

DM873

Deep Learning

Spring 2019

Lecture 1 - Introduction

Course Organization

■ Lectures

- We will have roughly 15 lectures
- We will cover a broad spectrum of deep learning

■ Exercises

- Consist of small projects to get you going
- End with a larger project
- The larger project is mandatory for the exam
- Teamwork will be allowed

■ Exam

- **The large assignment will be mandatory**
- **Oral Exam, external marking**
- **Details will follow**



Theory vs. Practice

- It is fundamental to understand the theory
 - When should you apply deep learning?
 - What are the benefits what are the downsides?
 - Why does the model not learn/what is going wrong?
- You can only answer these questions when you understand the theory!
- We try to achieve a healthy mixture

Content of the Course

1. Mathematical Foundations

- Linear Algebra
- Some Statistics

2. Linear Regression

- They form the basic operations for Neural Networks

3. Feed Forward Networks

- The simplest Form of DNN

4. Convolutional Neural Networks

5. Recurrent Neural Networks

6. Autoencoders

7. Generative Models

Introduction to the Course

- Course web-page (there is a link on blackboard):
http://imada.sdu.dk/~roettger/teaching/2019_spring_dm873.php
- Slides will be available
- All relevant course material will be available on the website
- TA: Mathias
- Contact: just email us
 - boegebjerg@imada.sdu.dk

Book and Slides

- **Deep Learning**

Ian Goodfellow and Yoshua Bengio and Aaron Courville
MIT Press, 2016

<http://www.deeplearningbook.org/>

- Most slides are based on:

- <https://www.cs.cmu.edu/~rsalakhu/10707/lectures.html>
- https://www.cc.gatech.edu/classes/AY2018/cs7643_fall/

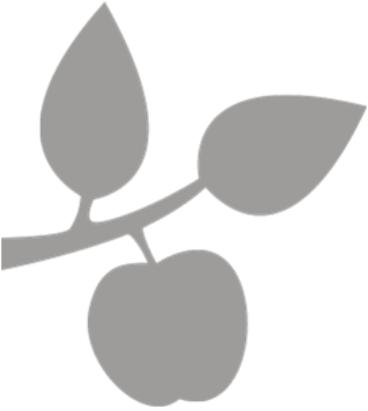
- Slides will be available on the website after the lecture

- **User: deep**
- **Pass: sodeep**

Announcement

No Exercise next Monday

18.02.2019



Introduction

What is it all about?

**Some of the most exciting developments
in Machine Learning, Vision, NLP,
Speech, Robotics & AI in general in the
last 5 years!**

Money Printing Machine

Google snaps up object recognition startup DNNr

Google has acq
Toronto, who

by [Josh Lowensohn](#) !

2 / 0

Google has acqui
research compan
image recognitior

DNNresearch. wh



Yan
Decem

Big news to

Facebook ha
long-term go
Intelligence

(C) DRIVV Data

« [Search needs a shake-up](#)

[Songbirds use grammar rules](#) »



Machine Learning Startup Acquired by ai-one

Press Release

For Immediate Release: August 4, 2011

[San Diego artificial intelligence startup acquired by leading
pro](#)

IBM acquires deep learning startup AlchemyAPI

by [Derrick Harris](#) Mar. 4, 2015 - 8:15 AM PDT

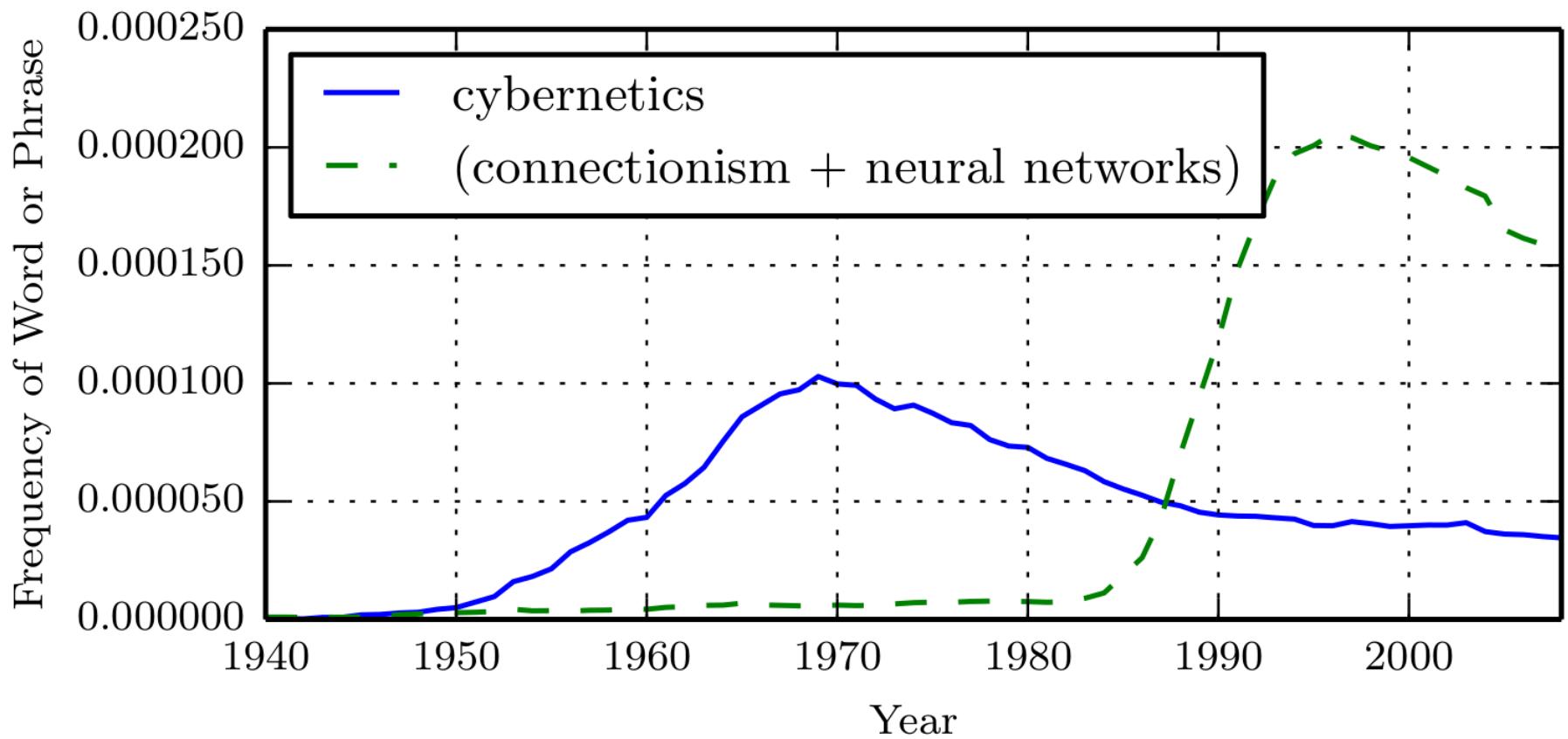
1 Comment

IBM Watson. Photo by Clockready/Wikimedia Commons



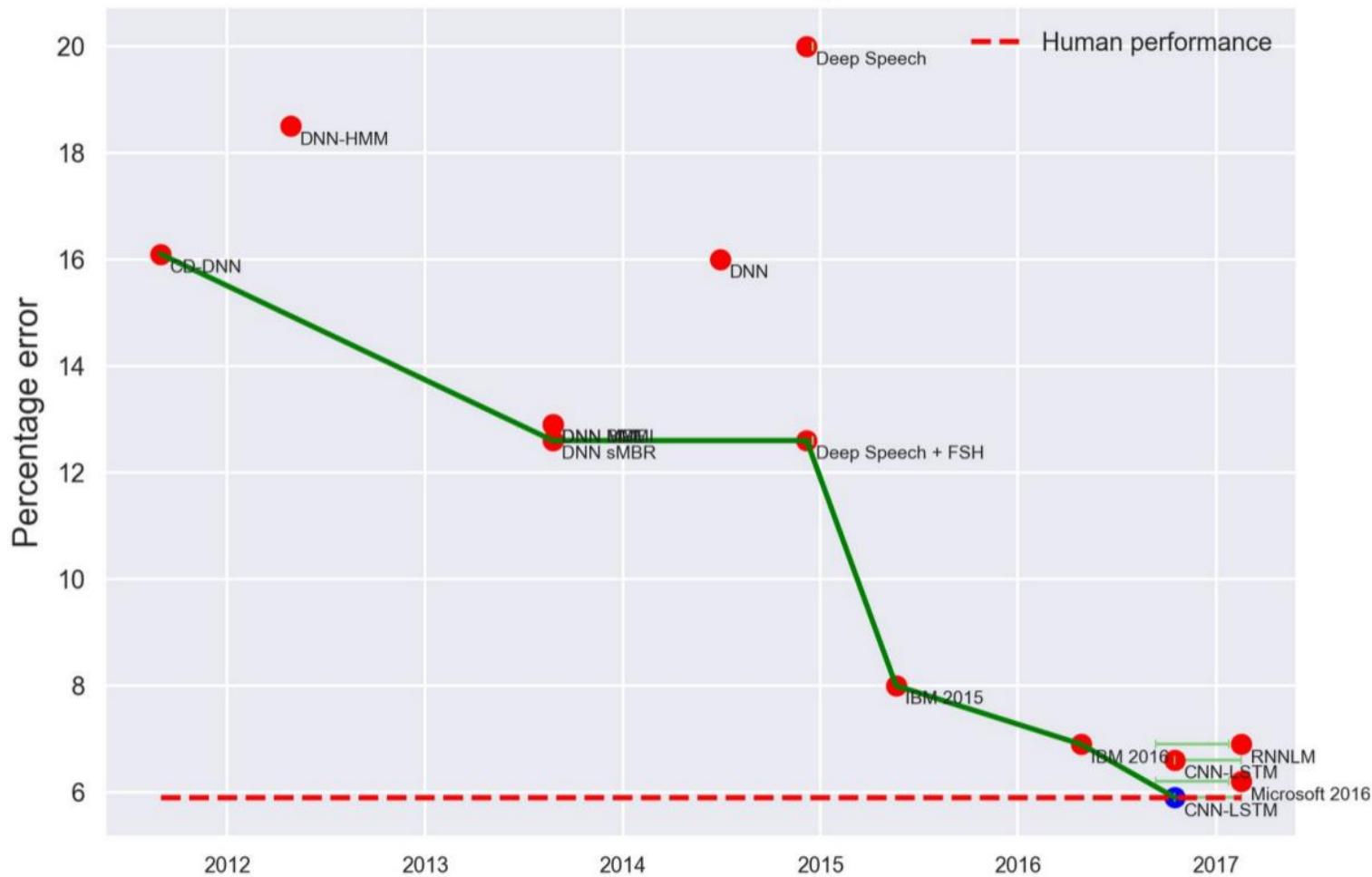
UNIVERSITY OF SOUTHERN DENMARK.DK

Historical Waves



Transcribing Phone Calls

Word error rate on Switchboard trained against the Hub5'00 dataset



Classification: Labradoodle or fried chicken?



➤ <https://www.cs.tau.ac.il/~dcor/Graphics/pdf.slides/YY-Deep%20Learning.pdf>

Classification: Puppy or bagel?



➤ <https://www.cs.tau.ac.il/~dcor/Graphics/pdf.slides/YY-Deep%20Learning.pdf>

Classification: Sheepdog or mop?



➤ <https://www.cs.tau.ac.il/~dcor/Graphics/pdf.slides/YY-Deep%20Learning.pdf>

Classification: Barn owl or apple?



@teenybiscuit

➤ <https://www.cs.tau.ac.il/~dcor/Graphics/pdf.slides/YY-Deep%20Learning.pdf>

Classification: Raw chicken or Donald Trump?



➤ <https://www.cs.tau.ac.il/~dcor/Graphics/pdf.slides/YY-Deep%20Learning.pdf>

Surprising Results

- This is a tremendously difficult task for a computer
 - A picture represents a huge feature space
 - The classification must be robust with respect to rotation, light conditions etc.

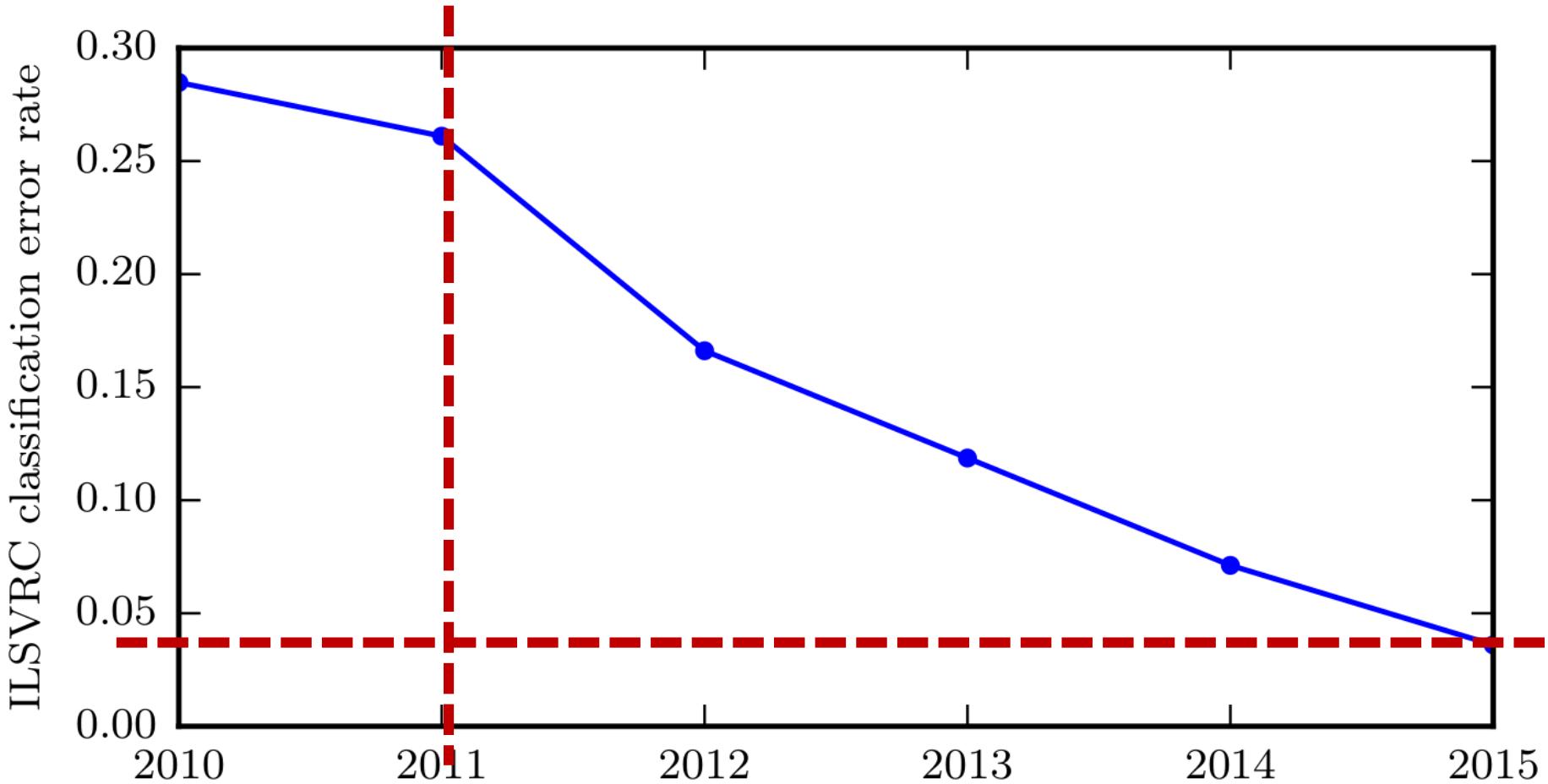
Surprising Results

- This is a tremendously difficult task for a computer
 - A picture represents a huge feature space
 - The classification must be robust with respect to rotation, light conditions etc.

Deep Neural Networks have
surpassed human
performance

Solving Object Recognition

IMAGENET Large Scale Visual Recognition Challenge (ILSVRC)



AlphaGo



AlphaGo

FINAL DEFEAT

The awful frustration of a teenage Go champion playing Google's AlphaGo

OUR PICKS LATEST POPULAR QUARTZ OBSSESSIONS ...

A photograph of a young man with dark hair, wearing a black shirt, sitting at a Go board. He is leaning forward with his head in his hands, appearing frustrated or exhausted. The Go board is in front of him, with stones on it. To the right, a small screen displays the name 'KE JIE' and the time '00:46:57'. The background is a blue wall with circular light fixtures.

Increasingly more difficult tasks

Visual Dialog



A cat drinking water out of a coffee mug.

What color is the mug?

White and red

Are there any pictures on it?

No, something is there can't tell what it is

Is the mug and cat on a table?

Yes, they are

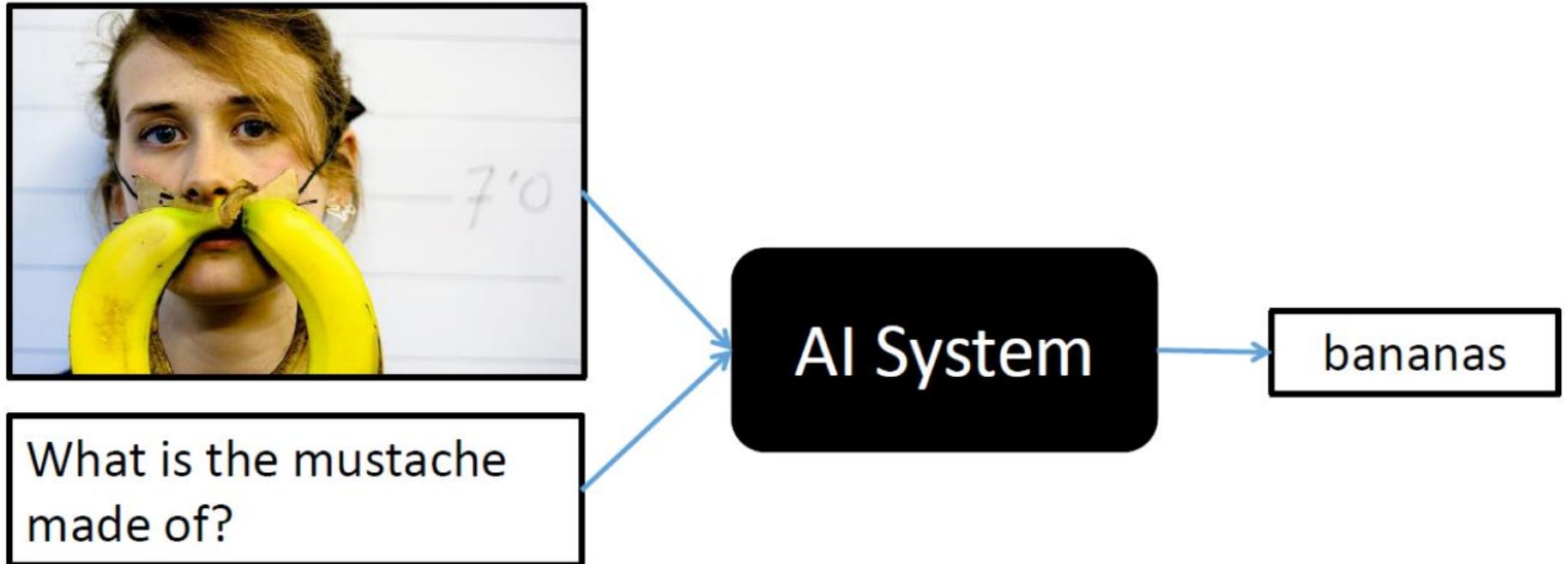
Are there other items on the table?

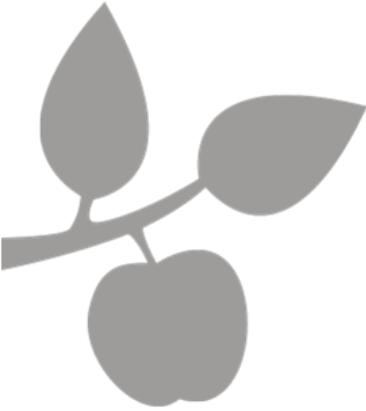
Yes, magazines, books, toaster and basket, and a plate

C Start typing question here ... >

➤ Das et al., 2017

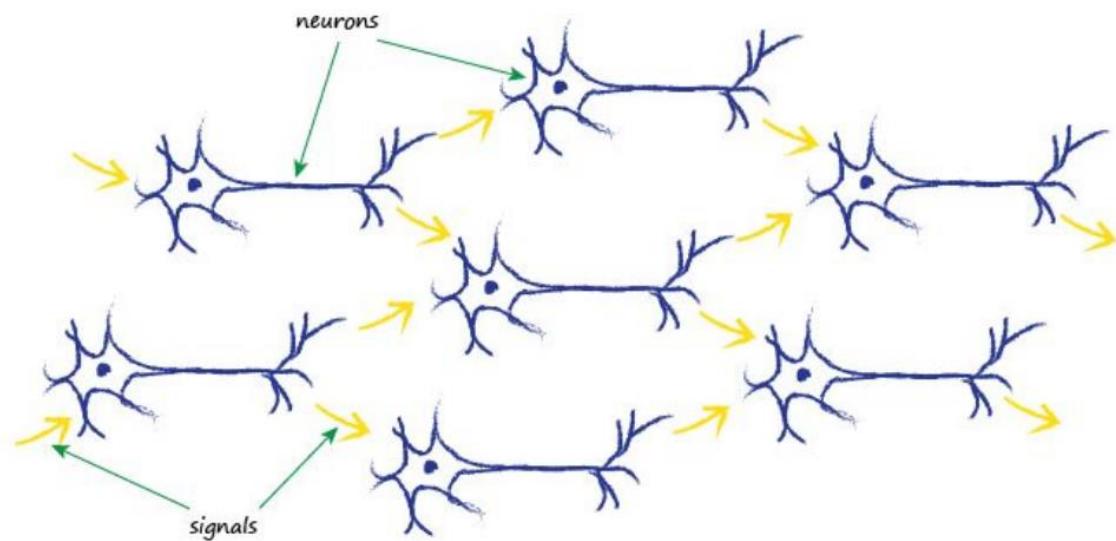
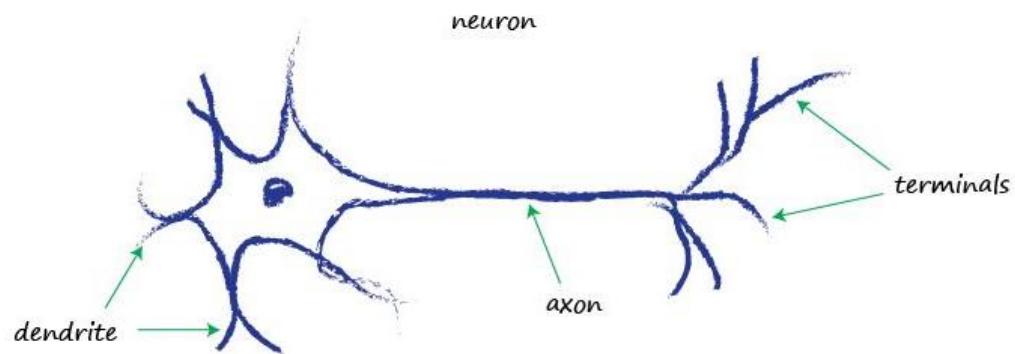
Visual Question Answering (vQA)



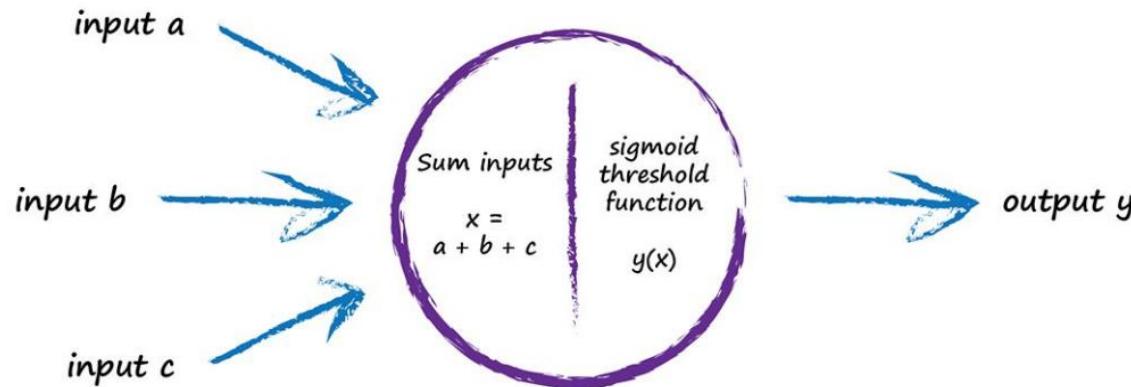


What is Deep Learning?

The Neuron ... in Nature



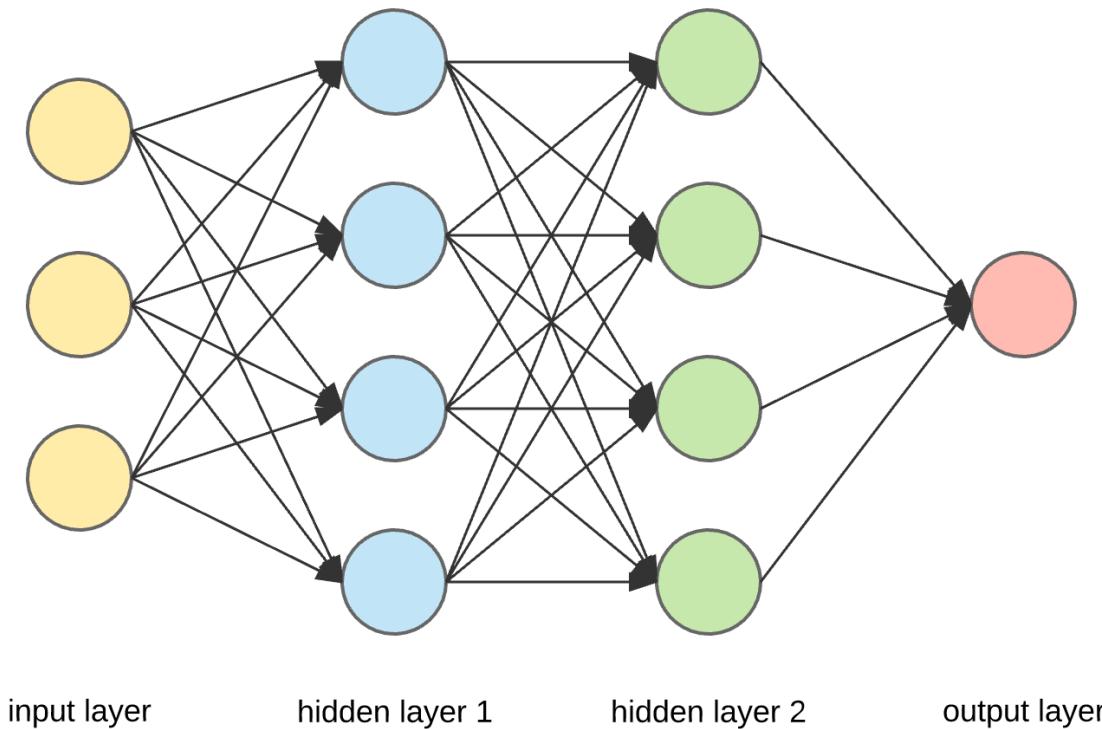
The Artificial Neuron



- An artificial Neuron consists of
 - A number of weighted inputs
 - An activation function
 - The generated output

Connecting to a Network

- We normally have more than one node
- Multiple Nodes are arranged in layers
- Each layer receives the generated output from the previous layer



The Driving Ideas of Deep Learning

- Representation Learning using deep neural networks
- (Hierarchical) Compositionality
 - Cascade of non-linear transformations
 - Multiple layers of representations
- End-to-End Learning
 - Learning (goal-driven) representations
 - Learning to feature extraction
- Distributed Representations
 - No single neuron “encodes” everything
 - Groups of neurons work together

The Driving Ideas of Deep Learning

- Representation Learning using deep neural networks
- **(Hierarchical) Compositionality**
 - Cascade of non-linear transformations
 - Multiple layers of representations
- End-to-End Learning
 - Learning (goal-driven) representations
 - Learning to feature extraction
- Distributed Representations
 - No single neuron “encodes” everything
 - Groups of neurons work together

Hierarchical Compositionality

VISION

pixels → edge → texton → motif → part → object

SPEECH

sample → spectral band → formant → motif → phone → word

NLP

character → word → NP/VP/.. → clause → sentence → story

Building A Complicated Function

Library of
simple functions

$$\begin{array}{ll} x^n & \cos(x) \\ \log(x) & \sin(x) \\ & e^x \end{array}$$

Compose



Building A Complicated Function

Library of simple functions

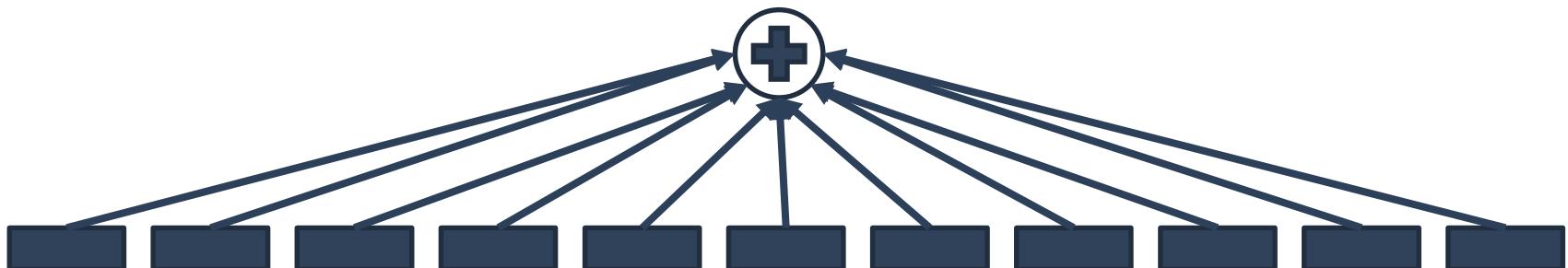
$$\begin{array}{ll} x^n & \cos(x) \\ \log(x) & \sin(x) \\ & e^x \end{array}$$



Linear Combinations

- Boosting
- Kernels
- ...

$$f(x) = \sum_i \alpha_i g_i(x)$$



Building A Complicated Function

Library of simple functions

$$\begin{array}{ll} x^n & \cos(x) \\ \log(x) & \sin(x) \\ & e^x \end{array}$$



Compositions

- Deep Learning
- Grammar models
- ...

$$f(x) = g_1(g_2(\dots(g_n(x))\dots))$$



Building A Complicated Function

Library of simple functions

$$\begin{array}{ll} x^n & \cos(x) \\ \log(x) & \sin(x) \\ e^x & \end{array}$$



Compositions

- Deep Learning
- Grammar models
- ...

$$f(x) = \log \left(\cos \left(\exp \left(\sin^3(x) \right) \right) \right)$$

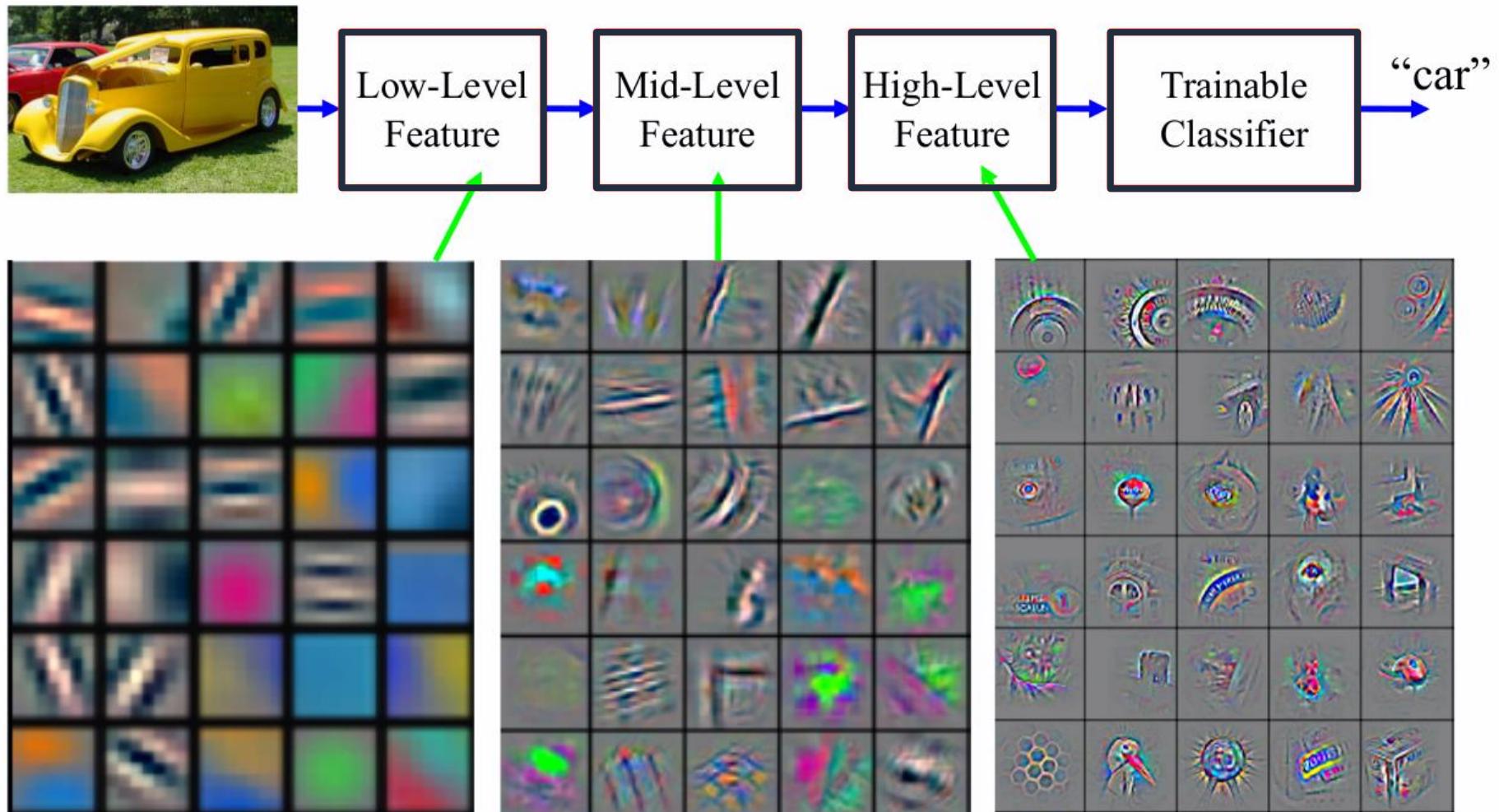


Hierarchical Composition



- Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Hierarchical Composition

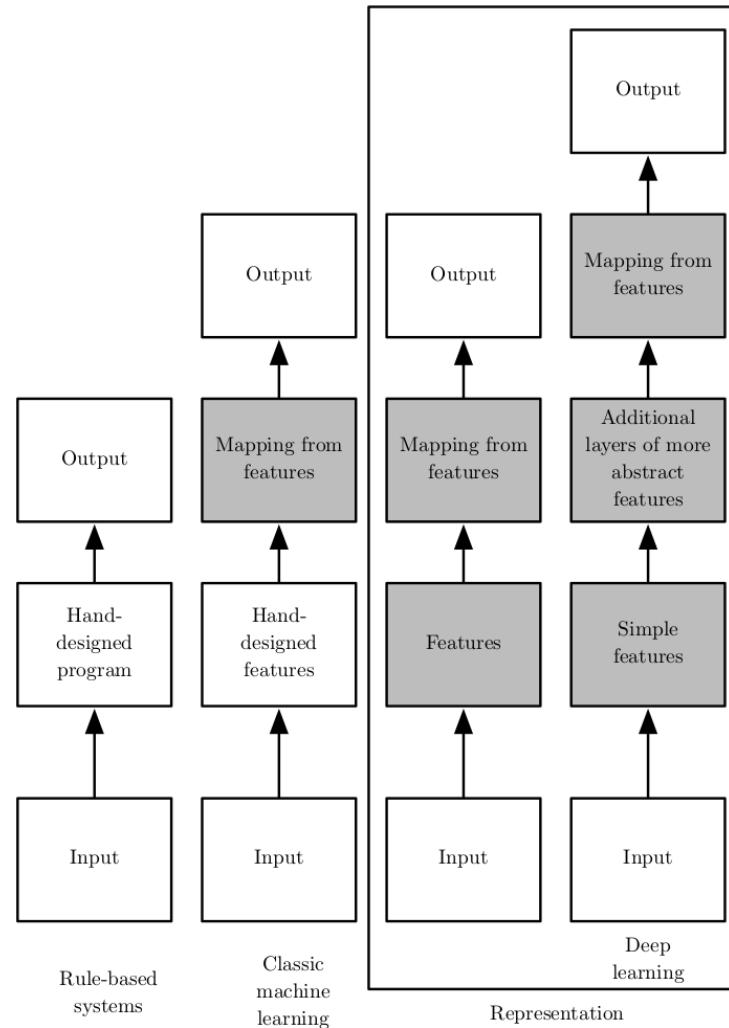


- Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

The Driving Ideas of Deep Learning

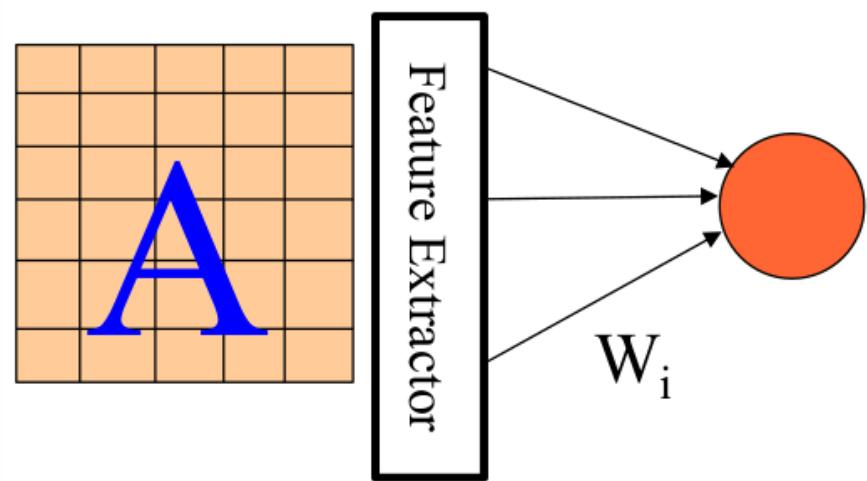
- Representation Learning using deep neural networks
- (Hierarchical) Compositionality
 - Cascade of non-linear transformations
 - Multiple layers of representations
- End-to-End Learning
 - Learning (goal-driven) representations
 - Learning to feature extraction
- Distributed Representations
 - No single neuron “encodes” everything
 - Groups of neurons work together

Learning Multiple Components



Old Paradigm

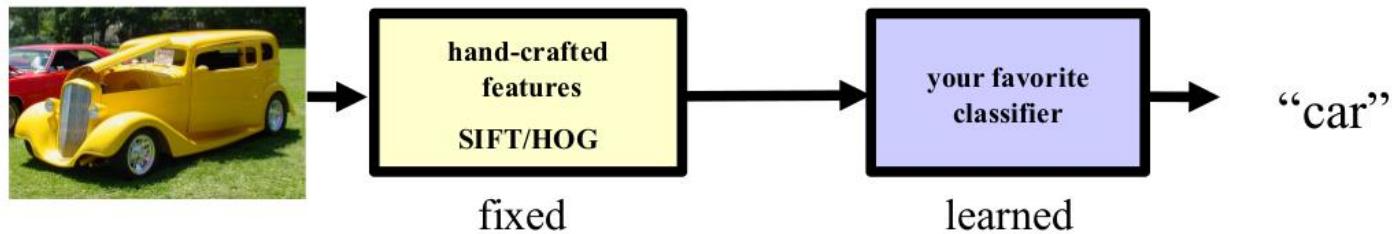
- The first learning machine: the Perceptron (~1960)
- The Perceptron was a linear classifier on top of a simple feature extractor
- The vast majority of practical applications of ML today use glorified linear classifiers or glorified template matching.
- **Designing a feature extractor requires considerable efforts by experts.**



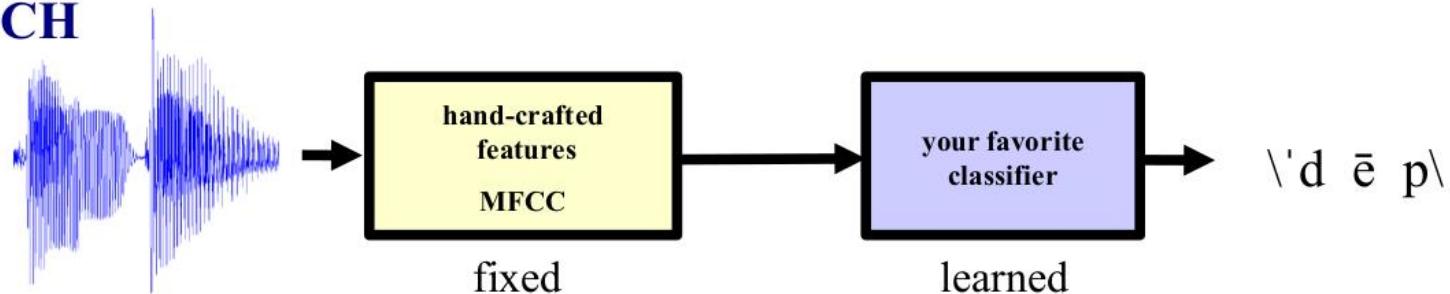
$$y = \text{sign} \left(\sum_{i=1}^N W_i F_i(X) + b \right)$$

Traditional Machine Learning

VISION

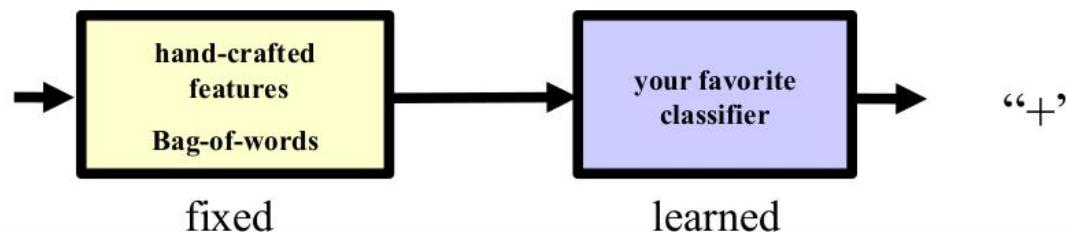


SPEECH



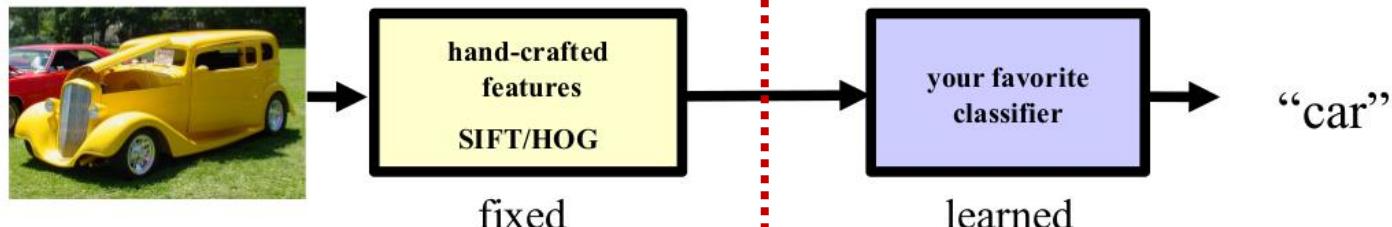
NLP

This burrito place
is yummy and fun!

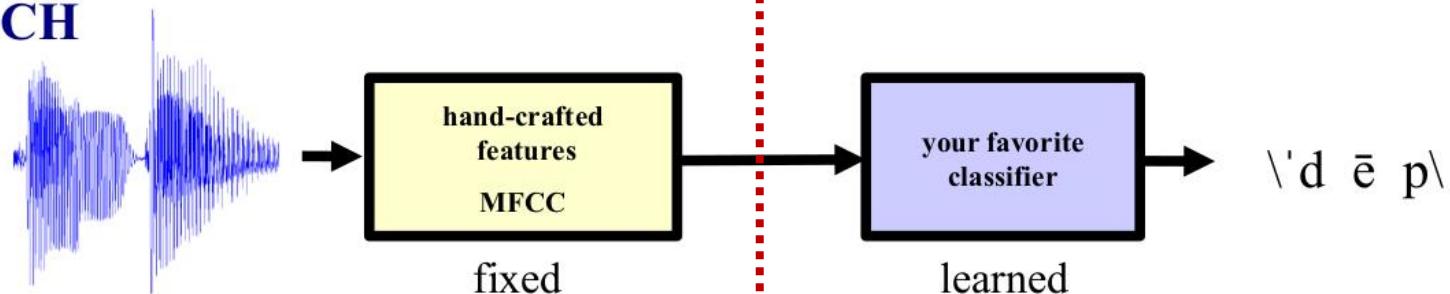


Traditional Machine Learning

VISION

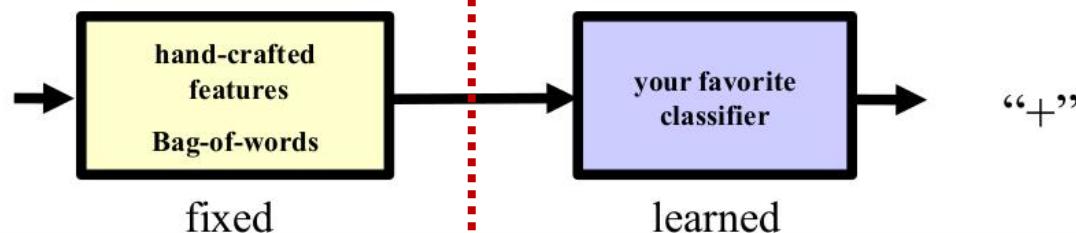


SPEECH



NLP

This burrito place
is yummy and fun!



Learned

your favorite
classifier

learned

"car"

your favorite
classifier

learned

\'d ē p\'

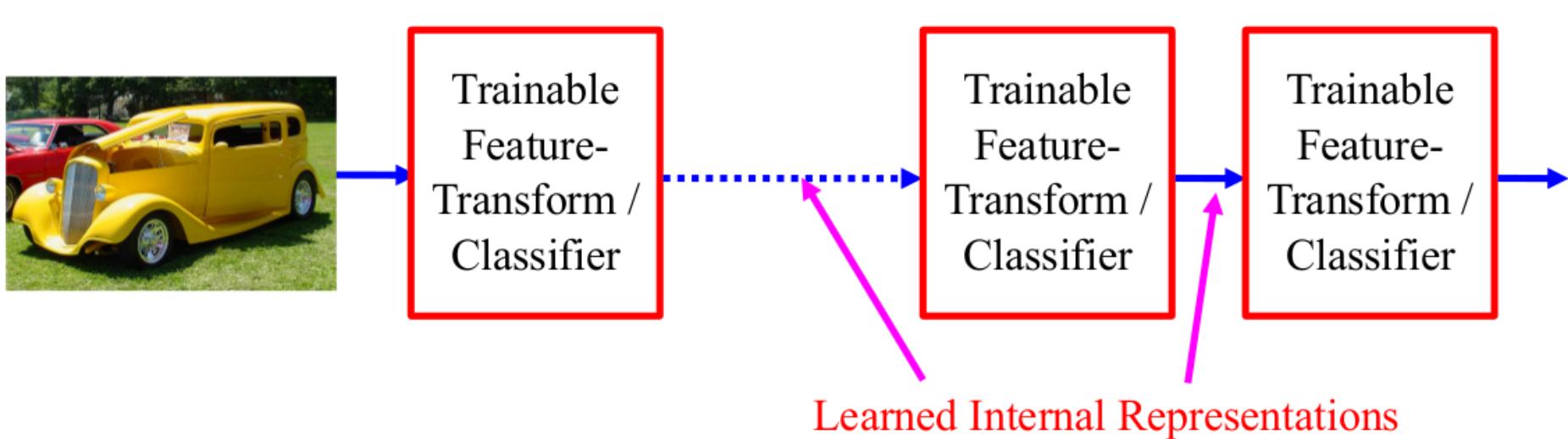
your favorite
classifier

learned

"+"

Deep Learning = End-to-End Learning

- A hierarchy of trainable feature transforms
 - Each module transforms its input representation into a higher-level one.
 - High-level features are more global and more invariant.
 - Low-level features are shared among categories.



The Driving Ideas of Deep Learning

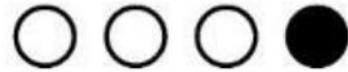
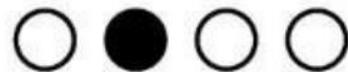
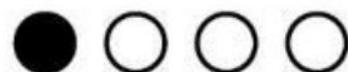
- Representation Learning using deep neural networks
- (Hierarchical) Compositionality
 - Cascade of non-linear transformations
 - Multiple layers of representations
- End-to-End Learning
 - Learning (goal-driven) representations
 - Learning to feature extraction
- **Distributed Representations**
 - **No single neuron “encodes” everything**
 - **Groups of neurons work together**

Distributed Representations

- Local

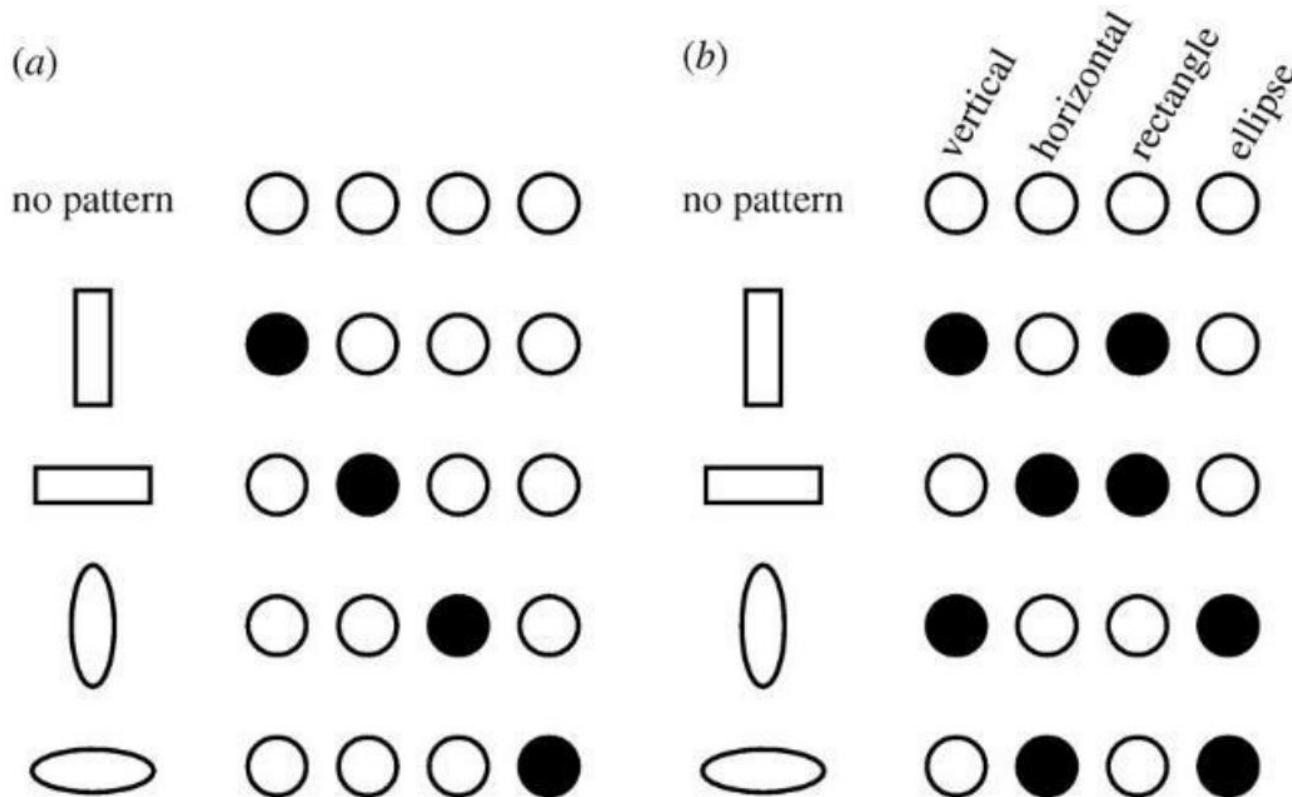
(a)

no pattern



Distributed Representations

- Local vs. Distributed
- Can we interpret each dimension?



Power of Distributed Representations!

Local

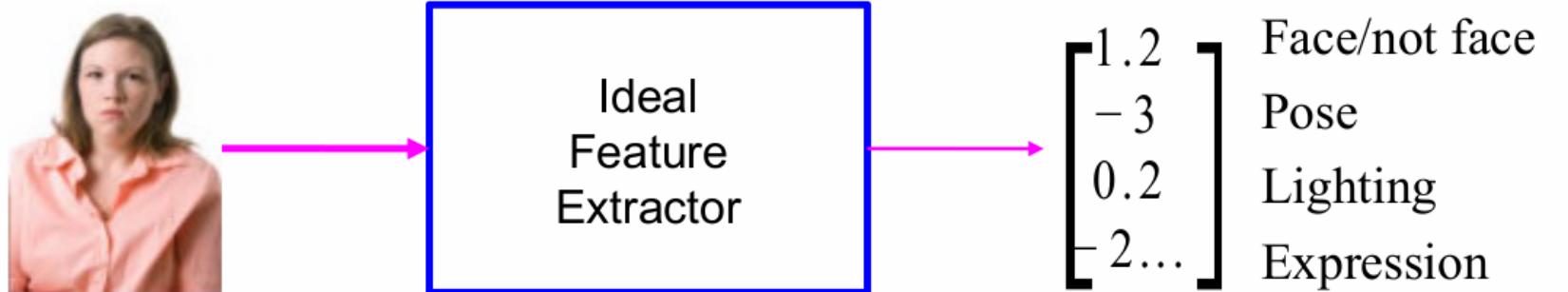
$$\bullet \bullet \circ \bullet = \text{VR} + \text{HR} + \text{HE} = ?$$

Distributed

$$\bullet \bullet \circ \bullet = \text{V} + \text{H} + \text{E} \approx \bigcirc$$

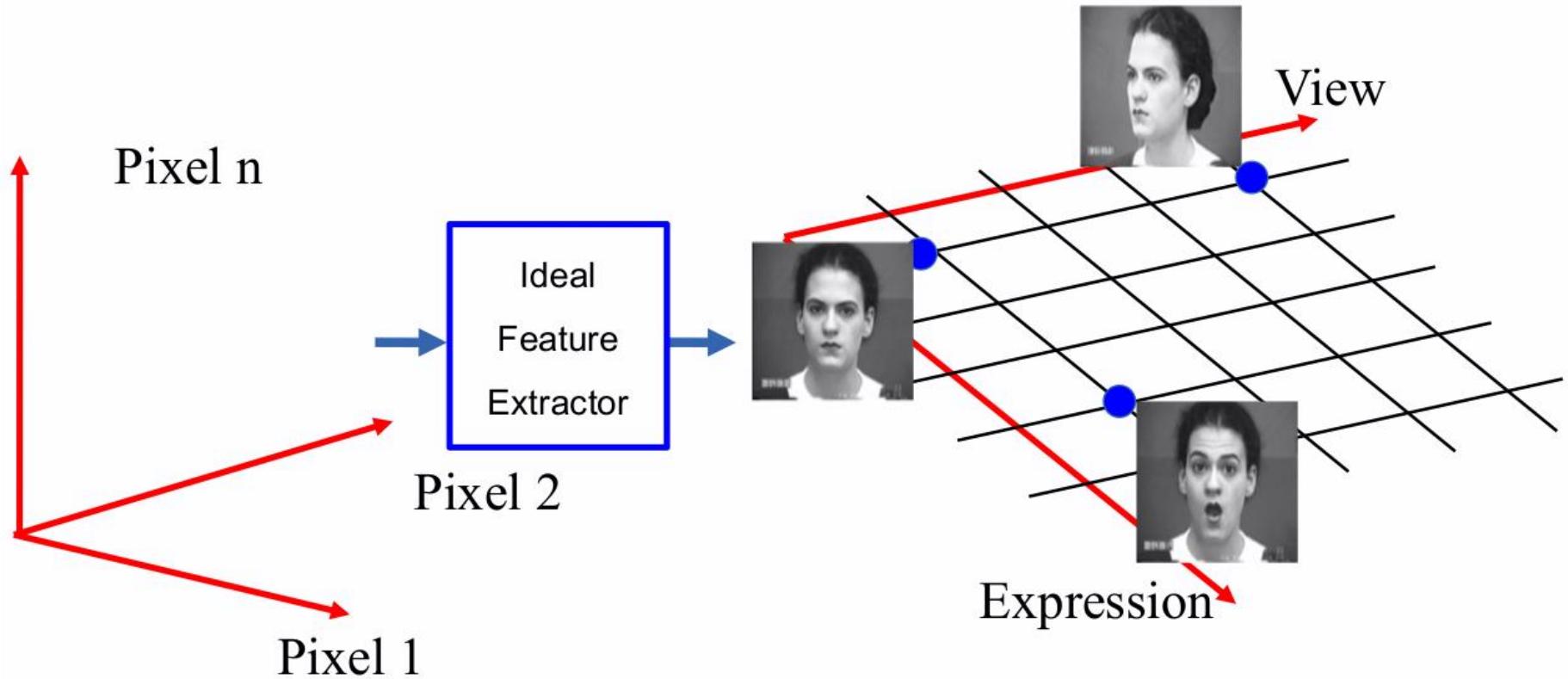
Power of Distributed Representations!

- Example: all face images of a person
 - 1000×1000 pixels = 1,000,000 dimensions
 - But the face has 3 cartesian coordinates and 3 Euler angles
 - And humans have about 50 muscles in the face
 - Hence the manifold of face images for a person has <56 dimensions
- The perfect representations of a face image:
 - Its coordinates on the face manifold
 - Its coordinates away from the manifold



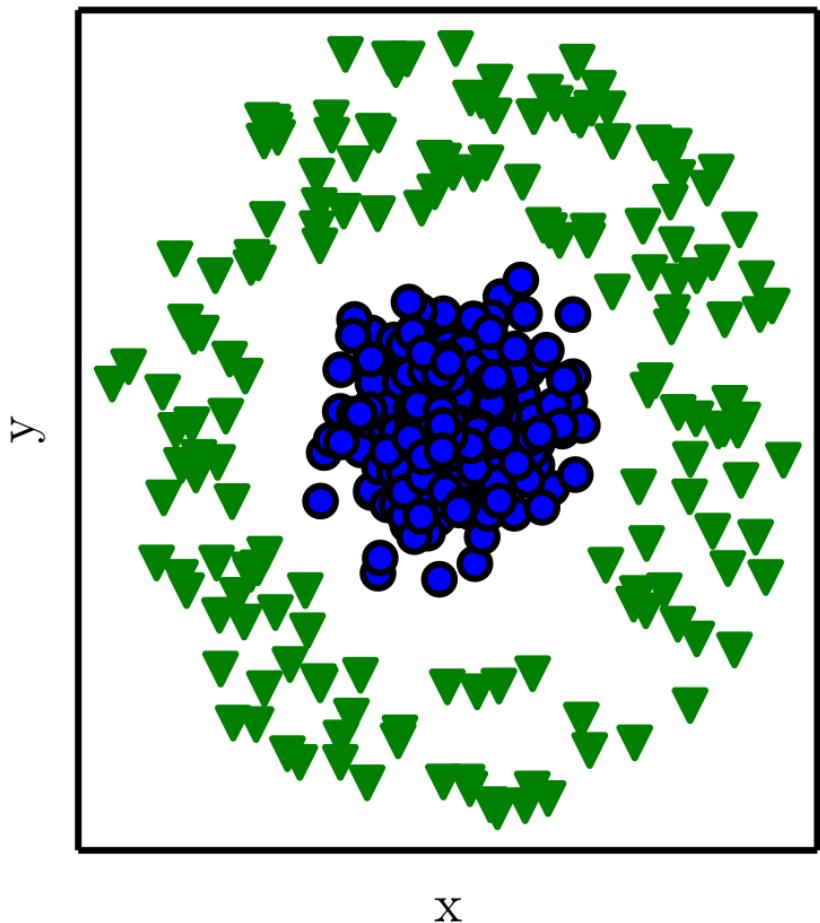
Power of Distributed Representations!

The Ideal Disentangling Feature Extractor

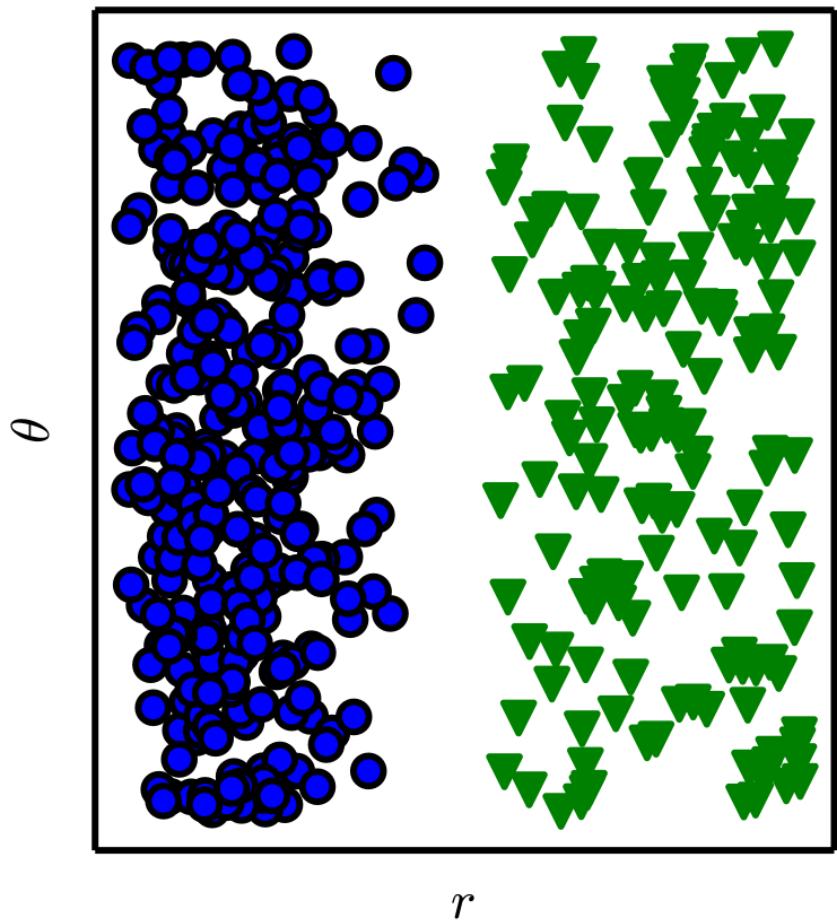


Representation Matters

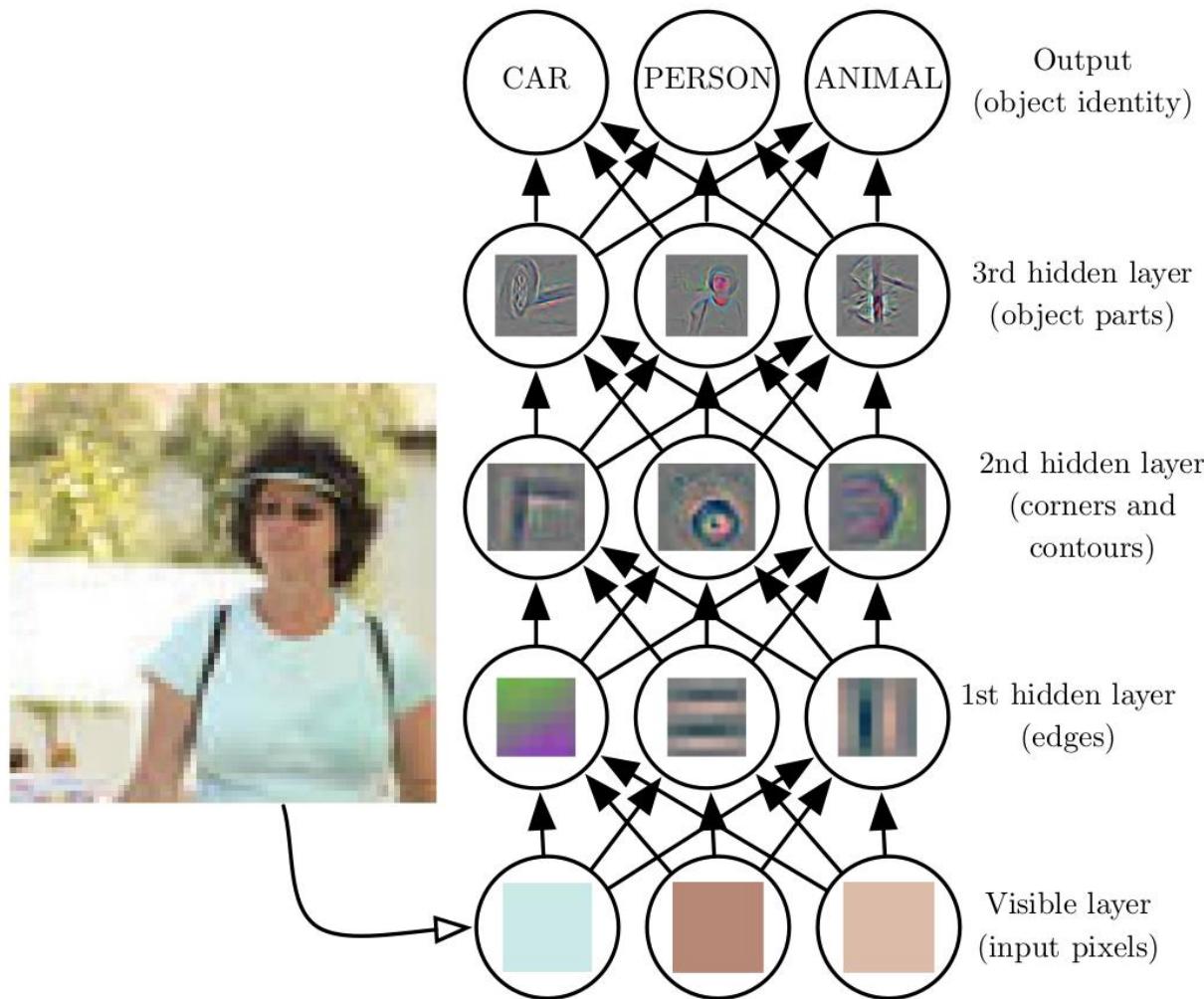
Cartesian coordinates

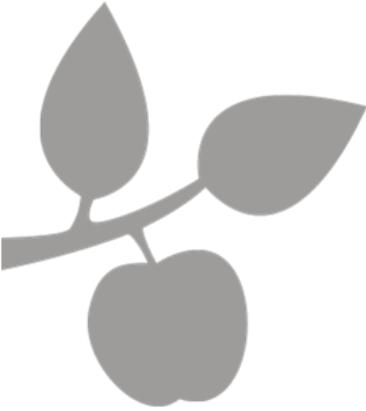


Polar coordinates



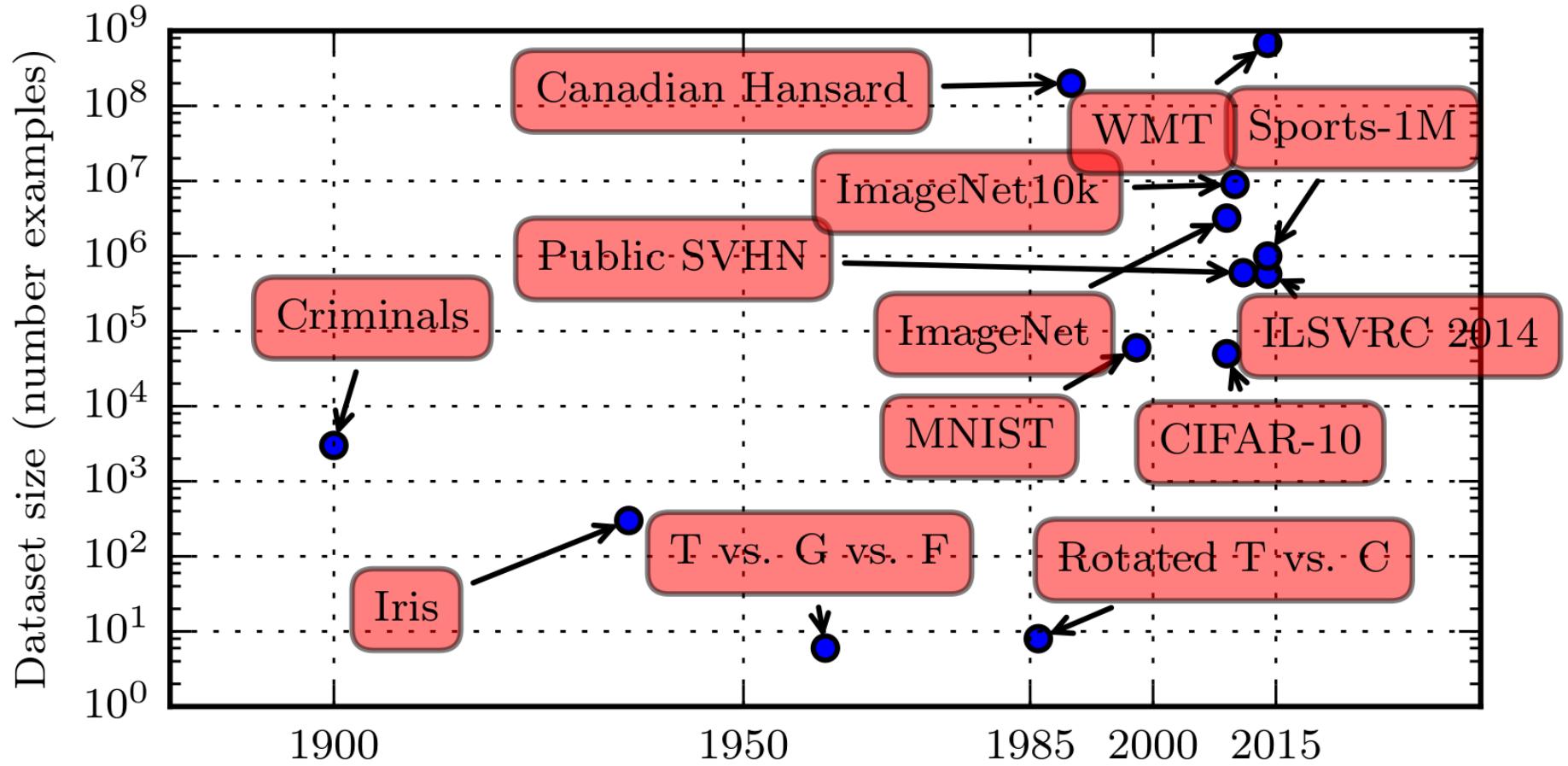
Depth: Repeated Composition





Enablers of Deep Learning

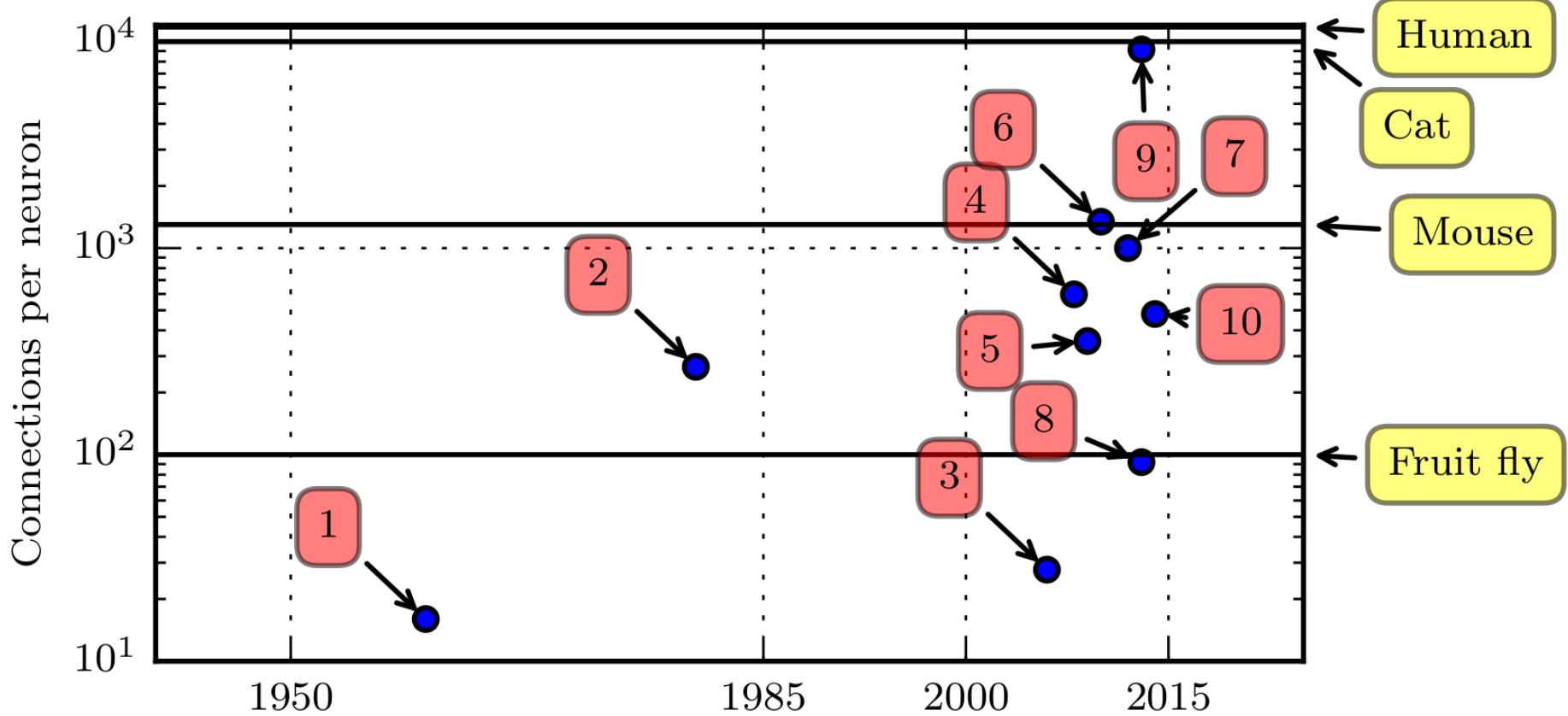
Growing Dataset Sizes



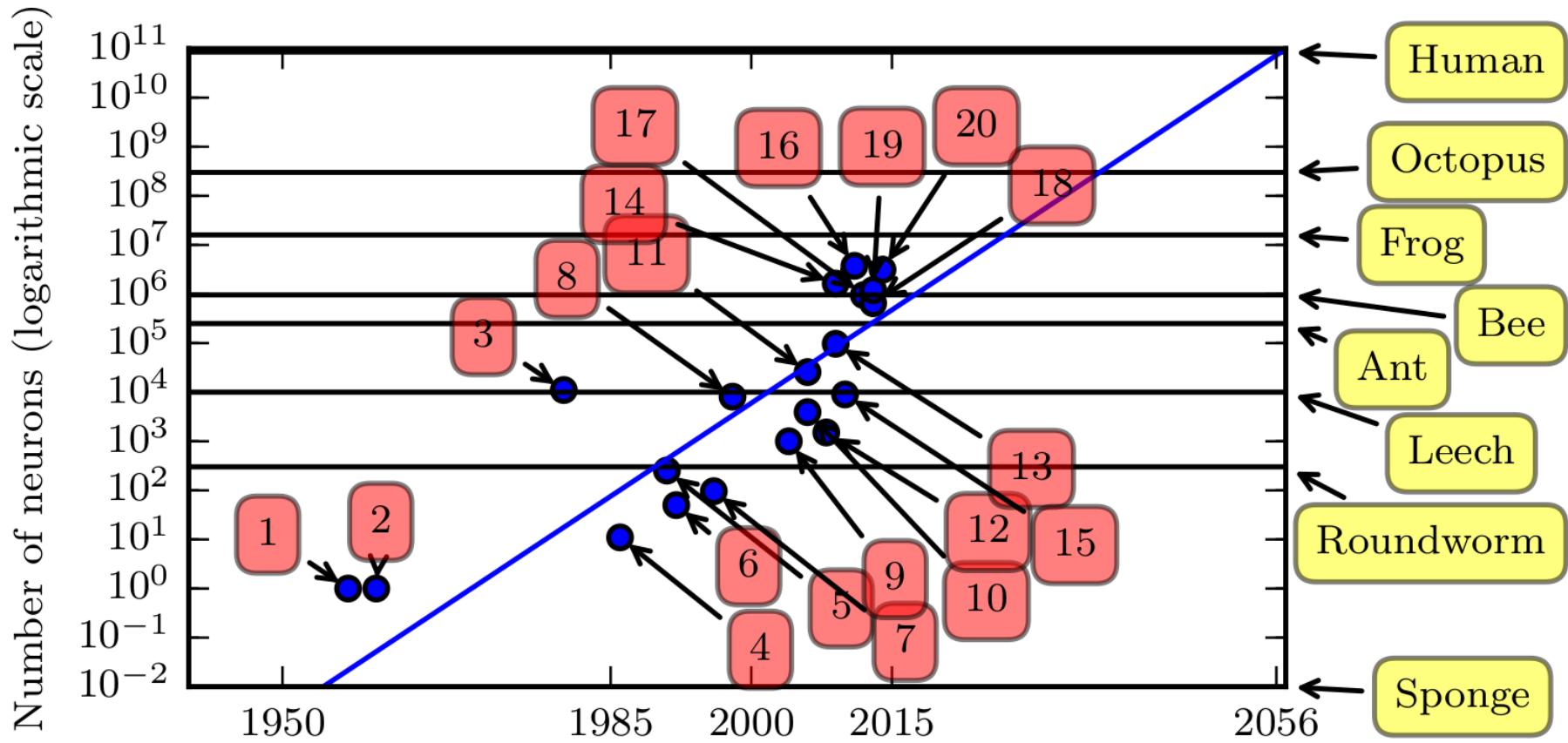
The MNIST Dataset

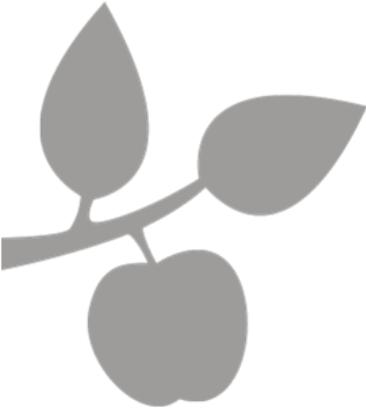
8	9	0	1	2	3	4	7	8	9	0	1	2	3	4	5	6	7	8	6
4	2	6	4	7	5	5	4	7	8	9	2	9	3	9	3	8	2	0	5
0	1	0	4	2	6	5	3	5	3	8	0	0	3	4	1	5	3	0	8
3	0	6	2	7	1	1	8	1	7	1	3	8	9	7	6	7	4	1	6
7	5	1	7	1	9	8	0	6	9	4	9	9	3	7	1	9	2	2	5
3	7	8	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	0
1	2	3	4	5	6	7	8	9	8	1	0	5	5	1	9	0	4	1	9
3	8	4	7	7	8	5	0	6	5	5	3	3	3	9	8	1	4	0	6
1	0	0	6	2	1	1	3	2	8	8	7	8	4	6	0	2	0	3	6
8	7	1	5	9	9	3	2	4	9	4	6	5	3	2	8	5	9	4	1
6	5	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7
8	9	0	1	2	3	4	5	6	7	8	9	6	4	2	6	4	7	5	5
4	7	8	9	2	9	3	9	3	8	2	0	9	8	0	5	6	0	1	0
4	2	6	5	5	5	4	3	4	1	5	3	0	8	3	0	6	2	7	1
1	8	1	7	1	3	8	5	4	2	0	9	7	6	7	4	1	6	8	4
7	5	1	2	6	7	1	9	8	0	6	9	4	9	9	6	2	3	7	1
9	2	2	5	3	7	8	0	1	2	3	4	5	6	7	8	0	1	2	3
4	5	6	7	8	0	1	2	3	4	5	6	7	8	9	2	1	2	1	3
9	9	8	5	3	7	0	7	7	5	7	9	9	4	7	0	3	4	1	4
4	7	5	8	1	4	8	4	1	8	6	6	4	6	3	5	7	2	5	9

Connections per Neuron



Number of Neurons



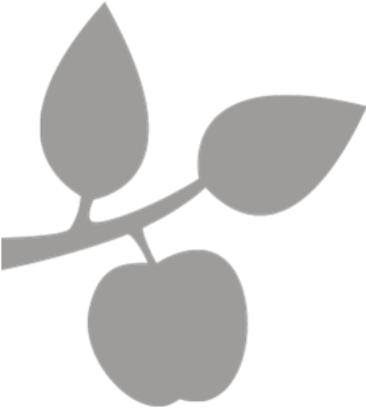


DM873

Deep Learning

Spring 2019

Lecture 2 – Recap: Math



Recap Math

- **Linear Algebra**
- **Probability**

Scalar and Vector

Scalar

- Single number
- Normally: $x \in \mathbb{R}$ or $x \in \mathbb{N}$

Vector

- An array of numbers
 - Arranged in order
 - Each no. identified by an index
- $\mathbf{x} = \begin{bmatrix} x_1 \\ \dots \\ x_d \end{bmatrix}$ and $\mathbf{x}^T = [x_1, \dots, x_d]$, $\mathbf{x} \in \mathbb{R}^d$
- We think of vectors as points in space
 - Each element gives coordinate along an axis

Matrix

- 2-D array of numbers
- Each element identified by two indices
- Denoted by bold typeface \mathbf{A}
- Elements indicated as $A_{m,n}$

$$\mathbf{A} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$$

- $A_{i:}$ is i th row of \mathbf{A} , $A_{:\mathit{j}}$ is j th column
- \mathbf{A} has m rows and n columns, then $A \in \mathbb{R}^{m \times n}$

Tensor

- Sometimes need an array with more than two axes
- An array arranged on a regular grid with variable number of axes is referred to as a **Tensor**
- We denote a tensor with bold non-italic typeface: **A** in comparison to a normal Matrix *A*
- I try to keep the notation consistent, but double-check 😊
- An Element (i, j, k) of tensor denoted by $A_{i,j,k}$

Transpose of a Matrix

- Mirror image across principal diagonal

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \end{bmatrix}, A^T = \begin{bmatrix} A_{1,1} & A_{2,1} & A_{3,1} \\ A_{1,2} & A_{2,2} & A_{3,2} \end{bmatrix}$$

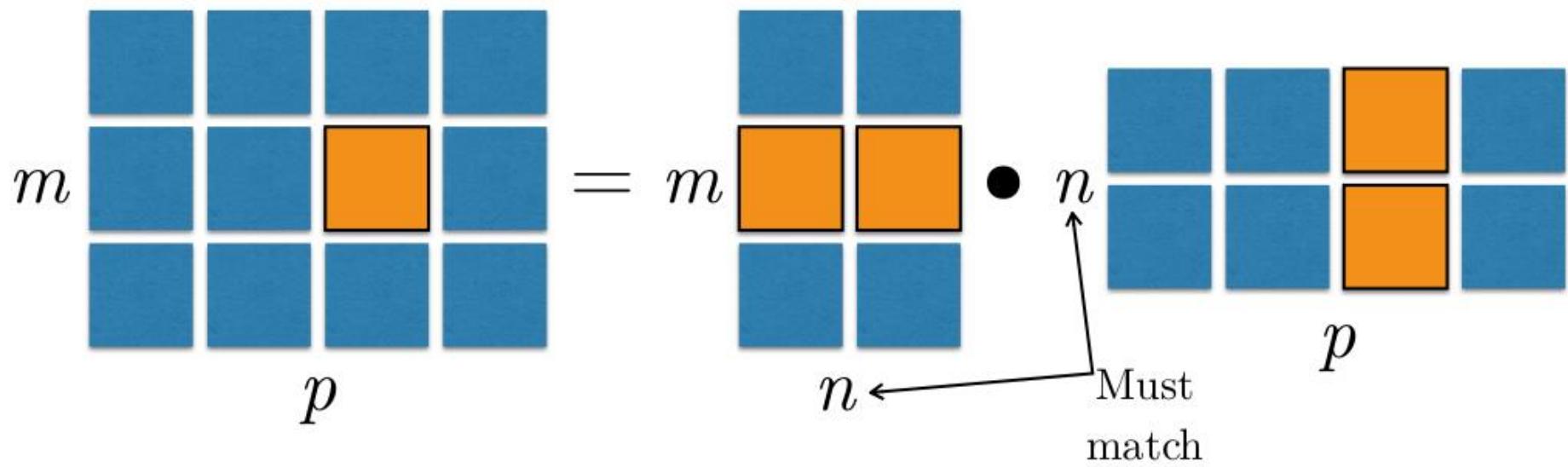
- Vectors are then either a Matrix with a single column or a single row; For convenience often written in a line: $x^T = [x_1, x_2, x_3]$
- Scalar is a Matrix with just one element, thus:

$$x^T = x$$

Matrix Product

$$C = AB.$$

$$C_{i,j} = \sum_k A_{i,k}B_{k,j}.$$



Matrix Product Properties

- Distributivity over addition:

$$A(B + C) = AB + AC$$

- Associativity:

$$A(BC) = (AB)C$$

- Not commutative: $AB = BA$ is not always true

- Dot product between vectors is commutative:

$$x^T y = y^T x$$

- Transpose of a matrix product has a simple form:

$$(AB)^T = B^T A^T$$

Square Matrices: Linear Transformations

$$Ax = b$$

- Where $A \in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$
- More Explicitly:

$$A_{1,1}x_1 + \cdots + A_{1,n}x_n = b_1$$

$$A_{2,1}x_1 + \cdots + A_{2,n}x_n = b_2$$

...

$$A_{n,1}x_1 + \cdots + A_{n,n}x_n = b_n$$

- In total: n equations with n unknowns
- Can view A as a linear transformation of vector x to vector b
- More interesting: We want to solve for unknowns $x = \{x_1, \dots, x_n\}$ with A and b being constraints.

Square Matrices: Determinant of a Matrix

- Determinant of a square matrix $\det(A)$ is a mapping to a scalar
- It is equal to the product of all eigenvalues of the matrix
- Measures how much multiplication by the matrix expands or contracts space

Square Matrices: How To Calculate a Determinant

- For a 1x1 matrix:

$$\det(a_{11}) = a_{11}$$

- For a 2x2 matrix:

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}$$

- For a 3x3 matrix:

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} =$$

$$a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} - a_{13}a_{22}a_{31} = \\ a_{11}(a_{22}a_{33} - a_{23}a_{32}) - a_{12}(a_{21}a_{33} - a_{23}a_{31}) + a_{13}(a_{21}a_{32} - a_{22}a_{31})$$

Square Matrices: More on Determinants

- The determinant of a matrix can be calculate by multiplying each element of one of its lines by the determinant of a sub-matrix formed by the elements that stay when one suppress the line and column containing this element. One give to the obtained product the sign $(-1)^{i+j}$.
- Determinant exists only for square matrices
- The determinant of a matrix is zero if and only if there exist a linear relationship between the lines or the columns of the matrix

Square Matrices: Identify and Inverse

- Matrix inversion is a powerful tool to analytically solve
$$Ax = b$$
- Needs concept of Identity matrix
- Identity matrix does not change value of vector when we multiply the vector by identity matrix
 - Denote identity matrix that preserves n-dimensional vectors as I_n
 - For Example:

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- The inverse of a **square** Matrix A is defined as:

$$A^{-1}A = I$$

Square Matrices: Solving for x

- Using the inverse of A , we can easily solve for x :

$$\begin{aligned} Ax &= b \\ A^{-1}Ax &= A^{-1}b \\ Ix &= A^{-1}b \\ x &= A^{-1}b \end{aligned}$$

- Numerically unstable, but useful for abstract analysis
- Matrix can't be inverted if...
 - More rows than columns
 - More columns than rows
 - Redundant rows/columns ("linearly dependent", "low rank")

Linear Equations: Closed-Form Solutions

1. Matrix Formulation: $\mathbf{Ax} = \mathbf{b}$
Solution: $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots &\quad \vdots &\quad \vdots &\quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned}$$

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

2. Gaussian Elimination
followed by back-substitution

$$\begin{aligned} x + 3y - 2z &= 5 \\ 3x + 5y + 6z &= 7 \\ 2x + 4y + 3z &= 8 \end{aligned}$$

$$\begin{array}{c} L_2 - 3L_1 \rightarrow L_2 \quad L_3 - 2L_1 \rightarrow L_3 \quad -L_2/4 \rightarrow L_2 \\ \hline \end{array}$$
$$\left[\begin{array}{ccc|c} 1 & 3 & -2 & 5 \\ 3 & 5 & 6 & 7 \\ 2 & 4 & 3 & 8 \end{array} \right] \sim \left[\begin{array}{ccc|c} 1 & 3 & -2 & 5 \\ 0 & -4 & 12 & -8 \\ 2 & 4 & 3 & 8 \end{array} \right] \sim \left[\begin{array}{ccc|c} 1 & 3 & -2 & 5 \\ 0 & -4 & 12 & -8 \\ 0 & -2 & 7 & -2 \end{array} \right] \sim \left[\begin{array}{ccc|c} 1 & 3 & -2 & 5 \\ 0 & 1 & -3 & 2 \\ 0 & -2 & 7 & -2 \end{array} \right]$$
$$\sim \left[\begin{array}{ccc|c} 1 & 3 & -2 & 5 \\ 0 & 1 & -3 & 2 \\ 0 & 0 & 1 & 2 \end{array} \right] \sim \left[\begin{array}{ccc|c} 1 & 3 & -2 & 5 \\ 0 & 1 & 0 & 8 \\ 0 & 0 & 1 & 2 \end{array} \right] \sim \left[\begin{array}{ccc|c} 1 & 0 & 0 & -15 \\ 0 & 1 & 0 & 8 \\ 0 & 0 & 1 & 2 \end{array} \right]$$

Disadvantages of Closed Form Solutions

- If A^{-1} exists, the same A^{-1} can be used for any given b
 - But A^{-1} cannot be represented with sufficient precision
 - It is not used in practice
- Gaussian elimination also has disadvantages
 - numerical instability (division by small no.)
 - $O(n^3)$ for a $n \times n$ matrix
- Software solutions use value of b in finding x
 - E.g., difference (derivative) between b and output is used iteratively

Norms

- Used for measuring the size of a vector
- Norms map vectors to non-negative values
- Norm of vector x is distance from origin to x
- A norm is any function f that satisfies:
 1. $f(x) = 0 \Rightarrow x = 0$
 2. $f(x + y) \leq f(x) + f(y)$
 3. $\forall \alpha \in \mathbb{R}, f(\alpha x) = |\alpha|f(x)$

L^p Norms for Vectors

$$L^p = \left(\sum_i |x_i|^p \right)^{1/p}$$

- L^2 Norm
 - Called Euclidean norm, written simply as $\|x\|$
 - Squared Euclidean norm is same as $x^T x$
- L^1 Norm
 - Useful when 0 and non-zero have to be distinguished (since L^2 increases slowly near origin, e.g., $0.1^2 = 0.01$)
- L^∞ Norm
 - $\|x\|_\infty = \max |x_i|$
 - Called max norm

Angle between Vectors

- Dot product of two vectors can be written in terms of their L^2 norms and angle θ between them

$$\mathbf{x}^T \mathbf{y} = \|\mathbf{x}\|_2 \|\mathbf{y}\|_2 \cos \theta$$

Size of a Matrix

- Frobenius norm

$$\|A\| = \left(\sum_{i,j} A_{i,j}^2 \right)^{1/2}$$

- It is analogous to L^2 norm of a vector

Special Matrices & Vectors

■ Unit Vector

- A vector with unit norm

$$\|x\|_2 = 1$$

■ Orthogonal Vectors

- A vector x and a vector y are orthogonal to each other if

$$x^T y = 0$$

- Vectors are at 90 degrees to each other

■ Orthogonal Matrix

- A square matrix columns and rows are orthogonal unit vectors

$$A^{-1} = A^T$$

Special Matrices & Vectors

■ Diagonal Matrix

- Mostly zeros, with non-zero entries in diagonal
- $\text{diag}(\mathbf{v})$ is a square diagonal matrix with diagonal elements given by entries of vector \mathbf{v}
- Multiplying $\text{diag}(\mathbf{v})$ by vector \mathbf{x} only needs to scale each element x_i by v_i

■ Symmetric Matrix

- Is equal to its transpose:

$$\mathbf{A} = \mathbf{A}^T$$

- E.g., a distance matrix is symmetric with $A_{ij} = A_{ji}$

Matrix Decomposition

- Matrices can be decomposed into factors to learn universal properties about them not discernible from their representation
 - E.g., from decomposition of integer into prime factors $12=2\times2\times3$ we can discern that
 - 12 is not divisible by 5 or
 - any multiple of 12 is divisible by 3
- Analogously, a matrix is decomposed into Eigenvalues and Eigenvectors to discern universal properties

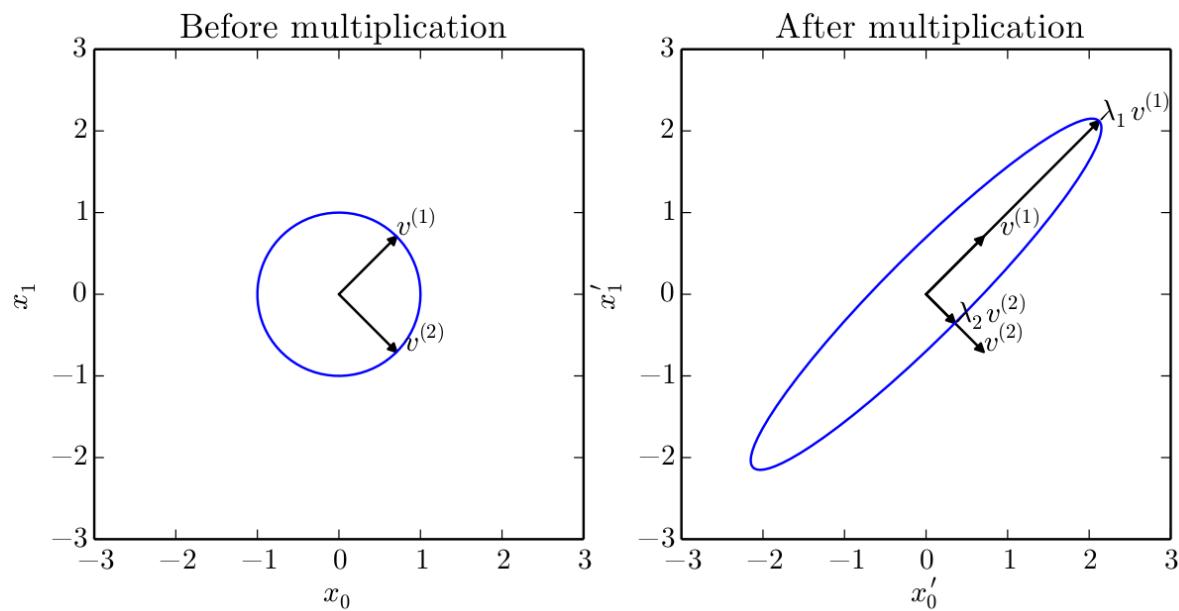
Eigenvector

- An eigenvector of a square matrix A is a non-zero vector v such that multiplication by A only changes the scale of v

$$Av = \lambda v$$

- The scalar λ is known as eigenvalue

- If v is an eigenvector of A , so is any rescaled vector sv .
- sv still has the same eigen value.
- Thus, the unit Eigenvector is used



Eigenvalue and Characteristic Polynomial

- Consider $A\mathbf{v} = \mathbf{w}$
- If \mathbf{v} and \mathbf{w} are scalar multipliers, then

$$A\mathbf{v} = \lambda\mathbf{v}$$

$$(A - \lambda I)\mathbf{v} = 0$$

- This has a non-zero solution if

$$|A - \lambda I| = 0$$

- The nulls of the polynomial of degree n are the eigenvalues of A

Example

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

- Then, the characteristic polynomial is

$$|A - \lambda I| = \begin{bmatrix} 2 - \lambda & 1 \\ 1 & 2 - \lambda \end{bmatrix} = 3 - 4\lambda + \lambda^2$$

- The polynomial has the roots $\lambda = 1$ and $\lambda = 3$ which are the eigenvalues of A .
- The eigenvectors can be found by solving the equation $Av = \lambda v$ using the different eigenvectors:

$$v_{\lambda=1} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, v_{\lambda=3} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Eigendecomposition

- Suppose that matrix A has n linearly independent eigenvectors $\{v^{(1)}, \dots, v^{(n)}\}$ with the eigenvalues $\{\lambda_1, \dots, \lambda_n\}$
- Concatenate eigenvectors to form matrix V
- Concatenate eigenvalues to form vector $\lambda = [\lambda_1, \dots, \lambda_n]$ (normally in descending order)
- The Eigendecomposition of A is given by

$$A = V \text{diag}(\lambda) V^{-1}$$

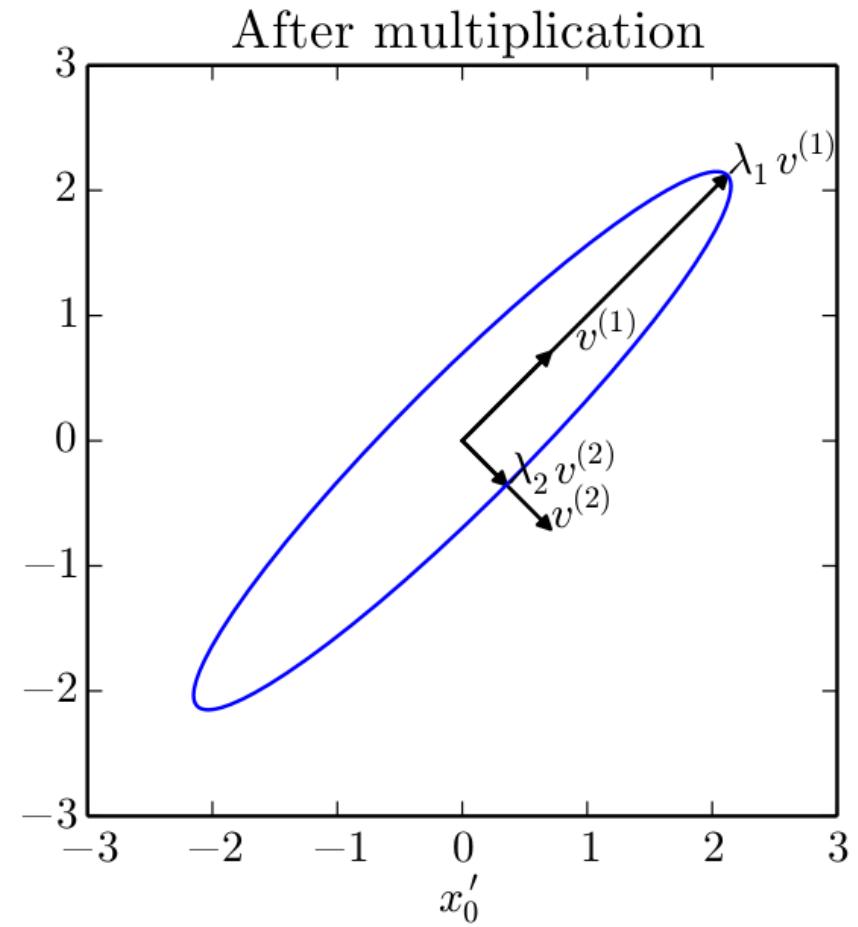
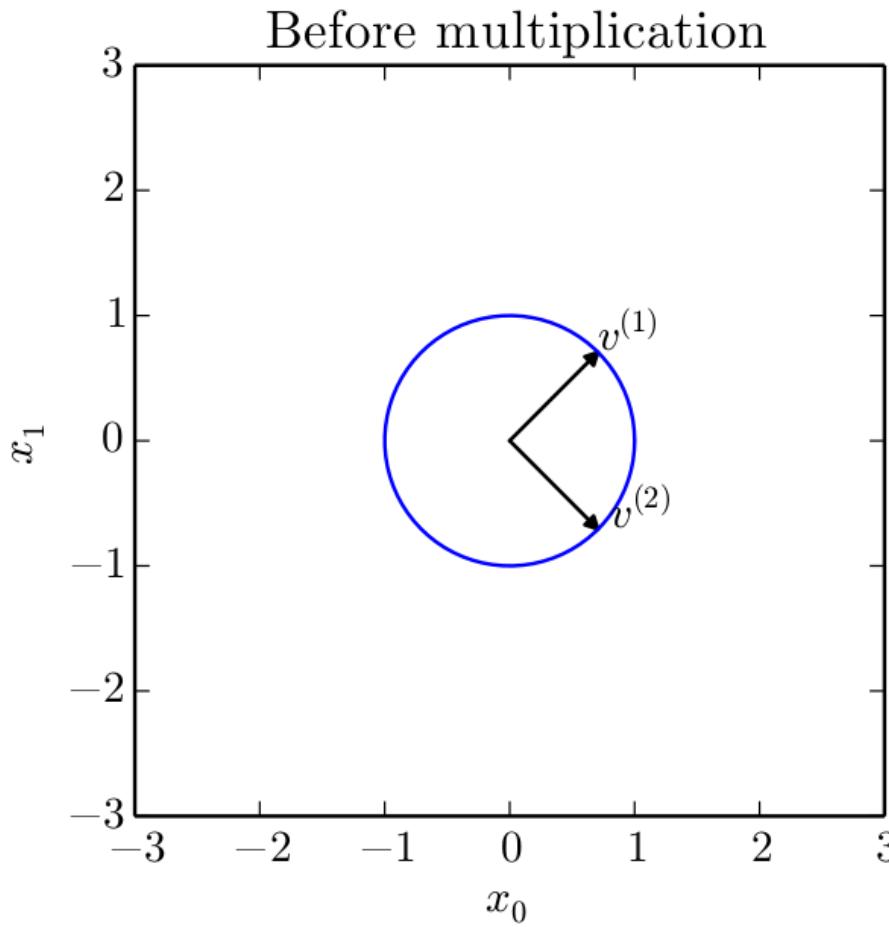
Decomposition of Symmetric Matrix

- Every real symmetric matrix A can be decomposed into real-valued eigenvectors and eigenvalues

$$A = Q\Lambda Q^T$$

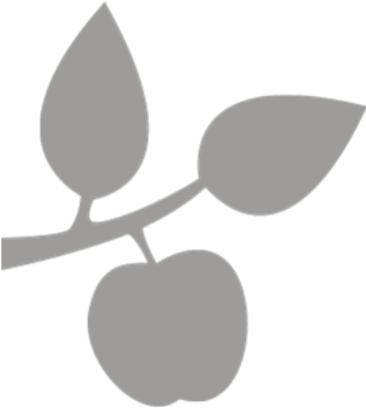
- where Q is an orthogonal matrix composed of the eigenvectors and Λ the diagonal matrix of eigenvalues.
- We can think of A as scaling space by λ_i in direction $v^{(i)}$

Effect of Eigenvectors and Eigenvalues



Effect of Eigenvectors and Eigenvalues

- A matrix whose eigenvalues are
 - all positive is called **positive definite**
 - all positive or zero-valued is called **positive semidefinite**.
 - all negative is called **negative definite**
 - all negative or zero-valued is called **negative semidefinite**. Positive
- Semidefinite matrices are interesting because they guarantee that
$$\forall \mathbf{x}, \mathbf{x}^T \mathbf{A} \mathbf{x} \geq 0$$
- Positive definite matrices additionally guarantee that
$$\mathbf{x}^T \mathbf{A} \mathbf{x} = 0 \Rightarrow \mathbf{x} = 0$$



Recap Math

- **Linear Algebra**
- **Probability**

Why Probability?

- Much of CS deals with entities that are certain
 - CPU executes flawlessly
 - At least almost ... there are CPU bugs and CPUs can also be broken
 - CS and software engineers work in clean and certain environment
 - Surprising that ML heavily uses probability theory
- Reasons for ML use of probability theory
 - Must always deal with uncertain quantities
 - Also with non-deterministic (stochastic) quantities
 - Many sources for uncertainty and stochasticity

Sources of Uncertainty

1. Inherent stochasticity of system being modeled
 - Subatomic particles are probabilistic
 - Cards shuffled in random order
2. Incomplete observability
 - Deterministic systems appear stochastic when not all variables are observed
3. Incomplete modeling
 - Discarded information results in uncertain predictions

Practical to use uncertain rule

- Simple rule “Most birds fly” is cheap to develop and broadly useful
- Rules of the form “Birds fly, except for very young birds that have not learned to fly, sick or injured birds that have lost ability to fly, flightless species of birds...” are expensive to develop, maintain and communicate
 - Also still brittle and prone to failure

Tools of Probability

- Probability theory was originally developed to analyze frequencies of events
 - Such as drawing a hand of cards in poker
 - These events are repeatable
 - If we repeated experiment infinitely many times, proportion of p of outcomes would result in that outcome
- Is it applicable to propositions not repeatable?
 - Patient has 40% chance of flu
 - Cannot make infinite replicas of the patient
 - We use probability to represent degree of belief
- Former is frequentist probability, latter Bayesian

Logic and Probability

- Reasoning about uncertainty behaves the same way as frequentist probabilities
- Probability is an extension of logic to deal with uncertainty
- Logic provides rules for determining what propositions are implied to be true or false
- Probability theory provides rules for determining the likelihood of a proposition being true given the likelihood of other propositions

Random Variables

- A **random variable** X is a variable that can take on different values randomly
- On its own, a random variable is just a description of the states that are possible;
- It must be coupled with a probability distribution that specifies how likely each of these states are.
- Random variables may be **discrete** or **continuous**

Probability Distributions

- A probability distribution is a description of how likely a random variable or a set of random variables is to take each of its possible states
- The way to describe the distribution depends on whether it is discrete or continuous

Probability Mass Functions

- The probability distribution over discrete variables is given by a probability mass function
- PMFs of variables are denoted by P and inferred from their argument, e.g., $P(x)$, $P(y)$
- They can act on many variables and is known as a joint distribution, written as $P(x, y)$
- To be a PMF it must satisfy:
 - Domain of P is the set of all possible states of x
 - $\forall x \in X, 0 \leq P(x) \leq 1$
 - $\sum_{x \in X} P(x) = 1$ (normalization)

Continuous Variables and PDFs

- When working with continuous variables, we describe probability distributions using probability density functions
- To be a pdf p must satisfy:
 - The domain of p must be the set of all possible states of X
 - $\forall x \in X, p(x) \geq 0$. Note, there is **no requirement** for $p(x) \leq 1$.
 - $\int p(x)dx = 1$

Marginal Probability

- Sometimes we know the joint distribution of several variables
- And we want to know the distribution over some of them
- It can be computed:
 - In the discrete case:

$$\forall x \in X, P(X = x) = \sum_y P(X = x, Y = y)$$

- In the continues case:

$$p(x) = \int p(x, y) dy$$

Conditional Probability

- We are often interested in the probability of an event given that some other event has happened
- This is called conditional probability
- It can be computed using

$$P(Y = y|X = x) = \frac{P(Y = y, X = x)}{P(X = x)}$$

- Bayes Theorem

$$P(Y = y|X = x) = \frac{P(X = x|Y = y) \cdot P(Y = y)}{P(X = x)}$$

Bayes Theorem

- The Disease:
 - Let's assume we invented a test to check if an individual has a disease D
 - The disease is rare, we have 100 sick in 1, 000, 000 people
- Our Test:
 - When presented with a sick patient, we receive a positive result in 99% of the cases
 - When presented with a healthy person, we receive a (falsely) positive result in 1% of the cases
- Question:
 - We are tested, and shockingly the result is positive!
 - How likely is it, that we are actually sick?

Bayes Theorem

- $\Pr[S] = \frac{100}{1000000} = 0.0001$ is the probability a random person is sick
- $\Pr[P] = \Pr[P|S] \cdot \Pr[S] + \Pr[P|\bar{S}] \Pr[\bar{S}] =$
 $0.99 \cdot 0.0001 + 0.01 \cdot 0.9999 = 0.0101$
 - The Probability that we receive a positive test
 - Is the probability of a true positive + the probability of a false negative
- We are interested in $\Pr[S|P]$

Bayes Theorem

- $\Pr[S] = \frac{100}{1000000} = 0.0001$ is the probability a random person is sick
- $\Pr[P] = \Pr[P|S] \cdot \Pr[S] + \Pr[P|\bar{S}] \Pr[\bar{S}] = 0.99 \cdot 0.0001 + 0.01 \cdot 0.9999 = 0.0101$
 - The Probability that we receive a positive test
 - Is the probability of a true positive + the probability of a false negative
- We are interested in $\Pr[S|P]$
- Lets use Bayes Theorem:

$$\Pr[S|P] = \frac{\Pr[P|S] \Pr[S]}{\Pr[P]} = \frac{0.99 \cdot 0.0001}{0.0101} = 0.01$$

- Even with a positive result, the chance that we are actually sick is only 1%
- Compared to the 99% that a random test is correct

Conditional Probability & Independence

- Any probability distribution over many variables can be decomposed into conditional distributions over only one variable

$$P(X^{(1)}, \dots, X^{(n)}) = P(X^{(1)}) \prod_{i=2}^n P(X^{(i)} | X^{(1)}, \dots, X^{(i-1)})$$

- For example:

$$P(A, B, C) = P(A|B, C)P(B|C)P(C)$$

- Two variables x and y are **independent** if their probability distribution can be expressed as a product of two factors:

$$\forall x \in X, y \in Y, p(X = x, Y = y) = p(X = x) \cdot p(Y = y)$$

Expectation

- Expectation or expected value of $f(x)$ w.r.t. $P(X)$ is the average or mean value that f takes on when x is drawn from P
- For discrete variables:

$$E[f(x)] = \sum_x P(x)f(x)$$

- For continuous variables:

$$E[f(x)] = \int p(x)f(x)dx$$

Variance

$$Var(f(x)) = E[(f(x) - E[f(x)])^2]$$

- Measure how much the value of $f(x)$ vary from the expectation
- Low variance means values cluster around its expectations
- Square root of the variance is the standard deviation

Covariance

- Covariance measures how two values are **linearly** related:

$$\text{Cov}(f(x), g(y)) = E[(f(x) - E[f(x)])(g(y) - E[g(y)])]$$

- Interpretation:

- High absolute values of covariance: Values change very much & are both far from their mean
 - Positive: High values of $f(x)$ coincide with high values of $g(y)$
 - Negative: High values of $f(x)$ coincide with low values of $g(y)$

- Covariance & independence are related but not same

- Zero covariance is necessary for independence but not sufficient
 - They must not have nonlinear relationship either

Independence vs. Covariance

- Independence stronger than covariance
- Covariance & independence are related but not same
- Zero covariance is necessary for independence
 - Independent variables have zero covariance
 - Variables with non-zero covariance are dependent
- Independence is a stronger requirement
 - They not only must not have linear relationship (zero covariance)
 - They must not have nonlinear relationship either

Dependence with zero covariance

- Suppose we sample real number x from $U[-1,1]$
- Next sample a random variable s
 - with prob $1/2$ we choose $s = 1$ otherwise $s = -1$
- Generate random variable y assigning $y = sx$
 - i.e., $y = -x$ or $y = x$ depending on s
 - Clearly x and y are not independent
 - x completely determines magnitude of y
- However $Cov(x, y) = 0$
 - Because when x has a high value y can be high or low depending on s

Bernoulli Distribution

- Distribution over a single binary random variable
- It is controlled by a single parameter $\phi \in [0,1]$
 - Which gives the probability a random variable being equal to 1
- It has the following properties

$$P(x = 1) = \phi$$

$$P(x = 0) = 1 - \phi$$

$$P(x = x) = \phi^x (1 - \phi)^{1-x}$$

$$\mathbb{E}_x[x] = \phi$$

$$\text{Var}_x(x) = \phi(1 - \phi)$$

Multinoulli Distribution

- Distribution over a single discrete variable with k different states
- It is parameterized by a vector $\mathbf{p} \in [0,1]^{k-1}$
 - where p_i is the probability of the i th state
 - The final k th state's probability is implicitly given by $1 - \mathbf{1}^T \mathbf{p}$
 - We must constrain $\mathbf{1}^T \mathbf{p} \leq 1$
- Multinoullis refer to distributions over categories
 - So we don't assume state 1 has value 1, etc.
 - For this reason we do not usually need to compute the expectation or variance of multinoulli variables since the states are not necessarily ordered

Gaussian Distribution

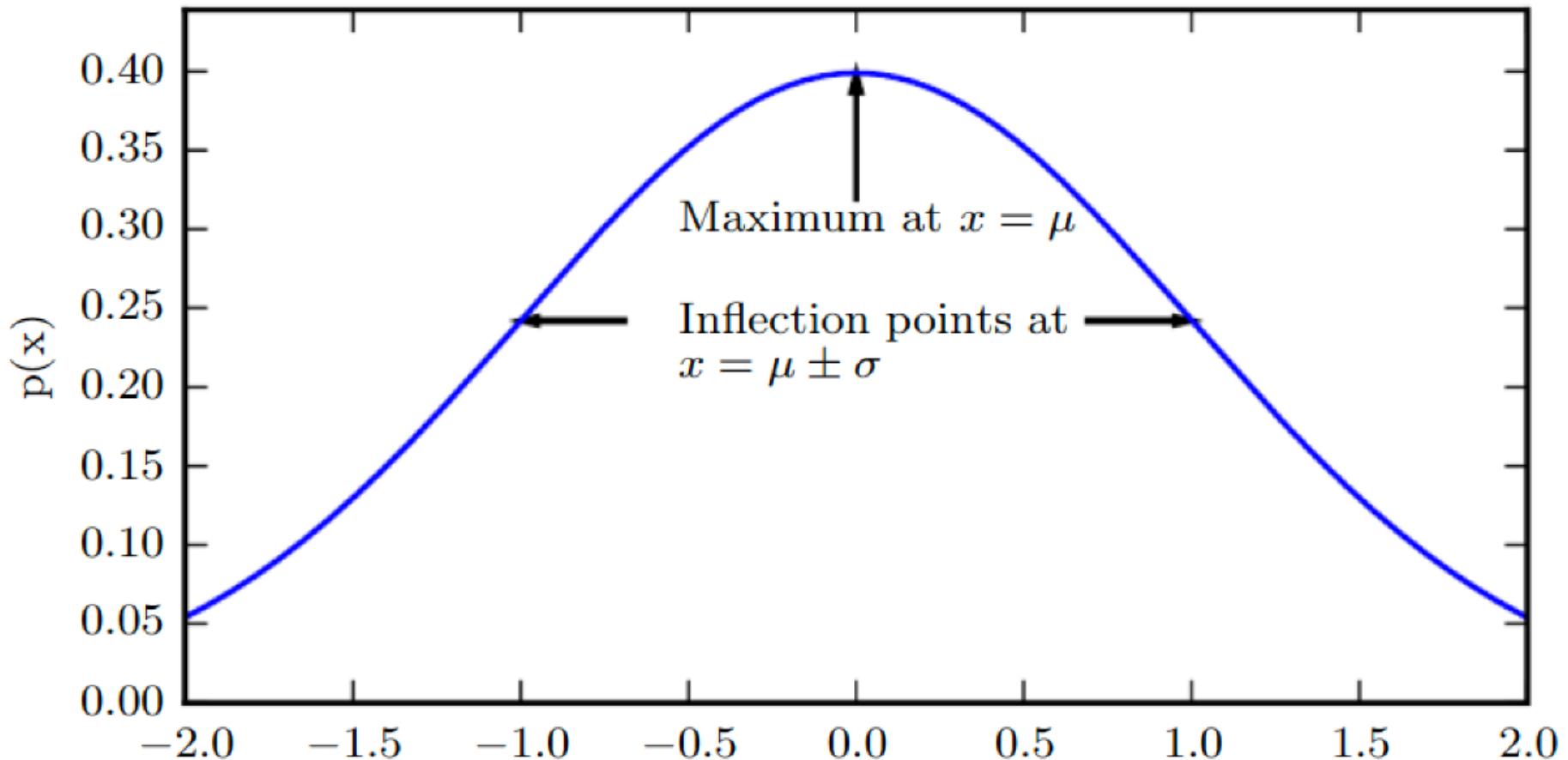
- Probably one of the most commonly used distributions

$$N(x; \mu, \sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right)$$

- Two parameters
 - μ gives the location of the central peak, which is also the mean of the distribution
 - The standard deviation is given by σ and variance by σ^2
- In case, this is evaluated frequently, sometimes parameterized with the inverse variance (or precision) β :

$$N(x; \mu, \beta^{-1}) = \sqrt{\frac{\beta}{2\pi}} \exp\left(-\frac{1}{2}\beta(x - \mu)^2\right)$$

Gaussian Distribution, $\mu = 0$, $\sigma = 1$



Multidimensional Gaussian Distributions

- The Gaussian Distribution can easily be extended to the multivariate case.
- Now, x and μ are a vector, Σ a positive semidefinite symmetric matrix (the covariance matrix):

$$N(x; \mu, \Sigma) = \sqrt{\frac{1}{(2\pi)^2 |\Sigma|}} \exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu)\right)$$

- Analogously, with the precision Matrix

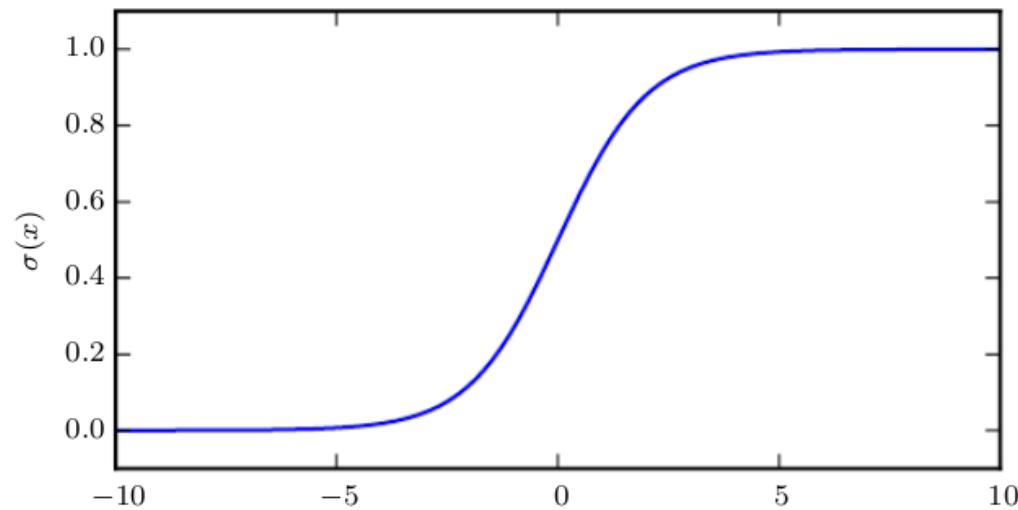
$$N(x; \mu, \beta^{-1}) = \sqrt{\frac{|\beta|}{(2\pi)^2}} \exp\left(-\frac{1}{2}(x - \mu)^T \beta (x - \mu)\right)$$

Helper Functions: Logistic Sigmoid

- Certain functions arise with probability distributions used in deep learning
- Logistic Sigmoid:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

- Commonly used to produce the ϕ parameter of a Bernoulli distribution because its range is $(0,1)$
- It saturates when x is very small/large
- Thus it is insensitive to small changes in input



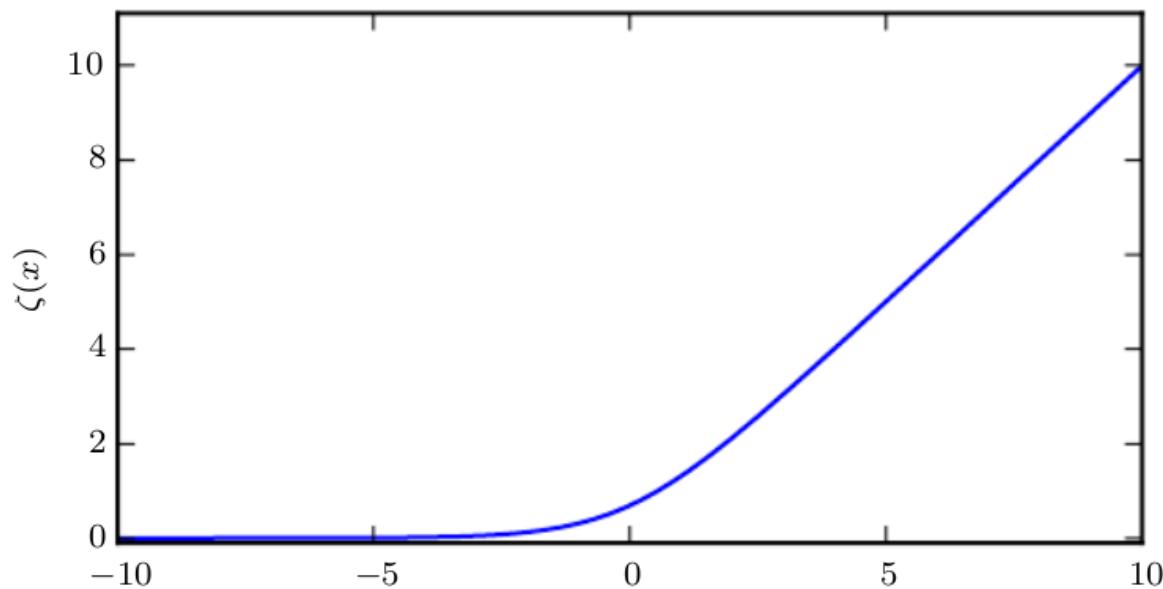
Helper Functions: Softplus Function

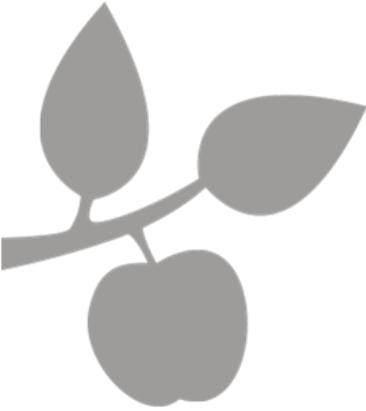
- It is defined as

$$\zeta(x) = \log(1 + \exp(x))$$

- Softplus is useful for producing the β or σ parameter of a normal distribution because its range is $(0, \infty)$

- Name arises as smoothed version of



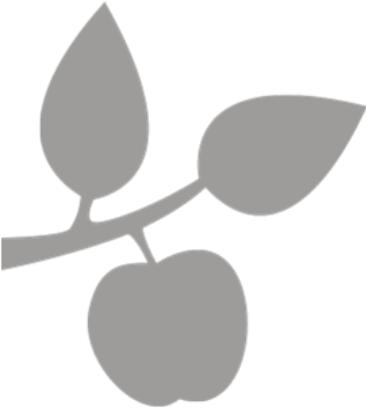


DM873

Deep Learning

Spring 2019

Lecture 3 – Machine Learning Basics



Machine Learning Basics

- **Introduction**
- **Statistical Learning**
- **Validation**

Machine Learning

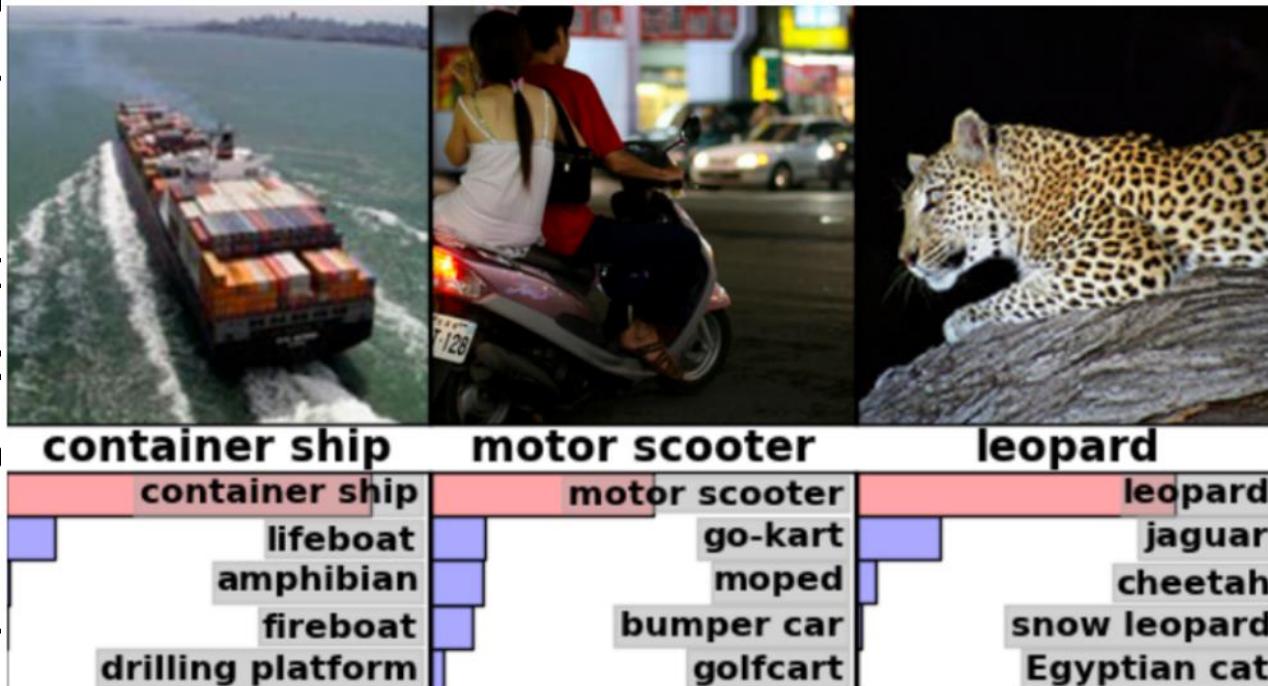
- An ML algorithm is an algorithm that is able to learn from data
 - A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at task T, as measured by P, improves with experience E
- The Task T:
 - Process of learning itself is not the task
 - Learning is our means of attaining ability to perform the task
 - Usually described in terms of how the machine learning system should process an example
 - Typically represent an example as a vector $x \in \mathbb{R}^n$ where each entry x_i of the vector is another feature

Kinds of Tasks solved using ML

- Classification
- Regression
- Transcription
- Machine Translation
- Structured Output
- Anomaly Detection
- Synthesis and Sampling
- Imputation of Missing Values
- Denoising
- Density Estimation
- ...

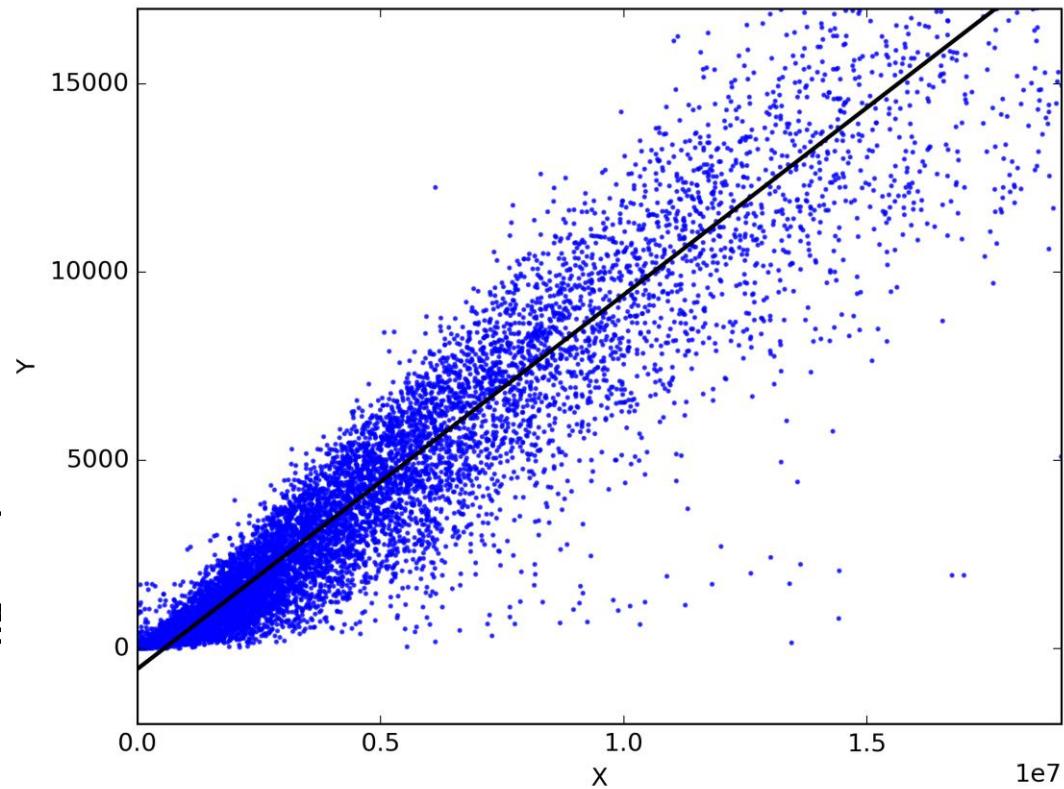
Kinds of Tasks solved using ML

- Classification
- Regression
- Transcription
- Machine Translation
- Structured Output
- Anomaly Detection
- Synthesis and Generation
- Imputation or Reconstruction
- Denoising
- Density Estimation
- ...



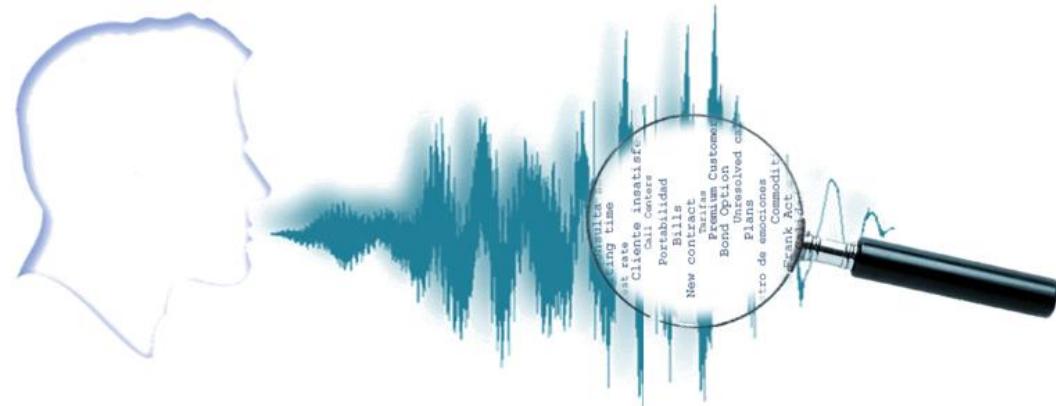
Kinds of Tasks solved using ML

- Classification
- **Regression**
- Transcription
- Machine Translation
- Structured Output
- Anomaly Detection
- Synthesis and Sampling
- Imputation of Missing
- Denoising
- Density Estimation
- ...



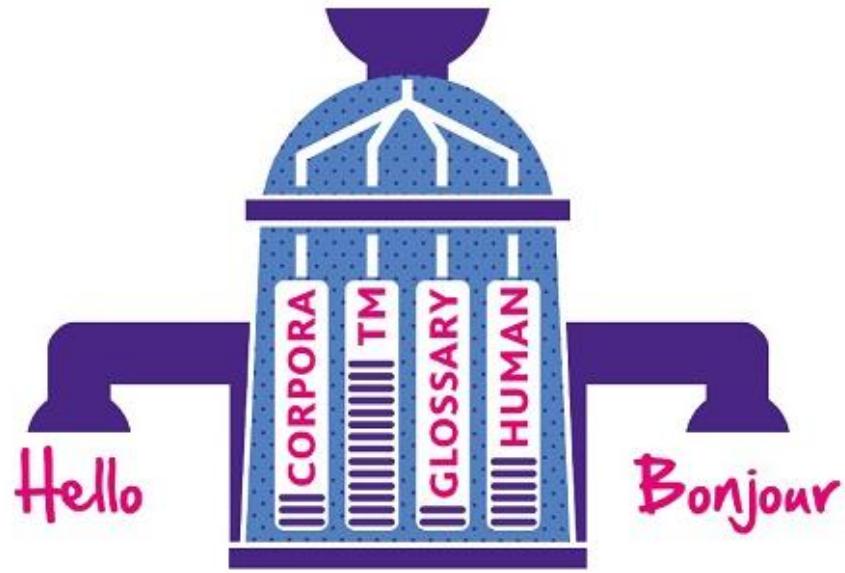
Kinds of Tasks solved using ML

- Classification
- Regression
- **Transcription**
- Machine Translation
- Structured Output
- Anomaly Detection
- Synthesis and Sampling
- Imputation of Missing Values
- Denoising
- Density Estimation
- ...



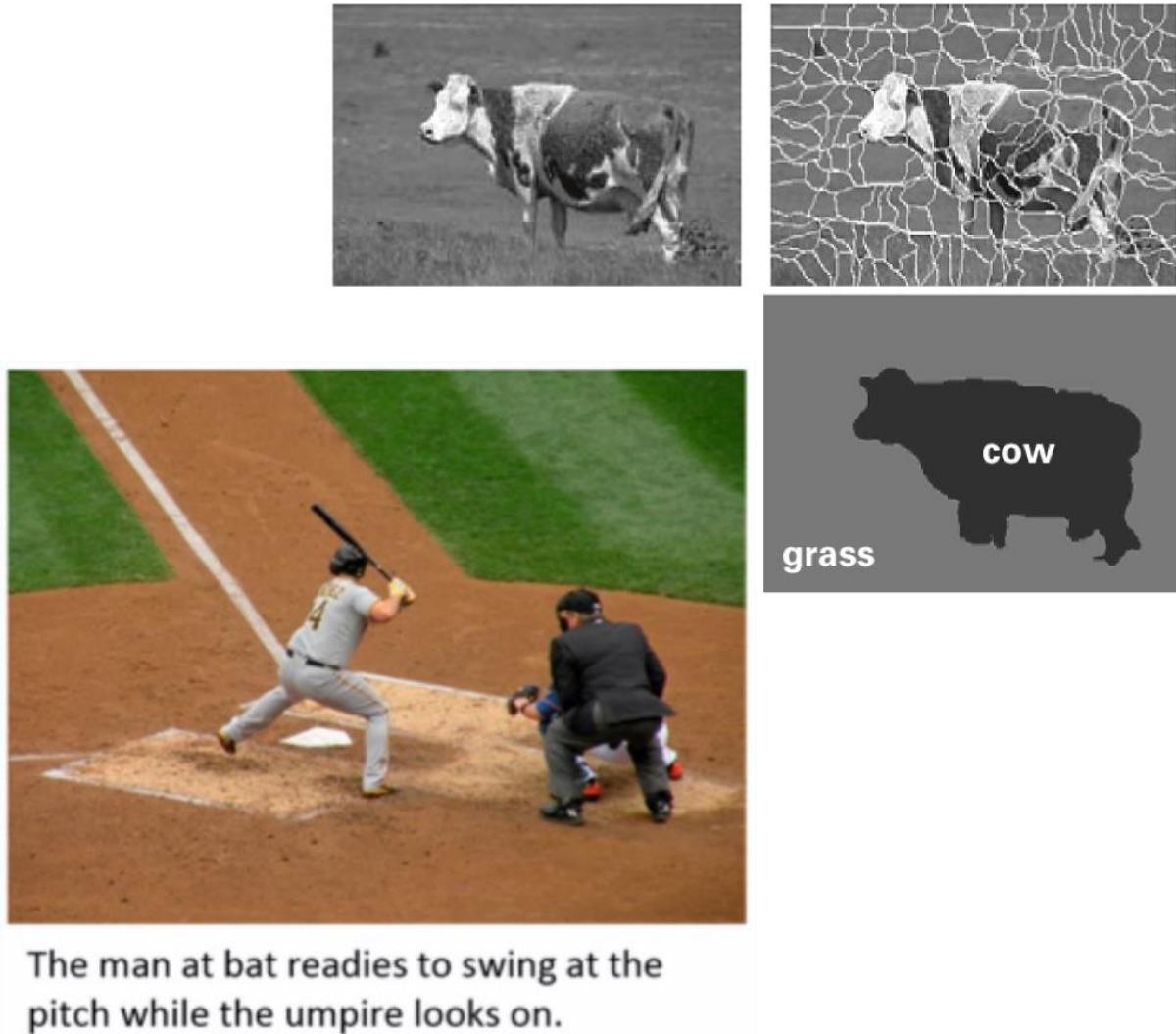
Kinds of Tasks solved using ML

- Classification
- Regression
- Transcription
- **Machine Translation**
- Structured Output
- Anomaly Detection
- Synthesis and Sampling
- Imputation of Missing Values
- Denoising
- Density Estimation
- ...



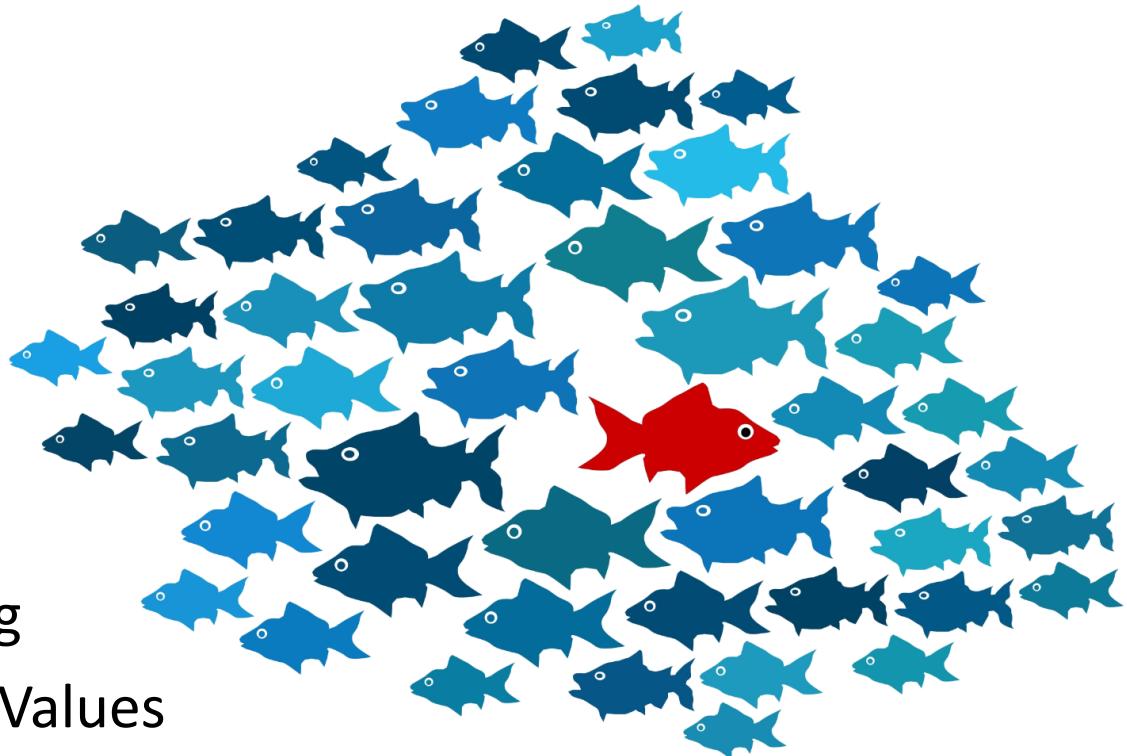
Kinds of Tasks solved using ML

- Classification
- Regression
- Transcription
- Machine Translation
- **Structured Output**
- Anomaly Detection
- Synthesis and Sampling
- Imputation of Missing Data
- Denoising
- Density Estimation
- ...

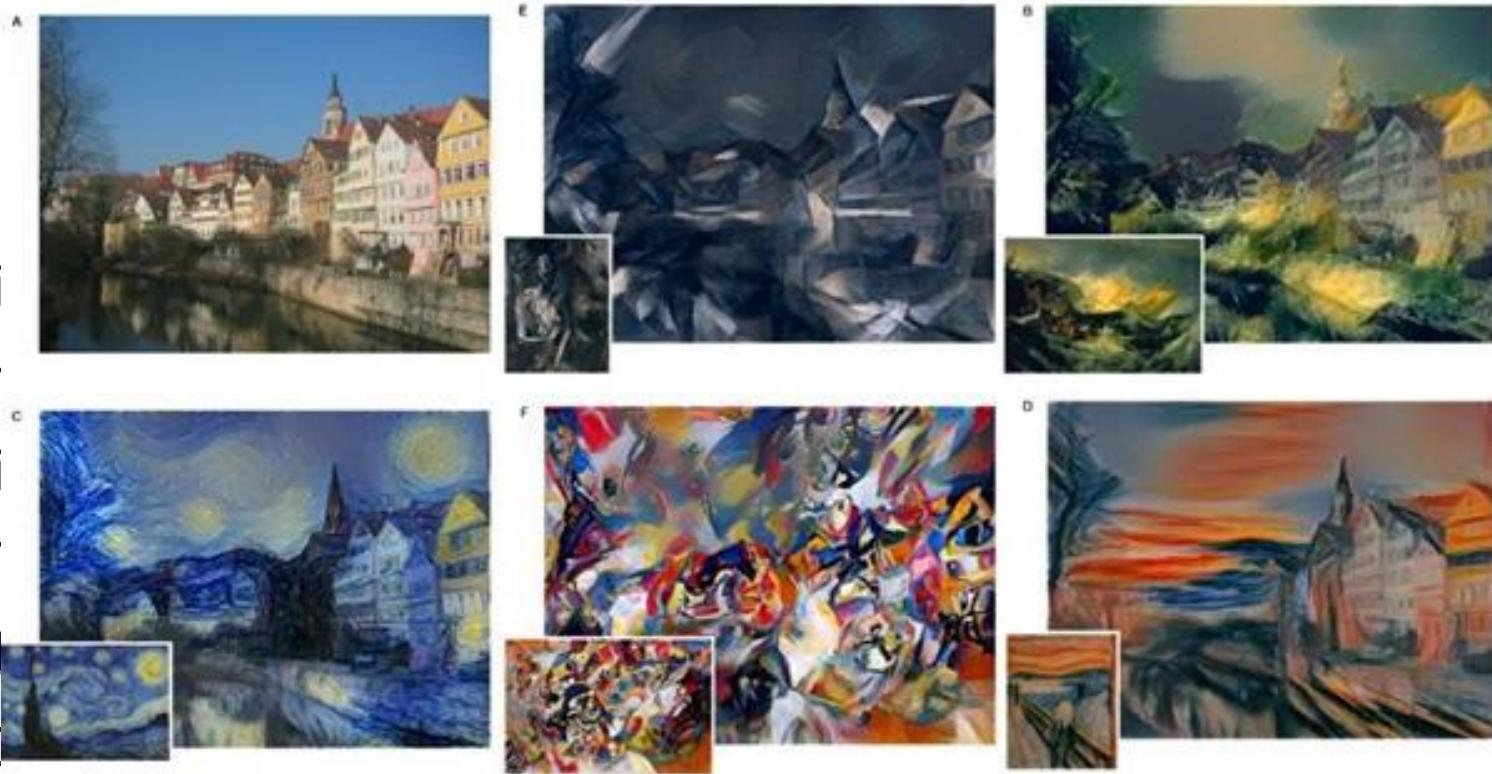


Kinds of Tasks solved using ML

- Classification
- Regression
- Transcription
- Machine Translation
- Structured Output
- **Anomaly Detection**
- Synthesis and Sampling
- Imputation of Missing Values
- Denoising
- Density Estimation
- ...

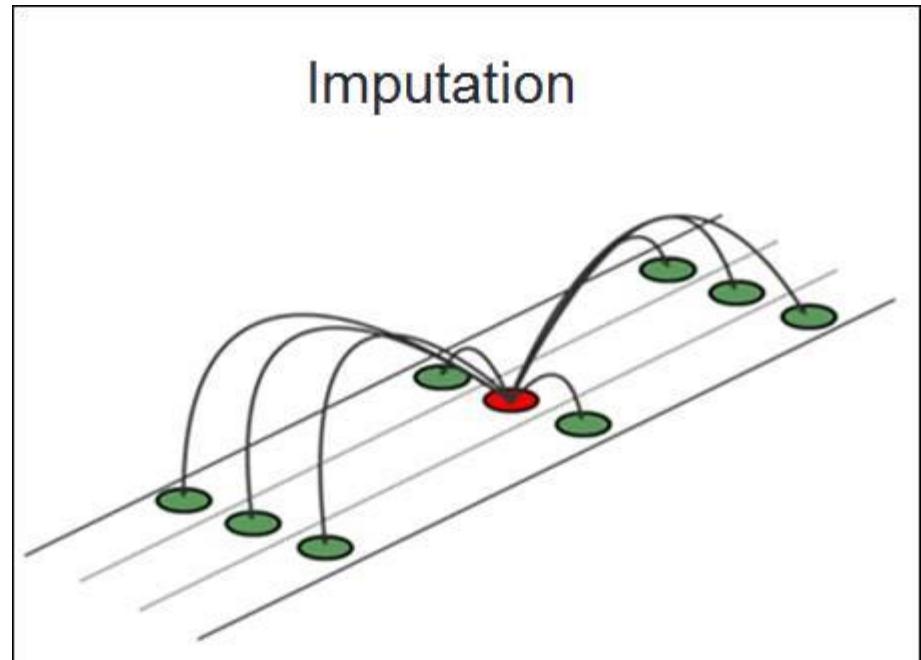


- Classification
- Regression
- Transcription
- Machine Translation
- Structured Prediction
- Anomaly Detection
- **Synthesis and Sampling**
- Imputation of Missing Values
- Denoising
- Density Estimation
- ...



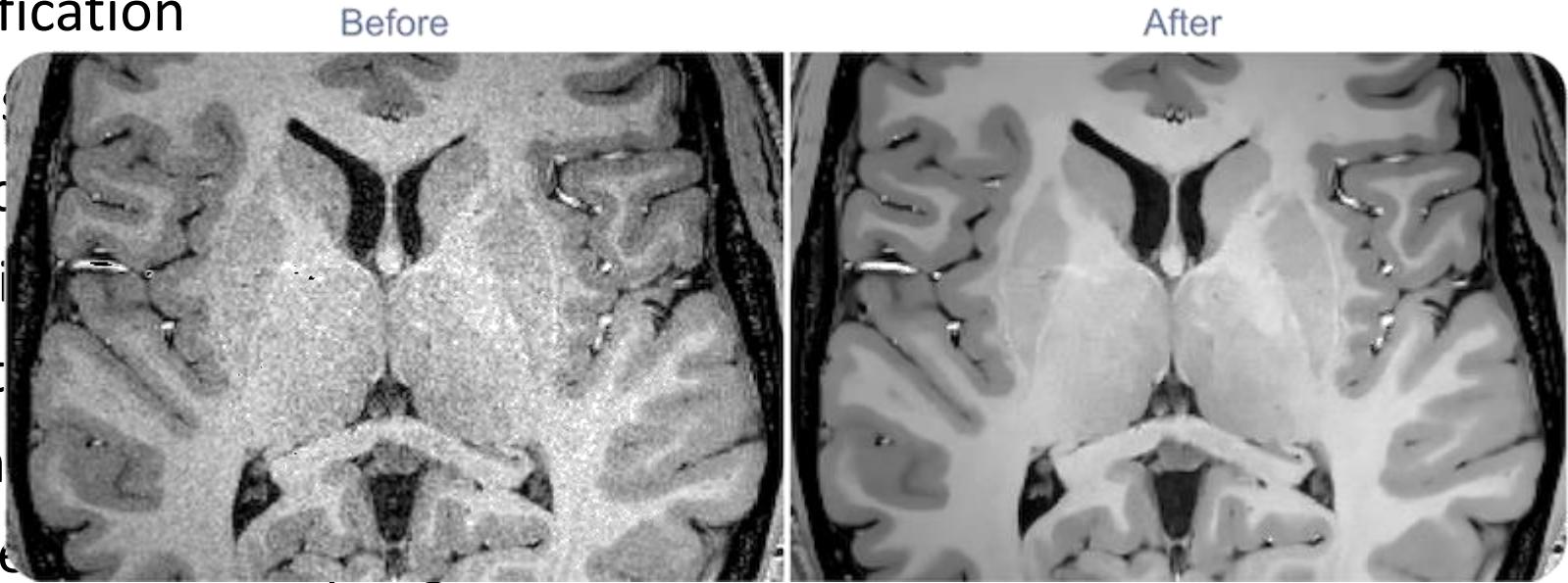
Kinds of Tasks solved using ML

- Classification
- Regression
- Transcription
- Machine Translation
- Structured Output
- Anomaly Detection
- Synthesis and Sampling
- **Imputation of Missing Values**
- Denoising
- Density Estimation
- ...



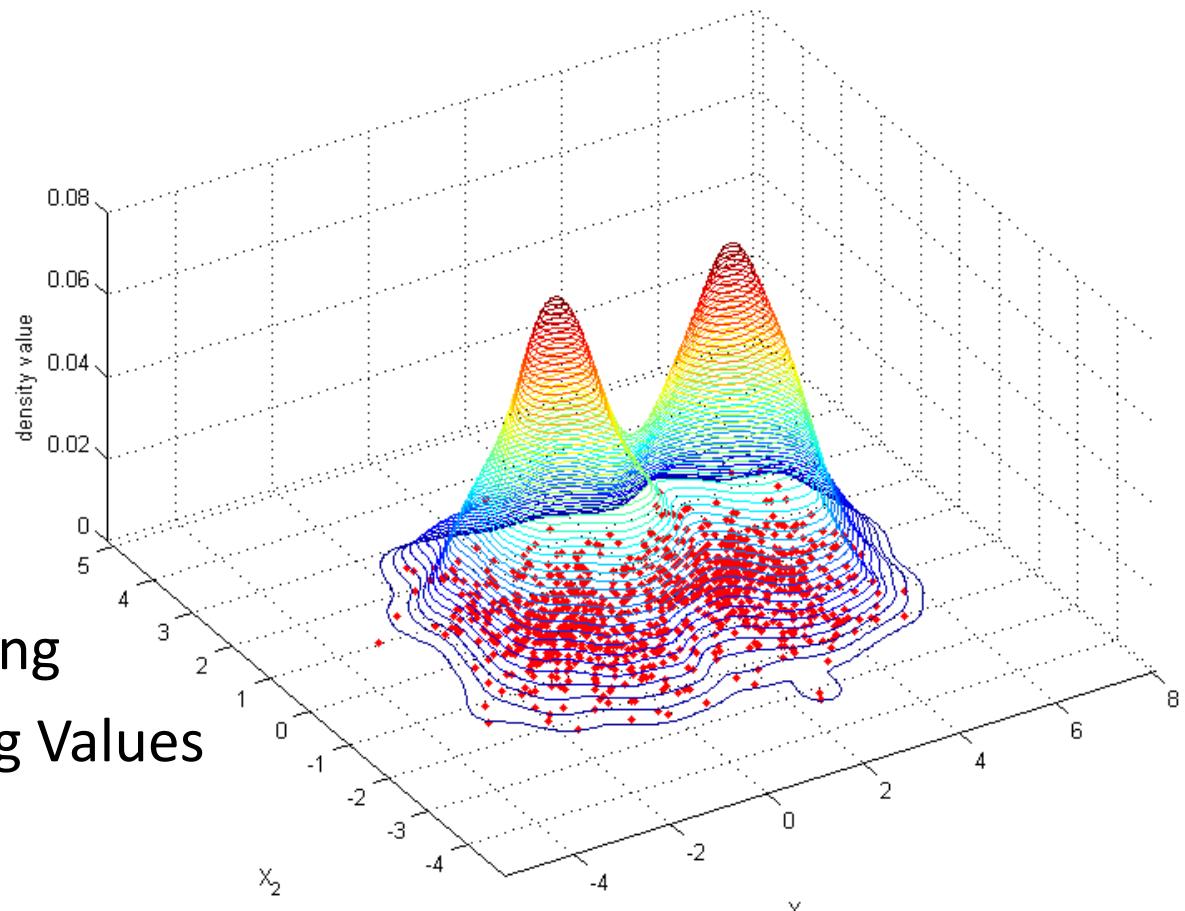
Kinds of Tasks solved using ML

- Classification
- Regression
- Translation
- Machine Translation
- Structured Prediction
- Anomaly Detection
- Synthesis
- Imputation of Missing Values
- Denoising
- Density Estimation
- ...



Kinds of Tasks solved using ML

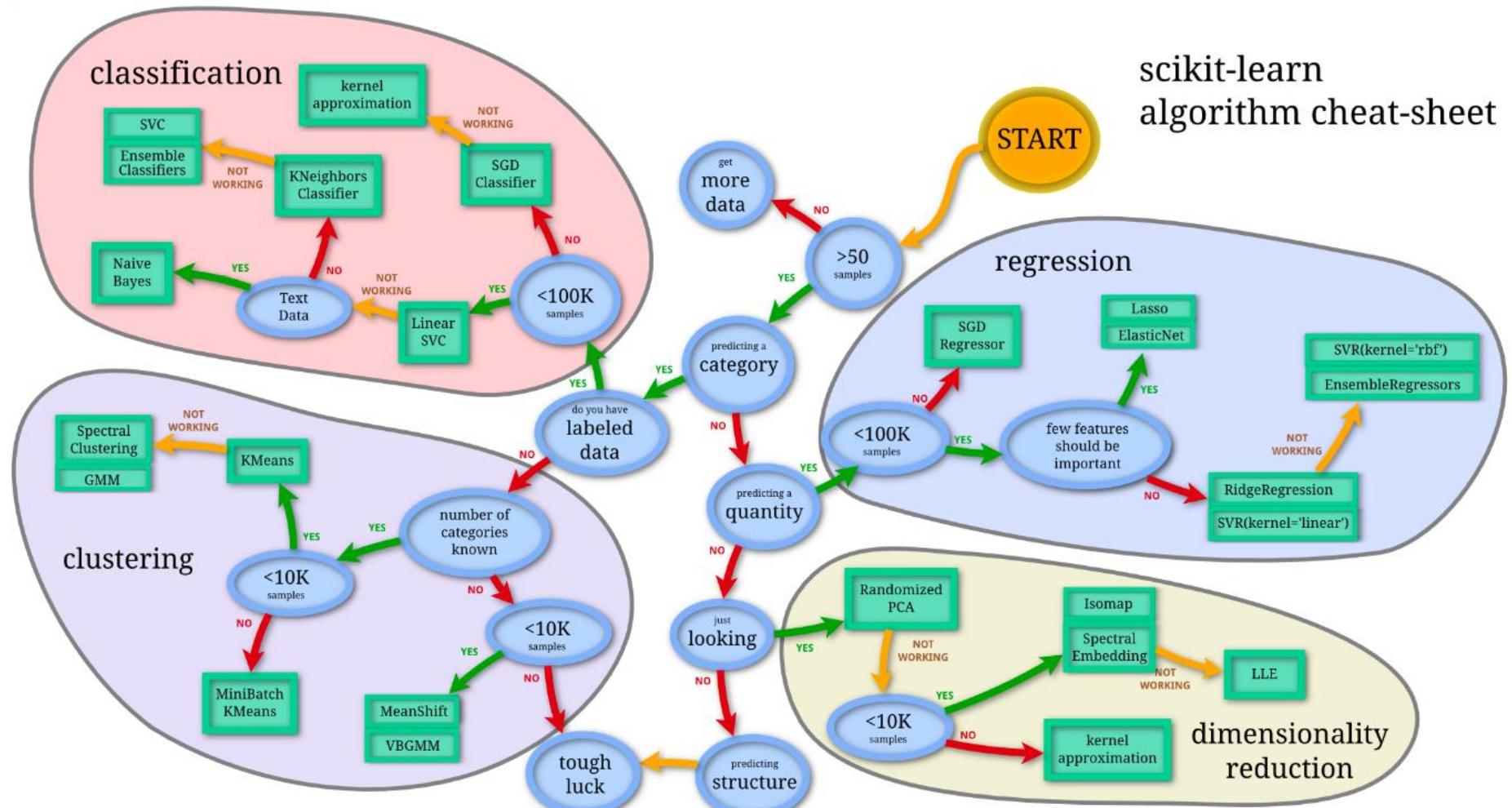
- Classification
- Regression
- Transcription
- Machine Translation
- Structured Output
- Anomaly Detection
- Synthesis and Sampling
- Imputation of Missing Values
- Denoising
- **Density Estimation**
- ...



General Approaches

	Generative	Discriminative
Supervised	<ul style="list-style-type: none">• Regression• Markov Random Fields• Bayesian Networks	<ul style="list-style-type: none">• Support Vector Machines• Conditional Random Fields• Decision Trees
Unsupervised	<ul style="list-style-type: none">• Neural Networks	<ul style="list-style-type: none">• Clustering Approaches

Little Cheat-Sheet



➤ http://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

Performance of a Method

- For classification, transcription, etc. we often measure the accuracy of the model.
 - Proportion of examples for which the model produces the correct output
- Equivalently, we can measure the error rate
 - Often referred to as the expected 0-1 loss
- Generally, there are hundreds of different performance measures
 - Often difficult to find a good performance measure
 - should we penalize the system more if it frequently makes medium-sized mistakes or if it rarely makes very large mistakes
 - The quantity we would ideally like to measure is impractical
 - ...

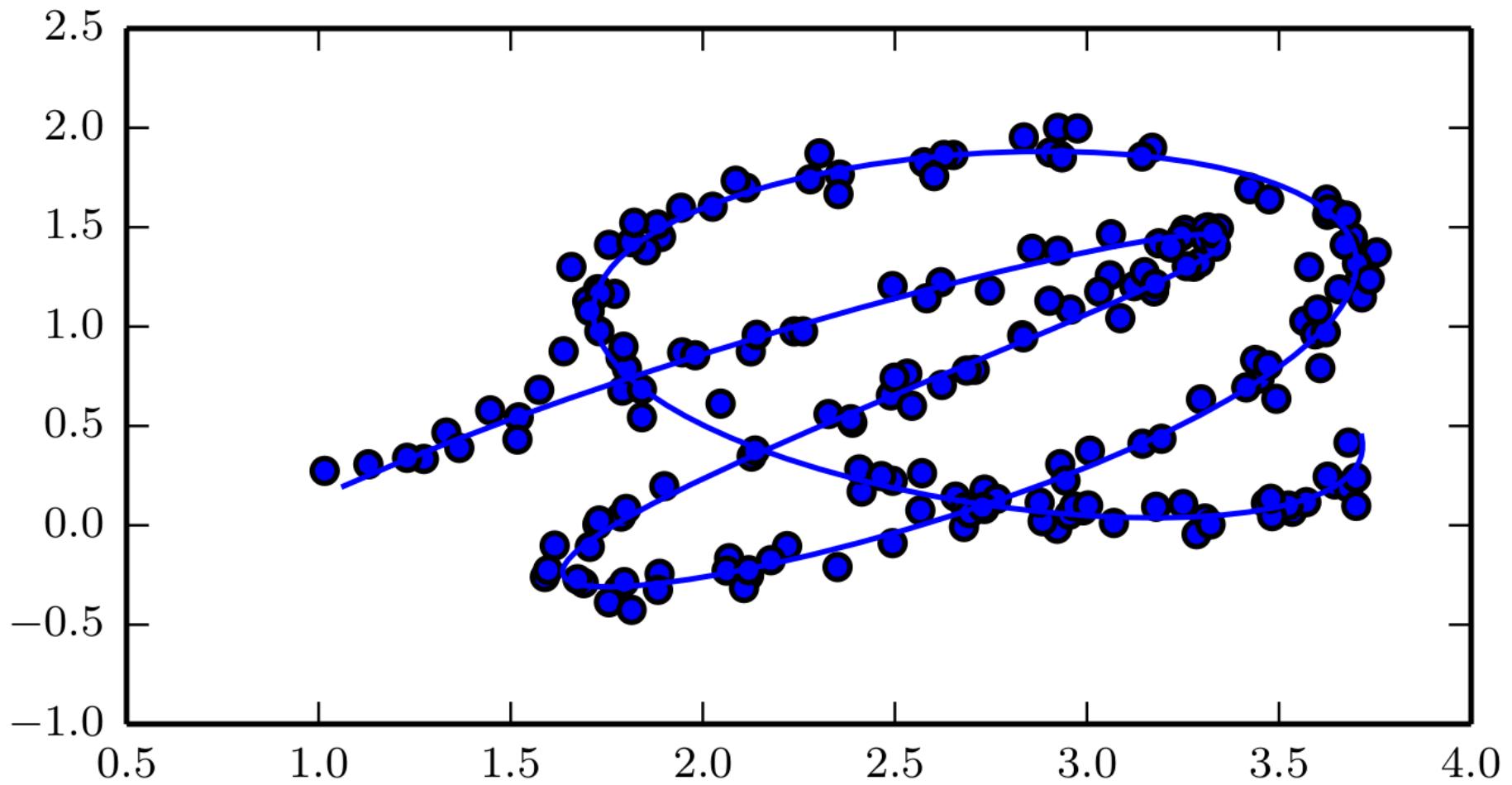
On Method Performance

- The No Free Lunch Theorem [Wolpert , 1996]

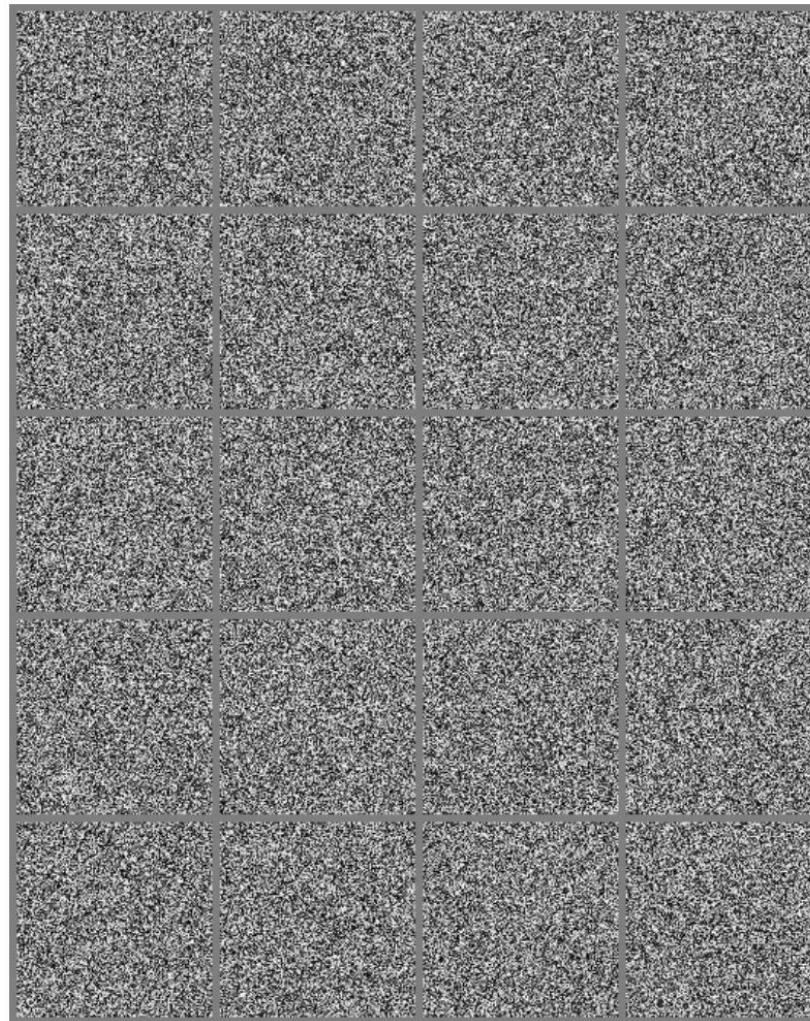
“averaged over all possible data generating distributions, every classification algorithm has the same error rate when classifying previously unobserved points”

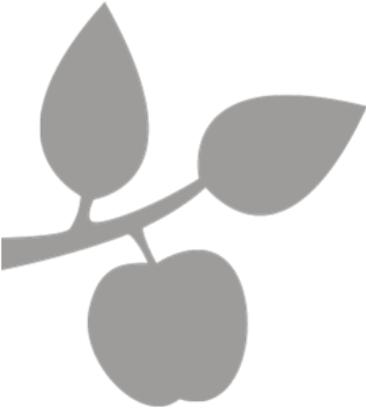
- In other words, no machine learning technique is per se better than another
- **BUT:** In reality, we do not encounter ALL possible distributions but only a few on which we need to perform good!

Manifold Learning



Uniformly Sampled Images





Machine Learning Basics

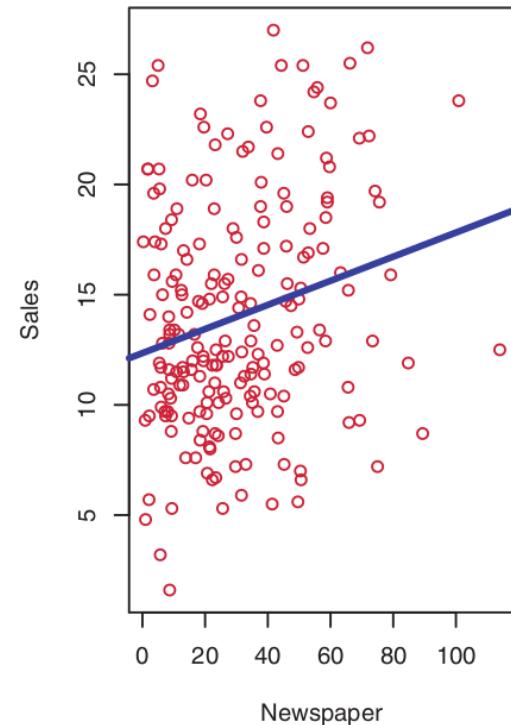
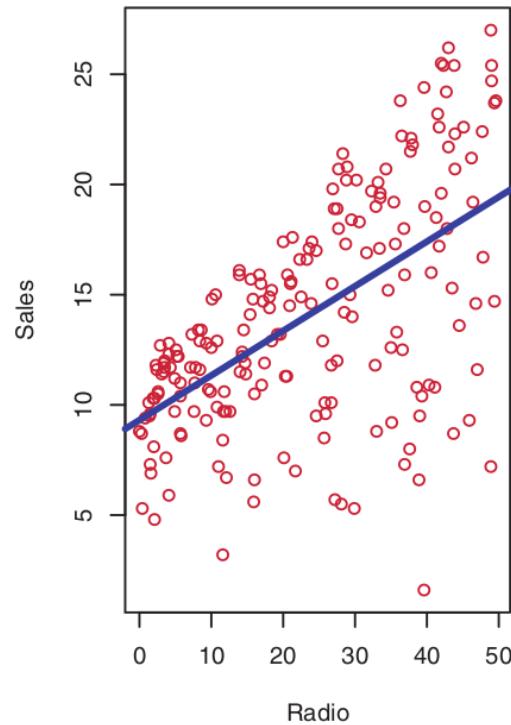
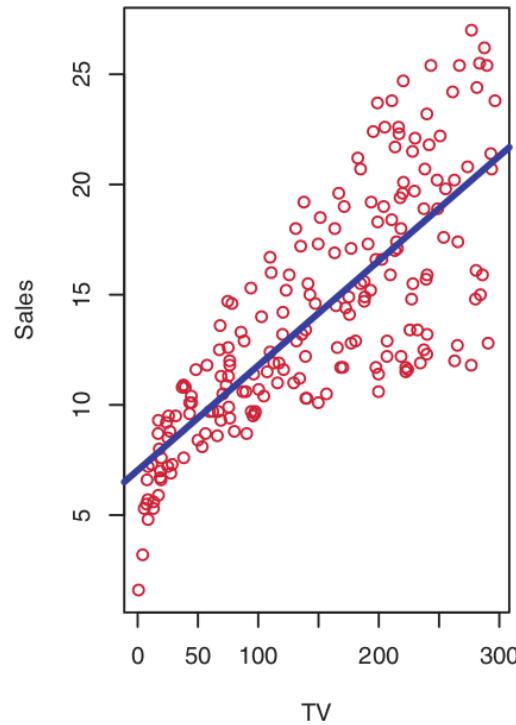
- **Introduction**
- **Statistical Learning**
- **Validation**

Wikipedia

Statistical learning theory is a framework for machine learning drawing from the fields of statistics and functional analysis.

Statistical learning theory deals with the problem of finding a predictive function based on data.

What is Statistical Learning?



- Shown are Sales vs. TV, Radio and Newspaper ads
- Can we predict Sales using these three?
- Perhaps we can do better using a model

Notation

- Here Sales is a response or target that we wish to predict. We generically refer to the response as Y .
- TV is a feature, or input, or predictor; we name it X_1 .
 - Likewise name Radio as X_2 , and so on.
- We can refer to the input vector collectively as

$$X = \begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix}$$

- Now we write our model as

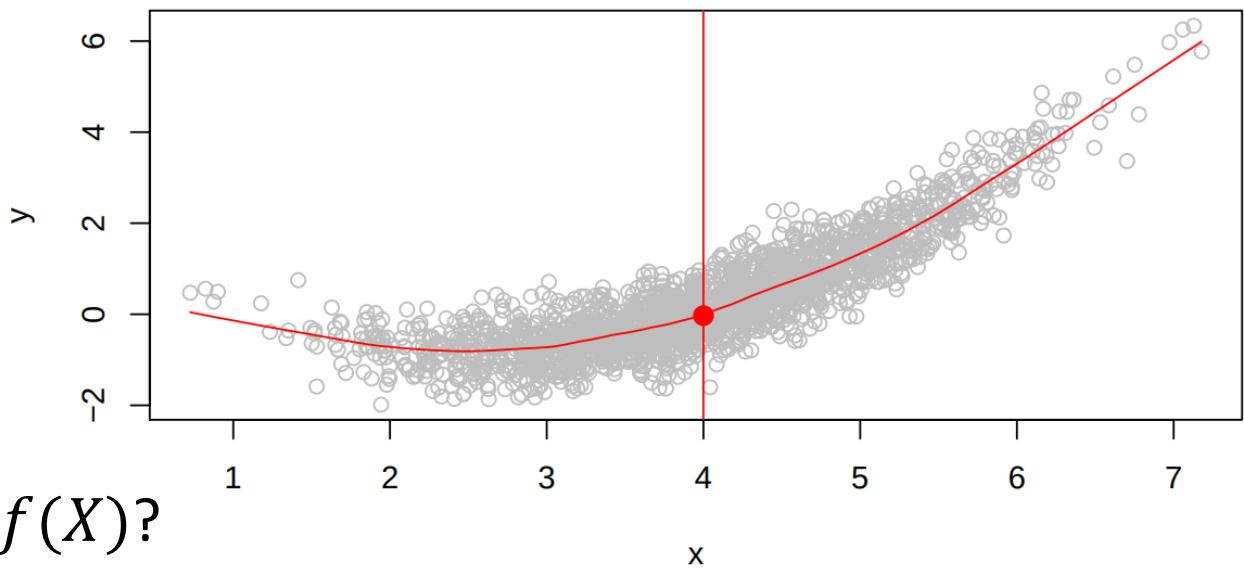
$$Y = f(X) + \epsilon$$

- where ϵ captures measurement errors and other discrepancies.

What is that good for?

- With a good f we can make predictions of Y at new points $X = x$.
- We can understand which components of $X = (X_1, X_2, \dots, X_p)$ are important in explaining Y , and which are irrelevant. e.g. Seniority and Years of Education have a big impact on Income, but Marital Status typically does not.
- Depending on the complexity of f , we may be able to understand how each component X_j of X affects Y .

The Regression Function



- Is there an optimal $f(X)$?
- What is the optimal value $f(X)$ at any selected value of X .
- For example: $X = 4$, then
$$f(4) = E[Y|X = 4]$$
- Ideally, we predict the **expected value** of Y for any given X
- This ideal $f(x) = E[Y|X = x]$ is called the **regression function**.

The Regression Function

- Is the ideal or optimal predictor of Y with regard to mean-squared prediction error: $f(x) = E(Y|X = x)$ is the function that minimizes $E[(Y - g(X))^2 | X = x]$ over all functions g at all points $X = x$.
- $\epsilon = Y - f(x)$ is the irreducible error — i.e. even if we knew $f(x)$, we would still make errors in prediction, since at each $X = x$ there is typically a distribution of possible Y values.
- For any estimate $\hat{f}(x)$ of $f(x)$, we have

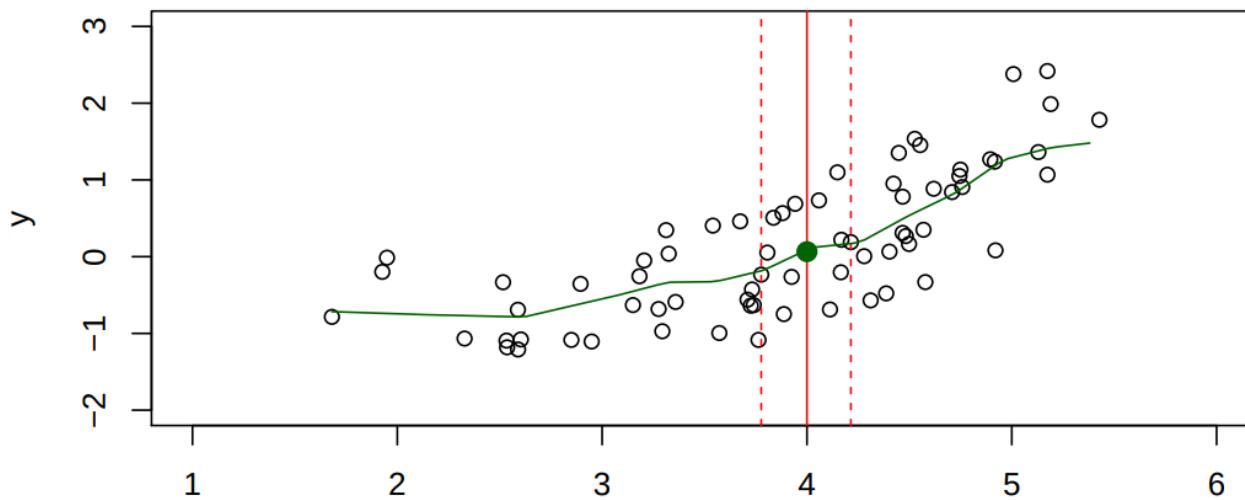
$$E[(Y - \hat{f}(X))^2 | X = x] = \underbrace{[f(x) - \hat{f}(x)]^2}_{\text{Reducible}} + \underbrace{\text{Var}(\epsilon)}_{\text{Irreducible}}$$

How to estimate f

- Typically we have few if any data points with $X = 4$ exactly.
 - So we cannot compute $E(Y|X = x)$!
- Relax the definition and let

$$\hat{f}(x) = \text{Ave}(Y|X \in \mathcal{N}(x))$$

where $\mathcal{N}(x)$ is some neighborhood of x .



How to estimate f

- Nearest neighbor averaging can be pretty good for small number of dimensions p , e.g. < 4 and large number of samples
- Nearest neighbor methods can be lousy when p is large.
 - Curse of dimensionality. Nearest neighbors tend to be far away in high dimensions.
 - We need to get a reasonable fraction of the N values of y_i to average to bring the variance down—e.g. 10%.
 - A 10% neighborhood in high dimensions need no longer be local, so we lose the spirit of estimating $E(Y|X = x)$ by local averaging.

Parametric and structured models

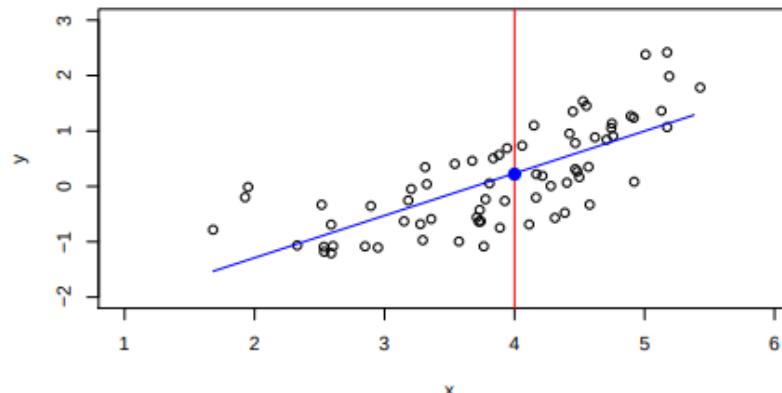
- The linear model is an important example of a parametric model:

$$f_L(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 \dots + \beta_p X_p$$

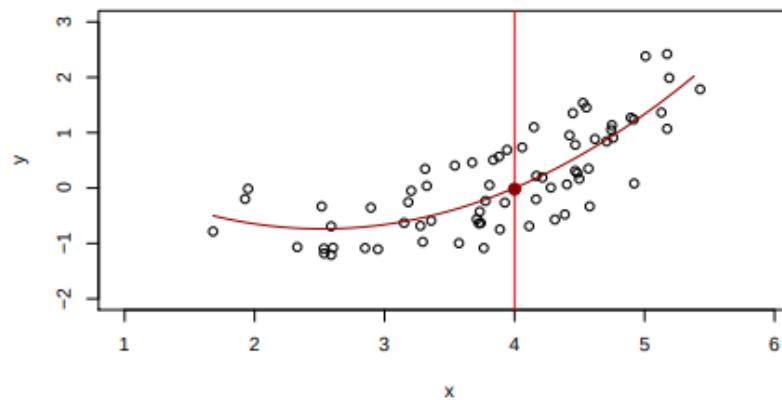
- A linear model is specified in terms of $p + 1$ parameters $\beta_0, \beta_1, \dots, \beta_p$.
- We estimate the parameters by fitting the model to training data.
- Although it is almost never correct, a linear model often serves as a good and interpretable approximation to the unknown true function $f(X)$.

Different Models

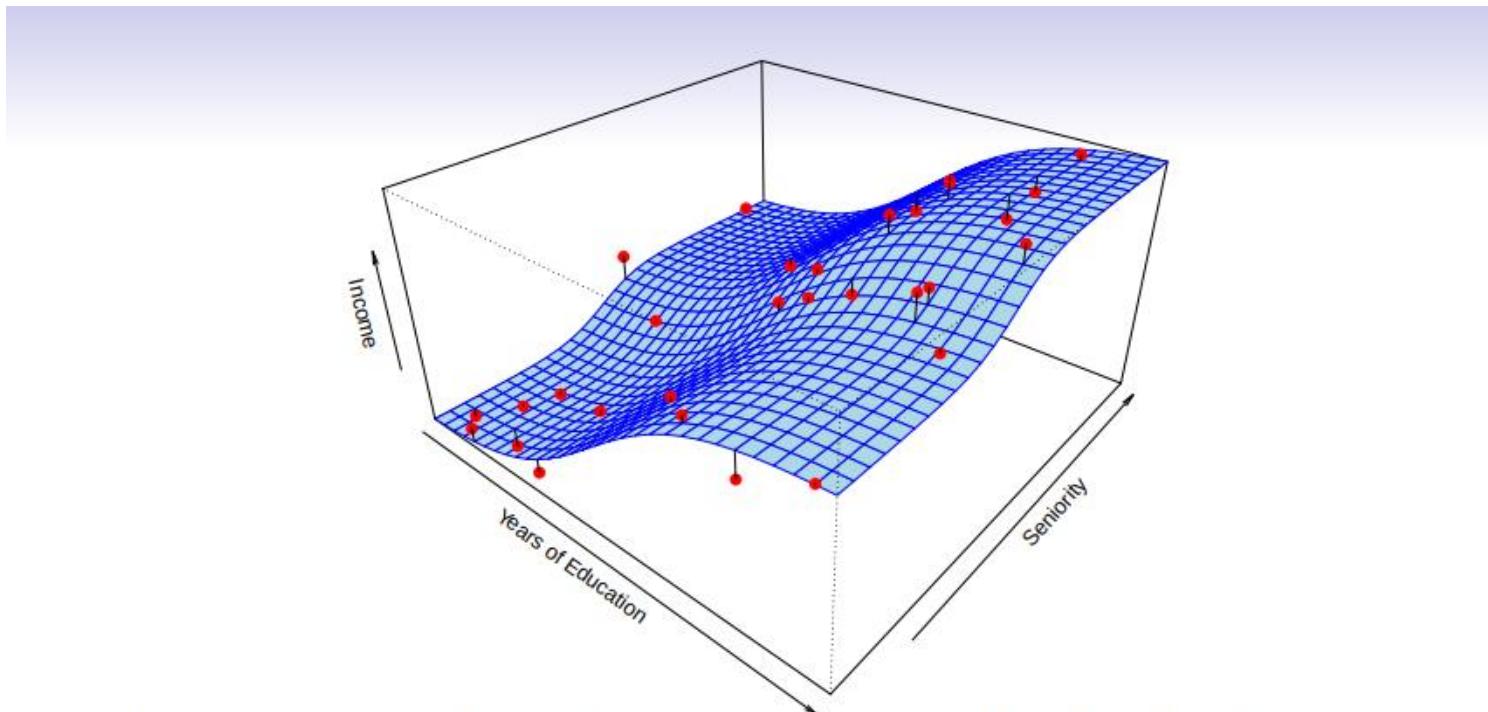
A linear model $\hat{f}_L(X) = \hat{\beta}_0 + \hat{\beta}_1 X$ gives a reasonable fit here



A quadratic model $\hat{f}_Q(X) = \hat{\beta}_0 + \hat{\beta}_1 X + \hat{\beta}_2 X^2$ fits slightly better.



Different Models

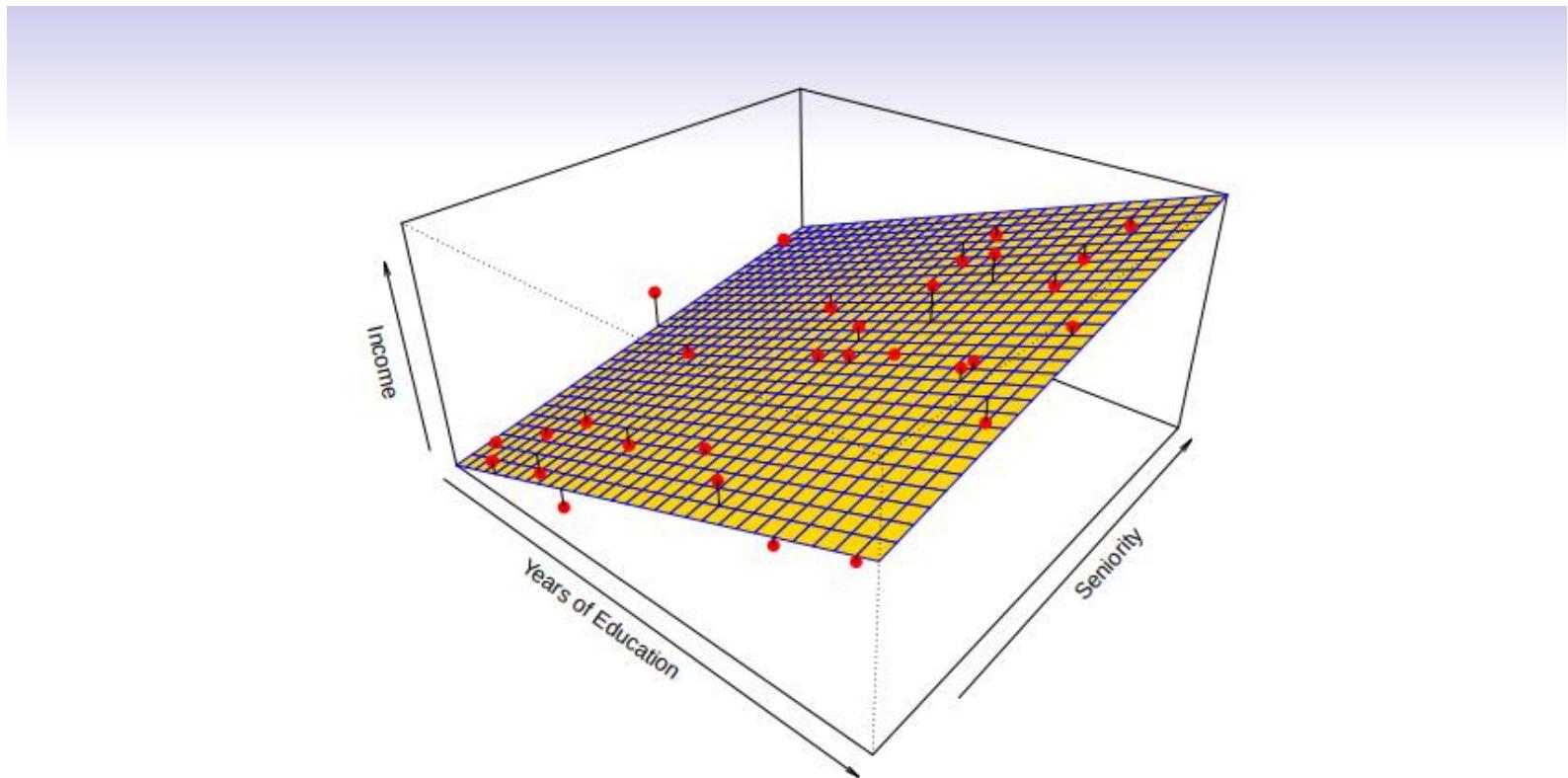


Simulated example. Red points are simulated values for `income` from the model

$$\text{income} = f(\text{education}, \text{seniority}) + \epsilon$$

f is the blue surface.

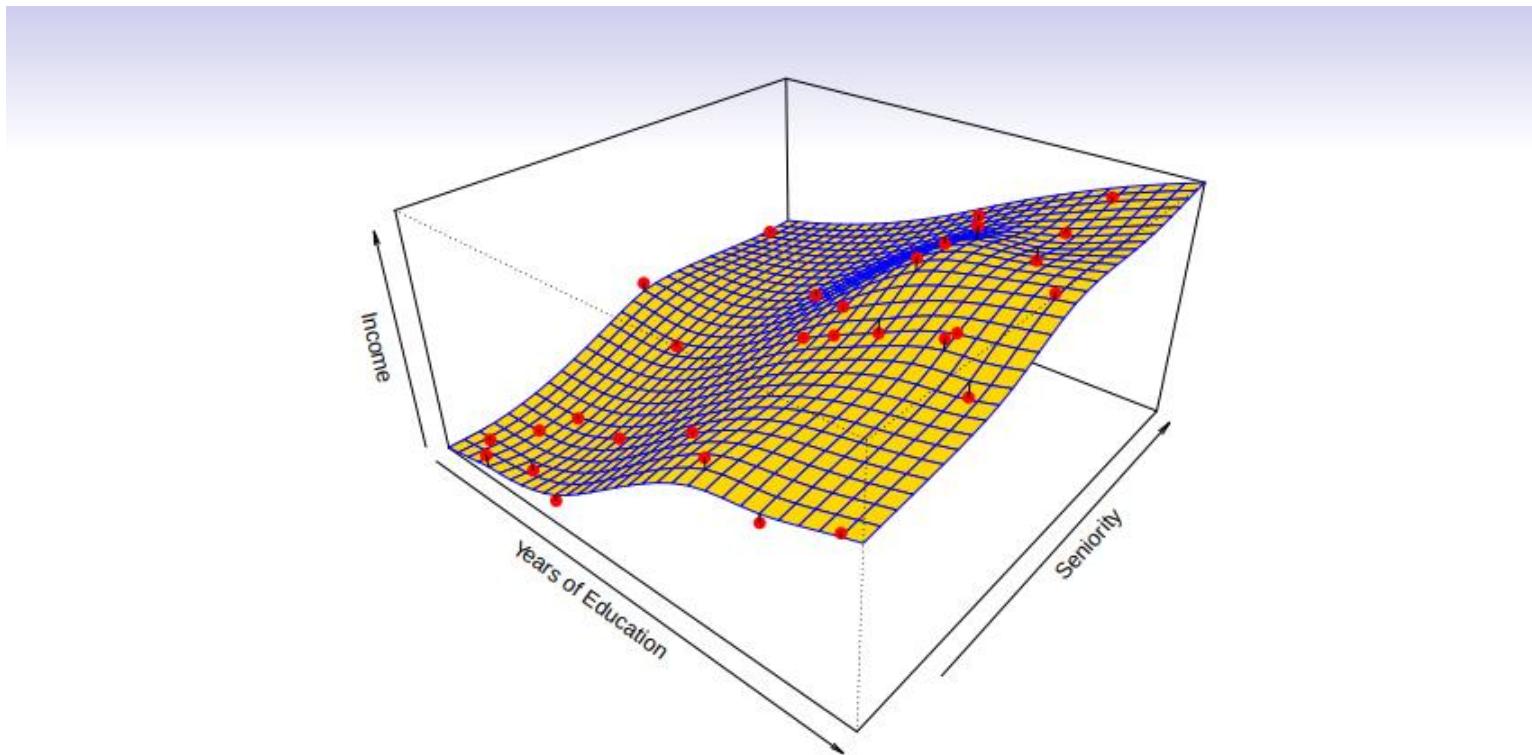
Model Complexity



Linear regression model fit to the simulated data.

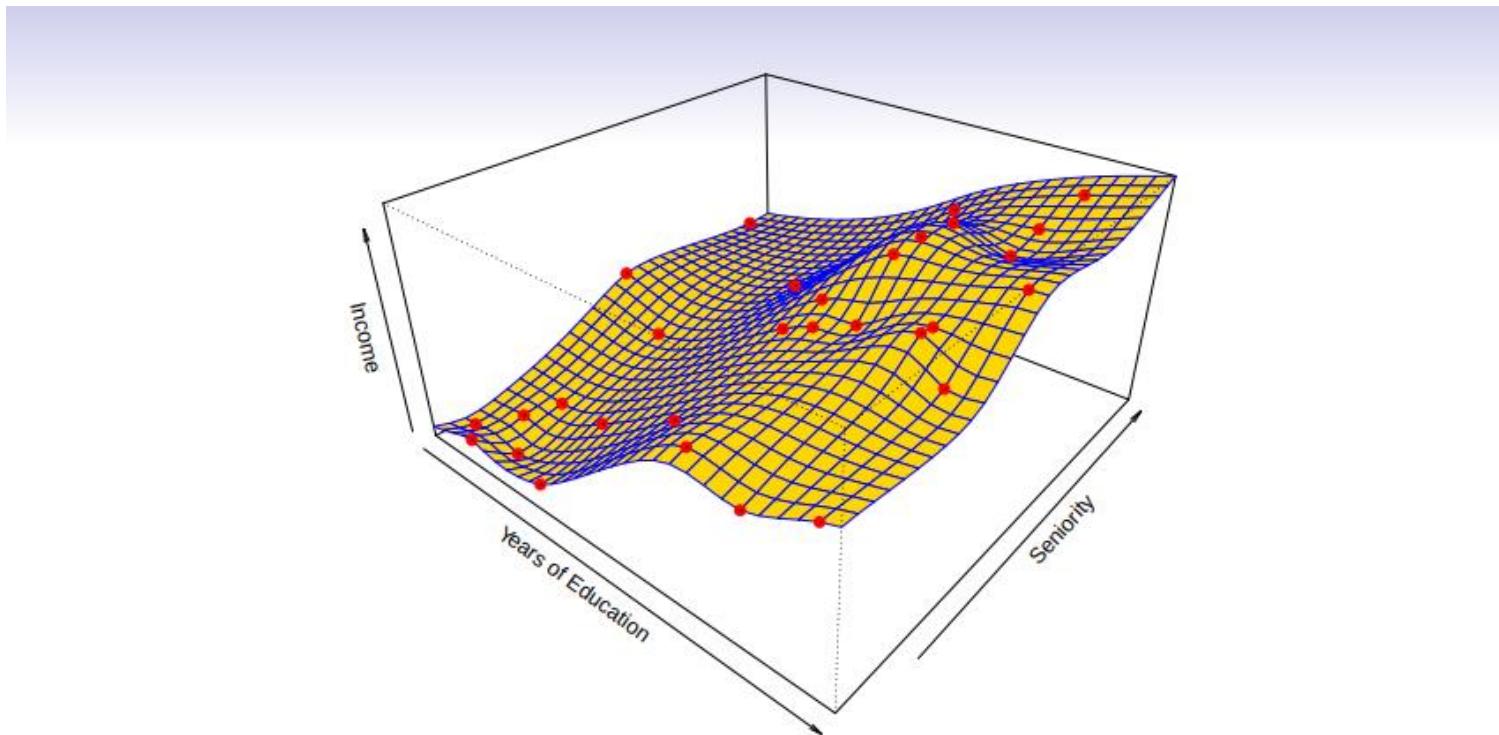
$$\hat{f}_L(\text{education}, \text{seniority}) = \hat{\beta}_0 + \hat{\beta}_1 \times \text{education} + \hat{\beta}_2 \times \text{seniority}$$

Model Complexity



More flexible regression model $\hat{f}_S(\text{education}, \text{seniority})$ fit to the simulated data. Here we use a technique called a *thin-plate spline* to fit a flexible surface.

Model Complexity



Even more flexible spline regression model
 $\hat{f}_S(\text{education, seniority})$ fit to the simulated data. Here the fitted model makes no errors on the training data! Also known as *overfitting*.

Model Complexity

Trade Offs:

- Prediction accuracy versus interpretability.
 - Linear models are easy to interpret; thin-plate splines are not. Deep Neural Networks are even harder to interpret
- Good fit versus over-fit or under-fit.
 - How do we know when the fit is just right?
- Parsimony versus black-box.
 - We often prefer a simpler model involving fewer variables over a black-box predictor involving them all.

Generalization

- Central challenge of ML is that the algorithm must perform well on new, **previously unseen** inputs
- Ability to perform well on previously unobserved inputs is called **generalization**
- Normally we have a test and training dataset
- Difference to normal optimization
 - We try to minimize the error on unseen data not on the training data, e.g. in the case of a linear regression: $\hat{y} = \mathbf{w}^T \mathbf{x} + b$

$$\frac{1}{m^{(\text{test})}} \left\| X^{(\text{test})} \mathbf{w} - y^{(\text{test})} \right\|_2^2$$

Estimating the generalization error

- To sum up with the example of linear regression:
- In linear regression example we train model by minimizing the training error

$$\frac{1}{m^{(\text{train})}} \|X^{(\text{train})}\mathbf{w} - \mathbf{y}^{(\text{train})}\|_2^2$$

- **BUT:** We are actually interested in the test error:

$$\frac{1}{m^{(\text{test})}} \|X^{(\text{test})}\mathbf{w} - \mathbf{y}^{(\text{test})}\|_2^2$$

- **How can we affect performance when we observe only the training set?**

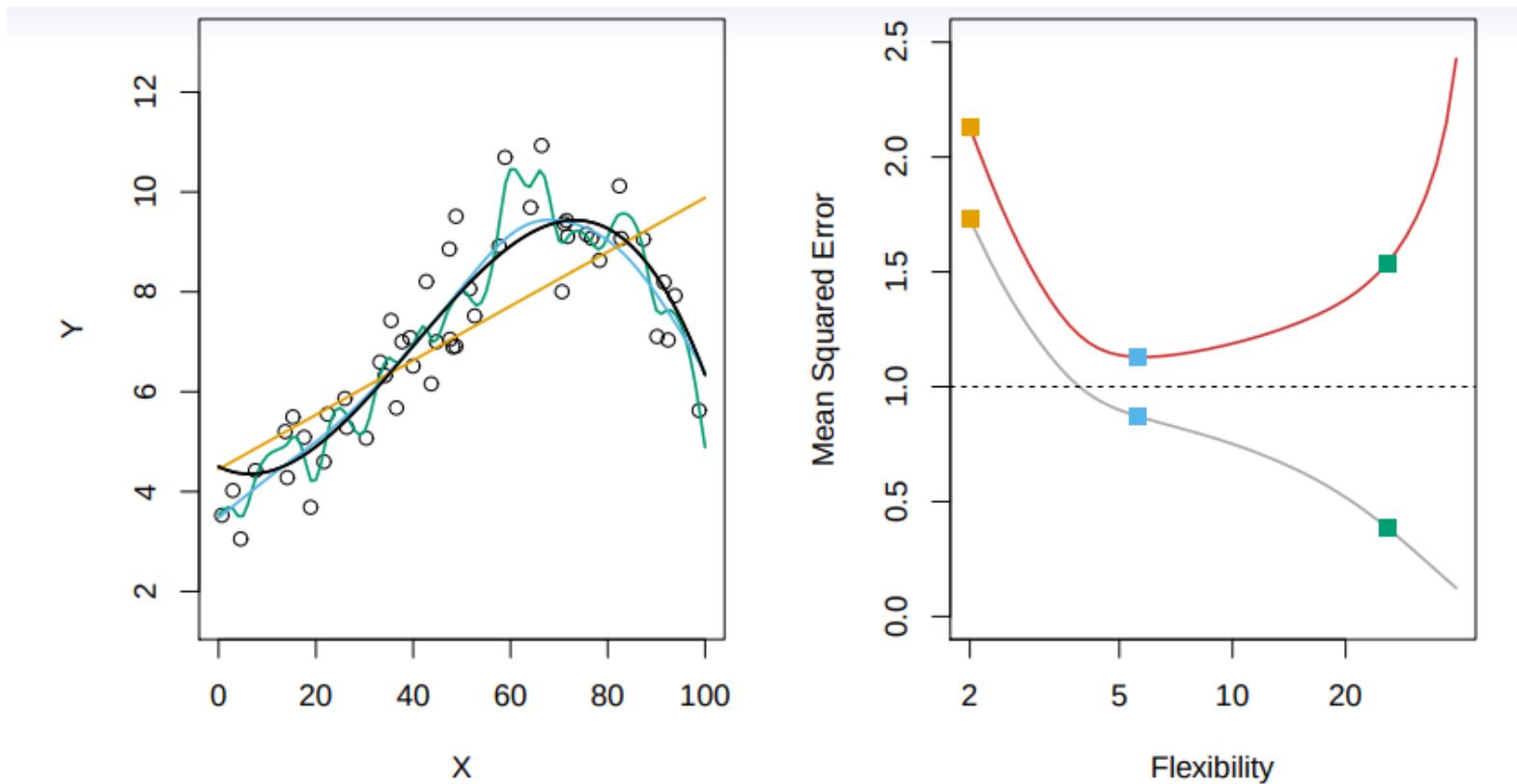
Under- and Over-fitting

- Factors determining how well an ML algorithm will perform are its ability to:
 1. Make the training error small
 2. Make gap between training and test errors small
- They correspond to two ML challenges
 - **Underfitting**
 - Inability to obtain low enough error rate on the training set
 - **Overfitting**
 - Gap between training error and testing error is too large
 - We can control whether a model is more likely to overfit or underfit by altering its capacity

Capacity of a model

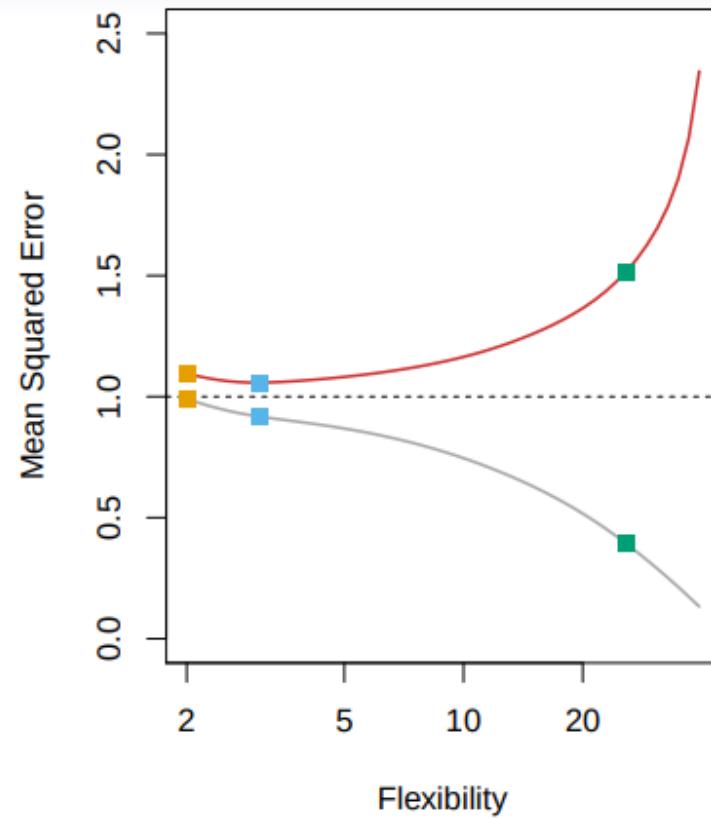
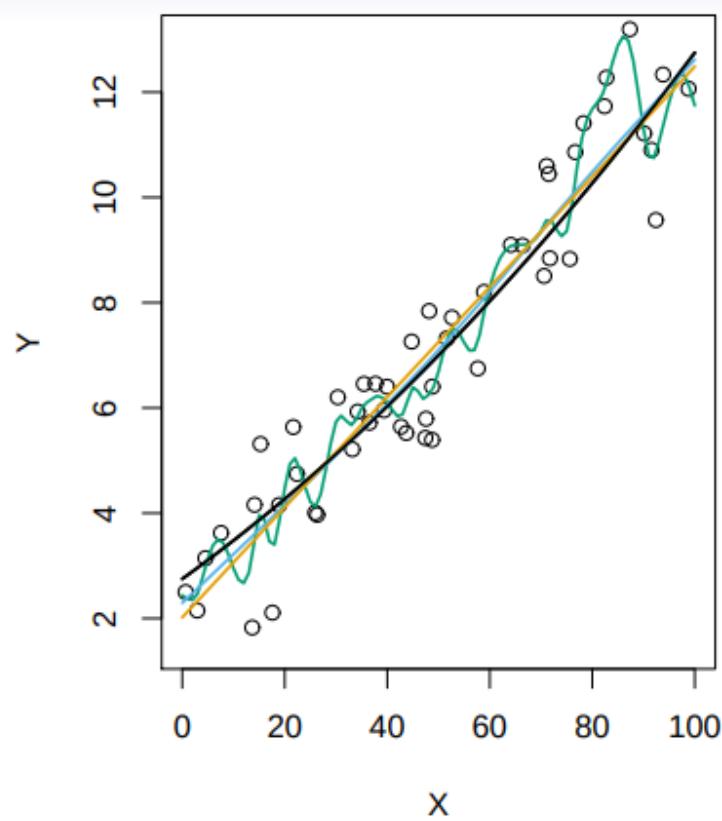
- Model capacity is ability to **fit variety of functions**
 - Model with Low capacity struggles to fit training set
 - A High capacity model can overfit by memorizing properties of training set which are not useful on test set
- When a model has higher capacity, it may overfit
 - One way to control capacity of a learning algorithm is by choosing the hypothesis space
 - i.e., set of functions that the learning algorithm is allowed to select as being the solution
 - e.g., the linear regression algorithm has the set of all linear functions of its input as the hypothesis space
 - **Regularization**

Assessing Model Accuracy



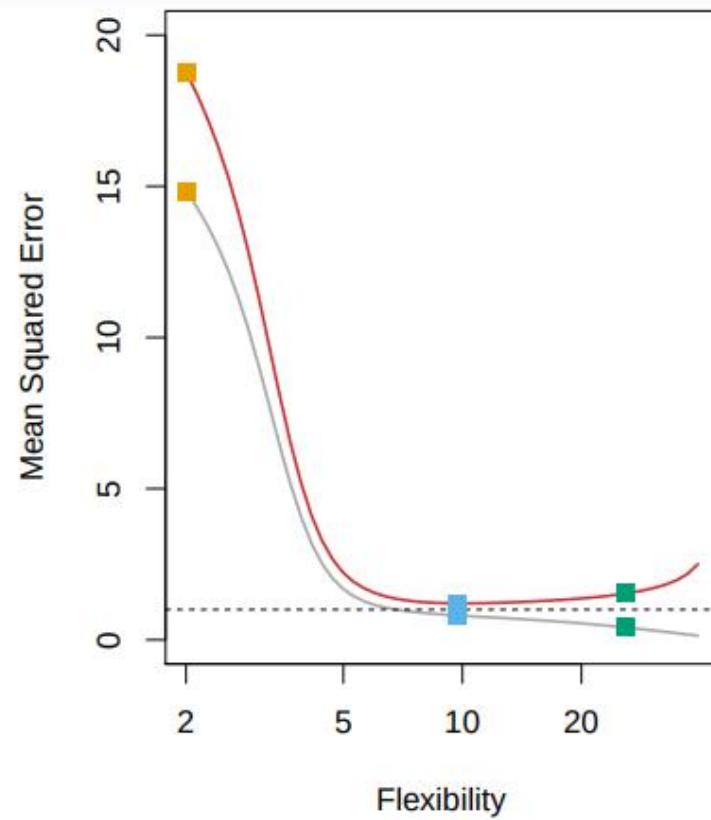
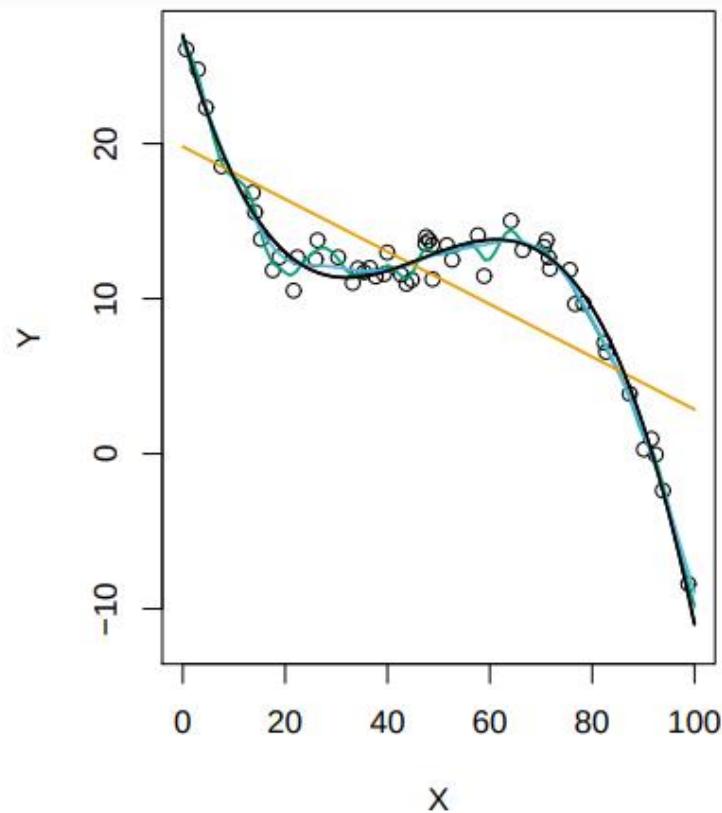
Black curve is truth. Red curve on right is MSE_{Te} , grey curve is MSE_{Tr} . Orange, blue and green curves/squares correspond to fits of different flexibility.

Assessing Model Accuracy



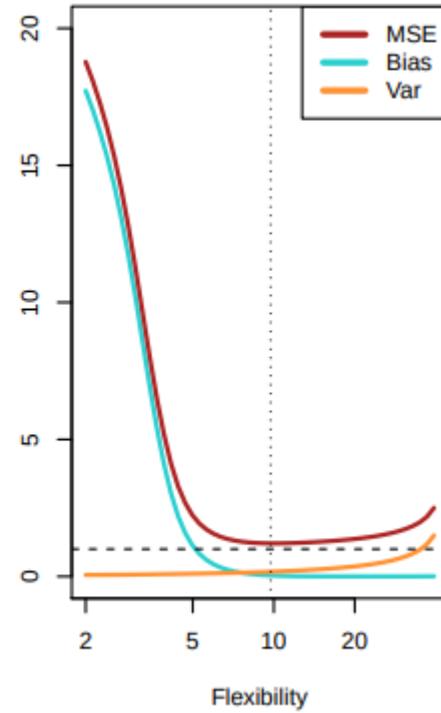
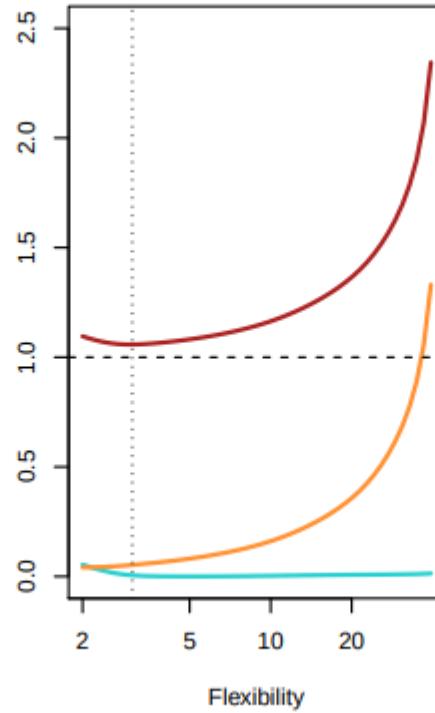
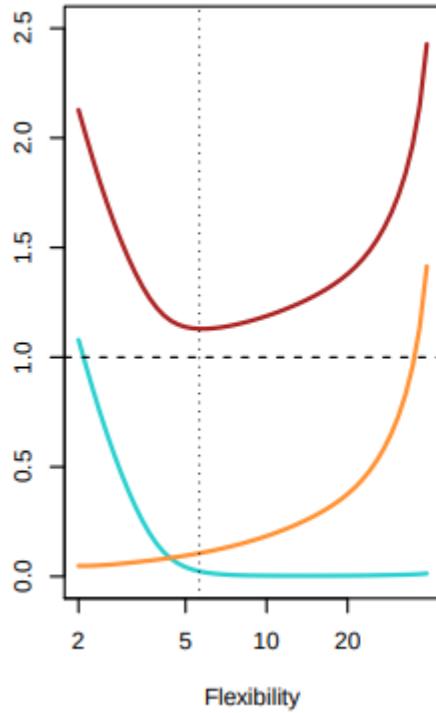
Here the truth is smoother, so the smoother fit and linear model do really well.

Assessing Model Accuracy



Here the truth is wiggly and the noise is low, so the more flexible fits do the best.

Bias Variance Trade-Off



Appropriate Capacity

- Machine Learning algorithms will perform well when their capacity is appropriate for the true complexity of the task that they need to perform and the amount of training data they are provided with
- Models with insufficient capacity are unable to solve complex tasks
- Models with high capacity can solve complex tasks, but when their capacity is higher than needed to solve the present task, they may overfit

Representational and Effective Capacity

Representational capacity:

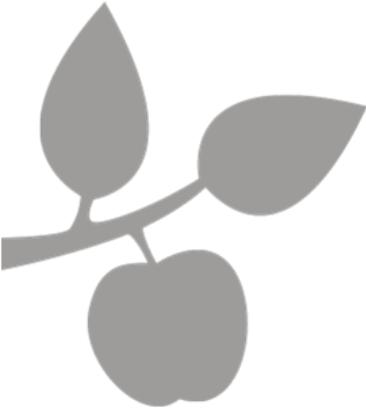
- Specifies family of functions learning algorithm can choose from

Effective capacity:

- Imperfections in optimization algorithm can limit representational capacity

Occam's razor:

- Among competing hypotheses that explain known observations equally well, choose the simplest one



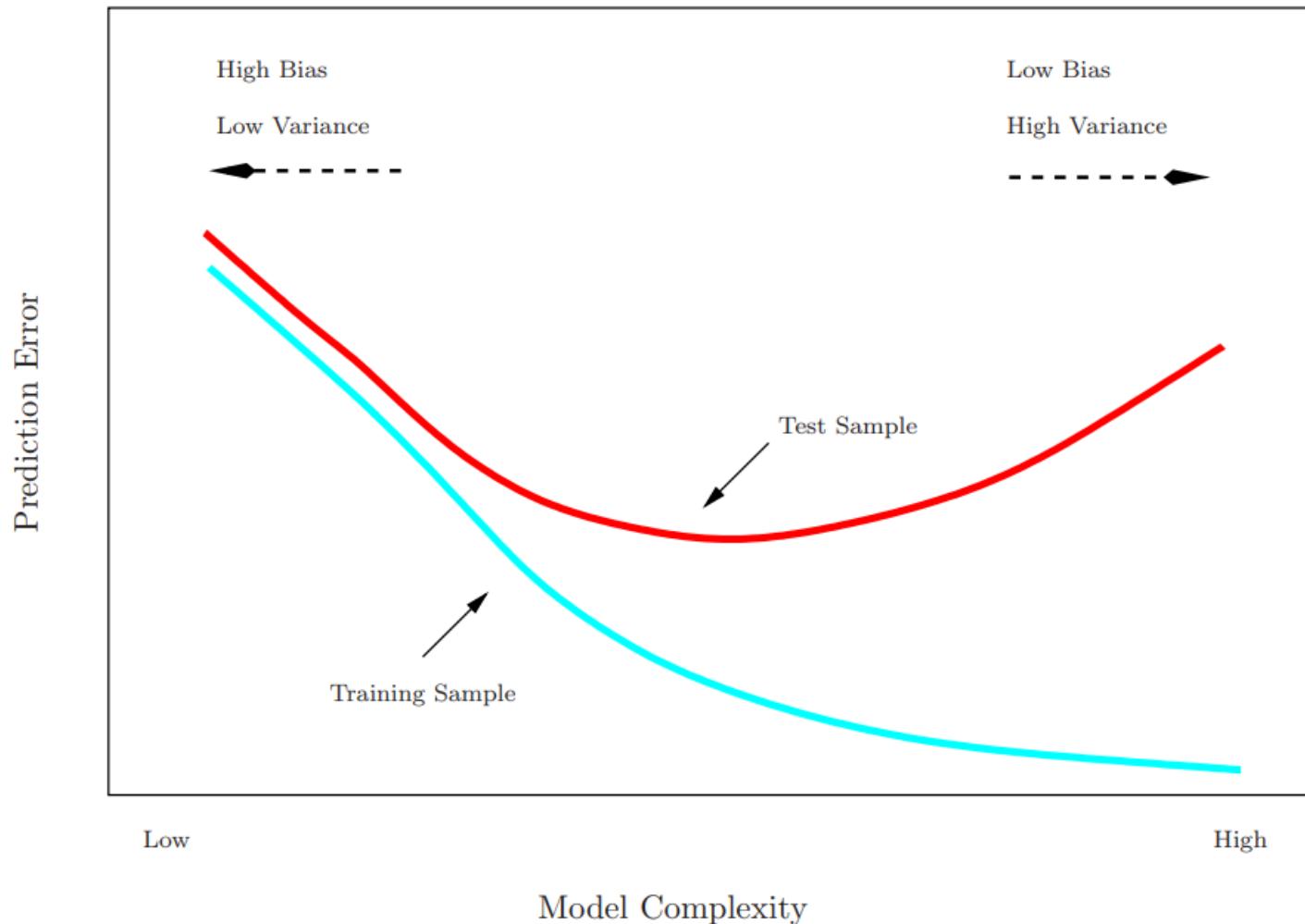
Machine Learning Basics

- **Introduction**
- **Statistical Learning**
- **Validation**

Training Error versus Test error

- Recall the distinction between the **test error** and the **training error**:
 - The test error is the average error that results from using a statistical learning method to predict the response on a new observation, one that was not used in training the method.
 - In contrast, the training error can be easily calculated by applying the statistical learning method to the observations used in its training.
 - **But the training error rate often is quite different from the test error rate, and in particular the former can dramatically underestimate the latter.**

Training- versus Test-Set Performance



Training- versus Test-Set Performance

- Best solution: a large designated test set.
 - Often not available
- Some methods make a mathematical adjustment to the training error rate in order to estimate the test error rate.
- Here we instead consider a class of methods that estimate the test error by holding out a subset of the training observations from the fitting process, and then applying the statistical learning method to those held out observations

Validation-set approach

- Here we randomly divide the available set of samples into two parts: a **training set** and a **validation** or **hold-out set**.
- The model is fit on the training set, and the fitted model is used to predict the responses for the observations in the validation set.
- The resulting validation-set error provides an estimate of the test error. This is typically assessed using MSE in the case of a quantitative response and misclassification rate in the case of a qualitative (discrete) response.

Validation-set approach



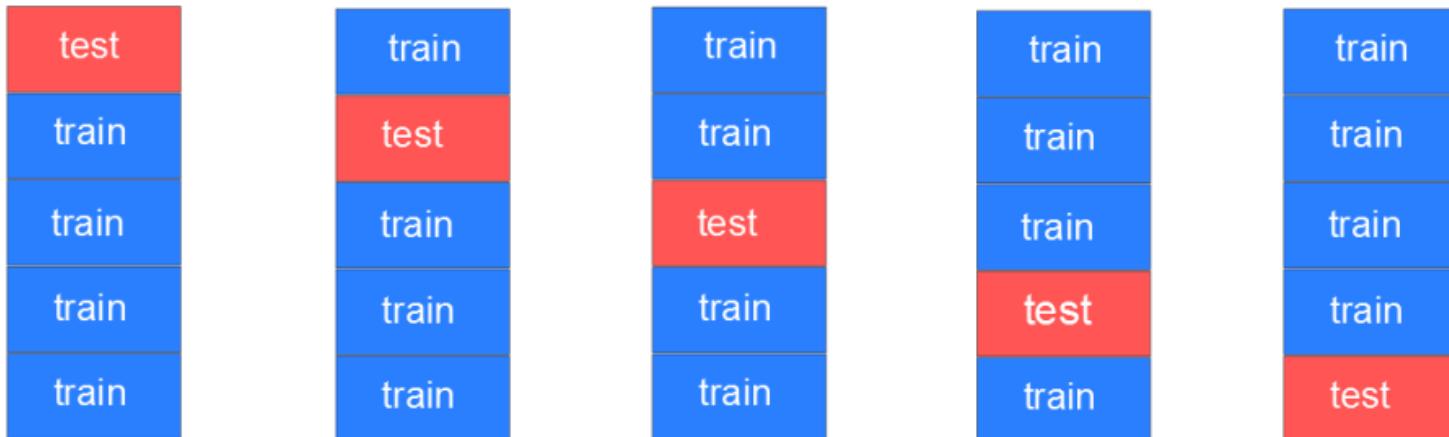
A random splitting into two halves: left part is training set,
right part is validation set

Drawbacks of Validation Set Approach

- The validation estimate of the test error can be highly variable, depending on precisely which observations are included in the training set and which observations are included in the validation set.
- In the validation approach, only a subset of the observations - those that are included in the training set rather than in the validation set - are used to fit the model.
- This suggests that the validation set error may tend to overestimate the test error for the model fit on the entire data set.

Cross-Validation

- When data set is too small, a fixed test set is problematic
- k -fold cross-validation:
 - All available data is partitioned into k groups
 - $k - 1$ groups are used to train and evaluated on remaining group
 - Repeat for all k choices of held-out group
 - Performance scores from k runs are averaged



The details

- Let the K parts be C_1, C_2, \dots, C_K , where C_k denotes the indices of the observations in part k . There are $n_k = n/K$ observations in part k .
- Compute

$$\text{CV}_{(K)} = \sum_{k=1}^K \frac{n_k}{n} \text{MSE}_k$$

- Setting $K = n$ yields n-fold or leave-one out cross-validation (LOOCV).
 - LOOCV sometimes useful, but typically doesn't shake up the data enough. The estimates from each fold are highly correlated and hence their average can have high variance.

Some Notes on Cross Validation

- Since each training set is only $(K - 1)/K$ as big as the original training set, the estimates of prediction error will typically be biased upward.
- This bias is minimized when $K = n$ (LOOCV), but this estimate has high variance, as just seen.
- $K = 5$ or 10 provides a good compromise for this bias-variance tradeoff.

Evaluate the Classifier Performance

- For classification, we need different measures than MSE
- Most simple measure: **Accuracy / Error Rate**
 - Accuracy: Percentage of correct classifications
 - Error Rate: Percentage of incorrect classifications
- Higher accuracy does not necessarily imply better performance on target task!
 - Implicit assumption: the class distribution among examples is relatively balanced
 - Biased in favor of the majority class!
 - Should be used with caution!

Confusion matrix, two classes only

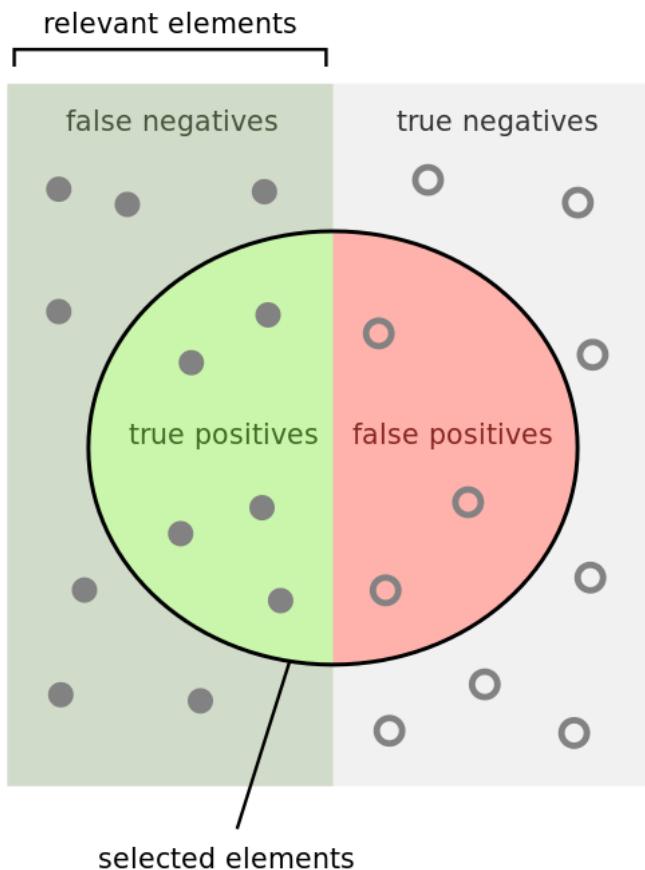
Performance measures calculated from the confusion matrix entries:

- ◆ Accuracy = $(a + d)/(a + b + c + d) = (TN + TP)/total$
- ◆ **True positive rate**, recall, sensitivity = $d/(c + d) = TP/actual\ positive$
- ◆ Specificity, true negative rate = $a/(a + b) = TN/actual\ negative$
- ◆ Precision, predicted positive value = $d/(b + d) = TP/predicted\ positive$
- ◆ **False positive rate**, false alarm = $b/(a + b) = FP/actual\ negative = 1 - specificity$
- ◆ False negative rate = $c/(c + d) = FN/actual\ positive$

		predicted	
		negative	positive
actual examples	negative	a TN - True Negative correct rejections	b FP - False Positive false alarms type I error
	positive	c FN - False Negative misses, type II error overlooked danger	d TP - True Positive hits

F-Measure

- Harmonic mean between Precision (Prec) and Recall (Rec):



$$F = 2 \cdot \frac{\text{Prec} \cdot \text{Rec}}{\text{Prec} + \text{Rec}}$$

How many selected items are relevant?

$$\text{Precision} = \frac{\text{green}}{\text{green} + \text{red}}$$

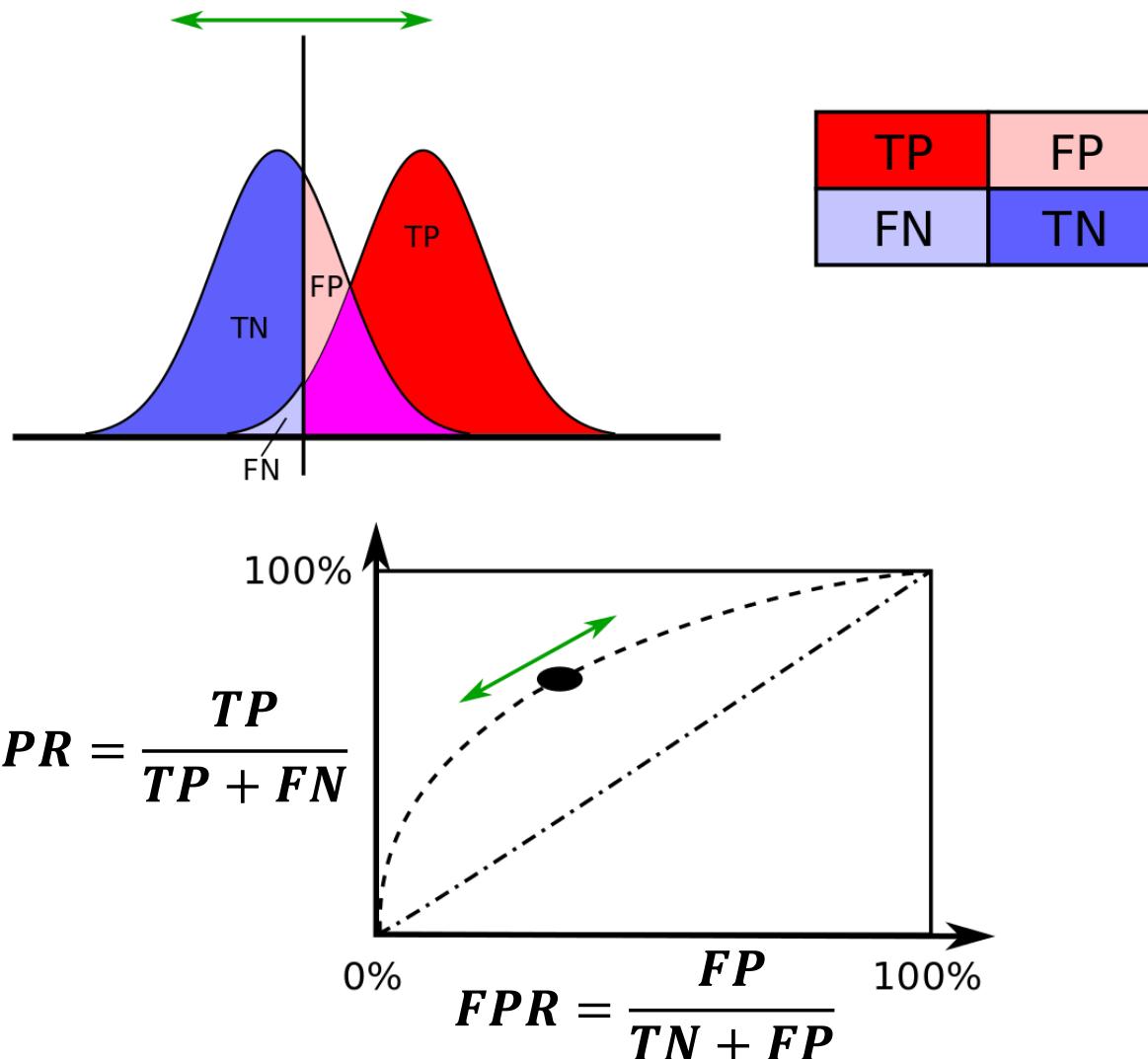
How many relevant items are selected?

$$\text{Recall} = \frac{\text{green}}{\text{green} + \text{blue}}$$

Roc Curve

- Receiver Operating Characteristic
 - The ROC curve was first used during World War II for the analysis of radar signals before it was employed in signal detection theory ... for these purposes they measured the ability of a radar receiver operator to make these important distinctions, which was called the Receiver Operating Characteristic.”
- ROC curve is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied.
- For our Regression, we simply vary the threshold t and
 - Predict $y = 1$ if $p(y|x, \theta) \geq t$
 - Predict $y = 0$ if $p(y|x, \theta) < t$

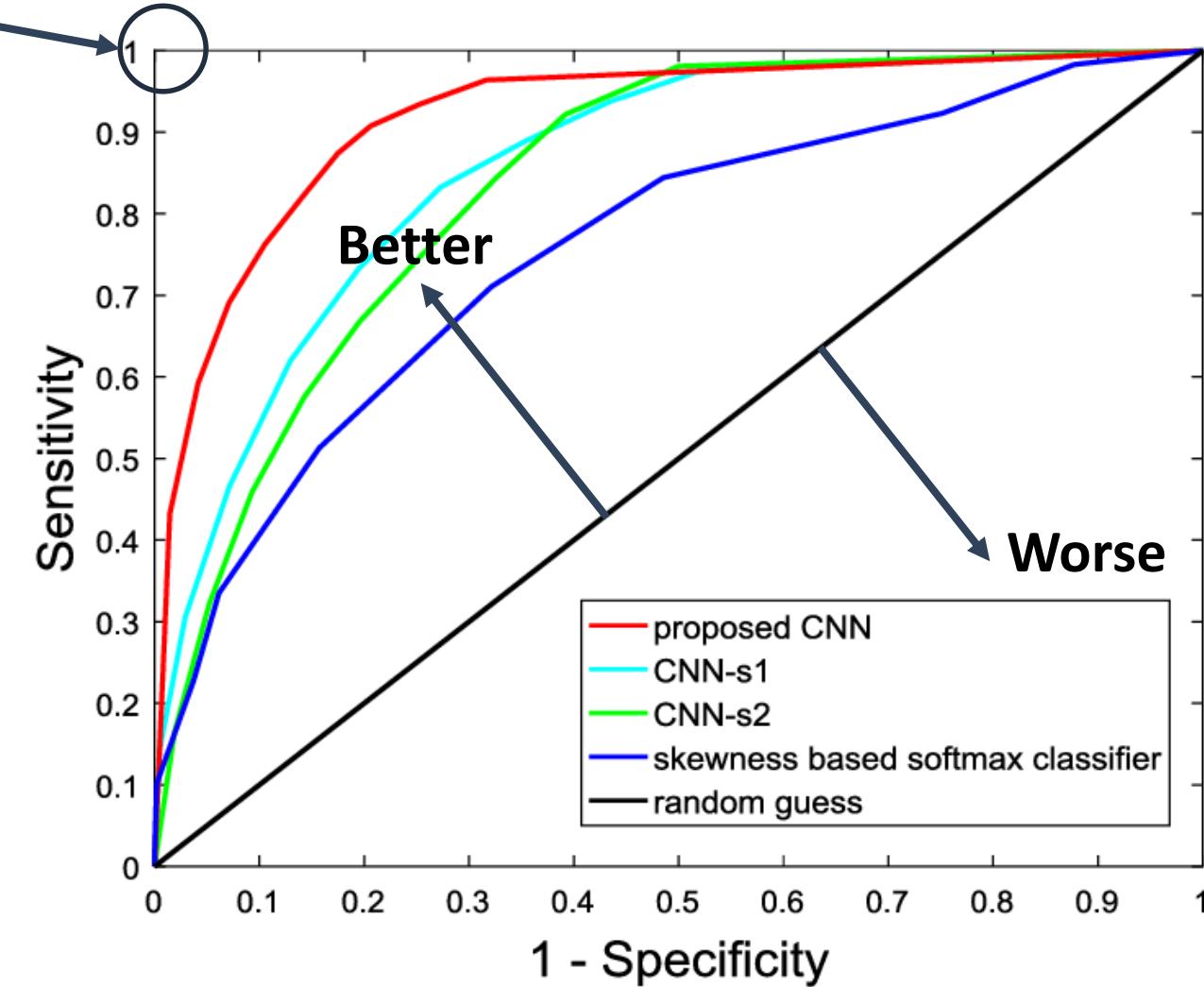
ROC Curve for Changing Threshold



➤ Source: wikipedia

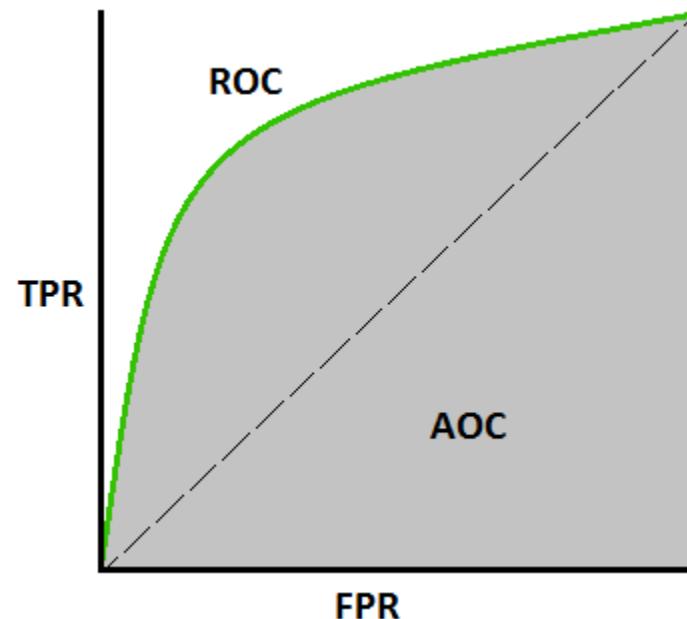
Interpreting a ROC Curve

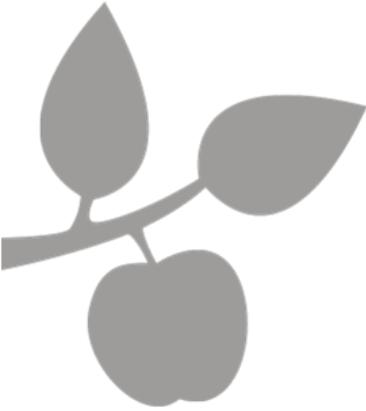
Perfect Classifier



AUC

- To summarize a ROC Curve, often the AUC is used
- AUC: Area under the curve
 - Perfect Classifier: AUC = 1
 - Random Classifier: AUC = 0.5





DM873

Deep Learning

Spring 2019

Lecture 4 – Regression

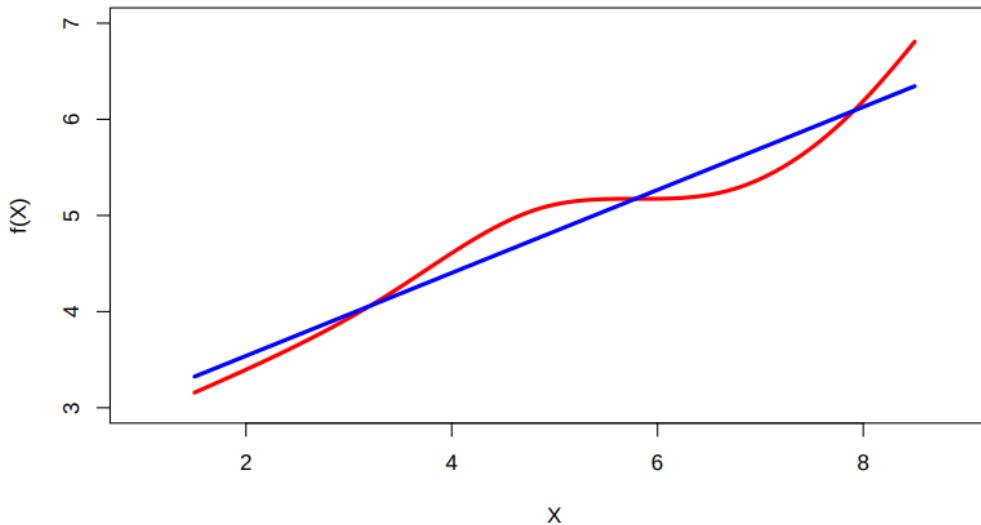


Regression

- **Linear Regression**
- **Logistic Regression**

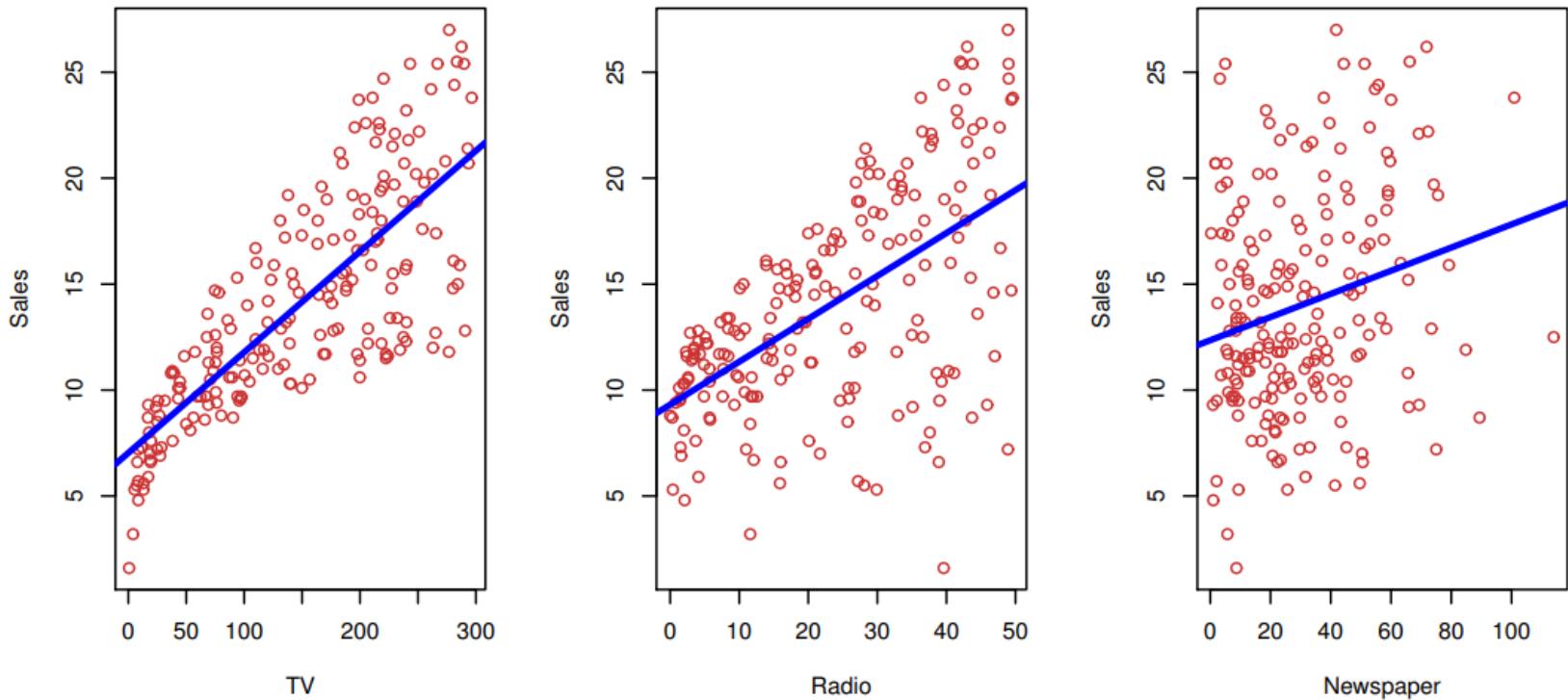
Linear Regression

- Linear regression is a simple approach to supervised learning. It assumes that the dependence of Y on X_1, X_2, \dots, X_p is linear.
- True regression functions are never linear!



- Although it may seem overly simplistic, linear regression is extremely useful both conceptually and practically.

Revisit: Advertising data



- The Advertising data set consists of the sales of that product in 200 different markets, along with advertising budgets for the product in each of those markets for three different media: TV, radio, and newspaper.

Revisit: Advertising data

Questions we might ask:

- Is there a relationship between advertising budget and sales?
- How strong is the relationship between advertising budget and sales?
- Which media contribute to sales?
- How accurately can we predict future sales?
- Is the relationship linear?
- Is there synergy among the advertising media?

Simple Linear Regression.

- We assume a model

$$Y = \beta_0 + \beta_1 X + \epsilon$$

- where β_0 and β_1 are two unknown constants that represent the intercept and slope, also known as coefficients or parameters, and ϵ is the error term.
- Given some estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ model coefficients, we **predict** future sales using

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x$$

Estimation of the Parameters by Least Squares

- Let $\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$ be the prediction of Y for the i th observation.
- Then the **residual** is defined as

$$e_i = y_i - \hat{y}_i$$

- We define the **residual sum of squares** as

$$\begin{aligned} \text{RSS} &= e_1^2 + e_2^2 + \cdots + e_n^2 = \\ (y_1 - \hat{\beta}_0 - \hat{\beta}_1 x_1)^2 &+ (y_2 - \hat{\beta}_0 - \hat{\beta}_1 x_2)^2 + \cdots + (y_n - \hat{\beta}_0 - \hat{\beta}_1 x_n)^2 \end{aligned}$$

- Our task is to find the $\hat{\beta}_0$ and $\hat{\beta}_1$ minimizing the RSS.

Estimation of the Parameters by Least Squares

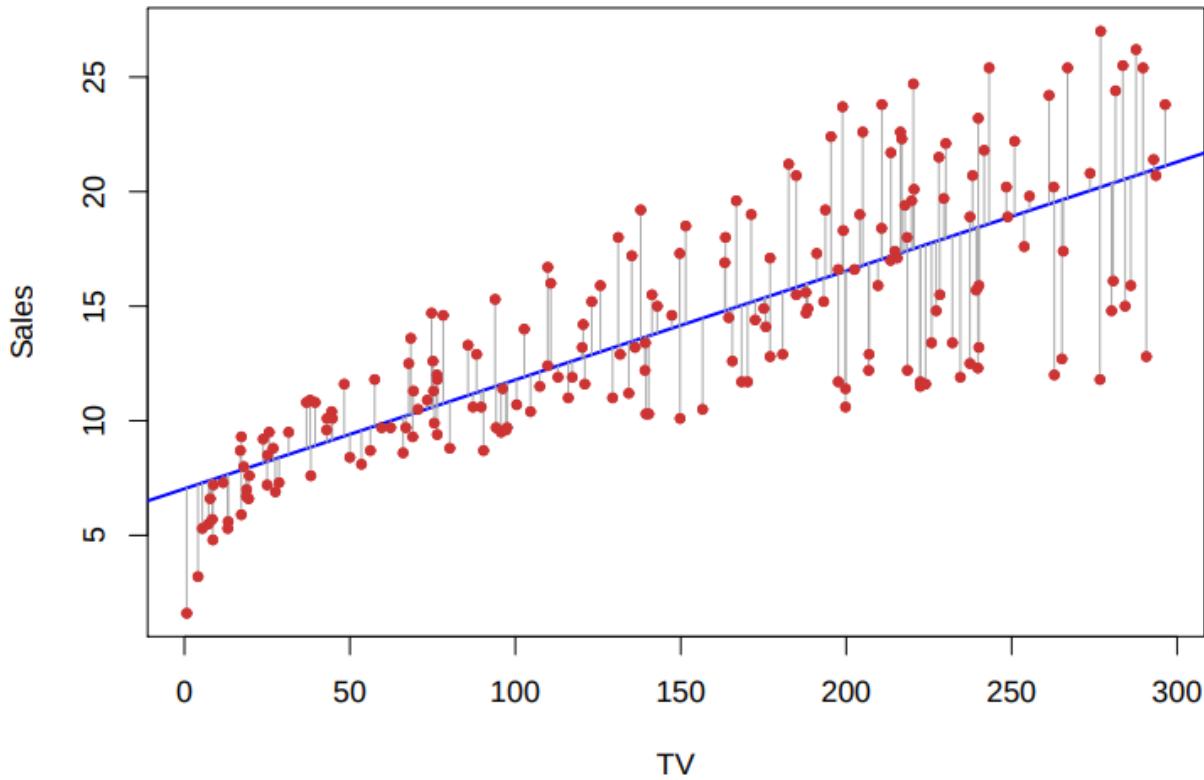
- With simple derivation of the RSS formula, we can show that

$$\hat{\beta}_1 = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

- with the sample means $\bar{y} = \frac{1}{n} \sum_{i=1}^n y_i$ and $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$

Example: Advertising data



- The least squares fit for the regression of sales onto TV.
- In this case a linear fit captures the essence of the relationship, although it is somewhat deficient in the left of the plot.

Assessing the Accuracy of the Coefficient Estimates

- The standard error of an estimator reflects how it varies under repeated sampling.
- For our example, we have ($\sigma = \text{Var}(\epsilon)$):

$$\text{SE}(\hat{\beta}_0)^2 = \sigma^2 \left[\frac{1}{n} + \frac{\bar{x}^2}{\sum_{i=1}^n (x_i - \bar{x})^2} \right]$$

$$\text{SE}(\hat{\beta}_1)^2 = \frac{\sigma^2}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

Hypothesis testing

- Standard errors can also be used to perform hypothesis tests on the coefficients.
- The most common hypothesis test involves testing the **null hypothesis** of:

H_0 : There is no relationship between X and Y

$$H_0: \beta_1 = 0$$

versus the alternative hypothesis

H_A : There is some relationship between X and Y

$$H_0: \beta_1 \neq 0$$

Hypothesis testing

- To test the null hypothesis, we compute a **t-statistic**, given by

$$t = \frac{\hat{\beta}_1 - 0}{\text{SE}(\hat{\beta}_1)}$$

- This will have a t -distribution with $n - 2$ degrees of freedom, assuming $\beta_1 = 0$.
- Using statistical software, it is easy to compute the probability of observing any value equal to $|t|$ or larger. We call this probability the **p-value**.

Assessing the Overall Accuracy

- We can compute the **Residual Sum of Squares (RSS)**

$$\text{RSS} = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

- With that, we can define the **Residual Standard Error (RSE)**

$$\text{RSE} = \sqrt{\frac{1}{n-2} \text{RSS}}$$

Assessing the Overall Accuracy

- To assess how much of the variance is explained, we use the **R-squared** measure

$$R^2 = \frac{TSS - RSS}{TSS} = 1 - \frac{RSS}{TSS}$$

where $TSS = \sum_{i=1}^n (y_i - \bar{y})^2$ is the **total sum of squares**.

- It can be shown that in this simple linear regression setting that $R^2 = r^2$ with r being the **correlation** between X and Y :

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

Multiple Linear Regression

- We can include several features or predictors into our model

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \epsilon$$

- We interpret β_j as the average effect on Y of a one unit increase in X_j , holding all other predictors fixed. In the advertising example, the model becomes

$$\text{sales} = \beta_0 + \beta_1 \cdot \text{TV} + \beta_2 \cdot \text{radio} + \beta_3 \cdot \text{newspaper} + \epsilon$$

In Matrix Form

- We can formulate the entire linear regression also in Matrixform

$$\mathbf{y} = \mathbf{X}\mathbf{b} + \mathbf{e}$$

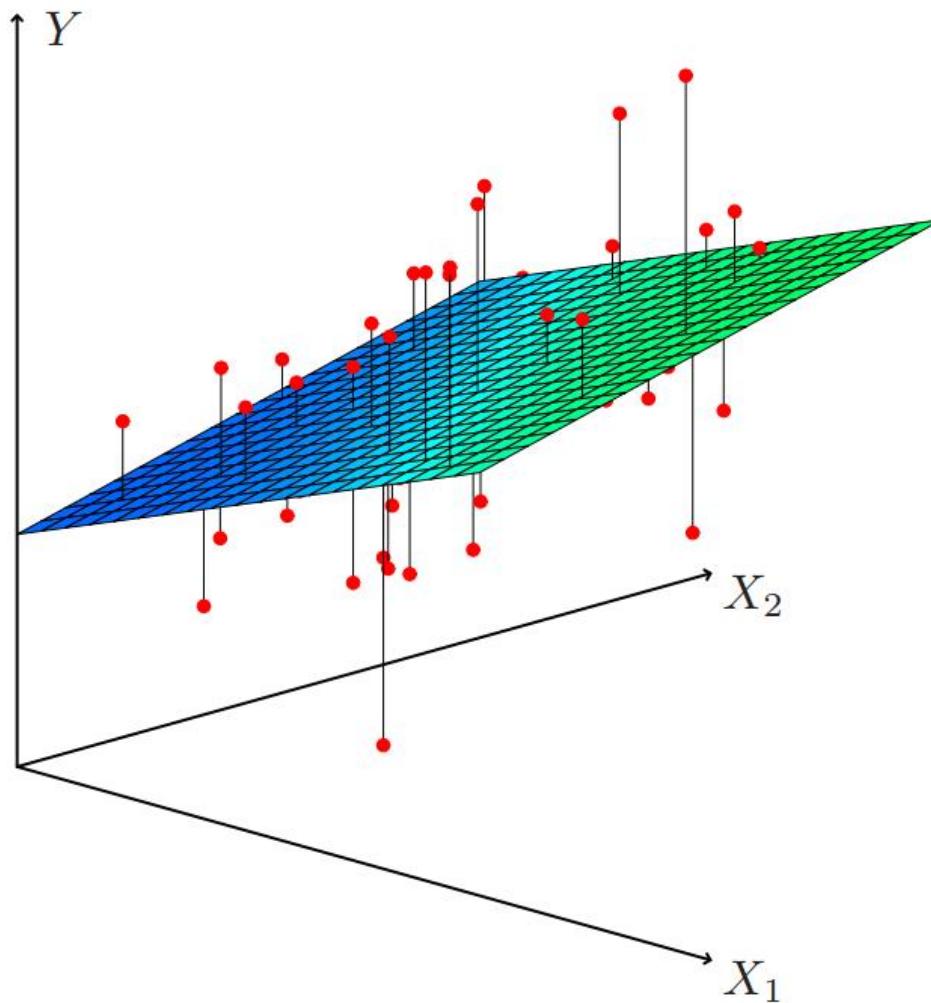
- $\mathbf{y} = (y_1, \dots, y_n) \in \mathbb{R}^n$ is the $n \times 1$ response vector
- $\mathbf{X} = [1_n, \mathbf{X}'] \in \mathbb{R}^{n \times (p+1)}$ is the $n \times (p + 1)$ design matrix
 - 1_n is an $n \times 1$ vector of ones
 - $\mathbf{X}' = [\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_p] \in \mathbb{R}^{n \times p}$ the $n \times p$ predictor Matrix
- $\mathbf{b} = (\beta_0, \beta_1, \dots, \beta_p) \in \mathbb{R}^{p+1}$ is regression coefficient vector
- $\mathbf{e} = (e_1, e_2, \dots, e_3) \in \mathbb{R}^n$ is the $n \times 1$ error vector

Example

#	TV	Radio	News	Sales
1	230.1	37.8	69.2	22.1
2	44.5	39.3	45.1	10.4
3	17.2	45.9	69.3	9.3
4	151.5	41.3	58.5	18.5
5	180.8	10.8	58.4	12.9

$$\begin{pmatrix} 22.1 \\ 10.4 \\ 9.3 \\ 18.5 \\ 12.9 \end{pmatrix} = \begin{pmatrix} 1 & 230.1 & 37.8 & 69.2 \\ 1 & 44.5 & 39.3 & 45.1 \\ 1 & 17.2 & 45.9 & 69.3 \\ 1 & 151.5 & 41.3 & 58.5 \\ 1 & 180.8 & 10.8 & 58.4 \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix} + \begin{pmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \end{pmatrix}$$

Linear Regression



Results for advertising data

	Coefficient	Std. Error	t-statistic	p-value
Intercept	2.939	0.3119	9.42	< 0.0001
TV	0.046	0.0014	32.81	< 0.0001
radio	0.189	0.0086	21.89	< 0.0001
newspaper	-0.001	0.0059	-0.18	0.8599

Correlations:

	TV	radio	newspaper	sales
TV	1.0000	0.0548	0.0567	0.7822
radio		1.0000	0.3541	0.5762
newspaper			1.0000	0.2283
sales				1.0000

Interpreting Regression Coefficients

- A regression coefficient β_j estimates the expected change in Y per unit change in X_j , with all other predictors held fixed. But predictors usually change together!
- Example: Y total amount of change in your pocket; $X_1 = \#$ of coins; $X_2 = \#$ of pennies, nickels and dimes. By itself, regression coefficient of Y on X_2 will be > 0 . But how about with X_1 in model?
- Y = number of tackles by a football player in a season; W and H are his weight and height. Fitted regression model is $\hat{Y} = b_0 + 0.5 \cdot W - 0.1 \cdot H$. How do we interpret $\beta_2 < 0$?

➤ “Data Analysis and Regression” Mosteller and Tukey 1977

Two quotes by famous Statisticians

“Essentially, all models are wrong, but some are useful”

George Box

“The only way to find out what will happen when a complex system is disturbed is to disturb the system, not merely to observe it passively”

Fred Mosteller and John Tukey, paraphrasing George Box

Some important questions

1. Is at least one of the predictors X_1, X_2, \dots, X_p useful in predicting the response?
2. Do all the predictors help to explain Y , or is only a subset of the predictors useful?
3. How well does the model fit the data?
4. Given a set of predictor values, what response value should we predict, and how accurate is our prediction?

Is at least one predictor useful?

- For the first question, we can use the F-statistic

$$F = \frac{(TSS - RSS)/p}{RSS/(n - p - 1)} \sim F_{p,n-p-1}$$

- It's similar to a T statistic from a T-Test; The T-test will tell you if a single variable is statistically significant and an F-test will tell you if a group of variables are jointly significant.

Deciding on the important variables

- The most direct approach is called **all subsets** or **best subsets regression**: we compute the least squares fit for all possible subsets and then choose between them based on some criterion that balances training error with model size.
- However we often can't examine all possible models, since they are 2^p of them; for example when $p = 40$ there are over a billion models!
- Instead we need an automated approach that searches through a subset of them. We discuss two commonly use approaches next.

Forward selection

- Begin with the **null model** - a model that contains an intercept but no predictors.
- Fit p simple linear regressions and add to the null model the variable that results in the lowest RSS.
- Add to that model the variable that results in the lowest RSS amongst all two-variable models.
- Continue until some stopping rule is satisfied, for example when all remaining variables have a p-value above some threshold.

Backward selection

- Start with all variables in the model.
- Remove the variable with the largest p-value - that is, the variable that is the least statistically significant.
- The new $(p - 1)$ -variable model is fit, and the variable with the largest p-value is removed.
- Continue until a stopping rule is reached. For instance, we may stop when all remaining variables have a significant p-value defined by some significance threshold.

Qualitative Predictors

- Some predictors are not quantitative but are qualitative, taking a discrete set of values.
- These are also called **categorical predictors** or **factor variables**

Qualitative Predictors

- For Example: We have given a dataset with female and male. When we want to incorporate that into our model, we create a new dummy variable “is_female”

$$x_i = \begin{cases} 1, & \text{if } i\text{th person is female} \\ 0, & \text{if } i\text{th person is male.} \end{cases}$$

Resulting Model:

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i = \begin{cases} \beta_0 + \beta_1 + \epsilon_i, & \text{if } i\text{th person is female} \\ \beta_0 + \epsilon_i & \text{if } i\text{th person is male.} \end{cases}$$

More than two levels

- If we have more categories than two, we simply create more dummy variables
- For example: `eye_color = {blue, brown, green, grey}`
- We create the dummy variables “`has_blue_eyes`”, “`has_green_eyes`”, “`has_grey_eyes`”.
- There is always one variable less than we have levels.
- The level with no dummy variable – brown eye color - is known as the baseline.

Extensions of the Linear Model

Removing the additive assumption: **interactions** and **nonlinearity**

Interactions:

- In our previous analysis of the Advertising data, we assumed that the effect on sales of increasing one advertising medium is independent of the amount spent on the other media.
- For example, the linear model

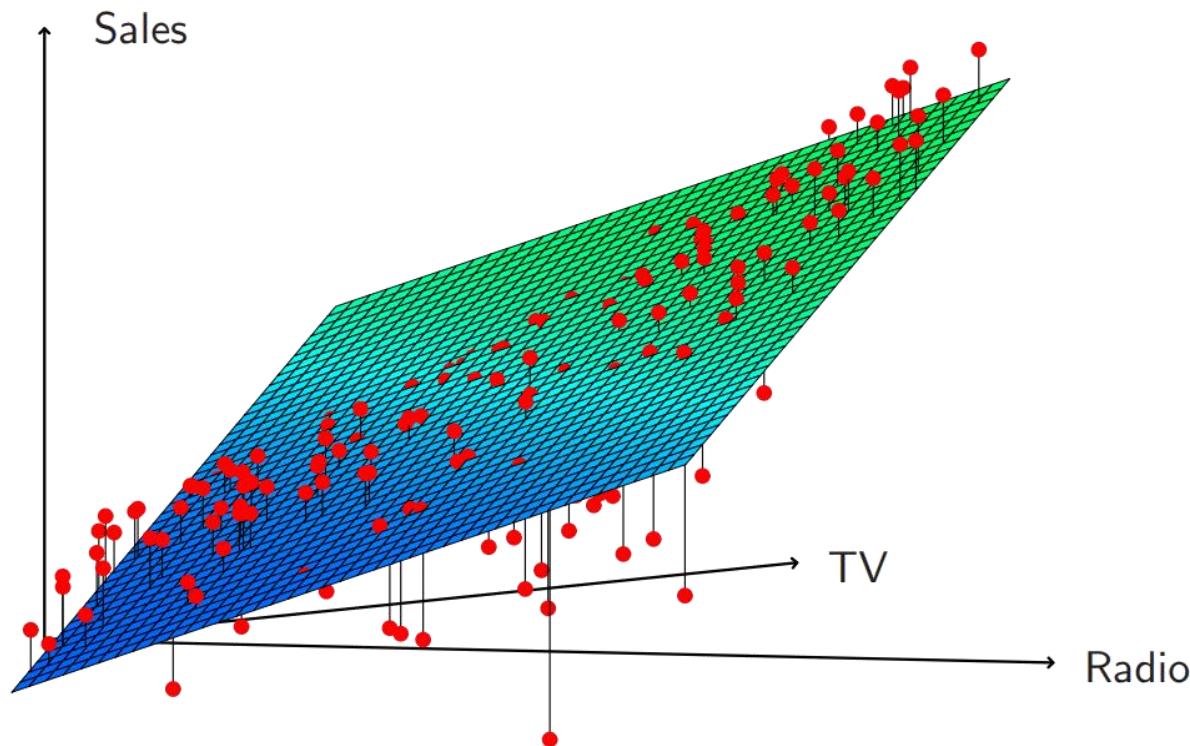
$$\text{sales} = \beta_0 + \beta_1 \cdot \text{TV} + \beta_2 \cdot \text{radio} + \beta_3 \cdot \text{newspaper} + \epsilon$$

states that the average effect on sales of a one-unit increase in TV is always β_1 , regardless of the amount spent on radio.

Interactions

- But suppose that spending money on radio advertising actually increases the effectiveness of TV advertising, so that the slope term for TV should increase as radio increases.
- In this situation, given a fixed budget of \$100,000, spending half on radio and half on TV may increase sales more than allocating the entire amount to either TV or to radio.
- In marketing, this is known as a synergy effect, and in statistics it is referred to as an interaction effect.

Interaction in the Advertising data?



- When levels of either TV or radio are low, then the true sales are lower than predicted by the linear model.
- But when advertising is split between the two media, then the model tends to underestimate sales

New Model

Model takes the form

$$\begin{aligned}\text{sales} &= \beta_0 + \beta_1 \times \text{TV} + \beta_2 \times \text{radio} + \beta_3 \times (\text{radio} \times \text{TV}) + \epsilon \\ &= \beta_0 + (\beta_1 + \beta_3 \times \text{radio}) \times \text{TV} + \beta_2 \times \text{radio} + \epsilon.\end{aligned}$$

Results:

	Coefficient	Std. Error	t-statistic	p-value
Intercept	6.7502	0.248	27.23	< 0.0001
TV	0.0191	0.002	12.70	< 0.0001
radio	0.0289	0.009	3.24	0.0014
TV×radio	0.0011	0.000	20.73	< 0.0001

Interpretation

- The results in this table suggests that interactions are important.
- The p-value for the interaction term $TV \times \text{radio}$ is extremely low, indicating that there is strong evidence for $H_A: \beta_3 \neq 0$.
- The R^2 for the interaction model is 96.8%, compared to only 89.7% for the model that predicts sales using TV and radio without an interaction term.

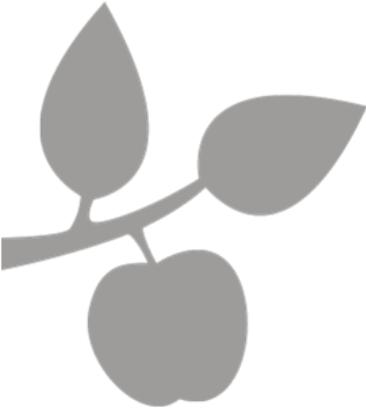
Interpretation — continued

- This means that $(96.8 - 89.7)/(100 - 89.7) = 69\%$ of the variability in sales that remains after fitting the additive model has been explained by the interaction term.
- The coefficient estimates in the table suggest that an increase in TV advertising of \$1, 000 is associated with increased sales of
$$(\beta_1 + \beta_3 \cdot \text{radio}) \times 1000 = 19 + 1.1 \times \text{radio}$$
- An increase in radio advertising of \$1, 000 will be associated with an increase in sales of
$$(\beta_2 + \beta_3 \cdot \text{TV}) \times 1000 = 19 + 1.1 \times \text{TV}$$

Generalizations

There exist many different extensions and generalizations to the simple linear regression model:

- **Classification problems:** logistic regression, support vector machines
- **Non-linearity:** kernel smoothing, splines and generalized additive models; nearest neighbor methods.
- **Interactions:** Tree-based methods, bagging, random forests and boosting (these also capture non-linearities)
- **Regularized fitting:** Ridge regression and lasso



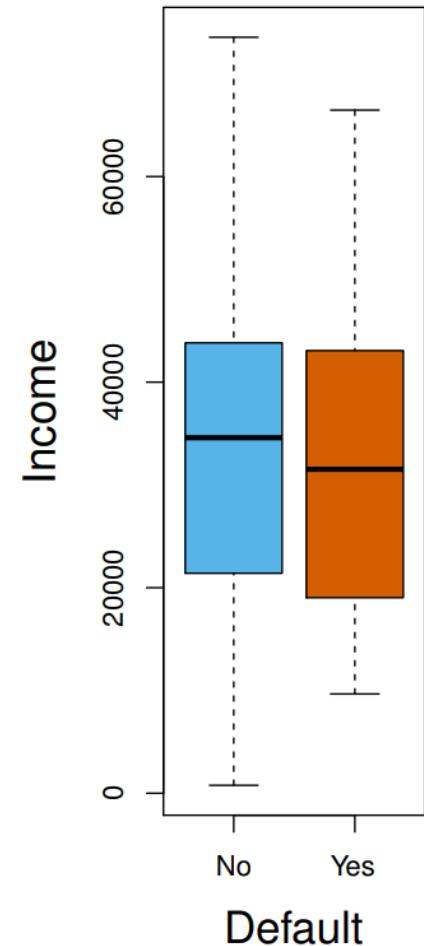
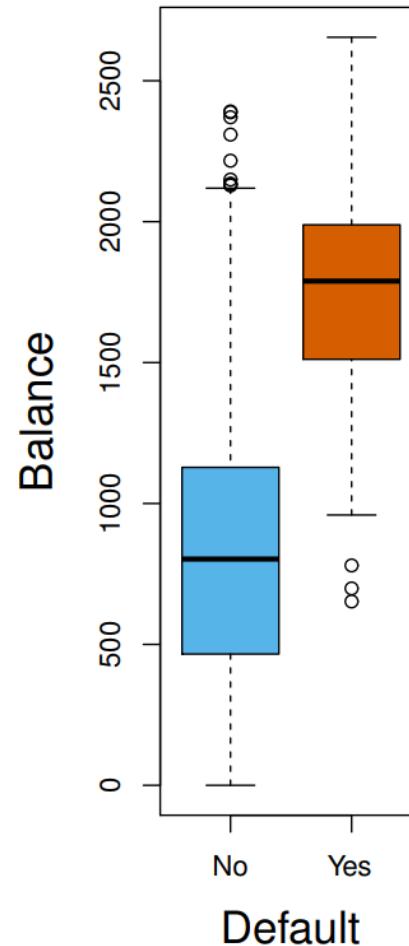
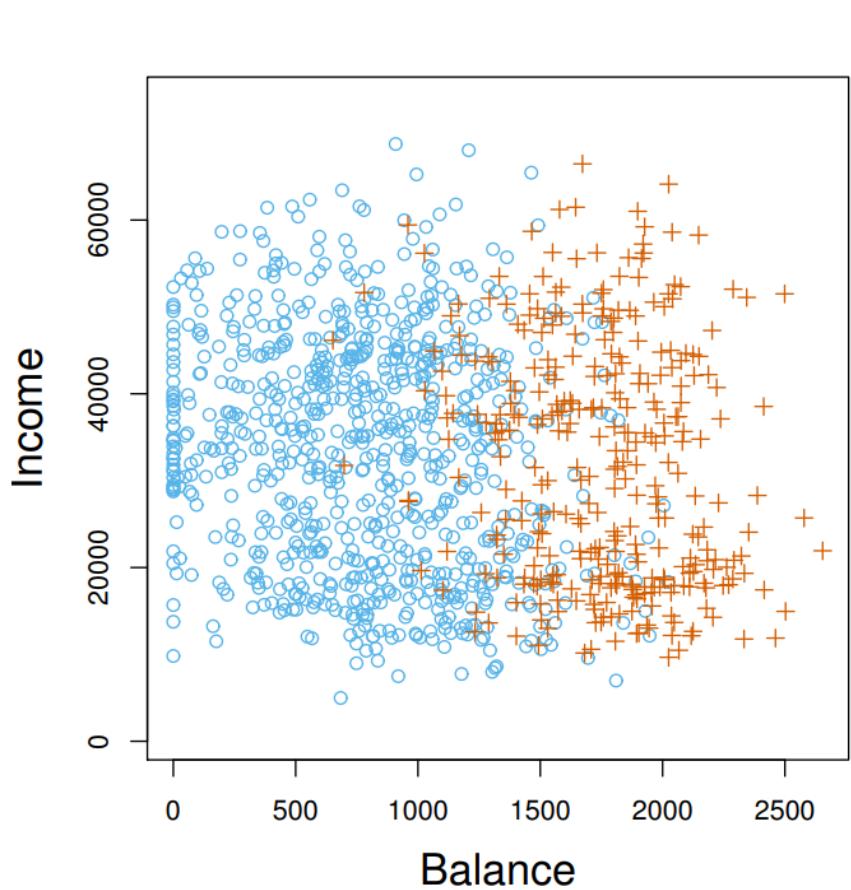
Regression

- Linear Regression
- Logistic Regression

Classification

- Qualitative variables take values in an unordered set C , such as:
 - eye color $\in \{\text{brown, blue, green}\}$
 - email $\in \{\text{spam, ham}\}$.
- Given a feature vector X and a qualitative response Y taking values in the set C , the classification task is to build a function $C(X)$ that takes as input the feature vector X and predicts its value for Y ; i.e. $C(X) \in C$.
- Often we are more interested in estimating the probabilities that X belongs to each category in C .
 - For example, it is more valuable to have an estimate of the probability that an insurance claim is fraudulent, than a classification fraudulent or not.

Example: Credit Card Default



Can we use Linear Regression?

- Consider the a binary classification task

$$f(x) = \begin{cases} 0, & \text{is No} \\ 1, & \text{if Yes} \end{cases}$$

- Can we simply perform a linear regression of Y on X and classify as Yes if $\hat{Y} > 0.5$?

Can we use Linear Regression?

- Consider the a binary classification task

$$f(x) = \begin{cases} 0, & \text{is No} \\ 1, & \text{if Yes} \end{cases}$$

- Can we simply perform a linear regression of Y on X and classify as Yes if $\hat{Y} > 0.5$?
 - In this case of a binary outcome, linear regression does a good job as a classifier, and is equivalent to linear discriminant analysis
 - Since in the population $E(Y|X = x) = \Pr(Y = 1|X = x)$, we might think that regression is perfect for this task.

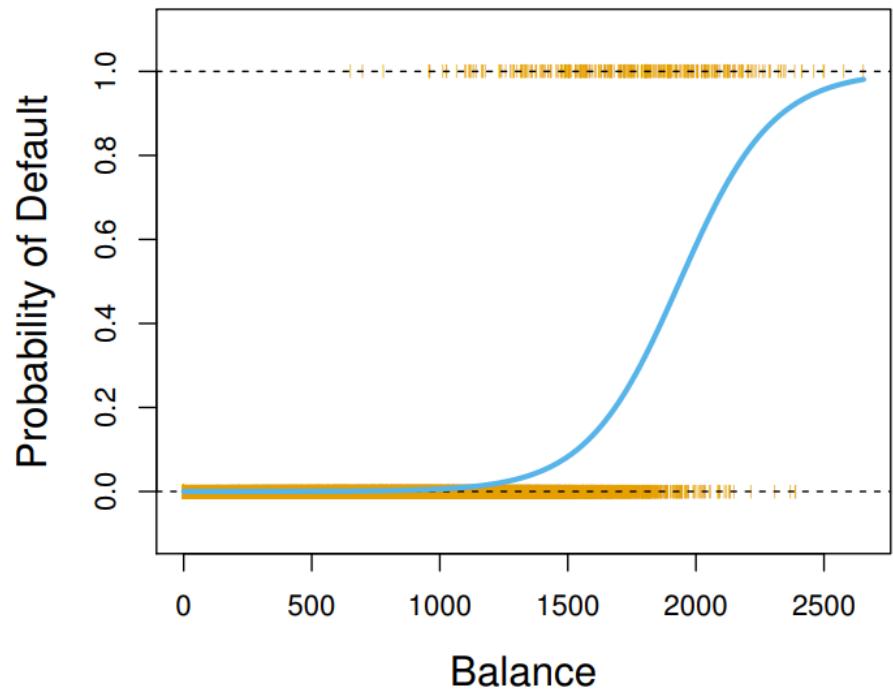
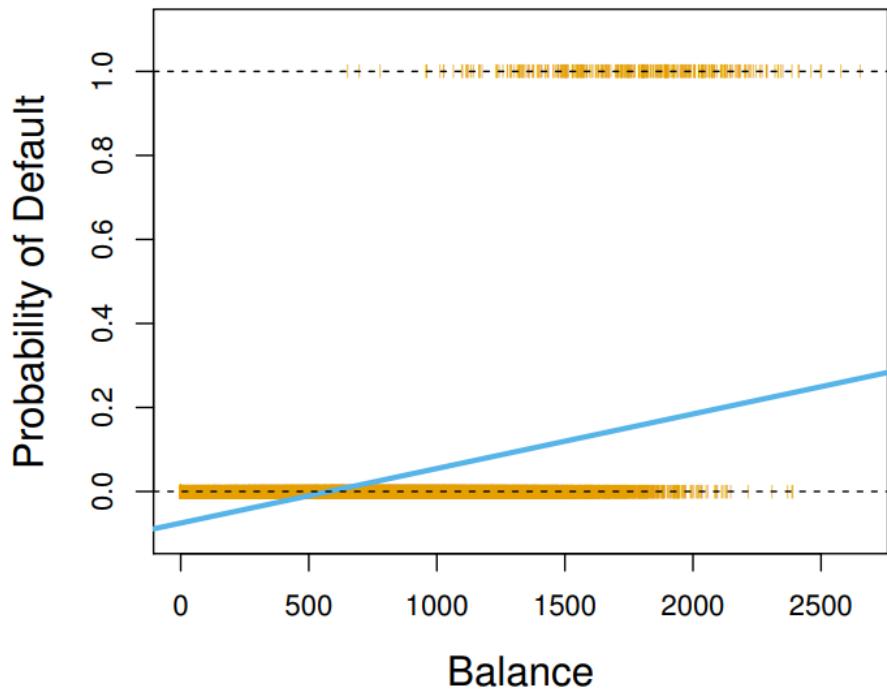
Can we use Linear Regression?

- Consider the a binary classification task

$$f(x) = \begin{cases} 0, & \text{is No} \\ 1, & \text{if Yes} \end{cases}$$

- Can we simply perform a linear regression of Y on X and classify as Yes if $\hat{Y} > 0.5$?
 - In this case of a binary outcome, linear regression does a good job as a classifier, and is equivalent to linear discriminant analysis
 - Since in the population $E(Y|X = x) = \Pr(Y = 1|X = x)$, we might think that regression is perfect for this task.
- However: Linear regression might produce probabilities less than zero or bigger than one. Logistic regression is more appropriate.

Linear versus Logistic Regression



- The orange marks indicate the response Y , either 0 or 1. Linear regression does not estimate $\Pr(Y = 1|X)$ well. Logistic regression seems well suited to the task.

Logistic Regression

- Let's write $p(X) = \Pr(Y = 1|X)$ for short and consider using balance to predict default. Logistic regression uses the form

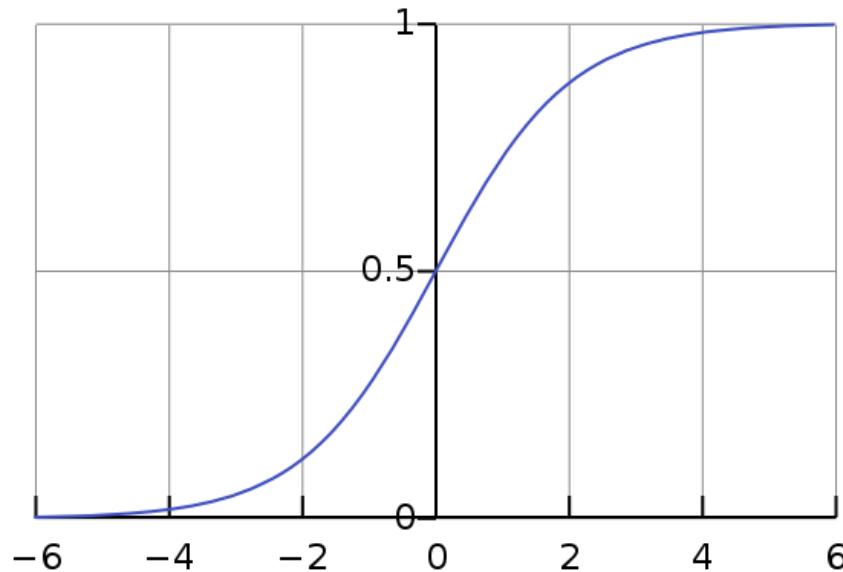
$$p(X) = \frac{e^{\beta_0 + \beta_1 X}}{1 + e^{\beta_0 + \beta_1 X}}$$

- It is easy to see that no matter what values β_0 , β_1 or X take, $p(X)$ will have values between 0 and 1.

Different Point of View

- The Sigmoid Function:

$$S(x) = \frac{e^x}{e^x + 1}$$



Different Point of View

- That means, we can rewrite

$$p(X) = \frac{e^{\beta_0 + \beta_1 X}}{1 + e^{\beta_0 + \beta_1 X}}$$

as

$$p(X) = S(\beta_0 + \beta_1 X)$$

- That means, we perform a standard linear regression and afterwards transform the result by means of a Sigmoid function.
- So, what are we estimating?

Logistic Regression

- Remember the Formula

$$p(X) = \frac{e^{\beta_0 + \beta_1 X}}{1 + e^{\beta_0 + \beta_1 X}}$$

- A bit of rearrangement yields

$$\log\left(\frac{p(x)}{1 - p(x)}\right) = \beta_0 + \beta_1 X.$$

- This monotone transformation is called the log odds or logit transformation of $p(X)$.
- In other words, logistic regression assumes that the log odds is a linear function of X

Yet Another Way Looking at the Problem

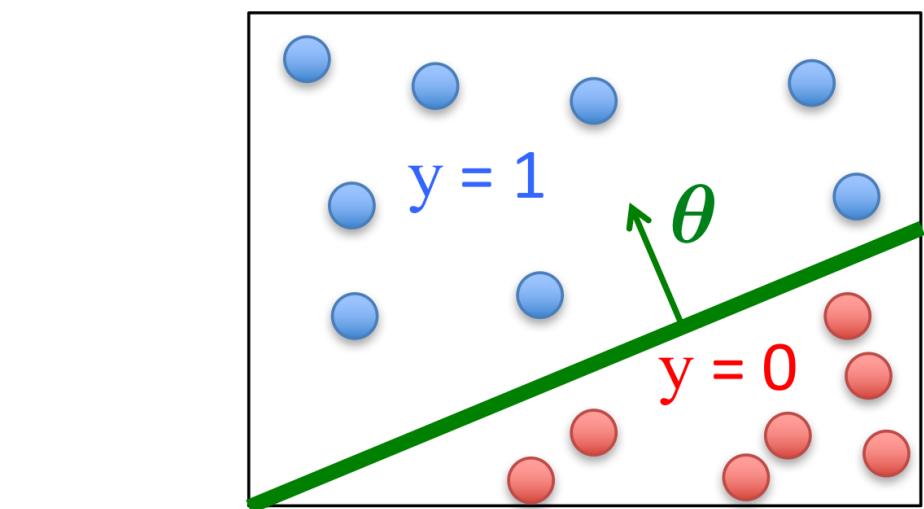
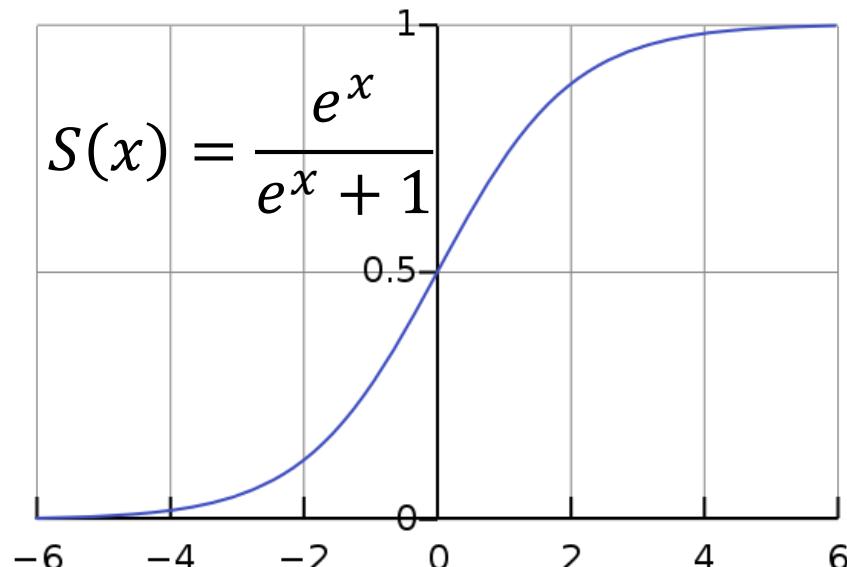
- What we are doing is, instead of predicting the class, we give the probability of the instance being in that class, i.e., learn

$$p(y|x, \theta)$$

- That means, we learn the parameters of a Bernoulli distribution depending on the given input
- We have to chose the θ in such a way, that is maximizes the agreement with our observations

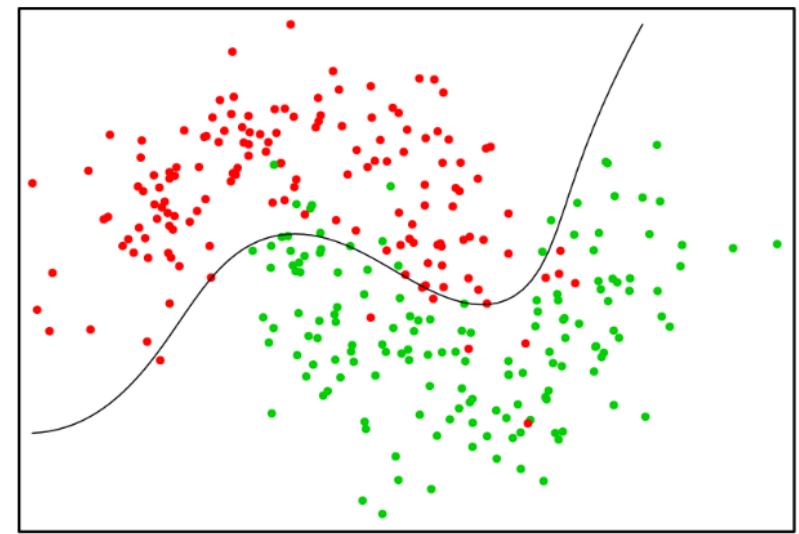
Learning Goal

- Our Model:
$$h_{\theta}(x) = p(y|x, \theta) = S(\theta^T x)$$
- $\theta^T x$ should be large negative for negative instances
- $\theta^T x$ should be large positive for positive instances
- Select a threshold t and
 - Predict $y = 1$ if $p(y|x, \theta) \geq t$
 - Predict $y = 0$ if $p(y|x, \theta) < t$



Non-Linear Decision Boundary

$$\mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1 x_2 \\ x_1^2 \\ x_2^2 \\ x_1^2 x_2 \\ x_1 x_2^2 \\ \vdots \end{bmatrix}$$



Logistic Regression Objective Function

- We cannot just use the least squared approach as with linear regression

$$J(\theta) = \frac{1}{n} \sum_i^n (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

- Since the logistic regression model with the sigmoid function results in a non-convex optimization problem.
- And due to other problem would not lead to the optimal parameter set

Maximum Likelihood Estimation

- The Maximum Likelihood Estimation (MLE) is a method of estimating the parameters of a model. This estimation method is one of the most widely used.
- The method of maximum likelihood selects the set of values of the model parameters that maximizes the likelihood function. Intuitively, this maximizes the "agreement" of the selected model with the observed data.
- The Maximum-likelihood Estimation gives an unified approach to estimation.

What is a Maximum Likelihood Estimator?

- The likelihood that one datapoint was generated by any distribution function is given by

$$l(\boldsymbol{x}, \boldsymbol{\theta}) = p(y|\boldsymbol{x}; \boldsymbol{\theta})$$

- For the entire dataset, we have the likelihood

$$L(\boldsymbol{X}, \boldsymbol{\theta}) = \prod_{i=1}^n p(y^{(i)}|\boldsymbol{x}^{(i)}; \boldsymbol{\theta})$$

What is a Maximum Likelihood Estimator?

- Goal of the MLE is to find $\boldsymbol{\theta}$ in such a way that $L(X, \boldsymbol{\theta})$ is maximized

$$\boldsymbol{\theta}_{\text{MLE}} = \operatorname{argmax}_{\boldsymbol{\theta}} L(X, \boldsymbol{\theta})$$

- Very often (for simplicity), the log likelihood is used

$$\boldsymbol{\theta}_{\text{MLE}} = \operatorname{argmax}_{\boldsymbol{\theta}} \log \prod_{i=1}^n p(y^{(i)} | \mathbf{x}^{(i)}; \boldsymbol{\theta})$$

$$= \operatorname{argmax}_{\boldsymbol{\theta}} \sum_{i=1}^n \log p(y^{(i)} | \mathbf{x}^{(i)}; \boldsymbol{\theta})$$

- This doesn't change the maximum of the function
- Eases many calculations

Example: MLE for a Gaussian

- Target: Estimate μ and σ for a Gaussian. The likelihood is

$$L(X|\theta) = \prod_{i=1}^n \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(x_i - \mu)^2}{2\sigma^2}\right]$$

- Taking the log results in an easier version:

$$LL(X|\theta) = -\frac{1}{2}n \log 2\pi - n \log \sigma - \sum_{i=1}^n \frac{(x_i - \mu)^2}{2\sigma^2}$$

Example: MLE for a Gaussian

- Derivate for μ :

$$\frac{\partial LL}{\partial \mu} = \frac{1}{\sigma^2} \sum_{i=1}^n (x_i - \mu)$$

- And for σ^2 :

$$\frac{\partial LL}{\partial \sigma^2} = \frac{1}{2\sigma^2} \left[\frac{1}{\sigma^2} \sum_{i=1}^n (x_i - \mu)^2 - n \right]$$

- These derivates are 0 if

$$\mu = \frac{1}{n} \sum_{i=1}^n x_i \quad \text{and} \quad \sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2}$$

Back to Logistic Regression

- Our MLE looks as follows:

$$\begin{aligned}\boldsymbol{\theta}_{\text{MLE}} &= \operatorname{argmax}_{\boldsymbol{\theta}} \sum_{i=1}^n \log p(y^{(i)} | \mathbf{x}^{(i)}; \boldsymbol{\theta}) = \\ &= \operatorname{argmax}_{\boldsymbol{\theta}} \sum_{i=1}^n \left[y^{(i)} \log p(y^{(i)} = 1 | \mathbf{x}^{(i)}; \boldsymbol{\theta}) + \right. \\ &\quad \left. + (1 - y^{(i)}) \log (1 - p(y^{(i)} = 1 | \mathbf{x}^{(i)}; \boldsymbol{\theta})) \right] = \\ &= \operatorname{argmax}_{\boldsymbol{\theta}} \sum_{i=1}^n [y^{(i)} \log h_{\boldsymbol{\theta}}(\mathbf{x}) + (1 - y^{(i)}) \log (1 - h_{\boldsymbol{\theta}}(\mathbf{x}))]\end{aligned}$$

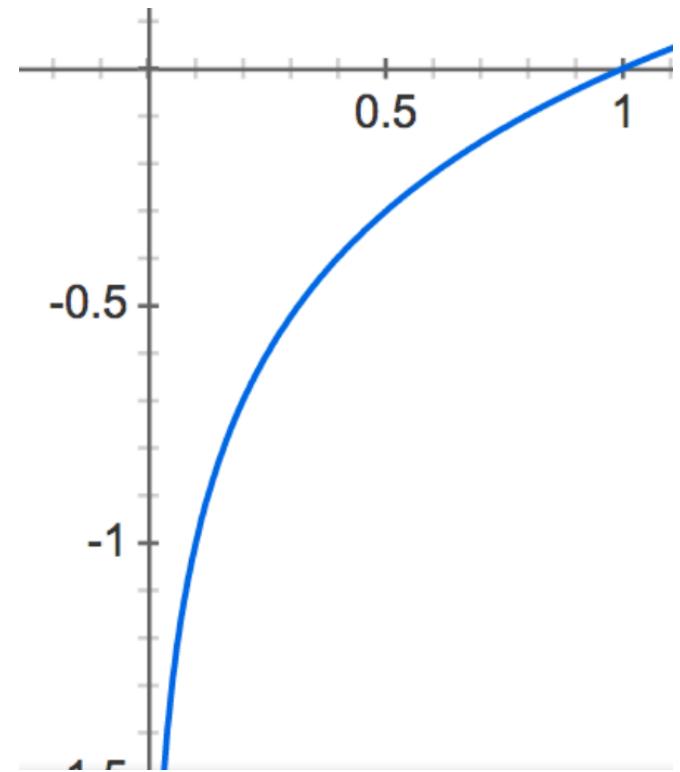
Intuition Behind the Objective

$$\operatorname{argmax}_{\theta} \sum_{i=1}^n [y^{(i)} \log h_{\theta}(x) + (1 - y^{(i)}) \log(1 - h_{\theta}(x))]$$

- Likelihood for a single Instance:

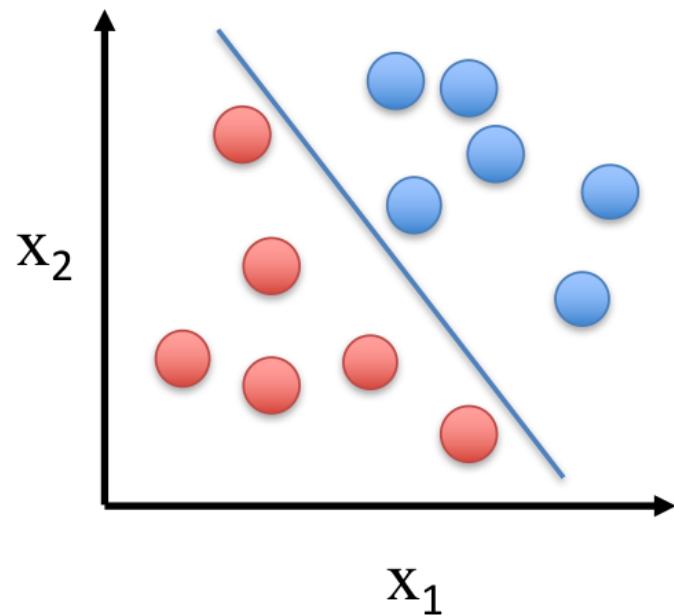
$$l(x, \theta) = \begin{cases} \log(h_{\theta}(x)) & \text{for } y = 1 \\ \log(1 - h_{\theta}(x)) & \text{for } y = 0 \end{cases}$$

- Effect: Extremely negative with wrong classifications
- Trained with gradient descent methods

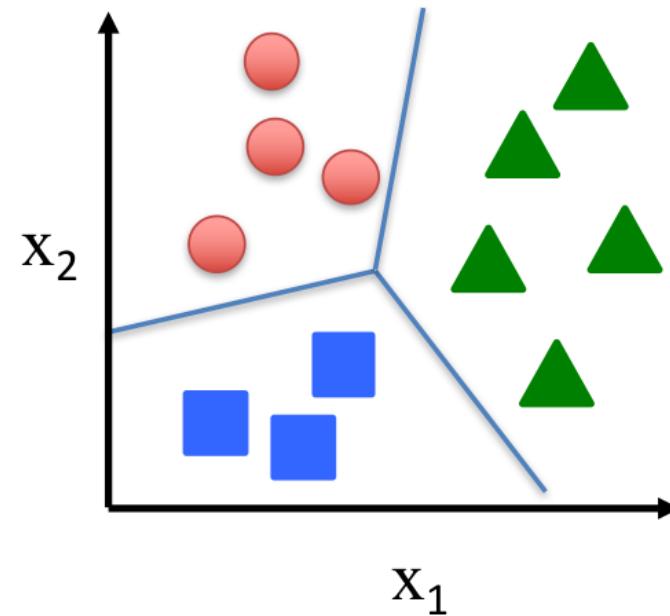


Multi-Class Classification

Binary classification:



Multi-class classification:



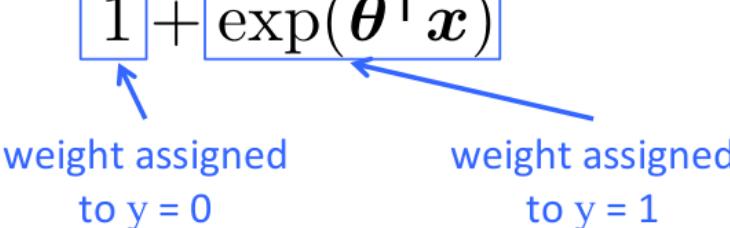
Disease diagnosis: healthy / cold / flu / pneumonia

Object classification: desk / chair / monitor / bookcase

Multi-Class Classification

- For 2 classes:

$$h_{\theta}(x) = \frac{1}{1 + \exp(-\theta^T x)} = \frac{\exp(\theta^T x)}{1 + \exp(\theta^T x)}$$



weight assigned
to $y = 0$ weight assigned
 to $y = 1$

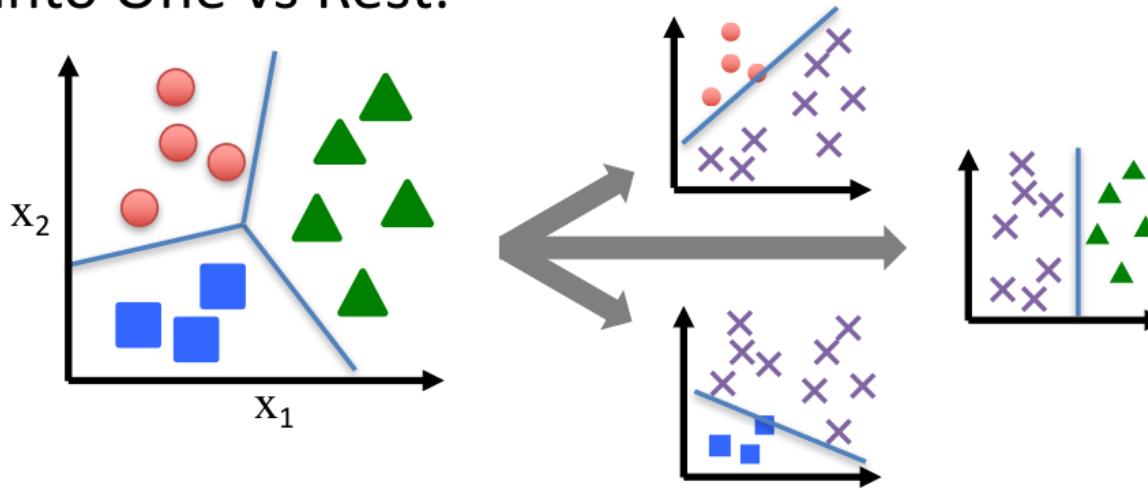
- For C classes $\{1, \dots, C\}$:

$$p(y = c \mid x; \theta_1, \dots, \theta_C) = \frac{\exp(\theta_c^T x)}{\sum_{c=1}^C \exp(\theta_c^T x)}$$

– Called the **softmax** function

Multi-Class Classification

Split into One vs Rest:



- Train a logistic regression classifier for each class i to predict the probability that $y = i$ with

$$h_c(\mathbf{x}) = \frac{\exp(\boldsymbol{\theta}_c^\top \mathbf{x})}{\sum_{c=1}^C \exp(\boldsymbol{\theta}_c^\top \mathbf{x})}$$

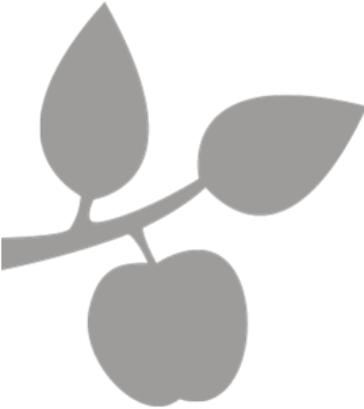


DM873

Deep Learning

Spring 2019

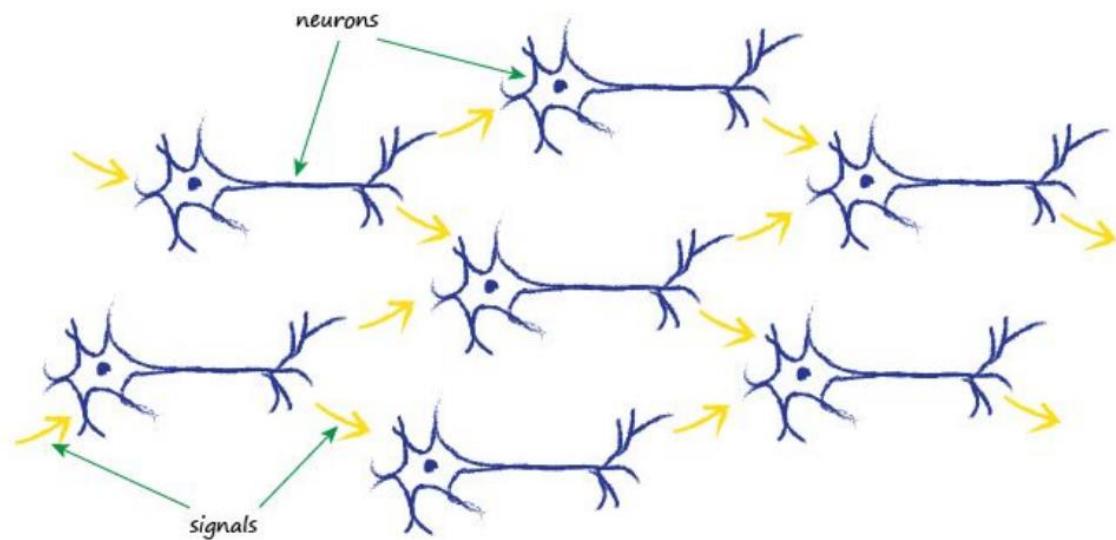
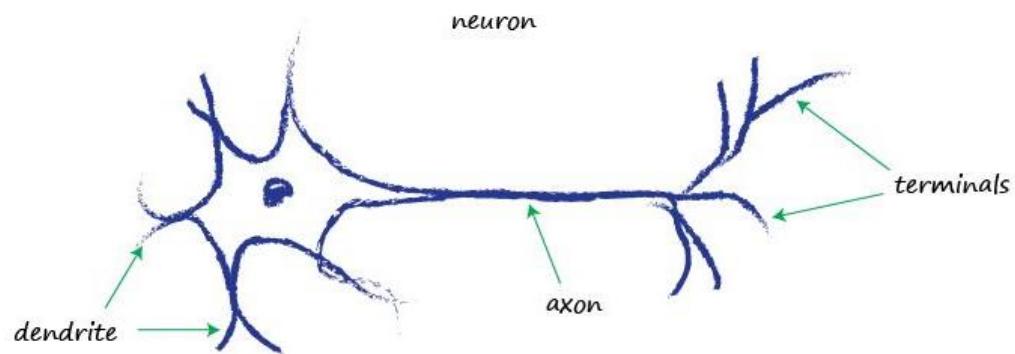
Lecture 5 – Feedforward Networks



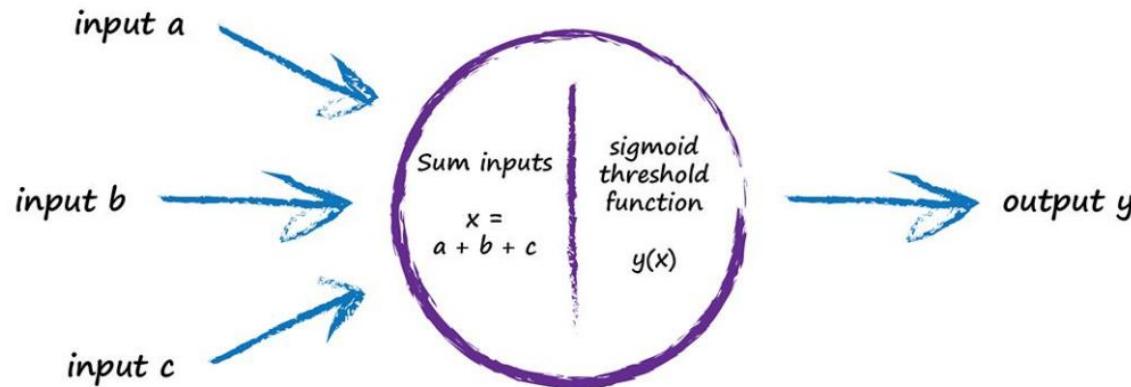
Deep Feedforward Networks

- **Feedforward Networks**
 - **Function Principle**
 - **The power of the activation function**
- **Output Units**
- **Hidden Units**
- **Architecture Design**

The Neuron ... in Nature



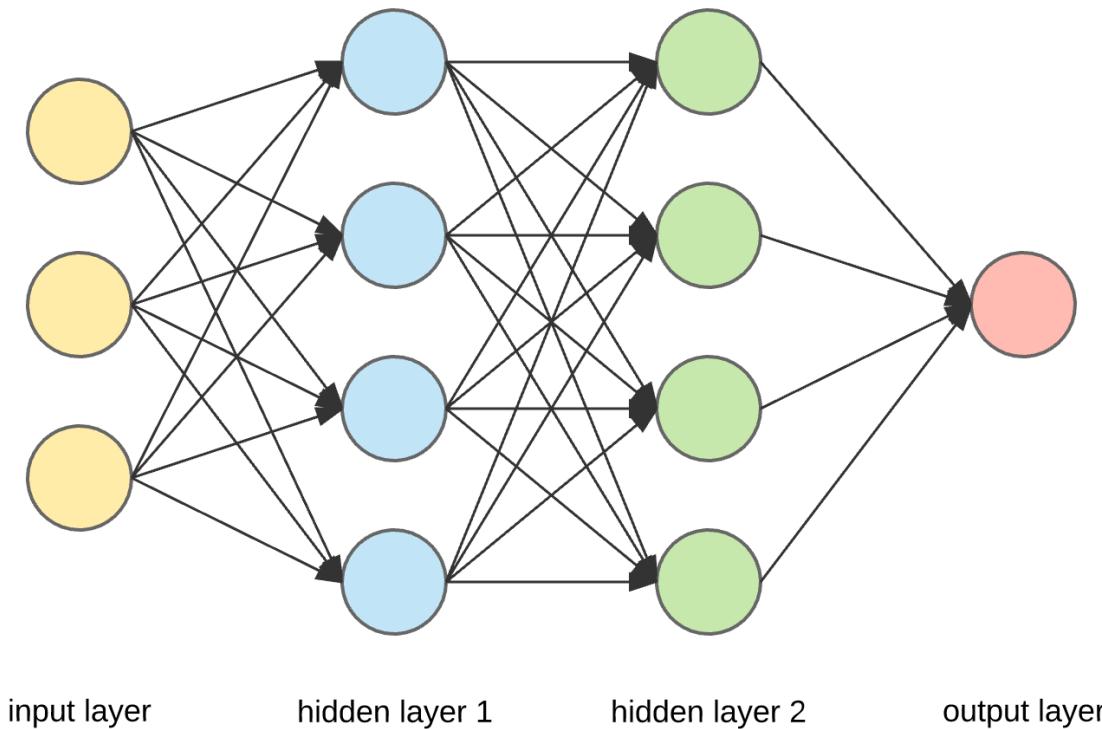
The Artificial Neuron



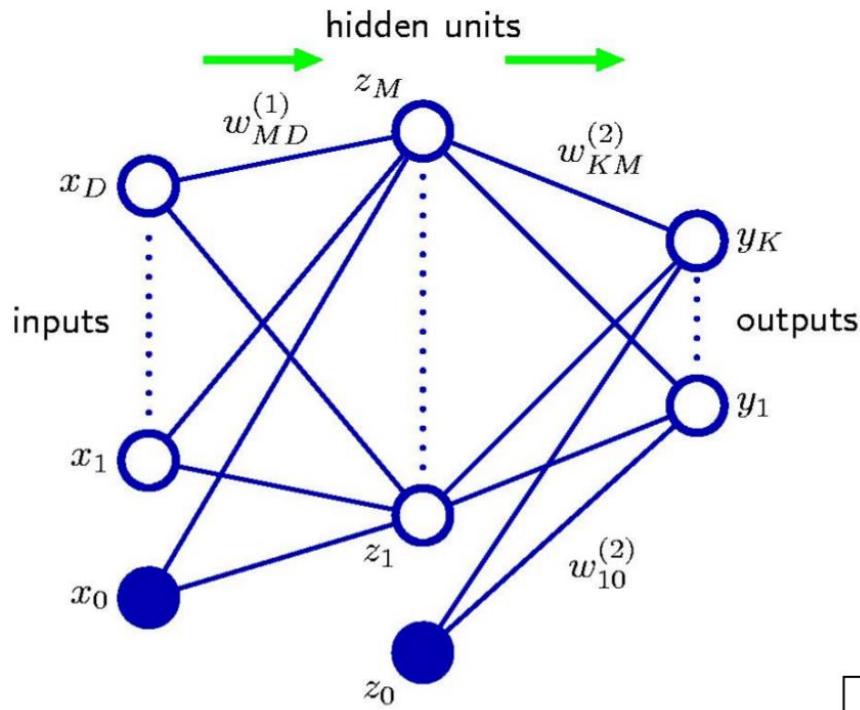
- An artificial Neuron consists of
 - A number of weighted inputs
 - An activation function
 - The generated output

Connecting to a Network

- We normally have more than one node
- Multiple Nodes are arranged in layers
- Each layer receives the generated output from the previous layer



More Mathematically View



K outputs y_1, \dots, y_K for a given input x
Hidden layer consists of M units

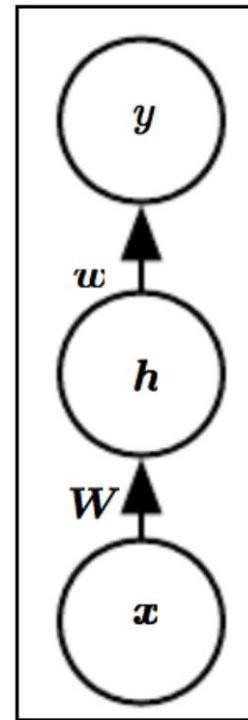
$$y_k(x, w) = \sigma \left(\sum_{j=1}^M w_{kj}^{(2)} h \left(\sum_{i=1}^D w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right)$$

$$\begin{aligned} f_m^{(1)} &= z_m = h(x, w_m^{(1)}), \quad m=1, \dots, M \\ f_k^{(2)} &= \sigma(z, w_k^{(2)}), \quad k=1, \dots, K \end{aligned}$$

Note that there are M versions of $f^{(1)}$ and K versions of $f^{(2)}$

More Compact Representation

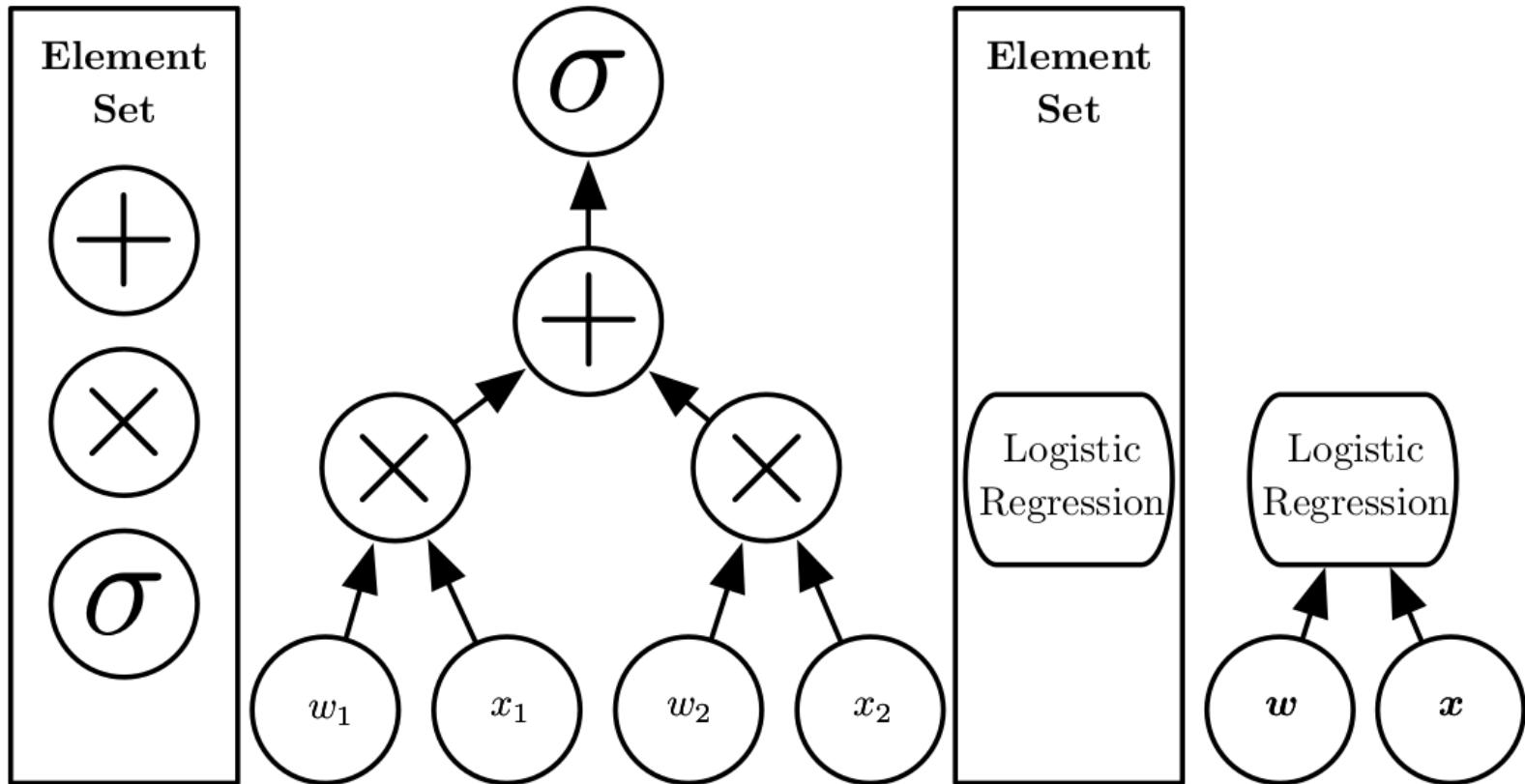
- We can summarize these networks to their essential parts:
- We receive the vector \mathbf{x} and perform an affine transformation according to the weight matrix
$$\mathbf{h}' = \mathbf{W}\mathbf{x} + \mathbf{b}$$
 - Does this ring a bell? Linear regression anyone?
- Afterwards, the internal representation is component wise feed through an activation function $h_i = g(h'_i)$ to generate the input vector \mathbf{h} for the next layer



Composition of Complicated Functions

- Model is associated with a directed acyclic graph describing how functions composed
 - E.g., functions $f^{(1)}, f^{(2)}, f^{(3)}$ connected in a chain to form
$$f(x) = f^{(3)} \left(f^{(2)} \left(f^{(1)}(x) \right) \right)$$
 - $f^{(1)}$ is the first layer, $f^{(2)}$ the second, etc.
 - The length of the chain is the depth of the model
- The name deep learning arises from this terminology
- Final layer of a feedforward network, (here $f^{(3)}$) is the output layer

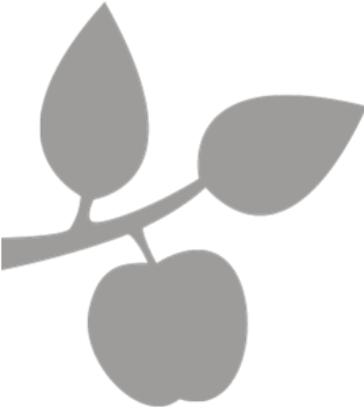
Regarding the Depth of a Model



$$p(y = 1 | \mathbf{x}; \mathbf{w}) = \sigma \left([\mathbf{w}_1, \mathbf{w}_2] \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} \right)$$

What are Hidden Layers

- Behavior of other layers is not directly specified by the data
- Learning algorithm must decide how to use those layers to produce value that is close to y
- Training data does not say what individual layers should do
 - This is one of the main tasks: Define the appropriate structure of the network and the hidden layers
- Since the desired output for these layers is not shown, they are called hidden layers



Deep Feedforward Networks

- **Feedforward Networks**
 - Function Principle
 - **The power of the activation function**
- Output Units
- Hidden Units
- Architecture Design

Extending Linear Models

To represent non-linear functions of x

- Apply linear model to transformed input $\phi(x)$ with non-linear ϕ
 - Equivalently kernel trick of SVM obtains nonlinearity
- Function is nonlinear wrt x but linear wrt $\phi(x)$
- We can think of ϕ as providing a set of features providing a new representation for x
- Get ϕ
 - Generic functions: There exists a set of useful kernel functions
 - Manually engineer ϕ : Laborious and not transformable
 - **Deep Learning:** Learn ϕ

Learn Features

- The Model is now

$$y = f(x; \theta, w) = \phi(x; \theta)^T w$$

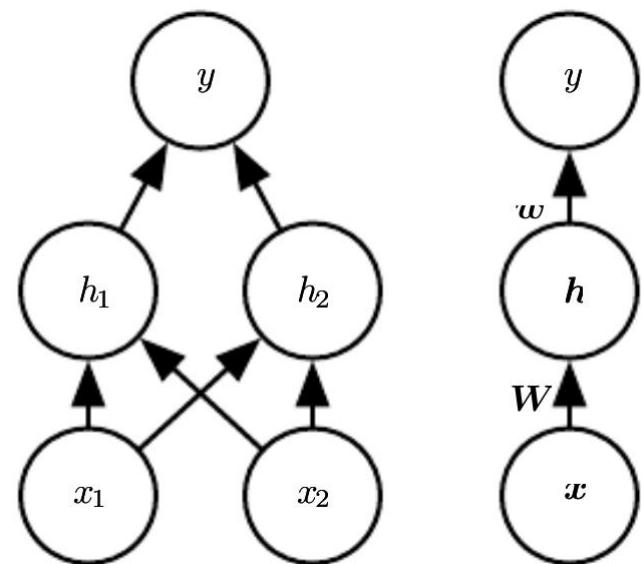
- θ is used to learn from a broad class of functions
- Parameters w map from $\phi(x)$ to output
- In a FFN, ϕ define a hidden layer
- BUT: This gives up on the convexity of the problem!
 - Bad for optimization algorithms
 - Normally, benefits outweigh harms

The XOR Problem

- We are trying to approximate the XOR function
 - Our training points: $X = \{[0,0]^T, [1,0]^T, [0,1]^T, [1,1]^T\}$
 - Using linear regression: $y = f([x_1, x_2]; \mathbf{w}, b) = \mathbf{x}^T \mathbf{w} + b$ to learn XOR
 - When using “mean squared error” as loss function, the result would be $\mathbf{w} = 0$ and $b = \frac{1}{2}$... horizontal line outputting 0.5 everywhere

The XOR Problem

- We are trying to approximate the XOR function
 - Our training points: $X = \{[0,0]^T, [1,0]^T, [0,1]^T, [1,1]^T\}$
 - Using linear regression: $y = f([x_1, x_2]; \mathbf{w}, b) = \mathbf{x}^T \mathbf{w} + b$ to learn XOR
 - When using “mean squared error” as loss function, the result would be $\mathbf{w} = 0$ and $b = \frac{1}{2}$... horizontal line outputting 0.5 everywhere
- Now lets use a FFN
 - One hidden layer, containing two units
 - Both representations are equivalent
 - Matrix \mathbf{W} describes mapping \mathbf{x} to \mathbf{h}
 - Vector \mathbf{w} describes mapping \mathbf{h} to y
 - (biases omitted)



What is gonna be computed

- Layer 1 (hidden layer): vector of hidden units \mathbf{h} computed by function

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

- c are bias variables

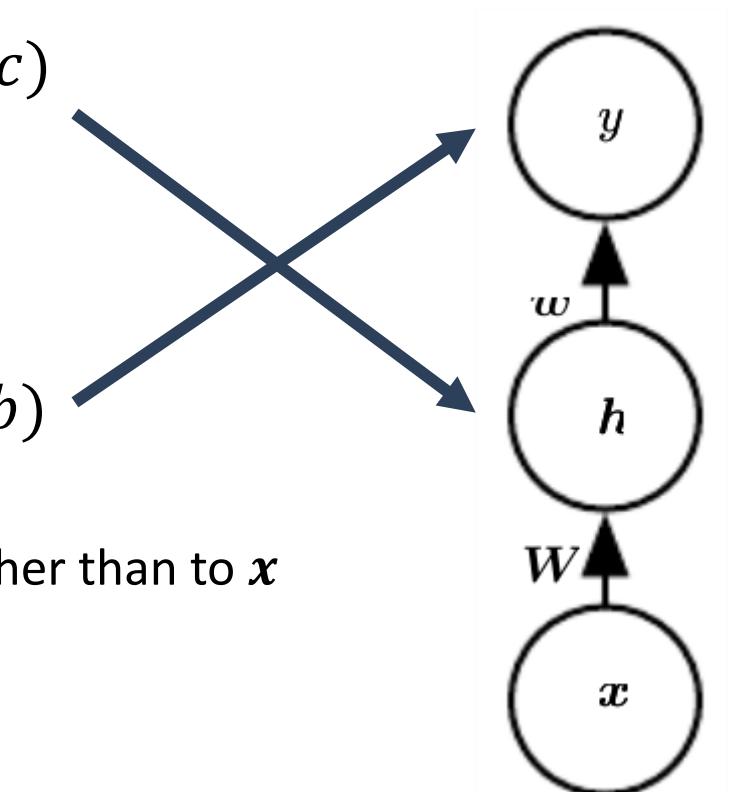
- Layer 2 (output layer) computes

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

- \mathbf{w} are linear regression weights
- Output is linear regression applied to \mathbf{h} rather than to \mathbf{x}

- Complete model is

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



Activation Function

- If we choose both $f^{(1)}$ and $f^{(2)}$ to be linear, the total function will still be linear; which is insufficient (as already seen)

Activation Function:

- In linear regression we used a vector of weights \mathbf{w} and scalar bias b

$$y = \mathbf{x}^T \mathbf{w} + b$$

- Now we describe an affine transformation from a vector \mathbf{x} to a vector \mathbf{h} , so an entire vector of bias parameters is needed
- Activation function g is typically chosen to be applied element-wise

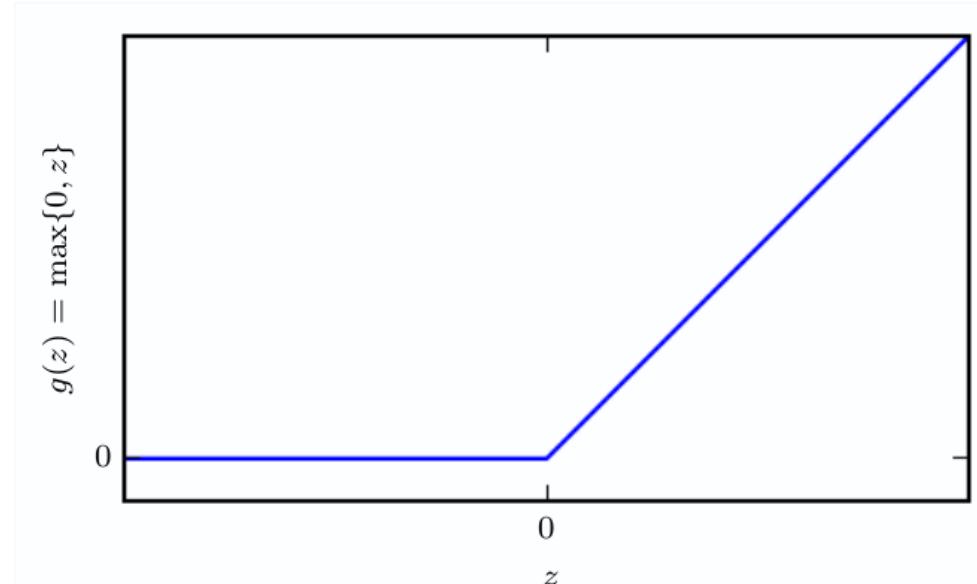
$$h_i = g(\mathbf{x}^T W_{:i} + c_i)$$

Default Activation Function: ReLU

- Activation:

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

- This already yields to a non-linear transformation
 - But still piecewise linear
 - Preserves couple of nice properties making gradient-based learning easy
 - Generalizes well



Back to our XOR Problem

- The complete equation is now:

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

Lets use the following weights:

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0 \text{ and } \mathbf{X} = \begin{bmatrix} 0101 \\ 0011 \end{bmatrix}^T$$

Back to our XOR Problem

- The complete equation is now:

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

Lets use the following weights:

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0 \text{ and } \mathbf{X} = \begin{bmatrix} 0101 \\ 0011 \end{bmatrix}^T$$

$$\mathbf{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

Back to our XOR Problem

- The complete equation is now:

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

Lets use the following weights:

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0 \text{ and } \mathbf{X} = \begin{bmatrix} 0101 \\ 0011 \end{bmatrix}^T$$

$$\mathbf{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$
$$\mathbf{XW} + \mathbf{c} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

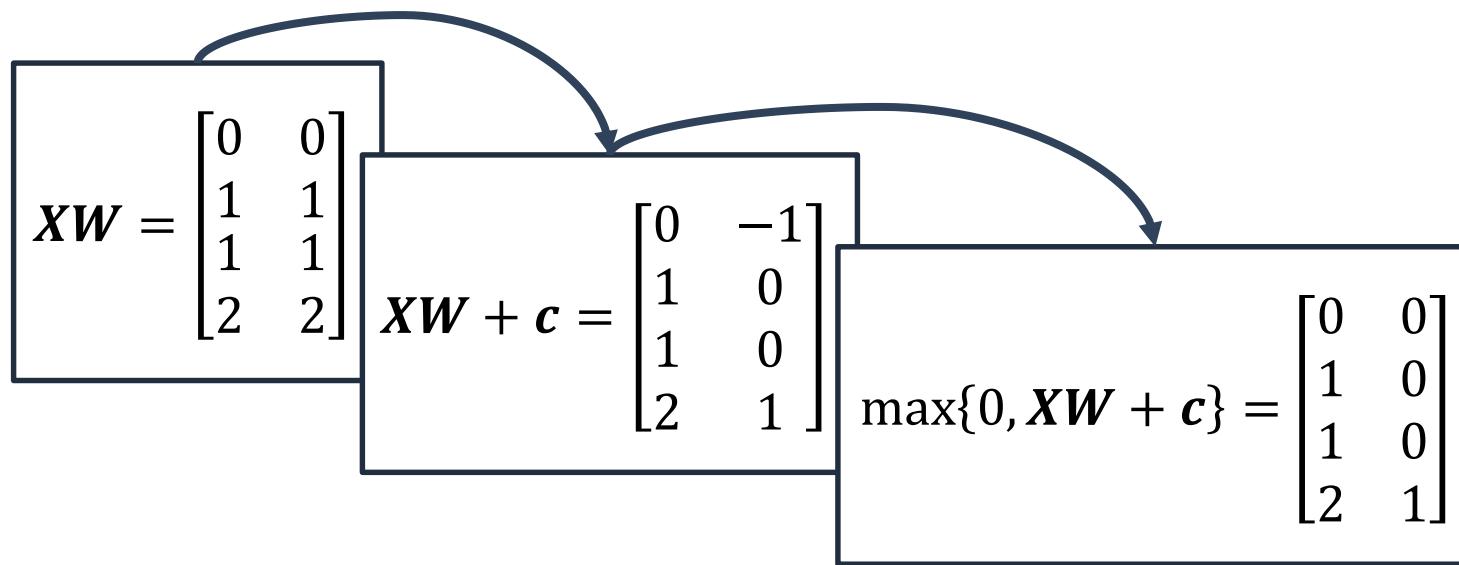
Back to our XOR Problem

- The complete equation is now:

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

Lets use the following weights:

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0 \text{ and } \mathbf{X} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}^T$$



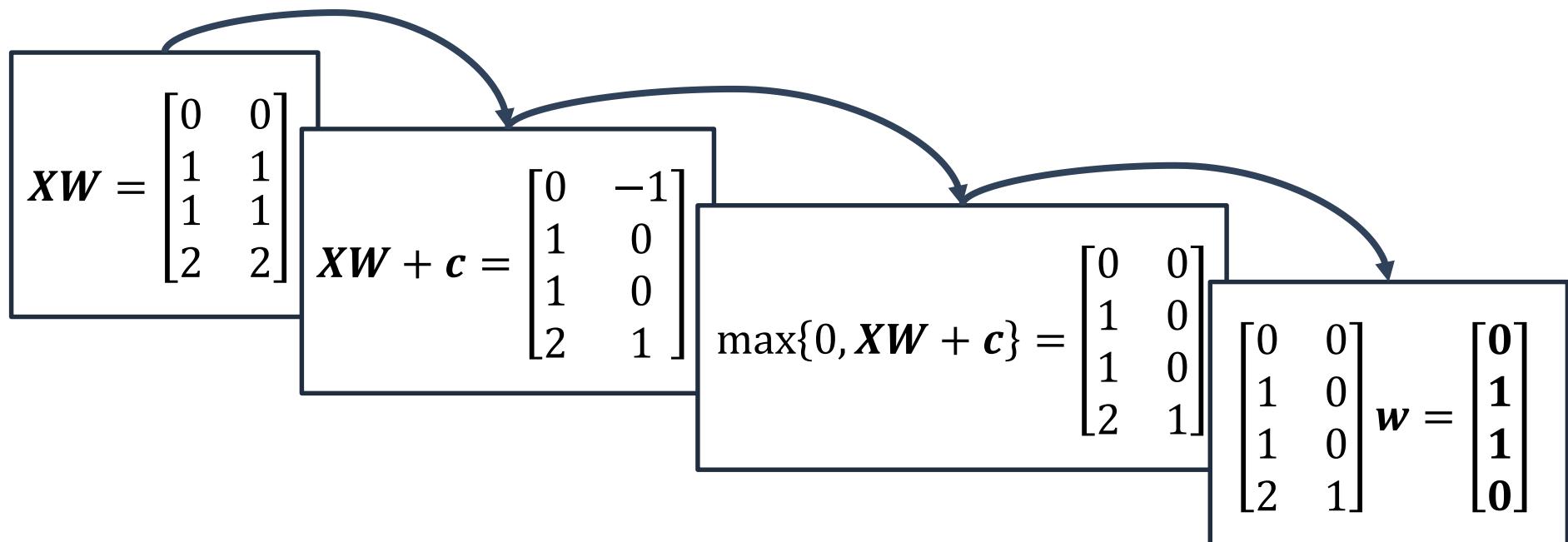
Back to our XOR Problem

- The complete equation is now:

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

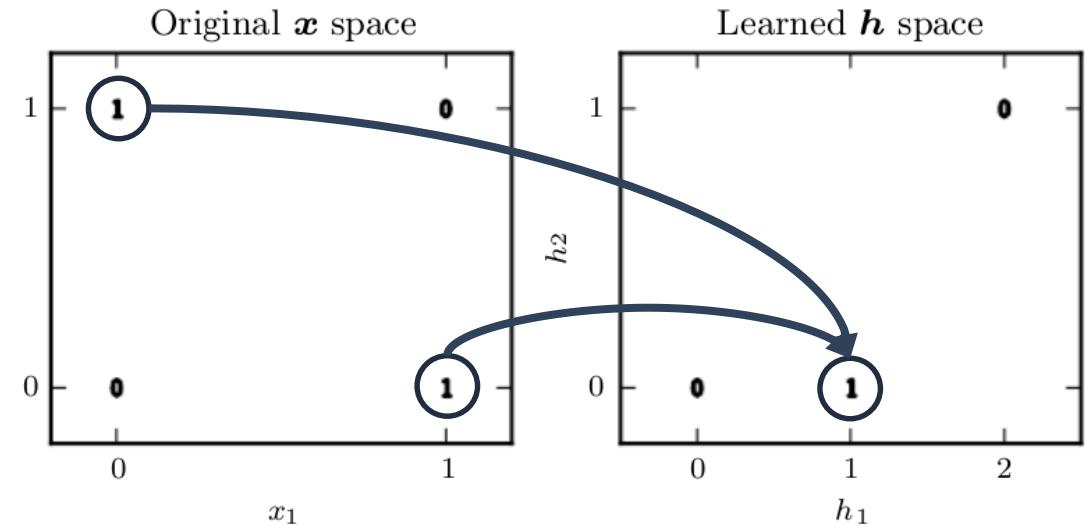
Lets use the following weights:

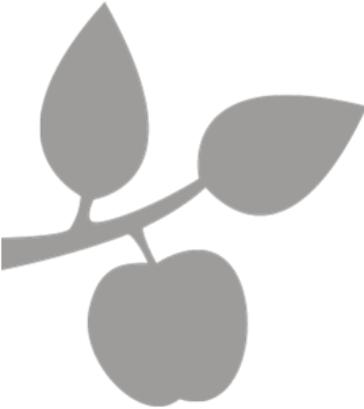
$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0 \text{ and } \mathbf{X} = \begin{bmatrix} 0101 \\ 0011 \end{bmatrix}^T$$



“Learned XOR”

- Two points that must have output 1 have been collapsed into one
- Can be separated in a linear model:
 - For fixed h_2 output has to increase in h_1
- Of course, we “cheated”: We have “guessed” the parameters correctly which lead to the desired result
- In reality: Millions of parameters which we have to actually learn



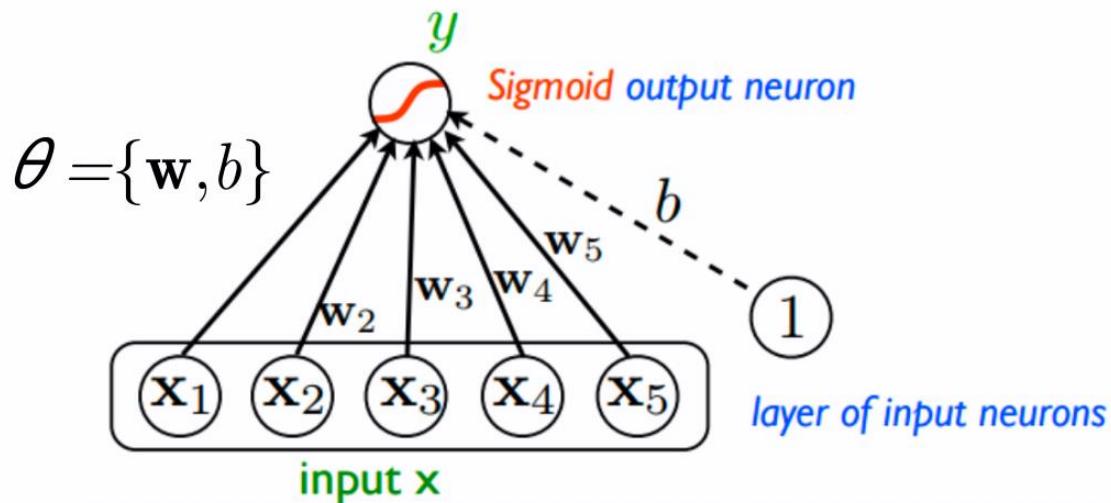


Deep Feedforward Networks

- **Feedforward Networks**
 - Function Principle
 - The power of the activation function
- **Output Units**
- **Hidden Units**
- **Architecture Design**

Output Units

- The choice of cost function is **tightly** coupled with the choice of output unit
- Most of the time we use cross-entropy between data distribution and model distribution
- Choice of how to represent the output then determines the form of the cross-entropy function



Role of a Output Unit

- Any output unit is in principal also usable as a hidden unit
- A feedforward network provides a hidden set of features
$$\mathbf{h} = f(\mathbf{x}; \boldsymbol{\theta})$$
- Role of output layer is to provide some additional transformation from the features to the task that network must perform
- Common Output Units
 - **Linear units:** no non-linearity (used for Gaussian distributions)
 - **Sigmoid units** (used for Bernoulli distributions)
 - **Softmax units** (used for Multinoulli distributions)

Linear Units for Gaussian Output Distributions

- Linear unit: simple output based on affine transformation with no nonlinearity
- Given features \mathbf{h} , a layer of linear output units produces a vector
$$\hat{\mathbf{y}} = \mathbf{W}^T \mathbf{h} + \mathbf{b}$$
- Linear units are often used to produce mean $\hat{\mathbf{y}}$ of a conditional Gaussian distribution
$$P(\mathbf{y}|\mathbf{x}) = N(\mathbf{y}; \hat{\mathbf{y}}, \sigma^2)$$

Sigmoid Units for Bernoulli Output Distributions

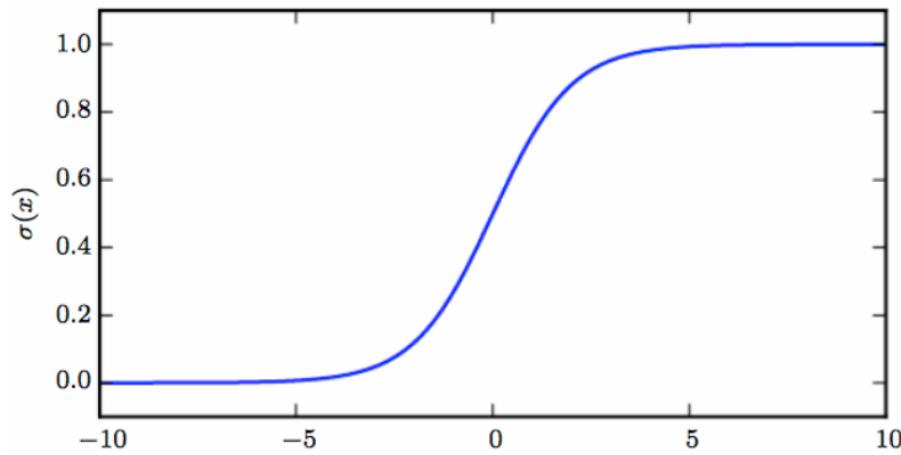
- Task of predicting value of binary variable y
 - Classification problem with two classes
- Maximum likelihood approach is to define a Bernoulli distribution over y conditioned on x
- Neural net needs to predict $p(y = 1|x) \in [0,1]$:
$$P(y = 1|x) = \max\{0, \min\{1, \mathbf{w}^T \mathbf{h} + b\}\}$$
 - We would define a valid conditional distribution, but cannot train it effectively with gradient descent
 - A gradient of 0: learning algorithm cannot be guided

Sigmoid Units

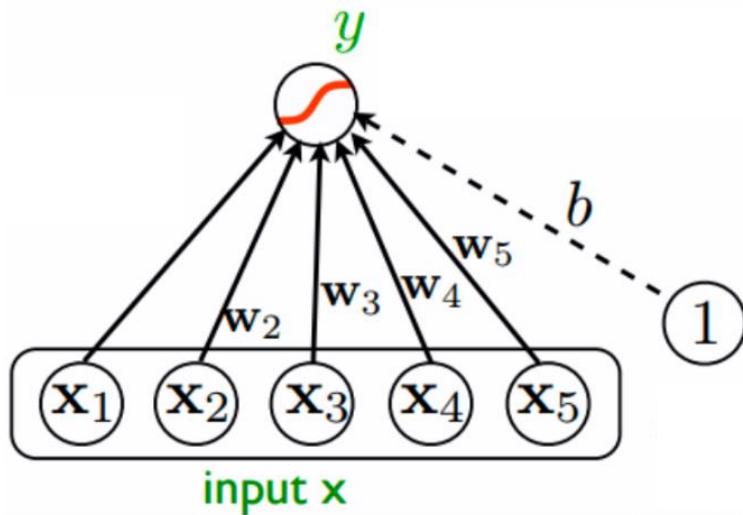
- Sigmoid always gives a gradient
$$\hat{y} = \sigma(\mathbf{w}^T \mathbf{h} + b)$$

with

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



- Sigmoid Unit has two components:
 - A linear layer to compute
$$z = \mathbf{w}^T \mathbf{h} + b$$
 - A sigmoid activation function to convert z into a probability



Softmax units for Multinoulli Output

- Any time we want a probability distribution over a discrete variable with n values we may us the softmax function
- Softmax most often used for output of classifiers to represent distribution over n classes
 - Also inside the model itself when we wish to choose between one of n options
- For the Binary case we have produced a single number
$$\hat{y} = P(y = 1|x)$$
- Now, we have to produce a vector $\hat{\mathbf{y}}$
$$\hat{y}_i = P(y = i|x)$$

Softmax Definition

- The same procedure as for the Bernoulli distribution
- We have a linear layer predicting unnormalized log probabilities

$$\mathbf{z} = \mathbf{W}^T \mathbf{h} + \mathbf{b}$$

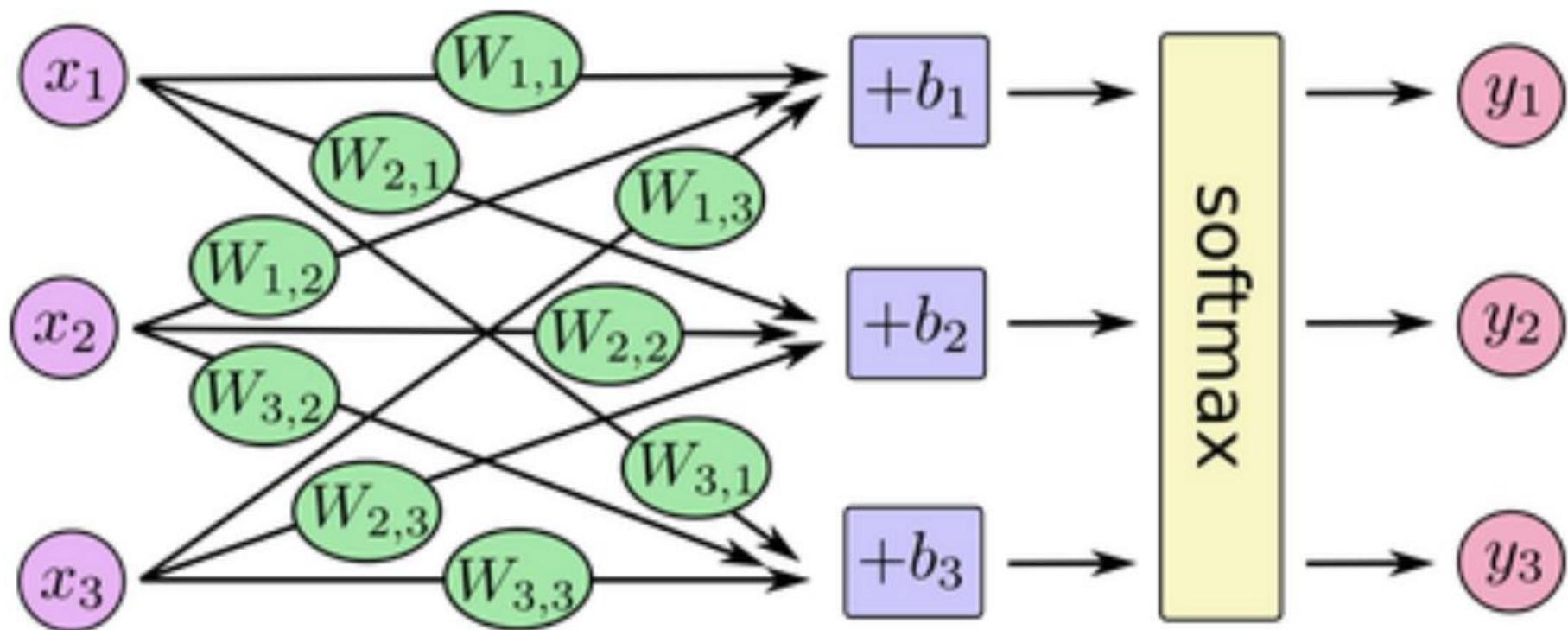
with

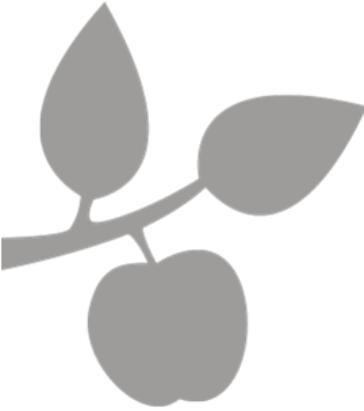
$$z_i = \log \hat{P}(y = i | \mathbf{x})$$

- Softmax can then exponentiate and normalize \mathbf{z} to get $\hat{\mathbf{y}}$

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Softmax Regression





Deep Feedforward Networks

- **Feedforward Networks**
 - Function Principle
 - The power of the activation function
- **Output Units**
- **Hidden Units**
- **Architecture Design**

General Construction of a Hidden Unit

- Very Similar to an output unit
- Accepts a vector of inputs x and computes an affine transformation $z = W^T x + b$
- Computes an element-wise non-linear function $g(z)$
- Most hidden units are distinguished from each other by the choice of activation function $g(z)$
 - We look at: ReLU, Sigmoid and tanh, and other hidden units

Choice of Hidden Unit

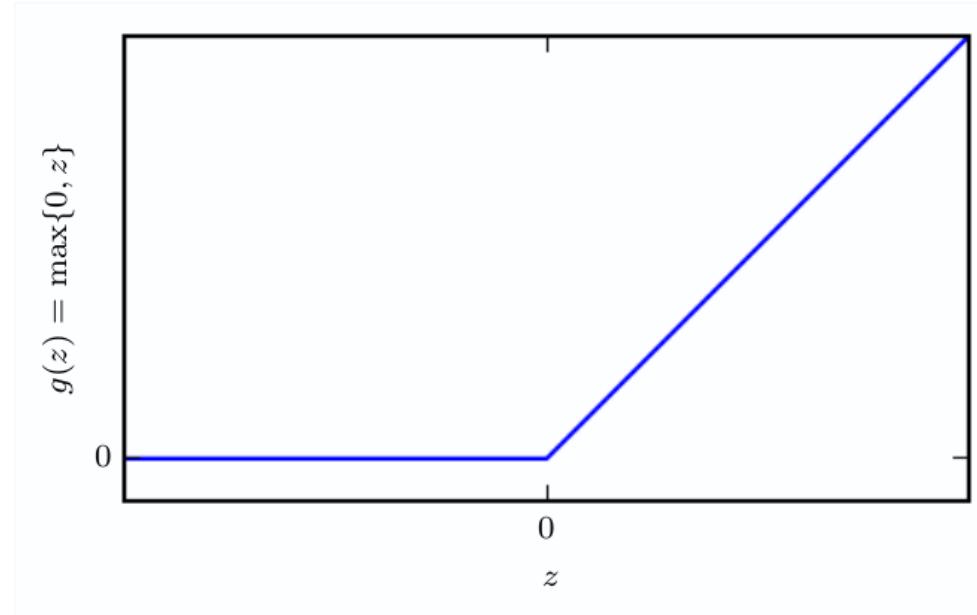
- We now look at how to choose the type of hidden unit in the hidden layers of the model
- Design of hidden units is an active research area that does not have many definitive guiding theoretical principles
- ReLU is an excellent default choice
- But there are many other types of hidden units available
- When to use which kind?
 - Impossible to predict in advance which will work best
 - Design process is trial and error

Is Differentiability necessary?

- Some hidden units are not differentiable at all input points
 - Most notably: Rectified Linear Function
$$g(z) = \max\{0, z\}$$
is not differentiable at $z = 0$
- May seem like it invalidates for use in gradient-based learning
- In practice gradient descent still performs well enough for these models to be used in ML tasks
- Not differentiable hidden units are usually non-differentiable at only a small number of points

Left and Right Differentiability

- Function $g(x)$ has a left and right derivate:
 - defined by the slope Immediately left of x
 - defined by the slope Immediately right of x
- In case of $g(x) = \max\{0, z\}$ the left derivate at $z = 0$ is 0, the right derivate is 1.
- Software normally reports either of the values as derivate



The ReLU

$$\begin{aligned}\mathbf{h} &= g(\mathbf{W}^T \mathbf{x} + \mathbf{b}) \\ g(x) &= \max\{0, x\}\end{aligned}$$

- Good practice to set all elements of \mathbf{b} to a small value such as 0.1
 - This makes it likely that ReLU will be initially active for most training samples and allow derivatives to pass through
- Generalizations of ReLU
 - Perform comparably to ReLU and occasionally perform better
 - ReLU cannot learn on examples for which the activation is zero
 - Generalizations guarantee that they receive gradient everywhere

Generalizations of ReLU

- Three methods based on using a non-zero slope α_i when $z_i < 0$:

$$h_i = g(\mathbf{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

- Absolute-value rectification:
 - fixes $\alpha_i = -1$ to obtain $g(z) = |z|$
- Leaky ReLU:
 - fixes α_i to a small value like 0.01
- Parametric ReLU or PReLU
 - treats α_i as a learnable parameter

Maxout Units

- Maxout units further generalize ReLUs
- Instead of applying element-wise function $g(z)$, maxout units divide z into groups of k values
- Each maxout unit then outputs the maximum element of one of these groups:

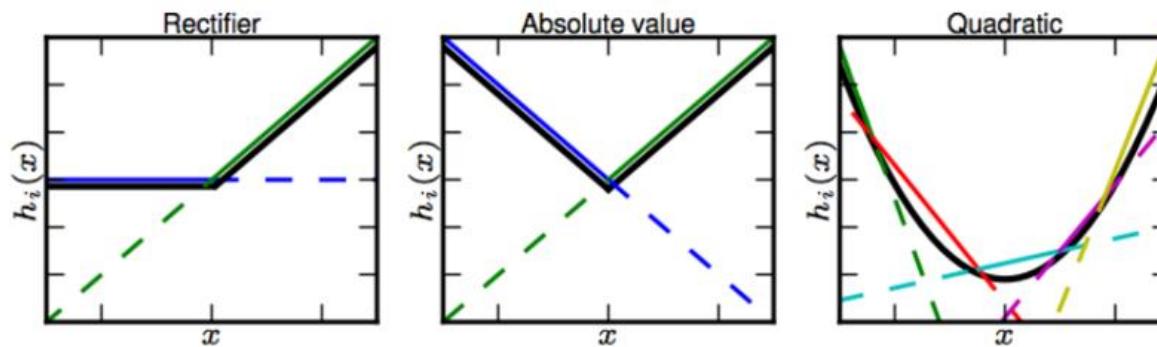
$$g(\mathbf{z})_i = \max_{j \in G^{(i)}} z_j$$

with $G^{(i)}$ is the set of indices for group i , $\{(i - 1)k + 1, \dots, ik\}$

- A maxout unit can learn a piecewise linear, convex function with up to k pieces

Maxout

- Thus seen as **learning the activation function itself** rather than just the relationship between units
- With large enough k , approximate any convex function
- A maxout layer with two pieces can learn to implement the same function as a traditional layer using ReLU or its generalizations



Logistic Sigmoid

- Prior to introduction of ReLU, most neural networks used logistic sigmoid activation

$$g(z) = \sigma(z)$$

- Or the hyperbolic tangent

$$g(z) = \tanh(z)$$

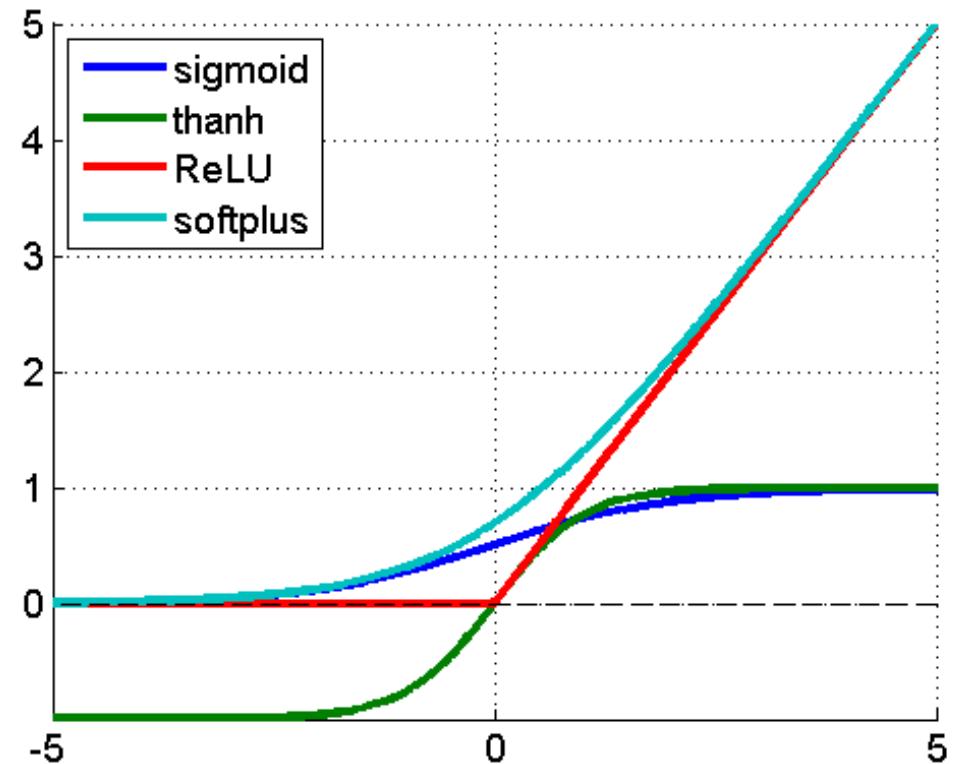
- These activation functions are closely related because

$$\tanh(z) = 2\sigma(2z) - 1$$

- Sigmoid units are used to predict probability that a binary variable is 1

Sigmoid Saturation

- Sigmoidals saturate across most of domain
 - Saturate to 1 when z is very positive and 0 when z is very negative
 - Strongly sensitive to input when z is near 0
 - Saturation makes gradient-learning difficult
- ReLU and Softplus increase for input > 0



Sigmoid vs *tanh* Activation

- Hyperbolic tangent typically performs better than logistic sigmoid
- It resembles the identity function more closely
 $\tanh(0) = 0$ while $\sigma(0) = 1 / 2$
- Because \tanh is similar to identity near 0, training a deep neural network

$$\hat{\mathbf{y}} = \mathbf{w}^T \tanh(\mathbf{U}^T \tanh(\mathbf{V}^T \mathbf{x}))$$

resembles training a linear model

$$\hat{\mathbf{y}} = \mathbf{w}^T \mathbf{U}^T \mathbf{V}^T \mathbf{x}$$

so long as the activations can be kept small

Other Uses of the Sigmoid Activation Function

- Sigmoidal more common in settings other than feed-forward networks
- Recurrent networks, many probabilistic models and autoencoders have additional requirements that rule out piecewise linear activation functions
- They make sigmoid units appealing despite saturation

Other Output Units

- **Cosine**

$$\mathbf{h} = \cos(\mathbf{W}\mathbf{x} + \mathbf{b})$$

- Feedforward networks on MNIST obtained error rate of less than 1%

- **Radial Basis**

$$h_i = \exp\left(-\frac{1}{\sigma^2} \|\mathbf{W}_{:,j} - \mathbf{x}\|^2\right)$$

- Becomes more active as \mathbf{x} approaches a template $\mathbf{W}_{:,i}$

- **Softplus**

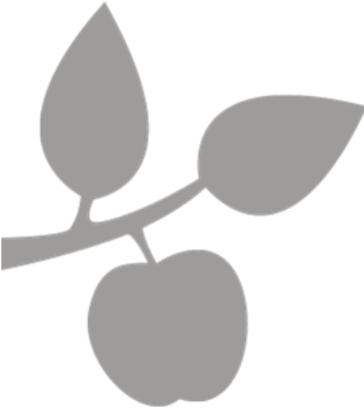
$$g(a) = \zeta(a) = \log(1 + e^a)$$

- Smooth version of the rectifier

- **Hard tanh**

$$g(a) = \max(-1, \min(1, a))$$

- Shaped similar to tanh and the rectifier but it is bounded



Deep Feedforward Networks

- **Feedforward Networks**
 - Function Principle
 - The power of the activation function
- **Output Units**
- **Hidden Units**
- **Architecture Design**

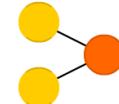
A mostly complete chart of Neural Networks

- Backfed Input Cell
- Input Cell
- △ Noisy Input Cell
- Hidden Cell
- Probabilistic Hidden Cell
- △ Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- △ Different Memory Cell
- Kernel
- Convolution or Pool

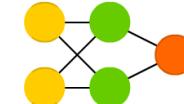
A mostly complete chart of Neural Networks

©2016 Fjodor van Veen - asimovinstitute.org

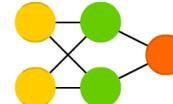
Perceptron (P)



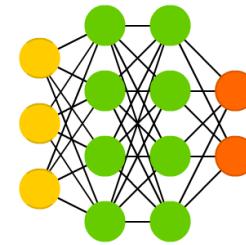
Feed Forward (FF)



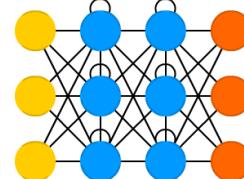
Radial Basis Network (RBF)



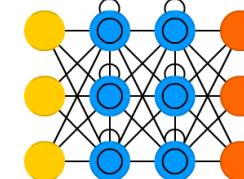
Deep Feed Forward (DFF)



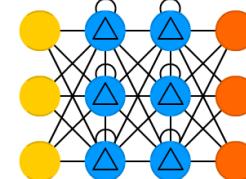
Recurrent Neural Network (RNN)



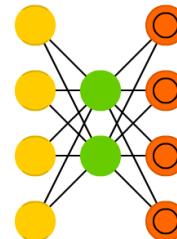
Long / Short Term Memory (LSTM)



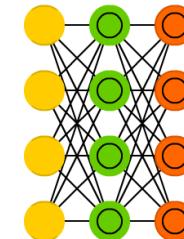
Gated Recurrent Unit (GRU)



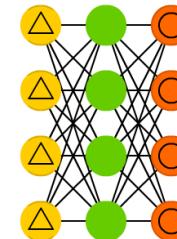
Auto Encoder (AE)



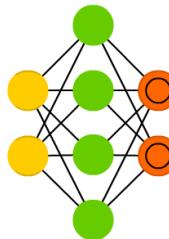
Variational AE (VAE)



Denoising AE (DAE)



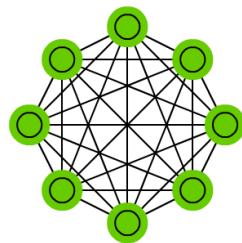
Sparse AE (SAE)



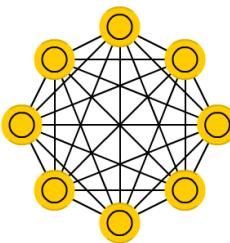
➤ <https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464>

A mostly complete chart of Neural Networks

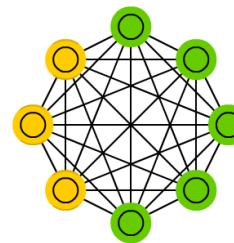
Markov Chain (MC)



Hopfield Network (HN)



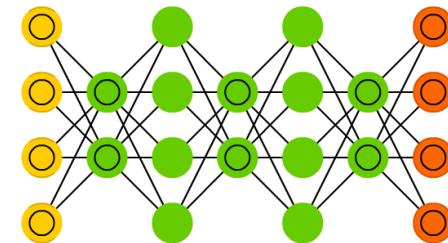
Boltzmann Machine (BM)



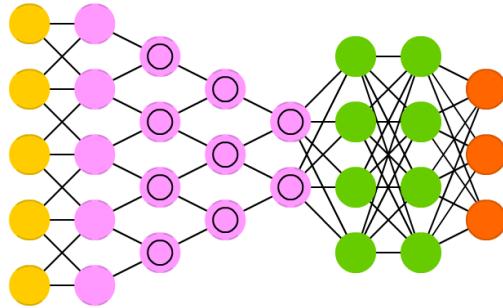
Restricted BM (RBM)



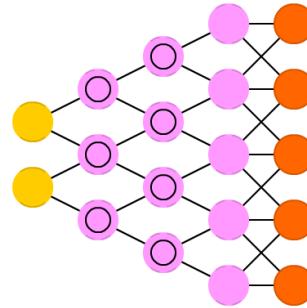
Deep Belief Network (DBN)



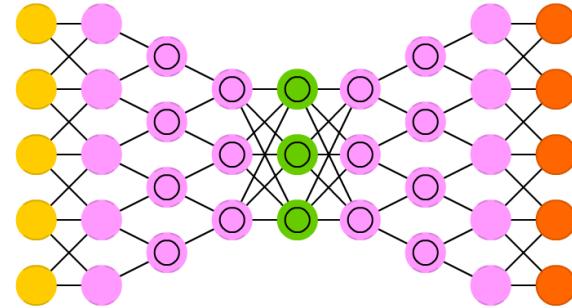
Deep Convolutional Network (DCN)



Deconvolutional Network (DN)



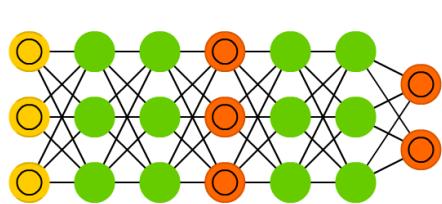
Deep Convolutional Inverse Graphics Network (DCIGN)



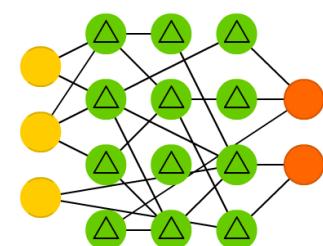
➤ <https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464>

A mostly complete chart of Neural Networks

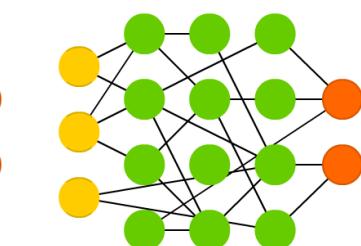
Generative Adversarial Network (GAN)



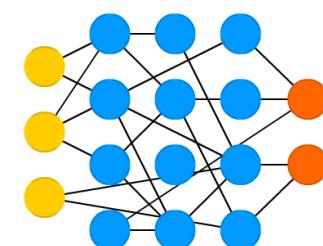
Liquid State Machine (LSM)



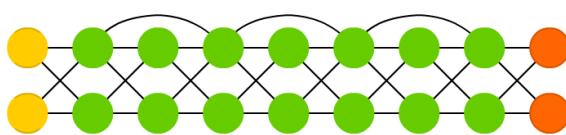
Extreme Learning Machine (ELM)



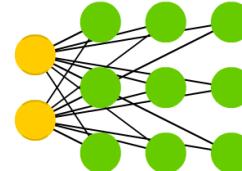
Echo State Network (ESN)



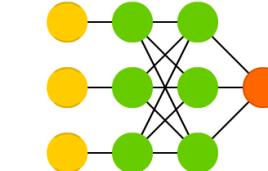
Deep Residual Network (DRN)



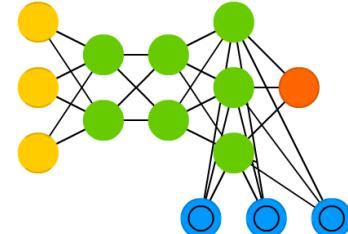
Kohonen Network (KN)



Support Vector Machine (SVM)



Neural Turing Machine (NTM)



➤ <https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464>

Architecture Terminology

- The word architecture refers to the overall structure of the network:
 - How many units should it have?
 - How the units should be connected to each other?
- Most neural networks are organized into groups of units called layers
 - Most neural network architectures arrange these layers in a chain structure
 - With each layer being a function of the layer that preceded it
 - First layer is given by $\mathbf{h}^{(1)} = g^{(1)}(\mathbf{W}^{(1)T}x + \mathbf{b}^{(1)})$
 - Second layer is given by $\mathbf{h}^{(2)} = g^{(2)}(\mathbf{W}^{(2)T}\mathbf{h}^{(1)} + \mathbf{b}^{(2)})$

Main Architectural Considerations

- 1. Choice of depth of network**
- 2. Choice of width of each layer**

- Deeper networks have
 - Far fewer units in each layer
 - Far fewer parameters
 - Often generalize well to the test set
 - But are often more difficult to optimize
- Ideal network architecture must be found via experimentation guided by validation set error

On the Power of Neural Networks

- A linear model, mapping from features to outputs via matrix multiplication, can by definition represent only linear functions.
 - Easy to learn
 - Cost function results in convex optimization problem

The **universal approximation theorem** states that a feedforward network with a linear output layer and at least one hidden layer with any “squashing” activation function and enough units (e.g. sigmoid) can approximate any Borel measurable function (with some restrictions).

- We don't need specialized model families for approximating functions
- However, it does not state how to learn and how to design the appropriate network

Implication of Theorem

- Whatever function we are trying to learn, a large MLP will be able to represent it
- However we are not guaranteed that the training algorithm will learn this function
 - Optimizing algorithms may not find the parameters
 - May choose wrong function due to over-fitting
- No Free Lunch: There is no universal procedure for examining a training set of samples and choosing a function that will generalize to points not in training set

On the Size of the Network

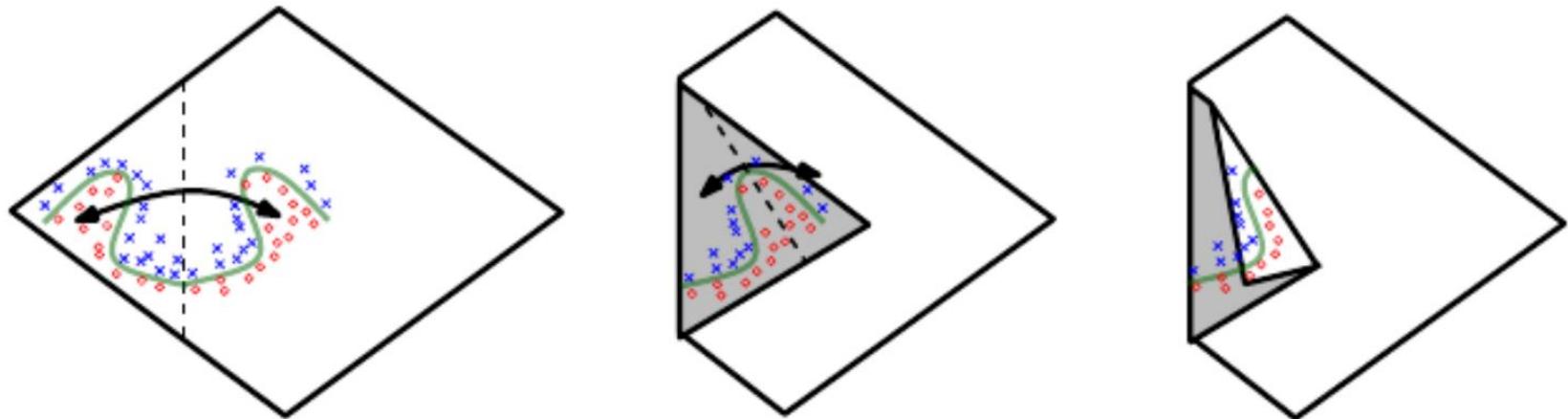
- Universal Approximation Theorem:
 - There is a network large enough to achieve any degree of accuracy
 - but does not say how large the network will be
- Some bounds on size of the single-layer network exist for a broad class of functions
 - But worst case is exponential no. of hidden units
- Example:
 - No. of possible binary functions on vectors $v \in \{0,1\}^n$ is 2^{2^n}
 - Selecting one such function requires 2^n bits which will require $O(2^n)$ degrees of freedom

Function Families and Depth

- Some families of functions can be represented efficiently in a deep network but require much larger networks if the model is shallow
- In some cases no. of hidden units required by shallow model is exponential in n
- Piecewise linear networks (which can be obtained from rectifier nonlinearities or maxout units) can represent functions with a no. of regions that is exponential in d

Advantage of deeper networks

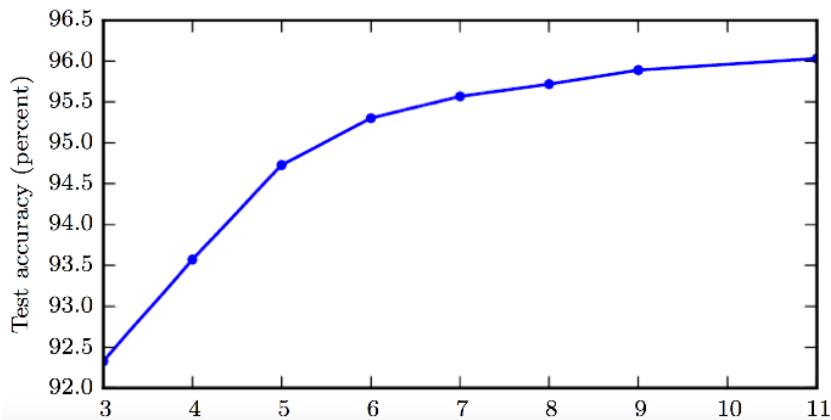
- Absolute value rectification creates mirror images of function computed on top of some hidden unit, wrt the input of that hidden unit.
- Each hidden unit specifies where to fold the input space in order to create mirror responses.
- By composing these folding operations we obtain an exponentially large no. of piecewise linear regions which can capture all kinds of repeating patterns



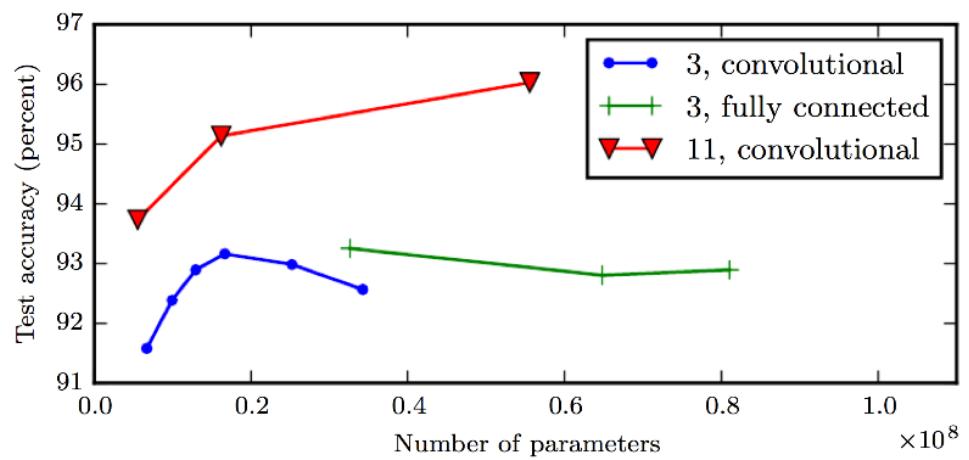
Intuition of Depth

- Any time we choose a ML algorithm we are implicitly stating a set of beliefs about what kind of functions that algorithm should learn
- Choosing a deep model encodes a belief that the function should be a composition several simpler functions
- The learning problem consists of discovering a set of underlying factors of variation that can in turn be described in terms of other, simpler underlying factors of variation.
- Empirically: Deeper Networks perform better ☺

Deeper Networks



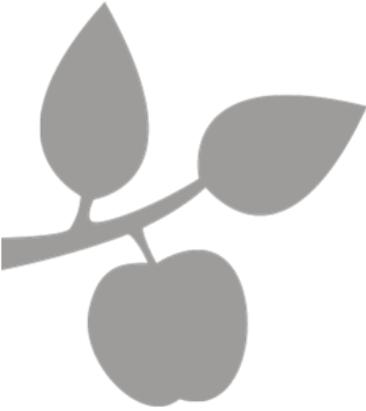
Test accuracy consistently increases with depth



Increasing parameters without increasing depth is not as effective

Other architectural considerations

- Most important specialized architectures are
- Convolutional Networks
 - Used heavily for computer vision
- Recurrent Neural Networks
 - Used for sequence processing
 - Have their own architectural considerations
- Non-chain architectures
 - Skipping going from layer i to layer $i+2$ or higher



DM873

Deep Learning

Spring 2019

Lecture 6 – Gradient Based Learning



Gradient Based Learning

- **Introduction**
- **Loss functions in Deep Learning**
 - Maximum Likelihood Estimation
 - Well Behaved Gradients
- **Stochastic Gradient Descent**

General Setting

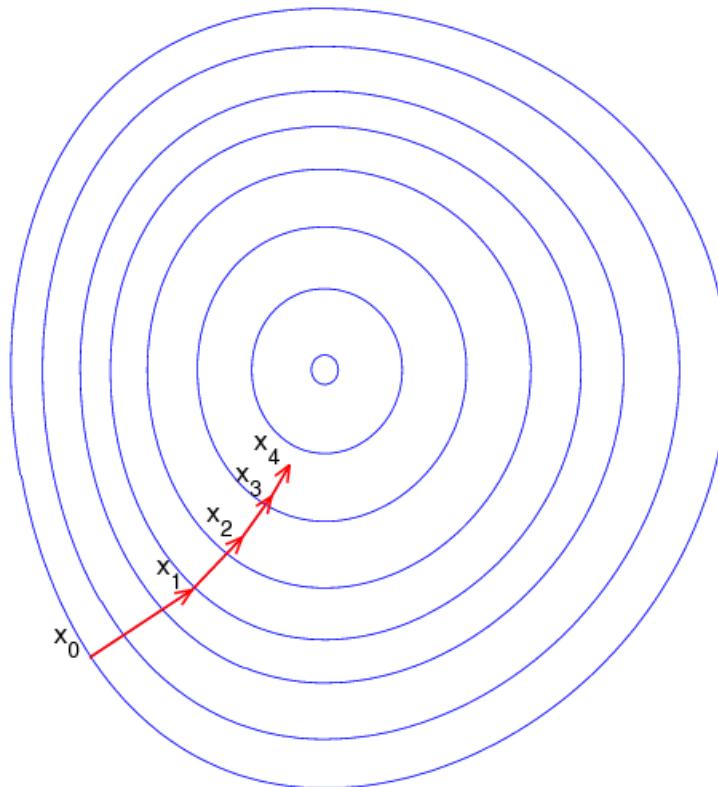
- We use the neural networks to construct a complicated, non-linear function
- We need to have a framework to be able to judge the function we have learned
- Therefore: We associate a cost / goodness to this function and seek to find a minimum/maximum.
 - Remember Logistic Regression:
 - $\text{argmax}_{\theta} \sum_{i=1}^n [y^{(i)} \log h_{\theta}(x) + (1 - y^{(i)}) \log(1 - h_{\theta}(x))]$
- **Two questions:**
 1. **How do we find the minima of such functions?**
 2. **How do we find these functions in the first place?**

Question 1: Gradient Based Learning



The Central Idea

- Update the model parameters following the steepest slope



More Mathematically

- Suppose function $y = f(x)$
- Derivative of function denoted: $f'(x)$ or as dy/dx
 - Derivative $f'(x)$ gives the slope of $f(x)$ at point x
 - It specifies how to scale a small change in input to obtain a corresponding change in the output:

$$f(x + \varepsilon) \approx f(x) + \varepsilon f'(x)$$

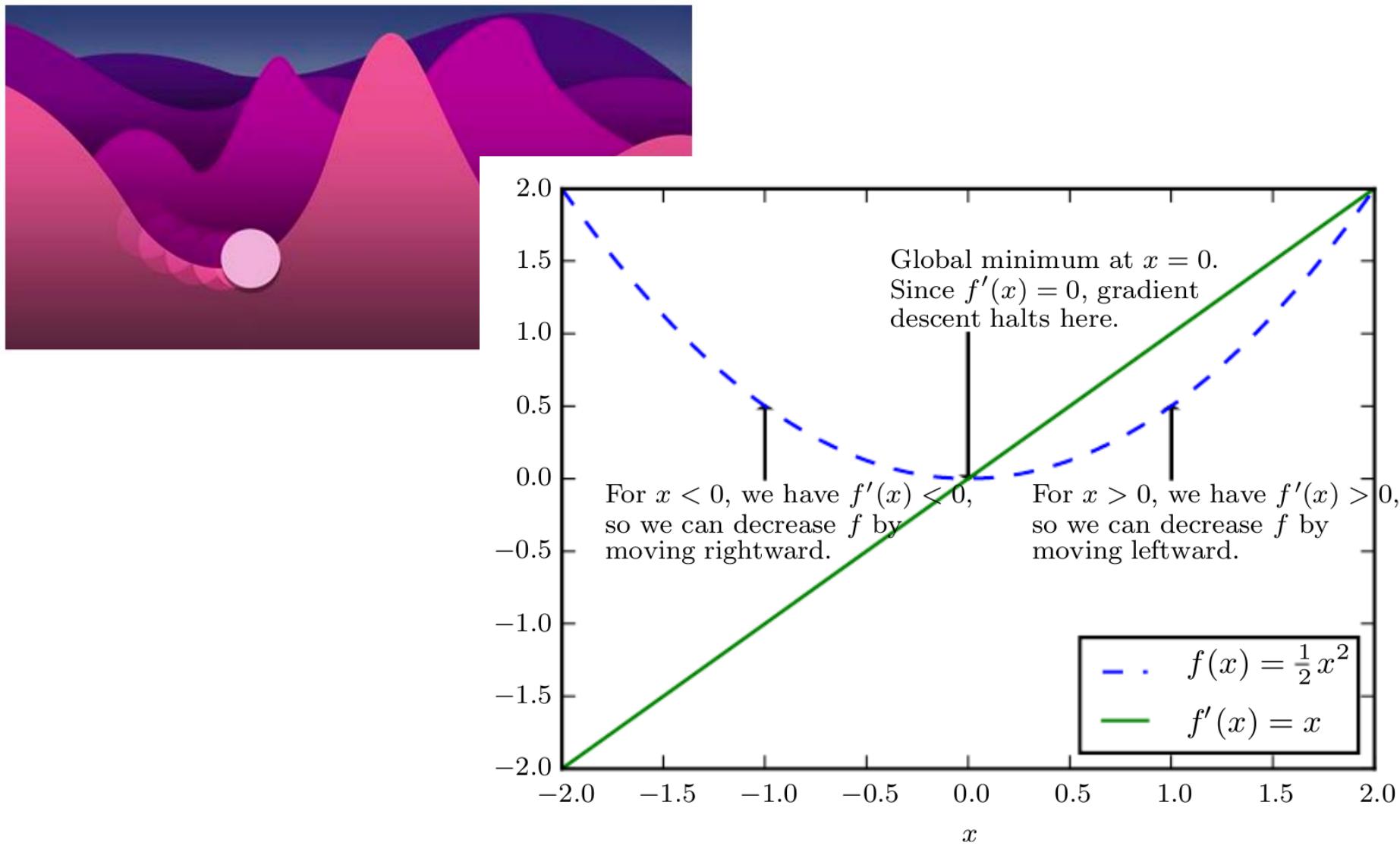
- We know that

$$f(x - \varepsilon \text{sign}(f'(x)))$$

is less than $f(x)$ for small ε .

- Thus we can reduce $f(x)$ by moving x in small steps with opposite sign of derivative
- This technique is called gradient descent (Cauchy 1847)

Usage of Derivates



Functions with multiple inputs

- Need partial derivatives

$$\frac{\partial}{\partial x_i} f(\mathbf{x})$$

- Measures how f changes as only variable x_i increases at point \mathbf{x}
- Gradient is vector containing all of the partial derivatives denoted with $\nabla_{\mathbf{x}} f(\mathbf{x})$
 - Element i of the gradient is the partial derivative of f wrt x_i
 - Critical points are where every element of the gradient is equal to zero
 - A function can be minimized when moving in the direction opposite to the gradient

Functions with multiple inputs

- We can decrease f by moving in the direction of the negative gradient vector
- Steepest descent proposes a new point

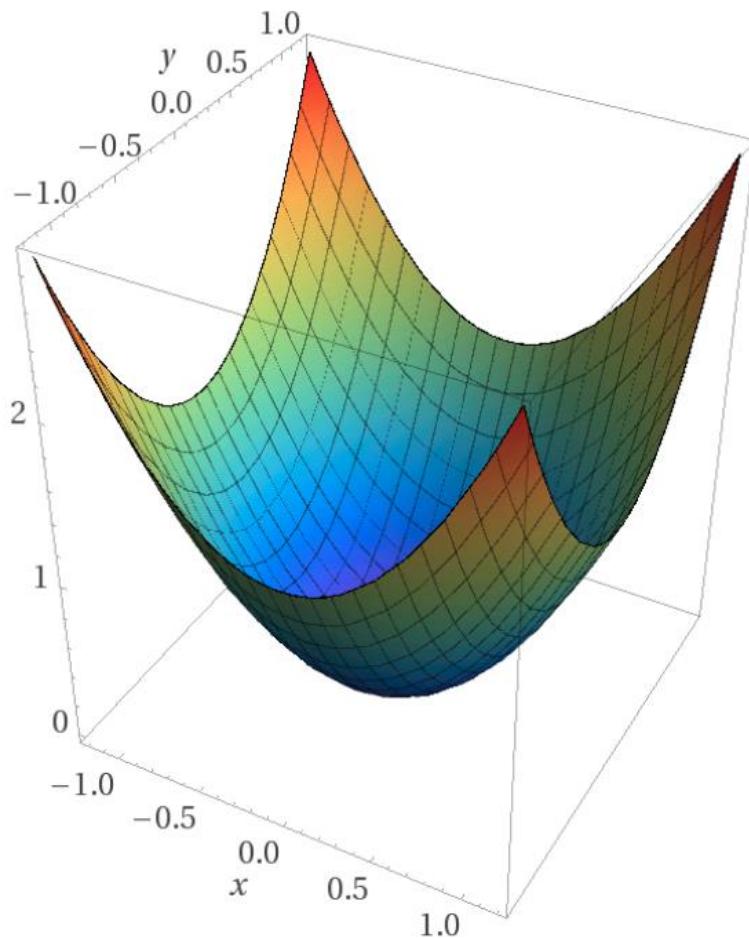
$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x})$$

- With ϵ being the **learning rate** (there are many methods of defining ϵ)
 - The learning rate is crucial ... we will have a lecture on that later on
 - Ascending an objective function of discrete parameters is called hill climbing

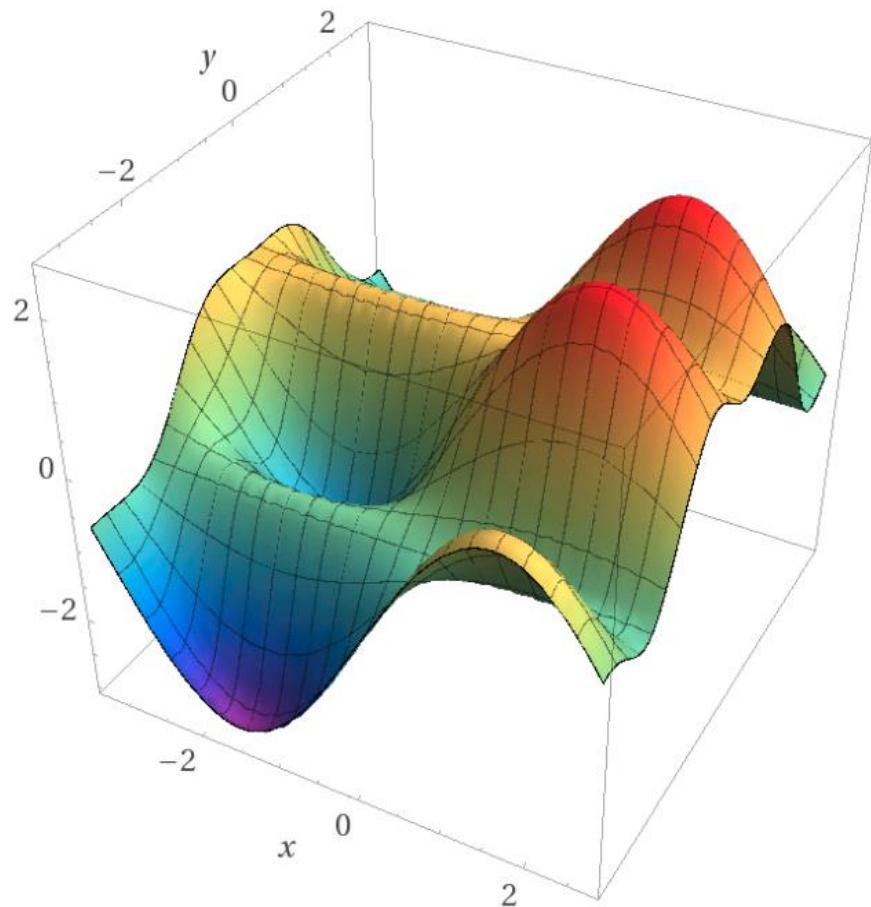
Specialties of Deep Learning

- Neural Network training not different from ML models with gradient descent. The components are needed:
 1. optimization procedure, e.g., gradient descent
 2. cost function (we will see shortly)
 3. model family, e.g., linear with basis functions
- Difference: **nonlinearity causes non-convex loss**
 - Use iterative gradient-based optimizers that merely drives cost to low value
 - No guarantees in comparison to convex optimizations
 - The initialization matters

Convex vs. Non-Convex



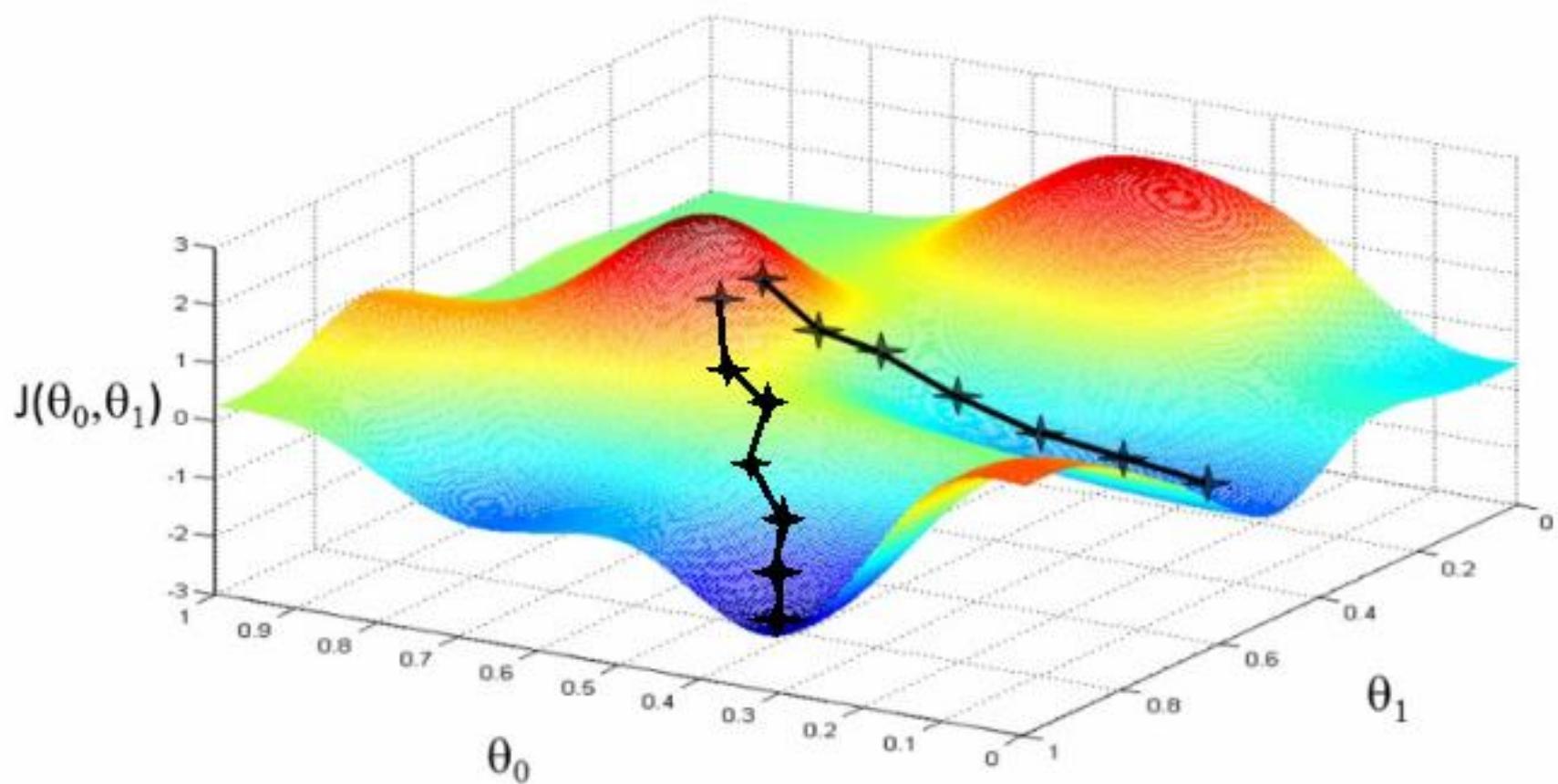
Computed by Wolfram|Alpha



Computed by Wolfram|Alpha

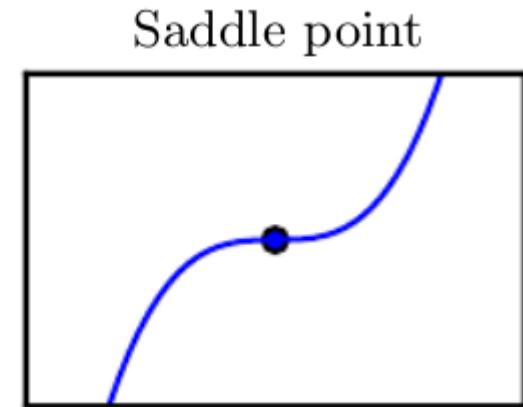
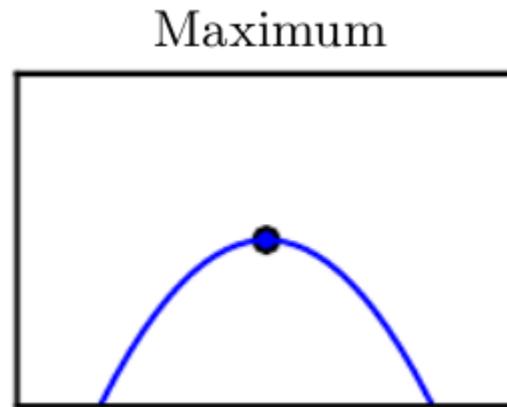
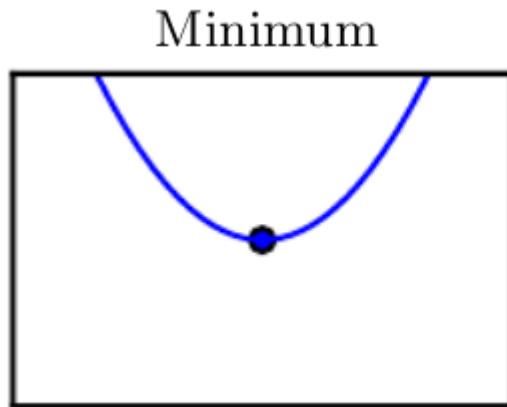
- <https://www.matroid.com/blog/post/the-hard-thing-about-deep-learning>

Problem: We can end-up in local minima

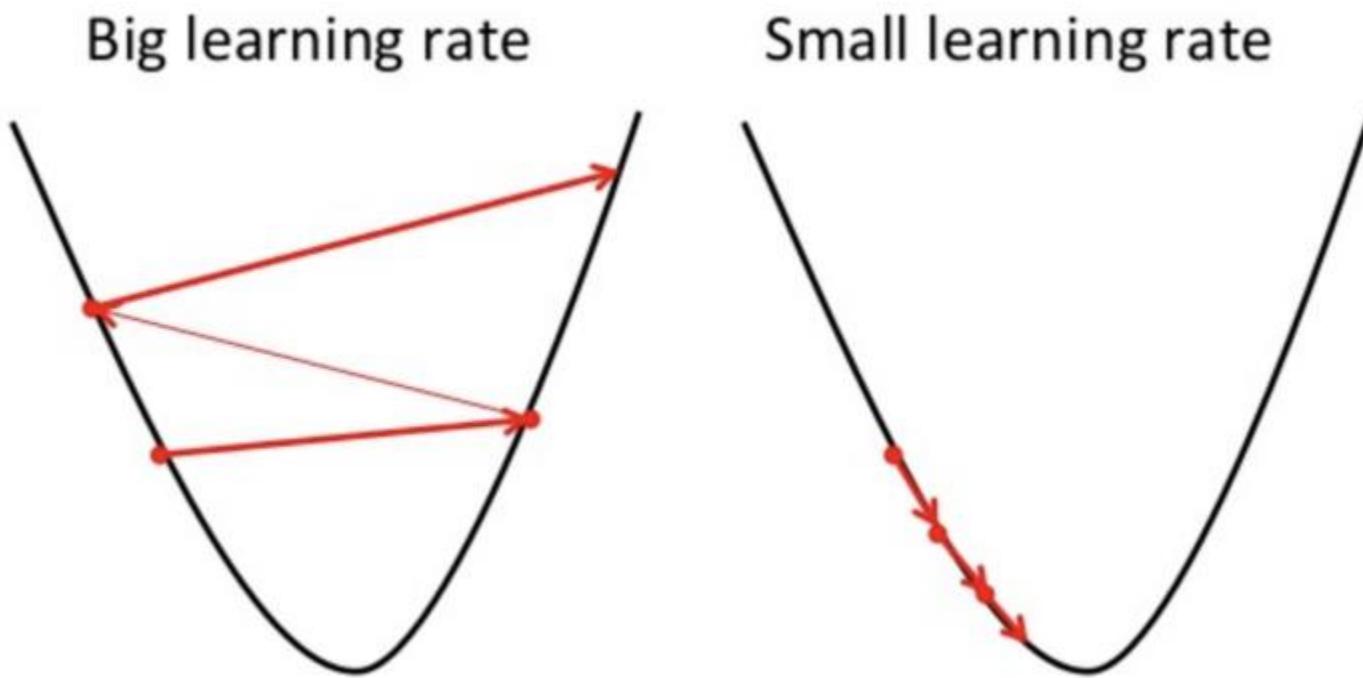


Problem: Stationary points, Local Optima

- When $f'(x) = 0$ derivative provides no information about direction of move
- Points where $f'(x) = 0$ are known as stationary or critical points
 - **Local minimum/maximum**: a point where $f(x)$ lower/ higher than all its neighbors
 - **Saddle Points**: neither maxima nor minima

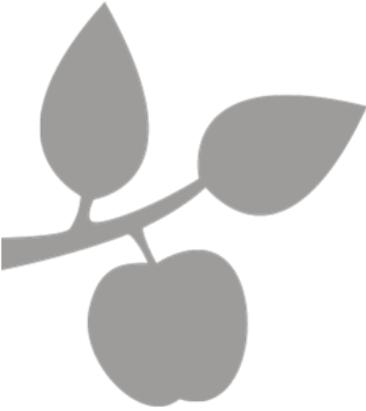


Problem: The Learning Rate



Convergence of Steepest Descent

- Steepest descent converges when every element of the gradient is zero
- Pure math way of life:
 - Find literally the smallest value of $f(x)$
 - Or maybe: find some critical point of $f(x)$ where the value is locally smallest
- Deep learning way of life:
 - Decrease the value of $f(x)$ a lot
 - But we have a highly non-convex problem (because of the activation functions) => No guarantees!



Gradient Based Learning

- Introduction
- Loss functions in Deep Learning
 - Maximum Likelihood Estimation
 - Well Behaved Gradients
- Stochastic Gradient Descent

Training Feedforward Networks

- We have seen how to construct a FNN
- We can input a data point into the FNN and receive a prediction
- **We now need to define a function which judges the quality of our predictions and allows us to optimize the network, i.e., train the network.**

Training Feedforward Networks

- We already two such error functions:

- Mean-Squared-Error (minimize):

$$J(\theta) = \frac{1}{n} \sum_i^n \|y^{(i)} - \hat{y}^{(i)}\|^2$$

- For Logistic Regression we have seen the MLE (maximize)

$$L(X, \boldsymbol{\theta}) = \prod_{i=1}^n p(y^{(i)} | \mathbf{x}^{(i)}; \boldsymbol{\theta})$$

Cross Entropy Loss

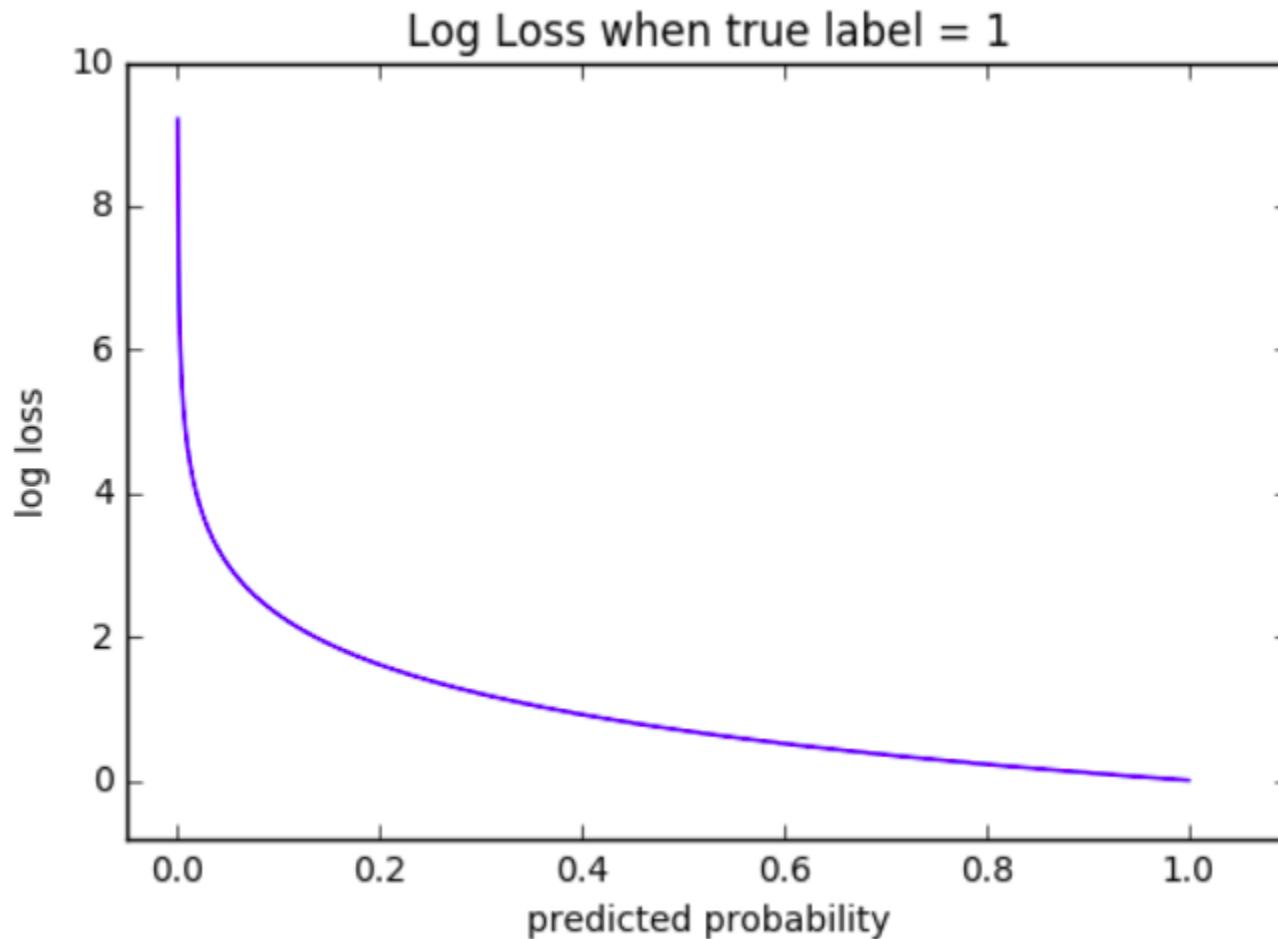
- For Neural Networks, we usually us the cross-entropy loss
- Minimizing the cross-entropy loss corresponds to maximize the log likelihood:

$$J(\boldsymbol{\theta}) = -\mathbb{E}[p(y|x; \boldsymbol{\theta})]$$

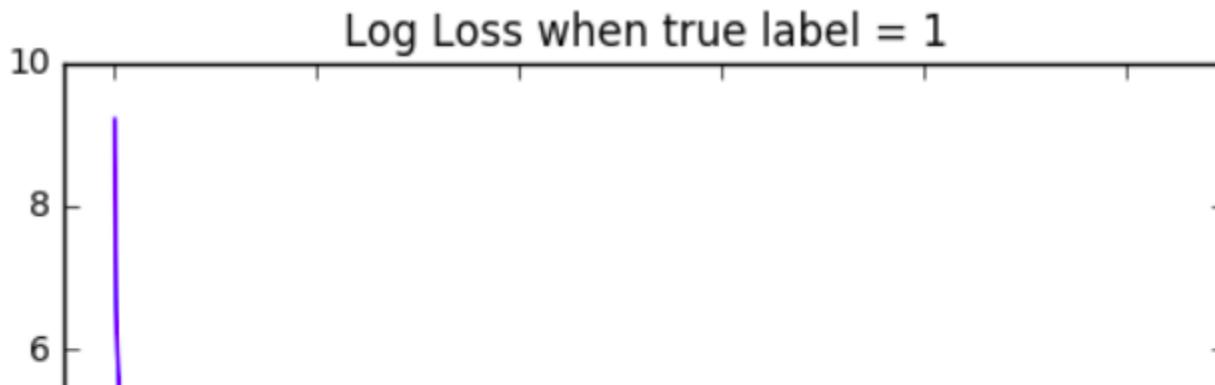
- In case of our logistic regression:

$$-\frac{1}{n} \sum_{i=1}^n [y^{(i)} \log h_{\boldsymbol{\theta}}(x) + (1 - y^{(i)}) \log(1 - h_{\boldsymbol{\theta}}(x))]$$

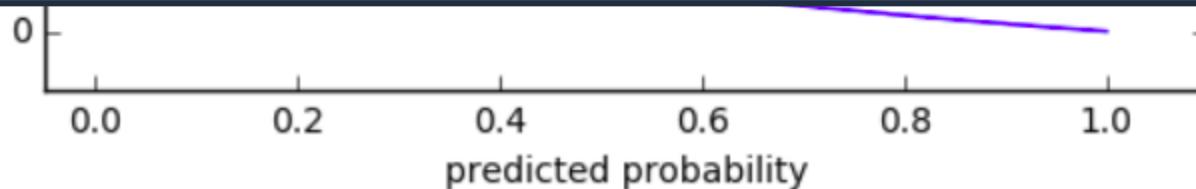
Cross Entropy Loss



Cross Entropy Loss



But How to find such loss functions?



Back to Statistical Learning

- To fully understand the origins of the most common loss functions, we have to resort back to the basics of statistical learning
- The key assumption is that our dataset $X = \{x^{(1)}, \dots, x^{(m)}\}$ is drawn independently from the true but unknown data generating distribution p_{data}
- Now, let $p_{model}(x; \theta)$ be a parametric family of probability distributions

The Maximum Likelihood Estimator

- The maximum likelihood estimator for θ is then defined as

$$\begin{aligned}\boldsymbol{\theta}_{ML} &= \operatorname{argmax}_{\boldsymbol{\theta}} p_{model}(X; \boldsymbol{\theta}) \\ &= \operatorname{argmax}_{\boldsymbol{\theta}} \prod_{i=1}^m p_{model}(\mathbf{x}^{(i)}; \boldsymbol{\theta})\end{aligned}$$

- The argmax function is invariant to rescaling, thus

$$\begin{aligned}\boldsymbol{\theta}_{ML} &= \operatorname{argmax}_{\boldsymbol{\theta}} \sum_i^m \log p_{model}(\mathbf{x}^{(i)}; \boldsymbol{\theta}) \\ &= \operatorname{argmax}_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x} \sim \hat{p}_{data}} \log p_{model}(\mathbf{x}; \boldsymbol{\theta})\end{aligned}$$

The Maximum Likelihood Estimator

- To interpret maximum likelihood estimation is to view it as minimizing the dissimilarity between the empirical distribution \hat{p}_{data} (defined by the training) and the model with Kullback-Leibler (KL) distance.

$$D_{KL}(\hat{p}_{data}, p_{model}) = \mathbb{E}_{x \sim \hat{p}_{data}} [\log \hat{p}_{data} - \log p_{model}(x; \theta)]$$

- Since \hat{p}_{data} is fixed, the MLE is equivalent to
$$-\mathbb{E}_{x \sim \hat{p}_{data}} [\log p_{model}(x; \theta)]$$
- Minimizing KL is the same as minimizing the **cross-entropy**

Conditional Log-Likelihood

- The log-likelihood estimator can easily be extended to cases where we try to estimate the conditional probability

$$\boldsymbol{\theta}_{ML} = \operatorname{argmax}_{\boldsymbol{\theta}} P(\mathbf{Y}|\mathbf{X}; \boldsymbol{\theta})$$

- Given that examples are i.i.d., then this can be decomposed to

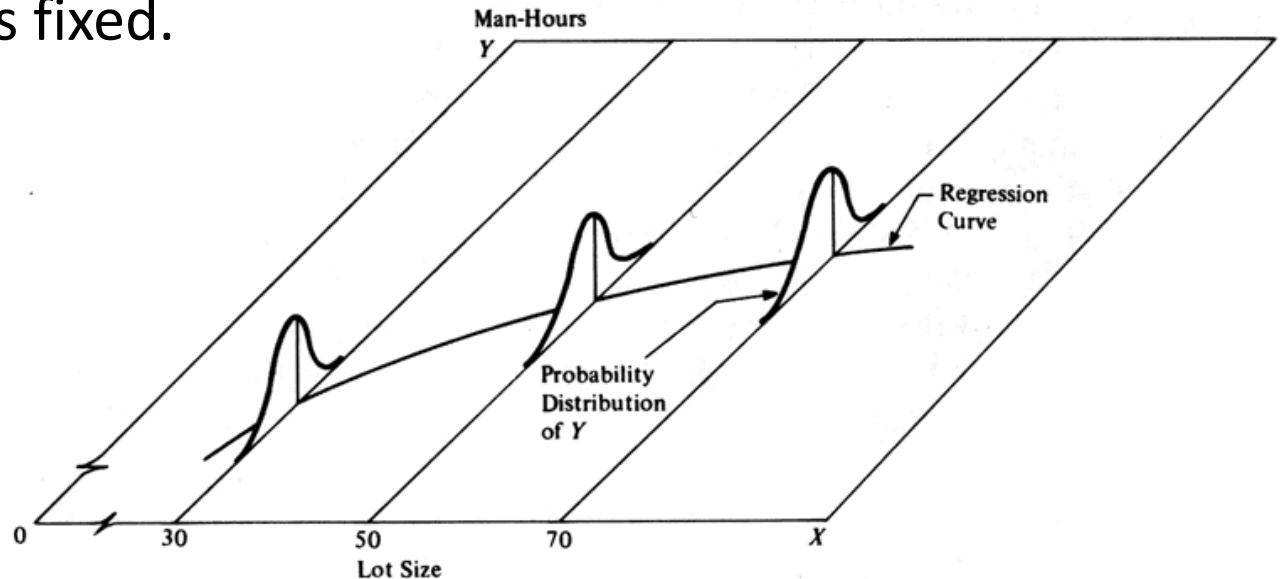
$$\boldsymbol{\theta}_{ML} = \operatorname{argmax}_{\boldsymbol{\theta}} \sum_i^m \log P(\mathbf{y}^{(i)} | \mathbf{x}^{(i)}; \boldsymbol{\theta})$$

Example: Linear Regression

- We now think of the model as producing a conditional distribution

$$p(y|x) = N(y; \hat{y}(x; w), \sigma^2)$$

- with $\hat{y}(x; w)$ being the linear regression predicting the mean of the gaussian, σ^2 is fixed.



source: Neter, Wasserman & Kutner (1983) *Applied Linear Regression Models*

Example: Linear Regression

- Let's see when we follow through:

$$N(x; \mu, \sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right)$$

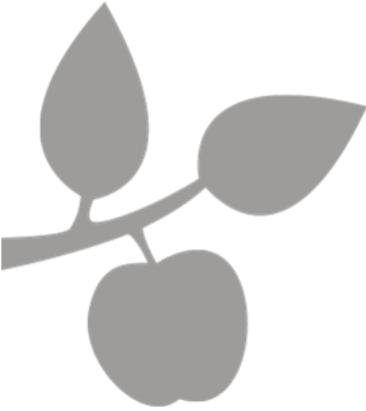
- Then the conditional log-likelihood is

$$\sum_i^m \log P(y^{(i)} | x^{(i)}; \theta) = -m \log \sigma - \frac{m}{2} \log(2\pi) - \sum_i^m \frac{\|\hat{y}^{(i)} - y^{(i)}\|^2}{2\sigma^2}$$

- Minimizing the negative log-likelihood is then the same as minimizing the MSE.

About the Gradient

- Gradient must be large and predictable enough to serve as good guide to the learning algorithm
- Functions that **saturate** (become very flat) undermine this
 - Because the gradient becomes very small
 - Happens when activation functions producing output of hidden/output units saturate
- **Negative log-likelihood** helps avoid saturation problem for many models
 - Many output units involve exp functions that saturate when its argument is very negative
 - Log function in Negative log likelihood cost function undoes exp of some units



Gradient Based Learning

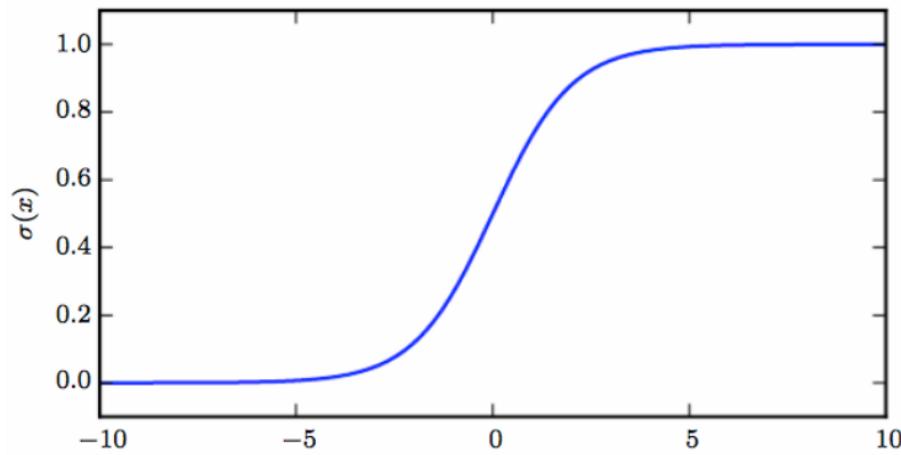
- Introduction
- Loss functions in Deep Learning
 - Maximum Likelihood Estimation
 - Well Behaved Gradients
- Stochastic Gradient Descent

Revisit: Sigmoid Units

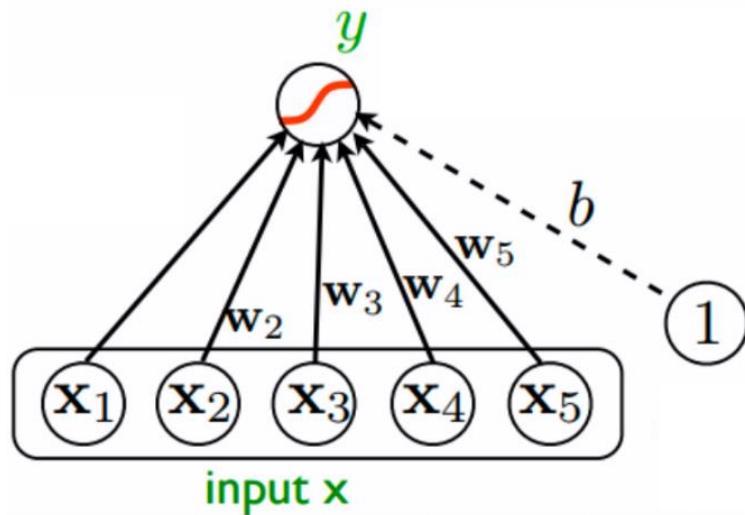
- Sigmoid always gives a gradient
$$\hat{y} = \sigma(\mathbf{w}^T \mathbf{h} + b)$$

with

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



- Sigmoid Unit has two components:
 - A linear layer to compute
$$z = \mathbf{w}^T \mathbf{h} + b$$
 - A sigmoid activation function to convert z into a probability



Cross-Entropy

- Using sigmoid directly:

$$p(y|\boldsymbol{\theta}) = \prod_{n=1}^N \sigma(\boldsymbol{\theta}^T \mathbf{x}_n)^{y_n} \cdot (1 - \sigma(\boldsymbol{\theta}^T \mathbf{x}_n))^{1-y_n}$$

- With the loss function:

$$J(\boldsymbol{\theta}) = -\ln p(y|\boldsymbol{\theta}) = -\sum_{n=1}^N (y_n \ln(\sigma(\boldsymbol{\theta}^T \mathbf{x}_n)) + (1 - y_n) \ln(1 - \sigma(\boldsymbol{\theta}^T \mathbf{x}_n)))$$

- Problem:

- Sigmoid saturates to 0 or 1 when z becomes very negative/positive
- Gradient can shrink to too small to be useful for learning, whether model has correct or incorrect answer

Different Formulation

- At the end of the day we need to produce a number which represents $\hat{y} = P(y = 1|x)$; thus lies between 0 and 1
- Therefore, let's predict a number

$$z = \log \tilde{P}(y = 1|x)$$

with \tilde{P} being an unnormalized random distribution:

$$\begin{aligned}\log \tilde{P}(y) &= yz \\ \tilde{P}(y) &= \exp(yz)\end{aligned}$$

- Normalizing leads to a real distribution:

$$\begin{aligned}P(y) &= \frac{\exp(yz)}{\sum_{y'=\{0,1\}} \exp(y'z)} \\ P(y) &= \sigma((2y - 1)z)\end{aligned}$$

Different Formulation

- At the end of the day we need to produce a number which represents $\hat{y} = P(y = 1|x)$; thus lies between 0 and 1
- Therefore, let's predict a number

$$z = \log \tilde{P}(y = 1|x)$$

with \tilde{P} being an unnormalized random distribution:

$$\log \tilde{P}(y) = yz$$

$$\tilde{P}(y) = \exp(yz)$$

- Normalizing leads to a real distribution:

$$P(y) = \frac{\exp(yz)}{\sum_{y'=\{0,1\}} \exp(y'z)}$$

$$P(y) = \sigma((2y - 1)z)$$

$$\begin{aligned}\log \tilde{P}(y = 1) &= z \\ \log \tilde{P}(y = 0) &= 0\end{aligned}$$

$$\begin{aligned}\tilde{P}(y = 1) &= e^z \\ \tilde{P}(y = 0) &= e^0 = 1\end{aligned}$$

$$\begin{aligned}P(y = 1) &= \frac{e^z}{1 + e^z} \\ P(y = 0) &= \frac{1}{1 + e^z}\end{aligned}$$

$$\left\{ \begin{array}{ll} \sigma(z) = \frac{1}{1 + e^{-z}} = \frac{e^z}{1 + e^z} & \text{for } y = 1 \\ \sigma(-z) = \frac{1}{1 + e^z} & \text{for } y = 0 \end{array} \right.$$

Sigmoid with Softplus

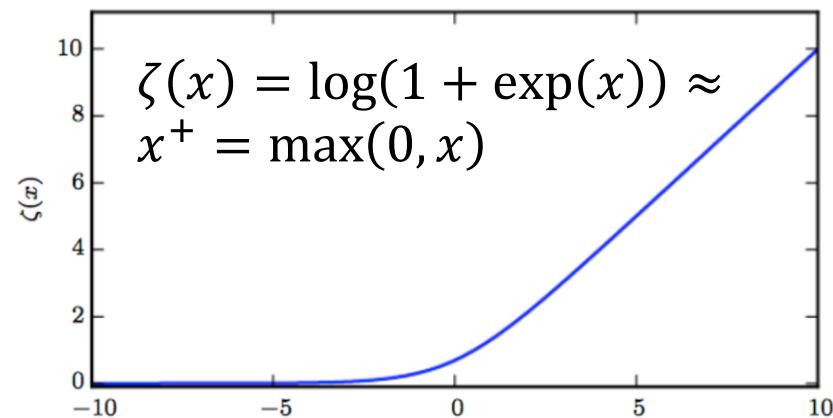
- Using

$$P(y|z) = \sigma((2y - 1)z)$$

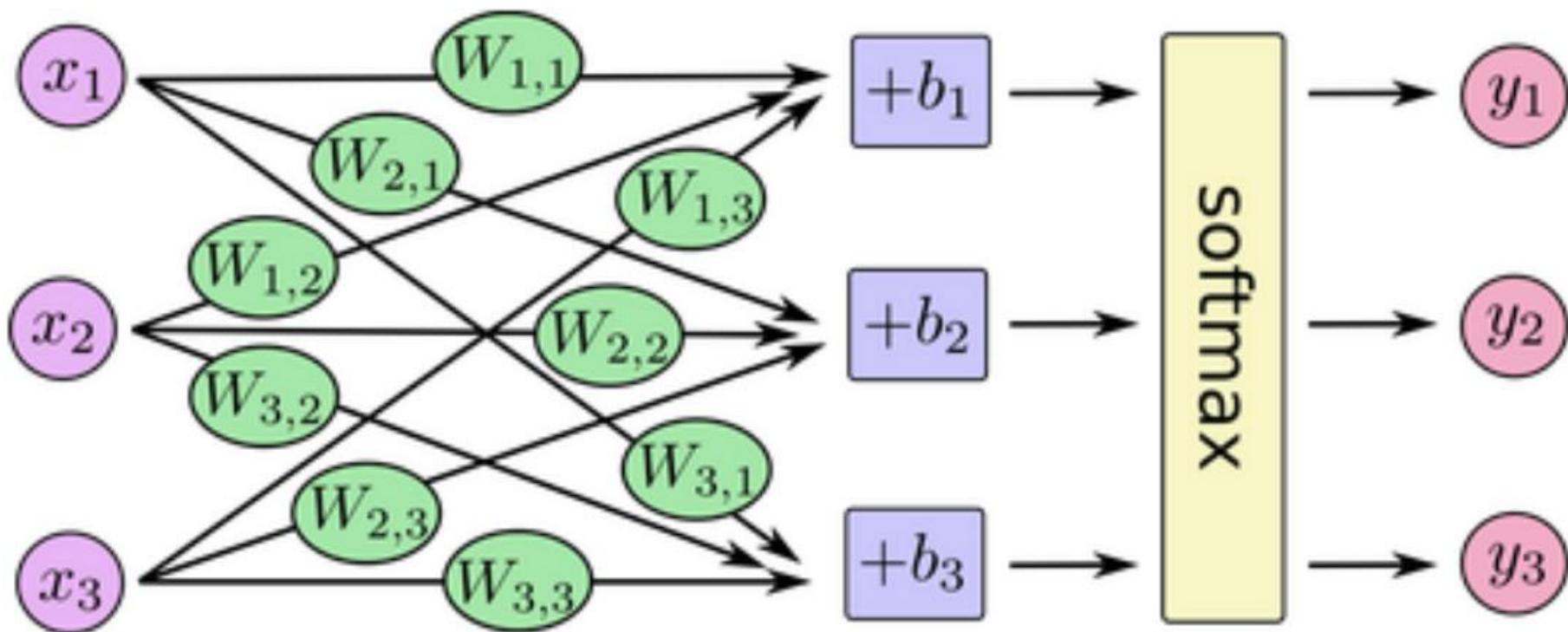
- The Loss function can be defined as:

$$J(\theta) = -\log P(y|x) = -\log \sigma((2y - 1)z) = \zeta((1 - 2y)z)$$

- With ζ being the softplus function
- only when saturates when $(1 - 2y)z$ is very negative
- That only happens, when the model already has the right answer



Revisit the Softmax Function



Log-likelihood of the Softmax Function

- As with the sigmoid, a linear layer produces the unnormalized log probabilities for each class:

$$\mathbf{z} = \mathbf{W}^T \mathbf{h} + \mathbf{b}$$

- Normalization leads to the softmax function:

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Log-likelihood of the Softmax Function

- As a general rule of thumb, exponentiations should be undone by the cost function
- Thus, the log-likelihood is a natural candidate for the loss function

$$\log \text{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j)$$

- Input z_i always has a direct contribution and can never saturate
 - This first term encourages z_i being pushed up
 - The second term encourages entire \mathbf{z} being pushed down

Second Term

- We consider the term

$$\log \sum_j \exp(z_j)$$

- Can be approximated to $\max_j z_j$ (when largest z_j is significantly larger than all other z_k)
- Costs penalizes most active incorrect prediction
- In case of a correct prediction, then z_i and $\log \sum_j \exp(z_j)$ will roughly cancel out.
- Thus, a correctly classified example will only little contribute to the overall costs

More about the Softmax Function

- Overall, unregularized maximum likelihood will drive the model to learn parameters that drive the softmax to predict the fraction of counts of each outcome observed in the training set

$$\text{softmax}(\mathbf{z}(\mathbf{x}; \boldsymbol{\theta}))_i \approx \frac{\sum_{j=1}^m \mathbf{1}_{y^{(j)}=i, \mathbf{x}^{(j)}=\mathbf{x}}}{\sum_{j=1}^m \mathbf{1}_{\mathbf{x}^{(j)}=\mathbf{x}}}$$

- Objective functions that do not use a log to undo the exp of softmax fail to learn when argument of exp becomes very negative, causing gradient to vanish
- Especially the squared error is a poor loss function for softmax units

Other Output Types

- Linear, Sigmoid and Softmax output units are the most common
- Neural networks can generalize to any kind of output layer
- Principle of maximum likelihood provides a guide for how to design a good cost function for any output layer
- If we define conditional distribution $p(y|x; \theta)$, principle of maximum likelihood suggests we use $-\log p(y|x; \theta)$ for our cost function



Gradient Based Learning

- **Introduction**
- **Loss functions in Deep Learning**
 - Maximum Likelihood Estimation
 - Well Behaved Gradients
- **Stochastic Gradient Descent**

Stochastic Gradient Descent (SGD)

- A recurring problem in machine learning:
 - large training sets are necessary for good generalization
 - but large training sets are also computationally expensive
- Nearly all deep learning is powered by one very important algorithm: **Stochastic Gradient Descent**

Function Principle of SGD

- Insight: The gradient for each sample can be regarded as a random variable with the **expectation value** being the gradient
- Expectations can be approximated using small set of samples
- In each step of SGD we sample a minibatch of examples

$$B = \{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m')}\}$$

- drawn uniformly from the training set
- Minibatch size m' is typically chosen to be small: 1 to a hundred
- Crucially m' is held fixed even if sample set is in billions
- We may fit a training set with billions of examples using updates computed on only a hundred examples

SGD Estimate on minibatch

- Estimate of gradient is formed as

$$\mathbf{g} = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

using only the examples of the minibatch

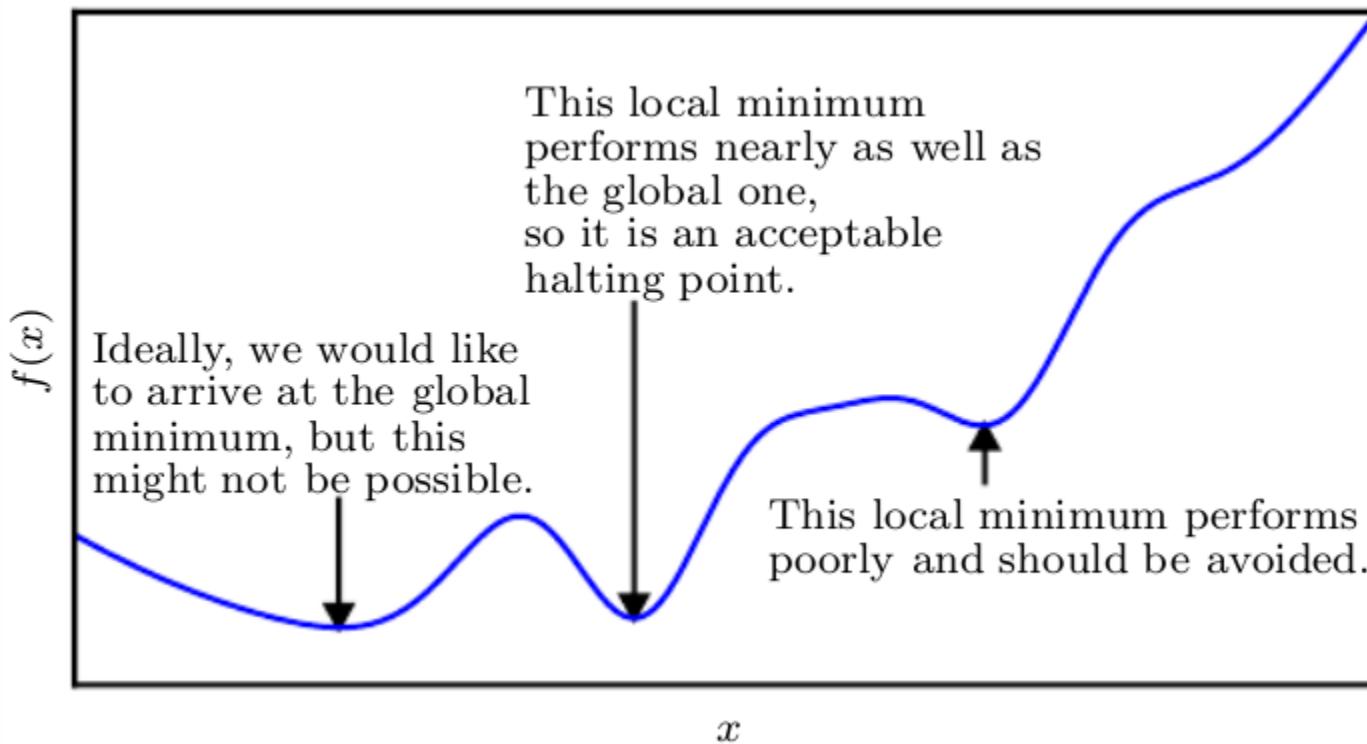
- SDG then simply follows the estimated gradient downhill

$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \epsilon \mathbf{g}$$

How good is SGD?

- In the past gradient descent was regarded as slow and unreliable
- Application of gradient descent to non-convex optimization problems was regarded as unprincipled
- SGD is not guaranteed to arrive at even a local minimum in reasonable time
- **But it often finds a very low value of the cost function quickly enough**
- As $m \rightarrow \infty$ the model will eventually converge to its best possible test error before SGD has sampled every example

Good Enough in Practice



Quality of sampling-based estimate

- Standard error for mean from n samples is

$$\frac{\sigma}{\sqrt{n}}$$

- where σ is the standard dev of samples
- Denominator shows that error decreases less than linearly with no. of samples
 - Ex: 100 samples vs 10,000 samples
 - Computation increases by a factor of 100 but
 - Error decreases by only a factor of 10
- Optimization algorithms converge much faster
 - if allowed to rapidly compute approximate estimates
 - rather than slowly compute exact gradient

A motivation for sampling: Redundancy

- Training set may be redundant
- Worst case: all m examples are identical
 - Sampling based estimate could use m times less computation
- In practice
 - unlikely to find worst case situation but
 - likely to find large no. of examples that all make similar contribution to gradient

Batch gradient vs. Stochastic methods

- Batch or deterministic gradient methods:
 - Optimization methods that use all training samples in a large batch
 - Somewhat confusing terminology
 - Batch also used to describe minibatch used by minibatch stochastic gradient descent
 - Batch gradient descent implies use of full training set
 - Batch size refers the size of a minibatch
- Those using a single sample are called stochastic or on-line
 - On-line typically means continually created samples, rather than multiple passes over a fixed size training set
 - Deep learning algorithms use more than 1 but fewer than all
 - Traditionally called minibatch or minibatch stochastic or simply stochastic

Minibatch Size

- Driven by following factors
 - Larger batches -> more accurate gradient but with less than linear returns
 - Multicore architectures are underutilized by extremely small batches
 - Use at least some minimum size below which there is no reduction in time to process a minibatch
 - If all examples processed in parallel, amount of memory scales with batch size
 - This is a limiting factor in batch size
 - GPU architectures more efficient with power of 2
 - Range from 32 to 256, sometimes with 16 for large models

Regularizing effect of small batches

- Small batches offer regularizing effect due to noise added in process
- Generalization is best for batch size of 1
- Small batch sizes require small learning rate
 - To maintain stability due to high variance in estimate of gradient
- Total run time can be high
 - Due to reduced learning rate and
 - Requires more time to observe entire training set

Use of minibatch information

- Different algorithms use different information from the minibatch
- Some algorithms more sensitive to sampling error
- Algorithms using only the gradient \mathbf{g} are robust and can handle smaller batch sizes below 100
- Second order methods using Hessian \mathbf{H} and compute updates such as $\mathbf{H}^{-1}\mathbf{g}$ require much larger batch sizes like 10,000

Random selection of minibatches

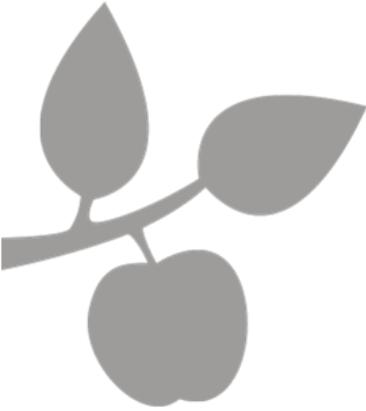
- Crucial to select minibatches randomly
- Computing expected gradient from a set of samples requires that sample independence
- Many data sets are arranged with successive samples highly correlated
 - E.g., blood sample data set has five samples for each patient
- Necessary to shuffle the samples
 - For a data set with billions of samples shuffle once and store it in shuffled fashion

Use of multiple epochs

- SGD minimizes generalization error when samples are not reused
- Yet best to make several passes through the training set
 - Unless training set is extremely large
- With multiple epochs, first epoch follows unbiased gradient of generalization error
- Additional epochs provide enough benefit to decrease training error
 - Although might start increasing gap between training and testing error

Impact of growing data sets

- Data sets are growing more rapidly than computing power
- More common to use each training example only once
- Or even make an incomplete pass through the data set
- With a large training set overfit is not an issue
 - Underfitting and computational efficiency become predominant concerns

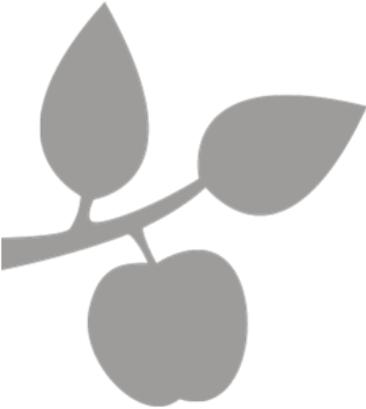


DM873

Deep Learning

Spring 2019

Lecture 7 – Backpropagation



Backpropagation

- **Function Principle**
- **Generalization to Vectors**

Chain Rule of Calculus

- If g is differentiable at x and f is differentiable at $g(x)$, then the composite function $F = f \circ g$ defined by $F(x) = f(g(x))$ is differentiable at x and F' is given by

$$F' = f'(g(x)) \cdot g'(x)$$

- In Leibnitz notation, if $y = f(u)$ and $u = g(x)$, then

$$\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx}$$

Forward vs. Backward Mode

■ Forward Accumulation:

- One first fixes the **independent variable** with respect to which differentiation is performed and computes the derivative of each sub-expression recursively

$$\frac{dy}{dx} = \frac{dy}{dw_{n-1}} \frac{dw_{i-1}}{dx} = \frac{dy}{dw_{n-1}} \left(\frac{dw_{n-1}}{dw_{n-2}} \frac{dw_{i-2}}{dx} \right) = \dots$$

■ Backward Accumulation:

- One first fixes the **dependent variable** to be differentiated and computes the derivative with respect to each sub-expression recursively

$$\frac{dy}{dx} = \frac{dy}{dw_1} \frac{dw_1}{dx} = \left(\frac{dy}{dw_2} \frac{dw_2}{dw_1} \right) \frac{dw_1}{dx} = \dots$$

Forward Accumulation Example

$$\begin{aligned} z &= f(x_1, x_2) \\ &= x_1 x_2 + \sin x_1 \\ &= w_1 w_2 + \sin w_1 \\ &= w_3 + w_4 \\ &= w_5 \end{aligned}$$

The choice of the independent variable defines the used seed. For example, we want to differentiate with respect to x_1 :

$$\dot{w}_1 = \frac{dx_1}{dx_1} = 1 \text{ and } \dot{w}_2 = \frac{dx_2}{dx_1} = 0$$

$$\frac{df}{dx_1} :$$

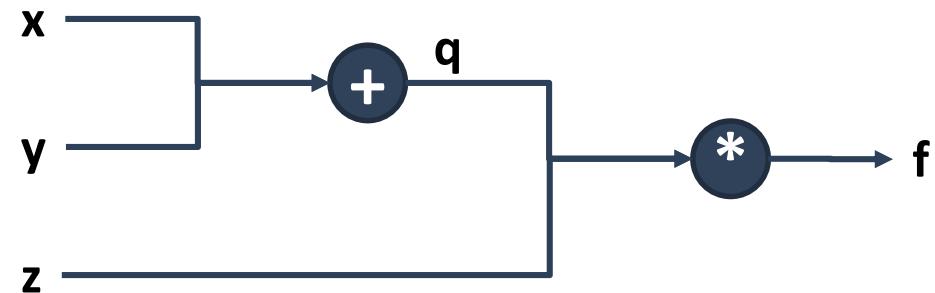
Compute Value	Compute derivative
$w_1 = x_1$	$\dot{w}_1 = 1$
$w_2 = x_2$	$\dot{w}_2 = 0$
$w_3 = w_1 \cdot w_2$	$\dot{w}_3 = w_2 \cdot \dot{w}_1 + w_1 \cdot \dot{w}_2$
$w_4 = \sin w_1$	$\dot{w}_4 = \cos w_1 \cdot \dot{w}_1$
$w_5 = w_3 + w_4$	$\dot{w}_5 = \dot{w}_3 + \dot{w}_4$

Observations

- In order to also get a derivative $\frac{df}{dx_2}$ another run would be required to receive the gradient
- Forward accumulation is good for functions
$$f: \mathbb{R}^n \rightarrow \mathbb{R}^m$$
with $m \gg n$
- In deep learning: Normally millions of weights (n) to optimize with only one output ($m = 1$), the costs
 - => Therefore, Backward accumulation, or Backpropagation

Backpropagation: A Simple Example

$$f(x, y, z) = (x + y)z$$

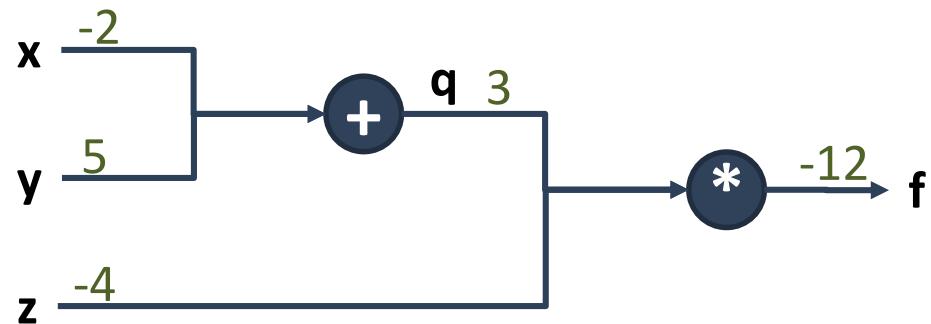


Backpropagation: A Simple Example

$$f(x, y, z) = (x + y)z$$

- With

- $x = -2$
- $y = 5$
- $z = -4$

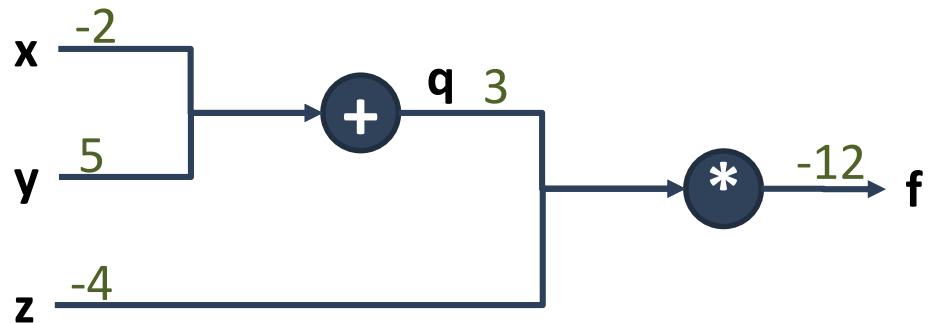


Backpropagation: A Simple Example

$$f(x, y, z) = (x + y)z$$

- With

- $x = -2$
- $y = 5$
- $z = -4$



$$q = x + y; \quad \frac{\partial q}{\partial x} = 1; \quad \frac{\partial q}{\partial y} = 1;$$

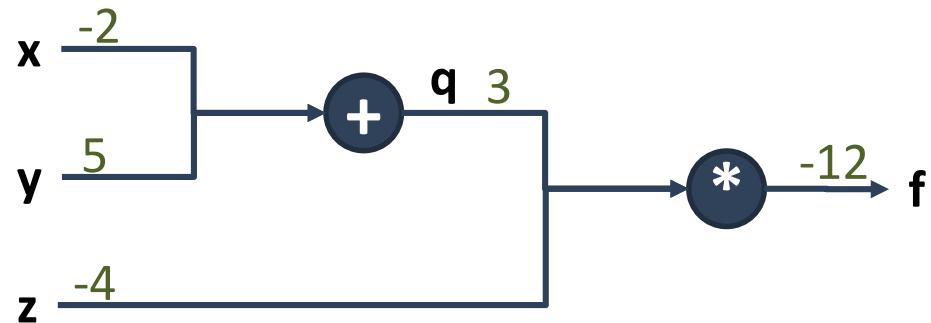
$$f = qz; \quad \frac{\partial f}{\partial q} = z; \quad \frac{\partial f}{\partial z} = q;$$

Backpropagation: A Simple Example

$$f(x, y, z) = (x + y)z$$

- With

- $x = -2$
- $y = 5$
- $z = -4$



$$q = x + y; \quad \frac{\partial q}{\partial x} = 1; \quad \frac{\partial q}{\partial y} = 1;$$

$$f = qz; \quad \frac{\partial f}{\partial q} = z; \quad \frac{\partial f}{\partial z} = q;$$

Looking for:

$$\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y} \quad \frac{\partial f}{\partial z}$$

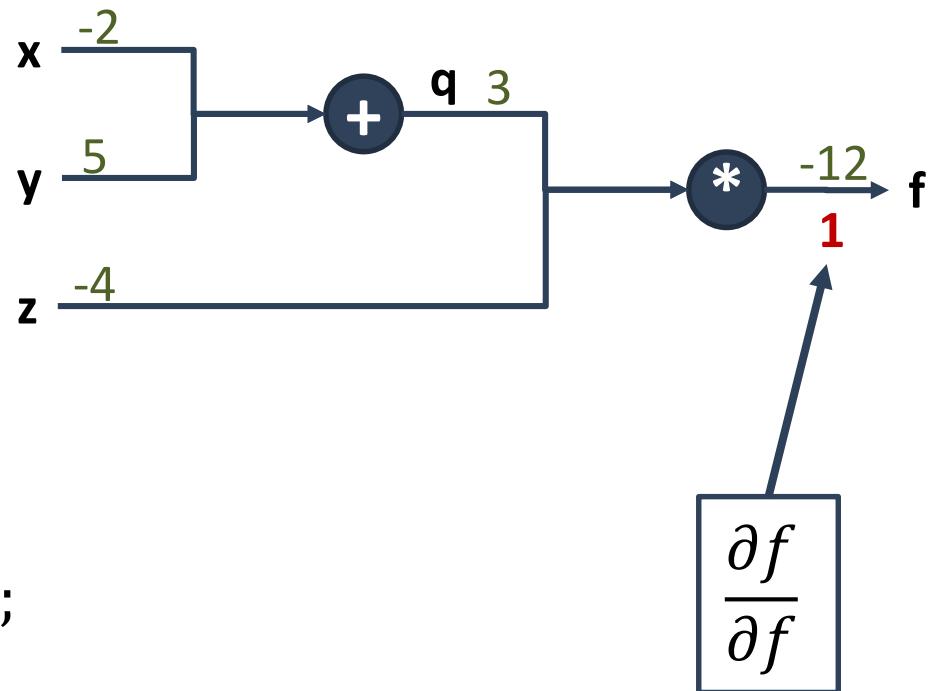
Backpropagation: A Simple Example

$$f(x, y, z) = (x + y)z$$

- With

- $x = -2$
- $y = 5$
- $z = -4$

$$q = x + y; \quad \frac{\partial q}{\partial x} = 1; \quad \frac{\partial q}{\partial y} = 1;$$



$$f = qz; \quad \frac{\partial f}{\partial q} = z; \quad \frac{\partial f}{\partial z} = q;$$

Looking for:

$$\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y} \quad \frac{\partial f}{\partial z}$$

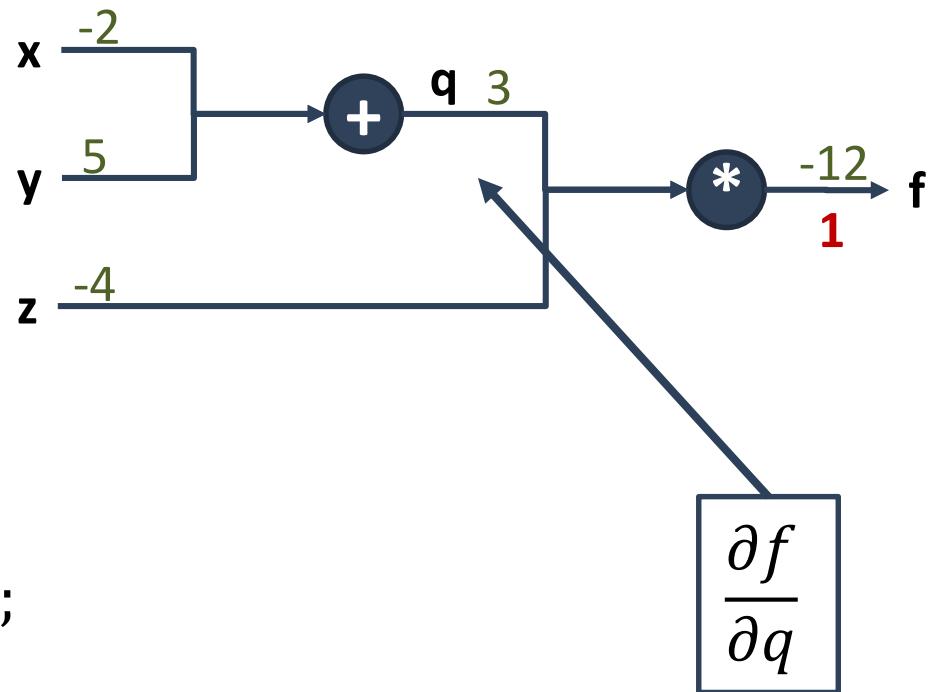
Backpropagation: A Simple Example

$$f(x, y, z) = (x + y)z$$

- With

- $x = -2$
- $y = 5$
- $z = -4$

$$q = x + y; \quad \frac{\partial q}{\partial x} = 1; \quad \frac{\partial q}{\partial y} = 1;$$



$$f = qz; \quad \frac{\partial f}{\partial q} = z; \quad \frac{\partial f}{\partial z} = q;$$

Looking for:

$$\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y} \quad \frac{\partial f}{\partial z}$$

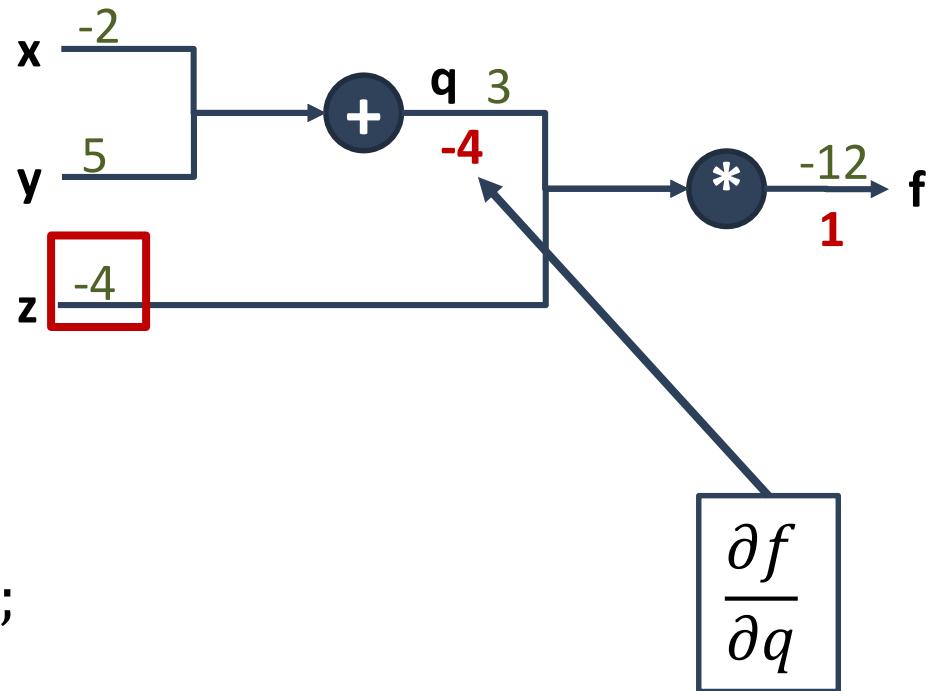
Backpropagation: A Simple Example

$$f(x, y, z) = (x + y)z$$

- With

- $x = -2$
- $y = 5$
- $z = -4$

$$q = x + y; \quad \frac{\partial q}{\partial x} = 1; \quad \frac{\partial q}{\partial y} = 1;$$



$$f = qz; \quad \boxed{\frac{\partial f}{\partial q} = z}; \quad \frac{\partial f}{\partial z} = q;$$

Looking for:

$$\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y} \quad \frac{\partial f}{\partial z}$$

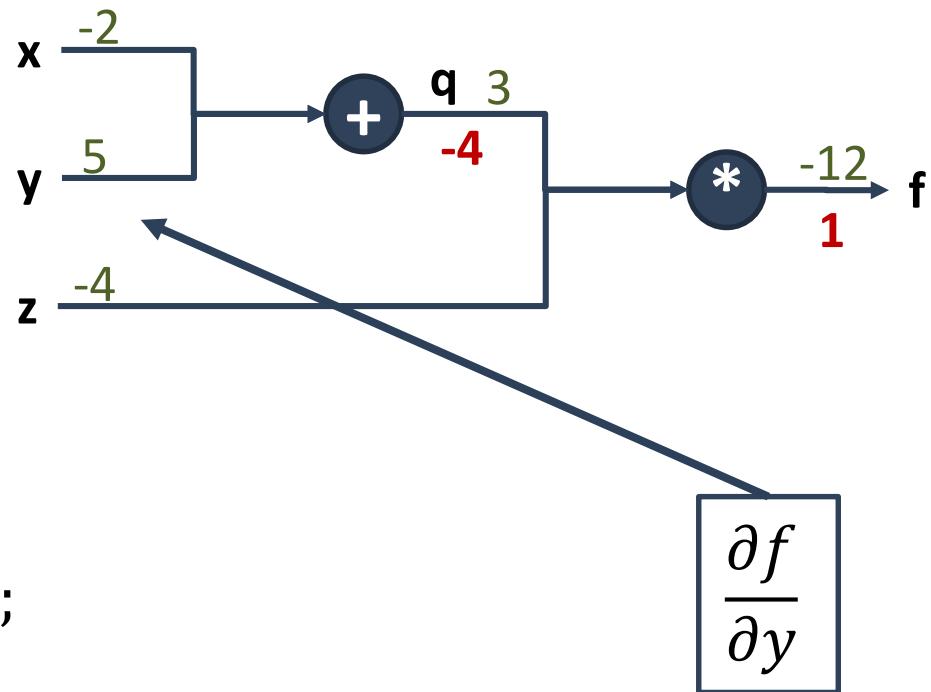
Backpropagation: A Simple Example

$$f(x, y, z) = (x + y)z$$

- With

- $x = -2$
- $y = 5$
- $z = -4$

$$q = x + y; \quad \frac{\partial q}{\partial x} = 1; \quad \frac{\partial q}{\partial y} = 1;$$



$$f = qz; \quad \frac{\partial f}{\partial q} = z; \quad \frac{\partial f}{\partial z} = q;$$

Looking for:

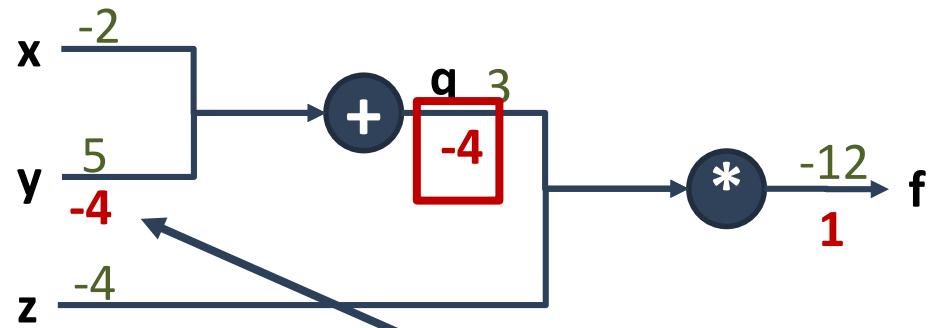
$$\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y} \quad \frac{\partial f}{\partial z}$$

Backpropagation: A Simple Example

$$f(x, y, z) = (x + y)z$$

- With

- $x = -2$
- $y = 5$
- $z = -4$



$$q = x + y; \quad \frac{\partial q}{\partial x} = 1; \quad \boxed{\frac{\partial q}{\partial y} = 1;}$$

Chain Rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}$$

$$\frac{\partial f}{\partial y}$$

$$f = qz; \quad \frac{\partial f}{\partial q} = z; \quad \frac{\partial f}{\partial z} = q;$$

Looking for:

$$\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y} \quad \frac{\partial f}{\partial z}$$

Backpropagation: A Simple Example

$$f(x, y, z) = (x + y)z$$

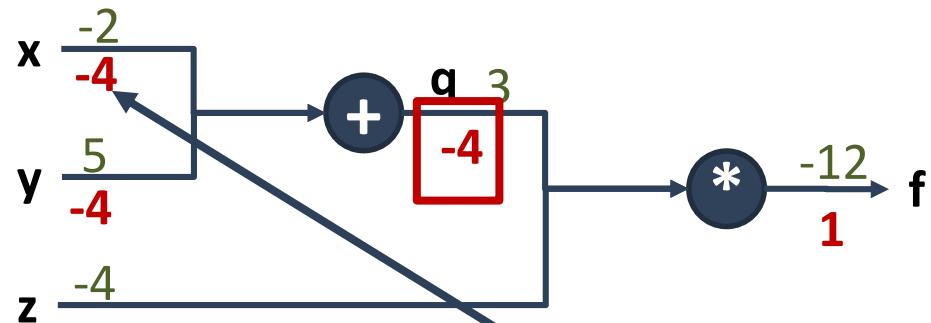
- With

- $x = -2$
- $y = 5$
- $z = -4$

$$q = x + y;$$

$$\frac{\partial q}{\partial x} = 1; \quad \frac{\partial q}{\partial y} = 1;$$

$$f = qz; \quad \frac{\partial f}{\partial q} = z; \quad \frac{\partial f}{\partial z} = q;$$



Chain Rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}$$

$$\frac{\partial f}{\partial x}$$

Looking for:

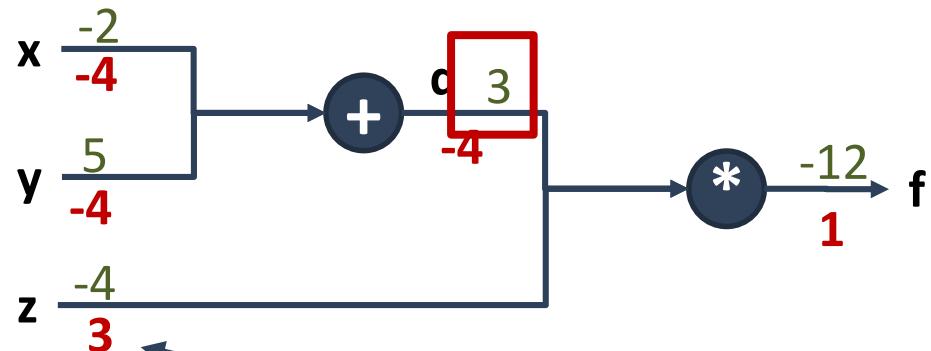
$$\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y} \quad \frac{\partial f}{\partial z}$$

Backpropagation: A Simple Example

$$f(x, y, z) = (x + y)z$$

- With

- $x = -2$
- $y = 5$
- $z = -4$



$$q = x + y; \quad \frac{\partial q}{\partial x} = 1; \quad \frac{\partial q}{\partial y} = 1;$$

$$f = qz; \quad \frac{\partial f}{\partial q} = z; \quad \frac{\partial f}{\partial z} = q;$$

$$\boxed{\frac{\partial f}{\partial z}}$$

Looking for:

$$\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y} \quad \frac{\partial f}{\partial z}$$

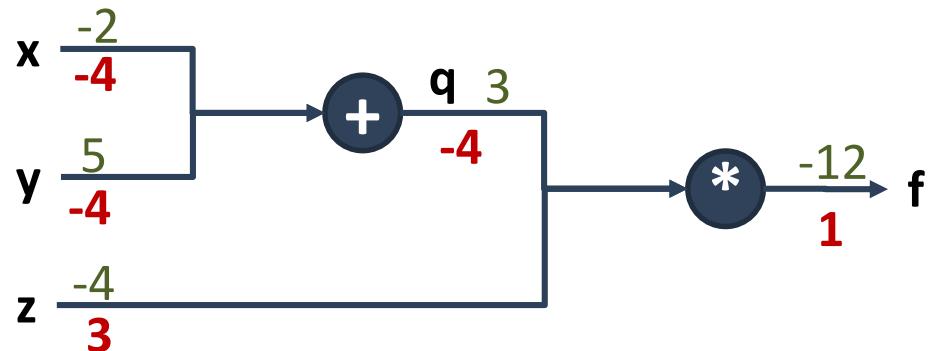
Backpropagation: A Simple Example

$$f(x, y, z) = (x + y)z$$

- With

- $x = -2$
- $y = 5$
- $z = -4$

$$q = x + y; \quad \frac{\partial q}{\partial x} = 1; \quad \frac{\partial q}{\partial y} = 1;$$



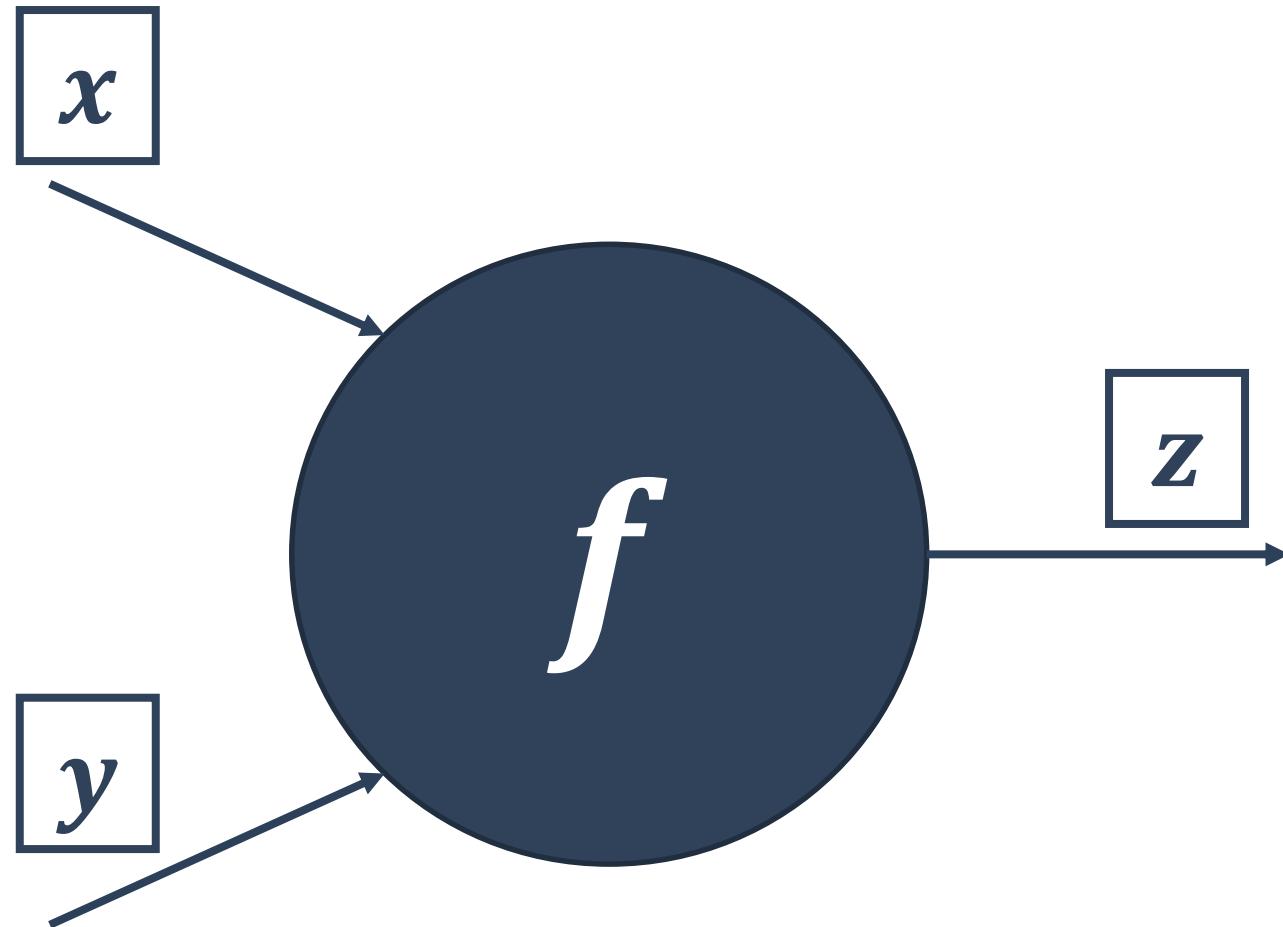
$$\nabla_{x,y,z} f = \begin{pmatrix} -4 \\ -4 \\ 3 \end{pmatrix}$$

$$f = qz; \quad \frac{\partial f}{\partial q} = z; \quad \frac{\partial f}{\partial z} = q;$$

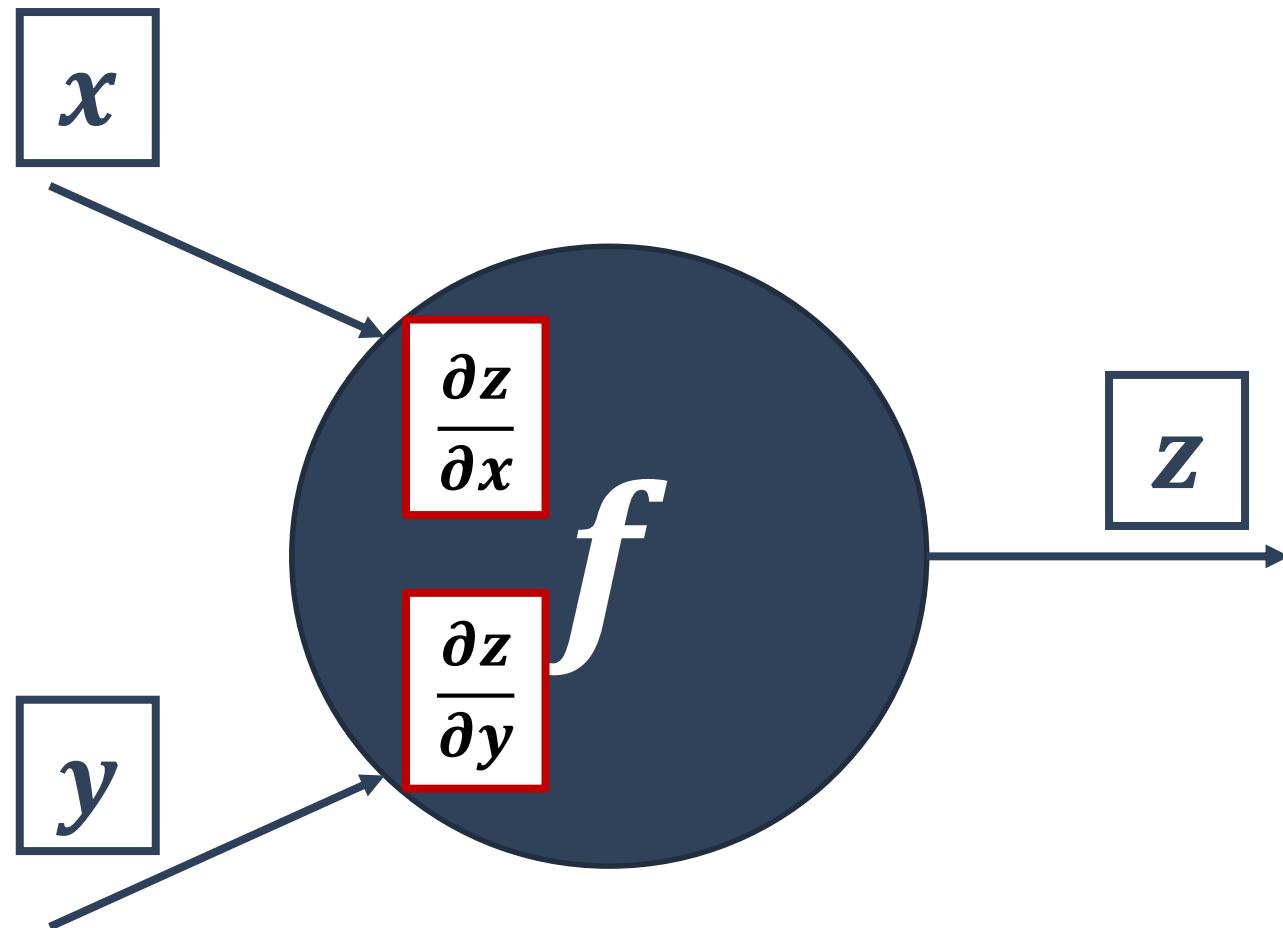
Looking for:

$$\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y} \quad \frac{\partial f}{\partial z}$$

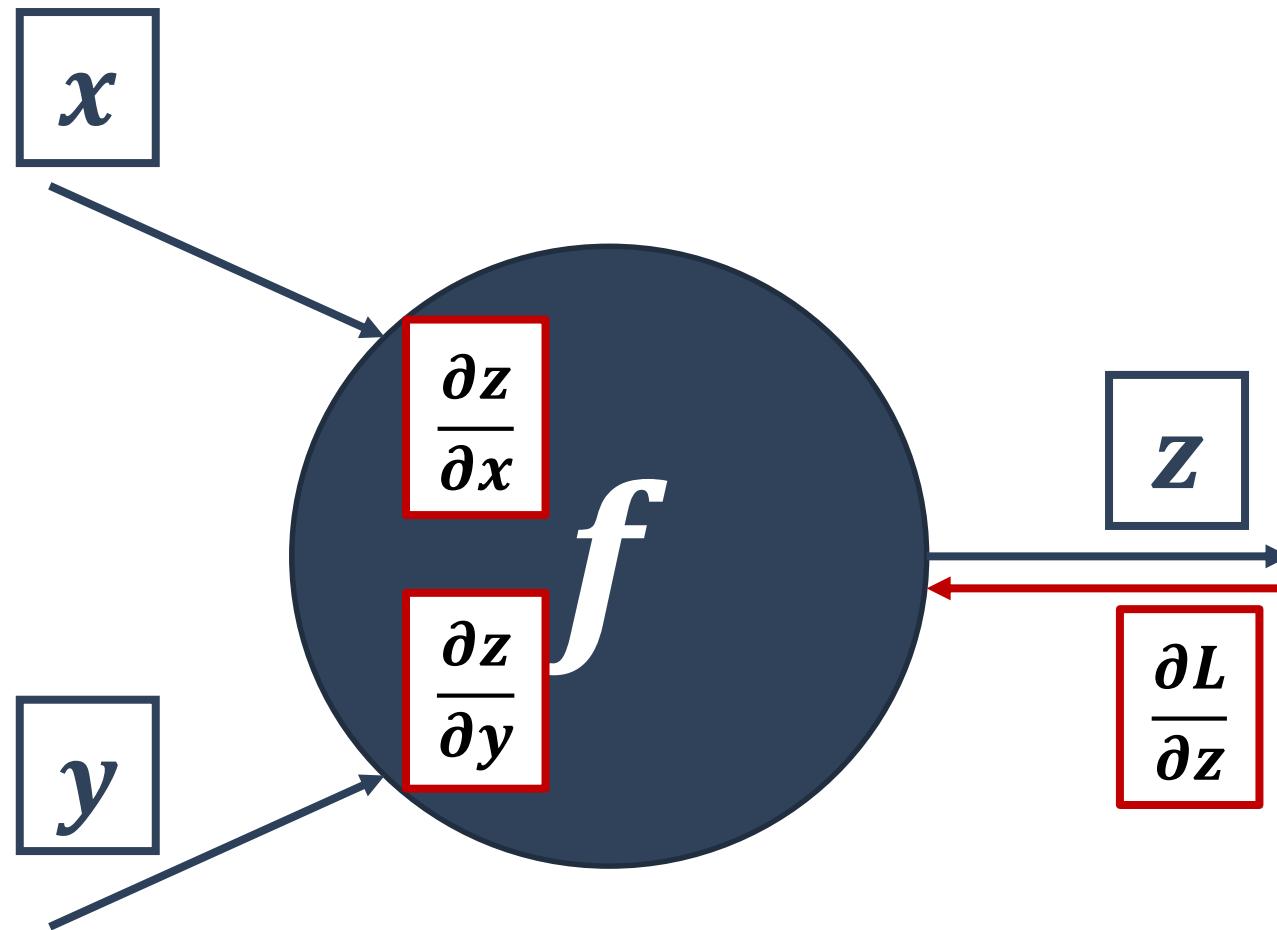
General Principle



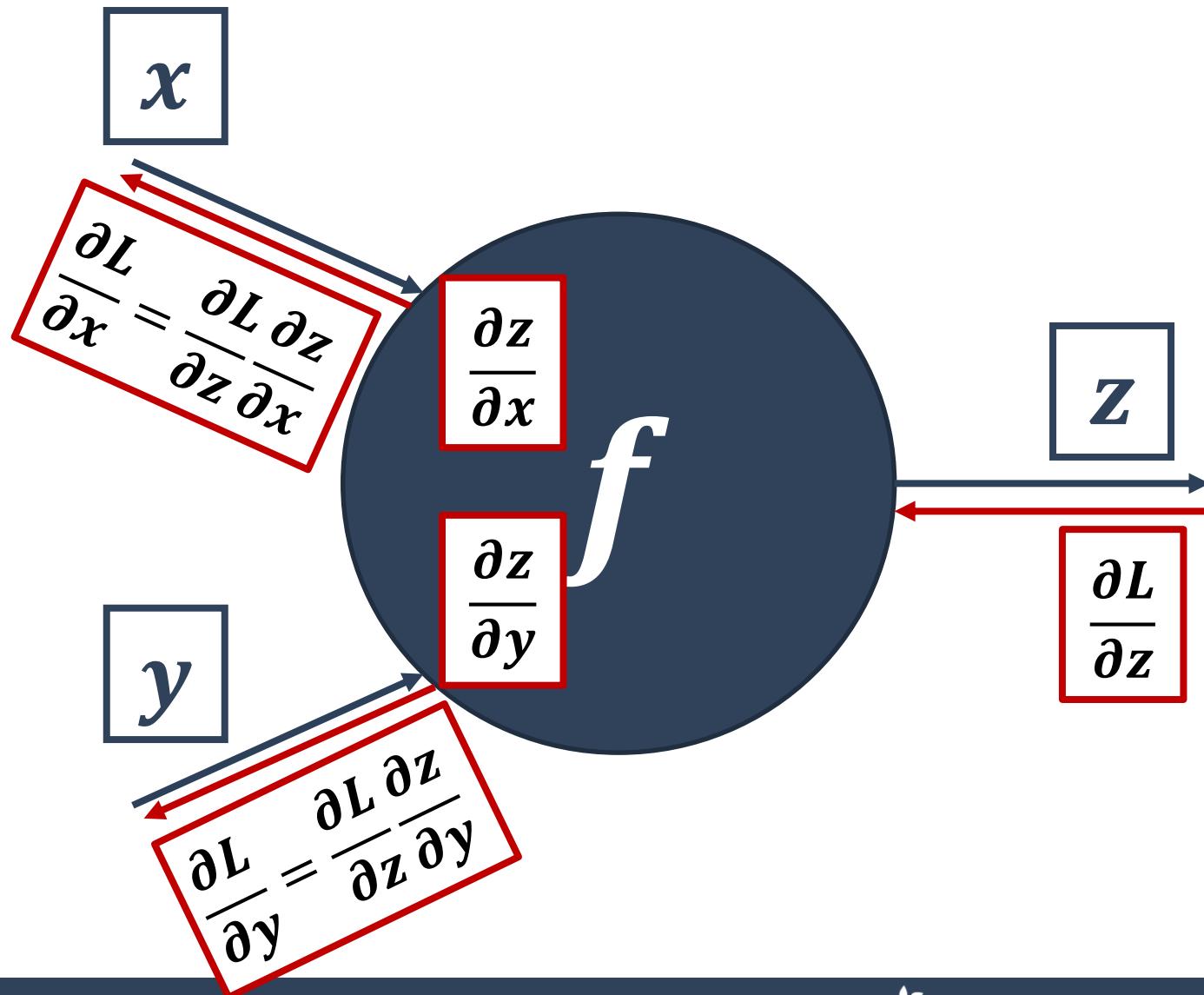
General Principle



General Principle

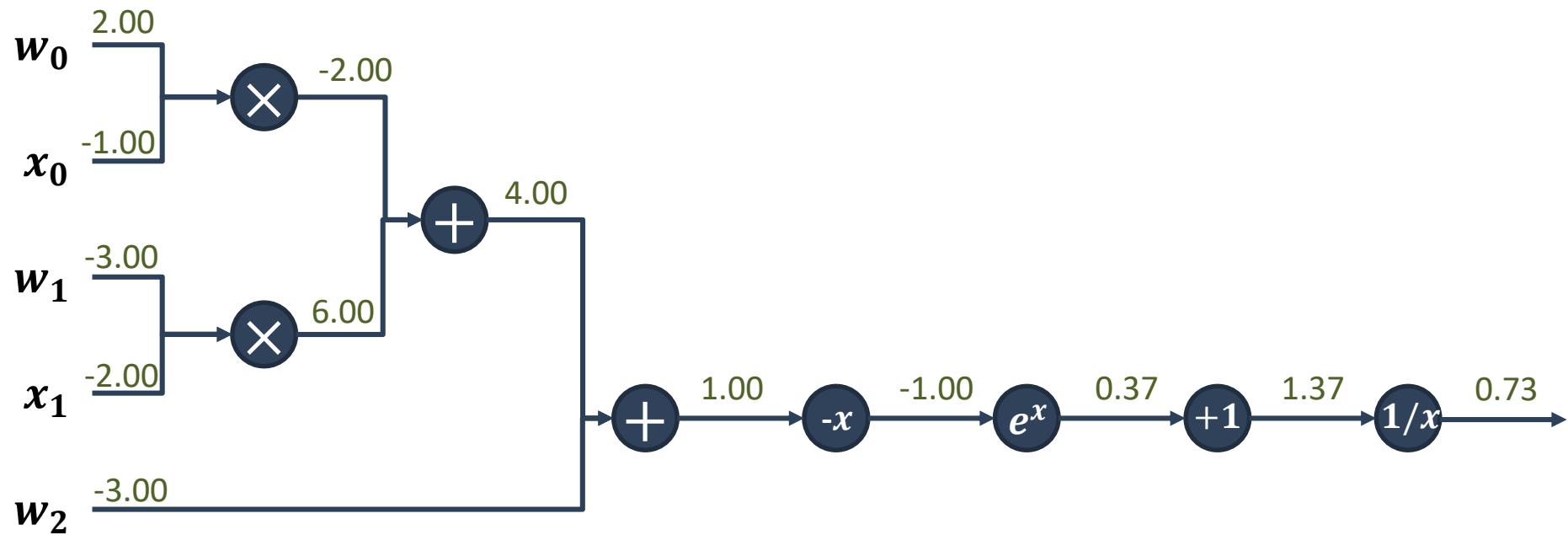


General Principle



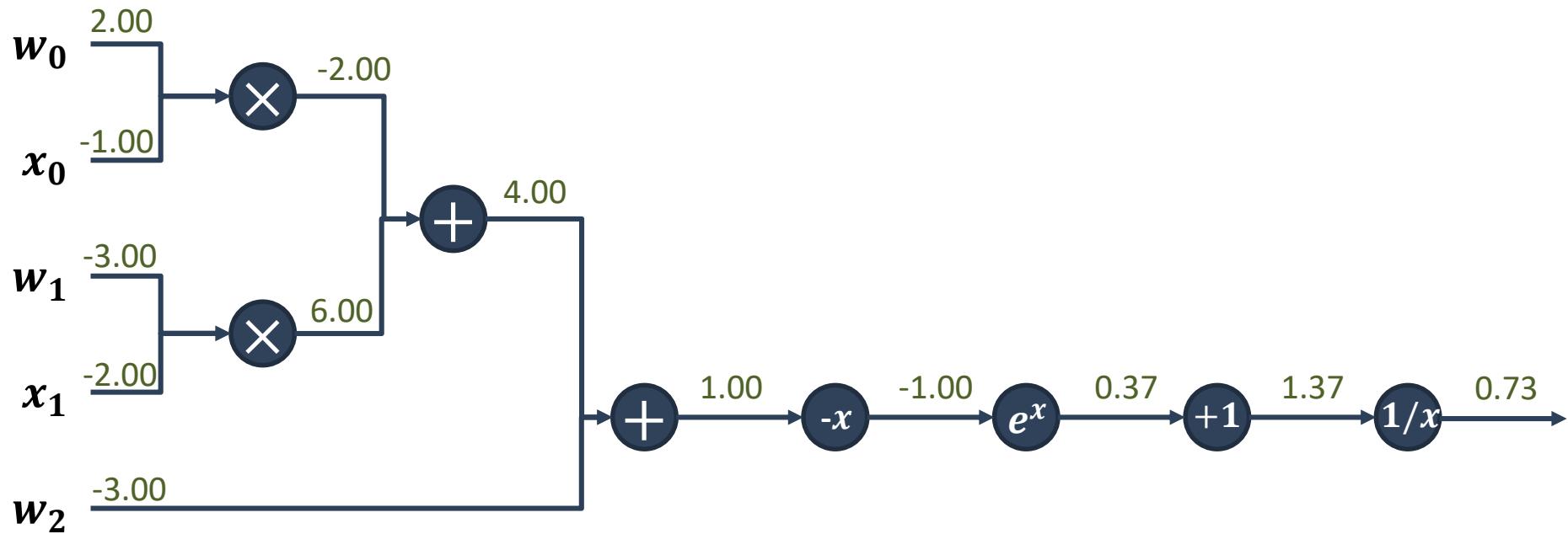
Different Example

$$f(w, x) = \frac{1}{1 + \exp(-(w_0x_0 + w_1x_1 + w_2))}$$



Different Example

$$f(w, x) = \frac{1}{1 + \exp(-(w_0x_0 + w_1x_1 + w_2))}$$



$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

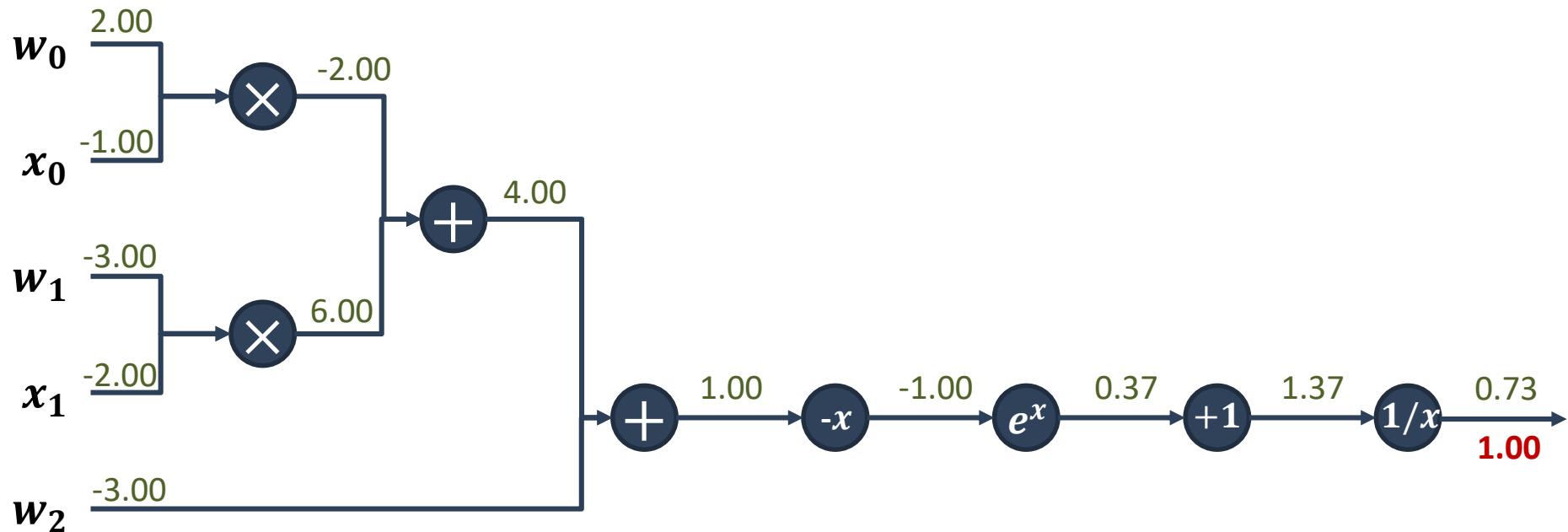
$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = -\frac{1}{x^2}$$

$$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$$

$$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$$

Different Example

$$f(w, x) = \frac{1}{1 + \exp(-(w_0x_0 + w_1x_1 + w_2))}$$



$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

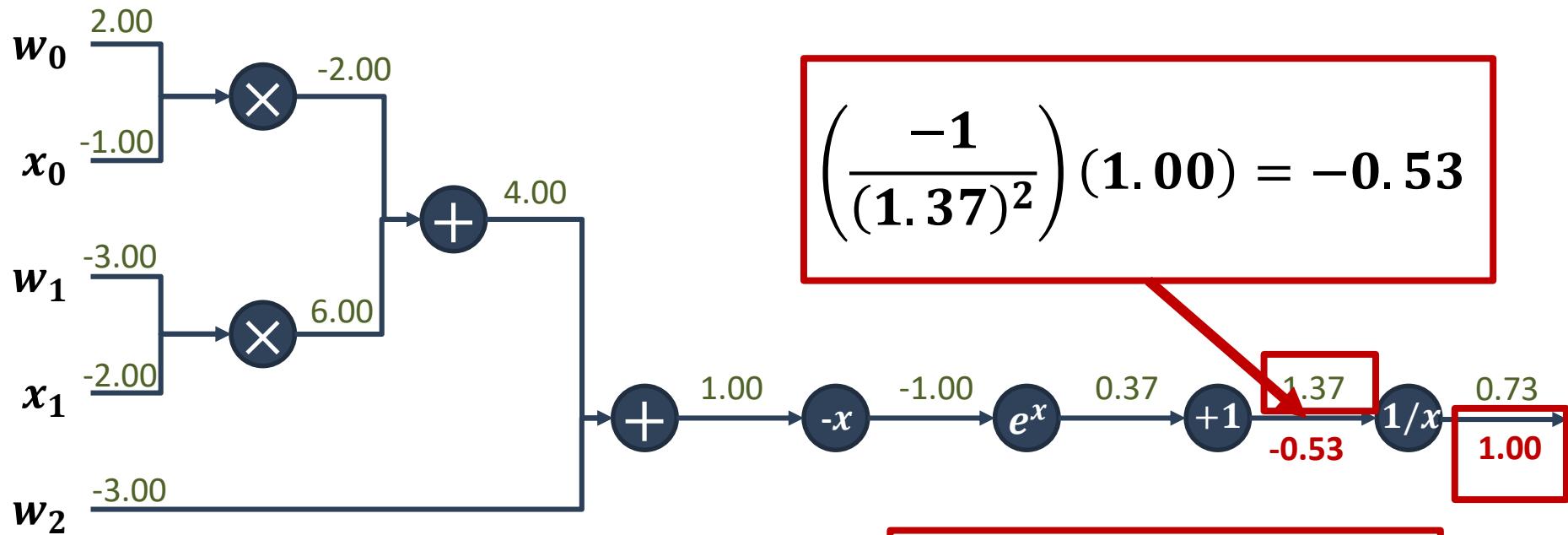
$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = -\frac{1}{x^2}$$

$$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$$

$$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$$

Different Example

$$f(w, x) = \frac{1}{1 + \exp(-(w_0x_0 + w_1x_1 + w_2))}$$



$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

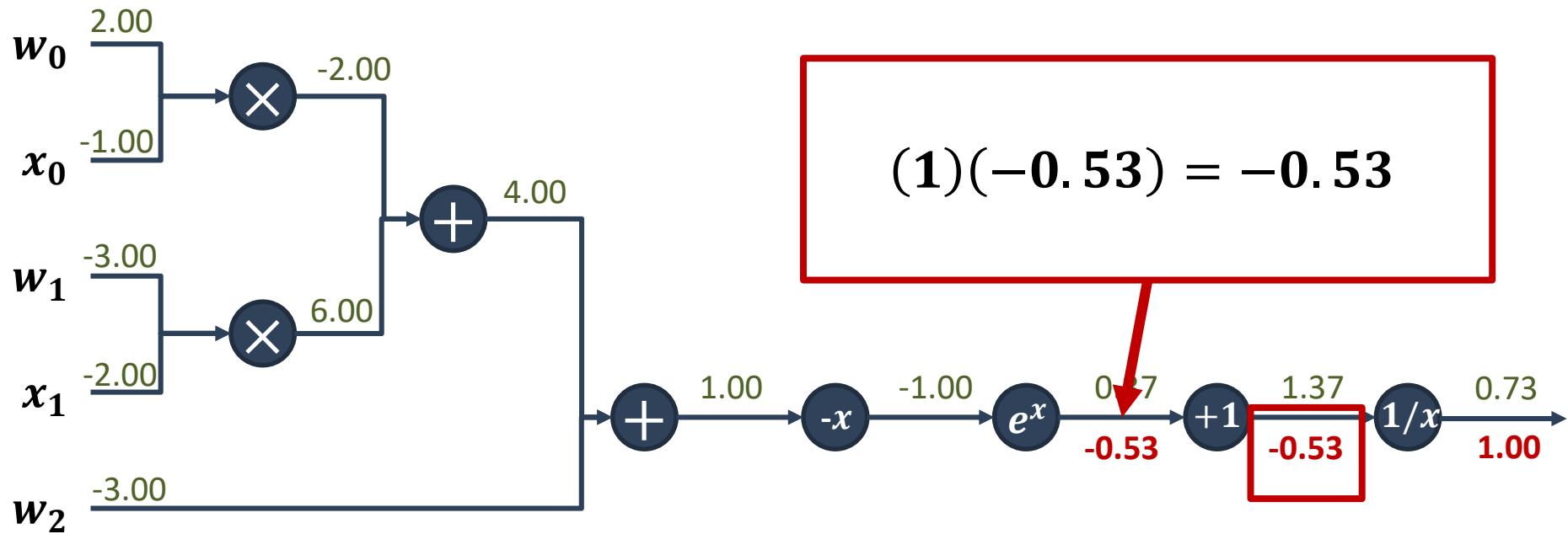
$$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$$

$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = -\frac{1}{x^2}$$

$$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$$

Different Example

$$f(w, x) = \frac{1}{1 + \exp(-(w_0x_0 + w_1x_1 + w_2))}$$



$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

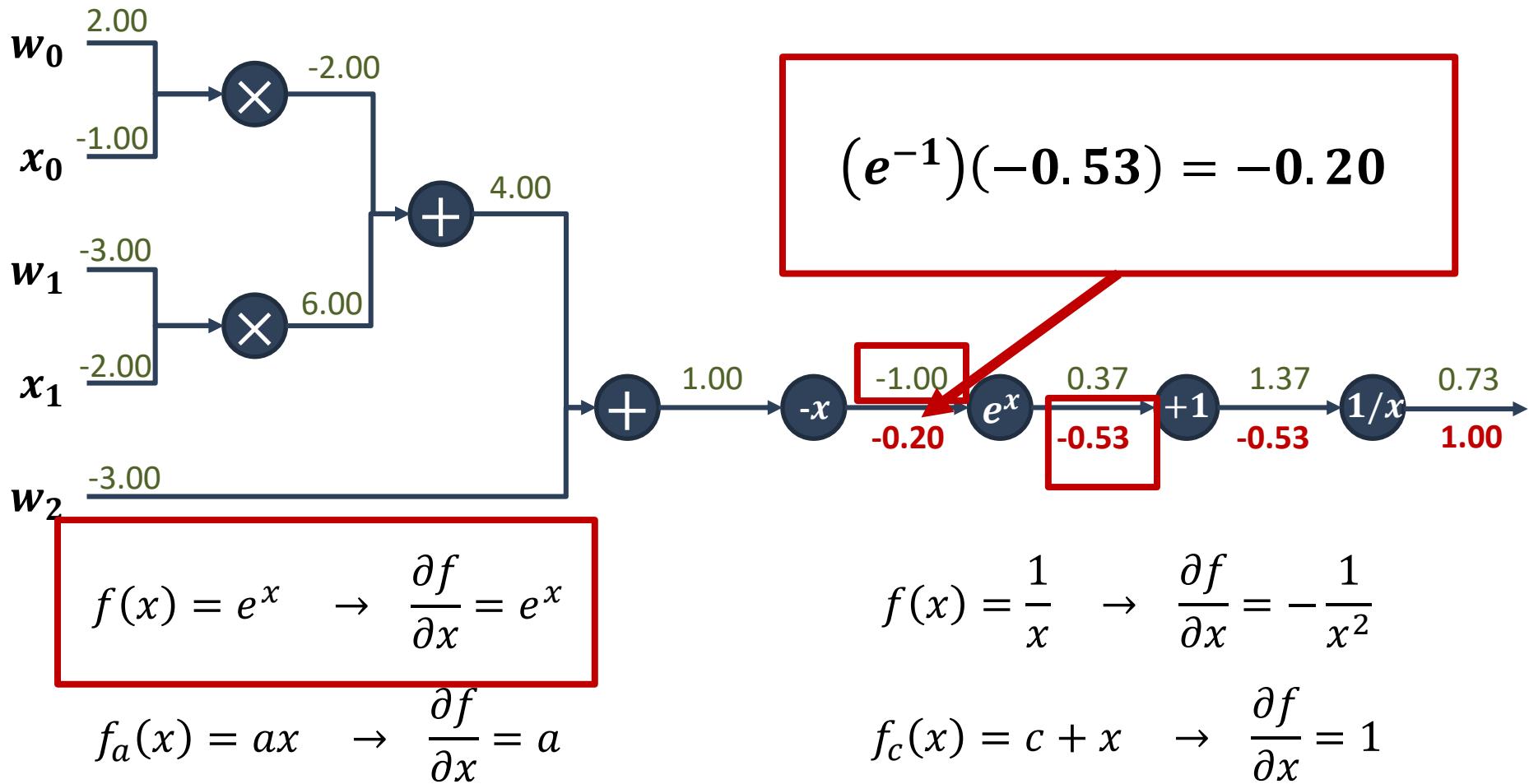
$$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$$

$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = -\frac{1}{x^2}$$

$$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$$

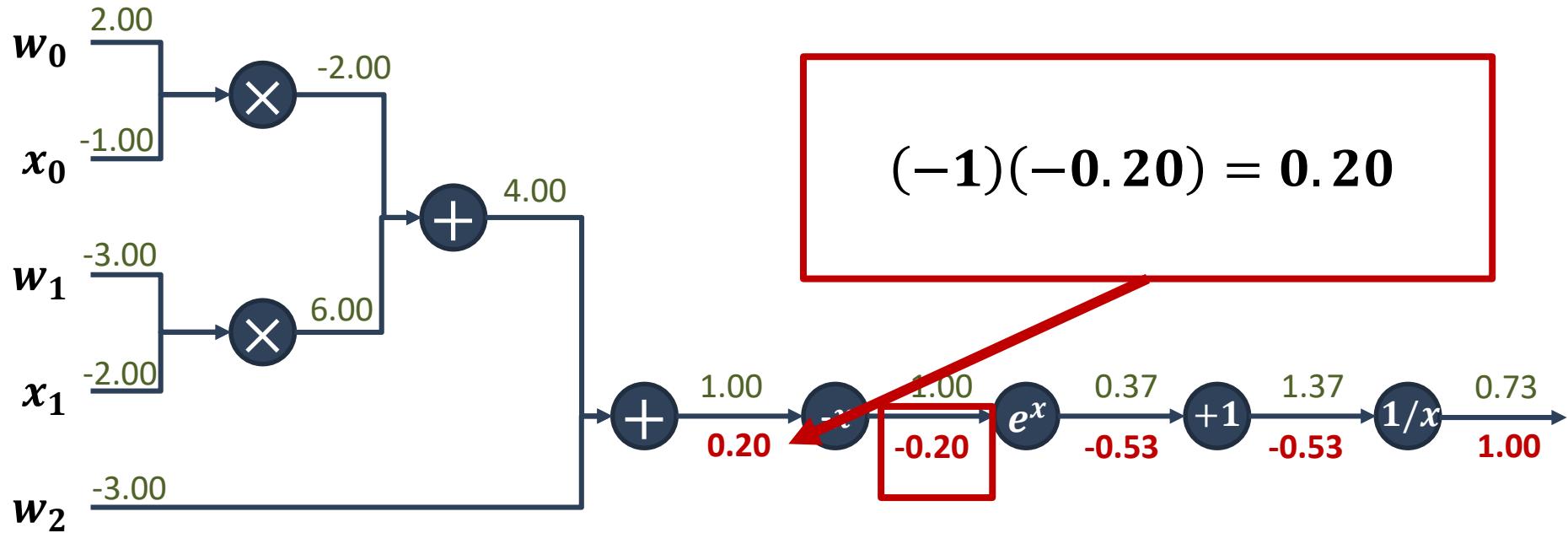
Different Example

$$f(w, x) = \frac{1}{1 + \exp(-(w_0x_0 + w_1x_1 + w_2))}$$



Different Example

$$f(w, x) = \frac{1}{1 + \exp(-(w_0x_0 + w_1x_1 + w_2))}$$



$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

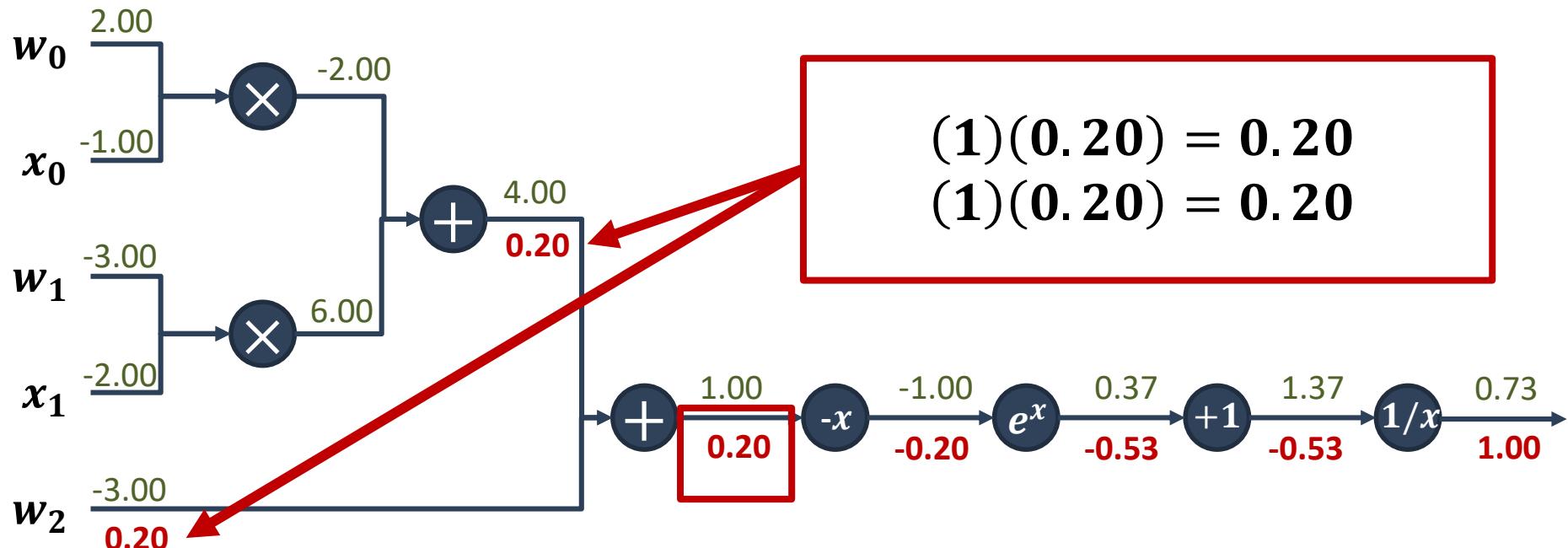
$$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$$

$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = -\frac{1}{x^2}$$

$$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$$

Different Example

$$f(w, x) = \frac{1}{1 + \exp(-(w_0x_0 + w_1x_1 + w_2))}$$



$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

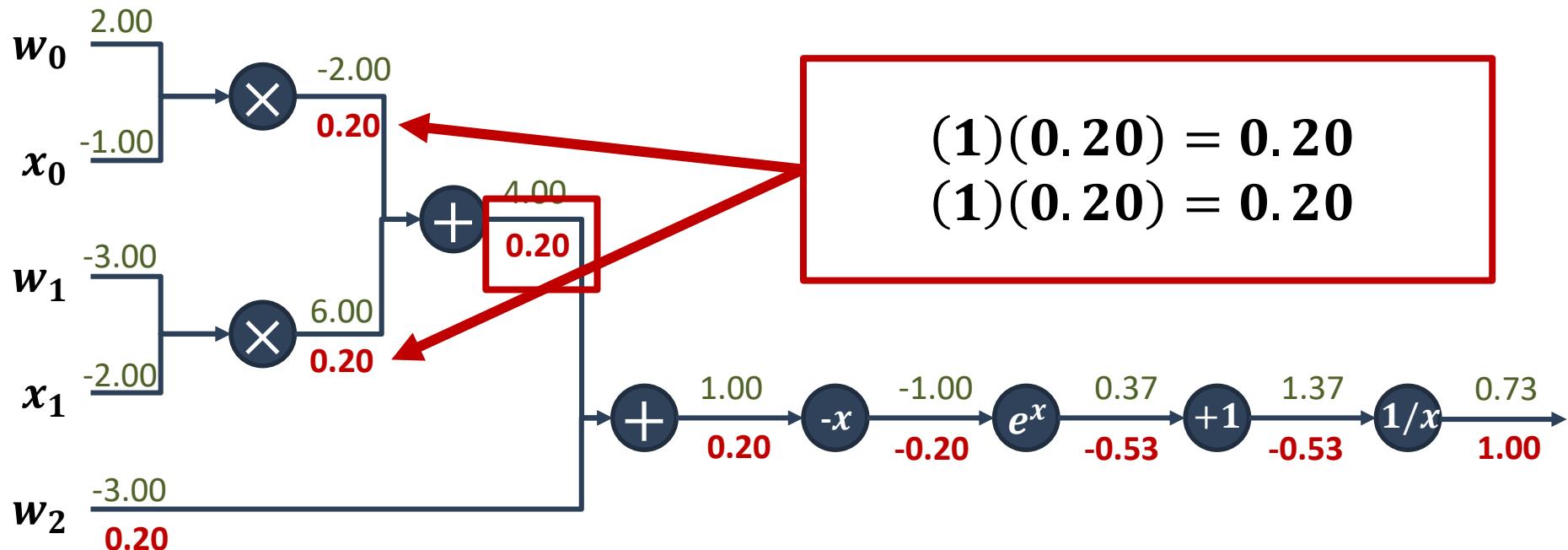
$$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$$

$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = -\frac{1}{x^2}$$

$$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$$

Different Example

$$f(w, x) = \frac{1}{1 + \exp(-(w_0x_0 + w_1x_1 + w_2))}$$



$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

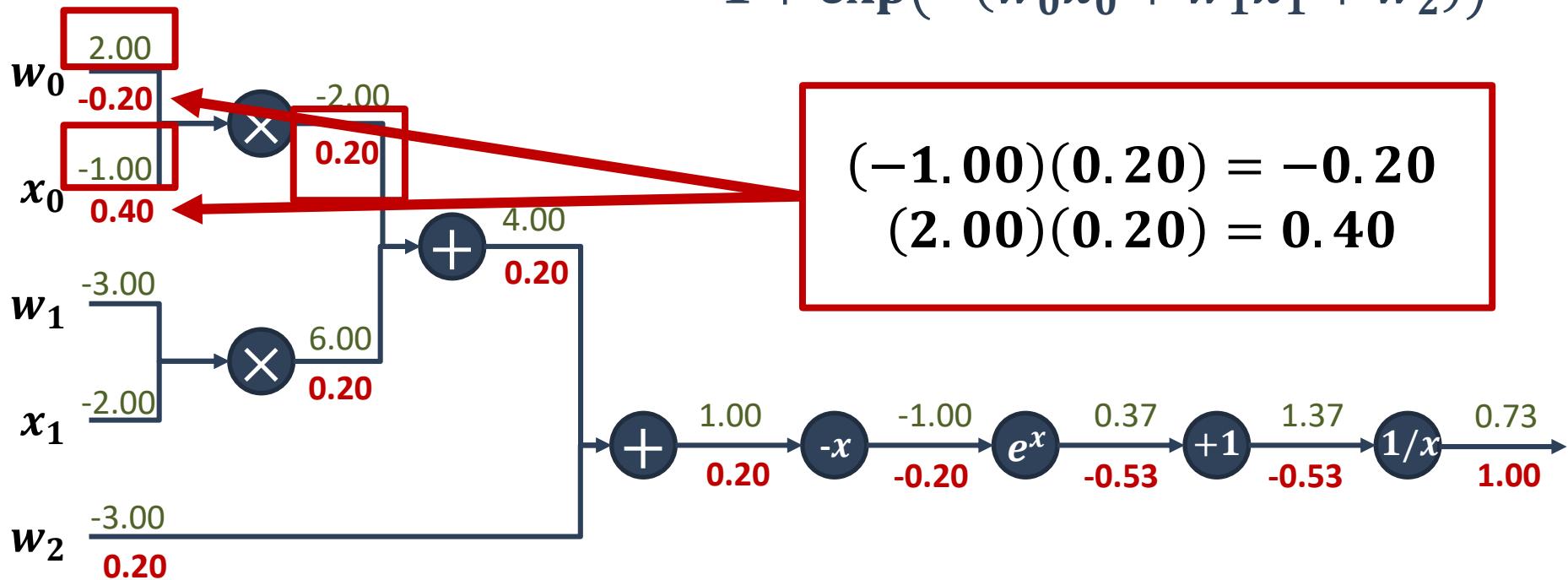
$$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$$

$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = -\frac{1}{x^2}$$

$$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$$

Different Example

$$f(w, x) = \frac{1}{1 + \exp(-(w_0x_0 + w_1x_1 + w_2))}$$



$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

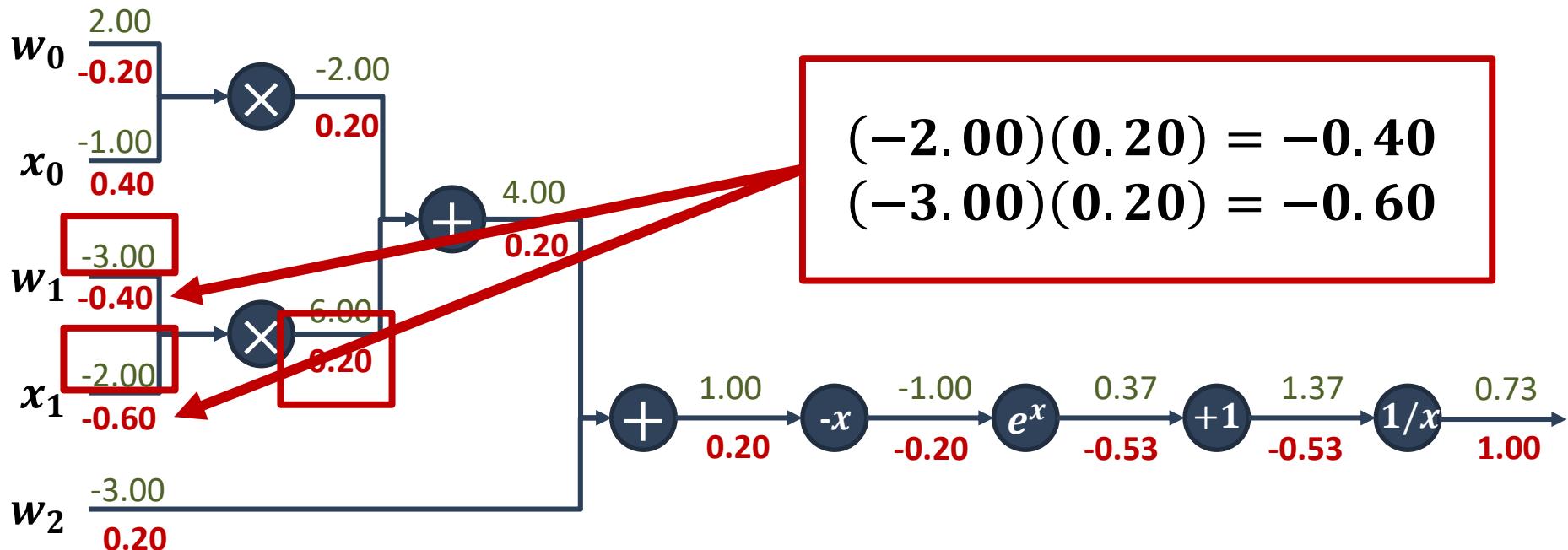
$$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$$

$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = -\frac{1}{x^2}$$

$$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$$

Different Example

$$f(w, x) = \frac{1}{1 + \exp(-(w_0x_0 + w_1x_1 + w_2))}$$



$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

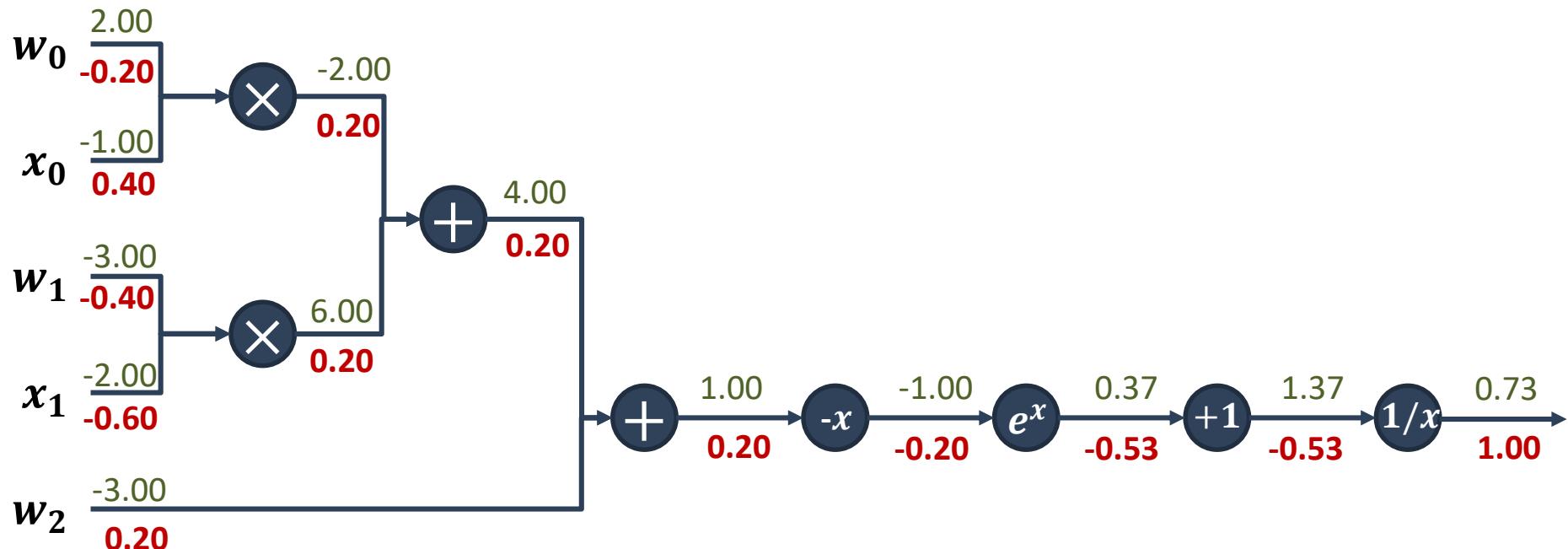
$$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$$

$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = -\frac{1}{x^2}$$

$$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$$

Different Example

$$f(w, x) = \frac{1}{1 + \exp(-(w_0x_0 + w_1x_1 + w_2))}$$



$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = -\frac{1}{x^2}$$

$$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$$

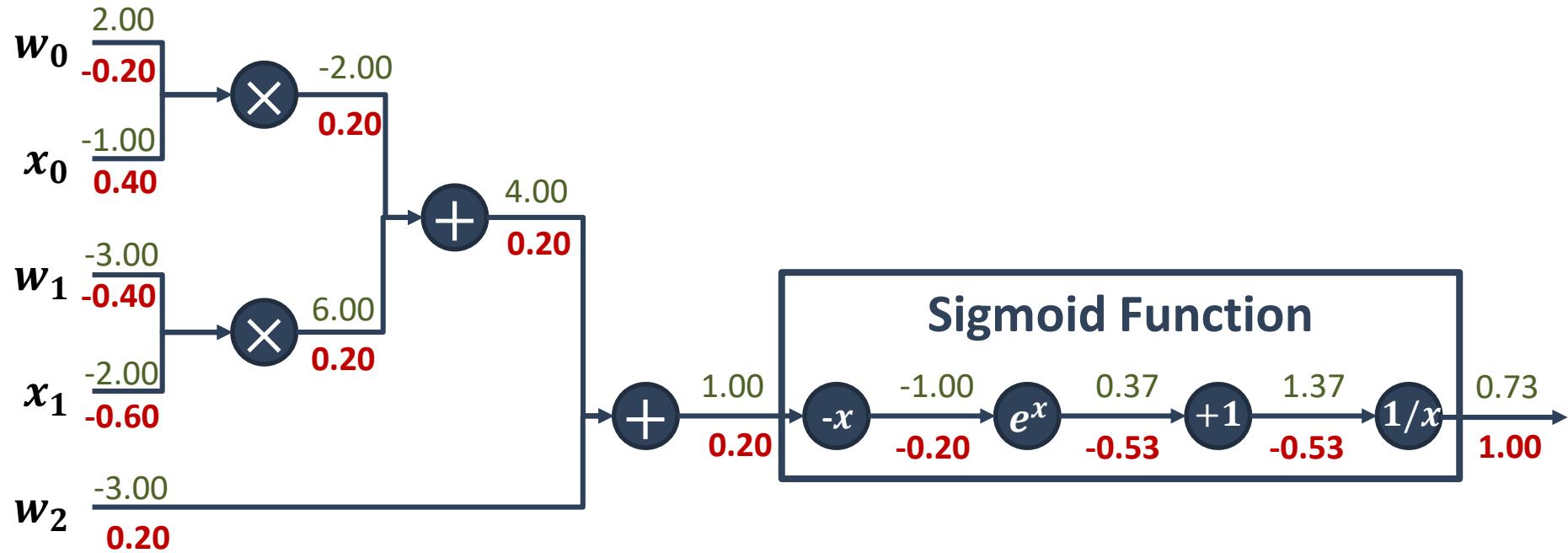
$$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$$

Different Example: Sigmoid Function

$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0x_0 + w_1x_1 + w_2))}$$

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

$$\frac{\partial \sigma}{\partial x} = (1 - \sigma(x))\sigma(x)$$

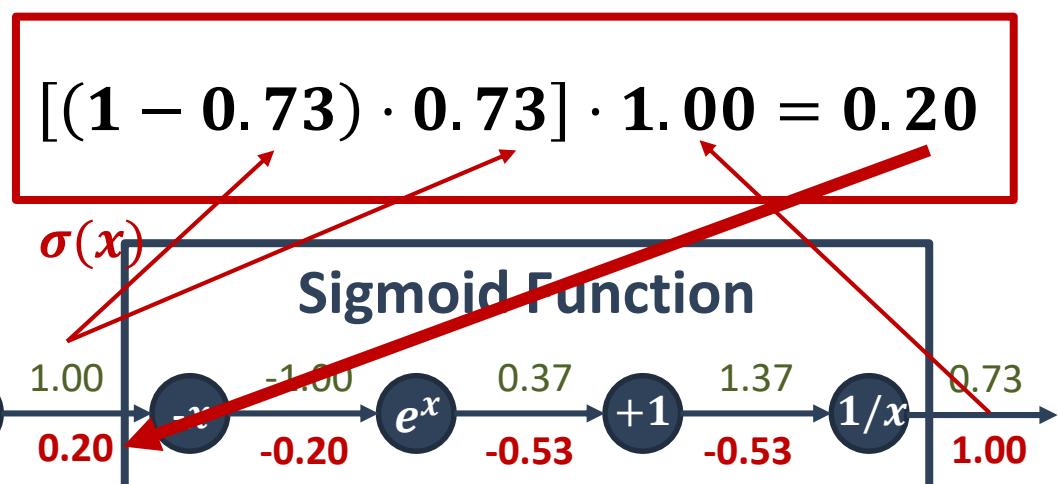
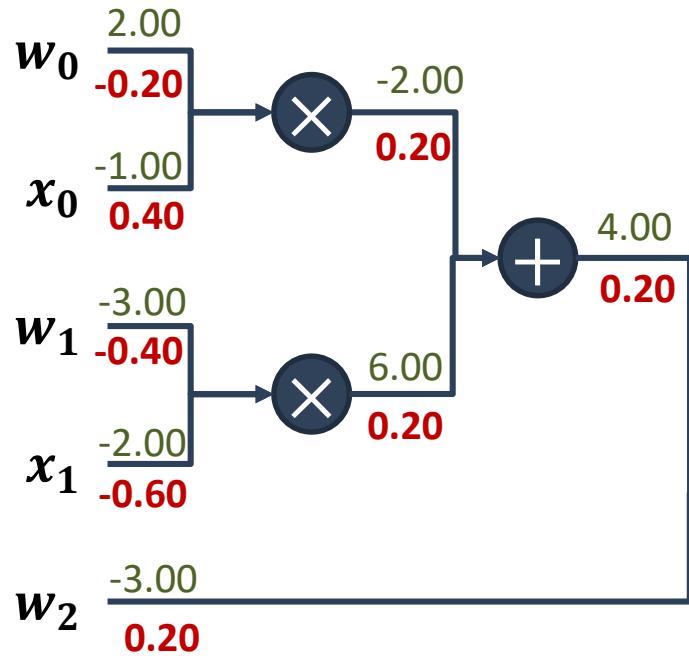


Different Example: Sigmoid Function

$$f(w, x) = \frac{1}{1 + \exp(-(w_0x_0 + w_1x_1 + w_2))}$$

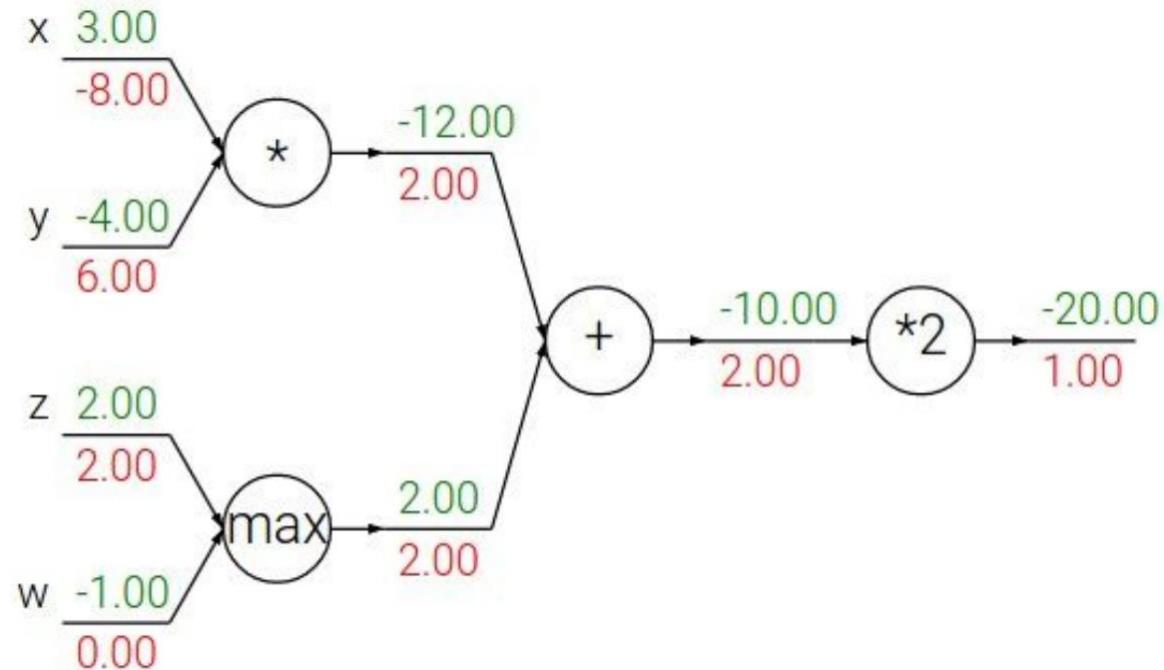
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

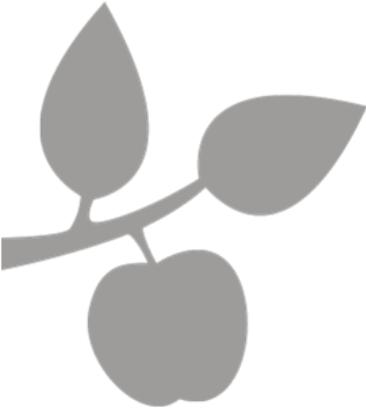
$$\frac{\partial \sigma}{\partial x} = (1 - \sigma(x))\sigma(x)$$



Patterns in Backflow of the Gradient

- **add**
 - Gradient distributor
- **max**
 - Gradient router
- **mul**
 - Gradient switcher





Backpropagation

- Function Principle
- Generalization to Vectors

Generalization to Vectors

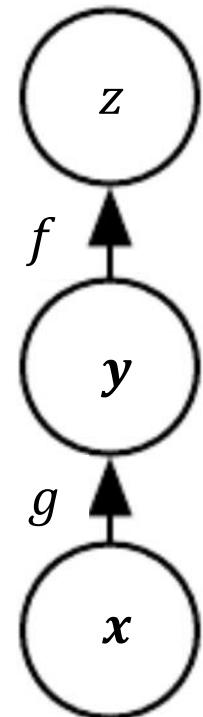
- Suppose $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$
 - g maps from \mathbb{R}^m to \mathbb{R}^n and
 - f maps from \mathbb{R}^n to \mathbb{R}
- If $\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$, then

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

- Or, in vector notation:

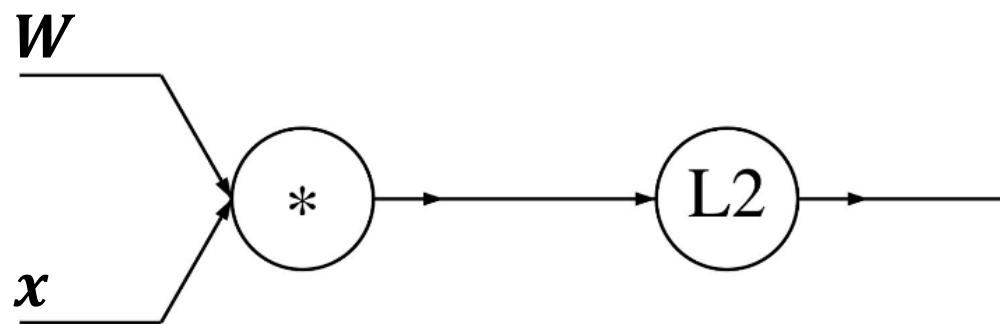
$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right)^T \nabla_{\mathbf{y}} z$$

- That is the product of the Jacobian matrix $\frac{\partial \mathbf{x}}{\partial \mathbf{y}}$ and the gradient vector $\nabla_{\mathbf{y}} z$.



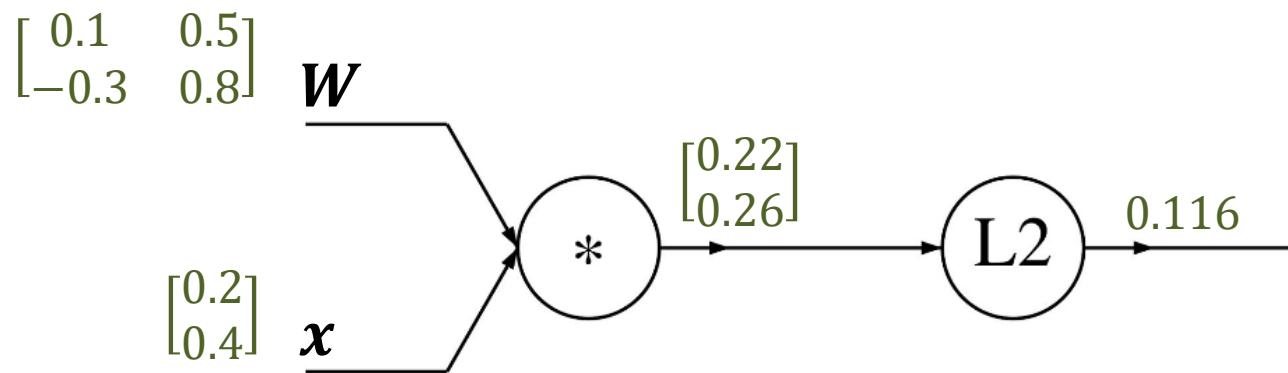
Vectorized Example

$$f(\mathbf{x}, \mathbf{W}) = \|\mathbf{W}\mathbf{x}\|^2 = \sum_i^n (\mathbf{W}\mathbf{x})_i^2$$



Vectorized Example

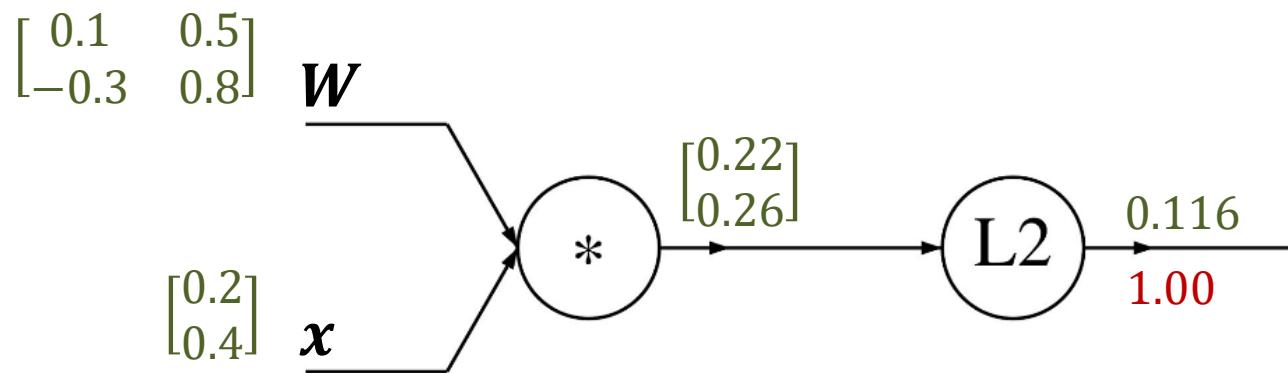
$$f(\mathbf{x}, \mathbf{W}) = \|\mathbf{W}\mathbf{x}\|^2 = \sum_i^n (\mathbf{W}\mathbf{x})_i^2$$



$$\begin{aligned} q &= \mathbf{W}\mathbf{x} = \begin{pmatrix} W_{1,1}x_1 & \cdots & W_{1,n}x_n \\ \vdots & \ddots & \vdots \\ W_{n,1}x_1 & \cdots & W_{n,n}x_n \end{pmatrix} \\ f(q) &= \|q\|^2 = q_1^2 + \cdots + q_n^2 \end{aligned}$$

Vectorized Example

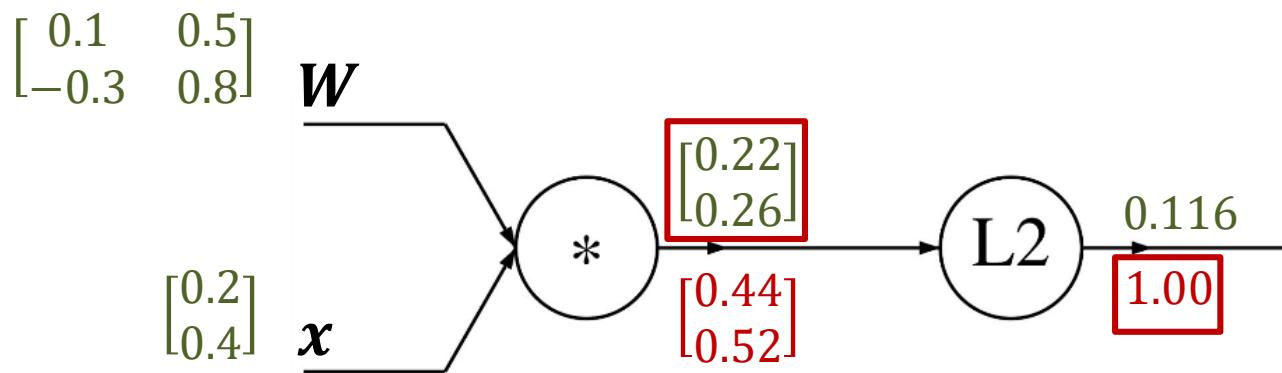
$$f(\mathbf{x}, \mathbf{W}) = \|\mathbf{W}\mathbf{x}\|^2 = \sum_i^n (\mathbf{W}\mathbf{x})_i^2$$



$$\begin{aligned} q &= \mathbf{W}\mathbf{x} = \begin{pmatrix} W_{1,1}x_1 & \cdots & W_{1,n}x_n \\ \vdots & \ddots & \vdots \\ W_{n,1}x_1 & \cdots & W_{n,n}x_n \end{pmatrix} \\ f(q) &= \|q\|^2 = q_1^2 + \cdots + q_n^2 \end{aligned}$$

Vectorized Example

$$f(\mathbf{x}, \mathbf{W}) = \|\mathbf{W}\mathbf{x}\|^2 = \sum_i^n (\mathbf{W}\mathbf{x})_i^2$$

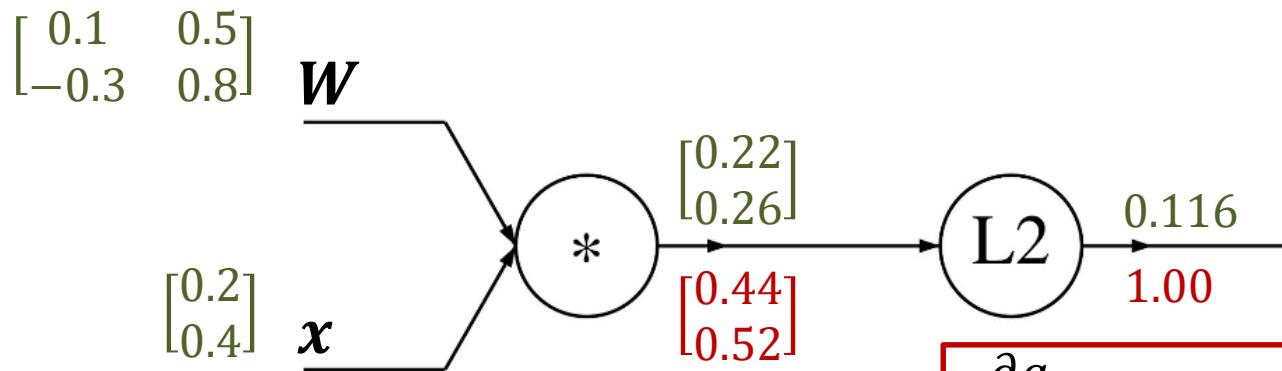


$$\begin{aligned} q &= \mathbf{W}\mathbf{x} = \begin{pmatrix} W_{1,1}x_1 & \cdots & W_{1,n}x_n \\ \vdots & \ddots & \vdots \\ W_{n,1}x_1 & \cdots & W_{n,n}x_n \end{pmatrix} \\ f(q) &= \|q\|^2 = q_1^2 + \cdots + q_n^2 \end{aligned}$$

$$\boxed{\begin{aligned} \frac{\partial f}{\partial q_i} &= 2q_i \\ \nabla_q f &= 2q \end{aligned}}$$

Vectorized Example

$$f(\mathbf{x}, \mathbf{W}) = \|\mathbf{W}\mathbf{x}\|^2 = \sum_i^n (\mathbf{W}\mathbf{x})_i^2$$

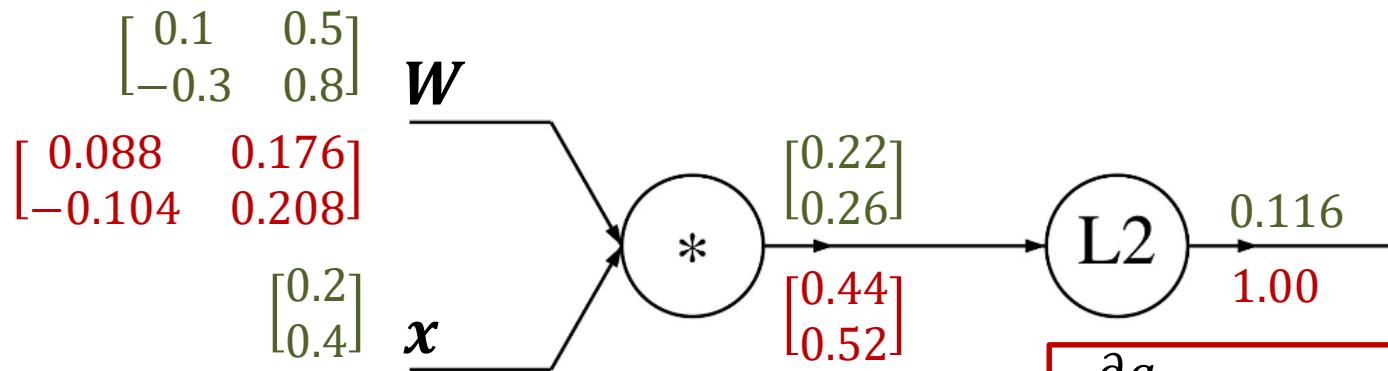


$$\begin{aligned} q &= \mathbf{W}\mathbf{x} = \begin{pmatrix} W_{1,1}x_1 & \cdots & W_{1,n}x_n \\ \vdots & \ddots & \vdots \\ W_{n,1}x_1 & \cdots & W_{n,n}x_n \end{pmatrix} \\ f(q) &= \|q\|^2 = q_1^2 + \cdots + q_n^2 \end{aligned}$$

$$\begin{aligned} \frac{\partial q_k}{\partial W_{i,j}} &= \mathbf{1}_{k=i} x_j \\ \frac{\partial f}{\partial W_{i,j}} &= \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial W_{i,j}} \\ &= \sum_k (2q_k)(\mathbf{1}_{k=i} x_j) \\ &= 2q_i x_j \end{aligned}$$

Vectorized Example

$$f(\mathbf{x}, \mathbf{W}) = \|\mathbf{W}\mathbf{x}\|^2 = \sum_i^n (\mathbf{W}\mathbf{x})_i^2$$



$$q = \mathbf{W}\mathbf{x} = \begin{pmatrix} W_{1,1}x_1 & \cdots & W_{1,n}x_n \\ \vdots & \ddots & \vdots \\ W_{n,1}x_1 & \cdots & W_{n,n}x_n \end{pmatrix}$$

$$f(q) = \|q\|^2 = q_1^2 + \cdots + q_n^2$$

$$\frac{\partial q_k}{\partial W_{i,j}} = \mathbf{1}_{k=i} x_i$$

$$\frac{\partial f}{\partial W} = \sum \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial W_{i,j}}$$

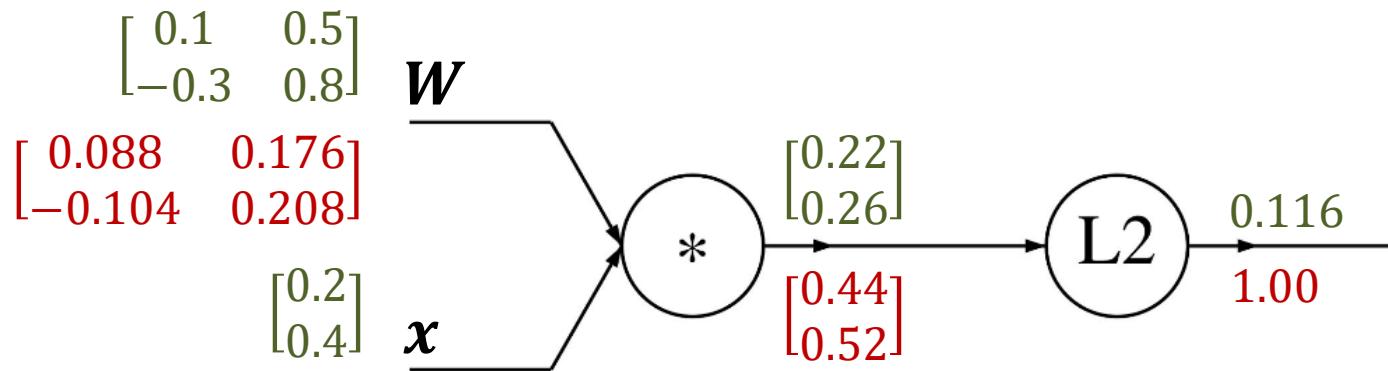
$$\nabla_W f = 2\mathbf{q} \cdot \mathbf{x}^T$$

$$= \sum_k (2q_k)(\mathbf{1}_{k=i} x_j)$$

$$= 2q_i x_j$$

Vectorized Example

$$f(\mathbf{x}, \mathbf{W}) = \|\mathbf{W}\mathbf{x}\|^2 = \sum_i^n (\mathbf{W}\mathbf{x})_i^2$$



$$q = \mathbf{W}\mathbf{x} = \begin{pmatrix} W_{1,1}x_1 & \cdots & W_{1,n}x_n \\ \vdots & \ddots & \vdots \\ W_{n,1}x_1 & \cdots & W_{n,n}x_n \end{pmatrix}$$

$$f(q) = \|q\|^2 = q_1^2 + \cdots + q_n^2$$

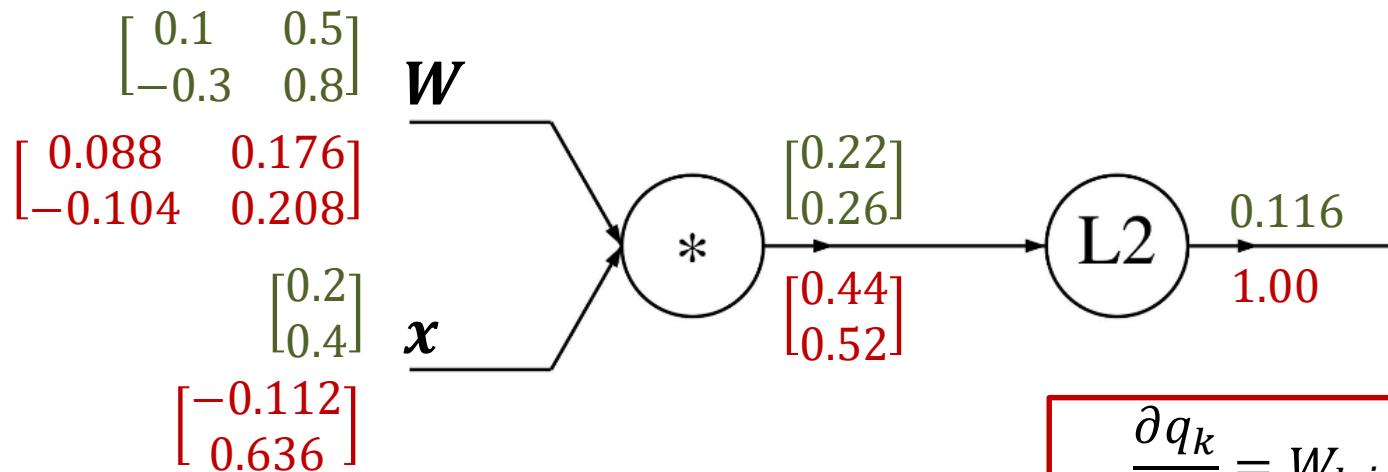
$$\frac{\partial q_k}{\partial x_i} = W_{k,i}$$

$$\frac{\partial f}{\partial x_i} = \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial x_i}$$

$$= \sum_k 2q_k W_{k,i}$$

Vectorized Example

$$f(\mathbf{x}, \mathbf{W}) = \|\mathbf{W}\mathbf{x}\|^2 = \sum_i^n (\mathbf{W}\mathbf{x})_i^2$$



$$\begin{aligned} q &= \mathbf{W}\mathbf{x} = \begin{pmatrix} W_{1,1}x_1 & \cdots & W_{1,n}x_n \\ \vdots & \ddots & \vdots \\ W_{n,1}x_1 & \cdots & W_{n,n}x_n \end{pmatrix} \\ f(q) &= \|q\|^2 = q_1^2 + \cdots + q_n^2 \end{aligned}$$

$$\frac{\partial q_k}{\partial x_i} = W_{k,i}$$

$$\frac{\partial f}{\partial f} = \nabla \frac{\partial f}{\partial q_k}$$

$$\nabla_{\mathbf{x}} f = 2\mathbf{W}^T \mathbf{q}$$

$$= \sum_k 2q_k W_{k,i}$$

Two approaches to backpropagation

1. Symbol-to-number differentiation

- Take a computational graph and a set of numerical values for inputs to the graph
- Return a set of numerical values describing gradient at those input values
- Used by libraries: Torch and Caffe

2. Symbol-to-symbol differentiation

- Take a computational graph
- Add additional nodes to the graph that provide a symbolic description of desired derivatives
- Used by libraries: Theano and Tensorflow

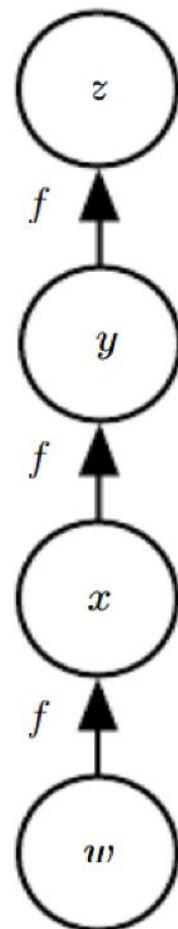
Symbol-to-symbol Derivatives

- To compute derivative using this approach, backpropagation does not need to ever access any actual numerical values
- Instead it adds nodes to a computational graph describing how to compute the derivatives for any specific numerical values
- A generic graph evaluation engine can later compute derivatives for any specific numerical values

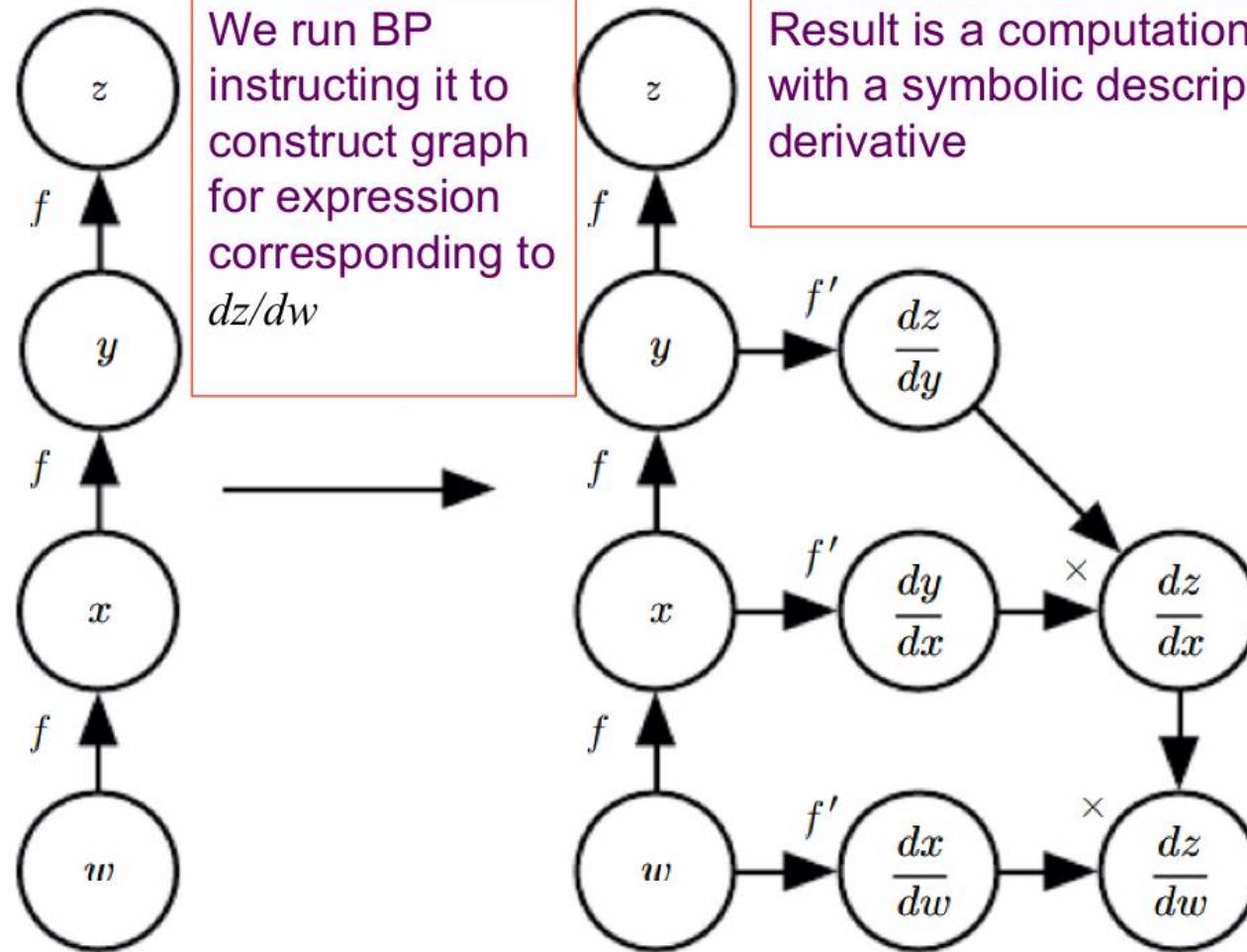
Example

- Consider the following function:

$$z = f(f(f(w)))$$



Symbol-to-Symbol Derivative Computation

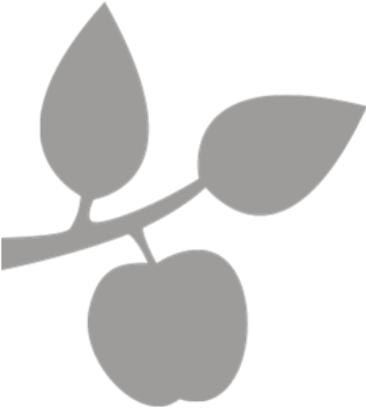


Advantages of Approach

- Derivatives are described in the same language as the original expression
- Because the derivatives are just another computational graph, it is possible to run back-propagation again
 - Differentiating the derivatives
 - Yields higher-order derivatives

What is Back-Propagation and what not!

- Often simply called backprop
 - Allows information from the cost to flow back through network to compute gradient
- The backpropagation algorithm does this using a simple and inexpensive procedure (and some optimizations, like dynamic programming to avoid evaluating the same expression twice)
- Backpropagation **is not Learning**
 - Only refers to the method for computing gradients
 - Needs to be coupled with a learning algorithm, e.g., stochastic gradient descent
 - Backprob is NOT specific to Deep Learning



DM873

Deep Learning

Spring 2019

Lecture 8 – CNN

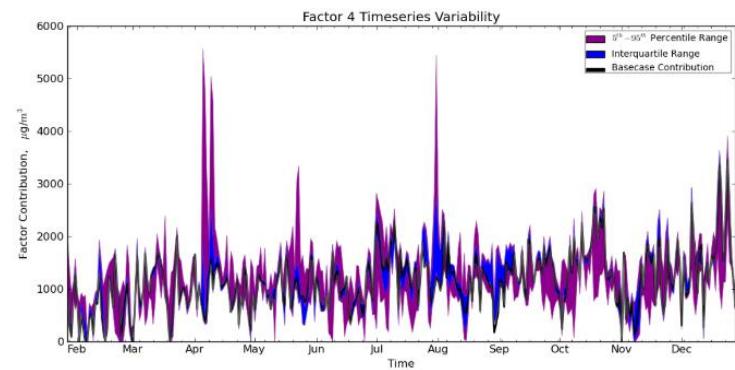


Convolutional Neural Networks

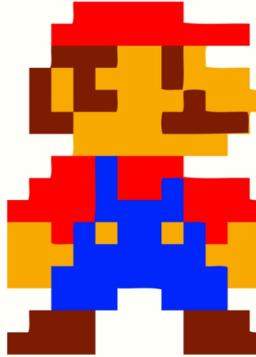
- **Convolution**
- **Pooling**
- **Variants of Convolution**
- **Efficient Convolution**
- **Convolution in Keras**

Convolutional Networks

- Convolutional neural networks (CNNs) are a specialized kind of neural network
 - Neural networks that use **convolution** in place of general matrix multiplication in at least one of their layers
- Well suited for data with a grid-like topology
- But: Convolution can be viewed as multiplication by a matrix



E.g. Ex: time-series data, which is a 1-D grid, taking samples at intervals



More obviously: Image data, which are 2-D grid of pixels

What is Convolution?

- Convolution is an operation on two functions of a real-valued argument
- Examples of the two functions
 - Tracking location of a spaceship by a laser sensor
 - A laser sensor provides a single output $x(t)$, the position of spaceship at time t
 - w a function of a real-valued argument
 - If laser sensor is noisy, we want a weighted average that gives more weight to recent observations
 - Weighting function is $w(a)$ where a is age of measurement
 - Convolution is the smoothed estimate of the position of the spaceship

$$s(t) = \int x(a)w(t - a)da$$

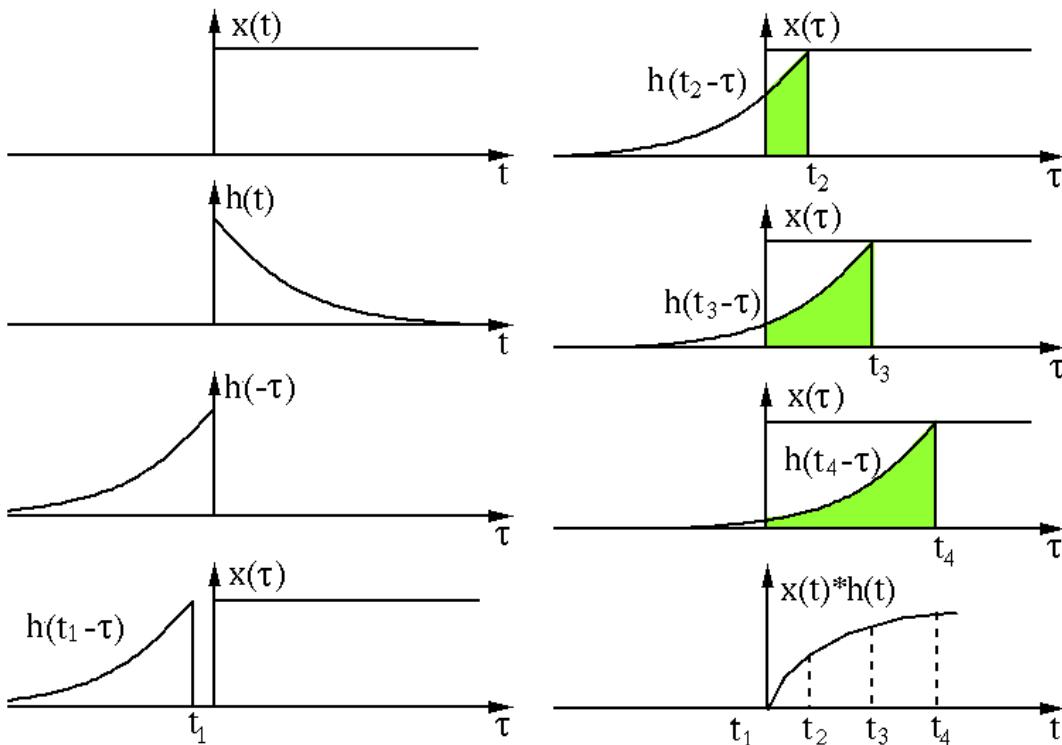
What is Convolution?

- One-dimensional continuous case
 - Input $f(t)$ is convolved with a kernel $g(t)$

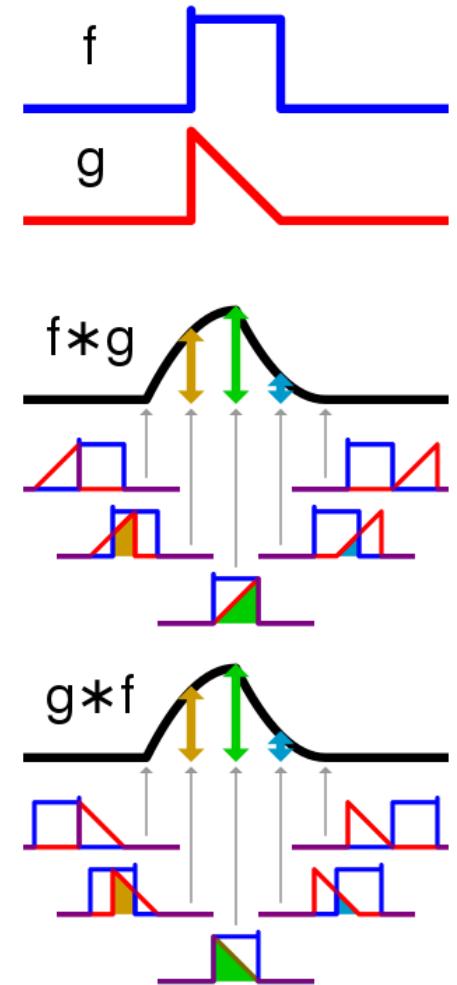
$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau$$

- Note that $(f * g)(t) = (g * f)(t)$
- Terminology:
 - The first argument (here the function f) to the convolution is often referred to as the **input** and
 - The second argument (in this example, the function g) as the **kernel**.
 - The output is sometimes referred to as the **feature map**.

Visual Explanations of Convolution!



Convolution



- <http://fourier.eng.hmc.edu/e161/lectures/convolution/index.html>
- www.wikipedia.org

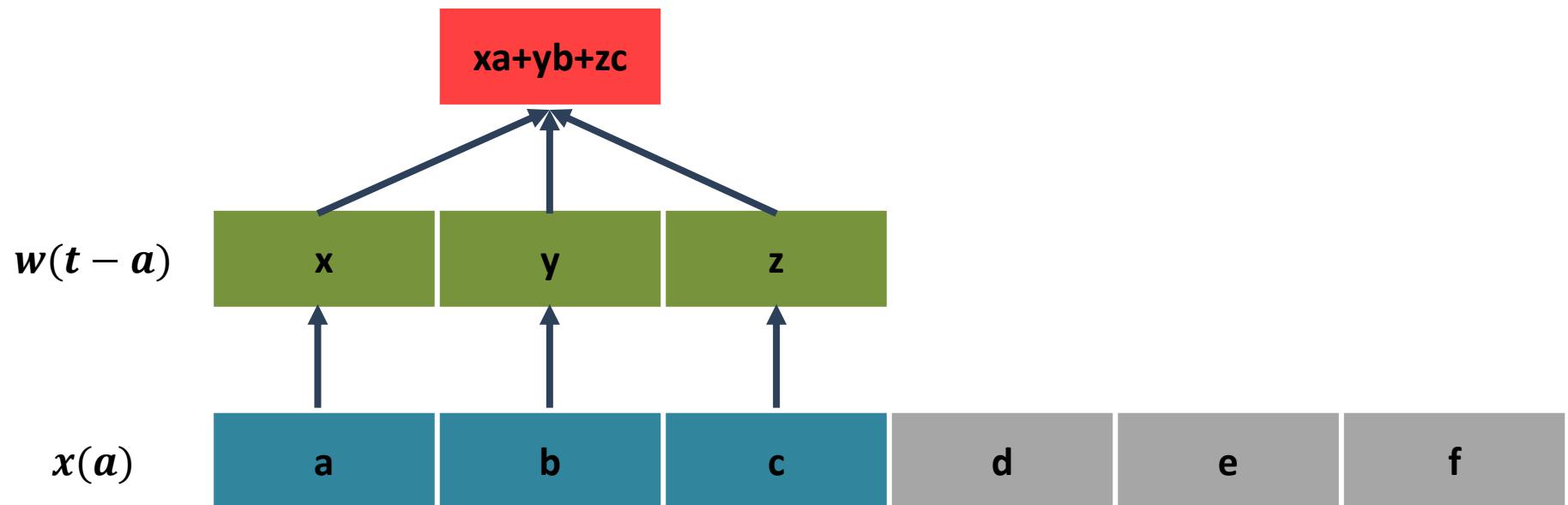
Convolution with Discrete Variables

- Laser sensor may only provide data at regular intervals
- Time index t can take on only integer values
 - x and w are defined only on integer t

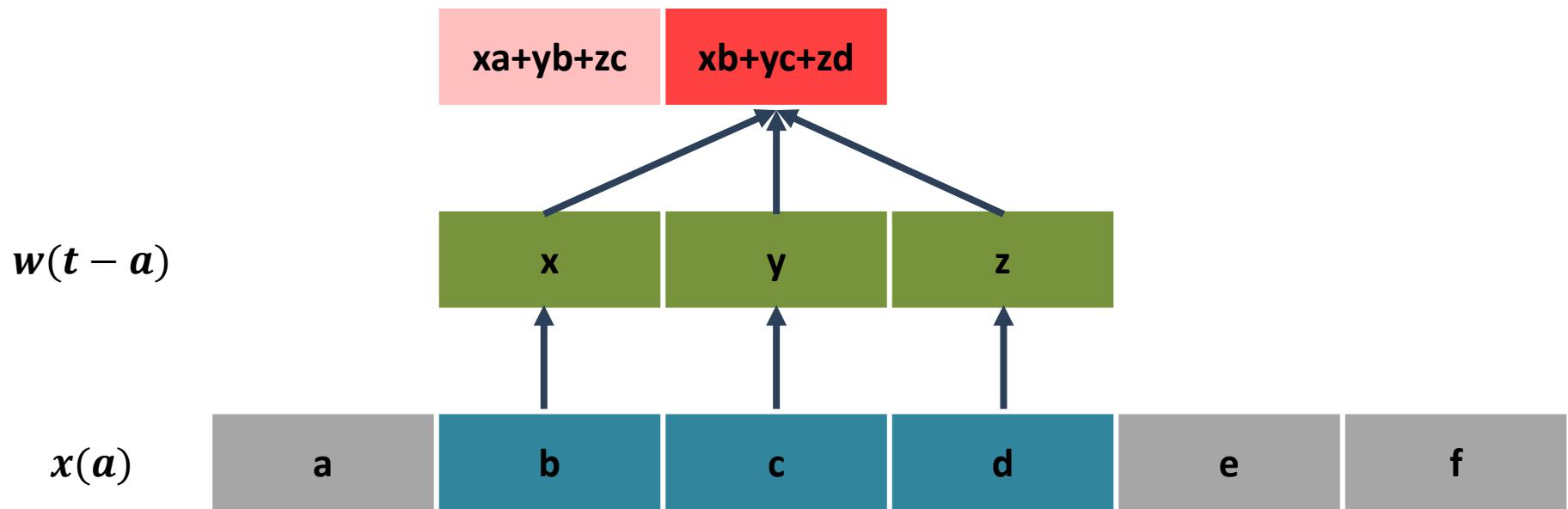
$$s(t) = (x * w)(t) \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

- In ML applications, input is a multidimensional array of data and the kernel is a multidimensional array of parameters that are adapted by the learning algorithm
- Input and kernel are explicitly stored separately

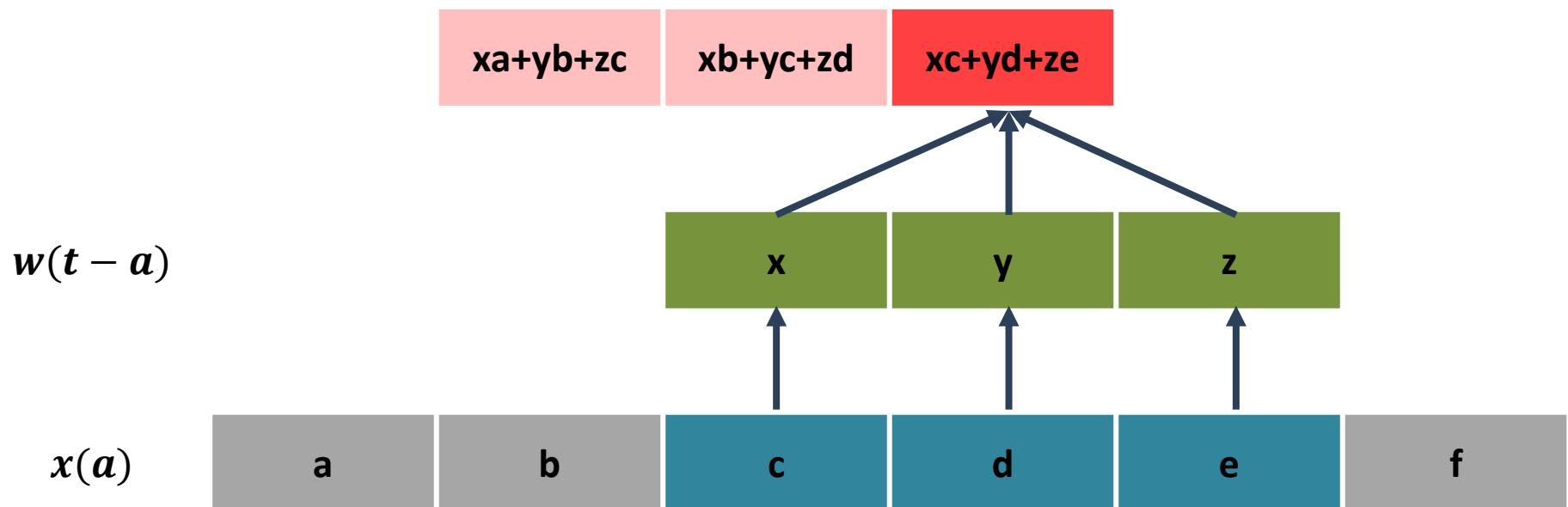
Discrete 1-Dimensional Case



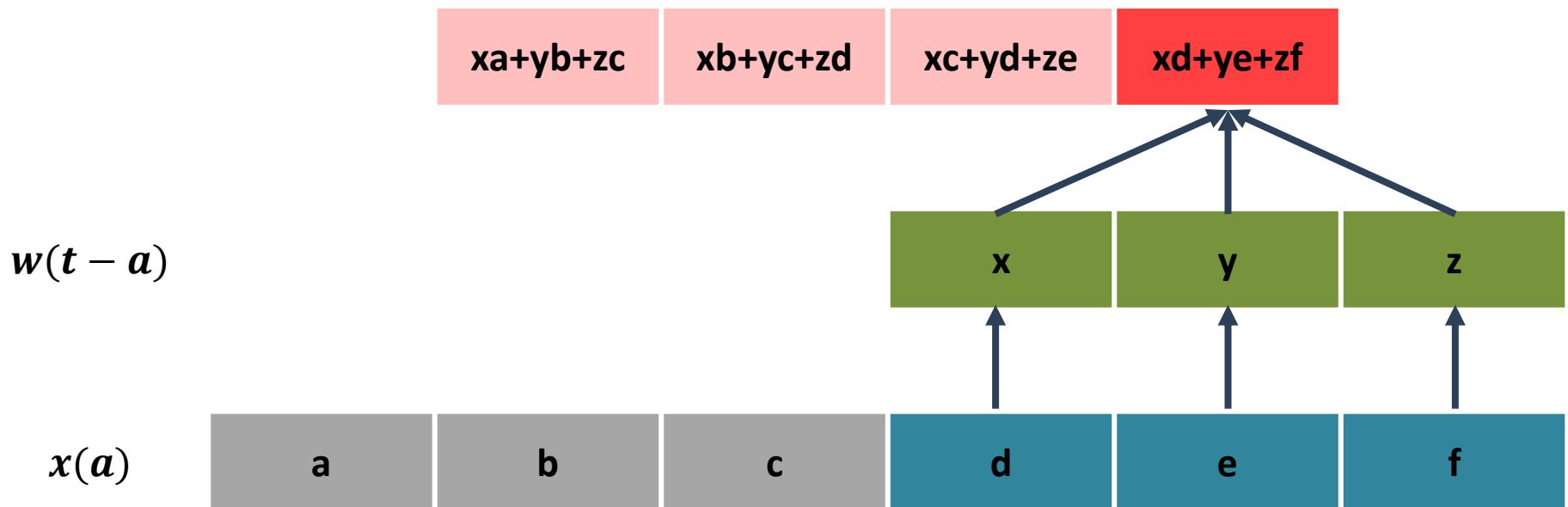
Discrete 1-Dimensional Case



Discrete 1-Dimensional Case

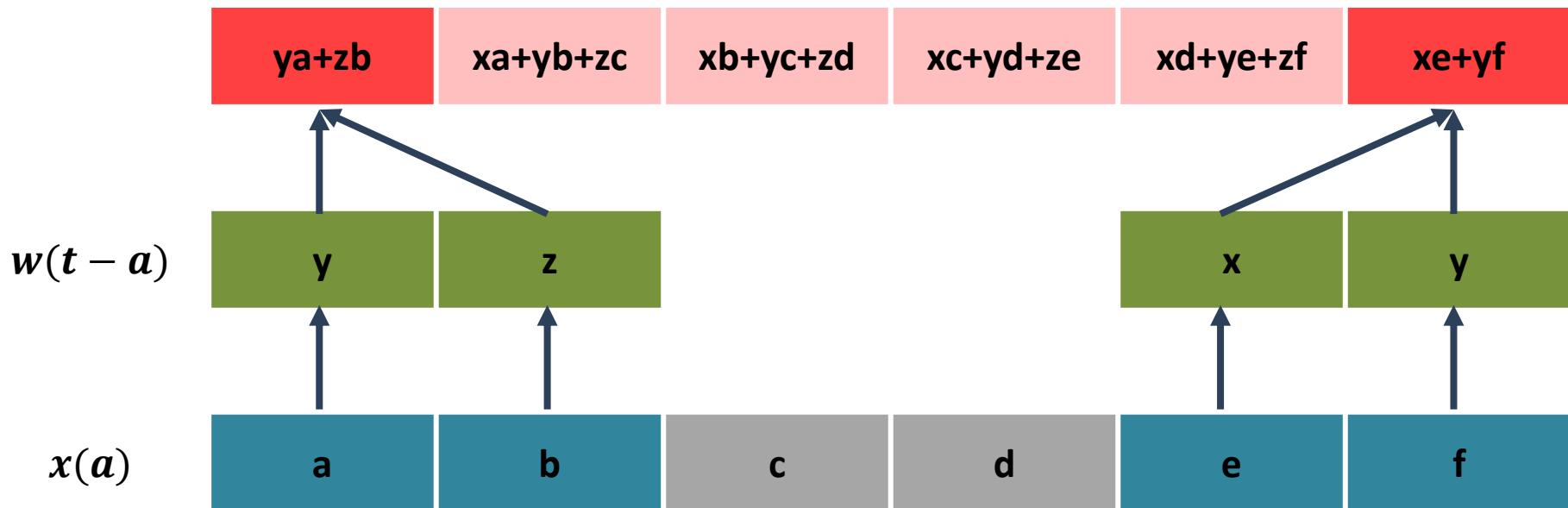


Discrete 1-Dimensional Case

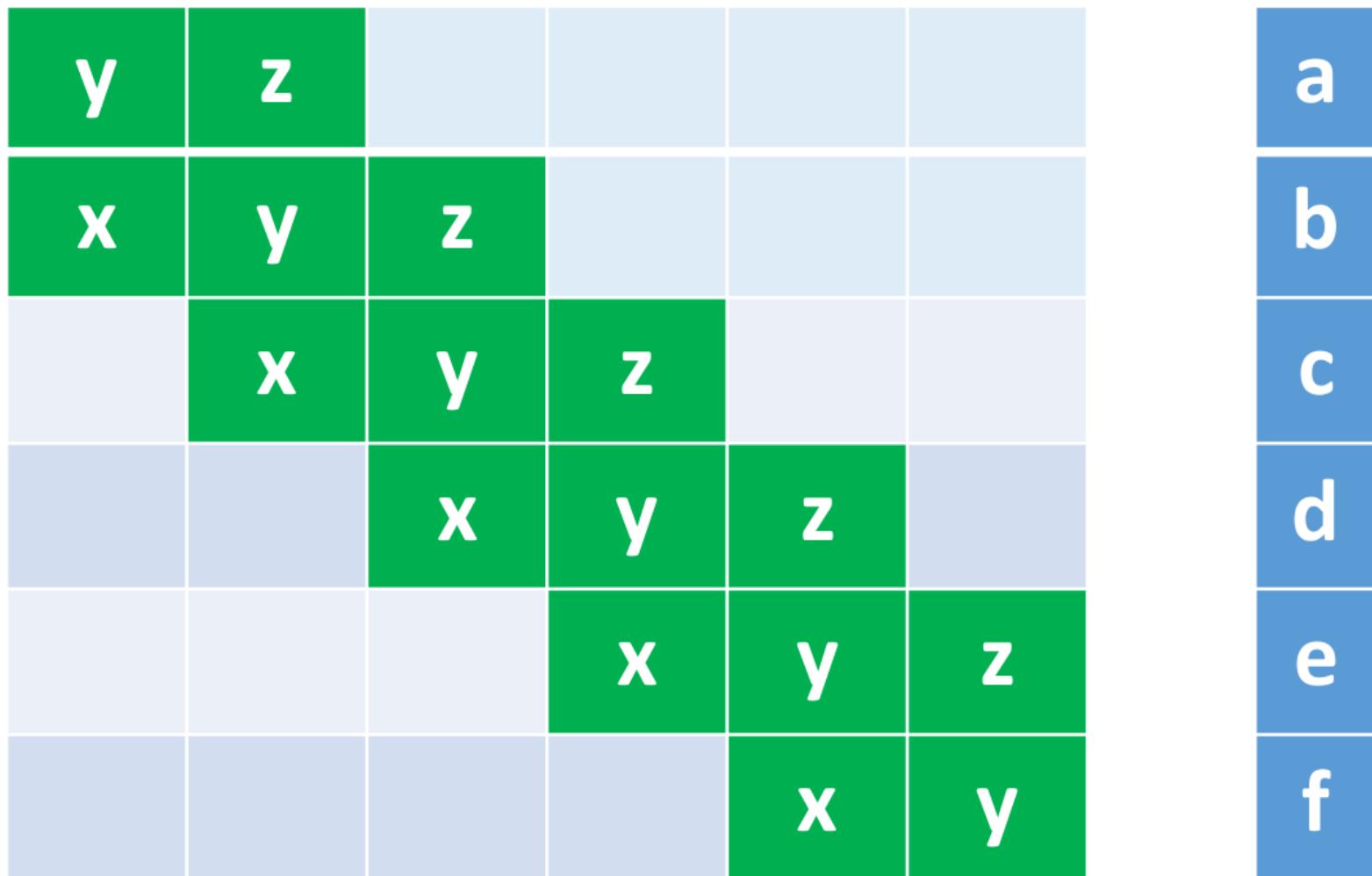


Discrete 1-Dimensional Case: Border Cases

- Border require special treatment
- Most commonly they are ignored
- Sometimes a smaller kernel is used, e.g.:

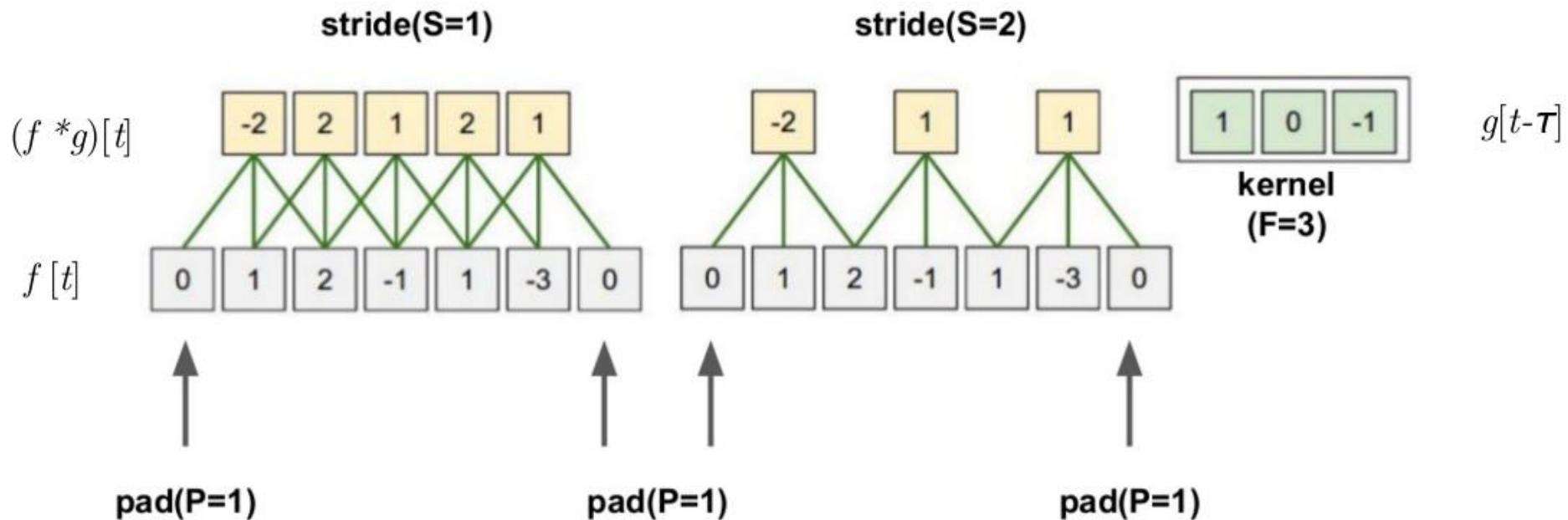


Seen as a Matrix Multiplication



Parameters of Convolution

- Kernel Size (F)
- Padding (P)
- Stride (S)



Two-dimensional Convolution

- If we use a 2D image I as input and use a 2D kernel K we have

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

- Since convolution is commutative, we can also write:

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

- Commutativity arises because we have flipped the kernel relative to the input
 - As m increases, index to the input increases, but index to the kernel decreases

Cross-Correlation

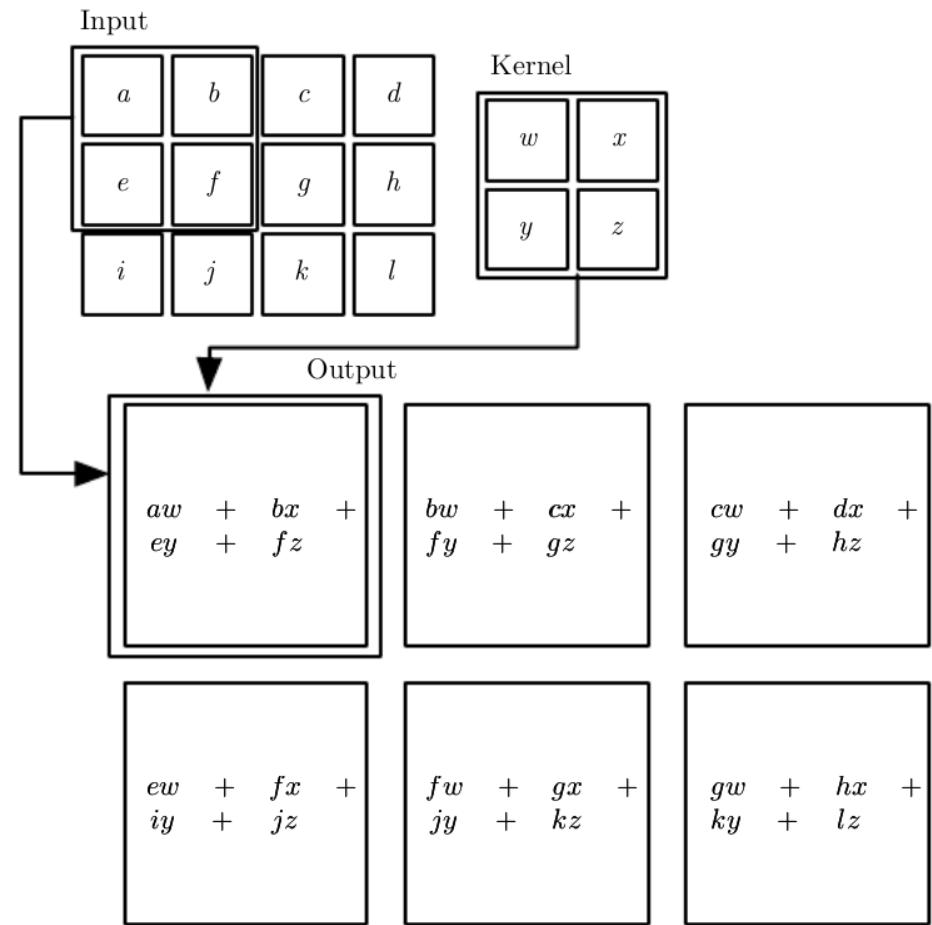
- Same as convolution, but without flipping the kernel

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

- Both referred to as convolution, and whether kernel is flipped or not
- The learning algorithm will learn appropriate values of the kernel in the appropriate place

Example of 2D convolution

- Convolution without kernel flipping applied to a 2D tensor
- Output is restricted to case where kernel is situated entirely within the image
- Arrows show how upper-left of input tensor is used to form upper-left of output tensor



The 2D Case in Matrix Interpretation

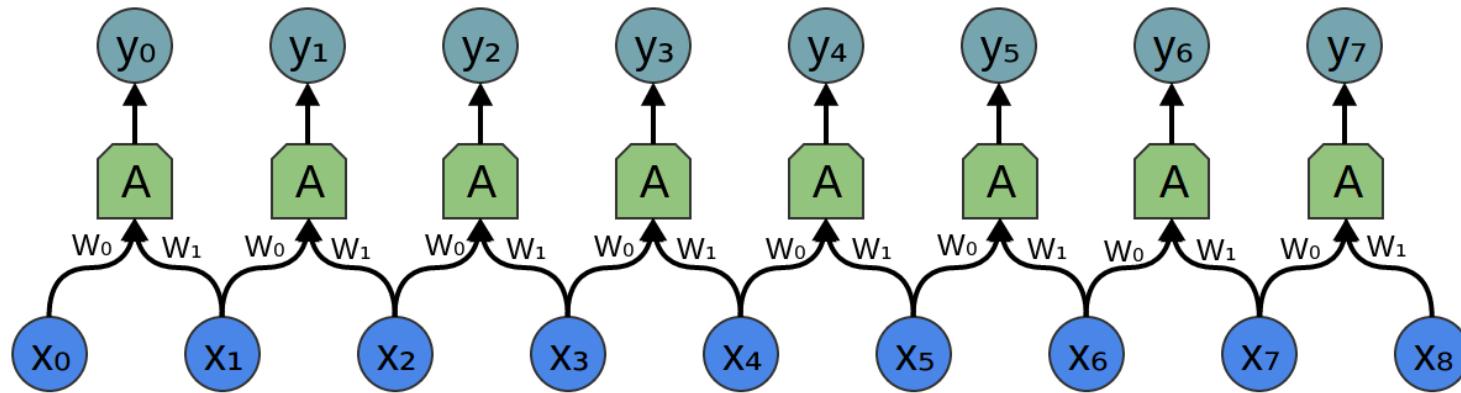
$$C = \begin{bmatrix} c_0 & c_{n-1} & \dots & c_2 & c_1 \\ c_1 & c_0 & c_{n-1} & & c_2 \\ \vdots & c_1 & c_0 & \ddots & \vdots \\ & & & \ddots & \ddots & c_{n-1} \\ c_{n-1} & c_{n-2} & \dots & c_1 & c_0 \end{bmatrix}$$

- Most parameters are shared
- Additionally, the matrix is very sparse

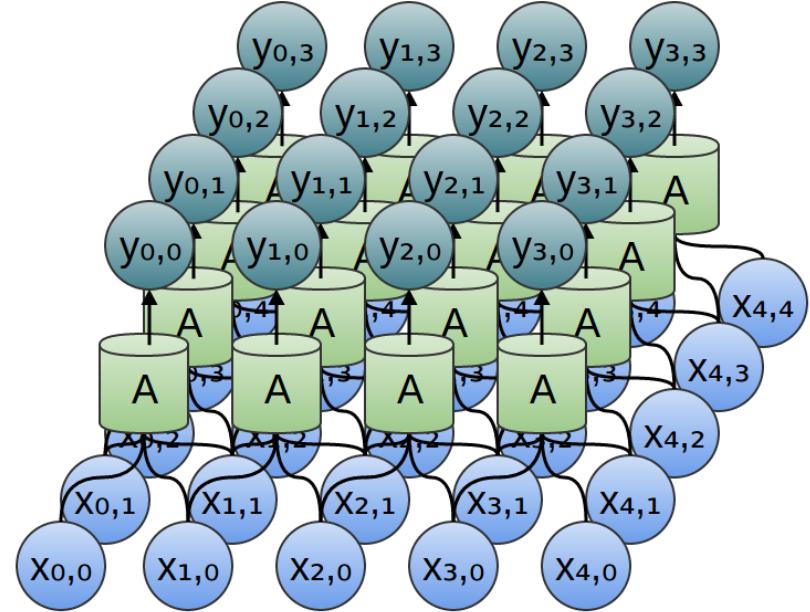
Motivation for using convolution networks

- Convolution leverages three important ideas to improve ML systems:
 1. Sparse interactions
 2. Parameter sharing
 3. Equivariant representations
- Convolution also allows for working with inputs of variable size

Sparse connectivity due to Convolution



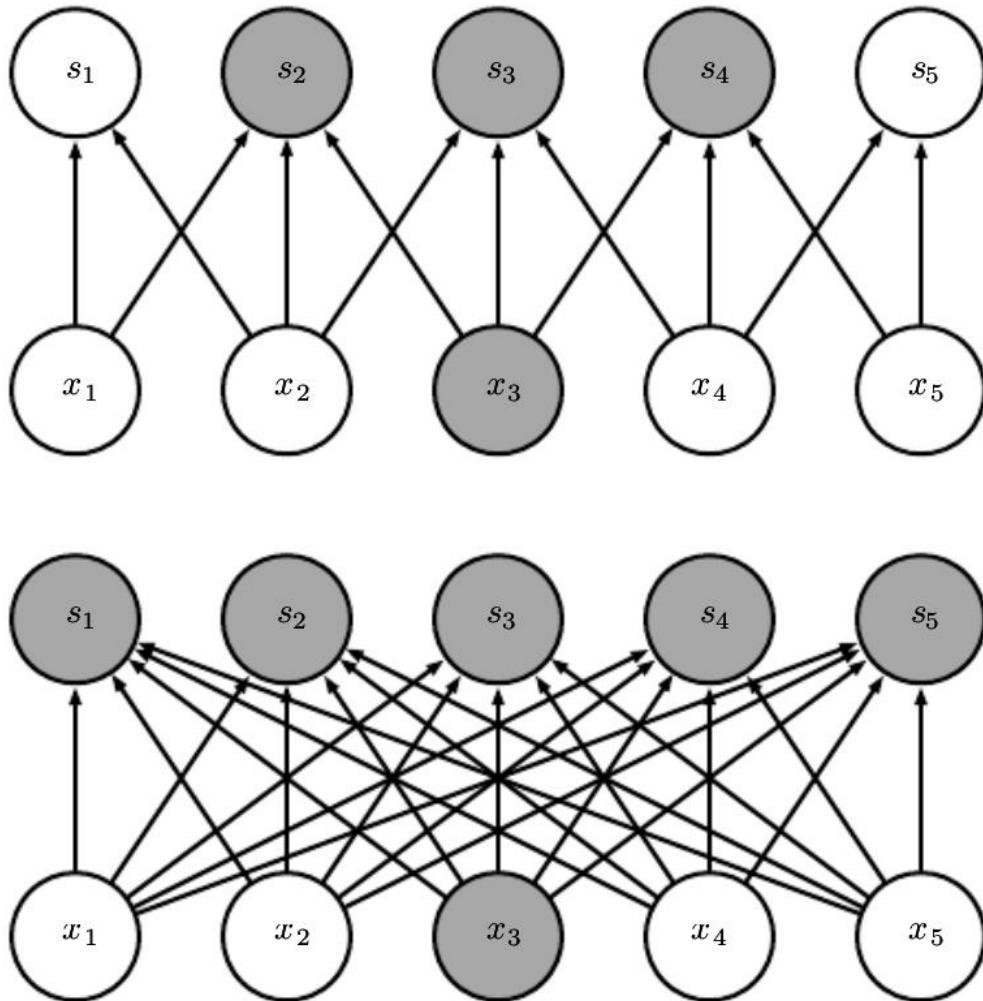
- Instead of learning a full matrix, only a couple of weights of the kernel need to be learned



Sparse Connectivity: Input Perspective

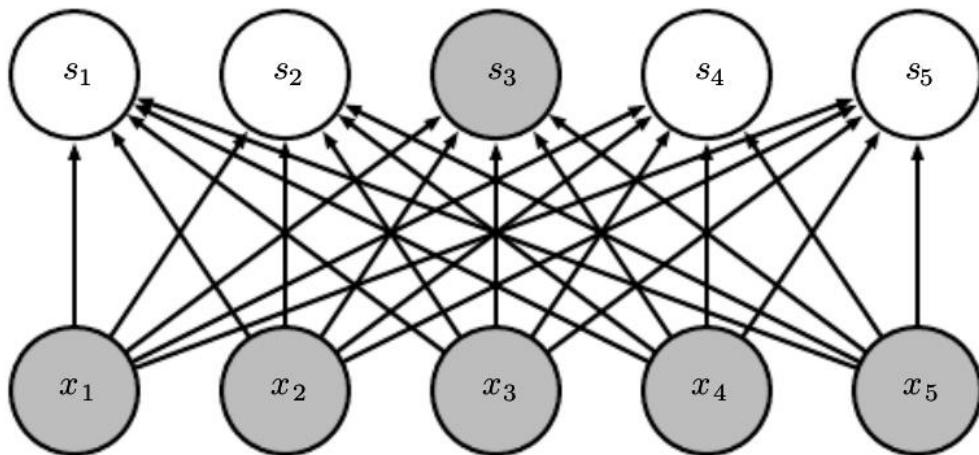
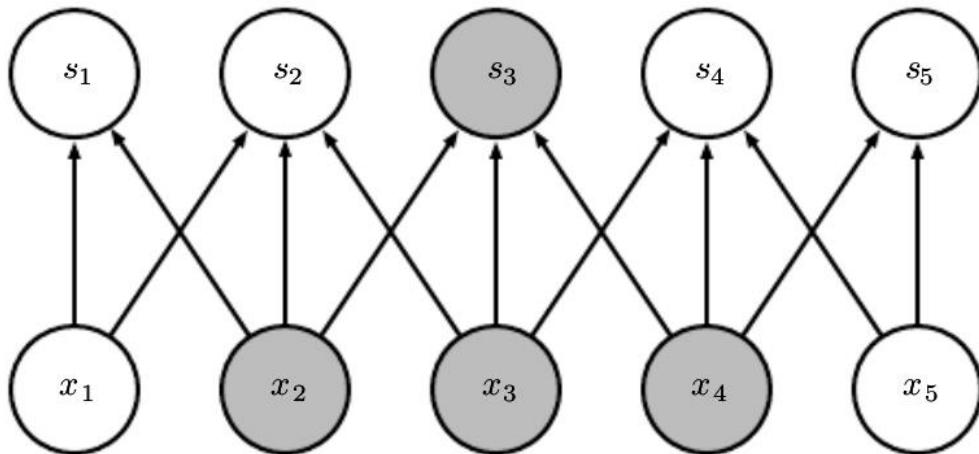
Highlight one input x_3 and output units s affected by it

- Top: when s is formed by convolution with a kernel of width 3, only three outputs are affected by x_3
- Bottom: when s is formed by matrix multiplication connectivity is no longer sparse => All outputs are affected by x_3



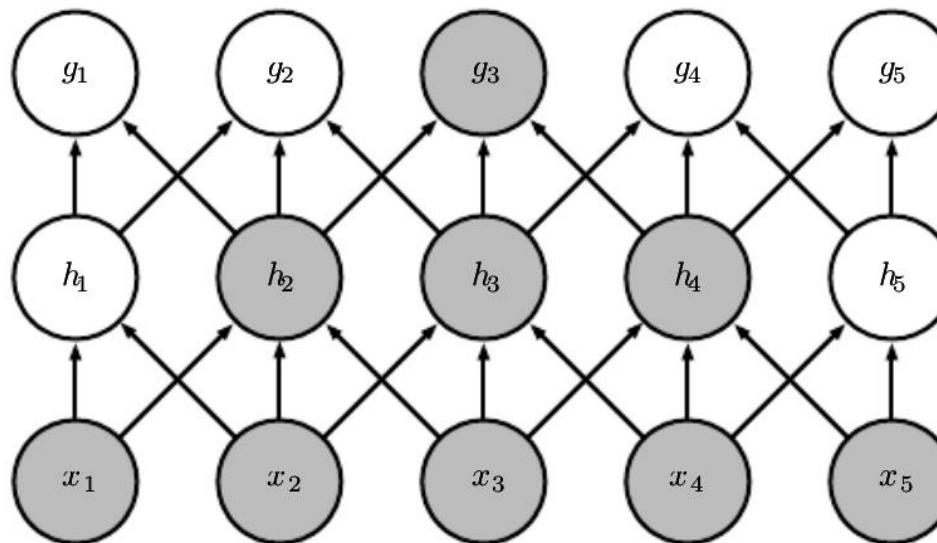
Sparse Connectivity: Output Perspective

Highlight one input s_3 and output units x affected by it



Performance with Reduced Connections

- It is possible to obtain good performance while keeping k several magnitudes lower than m
- In a deep neural network, units in deeper layers may indirectly interact with a larger portion of the input: Depth is the key
- This allows the network to efficiently describe complicated interactions between many variables from simple building blocks that only describe sparse interactions



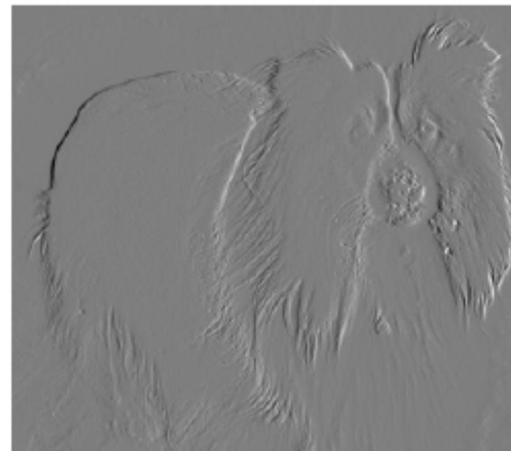
Parameter Sharing

- Parameter sharing refers to using the same parameter for more than one function in a model
- In a traditional neural net each element of the weight matrix is used exactly once when computing the output of a layer
 - It is multiplied by one element of the input and never revisited
- Parameter sharing is synonymous with tied weights
 - Value of the weight applied to one input is tied to a weight applied elsewhere
- In a Convolutional net, each member of the kernel is used in every position of the input (boundaries might be different)

Efficiency of Parameter Sharing

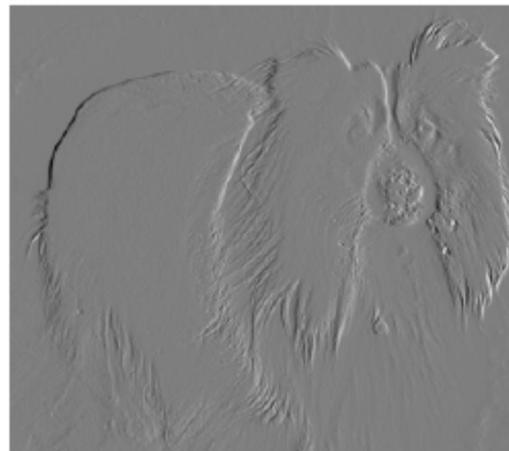
- Parameter sharing by convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set
- This does not affect runtime of forward propagation which is still $O(k \cdot n)$
 - Since m and n are roughly the same size, $k \cdot n$ is much smaller than $m \cdot n$
- But further reduces the storage requirements to k parameters
 - k is orders of magnitude less than m
- Convolution is way better in terms of memory and calculation
 - m are the number of inputs, n the number of outputs, and k the size of the kernel

Example: Efficiency of Convolution for Edge Detection



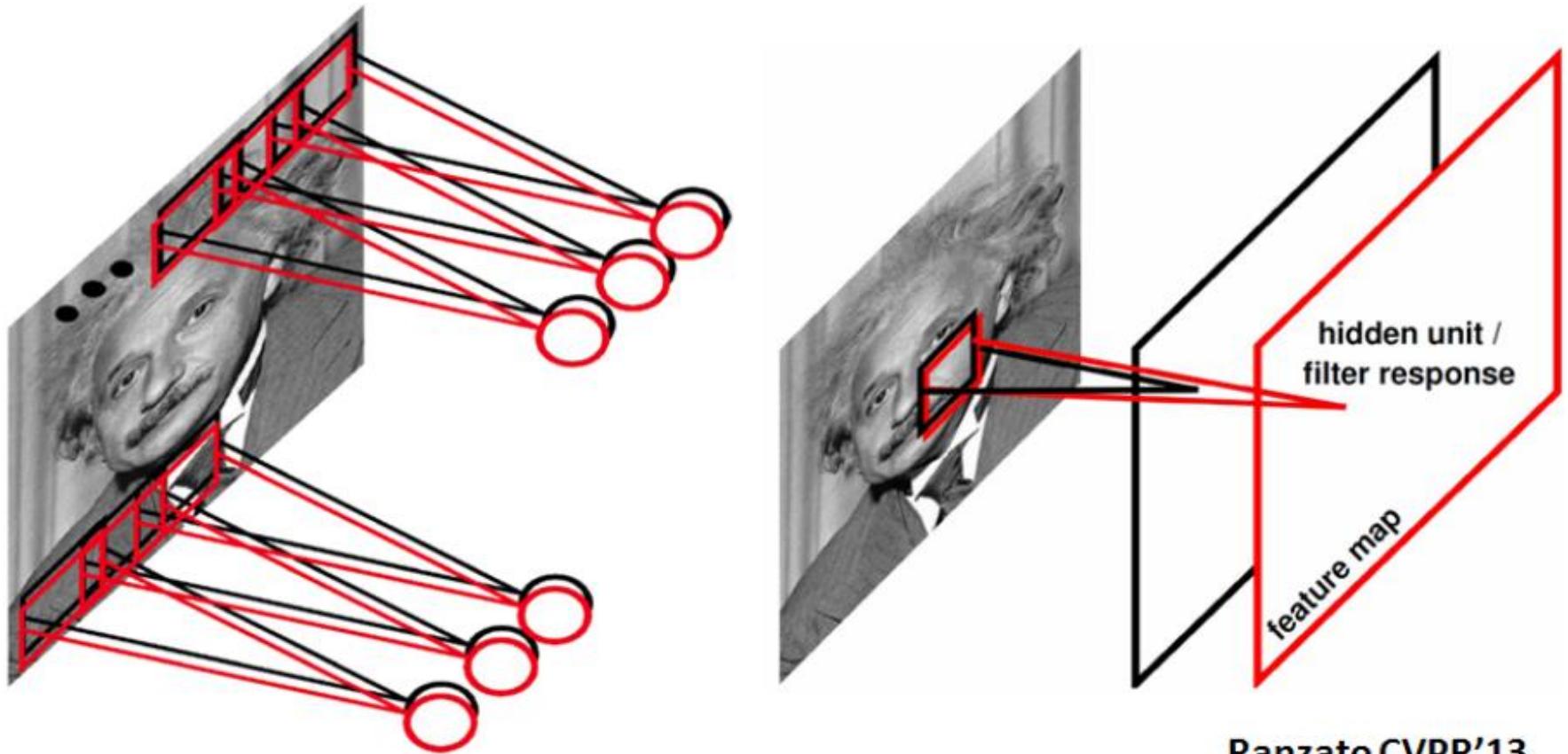
- Image on right formed by taking each pixel of input image and subtracting the value of its neighboring pixel on the left
- Input image is 320x280, output is 319x280
- Computational effort:
$$319 \cdot 320 \cdot 3 = 267,960 \text{ flops}$$
 - Each output pixel is calculated by a convolutional kernel with 2 entries, i.e. 2 multiplications, one add

Example: Efficiency of Convolution for Edge Detection



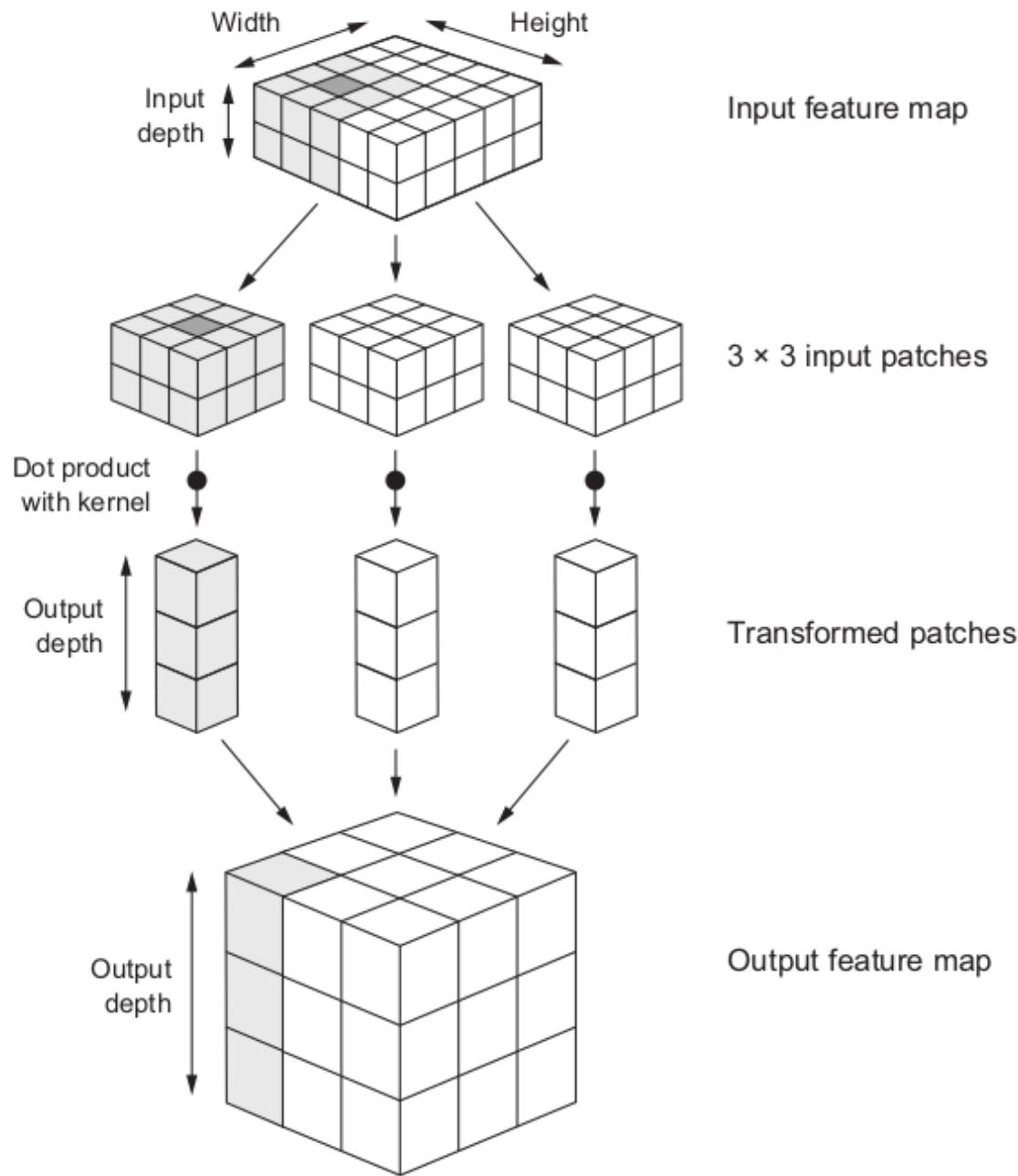
- Same operation with a full matrix with
 $320 \cdot 280 \cdot 319 \cdot 280 = 8 \text{ billion entries}$
- Storing the matrix (double vals): $8 \text{ billion} * 8\text{byte} \approx 60 \text{ Gbyte}$
- Performing the calculation: roughly 60,000 times less efficient computationally

Depth of the Output Volume



Depth of the Output Volume

- The depth of the output volume is a hyperparameter
- It corresponds to the number of filters we would like to use, each learning to look for something different in the input.
- For example, if the first Convolutional Layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color.
- We will refer to a set of neurons that are all looking at the same region of the input as a depth column (some people also prefer the term fiber).



Equivariance of Convolution to Translation

- The particular form of parameter sharing leads to equivariance to translation
 - A function $f(x)$ is equivariant to a function g if $f(g(x)) = g(f(x))$
 - Equivariant means that if the input changes, the output changes in the same way
- If g is a function that translates the input, i.e., that shifts it, then the convolution function is equivariant to g
 - $I(x, y)$ is image brightness at point (x, y)
 - $I' = g(I)$ is image function with $I'(x, y) = I(x - 1, y)$
 - If we apply g to I and then apply convolution, the output will be the same as if we applied convolution to I , then applied transformation g to the output

Example of equivariance

- With 2D images convolution creates a map where certain features appear in the input
- If we move the object in the input, the representation will move the same amount in the output
- Example:
 - It is useful to detect edges in first layer of convolutional network
 - Edges are roughly the same every where in the image
 - So it is practical to share parameters across entire image

Absence of equivariance

- In some cases, we may not wish to share parameters across entire image
 - If image is cropped to be centered on a face, we may want different features from different parts of the face
 - Part of the network processing the top of the face looks for eyebrows
 - Part of the network processing the bottom of the face looks for the chin
- Certain image operations such as scale and rotation are not equivariant to convolution
 - Other mechanisms are needed for such transformations



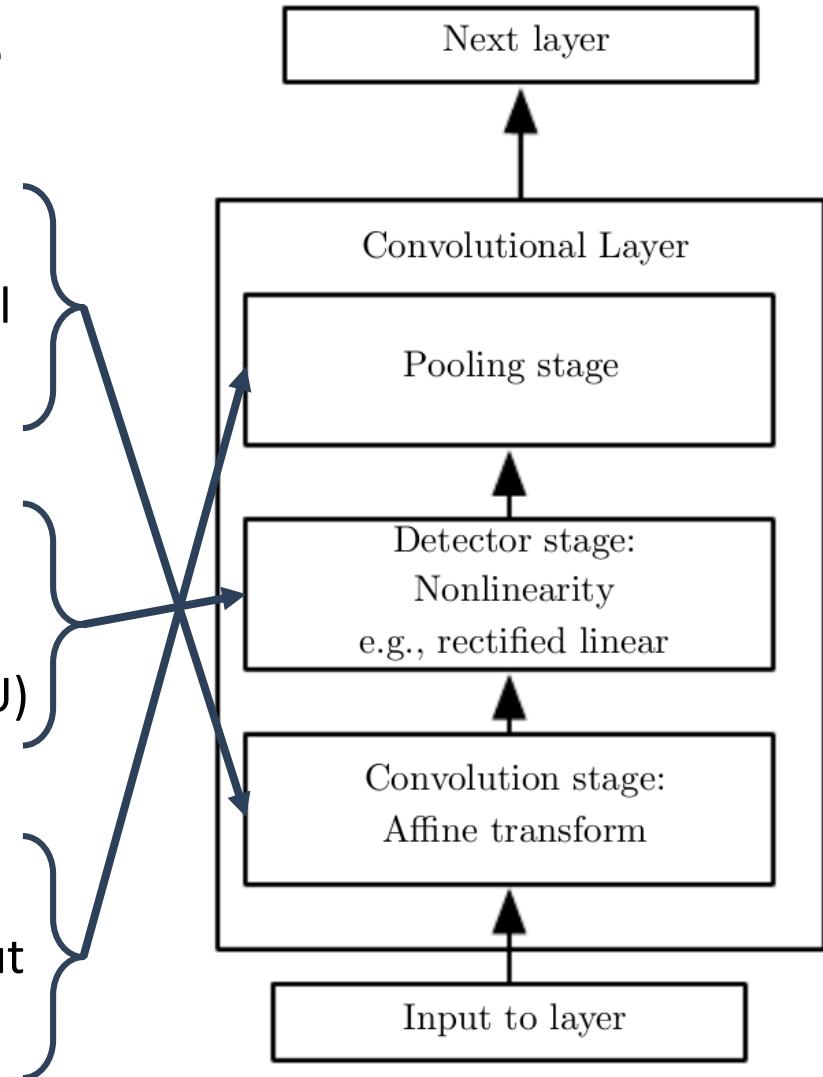
Convolutional Neural Networks

- Convolution
- Pooling
- Variants of Convolution
- Efficient Convolution
- Convolution in Keras

Pooling

Typical layer of a CNN consists of three stages:

- Stage 1 (**Convolution**):
 - Perform several convolutions in parallel to produce a set of linear activations
- Stage 2 (**Detector**):
 - Each linear activation is run through a nonlinear activation function (e.g. ReLU)
- Stage 3 (**Pooling**):
 - Use a pooling function to modify output of the layer further



Types of Pooling functions

- A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby inputs

Popular pooling functions:

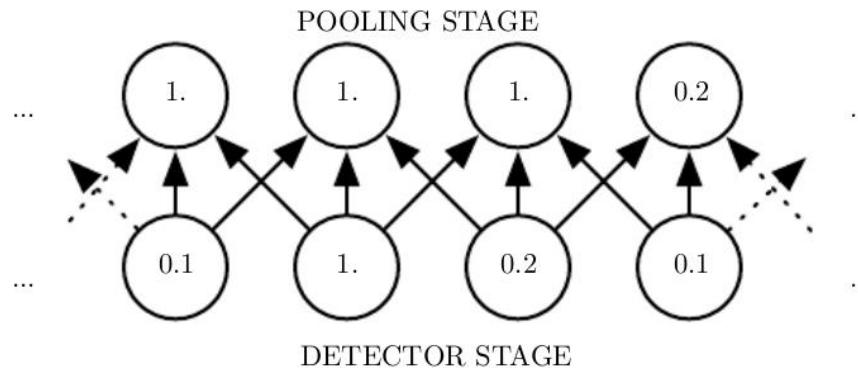
1. Max pooling operation reports the maximum output within a rectangular neighborhood
2. Average of a rectangular neighborhood
3. L^2 norm of a rectangular neighborhood
4. Weighted average based on the distance from the central pixel

Why Pooling?

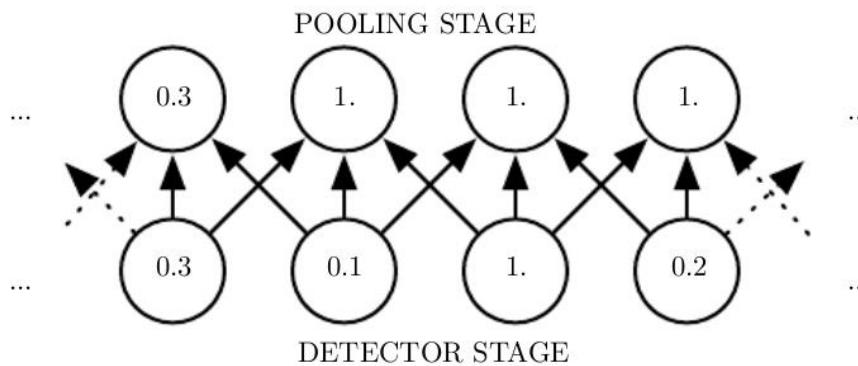
- In all cases, pooling helps make the representation become approximately invariant to small translations of the input
- If we translate the input by a small amount values of most of the outputs does not change
- **Pooling can be viewed as adding a strong prior that the function the layer learns must be invariant to small translations**

Max pooling introduces invariance to translation

- View of middle of output of a convolutional layer



- Same network after the input has been shifted by one pixel



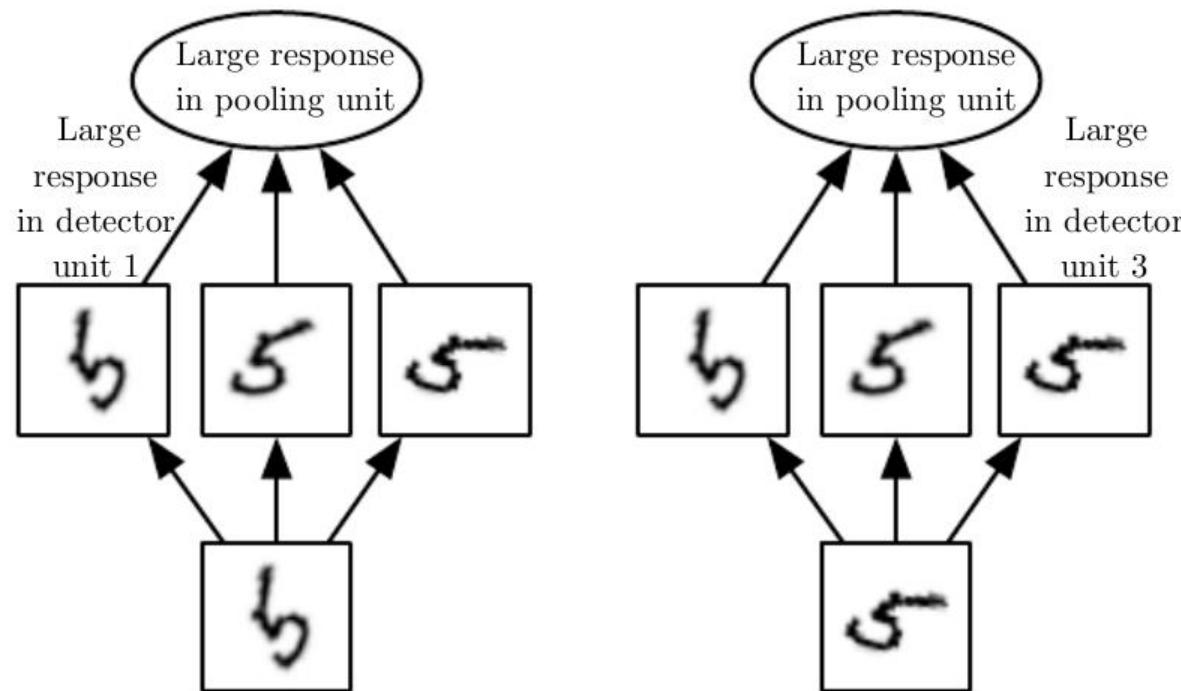
- Every input value has changed, but only half the values of output have changed

Importance of Translation Invariance

- Invariance to translation is important if we care about whether a feature is present rather than exactly where it is
 - For detecting a face we just need to know that an eye is present in a region, not its exact location
- In other contexts it is more important to preserve location of a feature
 - E.g., to determine a corner we need to know whether two edges are present and test whether they meet

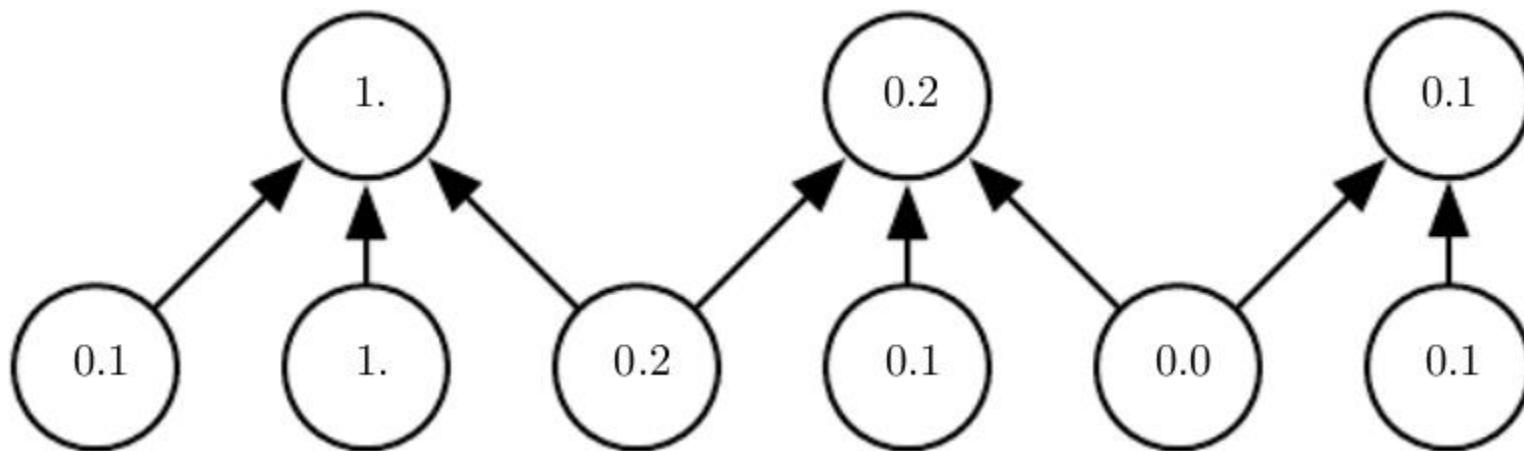
Learning other invariances

- Pooling over spatial regions produces invariance to translation
- But if we pool over the results of separately parameterized convolutions, the features can learn which transformations to become invariant to

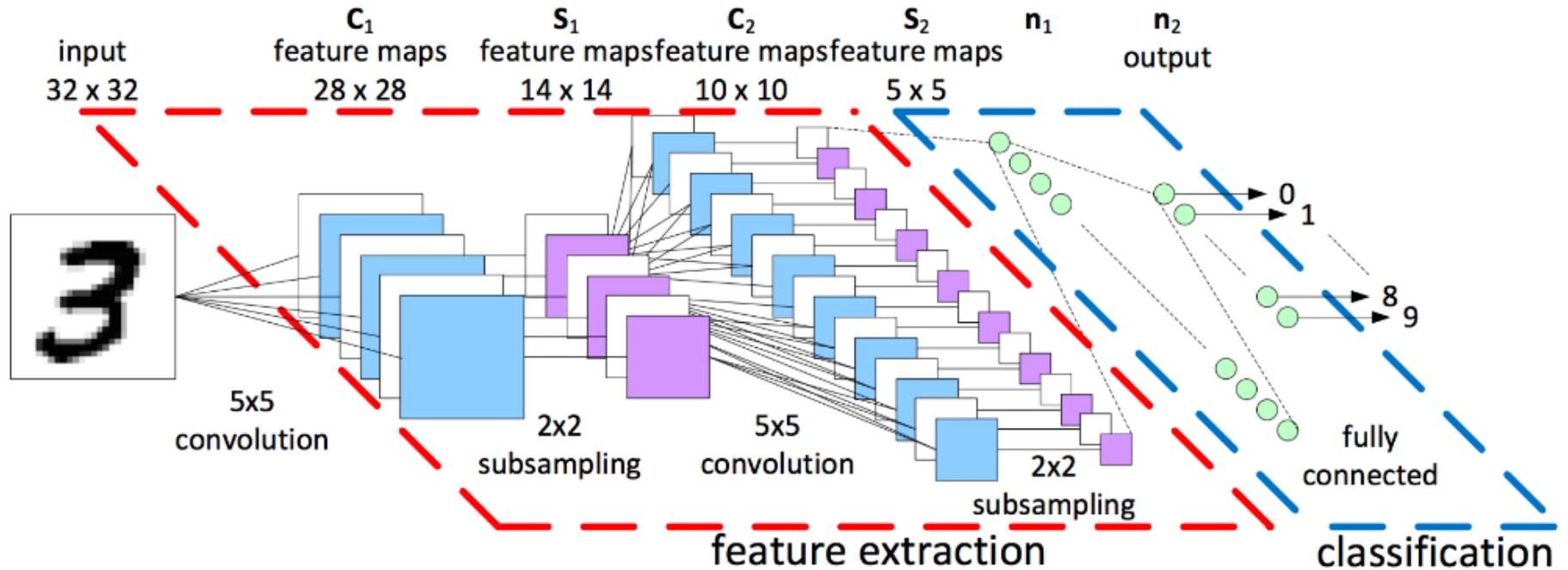


Pooling with Down-Sampling

- Because pooling summarizes the responses over a whole neighborhood, it is possible to use fewer pooling units than detector units
 - By reporting summary statistics for pooling regions spaced k pixels apart rather than one pixel apart
 - Next layer has k times fewer inputs to process

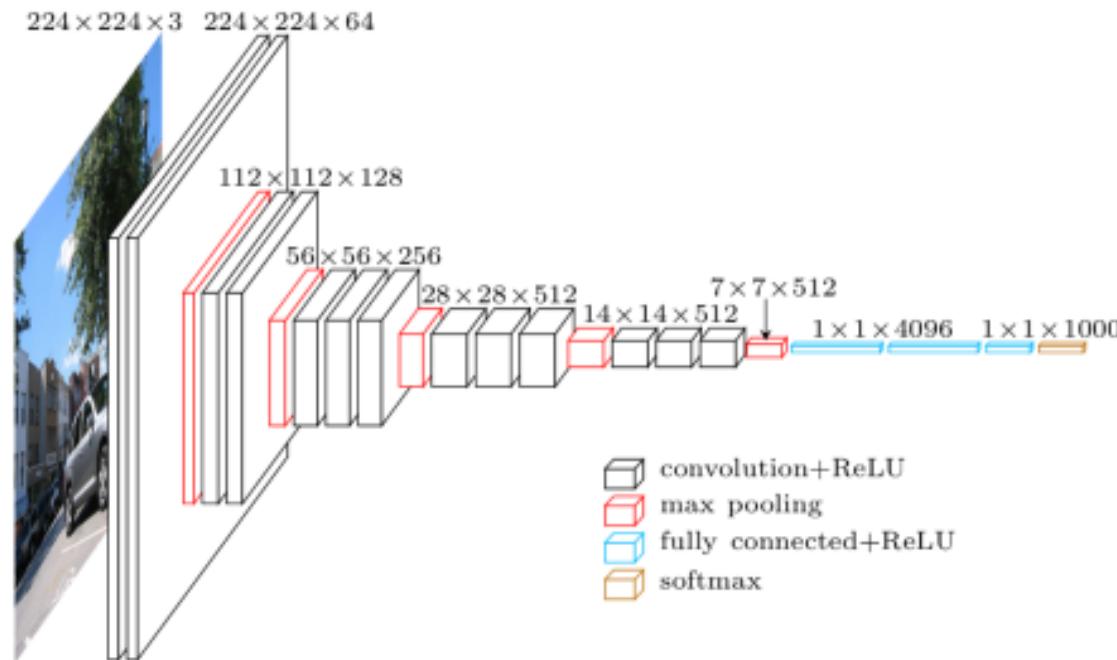


Example for a Convolutional Network



VGG Net

- VGG is a convolutional neural network model
 - “Very Deep Convolutional Networks for Large-Scale Image Recognition”
- The model achieves 93% top-5 test accuracy in ImageNet which is a dataset of over 14 million images belonging to 1000 classes.





Convolutional Neural Networks

- Convolution
- Pooling
- **Variants of Convolution**
- Efficient Convolution
- Convolution in Keras

Convolution Operation in Neural Networks

- Convolution in the context of neural networks does not refer exactly to the standard convolution operation in mathematics
 1. It refers to an operation that consists of many applications of convolution in parallel
 - This is because convolution with a single kernel can only extract one kind of feature, albeit at many locations
 - Usually we want to extract many kinds of features at many locations
 2. Input is usually not a grid of real values
 - Rather it is a vector of observations, e.g., a color image has R, G, B values at each pixel
 - Input to the next layer is the output of the first layer which has many different convolutions at each position
 - When working with images, input and output are 3-D tensors

Convolution Operation in Neural Networks

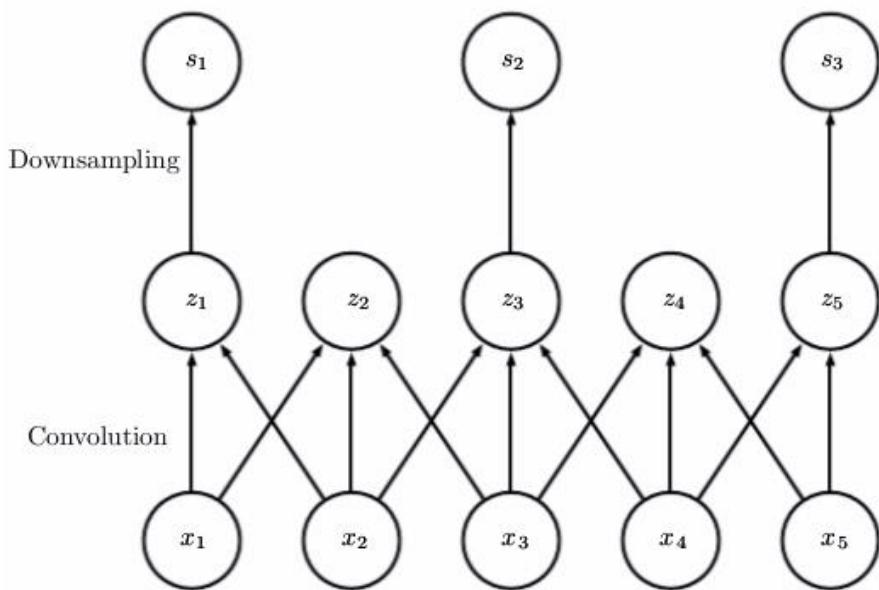
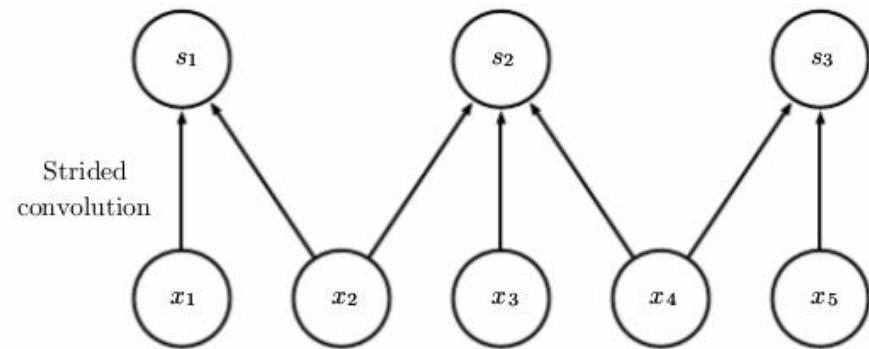
- Four indices with image software
 1. One index for Channel (RGB)
 2. Two indices for spatial coordinates of each channel (x,y)
 3. Fourth index for different samples in a batch (we omit that one)
- Because we are dealing with multichannel convolution, linear operations are usually not commutative, even when kernel flipping is used
- These multi-channel operations are only commutative if each operation has the same number of output channels as input channels.

Definition of 4-D kernel tensor

- Assume we have a 4-D kernel tensor \mathbf{K} with element $K_{i,j,l,k}$ giving the connection strength between
 - a unit in channel i of the output and
 - a unit in channel j of the input
 - with an offset of k rows and l columns between output and input units
- Assume our input consists of observed data \mathbf{V} with element $V_{i,j,k}$
 - within channel i at row j and column k
- Assume, output \mathbf{Z} is produced by convolving \mathbf{K} across \mathbf{V}

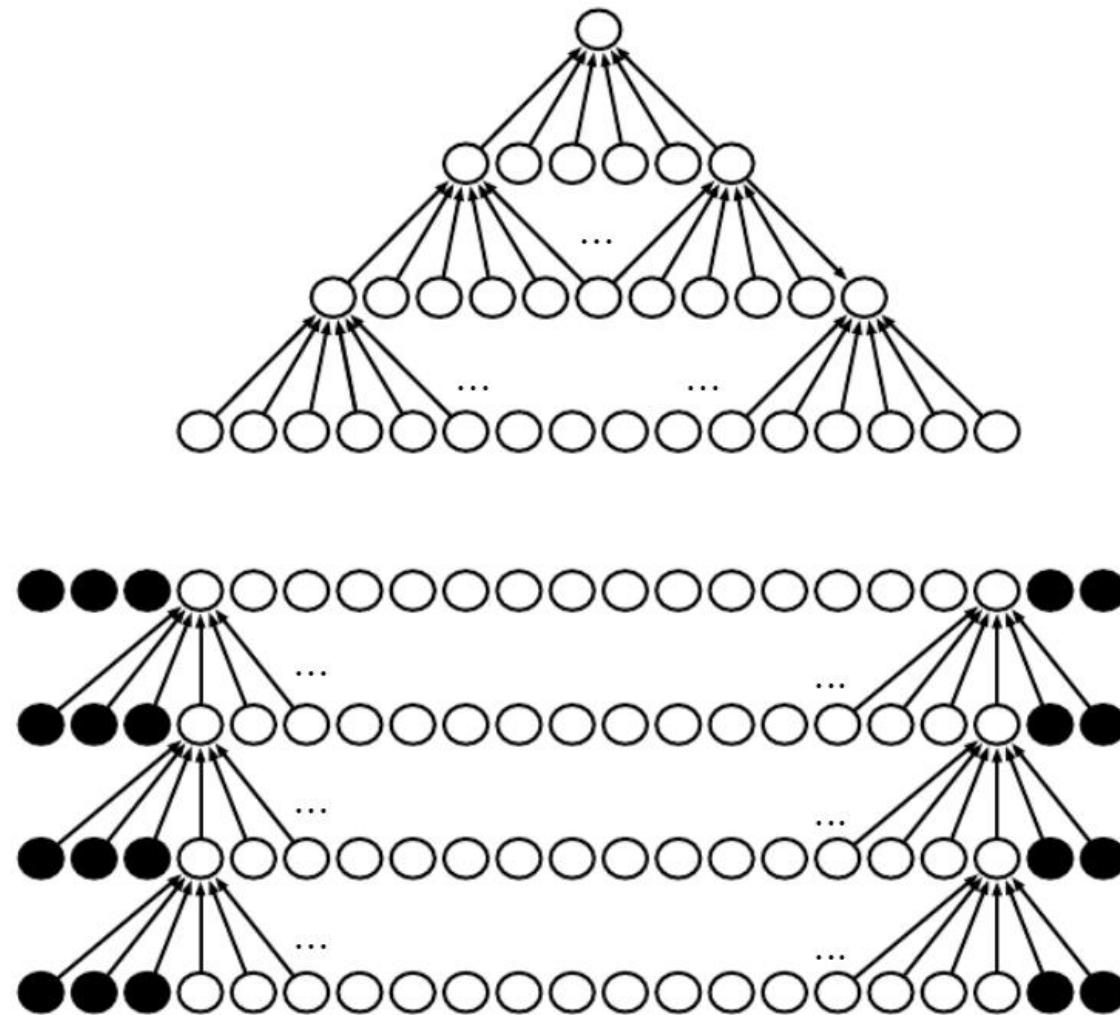
$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n}$$

Convolution with a stride: Implementation



- Strided convolution is equivalent to normal convolution followed by a downsampling
- Second approach is computationally wasteful

Effect of Zero-padding on network size



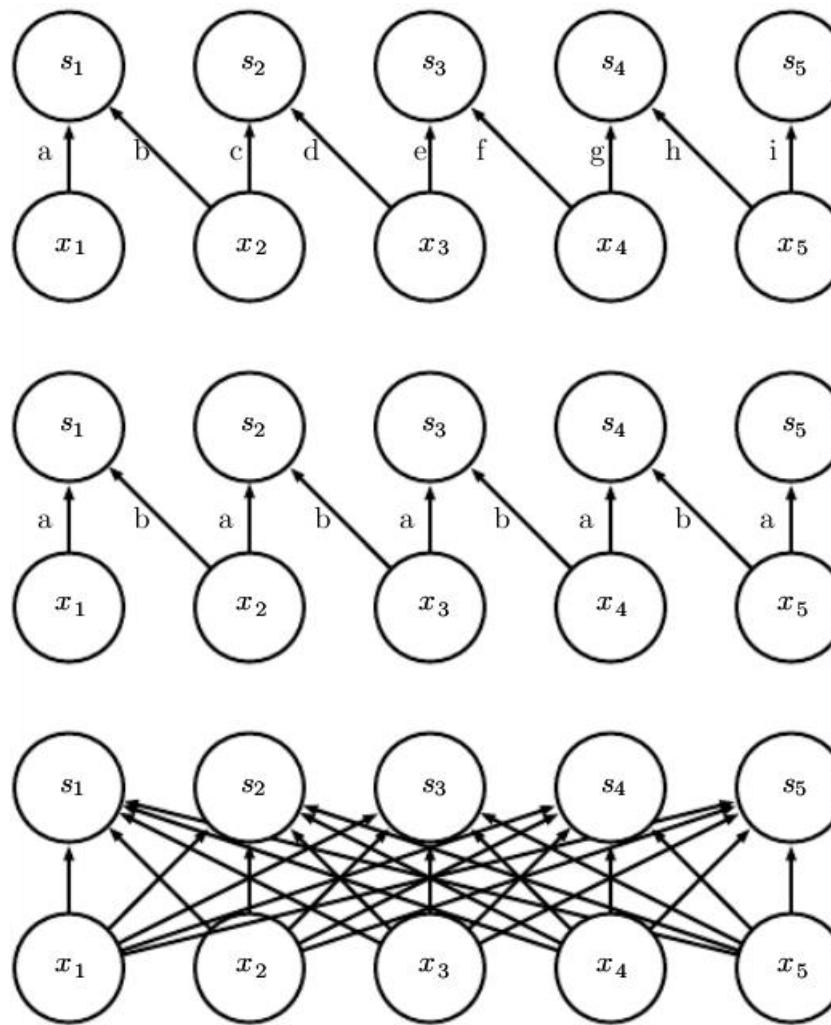
Locally connected layer

- In some cases, we do not actually want to use convolution, but rather sparsely connected layers
 - adjacency matrix in the graph of the network is the same, but every connection has its own weight, specified by a 6-D tensor \mathbf{W} :
 - i , the output channel
 - j , the output row,
 - k , the output column,
 - l , the input channel,
 - m , the row offset within the input, and
 - n , the column offset within the input.

$$Z_{i,j,k} = \sum_{l,m,n} [V_{l,j+m-1,k+n-1} w_{i,j,k,l,m,n}]$$

- Also called unshared convolution

Local Connections, Convolution, Full connections



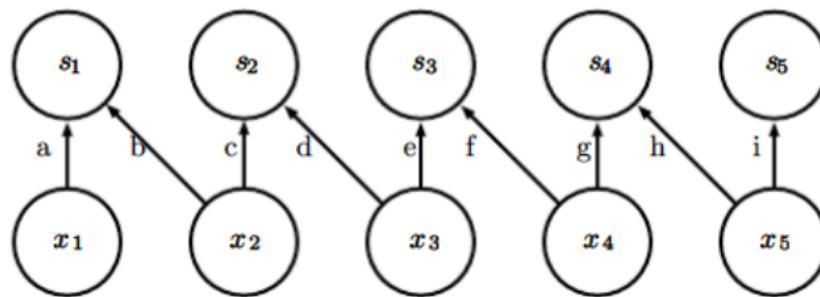
Use of Locally Connected Layers

- Locally connected layers are useful when we know that each feature should be a function of a small part of space, but there is no reason to think that the same feature should occur across all of space
- Ex: if we want to tell if an image is a picture of a face, we only need to look for the mouth in the bottom half of the image

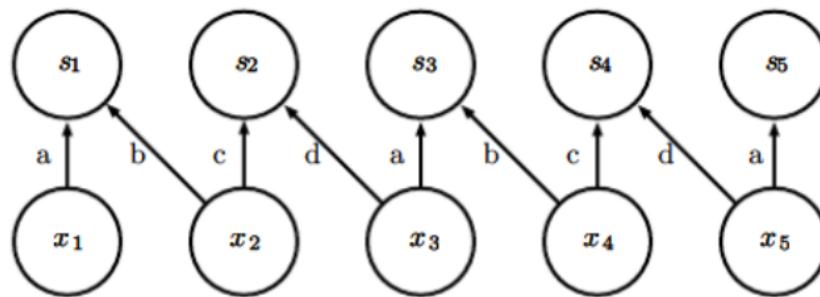
Tiled Convolution

- Compromise between a convolutional layer and a locally connected layer.
 - Rather than learning a separate set of weights at every spatial location, we learn a set of kernels that we rotate through as we move through space.
- This means that immediately neighboring locations will have different filters, like in a locally connected layer,
 - but the memory requirements for storing the parameters will increase only by a factor of the size of this set of kernels
 - rather than the size of the entire output feature map.

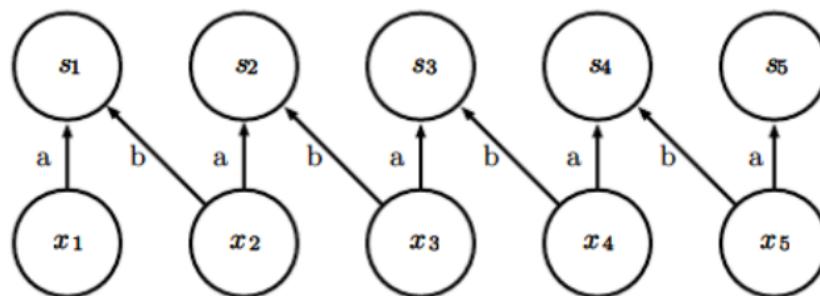
Locally Connected vs. Tiled Convolution vs. Convolution



A locally connected layer
Has no sharing at all
Each connection has its own weight



Tiled convolution
Has a set of different kernels
With $t=2$



Traditional convolution
Equivalent to tiled convolution
with $t=1$
There is only one kernel and it is
applied everywhere



Convolutional Neural Networks

- Convolution
- Pooling
- Variants of Convolution
- Efficient Convolution
- Convolution in Keras

Speed-Up Convolution

- Typically, the most expensive part of convolutional network training is learning the features
- The output layer is usually relatively inexpensive (small number of features after passing through pooling)
- When performing supervised training with gradient descent, every gradient step requires a complete run of forward propagation and backward propagation through the entire network.
- One way to reduce the cost of convolutional network training is to use features that are not trained in a supervised fashion.

Obtaining Kernels without Training

There are three basic strategies for obtaining convolution kernels without supervised training.

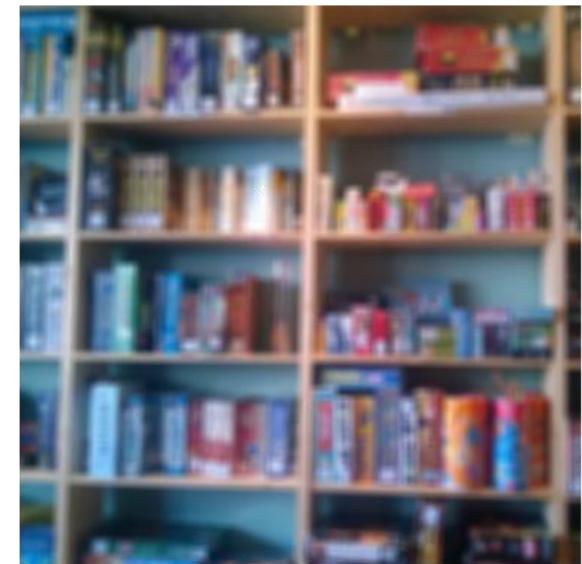
1. Simply initialize them randomly
2. Design them by hand, for example by setting each kernel to detect edges at a certain orientation or scale
3. Learn the kernels with an unsupervised criterion

Random Initialization

- Random filters often work surprisingly well in convolutional networks
- It was shown that layers consisting of convolution followed by pooling naturally become frequency selective and translation invariant when assigned random weights
- Strategy:
 - first evaluate the performance of several convolutional network architectures by training only the last layer
 - take the best of these architectures and train the entire architecture using a more expensive approach.

Blurring Filter

$$K_{box} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \hat{K}_{box} = \begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$



the same image without any filtering, after a 3x3 box blur and after a 9x9 box blur

- <https://www.taylorpetrick.com/blog/post/convolution-part3>

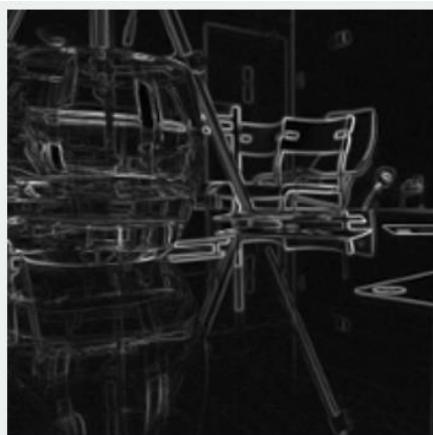
Edge Detection

■ Simple

$$\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$$

■ Sobel

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

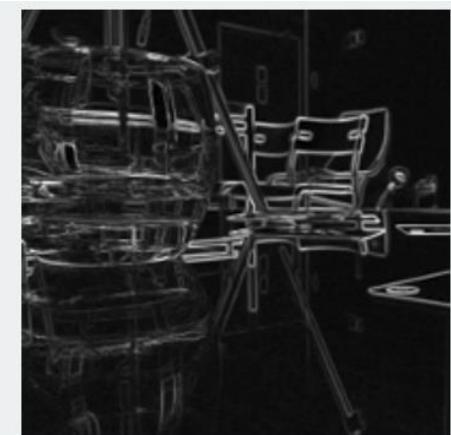
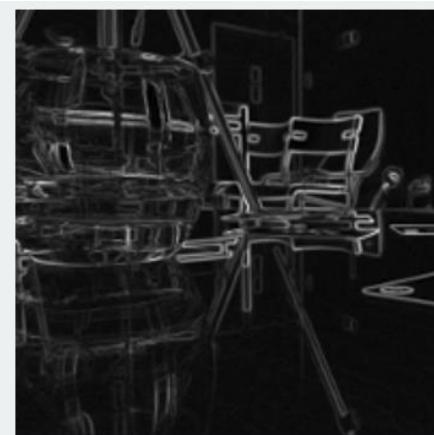


■ Prewitt

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{bmatrix} * \mathbf{A}$$

■ Kirsch (8 directions)

$$\mathbf{g}^{(1)} = \begin{bmatrix} +5 & +5 & +5 \\ -3 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix}, \quad \mathbf{g}^{(2)} = \begin{bmatrix} +5 & +5 & -3 \\ +5 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix}, \quad \mathbf{g}^{(3)} = \begin{bmatrix} +5 & -3 & -3 \\ +5 & 0 & -3 \\ +5 & -3 & -3 \end{bmatrix}$$



comparison of simple, sobel, prewitt and kirsch edge detection filters

➤ <https://www.taylorpetrick.com/blog/post/convolution-part3>

Sharpen

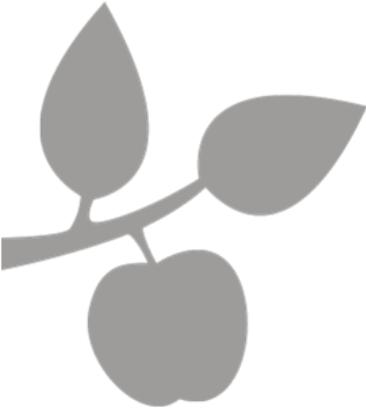
$$K_{sharp} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} * amount$$



comparison of unfiltered image and sharpened images with amount=2 and amount=8

Learn Kernels Unsupervised

- For example, k-means clustering to small image patches
- Use each learned centroid as a convolution kernel
- Learning the features with an unsupervised criterion allows them to be determined separately from the classifier layer at the top of the architecture.
- One can then extract the features for the entire training set just once, essentially constructing a new training set for the last layer



DM873

Deep Learning

Spring 2019

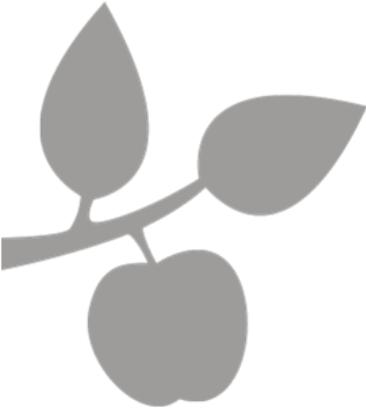
Lecture 9 – Regularization

What is Regularization?

- Central problem of ML is to design algorithms that will perform well not just on training data but on new inputs as well
- Regularization is:
 - “any modification we make to a learning algorithm to reduce its generalization error but not its training error”
 - Reduce test error even at the expense of increasing training error
- Some goals of regularization
 1. Encode prior knowledge
 2. Express preference for simpler model
 3. Needed to make underdetermined problem determined

Model Types and Regularization

- Three types of model families
 1. Excludes the true data generating process (Implies underfitting and inducing high bias)
 2. Matches the true data generating process
 3. Overfits (Includes true data generating process but also many other processes)
- Goal of regularization is to take model from third regime to second
- Best fitting model obtained not by finding the right number of parameters
- Instead, best fitting model is a large model that has been regularized appropriately



Regularization

- **Parameter Penalties**
- **Data Augmentation**
- **Noise Robustness**
- **Parameter Sharing / Semi-Supervised / Multitasking**
- **Early stopping**
- **Bagging**
- **Dropout**
- **Adversarial training**

Limiting Model Capacity

- Regularization has been used for decades prior to advent of deep learning
- Linear- and logistic-regression allow simple, straightforward and effective regularization strategies:
 - Adding a parameter norm penalty $\Omega(\theta)$ to the objective function J :
- with α is a hyperparameter that weight the relative contribution of the norm penalty term Ω
- Setting α to 0 results in no regularization. Larger values correspond to more regularization

Norm Penalty

- When our training algorithm minimizes the regularized objective function J and some measure of the size of the parameters θ
- Different choices of the parameter norm Ω can result in different solutions preferred
- Norm penalty Ω penalizes only weights at each layer and leaves biases unregularized
 - Biases require less data to fit than weights
 - Each bias controls only a single variable
- Let w indicate all weights affected by norm penalty, θ denotes both w and biases

L^2 parameter Regularization

- Simplest and most common kind
- Called Weight decay
- Drives weights closer to the origin by adding a regularization term to the objective function

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

- In other communities also known as **ridge regression** or Tikhonov regularization

A closer look

- Objective function (with no bias parameter)

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \frac{\alpha}{2} \mathbf{w}^T \mathbf{w} + J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

- Corresponding Gradient:

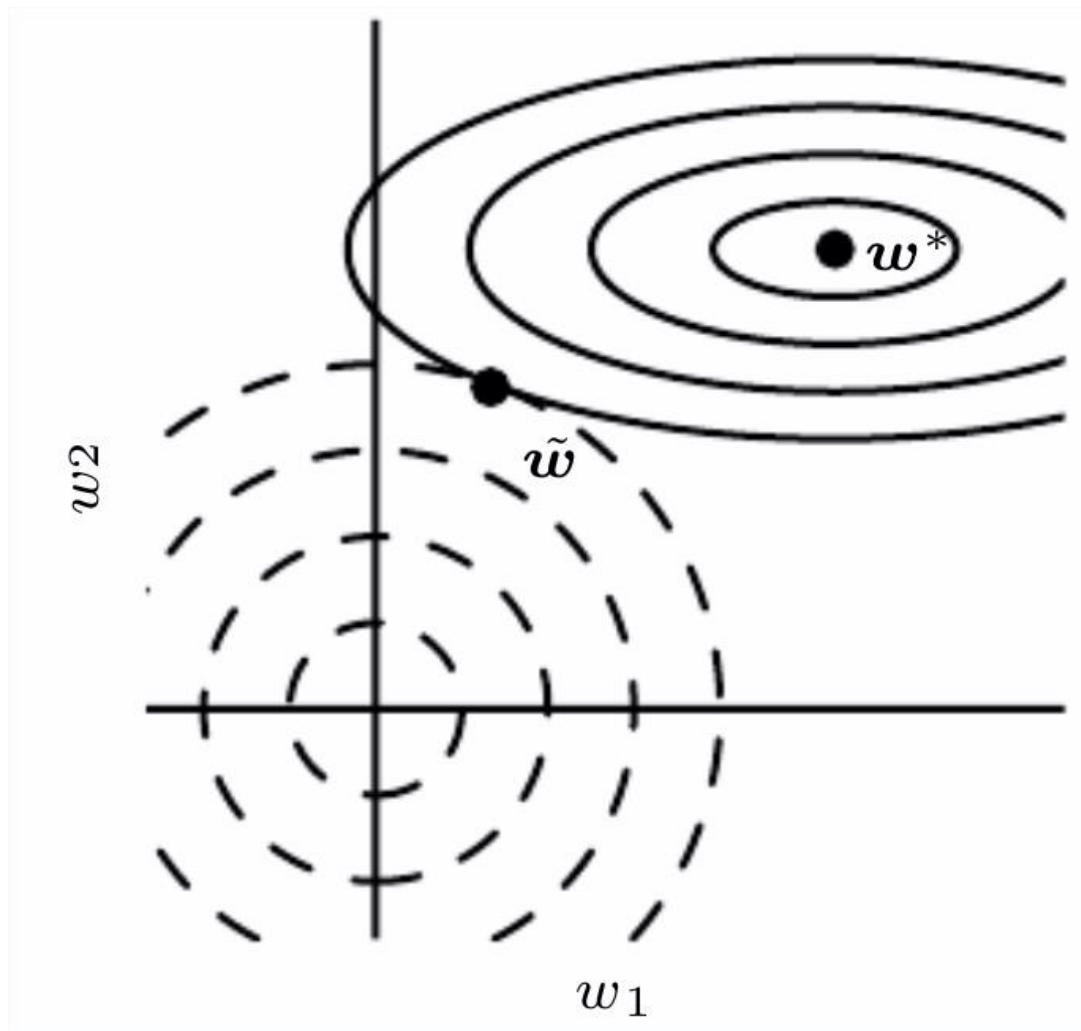
$$\nabla_{\mathbf{w}} \tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \mathbf{w} + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

- To perform single gradient step, perform update:

$$\mathbf{w}' \leftarrow (1 - \epsilon \alpha) \mathbf{w} - \epsilon \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

- We have modified learning rule to shrink \mathbf{w} by constant factor $1 - \epsilon \alpha$ at each step

An illustration of the effect of weight decay



L^1 Regularization

- While L^2 weight decay is the most common form of weight decay there are other ways to penalize the size of model parameters
- L^1 regularization is defined as

$$\Omega(\boldsymbol{\theta}) = \|\mathbf{w}\|_1 = \sum_i |w_i|$$

- Which is the sum of the absolute values of the individual parameters

A closer look

- Objective function (with no bias parameter)

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \|\mathbf{w}\|_1 + J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

- Corresponding Gradient:

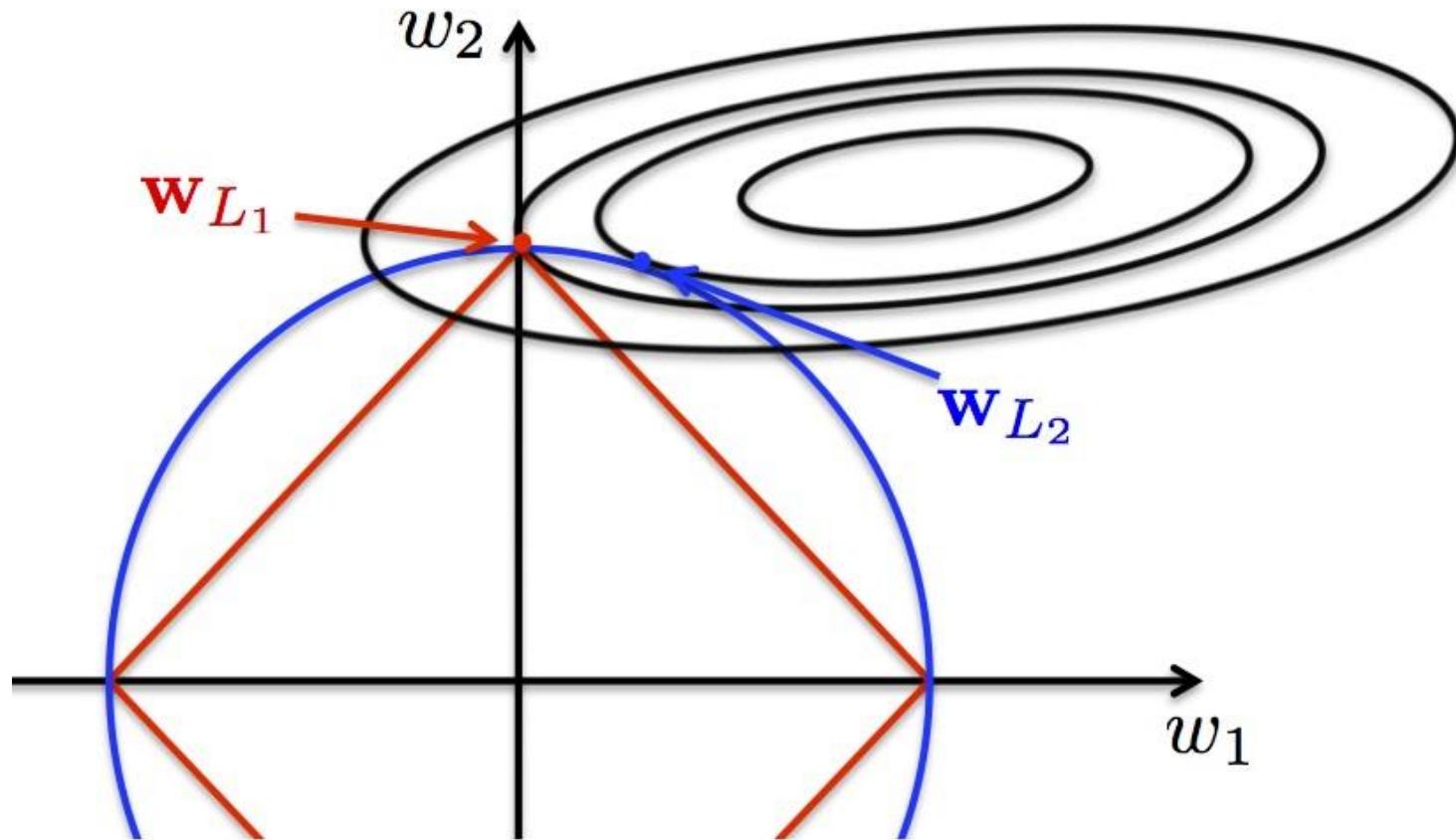
$$\nabla_{\mathbf{w}} \tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \text{sign}(\mathbf{w}) + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

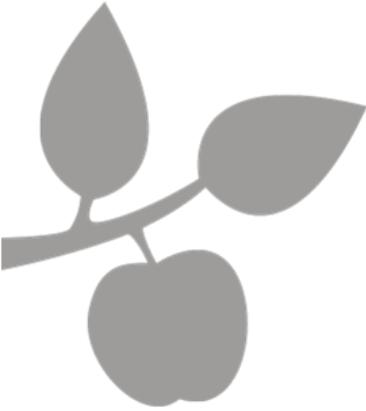
- Where $\text{sign}(\mathbf{w})$ is simply the element-wise sign of \mathbf{w}
 - Always a strong gradient unless $w_i = 0$
 - Leads to a sparse solution

Sparsity and Feature Selection

- The sparsity property induced by L^1 regularization has been used extensively as a feature selection mechanism
- Feature selection simplifies an ML problem by choosing subset of available features
- LASSO (Least Absolute Shrinkage and Selection Operator) integrates an L^1 penalty with a linear model and least squares cost function
- The L^1 penalty causes a subset of the weights to become zero, suggesting that those features can be discarded

L^1 vs. L^2 Regularization





Regularization

- Parameter Penalties
- Data Augmentation
- Noise Robustness
- Parameter Sharing / Semi-Supervised / Multitasking
- Early stopping
- Bagging
- Dropout
- Adversarial training

More Data is Better

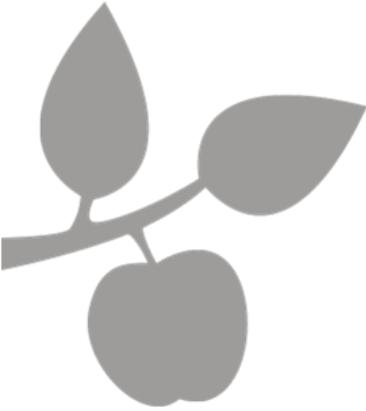
- Best way to make a ML model to generalize better is to train it on more data
- In practice amount of data is limited
- **Get around the problem by creating synthesized data**
- For some ML tasks it is straightforward to synthesize data
- Especially popular for classification/object recognition

Data Augmentation for Classification

- Data augmentation is easiest for classification
 - Classifier takes high-dimensional input x and summarizes it with a single category identity y
 - Main task of classifier is to be invariant to a wide variety of transformations
- Generate new samples (x, y) just by transforming inputs
 - Approach not easily generalized to other problems
 - E.g.: For density estimation problem: generating new data requires solving density estimation first
 - Good example: Images are high-dimensional and include a variety of variations, may easily simulated
 - Translating the images a few pixels can greatly improve performance
 - Rotating and scaling are also effective
 - Some other augmentations are hard to perform (e.g. out-of-pane rotation)

Injecting Noise

- Injecting noise into the input of a neural network can be seen as data augmentation
- **Neural networks are not robust to noise**
- To improve robustness, train them with random noise applied to their inputs
- Noise can also be applied to hidden units
- Dropout, a powerful regularization strategy, can be viewed as constructing new inputs by multiplying by noise



Regularization

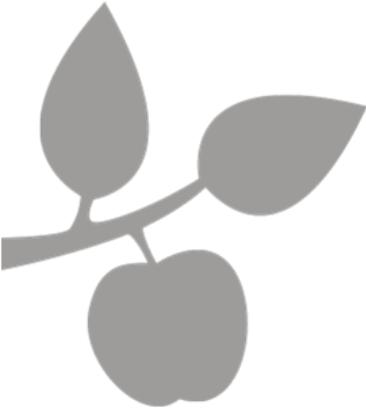
- Parameter Penalties
- Data Augmentation
- **Noise Robustness**
- Parameter Sharing / Semi-Supervised / Multitasking
- Early stopping
- Bagging
- Dropout
- Adversarial training

Noise injection is powerful

- Noise applied to inputs is a data augmentation (as we just saw)
 - For some models addition of noise with infinitesimal variance at the input is equivalent to imposing a penalty on the norm of the weights, e.g., $\alpha \mathbf{w}^T \mathbf{w}$
 - It basically forces the weights being in areas, where little changes on the input have little effects on the output, i.e., being rather small
- Noise applied to hidden units
 - Noise injection can be much more powerful than simply shrinking the parameters
 - Noise applied to hidden units is so important that it merits its own separate discussion
- Dropout is the main development of this approach

Injecting Noise at Output Targets

- Most datasets have some mistakes in y labels
 - Harmful to maximize $\log p(y|x)$ when y is a mistake
- To prevent it we explicitly model noise on labels
 - E.g.: we assume training set label y is correct with probability $1 - \epsilon$, and otherwise any of the other labels may be correct
 - This can be incorporated into the cost function
 - For example: Local Smoothing regularizes a model based on a softmax with k output values by replacing the hard 0 and 1 classification targets with targets of $\frac{\epsilon}{k-1}$ and $1 - \epsilon$ respectively



Regularization

- Parameter Penalties
- Data Augmentation
- Noise Robustness
- Parameter Sharing / Semi-Supervised / Multitasking
- Early stopping
- Bagging
- Dropout
- Adversarial training

Task of Semi-supervised Learning

- Both unlabeled examples from $P(\mathbf{x})$ and labeled examples from $P(\mathbf{x}, y)$ are used to estimate $P(y|\mathbf{x})$ or predict y from \mathbf{x}
- In the context of deep learning it refers to learning a representation $h = f(\mathbf{x})$
- The goal is to learn a representation so that examples from the same class have similar representations

How unsupervised learning helps

- Unsupervised learning can provide useful clues for how to group examples in representational space
- Examples that cluster tightly in the input space should be mapped to similar representations
- A linear classifier in the new space may achieve better generalization
- A variant is the application of PCA as a preprocessing step before applying a classifier to the projected data

Sharing Parameters

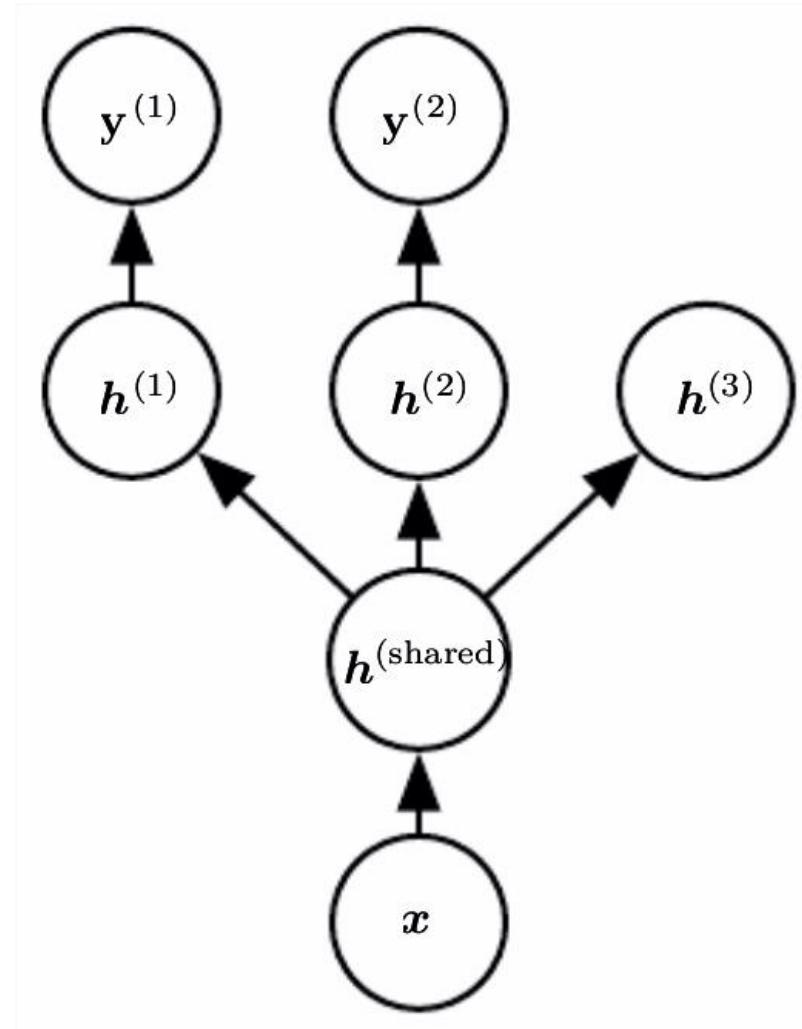
- Instead of separate unsupervised and supervised components in the model, construct models in which generative models of either $P(x)$ or $P(x, y)$ shares parameters with a discriminative model of $P(y|x)$
- One can then trade-off the supervised criterion – $\log P(y|x)$ with the unsupervised or generative one (such as $-\log P(x)$)
- The generative criterion then expresses a prior belief about the solution to the supervised problem

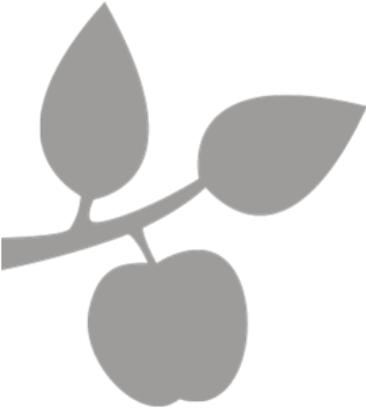
Sharing parameters over tasks

- Multi-task learning is a way to improve generalization by pooling the examples out of several tasks
 - Examples can be seen as providing soft constraints on the parameters
- In the same way that additional training examples put more pressure on the parameters of the model towards values that generalize well
- Different supervised tasks, predicting $y^{(i)}$ given x , share the same input x , as well as some intermediate representation \mathbf{h} (shared) capturing a common pool of factors

Common multi-task situation

- Task specific parameters
 - Which only benefit from the examples of their task to achieve good generalization
 - These are the upper layers of the neural network
- Generic parameters
 - Shared across all tasks
 - Which benefit from the pooled data of all tasks
 - These are the lower levels of the neural network

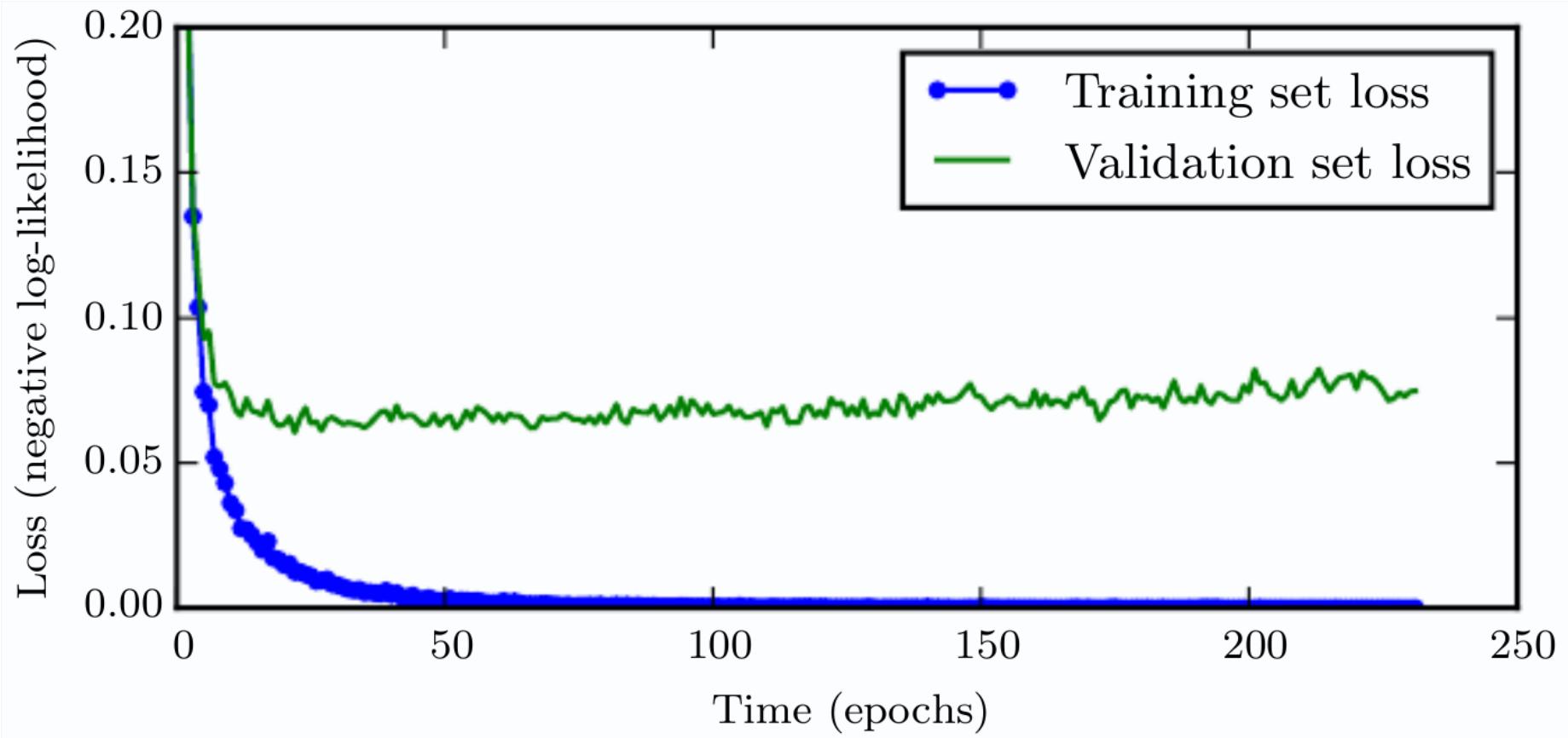




Regularization

- Parameter Penalties
- Data Augmentation
- Noise Robustness
- Parameter Sharing / Semi-Supervised / Multitasking
- **Early stopping**
- Bagging
- Dropout
- Adversarial training

Typical Learning Curves



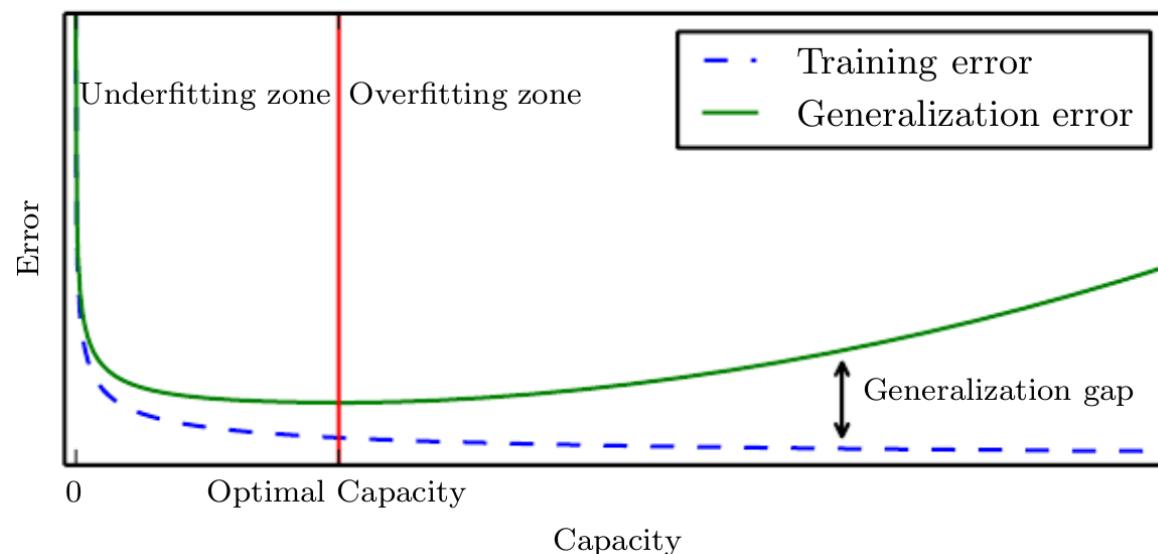
- In this example, we train a maxout network on MNIST. Observe that the training objective decreases consistently over time, but the validation set average loss eventually begins to increase again, forming an asymmetric U-shaped curve.

Early Stopping

- We can obtain a model with better validation set error (and thus better test error) by returning to the parameter setting at the point of time with the lowest validation set error
- Every time the error on the validation set improves, we store a copy of the model parameters.
- When the training algorithm terminates, we return these parameters, rather than the latest set
- It is the most common form of regularization in deep learning due to its effectiveness and its simplicity

Early Stopping as Hyperparameter Selection

- We can think of early stopping as a very efficient hyperparameter selection algorithm
 - In this view no. of training steps is just a hyperparameter
 - This hyperparameter has a U-shaped validation set performance curve
 - Most hyperparameters have such a U-shaped validation set performance curve, as seen below



Costs of Early Stopping

- Cost of this hyperparameter is running validation evaluation periodically during training
 - Ideally done in parallel to training process on a separate machine
 - Separate CPU or GPU from main training process, Or using small validation set or validating set less frequently
- Need to maintain a copy of the best parameters
 - This cost is negligible because they can be stored on a slower, larger memory
 - E.g., training in GPU, but storing the optimal parameters in host memory or on a disk drive

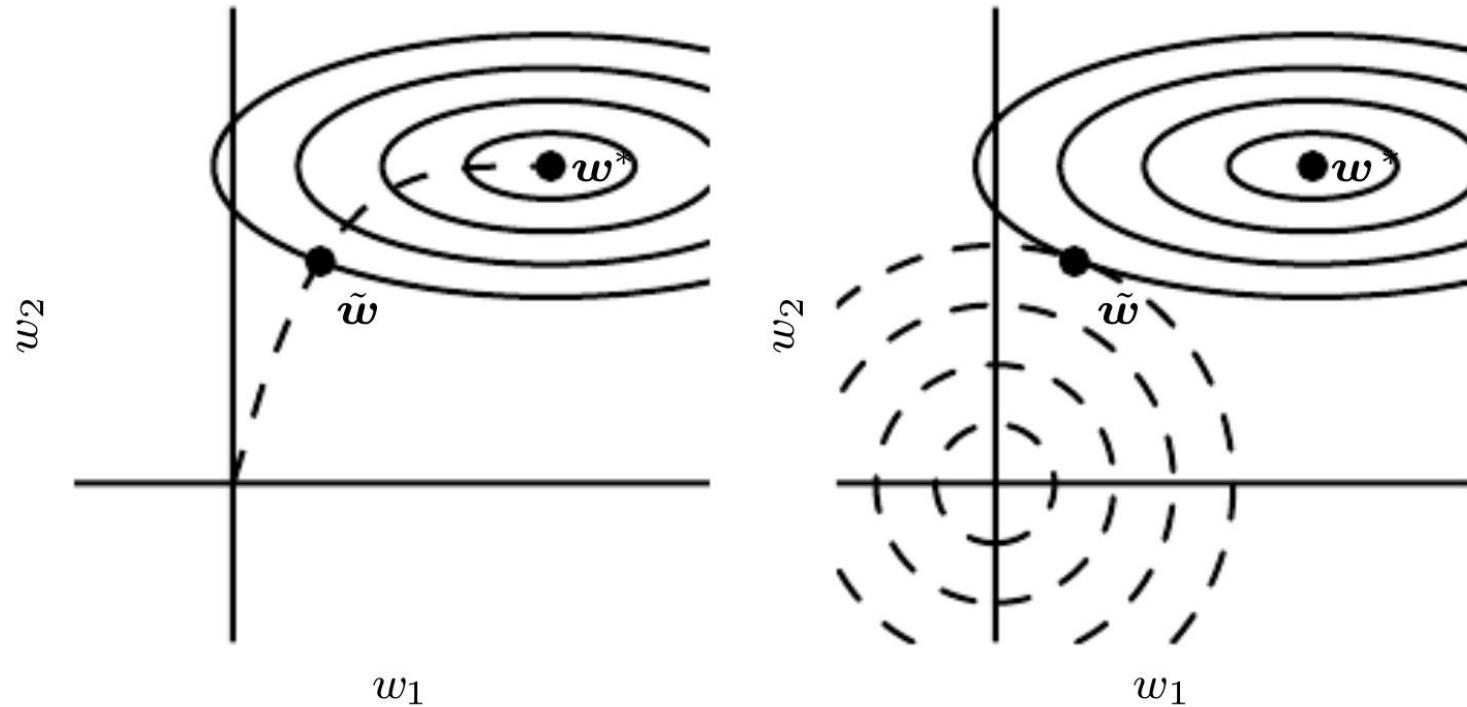
Early Stopping as Regularization

- Early stopping is an unobtrusive form of regularization
- It requires almost no change to the underlying training procedure, the objective function, or the set of allowable parameter values
- So it is easy to use early stopping without damaging the learning dynamics
- In contrast to weight decay, where we must be careful not to use too much weight decay
 - Otherwise we trap the network in a bad local minimum corresponding to pathologically small weights

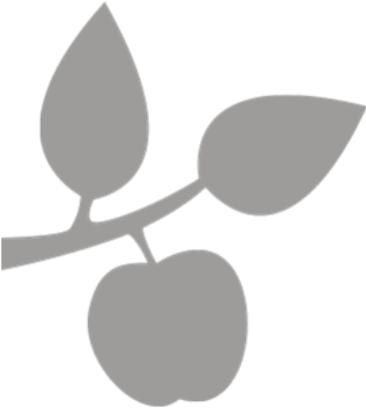
Use of a second training step

- Early stopping requires a validation set
 - Thus some training data is not fed to the model
- To best exploit this extra data, one can perform extra training after the initial training with early stopping has completed
- There are two basic strategies for including the test data
 1. Reinitialize the model to the untrained state and run for the same number of steps again
 2. Continue Training
 - Keep all parameters and continue with now the entire dataset
 - When to stop?
 - The training error will increase with including the entire dataset, stop when we reach the same training error as before

Early Stopping vs L^2 Regularization



- We now restrict the “distance” the weights can travel to find an optimal solution



Regularization

- Parameter Penalties
- Data Augmentation
- Noise Robustness
- Parameter Sharing / Semi-Supervised / Multitasking
- Early stopping
- **Bagging**
- Dropout
- Adversarial training

What is Bagging (Bootstrap Aggregating)

- It is a technique for reducing generalization error by combining several models
 - Idea is to train several models separately, then have all the models vote on the output for test examples
- This strategy is called model averaging
- Techniques employing this strategy are known as ensemble methods
- Model averaging works because different models will not make the same mistake

Ensemble vs Bagging

- Different ensemble methods construct the ensemble of models in different ways
- Example: each member of ensemble could be formed by training a completely different kind of model using a different algorithm or objective function
- Bagging is a method that allows the same kind of model, training algorithm and objective function to be reused several times

The Bagging Technique

- Given training set D of size N , generate k data sets of same no of examples as original by sampling with replacement
- Some observations may be repeated in D_i , some others are missing. This is known as a bootstrap sample.
- The differences in examples will result in differences between trained models
- The k models are combined by averaging the output (for regression) or voting (for classification)

Example

Original dataset



First resampled dataset



First ensemble member



Second resampled dataset

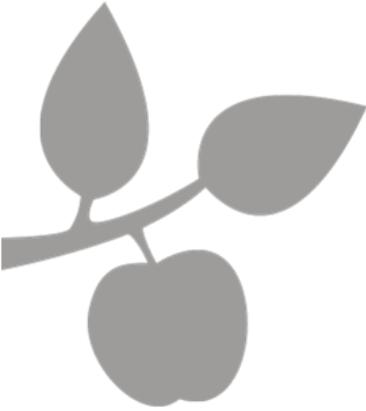


Second ensemble member



Bagging in Neural Nets

- Neural nets reach a wide variety of solution points
 - Thus they benefit from model averaging when trained on the same dataset
 - Differences in:
 - random initializations
 - random selection of minibatches, in hyperparameters,
 - cause different members of the ensemble to make partially independent errors
- Model averaging is a reliable method for reducing generalization error
 - Machine learning contests are usually won by model averaging over dozens of models, e.g., the Netflix grand prize
- But comes with significant computational costs



Regularization

- Parameter Penalties
- Data Augmentation
- Noise Robustness
- Parameter Sharing / Semi-Supervised / Multitasking
- Early stopping
- Bagging
- Dropout
- Adversarial training

Overfitting in Deep Neural Nets

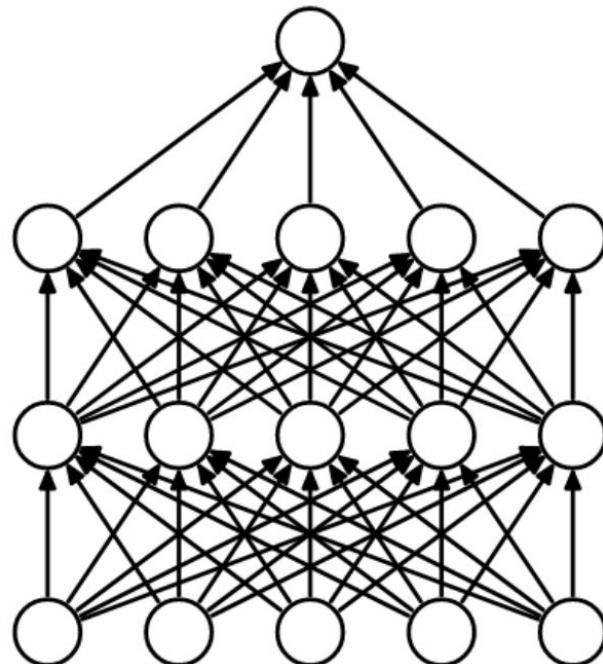
- Deep nets have many non-linear hidden layers
 - Making them very expressive to learn complicated relationships between inputs and outputs
 - But with limited training data, many complicated relationships will be the result of training noise
 - So they will exist in the training set and not in test set even if drawn from same distribution
- Many methods developed to reduce overfitting
 - Early stopping with a validation set
 - Weight penalties (L^1 and L^2 regularization)
 - Soft weight sharing

Dropout as Bayesian Approximation

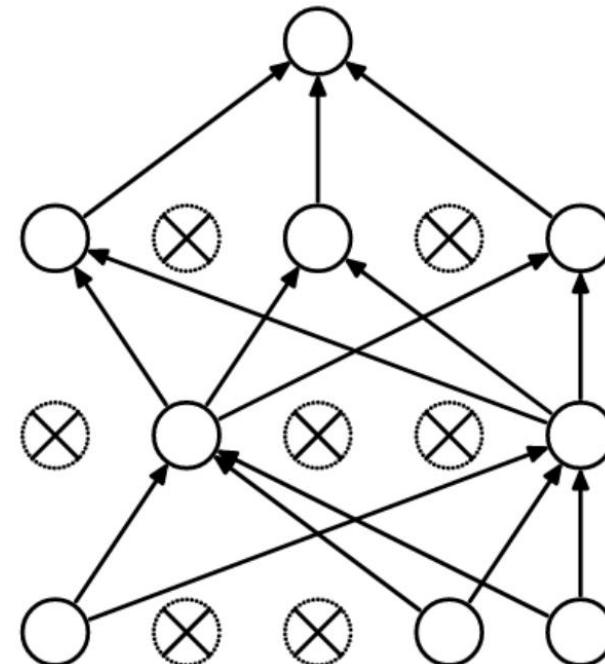
- Best way to regularize a fixed size model is:
 - Average the predictions of all possible settings of the parameters
 - Weighting each setting with the posterior probability given the training data
 - This would be the Bayesian approach
- Dropout does this using considerably less computation
 - Approximating an equally weighted geometric mean of the predictions of an exponential number of learned models that share parameters
 - Dropout makes it practical to apply bagging to very many large neural networks

Creating new Models by Removing Units

- Dropout trains an ensemble of subnetworks
 - formed by removing non-output units from an underlying base network
- We can effectively remove units by multiplying its output value by zero



(a) Standard Neural Net



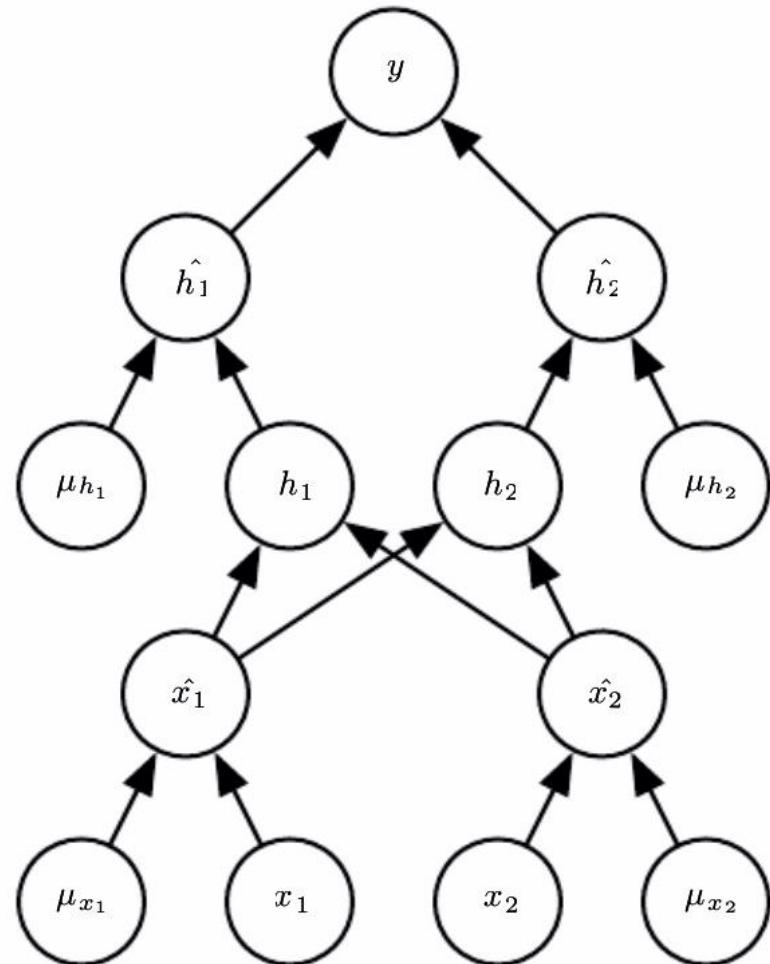
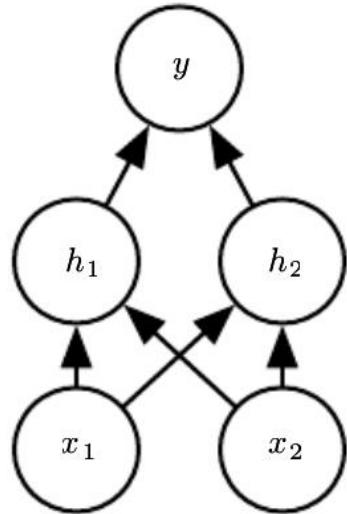
(b) After applying dropout.

Mask for dropout training

- To train with dropout we use minibatch based learning algorithm that takes small steps such as SGD
- At each step randomly sample a binary mask
 - Probability of including a unit is a hyperparameter
 - Normally: 0.5 for hidden units and 0.8 for input units
- We run forward & backward propagation as usual
- Equivalent to randomly selecting a subnetwork of the entire network

Example

- Modified network incorporating a binary vector μ
- Training and evaluation as normal



Formal Description

- Suppose that mask vector μ specifies which units to include

- Cost of the model is specified by

$$J(\boldsymbol{\theta}, \boldsymbol{\mu})$$

- Dropout training consists of minimizing

$$\mathbb{E}_{\boldsymbol{\mu}}(J(\boldsymbol{\theta}, \boldsymbol{\mu}))$$

- Expected value contains exponential no. of terms

- We can get an unbiased estimate of its gradient by sampling values of $\boldsymbol{\mu}$

Bagging training vs Dropout training

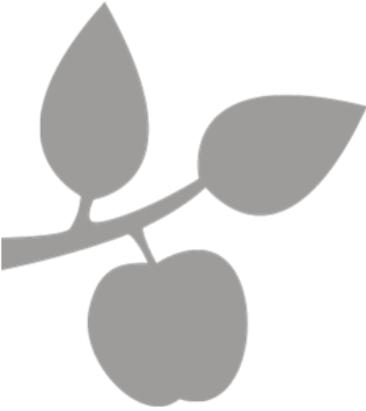
- Dropout training not same as bagging training
 - In bagging, the models are all independent
 - In dropout, models share parameters
 - Models inherit subsets of parameters from parent network
 - Parameter sharing allows an exponential no. of models with a tractable amount of memory
- In bagging each model is trained to convergence on its respective training set
 - In dropout, most models are not explicitly trained
 - Fraction of subnetworks are trained for a single step
 - Parameter sharing allows good parameter settings

Dropout Prediction

- Submodel defined by mask vector μ defines a probability distribution $p(y|x, \mu)$
- At testing time after training has finished, we would ideally like to find a sample average of all possible 2^n dropped-out networks
 - This is unfeasible for any realistic number n
- Approximation:
 - Each node's output weighted by a factor of p , so the expected value of the output of any node is the same as in the training stages.
 - This is the biggest contribution of the dropout method:
 - although it simulates 2^n different neural nets, at test time only a single network needs to be evaluated and tested

Another Interpretation of Dropout

- So far we have described dropout purely as a means of performing efficient, approximate bagging
- Dropout trains an ensemble of models that share hidden units
 - Each hidden unit must be able to perform well regardless of which other hidden units are in the model
 - Hidden units must be prepared to be swapped and interchanged between models
- **Dropout thus regularizes each hidden unit to be not merely a good feature but a feature that is good in many contexts.**



Regularization

- Parameter Penalties
- Data Augmentation
- Noise Robustness
- Parameter Sharing / Semi-Supervised / Multitasking
- Early stopping
- Bagging
- Dropout
- Adversarial training

The Understanding of Deep Nets

- In many cases, neural networks have begun to reach human level performance when evaluated on an i.i.d. test set
- Have they reached human level understanding?
- To probe the level of understanding we can construct examples that the model misclassifies
- Even neural networks that perform at human level accuracy have a 100% error rate on examples intentionally constructed!

Adversarial examples

- An optimization procedure is used to search for an input x' near data point x such that the model output is very different at x'
 - In many cases, x' can be so similar to x that a human observer cannot tell the difference between the original example and the adversarial example
- **But the network makes a highly different prediction**
- Adversarial examples have many implications
 - E.g., they are useful in computer security since they are hard to defend against
 - They are interesting in the context of regularization
 - Using adversarial perturbed samples we can reduce error rate on test set

Example



$$+ .007 \times$$



=



\mathbf{x}

$y = \text{"panda"}$
With 58% confidence

$$\text{sign}(\nabla_{\mathbf{x}} J(\boldsymbol{\theta}, \mathbf{x}, y))$$

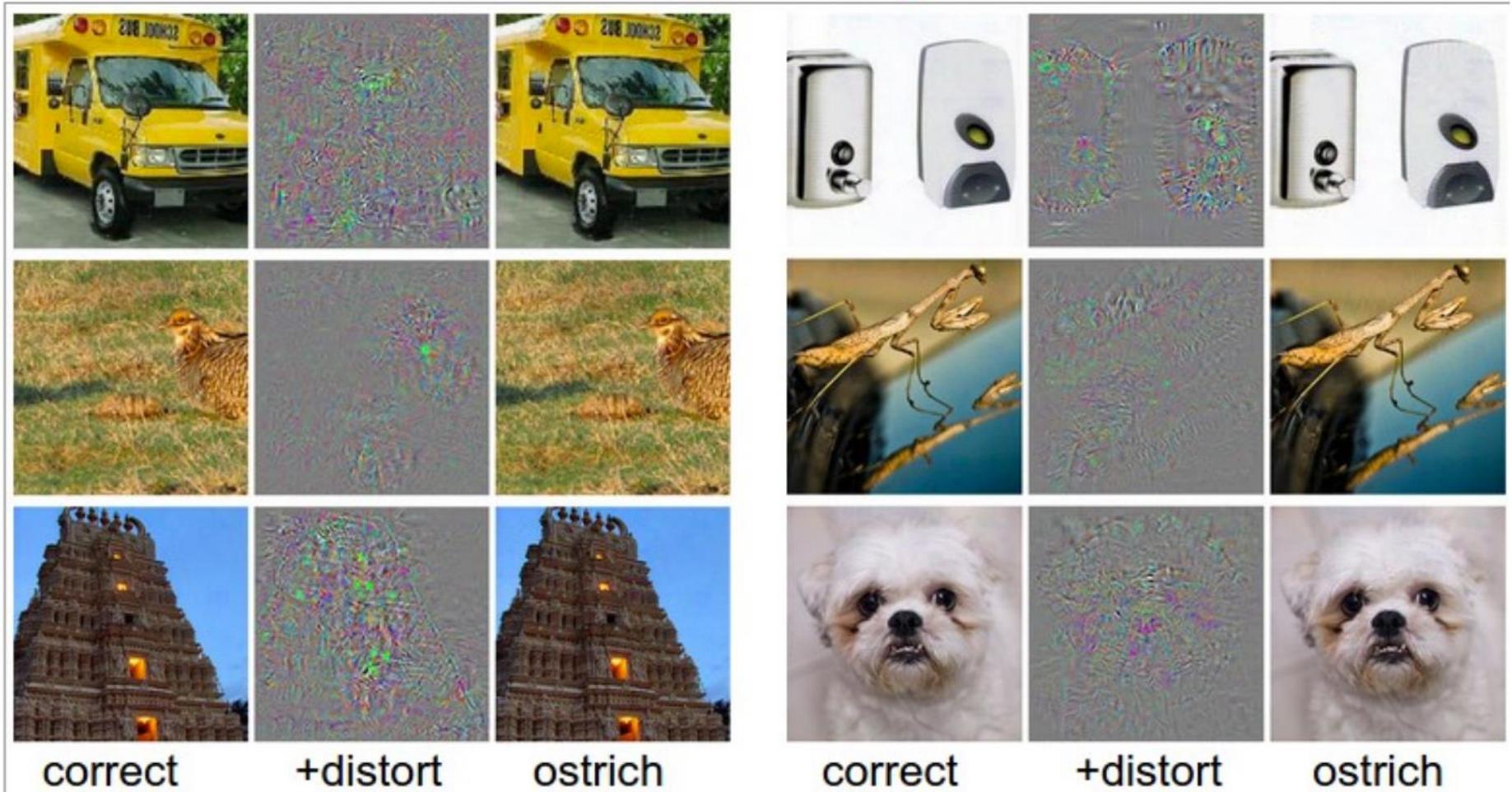
$$\frac{\mathbf{x} + \epsilon \text{sign}(\nabla_{\mathbf{x}} J(\boldsymbol{\theta}, \mathbf{x}, y))}{\epsilon \text{sign}(\nabla_{\mathbf{x}} J(\boldsymbol{\theta}, \mathbf{x}, y))}$$

$y = \text{"nematode"}$
With 8.2% confidence

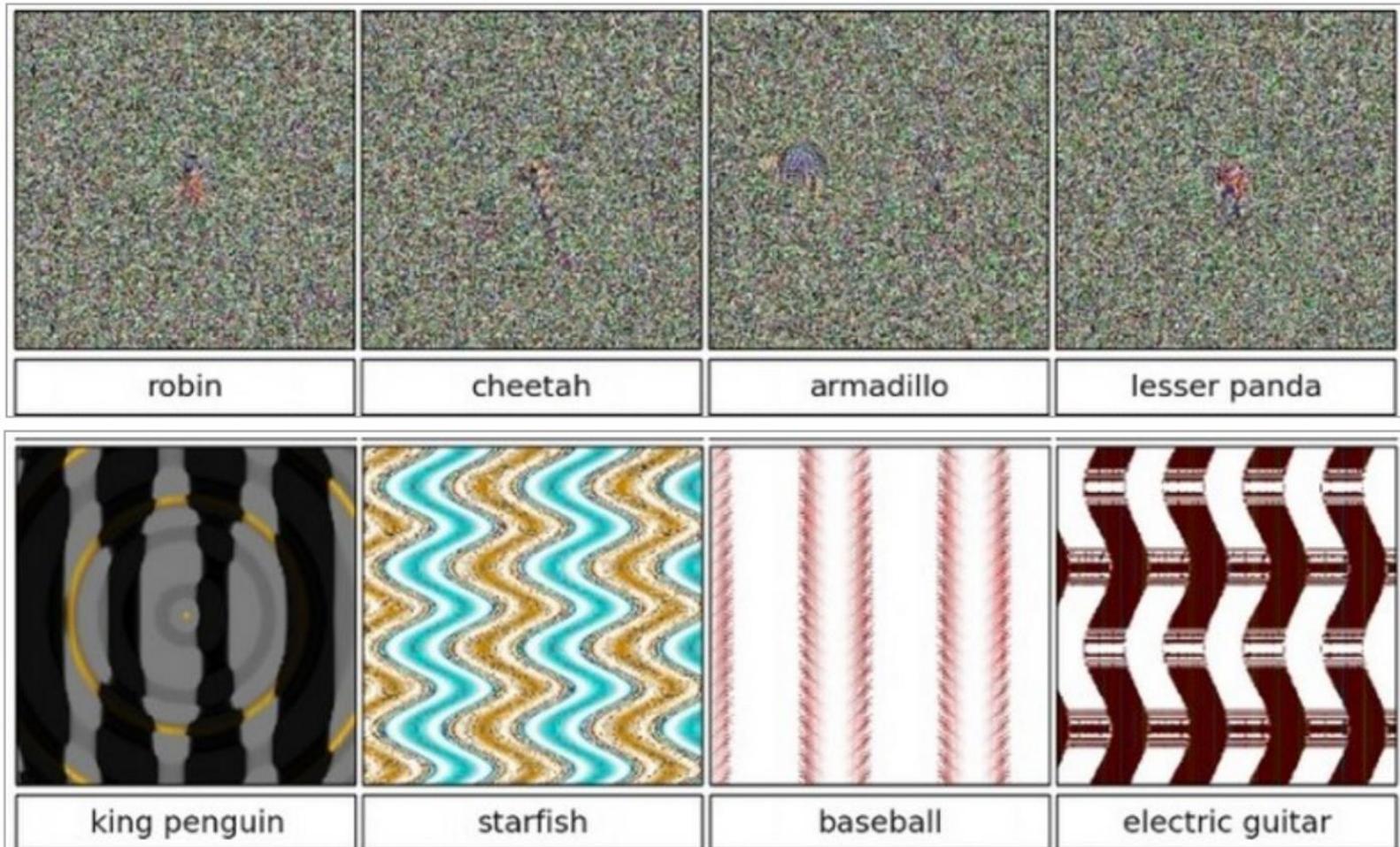
$y = \text{"gibbon"}$
With 99% confidence

4

Example



Example



These images are classified with >99.6% confidence as the shown class by a Convolutional Network.

Cause of adversarial examples

- Primary cause is excessive linearity
 - Neural networks are built primarily out of linear building blocks
 - The overall function often proves to be linear
 - Linear functions are easy to optimize
 - But the value of a linear function can change rapidly with numerous inputs
- If we change input by ϵ then a linear functions with weights w can change by $\epsilon\|w\|$ which can be very large in high-dimensional spaces

Adversarial Training

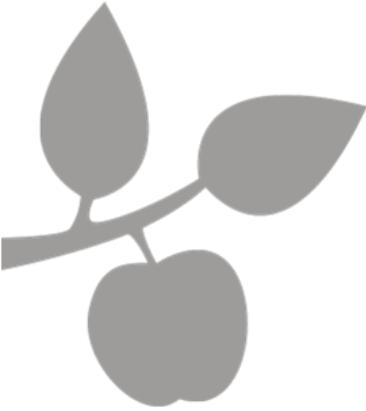
- Adversarial training discourages highly sensitive local behavior
- By encouraging network to be locally constant in the neighborhood of the training data
- This can be seen as a way of explicitly introducing a local constancy prior into supervised neural nets
- Adversarial training illustrates the power of using a large function family in combination with aggressive regularization
 - Purely linear models, like logistic regression, are unable to resist adversarial examples because they are forced to be linear

Relation to Semi-supervised Learning

- Adversarial examples provide a means of accomplishing semi-supervised learning
- At a point x that is not associated with a label in a dataset, the model itself assigns some label \hat{y}
- It may not be the true label, but if model is of high quality then \hat{y} has a probability of being the true label
- We can seek an adversarial example x' that causes the classifier to output a label y' with $y' \neq \hat{y}$

Virtual Adversarial Examples

- Adversarial examples generated with using not the true label but a label provided by a trained model are called **Virtual Adversarial Examples**
- The classifier may then be trained to assign the same label to x and x'
- This encourages the classifier to learn a function that is robust to small changes anywhere along the manifold
 - Assumption motivating this approach:
 - different classes lie on disconnected manifolds
 - A small perturbation should not be able to jump from one class manifold to another class manifold

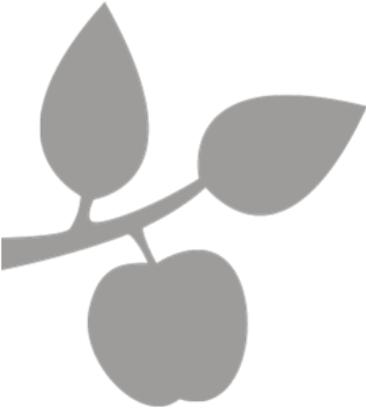


DM873

Deep Learning

Spring 2019

Lecture 10 – Recurrent Neural Networks



Recurrent Neural Networks

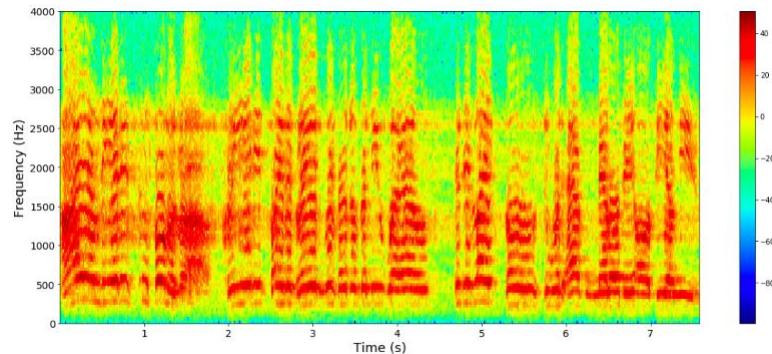
- **Introduction**
- **Unfolding a Graph**
- **Recurrent Neural Networks**
- **Long Term Memory**
 - Leaky Units
 - Long-Short Term Memory
- **Text in KERAS**

RNNs process sequential data

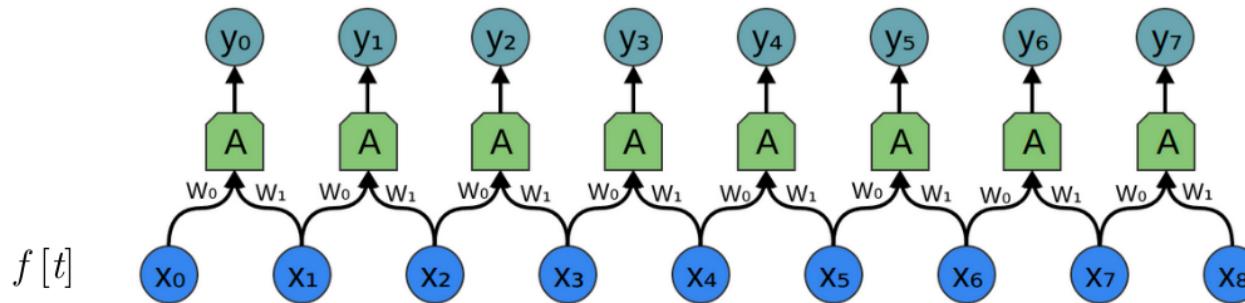
- Recurrent Neural Networks are a family of neural networks for processing sequential data
- Just as CNN is specialized for processing grid of values, e.g., image
 - RNN is specialized for processing a sequence of values $x^{(1)}, \dots, x^{(\tau)}$
- Just as CNNs can readily scale images with large width/height and process variable size images
 - RNNs can scale to much longer sequences than would be practical for networks without sequence-based specialization
 - RNNs can also process variable-length sequences

Examples of Sequential Data and Tasks

- Sequence data: sentences, speech, stock market, signal data
- Sequence-to-sequence Tasks
 - Speech recognition
 - decompose sound waves into frequency and amplitude using Fourier transforms yielding a spectrogram
 - Named Entity Recognition
 - Input: Jim bought 300 shares of Acme Corp. in 2006
 - NER: [Jim]_{Person} bought 300 shares of [Acme Corp.]_{Organization} in [2006]_{Time}
- Sequence-to-symbol
 - Sentiment
 - Speaker recognition



Recall: 1-D Convolution



Kernel $g(t)$:
[...0, w_1 , w_0 , 0...]

Equations for outputs of this network:

$$y_0 = \sigma(W_0x_0 + W_1x_1 - b)$$

$$y_1 = \sigma(W_0x_1 + W_1x_2 - b)$$
 etc. upto y_8

Note that kernel gets flipped in convolution

We can also write the equations in terms of elements of a general 8×8 weight matrix W as:

$$y_0 = \sigma(W_{0,0}x_0 + W_{0,1}x_1 + W_{0,2}x_2\dots)$$

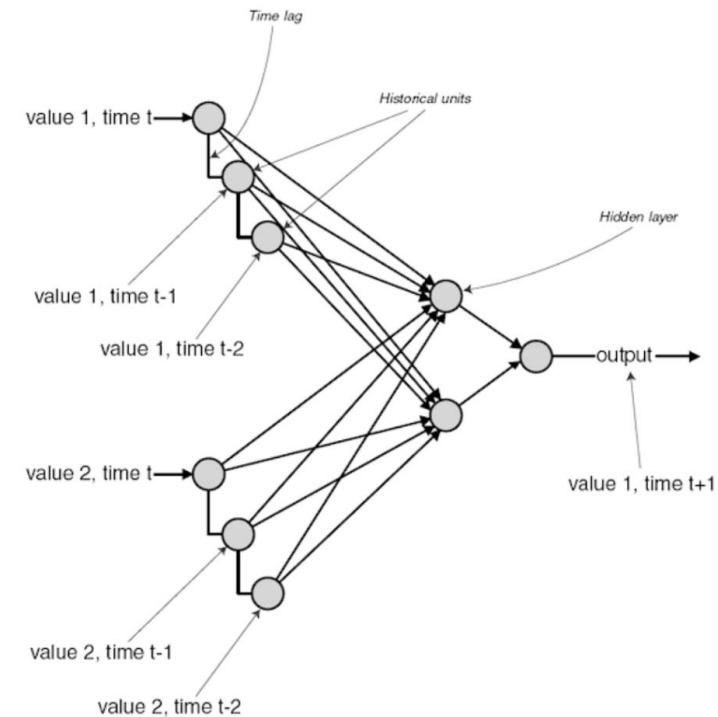
$$y_1 = \sigma(W_{1,0}x_0 + W_{1,1}x_1 + W_{1,2}x_2\dots)$$

where $W = \begin{bmatrix} w_0 & w_1 & 0 & 0 & \dots \\ 0 & w_0 & w_1 & 0 & \dots \\ 0 & 0 & w_0 & w_1 & \dots \\ 0 & 0 & 0 & w_0 & \dots \\ \dots & \dots & \dots & \dots & \dots \end{bmatrix}$

➤ <http://colah.github.io/posts/2014-07-Understanding-Convolutions/>

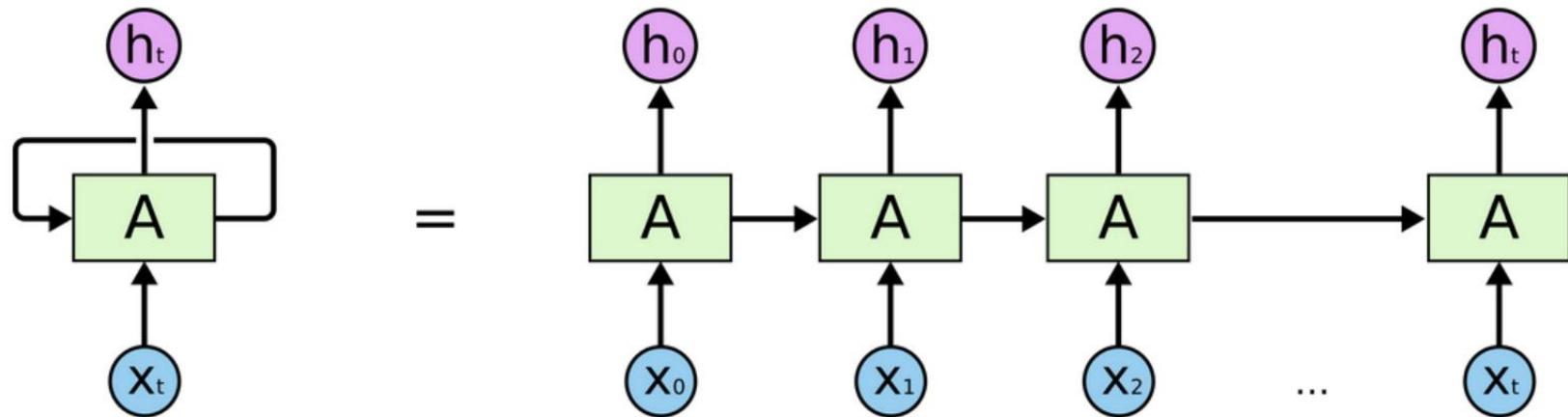
Time Delay Neural Networks

- Time-delay neural networks perform convolution across 1-D temporal sequence
 - Convolution operation allows a network to share parameters across time, but is shallow
 - Each member of output is dependent upon a small no. of neighboring members of the input
 - Parameter sharing manifests in the application of the same convolutional kernel at each time step
- A TDNN remembers the previous few training examples and uses them as input into the network.



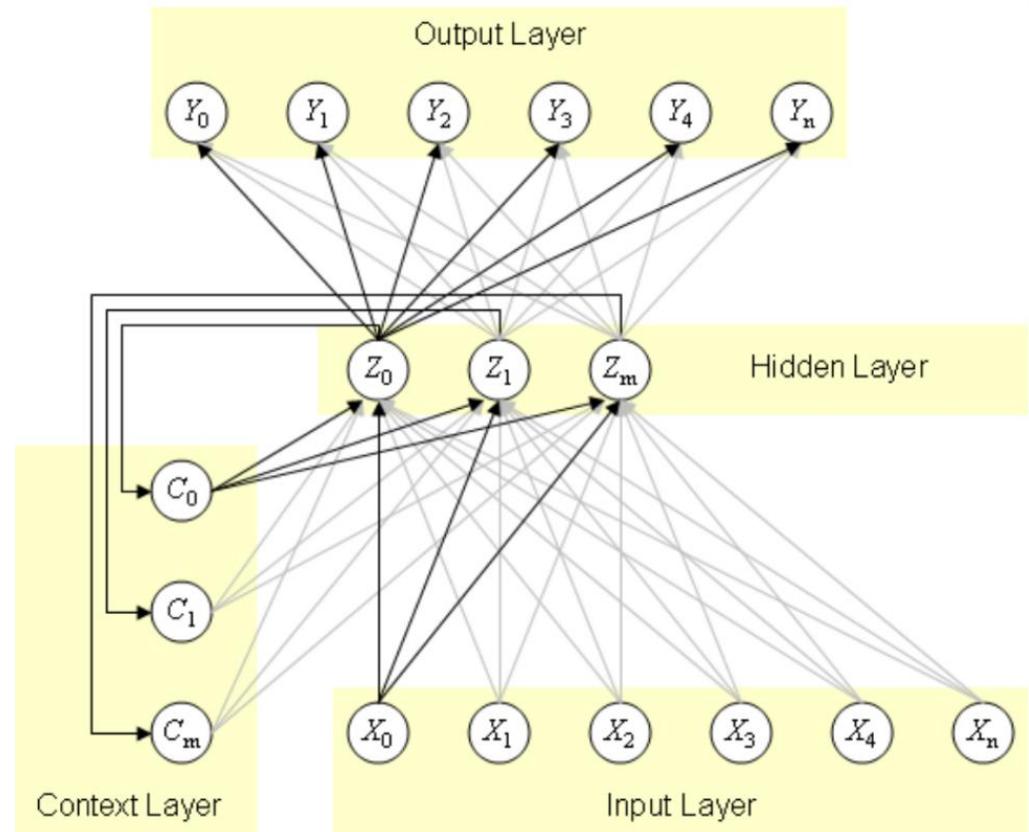
RNN vs. TDNN

- RNNs share parameters in a different way
 - Each member of output is a function of previous members of output
 - Each output produced using same update rule applied to previous outputs
 - This recurrent formulation results in sharing of parameters through a very deep computational graph
- An unrolled RNN



RNN as a network with cycles

- An RNN is a class of neural networks where connections between units form a directed cycle
- This creates an internal state of the network which allows it to exhibit dynamic temporal behavior
- The internal memory can be used to process arbitrary sequences of inputs

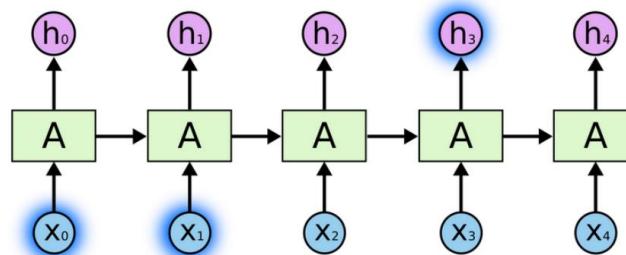


RNNs share same weights across Time Step

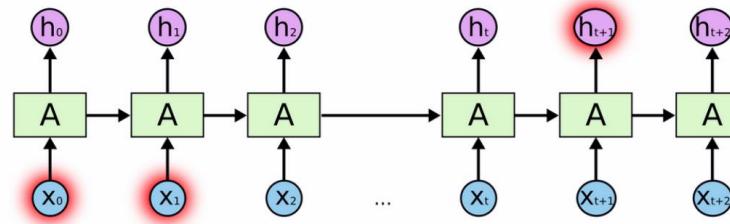
- To go from multi-layer networks to RNNs:
 - Need to share parameters across different parts of a model
 - Separate parameters for each value of input cannot generalize to sequence lengths not seen during training
 - Share statistical strength across different sequence lengths and across different positions in time
- Sharing important when information can occur at multiple positions in the sequence
 - Given “I went to Nepal in 1999 ” and “In 1999, I went to Nepal ”, an ML method to extract year, should extract 1999 whether in position 6 or 2
 - A feed-forward network that processes sentences of fixed length would have to learn all of the rules of language separately at each position
 - An RNN shares the same weights across several time steps

Problem of Long-Term Dependencies

- Easy to predict last word in “the clouds are in the sky”
 - When gap between relevant information and place that it’s needed is small, RNNs can learn to use the past information



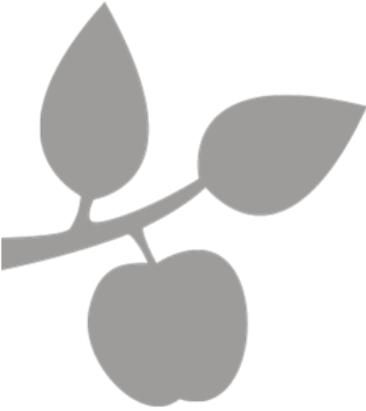
- “I grew up in France... I speak fluent French.”
 - Large gap between relevant information and point where it is needed



- In principle RNNs can handle it, but fail in practice
 - LSTMs offer a solution

RNN operating on a sequence

- RNNs operate on a sequence that contain vector $x^{(t)}$ with time step index t , ranging from 1 to τ
 - Sequence $x^{(1)}, \dots, x^{(\tau)}$
 - RNNs operate on minibatches of sequences of length τ
- Some remarks about sequences
 - The steps need not refer to passage of time in the real world
 - RNNs can be applied in two-dimensions across spatial data such as image
 - Even when applied to time sequences, network may have connections going backwards in time, provided entire sequence is observed before it is provided to network



Recurrent Neural Networks

- Introduction
- Unfolding a Graph
- Recurrent Neural Networks
- Long Term Memory
 - Leaky Units
 - Long-Short Term Memory
- Text in KERAS

Unfolding Computational Graphs

- Recall: A Computational Graph is a way to formalize the structure of a set of computations
 - Such as mapping inputs and parameters to outputs and loss
- We can unfold a recursive or recurrent computation into a computational graph that has a repetitive structure
 - Corresponding to a chain of events
- Unfolding this graph results in sharing of parameters across a deep network structure

Example of Unfolding a Recurrent Equation

- Classical form of a dynamical system is

$$\mathbf{s}^{(t)} = f(\mathbf{s}^{(t-1)}; \boldsymbol{\theta})$$

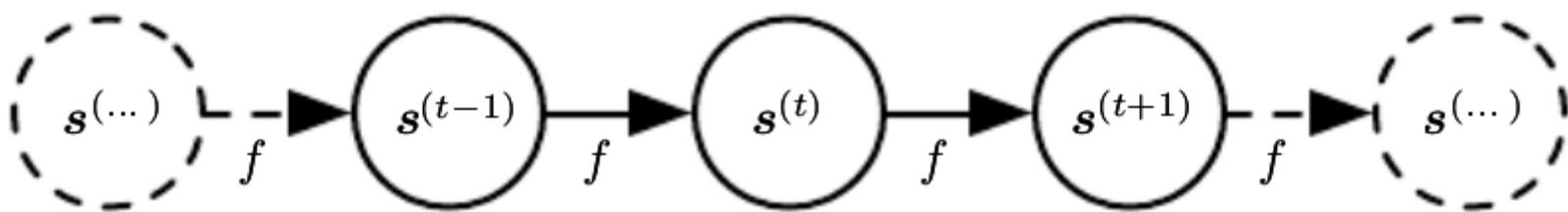
- where $\mathbf{s}^{(t)}$ is called the state of the system
- Equation is recurrent because the definition of \mathbf{s} at time t refers back to the same definition at time $t - 1$
- For a finite no. of time steps τ , the graph can be unfolded by applying the definition $\tau - 1$ times
 - E.g, for $\tau = 3$ time steps we get
- $$\mathbf{s}^{(3)} = f(\mathbf{s}^{(2)}; \boldsymbol{\theta}) = f(f(\mathbf{s}^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta})$$
- Unfolding equation by repeatedly applying the definition in this way has yielded expression without recurrence
 - $\mathbf{s}^{(1)}$ is ground state, the other states are calculated by applying f
 - Can now be represented by a traditional acyclic computational graph

Unfolded dynamical system

- The classical dynamical system described by

$$s^{(t)} = f(s^{(t-1)}; \theta)$$

is illustrated as an unfolded computational graph:



- Each node represents state at some time t
- Function f maps state at time t to the state at $t + 1$
- The same parameters (the same value of θ used to parameterize f) are used for all time steps

Dynamical system driven by external signal

- As another example, consider a dynamical system driven by external (input) signal $x^{(t)}$

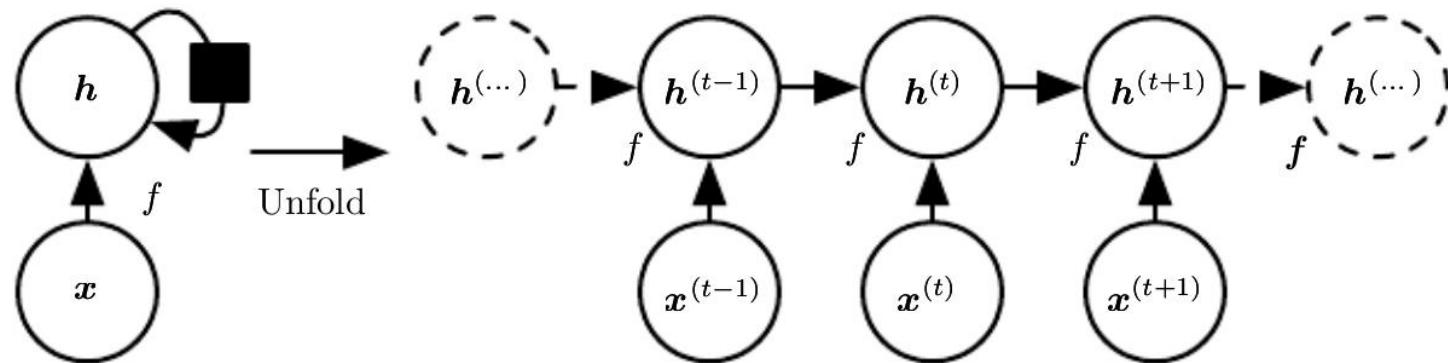
$$s^{(t)} = f(s^{(t-1)}, x^{(t)}; \theta)$$

- State now contains information about the whole past input sequence
- Recurrent neural networks can be built in many ways
 - Much as almost any function is a feedforward neural network, any function involving recurrence can be considered to be a recurrent neural network

Defining values of hidden units in RNNs

- Many recurrent neural nets use same equation (as dynamical system with external input) to define values of hidden units
 - To indicate that the state is hidden rewrite using variable \mathbf{h} for state

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



- Typical RNNs have extra architectural features such as output layers that read information out of state \mathbf{h} to make predictions

Predicting the Future from the Past

- When RNN is required to perform a task of predicting the future from the past, network typically learns to use $\mathbf{h}^{(t)}$ as a lossy summary of the past sequence of inputs upto time point t
- The summary is in general lossy since it maps a sequence of arbitrary length $(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)})$ to a fixed length vector $\mathbf{h}^{(t)}$
- Depending on criterion, summary keeps some aspects of past sequence more precisely than other aspects
 - RNN used in statistical language modeling, typically to predict next word from past words
 - it may not be necessary to store all information upto time t but only enough information to predict rest of sentence

Process of Unfolding

- We can represent unfolded recurrence after t steps with a function $g^{(t)}$:

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

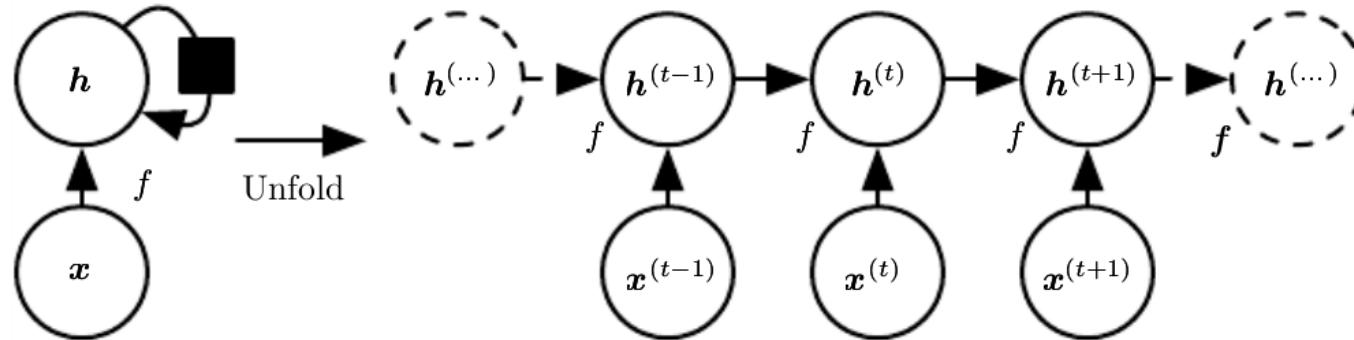
- Function $g^{(t)}$ takes in whole past sequence $(\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(1)})$ as input and produces the current state but the unfolded recurrent structure allows us to factorize $g^{(t)}$ into repeated application of a function f
- This has several advantages

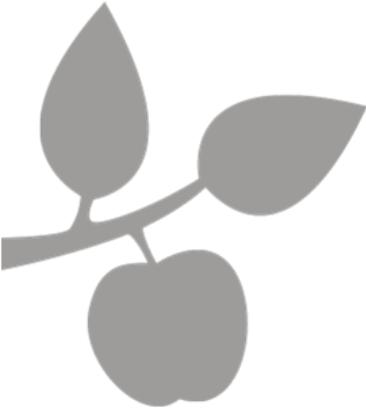
Unfolding process allows learning a single model

- The unfolding process introduces two major advantages:
 1. Regardless of sequence length, learned model has same input size
 - because it is specified in terms of transition from one state to another state rather than specified in terms of a variable length history of states
 2. Possible to use same function f with same parameters at every step
- These two factors make it possible to learn a single model f
 - that operates on all time steps and all sequence lengths
 - rather than needing separate model $g^{(t)}$ for all possible time steps
- Learning a single shared model allows:
 - Generalization to sequence lengths that did not appear in the training
 - Allows model to be estimated with far fewer training examples than would be required without parameter sharing

Recurrent Graph vs. Unrolled Graph

- Recurrent graph is succinct
- Unrolled graph provides explicit description of what computations to perform
 - Helps illustrate the idea of information flow forward in time
 - Computing outputs and losses
 - And backwards in time
 - Computing gradients
 - By explicitly showing path along which information flows





Recurrent Neural Networks

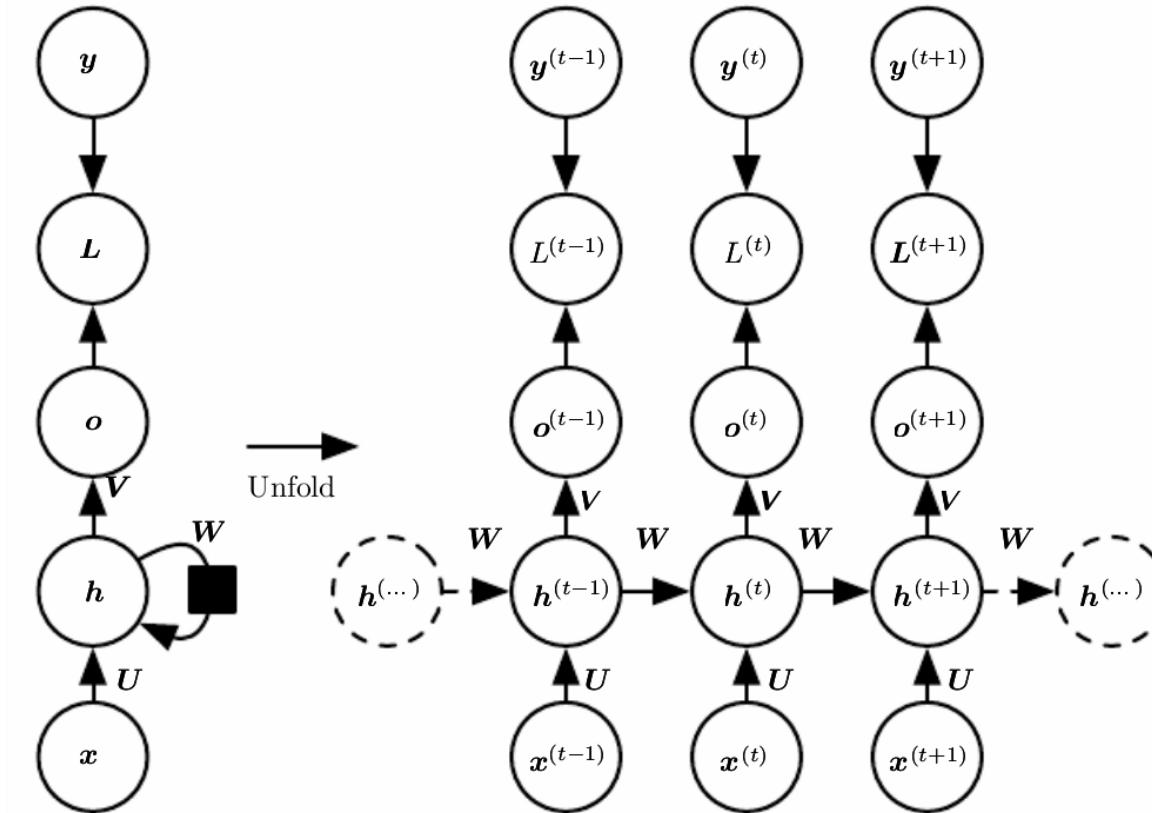
- Introduction
- Unfolding a Graph
- Recurrent Neural Networks
- Long Term Memory
 - Leaky Units
 - Long-Short Term Memory
- Text in KERAS

Three design patterns of RNNs

- Design 1: Output: each time step; Recurrence: hidden units
- Design 2: Output: each time step; Recurrence: output units
- Design 3: Output: one at the end; Recurrence: hidden units

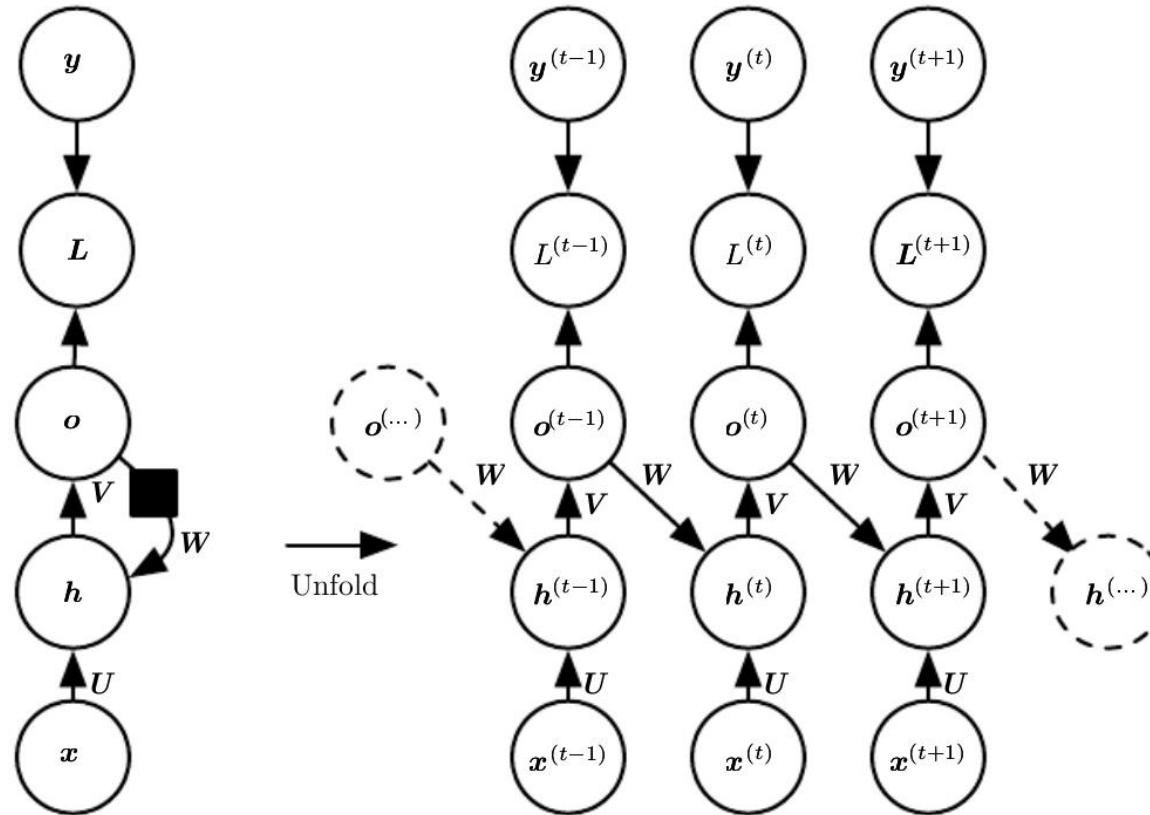
Three design patterns of RNNs

- Design 1: Output: each time step; Recurrence: hidden units
- Design 2: Output: each time step; Recurrence: output units
- Design 3: Output: one at the end; Recurrence: hidden units



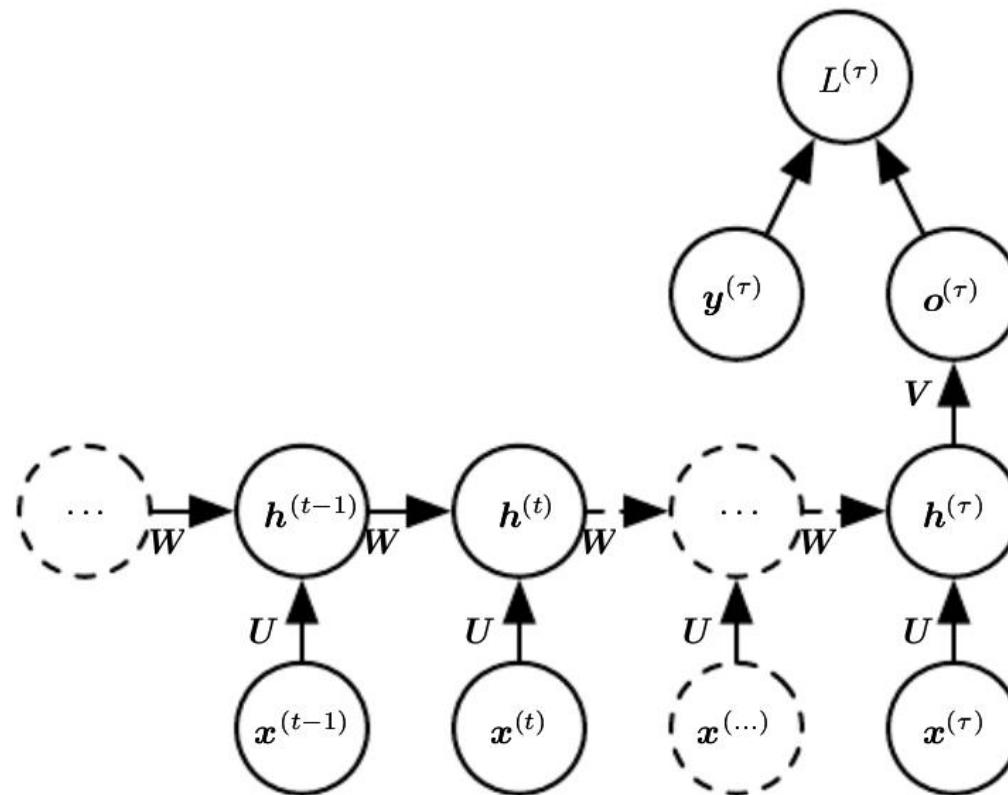
Three design patterns of RNNs

- Design 1: Output: each time step; Recurrence: hidden units
- **Design 2: Output: each time step; Recurrence: output units**
- Design 3: Output: one at the end; Recurrence: hidden units



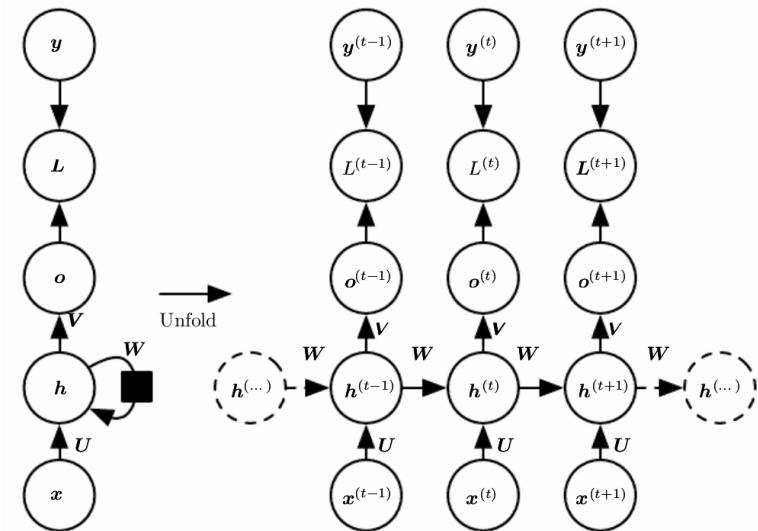
Three design patterns of RNNs

- Design 1: Output: each time step; Recurrence: hidden units
- Design 2: Output: each time step; Recurrence: output units
- **Design 3: Output: one at the end; Recurrence: hidden units**



Design 1: RNN with recurrence between hidden units

- Maps input sequence x to output o values
 - Loss L measures how far each o is from the corresponding target y
 - With softmax outputs we assume o is the unnormalized log probabilities
 - Loss L internally computes $y = \text{softmax}(o)$ and compares to target y
 - Let's assume we have tanh activation function
- Any function computable by a Turing Machine is computable by such an RNN of finite size



Design 1: Forward Calculation

- We need to specify initial state $\mathbf{h}^{(0)}$, then for each time point:

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)}$$

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

$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}$$

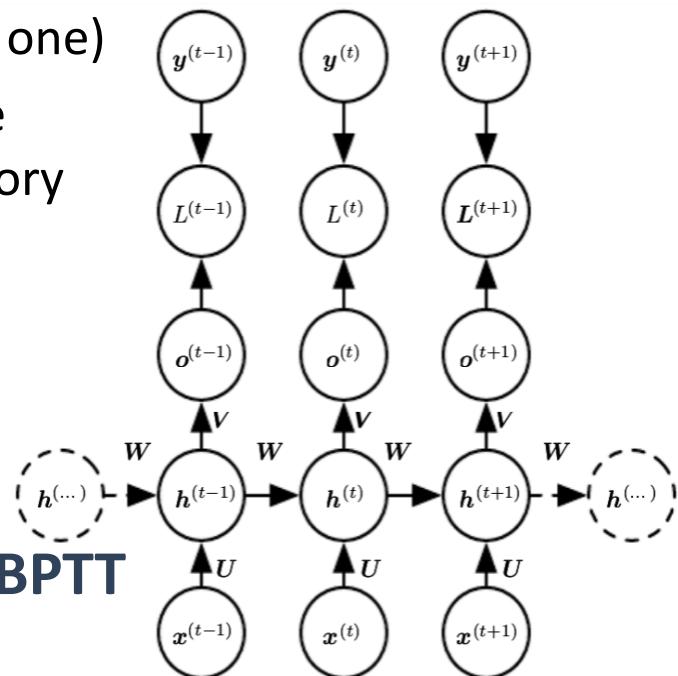
$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{o}^{(t)})$$

- The Loss function can simply defined as the sum of the loss at each timepoint:

$$L = \sum_t L^{(t)} = - \sum_t \log p_{model} (\mathbf{y}^{(t)} | \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}\})$$

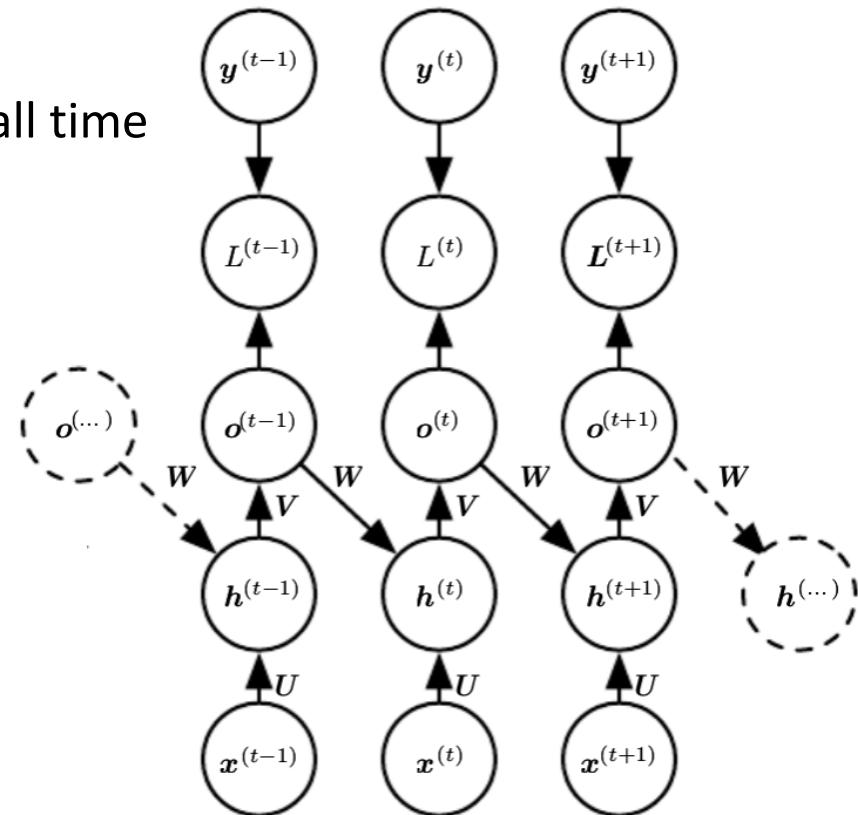
Design 1: Gradient Calculation

- Computing the gradient of this loss function with respect to the parameters is an expensive operation.
 - Requires forward propagation pass through the unrolled graph
 - followed by a backward pass
 - The runtime is $O(\tau)$ and cannot be reduced by parallelization (each time step may only be computed after the previous one)
 - The forward pass must be stored until they are reused during the backward pass, so the memory cost is also $O(\tau)$.
- The back-propagation algorithm applied to the unrolled graph with $O(\tau)$ cost is called **back-propagation through time** or **BPTT**



Design 2: Recurrence from Output to Hidden

- Less powerful than Design 1
 - It cannot simulate a universal Turing Machine
 - It requires that the output capture all information of past to predict future
- Advantage
 - In comparing loss function to output all time steps are decoupled
 - Each step can be trained in isolation
 - Training can be parallelized
 - Gradient for each step can be computed in isolation
 - No need to compute output for the previous step first, because training set provides ideal value of output
- Trained with **Teacher Forcing**



Teacher Forcing

- Teacher forcing during training means
 - Instead of summing activations from incoming units (possibly erroneous)
 - Each unit sums correct teacher activations as input for the next iteration
- It emerges from the maximum likelihood criterion

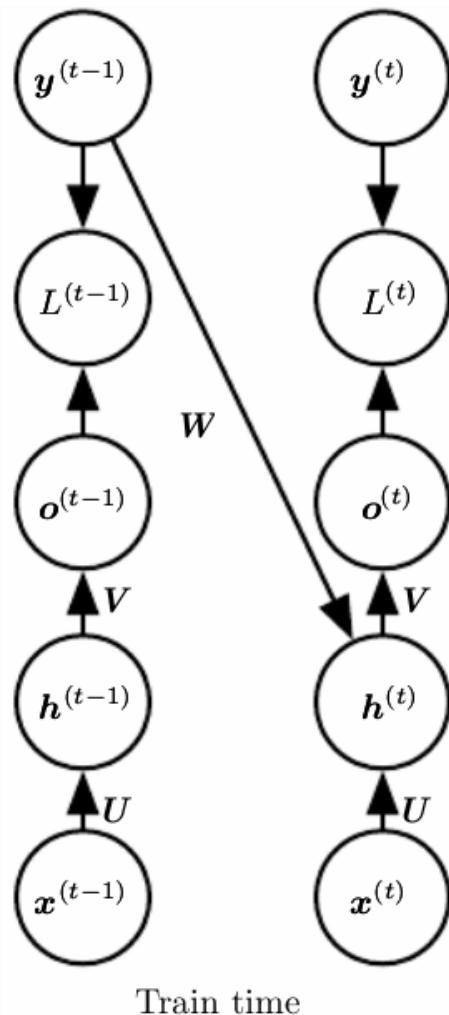
$$\log p(\mathbf{y}^{(1)}, \mathbf{y}^{(2)} | \mathbf{x}^{(1)}, \mathbf{x}^{(2)}) = \log p(\mathbf{y}^{(2)} | \mathbf{y}^{(1)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}) + \log p(\mathbf{y}^{(1)} | \mathbf{x}^{(1)}, \mathbf{x}^{(2)})$$

- At time $t = 2$, model is trained to maximize cond. prob. of $\mathbf{y}^{(2)}$ given both the \mathbf{x} sequence so far and the previous \mathbf{y} value from the training set.
 - We are using $\mathbf{y}^{(1)}$ as teacher forcing, rather than only $\mathbf{x}^{(1)}$.
- Maximum likelihood specifies that during training, rather than feeding the model's own output back to itself, target values should specify what the correct output should be

Illustration of Teacher Forcing

Train time:

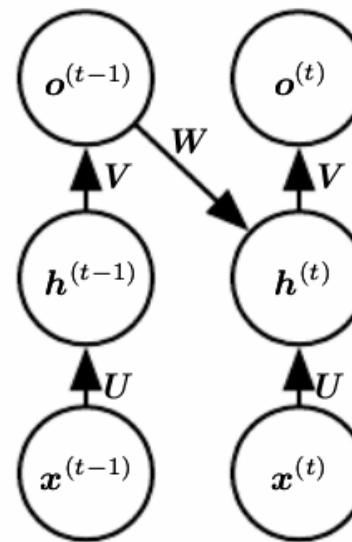
We feed the correct output $y^{(t)}$ (from teacher) drawn from the training set as input to $h^{(t+1)}$



Train time

Test time:

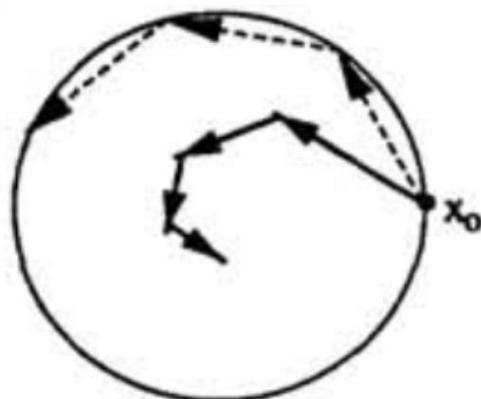
True output is not known.
We approximate the correct output $y^{(t)}$ with the model's output $o^{(t)}$ and feed the output back to the model



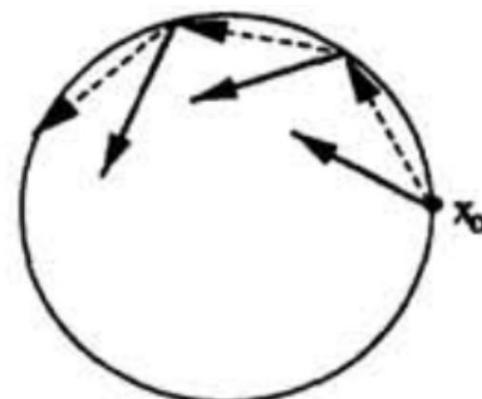
Test time

Visualizing Teacher Forcing

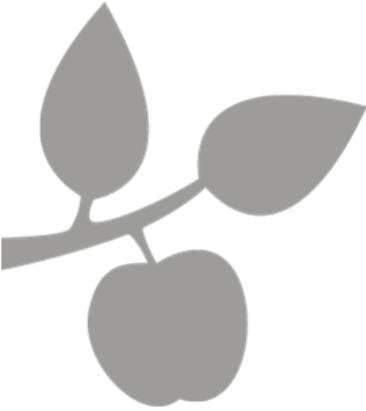
- Imagine that the network is learning to follow a trajectory
- It goes astray (because the weights are wrong) but teacher forcing puts the net back on its trajectory



a



b



Recurrent Neural Networks

- Introduction
- Unfolding a Graph
- Recurrent Neural Networks
- Long Term Memory
 - Leaky Units
 - Long-Short Term Memory
- Text in KERAS

Challenge of Long-Term Dependencies

- Neural network optimization face a difficulty when computational graphs become deep, e.g.,
 - Feedforward networks with many layers
 - RNNs that repeatedly apply the same operation at each time step of a long temporal sequence
- Gradients propagated over many stages tend to either vanish (most of the time) or explode (damaging optimization)
- The difficulty with long-term dependencies arise from exponentially smaller weights given to long-term interactions (involving multiplication of many Jacobians)

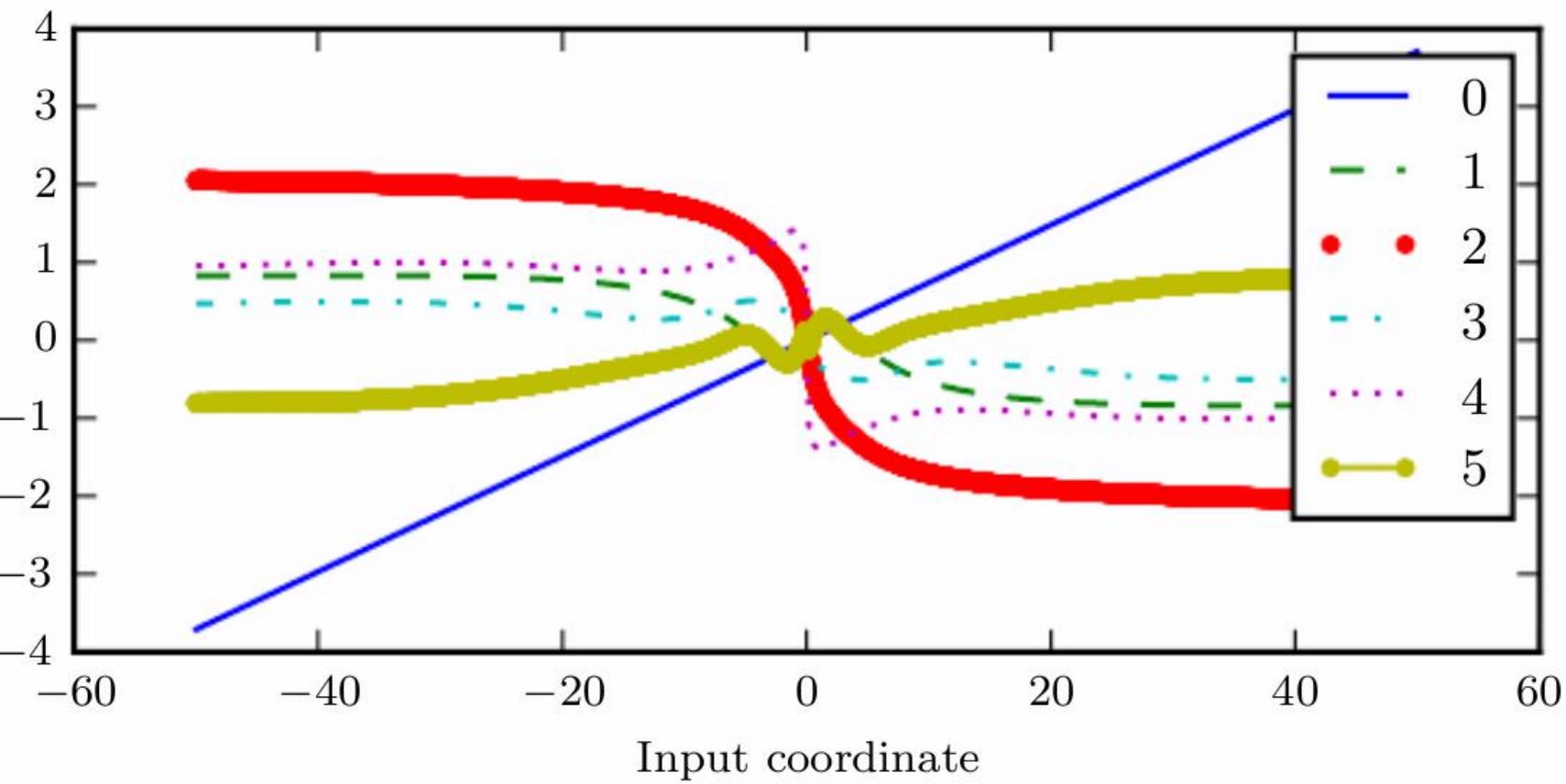
Vanishing and Exploding Gradient Problem

- Suppose a computational graph consists of repeatedly multiplying by a matrix \mathbf{W}
- After t steps this is equivalent to multiplying by \mathbf{W}^t
- Suppose \mathbf{W} has an eigendecomposition $\mathbf{W} = \mathbf{V}\text{diag}(\lambda)\mathbf{V}^{-1}$:

$$\mathbf{W}^t = (\mathbf{V}\text{diag}(\lambda)\mathbf{V}^{-1})^t = \mathbf{V}\text{diag}(\lambda)^t\mathbf{V}^{-1}$$

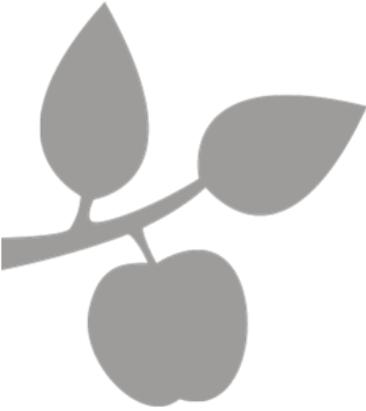
- Any eigenvalues λ_i that are not near an absolute value of 1 will either explode if they are greater than 1 in magnitude and vanish if they are less than 1 in magnitude
- Vanishing gradients make it difficult to know which direction the parameters should move to improve cost
- Exploding gradients make learning unstable

Projection of output



Vanishing and Exploding Gradient Problem

- This problem is particular to recurrent networks
- Imagine, if we had different weights at each time-step $w^{(t)}$ sampled with zero mean and variance ν :
 - The variance of the product is $O(\nu^n)$
 - We can limit the target variance to a value ν^* by choosing $\nu = \sqrt[n]{\nu^*}$
- Is staying with the parameters in a certain region a solution?
 - In order to store memories robust to small perturbations, the RNN must enter a region of parameter space where gradients vanish
 - The gradient of a long term interactions has exponentially smaller magnitude than the gradient of a short term interaction
 - This makes learning difficult
 - Probability of successful training of a traditional RNN via SGD rapidly reaching 0 for sequences of only length 10 or 20.



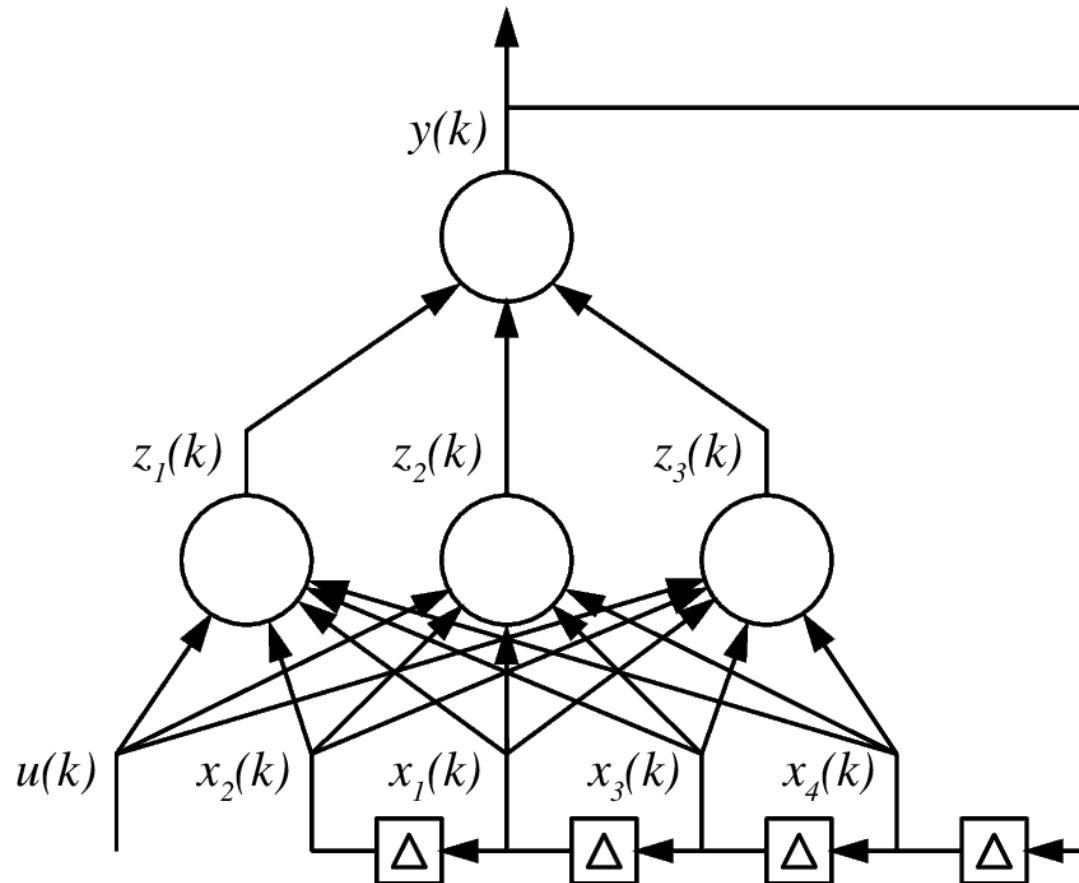
Recurrent Neural Networks

- Introduction
- Unfolding a Graph
- Recurrent Neural Networks
- Long Term Memory
 - Leaky Units
 - Long-Short Term Memory
- Text in KERAS

Adding skip connections through time

- One way to obtain coarse time scales is to add direct connections from variables in the distant past to variables in the present
- In an ordinary RNN, recurrent connection goes from time t to time $t + 1$.
 - Gradients can vanish/explode exponentially wrt no. of time steps
- Introduce time delay of d to mitigate this problem
- Gradients diminish as a function of $\frac{\tau}{d}$ rather than τ
- Allows learning algorithm to capture longer dependencies
 - Not all long-term dependencies can be captured this way

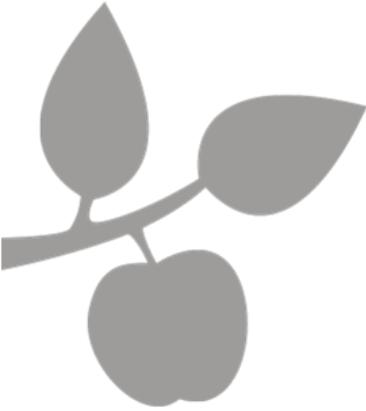
Network with 4 output delays



$$y(t) = \Psi \left(u(t), y(t-1), \dots, y(t-D) \right)$$

Leaky units and a spectrum of time scales

- Rather than an integer skip of d time steps, the effect can be obtained smoothly by adjusting a real-valued α
- Example: Running Average
 - Running average $\mu^{(t)}$ of some value $v^{(t)}$ is
$$\mu^{(t)} = \alpha\mu^{(t-1)} + (1 - \alpha)v^{(t)}$$
 - Called a linear self-correction
 - When α is close to 1, running average remembers information from the past for a long time and when it is close to 0, information is rapidly discarded.
- Hidden units with linear self connections behave similar to running average. They are called leaky units.
- Can obtain product of derivatives close to 1 by having linear self-connections and a weight near 1 on those connections

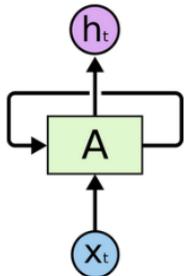


Recurrent Neural Networks

- **Introduction**
- **Unfolding a Graph**
- **Recurrent Neural Networks**
- **Long Term Memory**
 - Leaky Units
 - **Long-Short Term Memory**
- **Text in KERAS**

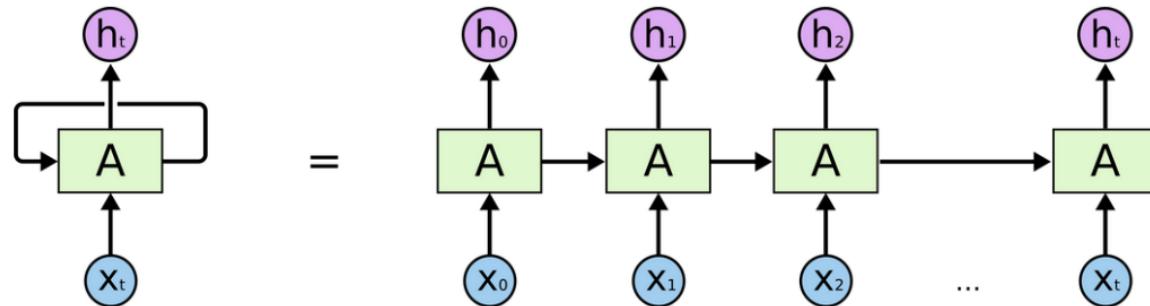
From RNN to LSTM

1. RNNs have loops



A chunk of neural network A looks at some input x_t and outputs a value h_t . A loop allows information to be passed from one step of the network to the next

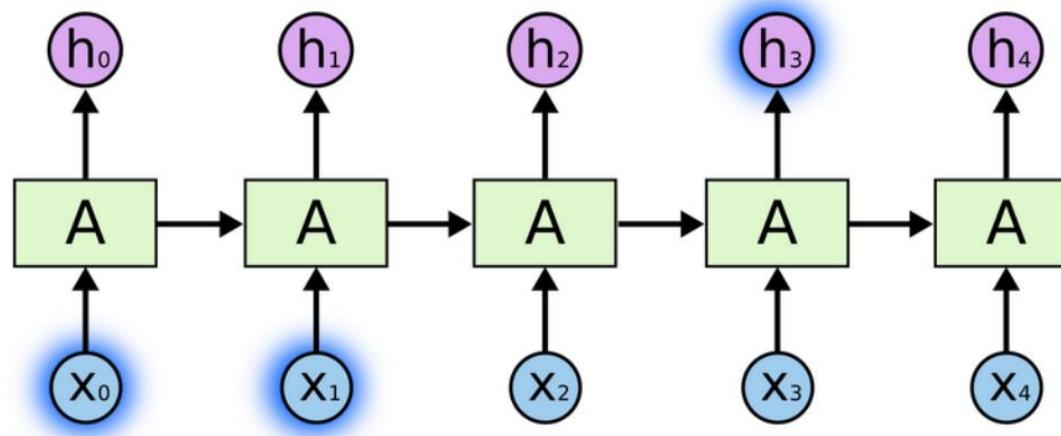
2. An unrolled RNN



Chain-like structure reveals that RNNs are intimately related to sequences and lists

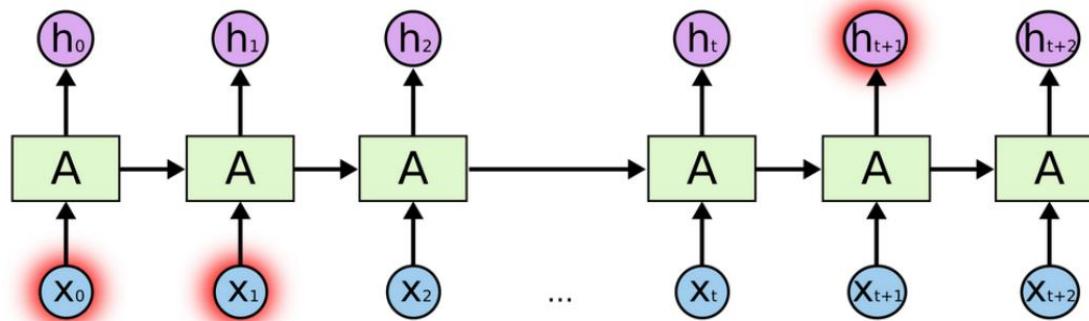
Prediction with only recent previous information

- RNNs connect previous information to the present task
 - Previous video frames may help understand present frame
- Sometimes we need only look at recent previous information to predict
 - To predict the last word of “The clouds are in the sky” we don’t need any further context. It is obvious that the word is “sky”



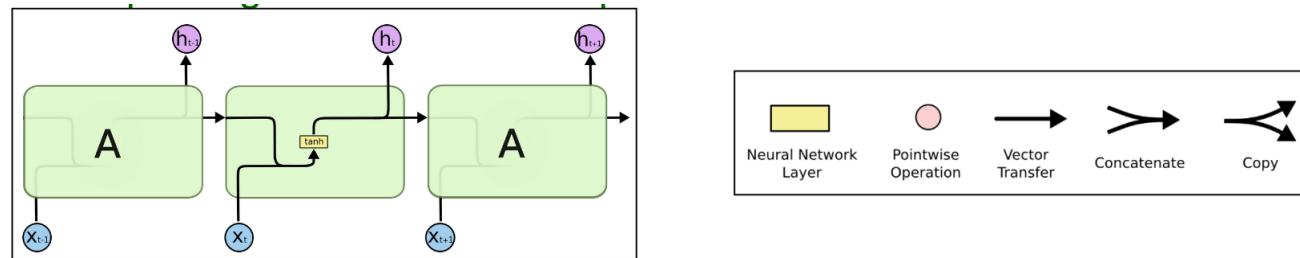
Long-term Connections not Possible

- here are cases where we need more context
- To predict the last word in the sentence “I grew up in France....I speak fluent French”
 - Using only recent information suggests that the last word is the name of a language. But more distant past indicates that it is French
 - It is possible that the gap between the relevant information and where it is needed is very large

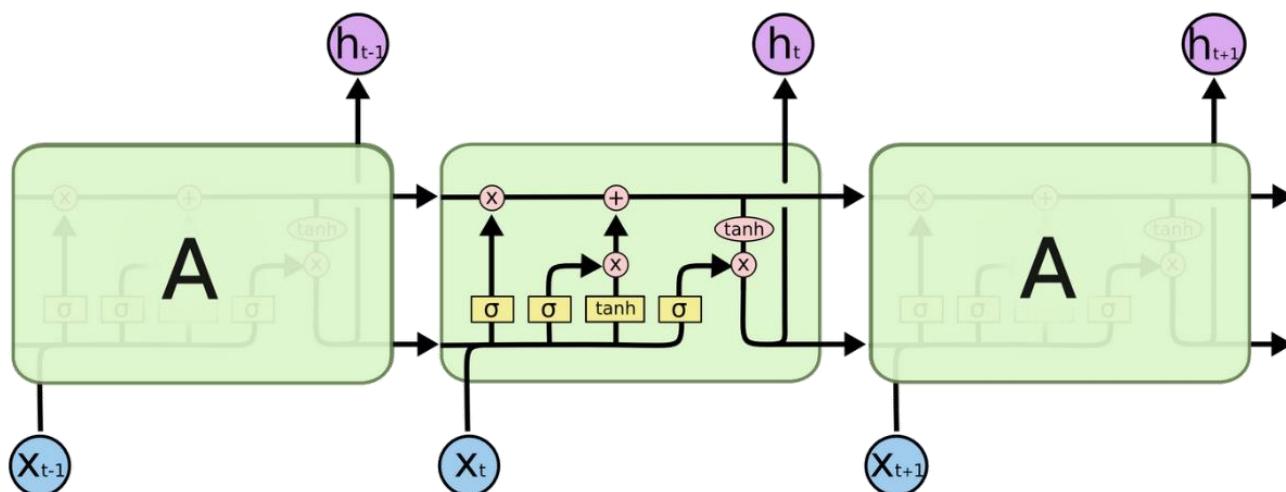


Long Short Term Memory

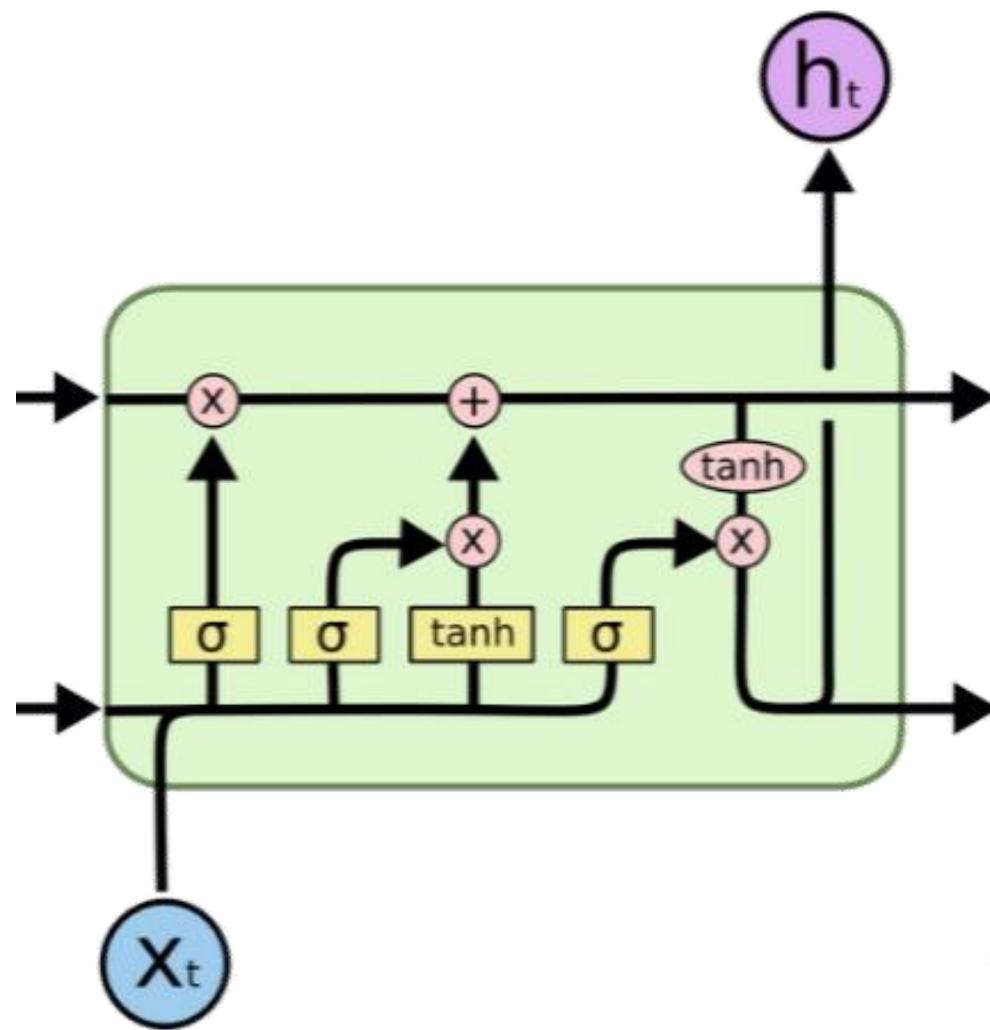
- Explicitly designed to avoid the long-term dependency problem
- RNNs have the form of a repeating chain structure
 - The repeating module has a simple structure such as tanh



- LSTMs also have a chain structure

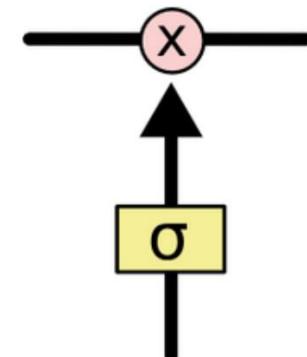
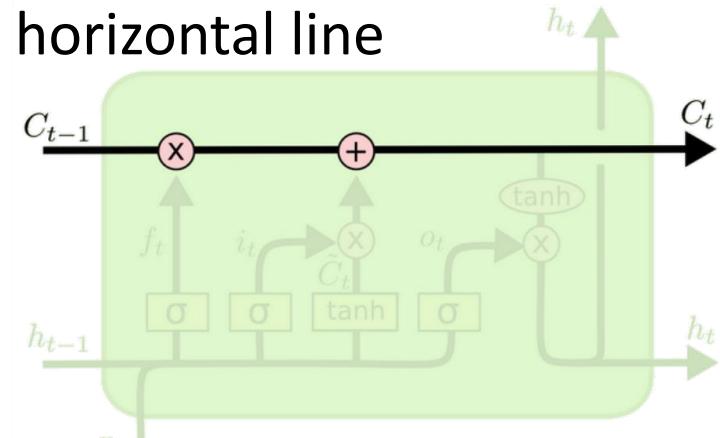


LSTM Unit



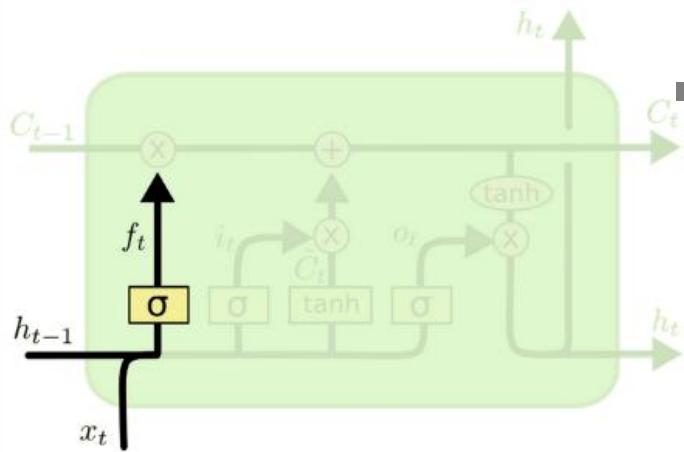
Core idea behind LSTM

- The key to LSTM is the cell state, C_t , the horizontal line running through the top of the diagram
 - Like a conveyor belt
 - Runs through entire chain with minor interactions
 - LSTM does have the ability to remove/add information to cell state regulated by structures called gates
- Gates are an optional way to let information through
- Consist of a sigmoid and a multiplication operation
- Sigmoid outputs a value between 0 and 1
 - 0 means let nothing through
 - 1 means let everything through
- LSTM has three of these gates, to protect and control cell state



Step-by-step LSM walk through

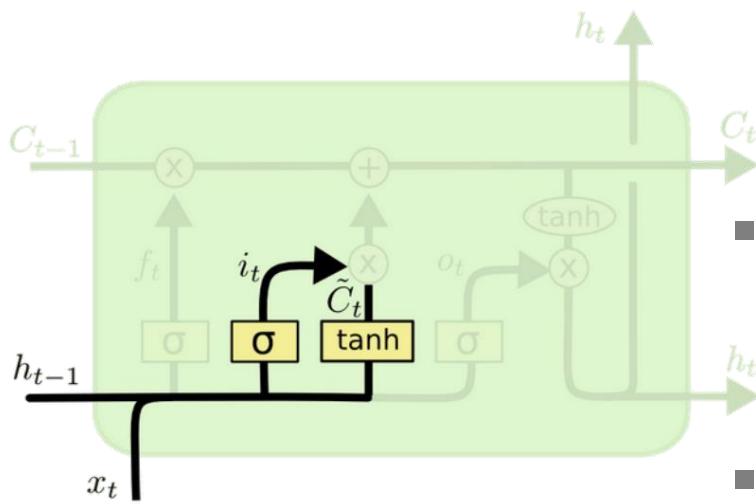
- Example: predict next word based on previous ones
 - Cell state may include the gender of the present subject
- First step: information to throw away from cell state



- Called forget gate layer
 - Uses $h^{(t-1)}$ and $x^{(t)}$ and outputs a number between 0 and 1 for each $C^{(t-1)}$
$$f^{(t)} = \sigma(W_f \cdot [h^{(t-1)}, x^{(t)}] + b_f)$$
- In language model
 - Consider trying to predict the next word based on all previous ones.
 - The cell state may include the gender of the present subject so that the proper pronouns can be used
 - When we see a new subject we want to forget old subject

LSM walk through: Second step

- Next step is to decide as to what new information we're going to store in the cell state



- In language model:

we'd want to add the gender of the new subject to the cell state, to replace the old One we are forgetting

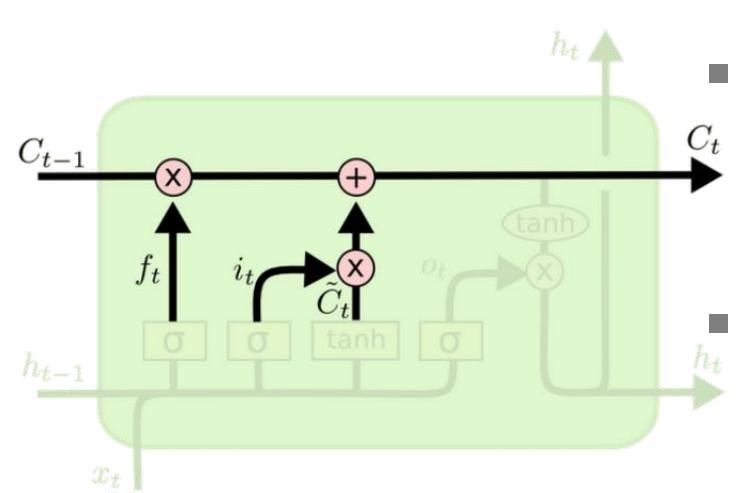
- First a sigmoid layer called Input gate layer:
 - decides which values we will update
- Next a tanh layer creates a vector of new candidate values $\tilde{C}^{(t)}$ that could be added to the state.
- In the third step we will combine these two to create an update to the state

$$i^{(t)} = \sigma(\mathbf{W}_i \cdot [\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + b_i)$$

$$\tilde{C}^{(t)} = \tanh(\mathbf{W}_c \cdot [\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + b_c)$$

LSM walk through: Third step

- It's now time to update old cell state C_{t-1} into new cell state C_t
 - The previous step decided what we need to do
 - We just need to do it



- We multiply the old state by $f^{(t)}$, forgetting the things we decided to forget earlier.
- Then we add $i^{(t)} \cdot \tilde{C}^{(t)}$

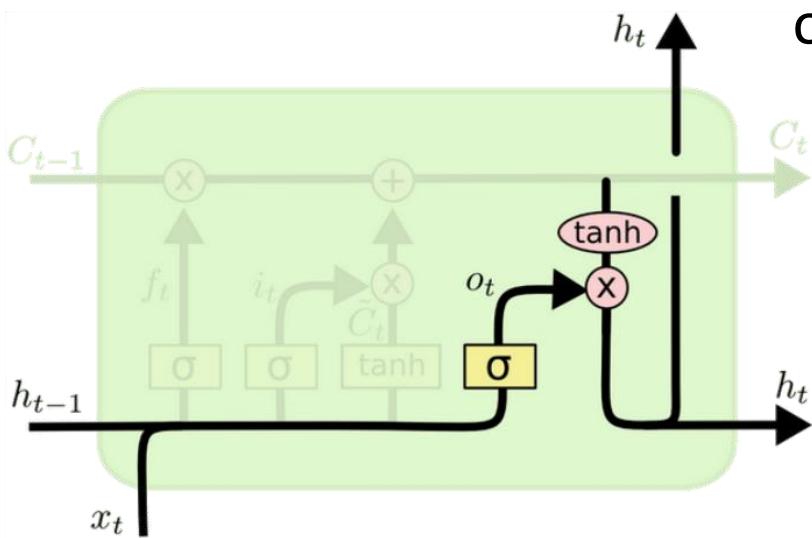
$$C^{(t)} = f^{(t)} \cdot C^{(t-1)} + i^{(t)} \cdot \tilde{C}^{(t)}$$

- In language model:

this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in previous steps

LSM walk through: Fourth step

- Finally we decide what we are going to output
 - This output will be a filtered version of our cell state



- First we run a sigmoid layer which decides what parts of cell state we're going to output.
- Then we put the cell state through tanh (to push values to be between -1 and 1)

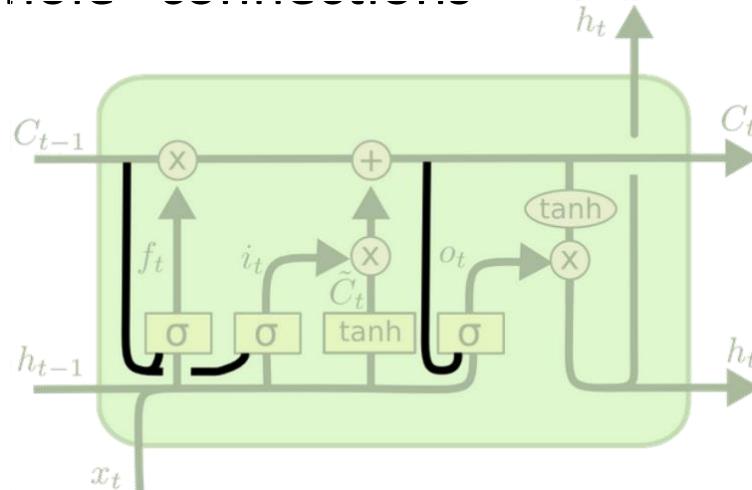
$$\begin{aligned}\mathbf{o}^{(t)} &= \sigma(\mathbf{W}_o \cdot [\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + \mathbf{b}_o) \\ \mathbf{h}^{(t)} &= \mathbf{o}_t \cdot \tanh(\mathbf{C}^{(t)})\end{aligned}$$

- In language model:

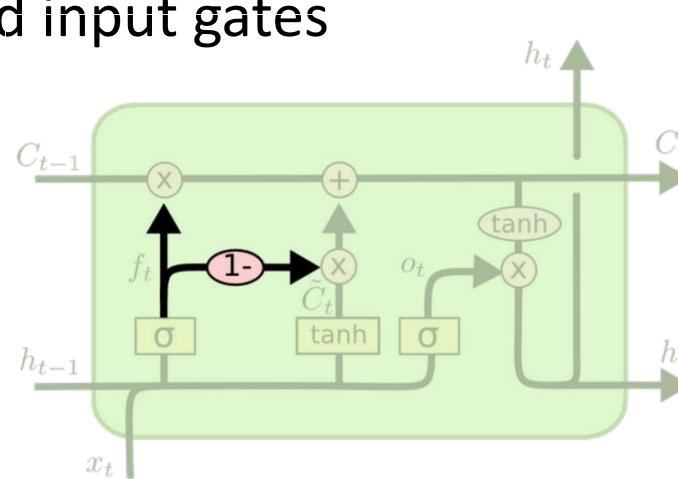
Since it just saw a subject it might want to output information relevant to a verb, in case that is what is coming next, e.g., it might output whether the subject is singular or plural so that we know what form a verb should be conjugated into if that's what follows next.

Variants of LSTM

- LSTM with “peephole” connections

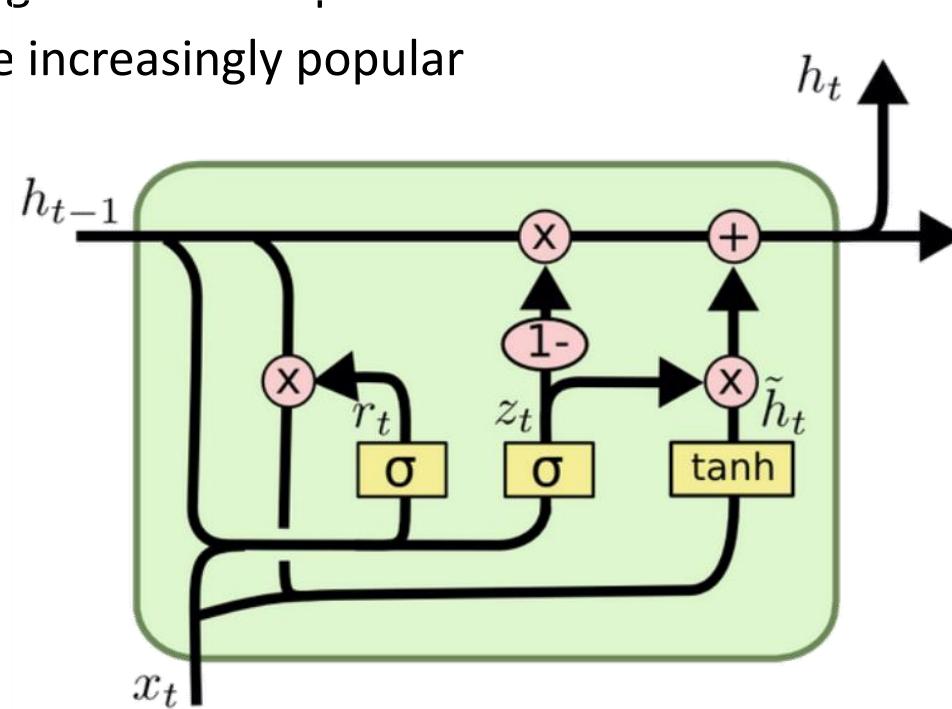


- Coupled forget and input gates



Gated Recurrent Unit (GRU)

- A dramatic variant of LSTM
 - It combines the forget and input gates into a single update gate
 - It also merges the cell state and hidden state, and makes some other changes
 - The resulting model is simpler than LSTM models
 - Has become increasingly popular



Idea of Gated RNNs

- Like leaky units, gated RNNs are based on the idea of creating paths through time that have derivatives that neither vanish nor explode
- Leaky units do this with connection weights that are manually chosen or were parameters α
 - generalizing the concept of discrete skipped connections
- Gated RNNs generalize this to connection weights that may change with each time step

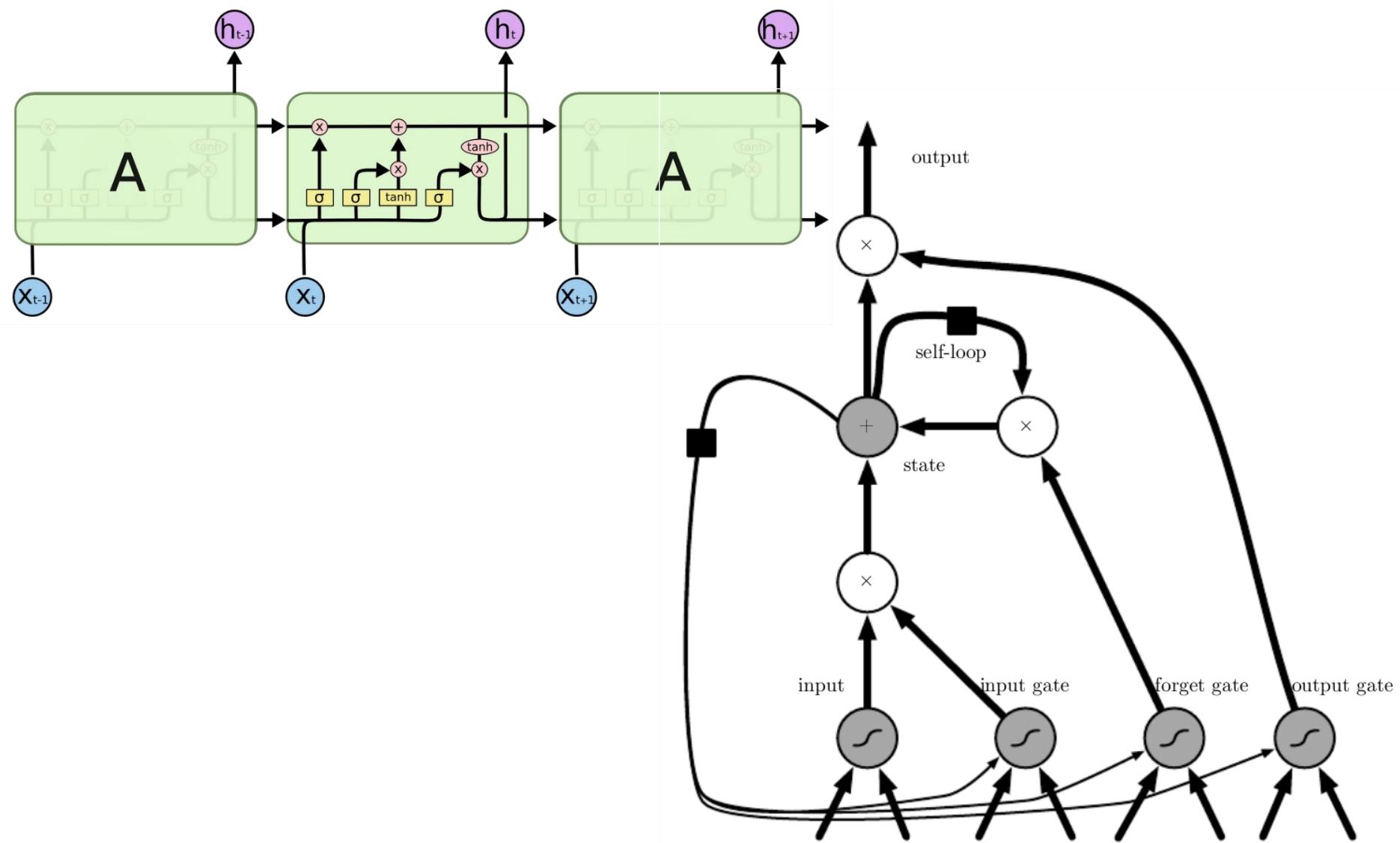
Accumulating Information over Longer Duration

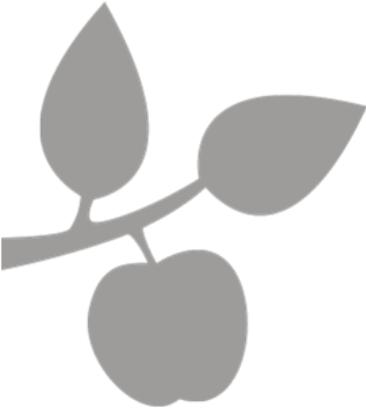
- Leaky Units Allow the network to accumulate information
 - Such as evidence for a particular feature or category
 - Over a long duration
- However, once the information has been used, it might be useful to forget the old state
 - E.g., if a sequence is made of sub-sequences and we want a leaky unit to accumulate evidence inside each sub-sequence, we need a mechanism to forget the old state by setting it to zero
- Gated RNN:
 - Instead of manually deciding when to clear the state, we want the neural network to learn to decide when to do it

LSTM

- Contribution of LSTMs:
 - clever idea of introducing self-loops to produce paths where the gradient can flow for long durations
- A crucial addition: make weight on this self-loop conditioned on the context, rather than fixed
 - By making weight of this self-loop gated (controlled by another hidden unit), time-scale can be changed dynamically
 - Even for an LSTM with fixed parameters, time scale of integration can change based on the input sequence
 - Because time constants are output by the model itself
- LSTM found extremely successful in:
 - Unconstrained handwriting recognition, Speech recognition
 - Handwriting generation, Machine Translation
 - Image Captioning, Parsing

LSTM Block Diagram





Recurrent Neural Networks

- **Introduction**
- **Unfolding a Graph**
- **Recurrent Neural Networks**
- **Long Term Memory**
 - Leaky Units
 - Long-Short Term Memory
- **Text in KERAS**

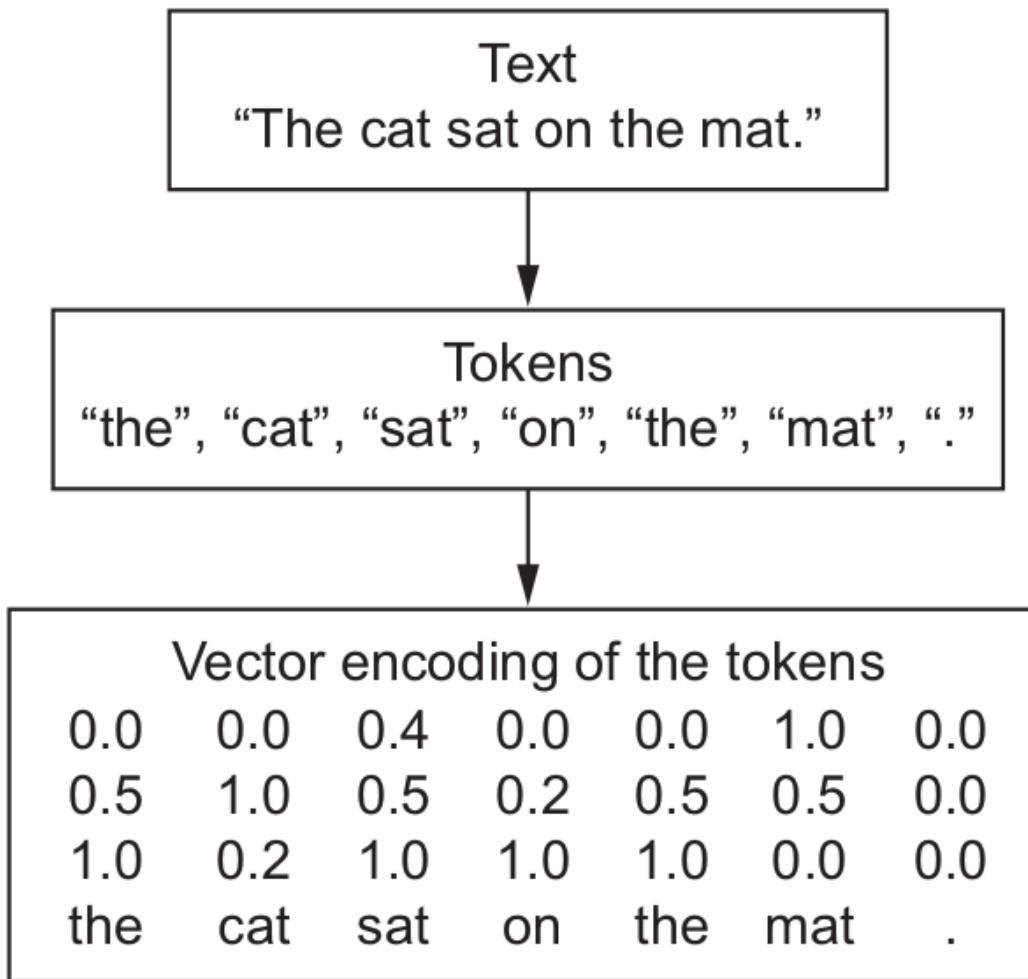
About Text

- Text is one of the most widespread forms of sequence data.
- Can be interpreted as sequence of characters or a sequence of words; interpretation as words more common.
- The deep-learning sequence-processing models can use text to produce a basic form of natural-language understanding
 - Document classification
 - Sentiment analysis
 - Author identification
 - question-answering (QA) (in a constrained context).

Text as Input

- We cannot feed text directly into networks
- They only “eat” numerical tensors
- We need to **vectorize** text into numeric tensors
 - Segment text into words, and transform each word into a vector.
 - Segment text into characters, and transform each character into a vector.
 - Extract n-grams of words or characters, and transform each n-gram into a vector.
 - N-grams are overlapping groups of multiple consecutive words or characters.

Example

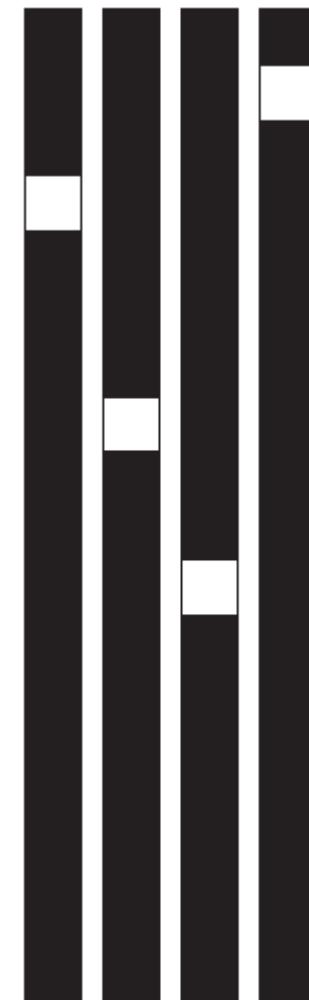


What are N-grams

- Word n-grams are groups of N (or fewer) consecutive words that you can extract from a sentence.
- Example: “The cat sat on the mat.”
- Set of 2-grams:
 - {"The", "The cat", "cat", "cat sat", "sat", "sat on", "on", "on the", "the", "the mat", "mat"}
- Set of 3-grams:
 - {"The", "The cat", "cat", "cat sat", "The cat sat", "sat", "sat on", "on", "cat sat on", "on the", "the", "sat on the", "the mat", "mat", "on the mat"}

One-hot Encoding of Words and Characters

- One-hot encoding is the most common, most basic way to turn a token into a vector.
 - Associate a unique integer index with every word (or token)
 - Turn the integer index i into a binary vector of size N (the size of the vocabulary); the vector is all zeros except for the i th entry, which is 1.
 - Like we have done for the encoding of categorical data



Using Keras for One-Hot Encoding

```
from keras.preprocessing.text import Tokenizer

samples = ['The cat sat on the mat.', 'The dog ate my homework.']

# Tokenizer, only consider the 1000 most common words
tokenizer = Tokenizer(num_words=1000)
tokenizer.fit_on_texts(samples)

# Turn strings into list of integers
sequences = tokenizer.texts_to_sequences(samples)

# Or directly get the one hot encoding
one_hot_results = tokenizer.texts_to_matrix(samples, mode='binary')

# Retrieve the word index
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

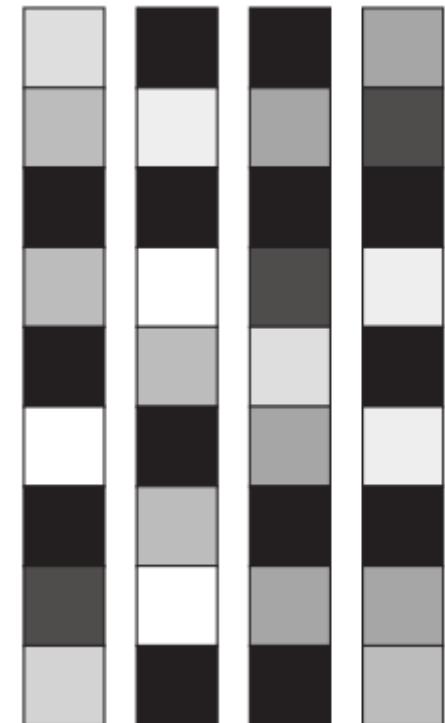
Vectorize a Sequence

```
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results
```

- You cannot feed a list of integer or one-hot encoded Matrix of different shapes into your model.
- Solution: One-hot encode your lists to turn them into vectors of 0s and 1s.
 - This would mean, for instance, turning the sequence [3, 5] into a 10,000-dimensional vector that would be all 0s except for indices 3 and 5, which would be 1s.
 - Then you could use as the first layer in your network a Dense layer, capable of handling floating-point vector data.

Word Embeddings

- Vectors obtained through one-hot encoding are binary, sparse (mostly made of zeros), and very high-dimensional (same dimensionality as the number of words in the vocabulary)
- Word embeddings are low-dimensional floating-point vectors (that is, dense vectors, as opposed to sparse vectors)



How to obtain Embeddings?

- Learn word embeddings jointly with the main task you care about (such as document classification or sentiment prediction).
 - In this setup, you start with random word vectors and then learn word vectors in the same way you learn the weights of a neural network.
- Load into your model word embeddings that were precomputed using a different machine-learning task than the one you're trying to solve. These are called pretrained word embeddings.

Embeddings in Keras

```
from keras.layers import Embedding  
embedding_layer = Embedding(1000, 64)
```

- The Embedding layer takes at least two arguments: the number of possible tokens (here, 1,000: 1 + maximum word index)
- The dimensionality of the embeddings (here, 64).
- The Embedding layer takes as input a 2D tensor of integers, of shape (samples, sequence_length)
 - Sequences that are shorter must be padded with zeros, sequences that are longer must be truncated.
- Returns a 3D floating-point tensor of shape (samples, sequence_length, embedding_dimensionality)

RNNs in Keras

- The simple RNN takes inputs of the shape (batch_size, timesteps, input_features)
- Can be run in two different modes:
 - It can return either the full sequences of successive outputs for each timestep:
(a 3D tensor of shape (batch_size, timesteps, output_features))
 - Only the last output for each input sequence
(a 2D tensor of shape (batch_size, output_features)).
- These two modes are controlled by the return_sequences constructor argument.
- **Available RNNs: SimpleRNN, GRU, LSTM, ...**

RNNs in Keras

```
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32))
model.summary()
```

Layer (type)	Output Shape	Param #
=====		
embedding_22 (Embedding)	(None, None, 32)	320000
=====		
simplernn_10 (SimpleRNN)	(None, 32)	2080
=====		
Total params: 322,080		
Trainable params: 322,080		
Non-trainable params: 0		

- Creates a simple RNN which only produces one output

RNNs in Keras

```
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32, return_sequences=True))
model.summary()
```

Layer (type)	Output Shape	Param #
=====		
embedding_22 (Embedding)	(None, None, 32)	320000
=====		
simplernn_10 (SimpleRNN)	(None, None, 32)	2080
=====		
Total params: 322,080		
Trainable params: 322,080		
Non-trainable params: 0		

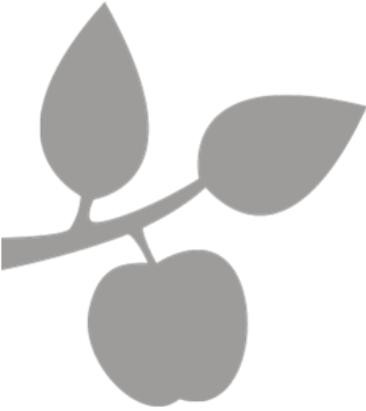
- Creates a simple RNN which produces an output at every position

Stack RNNs

```
model = Sequential()
model.add(Embedding(10000, 32))

# All RNNs need to return a value every timestep
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32, return_sequences=True))

# The last layer of RNN can only return one value
model.add(SimpleRNN(32))
```

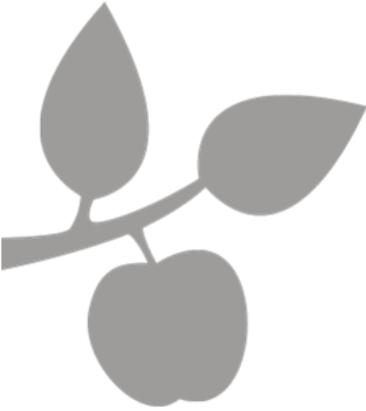


DM873

Deep Learning

Spring 2019

Lecture 11 – Optimization for Deep Learning



Optimization for Deep Learning

- **Learning vs. Optimization**
- **Neural Network Optimization**
- **Momentum**
- **Parameter Initialization**
- **Adaptive Learning**
- **Batch Normalization**
- **Pre-Training**

Learning vs Pure Optimization

- Optimization algorithms for deep learning differ from traditional optimization in several ways:
 - Machine learning acts indirectly
 - We care about some performance measure P defined wrt the training set which may be intractable
 - We reduce a different cost function $J(\theta)$ in the hope that doing so will reduce P
- Pure optimization: minimizing J is a goal in itself
- Optimizing algorithms for training Deep models:
 - Includes specialization on specific structure of ML objective function

Typical Cost Function

- Cost is average over the training set

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x}, \mathbf{y}) \sim \hat{p}_{data}}(L(f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y}))$$

- $f(\mathbf{x}; \boldsymbol{\theta})$ is the predicted output when the input is \mathbf{x}
- In supervised learning \mathbf{y} is target output
- L is the per-example loss function
- \hat{p}_{data} is the empirical distribution
- We consider the unregularized supervised case
 - Arguments of L are $f(\mathbf{x}; \boldsymbol{\theta})$ and y

Objective wrt data generation is risk

- Objective function wrt training set is

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(x,y) \sim \hat{p}_{data}}(L(f(x; \boldsymbol{\theta}), y))$$

- We would prefer to minimize an objective function where expectation is across the data generating distribution p_{data}

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(x,y) \sim p_{data}}(L(f(x; \boldsymbol{\theta}), y))$$

- The goal of a machine learning algorithm is to reduce this expected generalization error

Surrogate Loss: Log-likelihood

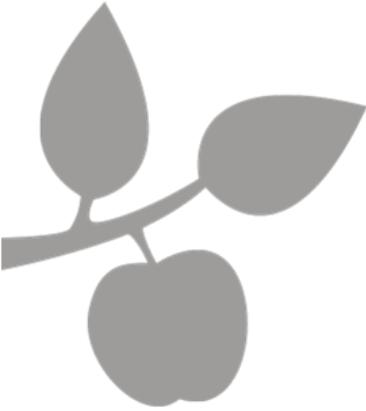
- Exactly minimizing 0-1 loss is typically intractable (exponential in the input dimension) even for a linear classifier
- In such situations use a surrogate loss function
 - Acts has a proxy but has advantages
- Negative log-likelihood of the correct class is a surrogate for 0-1 loss
 - It allows model to estimate conditional probability of classes given the input
 - If it does that well it can pick the classes that yield the least classification error in expectation

Surrogate may even learn more

- Using log-likelihood surrogate,
 - Test set 0-1 loss continues to decrease even after the training set 0-1 loss has reached zero when training
 - Because one can improve classifier robustness by further pushing the classes apart
 - Results in a more confident and robust classifier
 - Thus extracting more information from the training data than with minimizing 0-1 loss

Learning does not stop at minimum

- Important difference between Optimization in general and Optimization for Training:
 - Training does not halt at a local minimum
 - Early Stopping: Instead Learning algorithm halts on an early stopping criterion
 - Based on a true underlying loss function
 - Such as 0-1 loss measured on a validation set
 - Designed to cause algorithm to stop overfitting
- Often stops when derivatives are still large
 - In pure optimization, algorithm considered to converge when derivatives are very small



Optimization for Deep Learning

- Learning vs. Optimization
- **Problems with Neural Network Optimization**
- Momentum
- Parameter Initialization
- Adaptive Learning
- Batch Normalization
- Pre-Training

Optimization is a difficult task

- Traditionally ML has avoided difficulty of general optimization by carefully designing the objective function and constraints to ensure that optimization problem is convex
- When training neural networks, we must confront the nonconvex case
- We summarize challenges in optimization for training deep models

Local Minima

- In convex optimization, problem is one of finding a local minimum
- Some convex functions have a flat region rather than a global minimum point
- Any point within the flat region is acceptable
- With non-convexity of neural nets many local minima are possible
- Many deep models are guaranteed to have an extremely large no. of local minima
- This is not necessarily a major problem

Model Identifiability

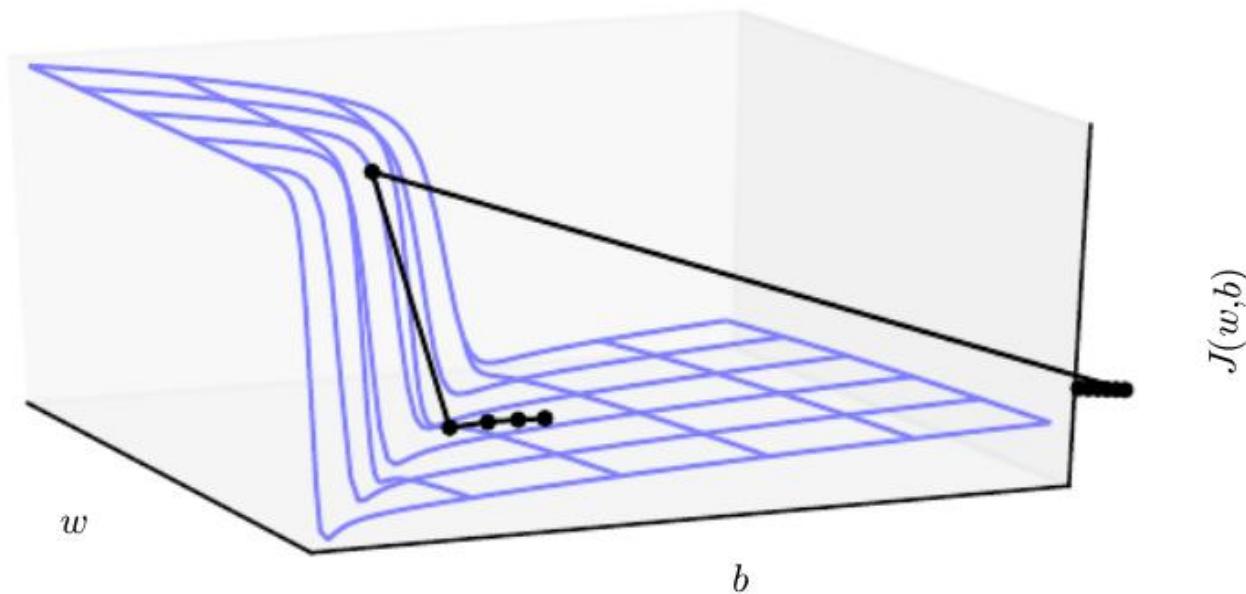
- Model is identifiable if large training sample set can rule out all but one setting of parameters
 - Models with latent variables are not identifiable
 - Because we can exchange latent variables
 - If we have m layers with n units each there are $n!^m$ ways of arranging the hidden units
 - This non-identifiability is weight space symmetry
 - Another is scaling incoming weights and biases
 - By a factor α and scale outgoing weights by $1/\alpha$
- Even if a neural net has uncountable no. of minima, they are equivalent in cost
 - So not a problematic form of non-convexity

Plateaus, Saddle Points etc

- More common than local minima/maxima are: saddle points
 - At saddle, Hessian has both positive and negative Eigenvalues
 - Positive: cost greater than saddle point
 - Negative: costs lower than saddle point
 - In low dimensions:
 - Local minima are more common
 - In high dimensions:
 - Local minima are rare, saddle points more common
- For Newton's saddle points pose a problem
 - Explains why second-order methods have not replaced gradient descent

Cliffs and Exploding Gradients

- Neural networks with many layers have steep regions i.e., cliffs
 - Result from multiplying several large weights
 - E.g., RNNs with many factors at each time step
- Gradient update step can move parameters extremely far, jumping off cliff altogether
- Cliffs dangerous from either direction
- Gradient clipping heuristics can be used



Inexact Gradients

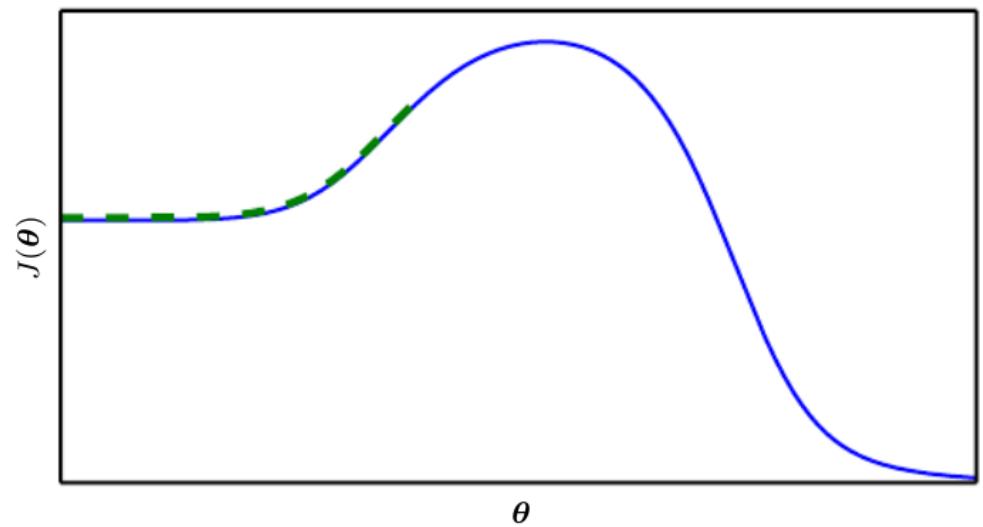
- Optimization algorithms assume we have access to exact gradient or Hessian matrix
- In practice we have a noisy or biased estimate
 - Every deep learning algorithm relies on sampling-based estimates
 - In using minibatch of training examples
 - In other case, objective function is intractable
 - In which case gradient is intractable as well

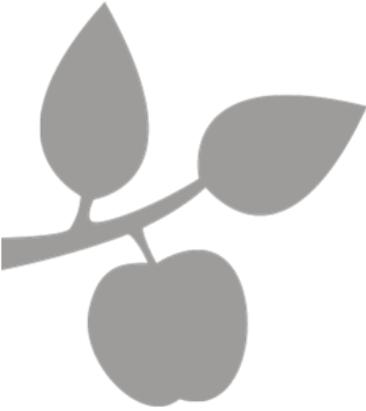
Poor Correspondence between Local and Global Structure

- It can be difficult to make a single step if:
 - $J(\theta)$ is poorly conditioned at the current point θ
 - θ lies on a cliff
 - θ is a saddle point hiding the opportunity to make progress downhill from the gradient
- It is possible to overcome all these problems and still perform poorly:
 - if the direction that makes most improvement locally does not point towards distant regions of much lower cost

Need for good initial points

- Optimization based on local downhill moves can fail if local surface does not point towards the global solution
- Research directions are aimed at finding good initial points for problems with a difficult global structure
 - For example: Trajectory of circumventing mountains may be long and result in excessive training time





Optimization for Deep Learning

- Learning vs. Optimization
- Problems with Neural Network Optimization
- **Momentum**
- Parameter Initialization
- Adaptive Learning
- Batch Normalization
- Pre-Training

Stochastic Gradient Descent

- We have seen:
 - Gradient descent that follows the gradient of an entire training set downhill
 - This can be accelerated considerably by SGD which follows the gradient of randomly selected minibatches downhill
- SGD and its variants are the most used optimization algorithms for ML in general and deep learning in particular
 - Average gradient on a minibatch is an estimate of the gradient
- A crucial parameter for Algorithm SGD is the learning rate ε
- It is necessary to let the learning rate decrease
 - At iteration k it is ε_k

Need for decreasing learning rate

- Batch gradient descent (i.e. using all samples) can use a fixed learning rate
 - Since true gradient becomes small and then 0
- SGD has a source of error
 - Random sampling of m training samples
 - Sufficient condition for SGD convergence

$$\sum \varepsilon = \infty; \sum \varepsilon^2 < \infty$$

- Common to decay learning rate linearly until iteration τ : $\varepsilon_k = (1 - \alpha)\varepsilon_0 + \alpha\varepsilon_\tau$ with $\alpha = \frac{k}{\tau}$.
- After iteration τ , it is common to leave ε constant

The Momentum Method

- SGD is a popular optimization strategy
- But it can be slow
- Momentum method accelerates learning, when:
 - Facing high curvature
 - Small but consistent gradients
 - Noisy gradients
- Algorithm accumulates moving average of past gradients and move in that direction, while exponentially decaying

The Momentum Method

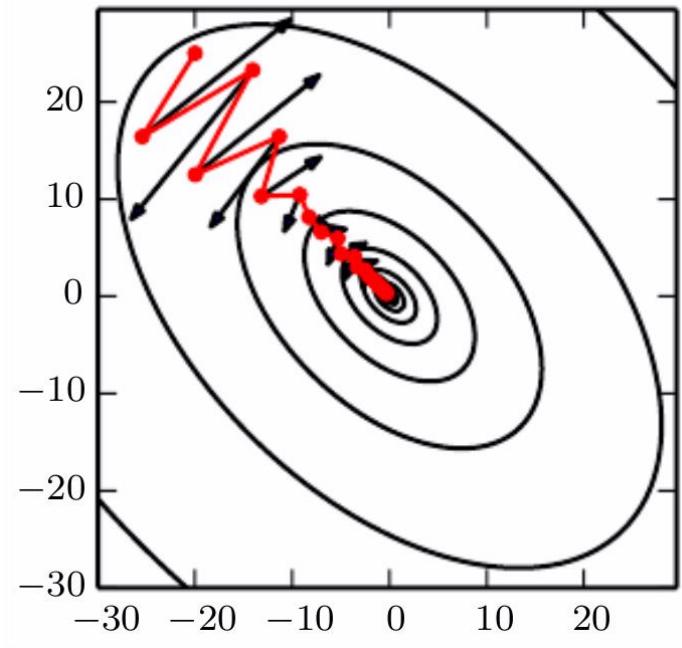
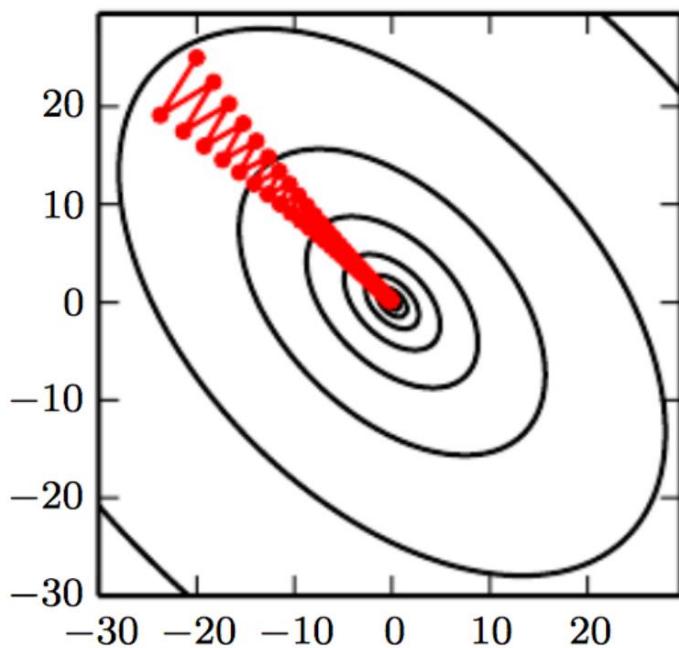
- Introduce variable v , or velocity
 - It is the direction and speed at which parameters move through parameter space
 - Momentum is physics is mass times velocity
 - The momentum algorithm assumes unit mass
 - A hyperparameter $\alpha \in [0,1)$ determines exponential decay

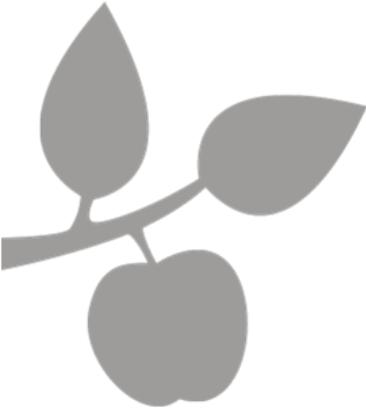
$$v \leftarrow \alpha v - \varepsilon \nabla_{\theta} \left(\frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}; \theta), y^{(i)}) \right)$$

$$\theta \leftarrow \theta + v$$

- The larger α is relative to ε , the more previous gradients affect the current direction

SGD with vs. w/o Momentum





Optimization for Deep Learning

- Learning vs. Optimization
- Problems with Neural Network Optimization
- Momentum
- Parameter Initialization
- Adaptive Learning
- Batch Normalization
- Pre-Training

Types of Initialization

1. Non-iterative optimization requires no initialization
 - Simply solve for solution point
2. Iterative but converge regardless of initialization
 - Acceptable solutions in acceptable time
3. Iterative but affected by choice of Initialization
 - Deep learning training algorithms are iterative
 - Initialization determines whether it converges at all
 - Can hugely determine how quickly learning converges

Modern Initialization Strategies

- They are simple and heuristic
- Based on achieving nice properties
- But problem is a difficult one
 - Some initial points are beneficial for optimization but detrimental to generalization
- Only property known with certainty: Initial parameters must be chosen to **break symmetry**
 - If two hidden units have the same inputs and same activation function then they must have different initial parameters
 - Usually best to initialize each unit to compute a different function
 - This motivates use random initialization of parameters

Choice of biases

- Biases for each unit are heuristically chosen constants
- Only the weights are initialized randomly
- Extra parameters such as conditional variance of a prediction are constants like biases

Weights drawn from Gaussian

- Weights are almost always drawn from a Gaussian or uniform distribution
 - Choice of Gaussian or uniform does not seem to matter much but not studied exhaustively
- Scale of the initial distribution does have an effect on outcome of optimization and ability to generalize
 - Larger initial weights will yield stronger symmetry-breaking effect, helping avoid redundant units
 - Too large may result in exploding values

Heuristics for initial scale of weights

- One heuristic is to initialize the weights of a fully connected layer with N inputs and M outputs by sampling each weights from $\text{Uniform}(-r, r)$ with

$$r = \frac{1}{\sqrt{N}}$$

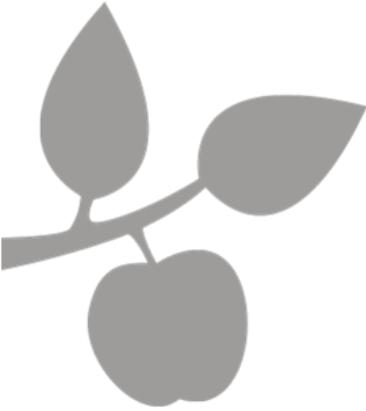
- Another heuristic is normalized initiation with

$$r = \sqrt{\frac{6}{N + M}}$$

- Which is a compromise between the goal of initializing all layers to have the same activation variance and the goal of having all layers having the same gradient variance

Initialization for the biases

- Bias settings must be coordinated with setting weights
- Setting biases to zero is compatible with most weight initialization schemes
- Situations for nonzero biases:
 - Bias for an output unit: initialize to obtain right marginal statistics for output
 - Set bias to inverse of activation function applied to the marginal statistics of the output in the training set
 - Assuming that the weights in the beginning are so small, that output is driven only by biases
 - Choose bias to causing too much saturation at initialization



Optimization for Deep Learning

- Learning vs. Optimization
- Problems with Neural Network Optimization
- Momentum
- Parameter Initialization
- Adaptive Learning
- Batch Normalization
- Pre-Training

Learning Rate is Crucial

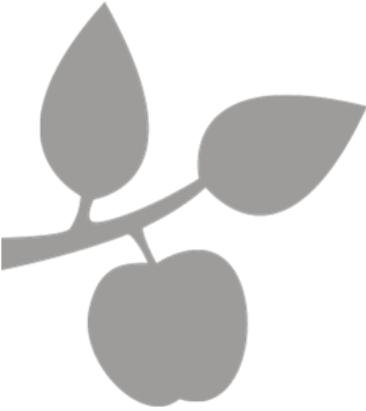
- Learning rate: most difficult hyperparam to set
- It significantly affects model performance
- Cost is highly sensitive to some directions in parameter space and insensitive to others
 - Momentum helps but introduces another hyperparameter
 - Is there another way?
- If direction of sensitivity is axis aligned, separate learning rate for each parameter and adjust them through learning

Heuristic Approaches

- Delta-bar-delta Algorithm
 - (Applicable to only full batch optimization)
 - If partial derivative of the loss wrt to a parameter remains the same sign, the learning rate should increase
 - If that partial derivative changes sign, the learning rate should decrease
- AdaGrad
 - Individually adapts learning rates of all params
 - By scaling them inversely proportional to the sum of the historical squared values of the gradient
- RMSProp
 - Modifies AdaGrad for a nonconvex setting
 - Change gradient accumulation into exponentially weighted moving average
 - Converges rapidly when applied to convex function

RMSProp is popular

- RMSProp is an effective practical optimization algorithm
- Go-to optimization method for deep learning practitioners
- Choosing the Right Optimizer
 - We have discussed several methods of optimizing deep models by adapting the learning rate for each model parameter
 - Which algorithm to choose?
 - There is no consensus
 - Most popular algorithms actively in use:
 - SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta and Adam
 - Choice depends on user's familiarity with algorithm



Optimization for Deep Learning

- Learning vs. Optimization
- Problems with Neural Network Optimization
- Momentum
- Parameter Initialization
- Adaptive Learning
- Batch Normalization
- Pre-Training

Topics in Batch Normalization

- Batch normalization: exciting recent innovation
- Method is to replace activations with zero-mean with unit variance activations
- Motivation is difficulty of choosing learning rate ϵ in deep networks
 - Method adds an additional step between layers, in which the output of the earlier layer is normalized
 - By standardizing the mean and standard deviation of each individual unit
 - It is a method of adaptive re-parameterization
 - It is not an optimization algorithm at all
 - A method to reduce internal covariate shift in neural networks

Motivation: Difficulty of composition

- Very deep models involve compositions of several functions or layers

$$f(\mathbf{x}, \mathbf{w}) = f^{(l)} \left(\dots f^{(3)} \left(f^{(2)} \left(f^{(1)}(\mathbf{x}) \right) \right) \right)$$

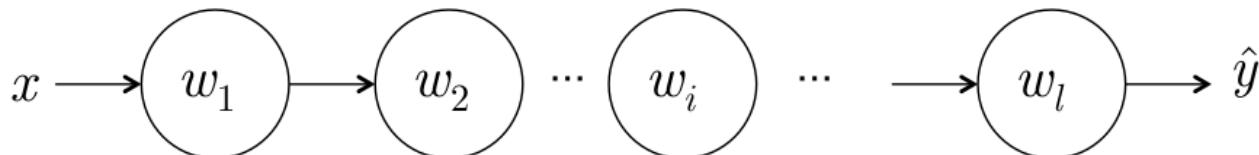
- The gradient tells us how to update each parameter

$$\mathbf{w}^{\tau+1} = \mathbf{w}^\tau - \epsilon \nabla_{\mathbf{w}} J(f(\mathbf{x}, \mathbf{w}), y)$$

- Under assumption that other layers do not change
- **BUT:** We update all l layers simultaneously
- When we make the change unexpected results can happen
- Because many functions are changed simultaneously, effects can accumulate

Choosing learning rate ϵ in multilayer

- Simple example:
 - l layers, one multiplication unit per layer, no activation function



- Network simply computes

$$\hat{y} = x \cdot w_1 \cdot w_2 \dots \cdot w_l$$

- Output of Layer i is $h_i = h_{i-1}w_i$
- Output is a linear function of input x but a nonlinear function of the weights w_i

Gradient in Simple example

- Suppose our cost function has put a gradient of 1 on \hat{y} , so we wish to decrease \hat{y} slightly.
- The back-propagation algorithm can then compute a gradient

$$\mathbf{g} = \nabla_{\mathbf{w}} \hat{y}$$

- Which corresponds to the gradient \mathbf{g} evaluated at $y = \hat{y}$
- When using the update $\mathbf{w} \leftarrow \mathbf{w} - \epsilon \mathbf{g}$ predicts that \hat{y} decreases by $\epsilon \mathbf{g}^T \mathbf{g}$
- If we want to decrease \hat{y} by 0.1, we could set the learning rate to

$$\epsilon = \frac{0.1}{\mathbf{g}^T \mathbf{g}}$$

Difficulty of Multilayer learning Rate

- With the first order information, we would set the learning rate to

$$\epsilon = \frac{0.1}{\mathbf{g}^T \mathbf{g}}$$

- Problem: We have to deal with many 2nd, 3rd ... effects. The new value in fact is

$$x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2) \dots (w_l - \epsilon g_l)$$

- One example for a second order term would be $\epsilon^2 g_1 g_2 \prod_{i=3}^l w_i$
- This term might be negligible if $\prod_{i=3}^l w_i$ is small or exponentially large if weights are larger than 1.
- This makes it very hard to choose ϵ because the effects of an update for one layer depend so strongly on all other layers
 - Higher order methods tackle this problem, but still only for small l

The Batch Normalization Solution

- Provides an elegant way of re-parameterizing almost any network
- Significantly reduces the problem of coordinating updates across many layers
- Can be applied to any input or hidden layer in a network

Batch Normalization Equations

- \mathbf{H} : minibatch of activations of layer to normalize

- arranged as a design matrix
- With activations for each example appearing in a row

$$H = \begin{bmatrix} \text{units} & \xrightarrow{\hspace{1cm}} \\ a_{1,1} & a_{1,2} & a_{1,3} \\ \cdot & \cdot & \cdot \\ a_{N,1} & a_{N,2} & a_{N,3} \end{bmatrix}$$

samples

- To normalize \mathbf{H} we replace it with $\mathbf{H}' = \frac{\mathbf{H} - \mu}{\sigma}$
 - Where μ is a vector containing the mean of each unit and σ is the std. deviation of each unit
 - The arithmetic here is based on broadcasting the vector μ and the vector σ to be applied to every row of \mathbf{H}
 - Within each row, the arithmetic is element-wise
 - $H_{i,j}$ is normalized by subtracting μ_j & dividing by σ_j

Normalization Details

- Rest of the network operates on \mathbf{H}' in the same way that the original network operated on \mathbf{H}
- At training time

$$\mu = \frac{1}{m} \sum_i H_i \quad \text{and} \quad \sigma = \sqrt{\delta + \frac{1}{m} \sum_i (H - \mu)_i^2}$$

- where δ is a small positive value such as 10^{-8} imposed to avoid encountering the undefined gradient of $z = 0$
 - Crucially we back propagate through these operations for computing the mean and std dev
 - And for applying them to normalize \mathbf{H}
 - This means that the gradient will never propose an operation that acts simply to increase std dev or mean of h_i , the normalization operations remove the effect of such an action and aero out the component in the gradient

Batch Normalization at Test time

- At test time, μ and σ may be replaced by running averages that were collected during training time
- This allows the model to be evaluated on a single example without needing to use definitions of μ and σ that depend on an entire minibatch

Revisiting the simple example

- Revisiting the $\hat{y} = x \cdot w_1 \cdot w_2 \dots \cdot w_l$ example we can mostly resolve the difficulties in learning the model by normalizing h_{l-1}
- Suppose that x is drawn from a unit Gaussian
- Then h_{l-1} will also come from a Gaussian, because the transformation from x to h_l is linear
- However h_{l-1} will no longer have zero mean, unit variance

Restoring zero-mean unit variance

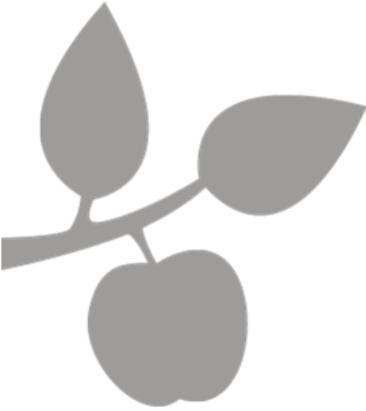
- After applying batch normalization, we obtain the normalized \hat{h}_{l-1} that restores zero mean and unit variance
 - For almost any update to the lower layers, \hat{h}_{l-1} will remain a unit Gaussian
 - Output \hat{y} may be learned as a simple linear function $\hat{y} = w_l \hat{h}_{l-1}$
- Learning in this model is now very simple
 - Because parameters at the lower layers do not have an effect in most cases
- Their output is always renormalized to a unit Gaussian

Batch Normalization -> learning easy

- Without normalization, updates would have an extreme effect of the statistics of h_{l-1}
- Batch normalization has thus made this model easier to learn
- In this example the ease of learning came from making the lower layers useless
 - In our linear example: Lower layers not harmful but not beneficial either
 - Because we have normalized-out all effects of higher order than 1st and 2nd order stats
- In a deep neural network lower levels still can perform useful nonlinear transformations

Reintroducing Expressive Power

- Normalizing the mean and standard deviation can reduce the expressive power of the neural network containing that unit
- To maintain the expressive power replace the batch of hidden unit activations \mathbf{H}' with $\gamma\mathbf{H}' + \beta$
 - γ and β are learned parameters that allow the new variable to have any mean and standard deviation
- Why did we normalize to zero and std. dev. when we then allow all means and deviations again?
 - Has different learning dynamics than unnormalized approach
 - It has the same power as \mathbf{H} , but the values of \mathbf{H} were determined by a complicated interaction between the parameters in the layers below
 - The new parametrization, the mean is only determined by β



Optimization for Deep Learning

- Learning vs. Optimization
- Problems with Neural Network Optimization
- Momentum
- Parameter Initialization
- Adaptive Learning
- Batch Normalization
- Pre-Training

Motivation

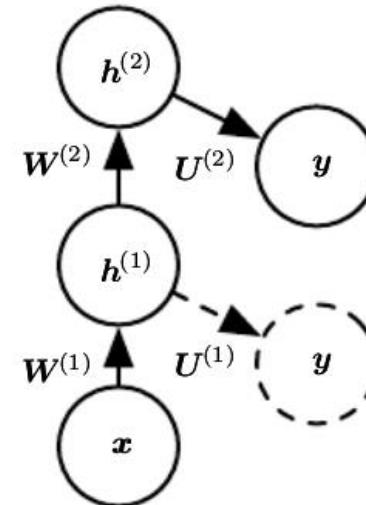
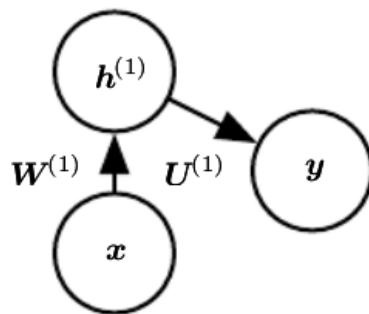
- Sometimes, directly training a model to solve a specific task can be too ambitious, if:
 - Model is too complex and hard to optimize
 - The task is very difficult
- It may be more effective to
 - Train a simpler model to solve the task, then move on to confront the final task
 - Methods collectively known as pretraining

Greedy Supervised Pretraining

- Greedy Algorithm:
 1. Break a problem into many components
 2. Solve for the optimal version of each component in isolation
 3. Combine the solutions
- Combining the component solutions may not yield an optimal complete solution
- However, greedy algorithms can be computationally much cheaper than algorithms that solve for the best joint solution
- Quality of a greedy solution is often acceptable if not optimal
- Initializing the joint optimization algorithm with a greedy solution can speed it up and improve the quality of the solution

Training each layer separately

- Supervised learning involving only a subset of the layers in the final neural network
- An example of greedy supervised pretraining
 - In which each added hidden layer is pretrained as part of a shallow supervised MLP
 - Taking as input the output of the previously trained hidden layer



Extension to Transfer Learning

- Pretraining extends the idea to transfer learning
- Pretrain convolutional net with k layers on tasks
 - E.g. on a subset of 1000 ImageNet object categories
- Then initialize same-size network with the first k layers of the first net
 - All layers of second network (with upper layers initialized randomly) are then jointly trained to perform a different set of tasks
 - E.g. another subset of 1000 ImageNet categories, with fewer training examples than for the first set of tasks

FitNets

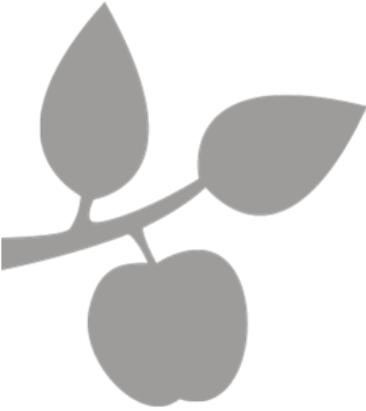
- While depth improves performance, it also makes gradient-based training more difficult since deeper networks are more non-linear.
- Solution is to train a network with low enough depth (e.g. 5) and great enough width (no. of units per layer) to be easy to train
- This network becomes a teacher for a second network, designated the student
- Student network is much deeper and thinner and would be difficult to train with SGD, (e.g., 15-20 layers)

Training the student network

- Task is made easier by training student network not only to predict output for original task, but also to predict value of middle layer of the teacher network
- This extra task provides a set of hints about how the hidden layers should be used and can simplify the optimization problem
- Additional parameters are introduced to regress the middle layer of the 5-layer teacher network from the middle layer of the deeper student network

Predicting Intermediate Layers

- Instead of predicting the final classification target, the objective is to predict the middle hidden layer of the teacher network
- Objectives of Lower layers of student network:
 1. Help outputs of student network accomplish task
 2. Predict intermediate layer of the teacher network
- A thin-deep network may be more difficult to train than a wide-shallow network, but may generalize better and has lower computational cost if it is thin enough to have far fewer parameters.

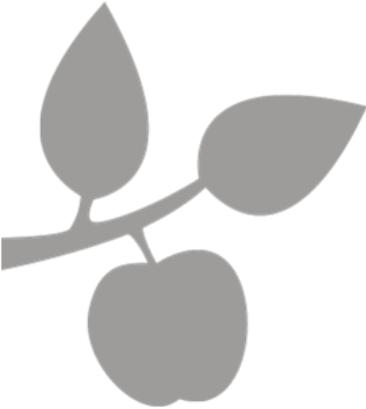


DM873

Deep Learning

Spring 2019

Lecture 12 – Autoencoders

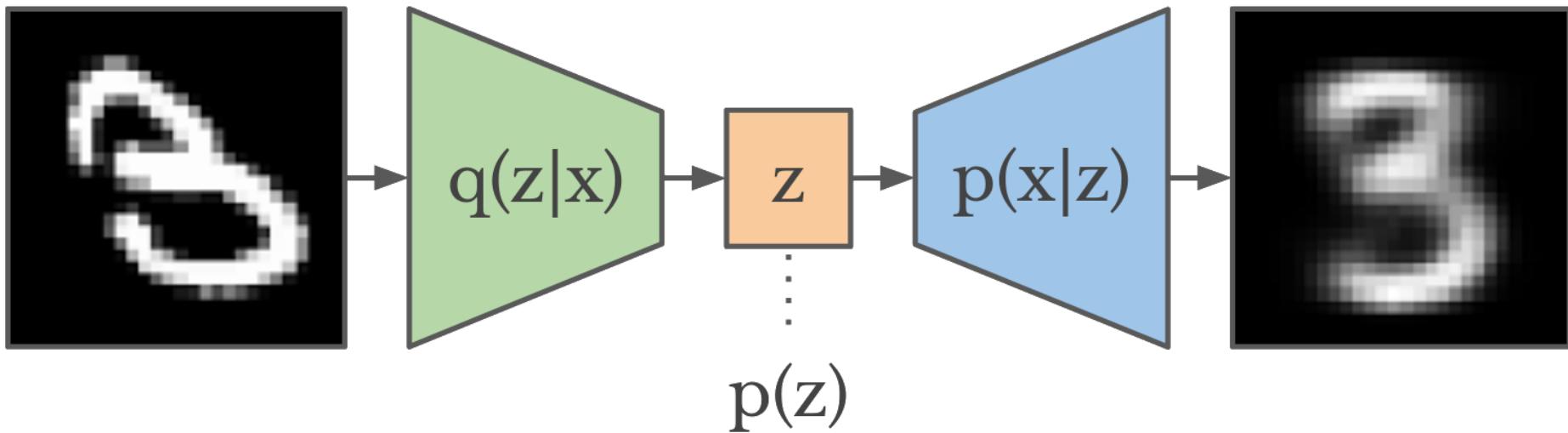


Autoencoders

- **Introduction**
- **Undercomplete Autoencoder**
- **Regularized Autoencoder**

What is an Autoencoder?

- An autoencoder is a data compression algorithm
 - Typically an artificial neural network trained to copy its input to its output



- Normally two stages:
 1. Compress input to (normally) smaller internal representation
 2. Reconstruct original input as closely as possible

Not to Mistake with General Data Compression

- Autoencoders are data-specific
 - i.e., only able to compress data similar to what they have been trained on
- This is different from, say, MP3 or JPEG compression algorithm
 - Which make general assumptions about "sound/images", but not about specific types of sounds/images
 - Autoencoder for pictures of cats would do poorly in compressing pictures of trees
- Autoencoders are lossy
 - i.e., exact reconstruction is normally not possible
- **Autoencoders are learnt**

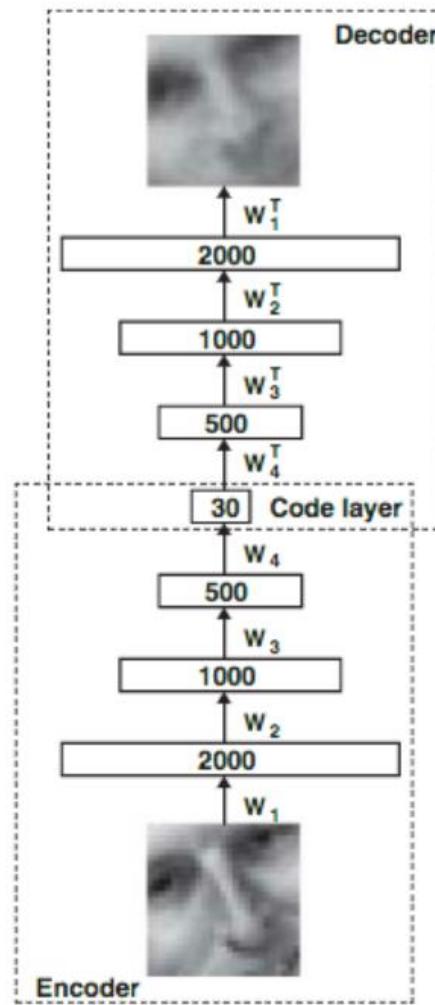
Rationale of an Autoencoder

- An autoencoder that simply learns to set $g(f(x)) = x$ everywhere is not really useful
- Autoencoders are designed to be unable to copy perfectly
 - They are restricted in ways to copy only approximately
 - Copy only input that resembles training data
- Because model is forced to prioritize which aspects of input should be copied, it often learns useful properties of the data
- Modern autoencoders have generalized the idea of encoder and decoder beyond deterministic functions to stochastic mappings $p_{\text{encoder}}(h|x)$ and $p_{\text{decoder}}(x|h)$

Autoencoder History

- Part of neural network landscape for decades
- Traditionally used for dimensionality reduction and feature learning
- Recent theoretical connection between autoencoders and latent variable models have brought them into forefront of generative models

Typical Autoencoder Architecture



Training and Loss Function

- Encoder f and decoder g

- $f: X \rightarrow h$
- $g: h \rightarrow X$
- $\text{argmin}_{f,g} \|X - (f \circ g)X\|^2$

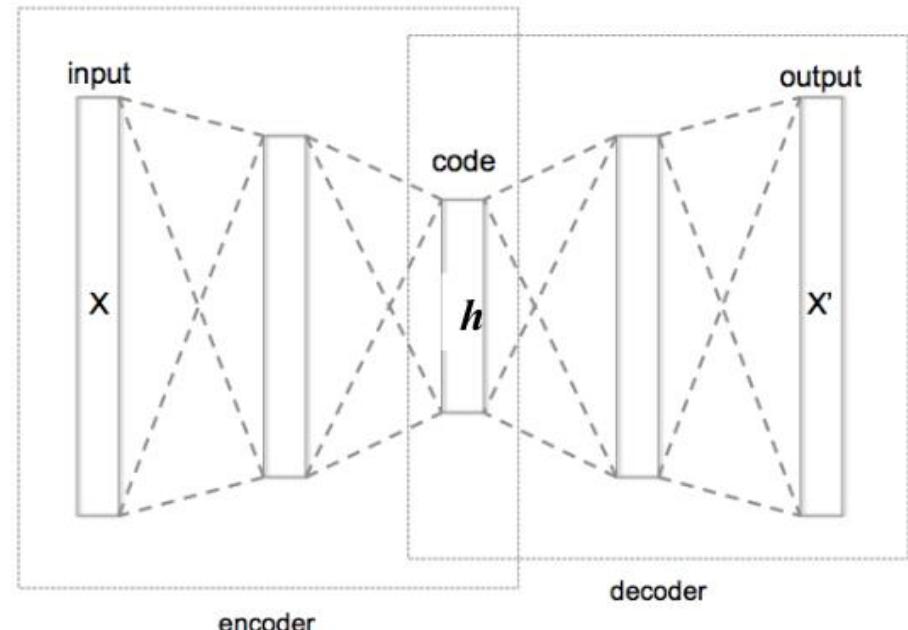
- One hidden layer

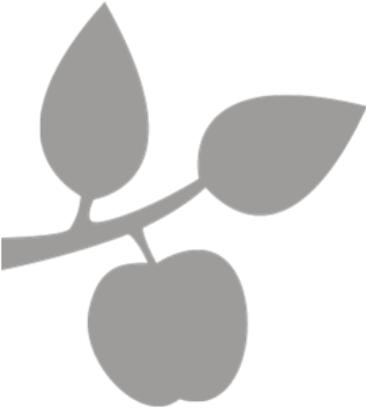
- Non-linear encoder
- Takes input $x \in \mathbb{R}^d$
- Maps into output $h \in \mathbb{R}^p$ (σ being any activation function)

$$h = \sigma_1(Wx + b) \quad x' = \sigma_2(W'h + b')$$

- Minimize reconstructions error

$$L(x, x') = \|x - x'\|^2 = \|x - \sigma_2(W'(\sigma_1(Wx + b)) + b')\|^2$$





Autoencoders

- Introduction
- Undercomplete Autoencoder
- Regularized Autoencoder

Undercomplete Autoencoder

- Copying input to output sounds useless
- But we are not interested in the output of the decoder
- We hope that training the autoencoder to perform copying task will result in h taking on useful properties
- To obtain useful features, constrain h to have lower dimension than x
 - Such an autoencoder is called undercomplete
 - Learning the undercomplete representation forces the autoencoder to capture most salient features of training data

Connection to PCA

- Learning process is that of minimizing a loss function

$$L(x, g(f(x)))$$

- When the decoder g is linear and L is the mean squared error, an undercomplete autoencoder learns to span the same subspace as PCA
- Autoencoders with nonlinear f and g can learn more powerful nonlinear generalizations of PCA
 - **BUT: Overcapacity is not always useful**

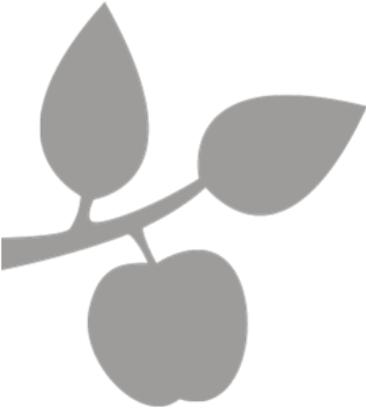
Encoder/Decoder Capacity

- If encoder f and decoder g are allowed too much capacity
 - autoencoder can learn to perform the copying task without learning any useful information about distribution of data
- Autoencoder with a one-dimensional code and a very powerful nonlinear encoder can learn to map $x^{(i)}$ to code i .
- The decoder can learn to map these integer indices back to the values of specific training examples
- Autoencoder trained for copying task fails to learn anything useful if f/g capacity is too great

Cases when Autoencoder Learning Fails

Where autoencoders fail to learn anything useful:

1. Capacity of encoder/decoder f/g is too high
 - Capacity controlled by depth
2. Hidden code h has dimension equal to input x
3. Overcomplete case: where hidden code h has dimension greater than input x
 - Even a linear encoder/decoder can learn to copy input to output without learning anything useful about data distribution



Autoencoders

- **Introduction**
- **Undercomplete Autoencoder**
- **Regularized Autoencoder**

Use Regularization

- Ideally, choose code size (dimension of h) small and capacity of encoder f and decoder g based on complexity of distribution modeled
- Regularized autoencoders provide the ability to do so
 - Rather than limiting model capacity by keeping encoder/decoder shallow and code size small
 - They use a loss function that encourages the model to have properties other than copy its input to output

Regularized Autoencoder Properties

- Regularized AEs have properties beyond copying input to output:
 - Sparsity of representation
 - Robustness to noise
 - Robustness to missing inputs
 - Smallness of the derivative of the representation
- Regularized autoencoder can be nonlinear and overcomplete
 - But still learn something useful about the data distribution even if model capacity is great enough to learn trivial identity function

Sparse Autoencoders

- A sparse autoencoder is simply an autoencoder whose training criterion involves a sparsity penalty $\Omega(h)$ on the code layer \mathbf{h} , in addition to the reconstruction error:

$$L(x, g(f(x))) + \Omega(h)$$

- They are often used to learn good features for other tasks, like classification
- The hidden layers then can be interpreted as latent variables of a generative model
 - An autoencoder that has been trained to be sparse must respond to unique statistical features of the dataset rather than simply perform copying

Denoising Autoencoders (DAE)

- Traditional autoencoders minimize

$$L(x, g(f(x)))$$

- A DAE minimizes

$$L(x, g(f(\tilde{x})))$$

- where \tilde{x} is a copy of x that has been corrupted by some form of noise
- The autoencoder must undo this corruption rather than simply copying the input
- Denoising training forces f and g to implicitly learn the structure of $p_{\text{data}}(x)$

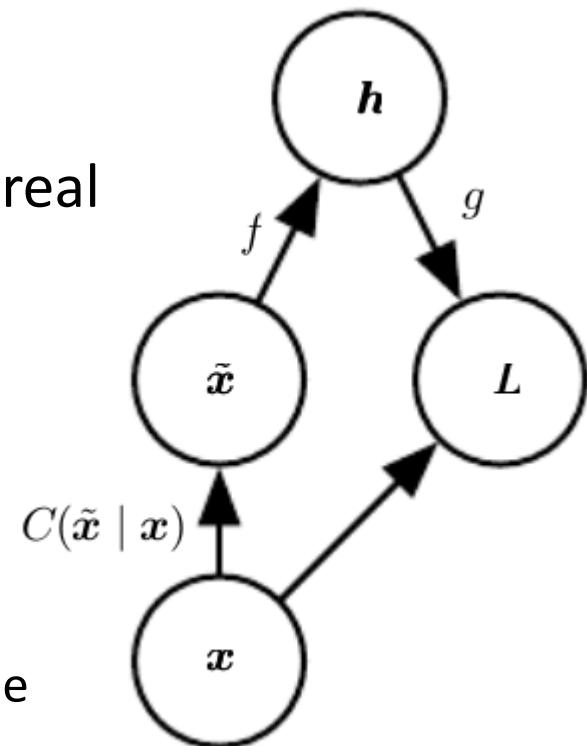
Denoising Autoencoders (DAE)

- Let's have a closer look to denoising autoencoders
- We train the encoder with data and a corruption process

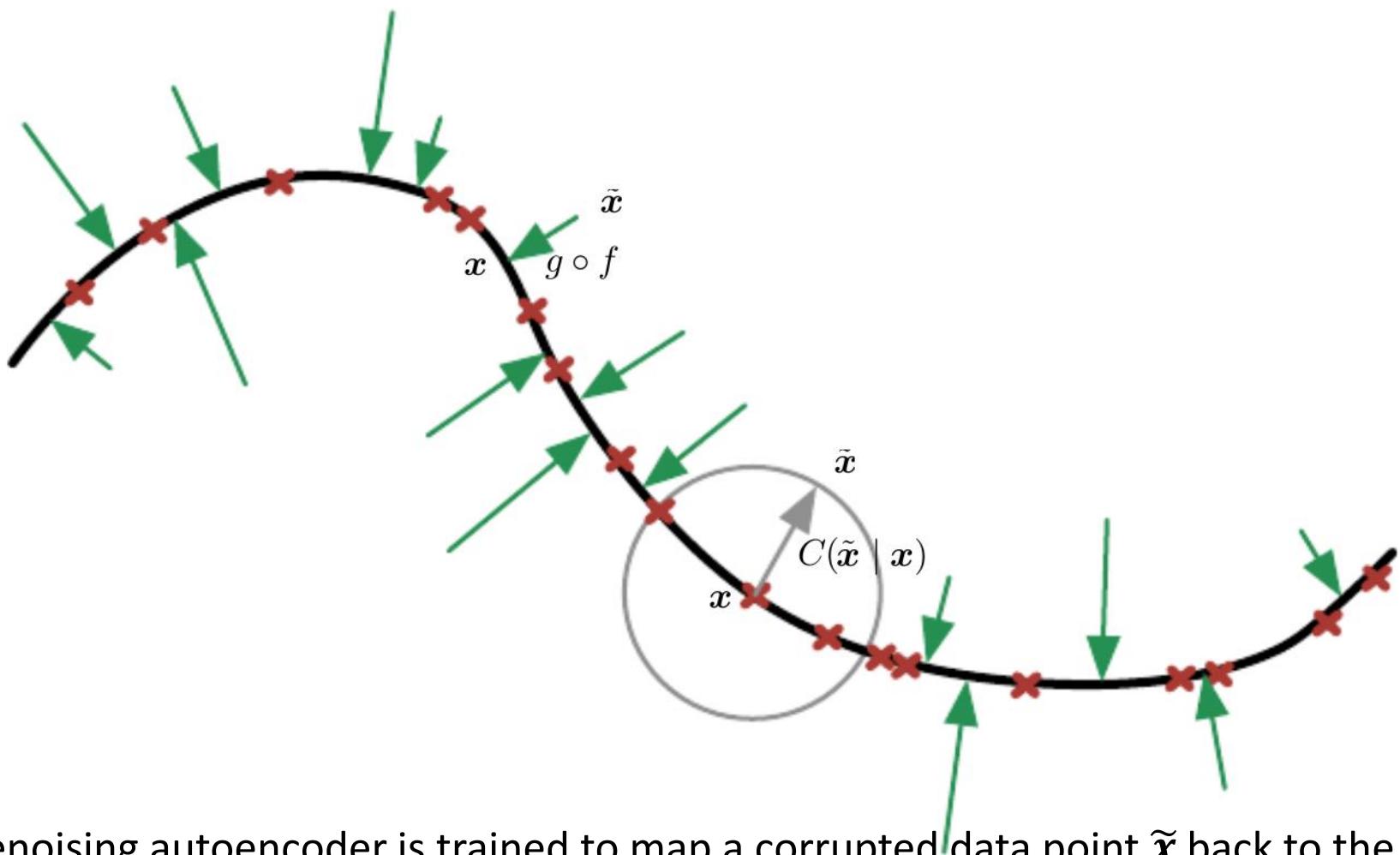
$$C(\tilde{x}|x)$$

which is a conditional distribution given the real input x

- Process:
 - Sample training example x
 - Create corrupted version \tilde{x}
 - Use (x, \tilde{x}) as training example for estimating the autoencoder reconstruction distribution



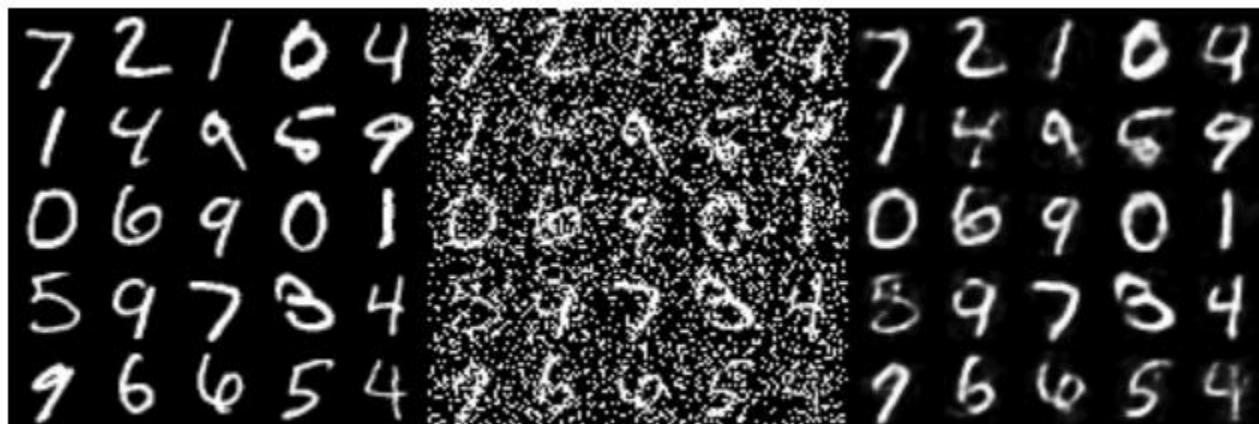
Denoising Autoencoders (DAE)



A denoising autoencoder is trained to map a corrupted data point \tilde{x} back to the original data point x .

Example

- An autoencoder with high capacity can end up learning an identity function (also called null function) where $\text{input}=\text{output}$
- A DAE can solve this problem by corrupting the data input
- Corrupt input nodes by setting 30-50% of random input nodes to zero



Original input, corrupted data and reconstructed data. Copyright by opendeep.org.

➤ <https://towardsdatascience.com/denoising-autoencoders-explained-dbb82467fc2>

Regularizing by Penalizing Derivatives

- Another strategy for regularizing an autoencoder is to use a penalty Ω as in sparse but with a different form of Ω :

$$\Omega(\mathbf{h}, \mathbf{x}) = \lambda \sum_i \|\nabla_{\mathbf{x}} h_i\|^2$$

- Note, that we are penalizing large derivatives with respect to the input \mathbf{x} : $\nabla_{\mathbf{x}}$
 - Forces the model to learn a function which is stable against slight changes in \mathbf{x}
- An autoencoder regularized in this way is called a **contractive autoencoder** or CAE.

Contractive Autoencoders

- Explicit regularizer on the code $\mathbf{h} = f(\mathbf{x})$, encouraging the derivatives of f to be as small as possible

$$\Omega(\mathbf{h}) = \lambda \left\| \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right\|_F^2$$

- As indicated, normally the squared Frobenius norm is used (sum of squared elements)
- The name contractive arises from the way that the CAE warps space:
 - Trained to resist perturbations of its input
 - Thus, maps a neighborhood of input points to a smaller neighborhood of output points

On Contraction

- To clarify, the CAE is contractive only locally
- All perturbations of a training point x are mapped near to $f(x)$
- Globally, two different points x and x' may be mapped to points that are farther apart than the original points
- But those points are outside of the manifold we are training

Practical Issues

- The CAE regularization criterion is cheap to compute in the case of a single hidden layer autoencoder
 - much more expensive in the case of deeper autoencoders
- Often used strategy:
 - separately train a series of single-layer autoencoders
 - Each autoencoder is trained to reconstruct the previous autoencoder's hidden layer
 - The composition of these autoencoders then forms a deep autoencoder
 - Because each layer was separately trained to be locally contractive, the deep autoencoder is contractive as well
 - The result is not the same as what would be obtained by jointly training the entire architecture but it captures many of the desirable qualitative characteristics.



DM873

Deep Learning

Spring 2019

Lecture 13 – Deep Generative Models

Sources

- Many of the slides are taken from a fantastic course of Stefano Ermon, Aditya Grover:
<https://deepgenerativemodels.github.io/>
- Also, from the same people:
Tutorial on Deep Generative Models
- Generative Deep Networks from Jakob Verbeek, INRIA, Grenoble, France



Deep Generative Models

- **Introduction**
- **Deep Generative Models**
- **Variational Autoencoder**
- **GAN**

Generative Models

Challenge: understand complex, unstructured inputs



Computer Vision



Computational Speech



Natural Language Processing



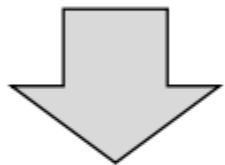
Robotics

Generative Models

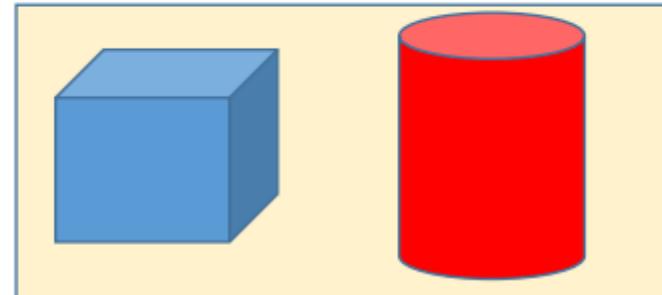
High level
description

Cube(color=blue, position=, size=, ...)
Cylinder(color=red, position=, size=,..)

Generation (graphics)



Inference (vision)

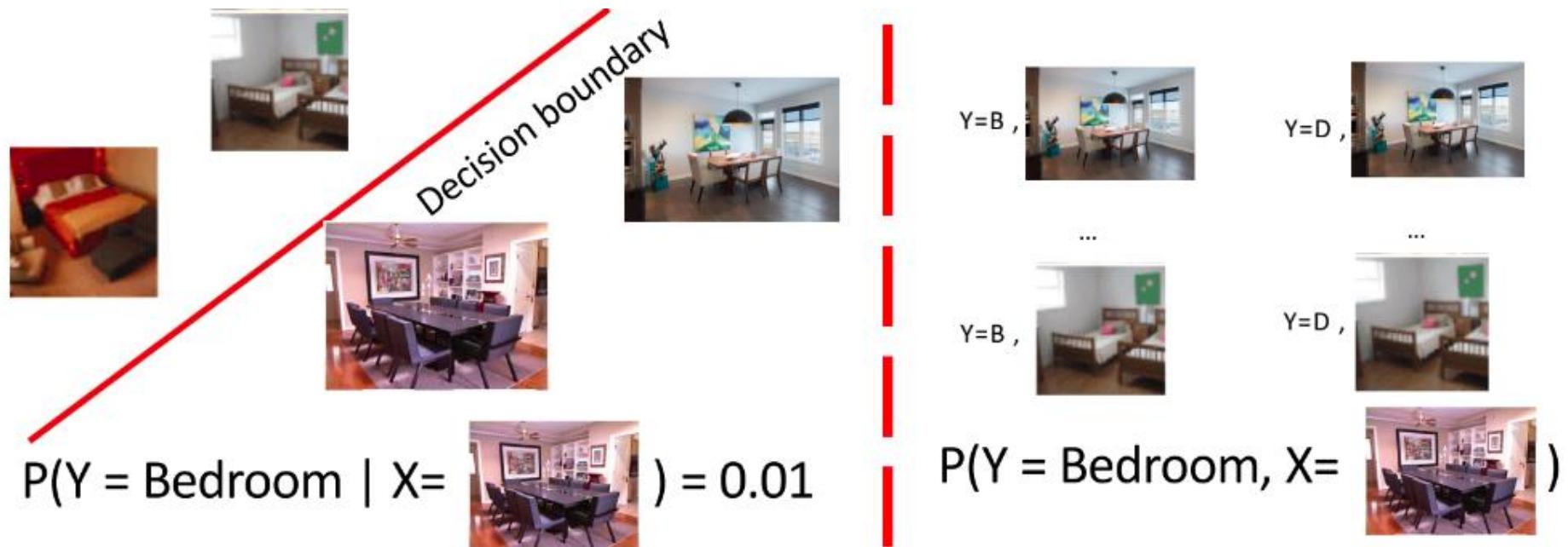


Raw sensory
outputs

Generative vs. Discriminative Models

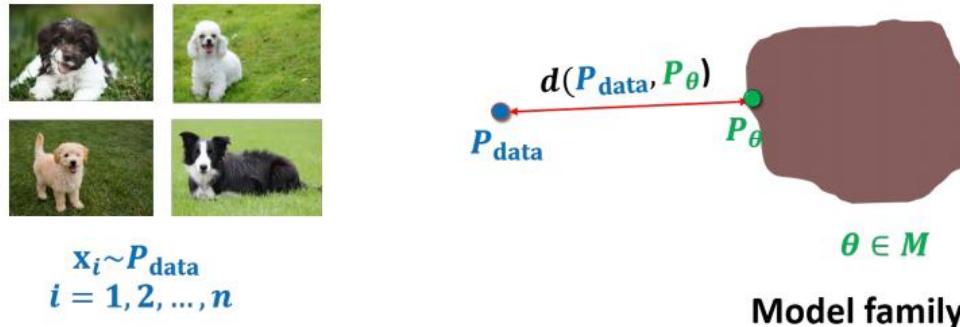
- Generative models are often contrasted against discriminative models.
- Consider a supervised learning task with features X and labels Y :
 - **Generative models want to learn $P(X, Y)$.**
 - **Discriminative models want to learn $P(Y|X)$.**
- Philosophically, it's hard to justify learning $P(X, Y)$ if you just want $P(Y|X)$.
 - "... one should solve the [classification] problem directly and never solve a more general problem as an intermediate step ..." Vapnik (1998)

Generative vs. Discriminative Models



Generative Models

- We are given a training set of examples, e.g., images of dogs



- Learn a probability distribution $p(x)$ over images x such that
 - Generation:** If we sample $x_{new} \sim p(x)$, x_{new} should look like a dog (sampling)
 - Density estimation:** $p(x)$ should be high if x looks like a dog, and low otherwise (anomaly detection)
 - Unsupervised representation learning:** We should be able to learn what these images have in common, e.g., ears, tail, etc. (features)
- First question: how to represent $p(x)$

Representing $p(x)$ - Compactness

- Suppose x_1, x_2, x_3 are binary variables. $P(x_1, x_2, x_3)$ can be specified with $(2^3 - 1) = 7$ parameters
- Main challenge: distributions over **high dimensional objects**

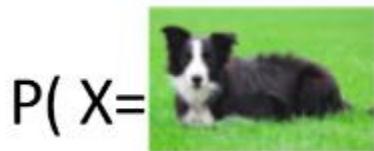


)

...

 $P(X =$ 

)

 $P(X = \text{[image of a gray textured surface]})$ 

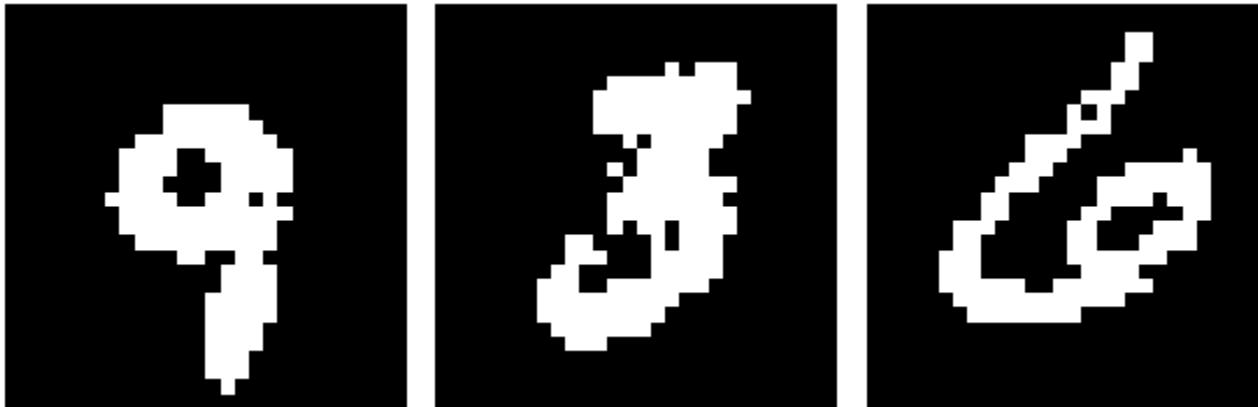
)

 $P(X =$ 

)

Too many possibilities!
N black/white pixels \rightarrow
 2^N parameters

Representing $p(x)$ - Compactness



- Suppose X_1, \dots, X_n are binary (Bernoulli) random variables, i.e., $X_i = \{0, 1\} = \{\text{Black, White}\}$.
- How many possible states?
$$\underbrace{2 \cdot 2 \cdot \dots \cdot 2}_{n \text{ times}} = 2^n$$
- Sampling of the pixel would require a model with $2^n - 1$ parameters

Representing $p(x)$ - Factorization

- Definition of conditional probability

$$P(x_1, x_2) = P(x_1) P(x_2 | x_1)$$

- Chain rule:

$$P(x_1, x_2, x_3) = P(x_1) P(x_2 | x_1) P(x_3 | x_1, x_2)$$

- Chain rule:

$$P(x_1, x_2, x_3, x_4) = P(x_1) P(x_2 | x_1) P(x_3 | x_1, x_2) P(x_4 | x_1, x_2, x_3)$$

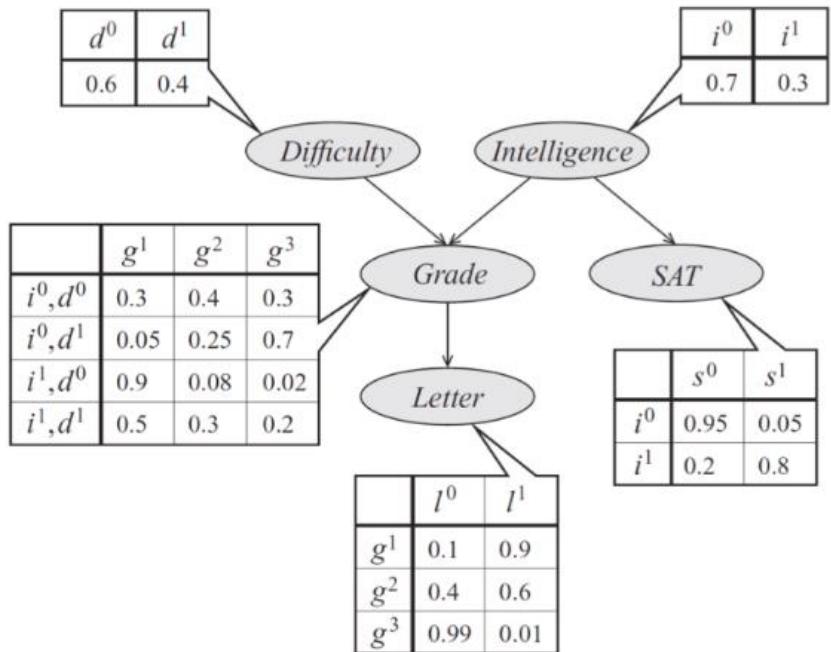
Simple!

Still complex!
Needs a distribution for all
possible values of x_1, x_2, x_3

Representing $p(x)$ - Bayes Nets

- Solution #1: Simplify the conditionals:

$$\begin{aligned} P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2|x_1)P(x_3|x_1, x_2)P(x_4|x_1, x_2, x_3) = \\ &= P(x_1)P(x_2|x_1)P(x_3|x_2)P(x_4|x_3) \end{aligned}$$



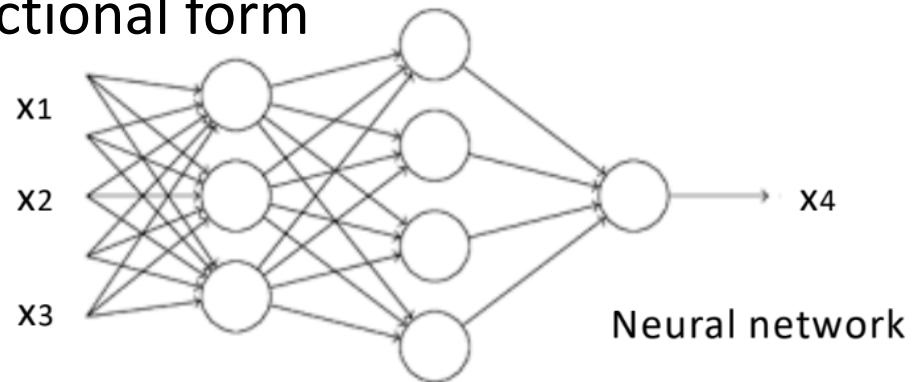
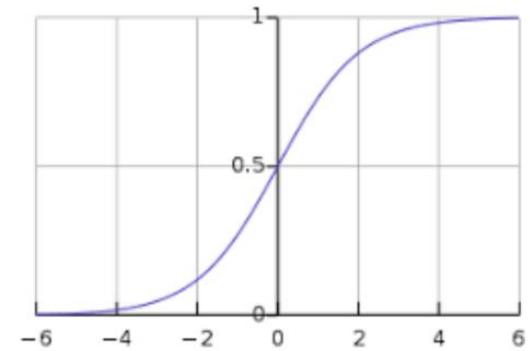


Deep Generative Models

- Introduction
- Deep Generative Models
- Variational Autoencoder
- GAN

Representing $p(x)$ – Deep Generative Models

- Solution #2a: use simple functional form for the conditionals
 - $P(x_4|x_1, x_2, x_3) \approx \sigma(a \cdot x_1 + b \cdot x_2 + c \cdot x_3)$
 - Only requires storing 3 parameters
 - Relationship between x_4 and (x_1, x_2, x_3) could be too simple
- Solution #2b: more complex functional form
 - More flexible
 - More parameters

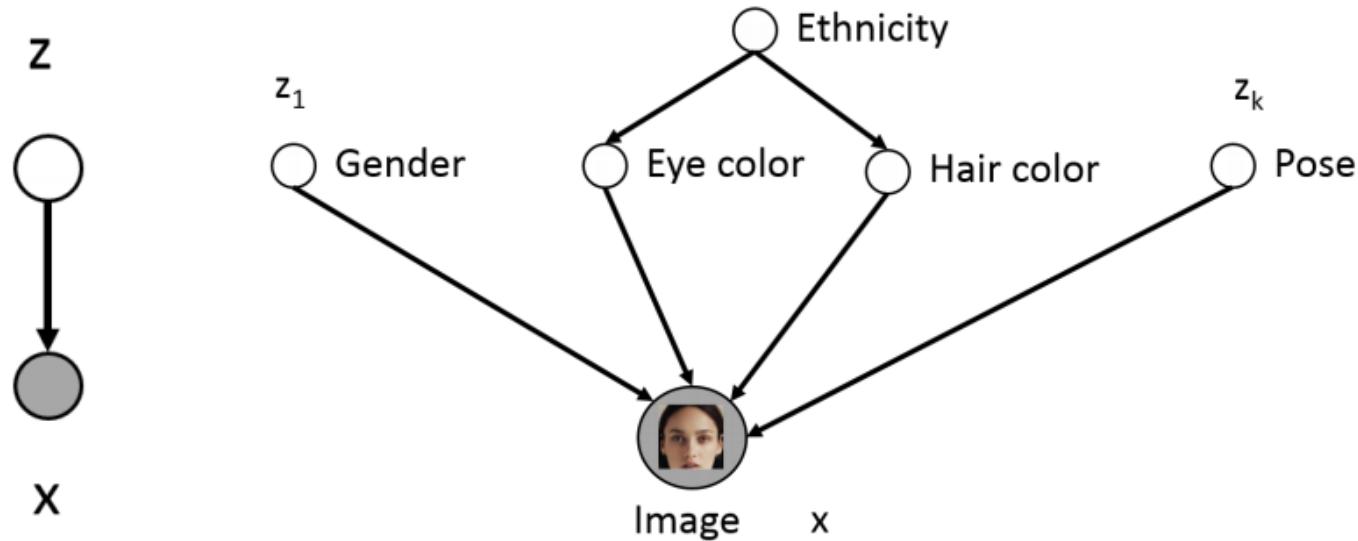


Latent variable models



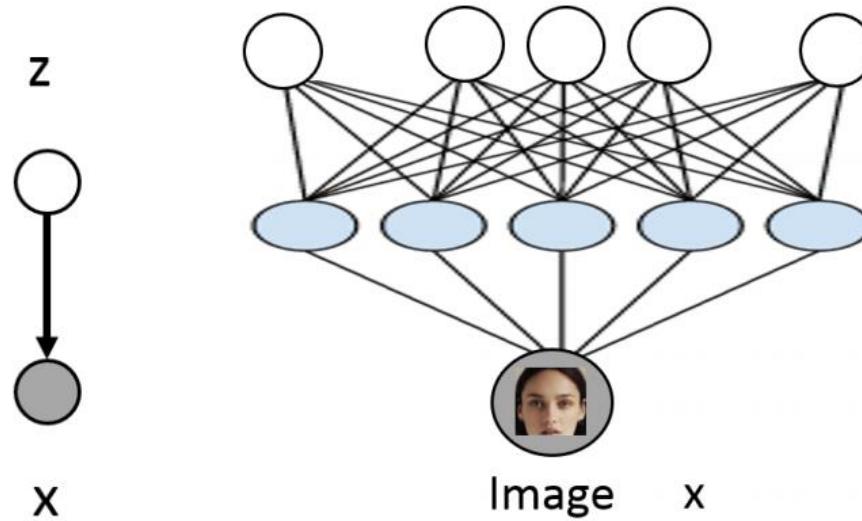
- Lots of variability in images x due to gender, eye color, hair color, pose, etc.
- However, unless images are annotated, these factors of variation are not explicitly available (latent).
- **Idea:** explicitly model these factors using latent variables z

Latent variable models



- Only shaded variables x are observed in the data (pixel values)
- Latent variables z correspond to high level features
 - If z chosen properly, $p(x|z)$ could be much simpler than $p(x)$
 - If we had trained this model, then we could identify features via $p(z|x)$, e.g., $p(\text{EyeColor} = \text{Blue}|x)$
- Challenge: Very difficult to specify these conditionals by hand

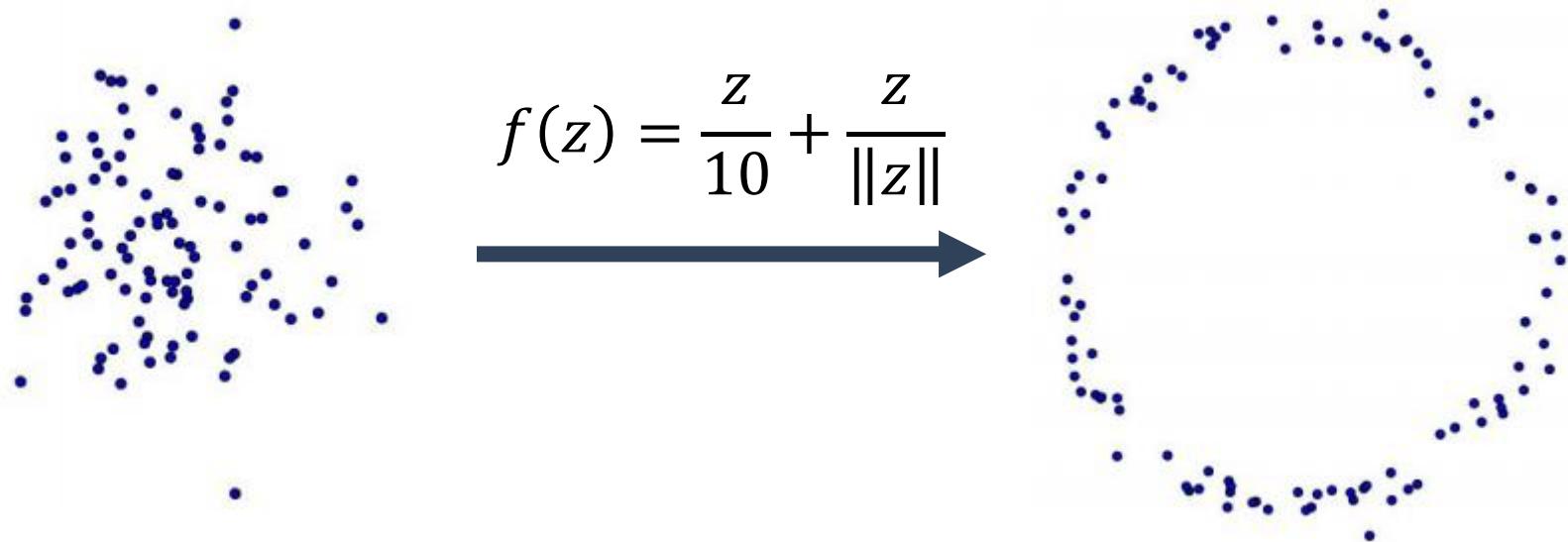
Deep Latent Variable Models



- $z \sim N(0, I)$
- $p(x|z) = N(\mu_\theta(z), \Sigma_\theta(z))$ where $\mu_\theta, \Sigma_\theta$ are neural networks
- Hope that after training, z will correspond to meaningful latent factors of variation (features)

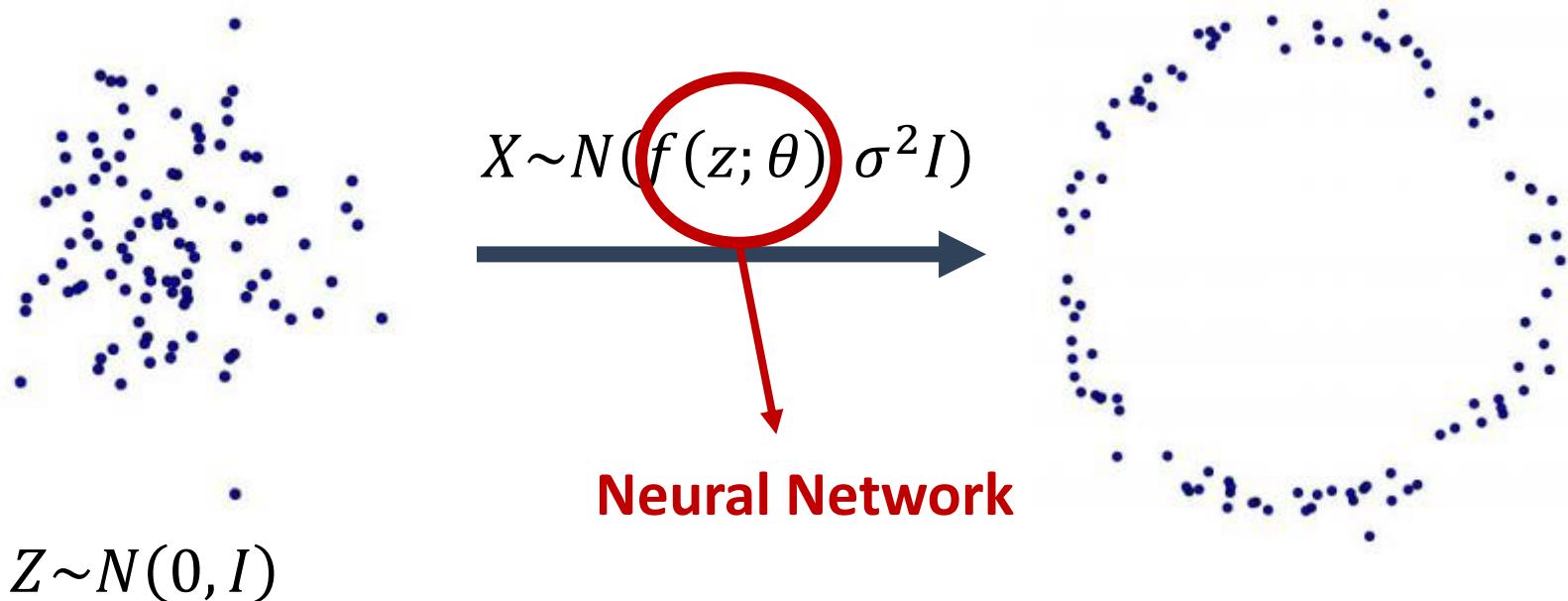
Rationale

- Intuition: given a bunch of random variables that can be sampled easily, we can generate some new random samples through a complicated non-linear mapping $x = f(z)$



Rationale

- Intuition: given a bunch of random variables that can be sampled easily, we can generate some new random samples through a complicated non-linear mapping $x = f(z)$





Deep Generative Models

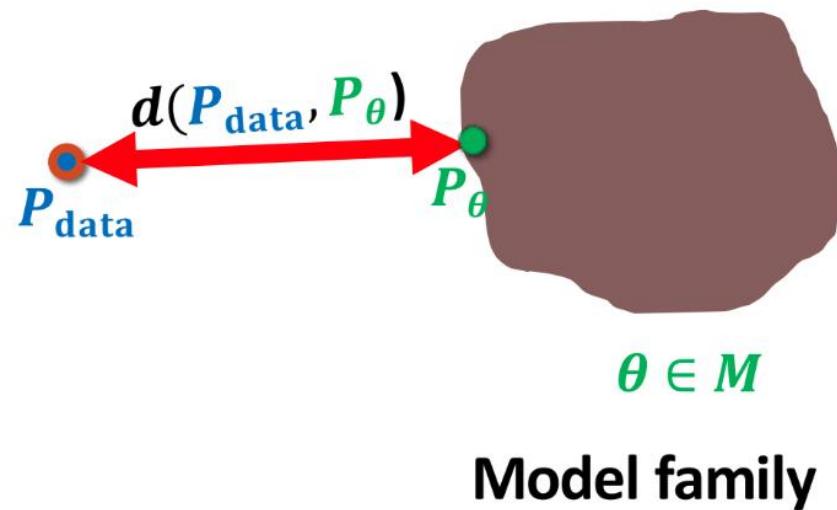
- Introduction
- Deep Generative Models
- Variational Autoencoder
- GAN

Learning Generative Models

- **Given:** Samples from a data distribution and model family
- **Goal:** Approximate a data distribution as closely as possible



$$\mathbf{x}_i \sim P_{\text{data}} \\ i = 1, 2, \dots, n$$



Learning Generative Models

- **Given:** Samples from a data distribution
- **Goal:** Approximate a data distribution as closely as possible

$$\min_{\theta \in M} d(P_{\text{data}}, P_{\theta})$$

- **Challenge:** How should we evaluate and optimize closeness between the **data distribution** and the **model distribution?**

Maximum Likelihood Estimation

- **Given:** Samples from a data distribution
- **Goal:** Approximate a data distribution as closely as possible
- Solution 1: KL divergence

$$\min_{\theta} \mathbb{E}_{x \sim P_{\text{data}}} [-\log p_{\theta}(x)]$$

- Statistically efficient
- Requires the ability to tractably evaluate or optimize likelihoods

Marginalization

- Remember, minimizing the KL divergence is the same as maximizing the log-likelihood

$$\max_{\theta} \sum_i \log p_{\theta}(x_i)$$

- With

$$p_{\theta}(x) = \int_{\mathbf{z}} p_{\theta}(x|\mathbf{z}) p_{\theta}(\mathbf{z}) d\mathbf{z}$$

Marginalization

- Remember, minimizing the KL divergence is the same as maximizing the log-likelihood

$$\max_{\theta} \sum_i \log p_{\theta}(x_i)$$

- With

$$p_{\theta}(x) = \int_{\mathbf{z}} p_{\theta}(x|\mathbf{z}) p_{\theta}(\mathbf{z}) d\mathbf{z}$$

Training Images should have high probability

Would require integrating over neural net => Difficult and Intractable

Marginalization

- Side note: Approximating the distribution by sampling also difficult:

$$\log p_\theta(x) \approx \log \frac{1}{N} \sum_j p_\theta(x|z_j)$$

- Most sampled z will have a close-to-zero $p_\theta(x|z_j)$!
- We need something more clever!

Maximize Variational Lower-Bound

- We try to maximize the so-called variational lower-bound:

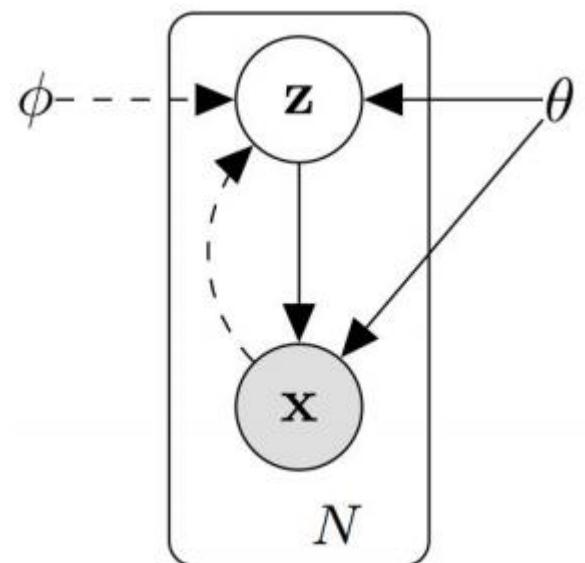
$$\log p_{\theta}(x_i) \geq \mathbb{E}_{q(\mathbf{z})}[\log p_{\theta}(x_i|\mathbf{z})] - KL[q(\mathbf{z})||p_{\theta}(\mathbf{z})]$$

- That means with maximizing the distance of a proposal distribution of \mathbf{z} and the KL distance of $q(\mathbf{z})$ and $p_{\theta}(\mathbf{z})$
 - Easy to sample
 - Differentiable wrt parameters
 - Given a training sample X , the sampled \mathbf{z} is likely to have a non-zero $p(x|\mathbf{z})$
- How to find such a distribution?

Maximize Variational Lower-Bound

- **Answer:** Another neural network + Gaussian to approximate the posterior

$$\log p_{\theta}(x_i) \geq \mathbb{E}_{q_{\Phi}(z)}[\log p_{\theta}(x_i|z)] - KL[q_{\Phi}(z)||p_{\theta}(z)]$$



Maximize Variational Lower-Bound

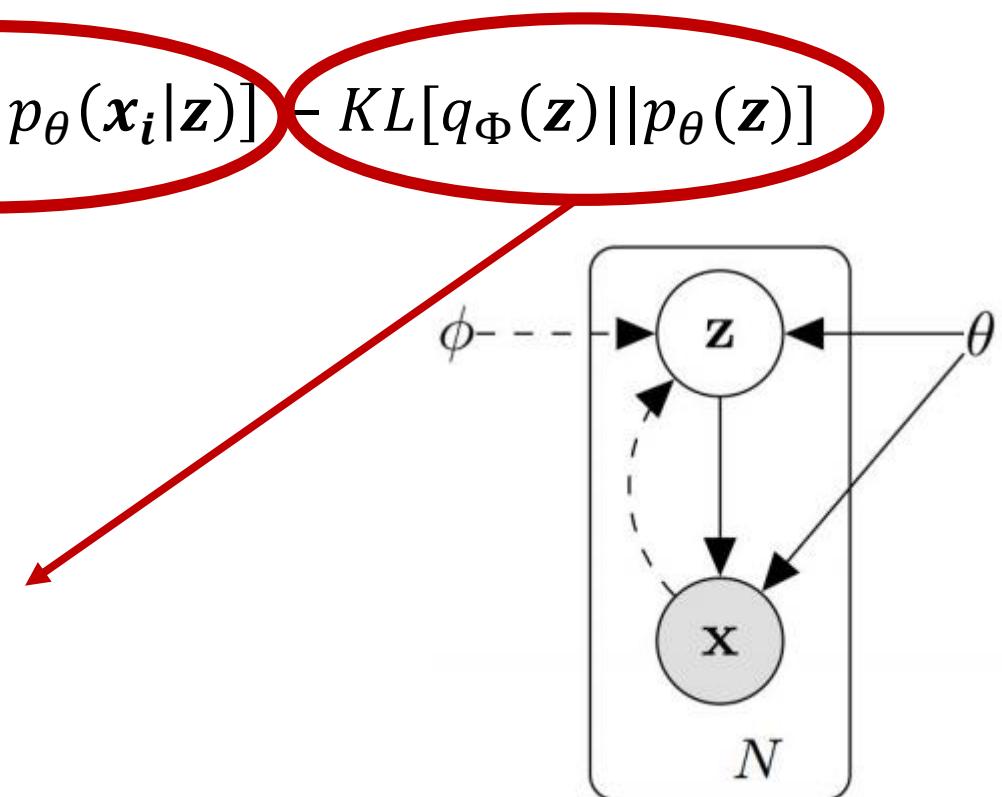
- Answer: Another neural network + Gaussian to approximate the posterior

$$\log p_{\theta}(x_i) \geq \mathbb{E}_{q_{\Phi}(z)}[\log p_{\theta}(x_i|z)] - KL[q_{\Phi}(z)||p_{\theta}(z)]$$

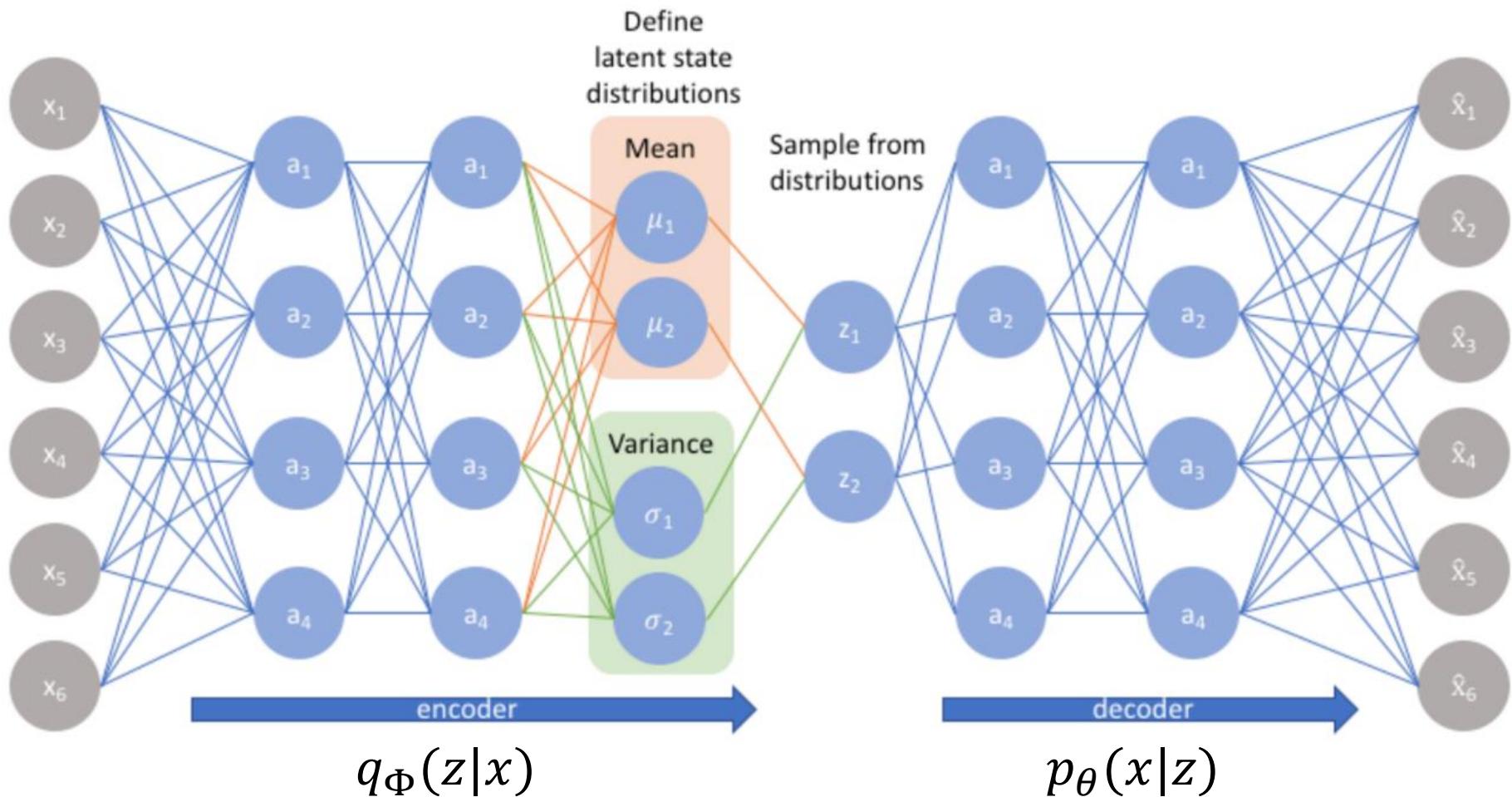
Reconstruction Error: Training samples should have high probability

Proposal Distribution should be like Gaussians:

- **closed-form and differentiable**



Variational Autoencoder





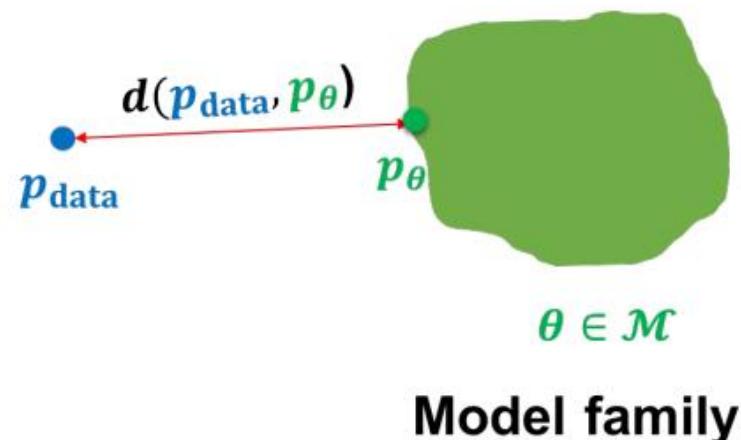
Deep Generative Models

- **Introduction**
- **Deep Generative Models**
- **Variational Autoencoder**
- **GAN**

Recap



$$\mathbf{x}^{(j)} \sim p_{\text{data}} \\ j = 1, 2, \dots, |\mathcal{D}|$$



- Variational Autoencoders:

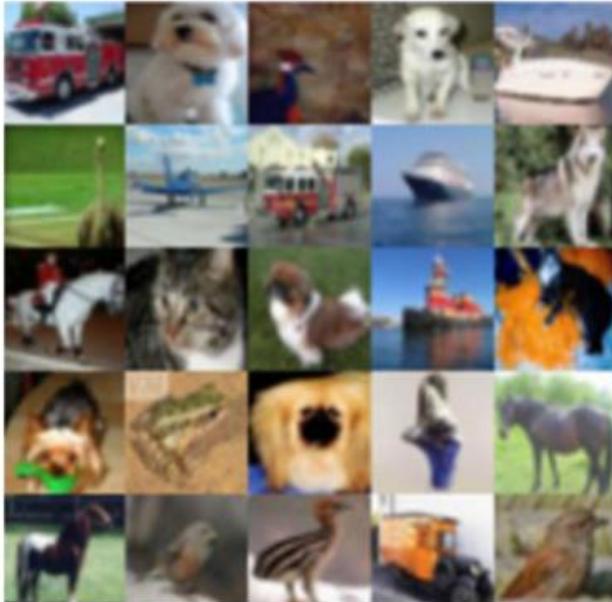
$$p_\theta(x) = \int_{\mathbf{z}} p_\theta(x|\mathbf{z}) p_\theta(\mathbf{z}) d\mathbf{z}$$

- Most model families are based on maximizing likelihoods
- Is the likelihood a good indicator of the quality of samples generated by the model?

Towards likelihood-free learning

- Case 1: Optimal generative model will give best sample quality and highest test log-likelihood
 - For imperfect models, achieving high log-likelihoods might not always imply good sample quality, and vice-versa
- Case 2: Great test log-likelihoods, poor samples.
 - When we have a poor signal to noise ratio
- Case 3: Great samples, poor test log-likelihoods.
 - E.g., Memorizing training set
 - Samples look exactly like the training set (cannot do better!)
 - Test set will have zero probability assigned (cannot do worse!)

Comparing Distributions via Samples



vs.



$$S_1 = \{\mathbf{x} \sim P\}$$

$$S_2 = \{\mathbf{x} \sim Q\}$$

- Given a finite set of samples from two distributions $S_1 = \{x \sim P\}$ and $S_2 = \{x \sim Q\}$, how can we tell if these samples are from the same distribution? (i.e., $P = Q?$)

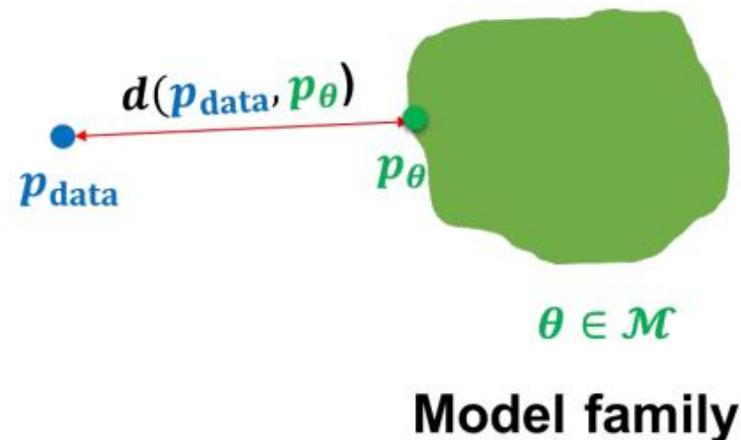
Two-sample Tests

- Given $S_1 = \{x \sim P\}$ and $S_2 = \{x \sim Q\}$, a two-sample test considers the following hypotheses
 - Null hypothesis $H_0: P = Q$
 - Alternate hypothesis $H_1: P \neq Q$
- Test statistic T compares S_1 and S_2 e.g., difference in means, variances of the two sets of samples
- If T is less than a threshold α , then accept H_0 else reject it
- **Key observation:** Test statistic is **likelihood-free** since it does not involve P or Q (only samples)

Generative Modeling and Two-Sample Tests

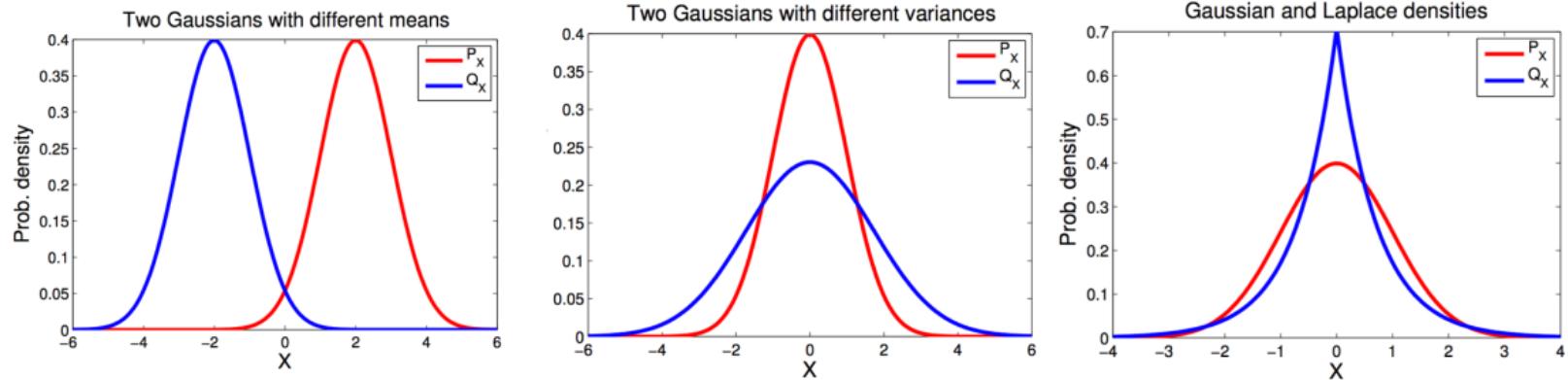


$$\begin{aligned} \mathbf{x}^{(j)} &\sim p_{\text{data}} \\ j &= 1, 2, \dots, |\mathcal{D}| \end{aligned}$$



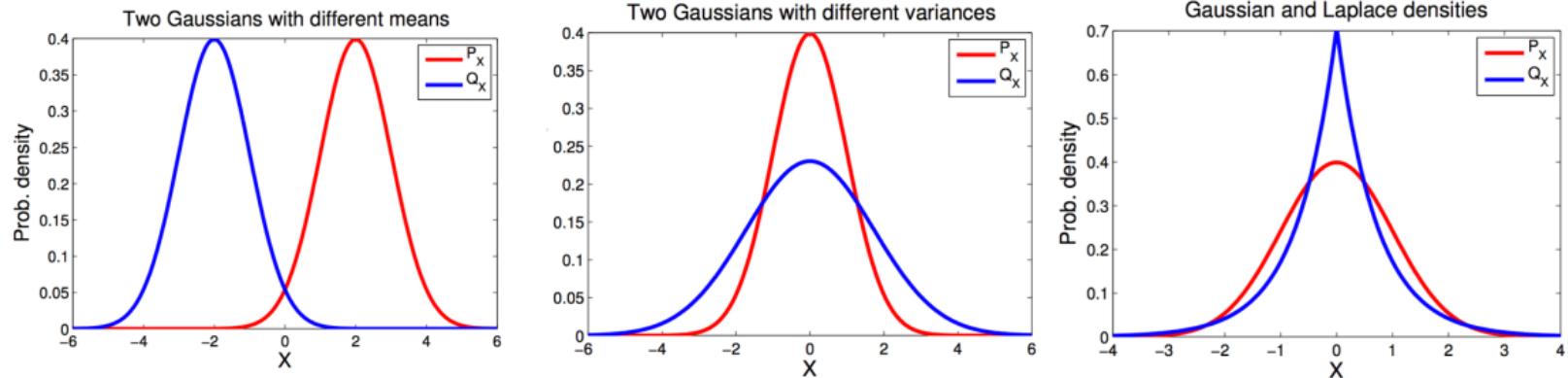
- Apriori we assume direct access to $S_1 = D = \{x \sim p_{\text{data}}\}$
- In addition, we have a model distribution p_{θ}
 - Assume that the model distribution permits efficient sampling (e.g., directed models). Let $S_2 = \{x \sim p_{\theta}\}$
- Alternate notion of distance between distributions: Train the generative model to minimize a two-sample test objective between S_1 and S_2

Two-Sample Test via a Discriminator



- Many different test statistics exist to distinguish distributions or samples statistics of them
- **But:** Finding a two-sample test objective in high dimensions is hard
- How should we find such a statistic for function we cannot fully assess?

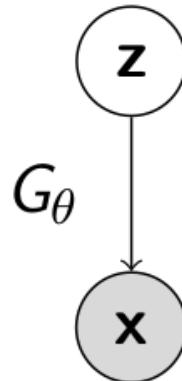
Two-Sample Test via a Discriminator



- Many different test statistics exist to distinguish distributions or samples
- Key idea: Learn a statistic that maximizes a suitable notion of distance between the two sets of samples S_1 and S_2**
- But how can we learn such a hard-to-assess?

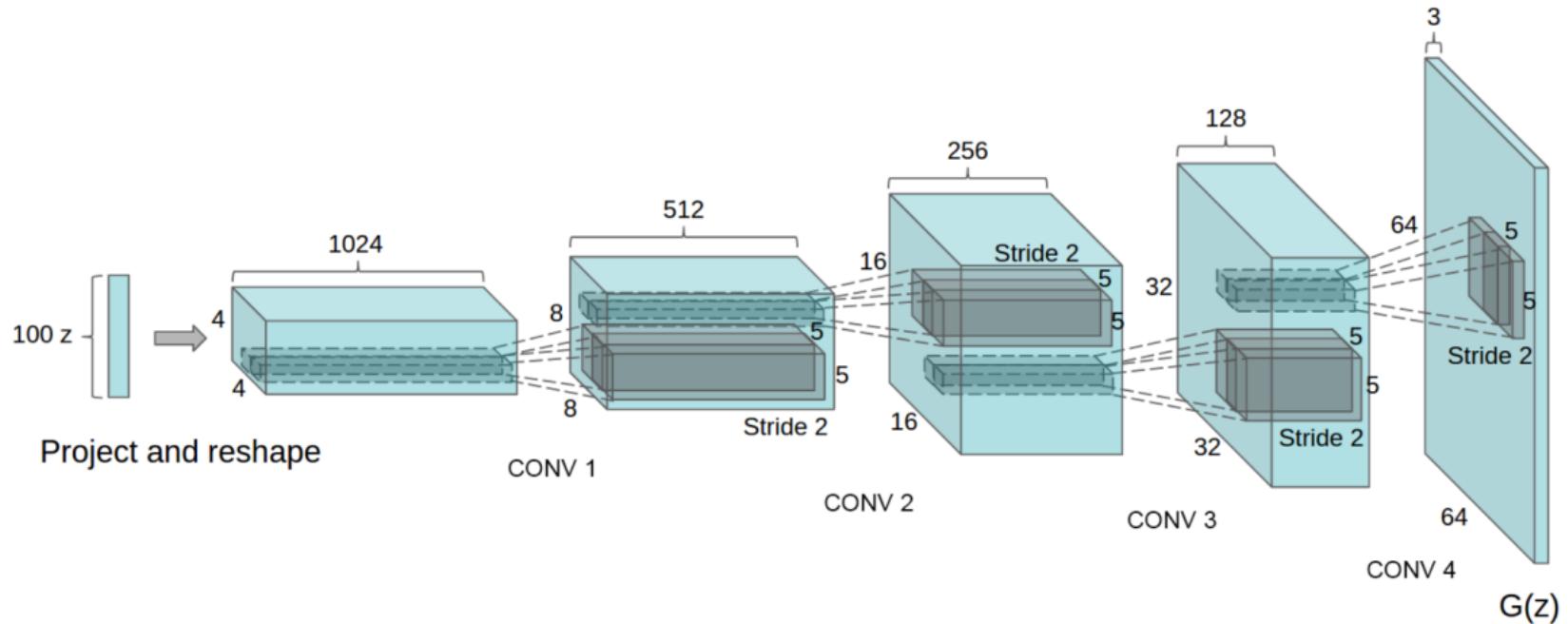
Generative Adversarial Networks

- A two player minimax game between a **generator** and a **discriminator**



- **Generator**
 - Directed, latent variable model with a deterministic mapping between z and x given by G_θ
 - Minimizes a two-sample test objective (in support of the null hypothesis $p_{\text{data}} = p_\theta$)

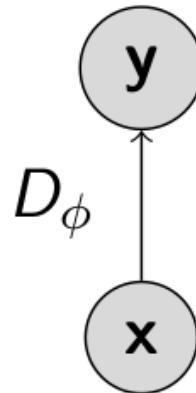
Typical Generator



- Unit Gaussian distribution on z , typically 10-100 dim.
- Up-convolutional deep network (reverse recognition CNN)

Generative Adversarial Networks

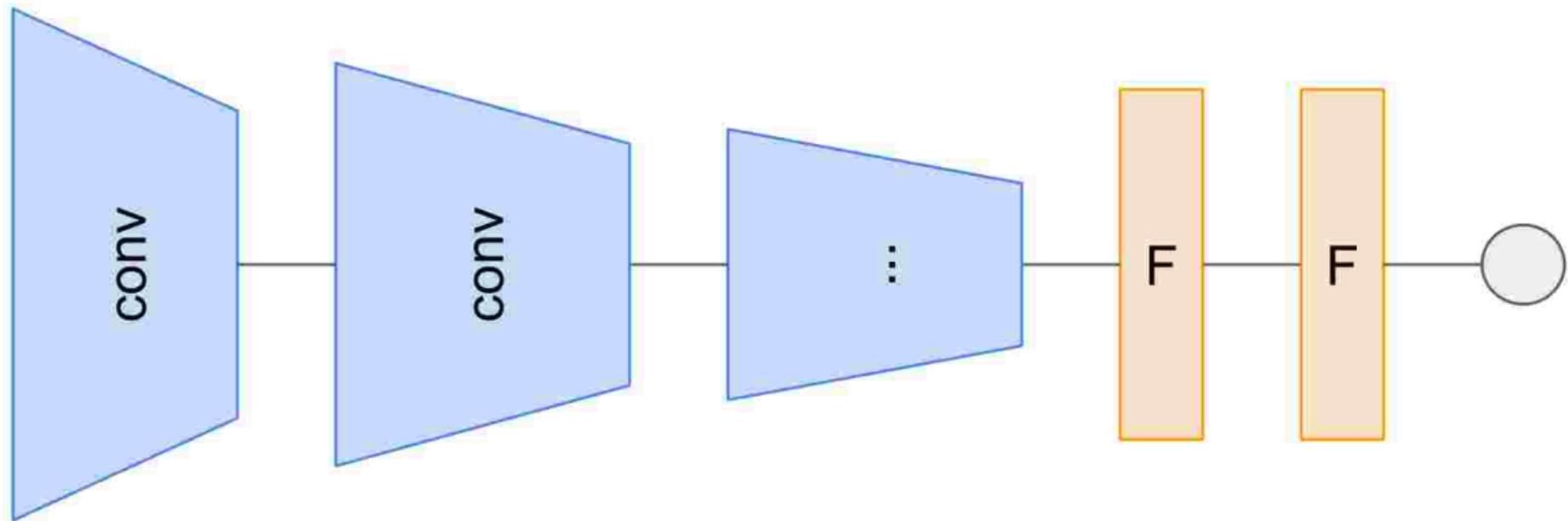
- A two player minimax game between a **generator** and a **discriminator**



- **Discriminator**

- Any function (e.g., neural network) which tries to distinguish “real” samples from the dataset and “fake” samples generated from the model
- Maximizes the two-sample test objective (in support of the alternate hypothesis $p_{\text{data}} \neq p_\theta$)

Typical Discriminator



Example of GAN objective

- **Training objective for discriminator:**

$$\max_D V(G, D) = \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{x \sim p_G} [\log(1 - D(x))]$$

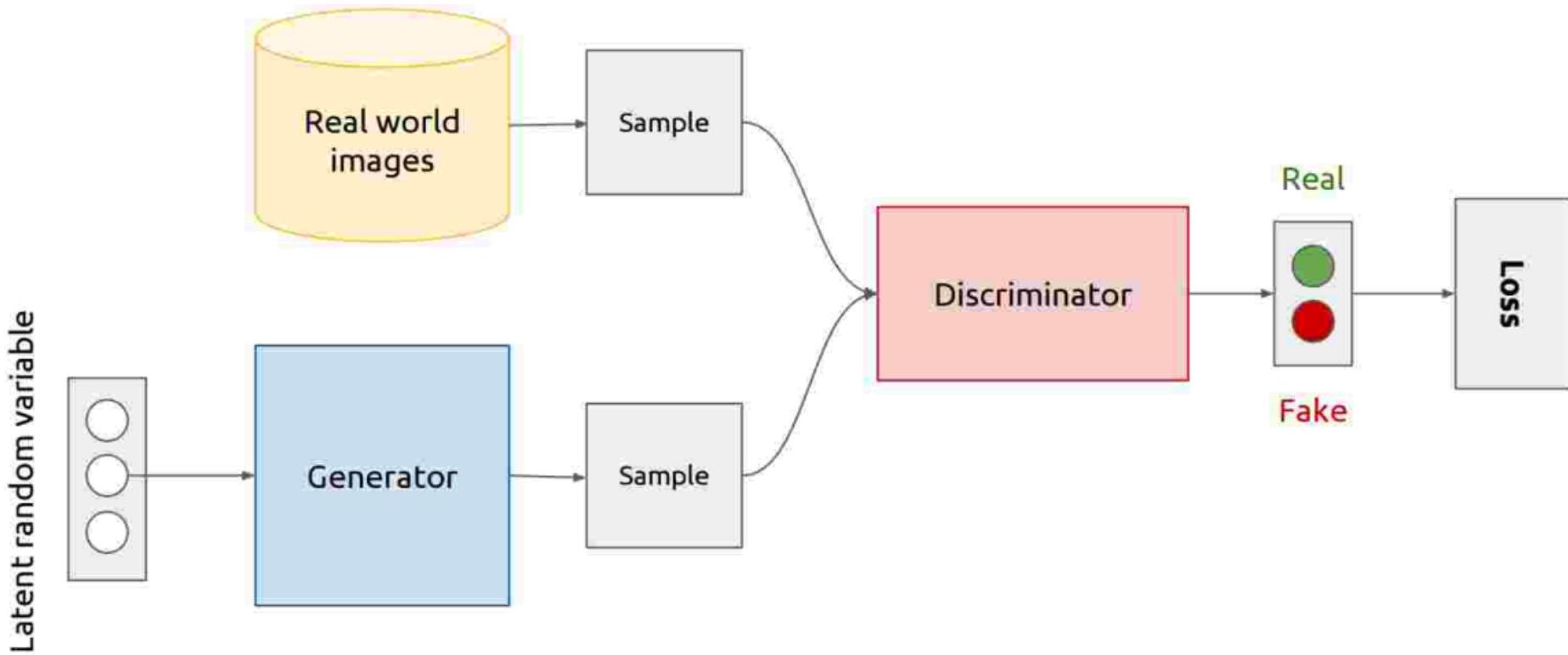
- For a fixed generator G , the discriminator is performing binary classification with the cross entropy objective

- Assign probability 1 to true data points $x \sim p_{\text{data}}$
 - Assign probability 0 to fake samples $x \sim p_G$

- **Training objective for generator:**

$$\min_G V(G, D) = \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{x \sim p_G} [\log(1 - D(x))]$$

Schematic Setup of Adversarial Training



The GAN training algorithm

- Sample minibatch of m training points $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(m)}$ from D
- Sample minibatch of m noise vectors $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \dots, \mathbf{z}^{(m)}$ from p_z
- Update the generator parameters θ by stochastic gradient **descent**

$$\nabla_{\theta} V(G_{\theta}, D_{\Phi}) = \frac{1}{m} \nabla_{\theta} \sum_{i=1}^m \log \left(1 - D_{\Phi} \left(G_{\theta}(\mathbf{z}^{(i)}) \right) \right)$$

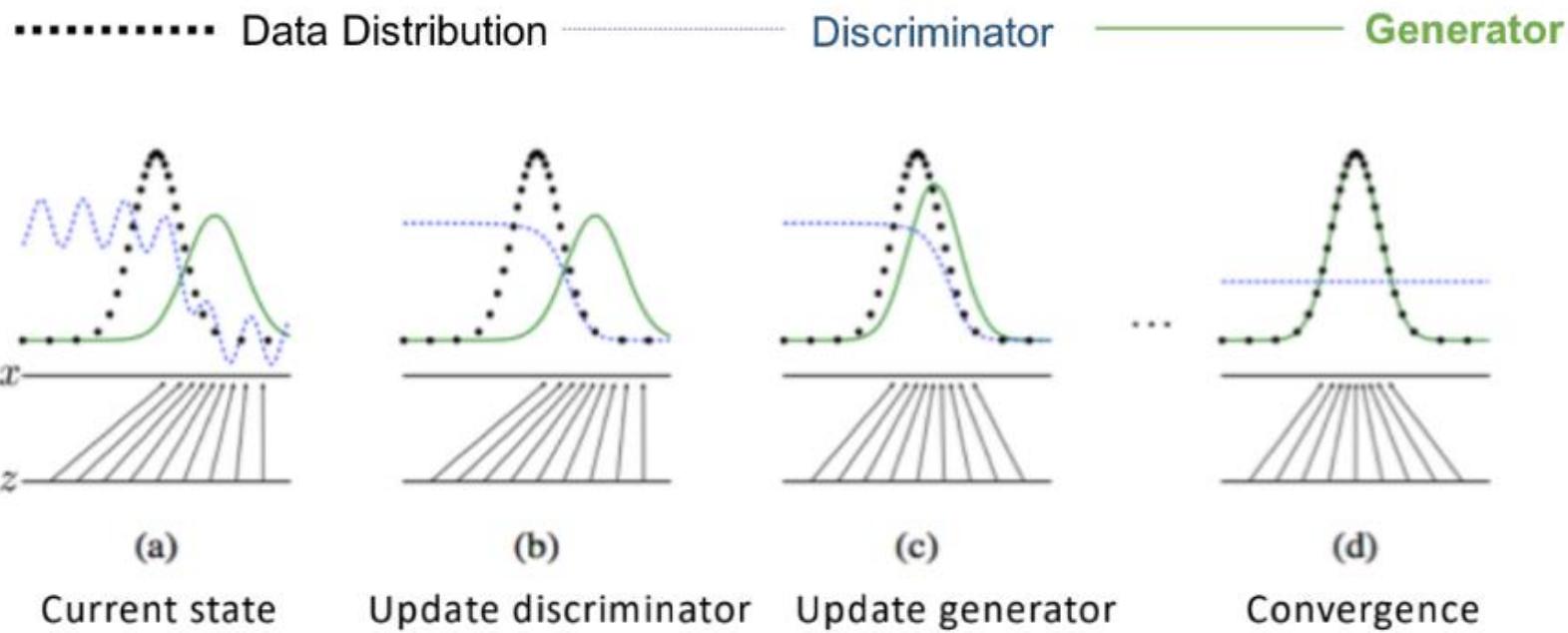
- Update the discriminator parameters Φ by stochastic gradient **ascent**

$$\nabla_{\Phi} V(G_{\theta}, D_{\Phi}) = \frac{1}{m} \nabla_{\Phi} \sum_{i=1}^m \left[\log D_{\Phi}(\mathbf{x}^{(i)}) + \log \left(1 - D_{\Phi} \left(G_{\theta}(\mathbf{z}^{(i)}) \right) \right) \right]$$

- Repeat for fixed number of epochs

Alternating optimization in GANs

$$\min_{\theta} \max_{\Phi} V(G_{\theta}, D_{\Phi}) = \mathbb{E}_{x \sim p_{\text{data}}} [\log D_{\Phi}(x)] + \mathbb{E}_{z \sim p(z)} [\log (1 - D_{\Phi}(G_{\theta}(z)))]$$



Goodfellow et al., 2014

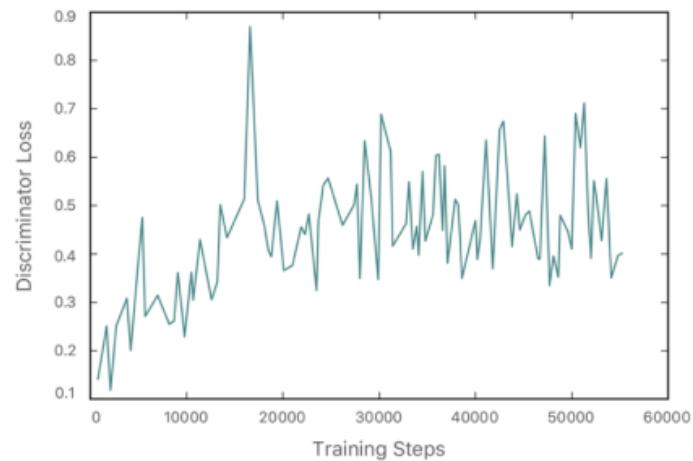
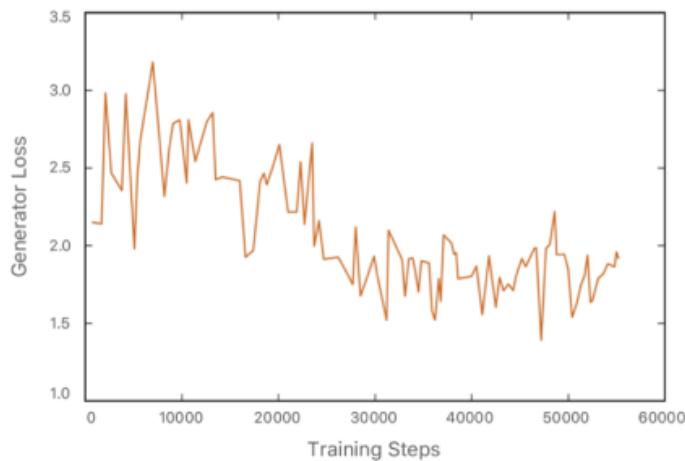
Both images are generated via GANs!



Source: Karras et al., 2018; The New York Times

Optimization challenges

- Theorem: If the generator updates are made in function space and discriminator is optimal at every step, then the generator is guaranteed to converge to the data distribution
- **Unrealistic assumptions!** In practice, the generator and discriminator loss keeps oscillating during GAN training



- No robust stopping criteria in practice!

Issues in Practice

- GANs are known to be very difficult to train in practice
- Formulated as mini-max objective between two networks
- Optimization can oscillate between solutions (Mode Collapse)
- Generator can collapse to represent part of the training data, and miss another part
- Hard to pick “compatible” architectures between generator and discriminator

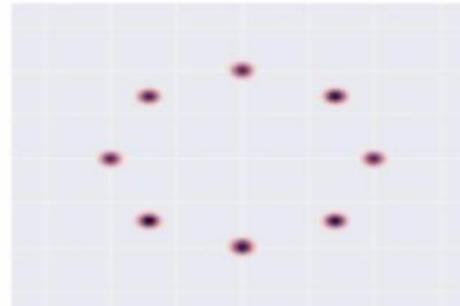
Mode Collapse

- GANs are notorious for suffering from mode collapse
- Intuitively, this refers to the phenomena where the generator of a GAN collapses to one or few samples (dubbed as “modes”)



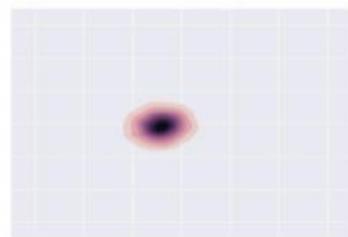
Arjovsky et al., 2017

Mode Collapse

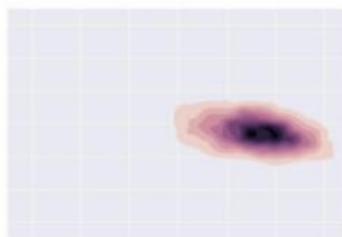


Target

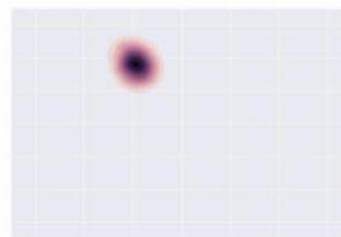
- True distribution is a mixture of Gaussians



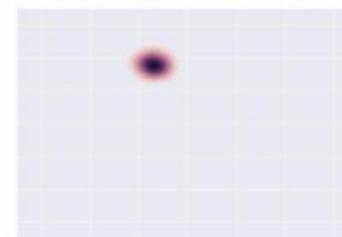
Step 0



Step 5k



Step 10k



Step 15k

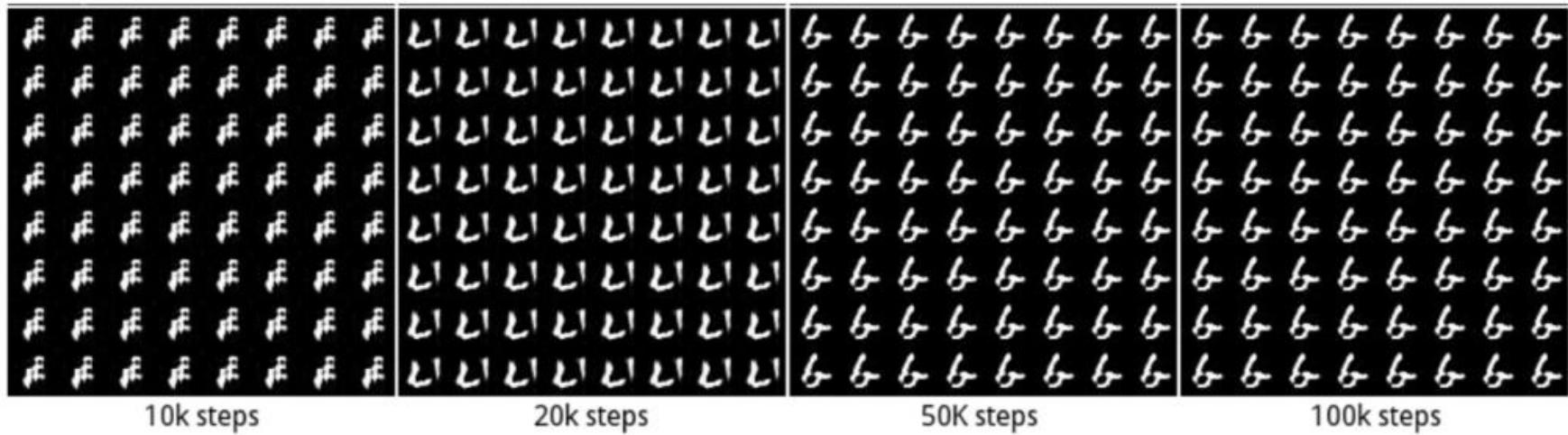


Step 20k

Source: Metz et al., 2017

- The generator distribution keeps oscillating between different modes

Mode Collapse



Source: Metz et al., 2017

- Fixes to mode collapse are mostly empirically driven: alternate architectures, adding regularization terms, injecting small noise perturbations etc.
- <https://github.com/soumith/ganhacks>
How to Train a GAN? Tips and tricks to make GANs work by Soumith Chintala

Beauty lies in the eyes of the discriminator

