

Machine Learning Systems

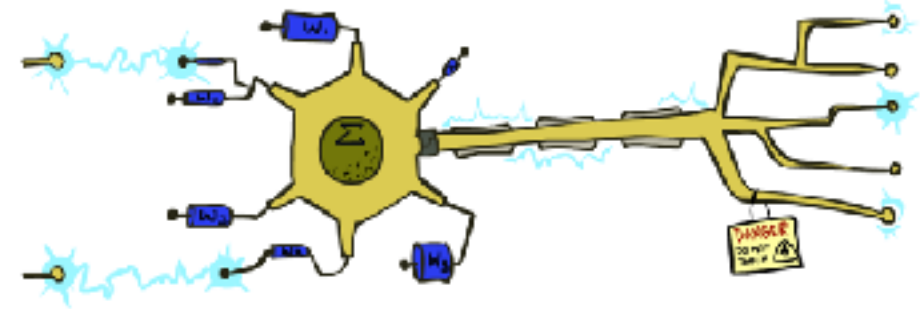
Lecture 6: Optimization and Neural Networks

Pooyan Jamshidi



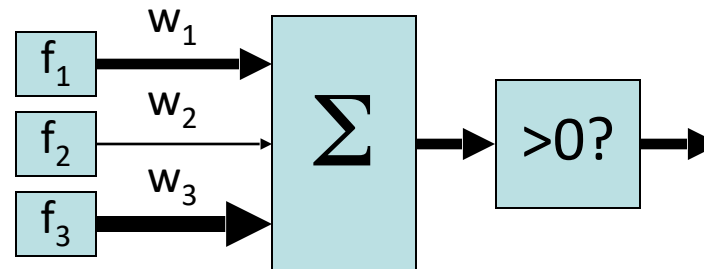
Reminder: Linear Classifiers

- Inputs are **feature values**
- Each feature has a **weight**
- Sum is the **activation**



$$\text{activation}_w(x) = \sum_i w_i \cdot f_i(x) = w \cdot f(x)$$

- If the activation is:
 - Positive, output +1
 - Negative, output -1

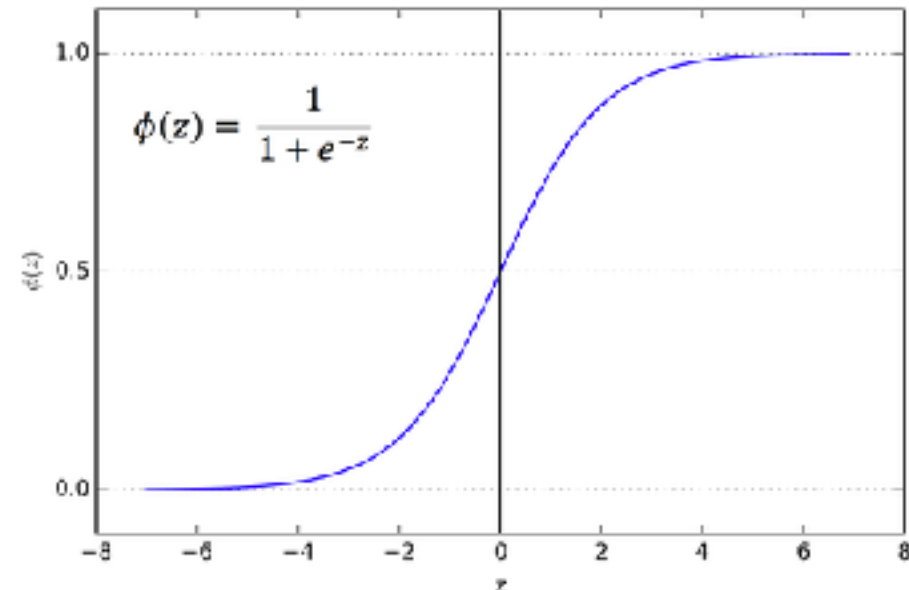


How to get probabilistic decisions?

- Activation: $z = w \cdot f(x)$
- If $z = w \cdot f(x)$ very positive \rightarrow want probability going to 1
- If $z = w \cdot f(x)$ very negative \rightarrow want probability going to 0

- Sigmoid function

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



Best w?

- Maximum likelihood estimation:

$$\max_w ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

with:

$$P(y^{(i)} = +1 | x^{(i)}; w) = \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}$$

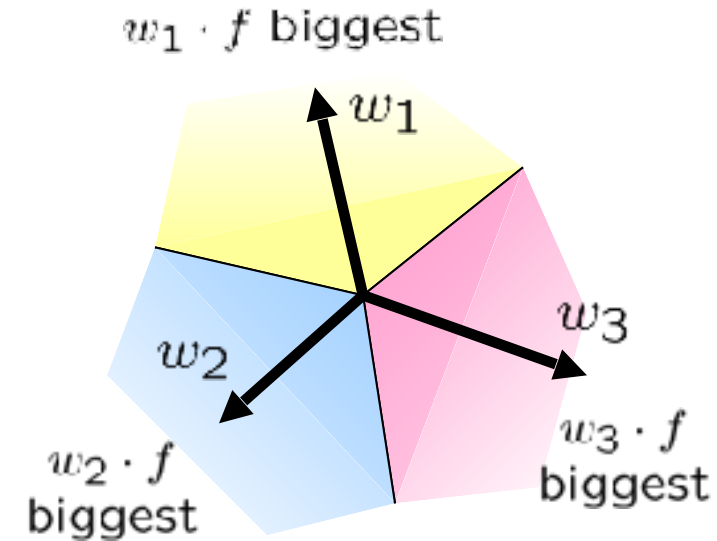
$$P(y^{(i)} = -1 | x^{(i)}; w) = 1 - \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}$$

= Logistic Regression

Multiclass Logistic Regression

■ Multi-class linear classification

- A weight vector for each class: w_y
- Score (activation) of a class y : $w_y \cdot f(x)$
- Prediction w/highest score wins: $y = \arg \max_y w_y \cdot f(x)$



■ How to make the scores into probabilities?

$$\underbrace{z_1, z_2, z_3}_{\text{original activations}} \rightarrow \underbrace{\frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}, \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}, \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}}}_{\text{softmax activations}}$$

Best w ?

- Maximum likelihood estimation:

$$\max_w ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

with:

$$P(y^{(i)} | x^{(i)}; w) = \frac{e^{w_{y^{(i)}} \cdot f(x^{(i)})}}{\sum_y e^{w_y \cdot f(x^{(i)})}}$$

= Multi-Class Logistic Regression

This Lecture

- Optimization

- i.e., how do we solve:

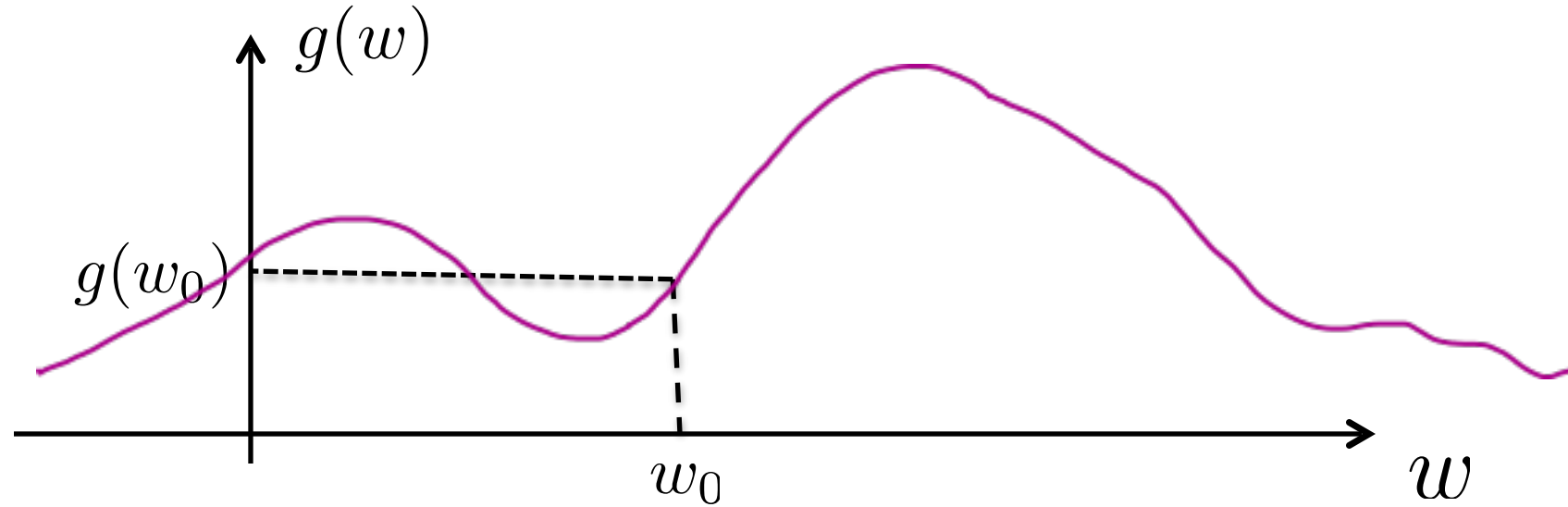
$$\max_w ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

Hill Climbing

- Recall from CSPs lecture: simple, general idea
 - Start wherever
 - Repeat: move to the best neighboring state
 - If no neighbors better than current, quit
- What's particularly tricky when hill-climbing for multiclass logistic regression?
 - Optimization over a continuous space
 - Infinitely many neighbors!
 - How to do this efficiently?



1-D Optimization



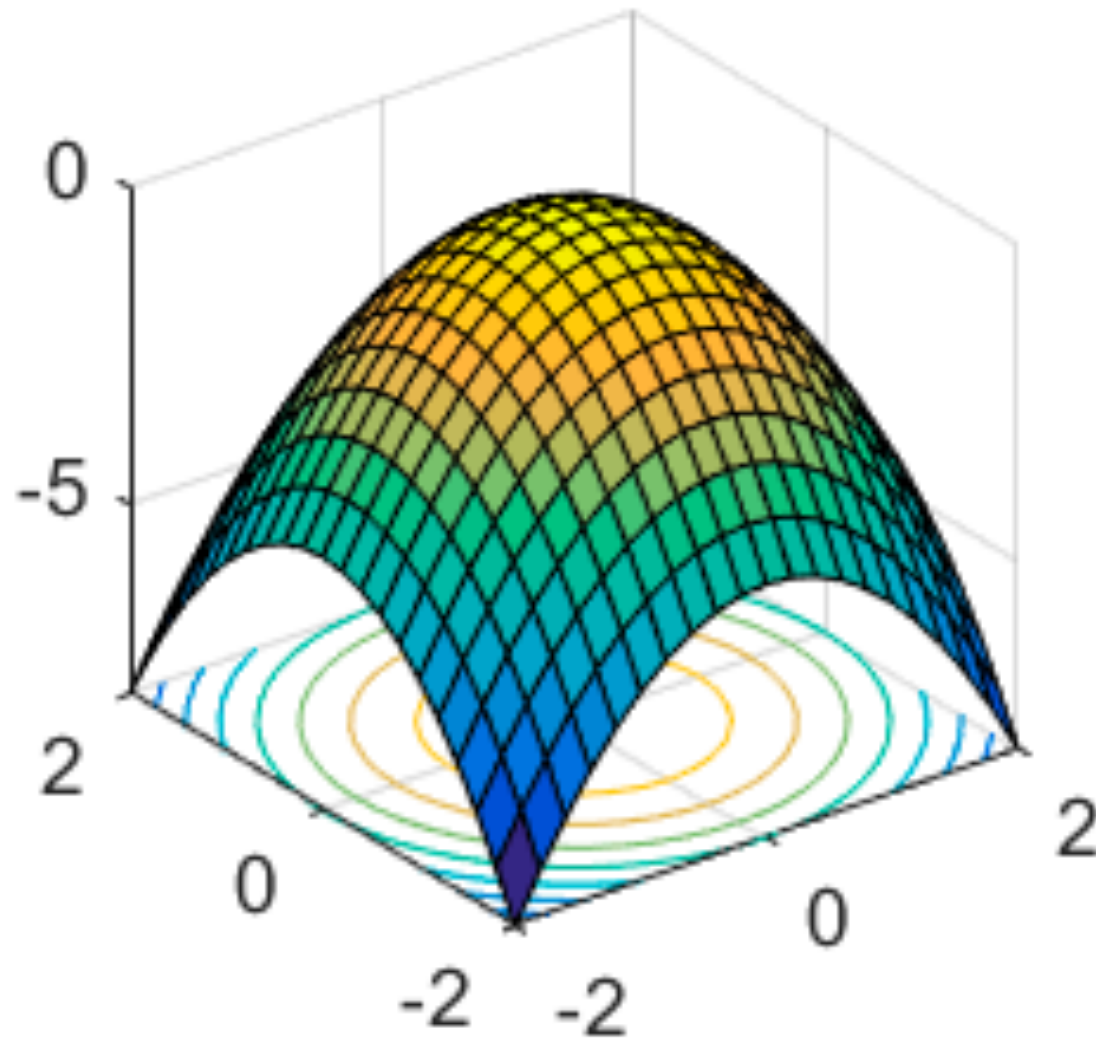
- Could evaluate $g(w_0 + h)$ and $g(w_0 - h)$

- Then step in best direction

- Or, evaluate derivative:
$$\frac{\partial g(w_0)}{\partial w} = \lim_{h \rightarrow 0} \frac{g(w_0 + h) - g(w_0 - h)}{2h}$$

- Tells which direction to step into

2-D Optimization



Gradient Ascent

- Perform update in uphill direction for each coordinate
- The steeper the slope (i.e. the higher the derivative) the bigger the step for that coordinate
- E.g., consider: $g(w_1, w_2)$

- Updates:

$$w_1 \leftarrow w_1 + \alpha * \frac{\partial g}{\partial w_1}(w_1, w_2)$$

$$w_2 \leftarrow w_2 + \alpha * \frac{\partial g}{\partial w_2}(w_1, w_2)$$

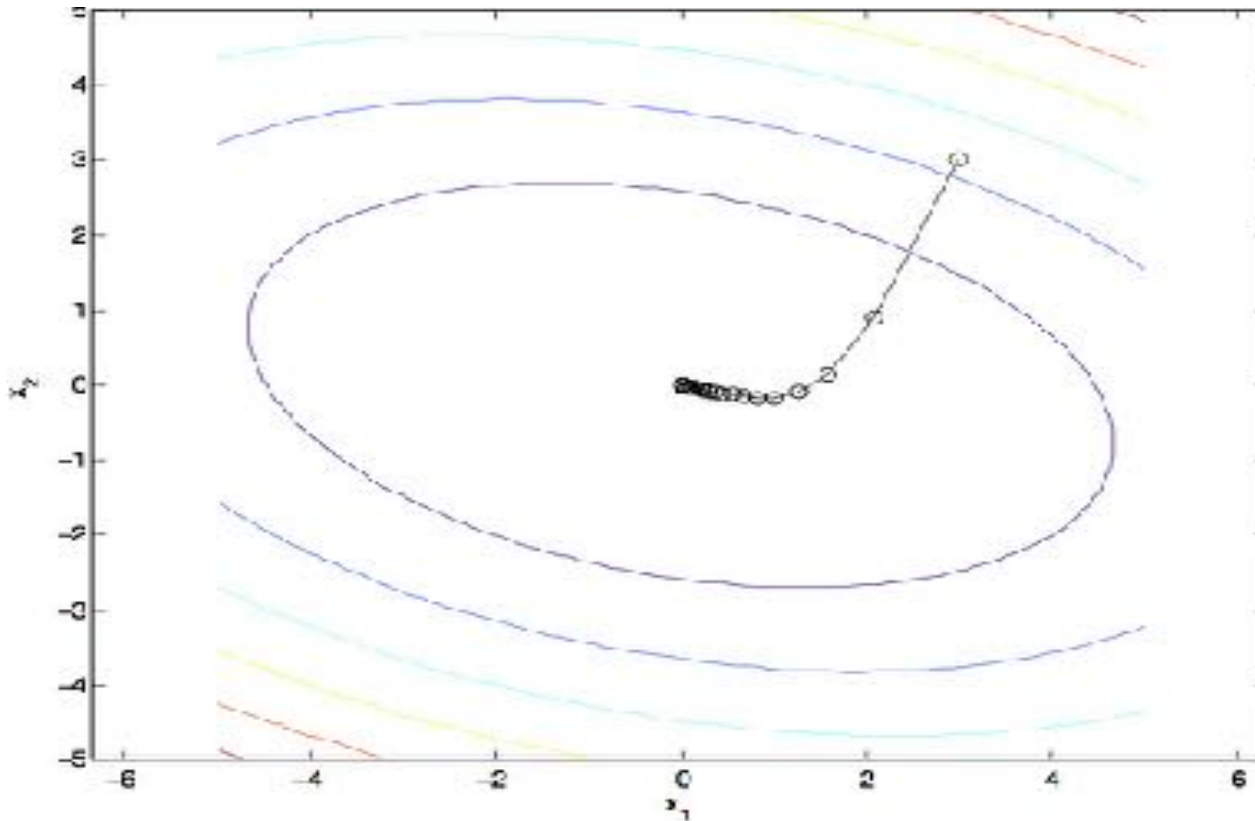
- Updates in vector notation:

$$w \leftarrow w + \alpha * \nabla_w g(w)$$

$$\text{with: } \nabla_w g(w) = \begin{bmatrix} \frac{\partial g}{\partial w_1}(w) \\ \frac{\partial g}{\partial w_2}(w) \end{bmatrix} = \text{gradient}$$

Gradient Ascent

- Idea:
 - Start somewhere
 - Repeat: Take a step in the gradient direction



What is the Steepest Direction?

$$\max_{\Delta: \Delta_1^2 + \Delta_2^2 \leq \varepsilon} g(w + \Delta)$$



- First-Order Taylor Expansion:

$$g(w + \Delta) \approx g(w) + \frac{\partial g}{\partial w_1} \Delta_1 + \frac{\partial g}{\partial w_2} \Delta_2$$

- Steepest Ascent Direction:

$$\max_{\Delta: \Delta_1^2 + \Delta_2^2 \leq \varepsilon} g(w) + \frac{\partial g}{\partial w_1} \Delta_1 + \frac{\partial g}{\partial w_2} \Delta_2$$

- Recall: $\max_{\Delta: \|\Delta\| \leq \varepsilon} \Delta^\top a \rightarrow$

$$\Delta = \varepsilon \frac{a}{\|a\|}$$

- Hence, solution: $\Delta = \varepsilon \frac{\nabla g}{\|\nabla g\|}$

Gradient direction = steepest direction!

$$\nabla g = \begin{bmatrix} \frac{\partial g}{\partial w_1} \\ \frac{\partial g}{\partial w_2} \end{bmatrix}$$

Gradient in n dimensions

$$\nabla g = \begin{bmatrix} \frac{\partial g}{\partial w_1} \\ \frac{\partial g}{\partial w_2} \\ \dots \\ \frac{\partial g}{\partial w_n} \end{bmatrix}$$

Optimization Procedure: Gradient Ascent

```
■ init  $w$   
■ for iter = 1, 2, ...  
     $w \leftarrow w + \alpha * \nabla g(w)$ 
```

- α : learning rate --- tweaking parameter that needs to be chosen carefully
- How? Try multiple choices
 - Crude rule of thumb: update changes w about 0.1 – 1 %

Batch Gradient Ascent on the Log Likelihood Objective

$$\max_w ll(w) = \max_w \underbrace{\sum_i \log P(y^{(i)} | x^{(i)}; w)}_{g(w)}$$

- `init` w
- `for` $iter = 1, 2, \dots$

$$w \leftarrow w + \alpha * \sum_i \nabla \log P(y^{(i)} | x^{(i)}; w)$$

Stochastic Gradient Ascent on the Log Likelihood Objective

$$\max_w ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

Observation: once gradient on one training example has been computed, might as well incorporate before computing next one

- `init` w
- `for` $iter = 1, 2, \dots$
 - pick random j

$$w \leftarrow w + \alpha * \nabla \log P(y^{(j)} | x^{(j)}; w)$$

Mini-Batch Gradient Ascent on the Log Likelihood Objective

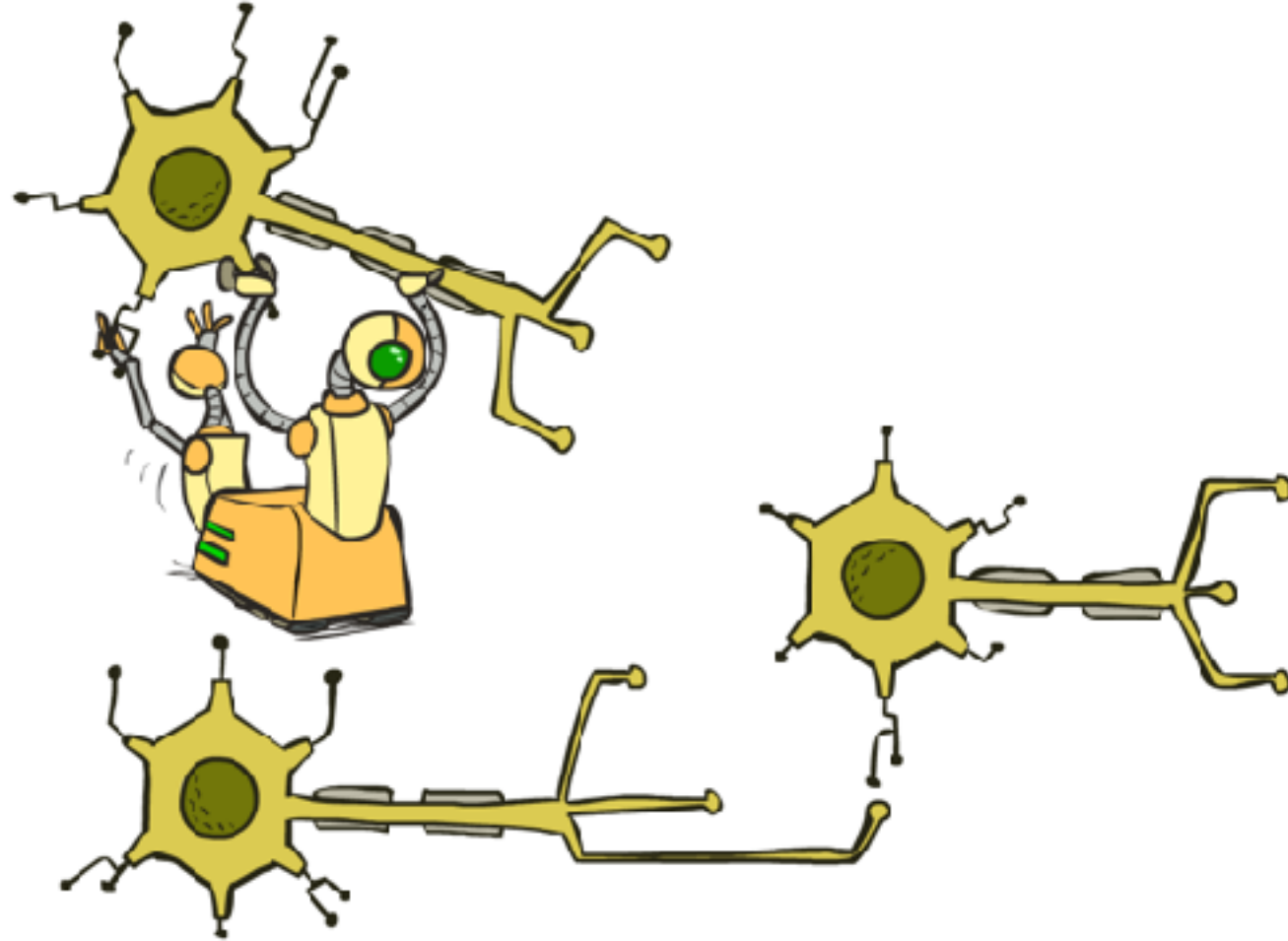
$$\max_w ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

Observation: gradient over small set of training examples (=mini-batch) can be computed in parallel, might as well do that instead of a single one

- `init` w
- `for` $iter = 1, 2, \dots$
 - pick random subset of training examples J

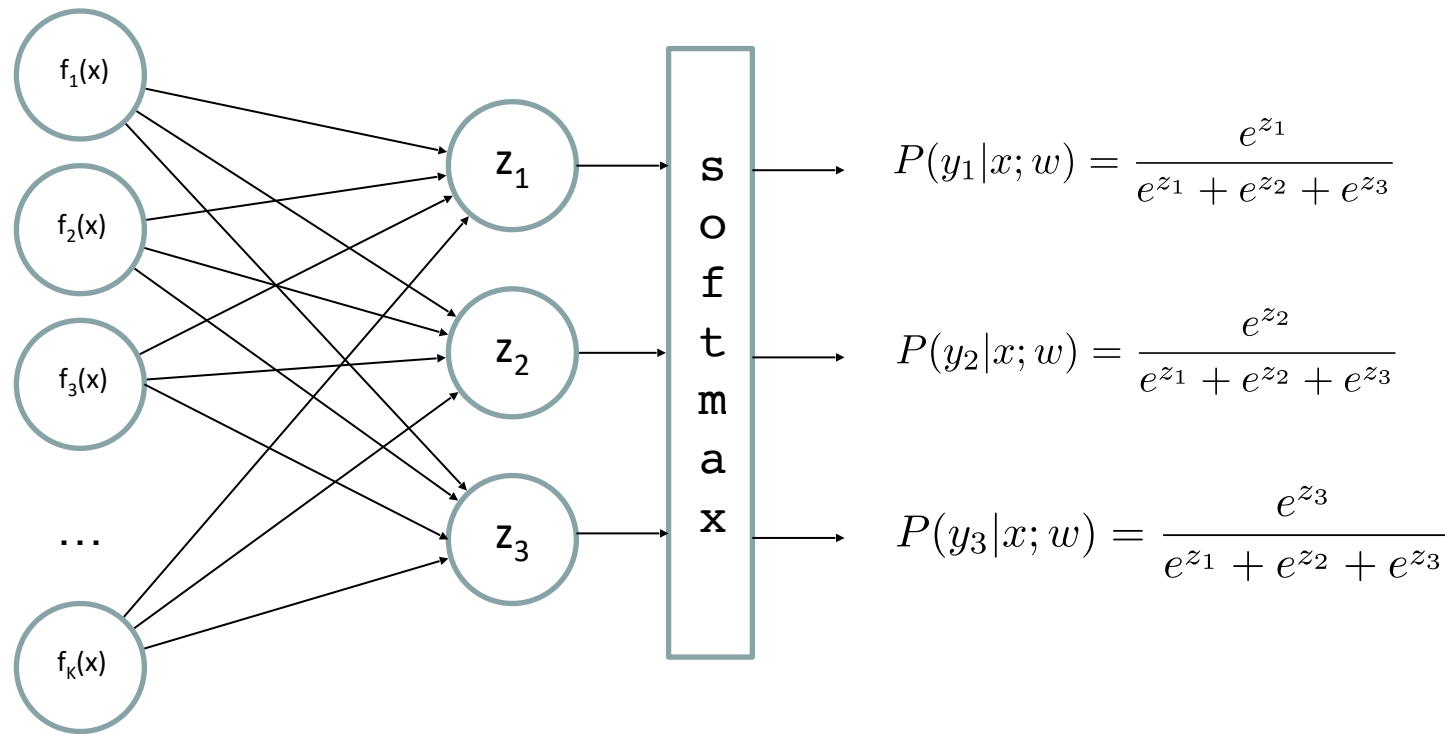
$$w \leftarrow w + \alpha * \sum_{j \in J} \nabla \log P(y^{(j)} | x^{(j)}; w)$$

Neural Networks

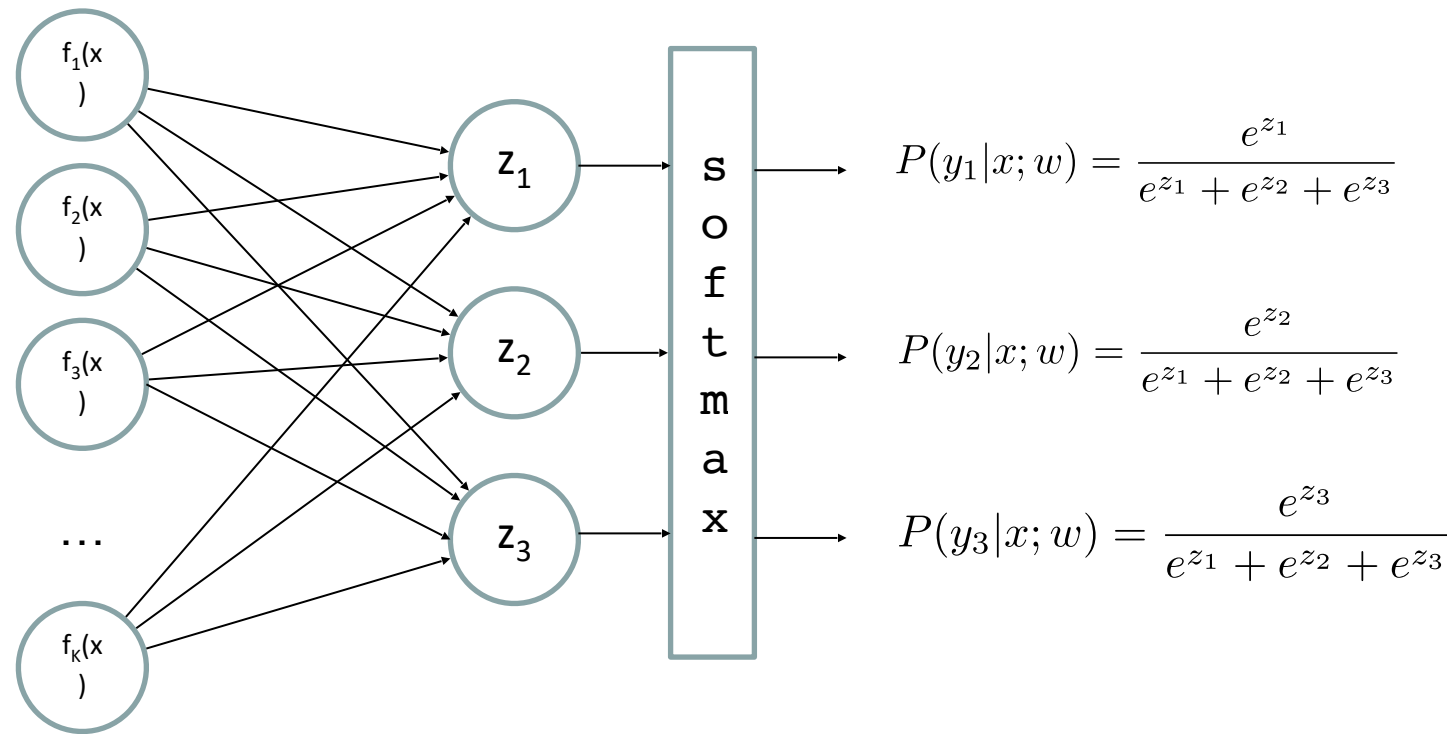


Multi-class Logistic Regression

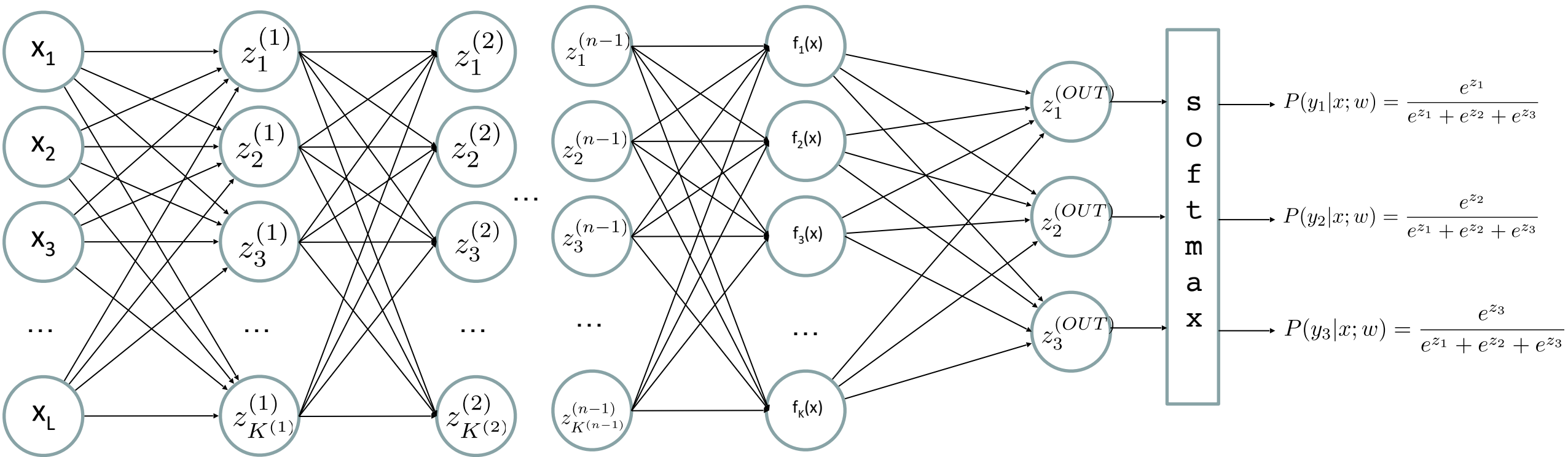
- = special case of neural network



Deep Neural Network = Also learn the features!



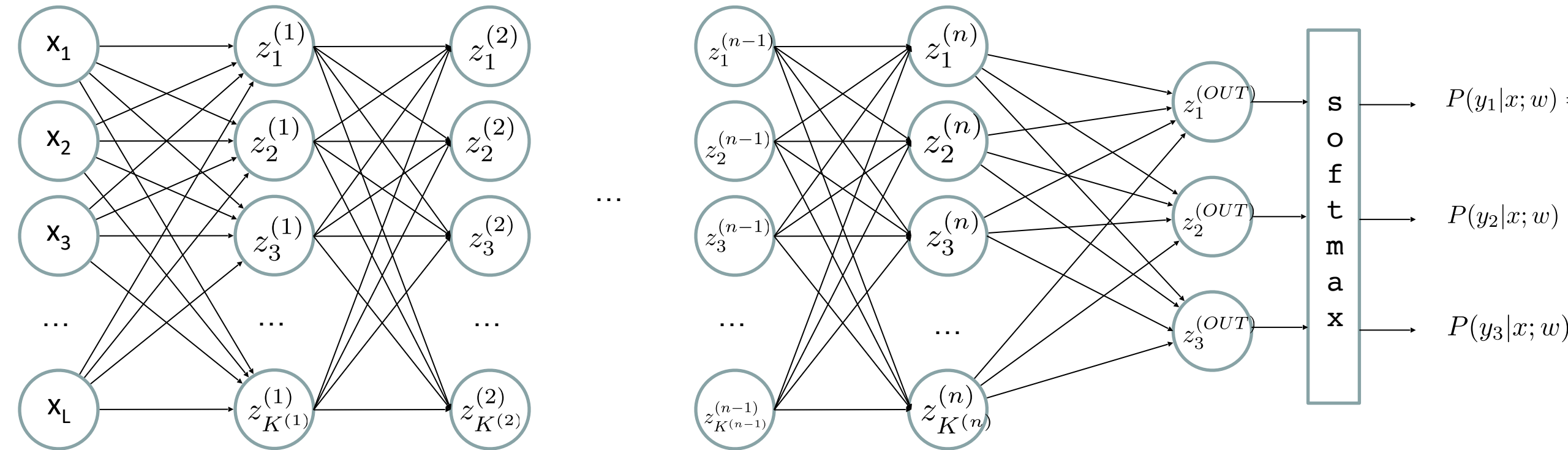
Deep Neural Network = Also learn the features!



$$z_i^{(k)} = g\left(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)}\right)$$

g = nonlinear activation function

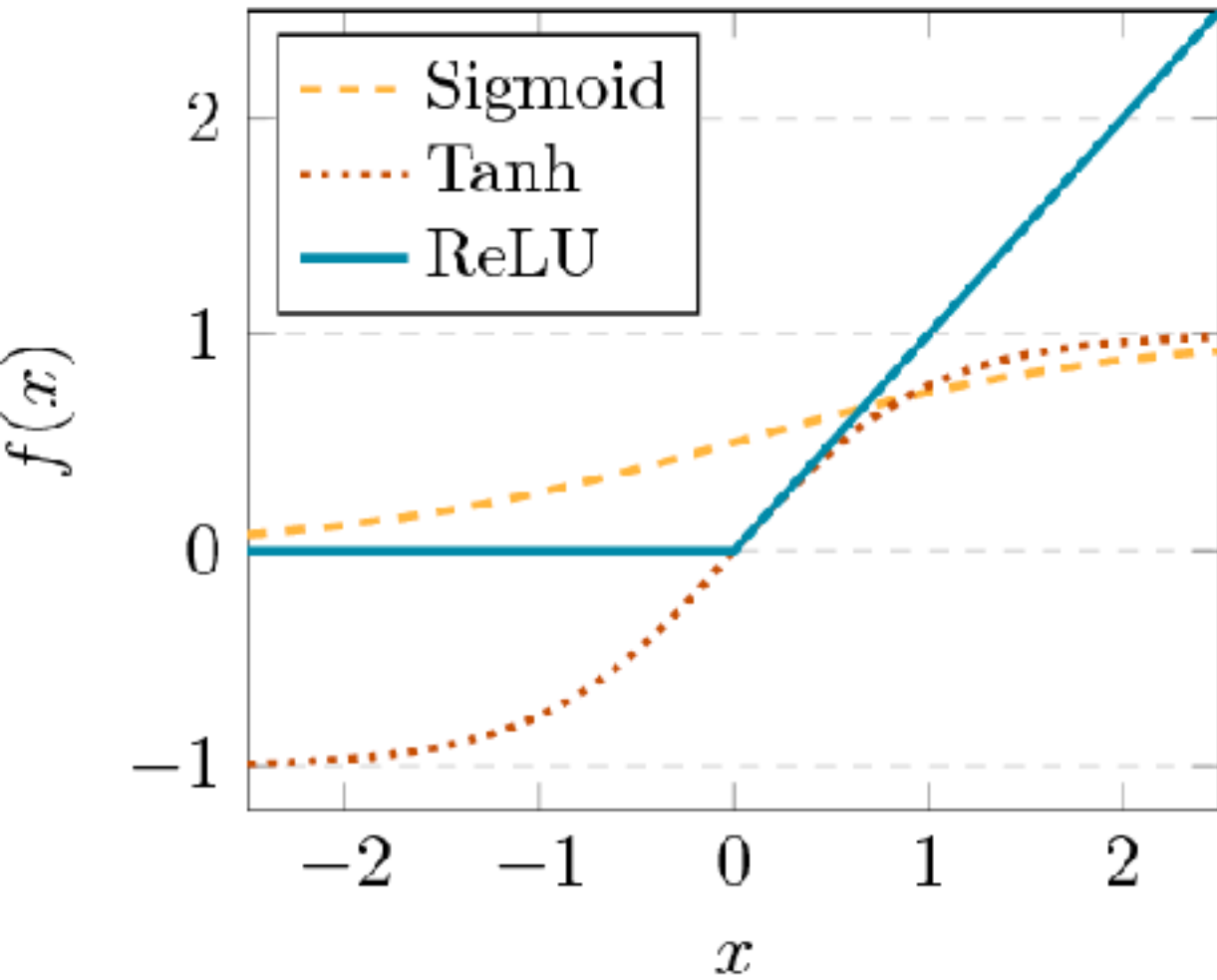
Deep Neural Network = Also learn the features!



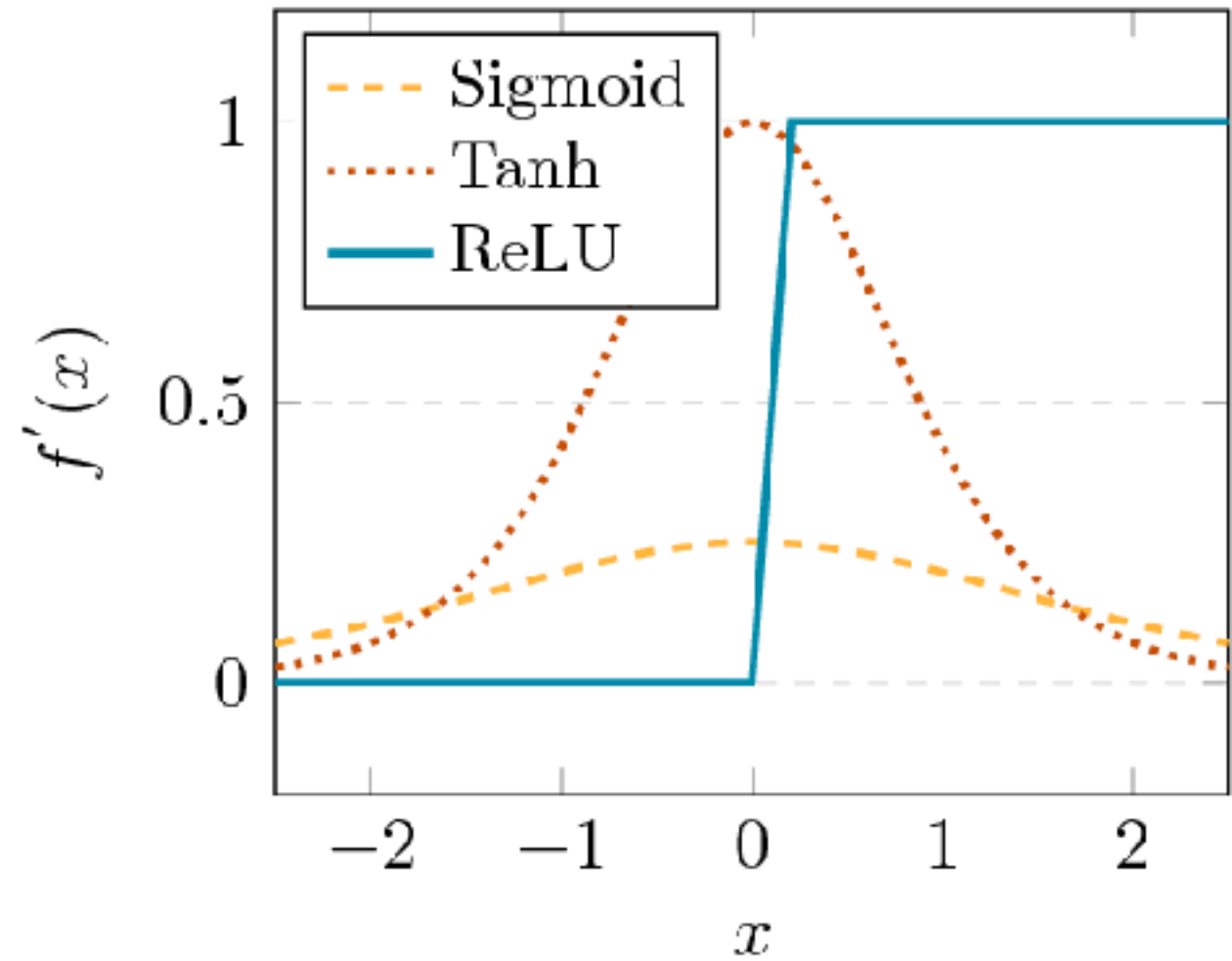
$$z_i^{(k)} = g\left(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)}\right)$$

g = nonlinear activation function

Common Activation Functions



(a) Activation functions



(b) Derivatives

Deep Neural Network: Also Learn the Features!

- Training the deep neural network is just like logistic regression:

$$\max_w ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

Neural Networks Properties

- Theorem (Universal Function Approximators). A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.
- Practical considerations
 - Can be seen as learning the features
 - Large number of neurons
 - Danger for overfitting
 - (hence early stopping!)

Universal Function Approximation Theorem*

Hornik theorem 1: Whenever the activation function is *bounded and nonconstant*, then, for any finite measure μ , standard multilayer feedforward networks can approximate any function in $L^p(\mu)$ (the space of all functions on R^k such that $\int_{R^k} |f(x)|^p d\mu(x) < \infty$) arbitrarily well, provided that sufficiently many hidden units are available.

Hornik theorem 2: Whenever the activation function is *continuous, bounded and non-constant*, then, for arbitrary compact subsets $X \subseteq R^k$, standard multilayer feedforward networks can approximate any continuous function on X arbitrarily well with respect to uniform distance, provided that sufficiently many hidden units are available.

- In words: Given any continuous function $f(x)$, if a 2-layer neural network has enough hidden units, then there is a choice of weights that allow it to closely approximate $f(x)$.

Cybenko (1989) "Approximations by superpositions of sigmoidal functions"

Hornik (1991) "Approximation Capabilities of Multilayer Feedforward Networks"

Leshno and Schocken (1991) "Multilayer Feedforward Networks with Non-Polynomial Activation

Functions Can Approximate Any Function"

Universal Function Approximation*

Math. Control Signals Systems (1989), 2, 303-314

Mathematics of Control,
Signals, and Systems
© 1989 Springer-Verlag New York, Inc.

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko

Abstract. In this paper we demonstrate that finite linear combinations of superpositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube, only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by such neural networks. Our results are valid only in a single hidden layer topology and suggest that more general topologies are necessary. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

1. Introduction

A number of diverse applications areas are concerned with the representation of general functions of n -dimensional real variables $x \in \mathbb{R}^n$, by finite linear combinations of the form

$$\sum_{j=1}^N c_j \sigma(y_j^T x + b_j) \quad (1)$$

where $y_j \in \mathbb{R}^n$ and $c_j, b_j \in \mathbb{R}$ are fixed, y_j^T is the transpose of y_j so that $y_j^T x$ is the inner product of y_j and x . Here the univariate function σ depends heavily on the context of the application. Our major concern is with so-called sigmoidal σ 's

$$\sigma(t) = \begin{cases} 1 & \text{as } t \rightarrow +\infty \\ 0 & \text{as } t \rightarrow -\infty \end{cases}$$

Such functions arise naturally in neural network theory as the activation function of a neural node (or unit as is becoming the preferred term) [Li], [RHW]. The main result of this paper is a demonstration of the fact that sums of the form (1) are dense in the space of continuous functions on the unit cube if σ is any continuous sigmoidal

*Duke received October 21, 1988. Duke revised February 17, 1989. This research was supported in part by NSF Grant DCR-8611803, ONR Contract N00014-86-G-0202 and DOD Grant D64F600-89F00584.

†Center for Supercomputing Research and Development and Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois 61821, U.S.A.

0890-4009/89/0002-0303\$01.00
© 1989 by John Wiley & Sons, Inc.

0890-4009/89/0002-0303\$01.00
© 1989 by John Wiley & Sons, Inc.

ORIGINAL CONTRIBUTION

Approximation Capabilities of Multilayer Feedforward Networks

KURT HORNIK

Yale University, New Haven, Conn.

Received 30 January 1989; revised and accepted 12 October 1989

Abstract. We show that standard multilayer feedforward networks with as few as a single hidden layer and arbitrary bounded and nonpolynomial activation functions are universal approximators with respect to L_2 for bounded criteria, for arbitrary finite input-output measures μ , provided only that sufficiently many hidden units are available. If the activation function is continuous, bounded and nonpolynomial, then continuous mappings can be learned uniformly over compact input sets. We also give very general conditions ensuring that networks with sufficiently smooth activation functions are capable of arbitrarily accurate approximation to a function and its derivatives.

Keywords: Multilayer feedforward networks, Activation functions, Universal approximation capabilities, Input-output measures, L_2 approximation, Uniform approximation, Spline spaces, Smooth approximation.

1. INTRODUCTION

The approximation capabilities of neural networks or, otherwise, have recently been investigated by many authors, including Cornill and Dickinson (1989), Cybenko (1989), Fausch (1989), Gallant and White (1988), Hecht-Nielsen (1989), Hornik, Stinchcombe, and White (1989, 1990), Liu and Wu (1988), Lapodas and Farris (1989), Stinchcombe and White (1989, 1990). [This list is by no means complete.]

If we think of the network architecture as a rule for computing values at l output units given values at k input units, hence implementing a class of mappings from \mathbb{R}^k to \mathbb{R}^l , we can ask how well arbitrary mappings from \mathbb{R}^k to \mathbb{R}^l can be approximated by the network. In particular, if as many hidden units as required internal representation and computation may be employed.

How to measure the accuracy of approximation depends on how we measure distances between functions, which in turn varies significantly with the specific problem to be dealt with. In many applications it is necessary to have the network perform well on all input samples taken from some compact input set \mathcal{X} in \mathbb{R}^k . In this case, closeness is

measured by the uniform distance between functions on \mathcal{X} , that is,

$$\|f, g\|_\infty = \sup_{x \in \mathcal{X}} |f(x) - g(x)|$$

In other applications, we think of the inputs as random variables and are interested in the mean square distance where the average is taken with respect to the input distribution measure μ , where $\mu(\mathcal{X}) = 1$. In this case, closeness is measured by the $L_2(\mu)$ distance

$$\|f, g\|_2 = \left(\int_{\mathcal{X}} |f(x) - g(x)|^2 d\mu(x) \right)^{1/2}$$

$1 \leq p < \infty$, the most popular choice being $p = 2$, corresponding to mean square error.

Of course, there are many other ways of measuring distances of functions. In particular, in many applications, it is also necessary that the derivatives of the approximating function implemented by the network closely resemble those of the function to be approximated, up to some order. This issue was first taken up in Hornik et al. (1990), who discuss the question of neural smooth function approximation in more detail. Typical examples arise in robotics (learning of smooth movements) and signal processing (analysis of chaotic time series); for a recent application to problems of nonparametric inference in statistics and econometrics, see Gallant and White (1988).

All papers establishing certain approximation cap-

MULTILAYER FEEDFORWARD NETWORKS WITH NON-POLYNOMIAL ACTIVATION FUNCTIONS CAN APPROXIMATE ANY FUNCTION

by

Moshe Leshno
Faculty of Management
Tel Aviv University
Tel Aviv, Israel 69978

and

Shimon Schocken
Leonard N. Stern School of Business
New York University
New York, NY 10003

September 1991

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN 91-91-26

Appeared previously as Working Paper No. 81/91 at The Israel Institute Of Business Research

Cybenko (1989) "Approximations by superpositions of sigmoidal functions"
Hornik (1991) "Approximation Capabilities of Multilayer Feedforward Networks"
Leshno and Schocken (1991) "Multilayer Feedforward Networks with Non-Polynomial Activation Functions Can Approximate Any Function"

Fun Neural Net Demo Site

- Demo-site:
 - <http://playground.tensorflow.org/>

How about computing all the derivatives?

■ Derivatives tables:

$$\frac{d}{dx}(a) = 0$$

$$\frac{d}{dx}(x) = 1$$

$$\frac{d}{dx}(au) = a \frac{du}{dx}$$

$$\frac{d}{dx}(u + v - w) = \frac{du}{dx} + \frac{dv}{dx} - \frac{dw}{dx}$$

$$\frac{d}{dx}(uv) = u \frac{dv}{dx} + v \frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{u}{v}\right) = \frac{1}{v} \frac{du}{dx} - \frac{u}{v^2} \frac{dv}{dx}$$

$$\frac{d}{dx}(u^n) = nu^{n-1} \frac{du}{dx}$$

$$\frac{d}{dx}(\sqrt{u}) = \frac{1}{2\sqrt{u}} \frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{1}{u}\right) = -\frac{1}{u^2} \frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{1}{u^n}\right) = -\frac{n}{u^{n+1}} \frac{du}{dx}$$

$$\frac{d}{dx}[f(u)] = \frac{d}{du}[f(u)] \frac{du}{dx}$$

$$\frac{d}{dx}[\ln u] = \frac{d}{dx}[\log_e u] = \frac{1}{u} \frac{du}{dx}$$

$$\frac{d}{dx}[\log_a u] = \log_a e \frac{1}{u} \frac{du}{dx}$$

$$\frac{d}{dx}e^u = e^u \frac{du}{dx}$$

$$\frac{d}{dx}a^u = a^u \ln a \frac{du}{dx}$$

$$\frac{d}{dx}(u^v) = v u^{v-1} \frac{du}{dx} + \ln u \cdot u^v \frac{dv}{dx}$$

$$\frac{d}{dx} \sin u = \cos u \frac{du}{dx}$$

$$\frac{d}{dx} \cos u = -\sin u \frac{du}{dx}$$

$$\frac{d}{dx} \tan u = \sec^2 u \frac{du}{dx}$$

$$\frac{d}{dx} \cot u = -\csc^2 u \frac{du}{dx}$$

$$\frac{d}{dx} \sec u = \sec u \tan u \frac{du}{dx}$$

$$\frac{d}{dx} \csc u = -\csc u \cot u \frac{du}{dx}$$

How about computing all the derivatives?

- But neural net f is never one of those?
 - No problem: CHAIN RULE:

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

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

→ **Derivatives can be computed by following well-defined procedures**

Automatic Differentiation

- Automatic differentiation software

- e.g. Theano, TensorFlow, PyTorch, Chainer
- Only need to program the function $g(x,y,w)$
- Can automatically compute all derivatives w.r.t. all entries in w
- This is typically done by caching info during forward computation pass of f , and then doing a backward pass = “backpropagation”
- Autodiff / Backpropagation can often be done at computational cost comparable to the forward pass

Summary of Key Ideas

- Optimize probability of label given input

$$\max_w ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

- Continuous optimization

- Gradient ascent:

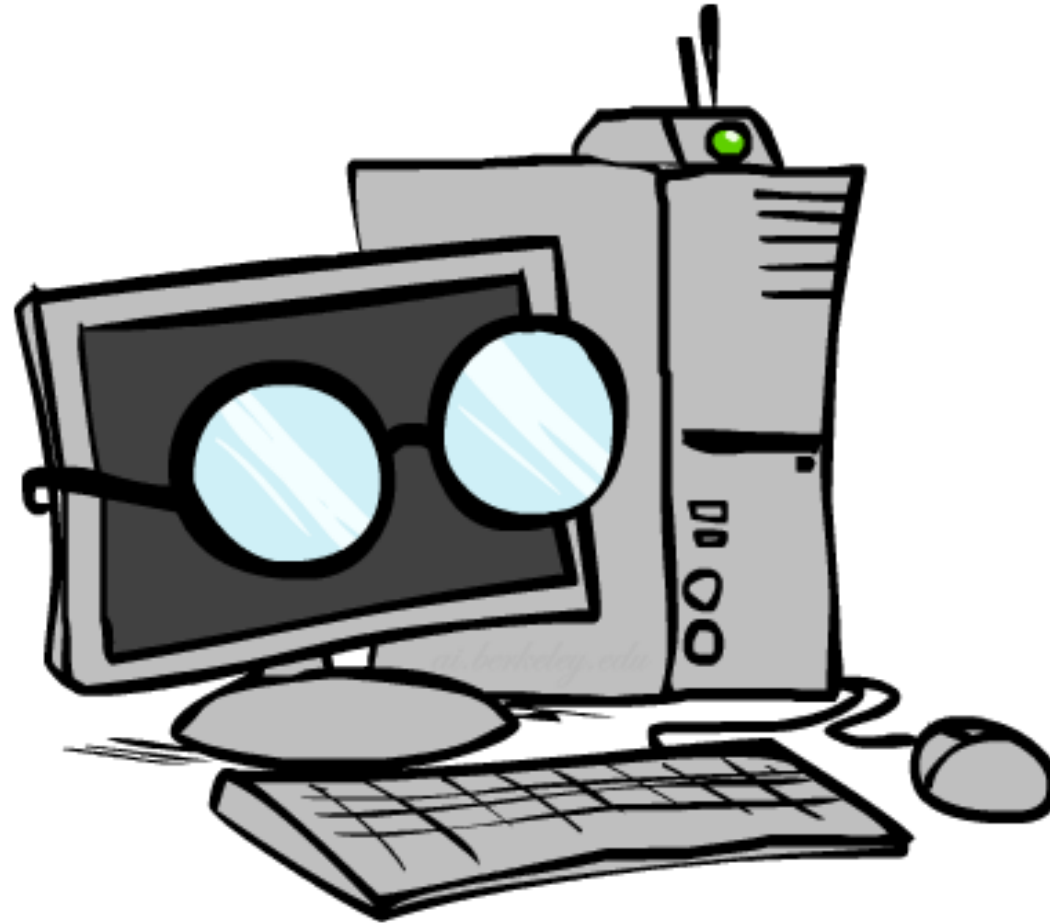
- Compute steepest uphill direction = gradient (= just vector of partial derivatives)
 - Take step in the gradient direction
 - Repeat (until held-out data accuracy starts to drop = “early stopping”)

- Deep neural nets

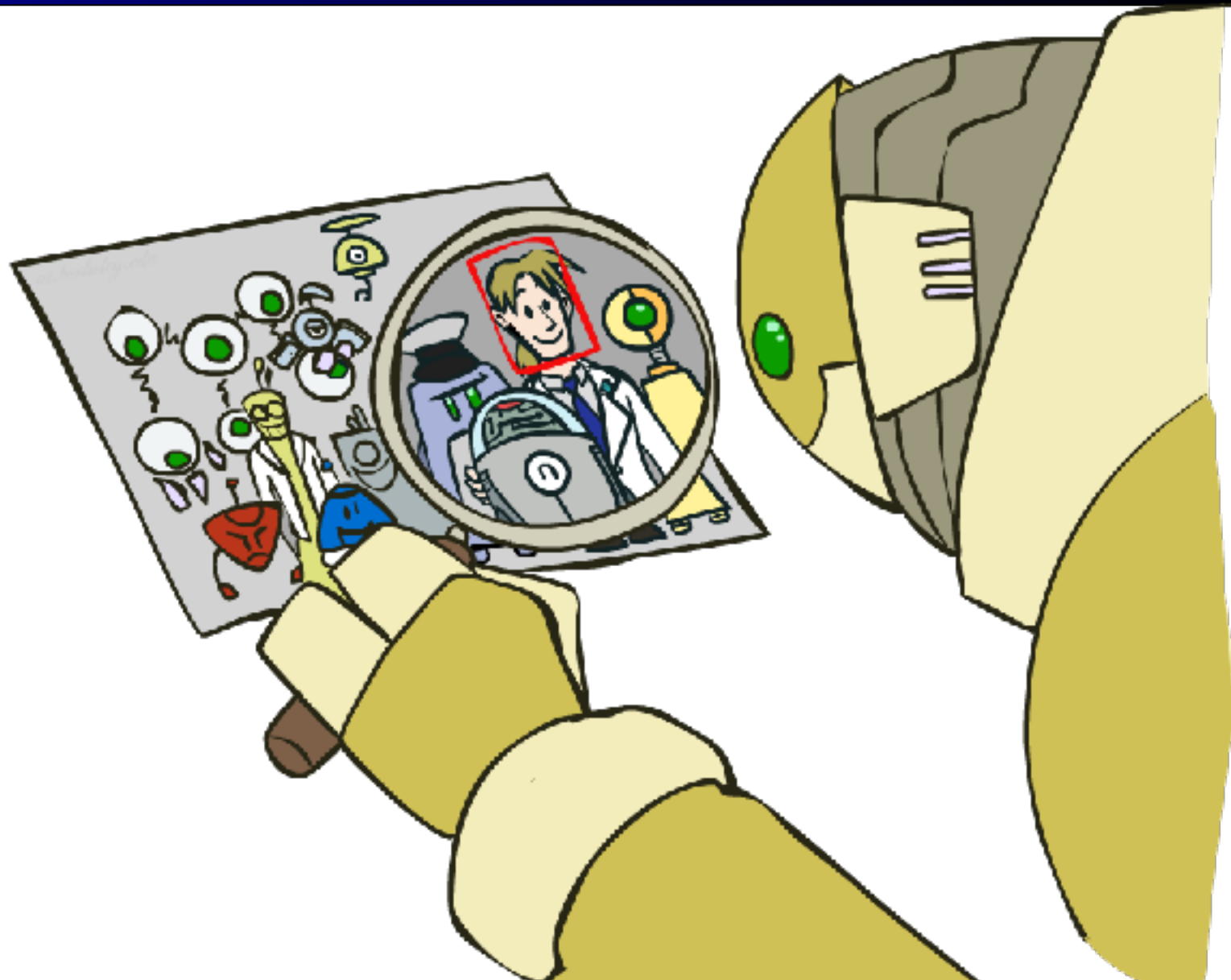
- Last layer = still logistic regression
 - Now also many more layers before this last layer
 - = computing the features
 - → the features are learned rather than hand-designed
 - Universal function approximation theorem
 - If neural net is large enough
 - Then neural net can represent any continuous mapping from input to output with arbitrary accuracy
 - But remember: need to avoid overfitting / memorizing the training data → early stopping!
 - Automatic differentiation gives the derivatives efficiently

How well does it work?

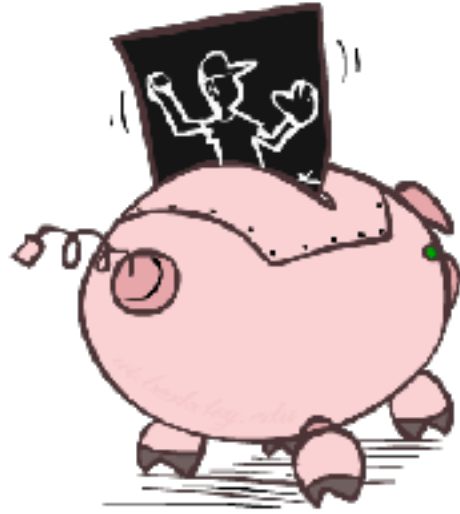
Computer Vision



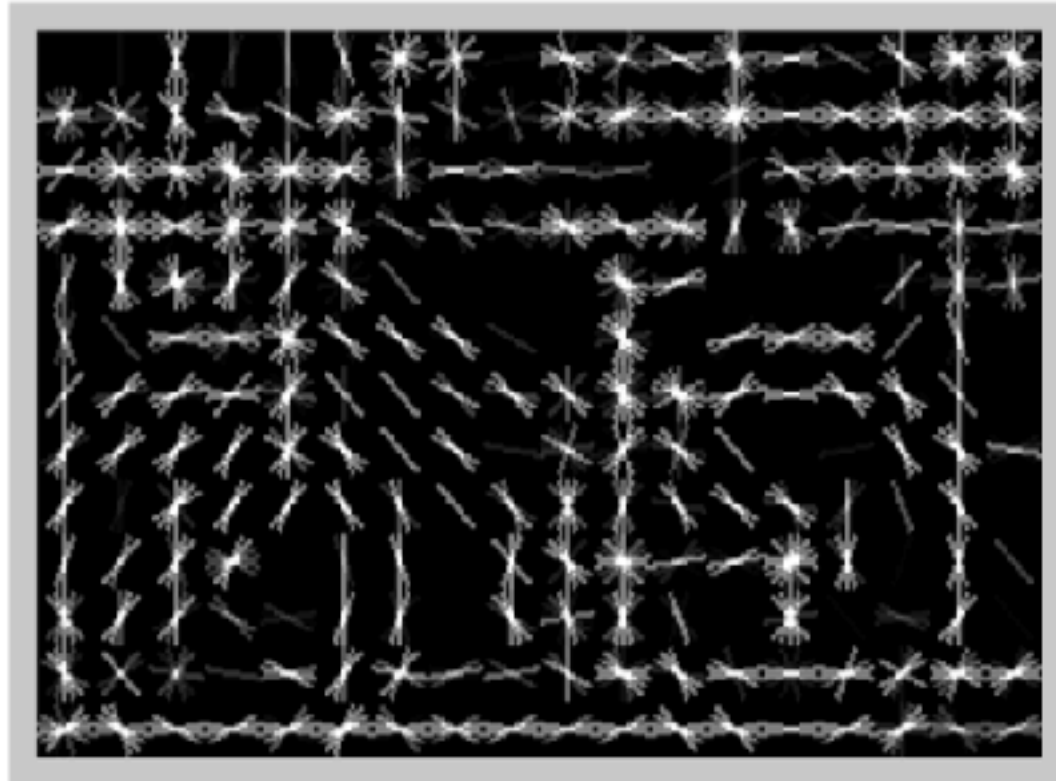
Object Detection



Manual Feature Design



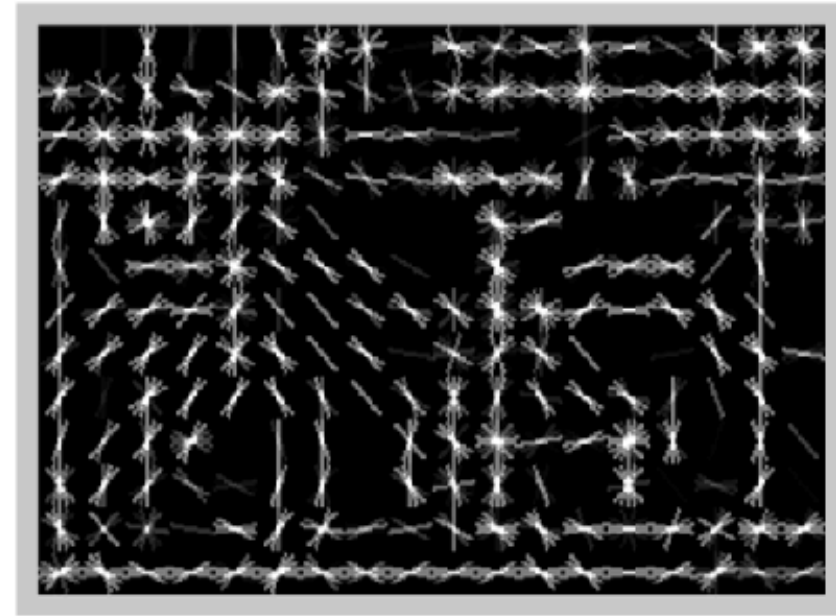
Features and Generalization



Features and Generalization



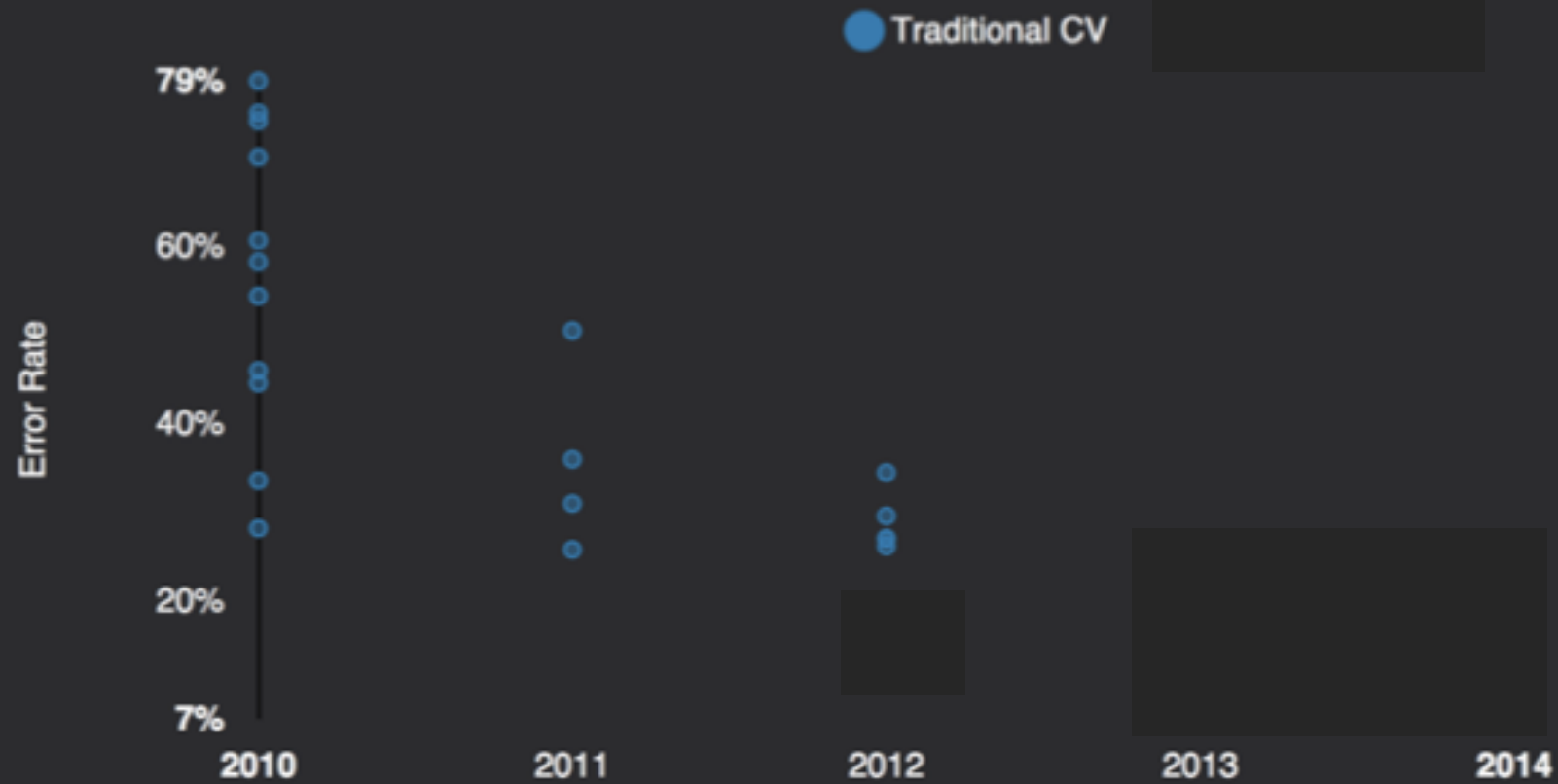
Image



HoG

Performance

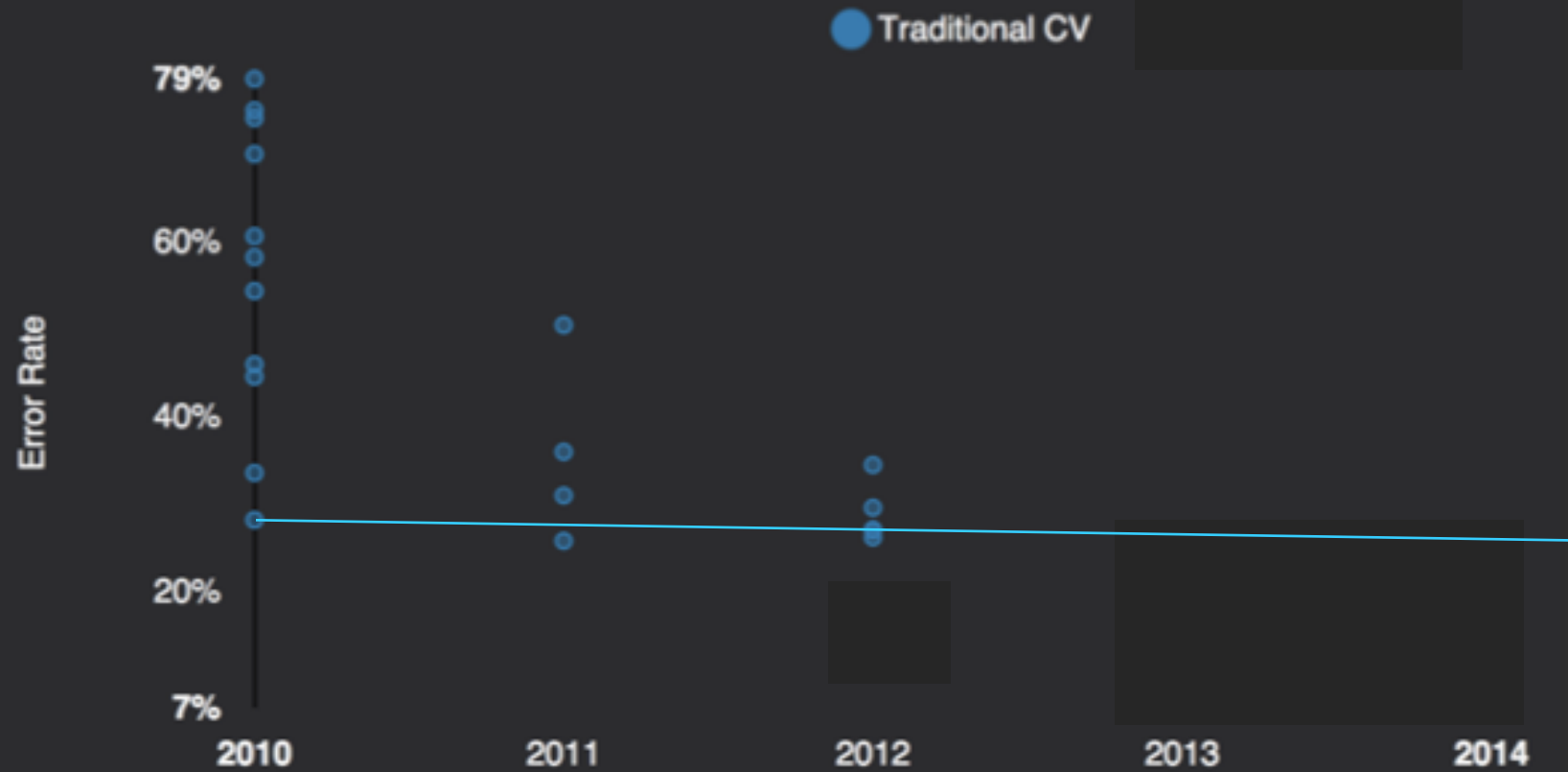
ImageNet Error Rate 2010-2014



graph credit Matt
Zeiler, Clarifai

Performance

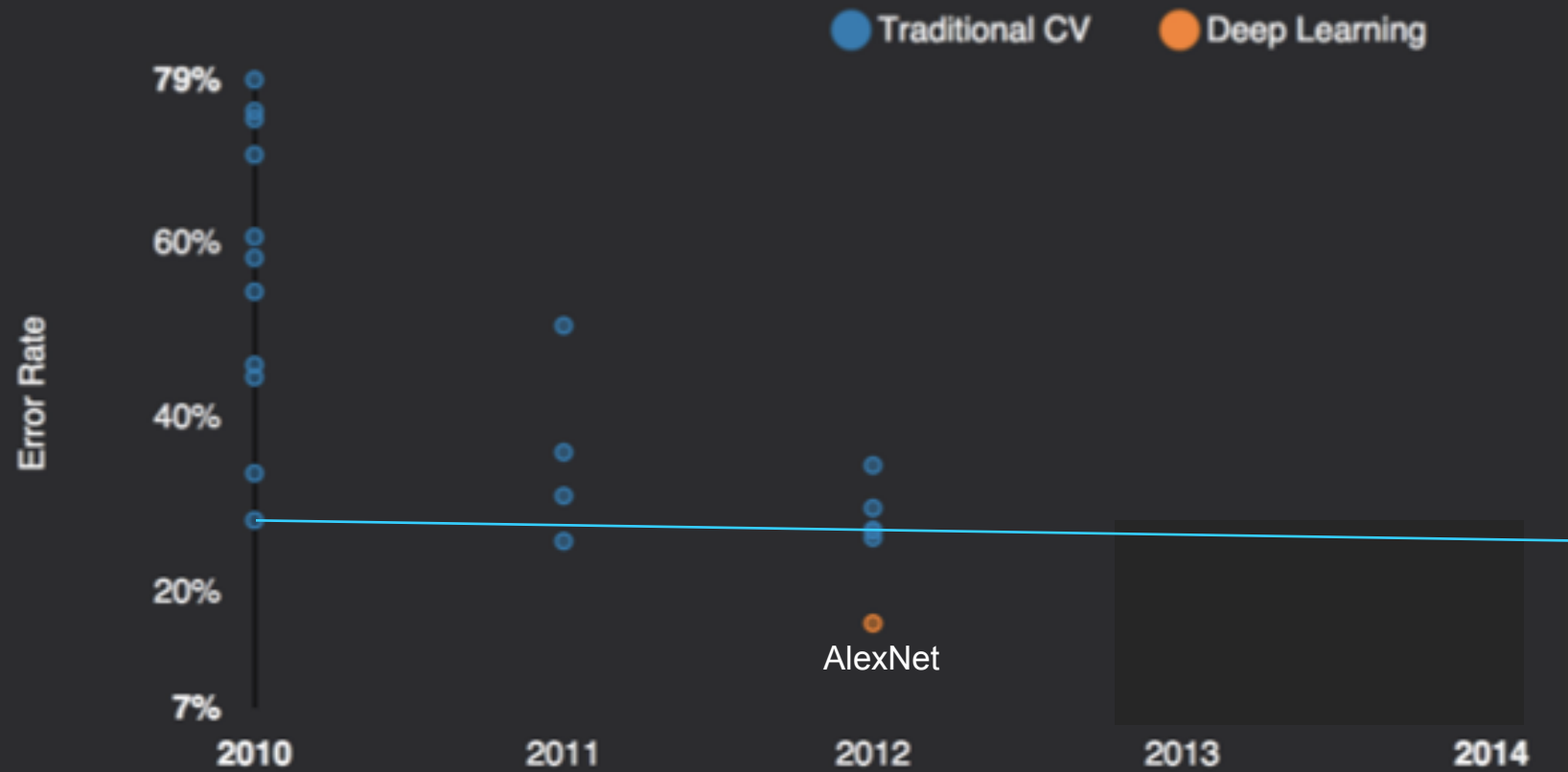
ImageNet Error Rate 2010-2014



graph credit Matt
Zeiler, Clarifai

Performance

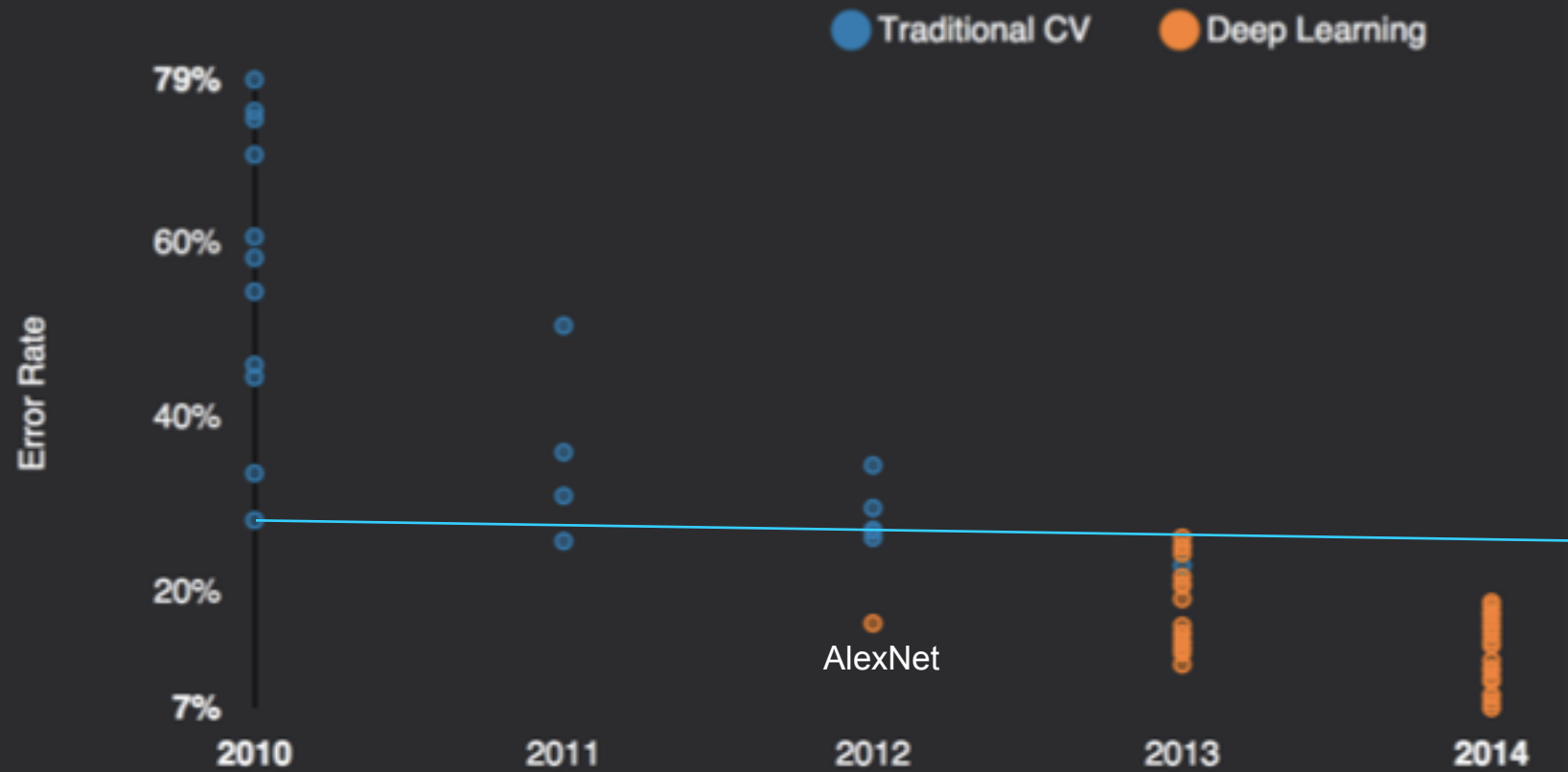
ImageNet Error Rate 2010-2014



graph credit Matt
Zeiler, Clarifai

Performance

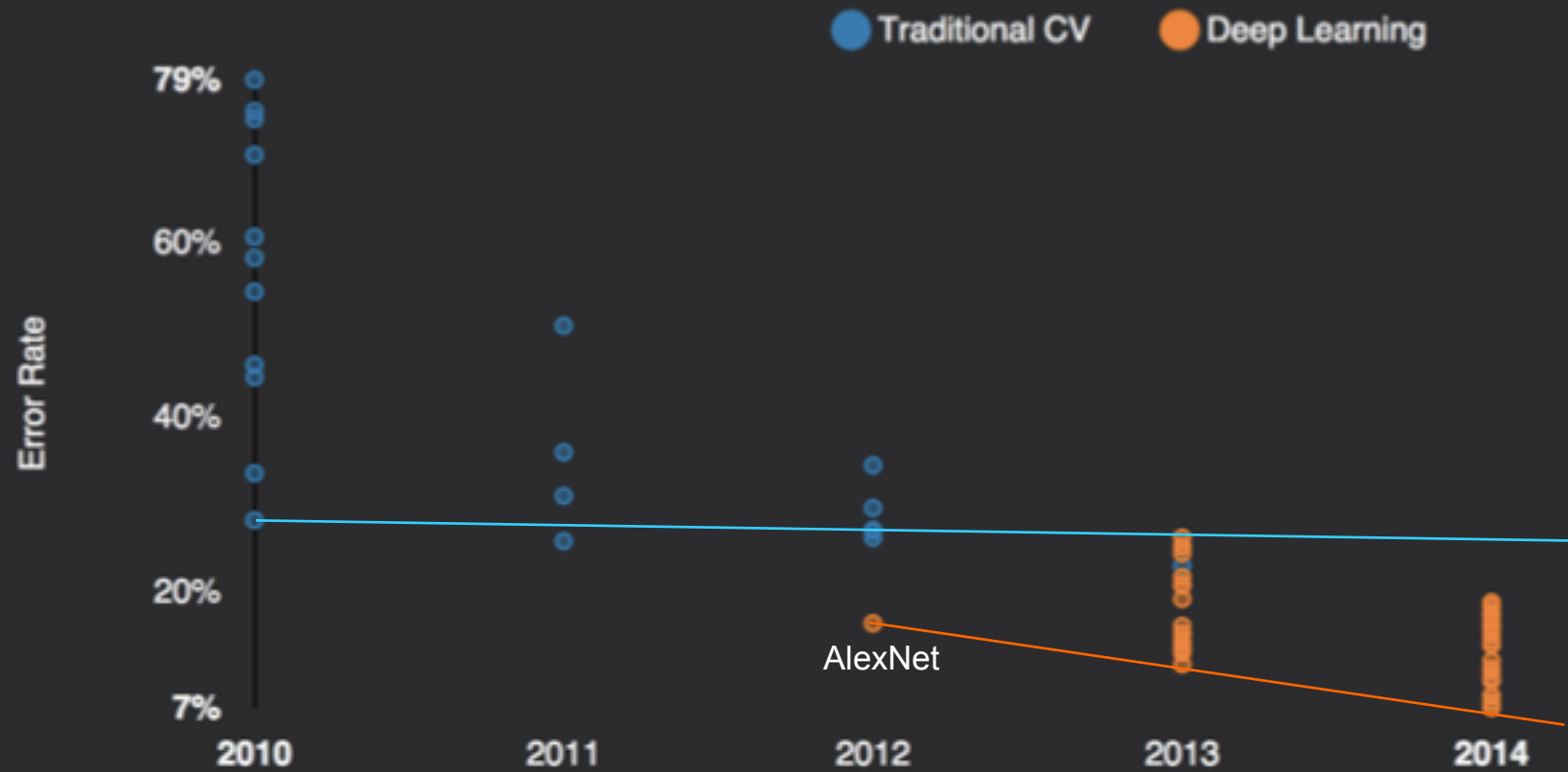
ImageNet Error Rate 2010-2014



graph credit Matt Zeiler, Clarifai

Performance

ImageNet Error Rate 2010-2014



graph credit Matt Zeiler, Clarifai

MS COCO Image Captioning Challenge



"man in black shirt is playing guitar."



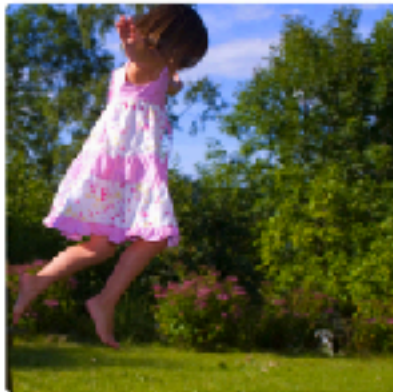
"construction worker in orange safety vest is working on road."



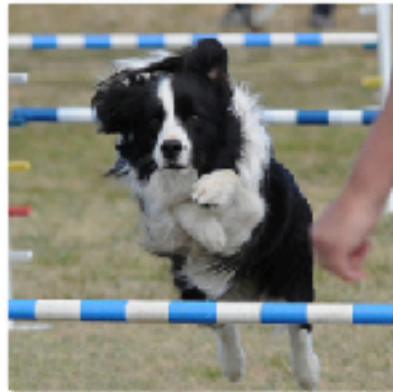
"two young girls are playing with lego toy."



"boy is doing backflip on wakeboard."



"girl in pink dress is jumping in air."



"black and white dog jumps over bar."



"young girl in pink shirt is swinging on swing."



"man in blue wetsuit is surfing on wave."

Karpathy & Fei-Fei, 2015; Donahue et al., 2015; Xu et al, 2015; many more

Visual QA Challenge

Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick, Devi Parikh



What vegetable is on the plate?

Neural Net: broccoli

Ground Truth: broccoli



What color are the shoes on the person's feet ?

Neural Net: brown

Ground Truth: brown



How many school busses are there?

Neural Net: 2

Ground Truth: 2



What sport is this?

Neural Net: baseball

Ground Truth: baseball



What is on top of the refrigerator?

Neural Net: magnets

Ground Truth: cereal



What uniform is she wearing?

Neural Net: shorts

Ground Truth: girl scout



What is the table number?

Neural Net: 4

Ground Truth: 40



What are people sitting under in the back?

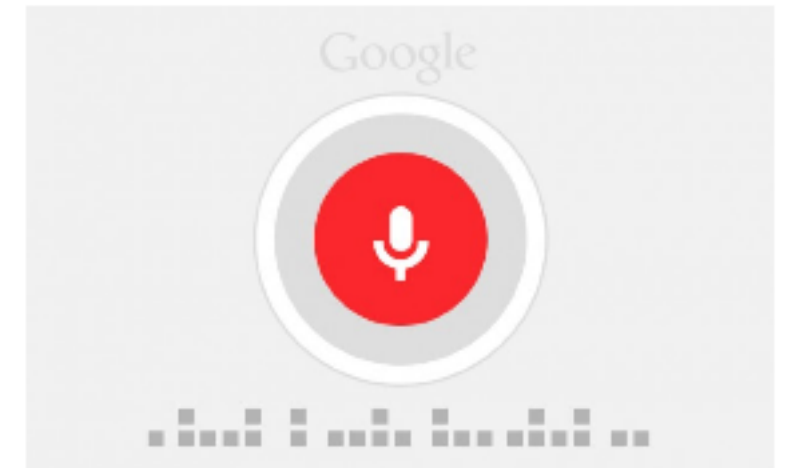
Neural Net: bench

Ground Truth: tent

Speech Recognition

TIMIT Speech Recognition

● Traditional ● Deep Learning



graph credit Matt Zeiler, Clarifai

Machine Translation

Google Neural Machine Translation (in production)

