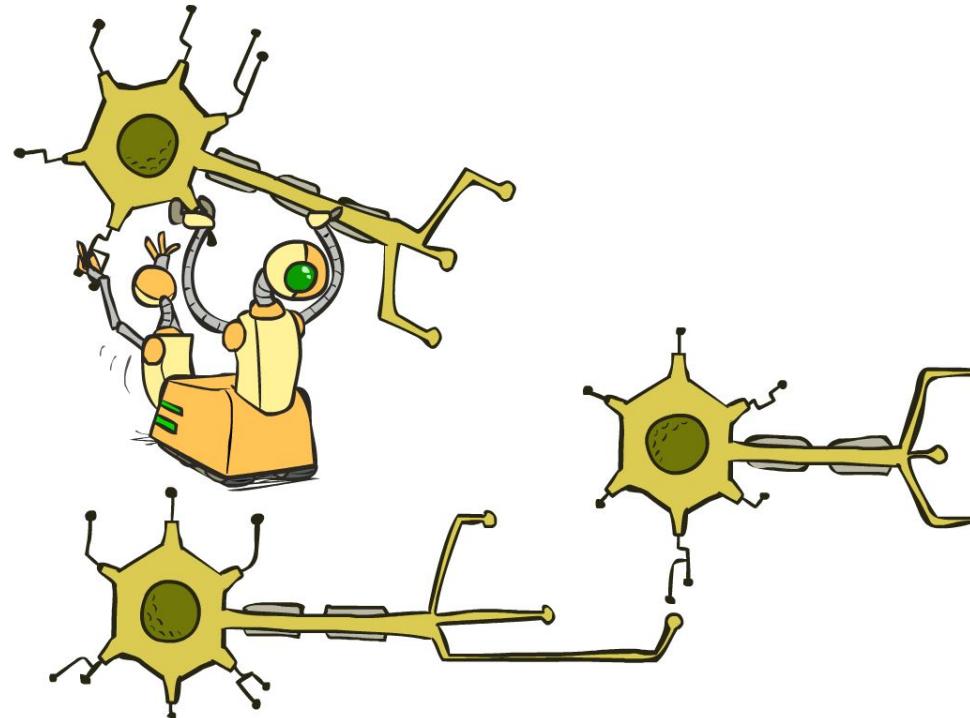


CS 4700: Foundations of Artificial Intelligence

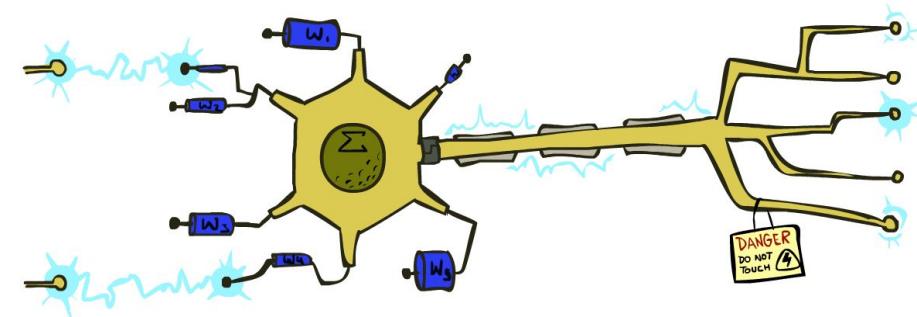
Optimization and Neural Nets



Instructors: Kevin Ellis --- Cornell University

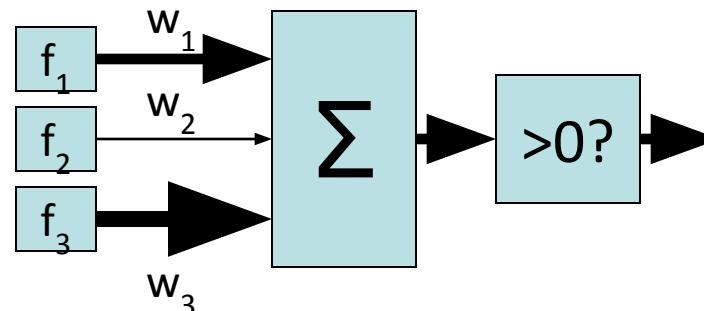
Reminder: Linear Classifiers

- 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



Cornell History

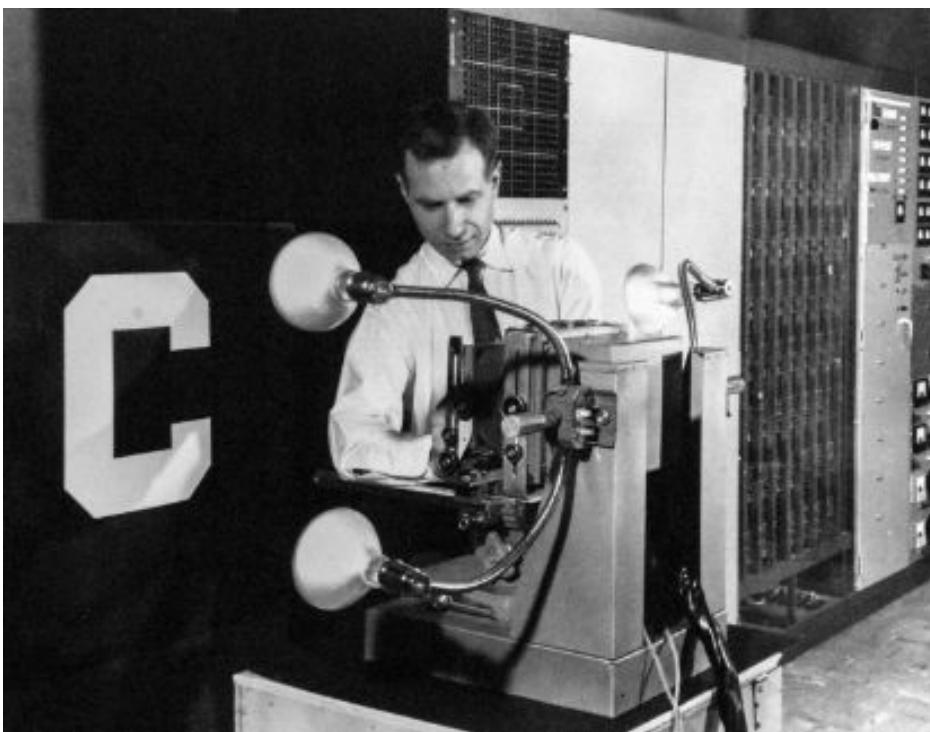


FIG. 1 — Organization of a biological brain. (Red areas indicate active cells, responding to the letter X.)

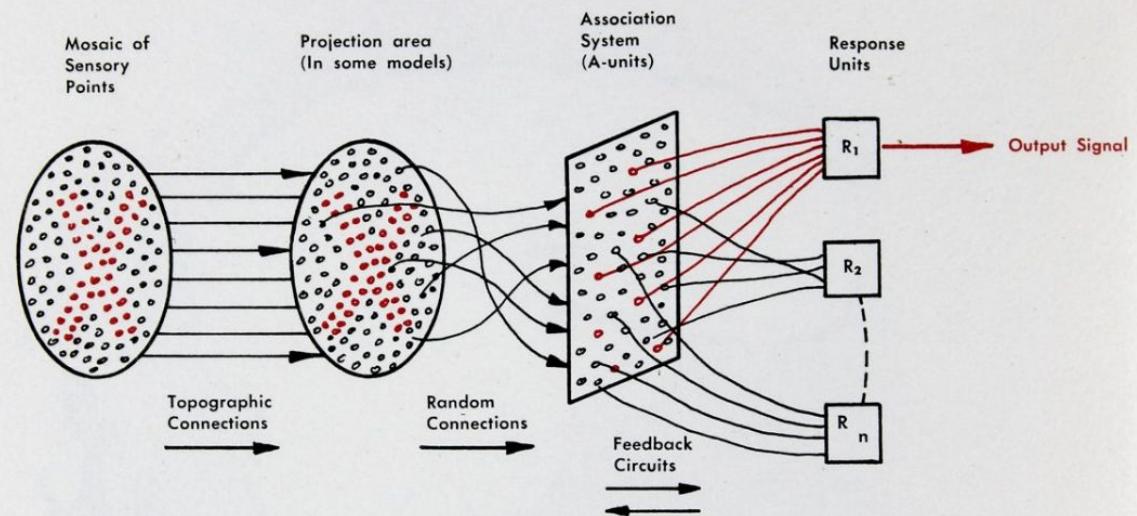


FIG. 2 — Organization of a perceptron.

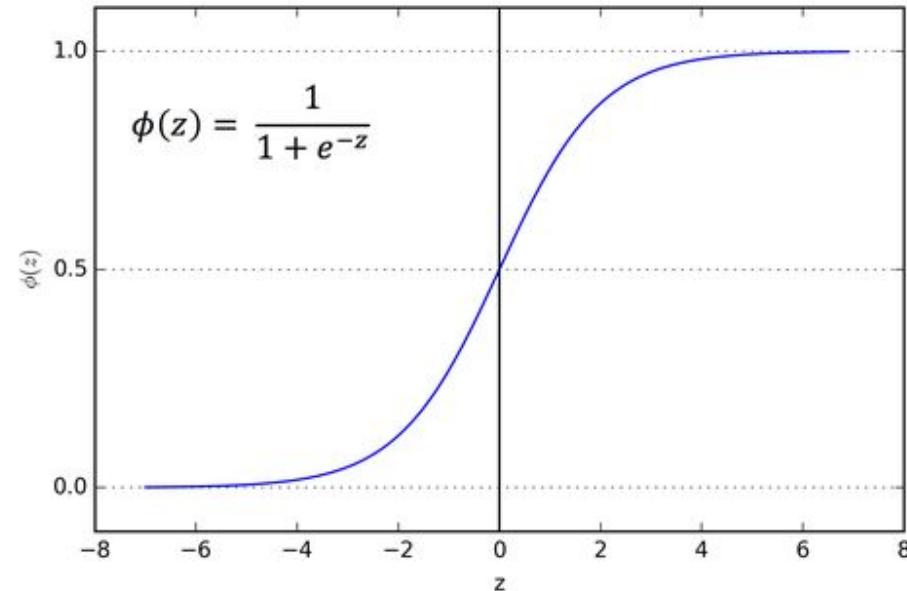
<https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon>

Creator “majored in social psychology and earned his Ph.D. in psychology”, then stayed to become a Cornell professor

How to get probabilistic decisions?

- Perceptron scoring: $z = w \cdot f(x)$
- 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:

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

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

- Recall Perceptron:

- A weight vector for each class:

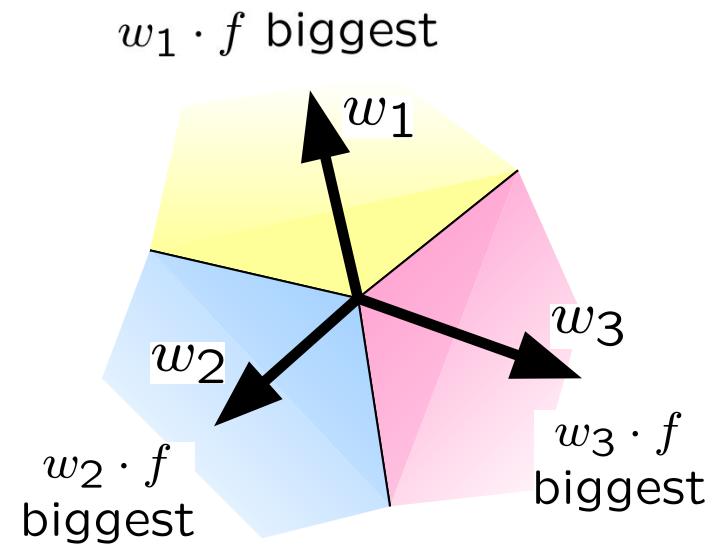
$$w_y$$

- Score (activation) of a class y :

$$w_y \cdot f(x)$$

- Prediction highest score wins

$$y = \arg \max_y w_y \cdot f(x)$$



- How to make the scores into probabilities?

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

original activations

Best w?

- Maximum likelihood estimation:

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

with:

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

= Multi-Class Logistic Regression

This Lecture

- Optimization
 - i.e., how do we solve:

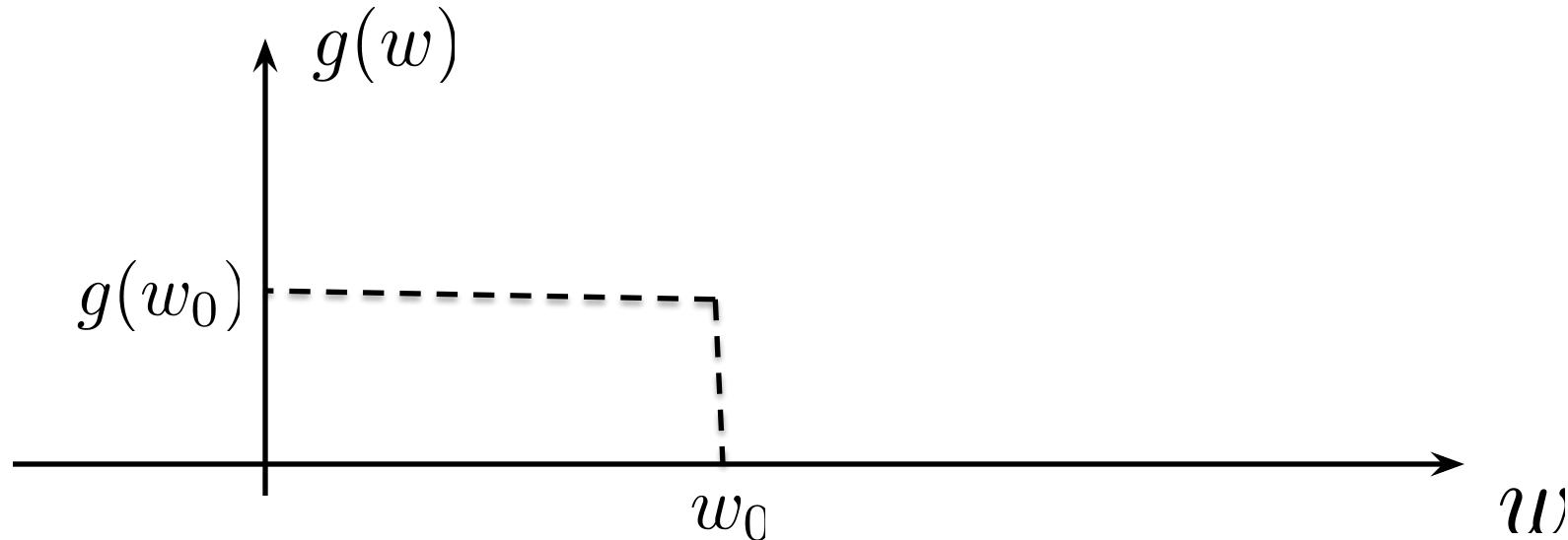
$$\max_w \text{ll}(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

Hill Climbing

- Recall: 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



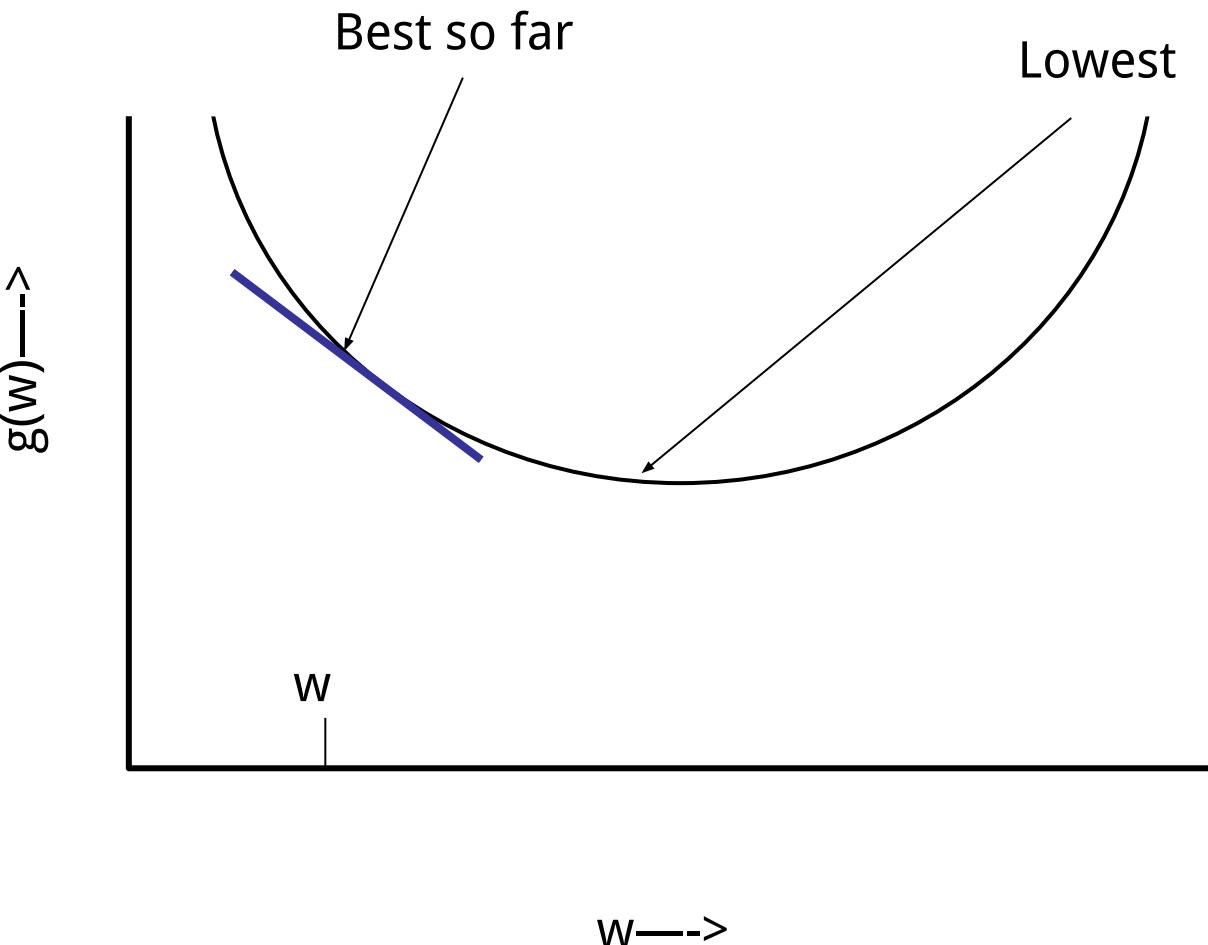
- Could evaluate $g(w_0 + h)$ and $g(w_0 - h)$
 - Then step in best direction
- Or, evaluate derivative: $\frac{\partial g(w_0)}{\partial w} = \lim_{h \rightarrow 0} \frac{g(w_0 + h) - g(w_0 - h)}{2h}$
 - Tells which direction to step into

Minimizing vs Maximizing

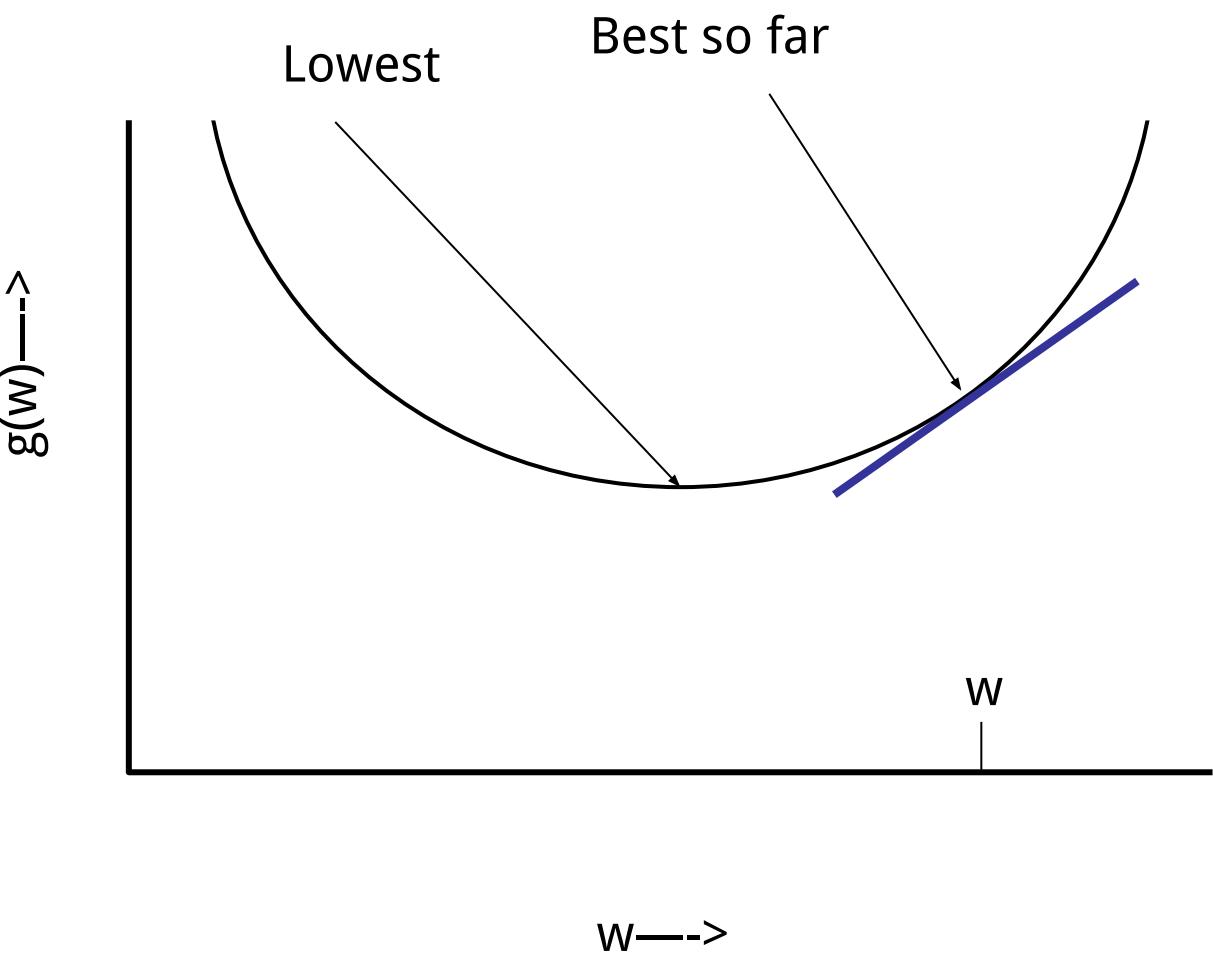
Gradient **Ascent**: Maximize Log Likelihood (LL)

Gradient **Descent**: Minimize “loss” ($\text{loss}=-\text{LL}$)

(you can also review lecture 4 for extra info on gradient descent)

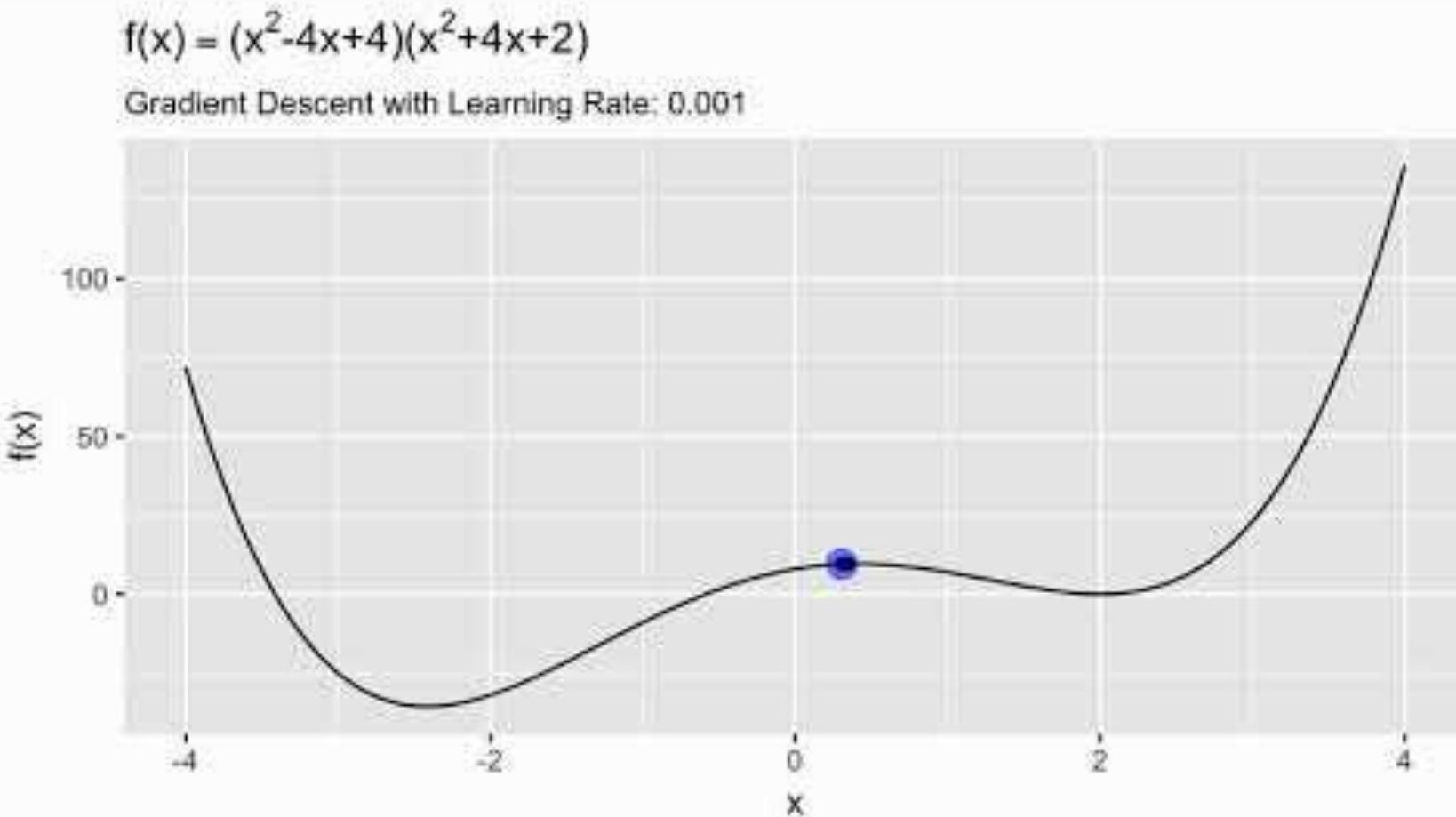


Negative derivative, increase w to decrease $g(w)$

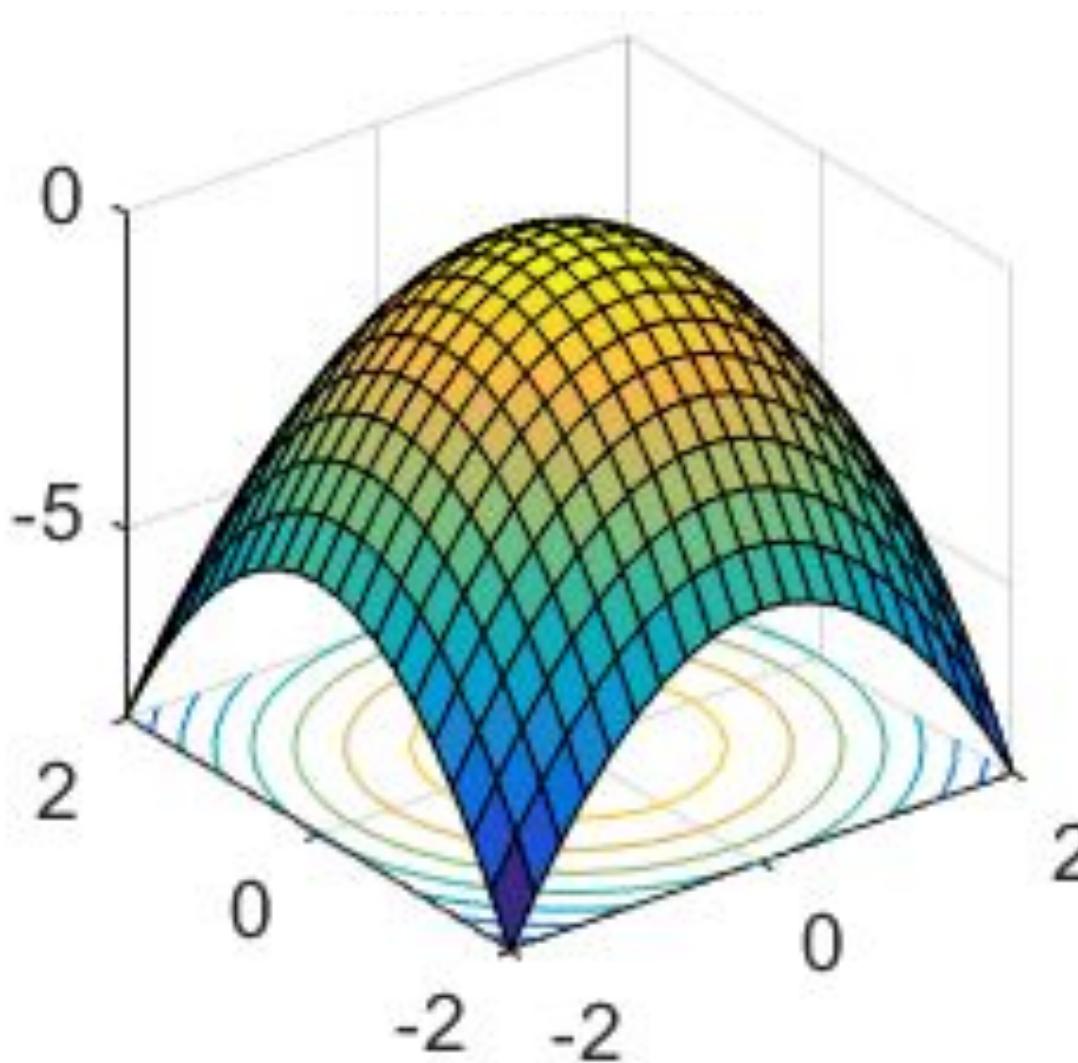


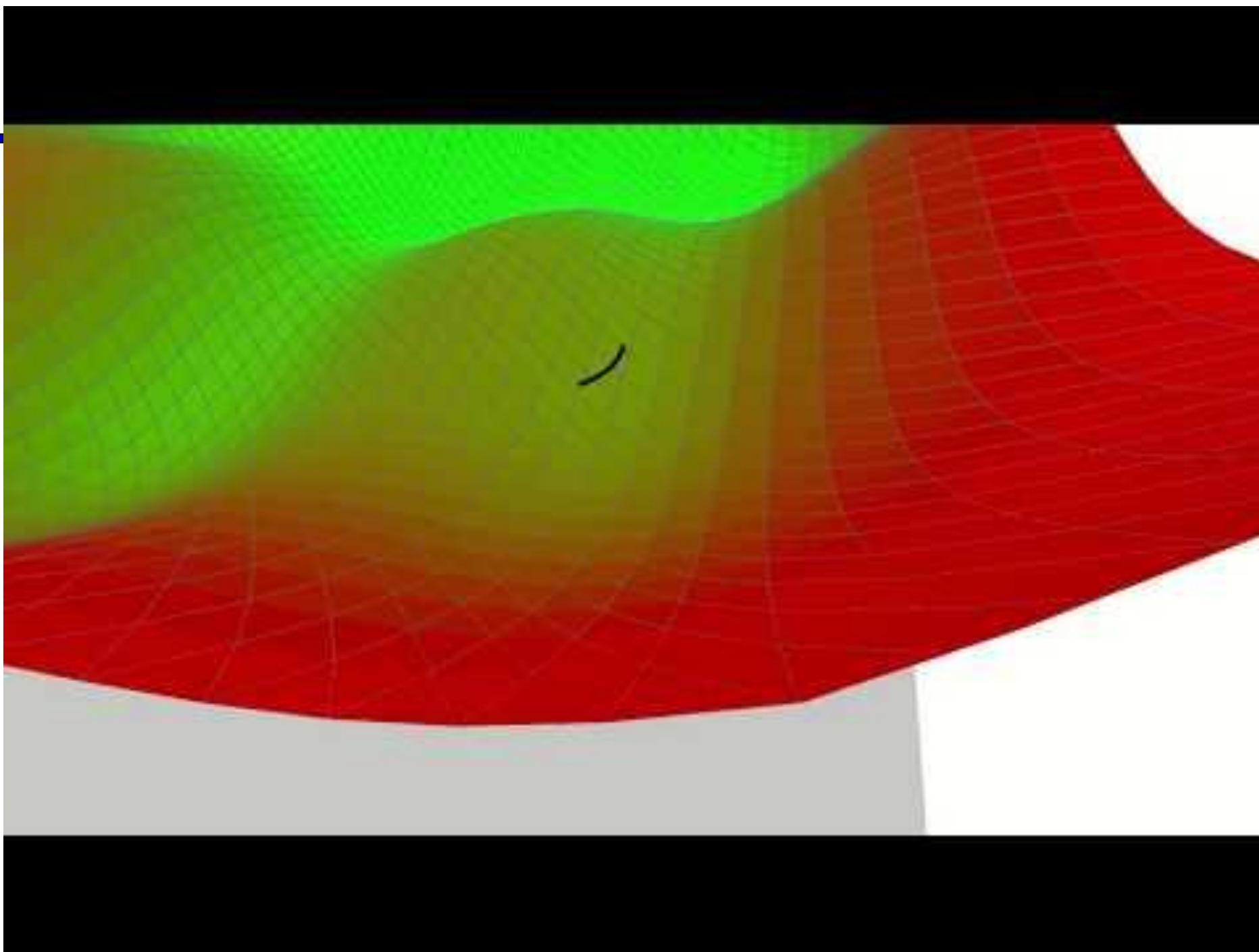
Positive derivative, decrease w
to decrease $g(w)$

Gradient Descent ~ Falling Downhill



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:

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

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

Gradient Ascent

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

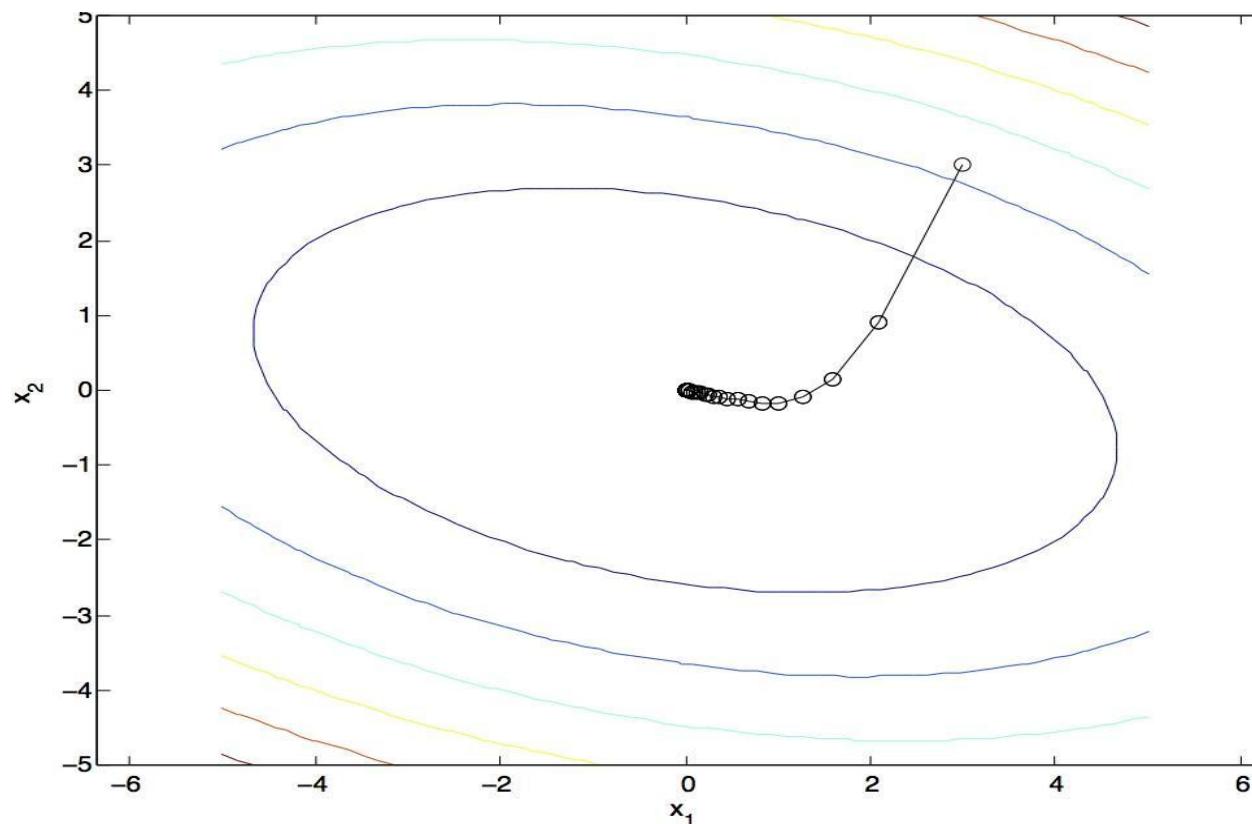


Figure source: Mathworks

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 Descent 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

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

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 %

Optimization Procedure: Gradient *Descent*

- init w
- for iter = 1, 2, ...

$$w \leftarrow w - \alpha * \nabla g(w)$$

Batch Gradient Ascent on the Log Likelihood Objective

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

- init w
- for iter = 1, 2, ...

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

Stochastic Gradient Ascent on the Log Likelihood Objective

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

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

- `init w`
- `for iter = 1, 2, ...`
 - pick random j

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

Mini-Batch Gradient Ascent on the Log Likelihood Objective

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

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

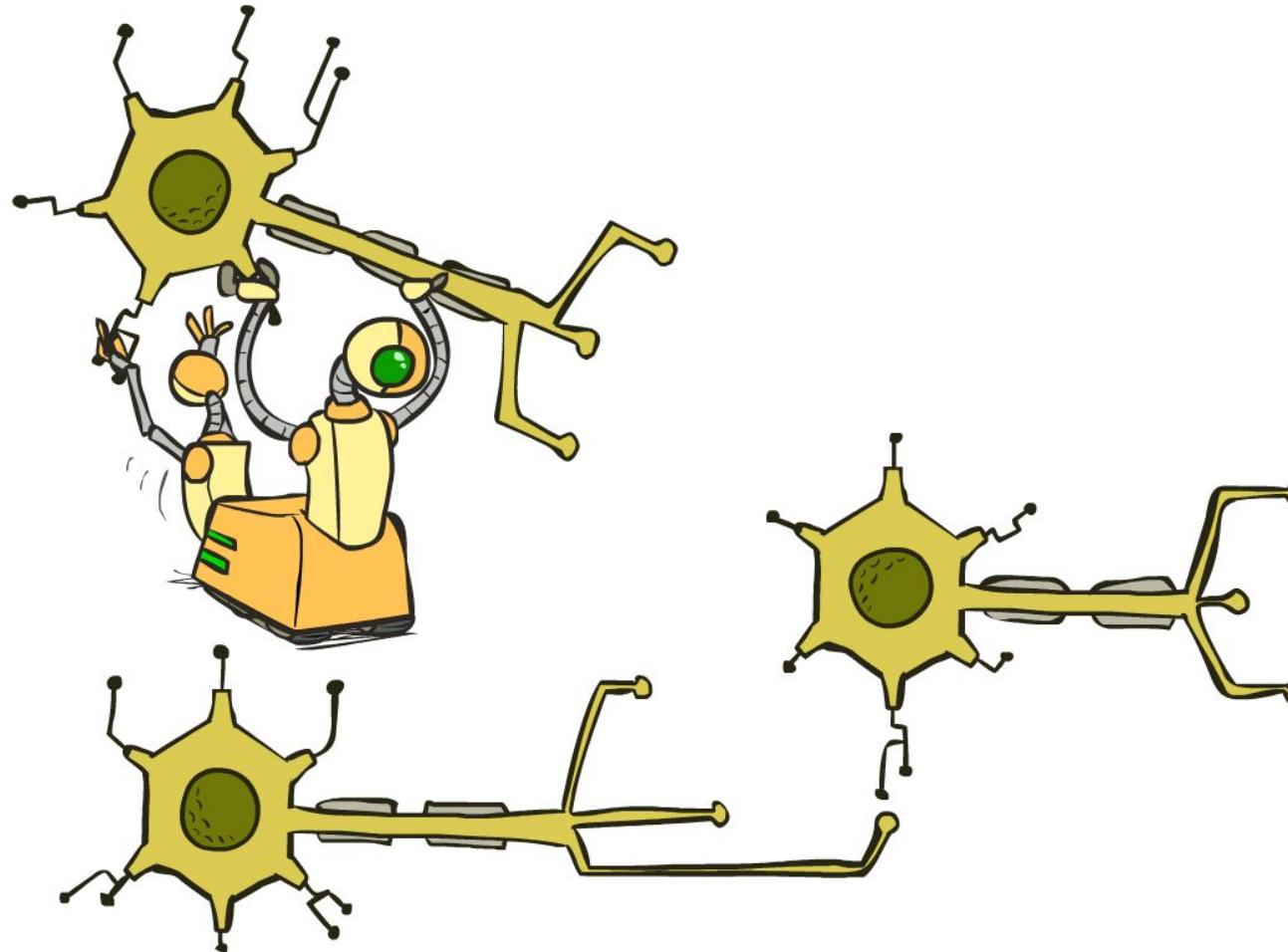
- `init w`
- `for iter = 1, 2, ...`
 - pick random subset of training examples J

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

How about computing all the derivatives?

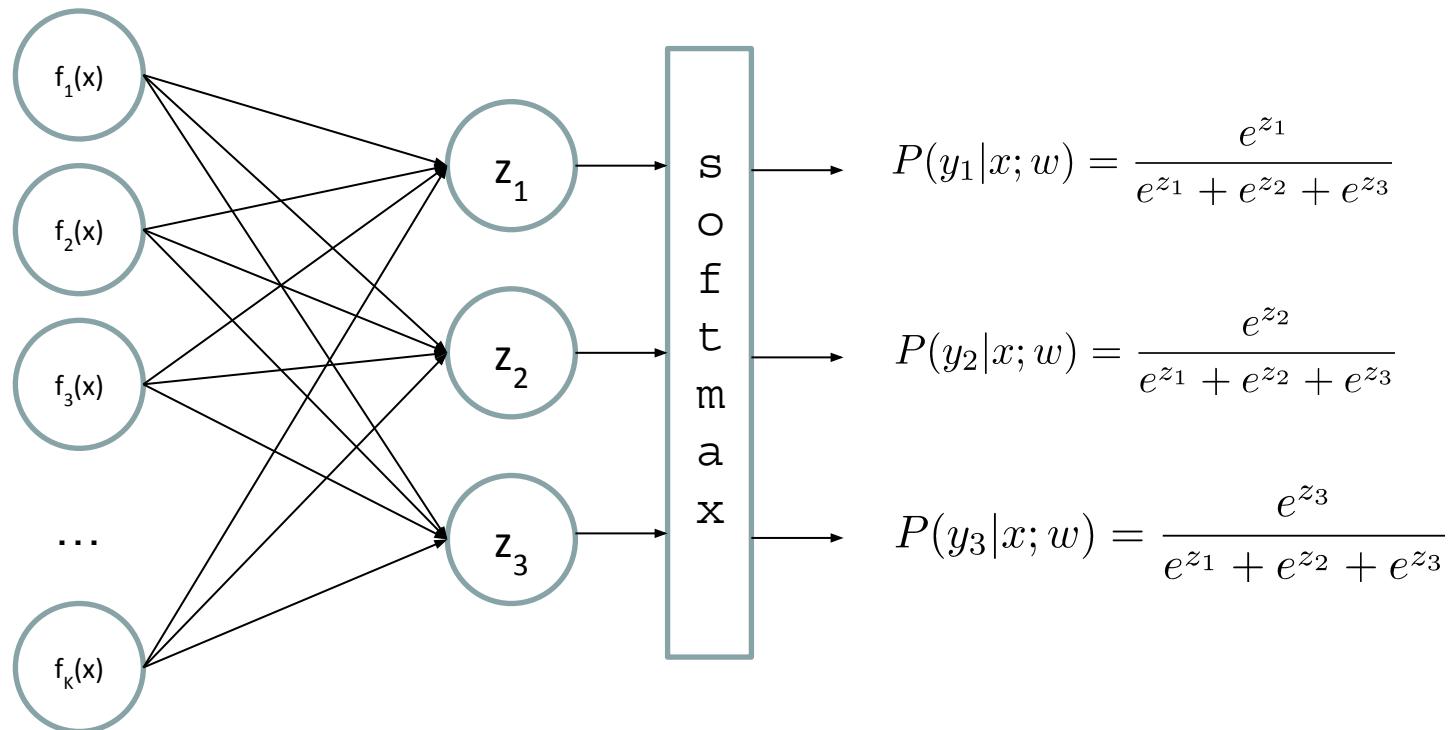
- We'll talk about that once we covered neural networks, which are a generalization of logistic regression

Neural Networks

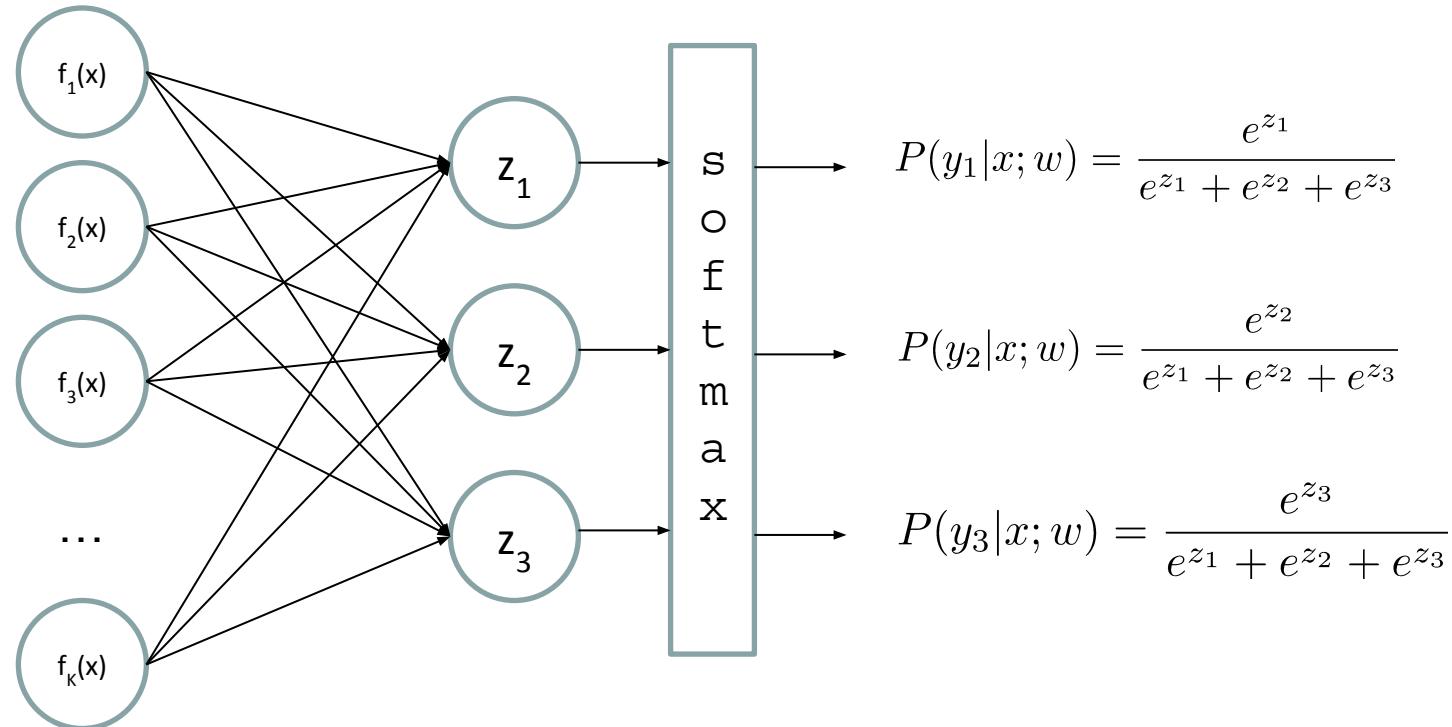


Multi-class Logistic Regression

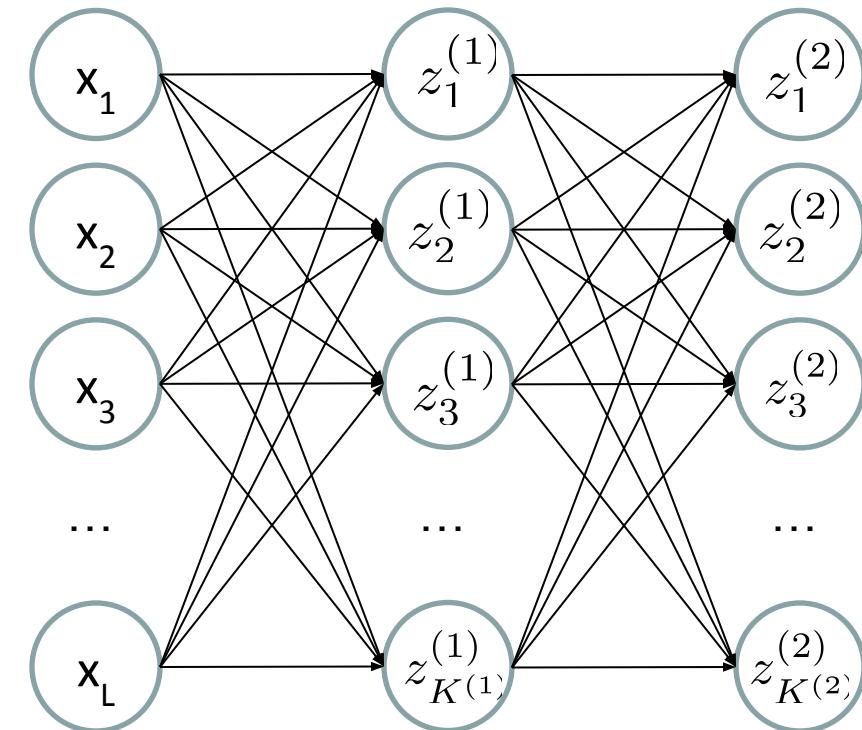
- = special case of neural network



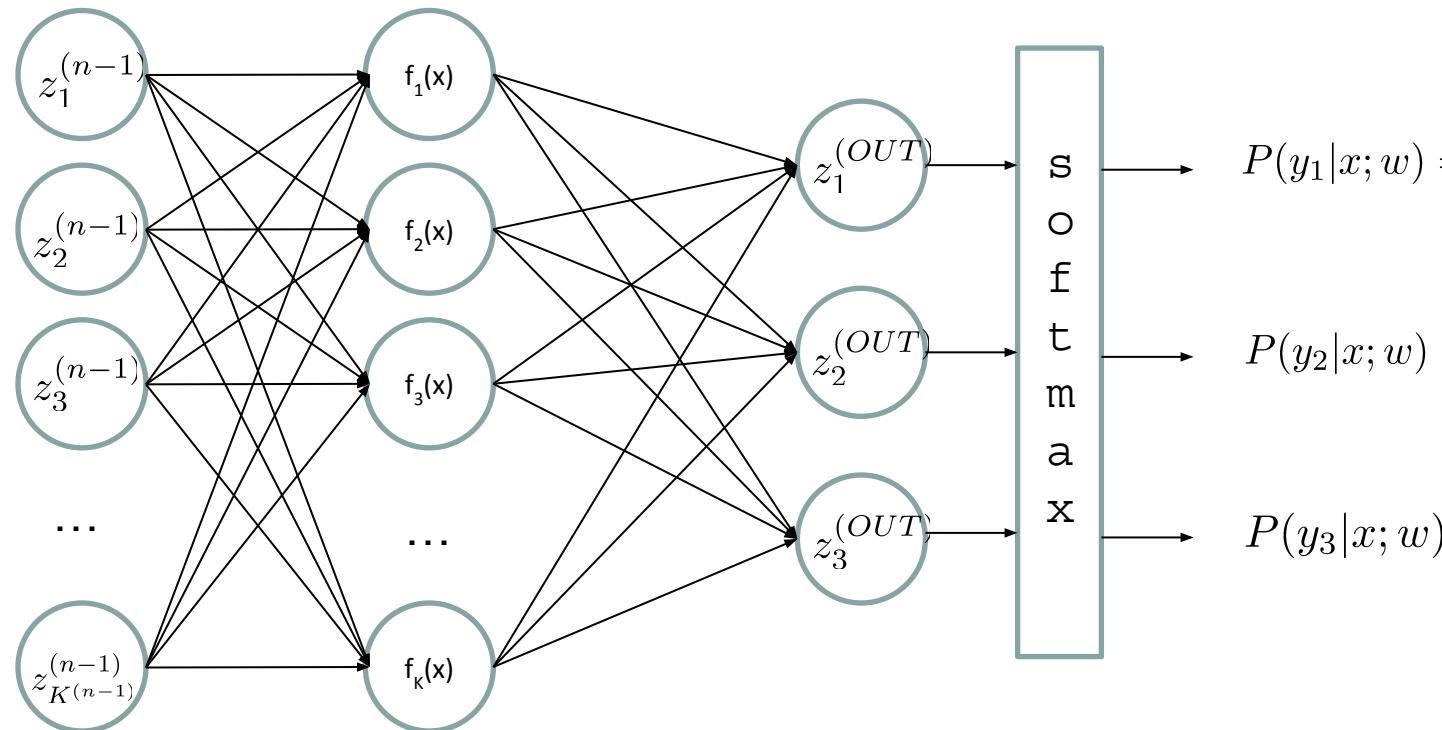
Deep Neural Network = Also learn the features!



Deep Neural Network = Also learn the features!

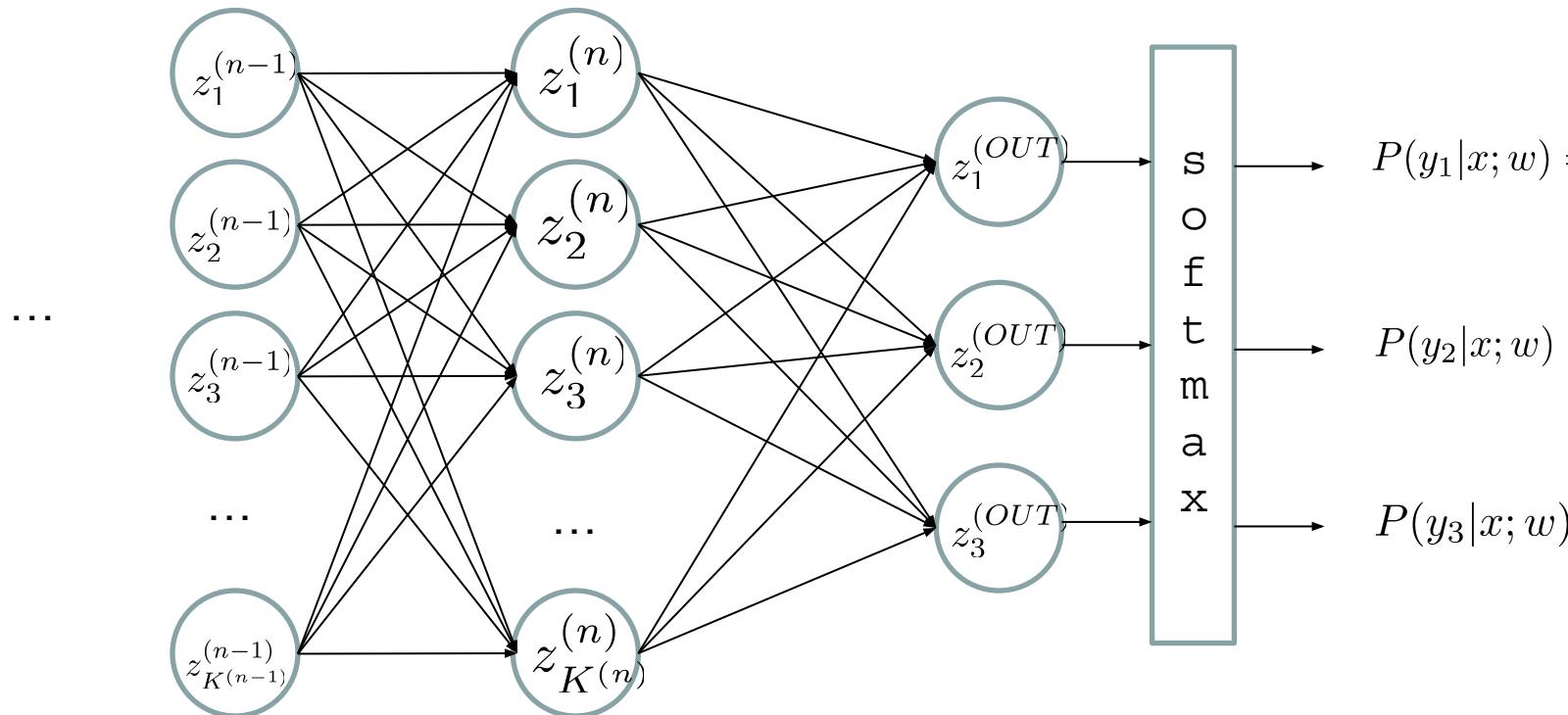
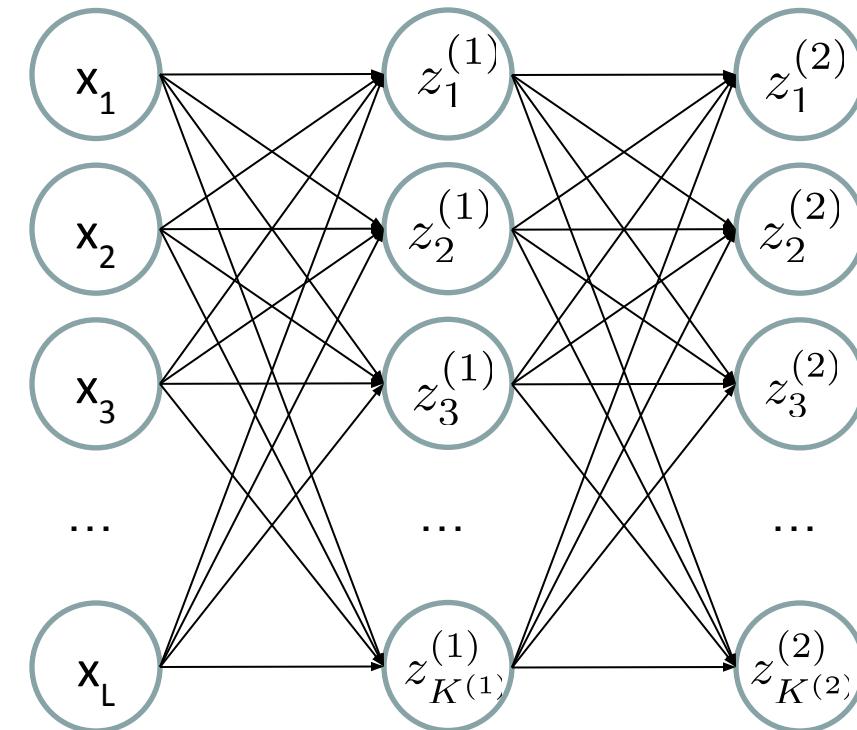


$$z_i^{(k)} = g\left(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)}\right)$$



g = nonlinear activation function

Deep Neural Network = Also learn the features!

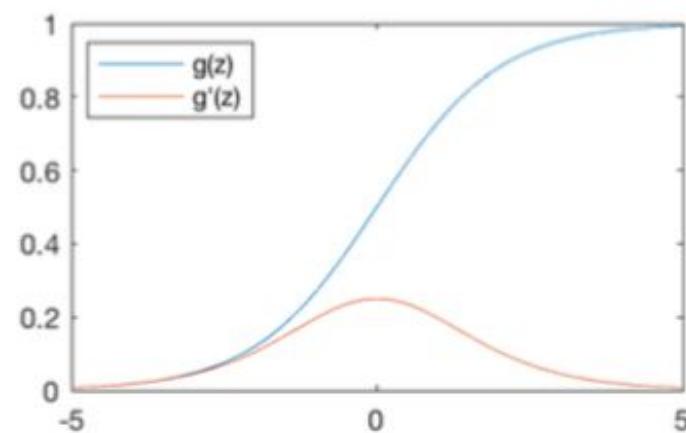


$$z_i^{(k)} = g\left(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)}\right)$$

g = nonlinear activation function

Common Activation Functions

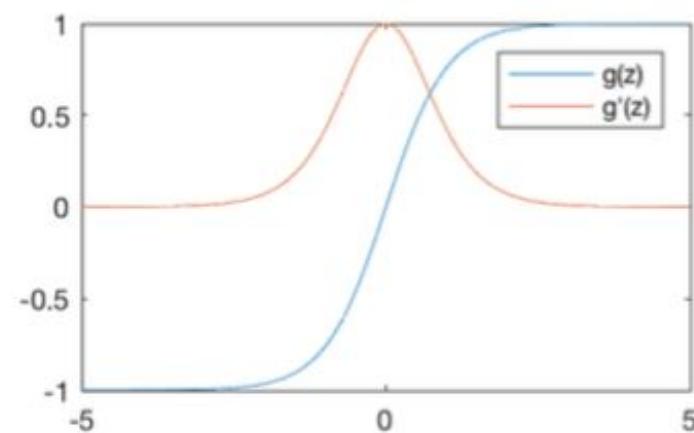
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

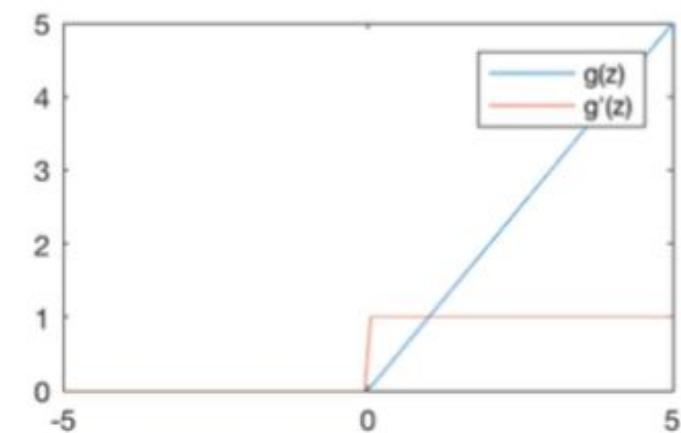
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)



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

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Deep Neural Network: Also Learn the Features!

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

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

just w tends to be a much, much larger vector 😊

- just run gradient ascent
 - + stop when log likelihood of hold-out data starts to decrease

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 Theorem*

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 compositions 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 continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. 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 application areas are concerned with the representation of general functions of an n -dimensional real variable, $x \in \mathbb{R}^n$, by finite linear combinations of the form

$$\sum_{j=1}^N \alpha_j \sigma(y_j^T x + \theta_j), \quad (1)$$

where $y_j \in \mathbb{R}^n$ and $\alpha_j, \theta \in \mathbb{R}$ are fixed. (y^T is the transpose of y so that $y^T x$ is the inner product of y 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:

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

Such functions arise naturally in neural network theory as the activation function of a neural node (or *unit* as is becoming the preferred term) [L1], [RHM]. 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

* Date received: October 21, 1988. Date revised: February 17, 1989. This research was supported in part by NSF Grant DCR-8619103, ONR Contract N000-86-G-0202 and DOE Grant DE-FG02-85ER25001.

† Center for Supercomputing Research and Development and Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois 61801, U.S.A.

Neural Networks, Vol. 4, pp. 251–257, 1991.
Printed in the USA. All rights reserved.

(893-6080/91) \$3.00 + .00
Copyright © 1991 Pergamon Press plc

ORIGINAL CONTRIBUTION

Approximation Capabilities of Multilayer Feedforward Networks

KURT HORNIK

Technische Universität Wien, Vienna, Austria

(Received 30 January 1990; revised and accepted 25 October 1990)

Abstract—We show that standard multilayer feedforward networks with as few as a single hidden layer and arbitrary bounded and nonconstant activation function are universal approximators with respect to $L^p(\mu)$ performance criteria, for arbitrary finite input environment measures μ , provided only that sufficiently many hidden units are available. If the activation function is continuous, bounded and nonconstant, 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 function, Universal approximation capabilities, Input environment measure, $L^p(\mu)$ approximation, Uniform approximation, Sobolev spaces, Smooth approximation.

1. INTRODUCTION

The approximation capabilities of neural network architectures have recently been investigated by many authors, including Carroll and Dickinson (1989), Cybenko (1989), Funahashi (1989), Gallant and White (1988), Hecht-Nielsen (1989), Hornik, Stinchcombe, and White (1989, 1990), Itoh and Miyake (1988), Lapedes and Farber (1988), 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 R^k to R^l , we can ask how well arbitrary mappings from R^k to R^l can be approximated by the network, in particular, if as many hidden units as required for internal representation and computation may be employed.

How to measure the accuracy of approximation depends on how we measure closeness 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 simultaneously well on all input samples taken from some compact input set X in R^k . In this case, closeness is measured by the uniform distance between functions on X , that is,

$$\rho_{\sigma,X}(f, g) = \sup_{x \in X} |f(x) - g(x)|.$$

In other applications, we think of the inputs as random variables and are interested in the *average performance* where the average is taken with respect to the input environment measure μ , where $\mu(R^k) < \infty$. In this case, closeness is measured by the $L^p(\mu)$ distances

$$\rho_{\sigma,\mu}(f, g) = \left[\int_{R^k} |f(x) - g(x)|^p d\mu(x) \right]^{1/p}.$$

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

Of course, there are many more ways of measuring closeness 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 sources of need of smooth functional 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 (1989).

All papers establishing certain approximation ca-

Requests for reprints should be sent to Kurt Hornik, Institut für Statistik und Wahrscheinlichkeitstheorie, Technische Universität Wien, Wiedner Hauptstraße 8-10/107, A-1040 Wien, Austria.

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 IS-91-26

Appeared previously as *Working Paper No. 21/91* 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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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
- Need to know this exists
- How this is done? -- outside of scope of CS4700

How well does this all work?

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

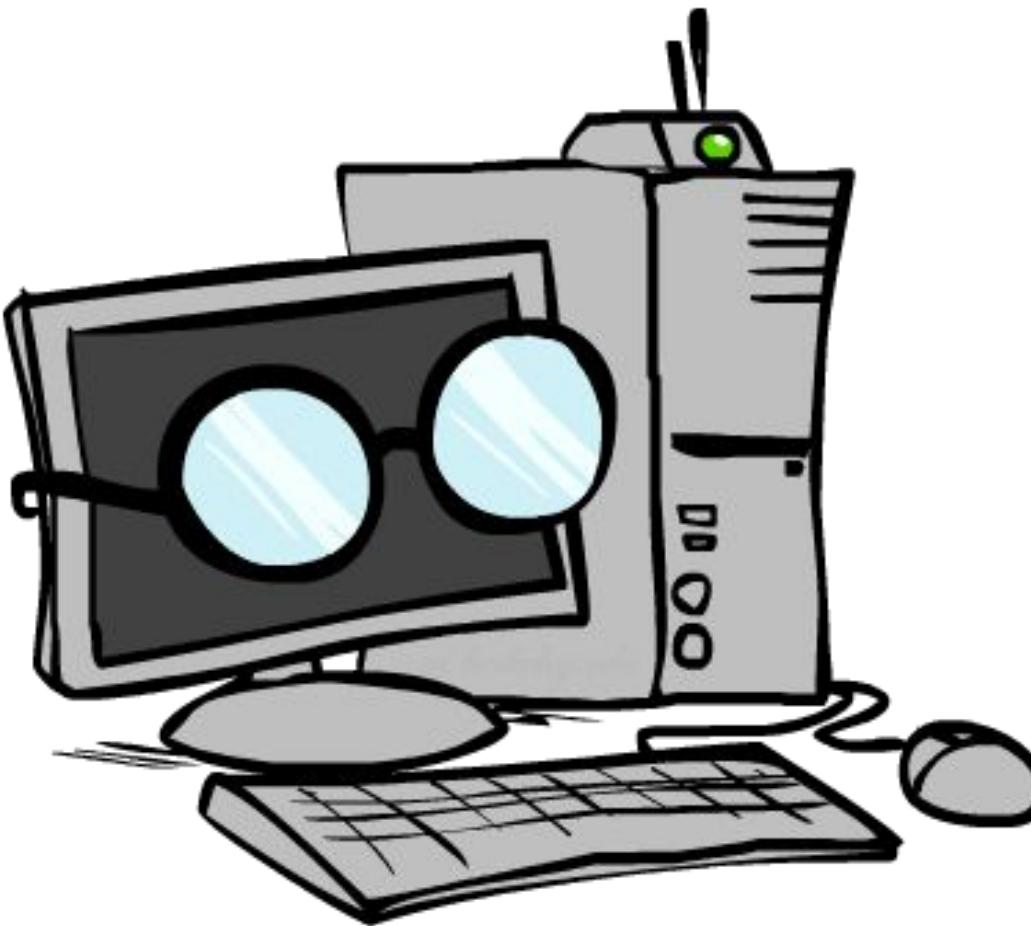
- Universal function approximation theorem

- 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!

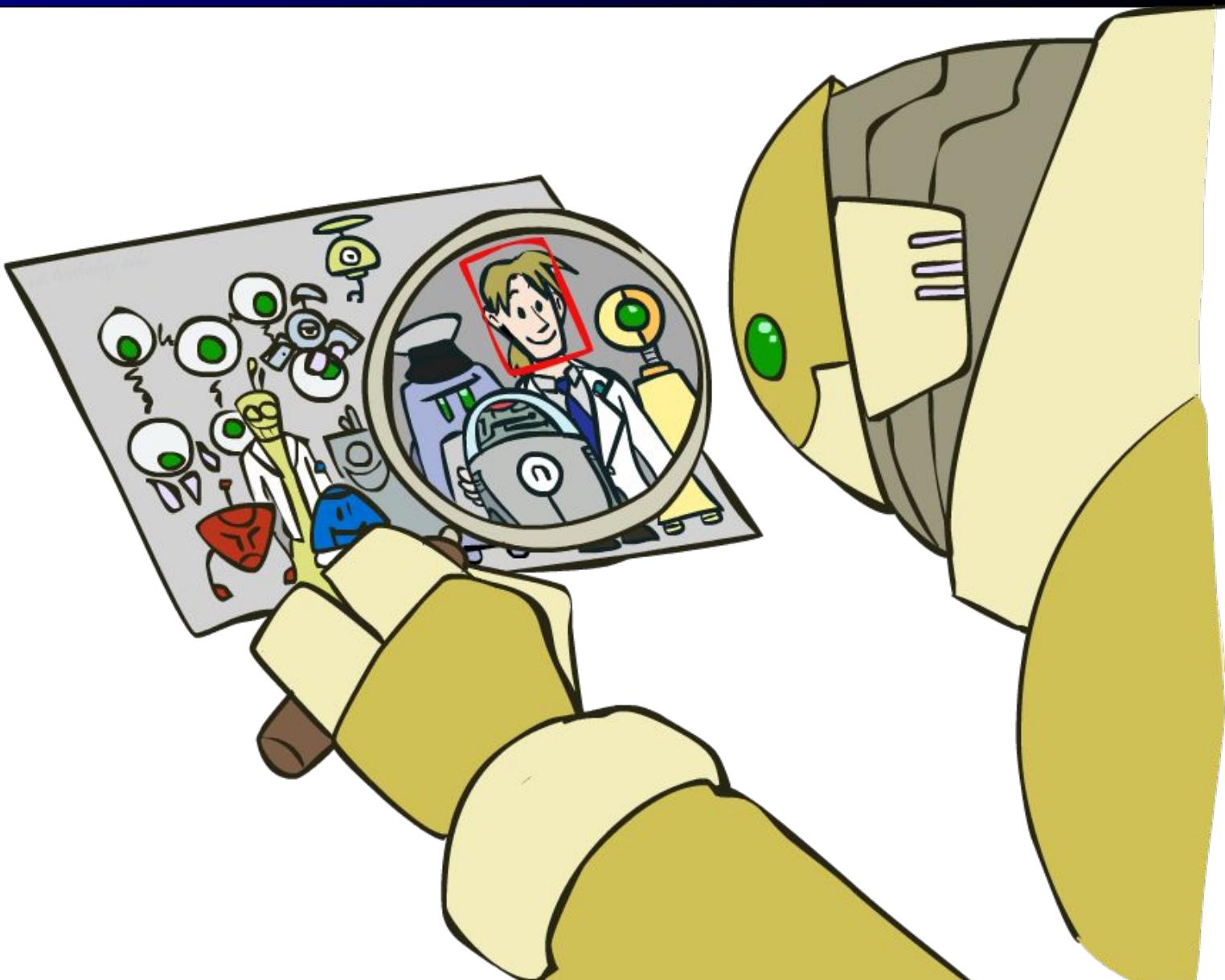
- Automatic differentiation gives the derivatives efficiently (how? = outside of scope of 4700)

How well does it work?

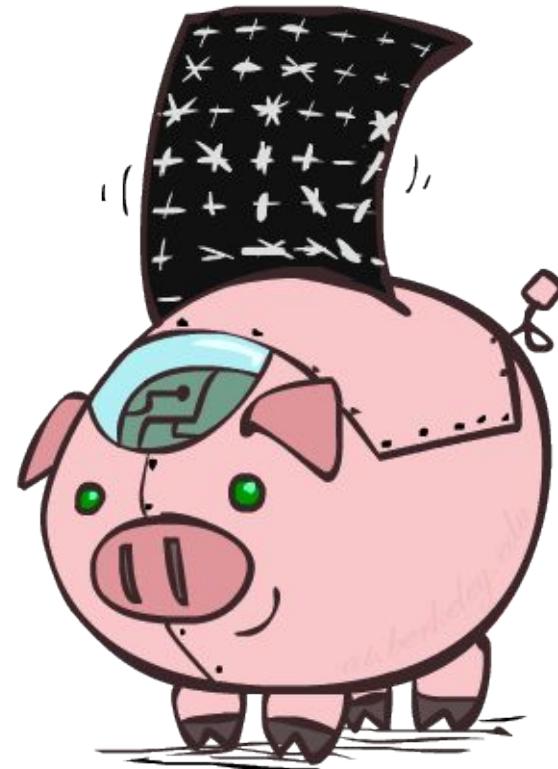
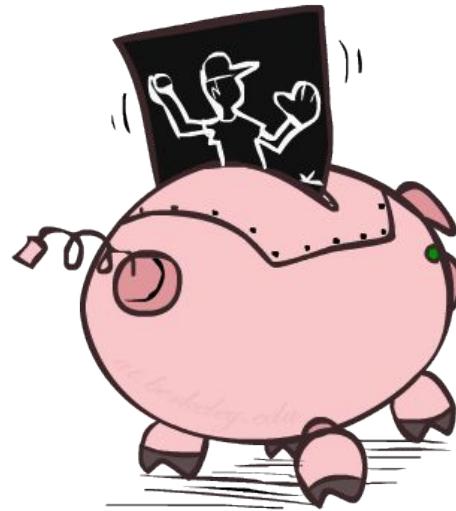
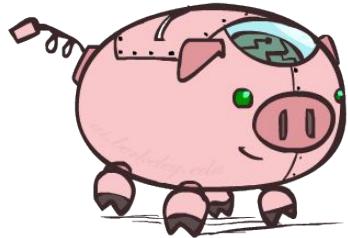
Computer Vision



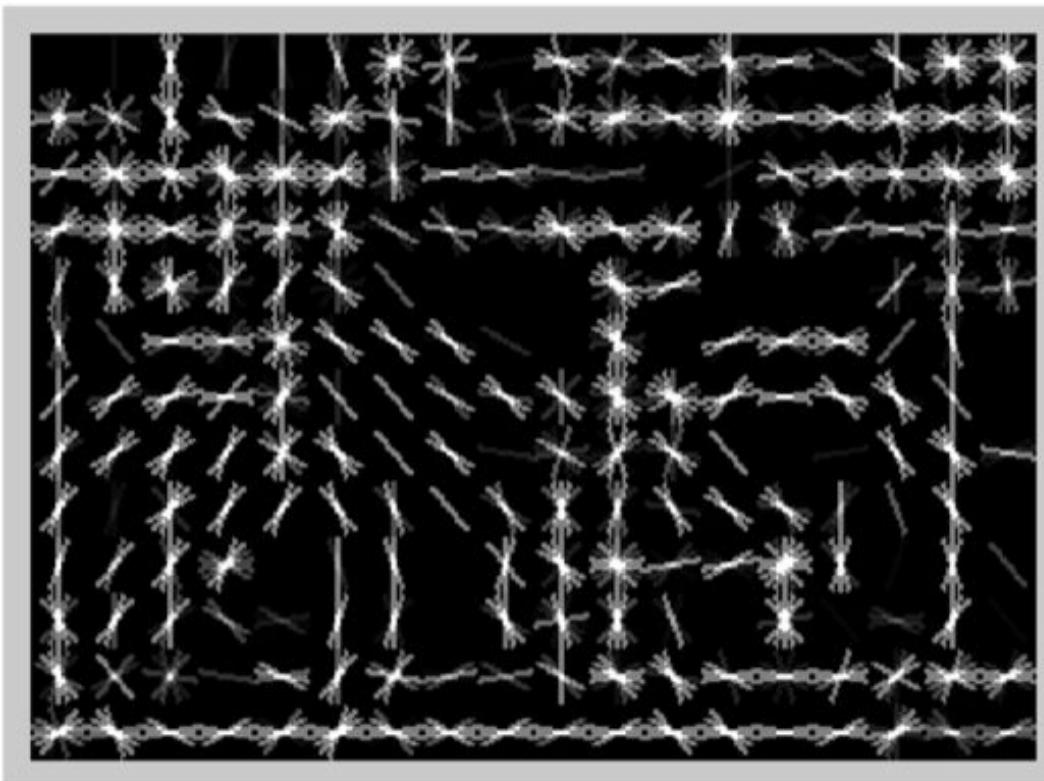
Object Detection



Manual Feature Design



Features and Generalization

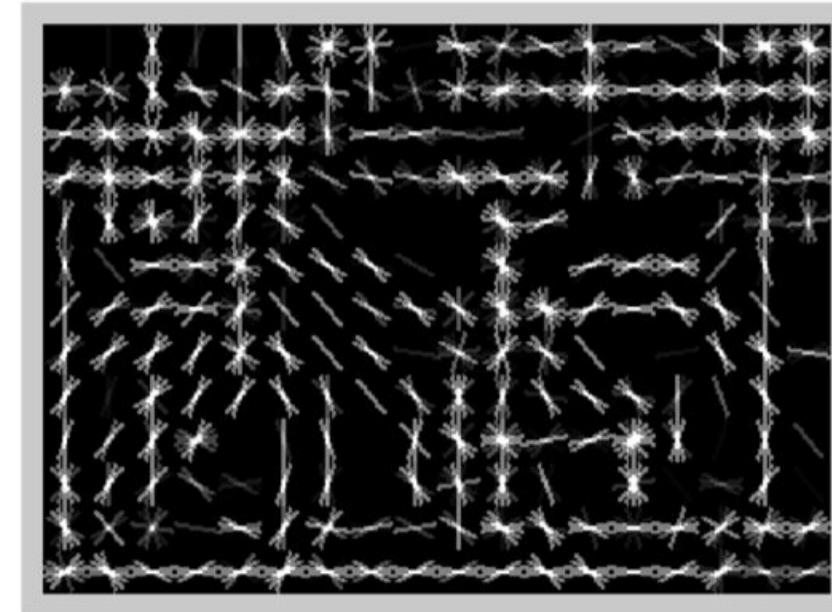


[HoG: Dalal and Triggs,
2005]

Features and Generalization



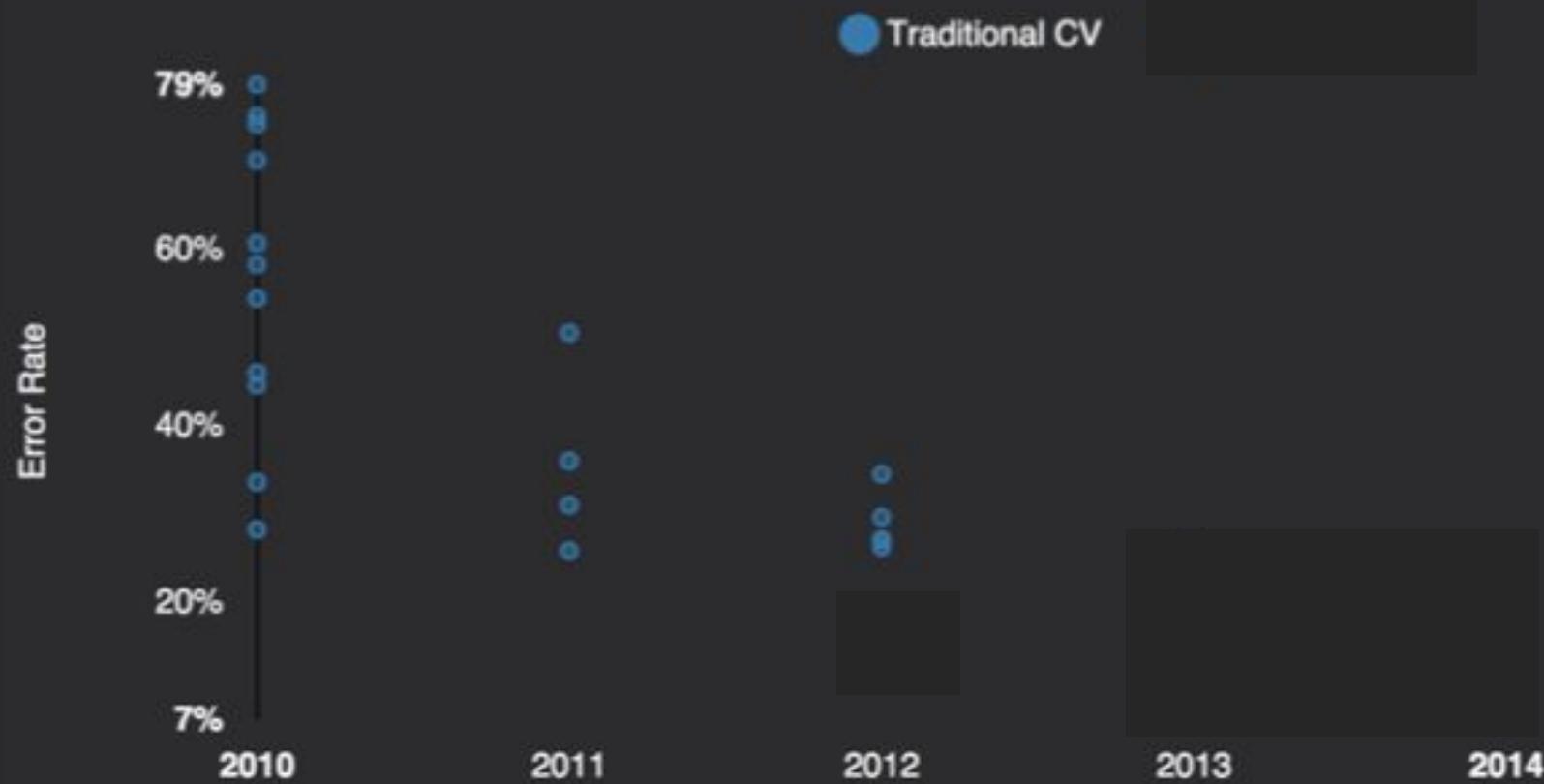
Image



HoG

Performance

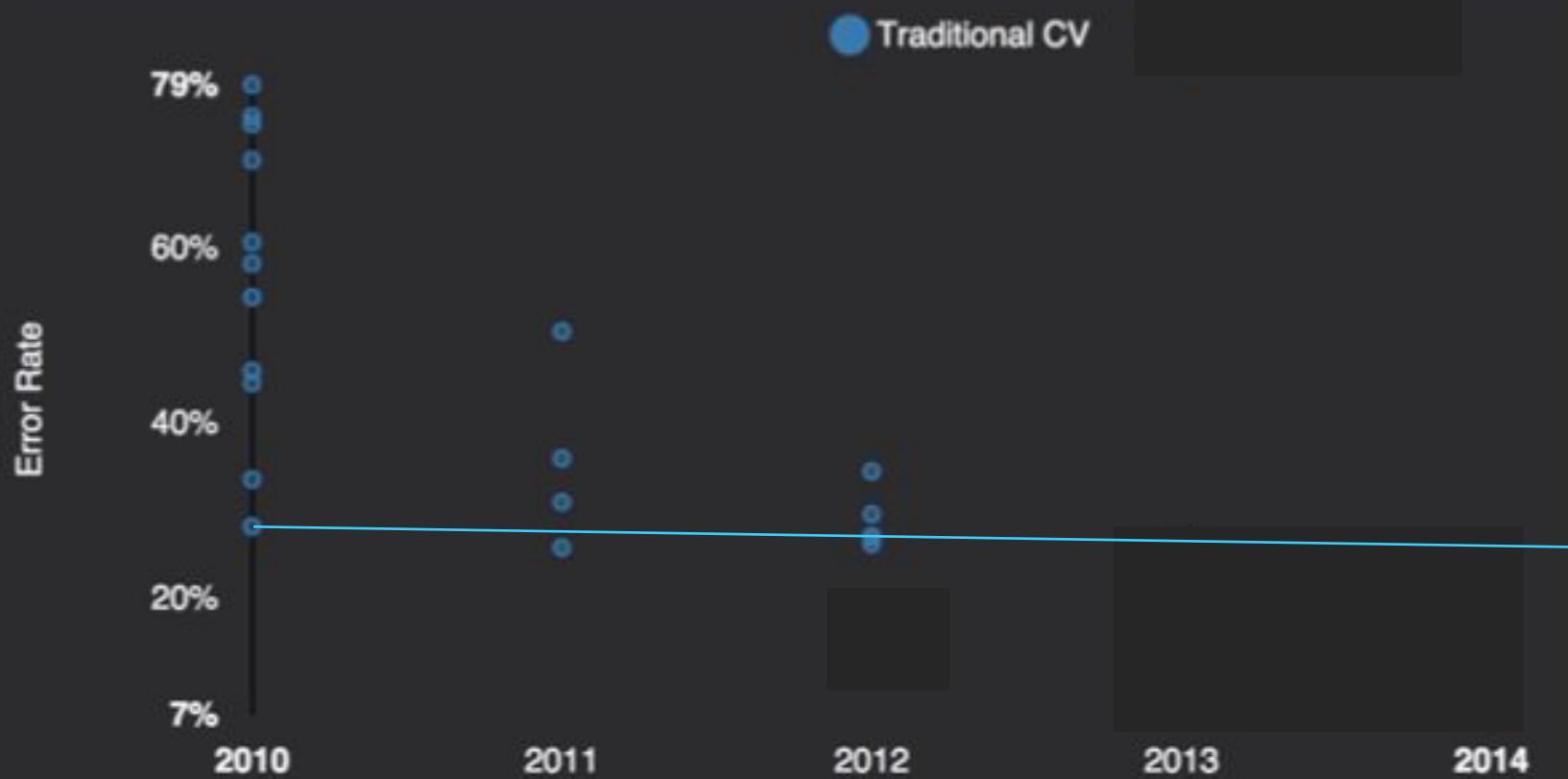
ImageNet Error Rate 2010-2014



graph credit Matt
Zeiler, Clarifai

Performance

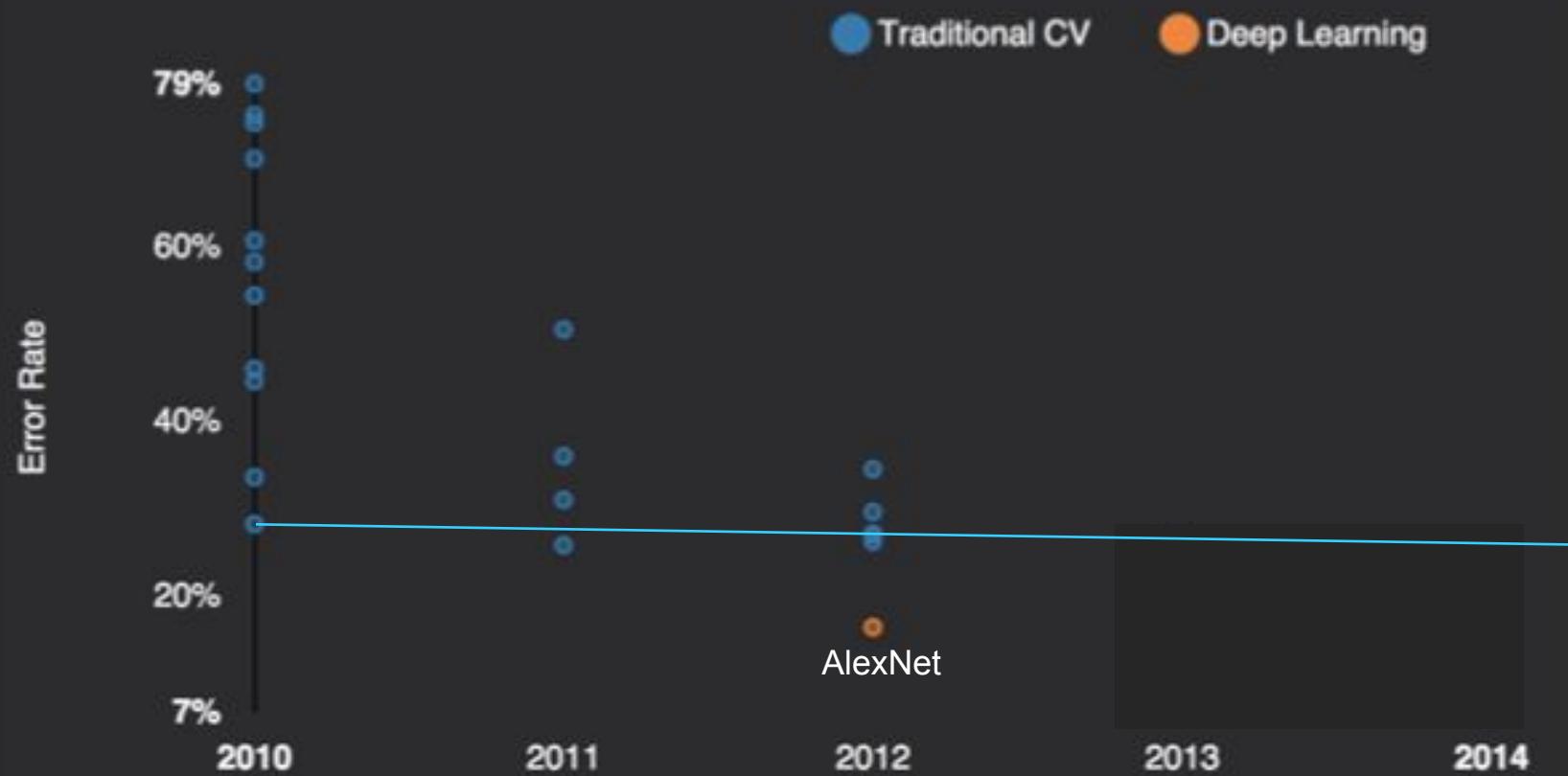
ImageNet Error Rate 2010-2014



graph credit Matt
Zeiler, Clarifai

Performance

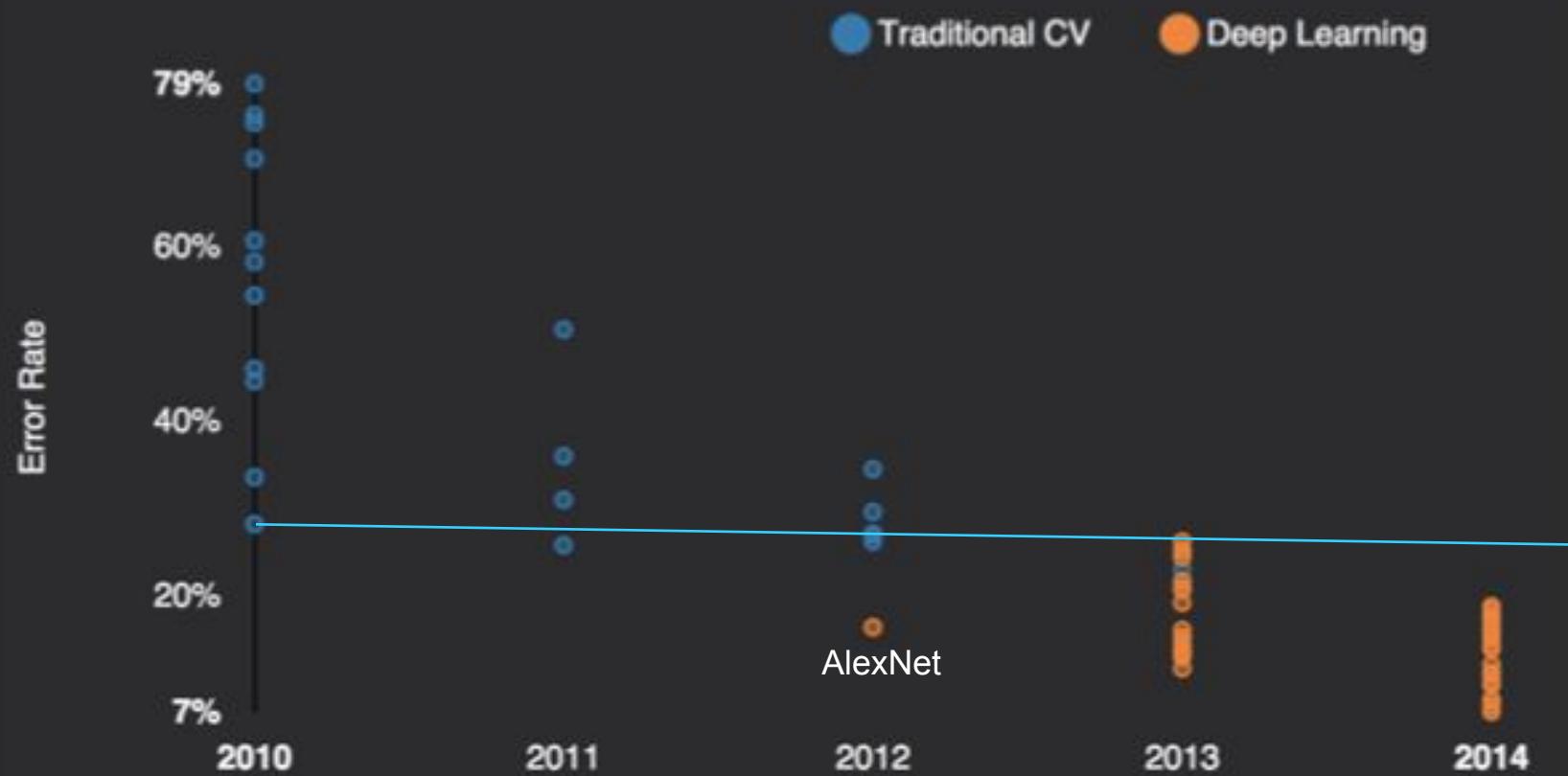
ImageNet Error Rate 2010-2014



graph credit Matt
Zeiler, Clarifai

Performance

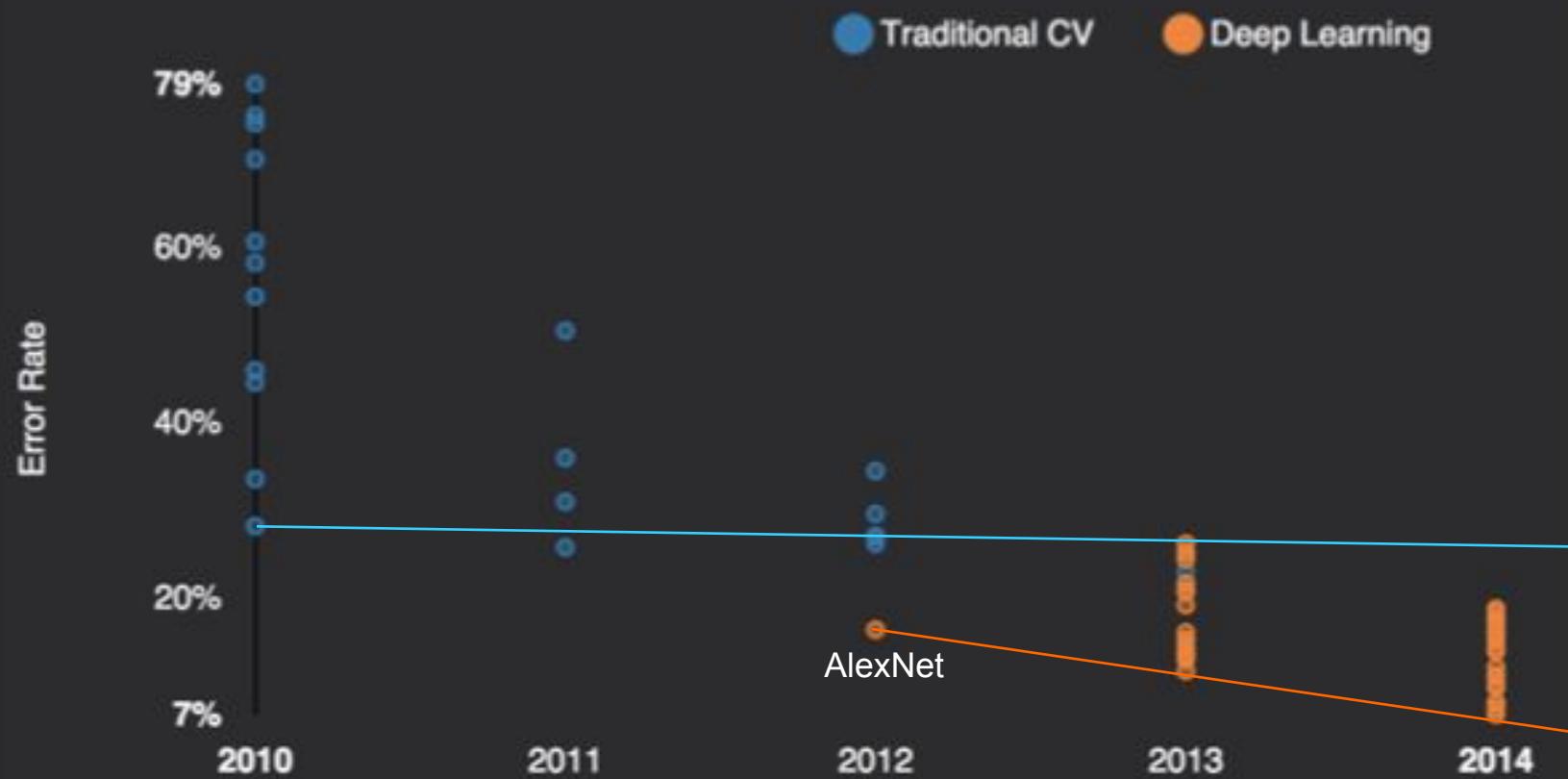
ImageNet Error Rate 2010-2014



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

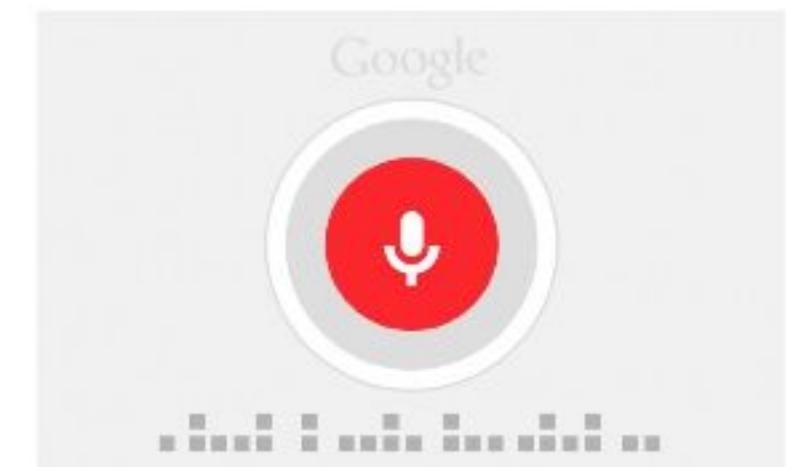
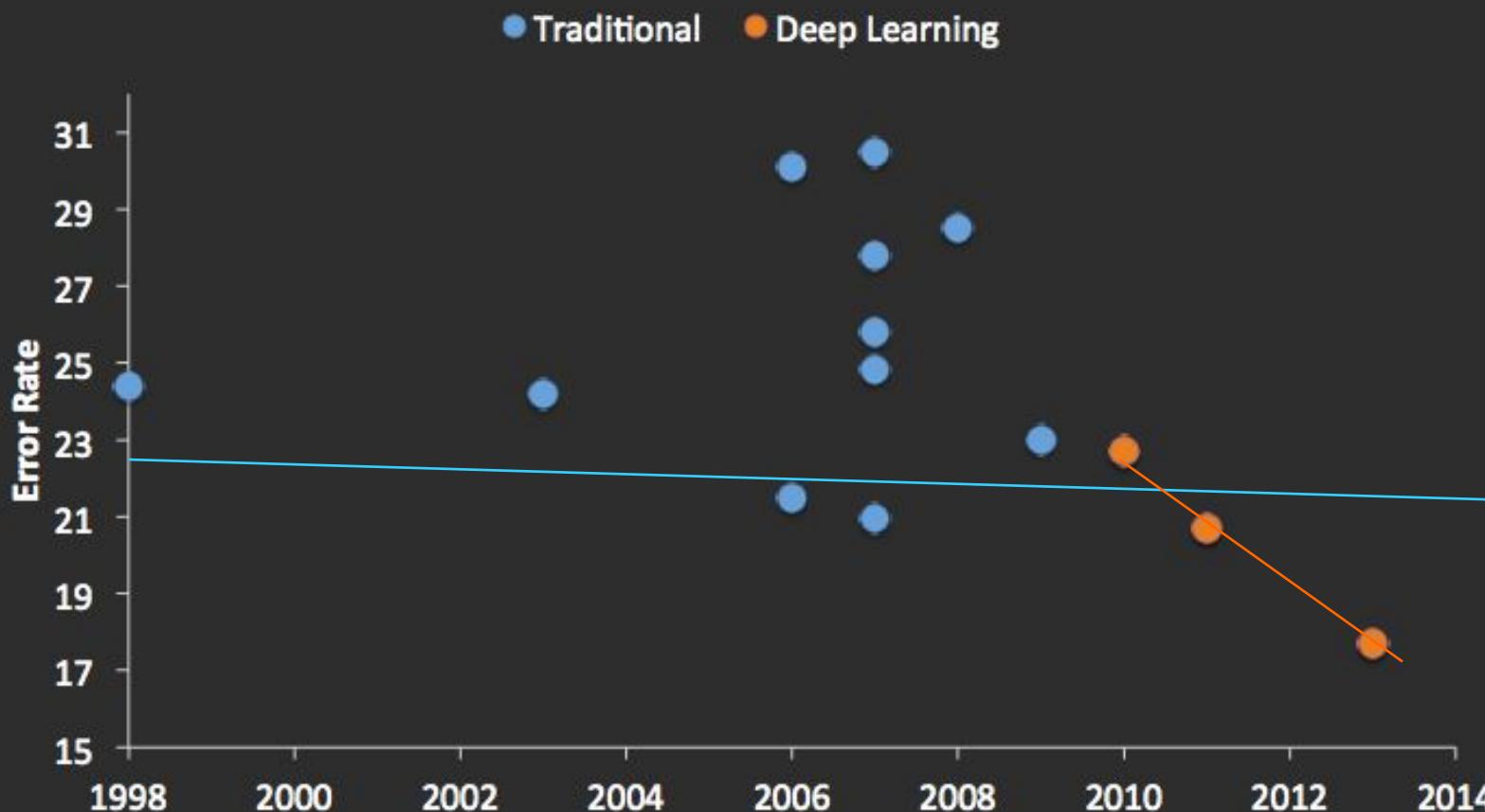


What are people sitting under in the back?

Neural Net: **bench**
Ground Truth: tent

Speech Recognition

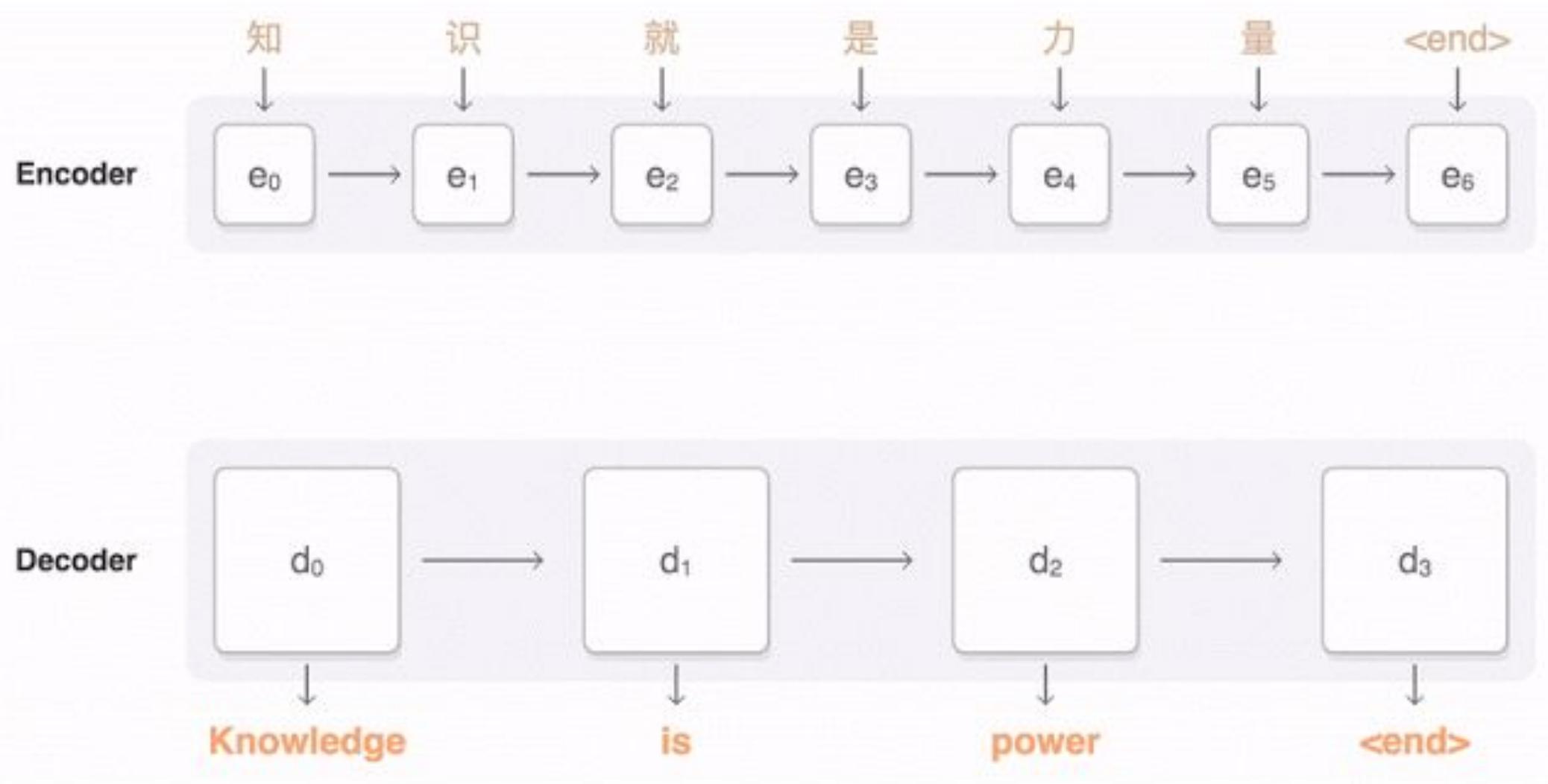
TIMIT Speech Recognition



graph credit Matt Zeiler, Clarifai

Machine Translation

Google Neural Machine Translation (in production)



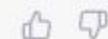
Language Modeling

KE

How are you today, ChatGPT?



As an AI language model, I don't have feelings or emotions, but I'm functioning well and ready to assist you with any questions or tasks you might have. How can I assist you today?



KE

Please explain gradient descent.



Gradient descent is an optimization algorithm used in machine learning and deep learning to minimize a loss function, which measures the difference between the predicted output and the actual output. The idea is to find the optimal values of the parameters of a model that will minimize the loss function, and thus improve the accuracy of the model's predictions.



The algorithm works by iteratively adjusting the values of the model's parameters, based on the gradient of the loss function with respect to each parameter. The gradient is a vector that indicates the direction of steepest increase of the loss

Language Modeling

KE

Please explain gradient descent as a sonnet.



In optimization, we seek a solution,
To minimize a loss function's confusion.
With gradient descent, we take each step,
Towards the valley's bottom, we won't forget.

Initialized with random values at start,
We calculate loss to play our part.
Gradient of the loss, we seek to find,
Adjusting parameters, we refine.

A fraction of the gradient we subtract,
Updating the parameters, in fact.
Iteration by iteration we proceed,
Towards the minimum, we will succeed.

Momentum and adaptive learning we apply,
To optimize gradient descent, we imply.
Thus, we traverse the loss function's path,
With gradient descent, we minimize with math.

Language Modeling

KE

Please write a python function that returns True if the number 74 is even.



Here's the Python function that checks whether the number 74 is even or not:

KE

Question: Regarding the number 74, is it even?

Answer (one word, yes/no):



No.

python

```
def is_even(num):
    if num % 2 == 0:
        return True
    else:
        return False

print(is_even(74)) # Output: True
```

Copy code

The function `is_even` takes a number as an argument and uses the modulo operator (`%`) to check whether the number is even or not. If the remainder of the number divided by 2 is 0, then the number is even and the function returns `True`. Otherwise, the function returns `False`. In the example above, we call the `is_even` function with the argument `74`, which returns `True` because 74 is indeed an even number.

Next: More Neural Net Applications!
