of weights based on errors calculated for hidden units.

$$\delta_j = o_j(1-o_j)\sum_k \delta_k w_{kj}$$

Update weights into j

$$\Delta w_{ji} = \eta \delta_j o_i$$



Backpropagation Training Algorithm

Create the 3-layer network with *H* hidden units with full connectivity between layers. Set weights to small random real values.

Until all training examples produce the correct value (within ε), or mean squared error ceases to decrease, or other termination criteria:

Begin epoch

For each training example, *d*, do:

Calculate network output for *d*’s input values

Compute error between current output and correct output for *d*

Update weights by backpropagating error and using learning rule

End epoch

Comments on Training Algorithm

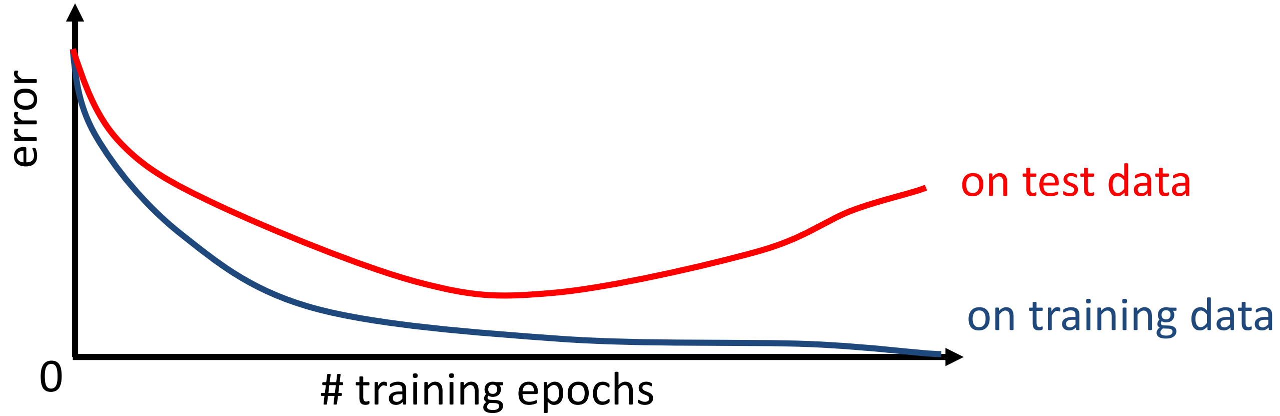
- Not guaranteed to converge to zero training error, may converge to local optima or oscillate indefinitely.
- However, in practice, does converge to low error for many large networks on real data.
- Many epochs (thousands) may be required, hours or days of training for large networks.
- To avoid local-minima problems, run several trials starting with different random weights (*random restarts*).
  – Take results of trial with lowest training set error.
  – Build a committee of results from multiple trials (possibly weighting votes by training set accuracy).

Hidden Unit Representations

- Trained hidden units can be seen as newly constructed features that make the target concept linearly separable in the transformed space.
- On many real domains, hidden units can be interpreted as representing meaningful features such as vowel detectors or edge detectors, etc..
- However, the hidden layer can also become a distributed representation of the input in which each individual unit is not easily interpretable as a meaningful feature.

Over-Training Prevention

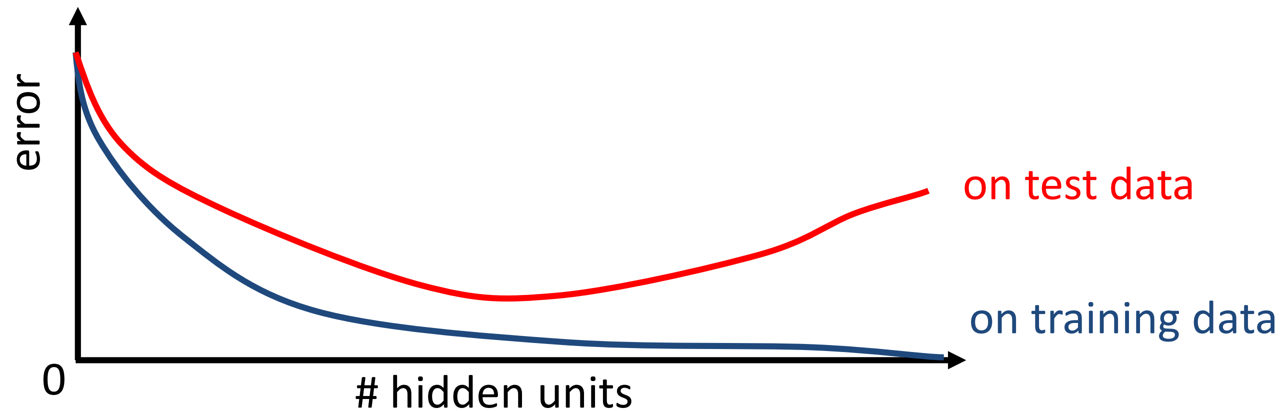
- Running too many epochs can result in over-fitting.



- Keep a hold-out validation set and test accuracy on it after every epoch. Stop training when additional epochs actually increase validation error.
- To avoid losing training data for validation:
  - Use internal 10-fold CV on the training set to compute the average number of epochs that maximizes generalization accuracy.
  - Train final network on complete training set for this many epochs.

Determining the Best Number of Hidden Units

- Too few hidden units prevents the network from adequately fitting the data.
- Too many hidden units can result in over-fitting.

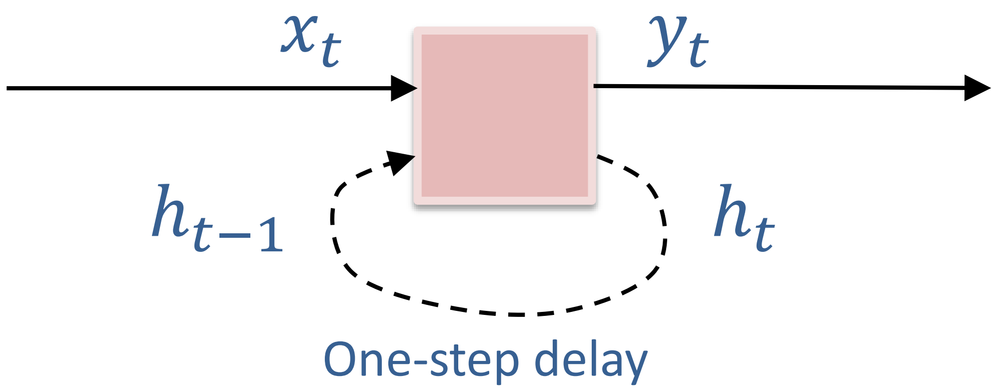


- Use internal cross-validation to empirically determine an optimal number of hidden units.

Recurrent Neural Networks (RNNs)

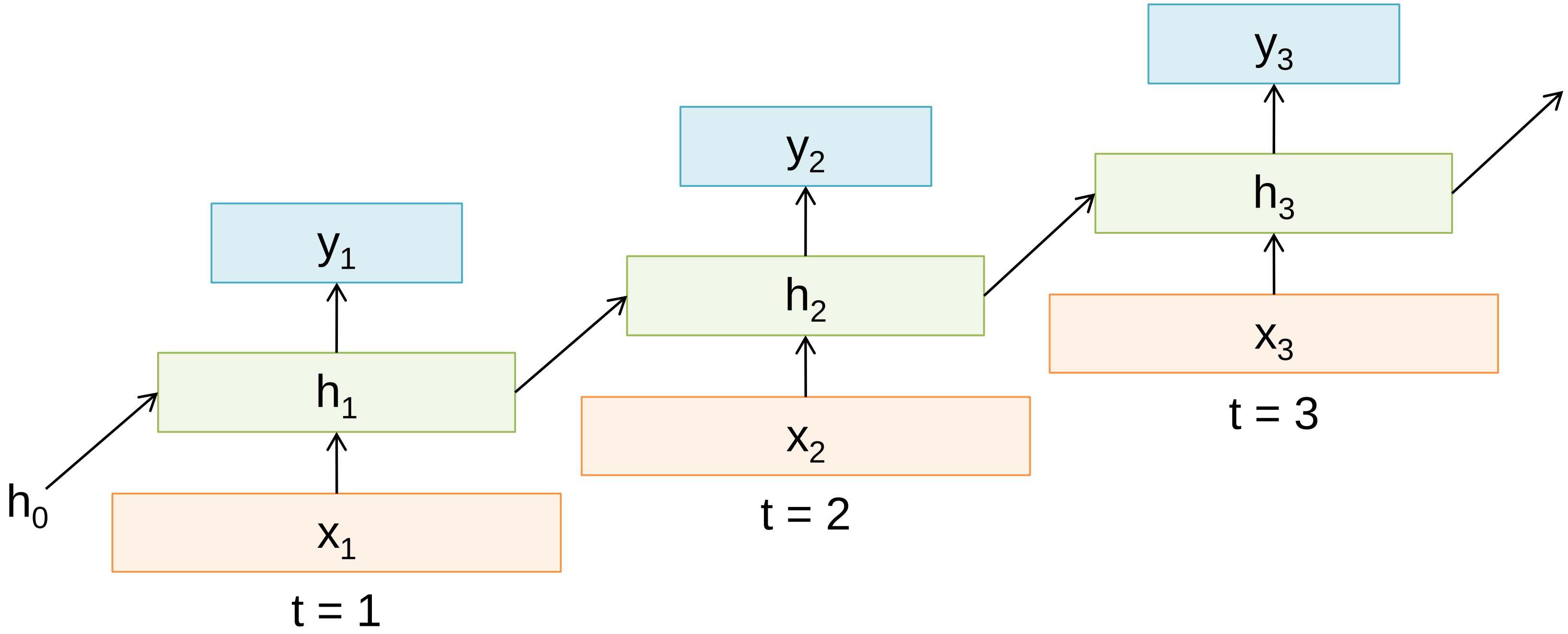
- Recurrent Neural Networks take the previous output or hidden states as inputs. Recurrent networks introduce cycles and a notion of time.

- The composite input at time t has some historical information about the happenings at time T < t



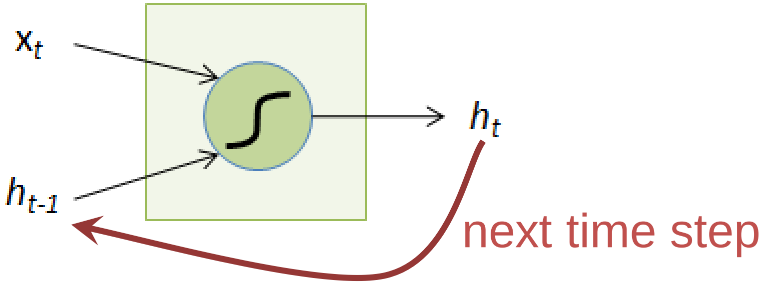
- They are designed to process sequences of data $x_1, \ldots, x_n$ and can produce sequences of outputs $y_1, \ldots, y_m$.

Sample RNN



The Recurrent Neuron

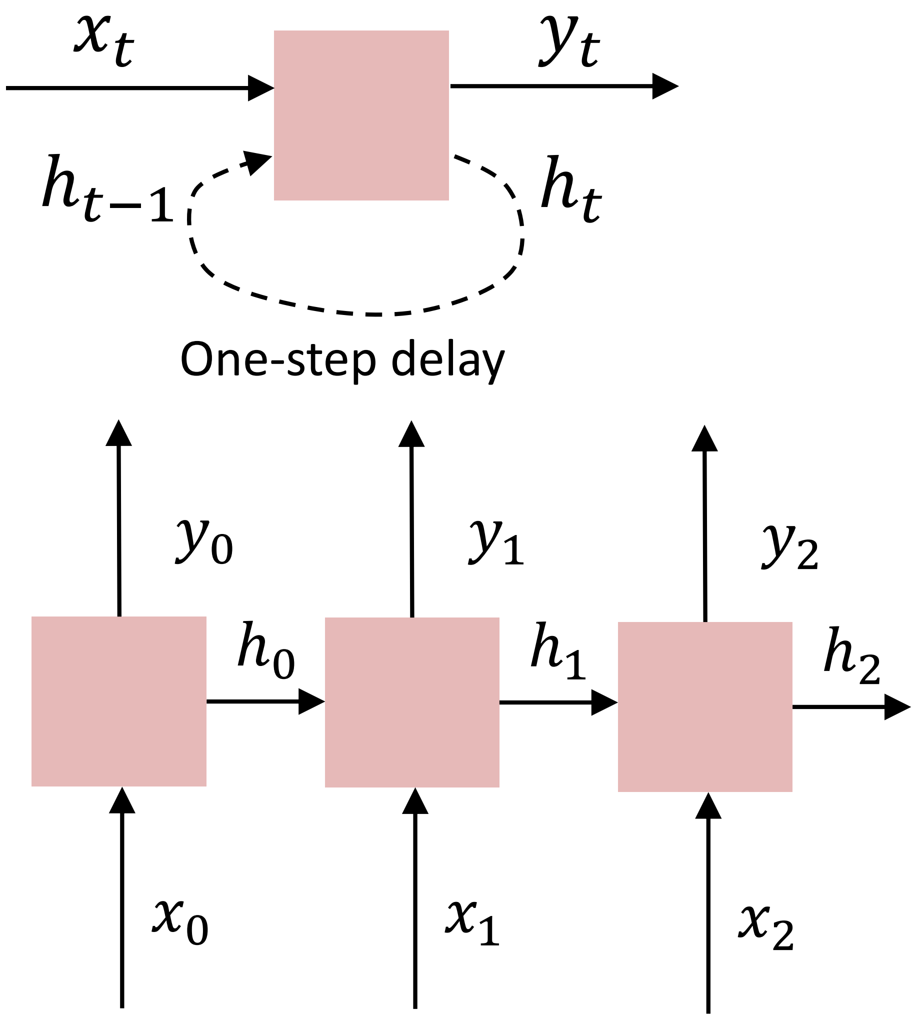
- x_t: Input at time t
- h_{t-1}: State at time t-1



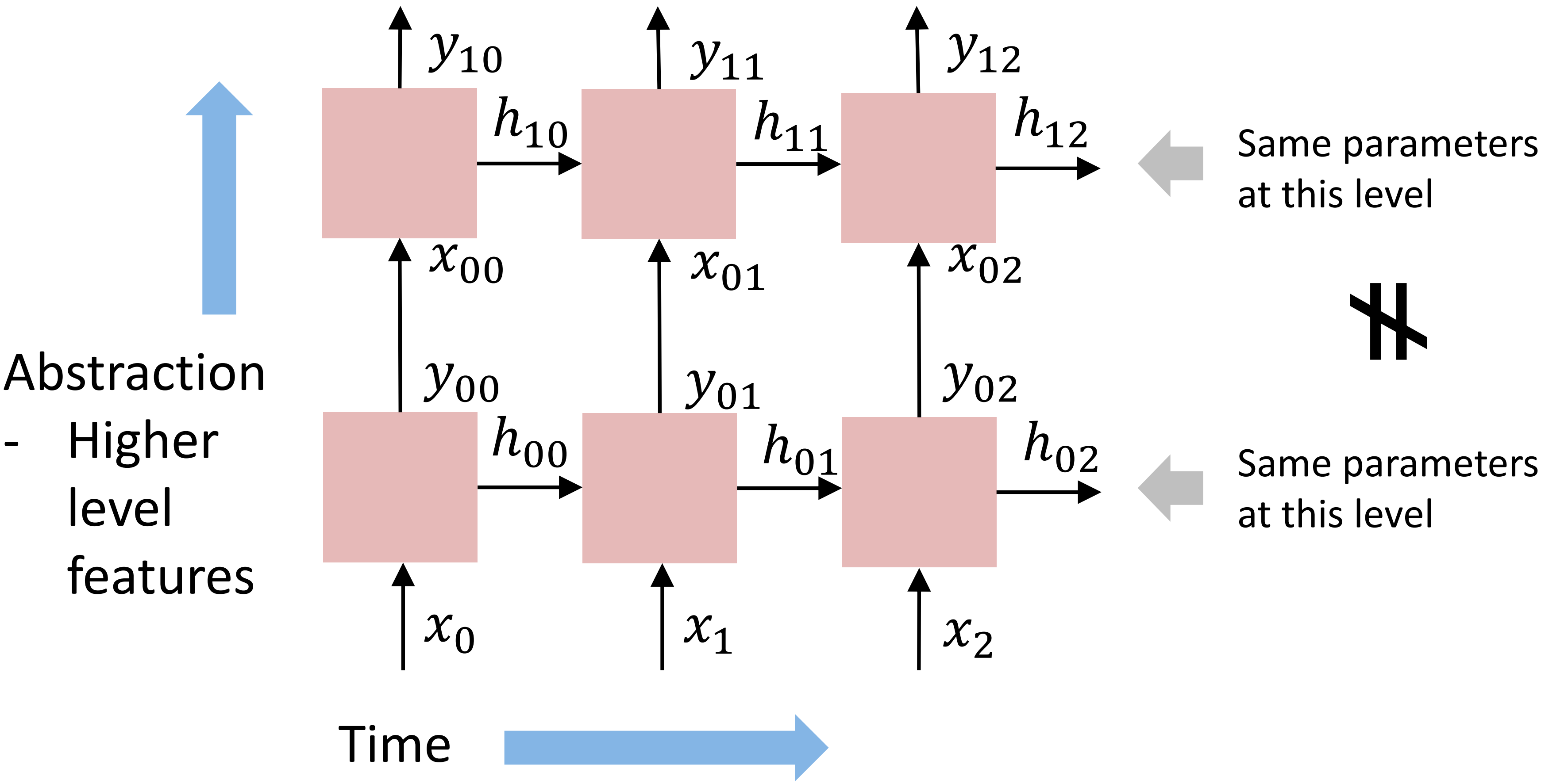
$$h_t = f(W_h h_{t-1} + W_x x_t)$$

Unrolling RNNs

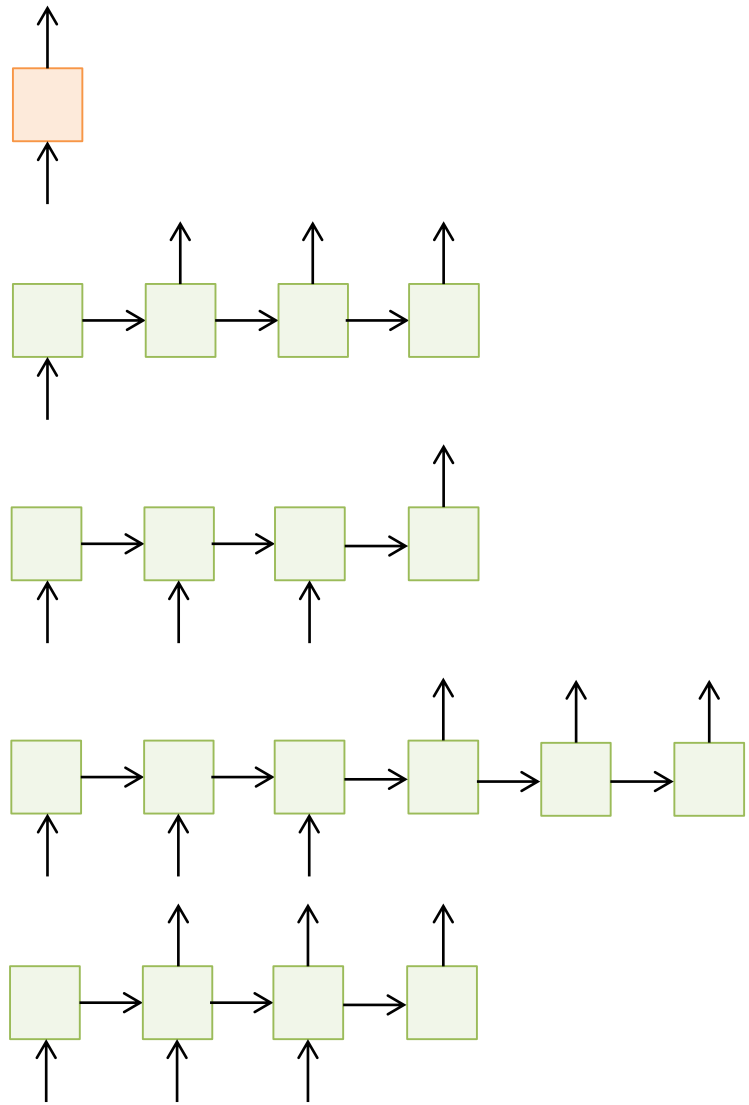
RNNs can be unrolled across multiple time steps.



Often layers are stacked vertically (deep RNNs):



Input – Output Scenarios



Single - Single	Feed-forward Network
Single - Multiple	Image Captioning
Multiple - Single	Sentiment Classification
Multiple - Multiple	Translation
	Image Captioning

Sentiment Classification

- Classify a
  restaurant review from Yelp! OR
  movie review from IMDB OR
  …
  as positive or negative

- Inputs: Multiple words, one or more sentences
- Outputs: Positive / Negative classification

- “The food was really good”
- “The chicken crossed the road because it was uncooked”

Sentiment Classification

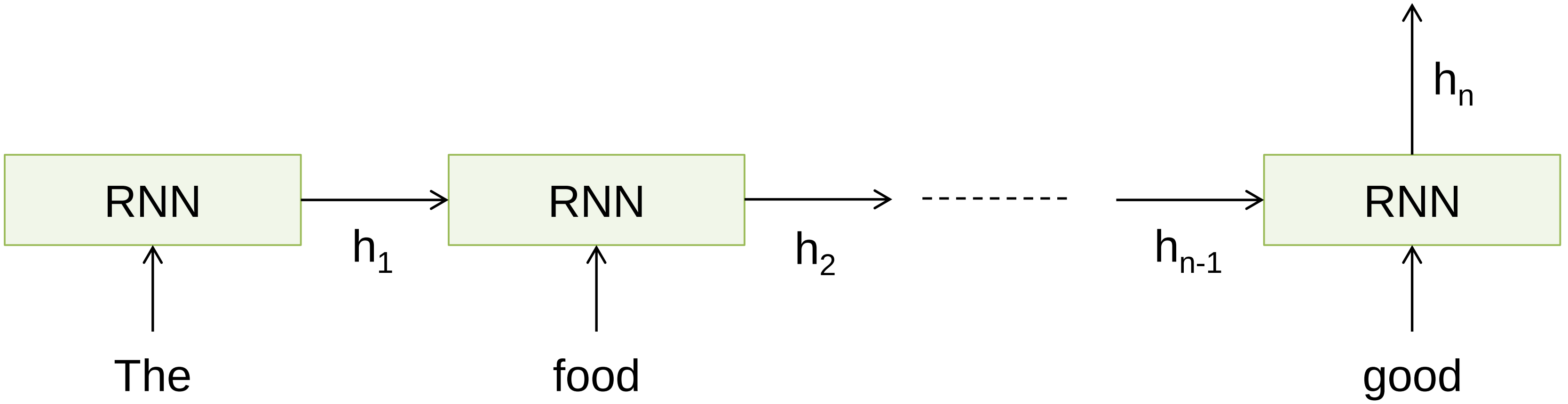


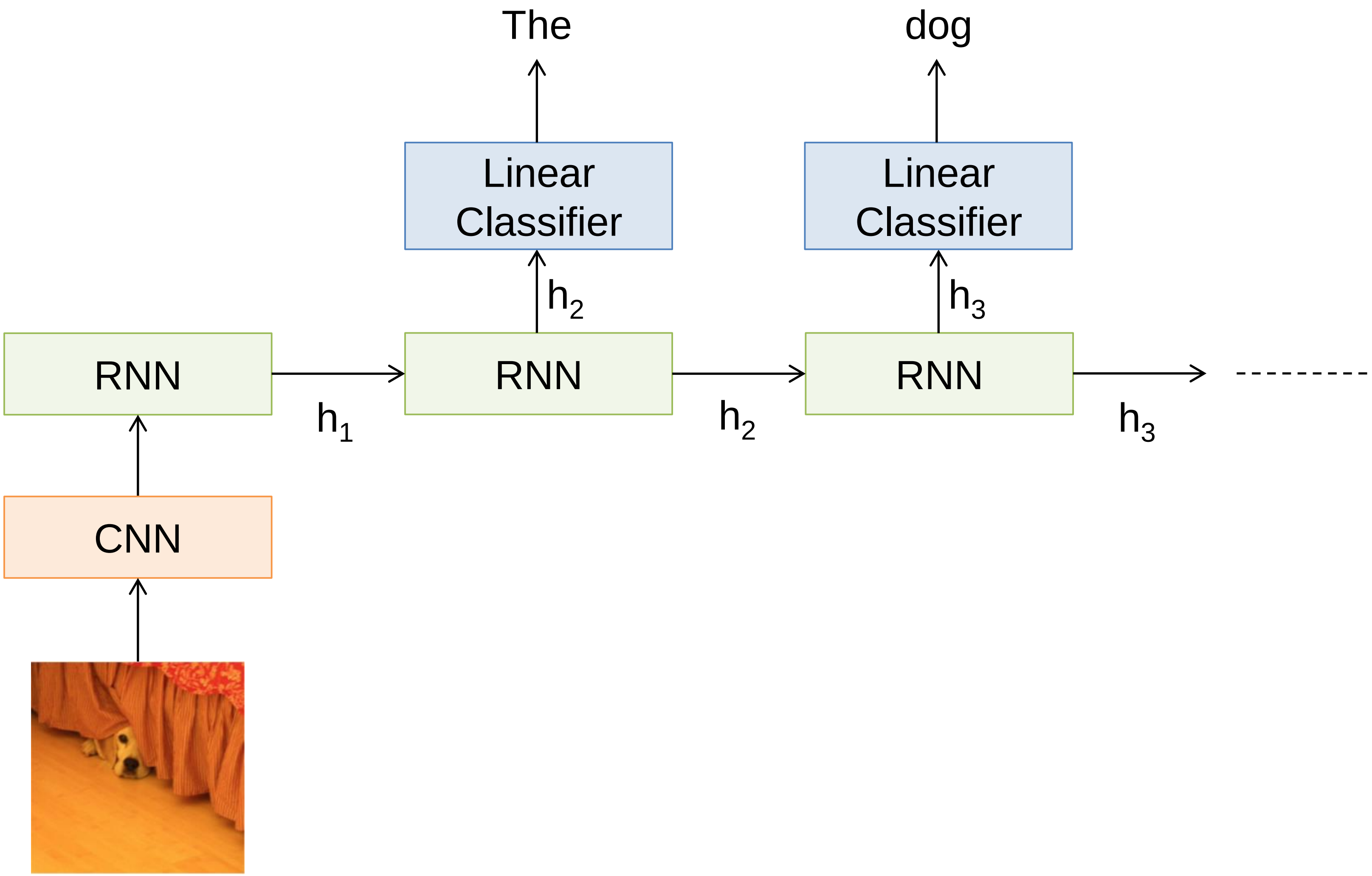
Image Captioning

- Given an image, produce a sentence describing its contents

- Inputs: Image feature (from a CNN)
- Outputs: Multiple words (let's consider one sentence)


: The dog is hiding

Image Captioning



RNN Outputs: Language Modeling

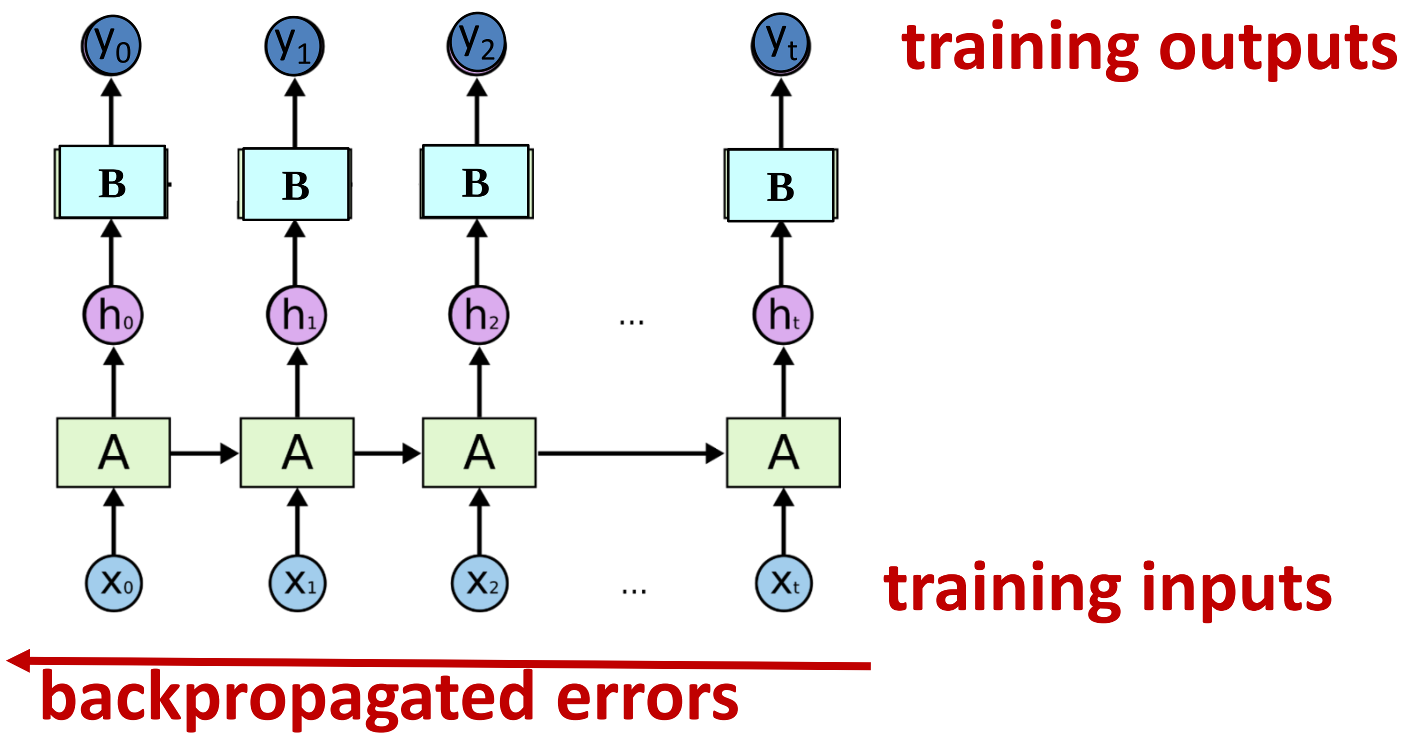
VIOLA:
Why, Salisbury must find his flesh and thought
That which I am not aps, not a man and in fire,
To show the reining of the raven and the wars
To grace my hand reproach within, and not a fair are hand,
That Caesar and my goodly father's world;
When I was heaven of presence and our fleets,
We spare with hours, but cut thy council I am great,
Murdered and by thy master's ready there
My power to give thee but so much as hell:
Some service in the noble bondman here,
Would show him to her wine.

KING LEAR:
O, if you were a feeble sight, the courtesy of your law,
Your sight and several breath, will wear the gods
With his heads, and my hands are wonder'd at the deeds,
So drop upon your lordship's head, and your opinion
Shall be against your honour.

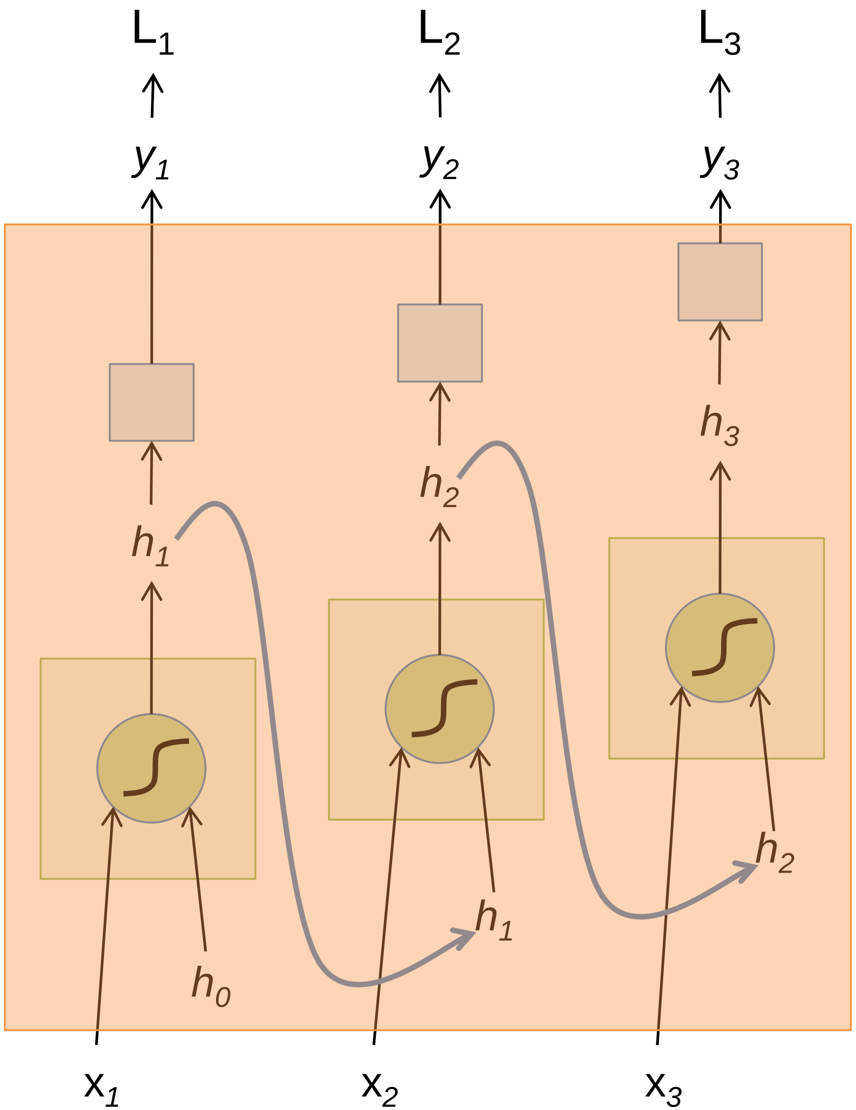
http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Training RNN's

- RNNs can be trained using "backpropagation through time."
- Can viewed as applying normal backprop to the unrolled network.

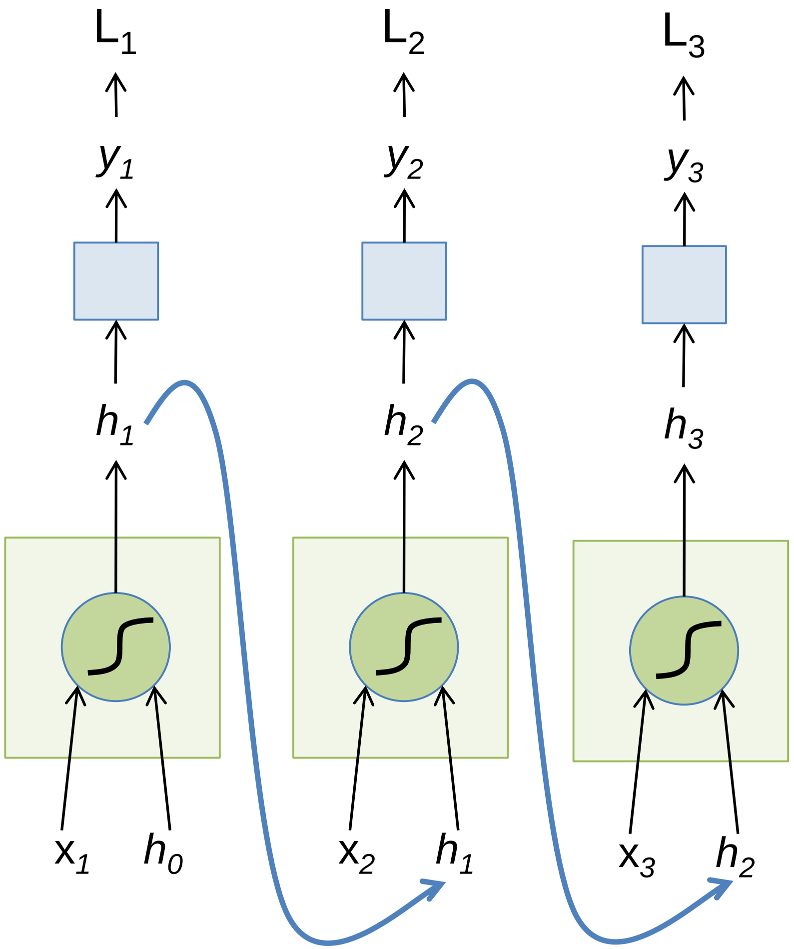


The Unfolded Vanilla RNN

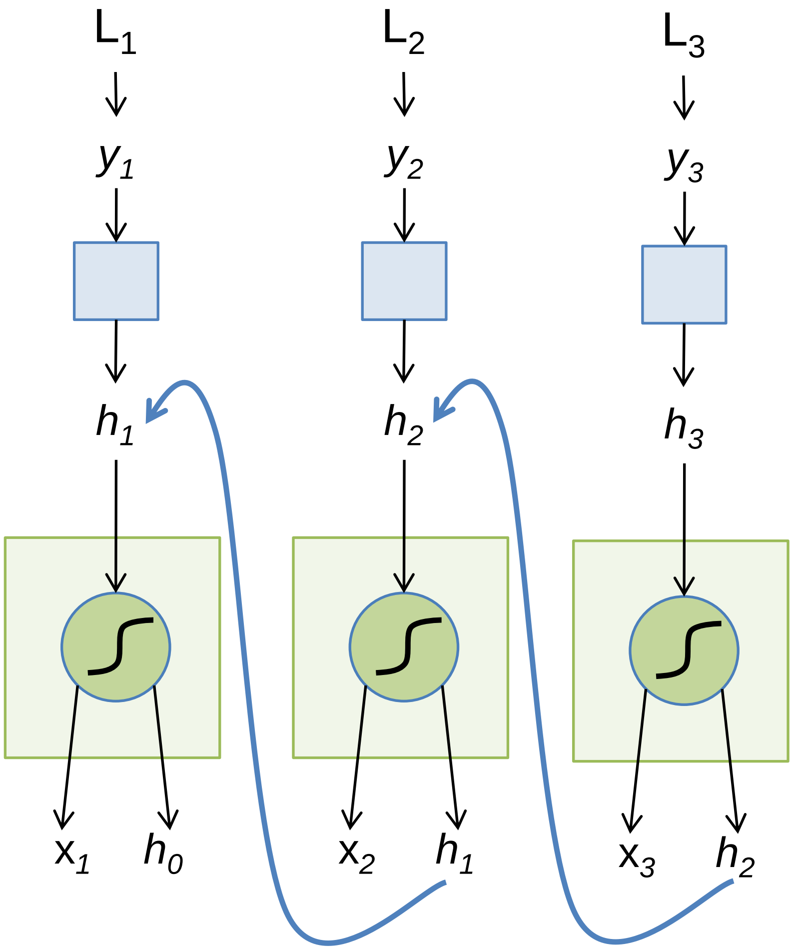


- Treat the unfolded network as one big feed-forward network!
- This big network takes in entire sequence as an input
- Compute gradients through the usual backpropagation
- Update shared weights

The Unfolded Vanilla RNN Forward



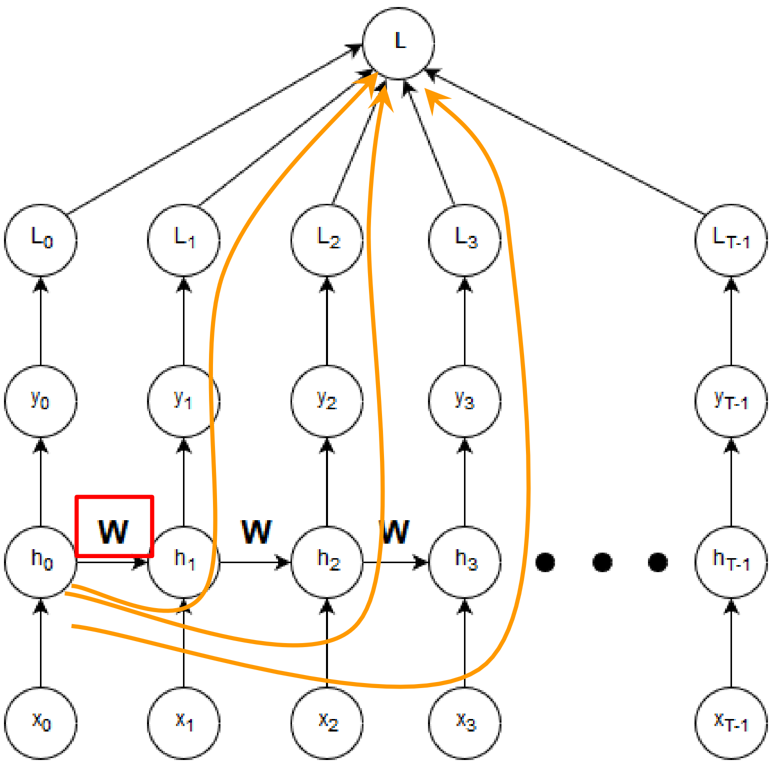
The Unfolded Vanilla RNN Backward



$$\frac{\partial L_t}{\partial h_1} = \left(\frac{\partial L_t}{\partial y_t}\right)\left(\frac{\partial y_t}{\partial h_1}\right)$$

$$= \left(\frac{\partial L_t}{\partial y_t}\right)\left(\frac{\partial y_t}{\partial h_t}\right)\left(\frac{\partial h_t}{\partial h_{t-1}}\right)\cdots\left(\frac{\partial h_2}{\partial h_1}\right)$$

Backpropagation Through Time (BPTT)



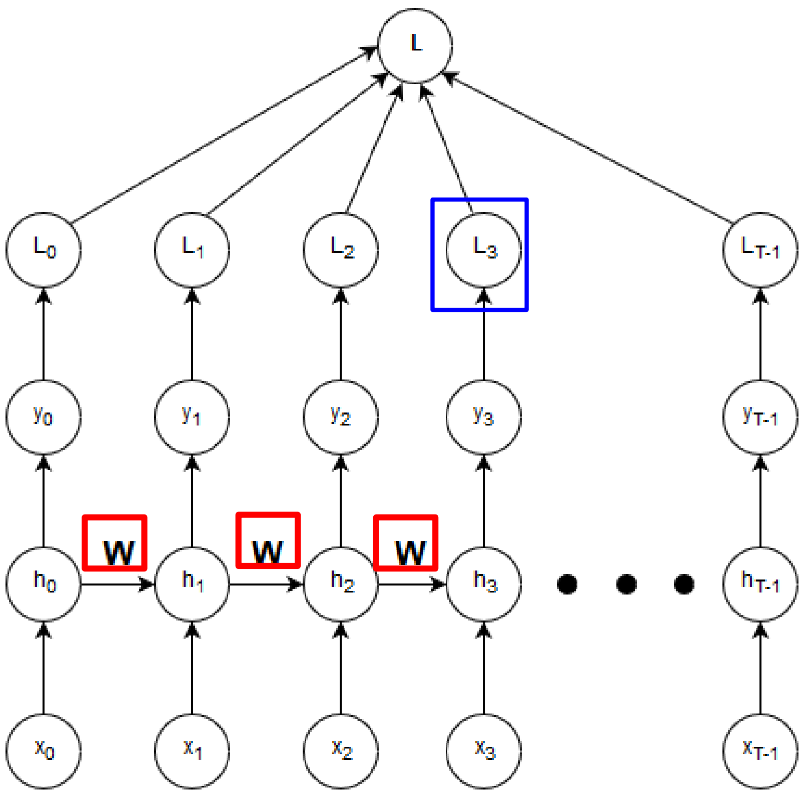
- Objective is to update the weight matrix:

$$\mathbf{W} \to \mathbf{W} - \alpha \frac{\partial L}{\partial \mathbf{W}}$$

- Issue: **W** occurs each timestep
- **Every** path from **W** to L is one dependency
- Find all paths from **W** to L!

(note: dropping subscript h from $\mathbf{W_h}$ for brevity)

Backpropagation as two summations



1. First summation over L

$$\frac{\partial L}{\partial \mathbf{W}} = \sum_{j=0}^{T-1} \frac{\partial L_j}{\partial \mathbf{W}}$$

2. Second summation over h: Each L_j depends on the weight matrices *before it*

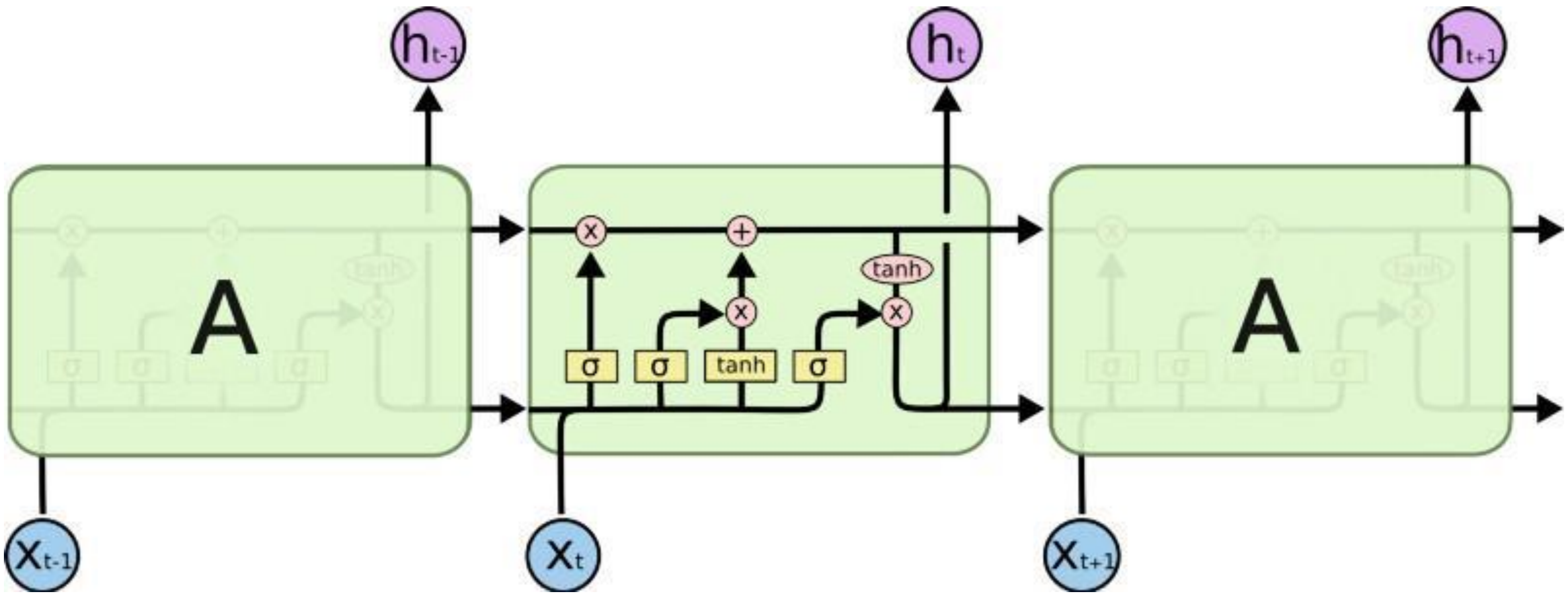
$$\frac{\partial L_j}{\partial \mathbf{W}} = \sum_{k=1}^{j} \boxed{\frac{\partial L_j}{\partial h_k}} \frac{\partial h_k}{\partial \mathbf{W}}$$

L_j depends on all h_k before it.

Why RNNs?

- Can model sequences having variable length
- Inputs, outputs can be different lengths in different examples
- **Efficient**: Weights shared across time-steps

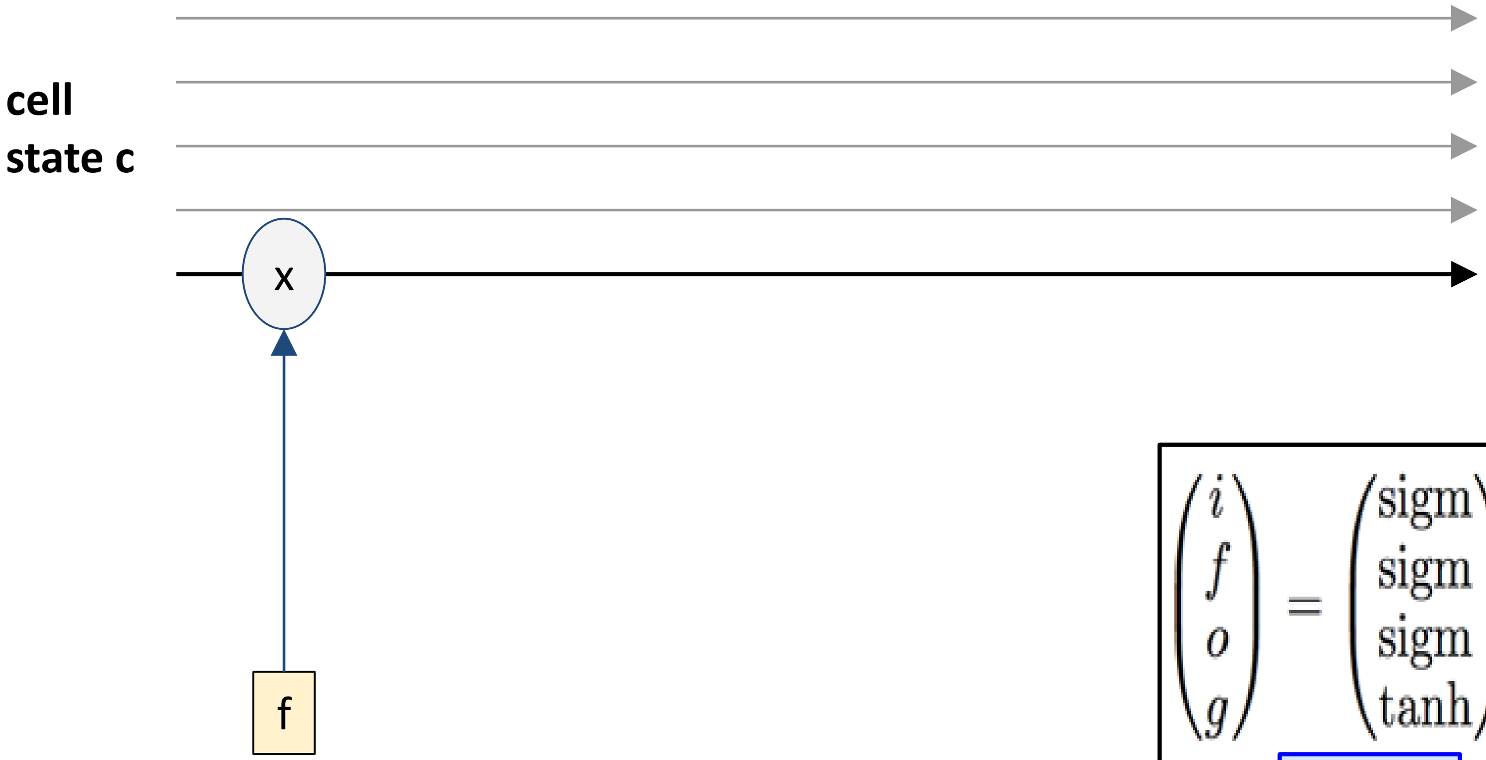
The LSTM Network



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

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



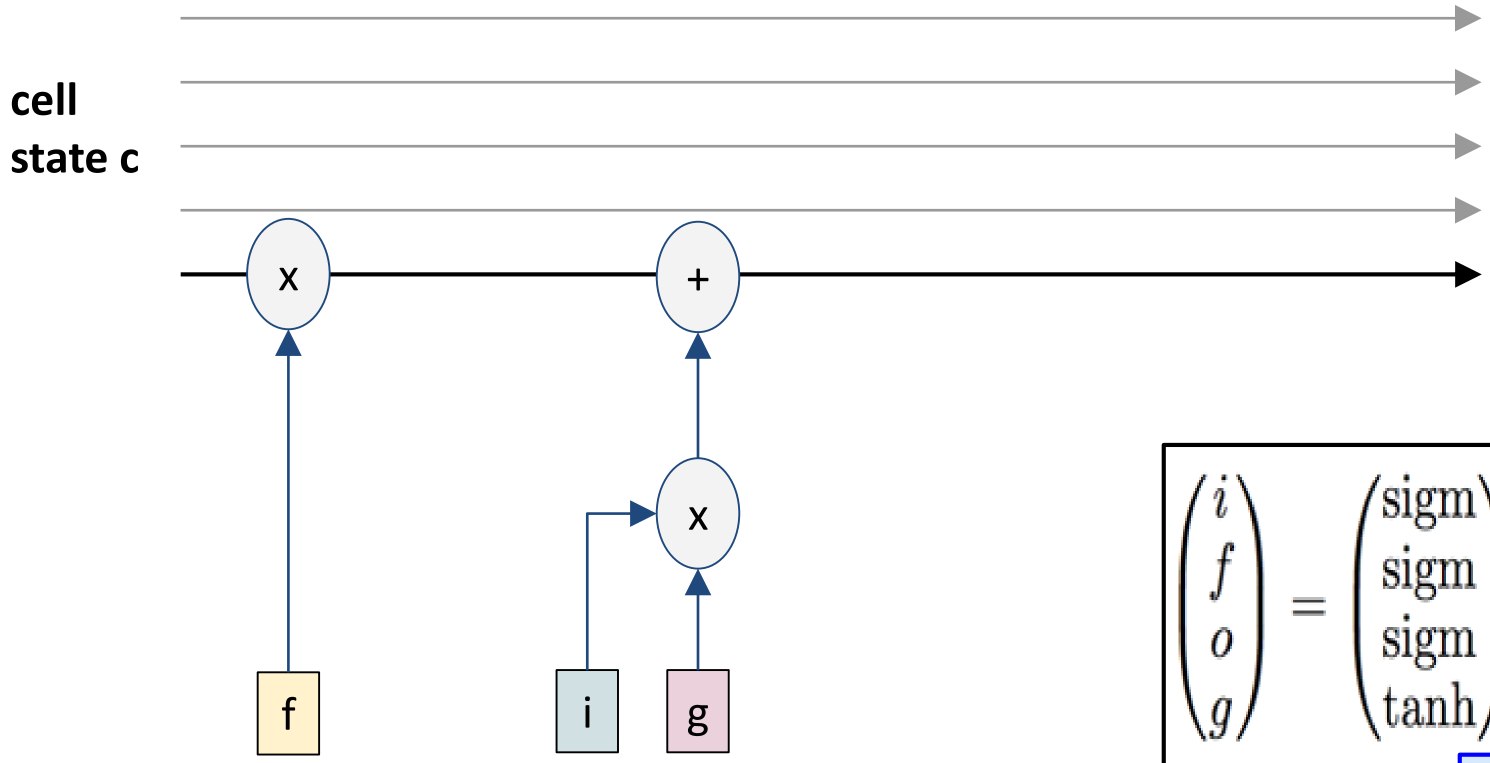
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \mathrm{sigm} \\ \mathrm{sigm} \\ \mathrm{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$$c_t^l = \boxed{f \odot c_{t-1}^l} + i \odot g$$

$$h_t^l = o \odot \tanh(c_t^l)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

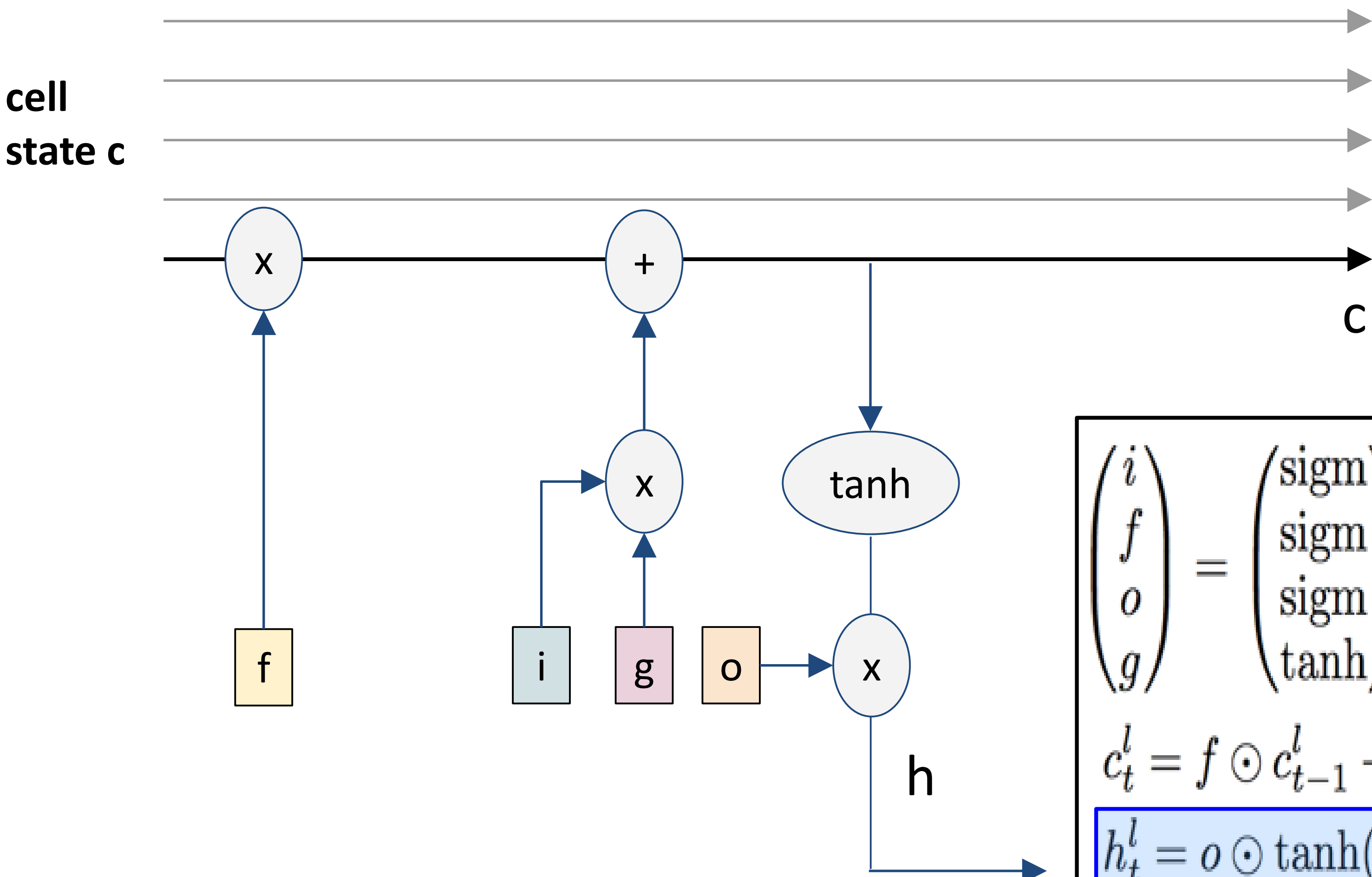


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$$c_t^l = f \odot c_{t-1}^l \boxed{+ i \odot g}$$

$$h_t^l = o \odot \tanh(c_t^l)$$

[Hochreiter et al., 1997]

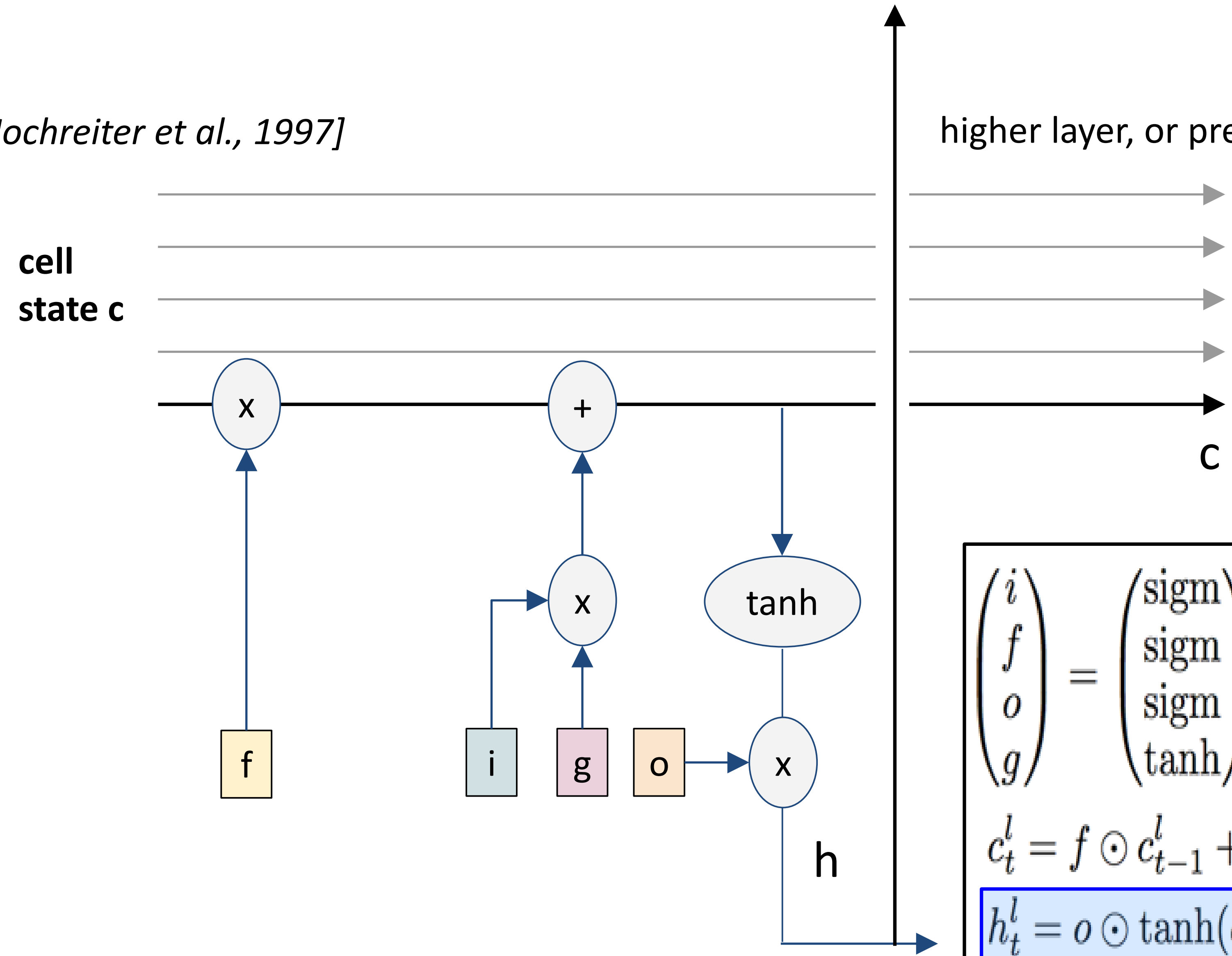


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

$$\boxed{h_t^l = o \odot \tanh(c_t^l)}$$

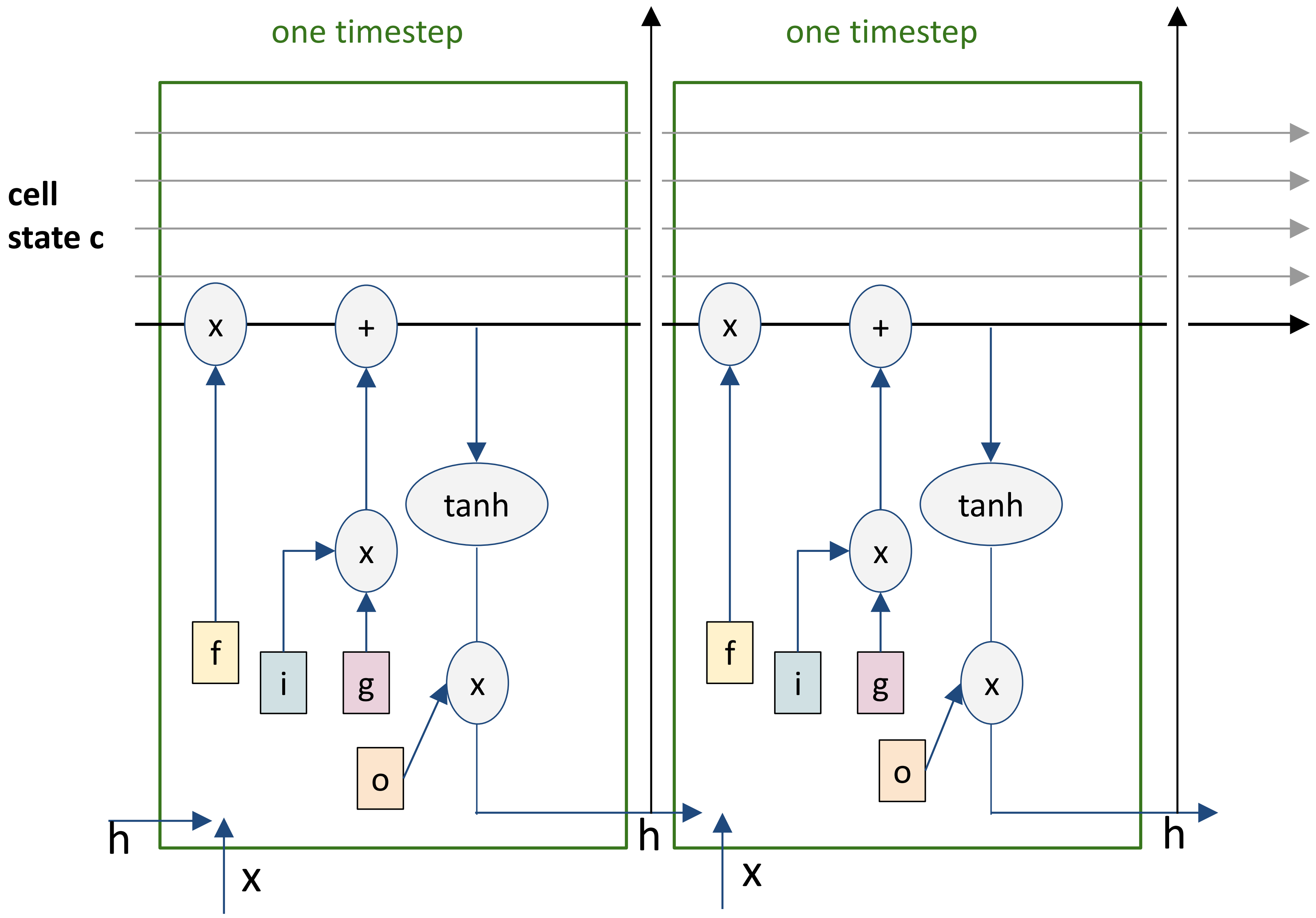
[Hochreiter et al., 1997]



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

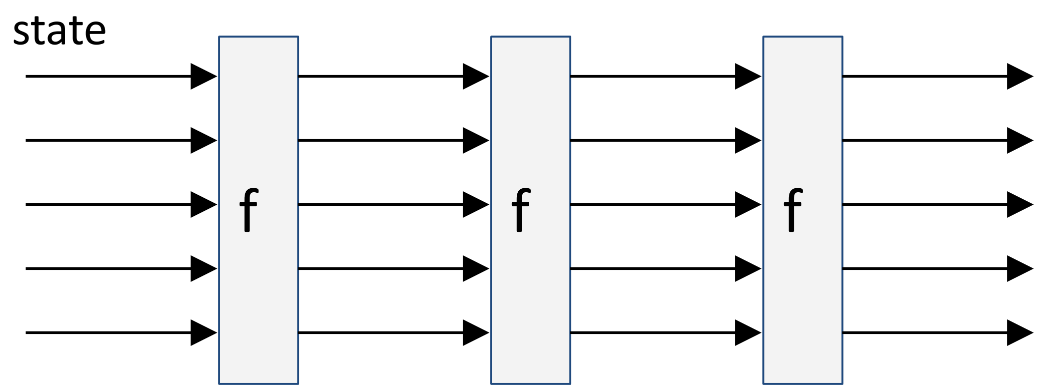
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

$$\boxed{h_t^l = o \odot \tanh(c_t^l)}$$

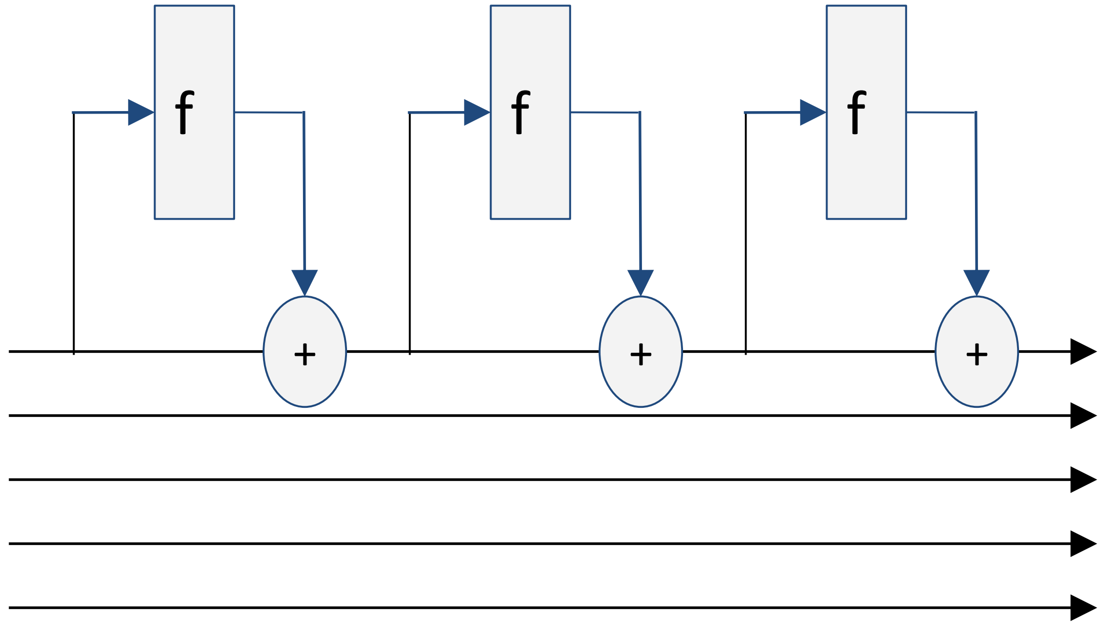


Based on cs231n by Fei-Fei Li & Andrej Karpathy & Justin Johnson

RNN

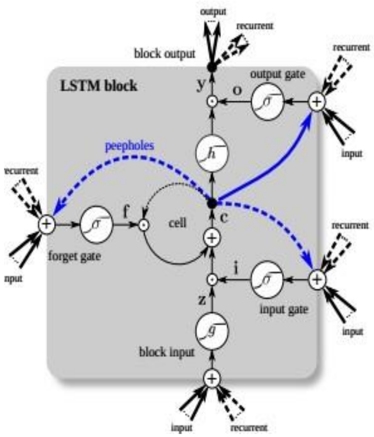


LSTM
(ignoring forget gates)



Based on cs231n by Fei-Fei Li & Andrej Karpathy & Justin Johnson

[*An Empirical Exploration of Recurrent Network Architectures,* Jozefowicz et al., 2015]



[*LSTM: A Search Space Odyssey,* Greff et al., 2015]

GRU [*Learning phrase representations using rnn encoder-decoder for statistical machine translation*, Cho et al. 2014]

$$\begin{aligned}
r_t &= \mathrm{sigm}\left(W_{\mathrm{xr}} x_t + W_{\mathrm{hr}} h_{t-1} + b_{\mathrm{r}}\right) \\
z_t &= \mathrm{sigm}(W_{\mathrm{xz}} x_t + W_{\mathrm{hz}} h_{t-1} + b_{\mathrm{z}}) \\
\tilde{h}_t &= \tanh(W_{\mathrm{xh}} x_t + W_{\mathrm{hh}}(r_t \odot h_{t-1}) + b_{\mathrm{h}}) \\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t
\end{aligned}$$

MUT1:

$$\begin{aligned}
z &= \mathrm{sigm}(W_{\mathrm{xz}} x_t + b_{\mathrm{z}}) \\
r &= \mathrm{sigm}(W_{\mathrm{xr}} x_t + W_{\mathrm{hr}} h_t + b_{\mathrm{r}}) \\
h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + \tanh(x_t) + b_{\mathrm{h}}) \odot z \\
&+ h_t \odot (1 - z)
\end{aligned}$$

MUT2:

$$\begin{aligned}
z &= \mathrm{sigm}(W_{\mathrm{xz}} x_t + W_{\mathrm{hz}} h_t + b_{\mathrm{z}}) \\
r &= \mathrm{sigm}(x_t + W_{\mathrm{hr}} h_t + b_{\mathrm{r}}) \\
h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + W_{xh} x_t + b_{\mathrm{h}}) \odot z \\
&+ h_t \odot (1 - z)
\end{aligned}$$

MUT3:

$$\begin{aligned}
z &= \mathrm{sigm}(W_{\mathrm{xz}} x_t + W_{\mathrm{hz}} \tanh(h_t) + b_{\mathrm{z}}) \\
r &= \mathrm{sigm}(W_{\mathrm{xr}} x_t + W_{\mathrm{hr}} h_t + b_{\mathrm{r}}) \\
h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + W_{xh} x_t + b_{\mathrm{h}}) \odot z \\
&+ h_t \odot (1 - z)
\end{aligned}$$

- RNNs allow a lot of flexibility in architecture design
- Vanilla RNNs are simple but don’t work very well
- Common to use LSTM or GRU: their additive interactions improve gradient flow
- Backward flow of gradients in RNN can explode or vanish. Exploding is controlled with gradient clipping. Vanishing is controlled with additive interactions (LSTM)
- Better/simpler architectures are a hot topic of current research
- Better understanding (both theoretical and empirical) is needed.