

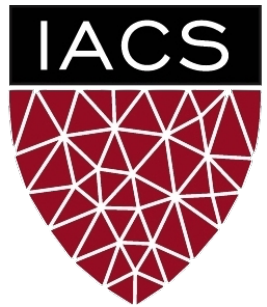


# Compute Fest 2018

## Introduction to Deep Learning Image Classification using Keras

Pavlos Protopapas and David Wihl

https://github.com/cs109/2018-ComputeFest/

# Outline

- Why Deep Neural Networks?
- Basic feed forward NN
- Notebook with examples

# AlphaZero (2017)

DeepMind

## AlphaZero AI beats champion chess program after teaching itself in four hours

Google's artificial intelligence sibling DeepMind repurposes Go-playing AI to conquer chess and shogi without aid of human knowledge

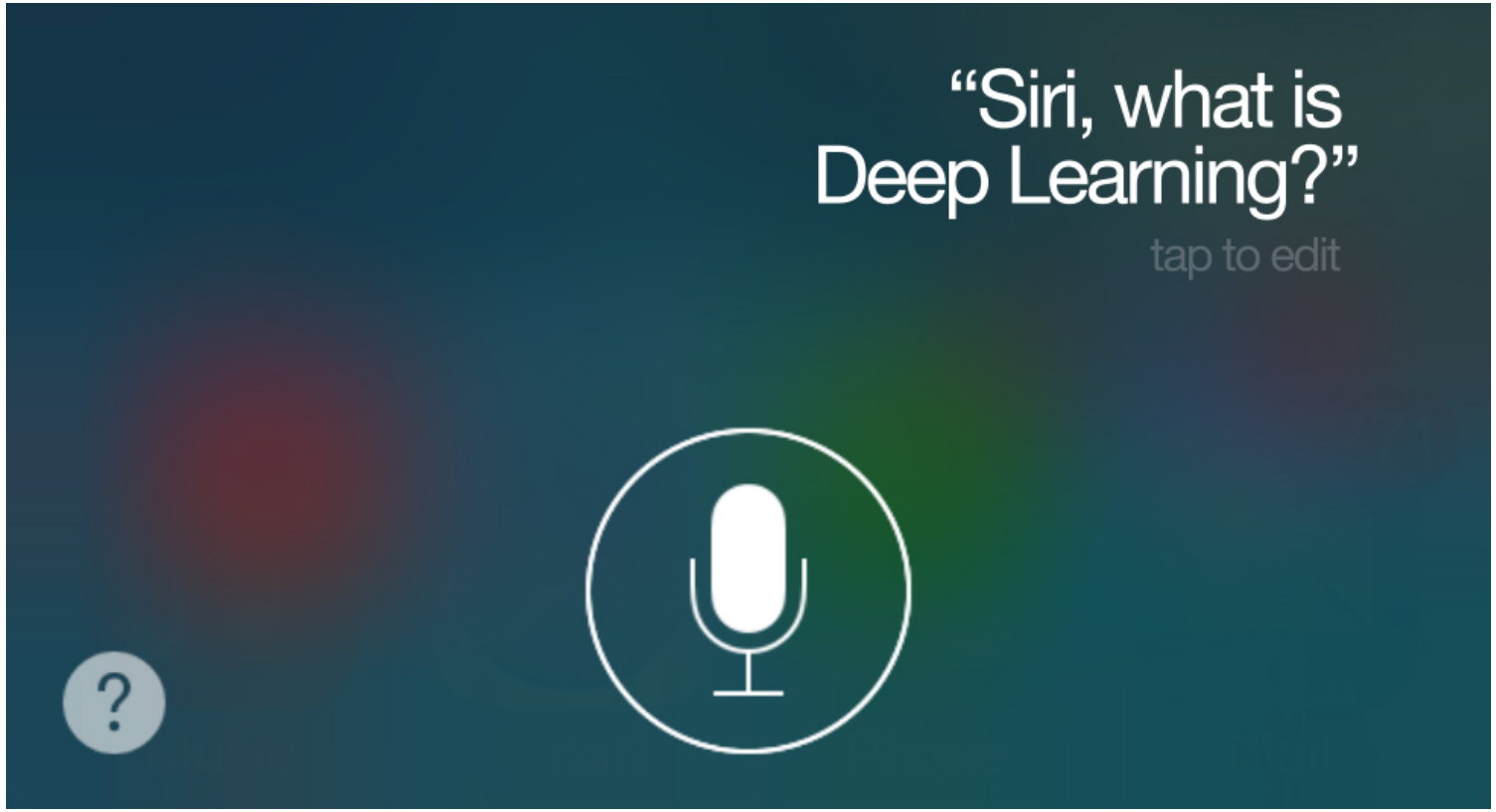
theguardian.com

# AlphaGo (2015)

First program to beat a professional Go player

# iOS Speech Synthesis (2016-)

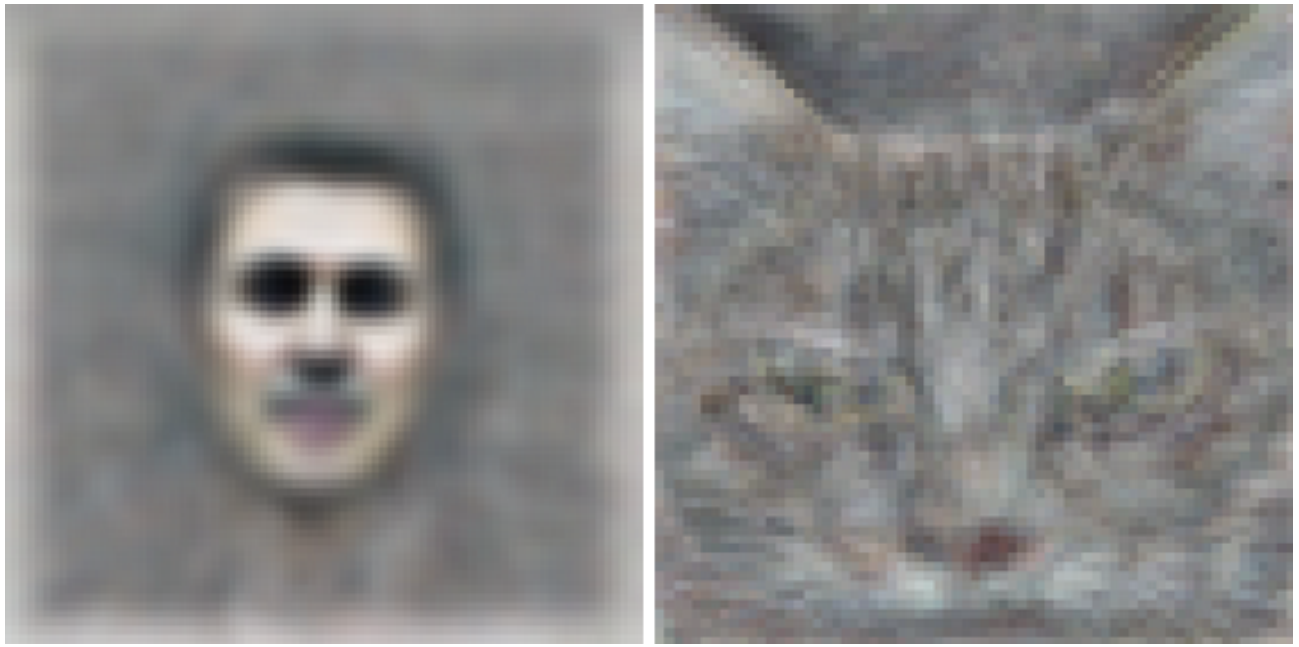
Trained from 20 hours of high quality speech




machinelearning.apple.com

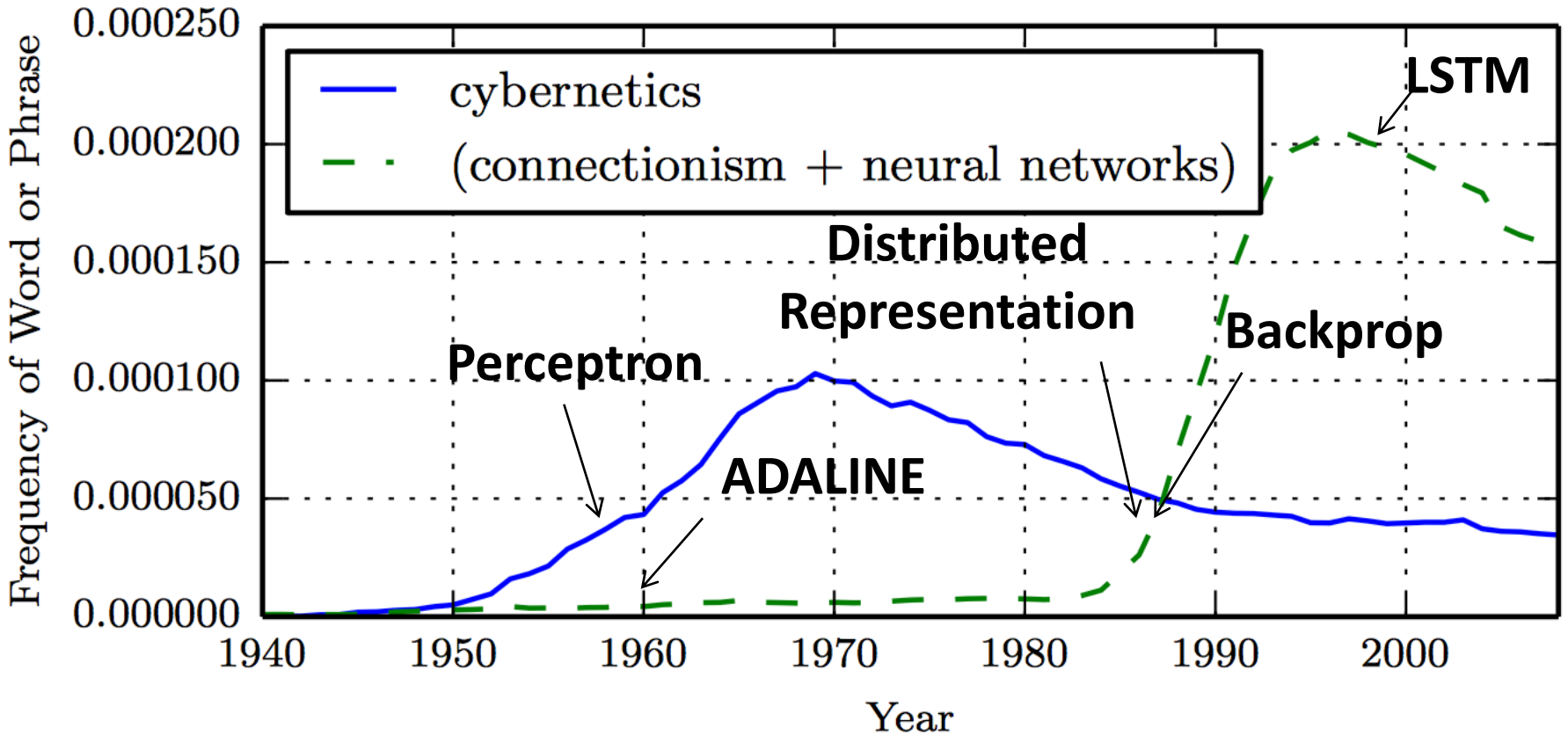
# Google Brain (2012)

- Differentiate between human face and cat
  - Neural network with 1 billion connections
  - 10 million 200x200 pixel images from YouTube



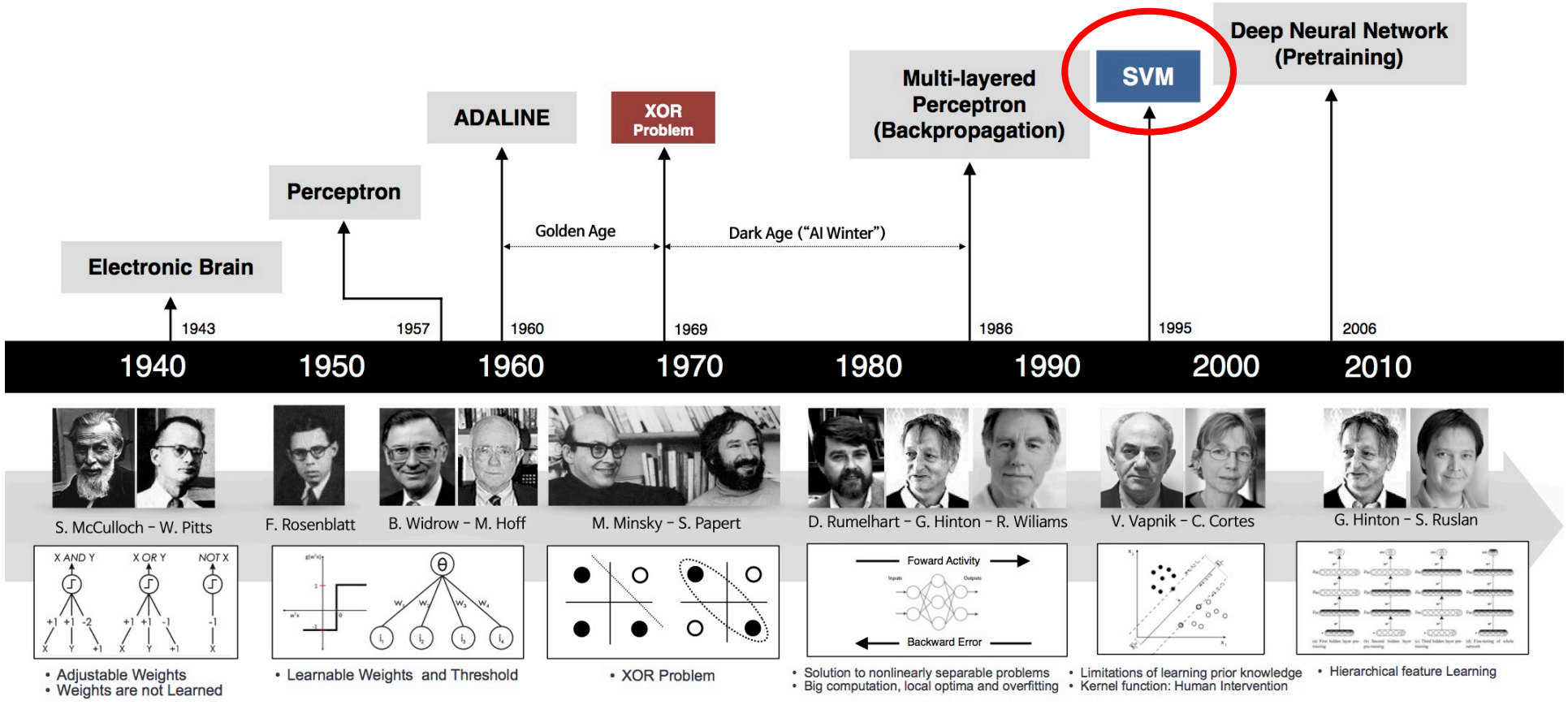
Le et al., ICML 2012

# Historical Trends




(Goodfellow 2016)

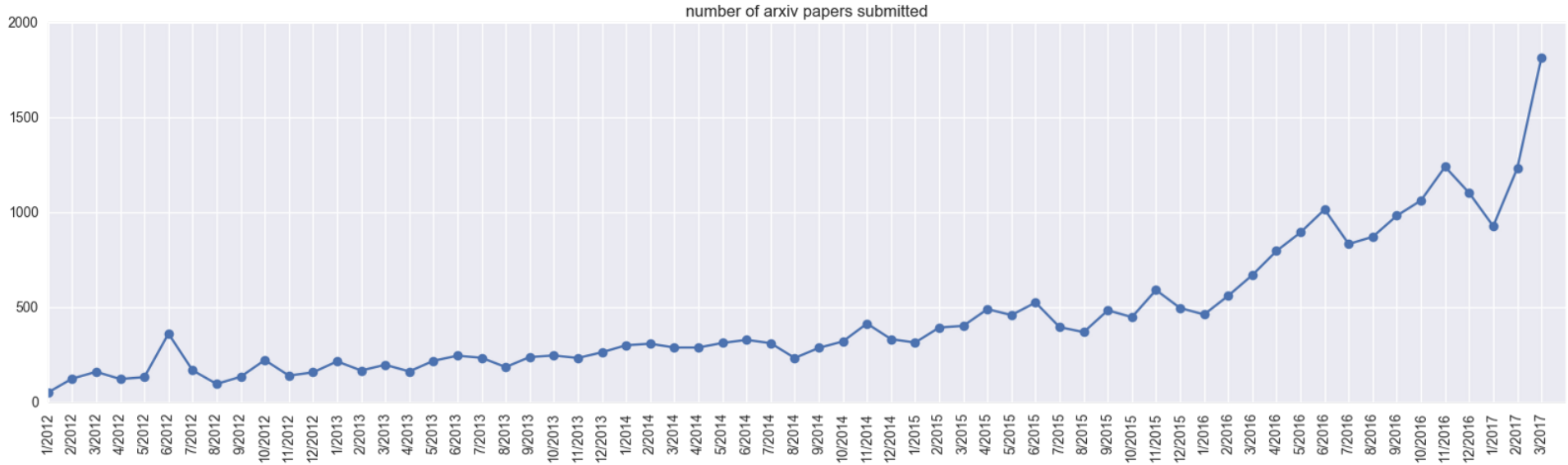
# Historical Trends



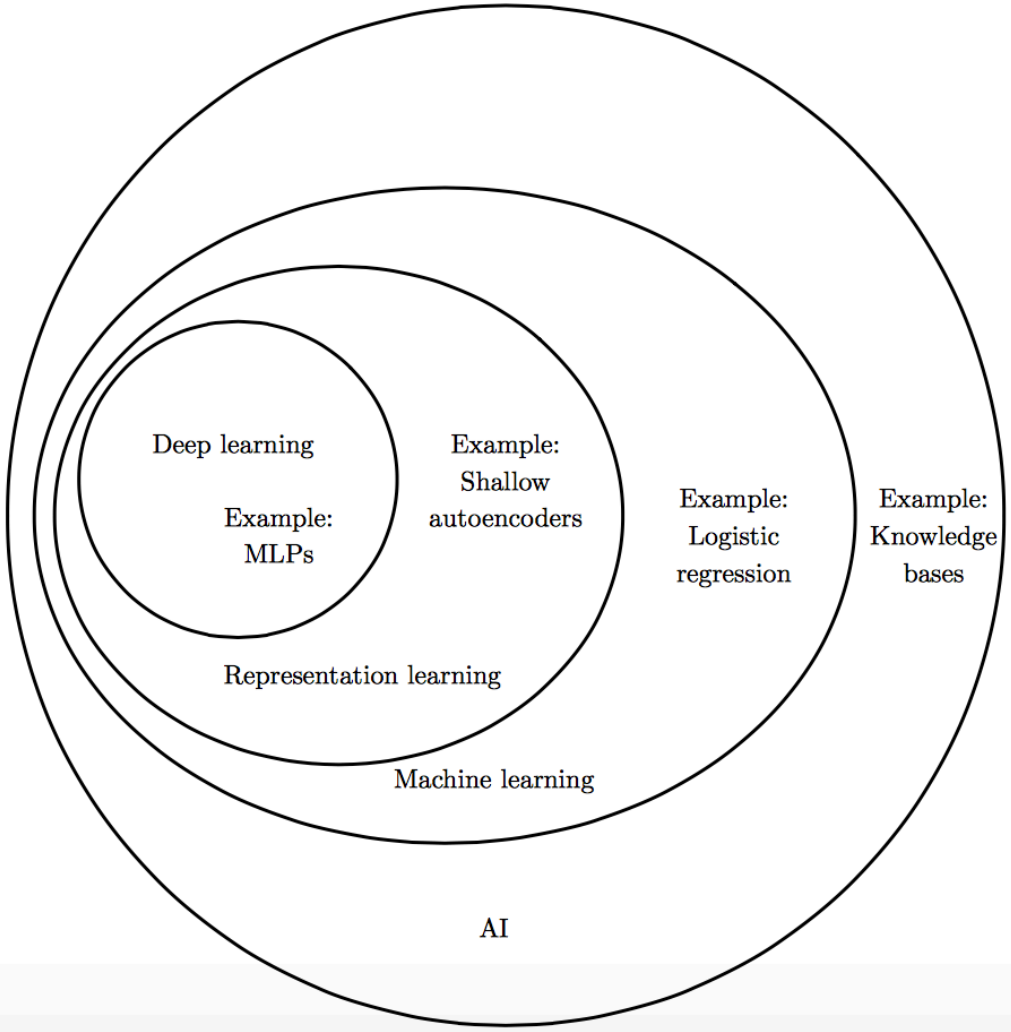

beamandrew.github.io/

# Historical Trends

## ArXiv papers on deep learning: 2012-2017

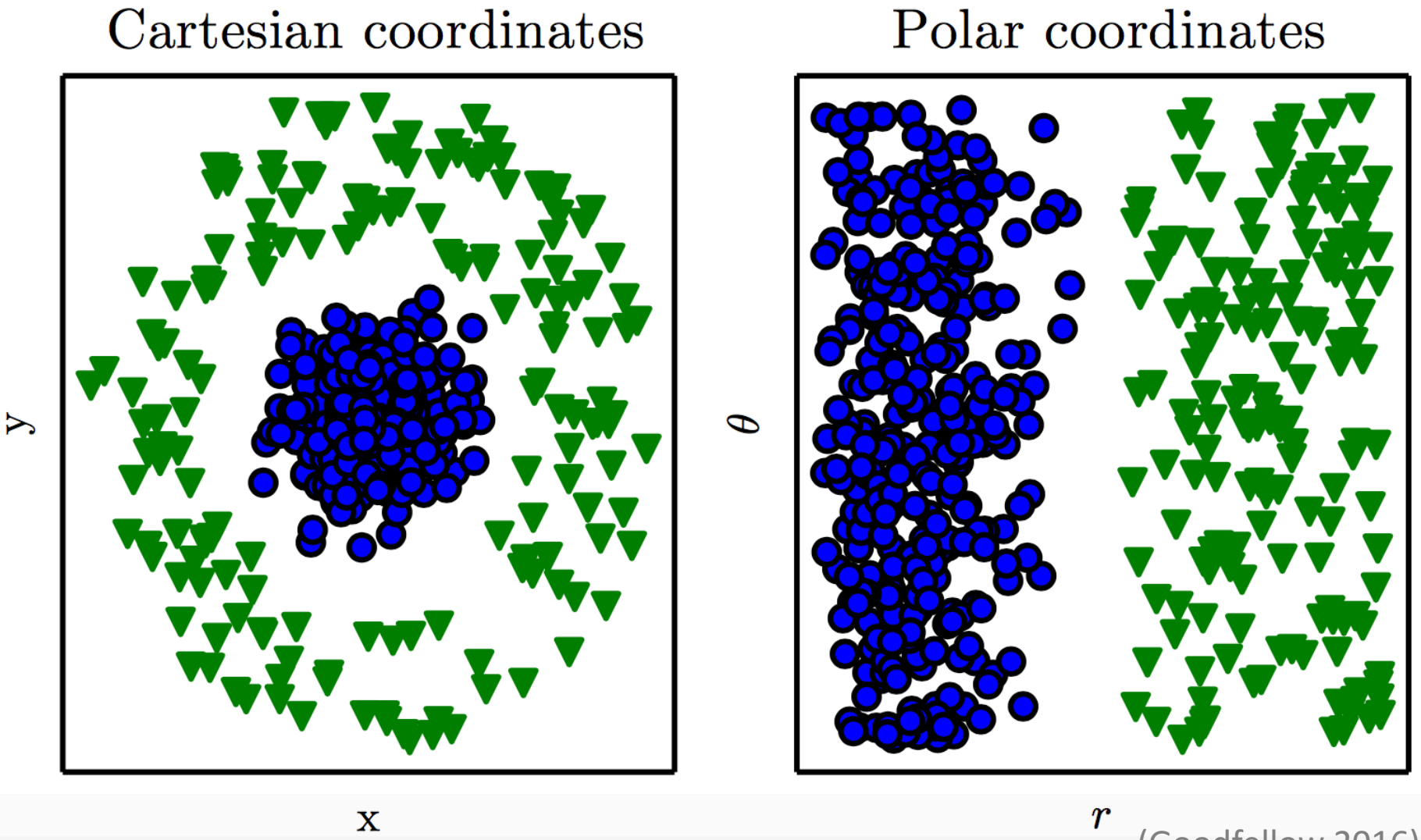


# Deep Learning vs Classical ML



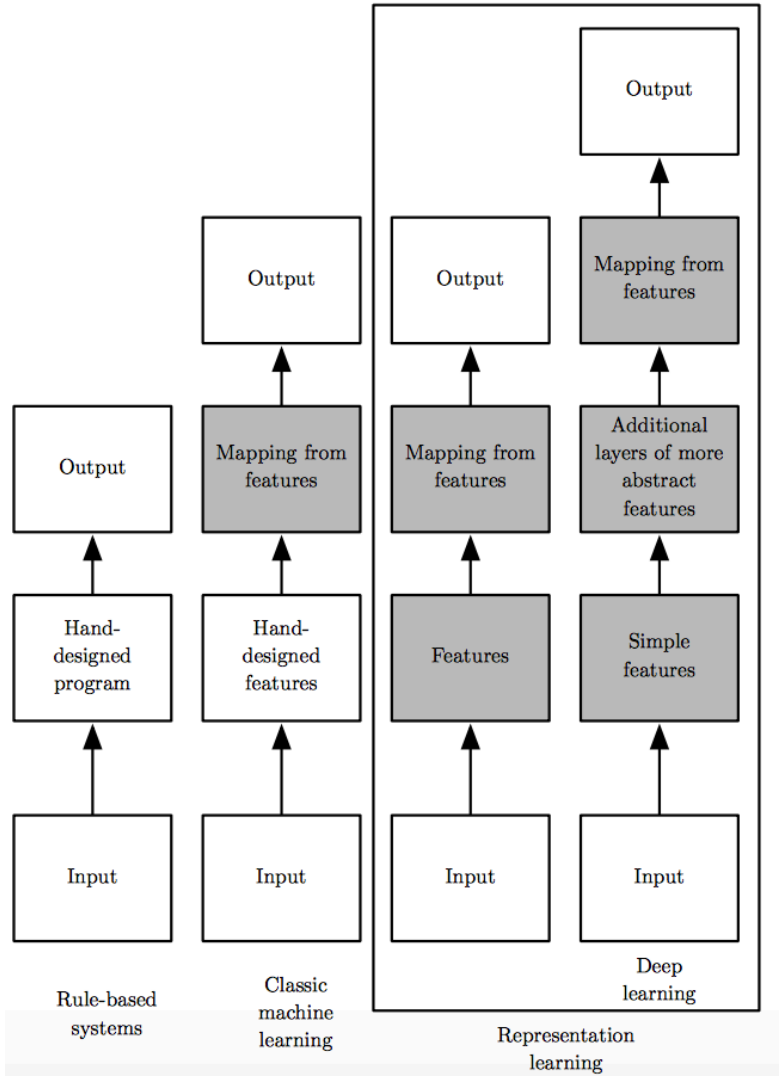

(Goodfellow 2016)

# Representation Matters



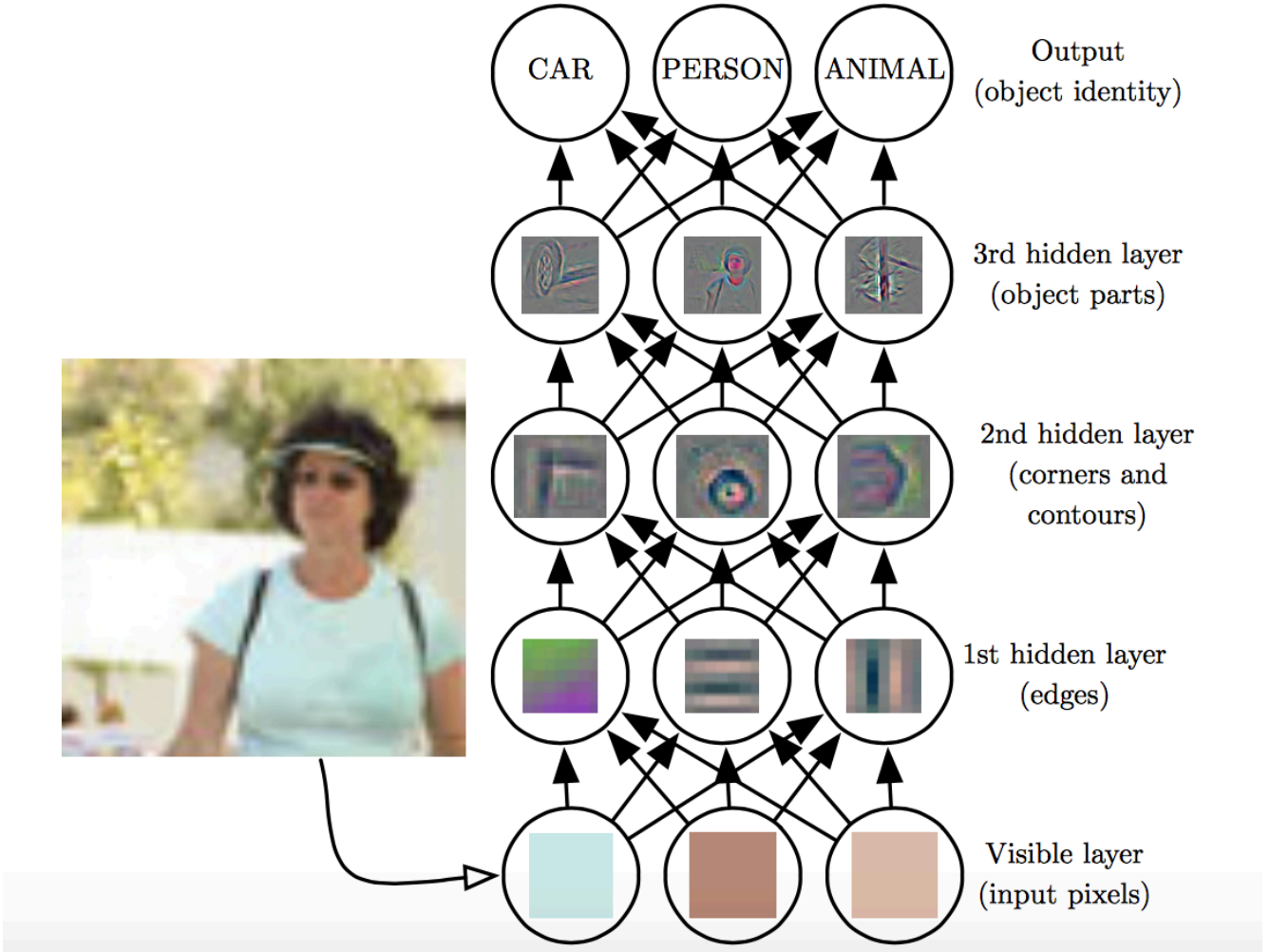

(Goodfellow 2016)

# Learning Multiple Components



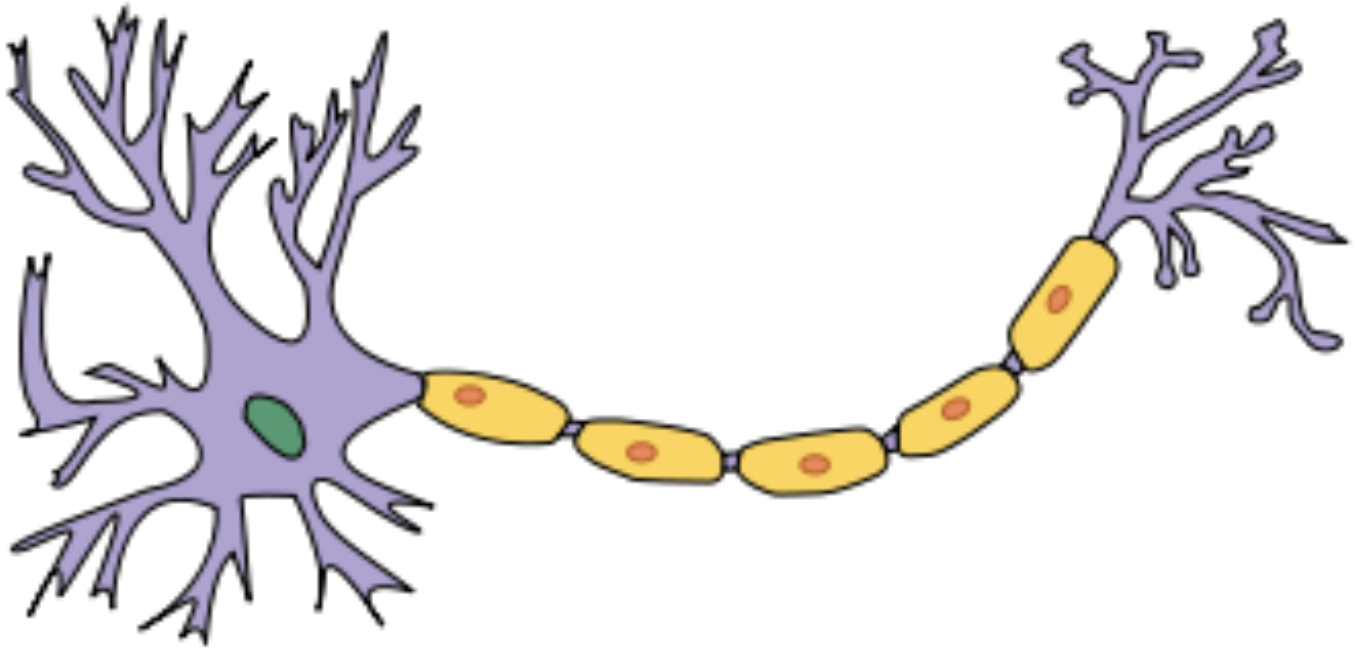

(Goodfellow 2016)

# Depth = Repeated Compositions




(Goodfellow 2016)

# Is deep learning inspired by the brain?



en.wikipedia.org

# Is deep learning inspired by the brain?

- Earliest learning algorithms intended to model biological learning
- Modern perspective:
  - Deep learning is a general principle of learning *multiple levels of compositions,* not necessarily neurally inspired

# Why is deep learning so popular now?

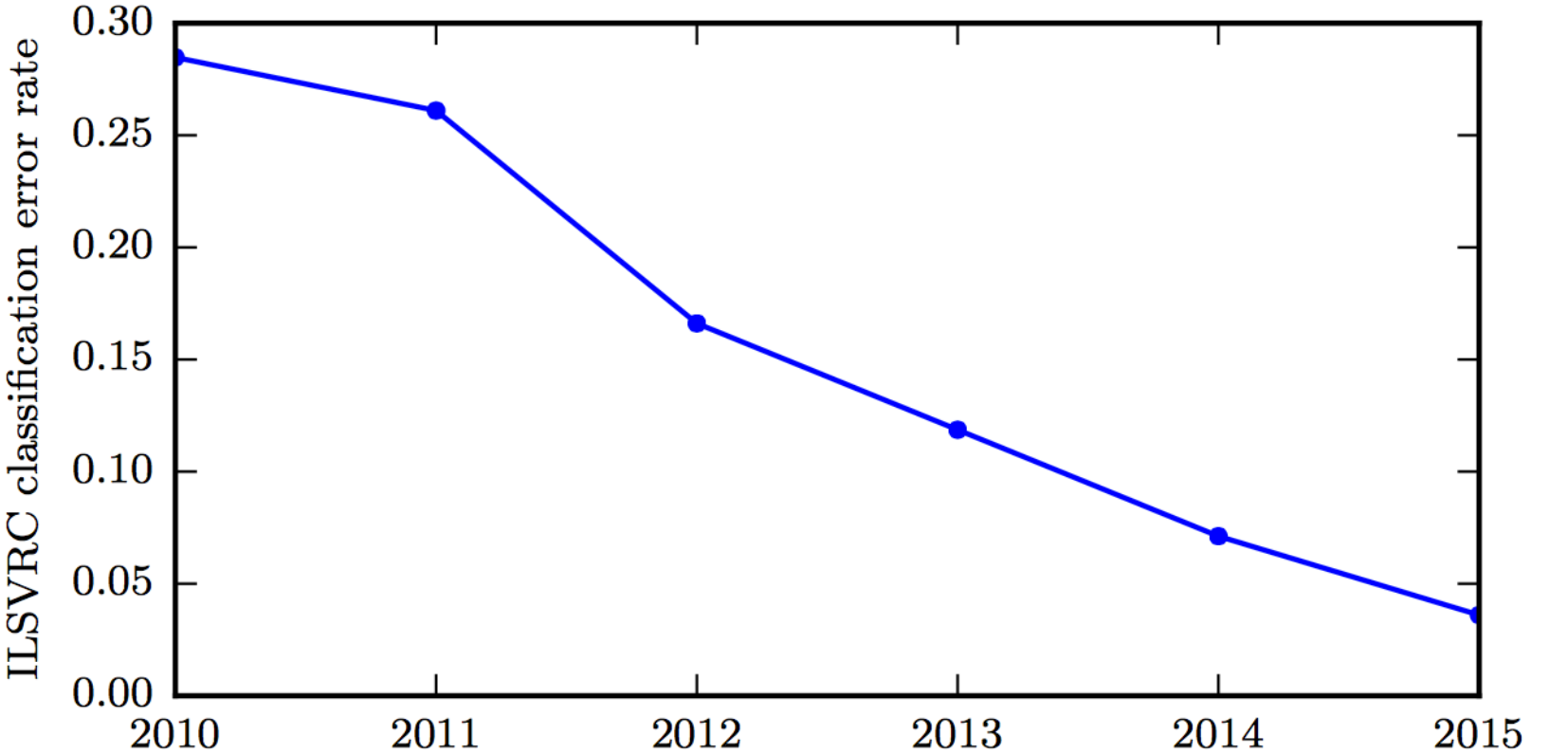
- Resurgence in 2006:

> G. Hinton, O. Simon, and Y-W The. “A fast learning algorithm for deep belief nets”, *Neural Computation*, 2006

- Breakthrough in 2012:
  - State-of-the-art performance in speech recognition
  - Best submission in Large Scale Visual Recognition Challenge (LSVRC)

# Increasing Accuracy

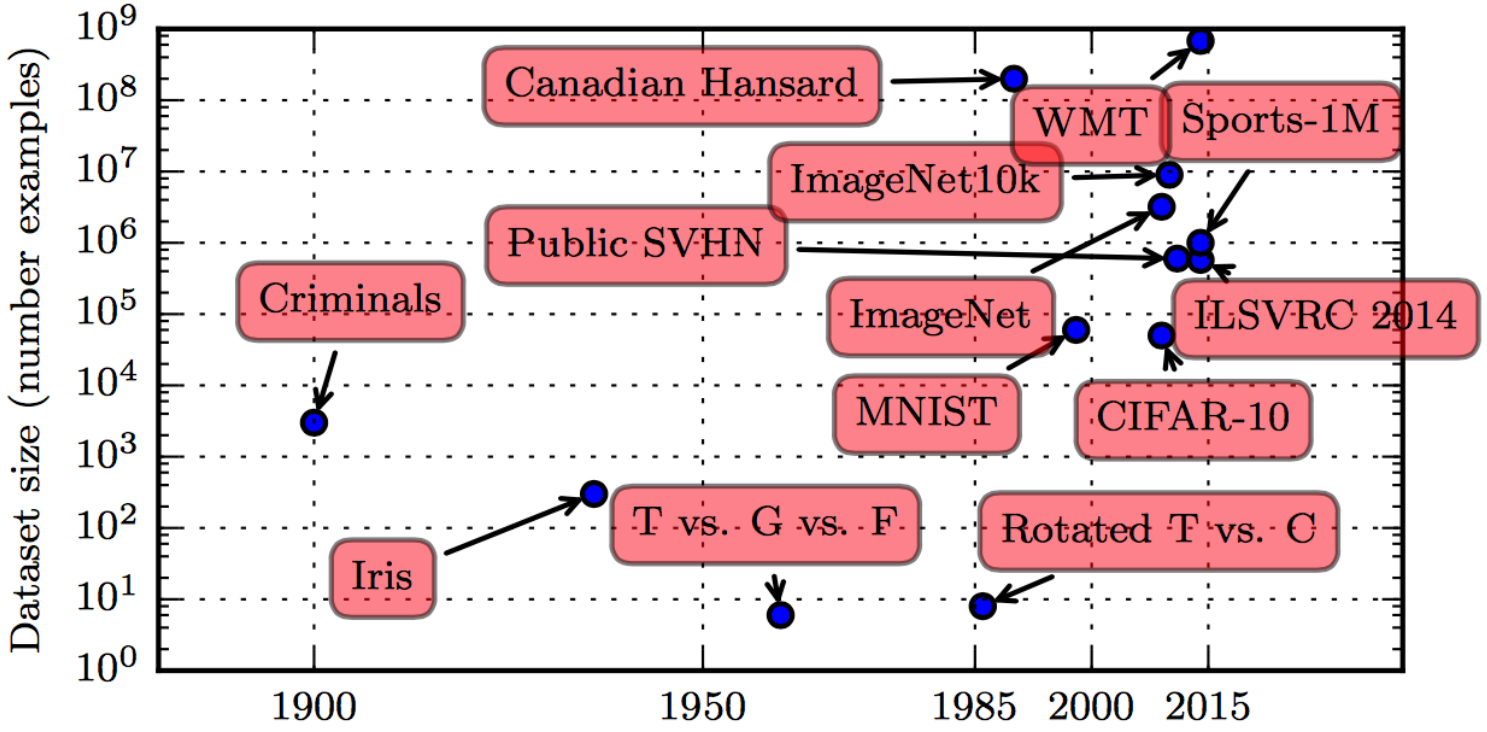
Large Scale Visual Recognition Challenge (LSVRC)




(Goodfellow 2016)

# What changed?

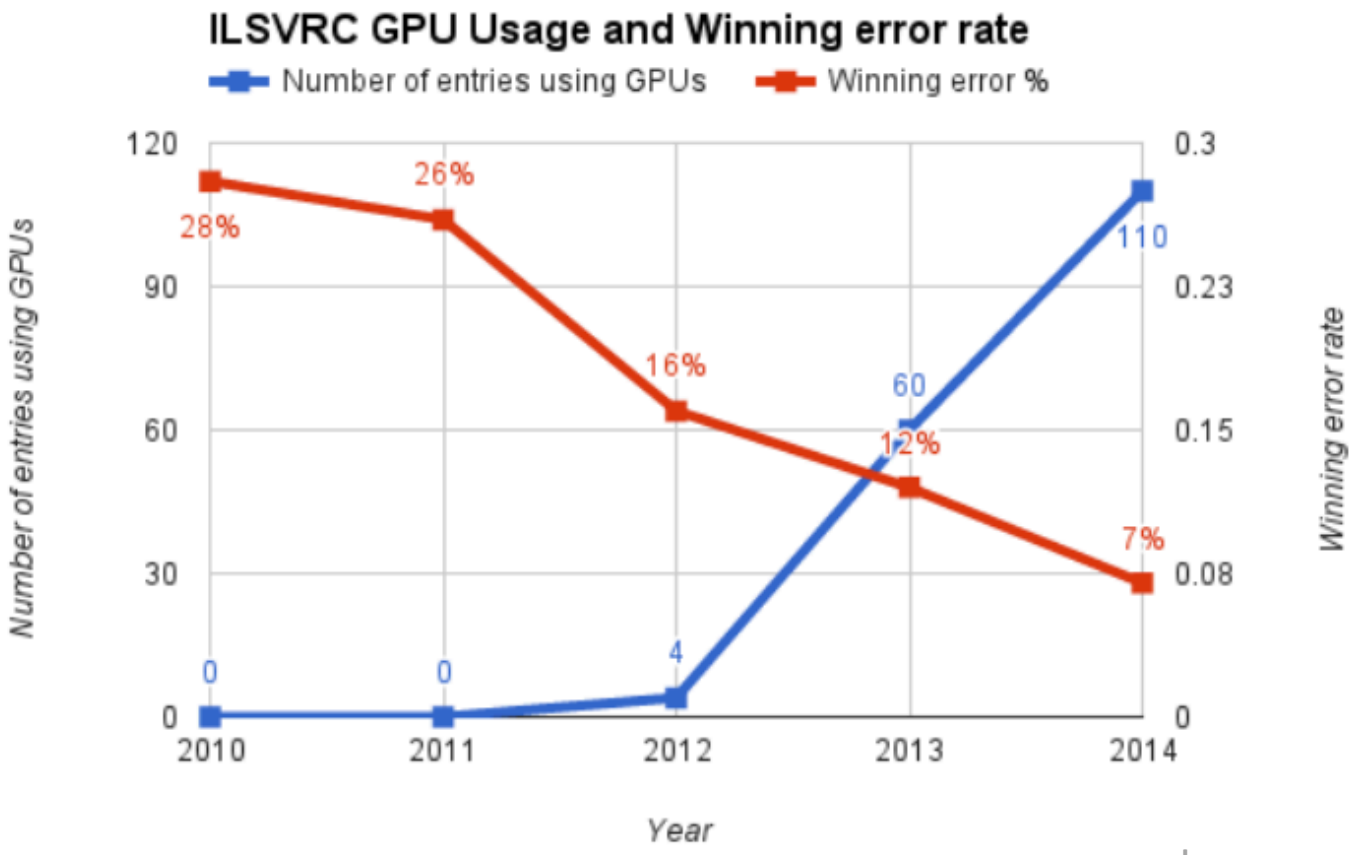
- Large, high-quality labeled data sets




(Goodfellow 2016)

# What changed?

- Massively parallel computing with GPUs



# What changed?

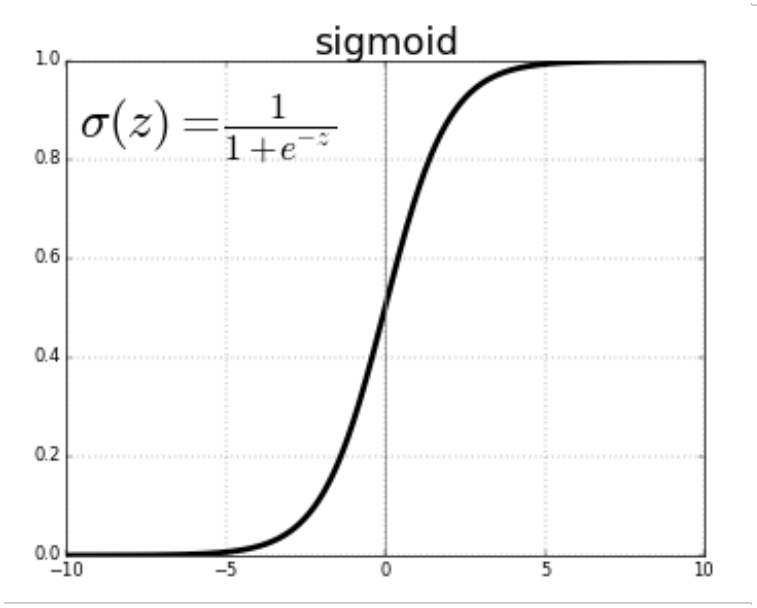
- New regularization techniques
  - Dropout
  - Batch normalization
  - Data augmentation

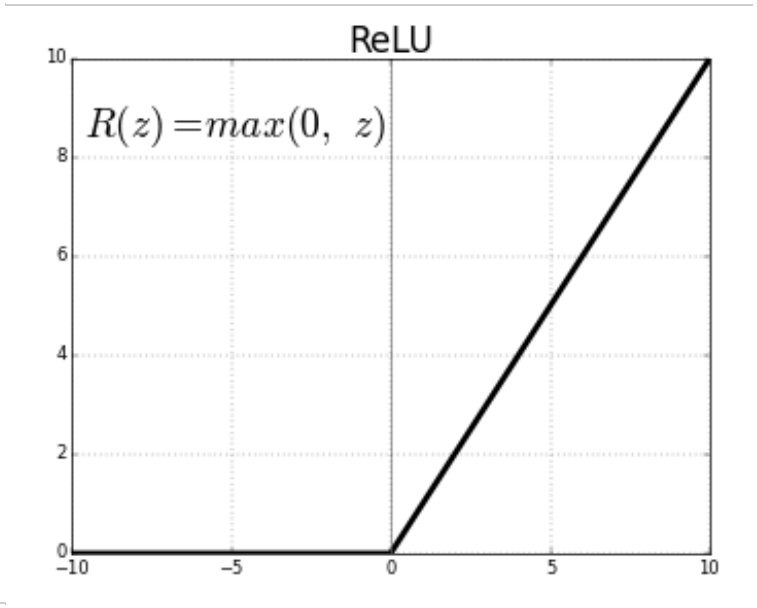
# What changed?

- Robust optimizers: Improvements to stochastic gradient descent
  - Momentum
  - RMSProp
  - ADAM
  - …

# What changed?

- Backprop-friendly activation functions
  - Sigmoid -> Rectified Linear Units (ReLU)



# What changed?

- Software platforms with auto-differentiation capabilities, seamless integration with GPUs

# Basic feed forward NN

# Beyond Linear Models

Non-linear transform

$$f(x) = w^T \phi(x)$$

# Traditional ML

- Manually engineer $\phi$
  - Domain specific, enormous human effort
- Generic transform
  - Maps to a higher-dimensional space
  - Kernel methods: e.g. RBF kernels
  - Over fitting: does not generalize well to unseen data
  - Cannot encode enough prior information

# Deep Learning

Directly learn
feature map?

Defined by
hidden layers of the
neural network

$$f(x) = w^T \phi(x)$$

Can encode prior beliefs, generalizes well

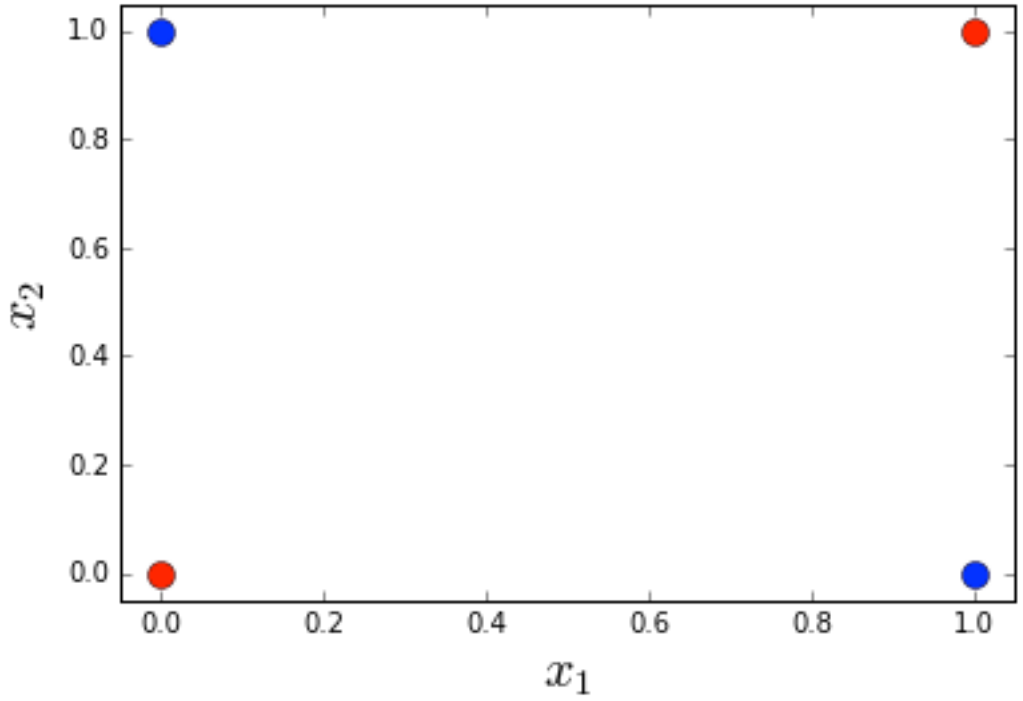
But, *non-convex optimization*

# SVM vs. Neural Networks

- Hand-written digit recognition: MNIST data

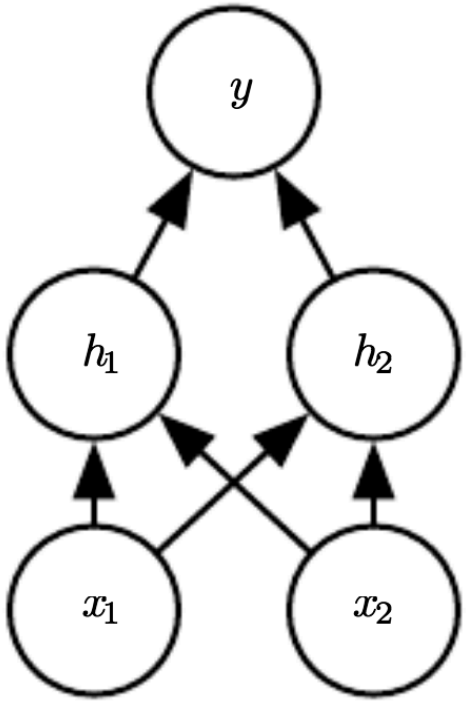
See illustration in notebook

# Example: Learning XOR




- Optimal linear model (sq. loss)
  - Predicts 0.5 on all points

# Example: Learning XOR




$$h_1 = \sigma(w_1^T x + c_1)$$

$$h_2 = \sigma(w_2^T x + c_2)$$

$$y = \sigma(w^T h + b)$$

where,

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

See illustration in notebook

# Design Choices

- Cost function
- Output units
- Hidden units
- Architecture
- Optimizer

# Cost Function

- Cross-entropy between training data and model distribution (i.e. negative log-likelihood)

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x})$$

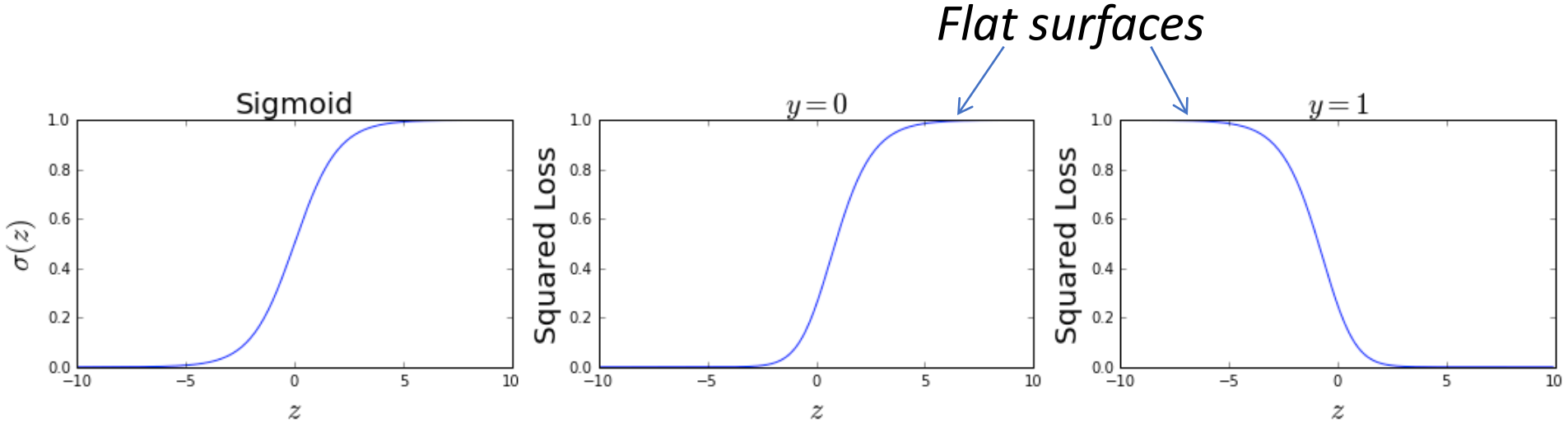
- Do not need to design separate cost functions
- Gradient of cost function must be large enough

# Cost Function

- Example: sigmoid output + squared loss

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

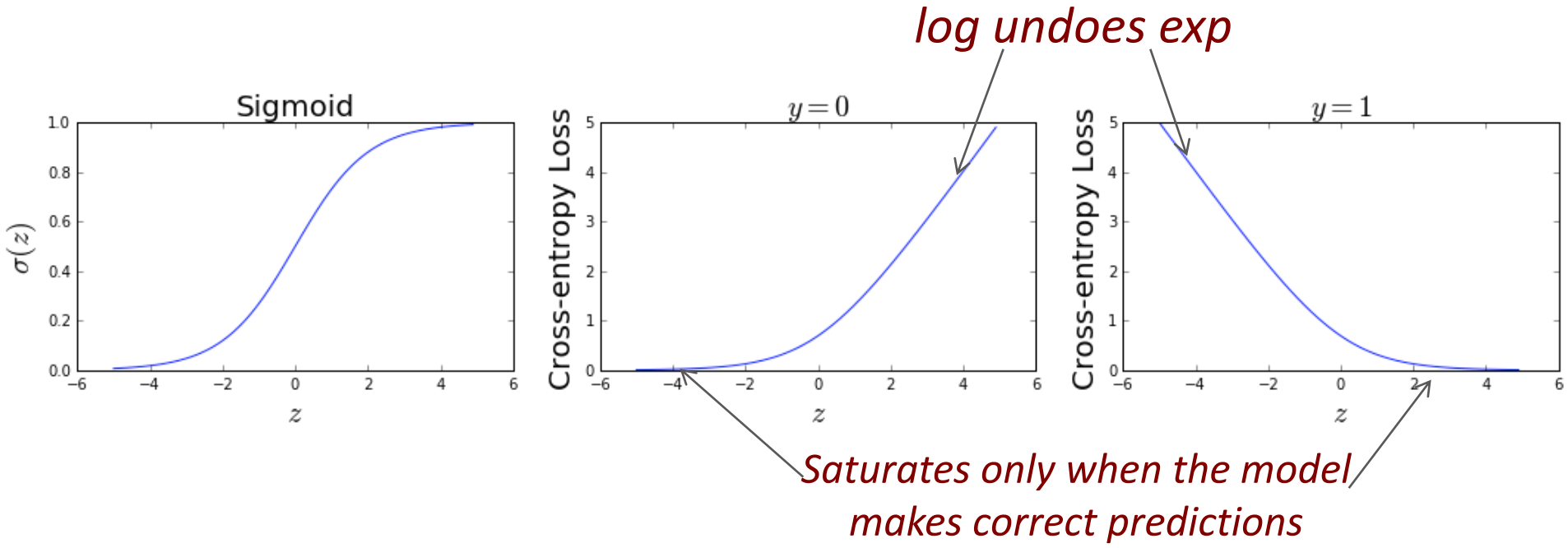
$$L_{sq}(y,z) = (y - \sigma(z))^2$$



# Cost Function

- Example: sigmoid output + cross-entropy loss

$$L_{ce}(y,z) = -(y\log(z) + (1-y)\log(1-z))$$



# Design Choices

- Cost function
- Output units
- Hidden units
- Architecture
- Optimizer

# Output Units

| Output Type | Output Distribution | Output Layer | Cost Function |
|---|---|---|---|
| Binary | Bernoulli | Sigmoid | Binary cross-entropy |
| Discrete | Multinoulli | Softmax | Discrete cross-entropy |
| Continuous | Gaussian | Linear | Gaussian cross-entropy (MSE) |
| Continuous | Mixture of Gaussian | Mixture Density | Cross-entropy |
| Continuous | Arbitrary | See part III: GAN, VAE, FVBN | Various |

(Goodfellow 2017)

# Softmax Output

- Discrete / Multinoulli output distribution
- For output scores $z_1, \ldots, z_n$

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

- Log-likelihood undoes exp

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

$$\approx z_i - \max_j z_j$$

(Score to target label – Maximum score)

# Design Choices

- Cost function
- Output units
- Hidden units
- Architecture
- Optimizer

# Hidden Units

$$\mathbf{h} = g(\mathbf{W}^T x + \mathbf{b})$$

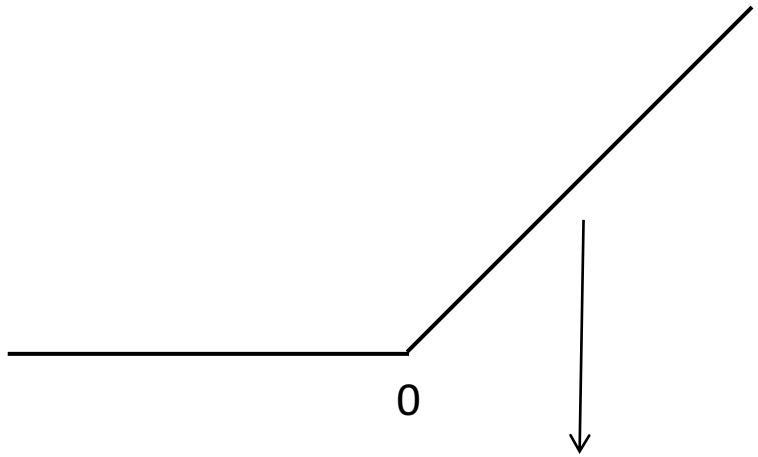
with activation function $g$

- Ensure gradients remain large through hidden unit
- Preferred: piece-wise linear activation
- Avoid sigmoid/tanh activation
  - Do not provide useful gradient info when they saturate

# ReLU

- Rectified Linear Units

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




- Gradient is 1 whenever unit is active
  - More useful for learning compared to sigmoid
  - No useful gradient information when z<0

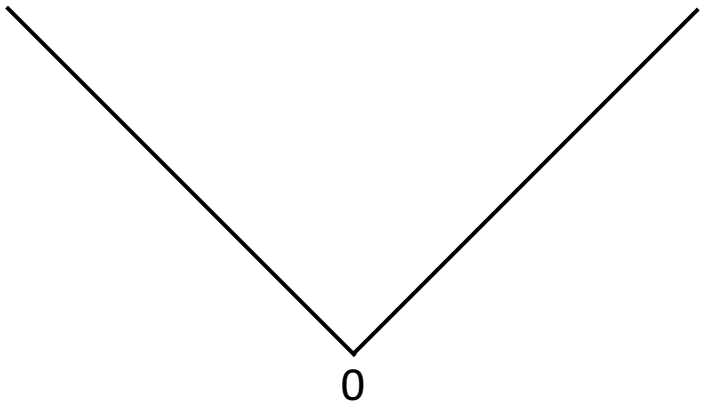
# Generalized ReLU

- Generalization: For $\alpha_i > 0$

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

- E.g. Absolute value ReLU:

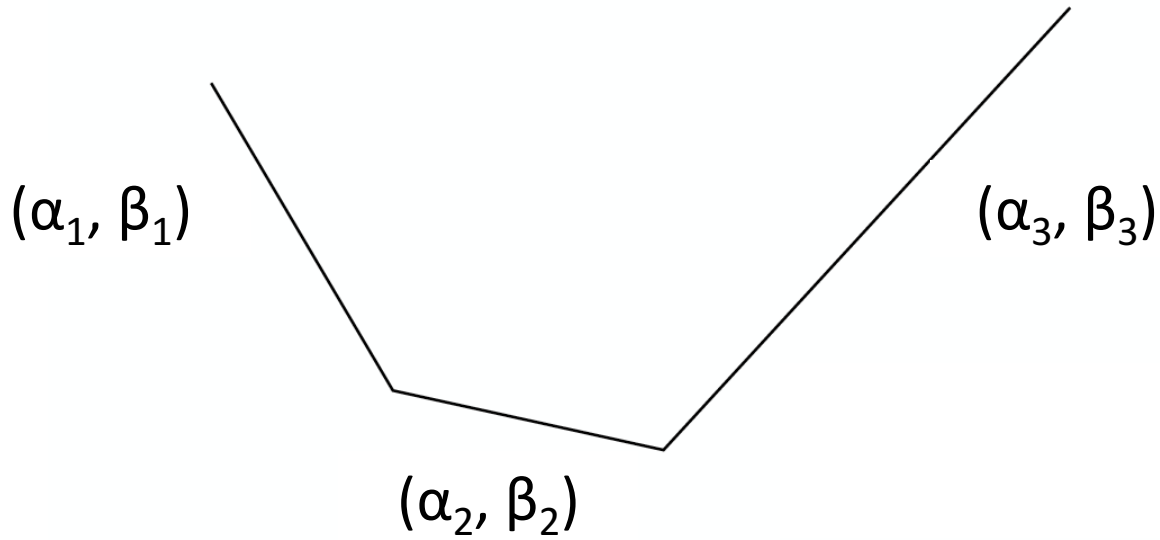
$$\alpha_i = -1 \Rightarrow g(z) = |z|$$



# Maxout

- Directly learn the activation function
  - Max of $k$ linear functions

$$g(z) = \max_{i \in \{1,\ldots,k\}} \alpha_i z_i + \beta_i$$

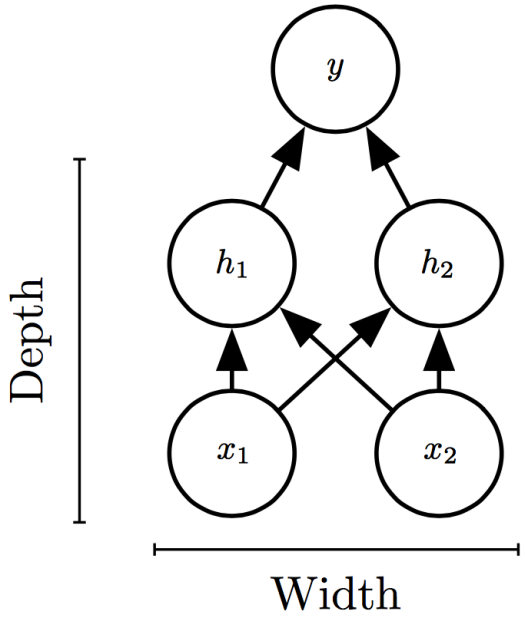


# Design Choices

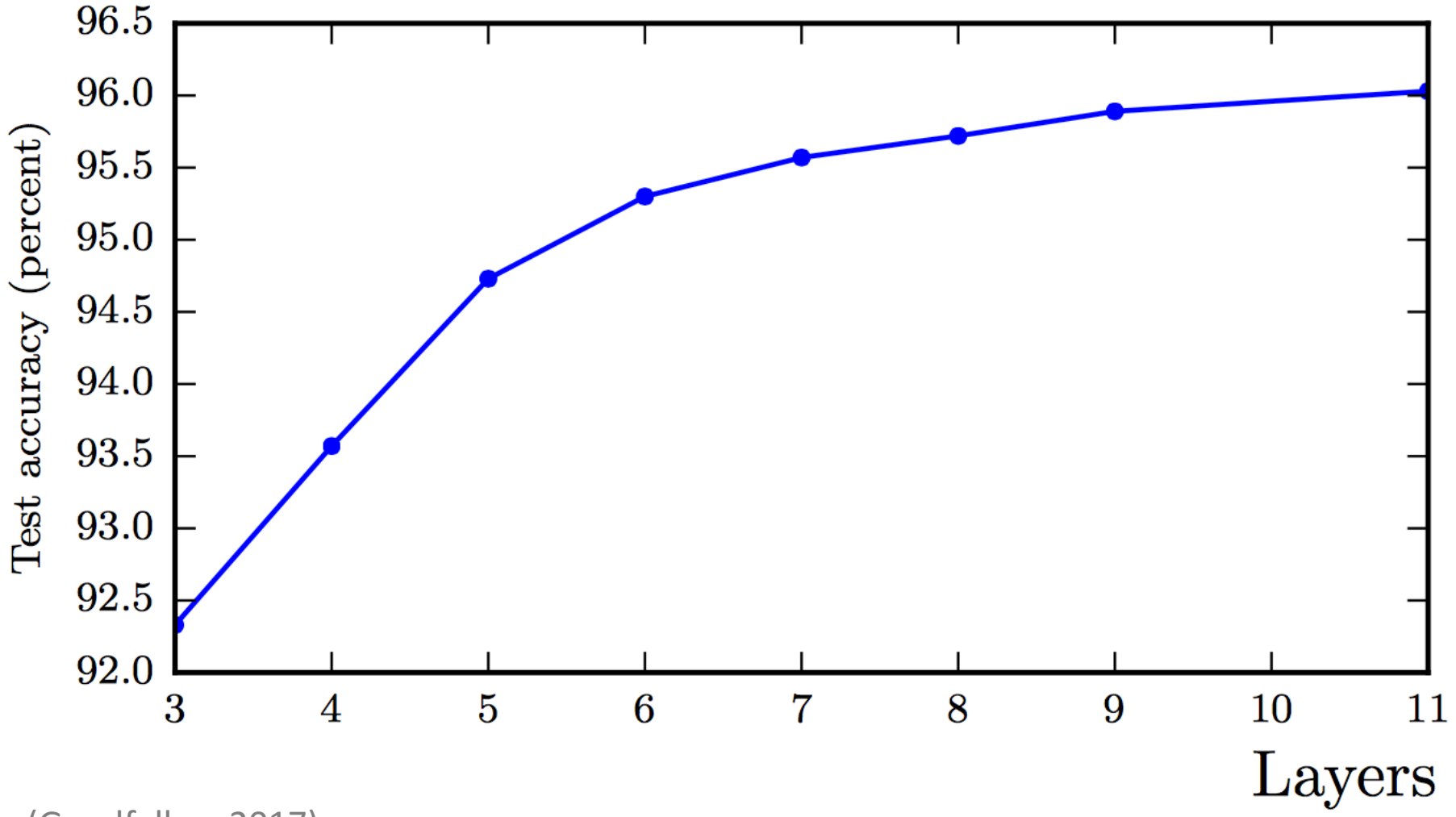
- Cost function
- Output units
- Hidden units
- Architecture
- Optimizer

# Universal Approximation Theorem

- One hidden layer is enough to *represent* an approximation of any function to an arbitrary degree of accuracy
- So why deeper?
  - Shallow net may need (exponentially) more width
  - Shallow net may overfit more



# Better Generalization with Depth




(Goodfellow 2017)

# Design Choices

- Cost function
- Output units
- Hidden units
- Architecture
- Optimizer

# Gradient-based Optimizer

(e.g. stochastic gradient descent)

- "Chain rule" for computing gradients:

$$\mathbf{y} = g(\mathbf{x}) \qquad z = f(\mathbf{y})$$

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

- For deeper networks

Naïve computation takes exponential time

$$\frac{\partial z}{\partial x_i} = \sum_{j_1} \cdots \sum_{j_m} \frac{\partial z}{\partial y_{j_1}} \cdots \frac{\partial y_{j_m}}{\partial x_i}$$

# Backpropagation

- Avoids repeated sub-expressions
- Uses dynamic programming (table filling)
- Trades-off memory for speed

# Backprop: Arithmetic

- Jacobian-gradient products

$$\mathbf{z} = g(\mathbf{x})$$
$$y = f(\mathbf{z})$$

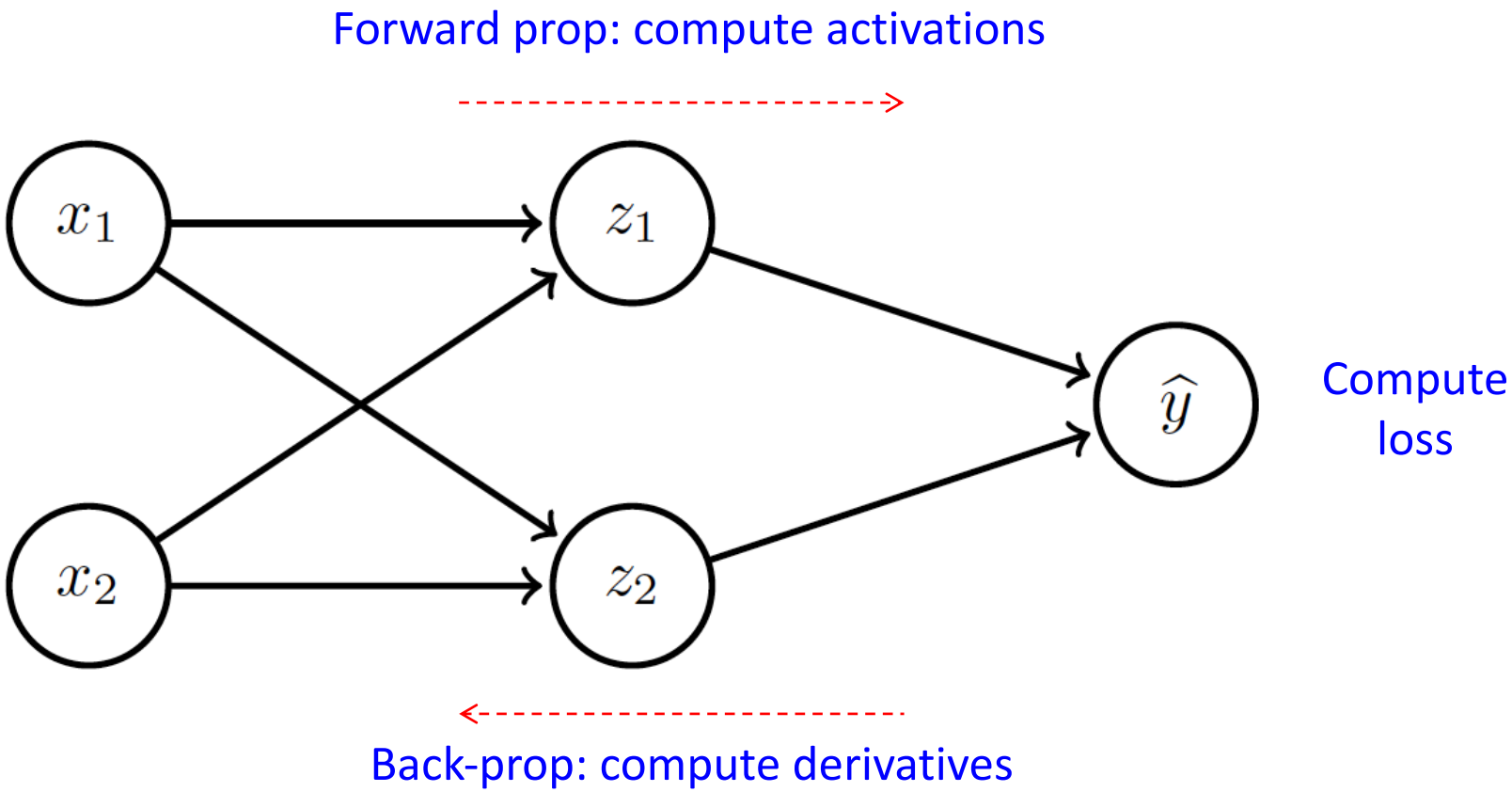
$$\begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \vdots \\ \frac{\partial y}{\partial x_m} \end{bmatrix} = \begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \cdots & \frac{\partial z_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_1}{\partial x_n} & \cdots & \frac{\partial z_m}{\partial x_n} \end{bmatrix} \times \begin{bmatrix} \frac{\partial y}{\partial z_1} \\ \vdots \\ \frac{\partial y}{\partial z_m} \end{bmatrix}$$

grad w.r.t. **x** — Jacobian of 'g' — grad w.r.t. **z**

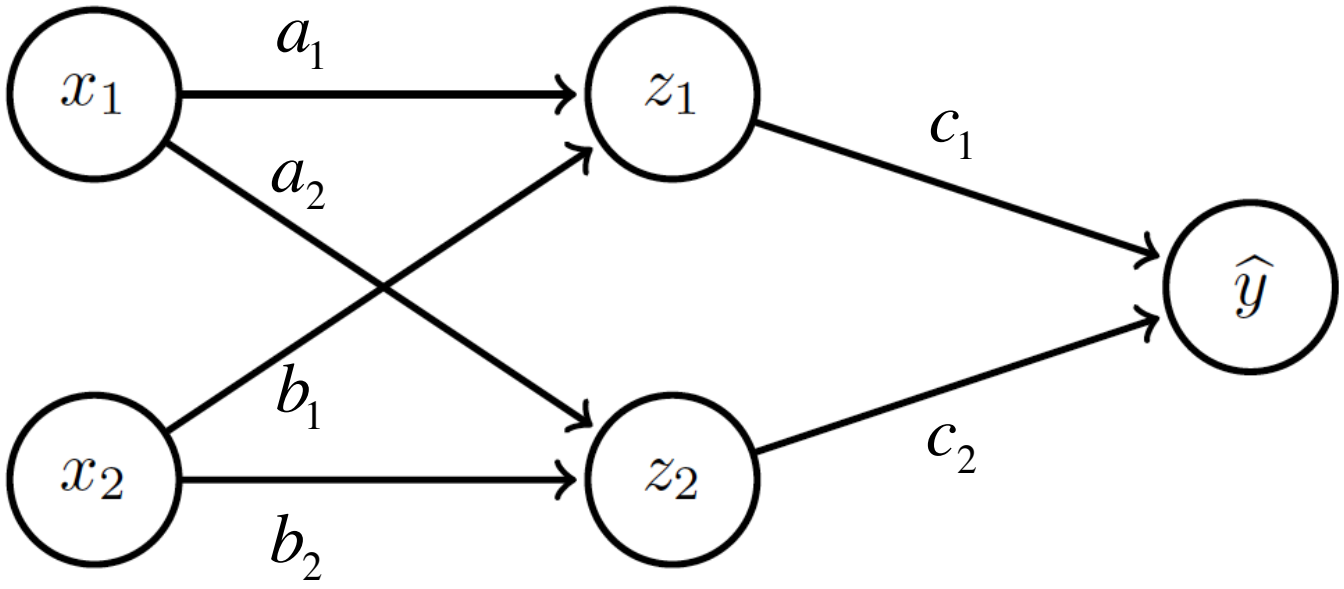
$$\nabla_{\mathbf{x}} y = \left(\frac{\partial \mathbf{z}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{z}} y$$

Apply recursively!

# Backprop: Overview



# Backprop: Example

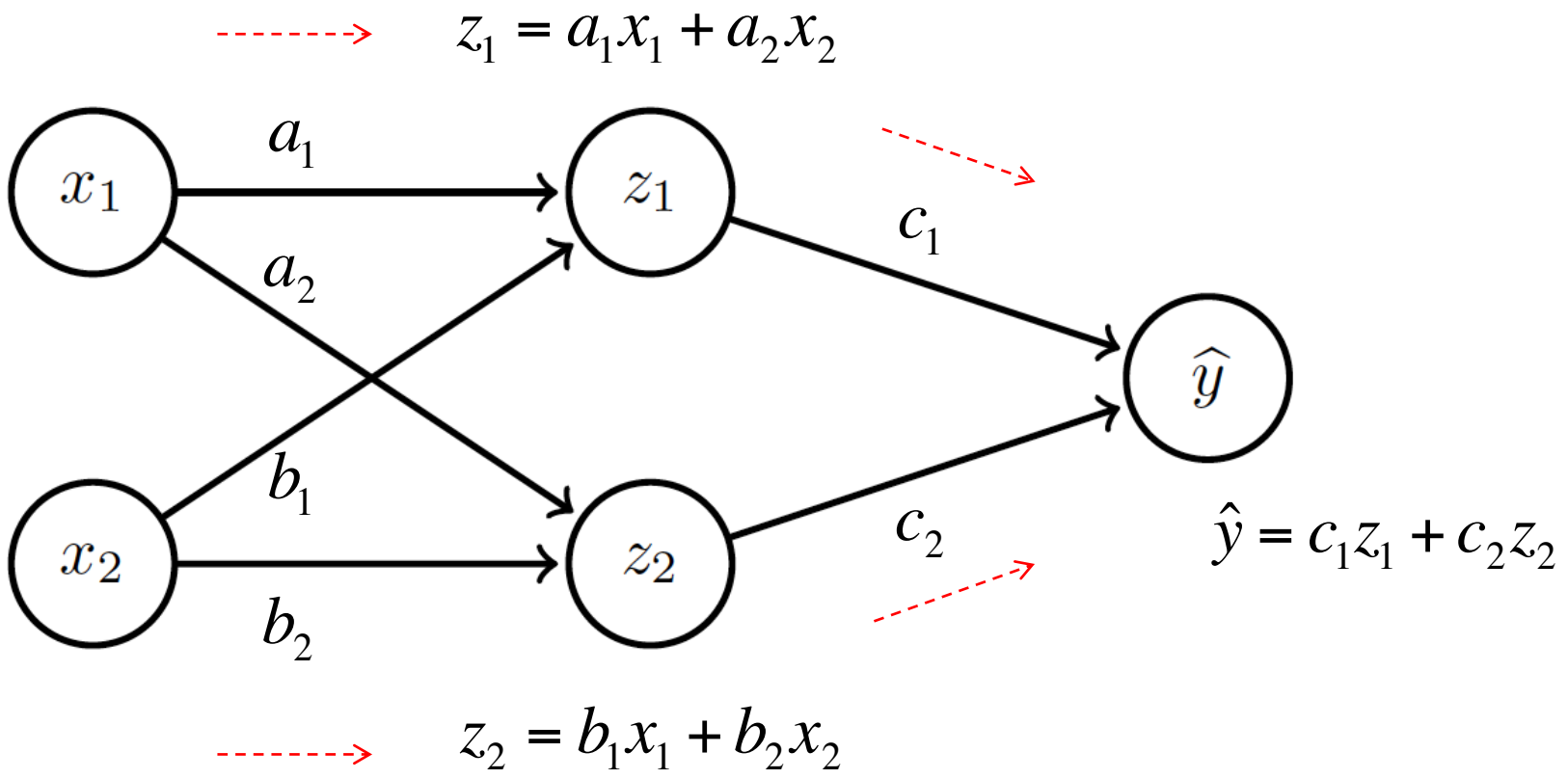



Linear activation functions

No bias

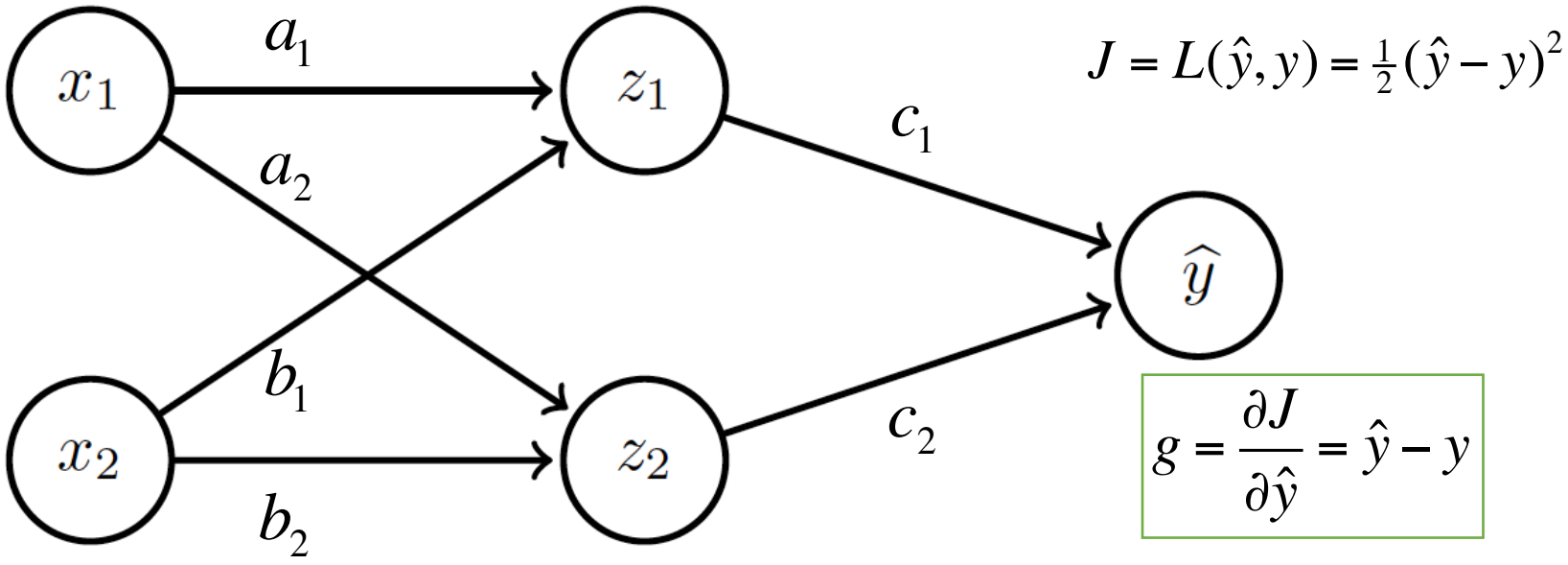
Squared loss

# Backprop: Example




Forward prop: Propagate activations to output layer

# Backprop: Example

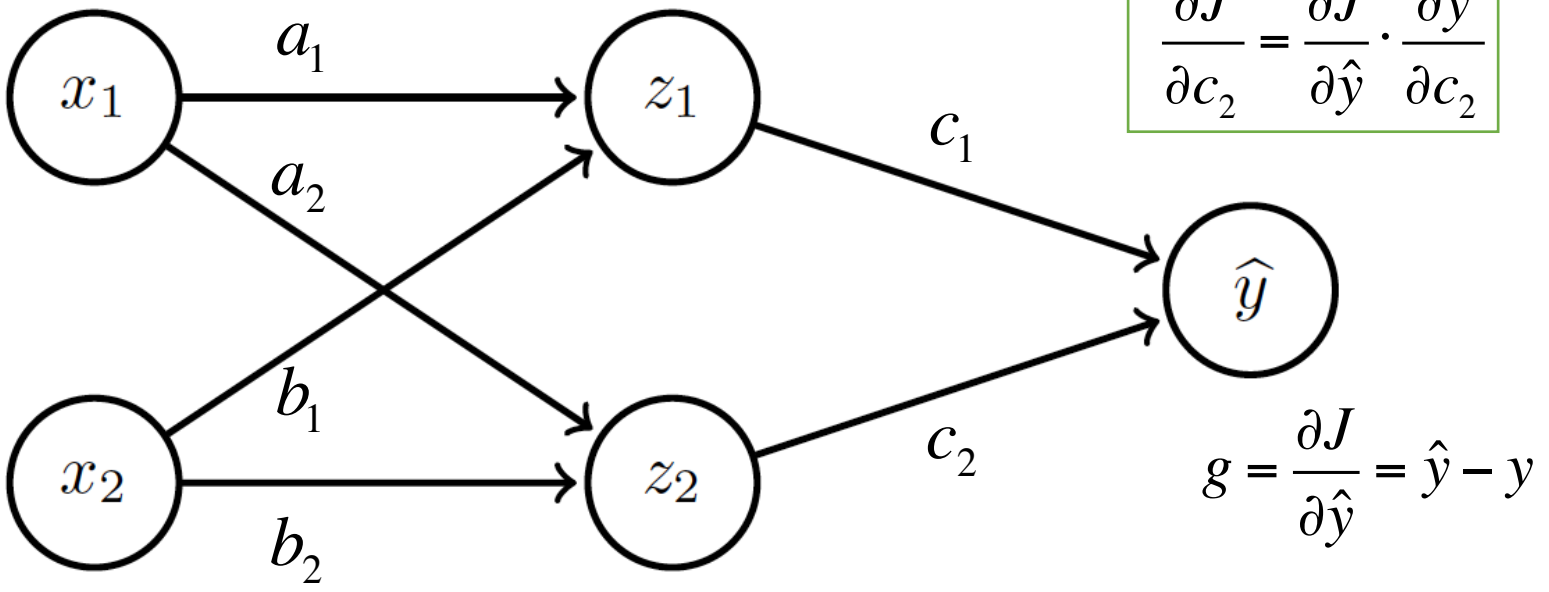



Backward prop: Compute loss and its derivative

# Backprop: Example

$$\frac{\partial J}{\partial c_1} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial c_1}$$

$$\frac{\partial J}{\partial c_2} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial c_2}$$



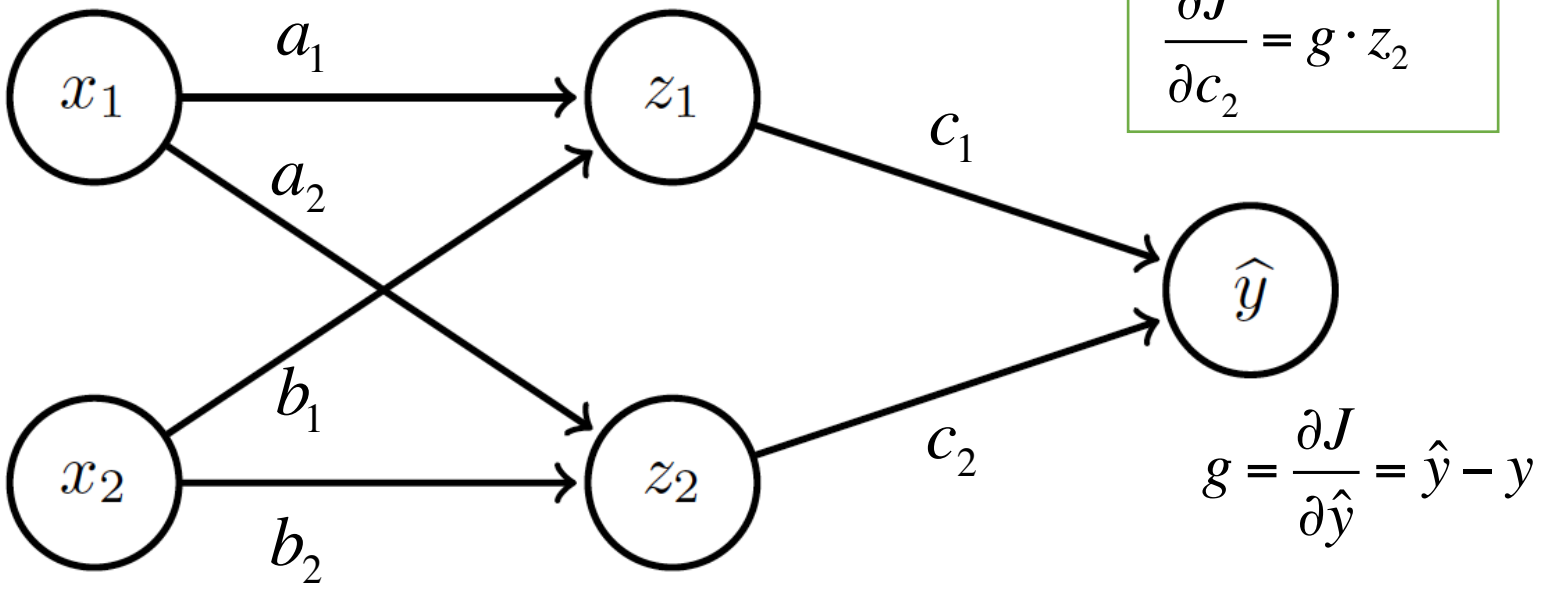

$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

Backward prop: Compute derivatives w.r.t. weights $c_1$ and $c_2$

# Backprop: Example

$$\frac{\partial J}{\partial c_1} = g \cdot z_1$$

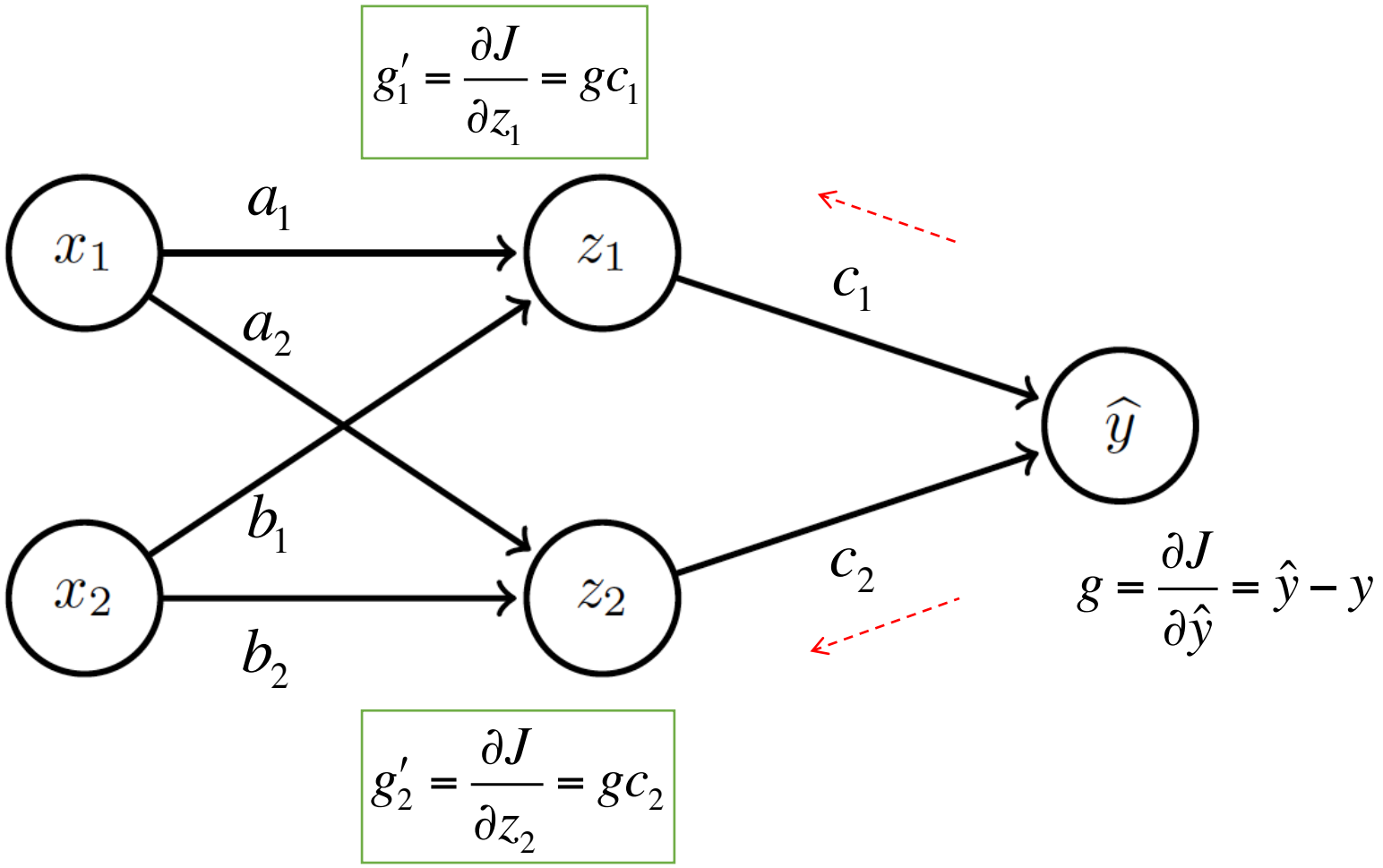
$$\frac{\partial J}{\partial c_2} = g \cdot z_2$$




$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

Backward prop: Compute derivatives w.r.t. weights $c_1$ and $c_2$

# Backprop: Example



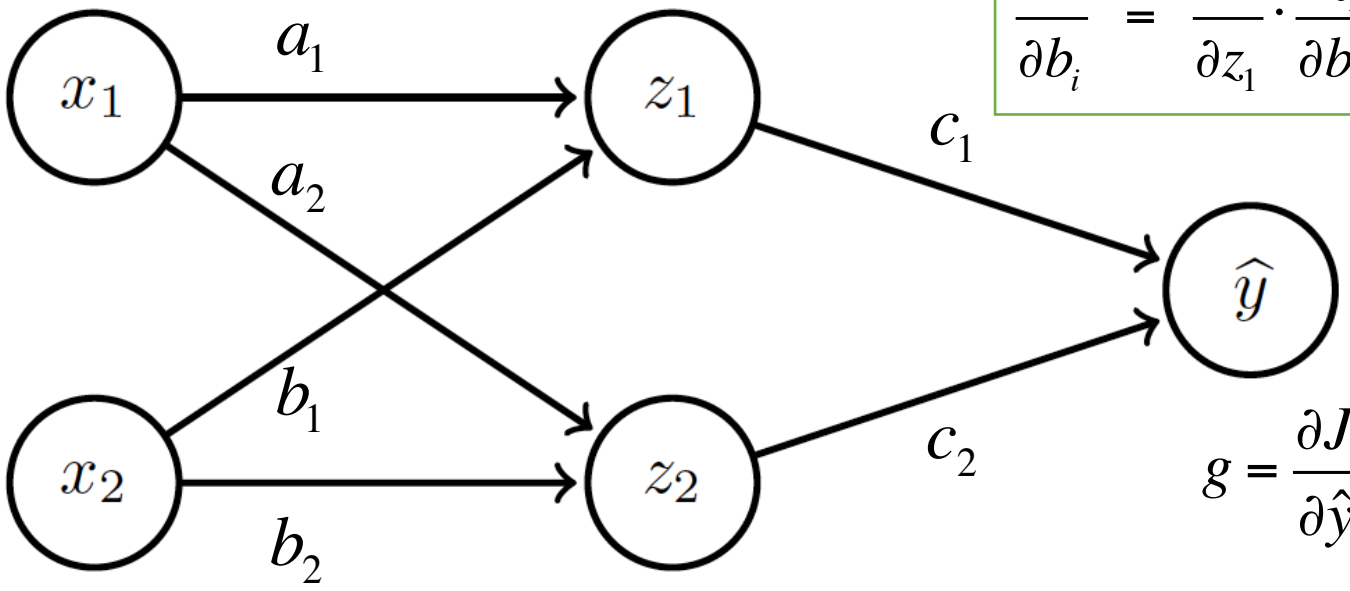

Backward prop: Propagate derivative to hidden layer

# Backprop: Example

$$\frac{\partial J}{\partial a_i} = \frac{\partial J}{\partial z_1} \cdot \frac{\partial z_1}{\partial a_i} + \frac{\partial J}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_i}$$

$$\frac{\partial J}{\partial b_i} = \frac{\partial J}{\partial z_1} \cdot \frac{\partial z_1}{\partial b_i} + \frac{\partial J}{\partial z_2} \cdot \frac{\partial z_2}{\partial b_i}$$

$$g_1' = \frac{\partial J}{\partial z_1} = gc_1$$




$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

$$g_2' = \frac{\partial J}{\partial z_2} = gc_2$$

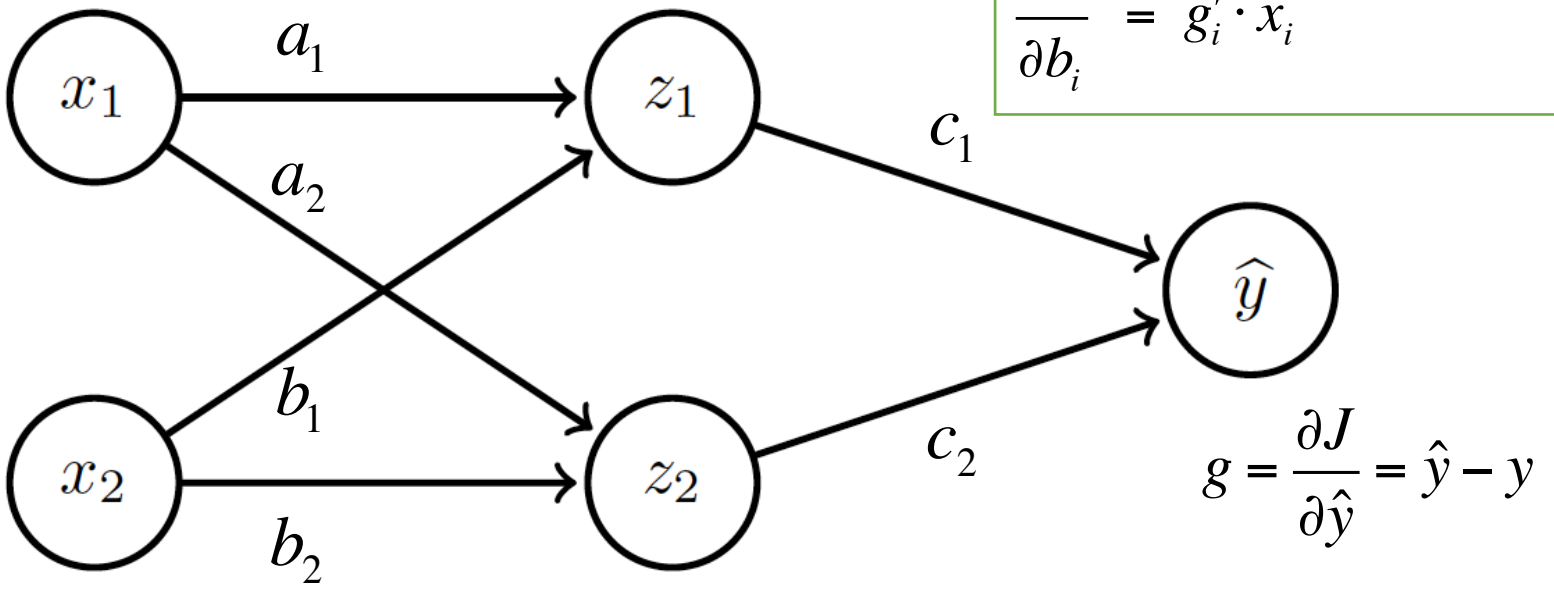
Backward prop: Compute derivatives w.r.t. weights $a_1$, $a_2$, $b_1$ and $b_2$

# Backprop: Example

$$\frac{\partial J}{\partial a_i} = g_i' \cdot x_i$$

$$\frac{\partial J}{\partial b_i} = g_i' \cdot x_i$$

$$g_1' = \frac{\partial J}{\partial z_1} = g c_1$$




$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

$$g_2' = \frac{\partial J}{\partial z_2} = g c_2$$

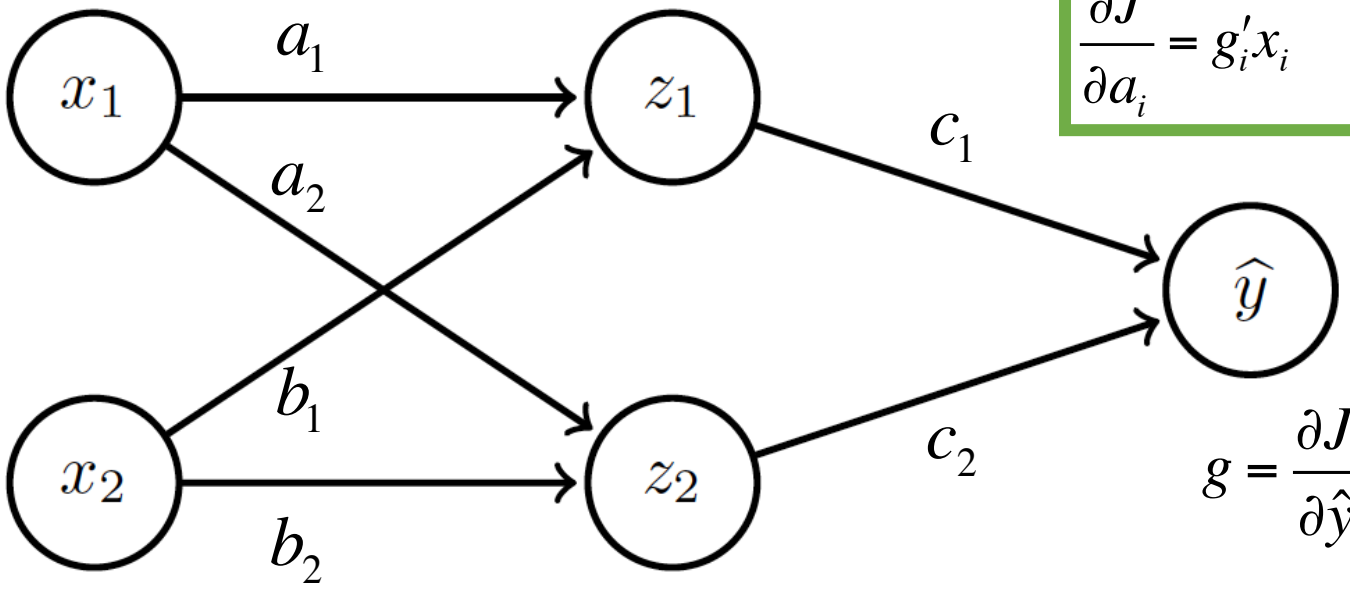
Backward prop: Compute derivatives w.r.t. weights $a_1$, $a_2$, $b_1$ and $b_2$

# Backprop: Example

$$\frac{\partial J}{\partial c_1} = g z_1 \qquad \frac{\partial J}{\partial c_2} = g z_2$$

$$\frac{\partial J}{\partial a_i} = g_i' x_i \qquad \frac{\partial J}{\partial b_i} = g_i' x_i$$

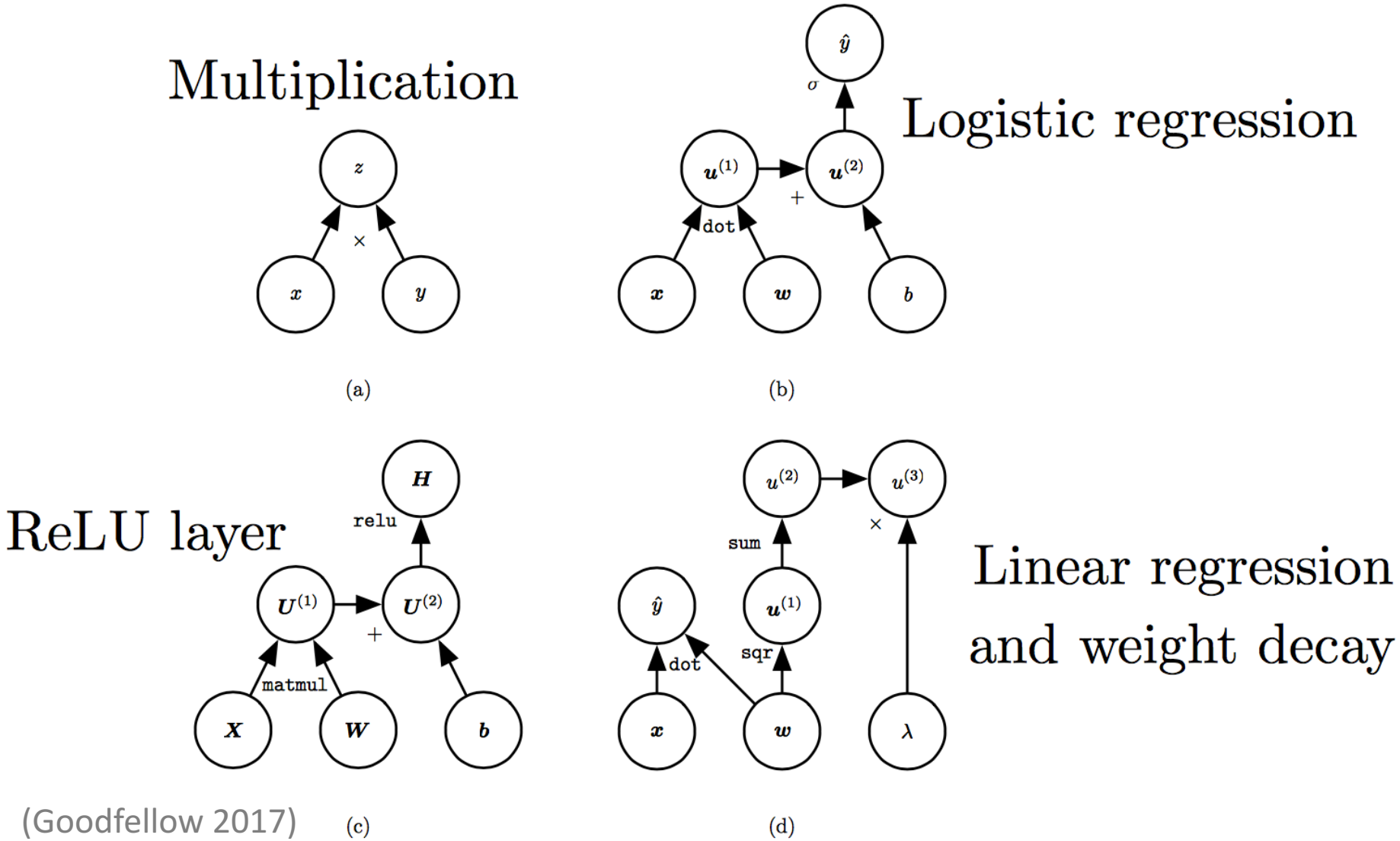
$$g_1' = \frac{\partial J}{\partial z_1} = g c_1$$




$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

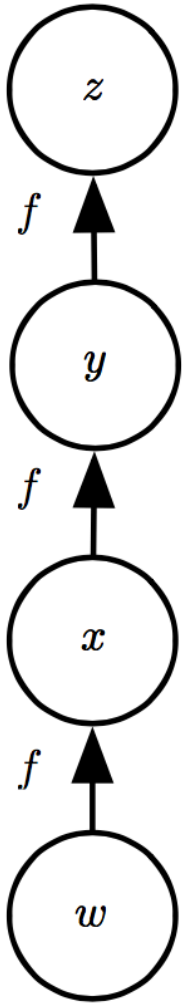
$$g_2' = \frac{\partial J}{\partial z_2} = g c_2$$

# Computation Graphs




(Goodfellow 2017)

# Repeated Sub-expressions




$$\frac{\partial z}{\partial w}$$
$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}$$
$$= f'(y)f'(x)f'(w)$$
$$= f'(f(f(w)))f'(f(w))f'(w)$$

Back-prop avoids computing this twice

(Goodfellow 2017)

# Backprop on Computation Graph

Maintain grad table

1: Initialize $\mathbf{g} \in R^n$ where $g_i$ denotes $\frac{\partial u^n}{\partial u^i}$

2: for $j = n-1$ to 1 do:

3: $$g_j = \sum_{i:j \in Pa(u^i)} g_i \frac{\partial u^i}{\partial u^j}$$

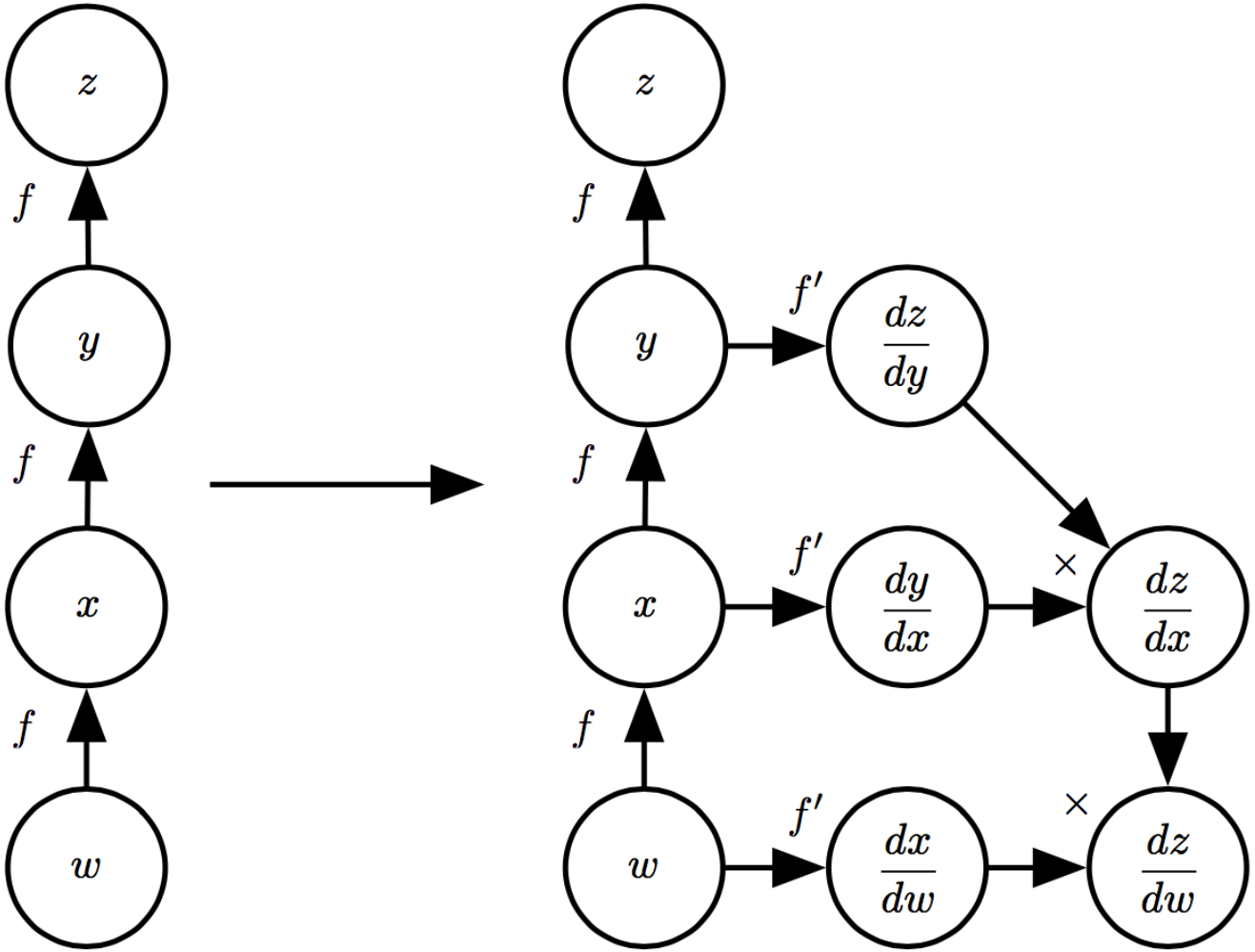
4: return $\mathbf{g}$

Parents of $u^i$

# Symbol-to-symbol Differentiation

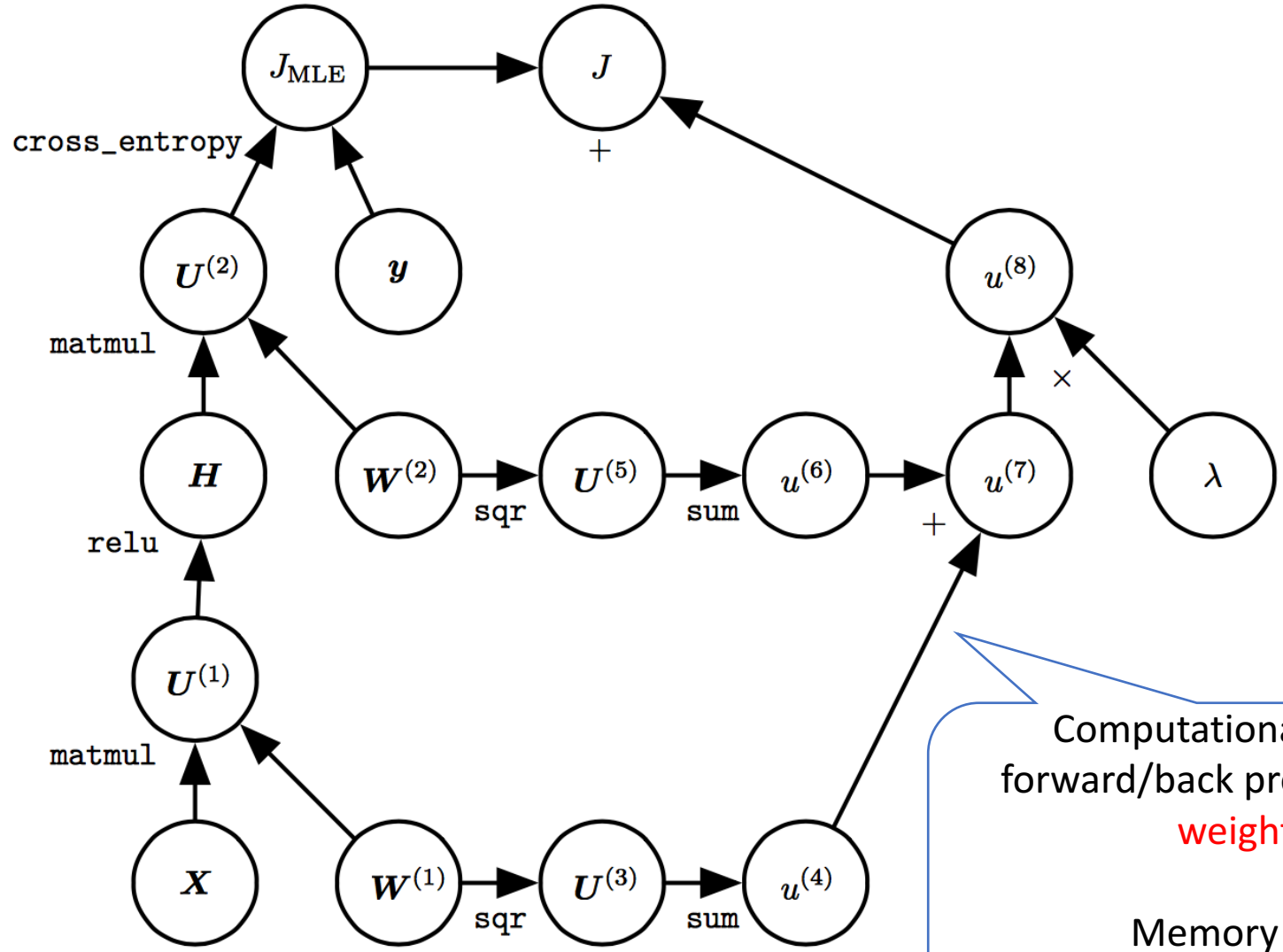
- Derivatives as computation graphs
  - Same language for both forward and back-propagation
- During execution, replace symbolic inputs with numeric value
- Used by Theano and TensorFlow
- Symbol-to-number differentiation: e.g. Torch and Caffe

# Symbol-to-symbol Differentiation




(Goodfellow et al. 2017)

# Training Feed-forward Nets




(Goodfellow et al. 2017)

Computational cost for forward/back prop: O(#num-weights)

Memory cost: O(#num-layers x minibatch-size)

# Regularization

Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error

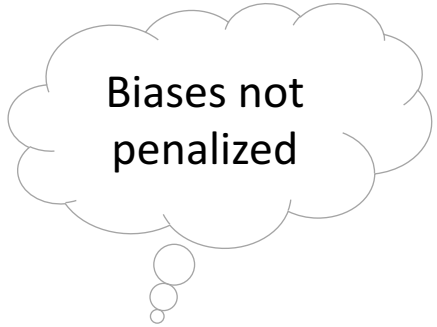
# Outline

- Norm Penalties
- Early Stopping
- Data Augmentation
- Bagging
- Dropout

# Norm Penalties

- Optimize:

$$J(\theta;\, X, y) \;+\; \alpha\, \Omega(\theta)$$



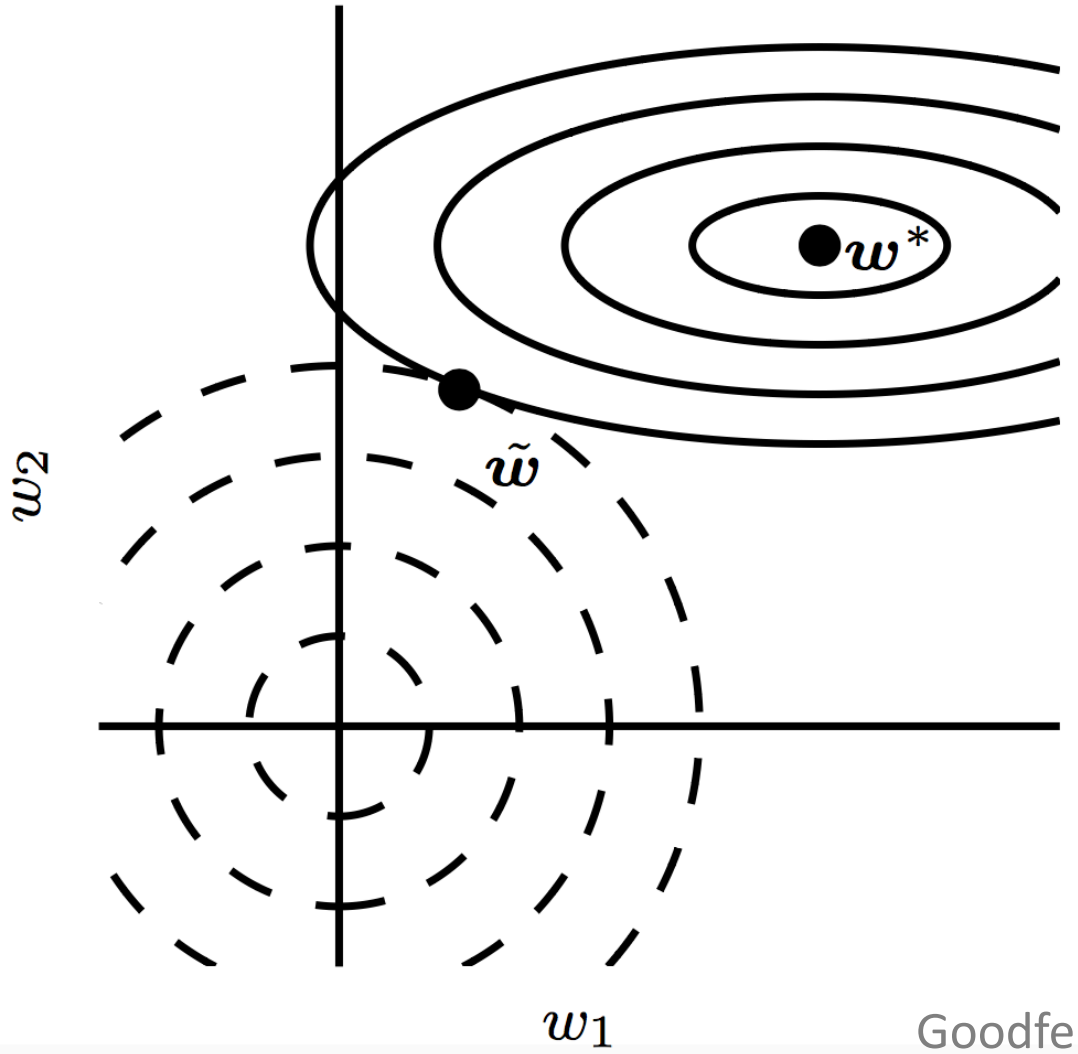

- $L_2$ regularization:
  - decays weights
  - MAP estimation with Gaussian prior

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

- $L_1$ regularization:
  - encourages sparsity
  - MAP estimation with Laplacian prior

$$\Omega(\theta) = \left\|\mathbf{w}\right\|_1$$

# $L_2$ Regularization



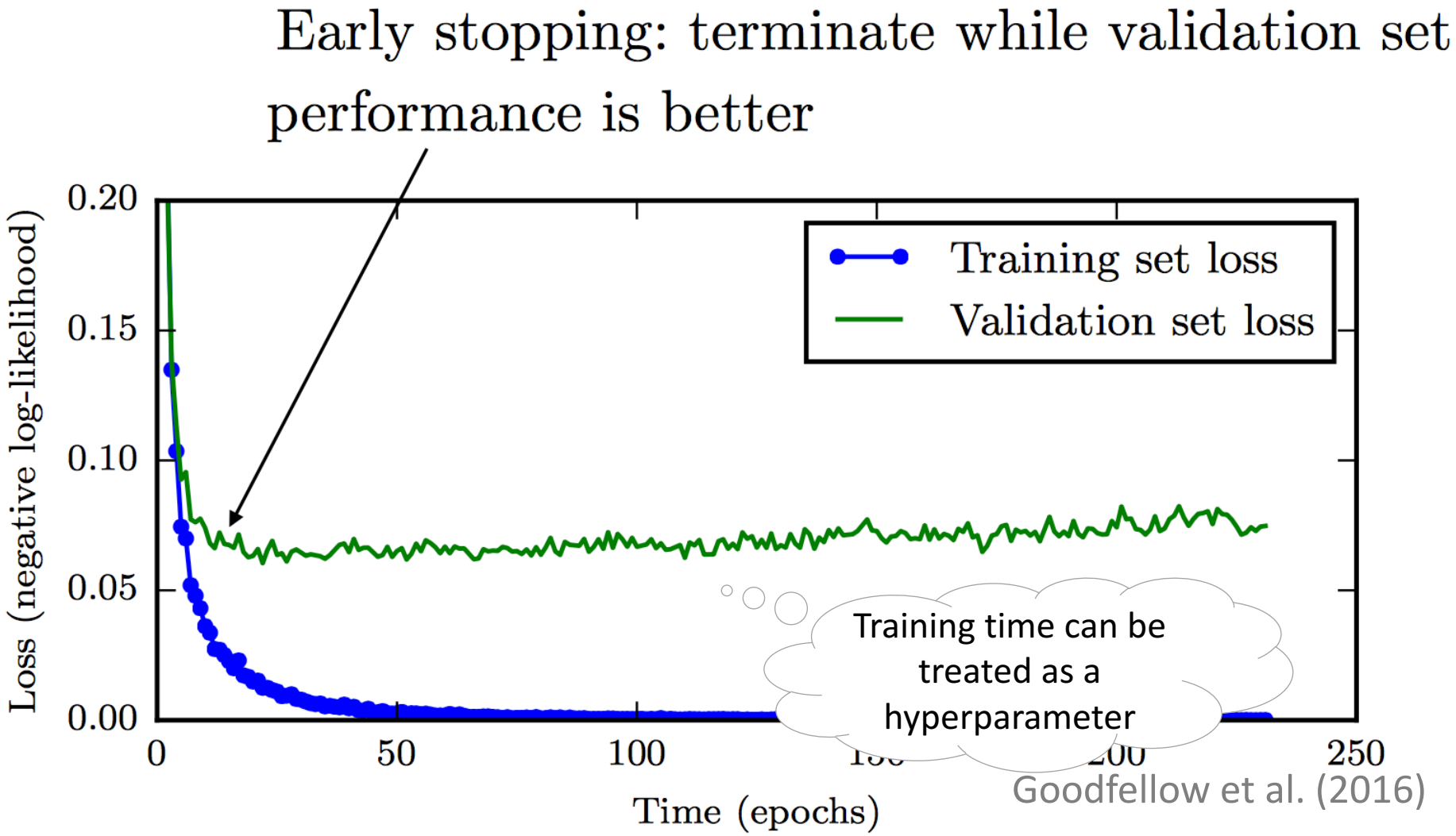

Goodfellow et al. (2016)

# Norm Penalties as Constraints

$$\min_{\Omega(\theta) \leq K} J(\theta; X, y)$$

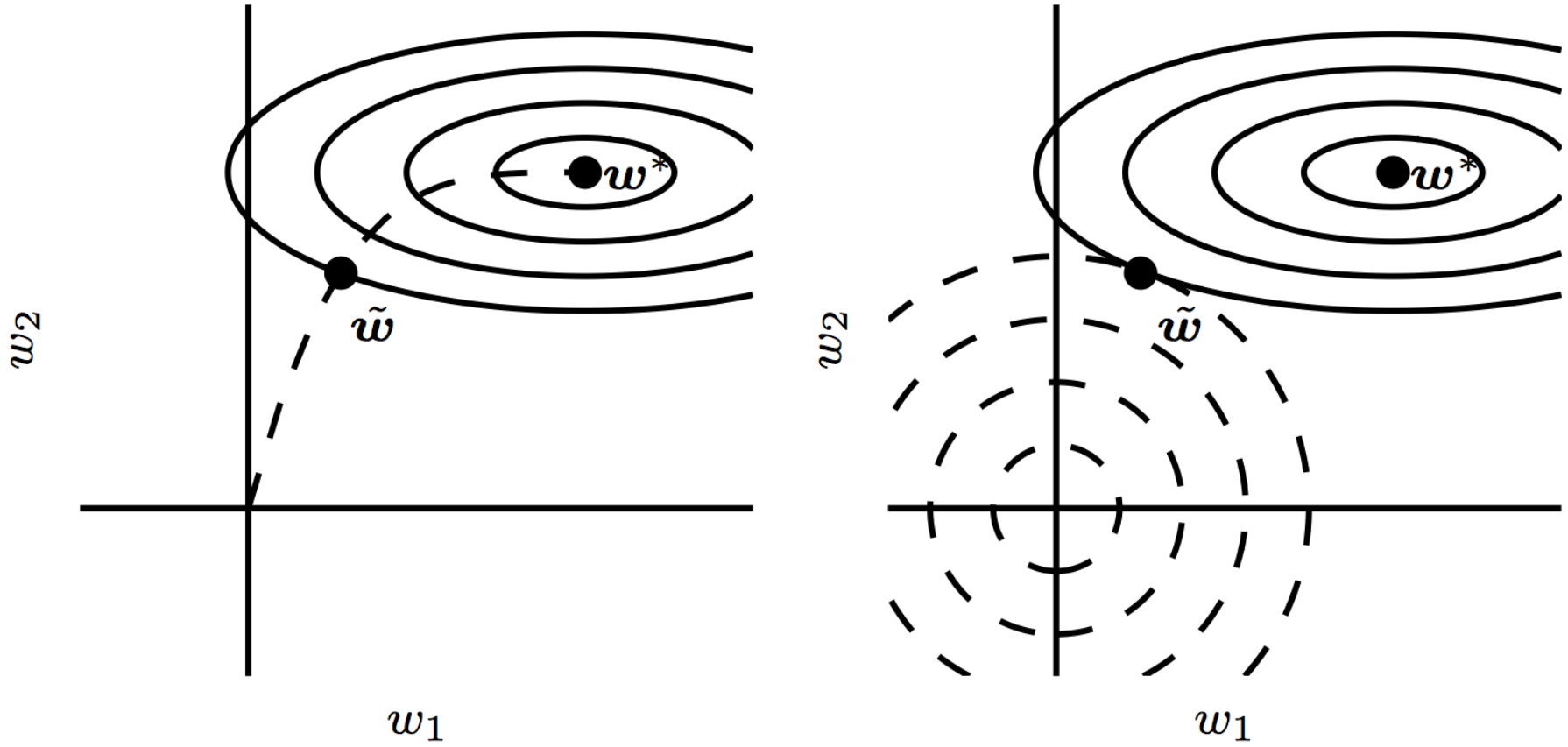
- Useful if *K* is known in advance
- Optimization:
  - Construct Lagrangian and apply gradient descent
  - Projected gradient descent

# Early Stopping




Goodfellow et al. (2016)

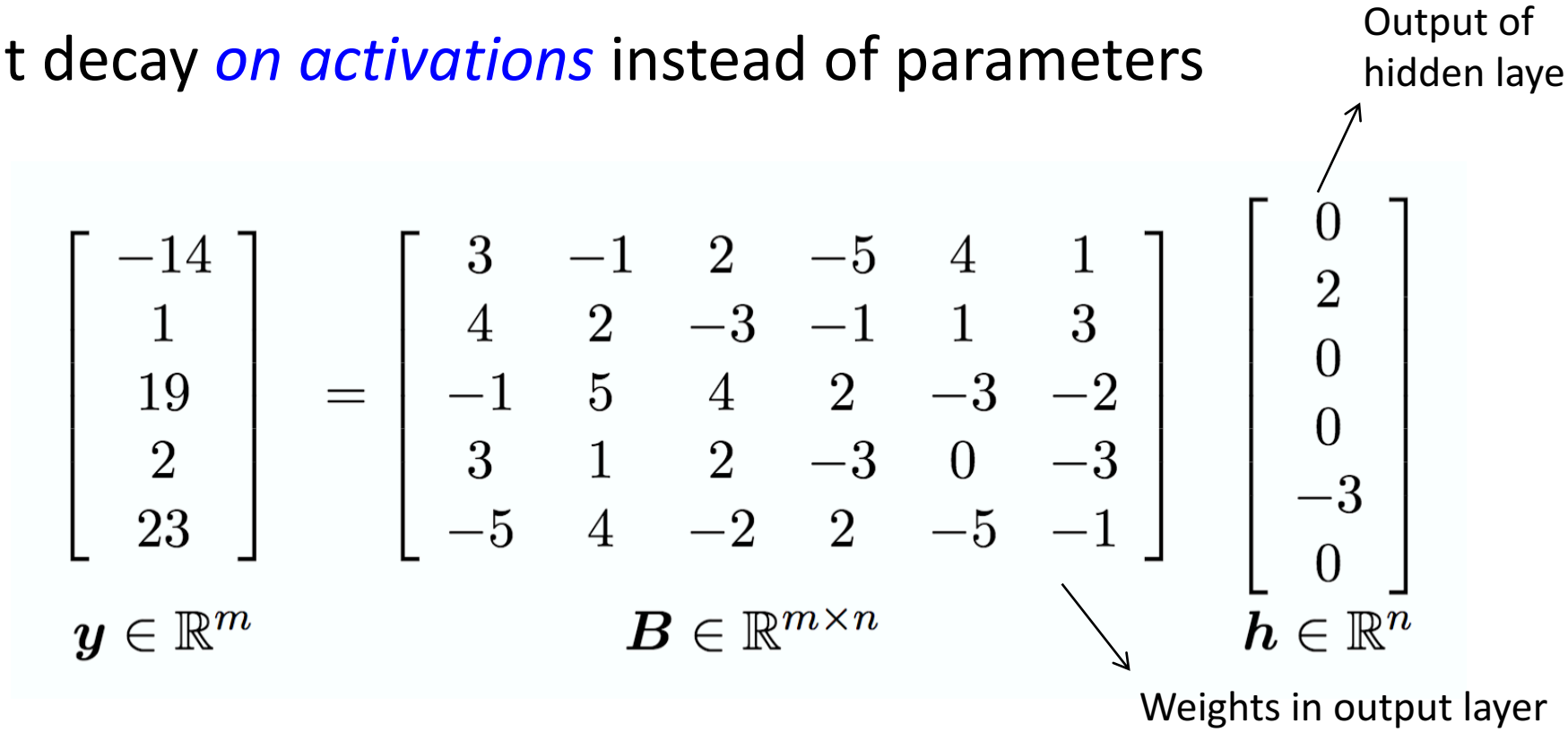
# Early Stopping ≈ Weight Decay




Goodfellow et al. (2016)

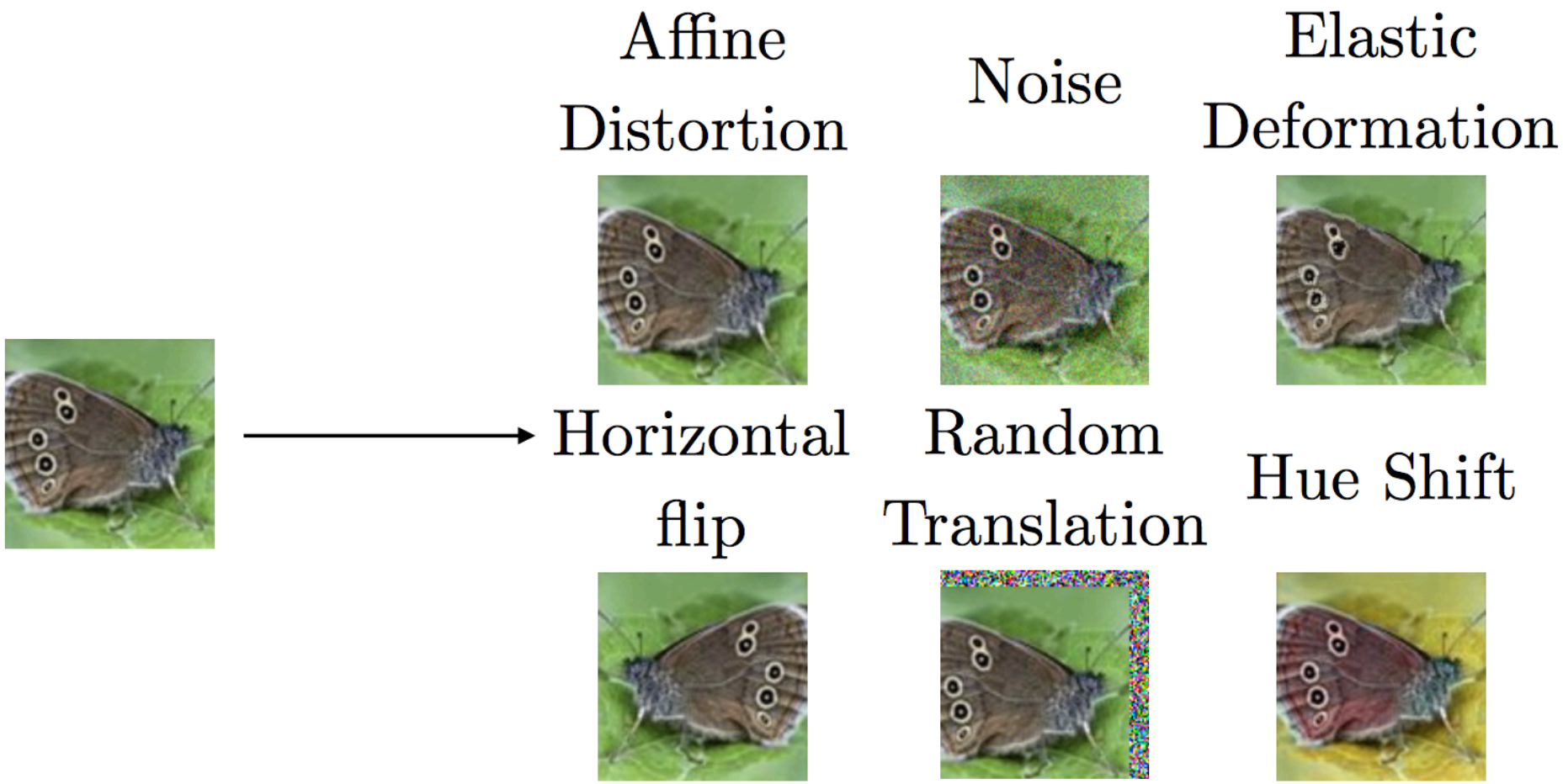
# Sparse Representations

- Weight decay *on activations* instead of parameters




$$J(\theta;\, X, y) \;+\; \alpha\, \Omega(h)$$

# Data Augmentation

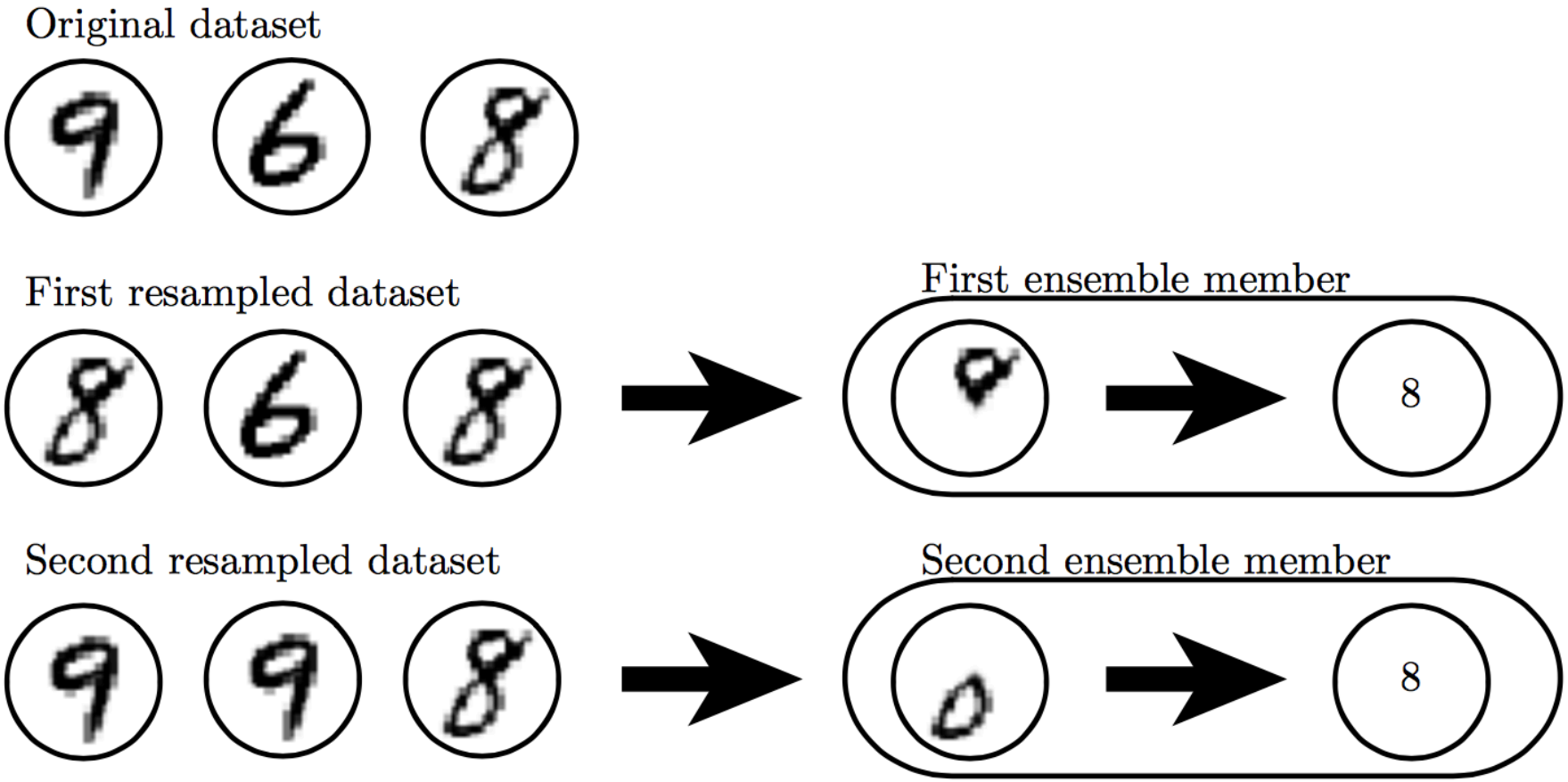



deeplearningbook.org

# Noise Robustness

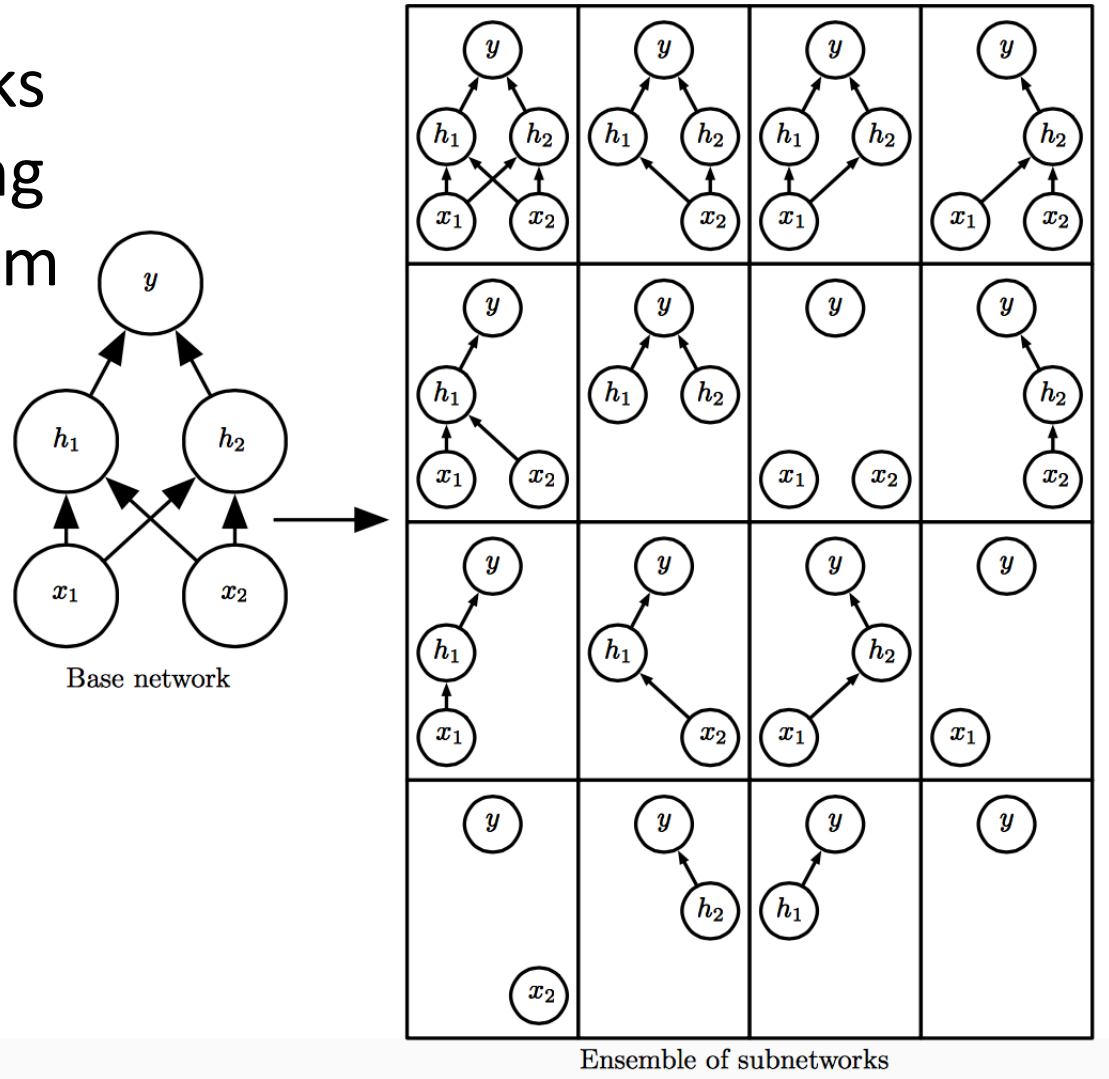
- Random perturbation of network weights
  - Gaussian noise: Equivalent to minimizing loss with regularization term
  - Encourages smooth function: small perturbation in weights leads to small changes in output $\mathbb{E}\left[\|\nabla_w y(x)\|\right]$
- Injecting noise in output labels
  - Better convergence: prevents pursuit of hard probabilities

# Bagging


Original dataset
First resampled dataset
First ensemble member
8
Second resampled dataset
Second ensemble member
8

# Dropout

Train all sub-networks obtained by removing non-output units from base network



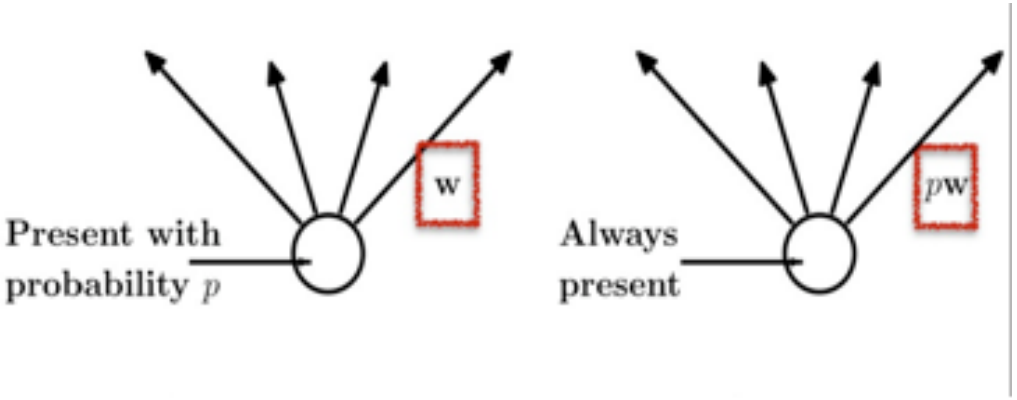

Goodfellow et al. (2016)

# Dropout: Stochastic GD

- For each new example/mini-batch:
  - Randomly sample a binary mask $\mu$ independently, where $\mu_i$ indicates if input/hidden node $i$ is included
  - Multiply output of node $i$ with $\mu_i$, and perform gradient update
- Typically, an input node is included with prob.0.8, hidden node with prob. 0.5

# Dropout: Weight Scaling

- During prediction time use all units, but scale weights with probability of inclusion




- Approximates the following inference rule:

$$\tilde{p}_{\text{ensemble}}(y \mid \boldsymbol{x}) = \sqrt[2^d]{\prod_{\boldsymbol{\mu}} p(y \mid \boldsymbol{x}, \boldsymbol{\mu})}$$

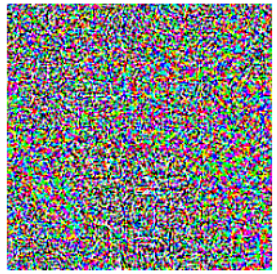
Cristina Scheau (2016)

# Adversarial Examples

$\boldsymbol{x}$

$y =$ "panda"
w/ 57.7%
confidence

$+ .007 \times$



$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

"nematode"
w/ 8.2%
confidence

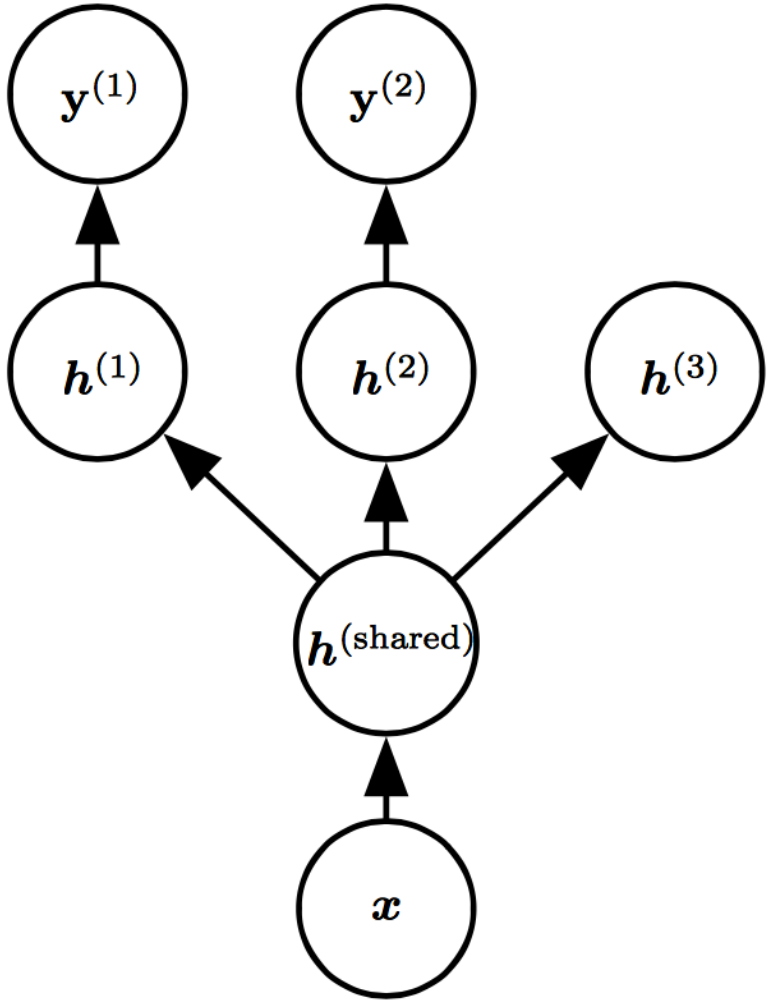
$=$

$\boldsymbol{x} +$
$\epsilon \, \text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

"gibbon"
w/ 99.3 %
confidence

Training on adversarial examples is mostly intended to improve security, but can sometimes provide generic regularization.

# Multi-task Learning




Goodfellow et al. (2016)

# Notebook

- https://github.com/cs109/2018-ComputeFest/