16-785: Integrated Intelligence in Robotics: Vision, Language, and Planning

Spring 2019

Deep Learning Basics

Overview

- Deep learning basics
  - Preliminaries
  - Feed forward networks
- Reference
  - Deep Learning (2017), Ch. 2 – 10
    Ian Goodfellow, Yoshua Bengio, and Aaron Courville, The MIT Press
    http://www.deeplearningbook.org/
  - Pattern Classification, The 2nd Edition (2000), Ch. 6
    Richard Duda, Peter E. Hart, David G. Stork
    http://cns-classes.bu.edu/cn550/Readings/duda-etal-00.pdf

Gradient-based optimization

- Minimizing f(x) by changing x
- Objective function f(x)

= Criterion

= Cost function

= Loss function

= Error function

Gradient

- $f(x) = f(x_1, x_2, \ldots, x_n)$: multiple input variables
- Partial derivative $\frac{\partial}{\partial x_i} f(x)$: How f(x) changes w.r.t. only x_i
- Gradient

  Vector of partial derivatives for all x_i

$$\nabla_x f(x) = \left[\frac{\partial}{\partial x_1} f(x), \frac{\partial}{\partial x_2} f(x), \ldots, \frac{\partial}{\partial x_i} f(x), \ldots, \frac{\partial}{\partial x_n} f(x)) \right]$$

Critical points: where all elements of gradient = 0

Jacobian matrix

- $f(x) = f(x_1, x_2, \ldots, x_m)$: multiple input variables
- Partial derivative $\frac{\partial}{\partial x_i} f(x)$: How f(x) changes w.r.t. only x_i
- Gradient $\nabla_x f(x)$: Vector of partial derivatives for all x_i

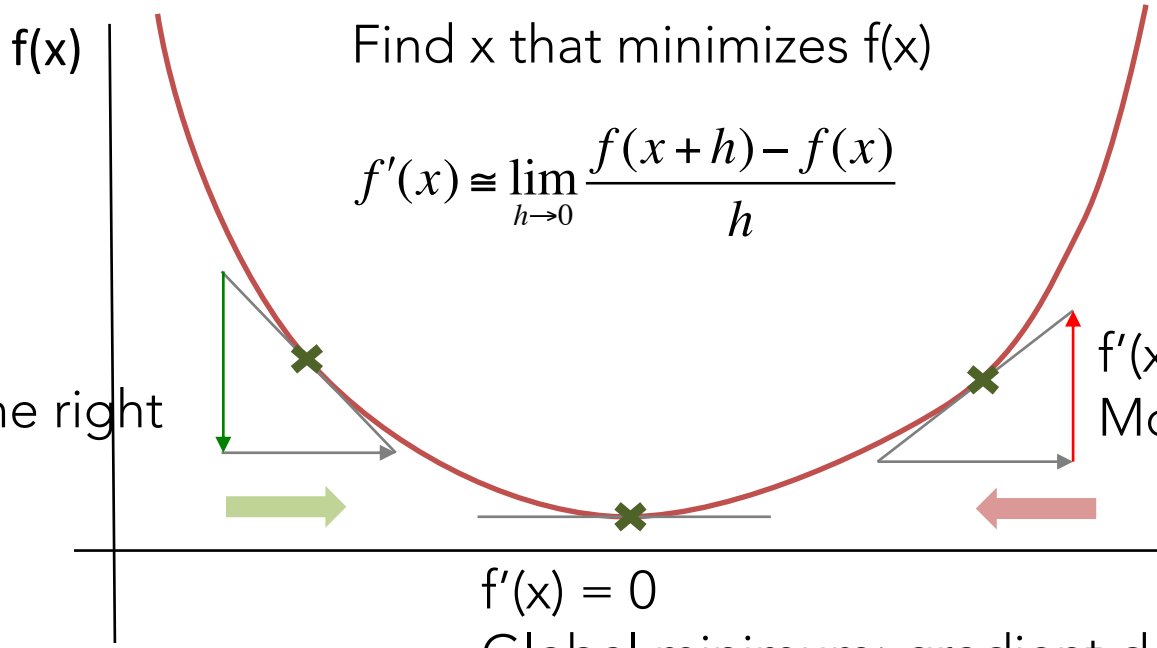
$$\nabla_x f(x) = \left[\frac{\partial}{\partial x_1} f(x), \frac{\partial}{\partial x_2} f(x), \ldots, \frac{\partial}{\partial x_i} f(x), \ldots, \frac{\partial}{\partial x_m} f(x)) \right]$$

- Multiple inputs, multiple outputs
  - f: $R^m \rightarrow R^n$
  - Jacobian matrix J_f

$$J_f = \begin{bmatrix} \ldots \\ \ldots J_{i,j} = \frac{\partial}{\partial x_j} f(x)_i \ldots \\ \ldots \end{bmatrix}$$

Gradient descent

[Cauchy 1847]



Find x that minimizes f(x)

$$f'(x) \cong \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

f′(x) < 0
Move x to the right

f′(x) > 0
Move x to the left

f′(x) = 0
Global minimum; gradient descent stops here.

Gradient descent

[Cauchy 1847]

- f(x) is descended when moving x in the negative direction of the gradient

$$x' = x - \varepsilon \nabla_x f(x)$$

Maximum Likelihood Estimation

- m i.i.d. examples: $X = \{ x^{(1)}, \ldots, x^{(m)} \}$
- $p_{data}(x)$: true probability distribution
- $p_{model}(x;\boldsymbol{\theta})$: parameterized probability model estimating true $p_{data}(x)$

$$\theta_{ML} = \underset{\theta}{\operatorname{argmax}}\, p_{model}(\mathrm{X};\theta) = \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{m} p_{model}(x^{(i)};\theta)$$

Maximum Likelihood Estimation

$$\theta_{ML} = \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{m} p_{model}(x^{(i)}; \theta)$$

Numerical underflow

Maximum Likelihood Estimation

$$\theta_{ML} = \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{m} p_{model}(x^{(i)};\theta)$$

Numerical underflow

$$= \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{m} \log p_{model}(x^{(i)};\theta)$$

Logarithm: product → sum

Maximum Likelihood Estimation

$$\theta_{ML} = \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{m} p_{model}(x^{(i)};\theta)$$

Numerical underflow

$$= \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{m} \log p_{model}(x^{(i)};\theta)$$

Logarithm: product → sum

$$= \underset{\theta}{\operatorname{argmax}} E_{x\sim\hat{p}_{data}} \log p_{model}(x;\theta)$$

Expectation

Gradient-based learning

- Optimization – Minimizing loss
- Loss function: commonly, maximum likelihood
  - Negative log-likelihood
  - Cross-entropy between training data & model distribution

$$J(\theta) = -E_{x,y\sim\hat{p}_{data}} \log p_{model}(y \mid x;\theta)$$

Deep Feedforward Networks

- A.k.a. Multilayer perceptron (MLP)
- Function approximation machines
  - Defines a mapping $y = f(x; \boldsymbol{\theta})$
  - Learns the values of $\boldsymbol{\theta}$ that best approximate the function
- Feedforward (as opposed to recurrent): information flows from input to output (without feedback loop)

Deep Feedforward Networks

Example

- $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$ is represented as a network of 3 functions $f^{(1)}, f^{(2)}, f^{(3)}$ connected in a chain

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$$

Input

1st layer

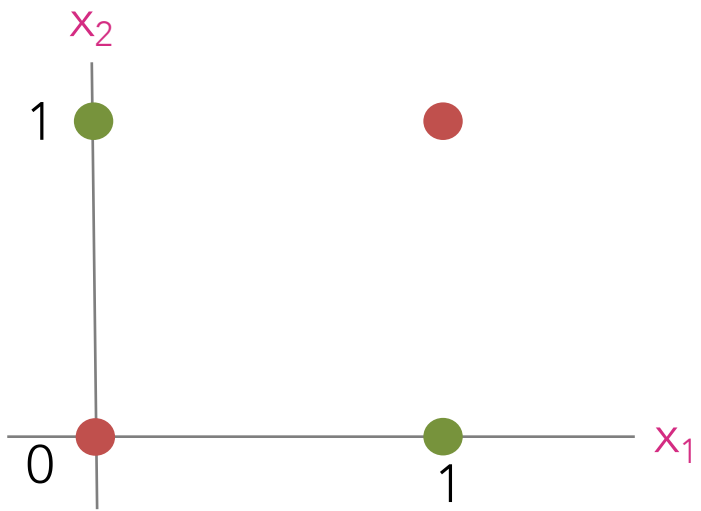
2nd layer

Output layer

} Hidden layers because values are not observed from training data

Exclusive OR (XOR)

- $X = \{ [0,0]^T, [0,1]^T, [1,0]^T, [1,1]^T \}$ $[x_1, x_2]^T$
- $Y = \{ \quad 0, \quad 1, \quad 1, \quad 0 \quad \}$



XOR as regression

- ~~Linear model~~

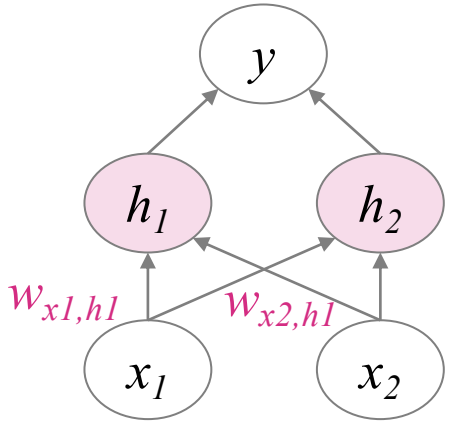
$$f(x;\theta) = f(x;w,b) = x^T w + b$$ w=0, b=0.5

- Mean Squared Error (MSE) loss function

$$J(\theta) = \frac{1}{4}\sum_{x \in X}(f^*(x) - f(x;\theta))^2$$ Minimize loss J(θ)

2-layer neural network

- 1 hidden layer with 2 hidden units

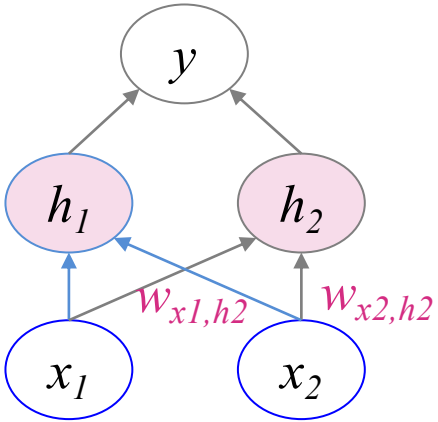


Explicit representation

Width: number of hidden units, |h|
Depth: number of layers in the network

2-layer neural network

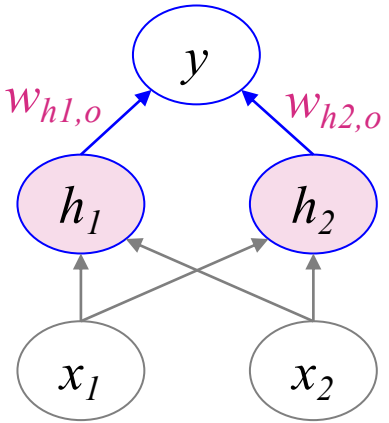
- 1 hidden layer with 2 hidden units



Explicit representation

2-layer neural network

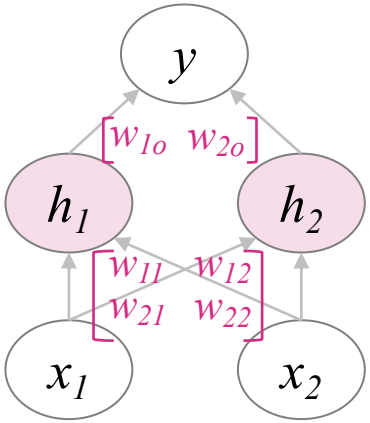
- 1 hidden layer with 2 hidden units



Explicit representation

2-layer neural network

- 1 hidden layer with 2 hidden units

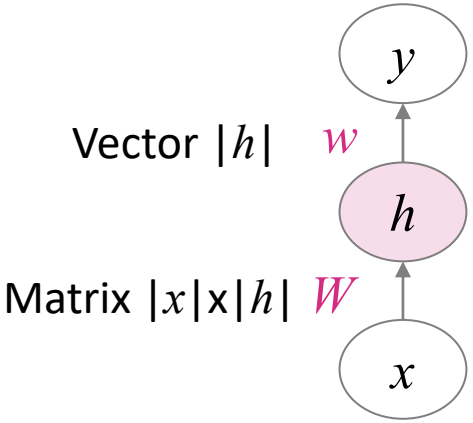


Explicit representation

$$f(x; W, c, w, b) = f^{(2)}(f^{(1)}(x))$$

Last|output (2nd) layer

$$y = f^{(2)}(h; w, b)$$



1st hidden layer

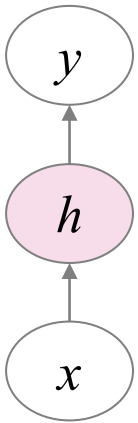
$$h = f^{(1)}(x; W, c)$$

Input layer

$$[x_1, x_2]^\top$$

Compact representation

Hidden layer function

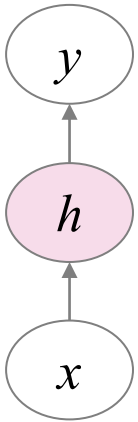


Affine transformation of input x

$$h = W^T x + c$$

linear bias

Hidden layer function



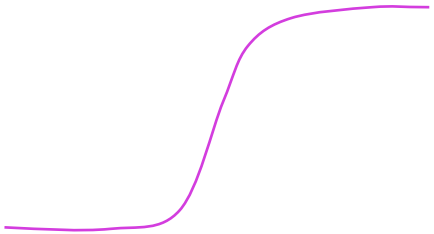
Affine transformation + nonlinear activation function g

$$h = g(W^T x + c)$$

nonlinear

Cost function

- Gradient of the cost function must be large and predictable
- Saturating functions (becoming flat) is problematic because gradient becomes very small
  - E.g., Sigmoid function



Cross-entropy cost function

- Most commonly used
- Training using maximum likelihood

  = Cost function is negative log-likelihood

  = Cross-entropy between training data & model distribution

$$J(\theta) = -E_{x,y\sim\hat{p}_{data}} \log p_{model}(y \mid x;\theta)$$

Output units

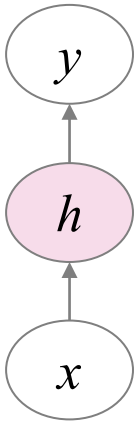
- Linear units for Gaussian distributions
- Sigmoid units for binary-class distributions
- Softmax units for multi-class distributions

$$soft\max(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}$$

Hidden units

- Logistic sigmoid activation function
- Hyperbolic tangent activation function
- ReLU

Hidden layer function



e.g., ReLU $g(z) = \max\{0, z\}$

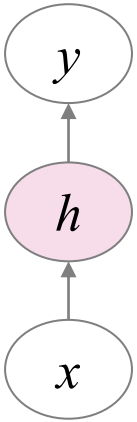
Affine transformation + nonlinear activation function g

$$h = g(W^T x + c)$$

Rectified Linear Unit (ReLU)

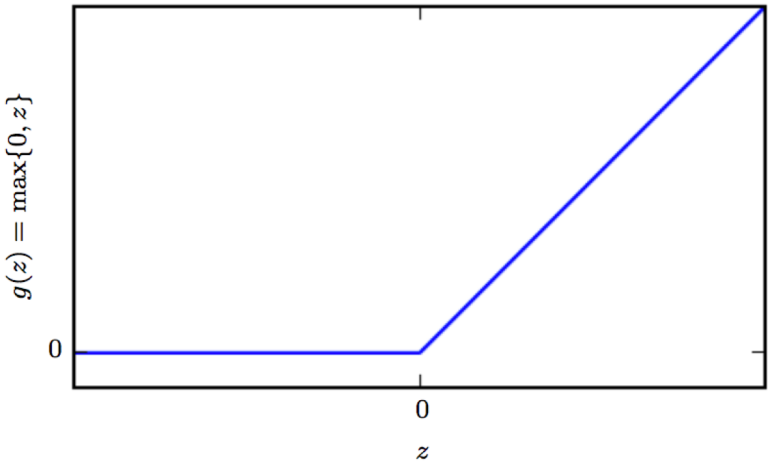
[Jarette et al., 2009]

$g(z) = \max\{0, z\}$



Affine transformation +
nonlinear activation function g

$h = g(W^T x + c)$



[DeepLearning Fig. 6.3]

Architecture design

- Decide on the number of units, the depth of network, and how units should be connected to each other
- Experimentation and tuning with validation set

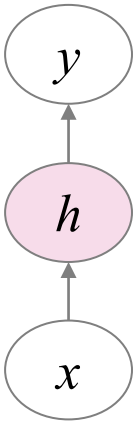
Network design

- Cost function
- Output units
- Hidden units
- Architecture

Two modes of network

- Feedforward: Predicting the values of output variables
- Learning: Training the values of network parameters
  - E.g., using backpropagation

Forward path: e.g., XOR



$$f(x; W, c, w, b) = f^{(2)}(g(f^{(1)}(x)))$$

$$= w^{\mathrm{T}} \max\{0, W^{\mathrm{T}}x + c\} + b$$

Forward path: e.g., XOR

$$f(x; W, c, w, b) = f^{(2)}(f^{(1)}(x)) = w^{\mathrm{T}} \max\{0, W^{\mathrm{T}}x + c\} + b$$

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad x = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \quad W^{T}x = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 2 \\ 0 & 1 & 1 & 2 \end{bmatrix}$$

$$c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$w = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \quad xW+c = \begin{bmatrix} 0 & 1 & 1 & 2 \\ -1 & 0 & 0 & 1 \end{bmatrix} \quad g(xW+c) = \begin{bmatrix} 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$b = 0$$

$$w^{\mathrm{T}}g(xW+c) = \begin{bmatrix} 1 & -2 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix}$$

Back-propagation

- Start with untrained network
- Given example X → Y
- Pass example input X through the network
- Compute the predicted output Z at the output layer
- Compare the prediction Z with true output Y
- Compute error (loss) between Z and Y
- Compute gradients for network parameters
- Adjust network parameters to reduce the error using gradient descent

Back-propagation

- Loss function J(w)
- At each iteration *m*, update network weights w

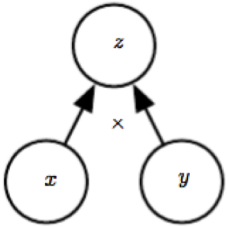
$$w(m+1) = w(m) + \nabla w(m) \quad \text{where} \quad \nabla w = -\eta \frac{\partial J}{\partial w}$$

Learning rate

Computation graph

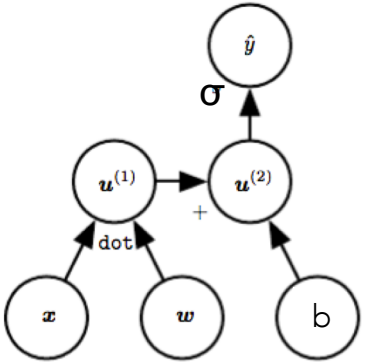
- Node represents variable of type scalar, vector, matrix, tensor, etc.
- Operator takes input nodes (variables) and returns output variable

Examples of computational graphs



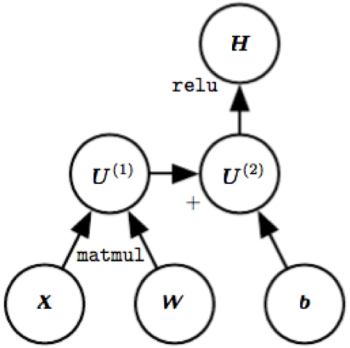
(a)

z = xy



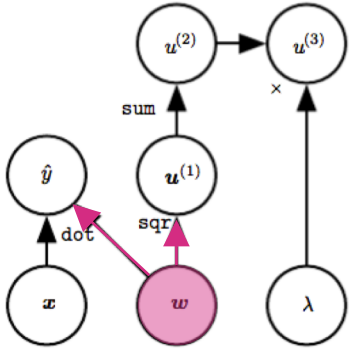
(b)

Logistric regression
y^=**σ**(wx + b)



(c)

H=max{0, WX+b}



(d)

y^=wx
λΣw^2

[DL Fig 6.8]

Chain rule of calculus

z = f(y)
y = g(x)

z = f(g(x))

$$\frac{\partial z}{\partial x} =$$

x, y in R
f: R → R
g: R → R

x in R^m, y in R^n
f: R^n → R
g: R^m → R^n

Chain rule of calculus

z = f(y)
y = g(x)

z = f(g(x))

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

x, y in R
f: R → R
g: R → R

x in R^m, y in R^n
f: R^n → R
g: R^m → R^n

Chain rule of calculus

z = f(y)
y = g(x)

z = f(g(x))

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

x, y in R
f: R → R
g: R → R

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

x in R^m, y in R^n
f: R^n → R
g: R^m → R^n

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^T \nabla_y z$$

Gradient vector of f

n x m Jacobian matrix of g

Chain rule applied to tensors

Tensor X, Y
z = f(Y)
Y = g(X)

Gradient of a value z w.r.t. a tensor X

$$\nabla_X z = \sum_j (\nabla_X Y_j) \frac{\partial z}{\partial Y_j}$$

$(\nabla_X z)_i$ Abstracting tensor indices into single variable i

Simple forward propagation

Assume that nodes have been sorted in the order of computation

for $i = 1, \ldots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for

Input of length n_i
Single scalar $u^{(n)}$

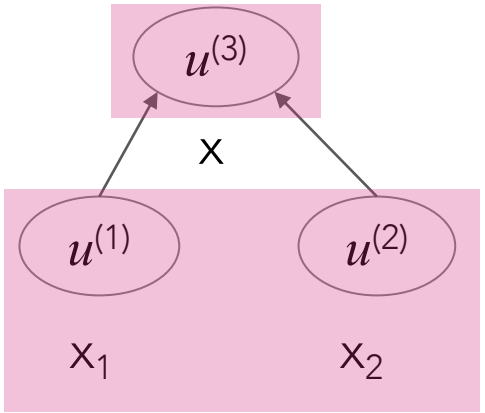
for $i = n_i + 1, \ldots, n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for

parents

return $u^{(n)}$

[DL Algorithm 6.1]

$Pa(u^{(3)}) = \{ u^{(1)}, u^{(2)} \}$



Simple backpropagation

Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network.

Initialize `grad_table`, a data structure that will store the derivatives that have been computed. The entry `grad_table`$[u^{(i)}]$ will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.

`grad_table`$[u^{(n)}] \leftarrow 1$

for $j = n - 1$ down to 1 **do**

i: children of j

The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:

`grad_table`$[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})}$ `grad_table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$

end for

return {`grad_table`$[u^{(i)}] \mid i = 1, \ldots, n_i$}

[DL Algorithm 6.2]

Forward propagation

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$ — Input layer

for $k = 1, \ldots, l$ **do**

$\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$ — Affine transformation

$\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$ — Activation function

} Hidden layers

end for

$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$ — Output layer

$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$ — Loss + regularizer

[DL Algorithm 6.3]

Back-propagation

After the forward computation, compute the gradient on the output layer:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$ $\quad a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$

Compute gradients on weights and biases (including the regularization term, where needed):

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$

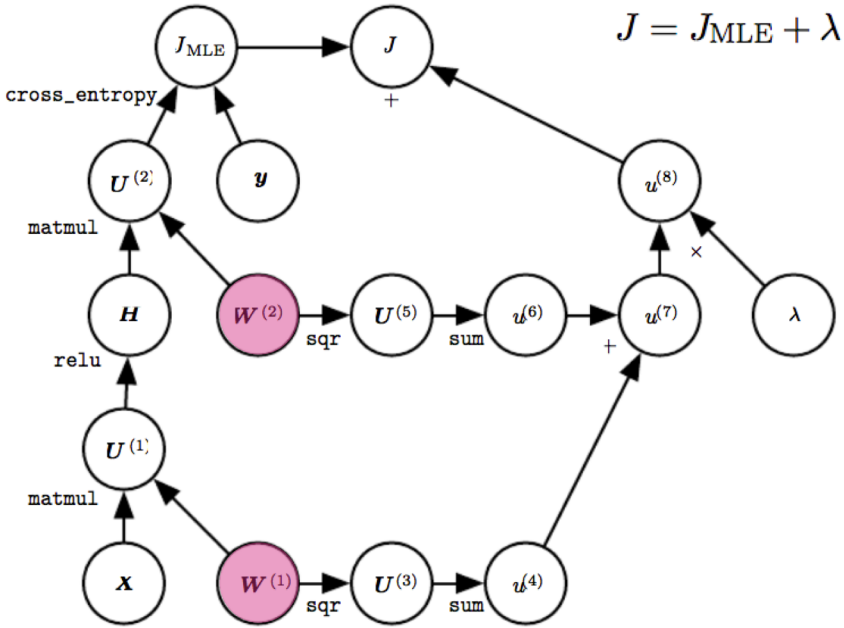
end for

[DL Algorithm 6.4]

Symbolic representations

- Symbol-to-number differentiation
  - Computational graph + a set of numerical values
  - Torch
  - Caffe
- Symbol-to-symbol derivatives
  - Computational graph + symbolic representation of derivatives (enables computation of higher derivatives)
  - Theano
  - TensorFlow

Computation graph for 1-layer MLP



$$J = J_{\text{MLE}} + \lambda \left(\sum_{i,j} \left(W_{i,j}^{(1)} \right)^2 + \sum_{i,j} \left(W_{i,j}^{(2)} \right)^2 \right) \tag{6.56}$$

Weight decay regularizer prefers: smaller weights

Figure 6.11: The computational graph used to compute the cost to train our example of a single-layer MLP using the cross-entropy loss and weight decay.

[DL Ch. 6.5.7]

General back-propagation

- get_operation(V): returns the operation that computes V, e.g., function pointer
- get_consumers(V, G): returns children of node V in computational graph G
- get_inputs(V, G): returns parents of V in G
- `op.bprop(inputs)`

$$\nabla_X z = \sum_j (\nabla_X Y_j) \frac{\partial z}{\partial Y_j}$$

`op.f(inputs)`$_j$

General back-propagation outer loop

Algorithm 6.5 The outermost skeleton of the back-propagation algorithm. This portion does simple setup and cleanup work. Most of the important work happens in the `build_grad` subroutine of algorithm 6.6

Require: $\mathbb{T}$, the target set of variables whose gradients must be computed.
Require: $\mathcal{G}$, the computational graph
Require: z, the variable to be differentiated
Let $\mathcal{G}'$ be $\mathcal{G}$ pruned to contain only nodes that are ancestors of z and descendents of nodes in $\mathbb{T}$.
Initialize `grad_table`, a data structure associating tensors to their gradients
`grad_table`$[z] \leftarrow 1$
for $\mathbf{V}$ in $\mathbb{T}$ **do**
  `build_grad`$(\mathbf{V}, \mathcal{G}, \mathcal{G}', \texttt{grad\_table})$
end for
Return `grad_table` restricted to $\mathbb{T}$

General back-propagation inner loop

Algorithm 6.6 The inner loop subroutine `build_grad`$(\mathbf{V}, \mathcal{G}, \mathcal{G}', \texttt{grad\_table})$ of the back-propagation algorithm, called by the back-propagation algorithm defined in algorithm 6.5.

Require: $\mathbf{V}$, the variable whose gradient should be added to $\mathcal{G}$ and `grad_table`
Require: $\mathcal{G}$, the graph to modify
Require: $\mathcal{G}'$, the restriction of $\mathcal{G}$ to nodes that participate in the gradient
Require: `grad_table`, a data structure mapping nodes to their gradients

```
if V is in grad_table then
  Return grad_table[V]
end if
i ← 1
for C in get_consumers(V, G') do
  op ← get_operation(C)
  D ← build_grad(C, G, G', grad_table)
  G^(i) ← op.bprop(get_inputs(C, G'), V, D)
  i ← i + 1
end for
G ← Σ_i G^(i)
grad_table[V] = G
Insert G and the operations creating it into G
Return G
```

Avoid redundant computation

children

parents

$$\nabla_X z = \sum_j (\nabla_X Y_j) \frac{\partial z}{\partial Y_j}$$

`op.f(inputs)`$_j$

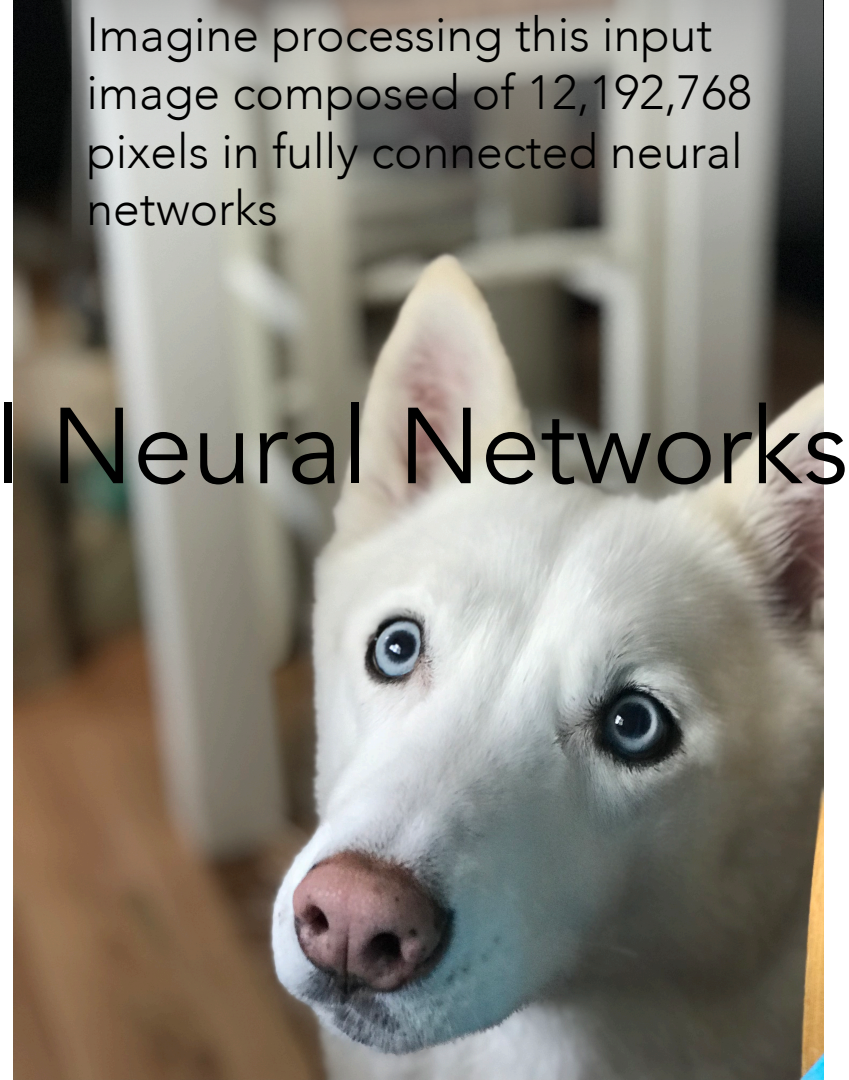
Computational cost of backprop

- Forward propagation stage:
  - O(number of weights) matrix multiplications
- Backward propagation state:
  - O(number of weights) matrix multiplications
  - Memory cost $O(mn_h)$ where m is the number of examples in minibatch; n_h, number of hidden units

Hyperparameters & validation sets

- Tuning parameters that we can use to control learning algorithms
- Setting that is inappropriate to learn from training set
- Validation sets: part of training data, e.g., 80% training set for learning model parameters, 20% validation set for tuning hyperparameters

Convolutional Neural Networks



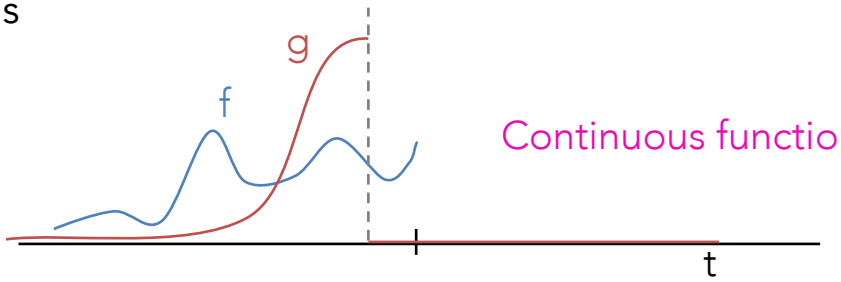
Convolutional networks [LeCun 1989]

are neural networks for processing data shaped in grid-like topology that use convolution in place of matrix multiplication in some layers.

Convolution

Convolution is an operator between two functions f and g of a real-valued argument t, defining an integral of piece-wise multiplication of f and g as one of the functions, g, is shifted over the other function f

e.g., f is noisy sensor reading; g is a weighting function, s(t) = f*g(t) is weighted average of sensor reading—higher weights for recent readings



Continuous functions

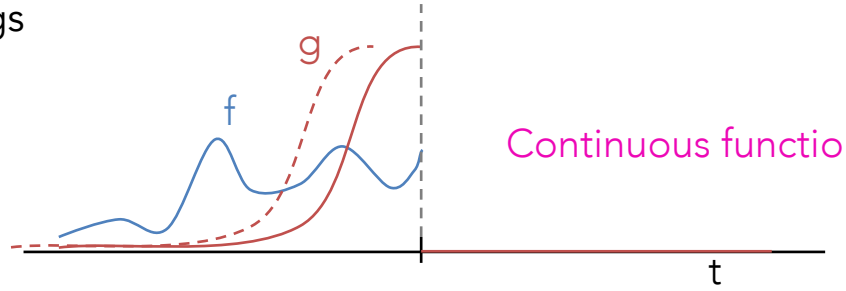
$$s(t) = (f * g)(t) = \int_i f(i)g(t-i)di$$

age shift over values of t

Convolution

Convolution is an operator between two functions f and g of a real-valued argument t, defining an integral of piece-wise multiplication of f and g as one of the functions, g, is shifted over the other function f

e.g., f is noisy sensor reading; g is a weighting function, s(t) = f*g(t) is weighted average of sensor reading—higher weights for recent readings



Continuous functions

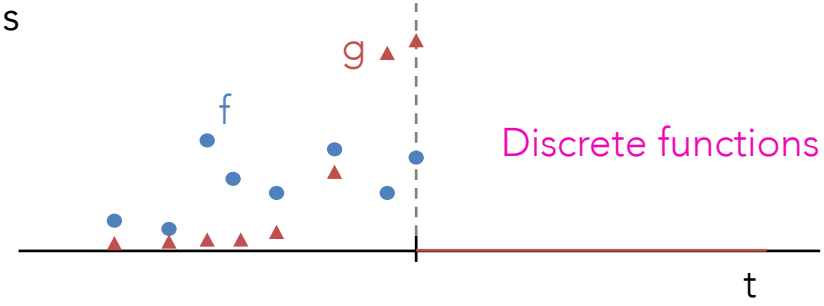
$$s(t) = (f * g)(t) = \int_i f(i)g(t-i)di$$

age shift over values of t

Convolution

Convolution is an operator between two functions f and g of a real-valued argument t, defining a sum of piece-wise multiplication of f and g as one of the functions, g, is shifted over the other function f

e.g., f is noisy sensor reading; g is a weighting function, s(t) = f*g(t) is weighted average of sensor reading—higher weights for recent readings



$$s(t) = (f * g)(t) = \sum_{i=-\infty}^{\infty} f(i)g(t-i)$$

age shift over values of t

Convolution in CNNs

Convolution is an operator between input (data) tensor I and kernel (parameters) tensor K of multiple real-valued arguments, defining a sum of piece-wise multiplication of I and K as the kernel K is shifted over input data I

e.g., 2-D image input data

input

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

kernel

shift over values of i and j

Convolution is commutative

Convolution is an operator between input (data) tensor I and kernel (parameters) tensor K of multiple real-valued arguments, defining a sum of piece-wise multiplication of I and K as the kernel K is shifted over input data I

e.g., 2-D image input data

input

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

kernel

shift over values of i and j

kernel

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

input

Kernel is flipped

Cross-correlation of convolution

Commonly also referred to as "convolution"

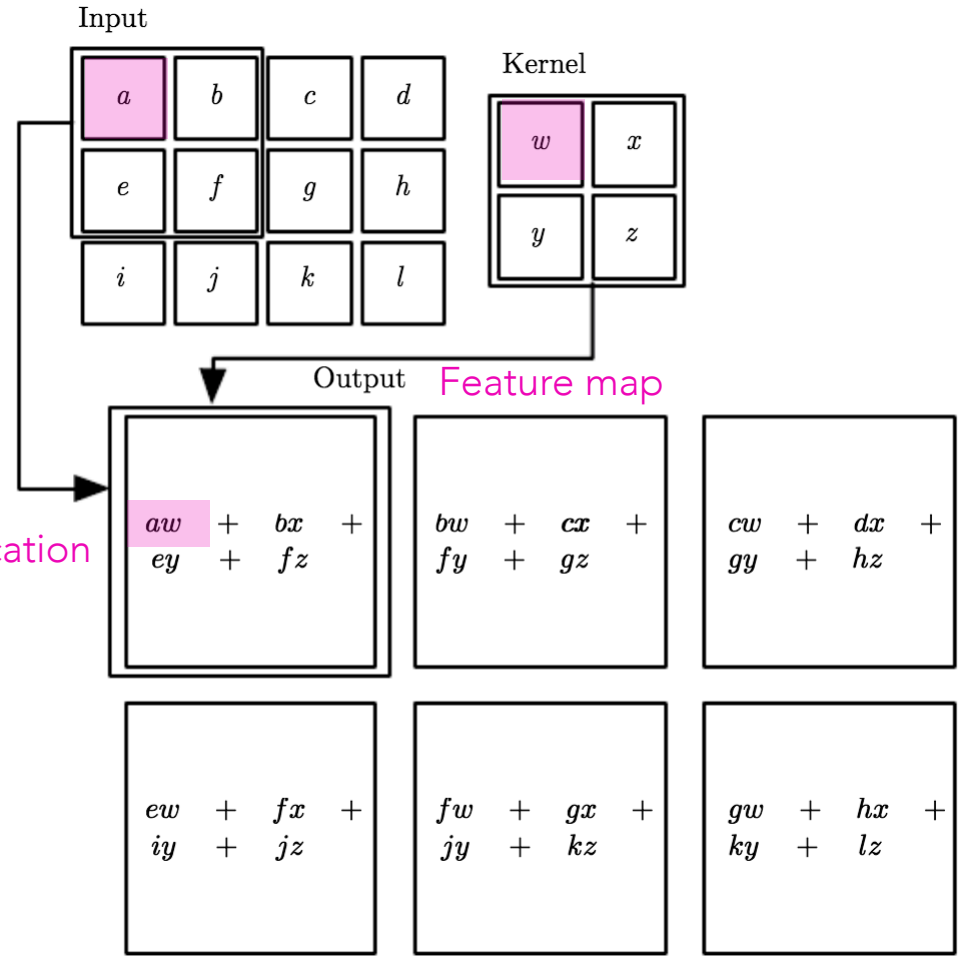
input

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n)$$

kernel

Kernel is not flipped

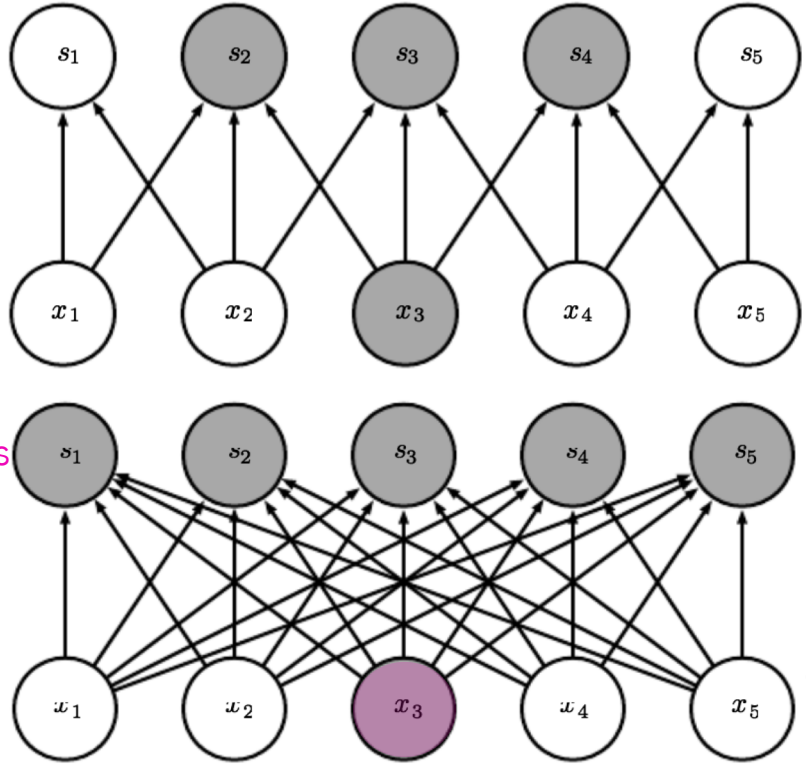
Example of 2-D convolution [Fig 9.2]



Properties of convolution in neural networks

- Sparse interactions
- Parameter sharing
- Equivariant representations

Sparse interactions



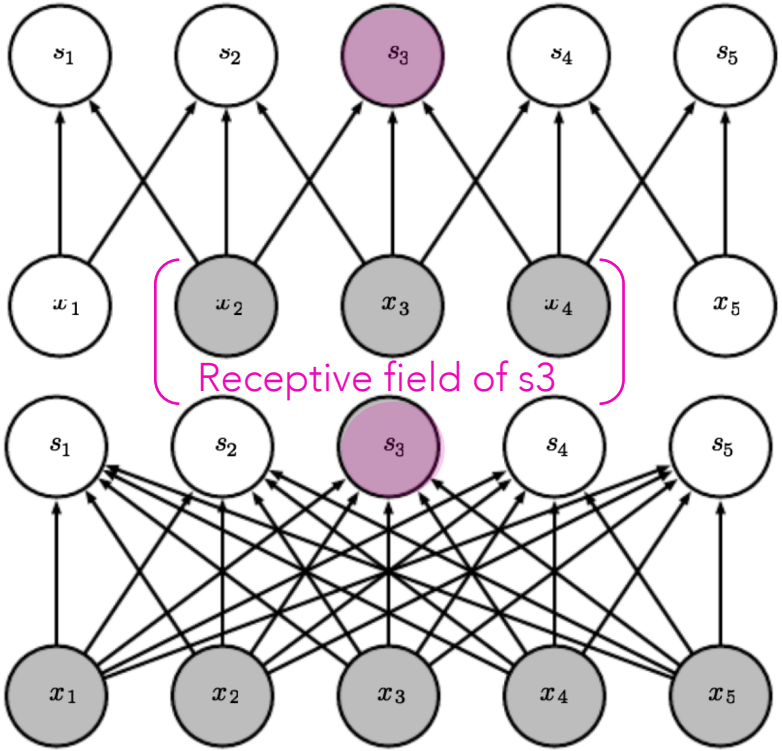
Convolution
O(k x n)

Matrix multiplication
O(m x n)

Output nodes affected by input node x_3

[Fig. 9.2]

Sparse interactions

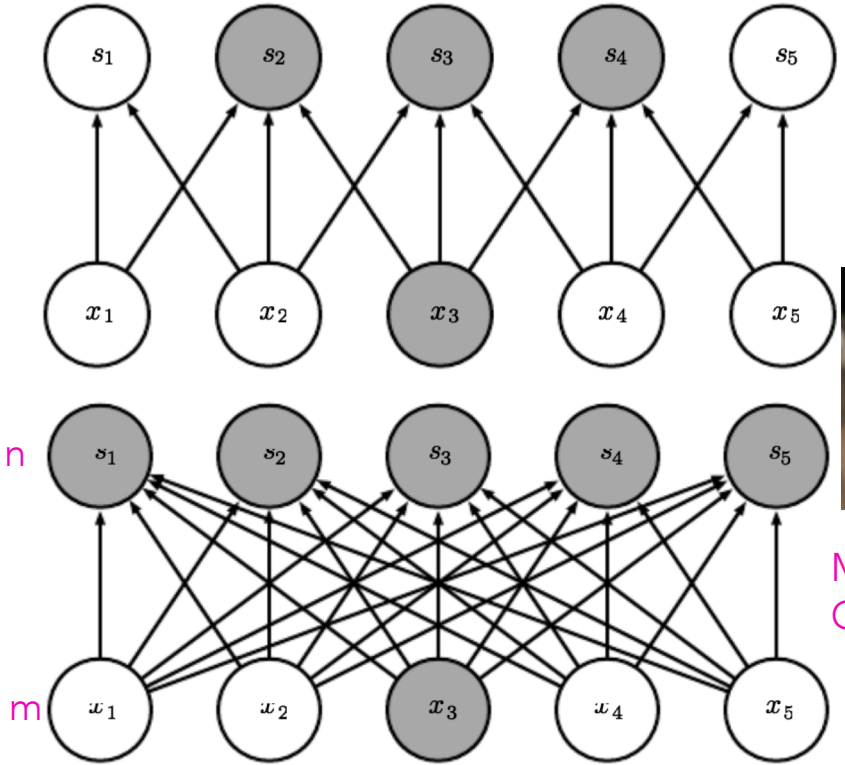


Matrix multiplication
O(m x n)

Input notes affecting output node s_3

[Fig. 9.3]

Sparse interactions



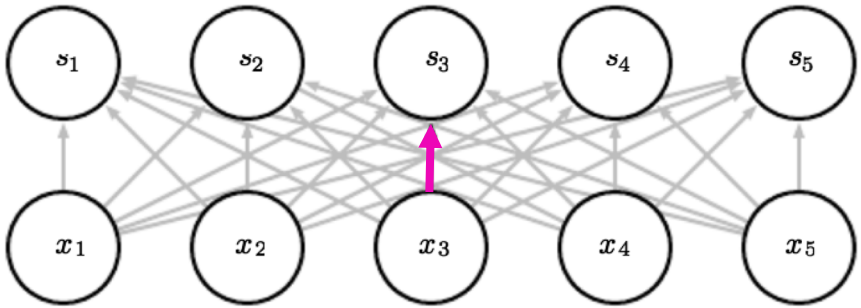
Convolution
O(k x n)

k= ~ 100s

m = 12,192,768

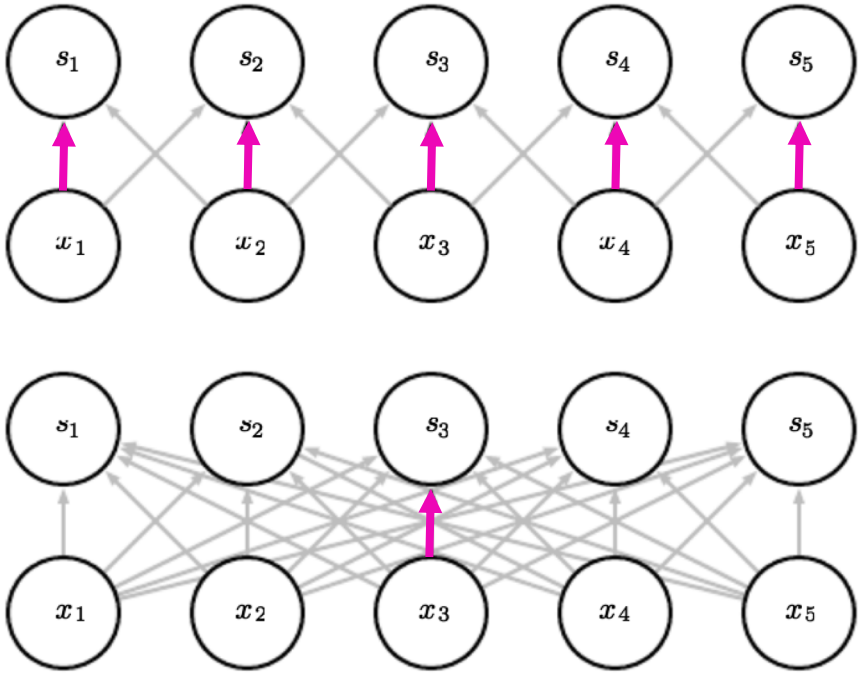
Matrix multiplication
O(m x n)

Parameter sharing



"Tied" weights
= Used once

Parameter sharing



Learn weights once
= Used for every location

Reduce memory requirement for storing parameters

"Tied" weights
= Used once

[Fig 9.5]

Equivariant to translation

Function f(x) is equivariant to function g if:

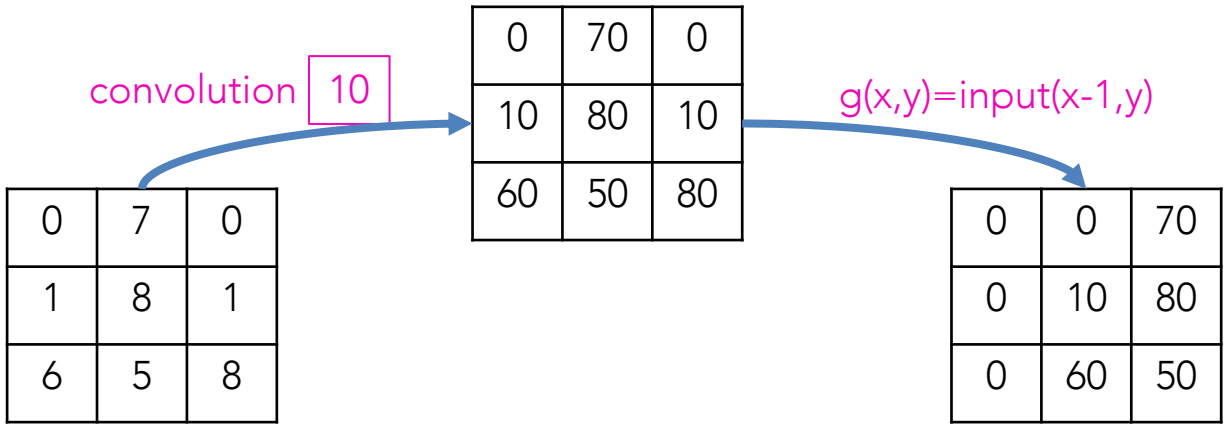
$$f(g(x)) = g(f(x))$$

Convolution is equivariant to translation, i.e., if the input changes, the output changes the same way.

Equivariant to translation

Convolution f is equivariant to translation function g if:

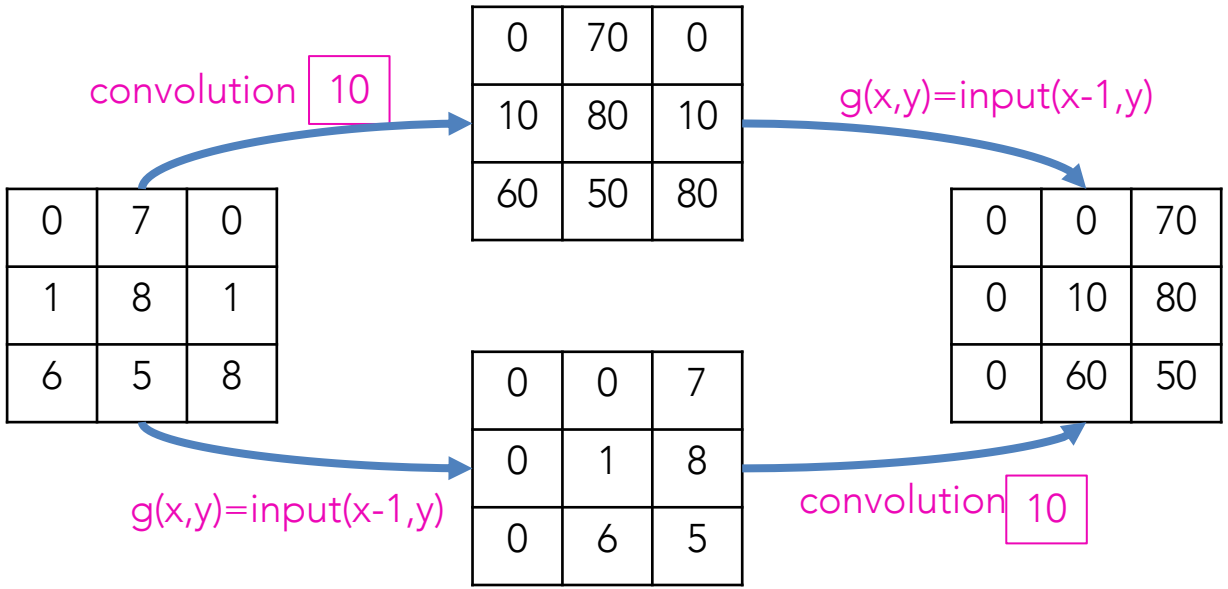
$$f(g(x)) = g(f(x))$$



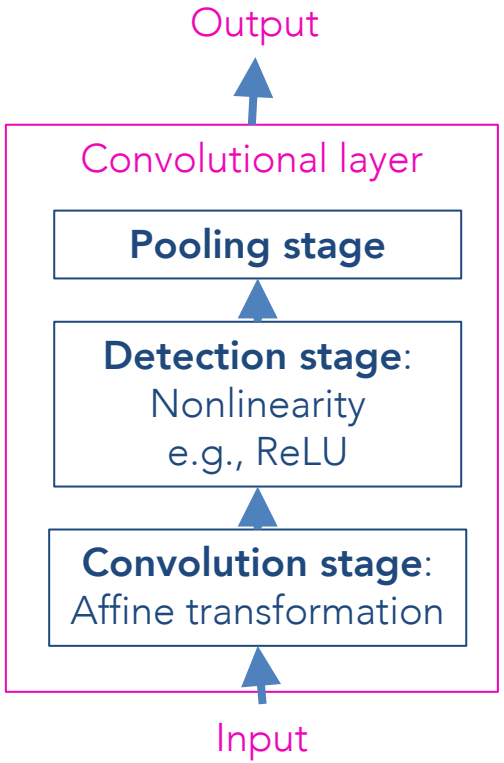
Equivariant to translation

Convolution f is equivariant to translation function g if:

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

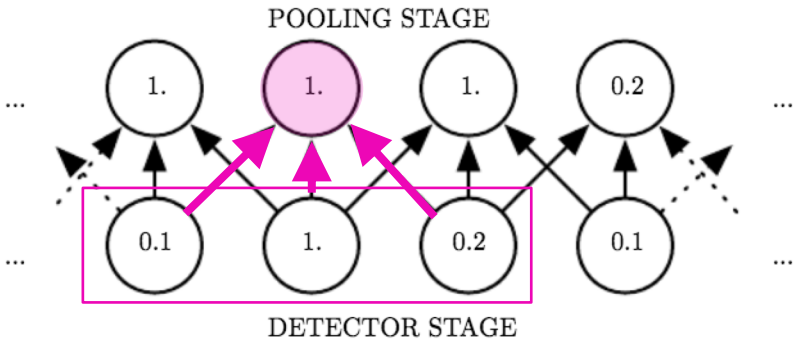


Components of convolutional layer



Pooling function: to refine the output

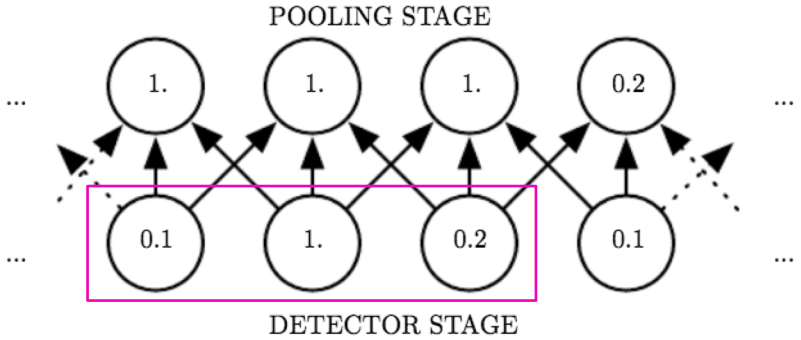
Pooling

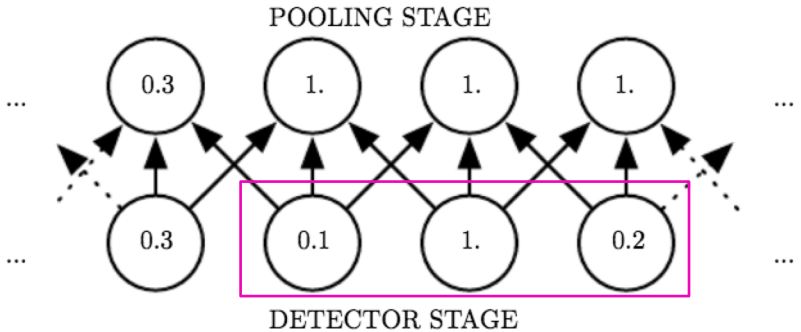


- Max pooling [Zhou & Chellappa, 1988]: report the maximum output among a rectangular neighborhood
- Average pooling
- L^2 norm pooling
- Weighted average pooling

Pooling over spatial regions

Invariance to translation (small shift)

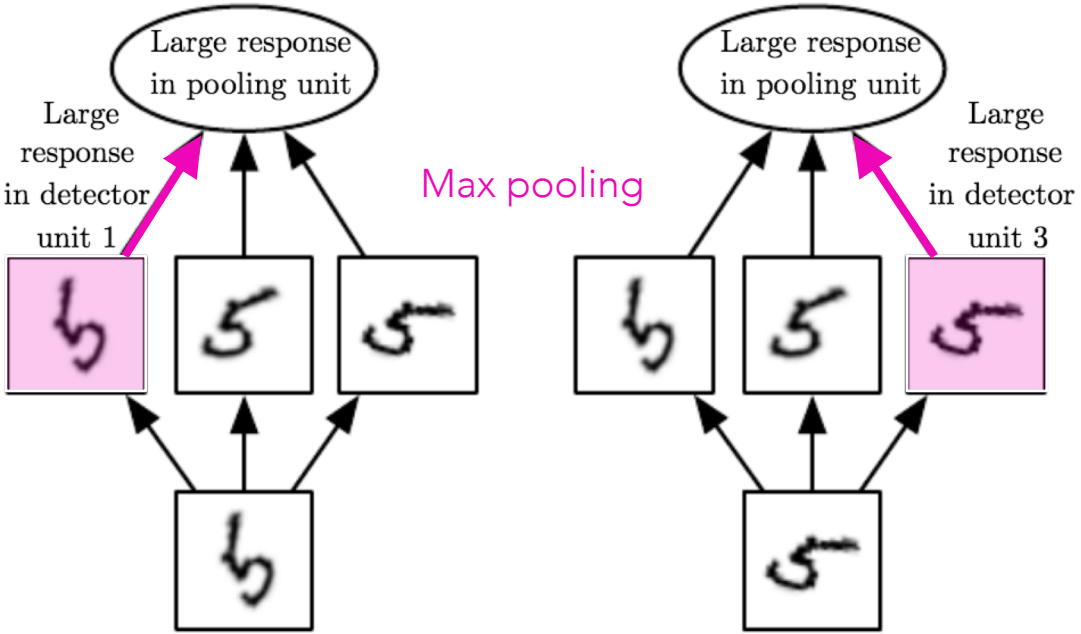




[Fig 9.8]

Pooling over features

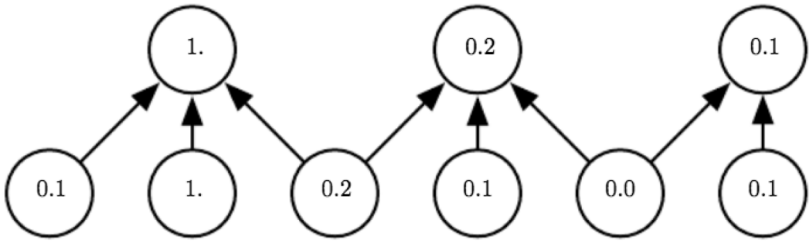
Invariance to features, e.g., rotation



[fig 9.9]

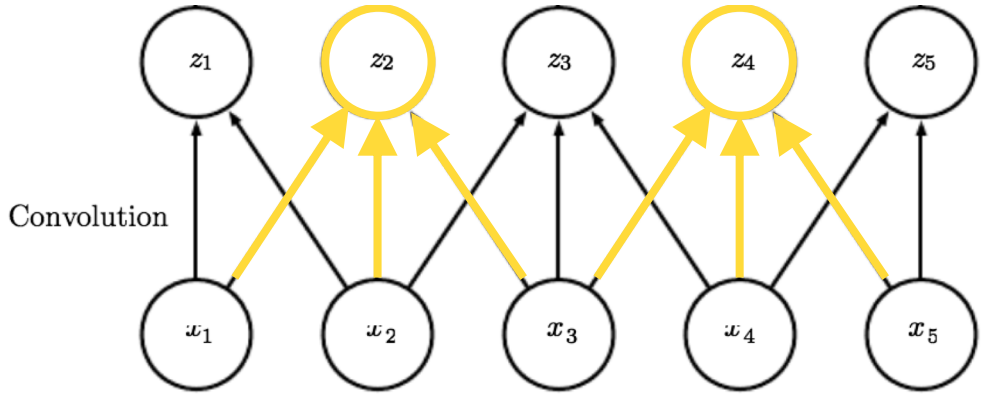
Pooling also provides

- Invariance to translation or other transformations
- Efficiency via downsampling—i.e., fewer pooling units than detection units
- Support for variable-size input by varying the size of offset between pooling regions to meet fixed size output



[Fig 9.10]

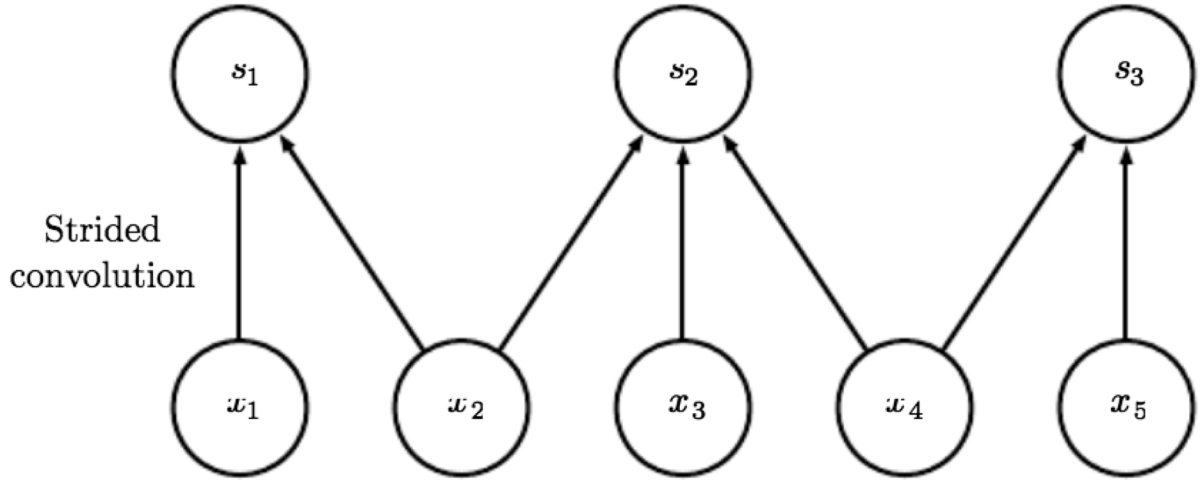
Convolution with a stride



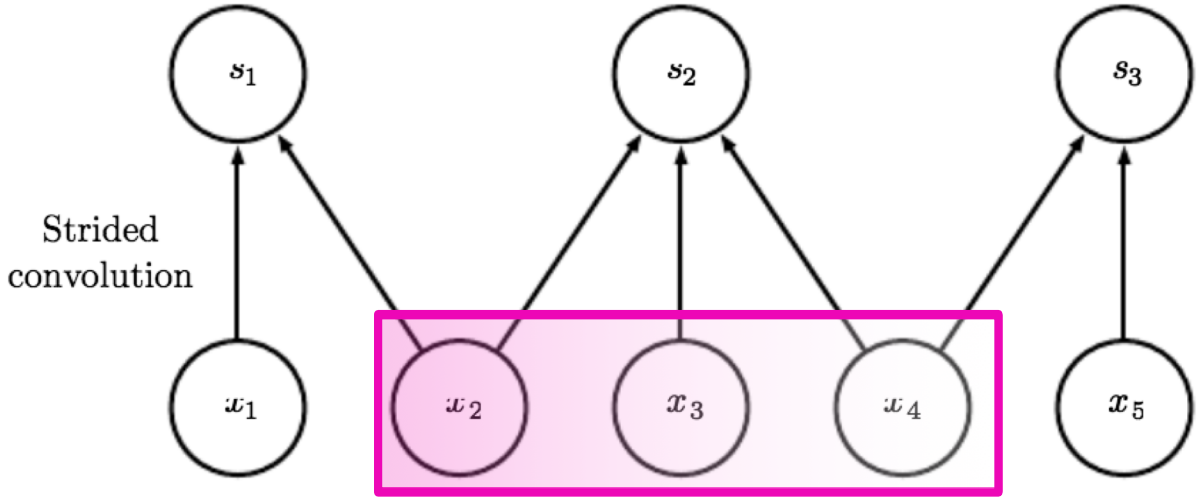
Computationally wasteful

[Fig 9.12]

Convolution with a stride



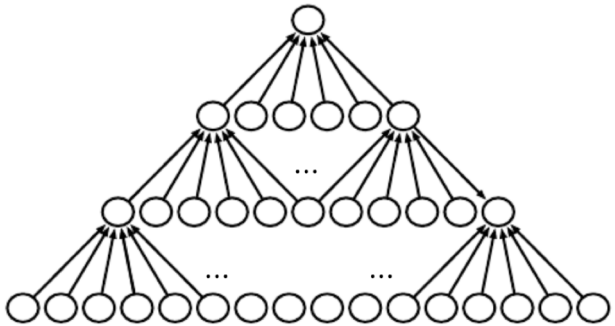
Convolution with a stride



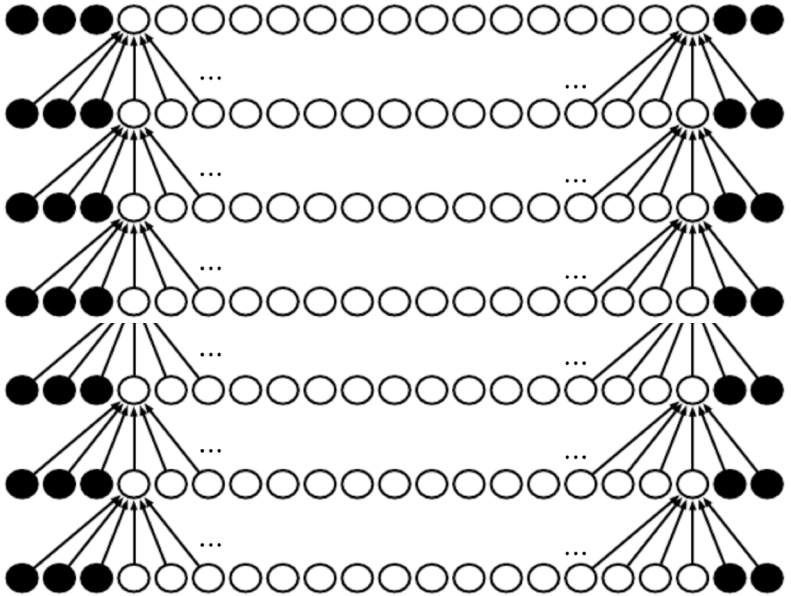
Stride of size 2

Zero padding

Convolution shrinks the network size …



No zero padding,
valid convolution



Full zero padding,
same convolution

Example: Image classification

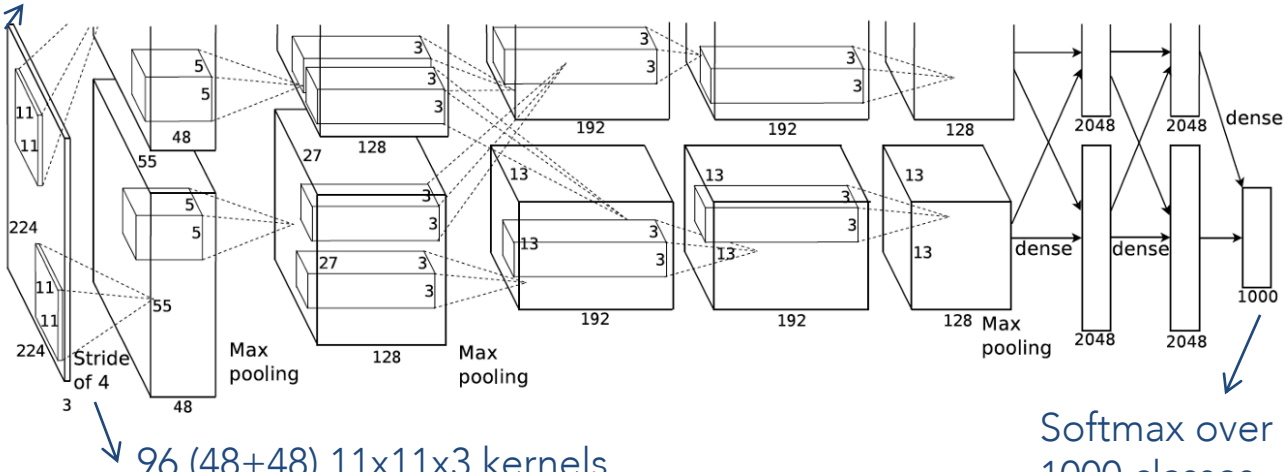
AlexNet [Krizhevsky et al., 2012]

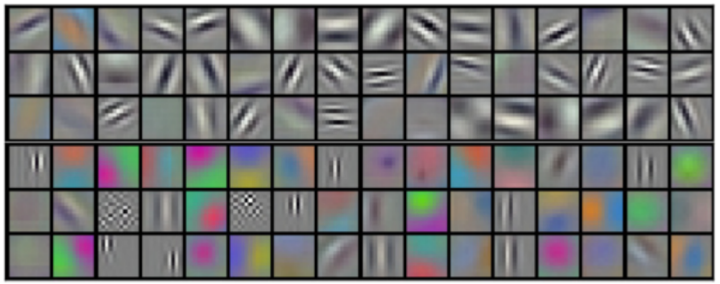
- ImageNet Large-Scale Visual Recognition Challenge (ILSVRC): 1.2M training, 50K validation, and 150K testing images
- Fixed-size input
- 5 Convolutional layers + 3 fully connected layers
- 2 GPUs
- 1000 classes image classification
- Reducing overfitting by:
  - Data augmentation (image translation, horizontal reflection, intensity alteration using PCA)
  - Dropout (zero out output with some probability, e.g., 0.5)

AlexNet

[Krizhevsky et al., 2012]

Input: 224x224x3 (RGB)





[Fig 3]

Regularization

Any modification we make to a learning algorithm to reduce “generalization error” as opposed to “training error”

Parameter norm penalties

- L^2 norm regularization (aka Tikhonov regularization or ridge regression)

$$\Omega(\theta) = \frac{1}{2}\|w\|_2^2$$

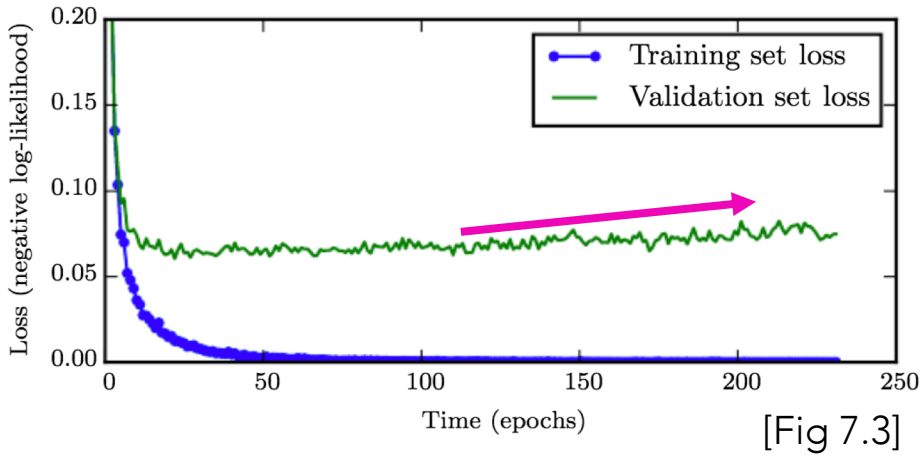
- L^1 norm regularization

$$\Omega(\theta) = \|w\|_2 = \sum_i |w_i|$$

Data augmentation

- Train on more data
- Invariant to various transformations
- Effective in image classification, object recognition
- But not for all problem domains
- Examples
  - Rotation
  - Horizontal, vertical flip
  - Translation

Early stopping



[Fig 7.3]

While training error is decreased, validation error may go up due to overfitting

During training, keep track of the parameters that give the lowest validation error, return them at the end

Batch normalization

[Ioffe & Szegedy, 2015]

- Internal covariance shift: the distribution of input to each layer changes as the parameters of previous layer changes
- Normalize layer inputs for each mini-batch
- Advantages:
  - higher learning rate, faster training
  - No dropout needed
  - Accepted as part of common architecture

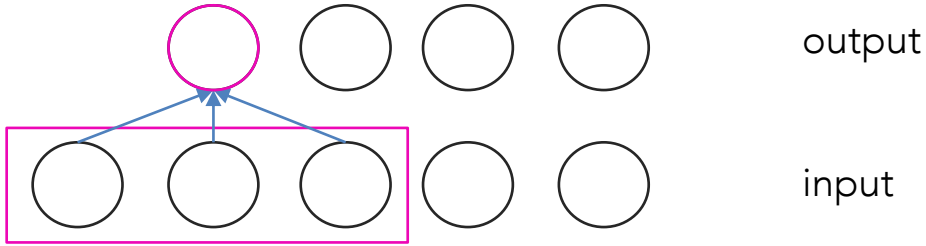
Other methods for regularization

- Noise injection, label smoothing
- Parameter sharing
  - CNN
- Bootstrapping aggregating (aka bagging) – ensemble or model averaging
- Dropout [Srivastava et al., 2014]
  - Ignore random output with some probability, meaning not used for learning at all
  - Popular earlier but not so much currently

Recurrent Neural Networks

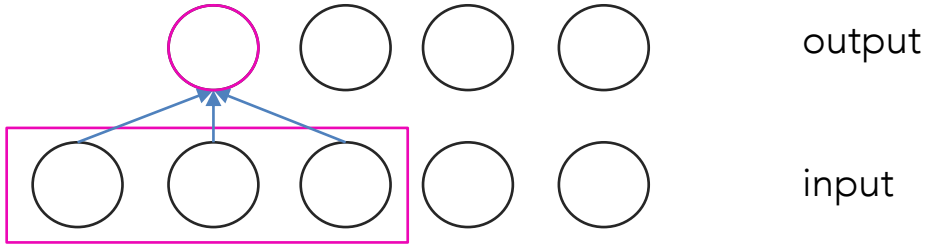
Whereas CNNs process a grid of values, Recurrent neural networks (RNNs) (Rumelhart et al., 1986) model and process sequential data.

Processing sequential data

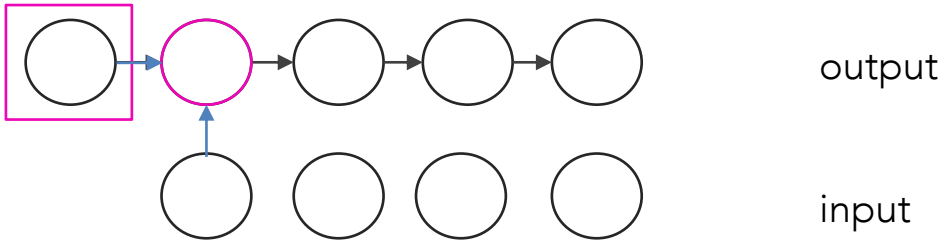


In 1-D CNN, an output is a function of neighboring inputs

Processing sequential data

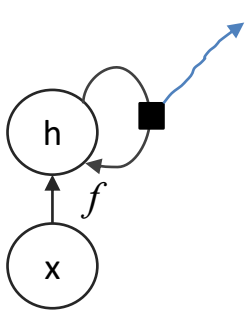


In 1-D CNN, an output is a function of neighboring inputs



In RNN, an output is a function of current input & previous output

RNN for processing sequence

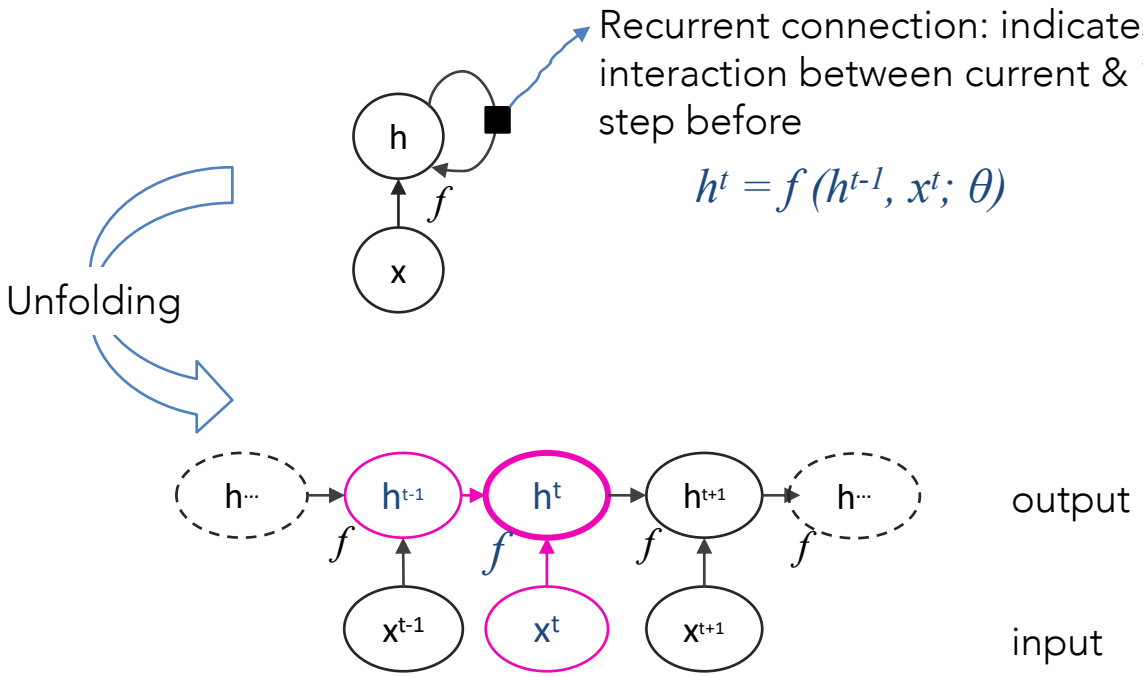


Recurrent connection: indicates interaction between current & 1 time step before

$$h^t = f(h^{t-1}, x^t; \theta)$$

In RNN, an output is a function of current input & previous output

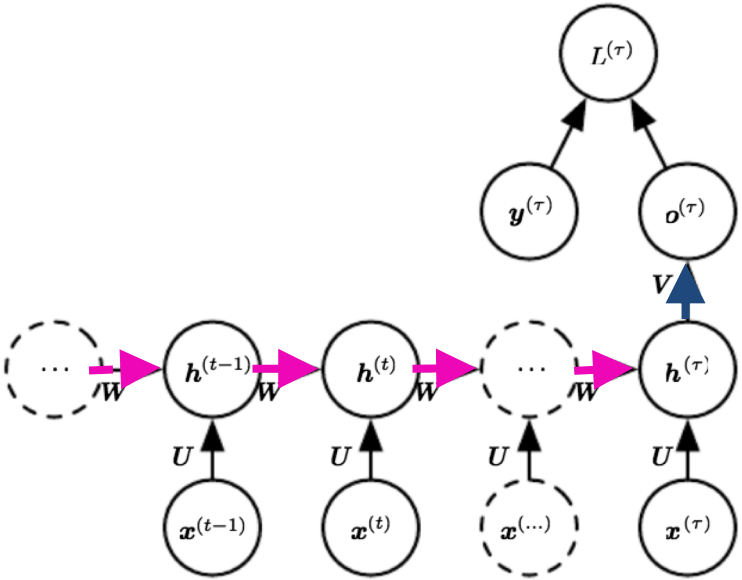
RNN for processing sequence



In RNN, an output is a function of current input & previous output

RNN Patten I

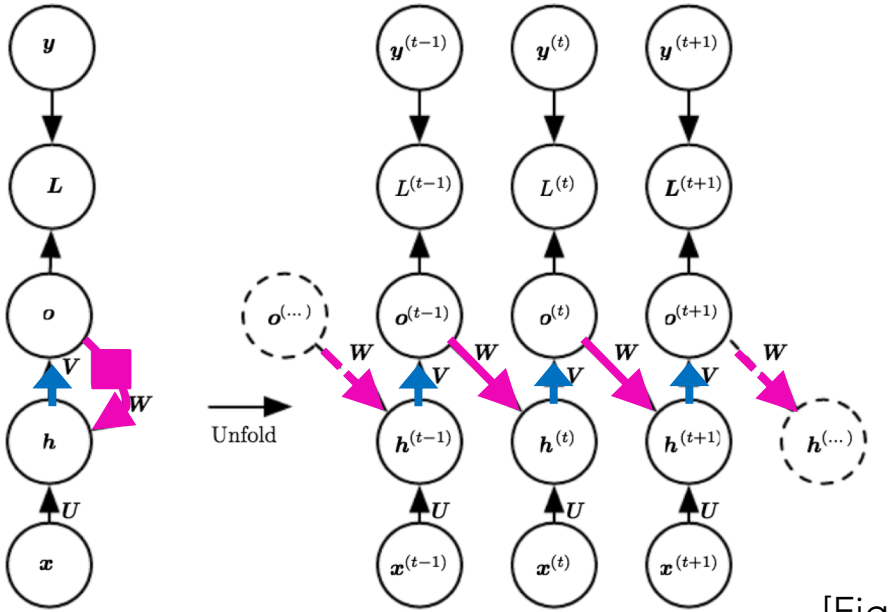
Recurrent connections between hidden units; output once at the end



[Fig. 10.5]

RNN Pattern II

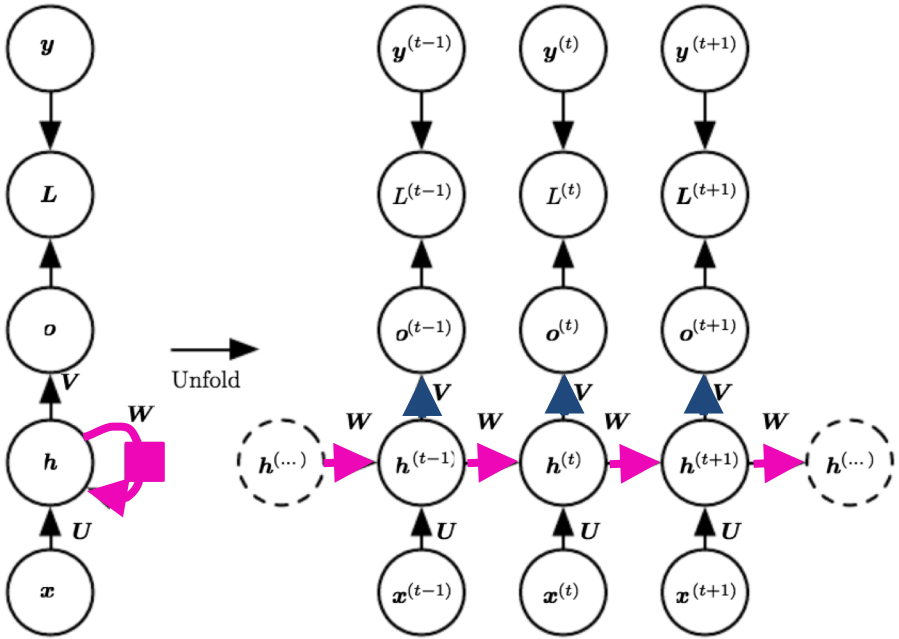
Recurrent connections between the current output and the next hidden unit; output every time step



[Fig. 10.4]

RNN Pattern III

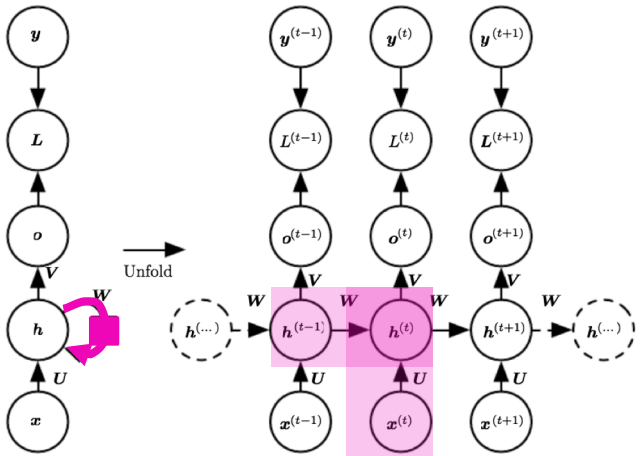
Recurrent connections between hidden units; output every time step



[Fig 10.3]

Forward propagation in RNN

Recurrent connections between hidden units; output every time step



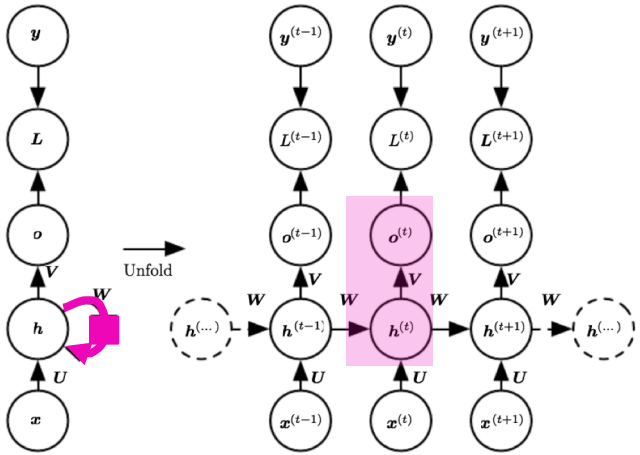
[Fig 10.3]

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)},$$
$$h^{t} = \tanh(a^{(t)}),$$

Activation (previous hidden unit + current input)

Forward propagation in RNN

Recurrent connections between hidden units; output every time step



[Fig 10.3]

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)},$$

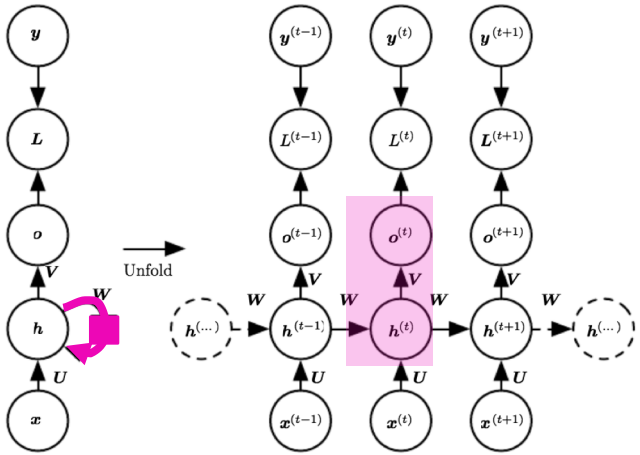
$$h^{t} = \tanh(a^{(t)}),$$

$$o^{(t)} = c + Vh^{(t)},$$

From hidden unit to output unit; e.g., unnormalized log probabilities of all values of a discrete variable such as words or characters

Forward propagation in RNN

Recurrent connections between hidden units; output every time step



[Fig 10.3]

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)},$$

$$h^{t} = \tanh(a^{(t)}),$$

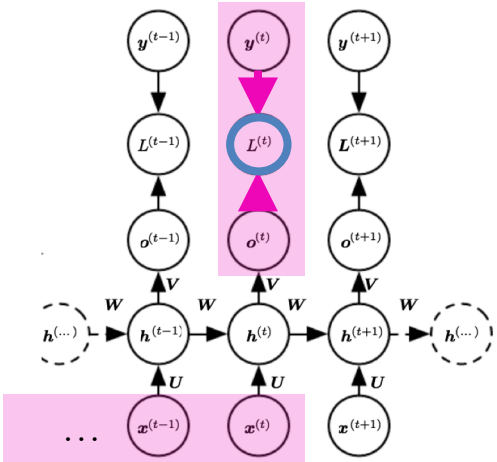
$$o^{(t)} = c + Vh^{(t)},$$

$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$

Softmax to get normalized probabilities

Back-propagation through time (BPTT)

An example RNN where input and out sequences have the same length

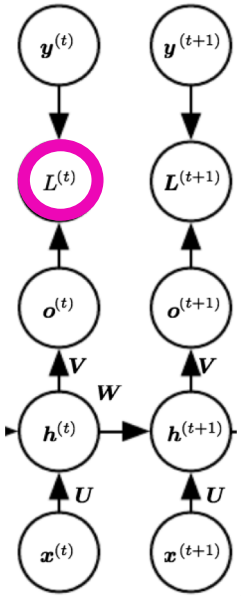


$$L\left(\left\{x^{(1)},\dots,x^{(\tau)}\right\},\left\{y^{(1)},\dots,y^{(\tau)}\right\}\right)$$
$$= \sum_t L^{(t)}$$
$$= -\sum_t \log p_{\text{model}}\left(y^{(t)} \middle| \left\{x^{(1)},\dots,x^{(t)}\right\}\right)$$

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)},$$
$$h^t = \tanh(a^{(t)}),$$
$$o^{(t)} = c + Vh^{(t)},$$
$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$
$$L^{(t)} = -\log p_{\text{model}}\left(y^{(t)} \middle| \left\{x^{(1)}, \ldots, x^{(t)}\right\}\right)$$

BPTT

Model parameters: U, V, W, b, and c
Nodes: $x^{(t)}$, $h^{(t)}$, $o^{(t)}$, and $L^{(t)}$

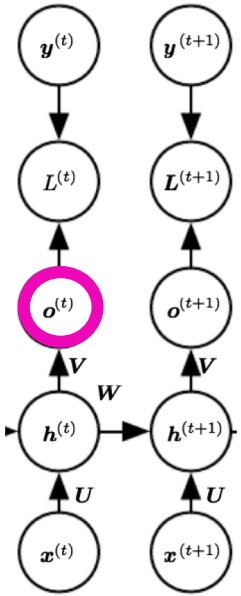


$$\nabla_{L(t)} L = \frac{\partial L}{\partial L^{(t)}} = 1$$

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)},$$
$$h^t = \tanh(a^{(t)}),$$
$$o^{(t)} = c + Vh^{(t)},$$ Input to softmax
$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$
$$L^{(t)} = -\log p_{\text{model}}\left(y^{(t)} \middle| \{x^{(1)}, \ldots, x^{(t)}\}\right)$$

BPTT

Model parameters: U, V, W, b, and c
Nodes: $x^{(t)}$, $h^{(t)}$, $o^{(t)}$, and $L^{(t)}$



$$\nabla_{L(t)} L = \frac{\partial L}{\partial L^{(t)}} = 1$$

$$\left(\nabla_{o(t)} L\right)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - 1_{i,y^{(t)}}$$

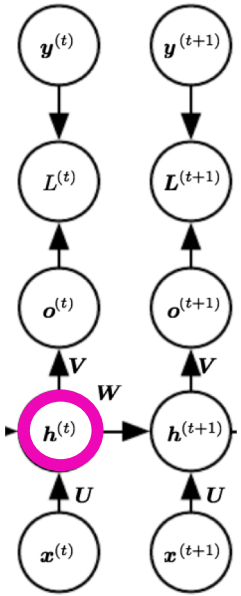
True answer (1 at index i, 0 elsewhere)

Loss between true answer and predicted output

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)},$$
$$h^{t} = \tanh(a^{(t)}),$$ $h^{(t+1)}$ & $o^{(t)}$
$$o^{(t)} = c + Vh^{(t)},$$
$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$
$$L^{(t)} = -\log p_{\text{model}}\left(y^{(t)} \middle| \{x^{(1)}, \ldots, x^{(t)}\}\right)$$

BPTT

Model parameters: U, V, W, b, and c
Nodes: $x^{(t)}$, $h^{(t)}$, $o^{(t)}$, and $L^{(t)}$

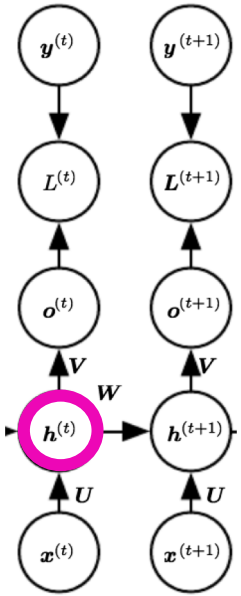


$$\nabla_{h(\tau)} L = V^T \nabla_{o^{(\tau)}} L$$ -- At the final time step (no $h^{(\tau+1)}$)

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)},$$
$$h^{t} = \tanh(a^{(t)}),$$ $h^{(t+1)}$ & $o^{(t)}$
$$o^{(t)} = c + Vh^{(t)},$$
$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$
$$L^{(t)} = -\log p_{\text{model}}\left(y^{(t)} \middle| \{x^{(1)}, \ldots, x^{(t)}\}\right)$$

BPTT

Model parameters: U, V, W, b, and c
Nodes: $x^{(t)}$, $h^{(t)}$, $o^{(t)}$, and $L^{(t)}$



$$\nabla_{h^{(\tau)}} L = V^T \nabla_{o^{(\tau)}} L$$ -- At the final time step (no $h^{(\tau+1)}$)

$$\nabla_{h^{(t)}} L = \left(\frac{\partial h^{(t+1)}}{\partial h^{(t)}}\right)^T (\nabla_{h^{(t+1)}} L) + \left(\frac{\partial o^{(t)}}{\partial h^{(t)}}\right)^T (\nabla_{o^{(t)}} L)$$

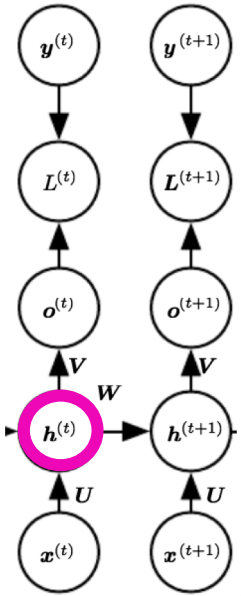
$$h^{(t+1)} = tanh\,(\mathrm{b} + \mathrm{W}h^{(\mathrm{t})} + \mathrm{U}x^{(\mathrm{t+1})})$$

$f(x) = \tanh(x)$
$f'(x) = 1 - \{f(x)\}^2$

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)},$$
$$h^{t} = \tanh(a^{(t)}),$$ $h^{(t+1)}$ & $o^{(t)}$
$$o^{(t)} = c + Vh^{(t)},$$
$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$
$$L^{(t)} = -\log p_{\text{model}}\left(y^{(t)} \middle| \{x^{(1)}, \ldots, x^{(t)}\}\right)$$

BPTT

Model parameters: U, V, W, b, and c
Nodes: $x^{(t)}$, $h^{(t)}$, $o^{(t)}$, and $L^{(t)}$



$$\nabla_{h^{(\tau)}} L = V^T \nabla_{o^{(\tau)}} L$$ -- At the final time step (no $h^{(\tau+1)}$)

$$\nabla_{h^{(t)}} L = \left(\frac{\partial h^{(t+1)}}{\partial h^{(t)}}\right)^T (\nabla_{h^{(t+1)}} L) + \left(\frac{\partial o^{(t)}}{\partial h^{(t)}}\right)^T (\nabla_{o^{(t)}} L)$$

$$= W^T (\nabla_{h^{(t+1)}} L)\,\text{diag}\left(1 - (h^{(t+1)})^2\right) + V^T (\nabla_{o^{(t)}} L)$$

$$h^{(t+1)} = tanh\,(\mathrm{b} + \mathrm{W}h^{(\mathrm{t})} + \mathrm{U}x^{(\mathrm{t}+1)})$$

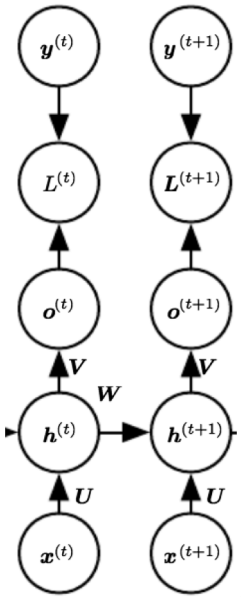
f(x) = tanh(x)
$f'(x) = 1 - \{f(x)\}^2$

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)},$$
$$h^t = \tanh(a^{(t)}),$$
$$o^{(t)} = c + Vh^{(t)},$$
$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$
$$L^{(t)} = -\log p_{\text{model}}\left(y^{(t)} \middle| \{x^{(1)}, \ldots, x^{(t)}\}\right)$$

BPTT

Model parameters: U, V, W, b, and c

Nodes: $x^{(t)}$, $h^{(t)}$, $o^{(t)}$, and $L^{(t)}$



$$\nabla_c L = \sum_t \left(\frac{\partial o^{(t)}}{\partial c}\right)^T \nabla_{o^{(t)}} L = \sum_t \nabla_{o^{(t)}} L \quad (= 1)$$

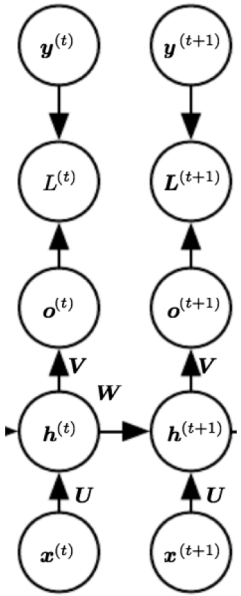
$$\nabla_b L = \sum_t \left(\frac{\partial h^{(t)}}{\partial b^{(t)}}\right)^T \nabla_{h^{(t)}} L = \sum_t \text{diag}\left(1 - (h^{(t)})^2\right) \nabla_{h^{(t)}} L$$

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)},$$
$$h^t = \tanh(a^{(t)}),$$
$$o^{(t)} = c + Vh^{(t)},$$
$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$
$$L^{(t)} = -\log p_{\text{model}}\left(y^{(t)} \middle| \{x^{(1)}, \ldots, x^{(t)}\}\right)$$

BPTT

Model parameters: U, V, W, b, and c

Nodes: $x^{(t)}$, $h^{(t)}$, $o^{(t)}$, and $L^{(t)}$



$$\nabla_c L = \sum_t \left(\frac{\partial o^{(t)}}{\partial c}\right)^T \nabla_{o^{(t)}} L = \sum_t \nabla_{o^{(t)}} L$$

$$\nabla_b L = \sum_t \left(\frac{\partial h^{(t)}}{\partial b^{(t)}}\right)^T \nabla_{h^{(t)}} L = \sum_t \text{diag}\left(1 - (h^{(t)})^2\right) \nabla_{h^{(t)}} L$$

$$\nabla_V L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}}\right) \nabla_V o_i^{(t)} = \sum_t \left(\nabla_{o^{(t)}} L\right) h^{(t)T}$$

$$\nabla_W L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{W^{(i)}} h_i^{(t)} = \sum_t \text{diag}\left(1 - (h^{(t)})^2\right) \left(\nabla_{h^{(t)}} L\right) h^{(t-1)T}$$

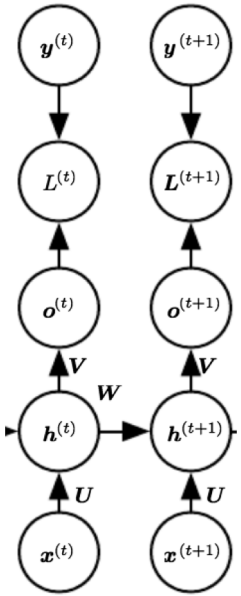
$$\nabla_U L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{U^{(i)}} h_i^{(t)} = \sum_t \text{diag}\left(1 - (h^{(t)})^2\right) \left(\nabla_{h^{(t)}} L\right) x^{(t)T}$$

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)},$$
$$h^t = \tanh(a^{(t)}),$$
$$o^{(t)} = c + Vh^{(t)},$$
$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$
$$L^{(t)} = -\log p_{\text{model}}\left(y^{(t)} \middle| \{x^{(1)}, \ldots, x^{(t)}\}\right)$$

BPTT

Model parameters: U, V, W, b, and c

Nodes: $x^{(t)}$, $h^{(t)}$, $o^{(t)}$, and $L^{(t)}$



$$\nabla_c L = \sum_t \left(\frac{\partial o^{(t)}}{\partial c}\right)^T \nabla_{o^{(t)}} L = \sum_t \nabla_{o^{(t)}} L$$

$$\nabla_b L = \sum_t \left(\frac{\partial h^{(t)}}{\partial b^{(t)}}\right)^T \nabla_{h^{(t)}} L = \sum_t \text{diag}\left(1 - (h^{(t)})^2\right) \nabla_{h^{(t)}} L$$

$$\nabla_V L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}}\right) \nabla_V o_i^{(t)} = \sum_t \left(\nabla_{o^{(t)}} L\right) h^{(t)T}$$

$$\nabla_W L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{W^{(i)}} h_i^{(t)} = \sum_t \text{diag}\left(1 - (h^{(t)})^2\right) \left(\nabla_{h^{(t)}} L\right) h^{(t-1)T}$$

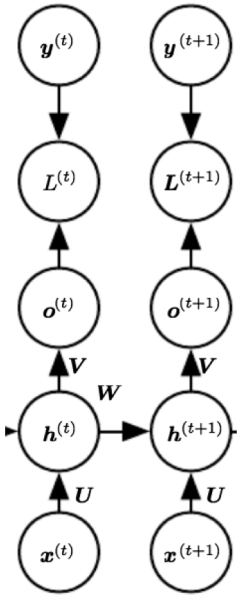
$$\nabla_U L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{U^{(i)}} h_i^{(t)} = \sum_t \text{diag}\left(1 - (h^{(t)})^2\right) \left(\nabla_{h^{(t)}} L\right) x^{(t)T}$$

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)},$$
$$h^t = \tanh(a^{(t)}),$$
$$o^{(t)} = c + Vh^{(t)},$$
$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$
$$L^{(t)} = -\log p_{\text{model}}\left(y^{(t)} \middle| \{x^{(1)}, \ldots, x^{(t)}\}\right)$$

BPTT

Model parameters: U, V, W, b, and c

Nodes: $x^{(t)}$, $h^{(t)}$, $o^{(t)}$, and $L^{(t)}$



$$\nabla_c L = \sum_t \left(\frac{\partial o^{(t)}}{\partial c}\right)^T \nabla_{o^{(t)}} L = \sum_t \nabla_{o^{(t)}} L$$

$$\nabla_b L = \sum_t \left(\frac{\partial h^{(t)}}{\partial b^{(t)}}\right)^T \nabla_{h^{(t)}} L = \sum_t \text{diag}\left(1 - (h^{(t)})^2\right) \nabla_{h^{(t)}} L$$

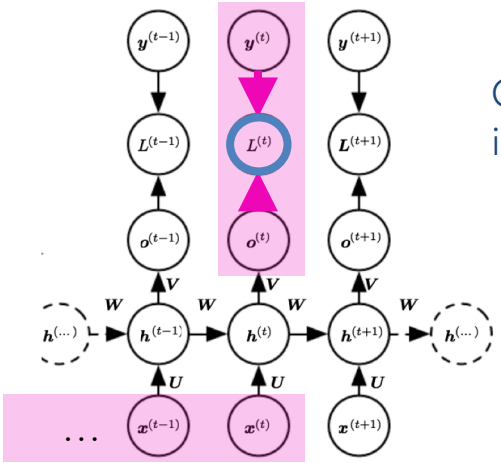
$$\nabla_V L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}}\right) \nabla_V o_i^{(t)} = \sum_t \left(\nabla_{o^{(t)}} L\right) h^{(t)T}$$

$$\nabla_W L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{W^{(i)}} h_i^{(t)} = \sum_t \text{diag}\left(1 - (h^{(t)})^2\right) \left(\nabla_{h^{(t)}} L\right) h^{(t-1)T}$$

$$\nabla_U L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{U^{(i)}} h_i^{(t)} = \sum_t \text{diag}\left(1 - (h^{(t)})^2\right) \left(\nabla_{h^{(t)}} L\right) x^{(t)T}$$

Back-propagation through time (BPTT)

An example RNN where input and out sequences have the same length



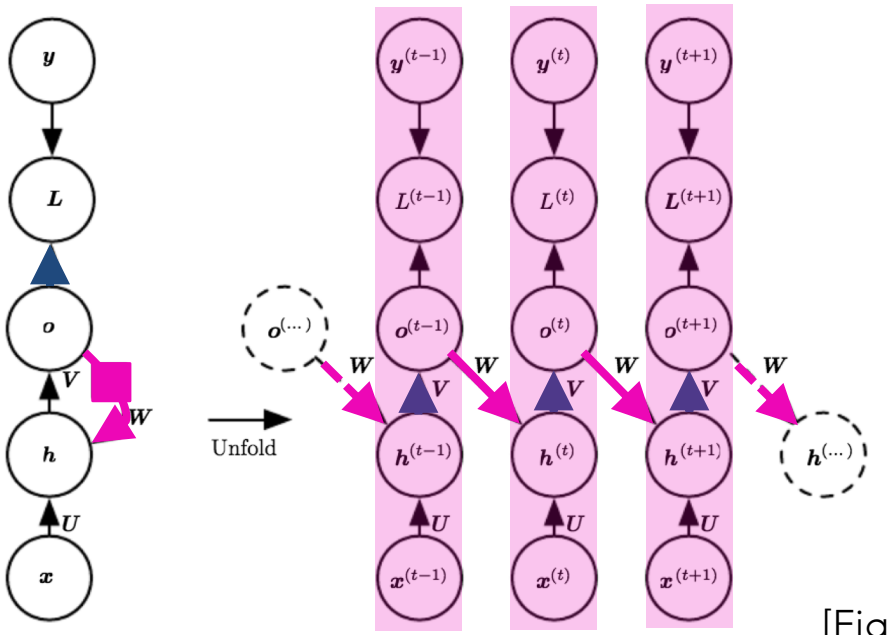
Computing this loss w.r.t. to model parameters is expensive; runtime O(τ), memory O(τ).

Can we parallelize this?

Recall: RNN Pattern II

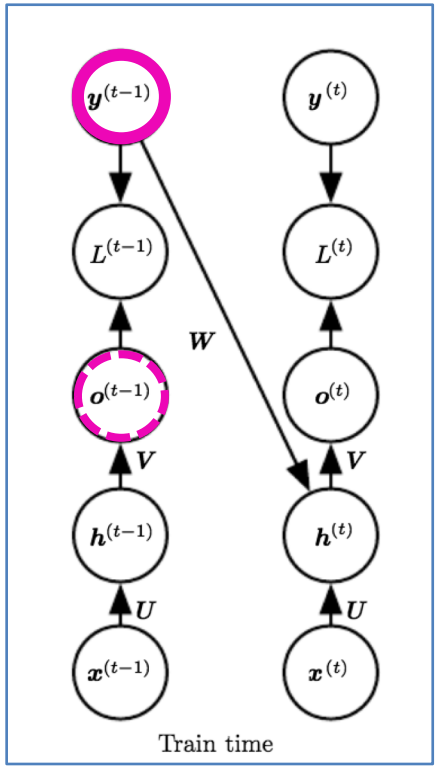
Con: Less powerful

Pro: Each time step can be trained independently in parallel



[Fig. 10.4]

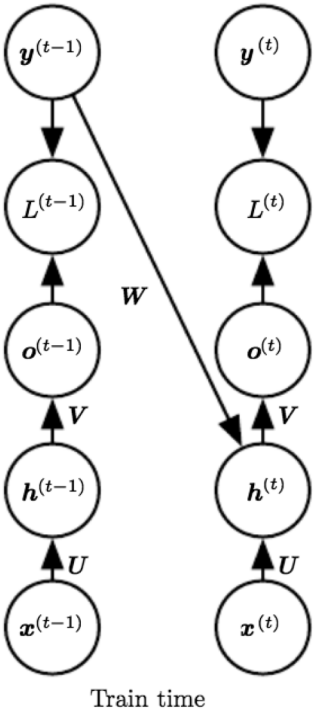
Teacher forcing



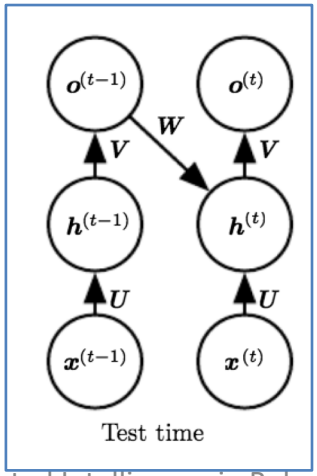
During training, use the ground truth from previous time step as input to compute the hidden unit of the current time step

[Fig. 10.6]

Teacher forcing



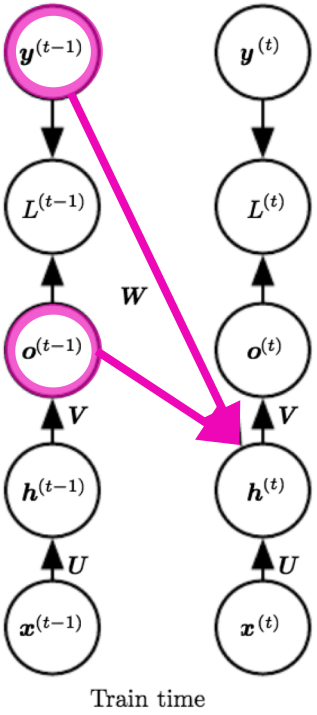
During training, use the ground truth from previous time step as input to compute the hidden unit of the current time step



During test time, use the predicted output from the previous time step to compute the current hidden unit

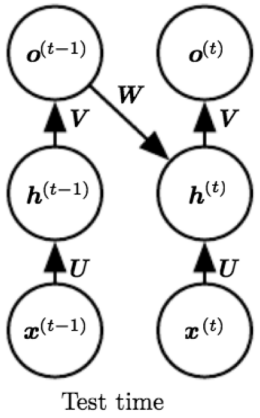
[Fig. 10.6]

Teacher forcing



During training, use the ground truth from previous time step as input to compute the hidden unit of the current time step

To become robust to open-loop mode, train with both teacher's inputs and self-generated inputs



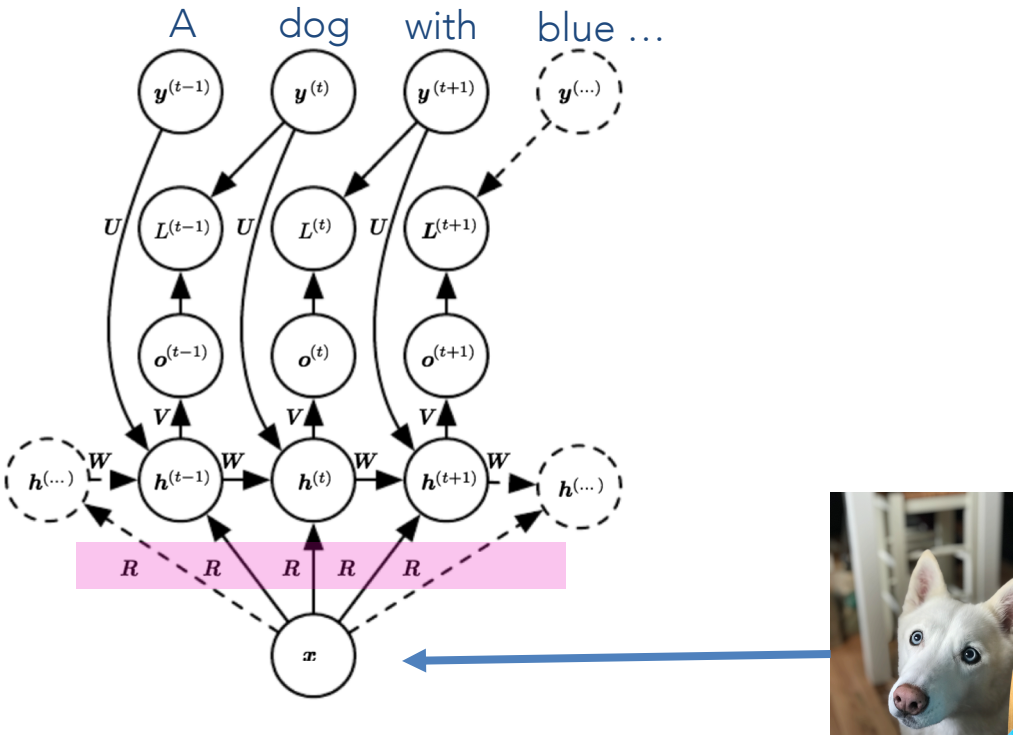
During test time, use the predicted output from the previous time step to compute the current hidden unit

[Fig. 10.6]

Determining when to end

- Special end of sequence symbol [Schmidhuber 2012]
- Bernoulli output at each time step to determine whether to continue or stop
- Add extra input to predict the length of output [Goodfellow et al., 2014]

Single vector input



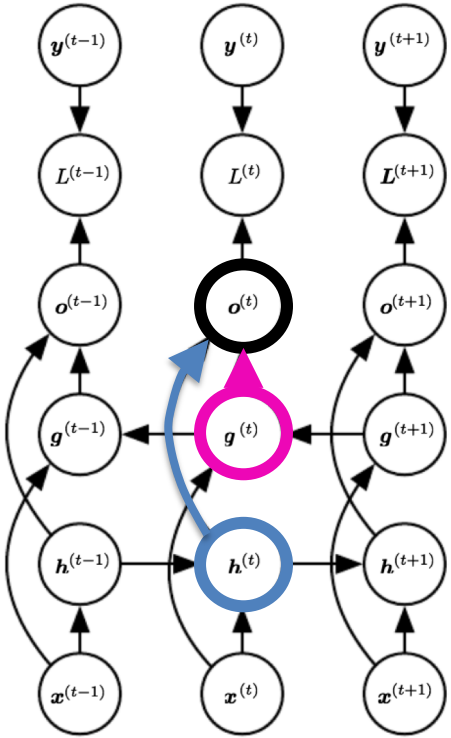
Bidirectional RNNs

(Schuster & Paliwal, 1997)

$x^{(1)}, \ldots, x^{(t-1)}, x^{(t)}, x^{(t+1)}, \ldots, x^{(\tau)}$

- Prediction may depend on the entire input
- Very successful in many applications:
  - Handwriting recognition (Graves et al., 2008, 2009)
  - Speech recognition (Graves & Schmidhuber 2005, 2013)
  - Bioinformatics (Baldi et al., 1999)

Bidirectional RNNs



Backward in time

Forward in time

[Fig. 10.11]

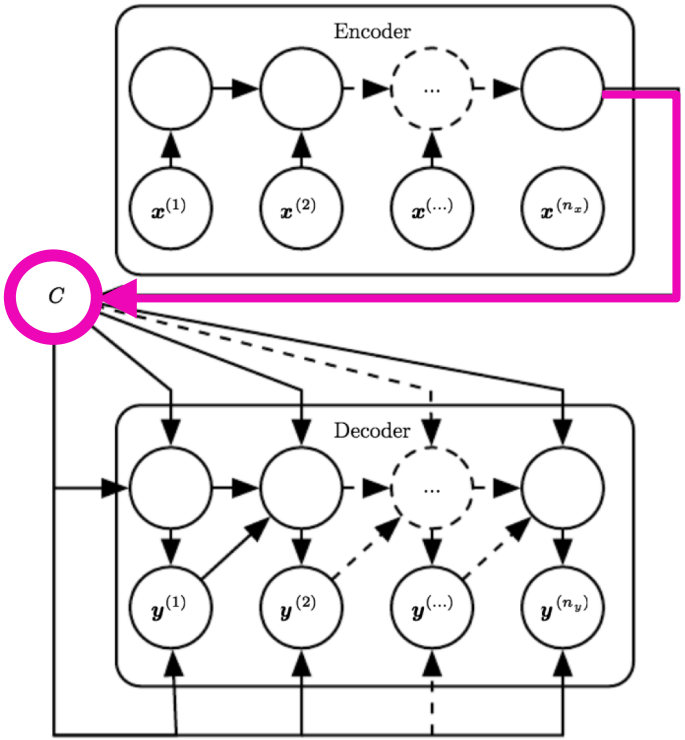
Encoder-Decoder

Sequence-to-Sequence Architecture

- Map variable-length input sequence to variable-length output sequence
- Machine translation [Cho et al., 2014][Sutskever et al., 2014]

Encoder-Decoder

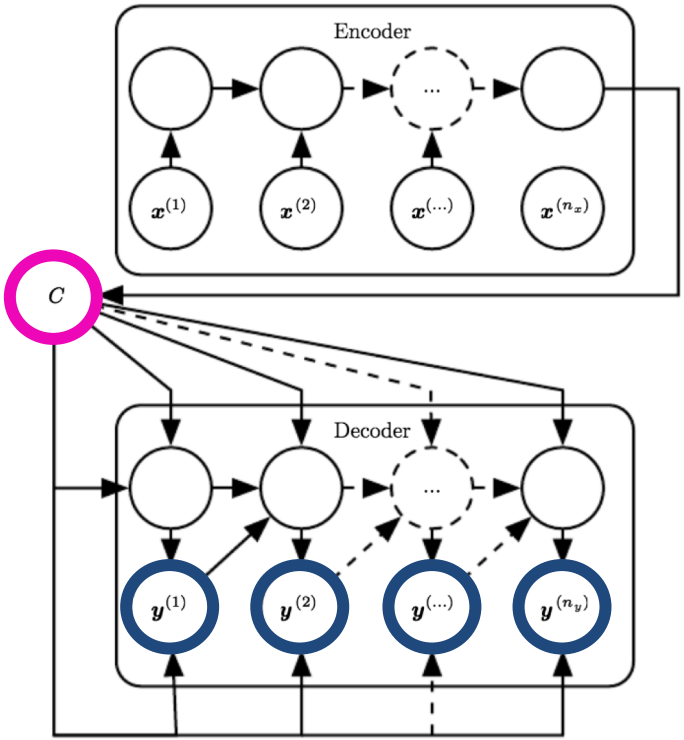
Sequence-to-Sequence Architecture



Encoder (reader or input) RNN processes input sequence $x=(x^{(1)}, \ldots, x^{(nx)})$and emits context C

Encoder-Decoder

Sequence-to-Sequence Architecture



Encoder (reader or input) RNN processes input sequence $x=(x^{(1)}, \ldots, x^{(nx)})$ and emits context C

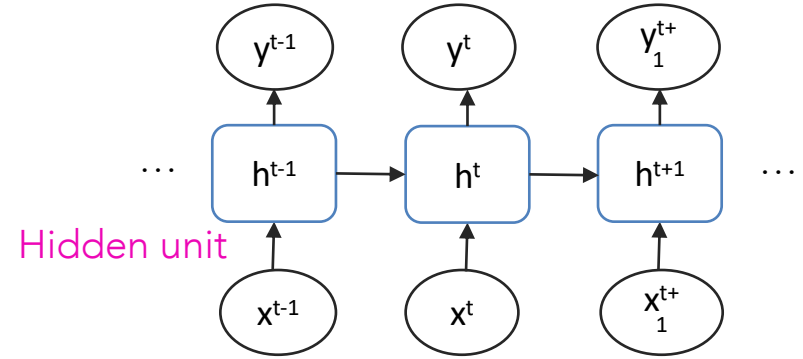
Decoder (writer or output) RNN is conditioned on the context C to generate output sequence $y=(y^{(1)}, \ldots, y^{(ny)})$

Long Short-Term Memory (LSTM)

(Hochreiter & Schmidhuber 1997)

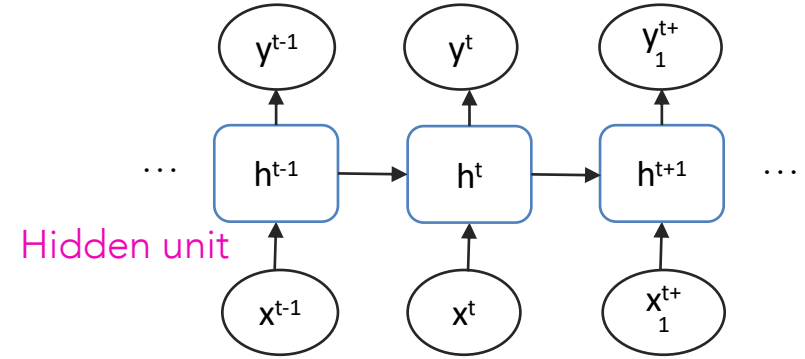
- To handle long-term dependence better
- Gated RNNs
- Self-loop gates controlled by another hidden units
- Extremely successful
  - Speech recognition
  - Handwriting
  - Machine translation
  - Social navigation
  - Image captioning
  - Parsing

Standard RNN

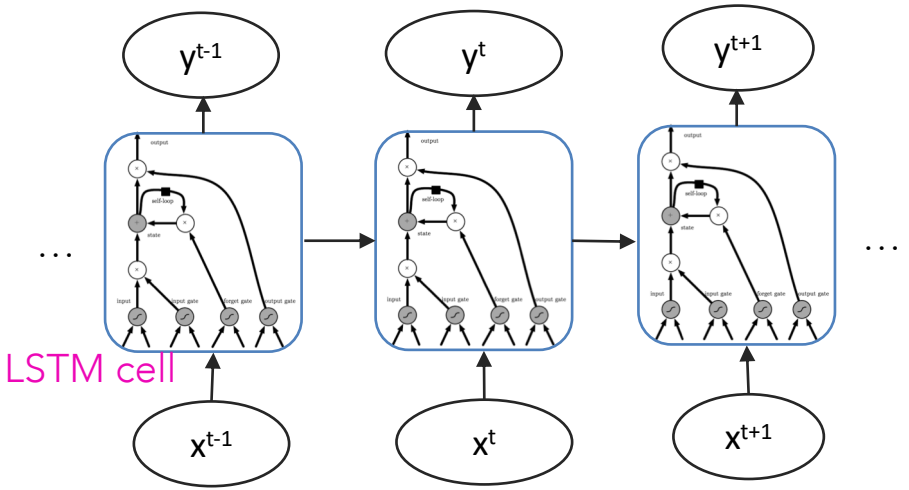


Hidden unit

Standard RNN

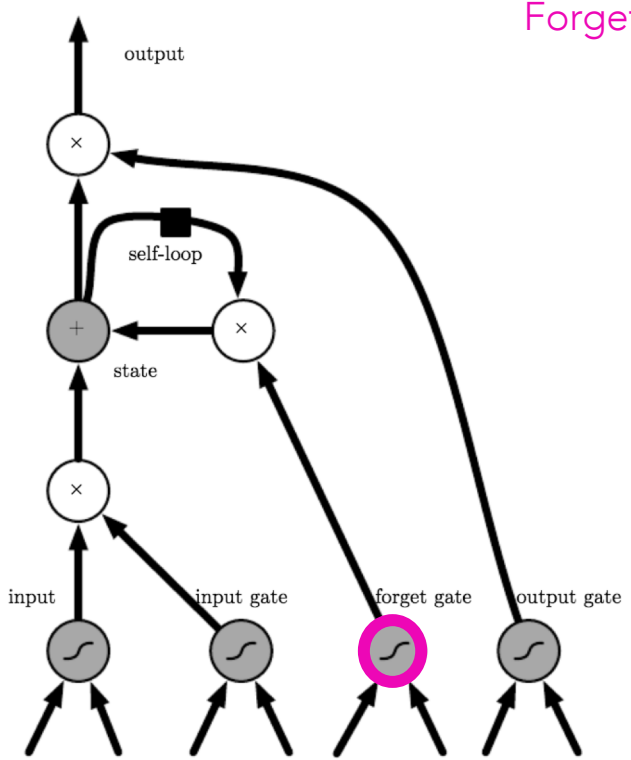


LSTM

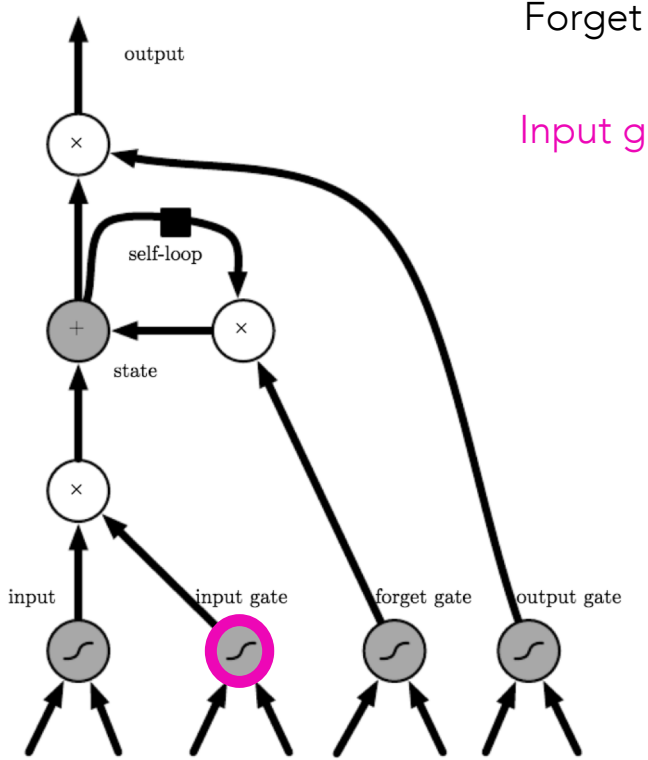


LSTM cell

Forget gate $f_t = \sigma\left(W_f h_{t-1} + U_f x_t + b_f\right)$



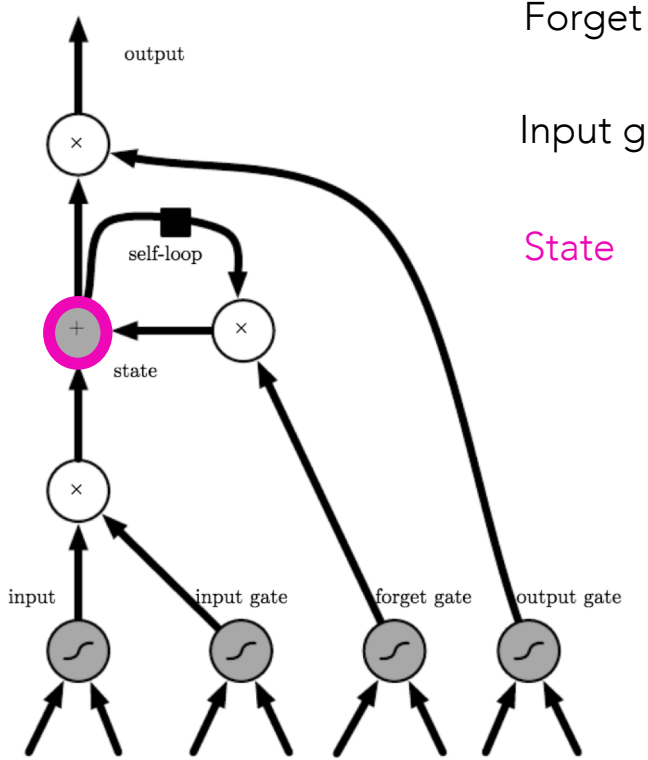
LSTM cell



Forget gate $f_t = \sigma\left(W_f h_{t-1} + U_f x_t + b_f\right)$

Input gate $i_t = \sigma\left(W_i h_{t-1} + U_i x_t + b_i\right)$

LSTM cell



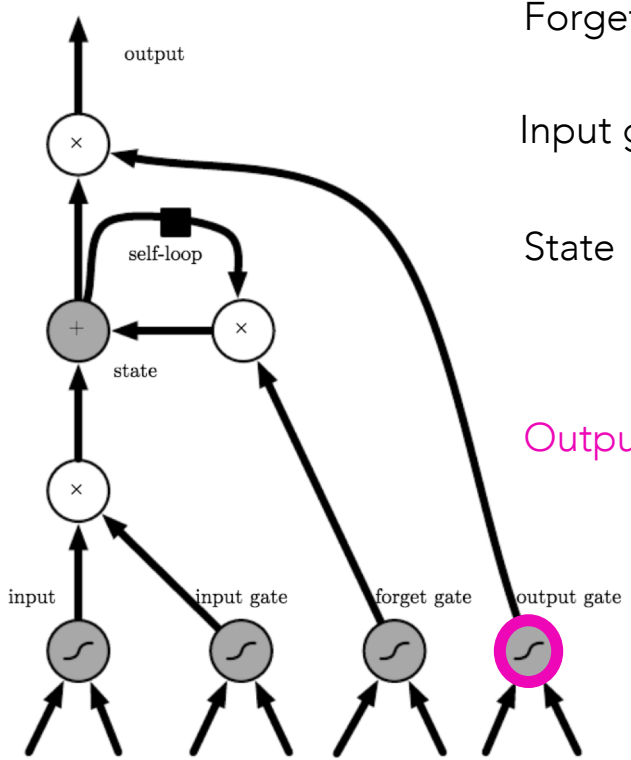
Forget gate $f_t = \sigma\left(W_f h_{t-1} + U_f x_t + b_f\right)$

Input gate $i_t = \sigma\left(W_i h_{t-1} + U_i x_t + b_i\right)$

State $\tilde{C}_t = \tanh\left(W_C h_{t-1} + U_C x_t + b_C\right)$

$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$

LSTM cell



Forget gate $f_t = \sigma\left(W_f h_{t-1} + U_f x_t + b_f\right)$

Input gate $i_t = \sigma\left(W_i h_{t-1} + U_i x_t + b_i\right)$

State $\tilde{C}_t = \tanh\left(W_C h_{t-1} + U_C x_t + b_C\right)$

$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$

Output gate $o_t = \sigma\left(W_o h_{t-1} + U_o x_t + b_o\right)$

$h_t = o_t * \tanh(C_t)$

Summary

- Feedforward neural networks
- Backpropagation
- Convolutional neural networks
- Recurrent neural networks