

Lecture Notes for **Machine Learning in Python**

Professor Eric Larson
Town Hall + MLP History

Class Logistics and Agenda

- Logistics:
 - Next time: Flipped Module on back propagation
- Multi Week Agenda:
 - Today: Neural Networks History, up to 1980
 - Today: Multi-layer Architectures
 - Town Hall, Lab 3 (if time)
 - Flipped: Programming Multi-layer training

A History of Neural Networks

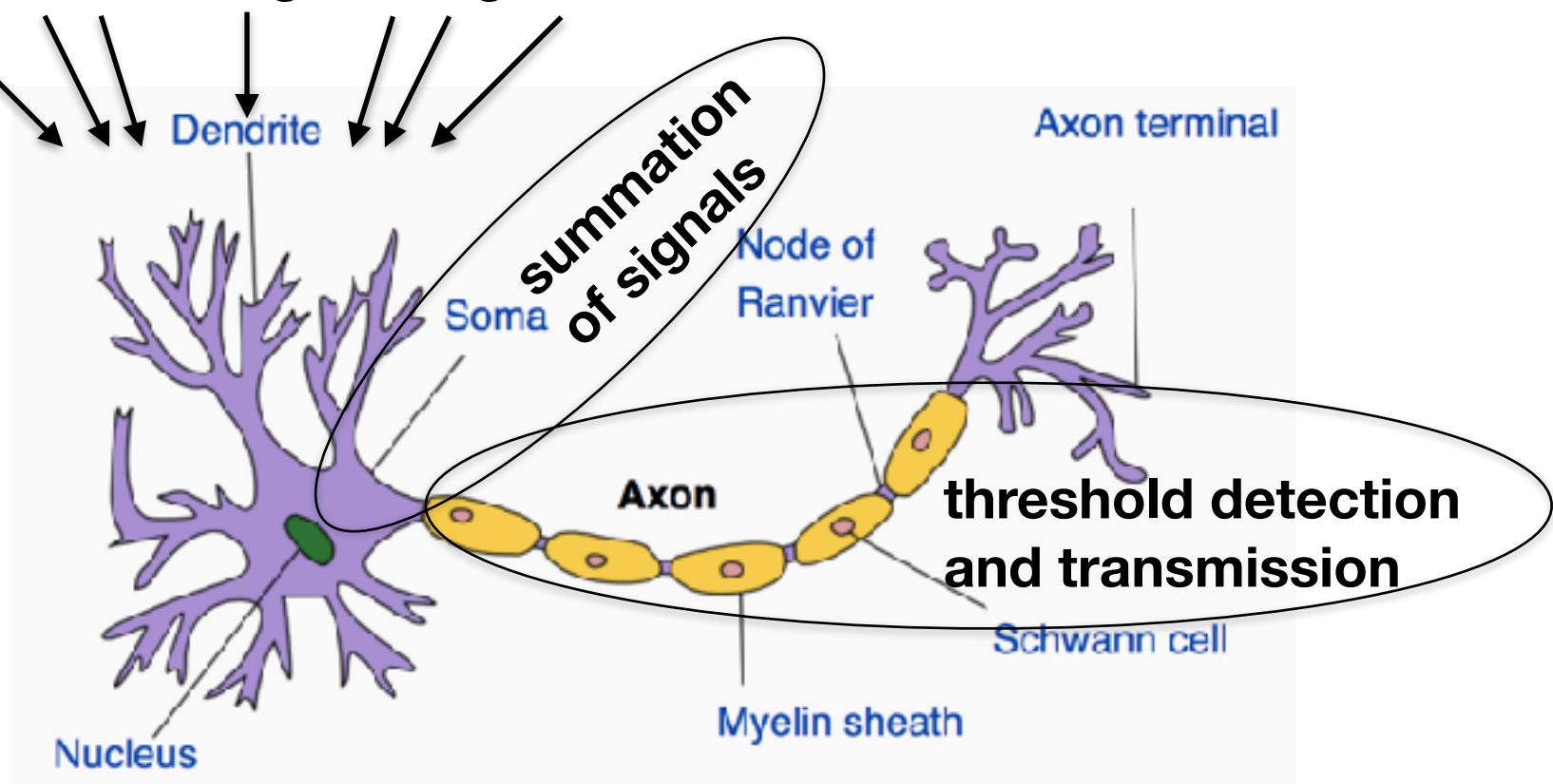


Machine Learning 101

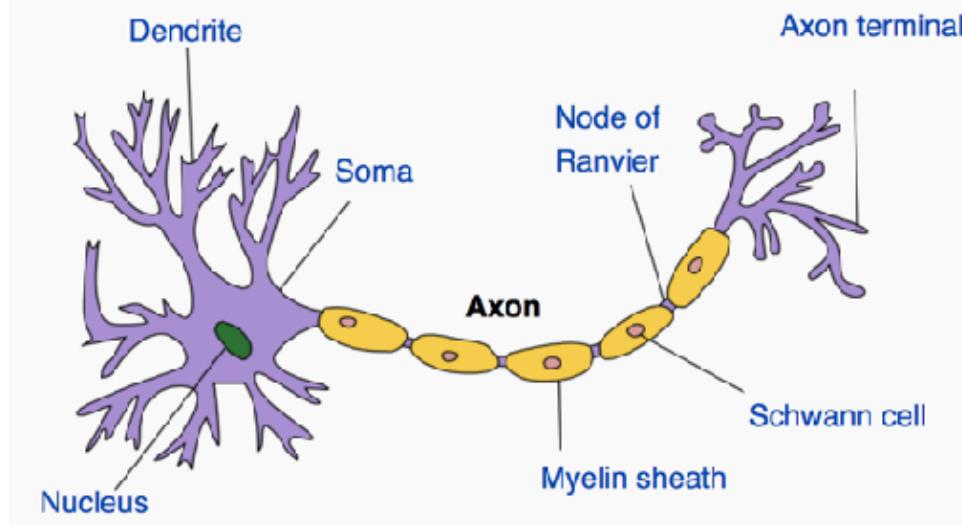
Neurons

- From biology to modeling:

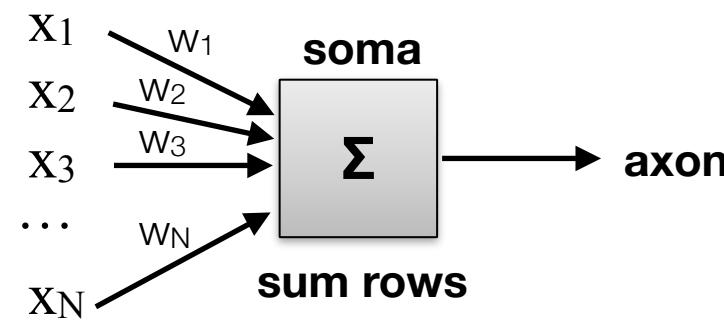
input from neighboring neurons



McCulloch and Pitts, 1943



dendrite

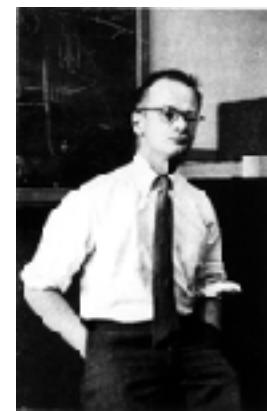


input

logic gates of the mind



Warren McCulloch



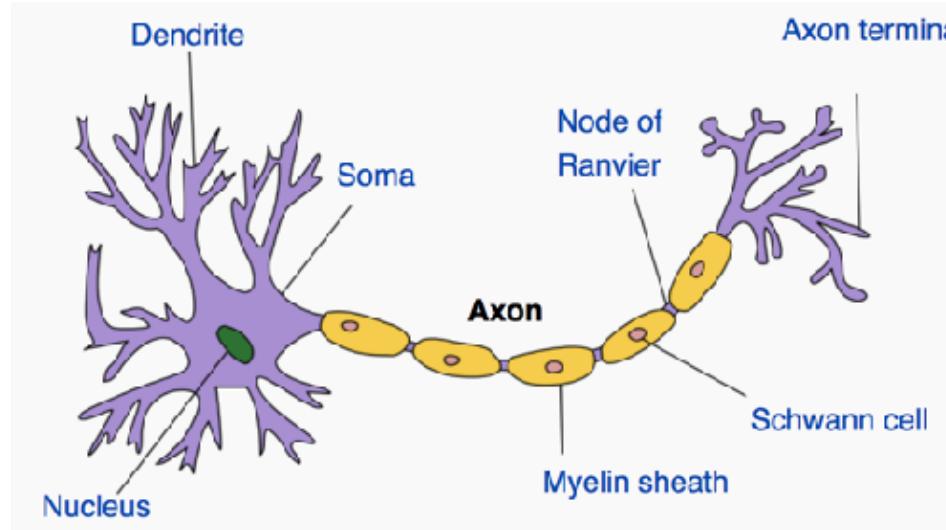
Walter Pitts

Neurons

- McCulloch and Pitts, 1943
- Donald Hebb, 1949
 - Hebb's Law: close neurons fire together
 - neurons “learn” to couple
 - easier synaptic transmission
 - basis of neural pathways



Donald O. Hebb

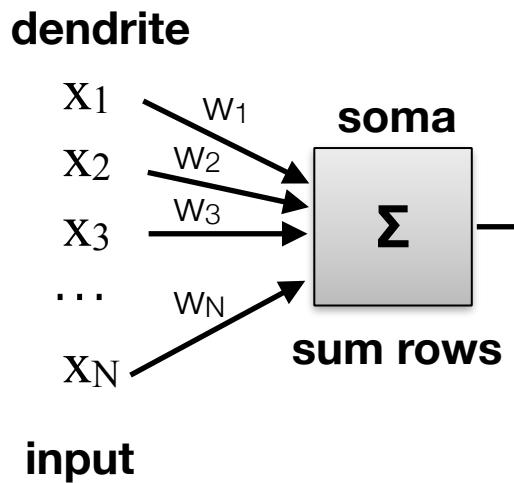
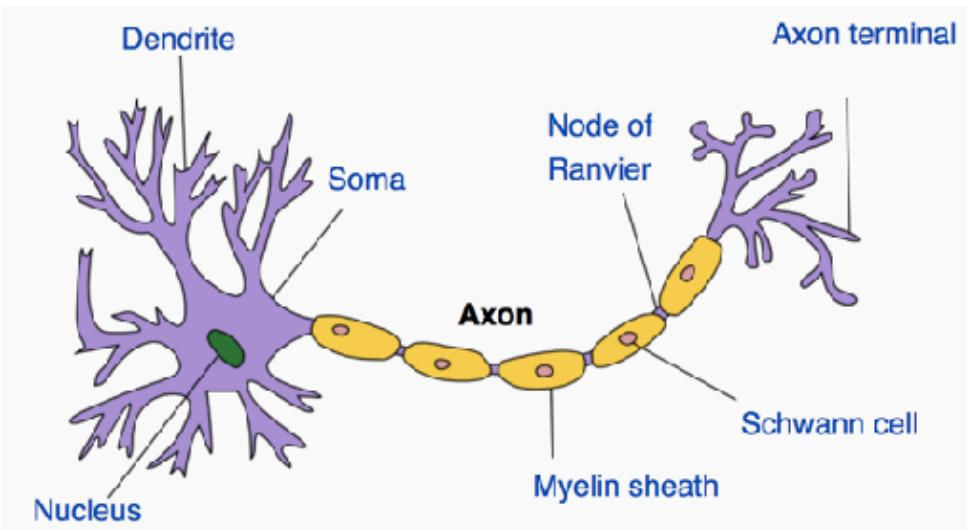


Warren McCulloch



Walter Pitts

Rosenblatt's perceptron, 1957



hard limit



linear

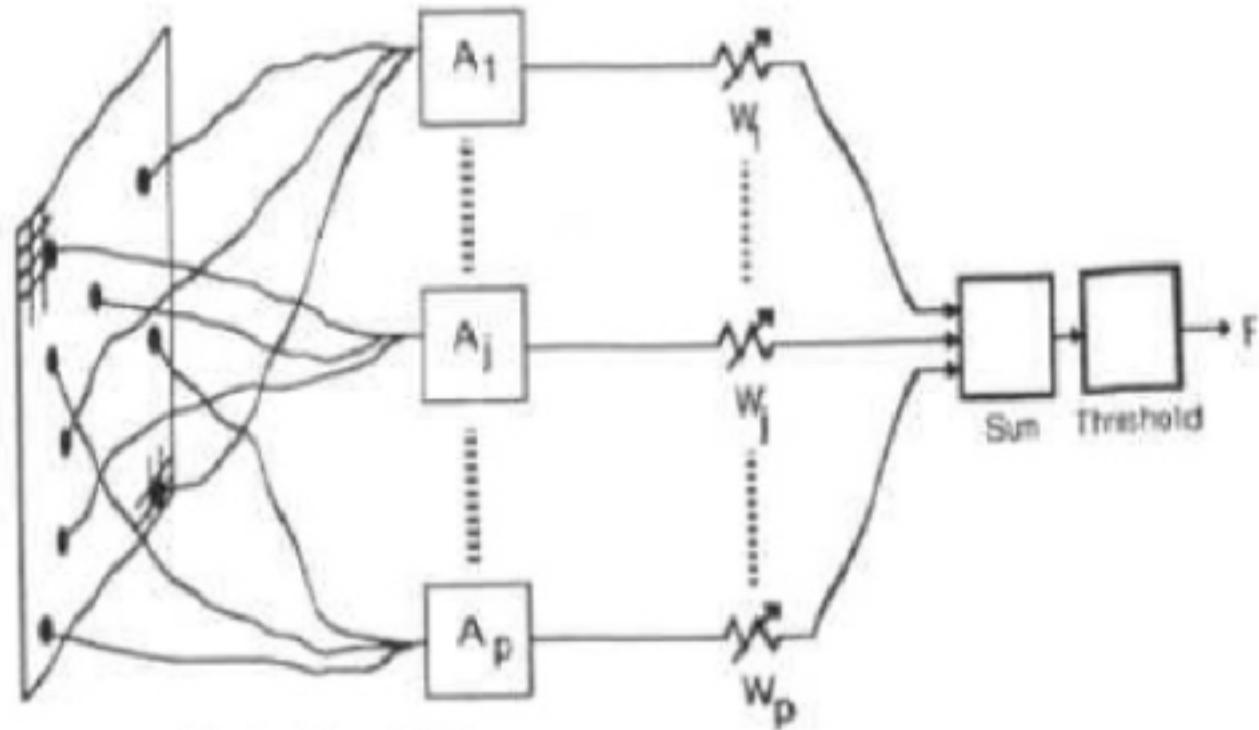


activation
function



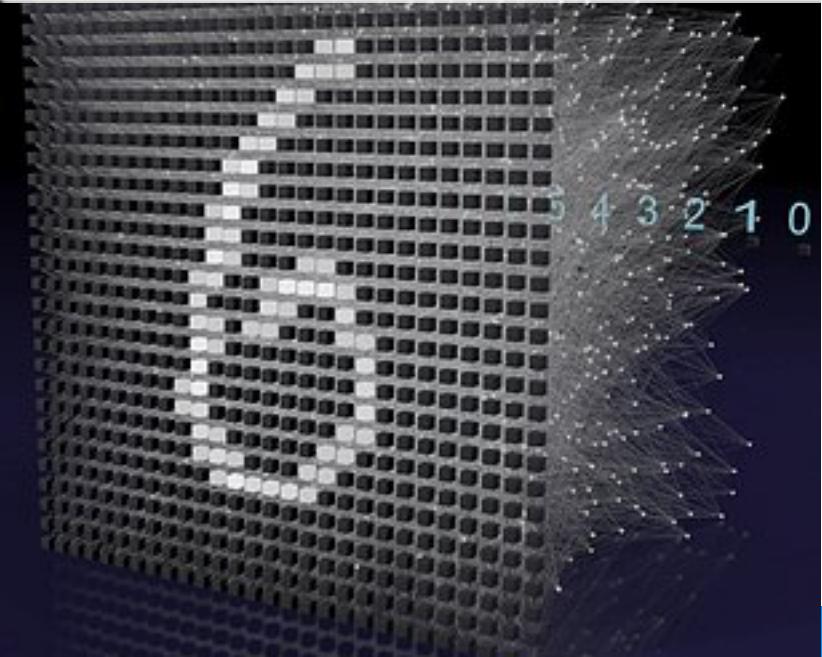
$$a = \frac{1}{1 + \exp(-z)}$$

The Mark 1



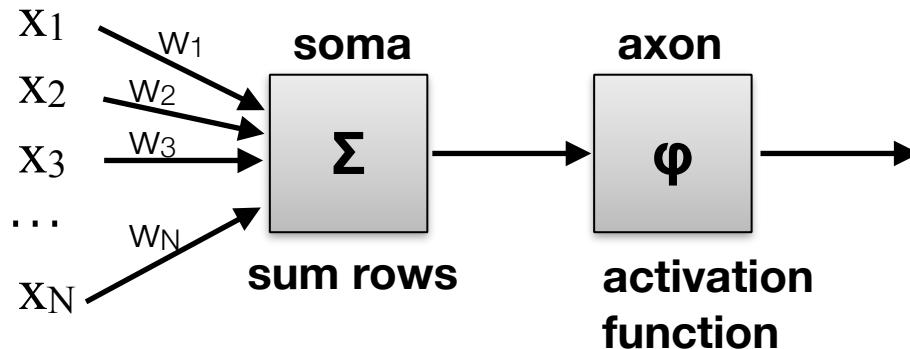
PERCEPTRON

Perceptron Learning Rule:
~Stochastic Gradient Descent

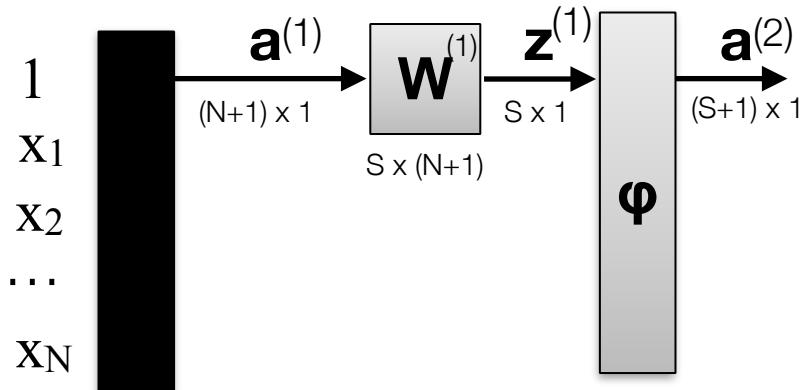


Some Notation

dendrite



input



- need bias term
- matrix representation
- multiple layers

$$\mathbf{a}^{(1)} = \mathbf{x} + \text{concat bias term}$$

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)} \mathbf{a}^{(1)}$$

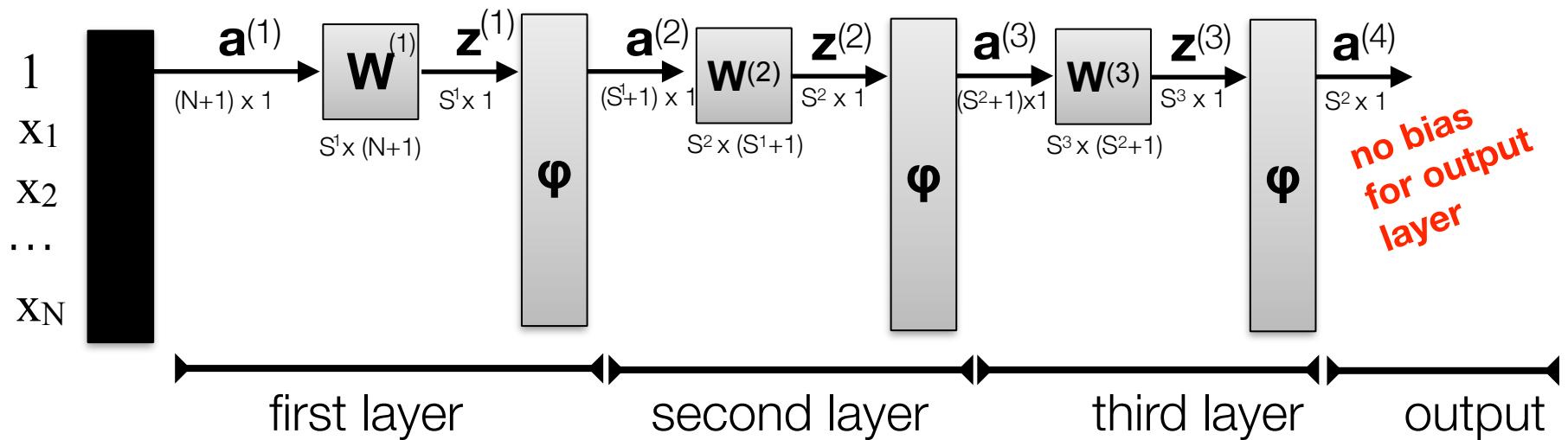
$$\mathbf{W} = \begin{bmatrix} W_{1,2} & W_{1,3} & W_{1,4} & \dots & W_{1,N+1} \\ \vdots & & & & \\ W_{S,2} & W_{S,3} & W_{S,4} & \dots & W_{S,N+1} \end{bmatrix}$$

$$\mathbf{a}^{(2)} = \Phi(\mathbf{z}) + \text{concat bias term}$$

$\mathbf{x}^{(i)}$ One row from Table data
becomes input column to model

notation adapted from *Neural Network Design*, Hagan, Demuth, Beale, and De Jesus

Multiple Layers Notation



$$\mathbf{a}^{(L+1)} = \Phi(\mathbf{z}^{(L)}) + \text{concat bias term}$$

$\mathbf{a}^{(4)}$ rows=unique classes

$$\mathbf{z}^{(L)} = \mathbf{W}^{(L)} \mathbf{a}^{(L)}$$

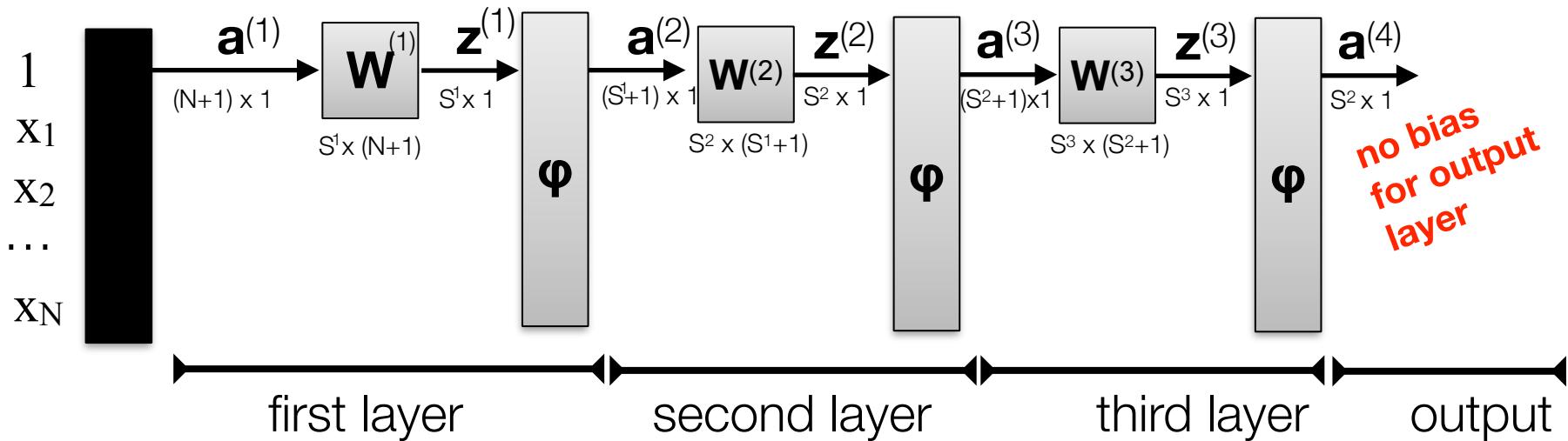
$$\mathbf{z}^{(L)} = \mathbf{W}^{(L)} \Phi(\mathbf{z}^{(L-1)})$$

$$\mathbf{W}^{(L)} = \begin{bmatrix} w^{(L)}_{1,1} & w^{(L)}_{1,2} & w^{(L)}_{1,3} & \dots & w^{(L)}_{1,S^{L-1}+1} \\ \vdots & & & & \\ w^{(L)}_{S^L,1} & w^{(L)}_{S^L,2} & \dots & & w^{(L)}_{S^L,S^{L-1}+1} \end{bmatrix}$$

$$\mathbf{z}^{(L)} = \mathbf{W}^{(L)} \Phi(\mathbf{W}^{(L-1)} \mathbf{a}^{(L-1)})$$

notation adapted from *Neural Network Design*, Hagan, Demuth, Beale, and De Jesus 10

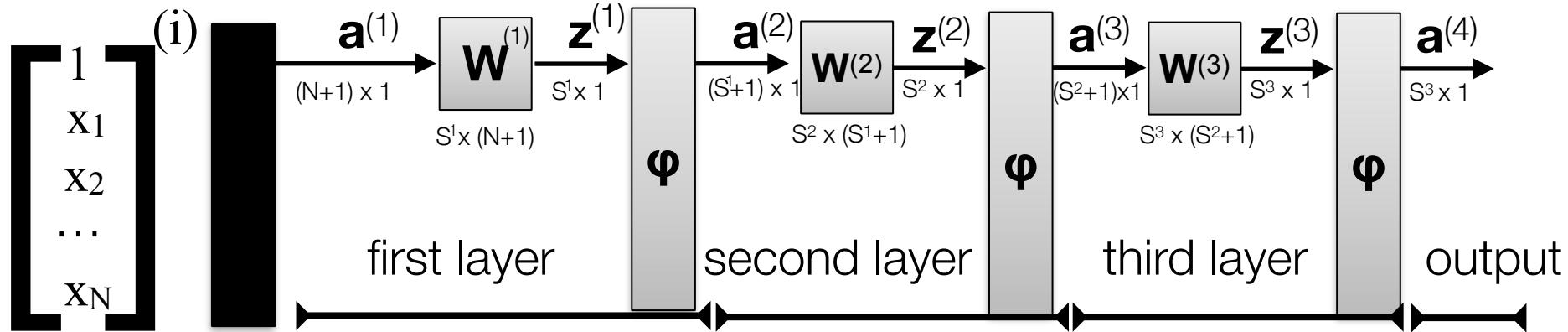
Multiple layers notation



- **Self test:** How many parameters need to be trained in the above network?
 - A. $[(N+1) \times S^1] + [(S^1+1) \times S^2] + [(S^2+1) \times S^3]$
 - B. $|W^{(1)}| + |W^{(2)}| + |W^{(3)}|$
 - C. can't determine from diagram
 - D. it depends on the sizes of intermediate variables, $z^{(i)}$

notation adapted from *Neural Network Design*, Hagan, Demuth, Beale, and De Jesus 11

Compact feedforward notation



$$z^{(L)} = \mathbf{W}^{(L)} \mathbf{a}^{(L)}$$

$$[z^{(L)}]^{(i)} = \mathbf{W}^{(L)} [\mathbf{a}^{(L)}]^{(i)}$$

$$\begin{bmatrix} z^{(L)}_1 \\ \vdots \\ z^{(L)}_{S^L} \end{bmatrix}^{(i)} = \begin{bmatrix} w^{(L)}_{1,1} & w^{(L)}_{1,2} & w^{(L)}_{1,3} & \dots & w^{(L)}_{1,S^{L-1}+1} \\ \vdots & \ddots & & & \vdots \\ w^{(L)}_{S^L,1} & w^{(L)}_{S^L,2} & \dots & & w^{(L)}_{S^L,S^{L-1}+1} \end{bmatrix} \begin{bmatrix} a^{(L)}_1 \\ \vdots \\ a^{(L)}_{S^{L-1}+1} \end{bmatrix}^{(i)}$$

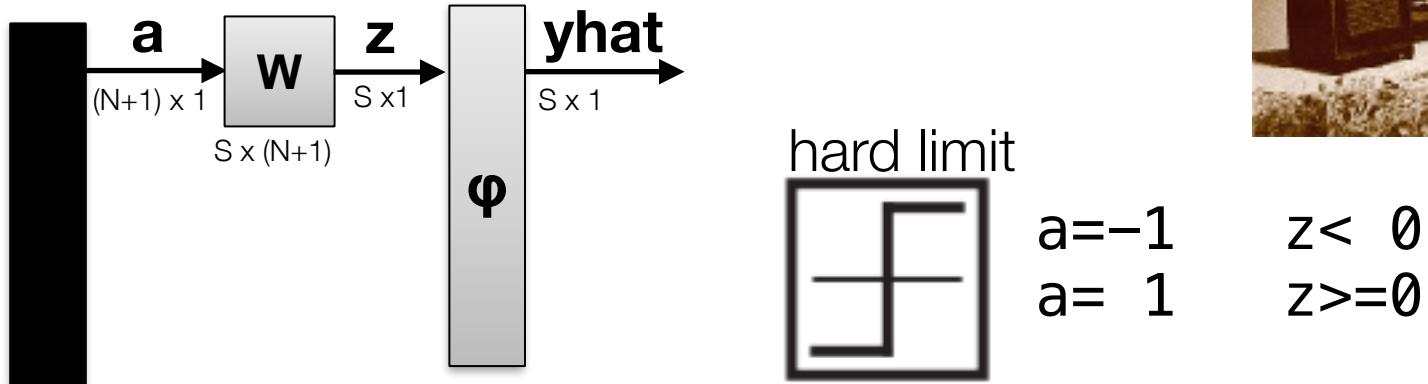
$$\begin{bmatrix} \begin{bmatrix} z^{(L)}_1 \\ \vdots \\ z^{(L)}_{S^L} \end{bmatrix}^{(1)} & \dots & \begin{bmatrix} z^{(L)}_1 \\ \vdots \\ z^{(L)}_{S^L} \end{bmatrix}^{(M)} \end{bmatrix} = [\mathbf{W}^{(L)}] \begin{bmatrix} \begin{bmatrix} a^{(L)}_1 \\ \vdots \\ a^{(L)}_{S^{L-1}+1} \end{bmatrix}^{(1)} & \dots & \begin{bmatrix} a^{(L)}_1 \\ \vdots \\ a^{(L)}_{S^{L-1}+1} \end{bmatrix}^{(M)} \end{bmatrix}$$

$$\mathbf{Z}^{(L)} = \mathbf{W}^{(L)} \mathbf{A}^{(L)}$$

$$\mathbf{Z}^{(L)} = \mathbf{W}^{(L)} \Phi(\mathbf{W}^{(L-1)} \mathbf{A}^{(L-1)})$$

Simple Architectures

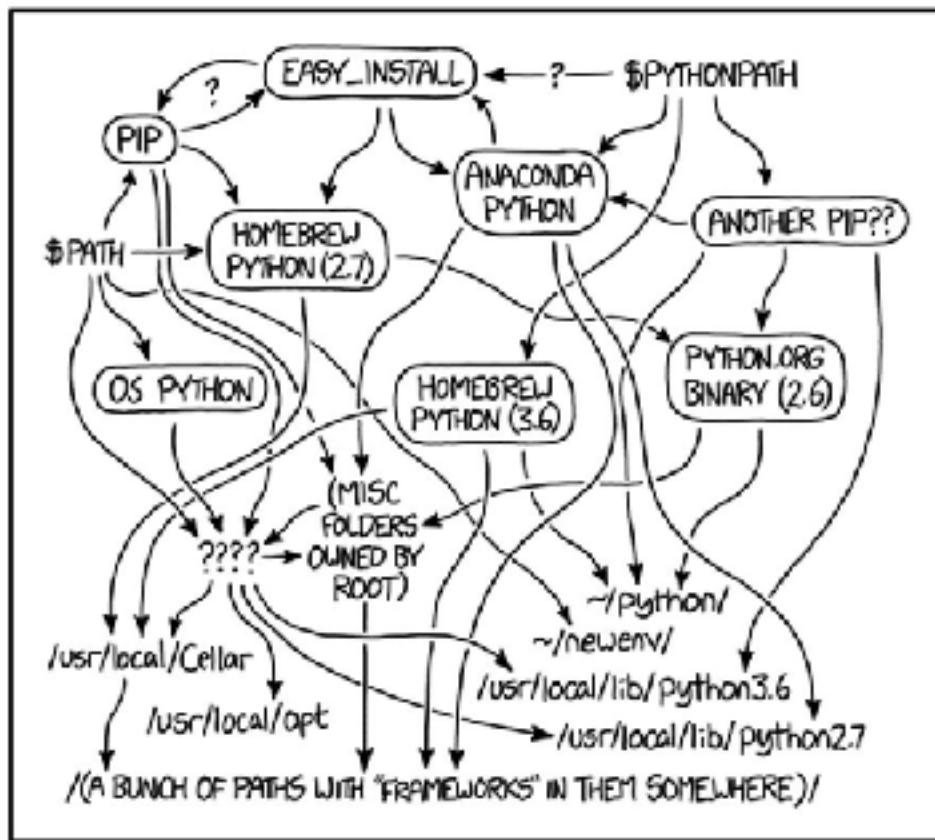
- Rosenblatt's perceptron, 1957



Self Test - If this is a binary classification problem, how large is S ?

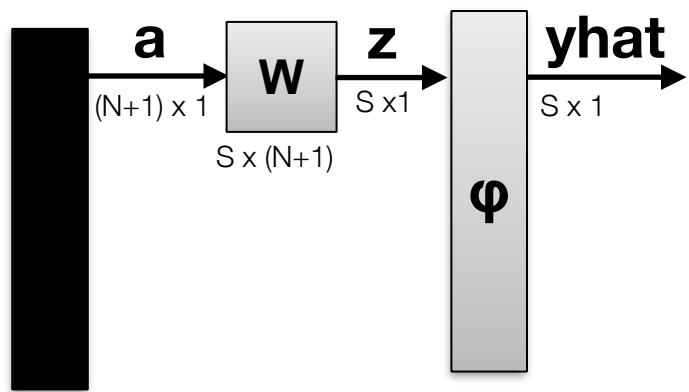
- A. Can't determine
- B. 2
- C. 1
- D. N

Training Neural Network Architectures



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

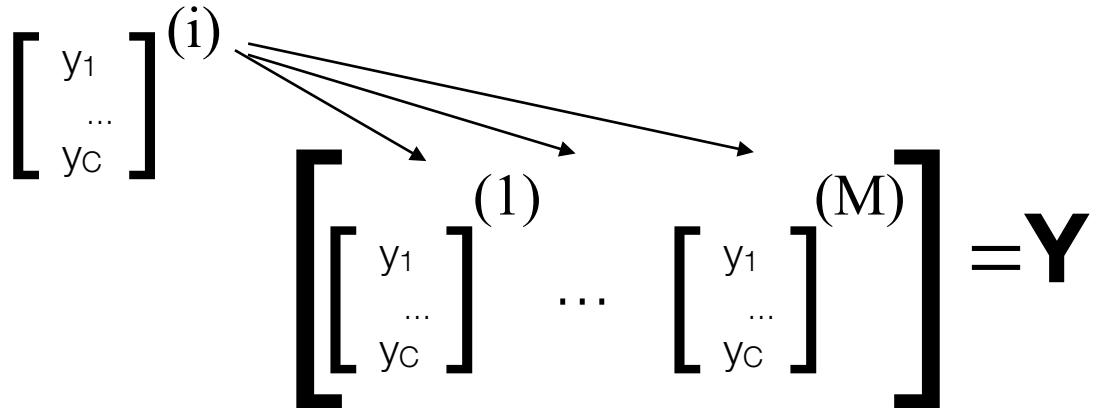
Rosenblatt's Perceptron, 1957



$$\sum_i^M (\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)})^2$$

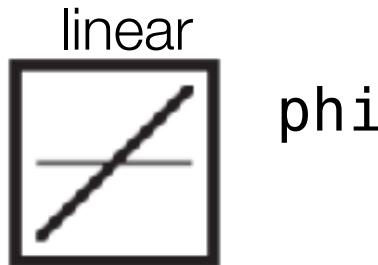
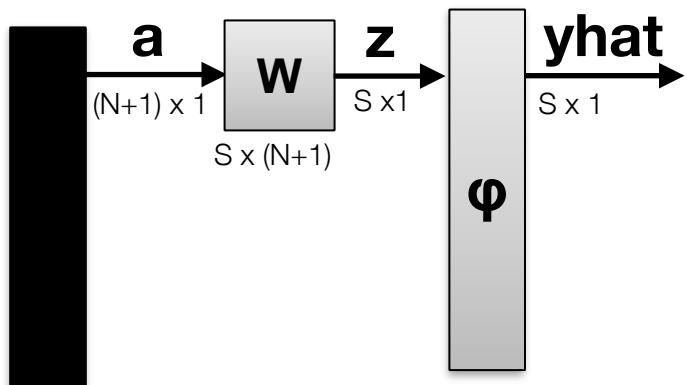
Need objective Function, minimize MSE $J(\mathbf{W}) = \|\mathbf{Y} - \hat{\mathbf{Y}}\|^2$

where ground truth $\mathbf{y}^{(i)}$ is one-hot encoded!



Simple Architectures

- Adaline network, Widrow and Hoff, 1960



Marcian "Ted" Hoff

Bernard Widrow

Objective Function, minimize MSE $J(\mathbf{W}) = ||\mathbf{Y} - \hat{\mathbf{Y}}||^2$

New objective function becomes: $J(\mathbf{W}) = ||\mathbf{Y} - \mathbf{W} \cdot \mathbf{A}||^2$

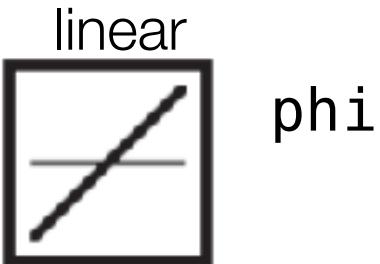
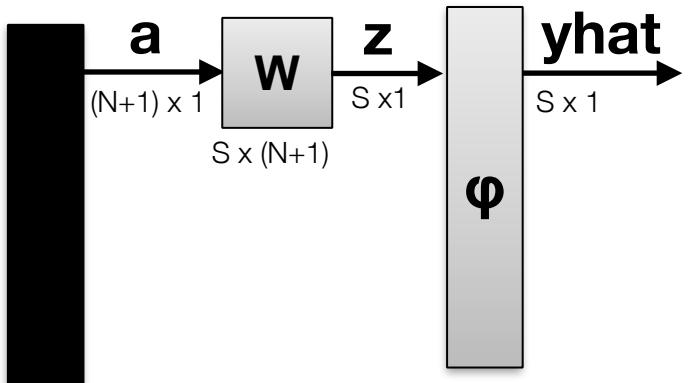
Need gradient $\nabla J(\mathbf{W})$ for update equation $\mathbf{W} \leftarrow \mathbf{W} + \eta \nabla J(\mathbf{W})$

We have been using the **Widrow-Hoff Learning Rule**

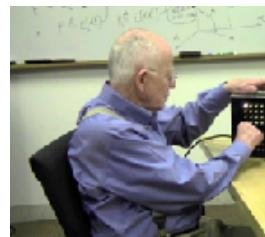


Simple Architectures

- Adaline network, Widrow and Hoff, 1960



Marcian “Ted” Hoff



Bernard Widrow

need gradient $\nabla J(\mathbf{W})$ for update equation $\mathbf{W} \leftarrow \mathbf{W} + \eta \nabla J(\mathbf{W})$

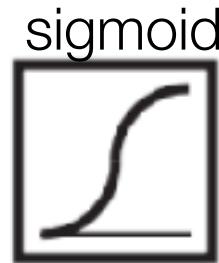
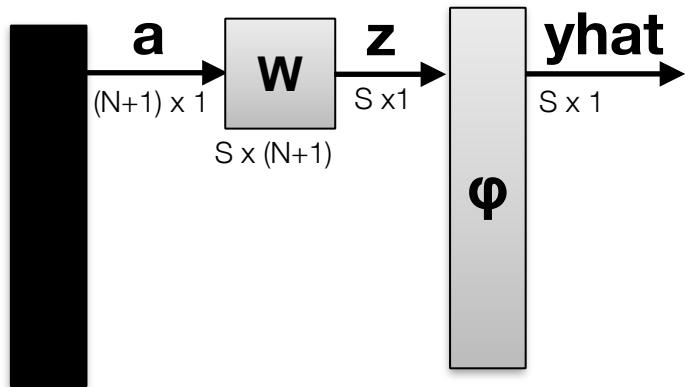
For case $S=1$, \mathbf{W} has only one row, \mathbf{w}
this is just **linear regression...**

$$\mathbf{w} \leftarrow \mathbf{w} + \eta [\mathbf{X} * (\mathbf{y} - \hat{\mathbf{y}})]$$



Simple Architectures

- Modern Perceptron network



$$\text{phi} = \frac{1}{1 + \exp(-z)}$$

need gradient $\nabla J(\mathbf{W})$ for update equation $\mathbf{W} \leftarrow \mathbf{W} + \eta \nabla J(\mathbf{W})$

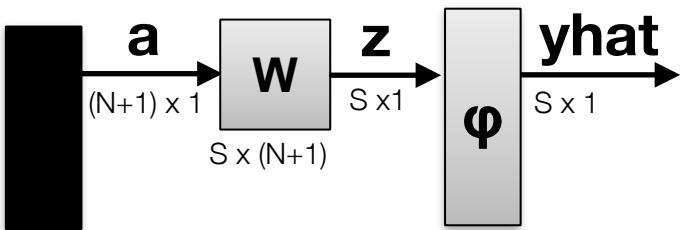
For case $S=1$, this is just **logistic regression...**
and **we have already solved this!**

$$\mathbf{w} \leftarrow \mathbf{w} + \eta [\mathbf{X} * (\mathbf{y} - g(\mathbf{x}))]$$



What happens when $S > 1$?

What if we have more than S=1?



$$\begin{bmatrix} [\phi(z_1)] \\ \dots \\ [\phi(z_S)] \end{bmatrix}^{(1)} \dots \begin{bmatrix} [\phi(z_1)] \\ \dots \\ [\phi(z_S)] \end{bmatrix}^{(M)} = \hat{\mathbf{Y}}$$

$$\begin{bmatrix} [y_1] \\ \dots \\ [y_S] \end{bmatrix}^{(1)} \dots \begin{bmatrix} [y_1] \\ \dots \\ [y_S] \end{bmatrix}^{(M)} = \mathbf{Y}$$

$$J(\mathbf{W}) = \|\mathbf{Y} - \hat{\mathbf{Y}}\|^2$$

Each target class in \mathbf{Y} can be independently optimized

$$yhat^{(i)} = \begin{bmatrix} \varphi_{\text{row}=1} \mathbf{w} \cdot \mathbf{x}^{(i)} \\ \varphi_{\text{row}=2} \mathbf{w} \cdot \mathbf{x}^{(i)} \\ \dots \\ \varphi_{\text{row}=S} \mathbf{w} \cdot \mathbf{x}^{(i)} \end{bmatrix}$$



which is one-versus-all!

$$J(1\mathbf{w}) = \sum_{i=1}^n [y_1(i) - \varphi(1\mathbf{w} \cdot \mathbf{x}(i))]^2$$

$$J(2\mathbf{w}) = \sum_{i=1}^n [y_2(i) - \varphi(2\mathbf{w} \cdot \mathbf{x}(i))]^2$$

...

$$J(S\mathbf{w}) = \sum_{i=1}^n [y_S(i) - \varphi(S\mathbf{w} \cdot \mathbf{x}(i))]^2$$

Simple Architectures: Summary

- Adaline network, Widrow and Hoff, 1960
 - linear regression
- Perceptron
 - *with sigmoid*: logistic regression
- One-versus-all implementation is the same as having $\mathbf{w}_{\text{class}}$ be rows of weight matrix, \mathbf{W}
 - works in adaline
 - works in logistic regression



these networks were created in the 50's and 60's
but were abandoned

why were they not used?

The Rosenblatt-Widrow-Hoff Dilemma

- 1960's: Rosenblatt got into a public academic argument with Marvin Minsky and Seymour Papert

"Given an elementary α -perceptron, a stimulus world W , and any classification $C(W)$ for which a solution exists; let all stimuli in W occur in any sequence, provided that each stimulus must reoccur in finite time; then beginning from an arbitrary initial state, an error correction procedure will always yield a solution to $C(W)$ in finite time..."

- Minsky and Papert publish limitations paper, 1969:

TED Ideas worth spreading

WATCH

DISCOVER

ATTEND

PARTICIPATE

Marvin Minsky:

Health and the human mind

TED2003 · 13:33 · Filmed Feb 2003

21 subtitle languages

View interactive transcript



More Advanced Architectures: history

- 1986: *Rumelhart, Hinton, and Williams* popularize gradient calculation for multi-layer network
 - *technically introduced by Werbos in 1982*
- **difference:** Rumelhart *et al.* validated ideas with a computer
- until this point no one could train a multiple layer network consistently
- algorithm is popularly called **Back-Propagation**
- wins pattern recognition prize in 1993, becomes de-facto machine learning algorithm until: SVMs and Random Forests in ~2004
- would eventually see a resurgence for its ability to train algorithms for Deep Learning applications: **Hinton is widely considered the father of deep learning**

David Rumelhart



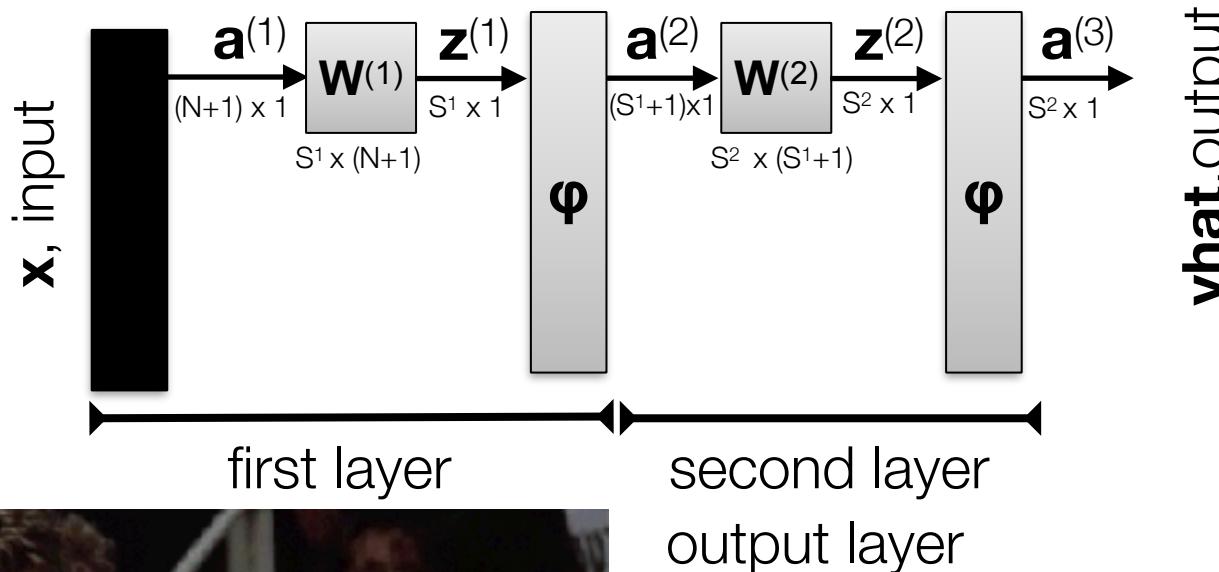
1942-2011

Geoffrey Hinton



More Advanced Architectures: MLP

- The multi-layer perceptron (MLP):
 - two layers shown, but could be arbitrarily many layers
 - algorithm is agnostic to number of layers (*kinda*)



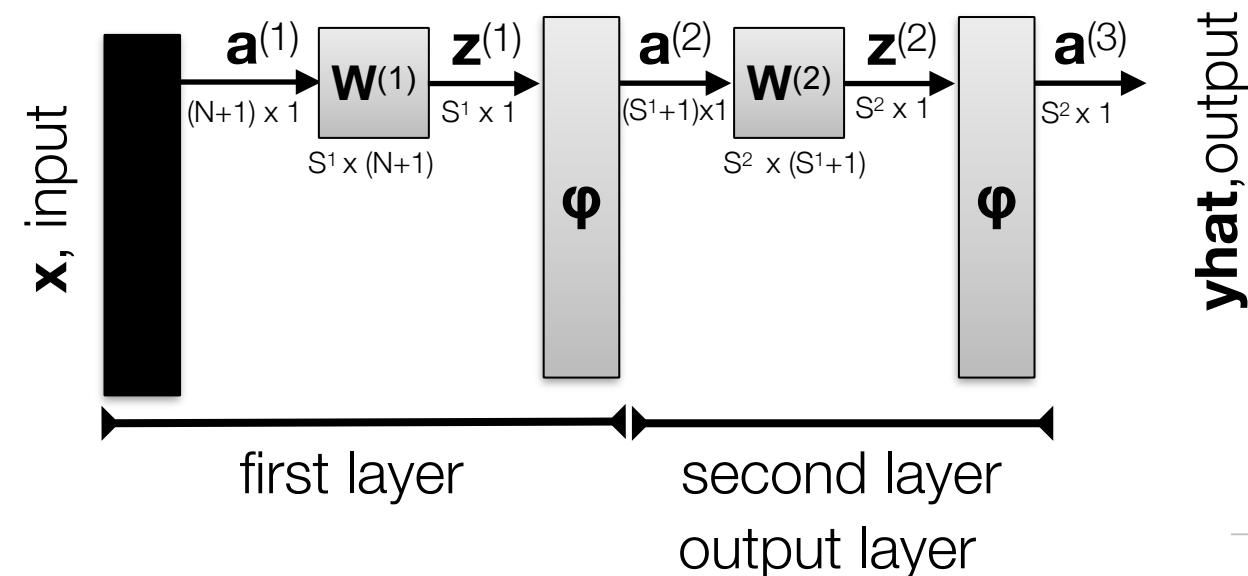
each row of $y\hat{}$
is no longer
independent of
the rows in W
so we cannot
optimize using
one versus all!!!



$$y\hat{^{(1)}} = \begin{bmatrix} \varphi_{(\text{row}=1)} \mathbf{w}^{(2)} \cdot \varphi(\mathbf{W}^{(1)} \mathbf{a}^{(1)}) \\ \vdots \\ \varphi_{(\text{row}=S)} \mathbf{w}^{(2)} \cdot \varphi(\mathbf{W}^{(1)} \mathbf{a}^{(1)}) \end{bmatrix}$$

Back Propagation

- Steps:
 - propagate weights forward
 - calculate gradient at final layer
 - back propagate gradient for each layer
 - via recurrence relation

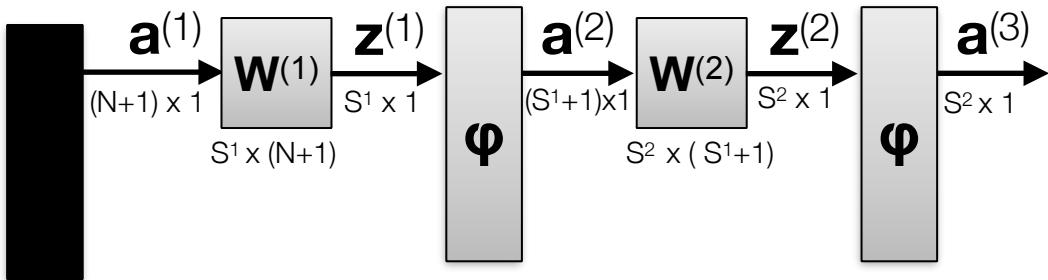


$$J(\mathbf{W}) = \sum_k^M (\hat{\mathbf{y}}^{(k)} - \mathbf{a}^{(L)})^2$$

$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta \frac{\partial J(\mathbf{W})}{\partial w_{i,j}^{(l)}}$$

Back propagation

$$J(\mathbf{W}) = \sum_k^M (\mathbf{y}^{(k)} - \mathbf{a}^{(L)})^2$$



$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta \frac{\partial J(\mathbf{W})}{\partial w_{i,j}^{(l)}}$$

use chain rule:

$$\frac{\partial J(\mathbf{W})}{\partial w_{i,j}^{(l)}} = \frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}} \frac{\partial \mathbf{z}^{(l)}}{\partial w_{i,j}^{(l)}}$$

Solve this in explainer video for next
in class assignment!

Town Hall (if time)



End of Session

- thanks! **Next time is Flipped Assignment!!!**

More help on neural networks to prepare for next time:

Sebastian Raschka

<https://github.com/rasbt/python-machine-learning-book/blob/master/code/ch12/ch12.ipynb>

Martin Hagan

[https://www.google.com/url?
sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwioprvn
27fPAhWMx4MKHYbwDlwQFggeMAA&url=http%3A%2F%2Fhagan.okstate.edu%
2FNNDesign.pdf&usg=AFQjCNG5YbM4xSMm6K5HNsG-4Q8TvOu_Lw&sig2=bgT3
k-5ZDDTPZ07Qu8Oreg](https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwioprvn27fPAhWMx4MKHYbwDlwQFggeMAA&url=http%3A%2F%2Fhagan.okstate.edu%2FNNDesign.pdf&usg=AFQjCNG5YbM4xSMm6K5HNsG-4Q8TvOu_Lw&sig2=bgT3k-5ZDDTPZ07Qu8Oreg)

Michael Nielsen

<http://neuralnetworksanddeeplearning.com>

Lecture Notes for **Machine Learning in Python**

Professor Eric Larson
Optimizing Neural Networks

Class Logistics and Agenda

- Logistics
- Agenda:
 - Finish Town Hall
 - Practical Multi-layer Architectures
 - Programming Examples
- Next Time: More MLPs

Town Hall



Tyler Rablin @Mr_Rablin · 2d



You're not grading assignments.

You're collecting evidence to determine student progress and pointing them towards their next steps.

Make the mental switch. It matters.

Semester Summary, so far!

- Adaline network, Widrow and Hoff, 1960
 - iterative linear regression
- Perceptron
 - *with sigmoid*: logistic regression
- One-versus-all implementation is the same as having $\mathbf{w}_{\text{class}}$ be rows of weight matrix, \mathbf{W}
 - works in adaline
 - works in logistic regression

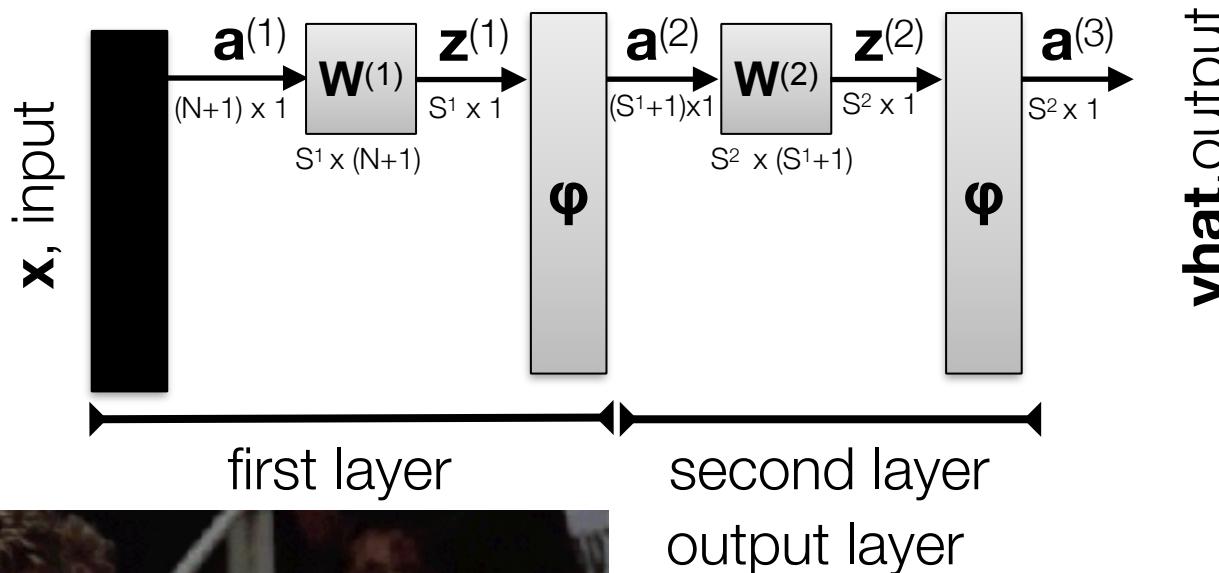


these networks were created in the 50's and 60's
but were abandoned

why were they not used?

More Advanced Architectures: MLP

- The multi-layer perceptron (MLP):
 - two layers shown, but could be arbitrarily many layers



each row of \hat{y}
is no longer
independent of
the rows in W
so we cannot
optimize using
one versus all!!!



$$\hat{y}^{(1)} = \begin{bmatrix} \varphi(\text{row}=1 W^{(2)} \cdot \varphi(W^{(1)} a^{(1)})) \\ \vdots \\ \varphi(\text{row}=S W^{(2)} \cdot \varphi(W^{(1)} a^{(1)})) \end{bmatrix}$$

The Rosenblatt-Widrow-Hoff Dilemma

- 1960's: Rosenblatt got into a public academic argument with Marvin Minsky and Seymour Papert

"Given an elementary α -perceptron, a stimulus world W , and any classification $C(W)$ for which a solution exists; let all stimuli in W occur in any sequence, provided that each stimulus must reoccur in finite time; then beginning from an arbitrary initial state, an error correction procedure will always yield a solution to $C(W)$ in finite time..."

- Minsky and Papert publish limitations paper, 1969:

TED Ideas worth spreading

WATCH

DISCOVER

ATTEND

PARTICIPATE

Marvin Minsky:

Health and the human mind

TED2003 · 13:33 · Filmed Feb 2003

21 subtitle languages

View interactive transcript



More Advanced Architectures: history

- 1986: *Rumelhart, Hinton, and Williams* popularize gradient calculation for multi-layer network
 - *actually* introduced by Werbos in 1982
- **difference:** Rumelhart *et al.* validated ideas with a computer
- until this point no one could train a multiple layer network consistently
- algorithm is popularly called **Back-Propagation**
- wins pattern recognition prize in 1993, becomes de-facto machine learning algorithm until: SVMs and Random Forests in ~2004
- would eventually see a resurgence for its ability to train algorithms for Deep Learning applications: **Hinton is widely considered the father of deep learning**

David Rumelhart



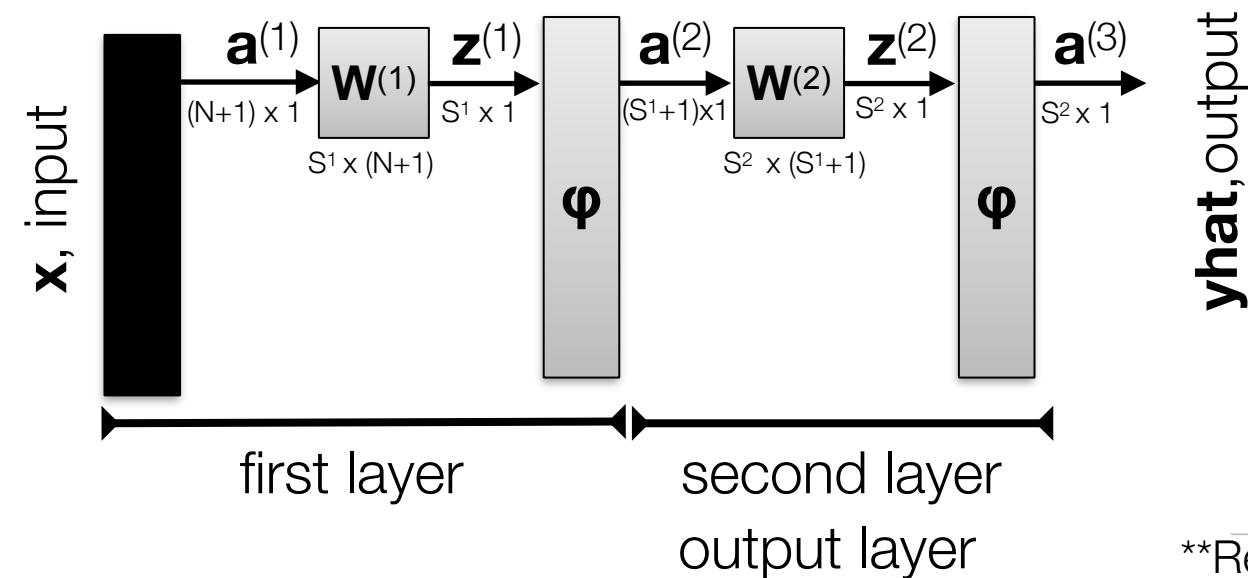
1942-2011

Geoffrey Hinton



Back propagation

- Steps:
 - propagate weights forward
 - calculate gradient at final layer
 - back propagate gradient for each layer
 - via recurrence relation

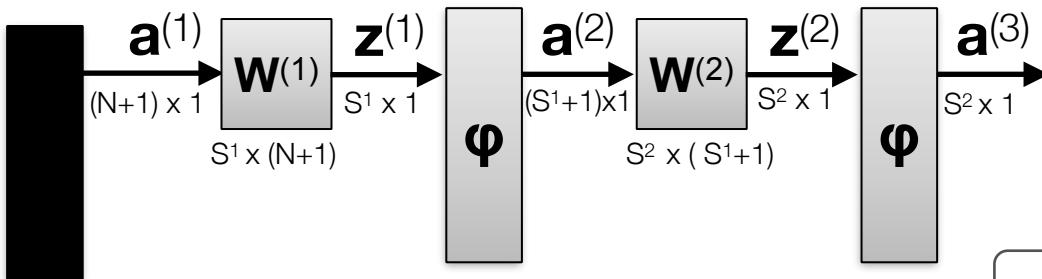


$$J(\mathbf{W}) = \|\mathbf{Y} - \hat{\mathbf{Y}}\|^2$$

$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta \frac{\partial J(\mathbf{W})}{\partial w_{i,j}^{(l)}}$$

**Recall from Flipped Assignment!

Back Propagation Summary



1. Forward propagate to get Z, A

2. Get final layer gradient

3. Back propagate sensitivities

4. Update each $W^{(l)}$

$$\mathbf{V}^{(2)} = -2(\mathbf{Y} - \mathbf{A}^{(3)}) * \mathbf{A}^{(3)} * (1 - \mathbf{A}^{(3)})$$

$$\nabla^{(2)} = \mathbf{V}^{(2)} \cdot [\mathbf{A}^{(2)}]^T$$

$$\mathbf{V}^{(1)} = \mathbf{A}^{(2)} * (1 - \mathbf{A}^{(2)}) * [\mathbf{W}^{(2)}]^T \cdot \mathbf{V}^{(2)}$$

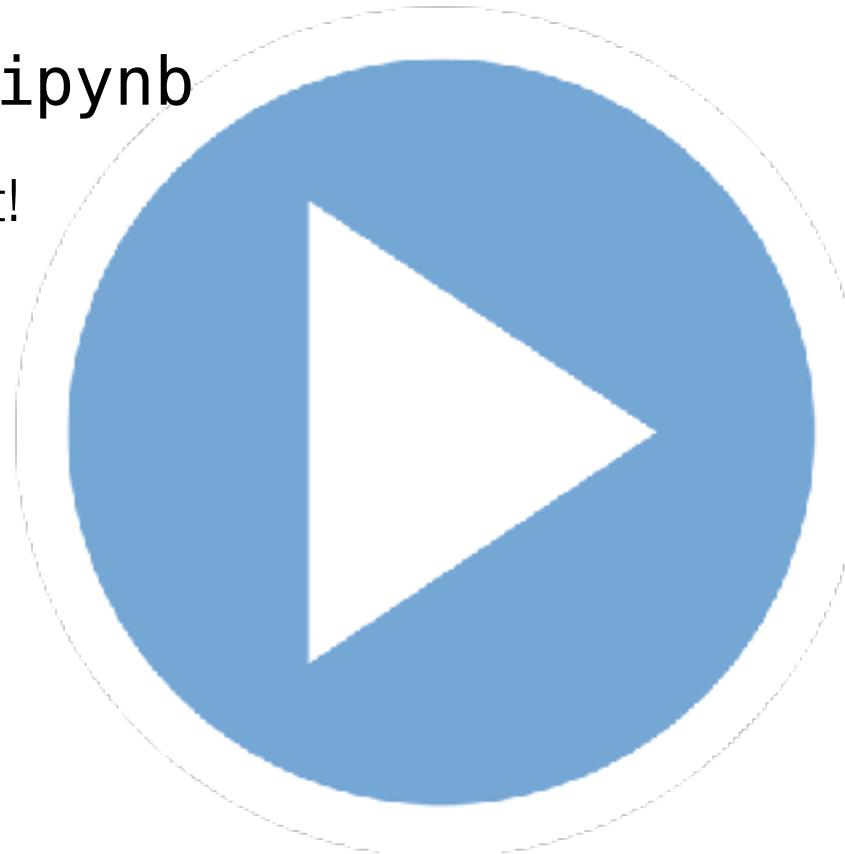
$$\nabla^{(1)} = \mathbf{V}^{(1)} \cdot [\mathbf{A}^{(1)}]^T$$

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \nabla^{(l)}$$

**Recall from Flipped Assignment!

07. MLP Neural Networks.ipynb

same as Flipped Assignment!
with regularization
and vectorization

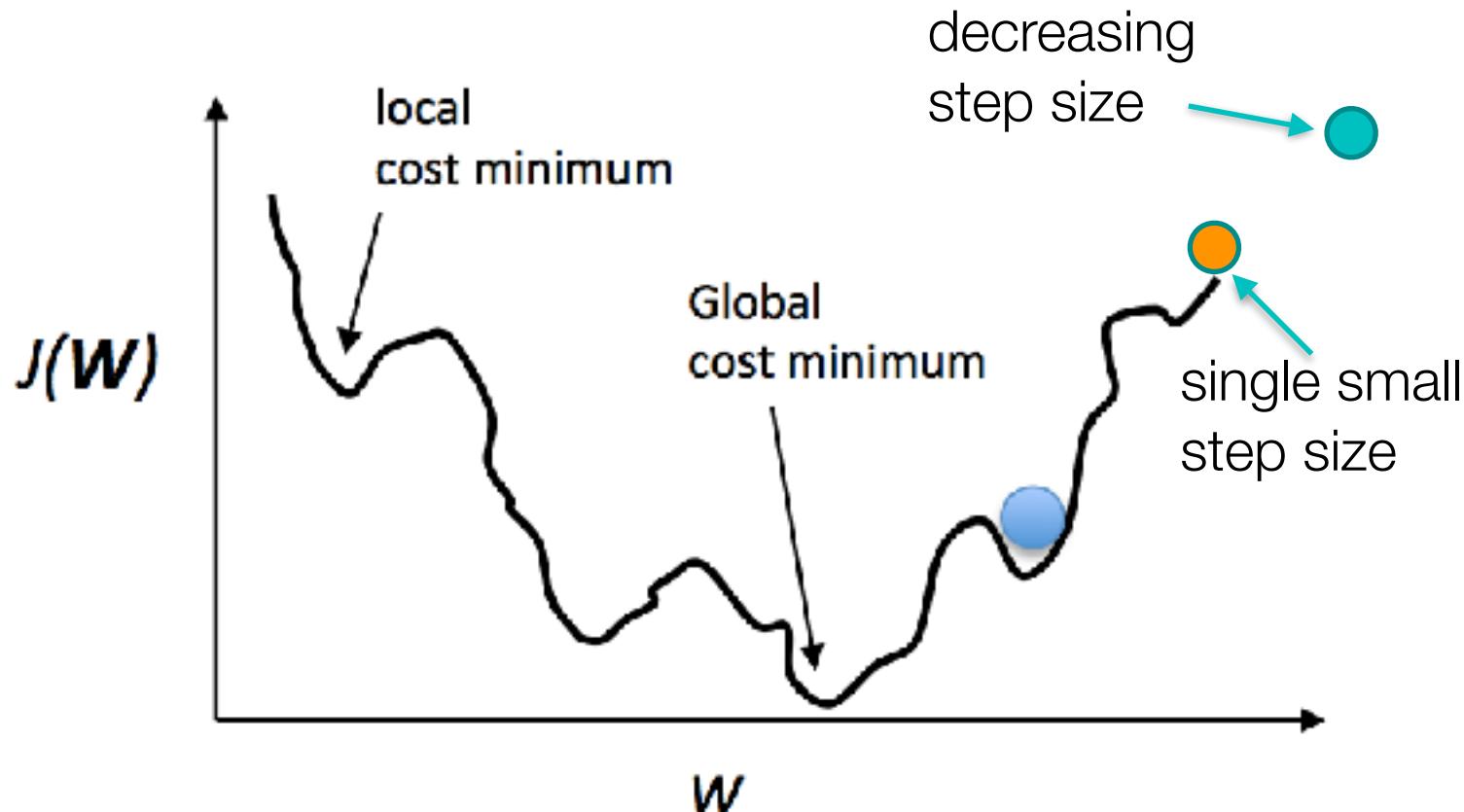


Problems with Advanced Architectures

- Numerous weights to find gradient update
 - minimize number of instances
 - **solution:** mini-batch
- **new problem:** mini-batch gradient can be erratic
 - **solution:** momentum
 - use previous update in current update

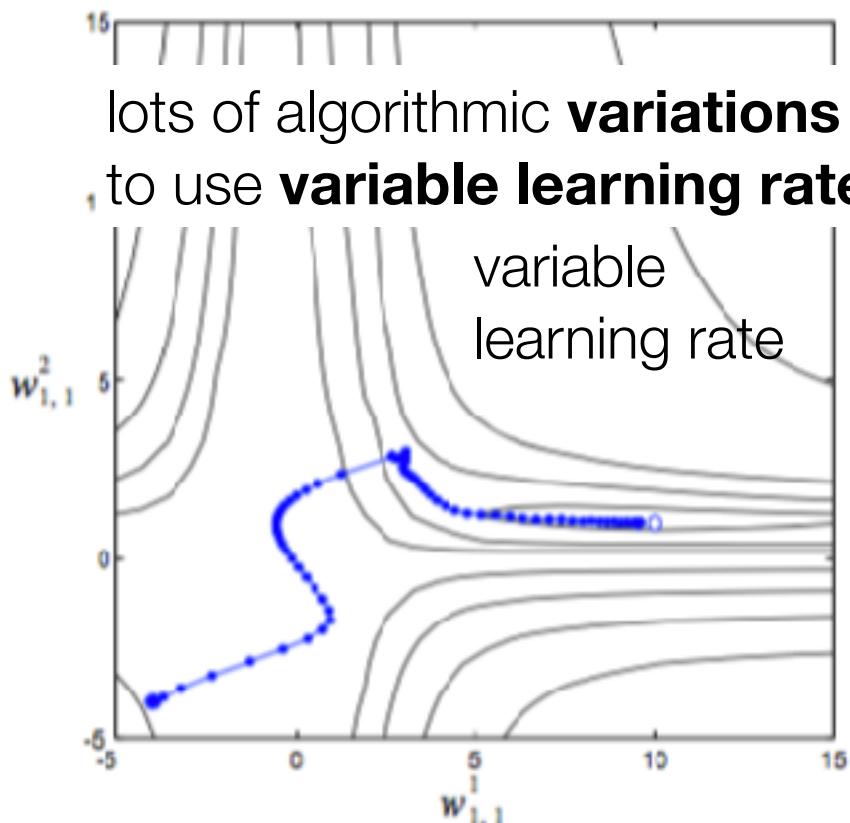
Problems with Advanced Architectures

- Space is no longer convex
 - **One solution:**
 - start with large step size
 - “cool down” by decreasing step size for higher iterations

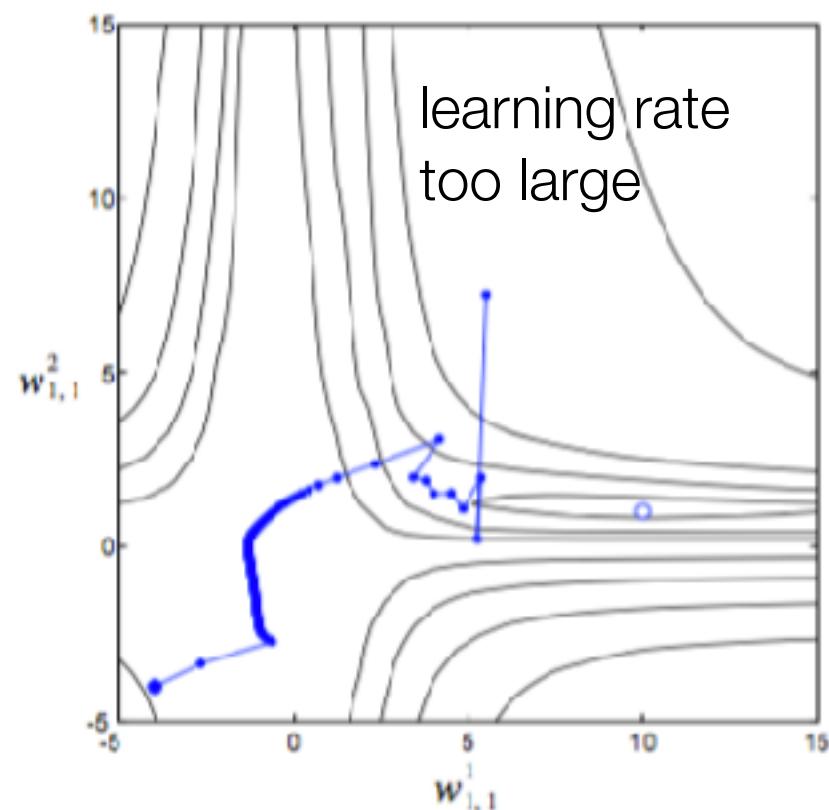


Problems with Advanced Architectures

- Space is no longer convex
 - **another solution:**
 - start with arbitrary step size
 - only decrease when successive iterations do not decrease cost

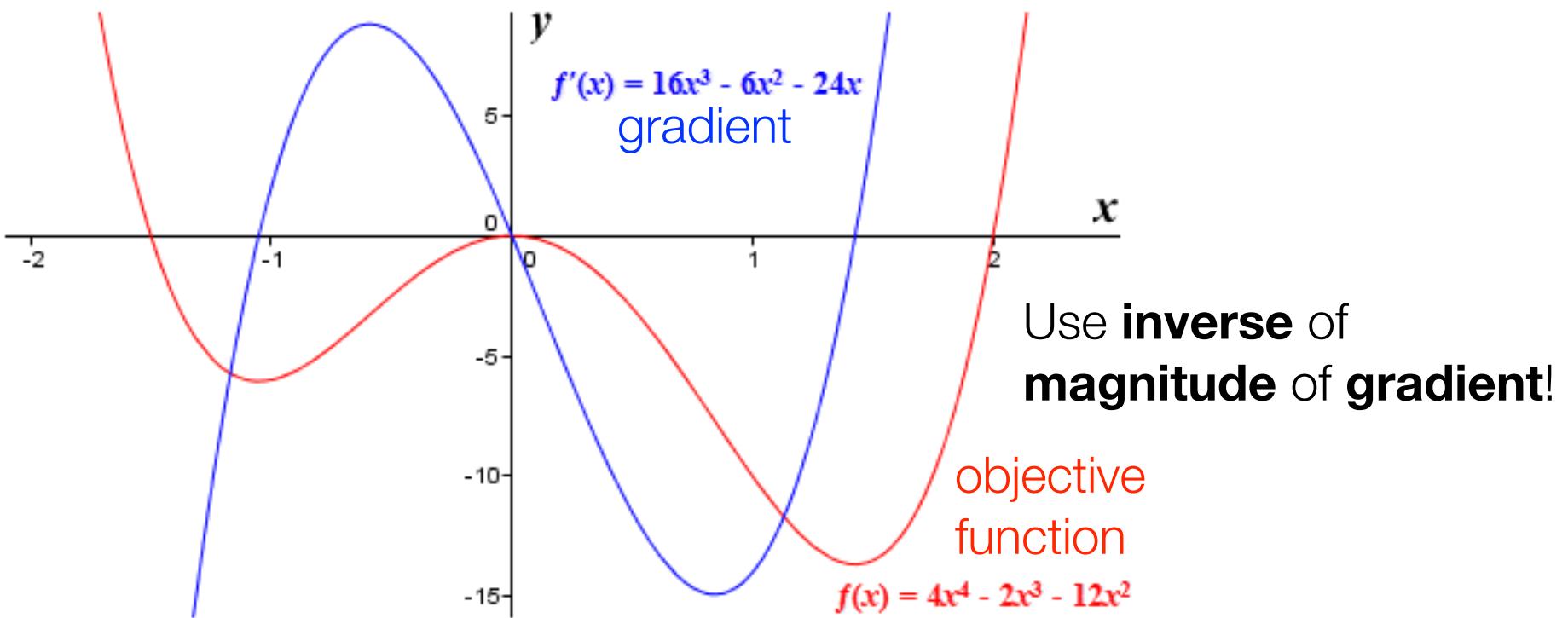


lots of algorithmic **variations**
to use **variable learning rate**



Learning Rate from Gradient Magnitude?

- Deccelerate down regions that are steep
- Accelerate on plateaus



Also **accumulate inverse** to be robust to many **changes** in **steepness**...

Common Adaptive Strategies

$$\mathbf{W}_{k+1} = \mathbf{W}_k - \rho_k$$

- Momentum

$$\rho_k = \alpha \nabla J(\mathbf{W}_k) + \beta \nabla J(\mathbf{W}_{k-1})$$

- Nesterov's Accelerated Gradient

$$\rho_k = \underbrace{\beta \nabla J(\mathbf{W}_k + \alpha \nabla J(\mathbf{W}_{k-1}))}_{\text{step twice}} + \alpha \nabla J(\mathbf{W}_{k-1})$$

- AdaGrad

$$\rho_k = \frac{\eta}{\sqrt{\text{trace}(G_k) + \epsilon}} * \nabla J(\mathbf{W}_k)$$

where

$$G_k = G_{k-1} + \nabla J(\mathbf{W}_k) \cdot \nabla J(\mathbf{W}_k)^T$$

- AdaDelta

G accumulated over a fixed time window, not all previous iterations

- RMSProp

G learning rate decreases with squared gradient magnitude

- AdaM

G updates with decaying momentum of J and J^2

- NAdaM

same as Adam, but with nesterov's acceleration

- None** of these are “**one-size-fits-all**” because the space of neural network **optimization varies** by problem

- ADAM is **popular** but **not a panacea**
- Global Optimum == Overfit**

Adam, a more detailed look

Published as a conference paper at ICLR 2015

ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

Diederik P. Kingma*

University of Amsterdam, OpenAI

Jimmy Lei Ba*

University of Toronto

Algorithm 1: Adam, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

Require: α : Stepsize

Require: $\beta_1, \beta_2 \in [0, 1]$: Exponential decay rates for the moment estimates

Require: $f(\theta)$: Stochastic objective function with parameters θ

Require: θ_0 : Initial parameter vector

$m_0 \leftarrow 0$ (Initialize 1st moment vector)

$v_0 \leftarrow 0$ (Initialize 2nd moment vector)

$t \leftarrow 0$ (Initialize timestep)

while θ_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t) get gradient

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate) gradient momentum

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate) squared gradient momentum

$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate) boost moments magnitudes

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate) (notice t in exponent)

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters) update gradient, normalized by second moment

end while similar to Adaptive Gradient (AdaGrad)

return θ_t (Resulting parameters)

07. MLP Neural Networks.ipynb

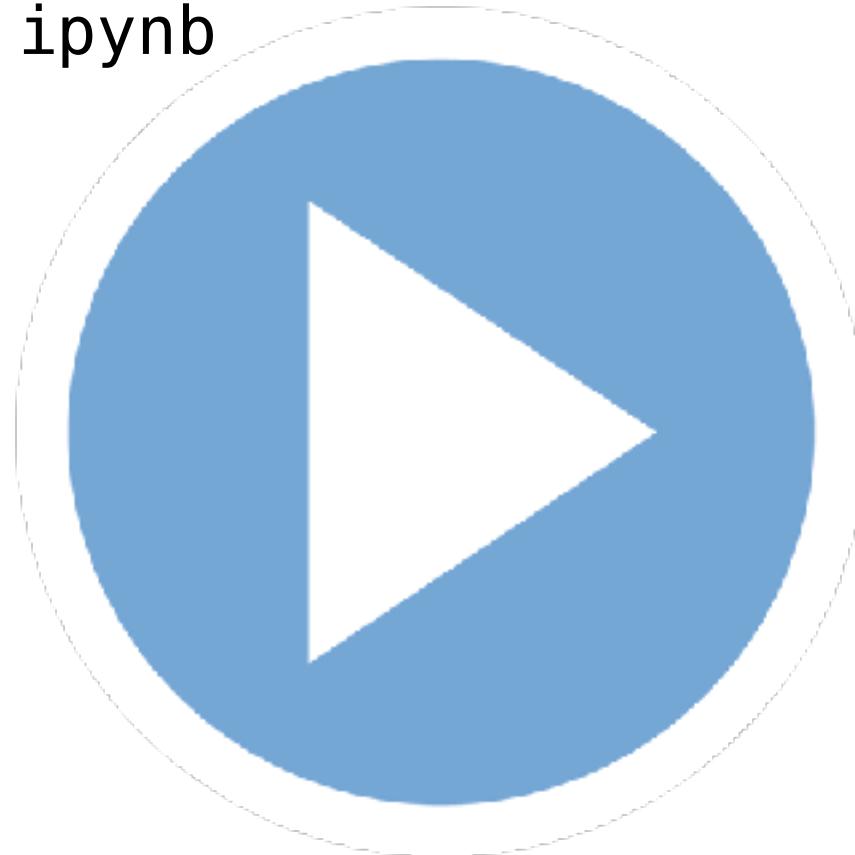
comparison:

mini-batch

momentum

adaptive learning

L-BFGS



Lecture Notes for **Machine Learning in Python**

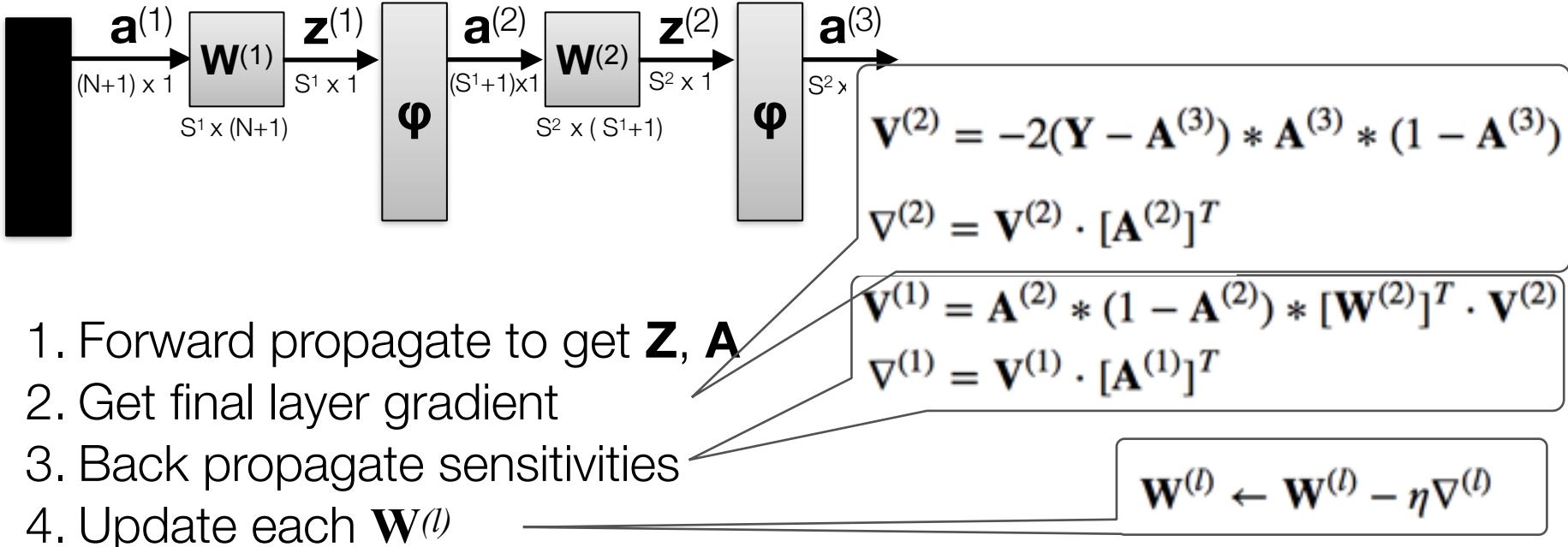
Professor Eric Larson

Neural Network Optimization and Activation

Class Logistics and Agenda

- Agenda:
 - More optimization and architectures
 - Programming Examples
 - Town Hall

Review:



- **Self Test:**

True or False: If we change the cost function, $J(W)$, we only need to update the final layer sensitivity calculation, $V^{(2)}$, of the back propagation steps. The remainder of the algorithm is unchanged.

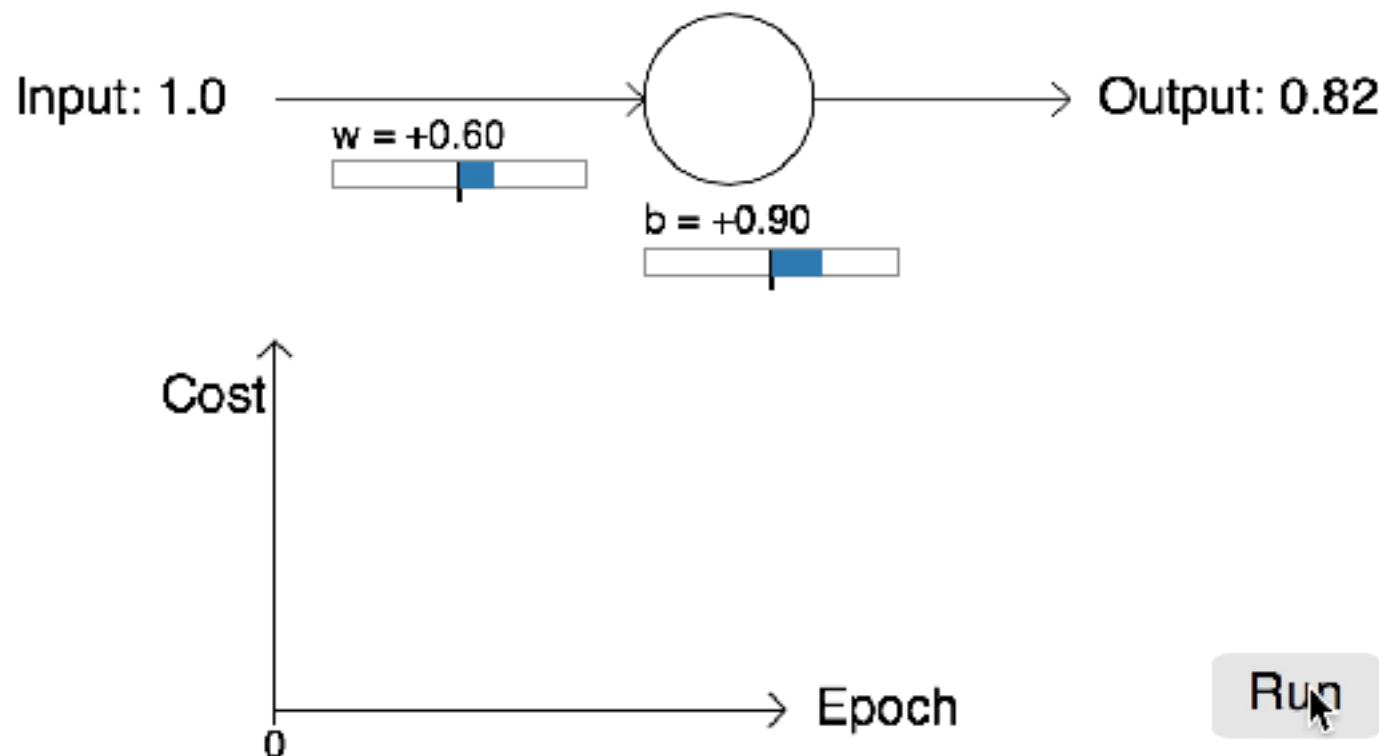
- A. True
- B. False

Practical Implementation of Architectures

- MSE

$$J(\mathbf{W}) = \sum_k^M (\mathbf{y}^{(k)} - \mathbf{a}^{(L)})^2$$

least squares objective,
tends to slow training initially

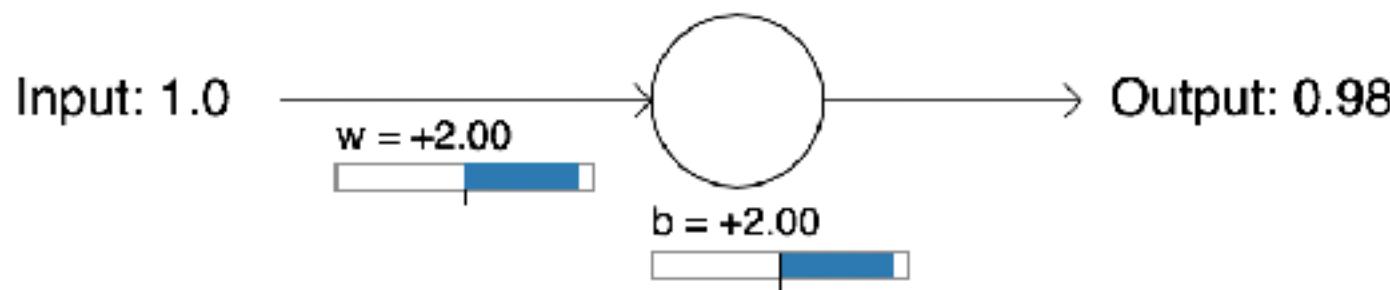


Practical Implementation of Architectures

- MSE

$$J(\mathbf{W}) = \sum_k^M (\mathbf{y}^{(k)} - \mathbf{a}^{(L)})^2$$

least squares objective,
tends to slow training initially

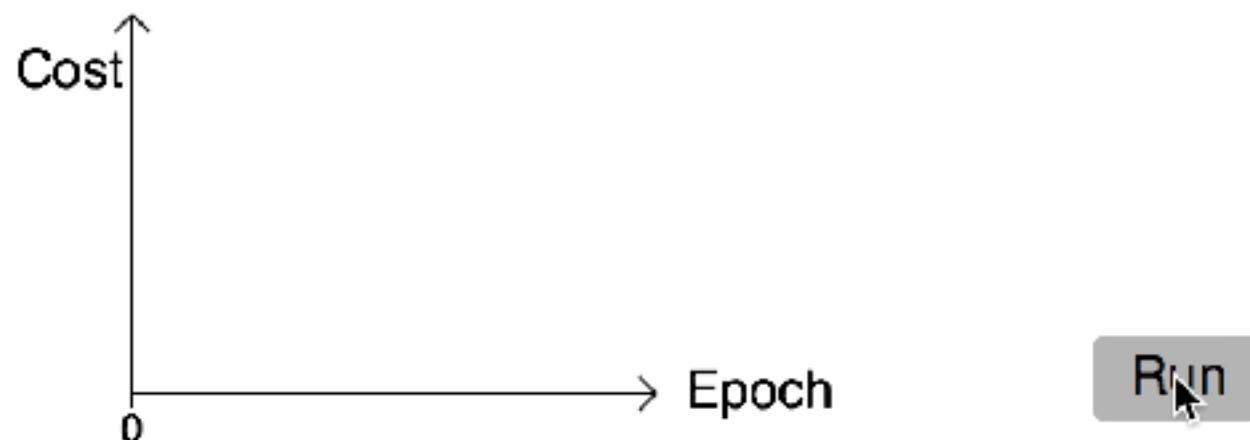
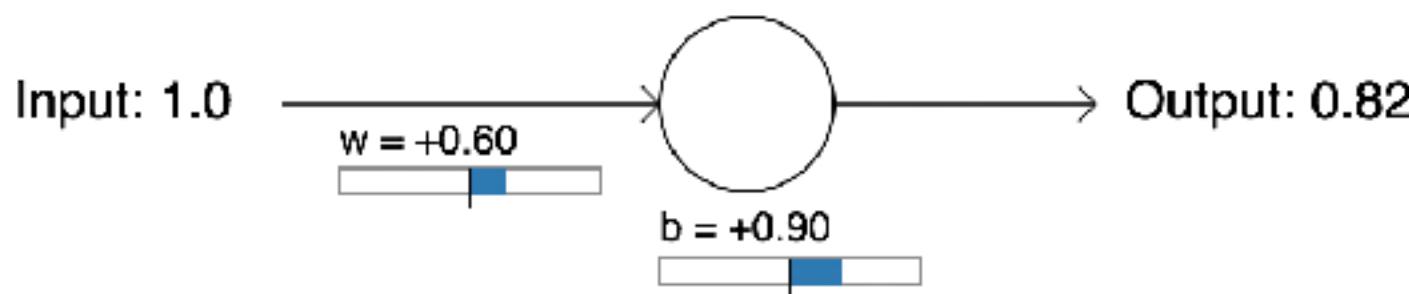


Practical Implementation of Architectures

- Back to our old friend: **Cross entropy**

$$J(\mathbf{W}) = - \left[\mathbf{y}^{(i)} \ln([\mathbf{a}^{(L+1)}]^{(i)}) + (1 - \mathbf{y}^{(i)}) \ln(1 - [\mathbf{a}^{(L+1)}]^{(i)}) \right]$$

speeds up
initial training



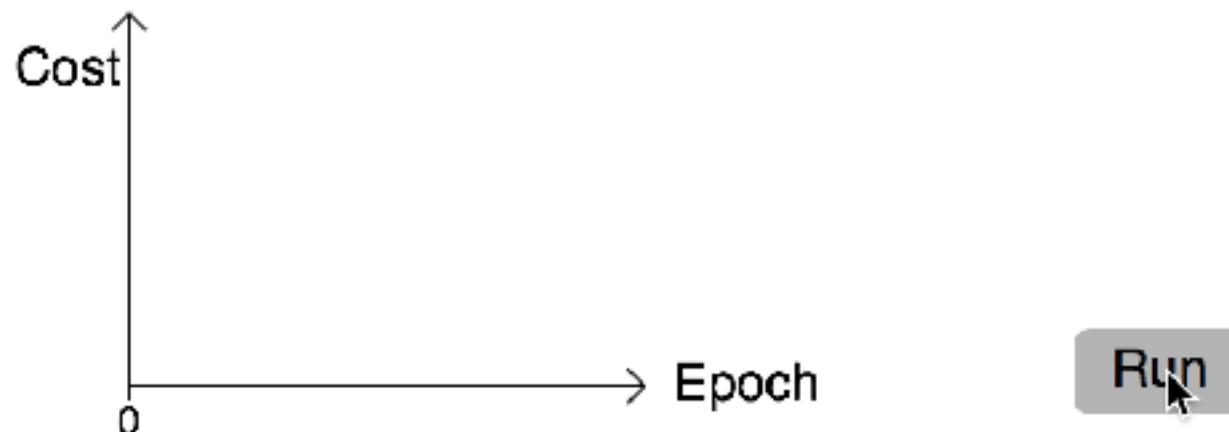
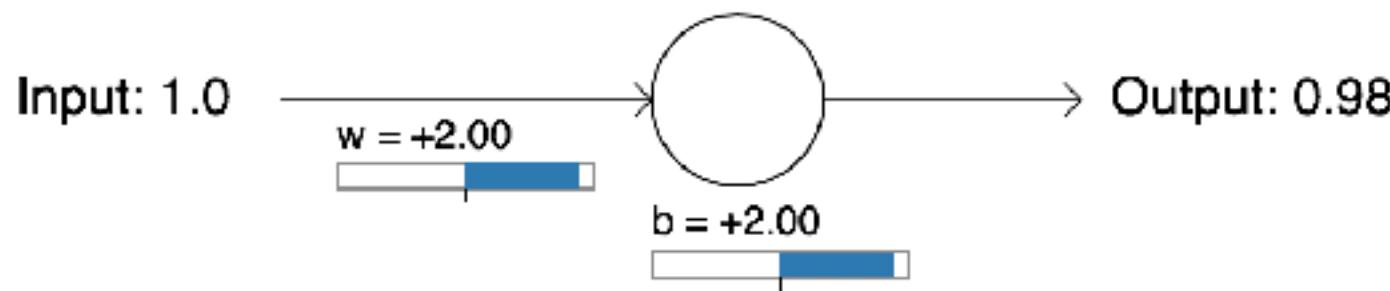
Neural Networks and Deep Learning, Michael Nielson, 2015

Practical Implementation of Architectures

- Back to our old friend: **Cross entropy**

$$J(\mathbf{W}) = - \left[\mathbf{y}^{(i)} \ln([\mathbf{a}^{(L+1)}]^{(i)}) + (1 - \mathbf{y}^{(i)}) \ln(1 - [\mathbf{a}^{(L+1)}]^{(i)}) \right]$$

speeds up
initial training



Practical Implementation of Architectures

- Back to our old friend: **Cross entropy**

$$J(\mathbf{W}) = - \left[\mathbf{y}^{(i)} \ln([\mathbf{a}^{(L+1)}]^{(i)}) + (1 - \mathbf{y}^{(i)}) \ln(1 - [\mathbf{a}^{(L+1)}]^{(i)}) \right]$$

speeds up
initial training

$$\left[\frac{\partial J(\mathbf{W})}{\mathbf{z}^{(L)}} \right]^{(i)}$$

$$\mathbf{V}^{(2)} = -2(\mathbf{Y} - \mathbf{A}^{(3)}) * \mathbf{A}^{(3)} * (1 - \mathbf{A}^{(3)}) \quad \text{old update}$$

Practical Implementation of Architectures

- Back to our old friend: **Cross entropy**

$$J(\mathbf{W}) = - \left[\mathbf{y}^{(i)} \ln([\mathbf{a}^{(L+1)}]^{(i)}) + (1 - \mathbf{y}^{(i)}) \ln(1 - [\mathbf{a}^{(L+1)}]^{(i)}) \right]$$

speeds up
initial training

$$\left[\frac{\partial J(\mathbf{W})}{\mathbf{z}^{(L)}} \right]^{(i)} = ([\mathbf{a}^{(L+1)}]^{(i)} - \mathbf{y}^{(i)})$$

$$\left[\frac{\partial J(\mathbf{W})}{\mathbf{z}^{(2)}} \right]^{(i)} = ([\mathbf{a}^{(3)}]^{(i)} - \mathbf{y}^{(i)})$$

$$\mathbf{V}^{(2)} = \mathbf{A}^{(3)} - \mathbf{Y}$$

new update

```
# vectorized backpropagation
v2 = (A3-Y_enc) # <- this is only line t
v1 = A2*(1-A2)*(W2.T @ v2)

grad2 = v2 @ A2.T
grad1 = v1[1:,:,:] @ A1.T
```

$$\mathbf{V}^{(2)} = -2(\mathbf{Y} - \mathbf{A}^{(3)}) * \mathbf{A}^{(3)} * (1 - \mathbf{A}^{(3)}) \quad \text{old update}$$

bp-5

53

cross entropy



Practical Implementation of Architectures

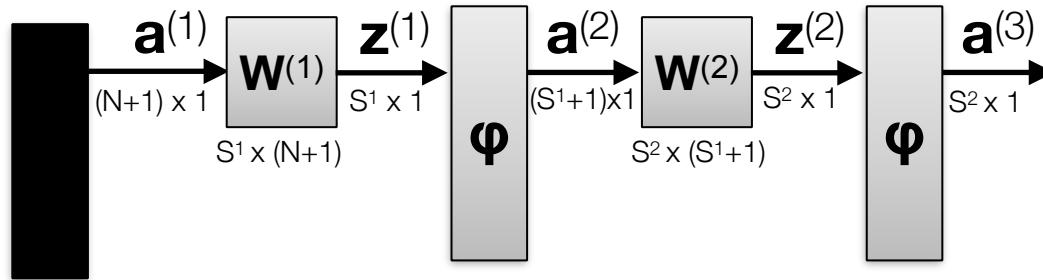
**Gradient when using cosine annealing with
warm restarts learning rate scheduler**



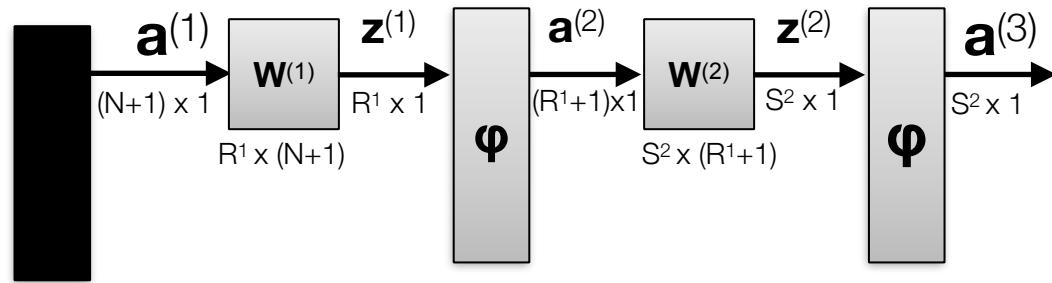
Expansion and Dropout

Practical Implementation of Architectures

- Dropout
 - don't train all hidden neurons at the same time



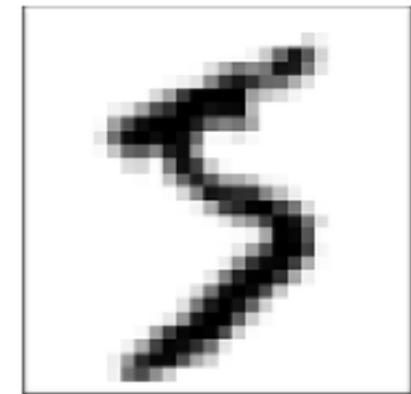
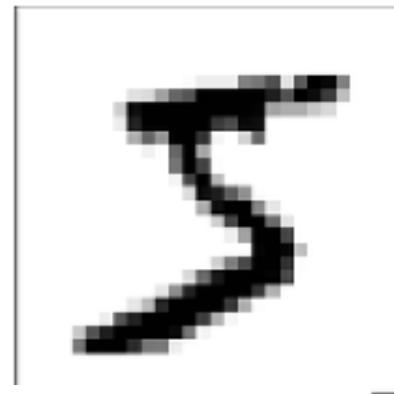
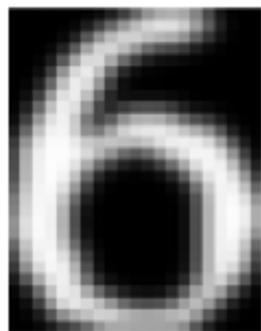
For each mini-batch,
 R^1 is subset of S^1



Why does this work?

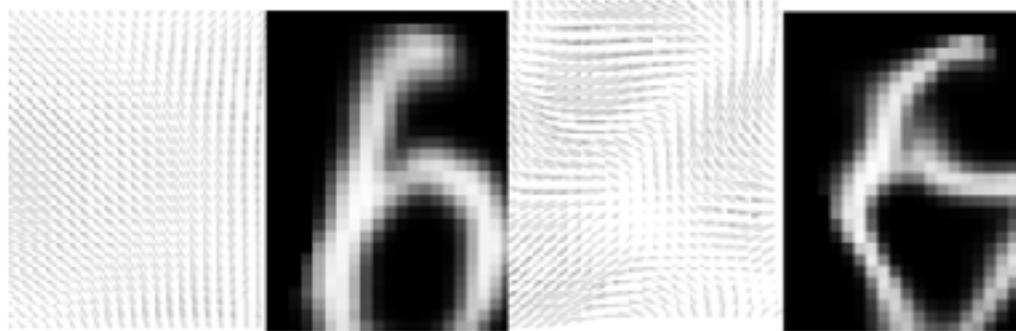
Practical Implementation of Architectures

- Expansion
 - get more data
 - perturb your data



Neural Networks and Deep Learning,
Michael Nielson, 2015

Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis, by Patrice Simard, Dave Steinkraus, and John Platt (2003)



98.4% → 99.3%

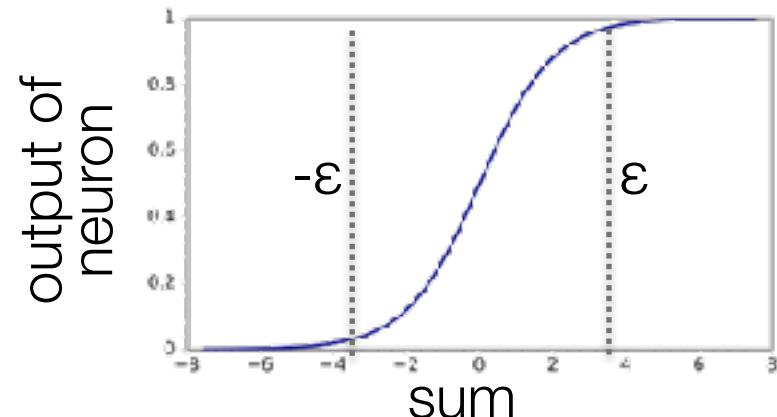
Practical Implementation of Architectures

- Weight initialization
 - try not to **saturate** your neurons right away!

$$\mathbf{a}^{(L+1)} = \phi(\mathbf{W}^{(L)} \mathbf{a}^{(L)})$$



each row is summed before sigmoid



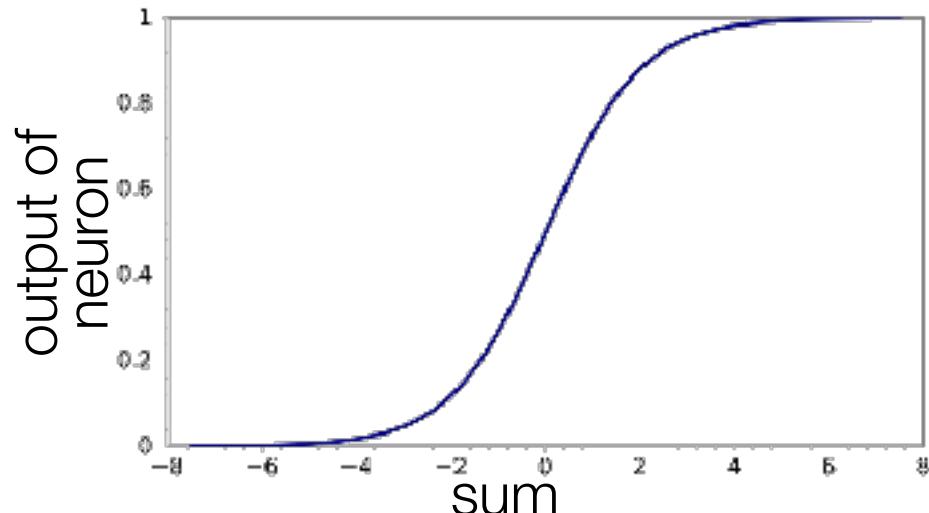
want sum to be between $-\epsilon < \sum < \epsilon$ for no saturation

solution: squash initial weights magnitude

- one choice: each element of \mathbf{W} selected from a Gaussian with **zero mean** and **unit standard deviation**
- for adding Gaussian distributions, variances add together:
 - make each variance $1/\mathbf{W}_{\text{num_elements_in_row}}$
 - which is the same as standard deviation = $1/\sqrt{\mathbf{W}_{\text{num_elements_in_row}}}$

Self Test

- for adding Gaussian distributions, variances add together
$$\mathbf{a}^{(L+1)} = \Phi(\mathbf{W}^{(L)} \mathbf{a}^{(L)})$$
- If you initialized the weights with too large variance, you would expect the output of the neuron, Φ , to be:
 - A. saturated to “1”
 - B. saturated to “0”
 - C. could either be saturated to “0” or “1”
 - D. would not be saturated



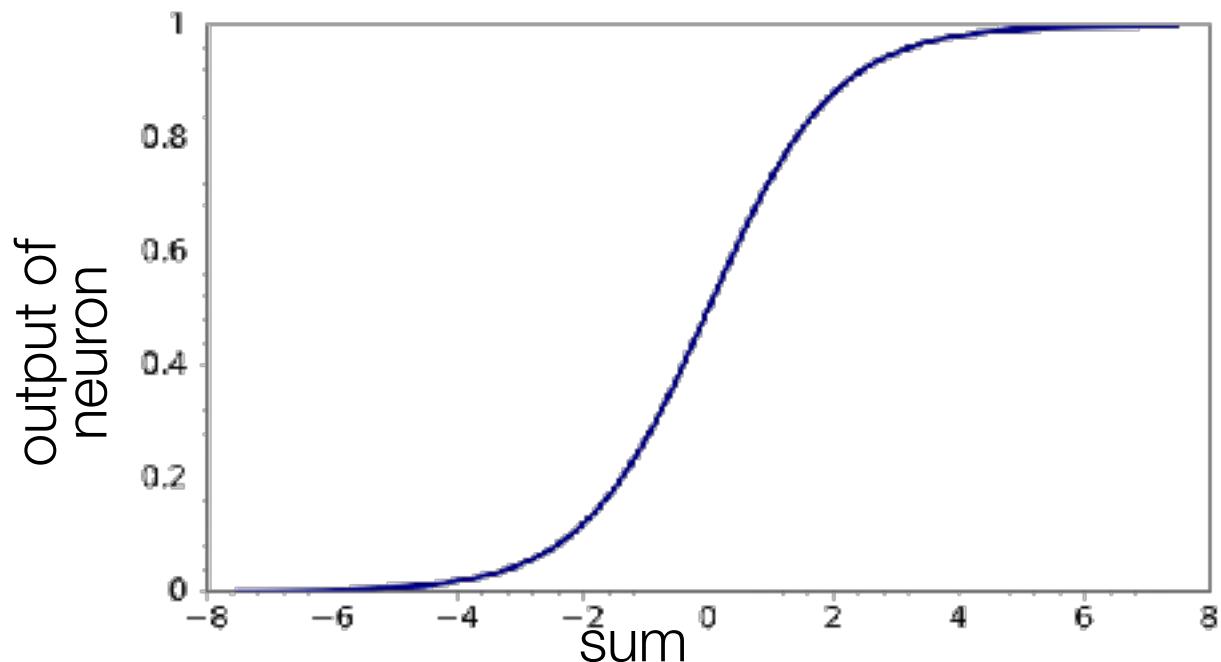
Self Test

- for adding Gaussian distributions, variances add together

$$\mathbf{a}^{(L+1)} = \phi(\mathbf{W}^{(L)}\mathbf{a}^{(L)})$$

- What is the derivative of a saturated sigmoid neuron?

- A. zero
- B. one
- C. $a * (1-a)$
- D. it depends



More Weight Initialization

Understanding the difficulty of training deep feedforward neural networks

Xavier Glorot

DIRO, Université de Montréal, Montréal, Québec, Canada

Yoshua Bengio

We should not saturate **feedforward** or **back propagated** variance

$$y_i = \sum_j^{n^{(L)}} w_{ij} x_j \quad \text{break down feed forward by each multiply}$$

$$\text{Var}[y_i] = \sum_j^{n^{(L)}} \underbrace{E[w_{ij}]^2 \text{Var}[x_j] + \text{Var}[w_{ij}]E[x_j]^2 + \text{Var}[w_{ij}]\text{Var}[x_j]}_{0, \text{ if uncorrelated}} \quad \text{assume i.i.d.}$$

Similar for back prop.

$$\text{Var}[y_i] = n^{(L)} \text{Var}[w_{ij}] \text{Var}[x_j]$$

$$\text{Var}[s_i^{(L)}] = n^{(L+1)} \text{Var}[w_{ij}] \text{Var}[s_j^{(L+1)}]$$

$$w_{ij} \approx \mathcal{N}\left(0, \sqrt{\frac{1}{n^{(L)}}}\right)$$

forward
from data

$$w_{ij} \approx \mathcal{N}\left(0, \sqrt{\frac{1}{n^{(L+1)}}}\right)$$

backward
from sensitivity.

$$w_{ij} \approx \mathcal{N}\left(0, \sqrt{\frac{2}{n^{(L)} + n^{(L+1)}}}\right)$$

compromise

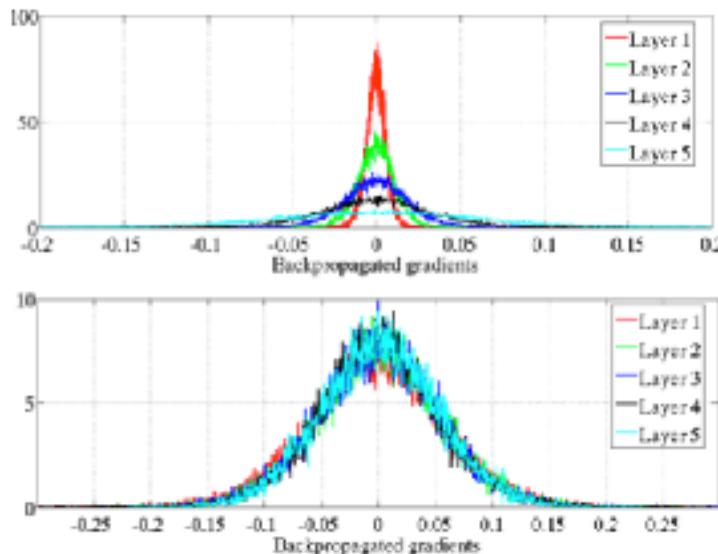
More Weight Initialization

Understanding the difficulty of training deep feedforward neural networks

Xavier Glorot

DIRO, Université de Montréal, Montréal, Québec, Canada

Yoshua Bengio



Starting gradient magnitudes
per layer
standard normalization

Starting gradient magnitudes
per layer
Glorot normalization

Figure 7: Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.

Glorot and He Initialization

We have solved this assuming the weights need be in the range -1 to 1 and assuming that x is distributed Gaussian (\tanh)

This range is different depending on the activation and assuming Gaussian or Uniform

	Uniform	Gaussian
Tanh	$\sigma = \sqrt{\frac{6}{n^{(L)} + n^{(L+1)}}}$	$\sigma = \sqrt{\frac{2}{n^{(L)} + n^{(L+1)}}}$
Sigmoid	$\sigma = 4\sqrt{\frac{6}{n^{(L)} + n^{(L+1)}}}$	$\sigma = 4\sqrt{\frac{2}{n^{(L)} + n^{(L+1)}}}$
ReLU	$\sigma = \sqrt{2}\sqrt{\frac{6}{n^{(L)} + n^{(L+1)}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n^{(L)} + n^{(L+1)}}}$

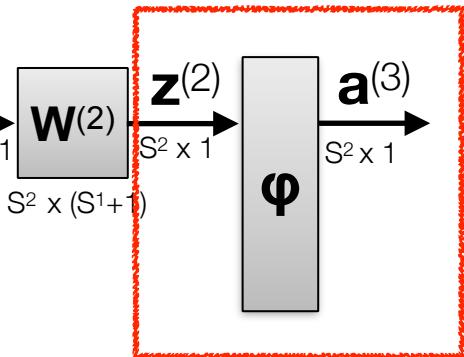
Summarized by Glorot and He

Dropout
Smarter Weight Initialization



Practical Implementation of Architectures

- A new nonlinearity: **softmax**



$$a_j^{(L+1)} = \frac{\exp(z_j^{(L)})}{\sum_i \exp(z_i^{(L)})}$$

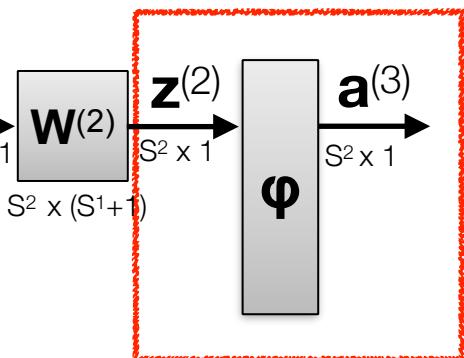
instead of final layer sigmoid!

it has many of the same properties as sigmoid mapping
but also the advantage of **interpretation as a true probability**

Update equations are **identical** to that of Cross Entropy with a sigmoid.
We typically do not try to take derivative of softmax, therefore it is
only used in final layer.

Practical Implementation of Architectures

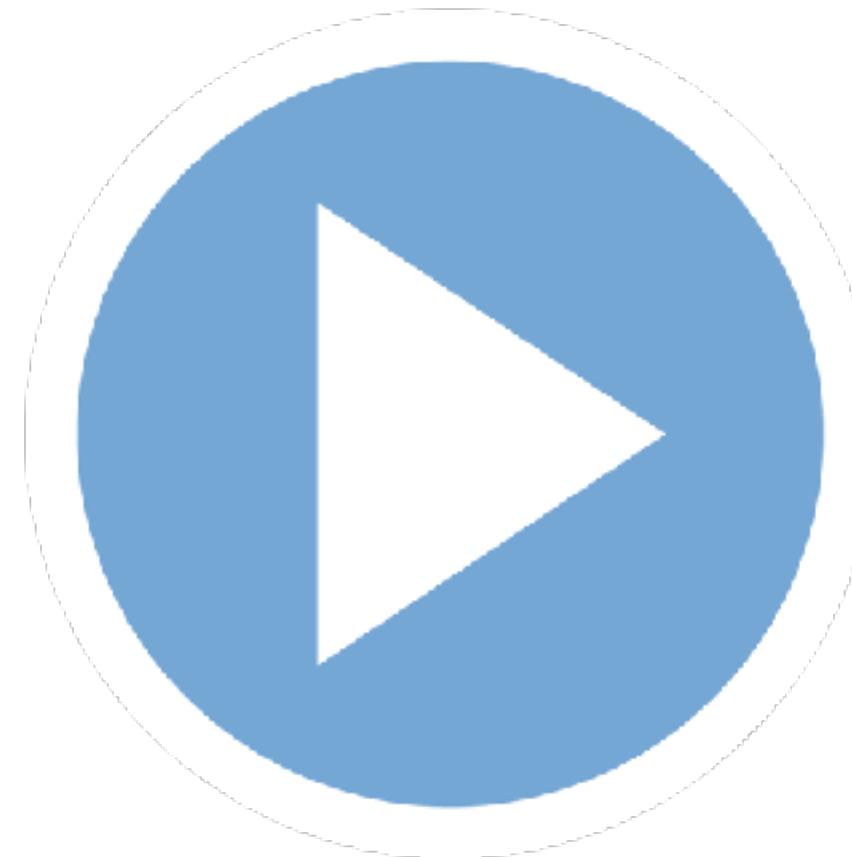
- A new nonlinearity: **rectified linear units**



$$\phi(\mathbf{z}^{(i)}) = \begin{cases} \mathbf{z}^{(i)}, & \text{if } \mathbf{z}^{(i)} > 0 \\ \mathbf{0}, & \text{else} \end{cases}$$

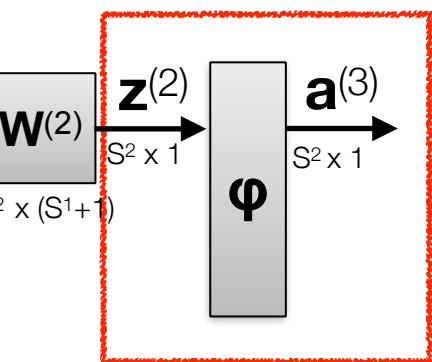
it has the advantage of **large gradients** and
extremely simple derivative

$$\frac{\partial \phi(\mathbf{z}^{(i)})}{\partial \mathbf{z}^{(i)}} = \begin{cases} 1, & \text{if } \mathbf{z}^{(i)} > 0 \\ 0, & \text{else} \end{cases}$$



ReLU Nonlinearities

Other Activation Functions



- Sigmoid Weighted Linear Unit
SiLU
 - also called Swish
- Mixing of sigmoid, σ , and ReLU

$$\varphi(z) = z \cdot \sigma(z)$$

$$\frac{\partial \varphi(z)}{\partial z} = \varphi(z) + \sigma(z)[1 - \varphi(z)]$$

$$= a^{(l+1)} + \sigma(z^{(l)}) \cdot [1 - a^{(l+1)}]$$

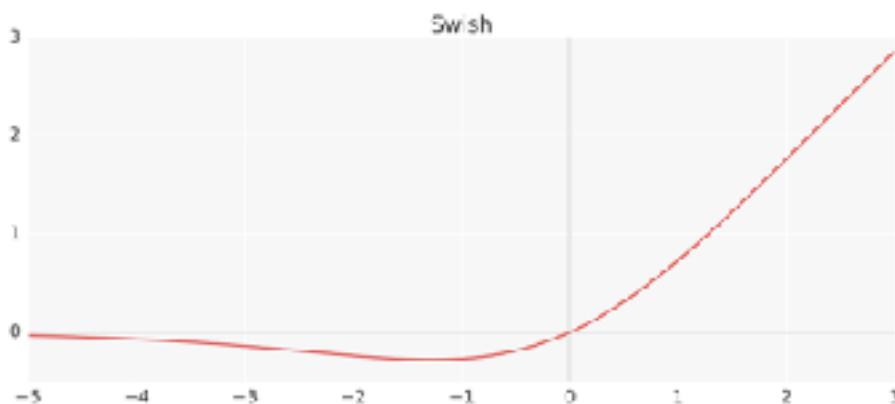


Figure 1: The Swish activation function.

Ramachandran P, Zoph B, Le QV.
Swish: a Self-Gated Activation Function. arXiv preprint
arXiv:1710.05941. 2017 Oct 16

Elfwing, Stefan, Eiji Uchibe, and Kenji Doya. "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning." Neural Networks (2018).

Derivative Calculation:

$$= \sigma(x) + x \cdot \sigma(x)(1 - \sigma(x))$$

$$= \sigma(x) + x \cdot \sigma(x) - x \cdot \sigma(x)^2$$

$$= x \cdot \sigma(x) + \sigma(x)(1 - x \cdot \sigma(x))$$

Practical Details

- Neural networks can separate any data through multiple layers. The true realization of Rosenblatt:

"Given an elementary α -perceptron, a stimulus world W , and any classification $C(W)$ for which a solution exists; let all stimuli in W occur in any sequence, provided that each stimulus must reoccur in finite time; then beginning from an arbitrary initial state, an error correction procedure will always yield a solution to $C(W)$ in finite time..."
- **Universality:** No matter what function we want to compute, we know that there is a neural network which can do the job.
- One nonlinear hidden layer with an output layer can perfectly train any problem with enough data, but might just be memorizing...
 - ... it might be better to have even more layers for decreased computation and generalizability

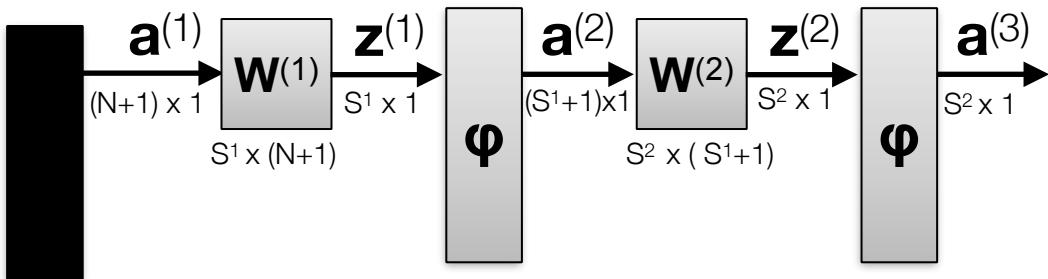
End of Session

- Next Time: Town Hall
- Next Time: Deep Learning in Keras

Back Up Slides

Back propagation

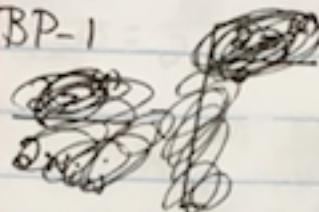
$$J(\mathbf{W}) = \sum_k^M (\mathbf{y}^{(k)} - \mathbf{a}^{(L)})^2$$



use chain rule:
$$\frac{\partial J(\mathbf{W})}{\partial w_{i,j}^{(l)}} = \frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}} \frac{\partial \mathbf{z}^{(l)}}{\partial w_{i,j}^{(l)}}$$

$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta \frac{\partial J(\mathbf{W})}{\partial w_{i,j}^{(l)}}$$

BP-1



$$\frac{\partial z^{(k)}}{\partial w_{i,j}^{(k)}} = \begin{bmatrix} \frac{\partial z_1^{(k)}}{\partial w_{i,j}^{(k)}} \\ \vdots \\ \frac{\partial z_s^{(k)}}{\partial w_{i,j}^{(k)}} \end{bmatrix}$$

$$z^{(k)} = W^{(k)} a^{(k)}$$

$$\begin{aligned} z_k^{(k)} &= W^{(k)} a^{(k)} \\ &\text{ROW } k^{\text{TH}} \\ &= \sum_{j=1}^s w_{k,j}^{(k)} a_j^{(k)} \end{aligned}$$

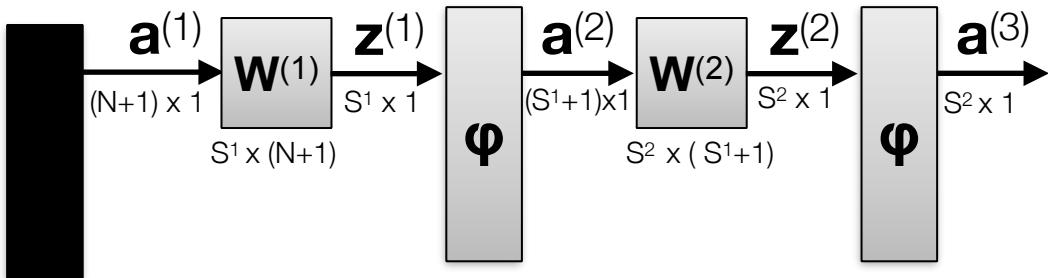
$$\begin{aligned} \frac{\partial z_i^{(k)}}{\partial w_{i,j}^{(k)}} &= \sum_{j=1}^s \frac{\partial z_k^{(k)}}{\partial w_{i,j}^{(k)}} a_j^{(k)} w_{k,j}^{(k)} \\ &= 0 \text{ if } k \neq i \end{aligned}$$

$$= a_j^{(k)} \text{ if } k = i$$

$$\boxed{\frac{\partial z^{(k)}}{\partial w_{i,j}^{(k)}} = a_j^{(k)}} = \begin{bmatrix} 0 \\ 0 \\ a_j^{(k)} \\ 0 \\ 0 \end{bmatrix} \leftarrow i^{\text{TH POSITION}}$$

Back propagation

$$J(\mathbf{W}) = \sum_k^M (\mathbf{y}^{(k)} - \mathbf{a}^{(L)})^2$$



can we use
chain rule
again?

$$\frac{\partial J(\mathbf{W})}{\partial w_{i,j}^{(l)}} = \boxed{\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}}} a_j^{(l)}$$

$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta \boxed{\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}}} a_j^{(l)}$$

- Steps:
 - propagate weights forward
 - calculate gradient at final layer
 - back propagate gradient for each layer
 - via recurrence relation

BP-2, setup $\frac{\partial J(w)}{\partial z^{(l+1)}}$

CHAIN RULE
AGAIN

$$\frac{\partial J(w)}{\partial z^{(l)}} = \underbrace{\frac{\partial J(w)}{\partial z^{(l+1)}}}_{\text{VECTOR}} \underbrace{\frac{\partial z^{(l+1)}}{\partial z^{(l)}}}_{\text{MATRIX}}$$

$1 \times S^{(l+1)}$

$S^{(l+1)} \times S^{(l)}$

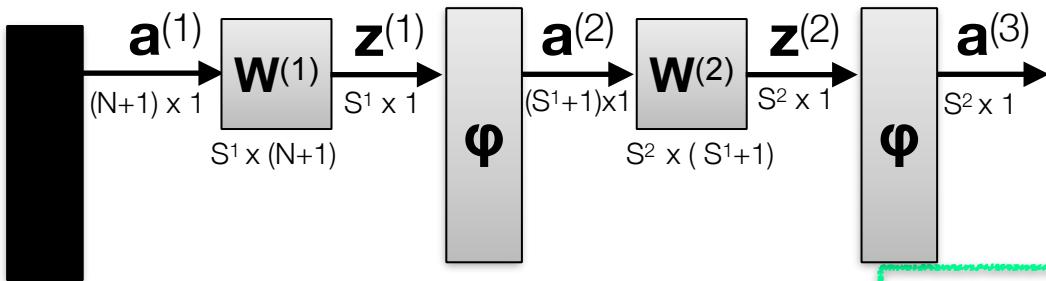
$$\begin{bmatrix} \frac{\partial J(w)}{\partial z_1^{(l+1)}} \\ \vdots \\ \frac{\partial J(w)}{\partial z_{S^{(l+1)}}^{(l+1)}} \end{bmatrix} \begin{bmatrix} \frac{\partial z_1^{(l+1)}}{\partial z_1^{(l)}} & \frac{\partial z_2^{(l+1)}}{\partial z_1^{(l)}} & \dots & \frac{\partial z_{S^{(l+1)}}^{(l+1)}}{\partial z_1^{(l)}} \\ \frac{\partial z_1^{(l+1)}}{\partial z_2^{(l)}} & \ddots & & \frac{\partial z_{S^{(l+1)}}^{(l+1)}}{\partial z_2^{(l)}} \\ \vdots & \ddots & \ddots & \vdots \\ \frac{\partial z_1^{(l+1)}}{\partial z_{S^l}^{(l)}} & \dots & \dots & \frac{\partial z_{S^{(l+1)}}^{(l+1)}}{\partial z_{S^l}^{(l)}} \end{bmatrix}$$

$$\frac{\partial z_i^{(l+1)}}{\partial z_j^{(l)}} = \frac{\partial}{\partial z_j^{(l)}} \sum_{k=1}^{S^{(l+1)}} w_{i,j,k}^{(l+1)} q_k^{(l+1)}$$

#

Back propagation

$$J(\mathbf{W}) = \sum_k^M (\mathbf{y}^{(k)} - \mathbf{a}^{(L)})^2$$



can we get this gradient?

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}} = \frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l+1)}} \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{z}^{(l)}}$$

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}} a_j^{(l)}$$

BP-3

$$\frac{\partial z_i^{(t+1)}}{\partial z_j^{(t)}} = \frac{\partial}{\partial z_j^{(t)}} \left(\sum_{k=1}^{S^{(t+1)}} w_{i,k}^{(t+1)} q_k^{(t+1)} \right)$$

$$= w_{i,j}^{(t+1)} \frac{\partial}{\partial z_j^{(t)}} \phi(z_j^{(t)})$$

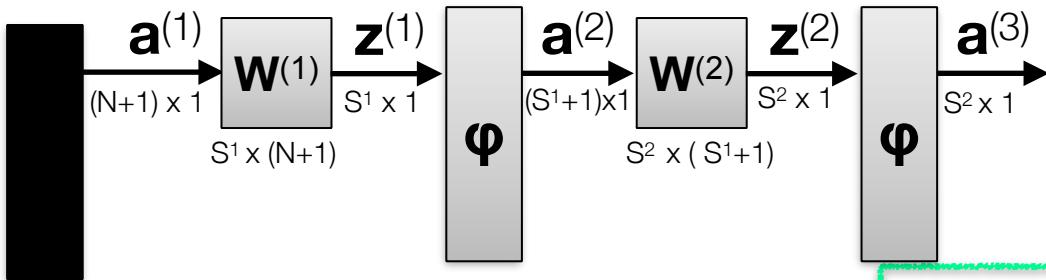
$$= w_{i,j}^{(t+1)} q_j^{(t+1)} (1 - q_j^{(t+1)})$$

$$\frac{\partial z^{(t+1)}}{\partial z^{(t)}} = w^{(t+1)} \underbrace{\text{diag}(q^{(t+1)}(1 - q^{(t+1)}))}_{S^{(t+1)} \times S^{(t+1)}} \underbrace{w^{(t+1)}}_{S^{(t+1)} \times S^2}$$

ENTIRE MATRIX
 $S^{(t+1)} \times S^2$

Back propagation

$$J(\mathbf{W}) = \sum_k^M (\mathbf{y}^{(k)} - \mathbf{a}^{(L)})^2$$



use chain rule:

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}} = \frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l+1)}} \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{z}^{(l)}}$$

$$\frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{z}^{(l)}} = \text{diag}[\mathbf{a}^{(l+1)} * (1 - \mathbf{a}^{(l+1)})] \cdot \mathbf{W}^{(l+1)}$$

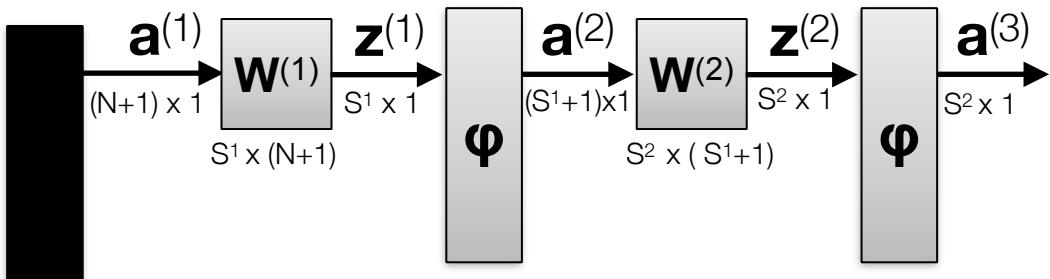
$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}} = \text{diag}[\mathbf{a}^{(l+1)} * (1 - \mathbf{a}^{(l+1)})] \cdot \mathbf{W}^{(l+1)} \frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l+1)}}$$

recurrence relation

If we know last layer, we can **back propagate** towards previous layers!

Back propagation

$$J(\mathbf{W}) = \sum_k^M (\mathbf{y}^{(k)} - \mathbf{a}^{(L)})^2$$



$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}} a_j^{(l)}$$

one more step

need last layer
gradient:

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(2)}} = \frac{\partial}{\partial \mathbf{z}^{(2)}} (\mathbf{y}^{(k)} - \phi(\mathbf{z}^{(2)}))^2$$

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l+1)}}$$

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(2)}} = -2(\mathbf{y}^{(k)} - \mathbf{a}^{(3)}) * \mathbf{a}^{(3)} * (1 - \mathbf{a}^{(3)})$$

Final Layer BP-4

$$\frac{\partial J(w)}{\partial z_i^{(L)}} = \frac{\partial}{\partial z_i^{(L)}} \sum_{k=1}^m (y^{(k)} - \phi(z^{(L)}))^2$$

$$= - \sum_k 2(y^{(k)} - \phi^{(L+1)}) \frac{\partial}{\partial z_i^{(L)}} \phi(z^{(L)})$$

$$= -2 \sum_k (y^{(k)} - q^{(L+1)}) q_i^{(L+1)} (1 - q_i^{(L+1)}) \frac{\partial}{\partial z_i^{(L)}} z^{(L)}$$

$$= -2 \sum_k (y^{(k)} - q^{(L+1)}) q_i^{(L+1)} (1 - q_i^{(L+1)})$$

$$\frac{\partial J}{\partial z^{(L)}} = -2 \sum_k (y^{(k)} - q^{(L+1)}) q^{(L+1)} * (1 - q^{(L+1)})$$

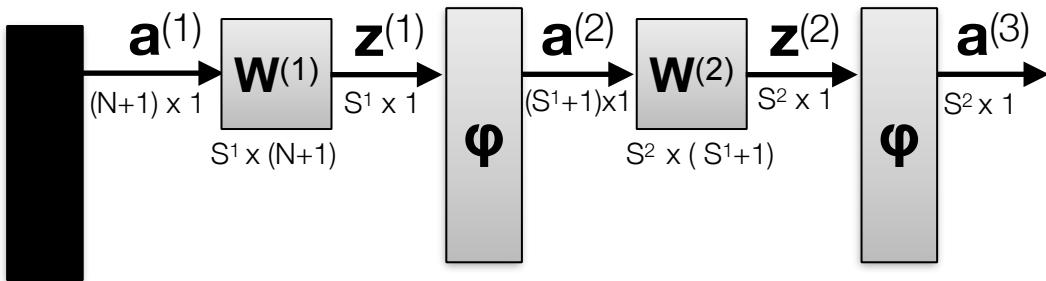
↑ ONE DIM, TAKE OUT K

OR w/ layers

$$\frac{\partial J(w)}{\partial z^{(2)}} = \frac{\partial}{\partial z^{(2)}} \sum_k (y^{(k)} - \phi(z^{(2)}))^2$$

$$= -2 \sum_k (y^{(k)} - q^{(3)}) * q^{(3)} * (1 - q^{(3)})$$

Back propagation summary



- **Self Test:**

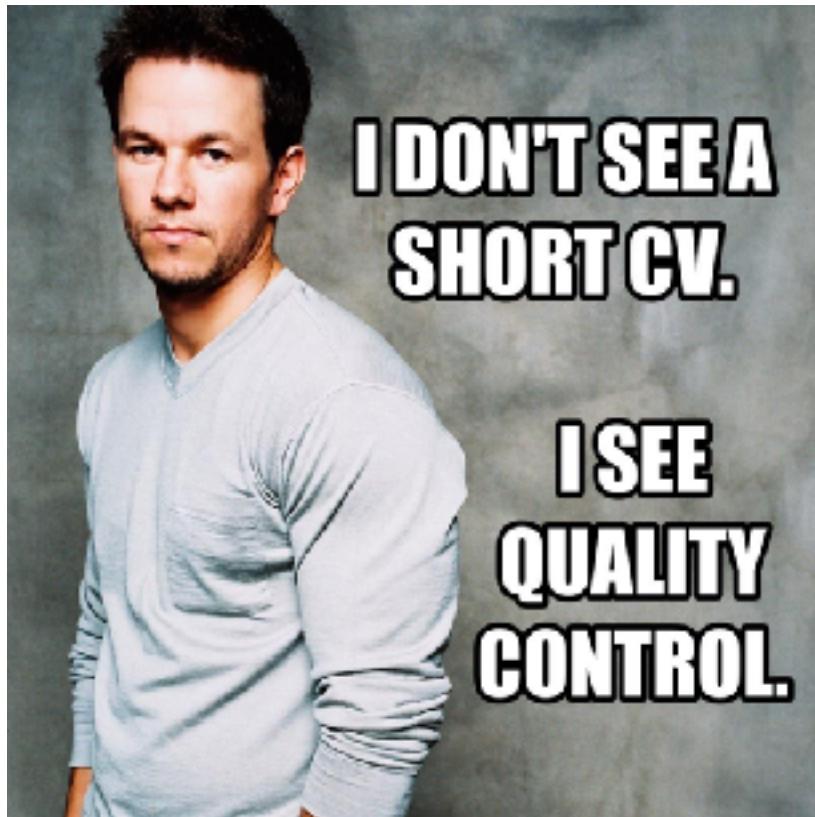
True or False: If we change the cost function, $J(\mathbf{W})$, we only need to update the final layer calculation of the back propagation steps. The remainder of the algorithm is unchanged.

- A. True
- B. False

$$\boxed{\begin{aligned}\mathbf{V}^{(2)} &= -2(\mathbf{Y} - \mathbf{A}^{(3)}) * \mathbf{A}^{(3)} * (1 - \mathbf{A}^{(3)}) \\ \nabla^{(2)} &= \mathbf{V}^{(2)} \cdot [\mathbf{A}^{(2)}]^T\end{aligned}}$$

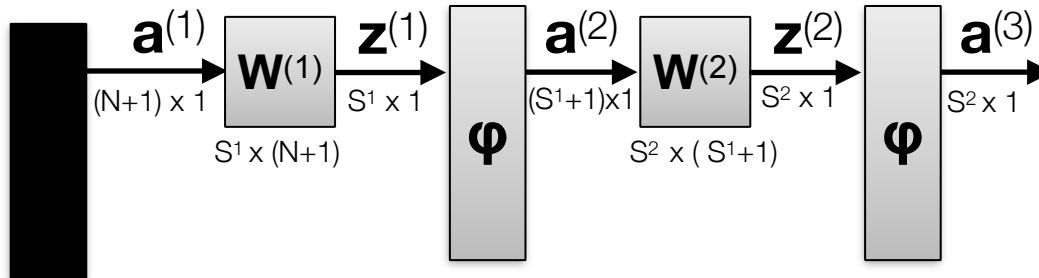
Programming Multi-layer Neural Networks

07. MLP Neural Networks.ipynb



Guided Example

Recall from the in-class assignment



1. Forward propagate to get \mathbf{Z}, \mathbf{A}

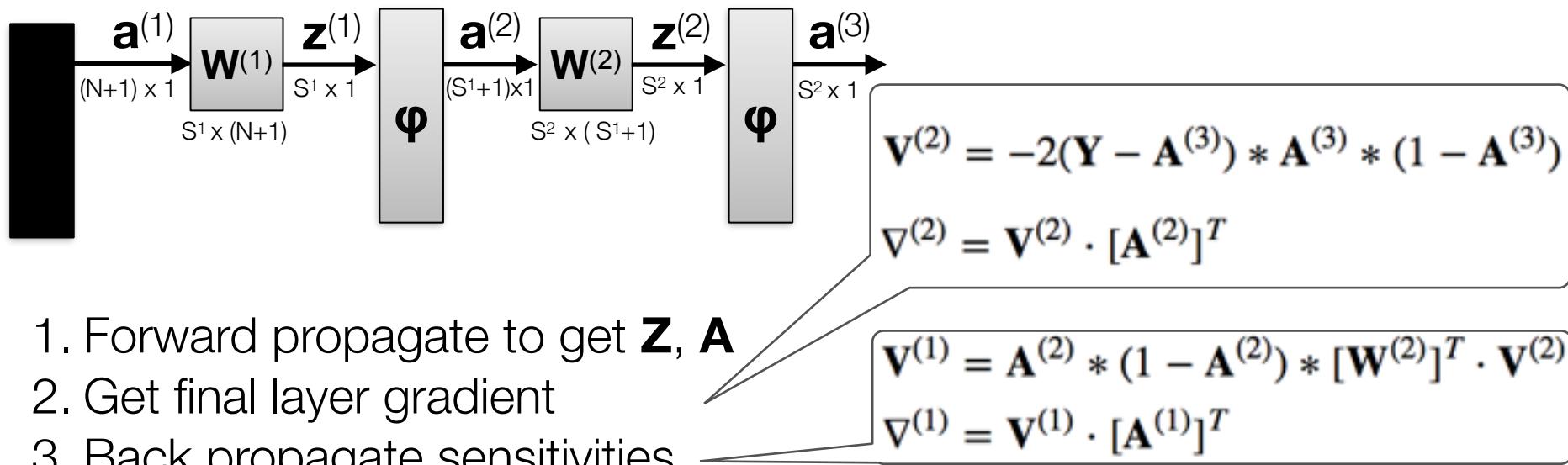
```
# feedforward all instances
A1, z1, A2, z2, A3 = self._feedforward(X_data, self.W1, self.W2)
```



```
def _feedforward(self, X, W1, W2):
    A1 = self._add_bias_unit(X.T, how='row')
    Z1 = W1 @ A1
    A2 = self._sigmoid(Z1)
    A2 = self._add_bias_unit(A2, how='row')
    Z2 = W2 @ A2
    A3 = self._sigmoid(Z2)
    return A1, Z1, A2, Z2, A3
```

these are more than just vectors for **one instance!**
these are for **all** instances

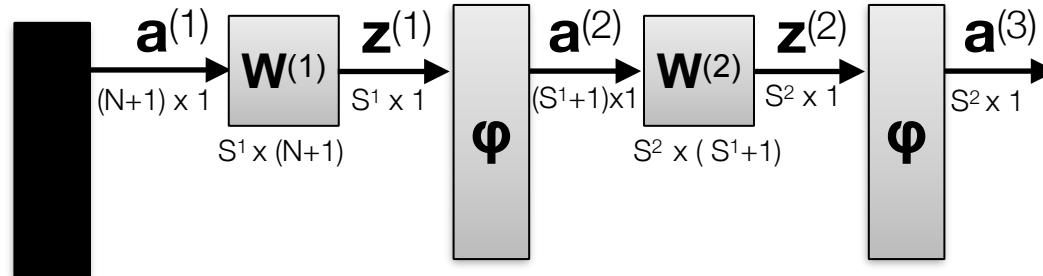
Back propagation implementation



```
def _get_gradient(self, A1, A2, A3, z1, z2, Y_enc, W1, W2):
    """ Compute gradient step using backpropagation.
    """
    # vectorized backpropagation
    V2 = -2*(Y_enc-A3)*A3*(1-A3)
    V1 = A2*(1-A2)*(W2.T @ V2)

    grad2 = V2 @ A2.T
    grad1 = V1[1:,:,:] @ A1.T
```

Back propagation implementation



1. Forward propagate to get \mathbf{Z} , \mathbf{A} for all layers
2. Get final layer gradient
3. Back propagate sensitivities
4. Update each $\mathbf{W}^{(l)}$

for each layer:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \nabla^{(l)}$$

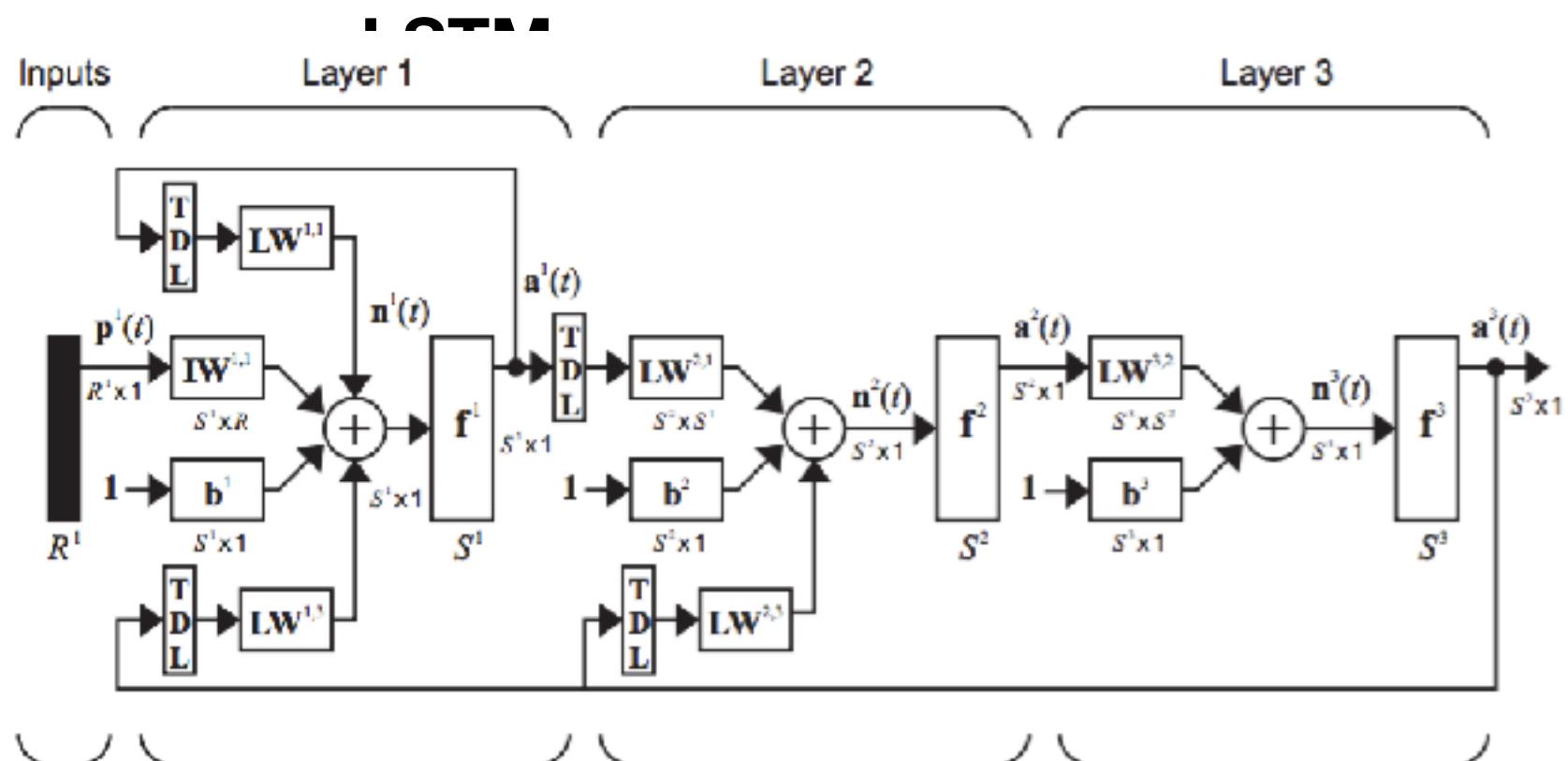
```
# feedforward all instances
A1, Z1, A2, Z2, A3 = self._feedforward(X_data, self.W1, self.W2)

# compute gradient via backpropagation
grad1, grad2 = self._get_gradient(A1=A1, A2=A2,
                                    A3=A3, Z1=Z1,
                                    Y_enc=Y_enc,
                                    W1=self.W1, W2=self.W2)

self.W1 -= self.eta * grad1
self.W2 -= self.eta * grad2
```

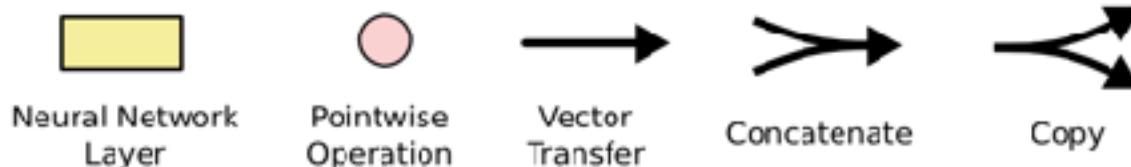
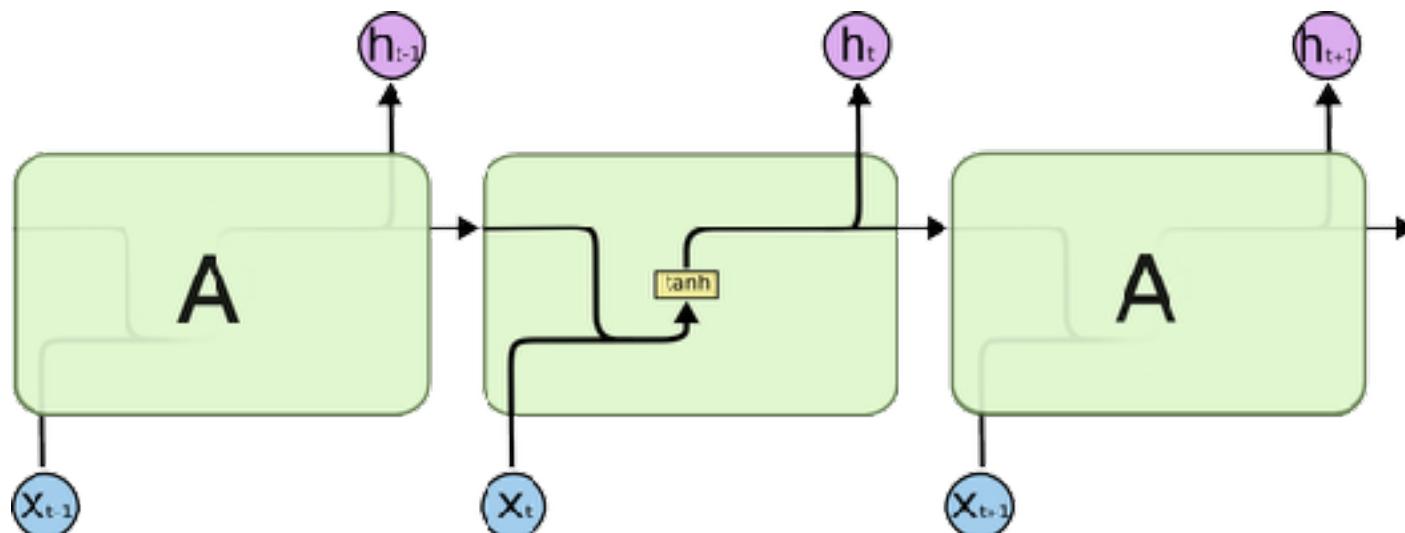
More Advanced Architectures

- Dynamic Networks (recurrent networks)
 - can use current and previous inputs, in time
 - still popular, but ultimately extremely hard to train
 - **highly successful variant:** long short term



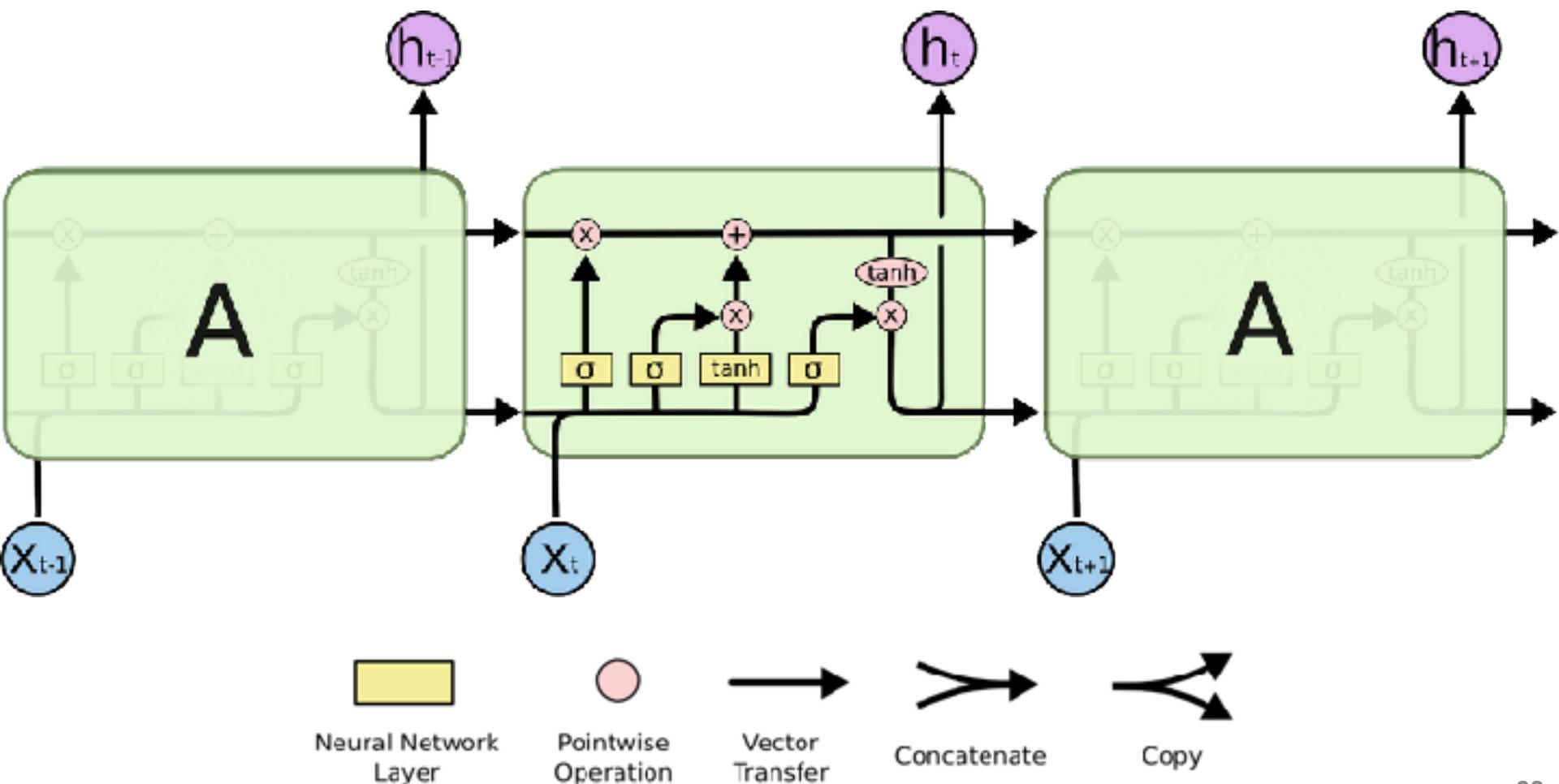
More Advanced Architectures

- **LSTM key idea:** limit how past data can affect output



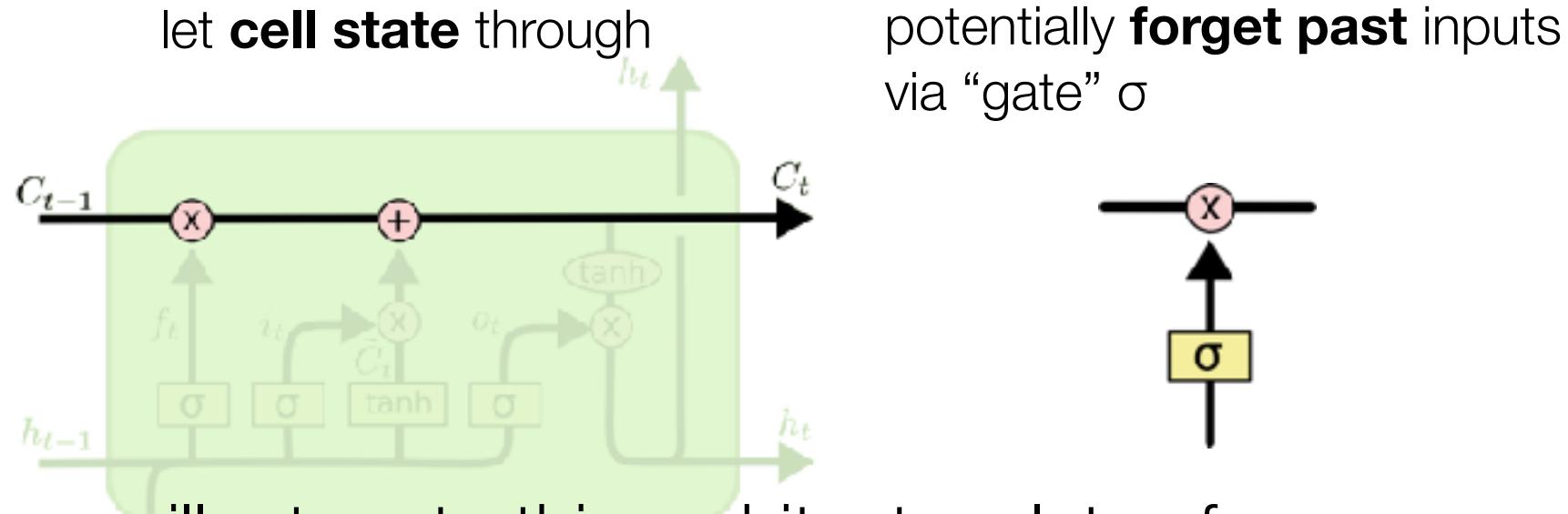
More Advanced Architectures

- **LSTM key idea:** limit how past data can affect output



More Advanced Architectures

- **LSTM key idea:** limit how past data can affect output



we will return to this architecture later, for now:

put it in long term memory 😂



Neural Network
Layer



Pointwise
Operation



Vector
Transfer



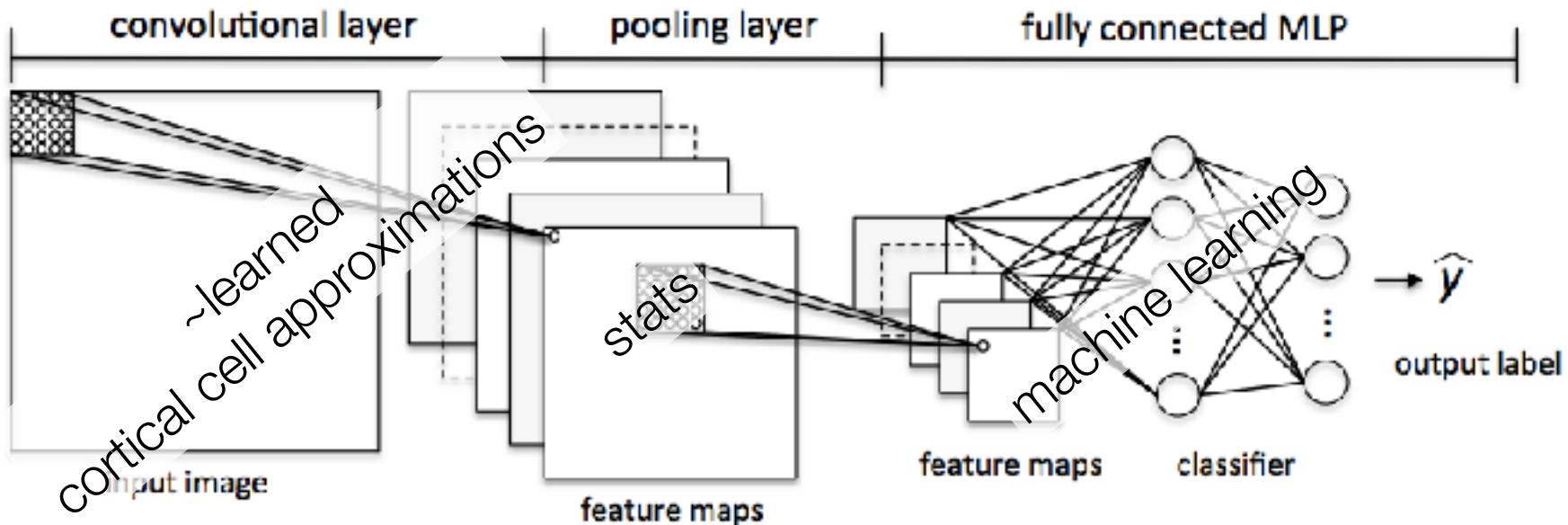
Concatenate



Copy

More Advanced Architectures

- Convolutional Neural Networks
 - image processing operations



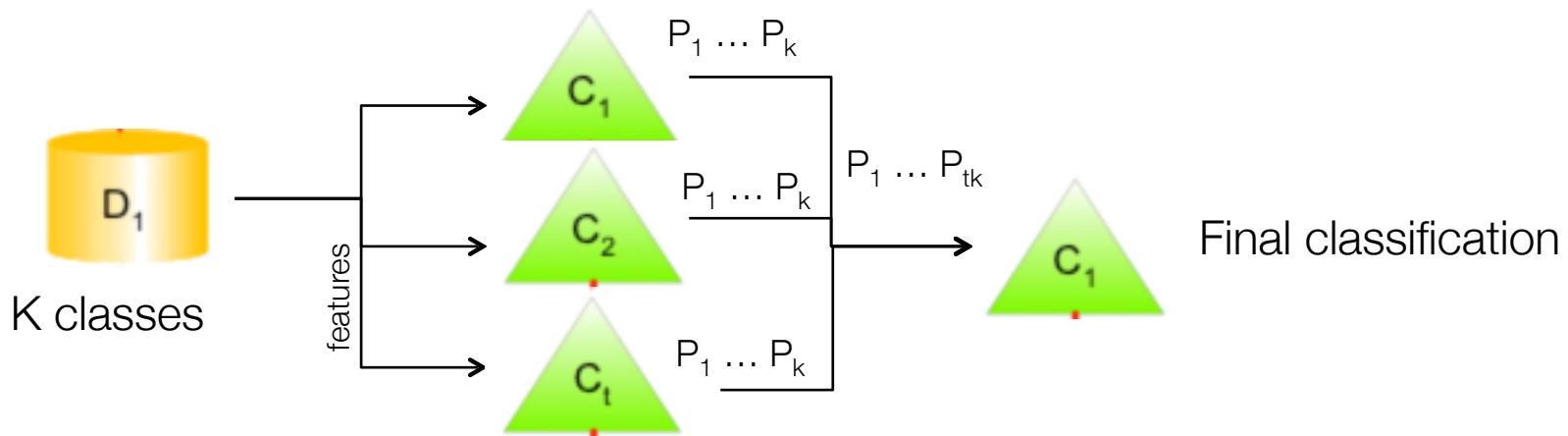
we will return to this architecture later, for now:
put it in long term memory

Problems with these Advanced Architectures

- These architectures have been around for 30 years
- And solved some amazingly hard problems
- but they had **big training problems** that back propagation could not solve readily:
 - unstable gradients (vanishing/exploding)
 - **extremely** non-convex space
 - more layers==many more local optima to get stuck
 - sometimes **gradient optimization is too computational** for weight updates
 - might need better optimization strategy than SGD
- The solution to these problems came from having large amounts of training data, better setup of the optimization
 - eventually was termed **deep learning**

Stacking and Cascading ensembles

- Train an initial classifier (or ensemble)
- Use the output probabilities of each class (from each classifier, if ensemble), as features for another classifier
- Train final classifier on inferences from previous classes



Self test

- How many features (attributes) does scikit-learn use in each iteration of bagging for a random forest?
- Which best completes this statement: In scikit, the number of features considered changes
 - a: once for each tree in the forest
 - b: once for each split in each tree

each iteration!!!

`max_features=sqrt(n_features)`

SVMs Summary: Linear

- Linear SVMs
 - Architecture identical to logistic regression, except:
 - maximize margin
 - constrained optimization
 - **Self Test:** What are the trained parameters for the linear SVM?
 - A: Coefficients of w
 - B: Intercept
 - C: Slack Variables
 - D: Support Vector locations

$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^n \zeta_i$$

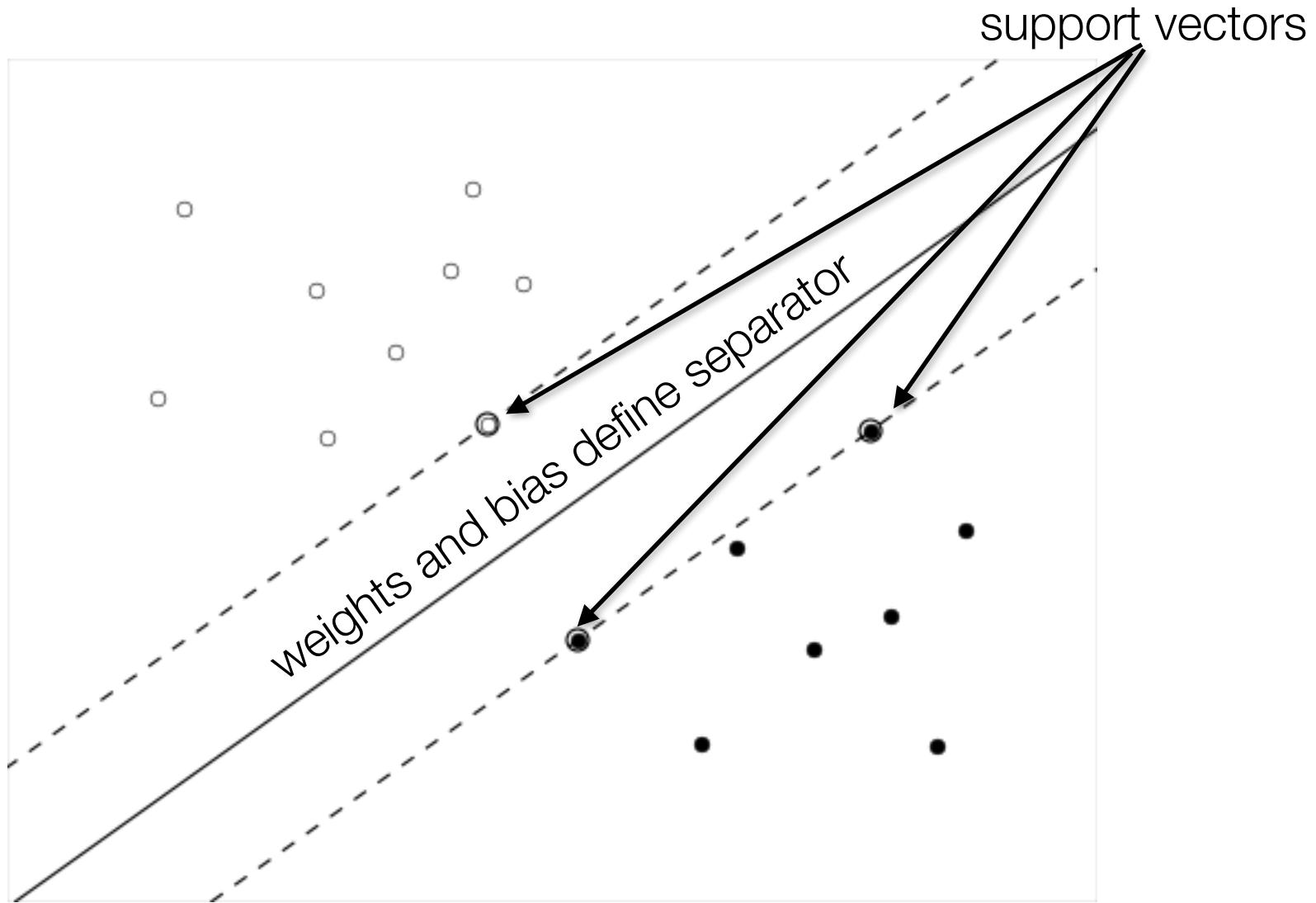
subject to $y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i,$
 $\zeta_i \geq 0, i = 1, \dots, n$

SVMs Summary: Linear

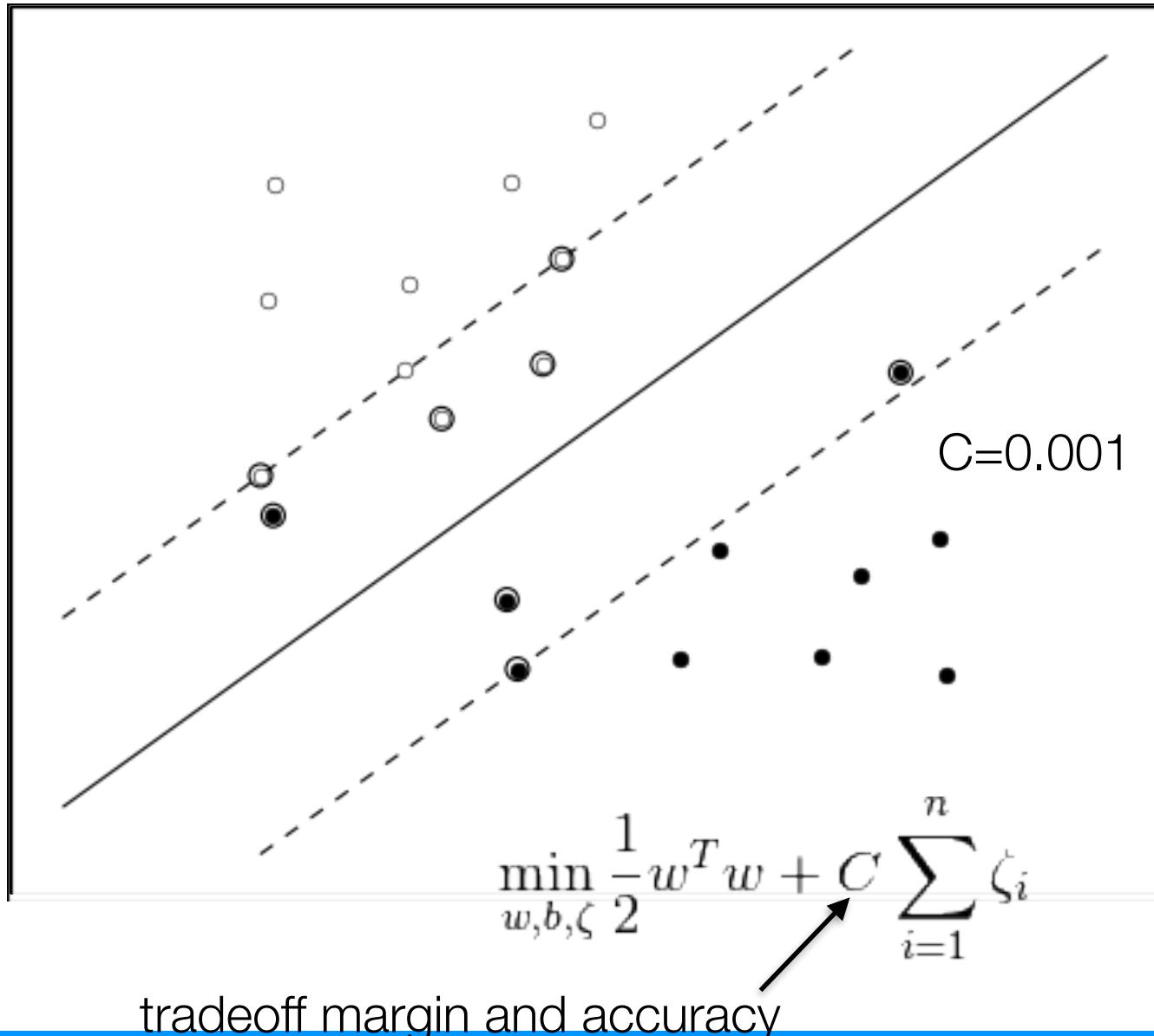
- Linear SVMs
 - Architecture near identical to logistic regression, except:
 - maximize margin
 - constrained optimization
 - Trained Parameters:
 - intercept and weights for each class
 - support vectors chosen for margin calculation
 - slack variables only needed during training

$$\begin{aligned} & \min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^n \zeta_i \\ & \text{to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i, \\ & \zeta_i \geq 0, i = 1, \dots, n \end{aligned}$$

SVMs Summary: Linear

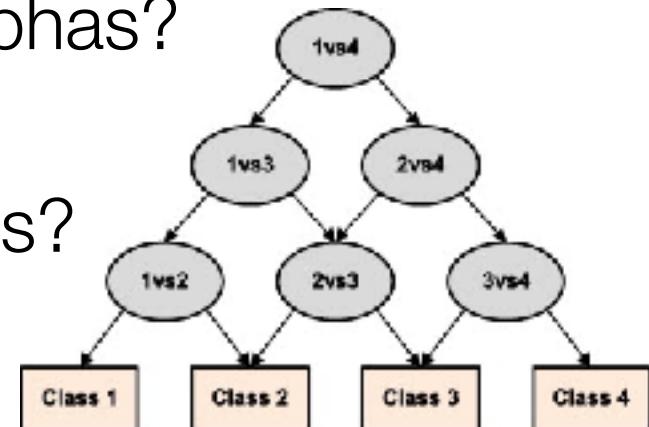


SVMs Summary: Linear



SVMs Summary: non-linear

- Non-linear SVMs
 - Architecture not like logistic regression
 - kernels == high dimensional dot-product
 - impossible to store weights
 - use kernel trick so no need to store them!
 - Trained Parameters
 - biases? how many?
 - selected support vectors? alphas?
 - slack variables?
 - parameters specific to kernels?



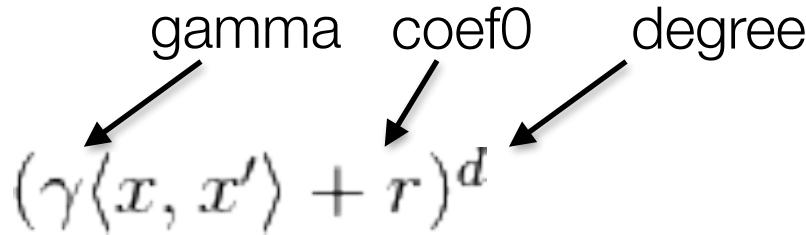
SVMs Summary: non-linear

- Popular Kernels

- polynomial

$$(\gamma \langle x, x' \rangle + r)^d$$

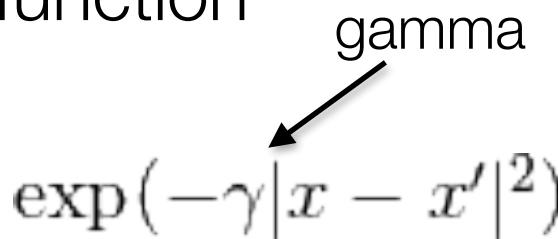
gamma coef0 degree



- radial basis function

$$\exp(-\gamma |x - x'|^2)$$

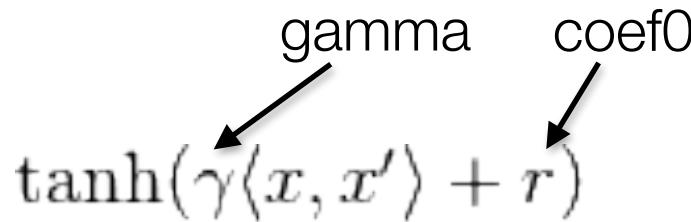
gamma



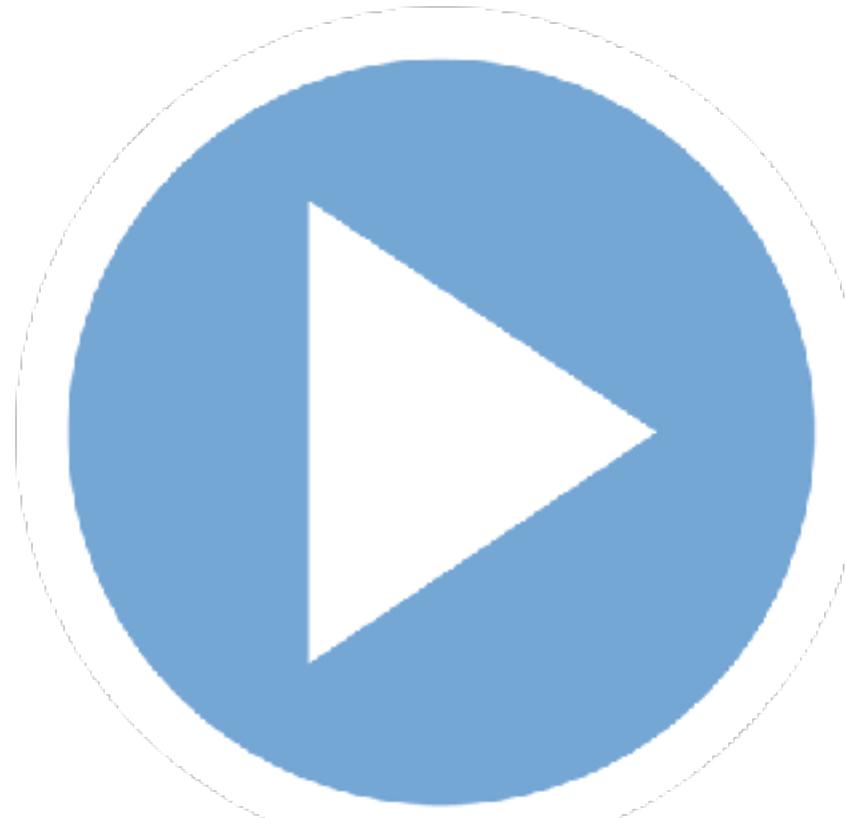
- sigmoid

$$\tanh(\gamma \langle x, x' \rangle + r)$$

gamma coef0



svm_gui.py



The End of SVMs