

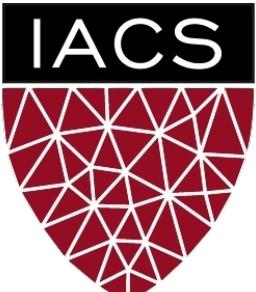
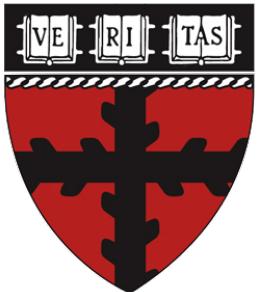


Compute Fest 2018

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



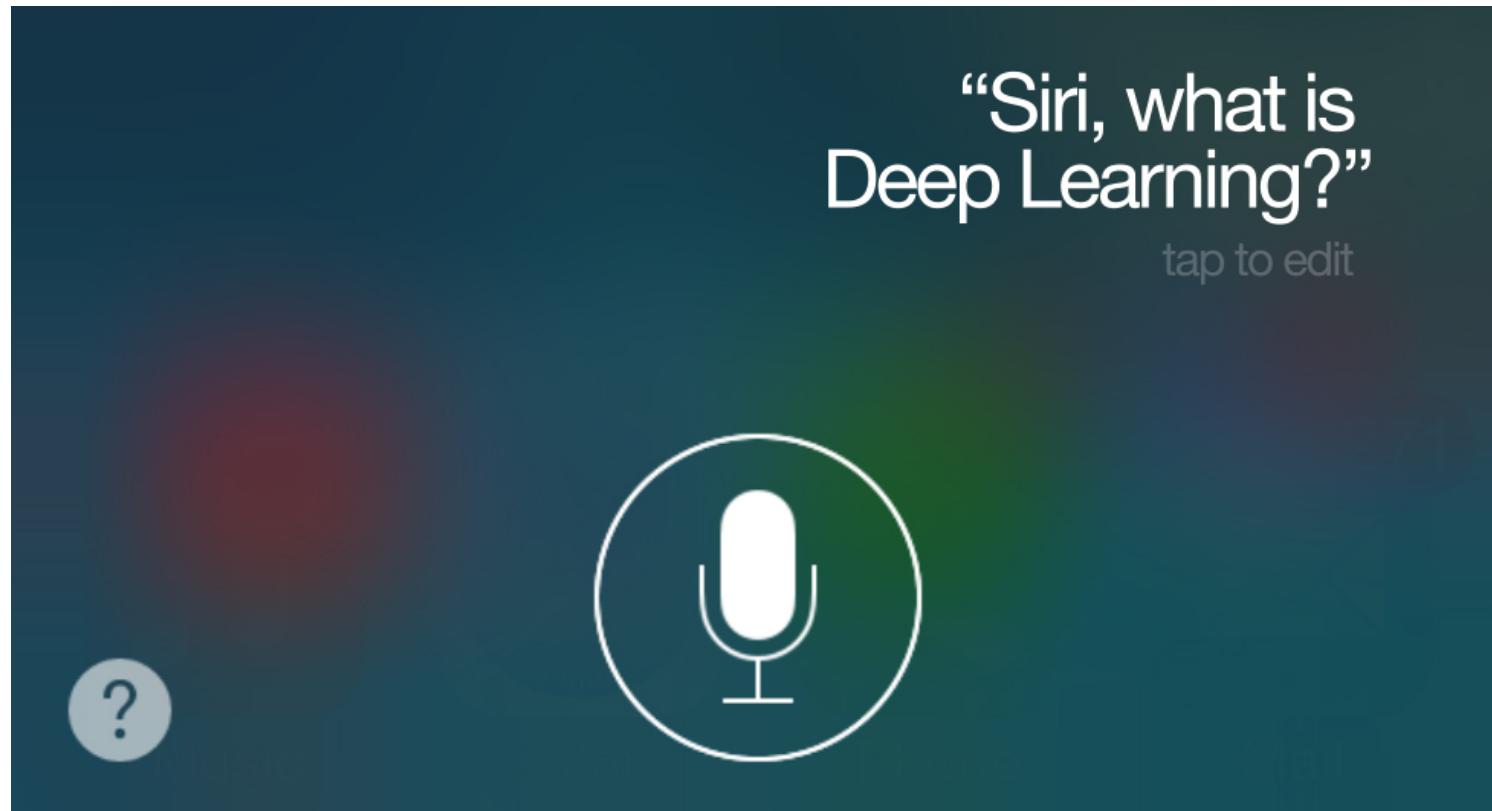
AlphaGo (2015)

First program to beat a professional Go player



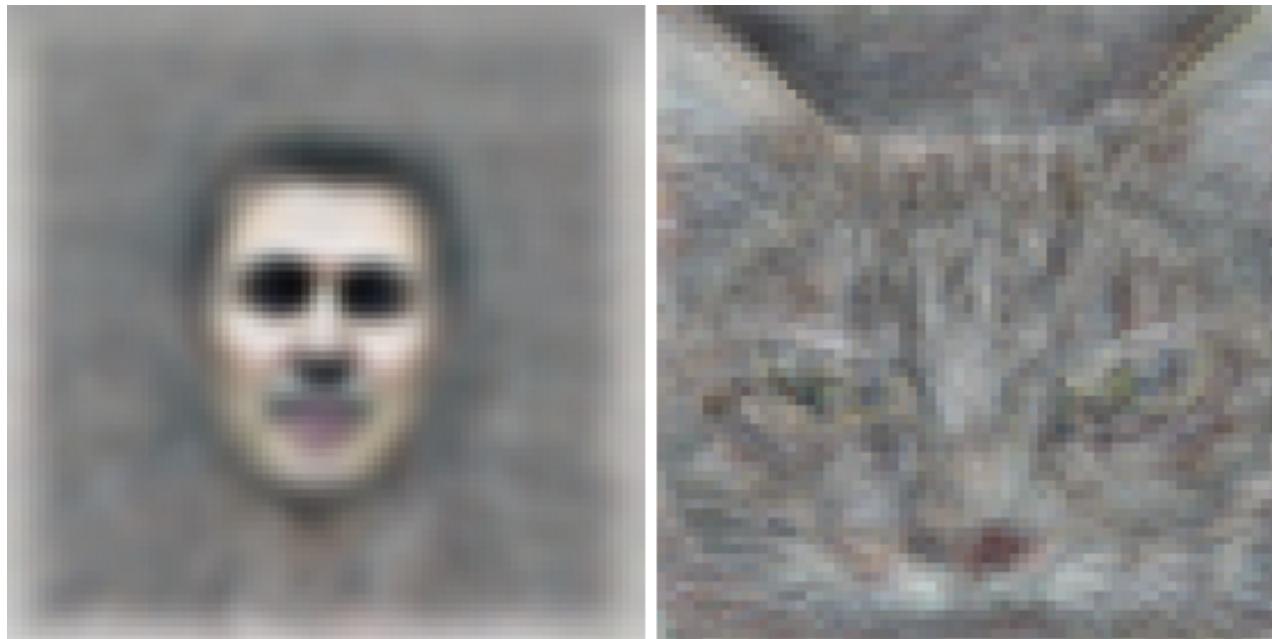
iOS Speech Synthesis (2016-)

Trained from 20 hours of high quality speech

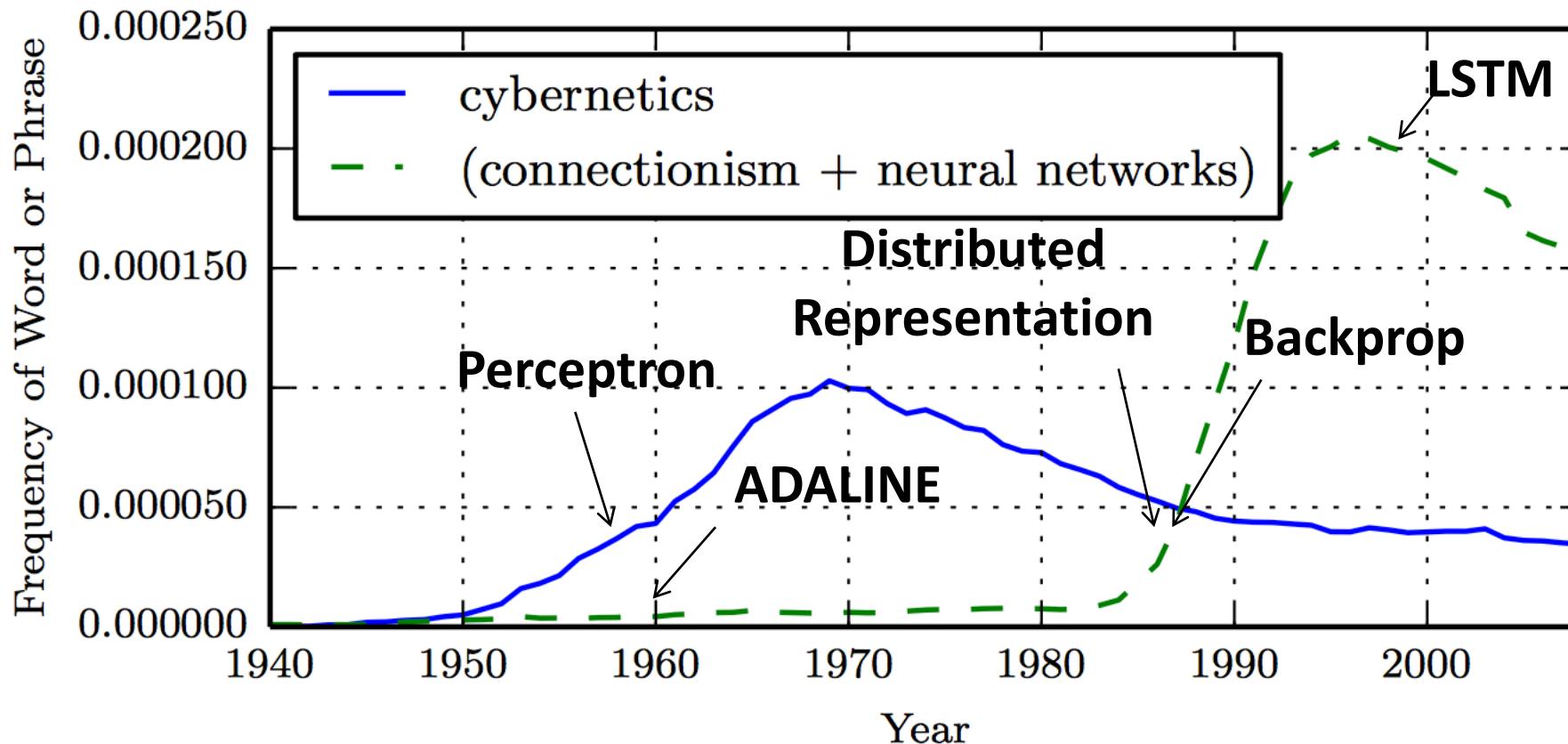


Google Brain (2012)

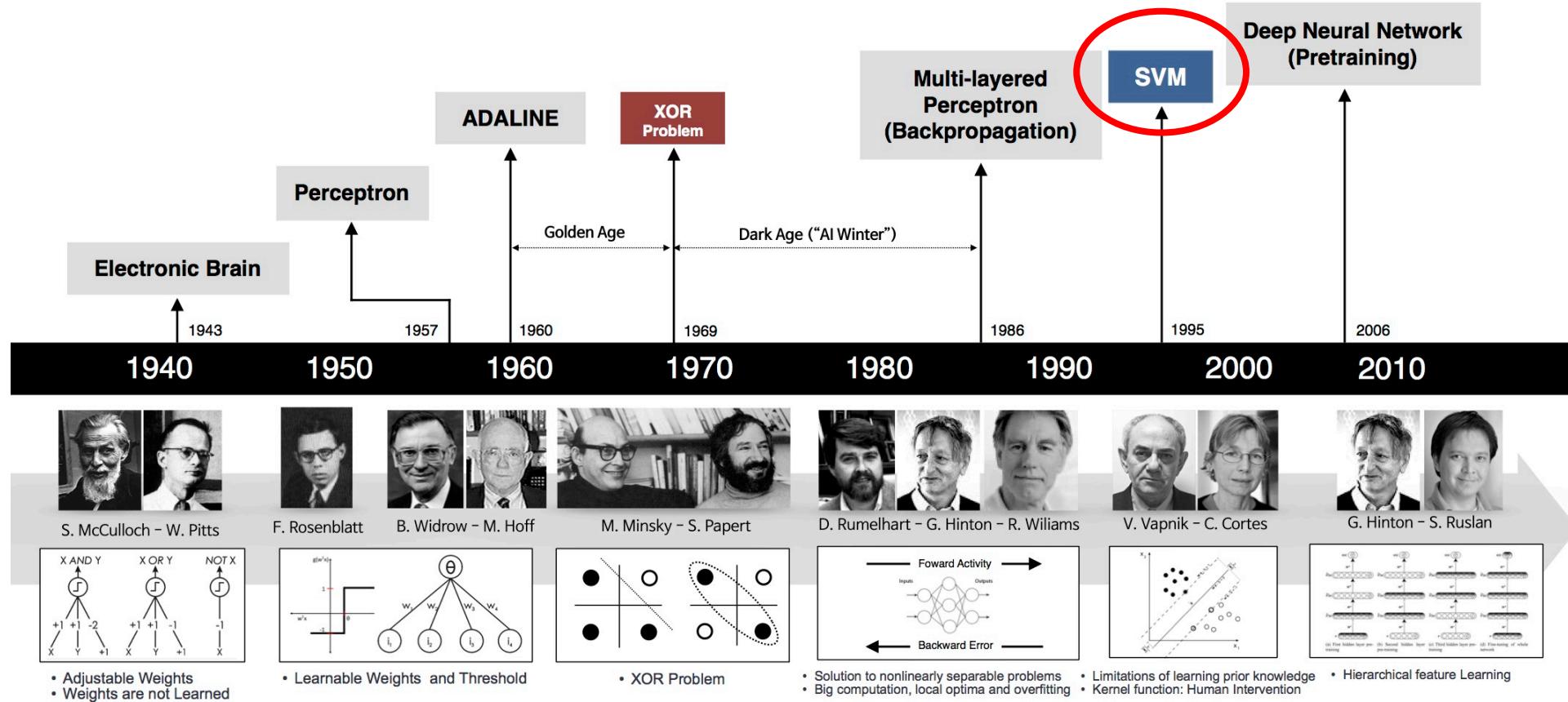
- Differentiate between human face and cat
 - Neural network with 1 billion connections
 - 10 million 200x200 pixel images from YouTube



Historical Trends

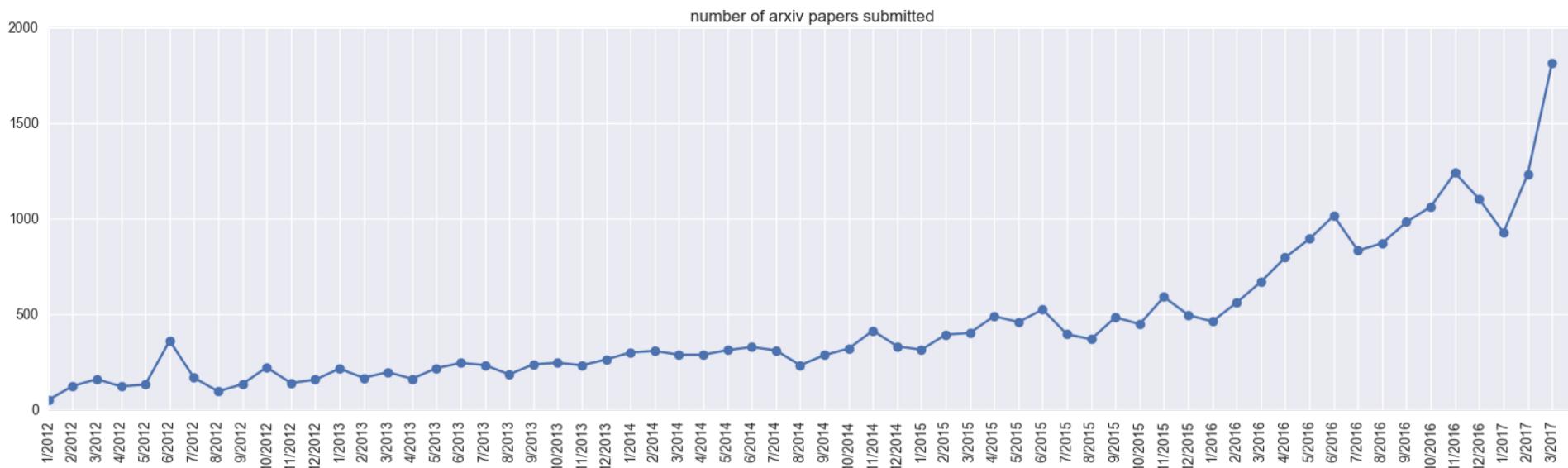


Historical Trends

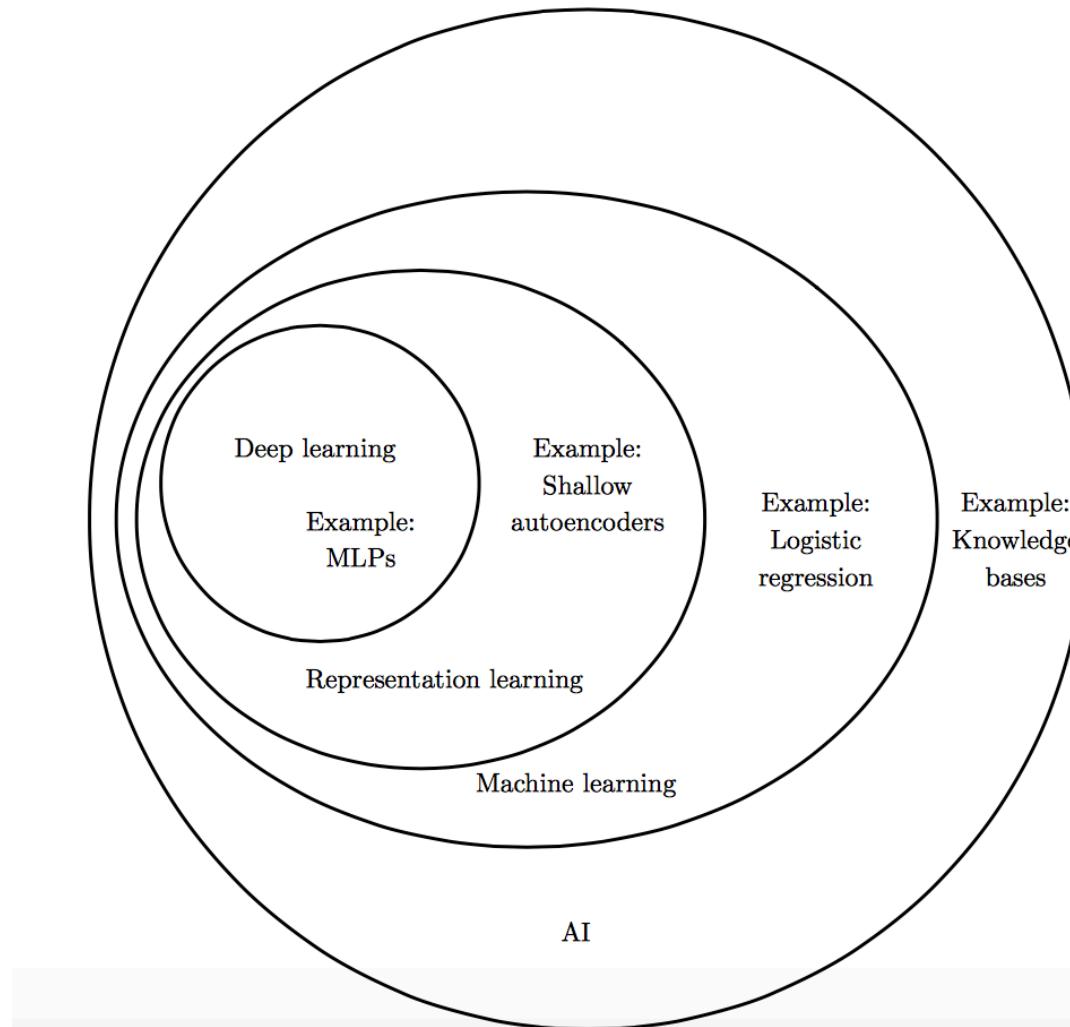


Historical Trends

ArXiv papers on deep learning: 2012-2017

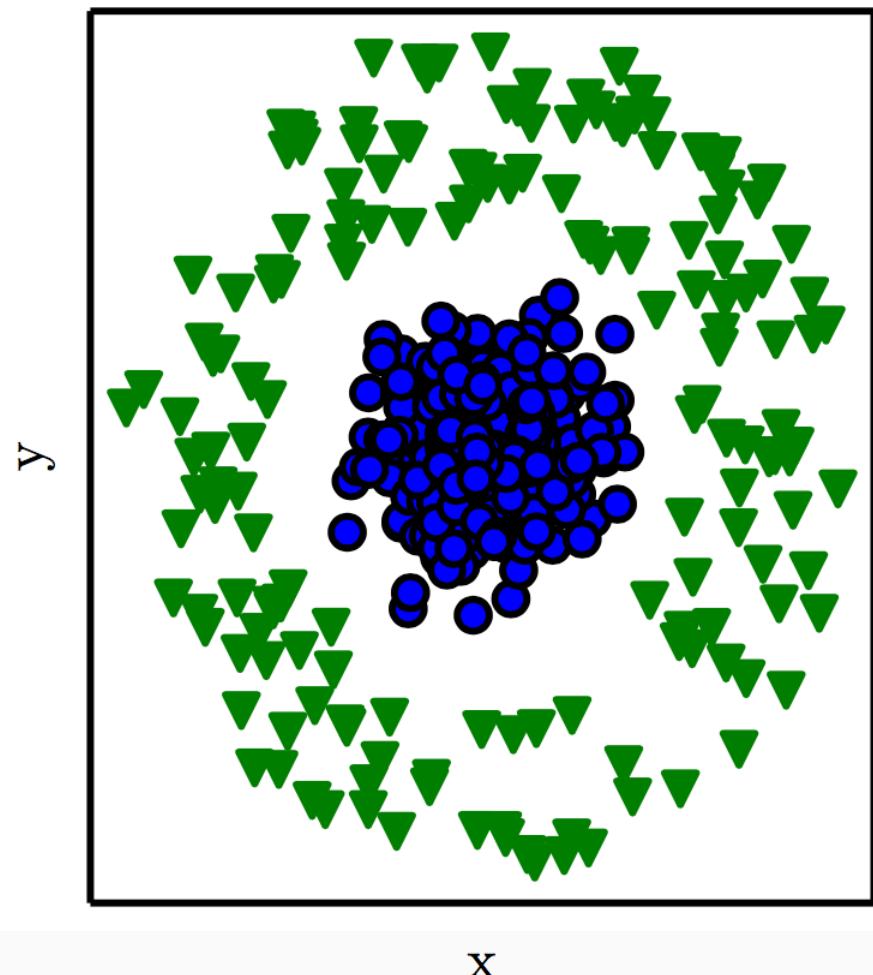


Deep Learning vs Classical ML

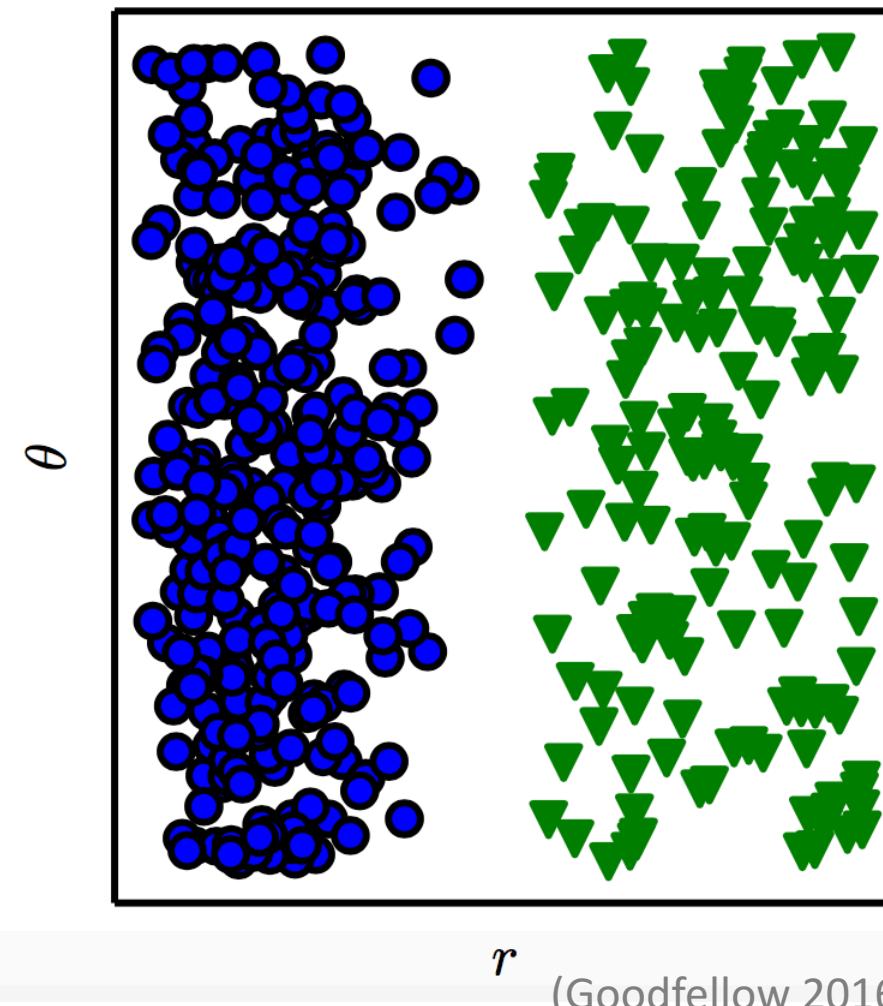


Representation Matters

Cartesian coordinates

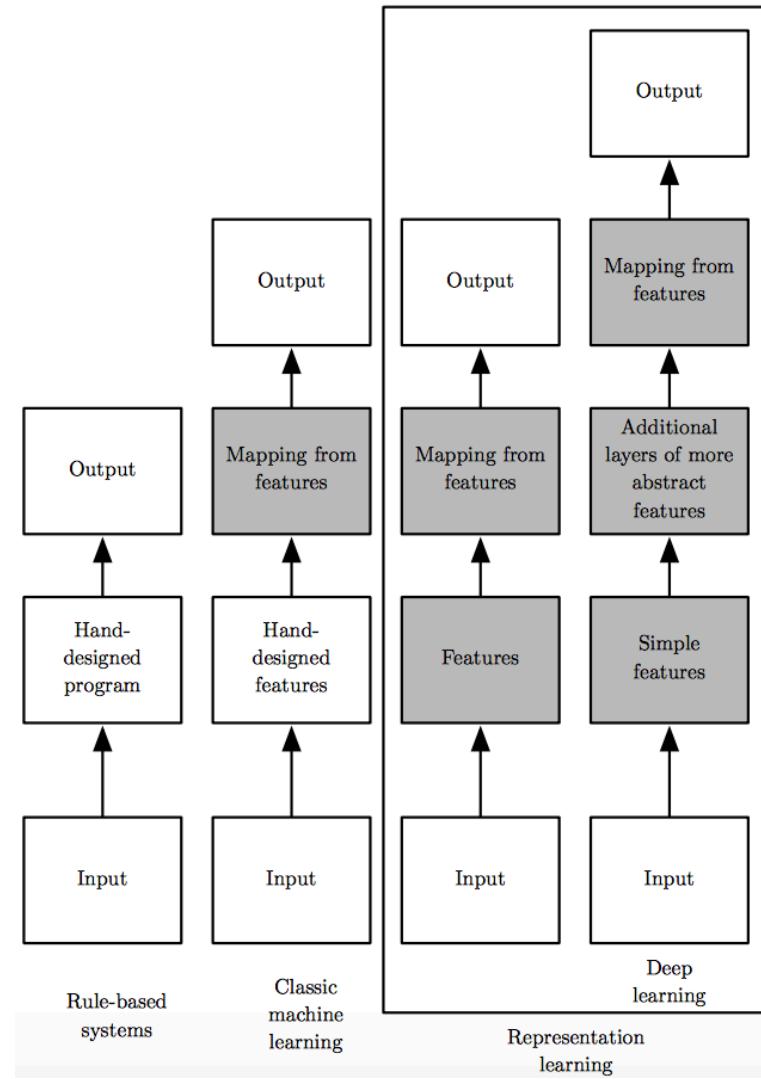


Polar coordinates



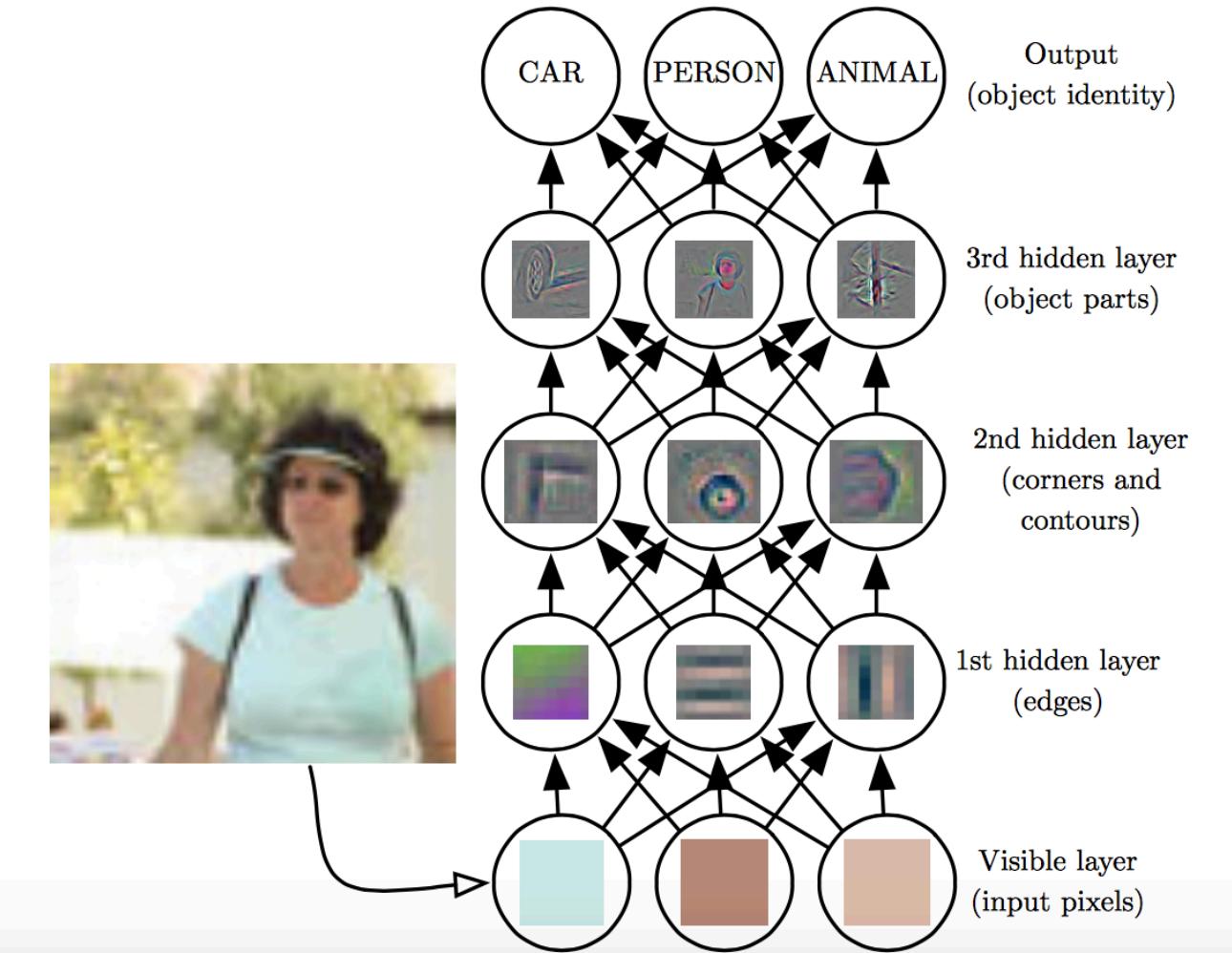
r
(Goodfellow 2016)

Learning Multiple Components



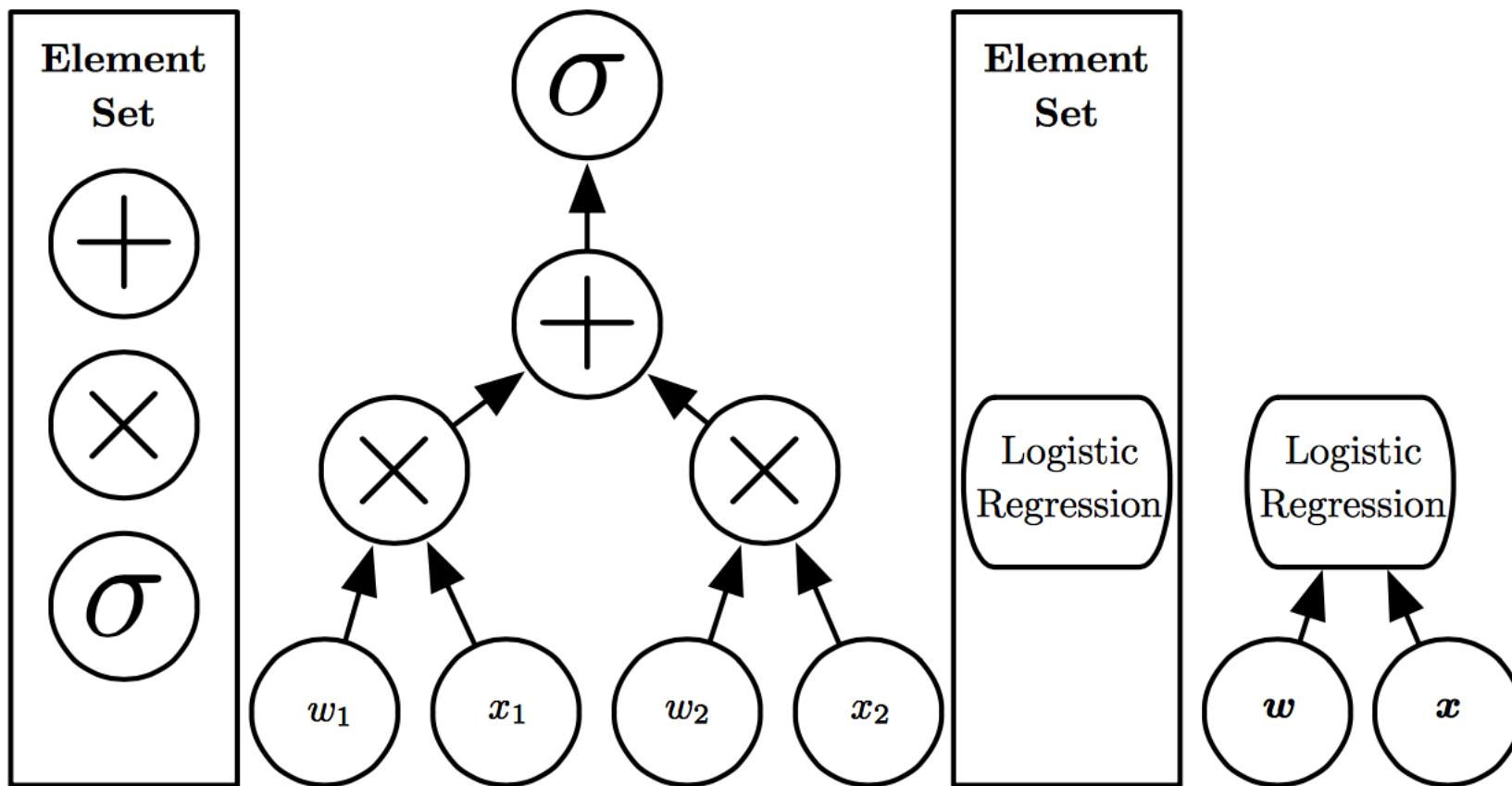
(Goodfellow 2016)

Depth = Repeated Compositions

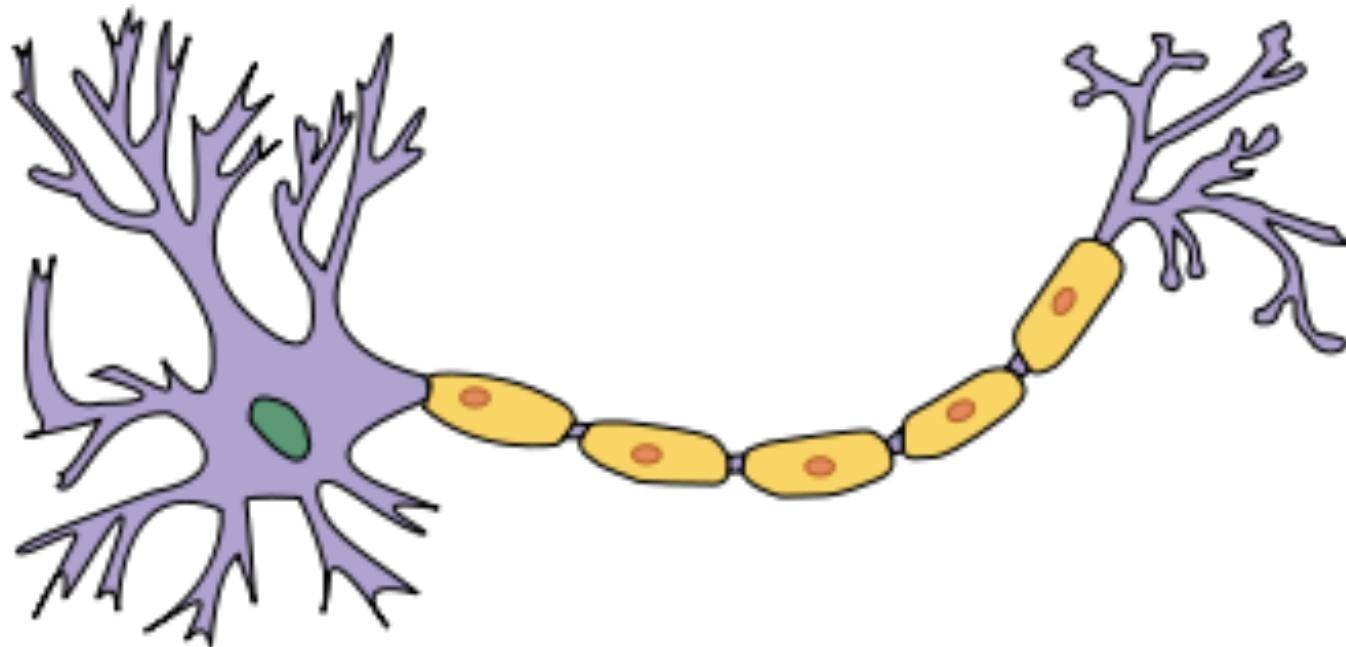


(Goodfellow 2016)

How to measure depth?



Is deep learning inspired by the brain?



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

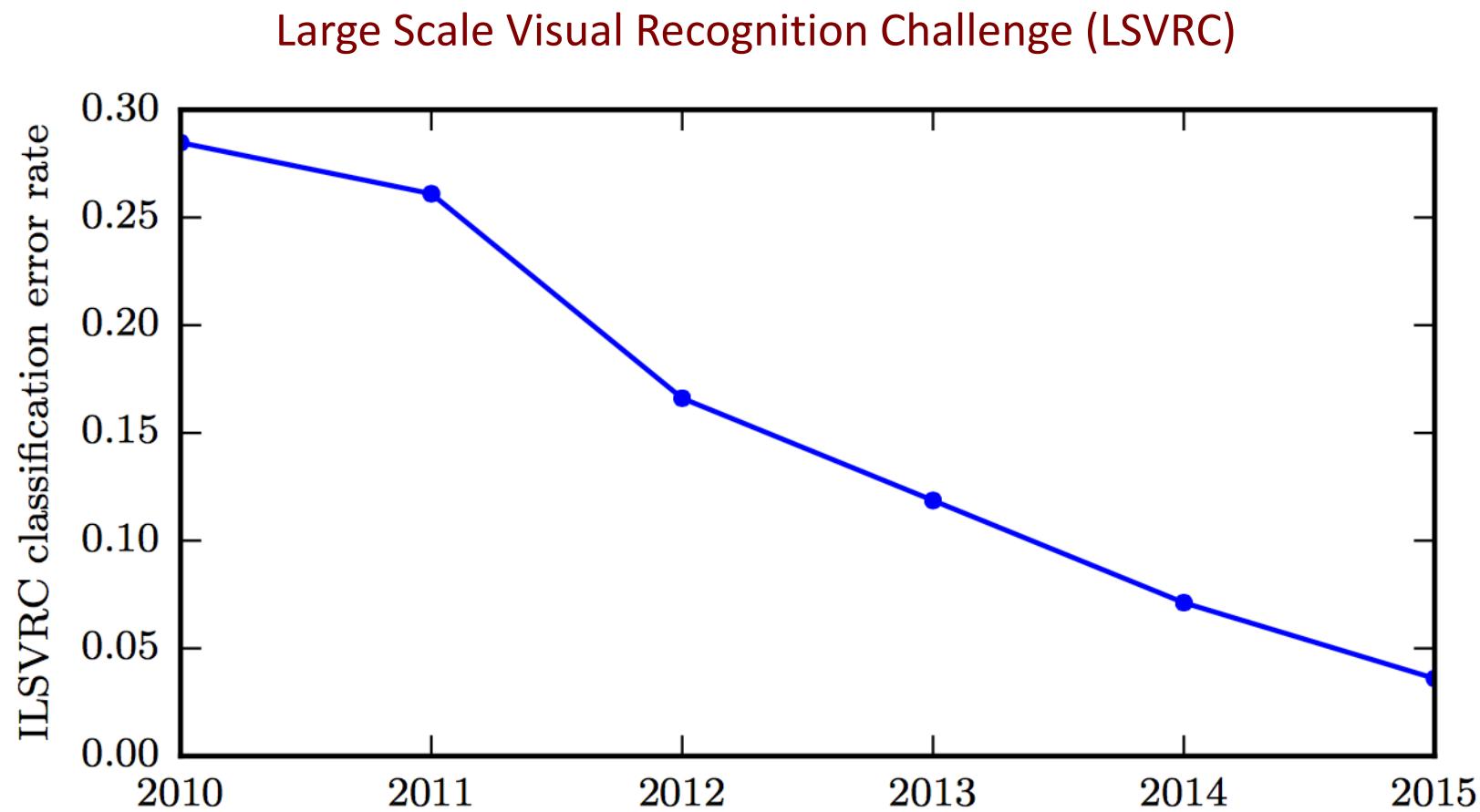
Why is deep learning so popular now?

- 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)

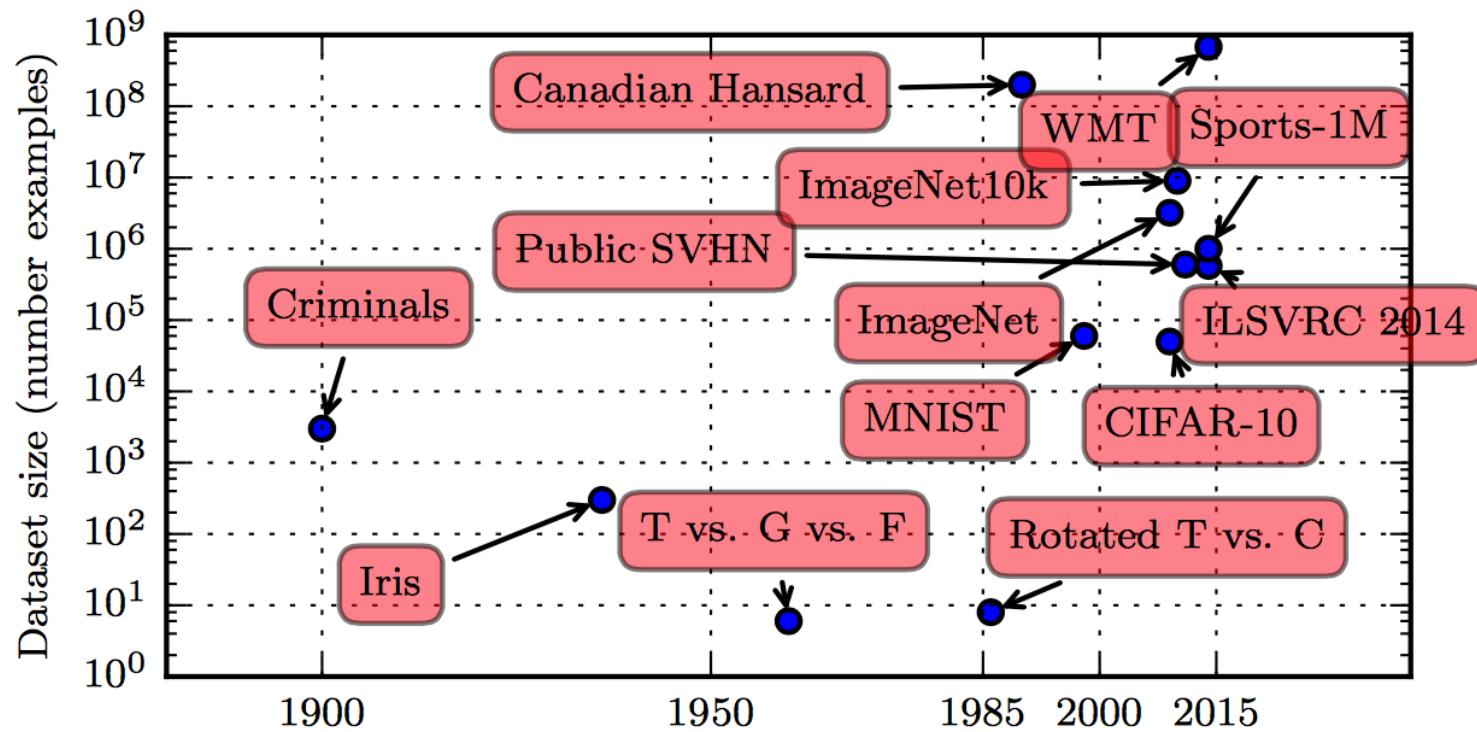
Increasing Accuracy



(Goodfellow 2016)

What changed?

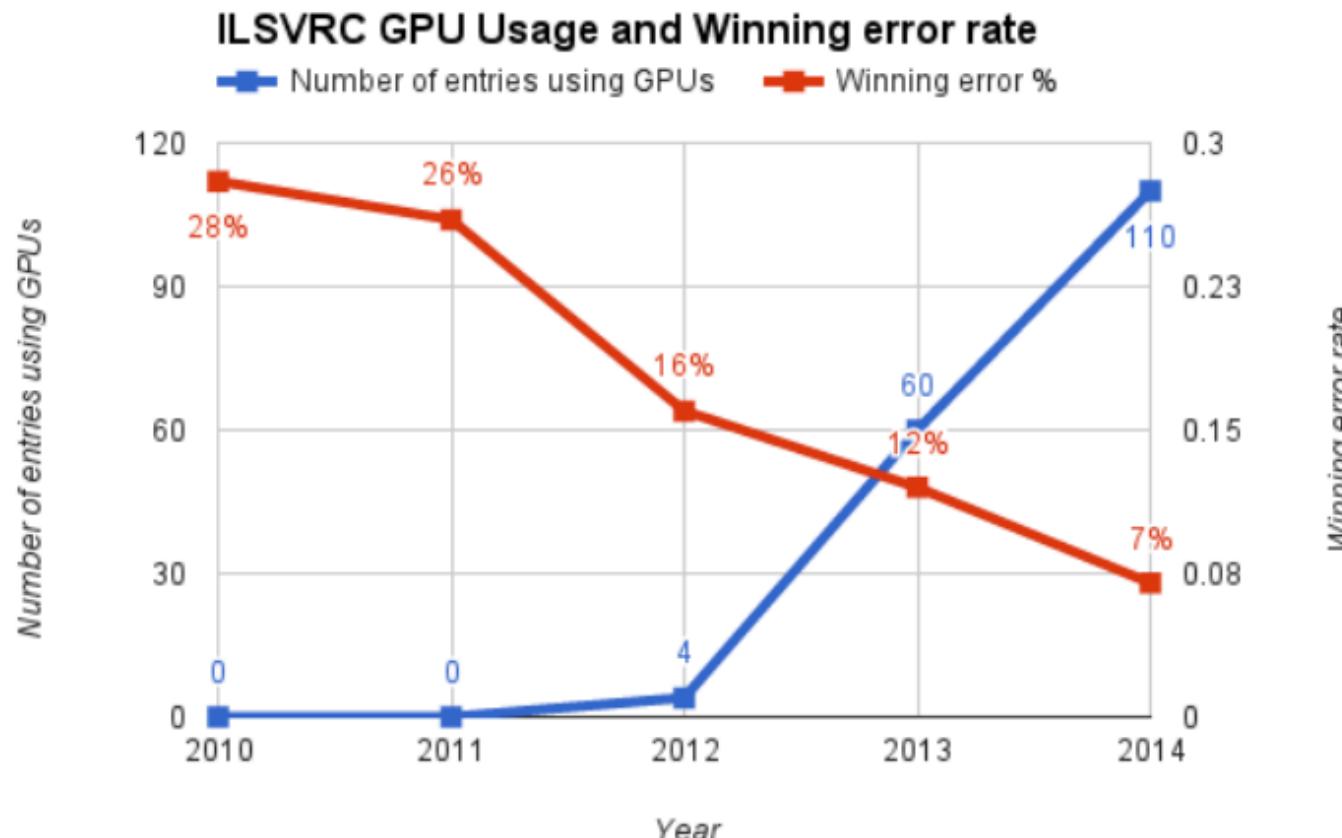
- Large, high-quality labeled data sets



(Goodfellow 2016)

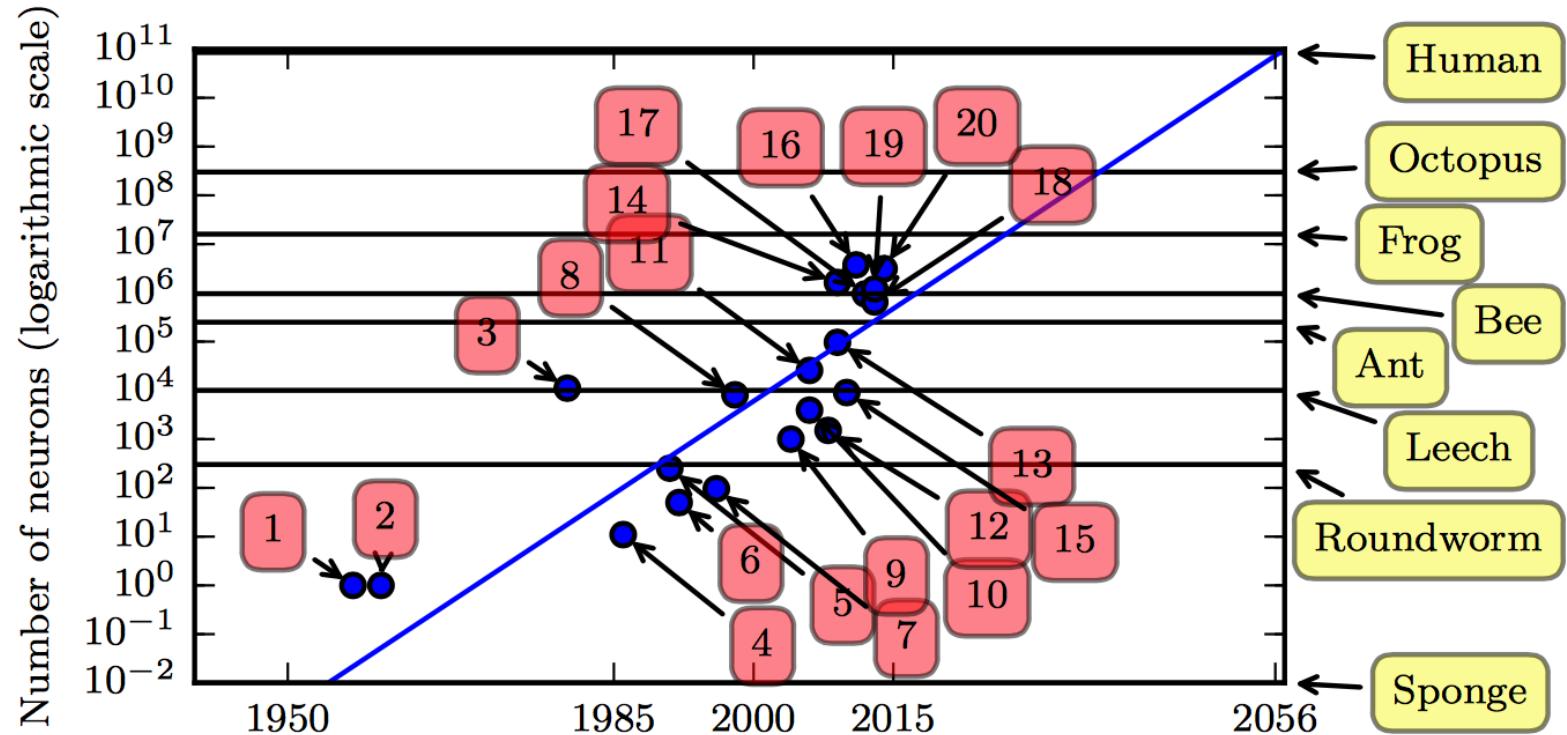
What changed?

- Massively parallel computing with GPUs



What changed?

- Better architectures, improved model complexity



(Goodfellow 2016)

What changed?

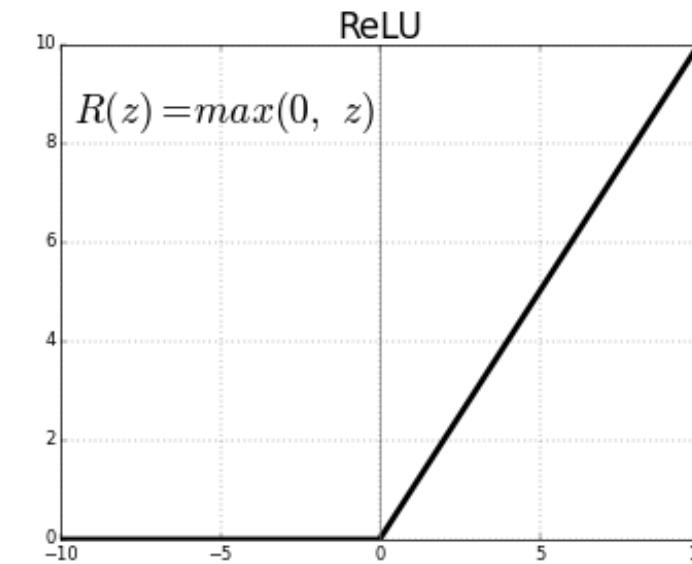
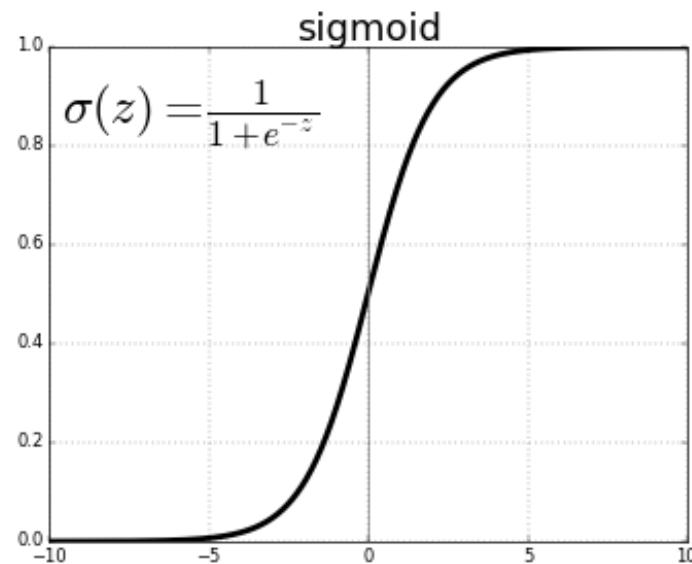
- New regularization techniques
 - Dropout
 - Batch normalization
 - Data augmentation

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

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

Non-linear transform

The diagram illustrates a non-linear transformation. It shows a blue arrow pointing from the input variable x to a blue bracket labeled "Non-linear transform". This bracket encompasses the term $\phi(x)$ in the equation. The output of this transformation is then multiplied by the weight vector w^T to produce the final output $f(x)$.

Traditional ML

- Manually engineer ϕ
 - 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?

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

Defined by
hidden layers of the
neural network

Can encode prior beliefs, generalizes well
But, *non-convex optimization*

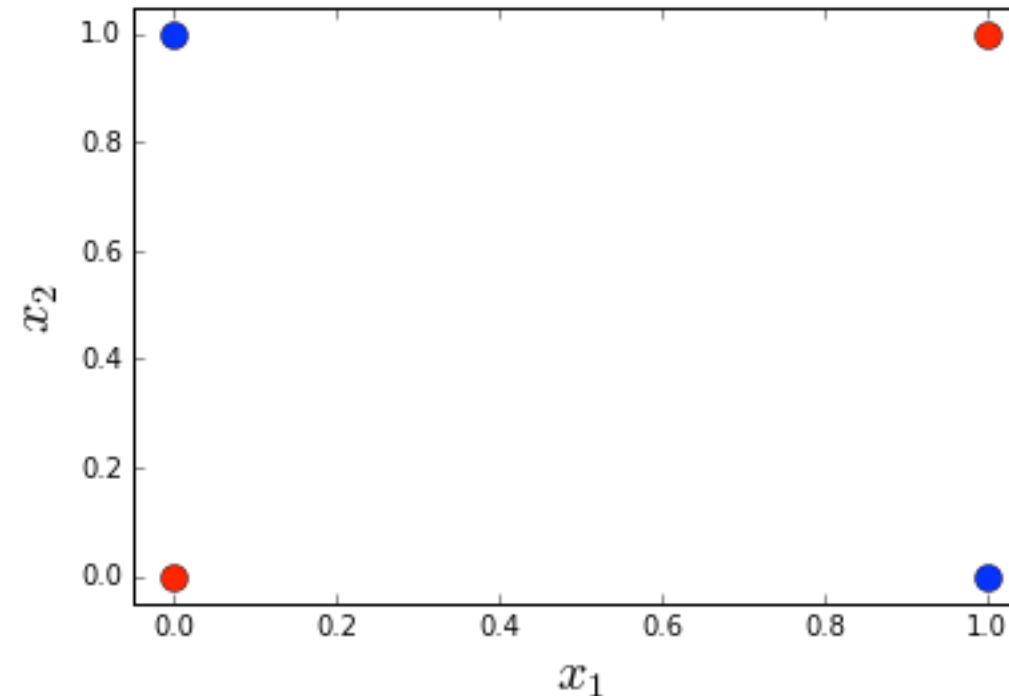
SVM vs. Neural Networks

- Hand-written digit recognition: MNIST data



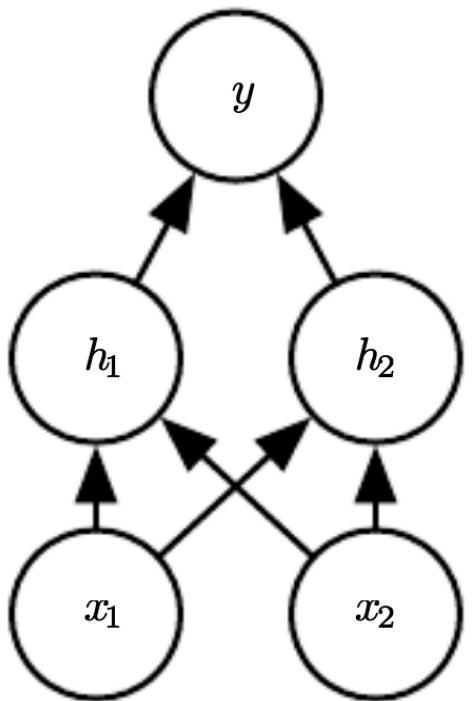
See illustration in notebook

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**)

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

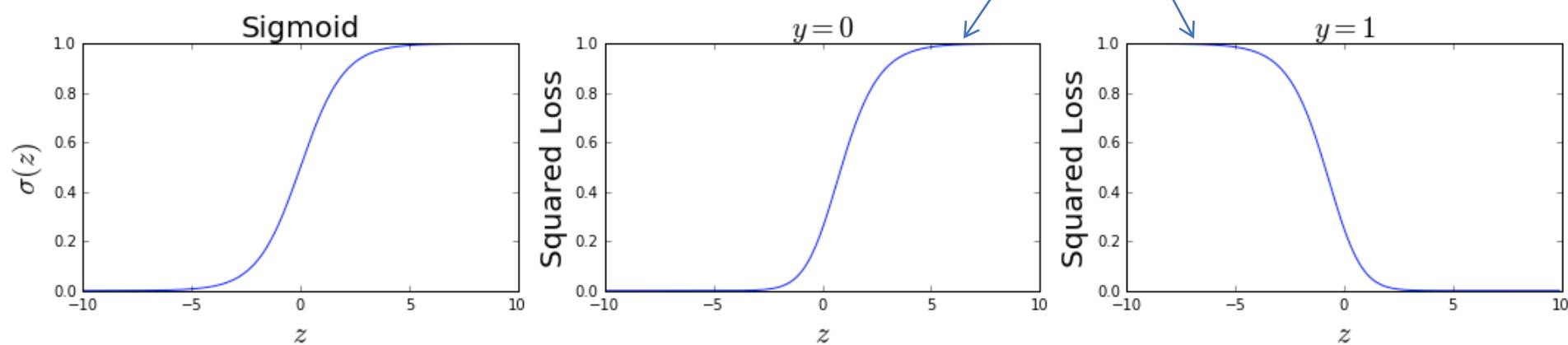
- Do not need to design separate cost functions
- Gradient of cost function must be large enough

Cost Function

- Example: sigmoid output + squared loss

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

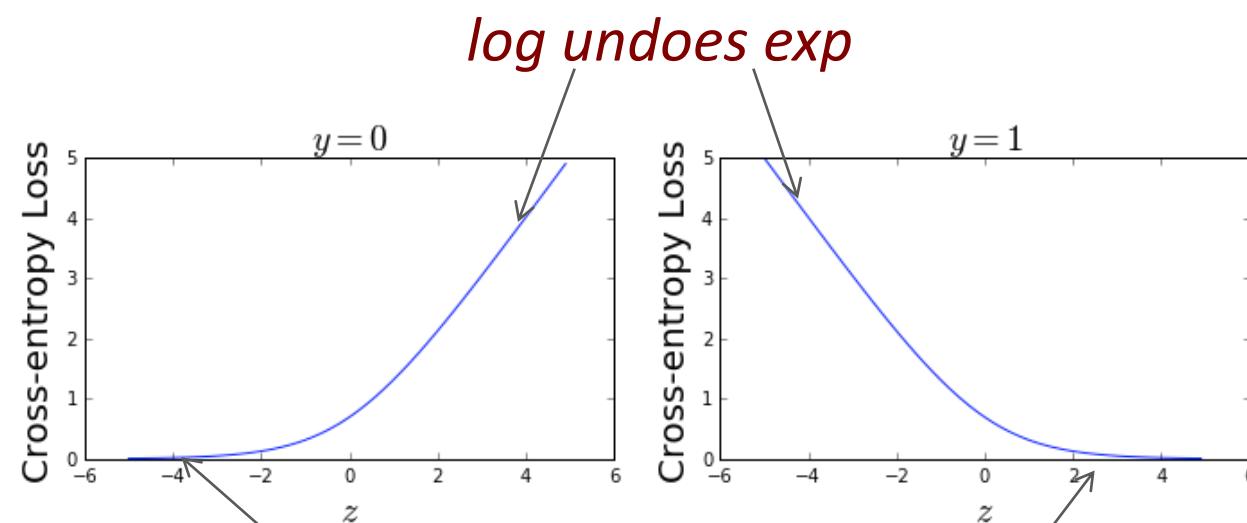
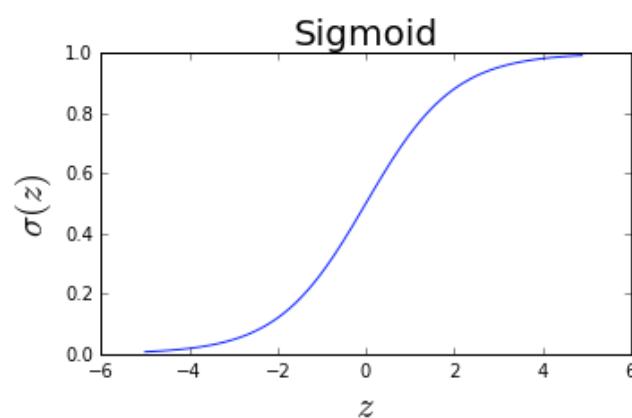
$$L_{sq}(y, z) = (y - \sigma(z))^2$$



Cost Function

- Example: sigmoid output + cross-entropy loss

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



Saturates only when the model makes correct predictions

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

Softmax Output

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

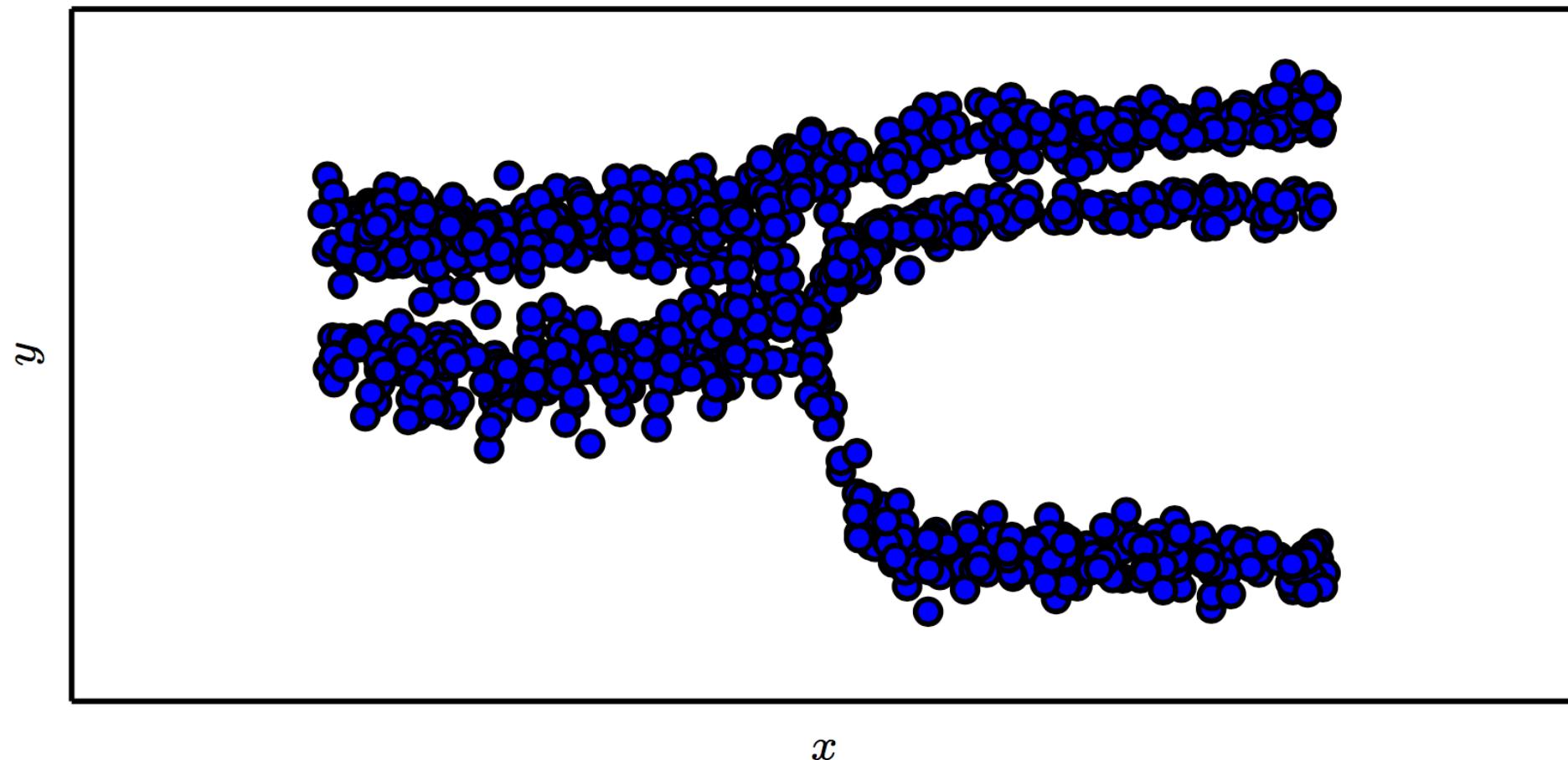
$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- Log-likelihood undoes exp

$$\begin{aligned}\log \text{softmax}(z)_i &= z_i - \log \sum_j \exp(z_j) \\ &\approx z_i - \max_j z_j\end{aligned}$$

(Score to target label – Maximum score)

Mixture Density Output



(Goodfellow 2017)

Design Choices

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

Hidden Units

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

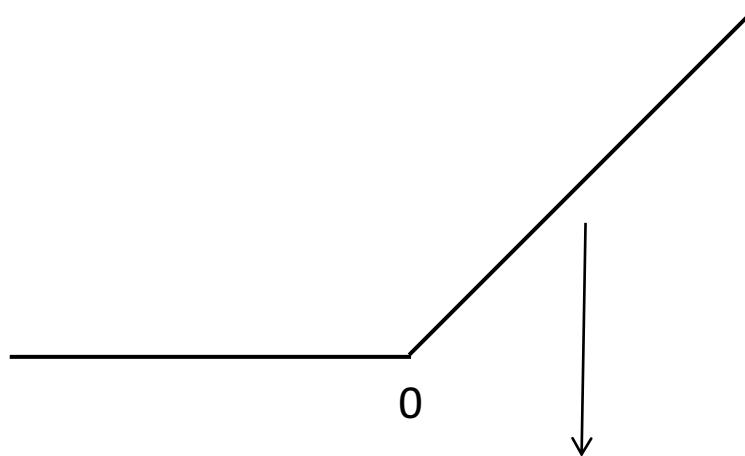
with activation function g

- 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 $z < 0$

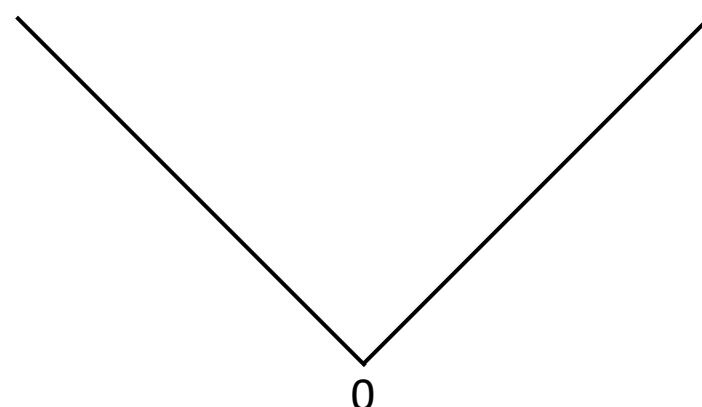
Generalized ReLU

- Generalization: For $\alpha_i > 0$

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

- E.g. Absolute value ReLU:

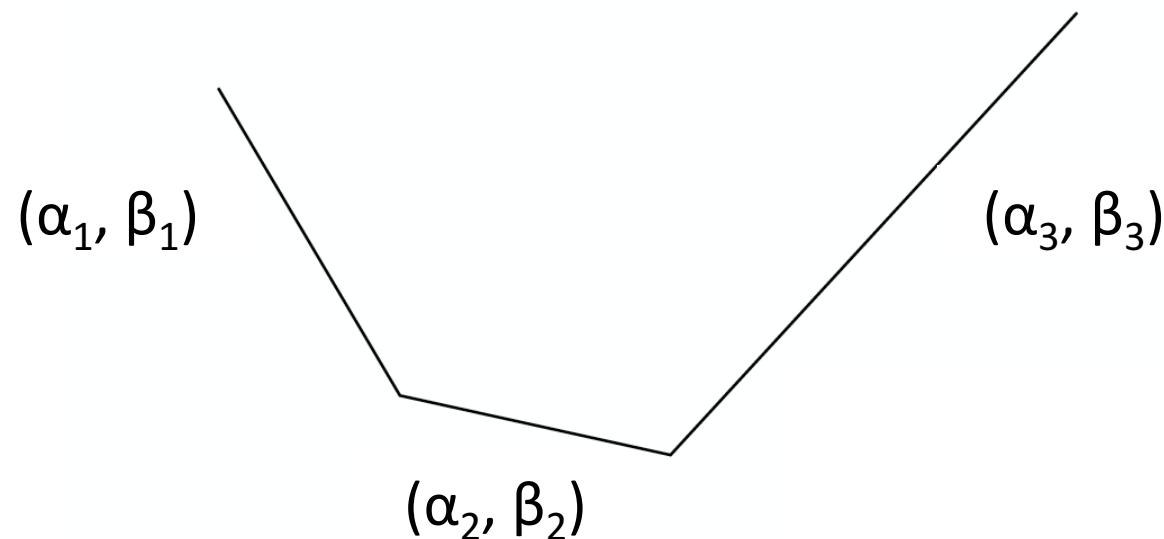
$$\alpha_i = -1 \Rightarrow g(z) = |z|$$



Maxout

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

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

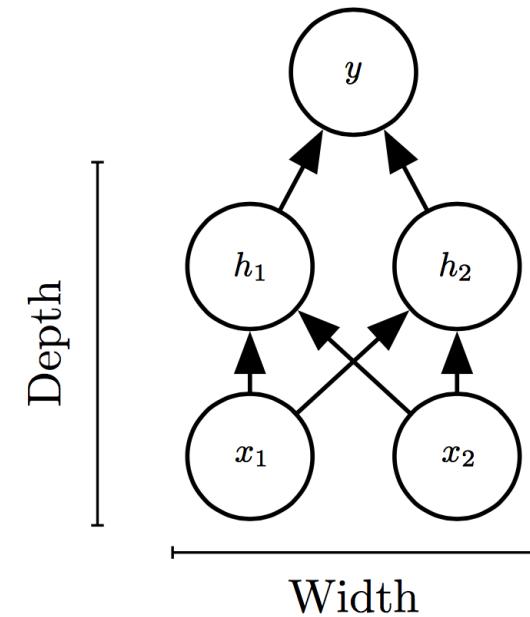


Design Choices

- 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

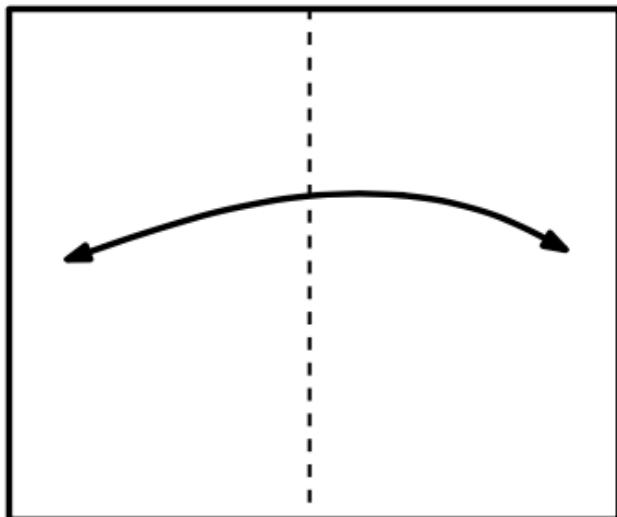


Exponential Gain with Depth

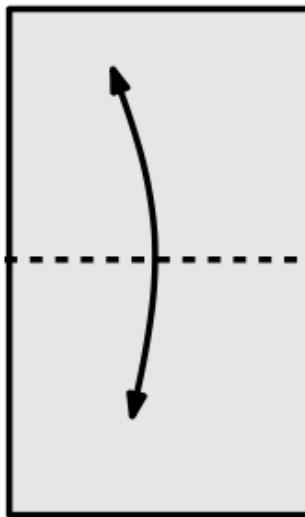
- Each hidden layer **folds** the space of activations of the previous layer. E.g. abs activation

$$g(z) = |z|$$

Montúfar (2014)



1. Fold along the vertical axis



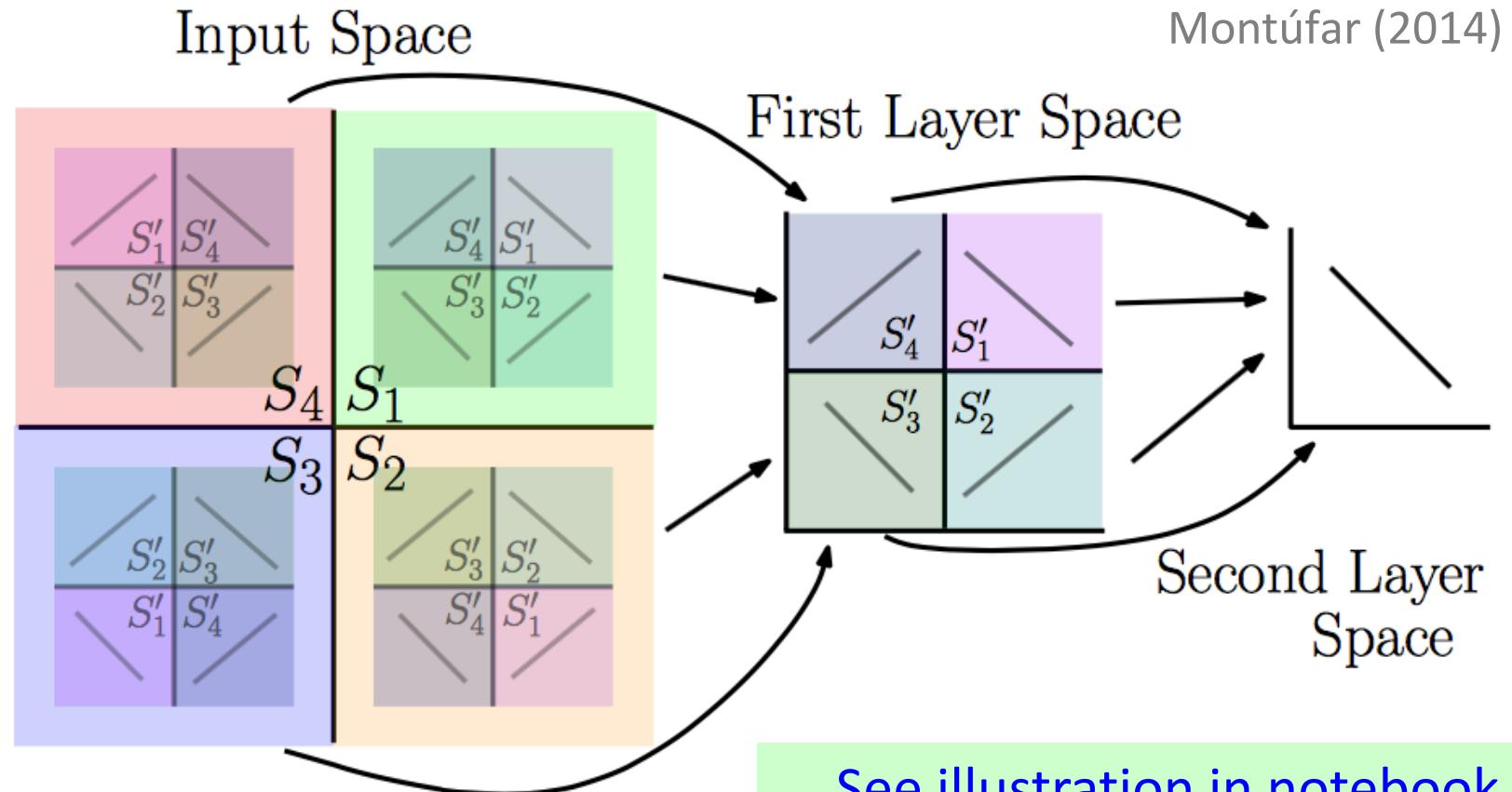
2. Fold along the horizontal axis



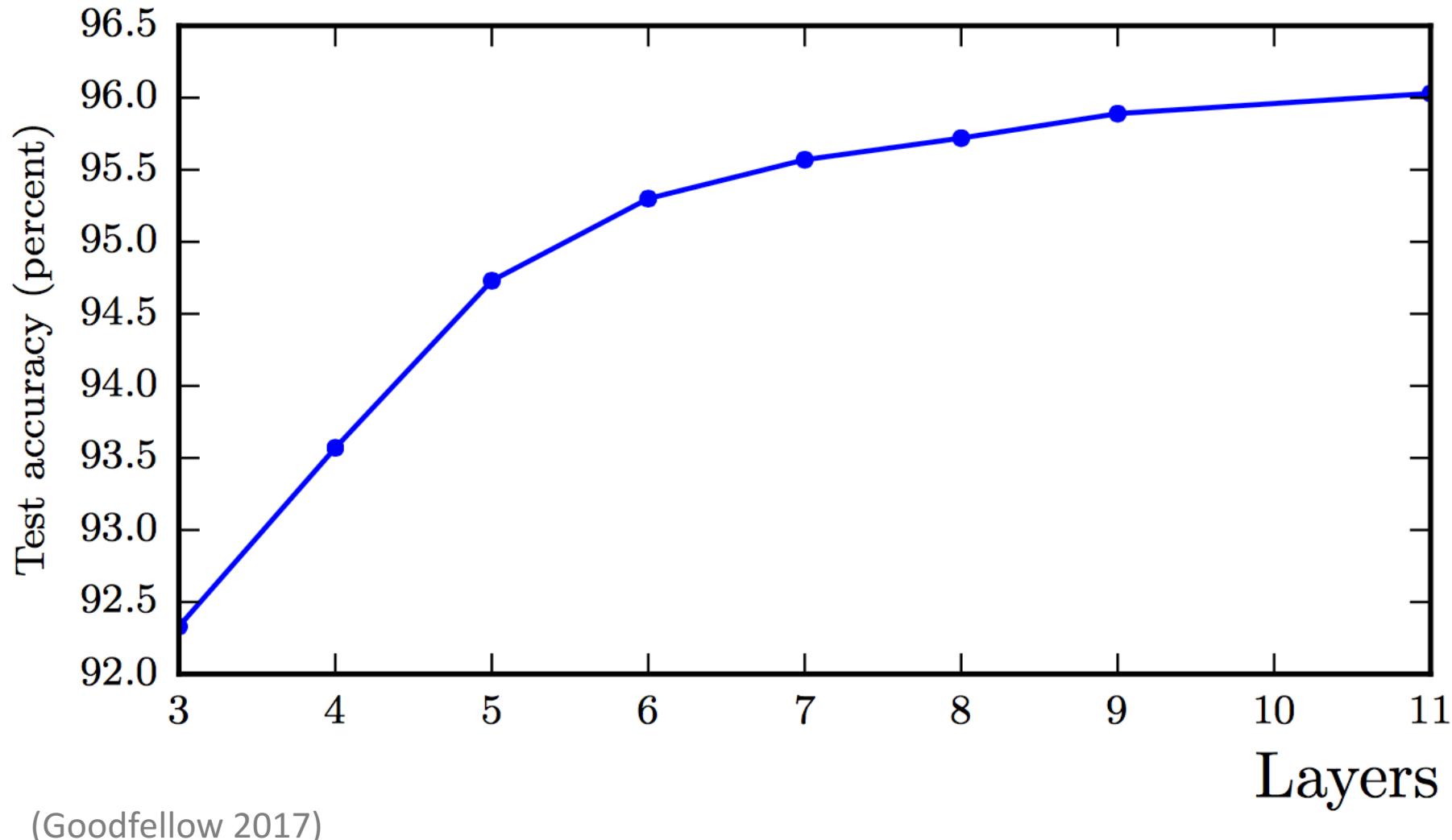
3.

Exponential Gain with Depth

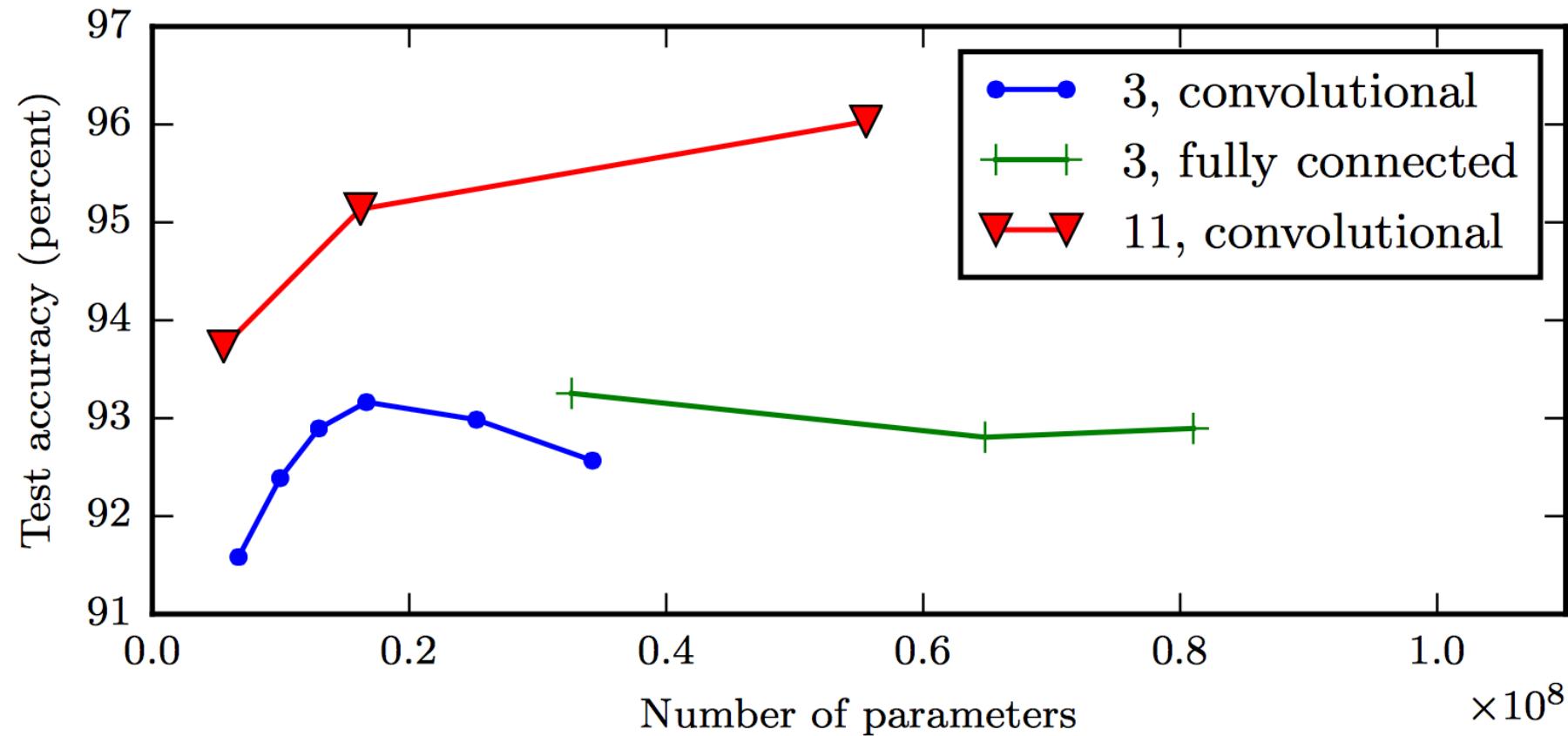
- With N hidden layers, there are $O(4^N)$ piecewise linear regions



Better Generalization with Depth



Large, Shallow Nets Overfit More



(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:

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

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

- For deeper networks

Naïve computation
takes exponential time

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

Backpropagation

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

Backprop: Arithmetic

- Jacobian-gradient products

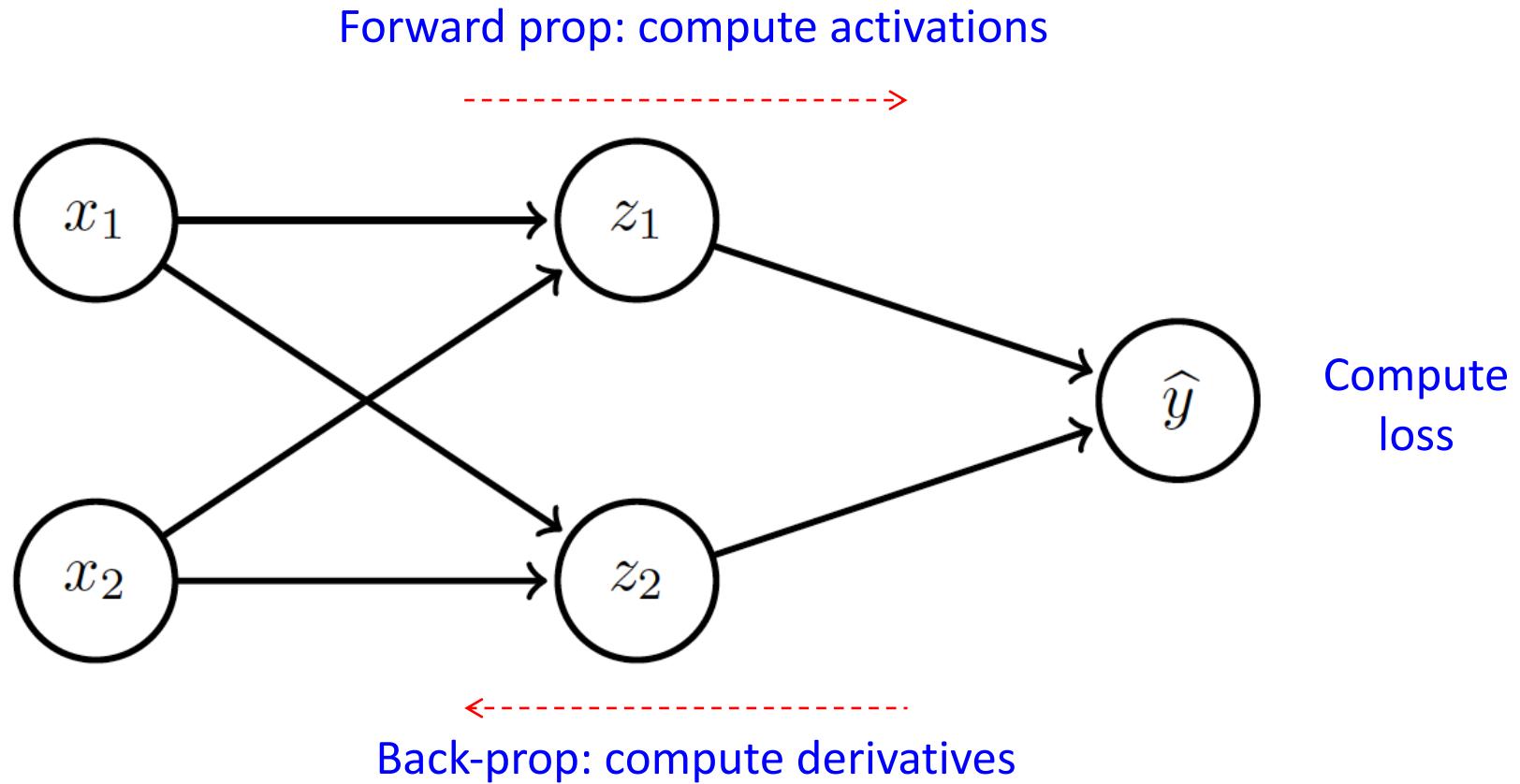
$$\begin{aligned} \mathbf{z} &= g(\mathbf{x}) \\ y &= f(\mathbf{z}) \end{aligned}$$
$$\left[\begin{array}{c} \frac{\partial y}{\partial x_1} \\ \vdots \\ \frac{\partial y}{\partial x_m} \end{array} \right] = \left[\begin{array}{ccc} \frac{\partial z_1}{\partial x_1} & \dots & \frac{\partial z_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_1}{\partial x_n} & \dots & \frac{\partial z_m}{\partial x_n} \end{array} \right] \times \left[\begin{array}{c} \frac{\partial y}{\partial z_1} \\ \vdots \\ \frac{\partial y}{\partial z_m} \end{array} \right]$$

grad w.r.t. \mathbf{x} Jacobian of 'g' grad w.r.t. \mathbf{z}

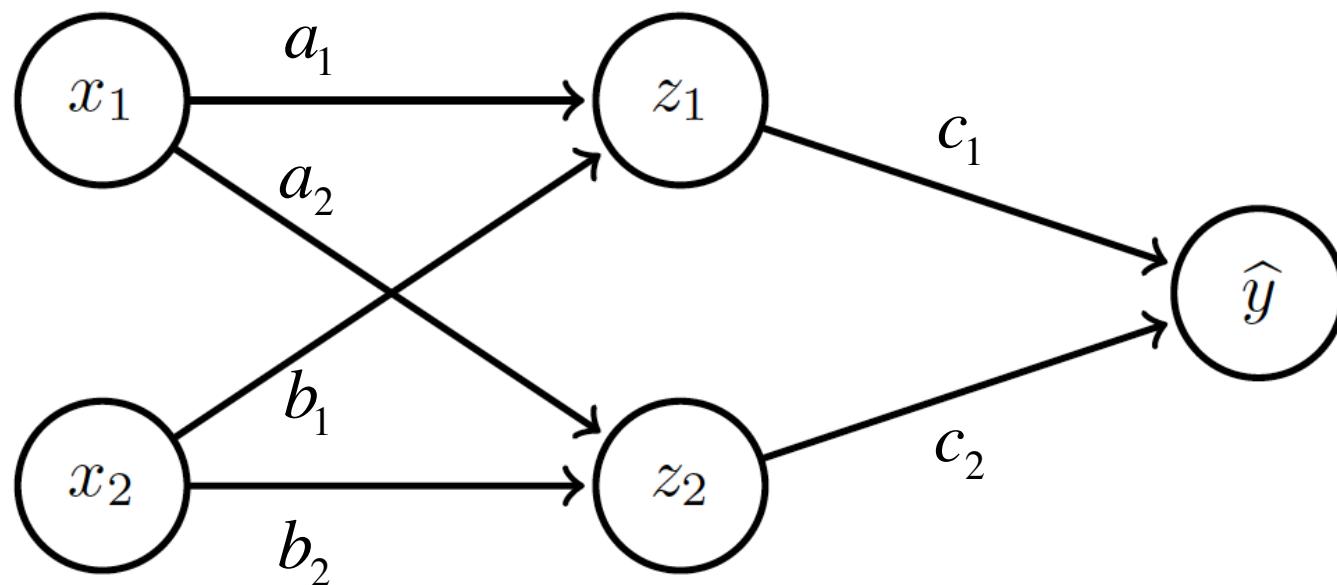
$$\nabla_{\mathbf{x}} y = \left(\frac{\partial \mathbf{z}}{\partial \mathbf{x}} \right)^T \nabla_{\mathbf{z}} y$$

Apply
recursively!

Backprop: Overview

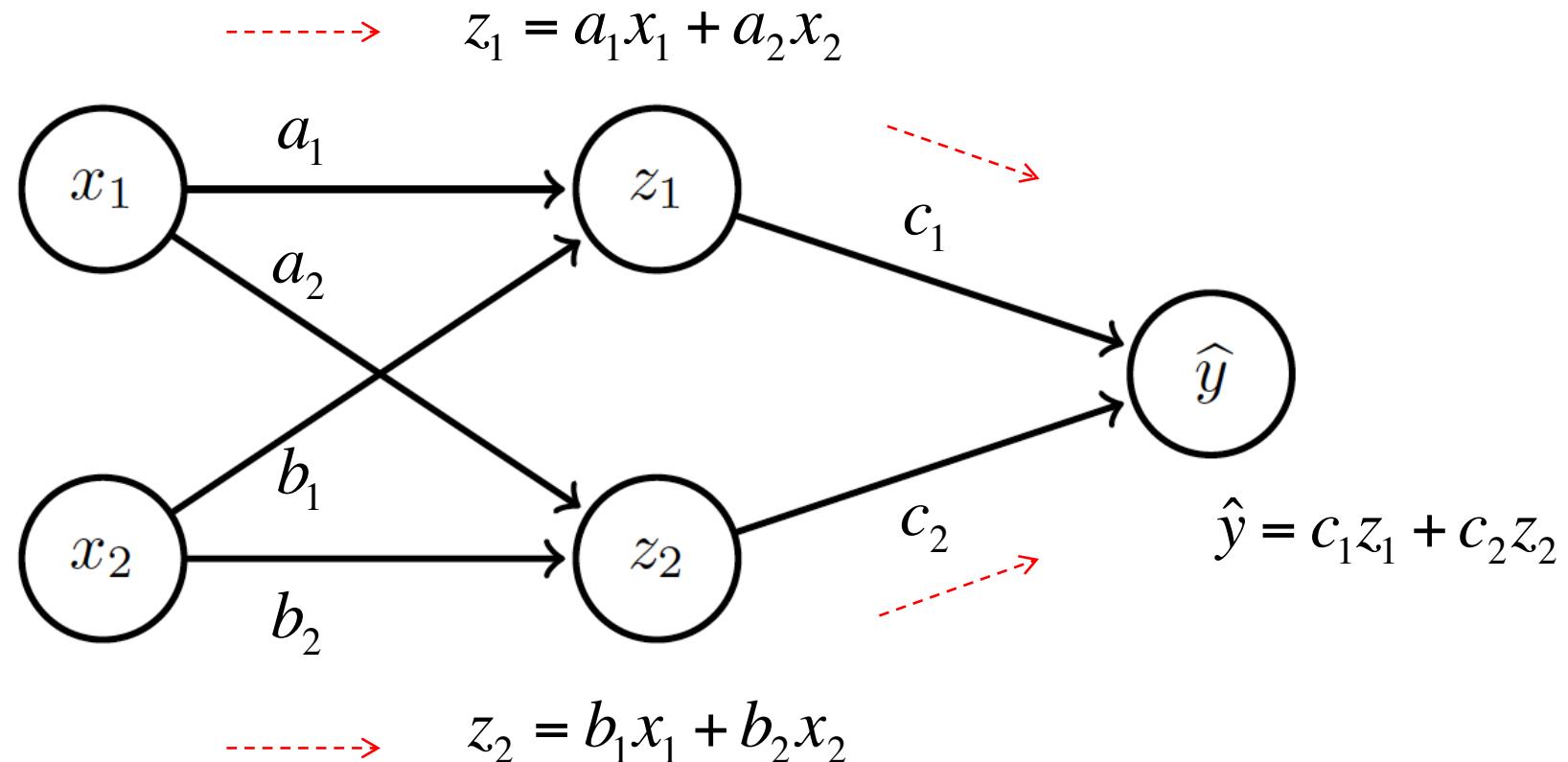


Backprop: Example



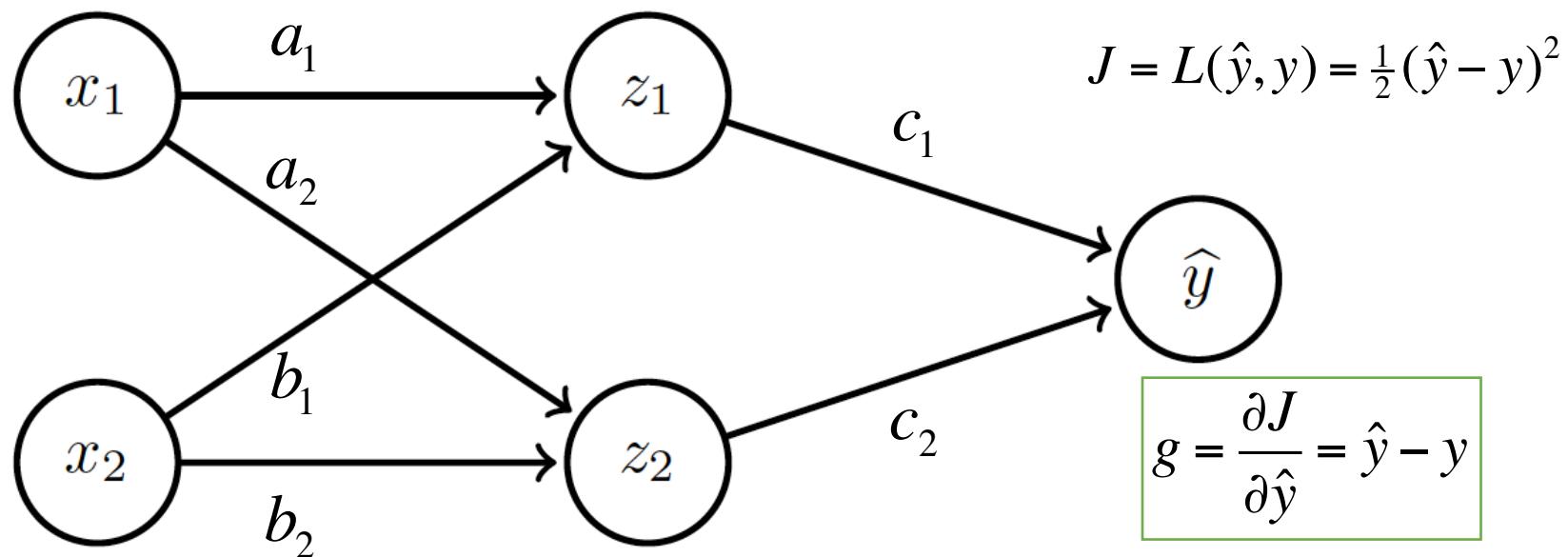
Linear activation functions
No bias
Squared loss

Backprop: Example



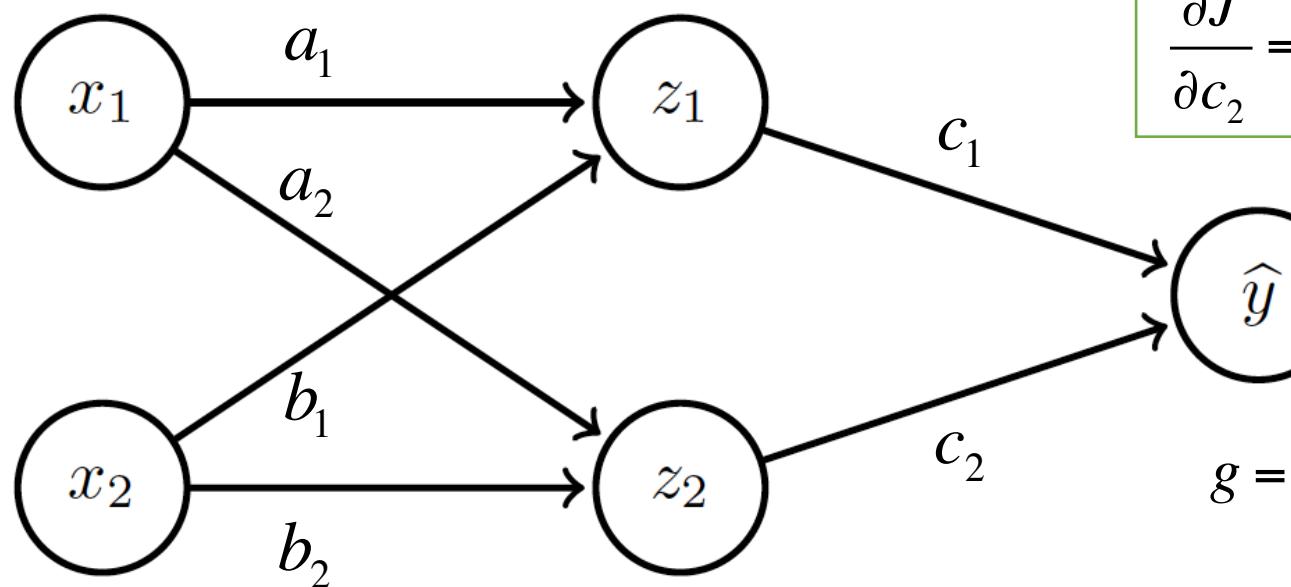
Forward prop: Propagate activations to output layer

Backprop: Example



Backward prop: Compute loss and its derivative

Backprop: Example



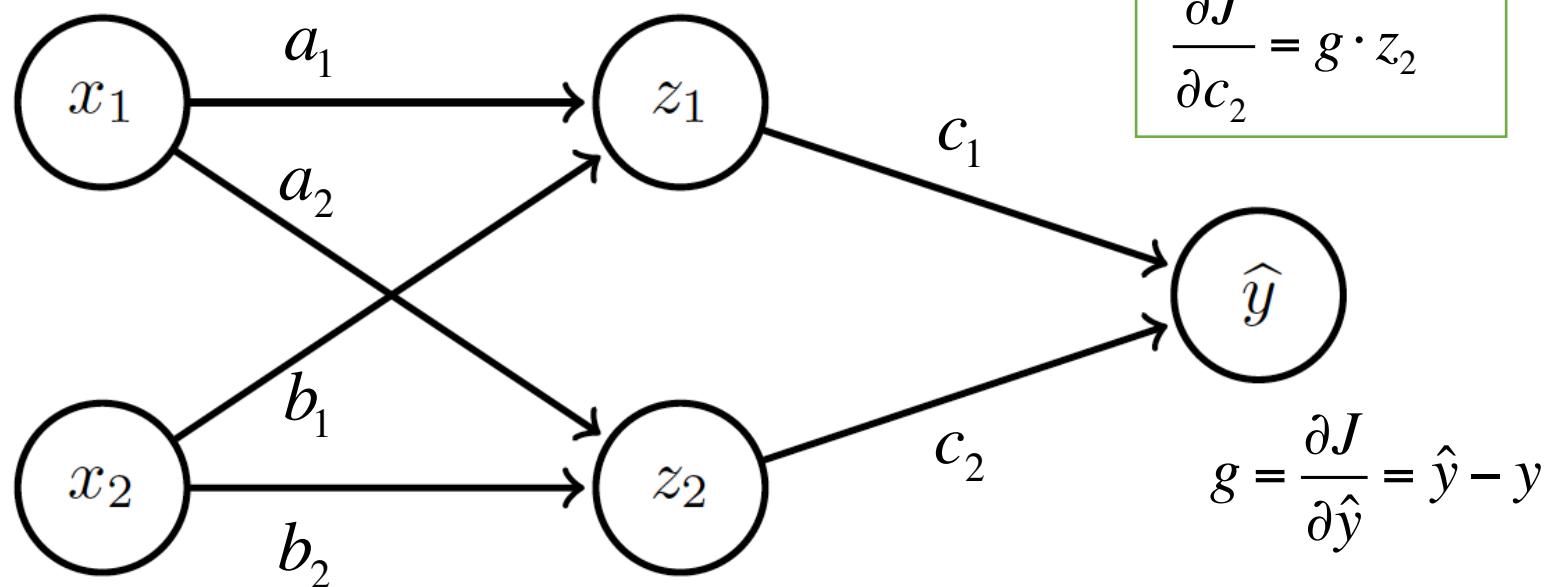
$$\frac{\partial J}{\partial c_1} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial c_1}$$

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

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

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

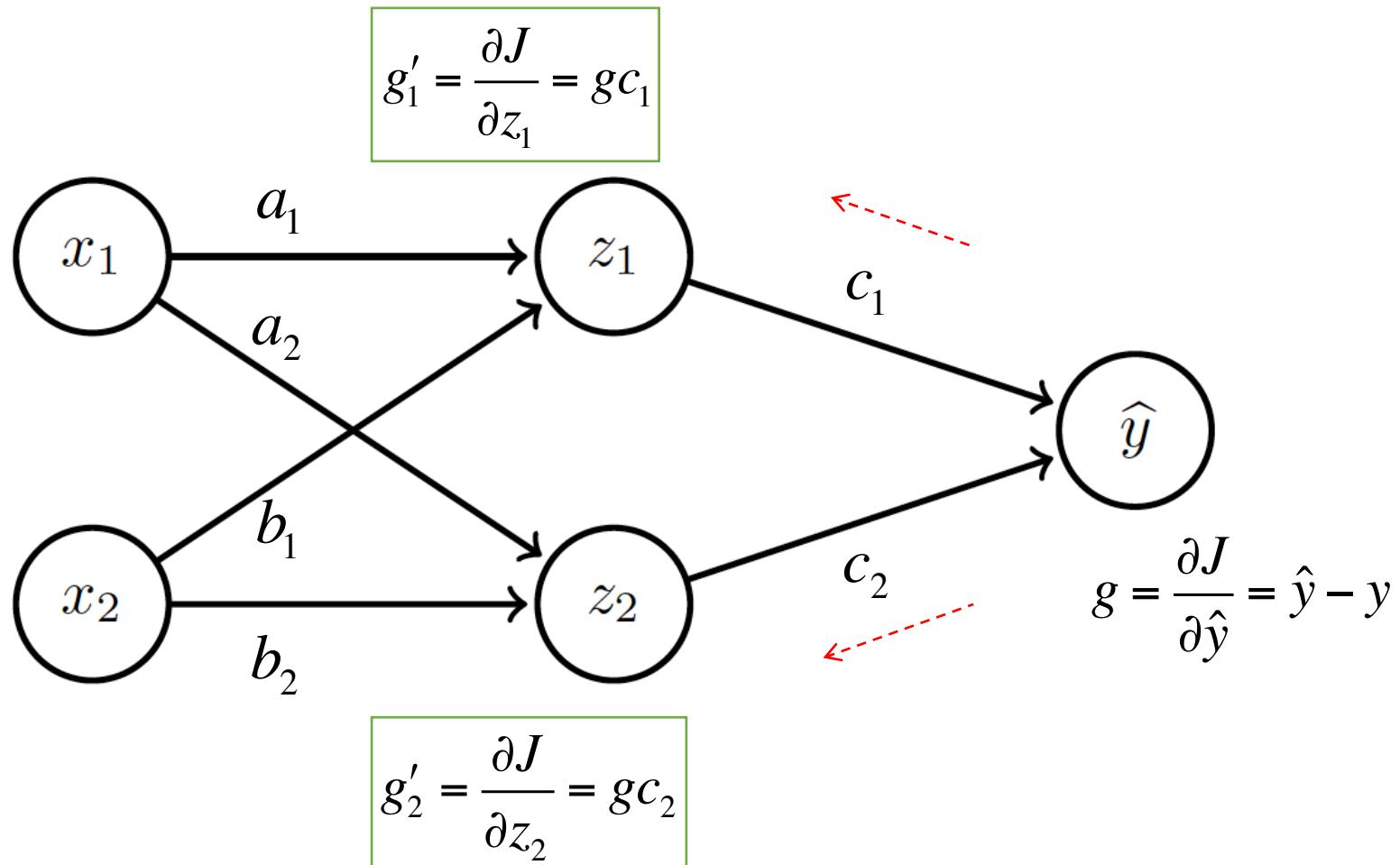
Backprop: Example



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

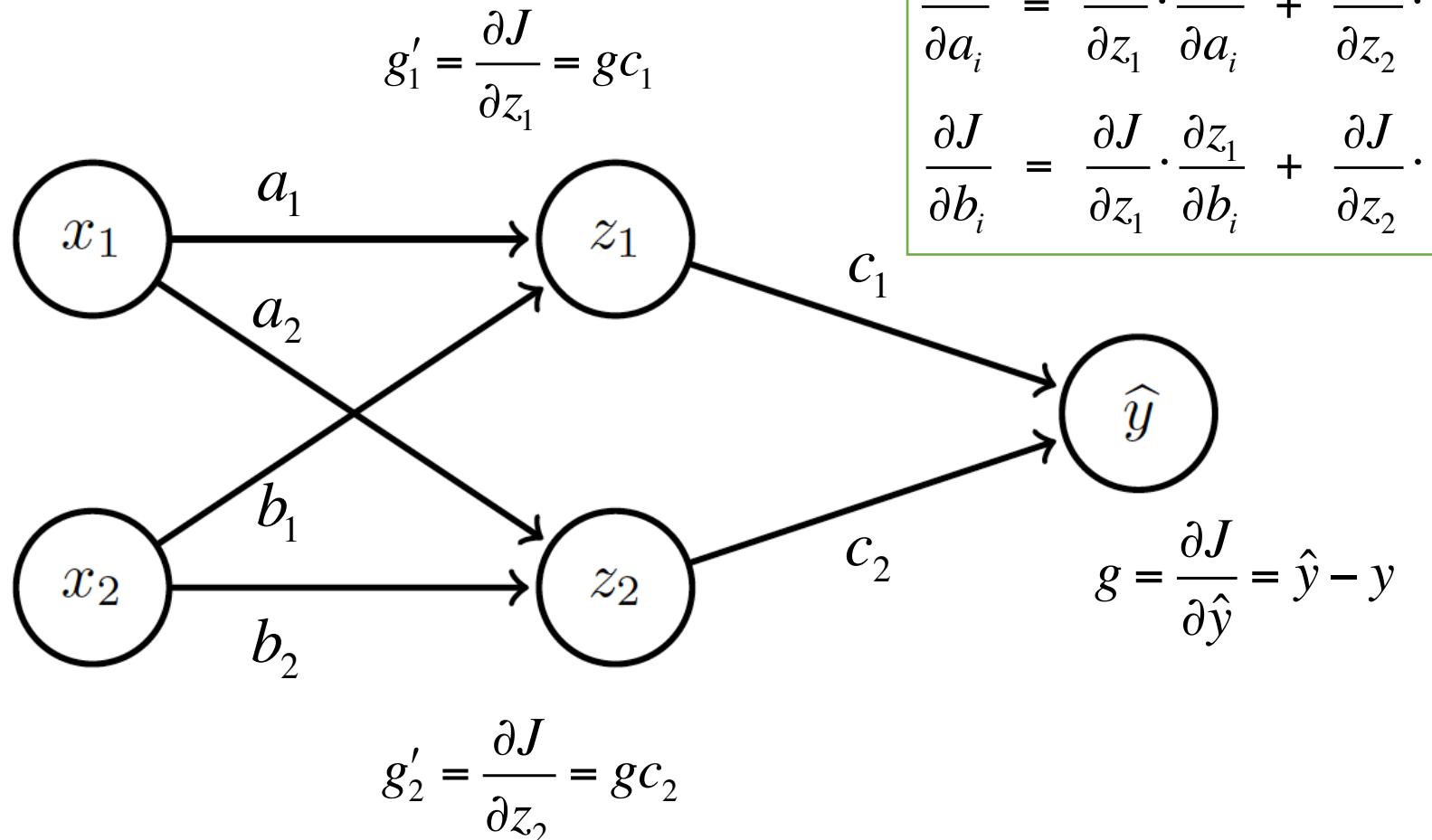
Backward prop: Compute derivatives w.r.t. weights c_1 and c_2

Backprop: Example



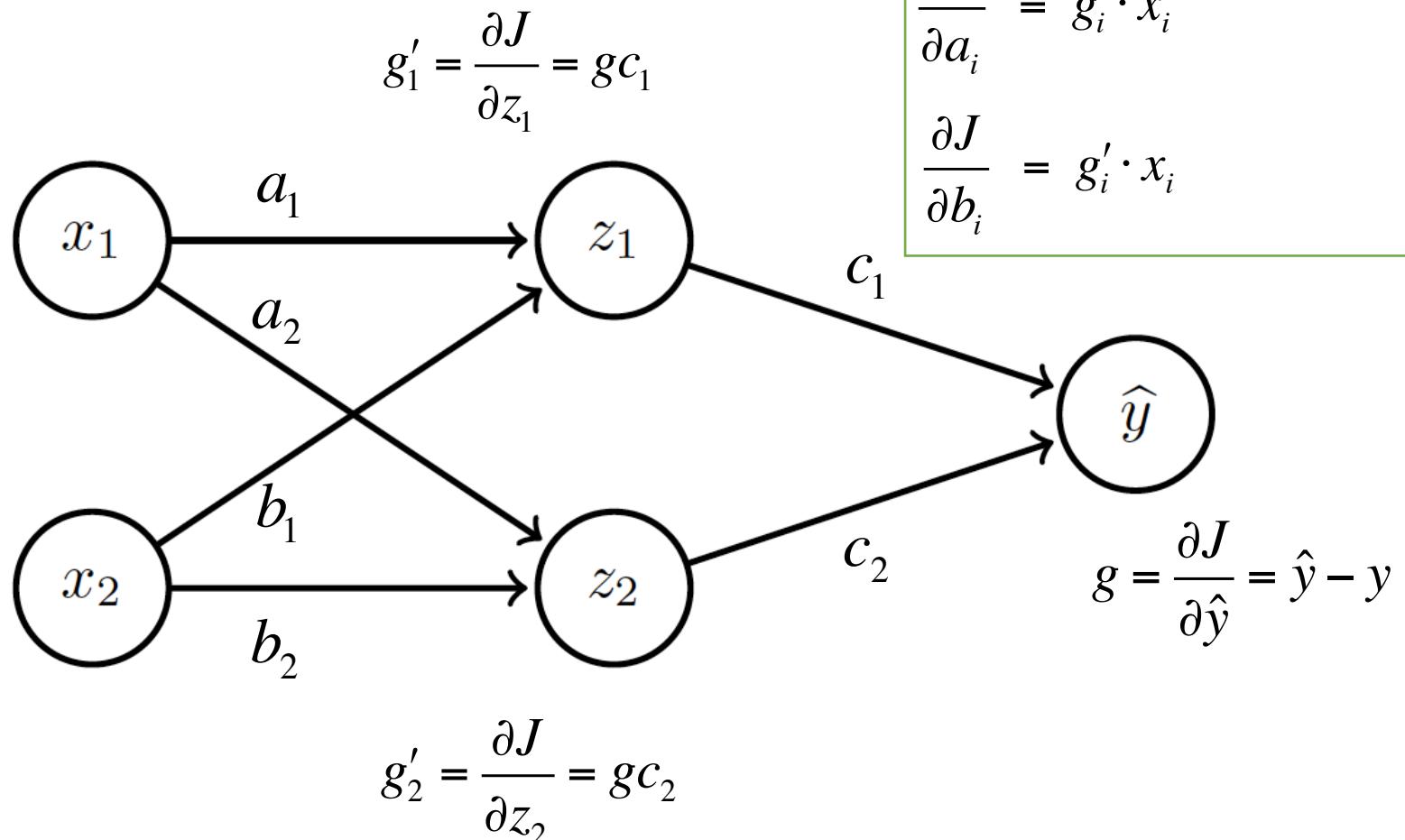
Backward prop: Propagate derivative to hidden layer

Backprop: Example



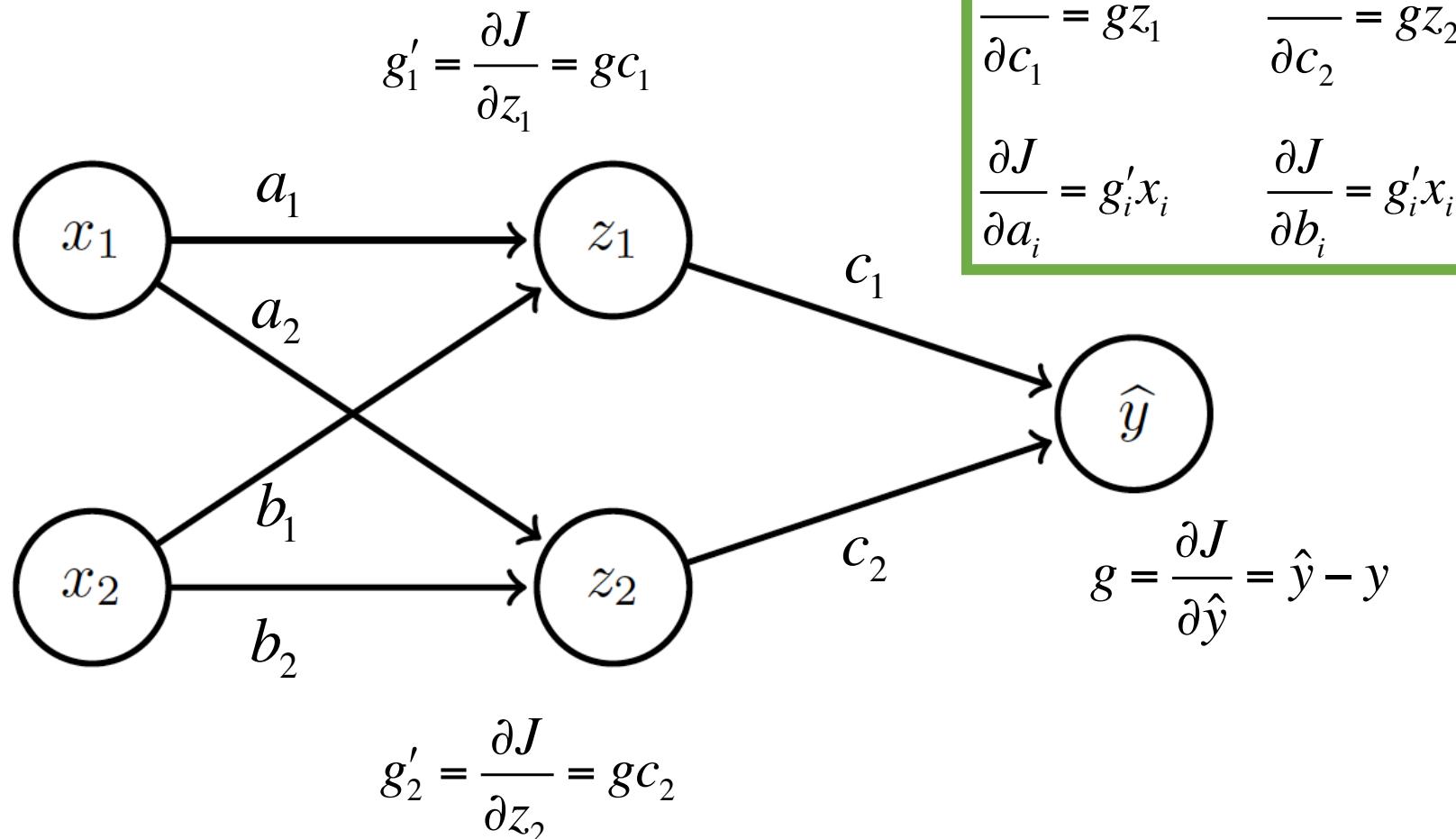
Backward prop: Compute derivatives w.r.t. weights a_1, a_2, b_1 and b_2

Backprop: Example



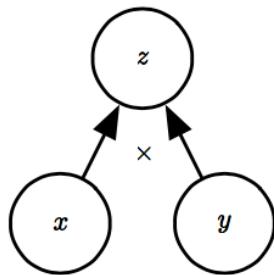
Backward prop: Compute derivatives w.r.t. weights a_1, a_2, b_1 and b_2

Backprop: Example



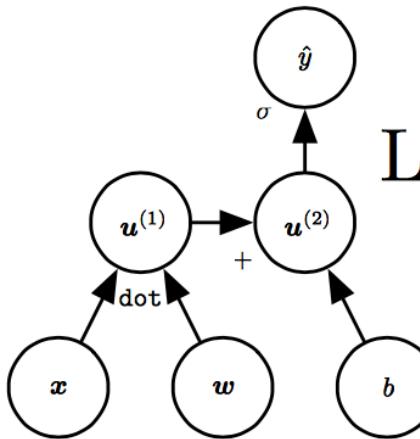
Computation Graphs

Multiplication



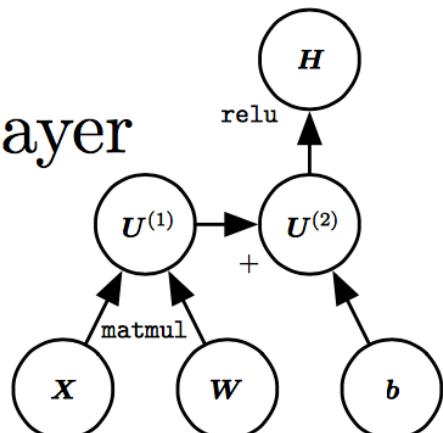
(a)

Logistic regression



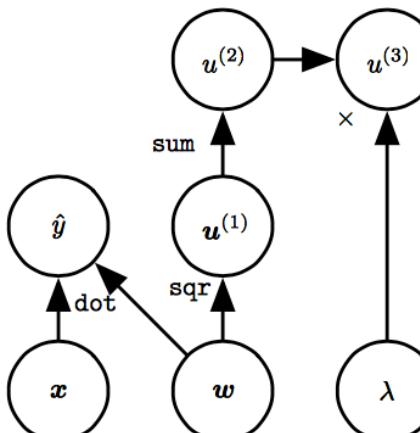
(b)

ReLU layer



(c)

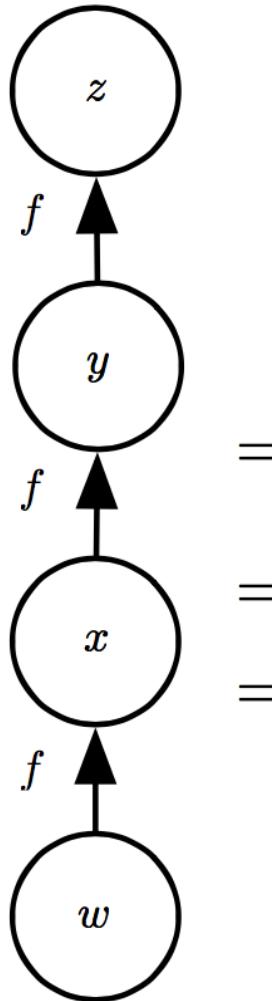
Linear regression
and weight decay



(d)

(Goodfellow 2017)

Repeated Sub-expressions



$$\begin{aligned}\frac{\partial z}{\partial w} &= \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \frac{\partial x}{\partial w} \\ &= f'(y) f'(x) f'(w) \\ &= f'(f(f(w))) f'(f(w)) f'(w)\end{aligned}$$

Back-prop avoids computing this twice

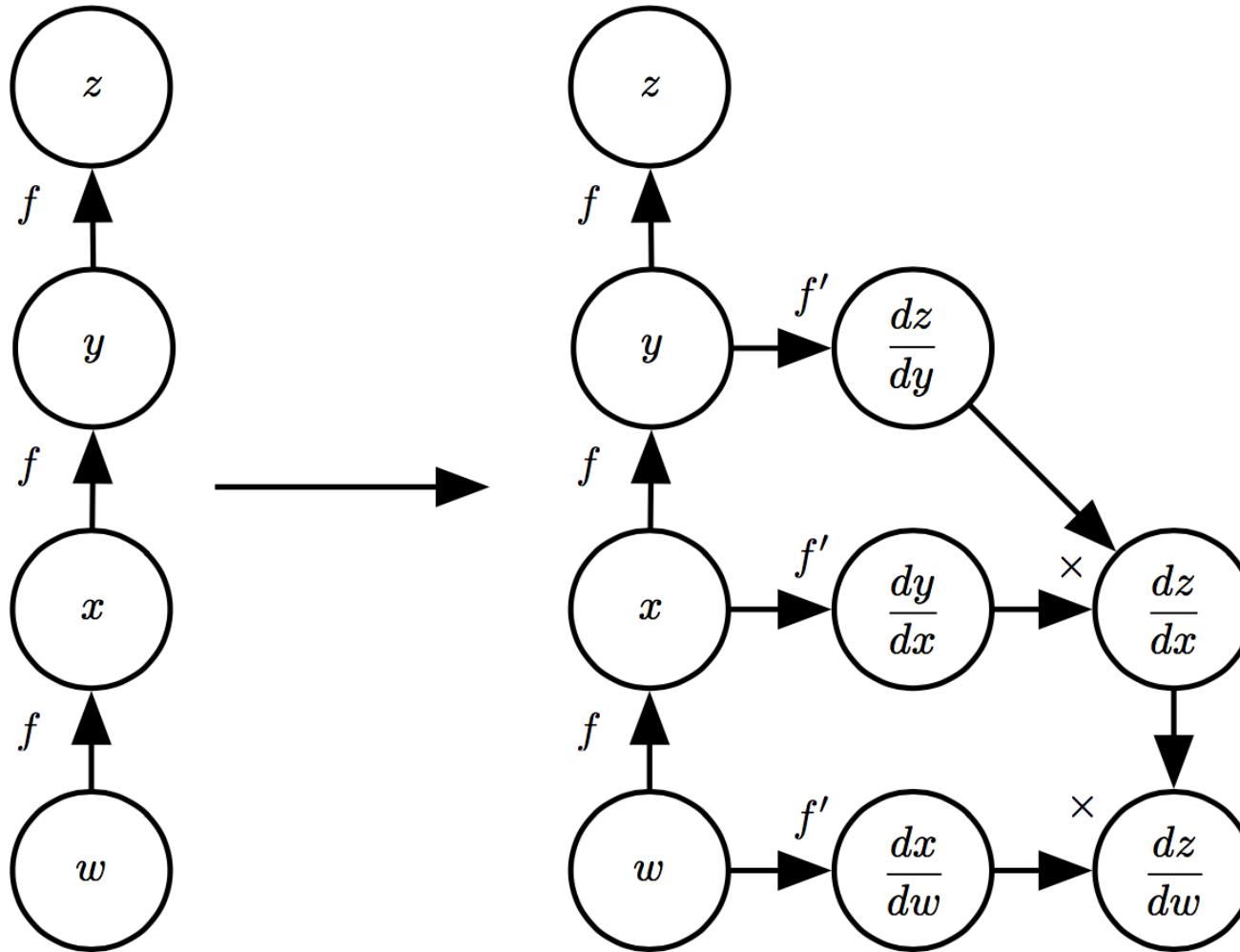
Backprop on Computation Graph

-
- The diagram illustrates a computation graph for backpropagation. It consists of four numbered steps:
- 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}
- Annotations in blue text provide additional context:
- An arrow points from step 1 to the text "Maintain grad table".
 - An arrow points from the summation term in step 3 to the text "Parents of u^i ".

Symbol-to-symbol Differentiation

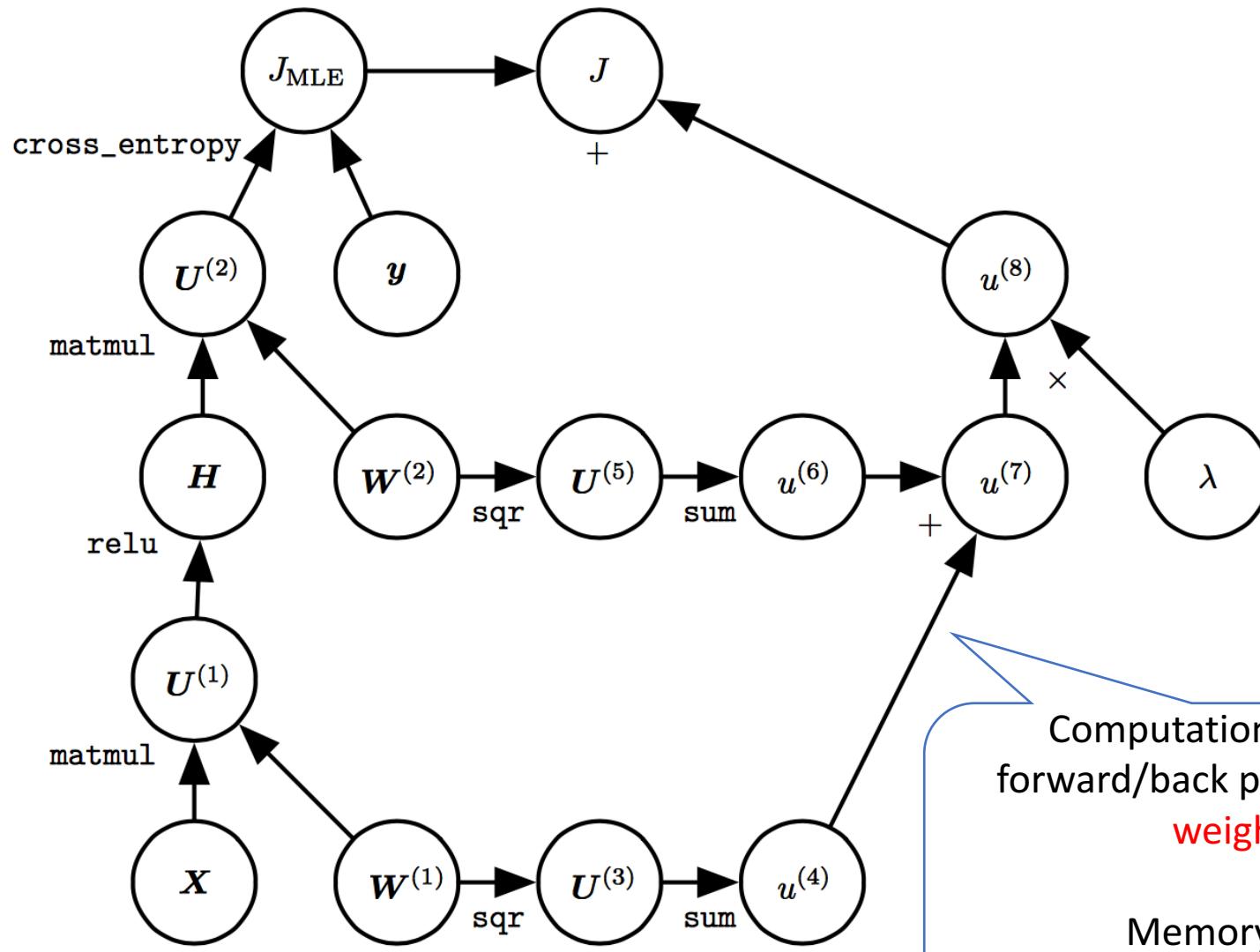
- Derivatives as computation graphs
 - Same language for both forward and back-propagation
- During execution, replace symbolic inputs with numeric value
- Used by [Theano](#) and [TensorFlow](#)
- Symbol-to-number differentiation: e.g. Torch and Caffe

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:
 $O(\#num-layers \times minibatch-size)$

Notebook

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