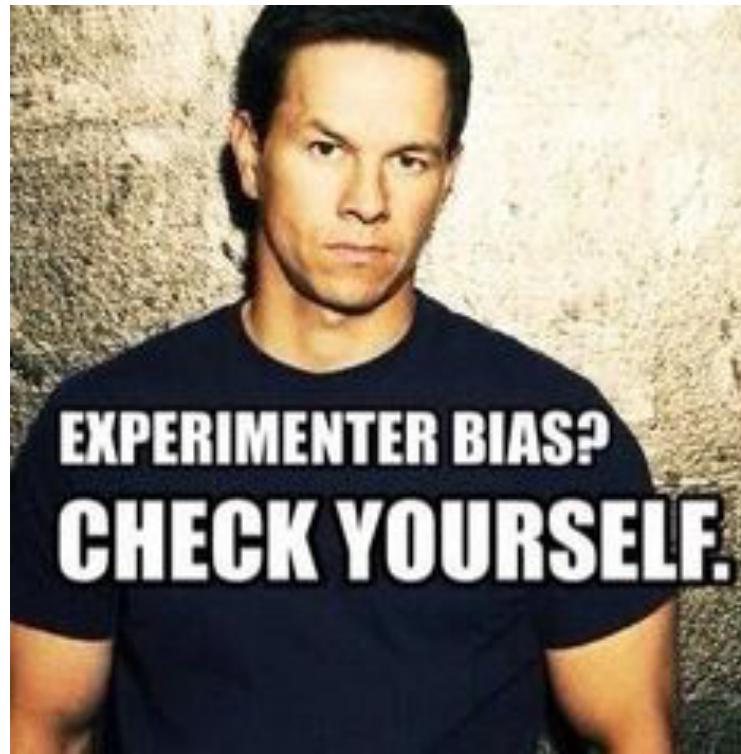

Lecture Notes for Machine Learning in Python

Professor Eric Larson
SVM Review

Class Logistics and Agenda

- Logistics:
 - Lab 3 extended!
 - ICA next time (back propagation)
 - ICA after break (validation)
- Agenda:
 - SVM Review
 - Town Hall

Support Vector Machine Review



SVMs Summary: Linear

- Linear SVMs
 - Architecture near identical to logistic regression, except:
 - maximize margin
 - constrained optimization to $y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i$,
 $\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^n \zeta_i$ subject to $\zeta_i \geq 0, i = 1, \dots, n$
 - **Self Test:** What are the trained parameters for the linear SVM?
 - A: Coefficients of w
 - B: Bias terms
 - C: Slack Variables
 - D: Support Vector locations

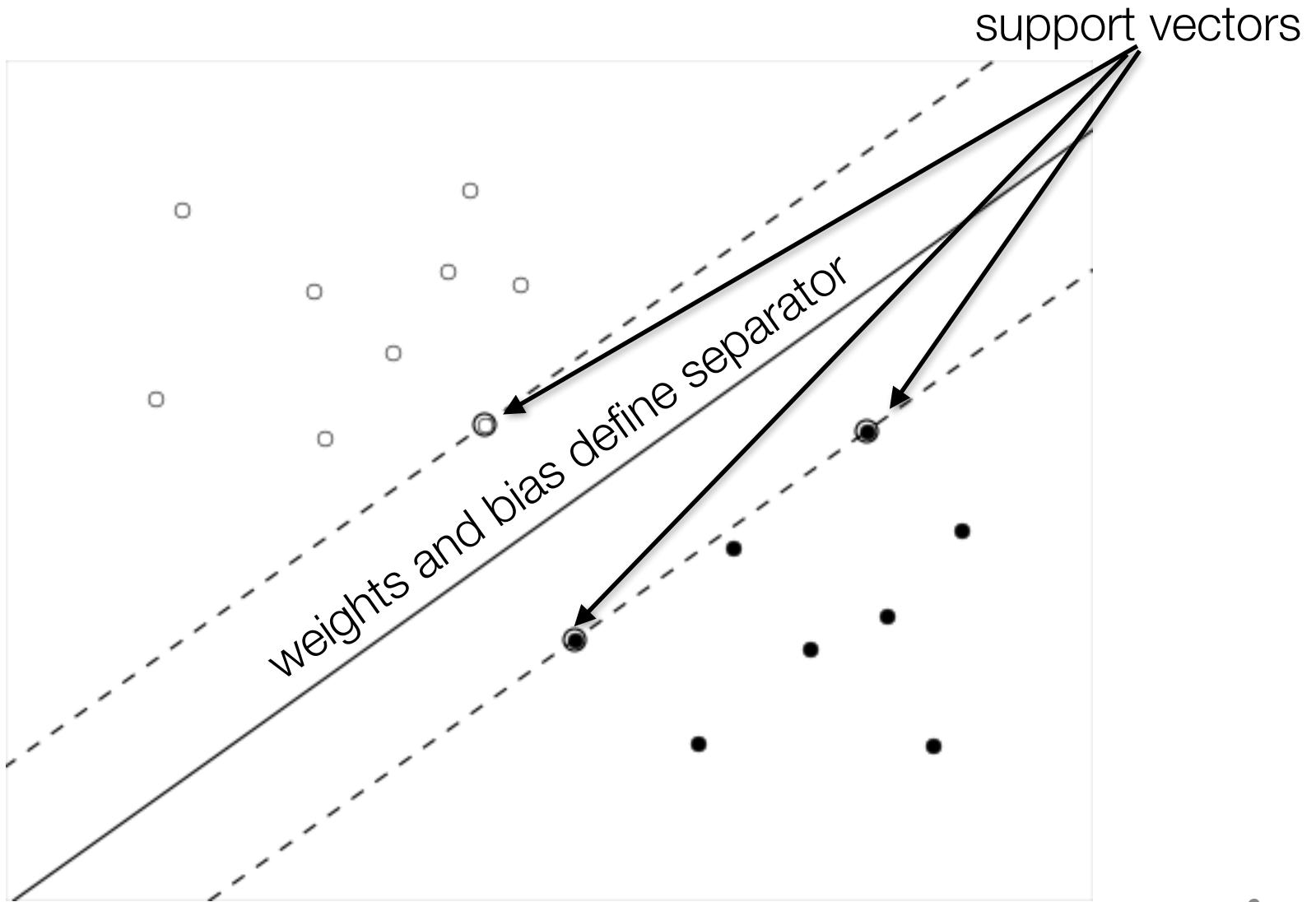
SVMs Summary: Linear

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

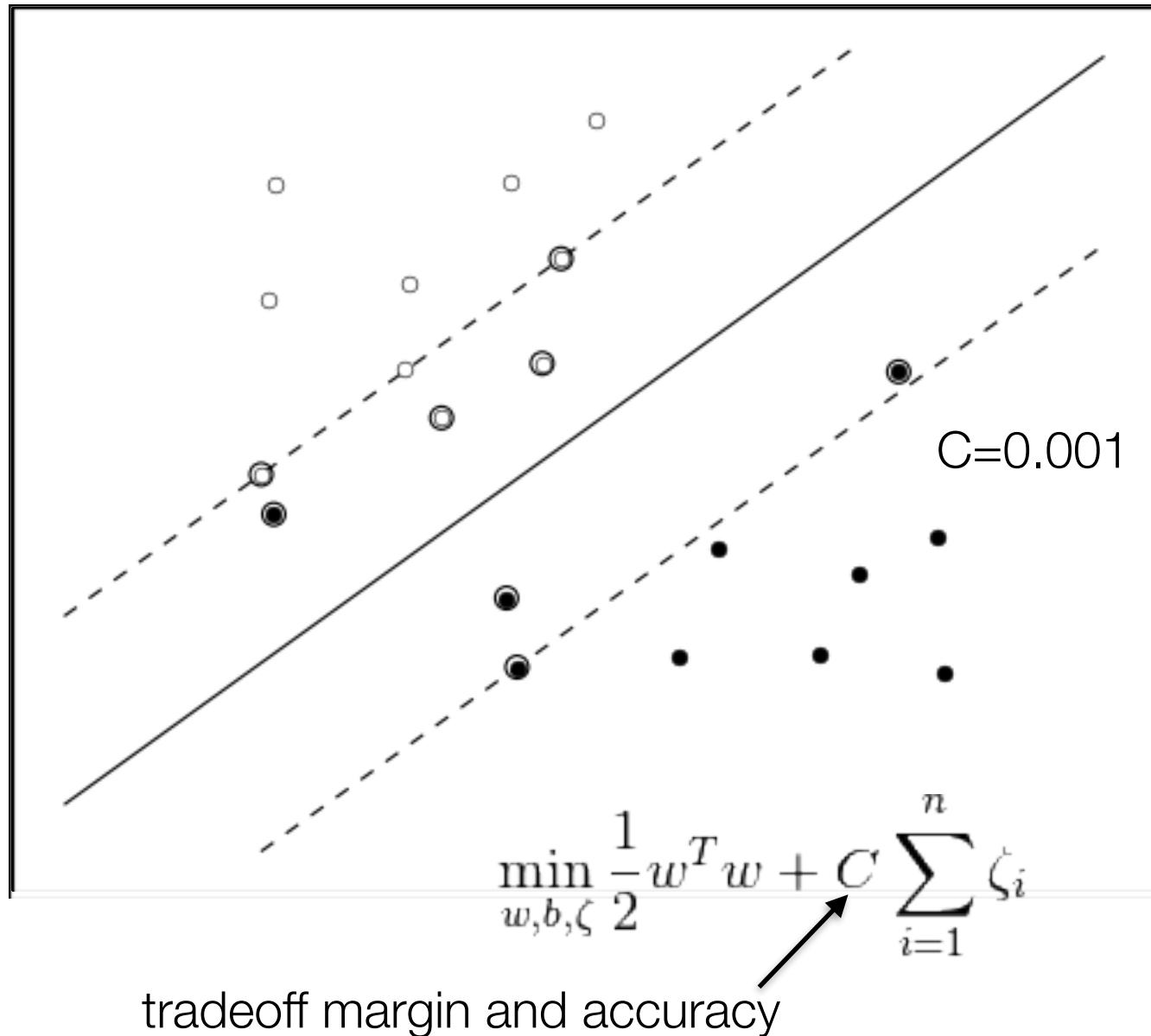
$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^n \zeta_i$$

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

SVMs Summary: Linear

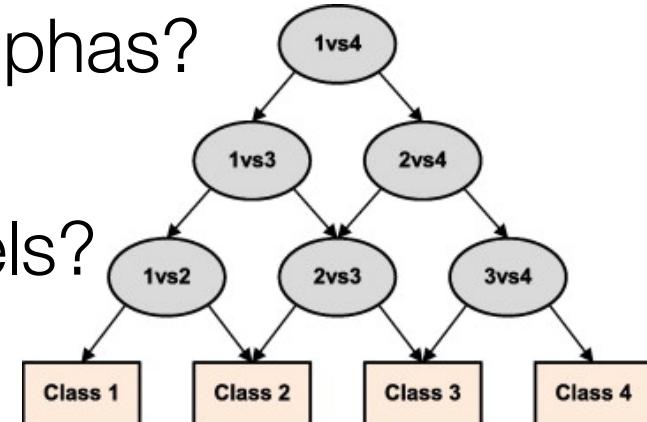


SVMs Summary: Linear



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!
 - Parameters
 - biases? How many?
 - selected support vectors? alphas?
 - slack variables?
 - parameters specific to kernels?



SVMs Summary: non-linear

- Popular Kernels

- polynomial

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

Diagram illustrating the components of a polynomial kernel. Three arrows point from the labels 'gamma', 'coef0', and 'degree' to the respective terms in the equation $(\gamma \langle x, x' \rangle + r)^d$.

- radial basis function

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

Diagram illustrating the component of a radial basis function kernel. An arrow points from the label 'gamma' to the term $-\gamma |x - x'|^2$ in the equation $\exp(-\gamma |x - x'|^2)$.

- sigmoid

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

Diagram illustrating the components of a sigmoid kernel. Two arrows point from the labels 'gamma' and 'coef0' to the respective terms in the equation $\tanh(\gamma \langle x, x' \rangle + r)$.

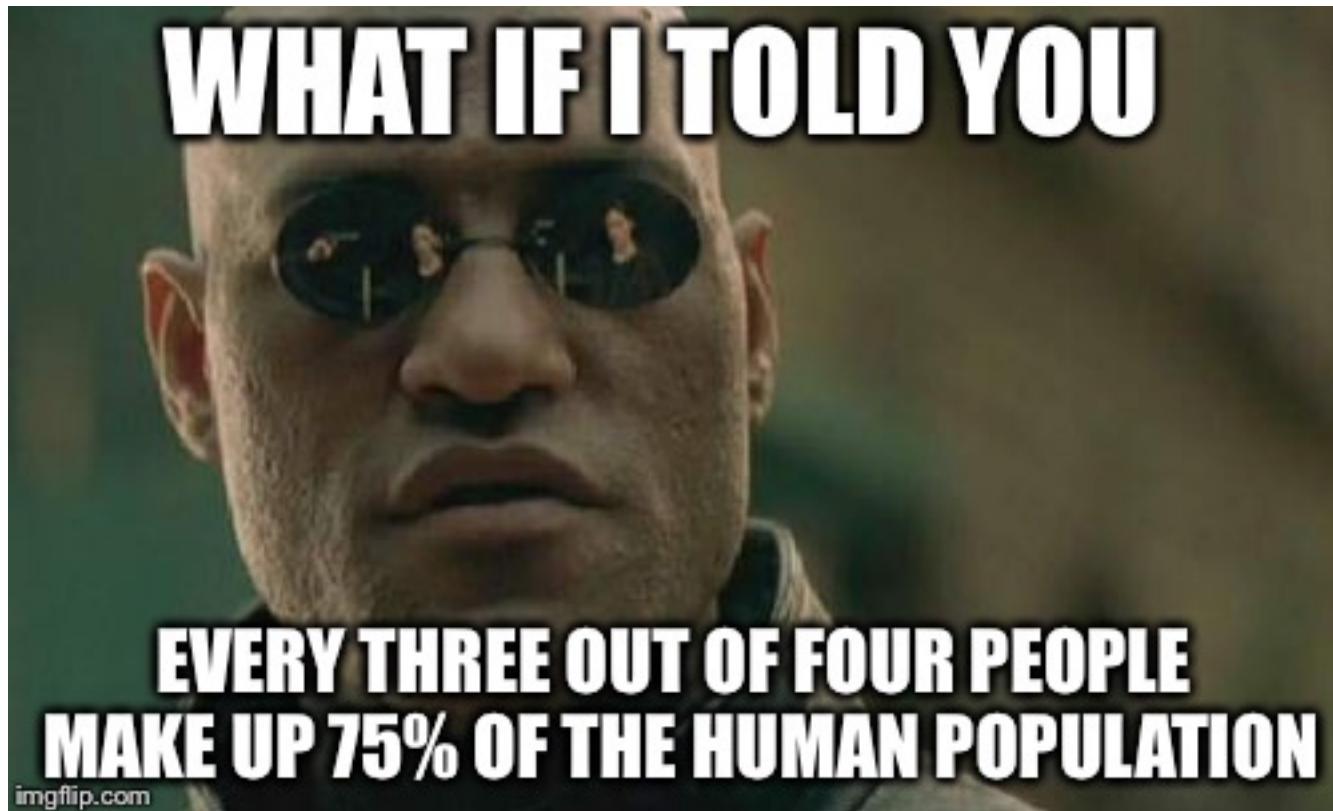
SVM parameterization

svm_gui.py



The End of SVMs

Town Hall



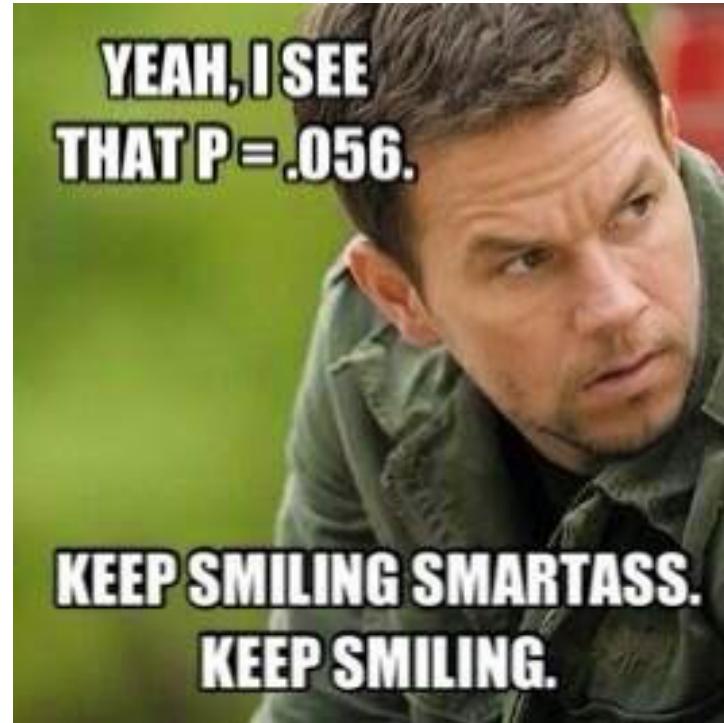
Lecture Notes for Machine Learning in Python

Professor Eric Larson
Neural Network Notation

Class Logistics and Agenda

- Logistics:
 - Lab 3 extended to Wed.
 - Lab 4 due in a while
 - ICA next time (back propagation)
 - ICA following break (validation)
- Multi Week Agenda:
 - Neural Networks History, up to 1980
 - Multi-layer Architectures
 - Programming Multi-layer training

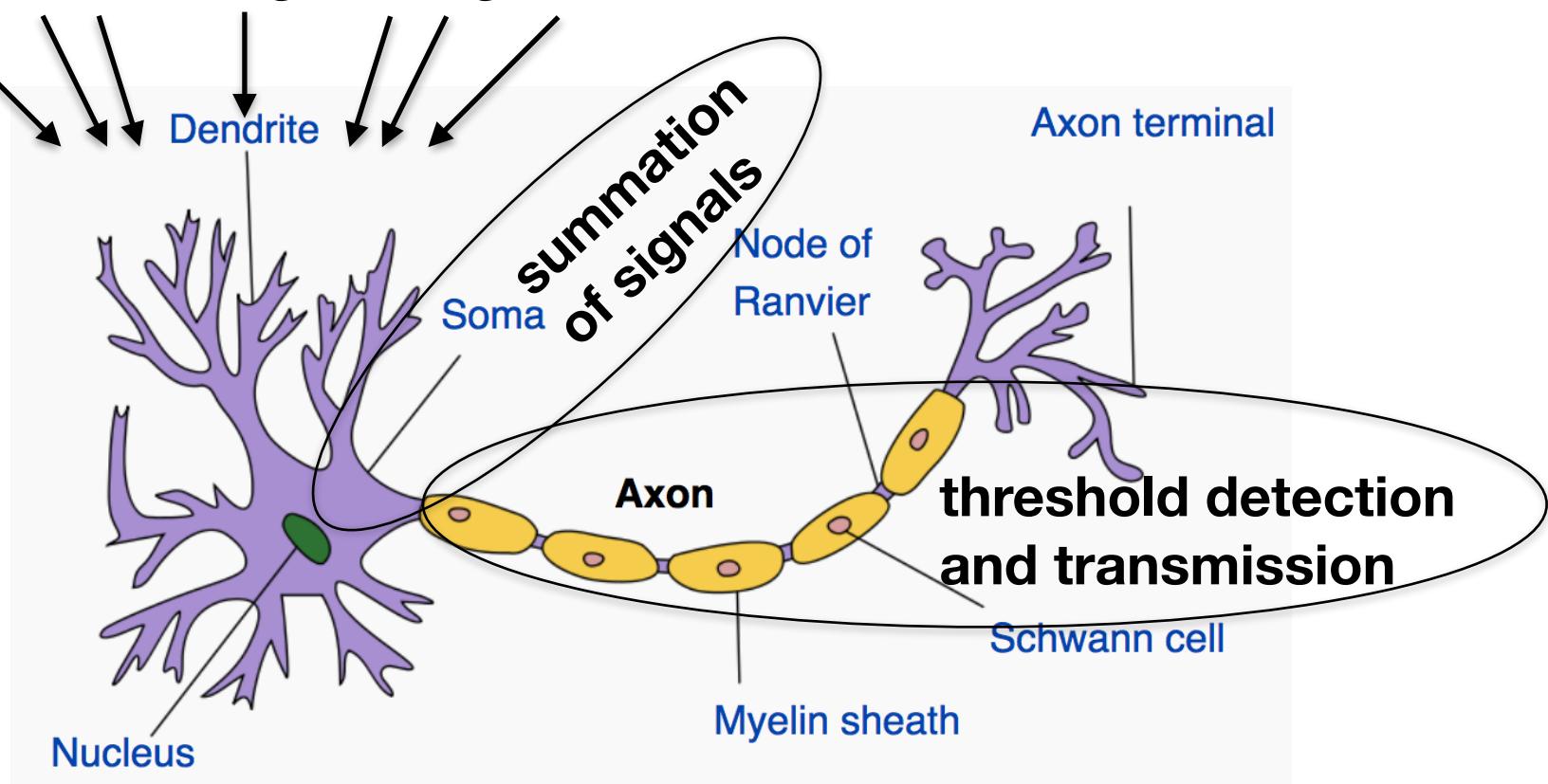
A history of Neural Networks



Neurons

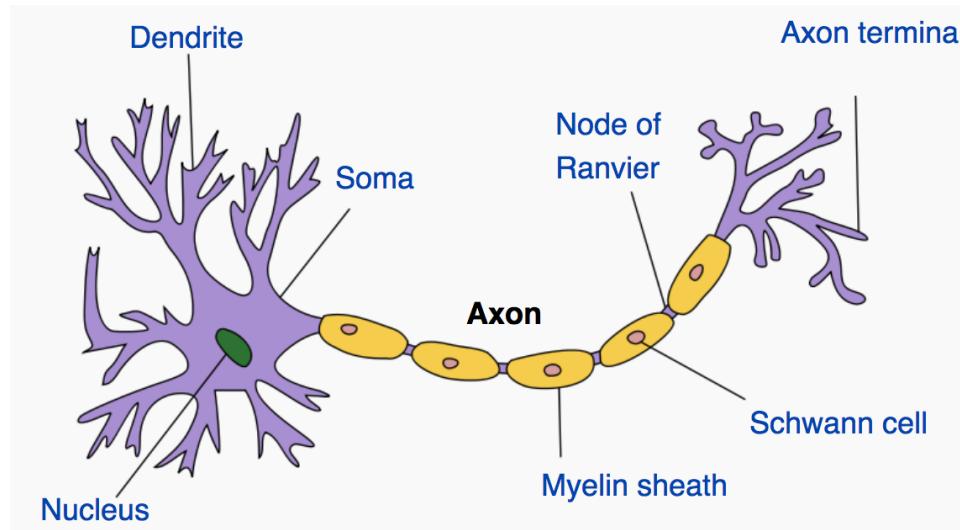
- From biology to modeling:

input from neighboring neurons

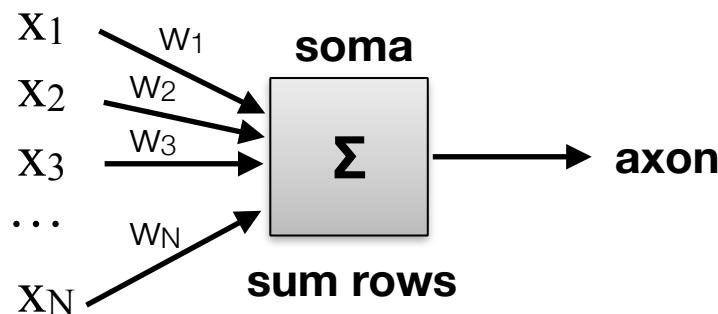


Neurons

- McCulloch and Pitts, 1943



dendrite



input

logic gates of the mind



Warren McCulloch



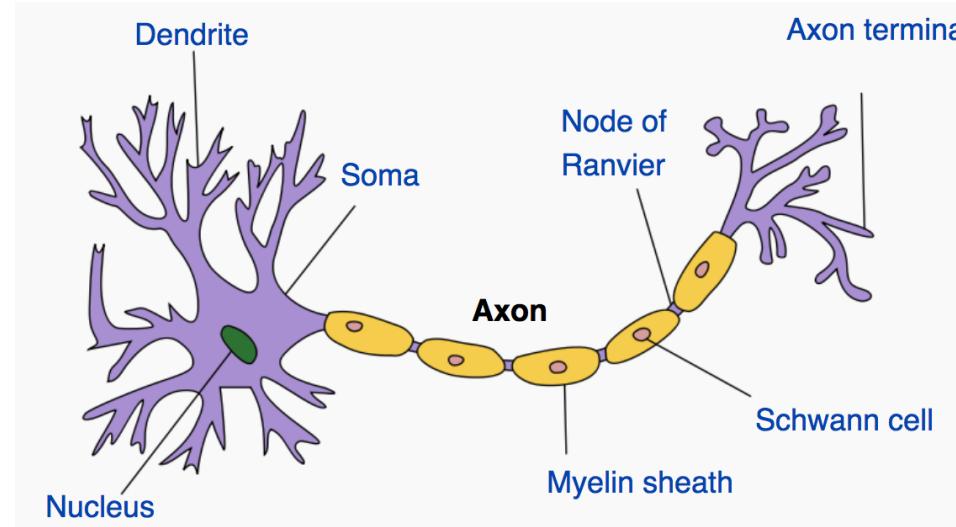
Walter Pitts

Neurons

- McCulloch and Pitts, 1943
- Donald Hebb, 1949
 - Hebb's Law: close neurons fire together
 - makes coupling tighter
 - synaptic transmission
 - basis of neural pathways



Donald O. Hebb



Warren McCulloch

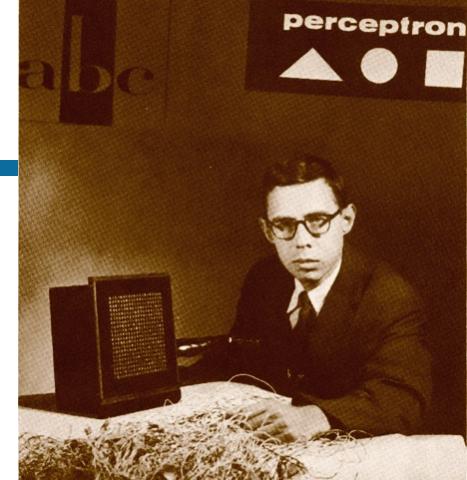
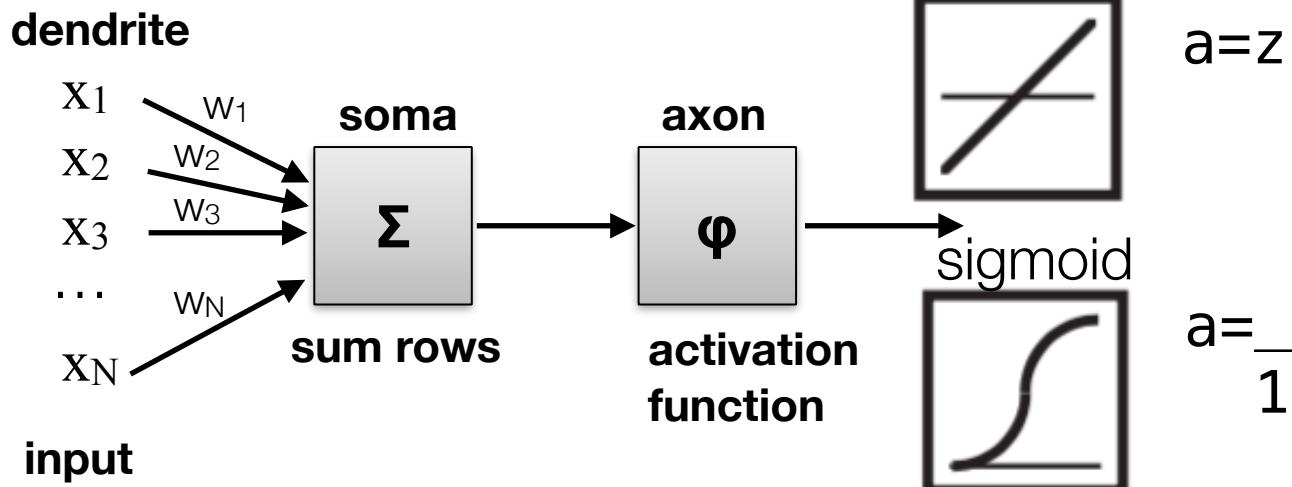
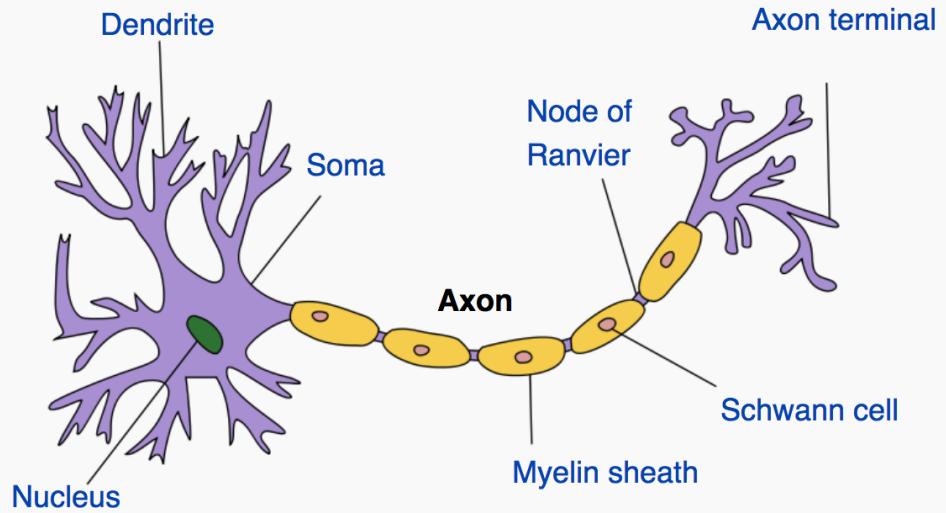


Walter Pitts

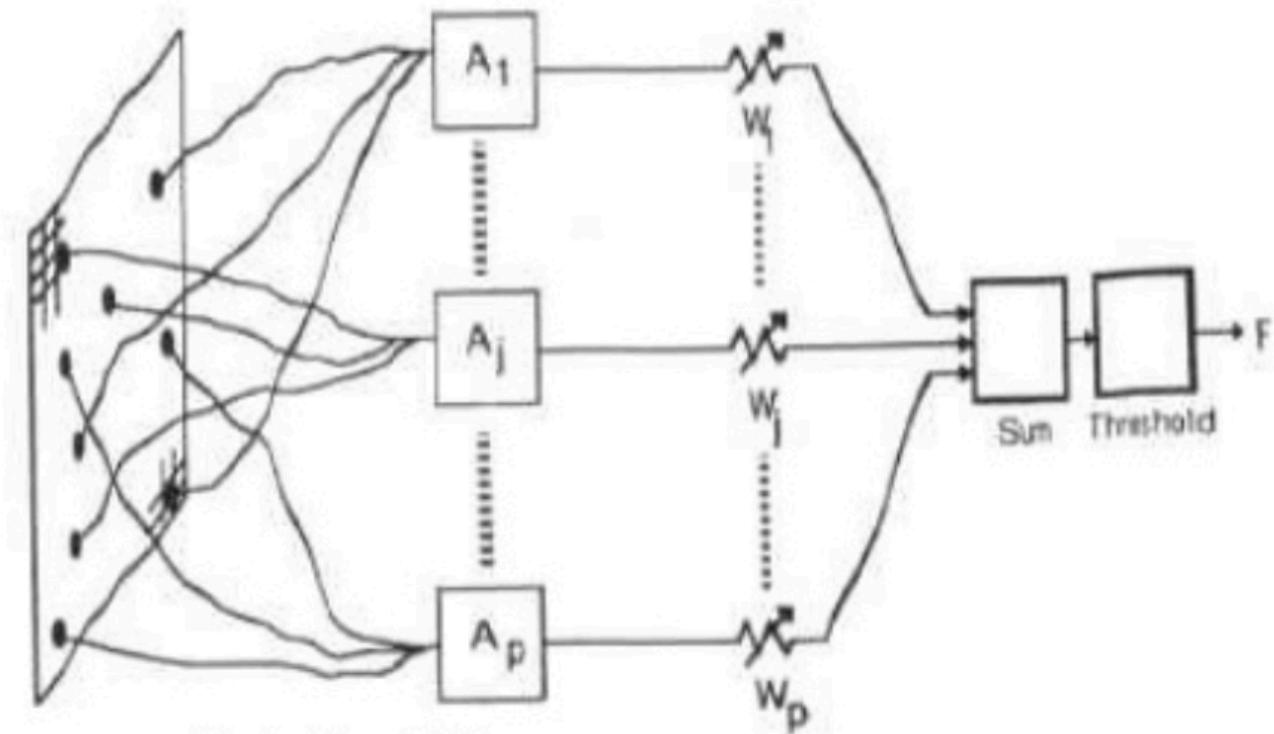
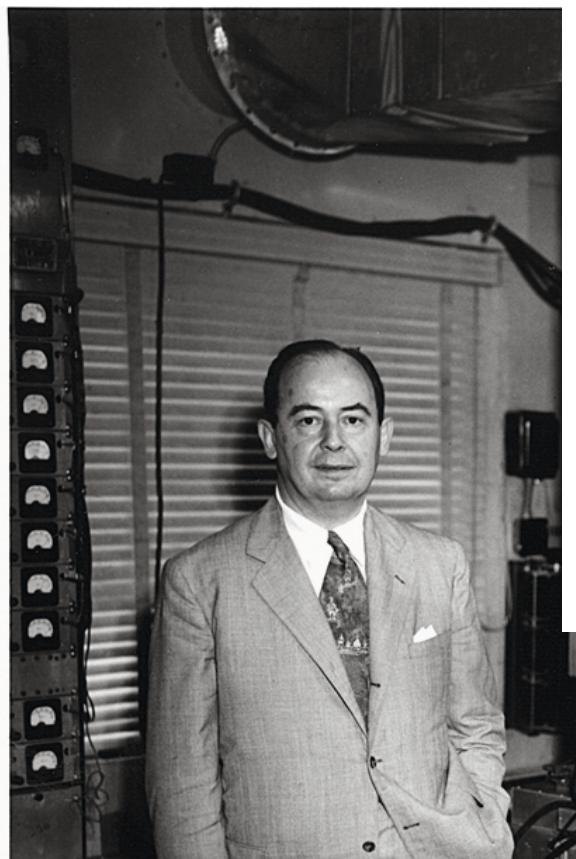


Neurons

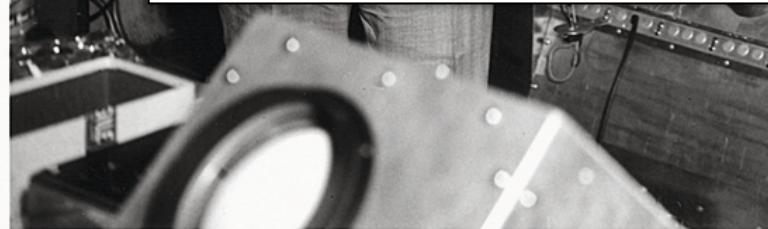
- Rosenblatt's perceptron, 1957



The Mark 1

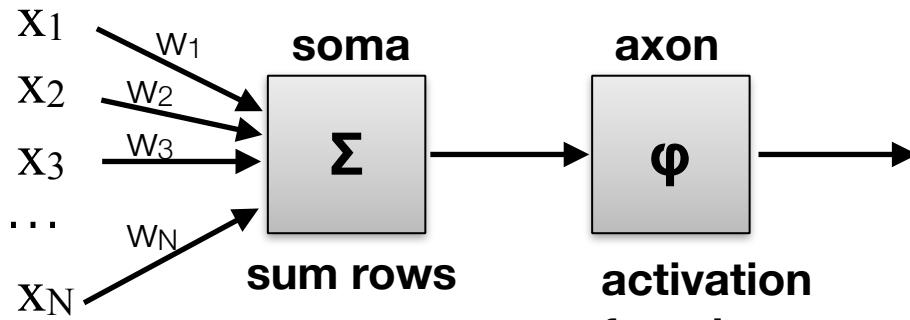


Perceptron Learning Rule:
~Stochastic Gradient Descent

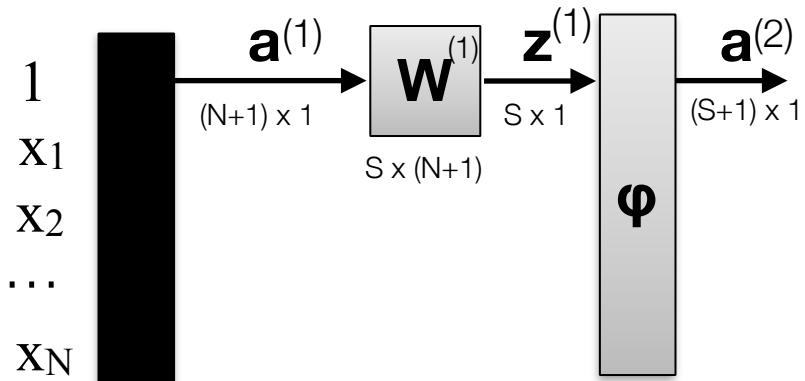


Neurons

dendrite



input



- **Some notation**

- need bias term
- matrix representation
- multiple layers

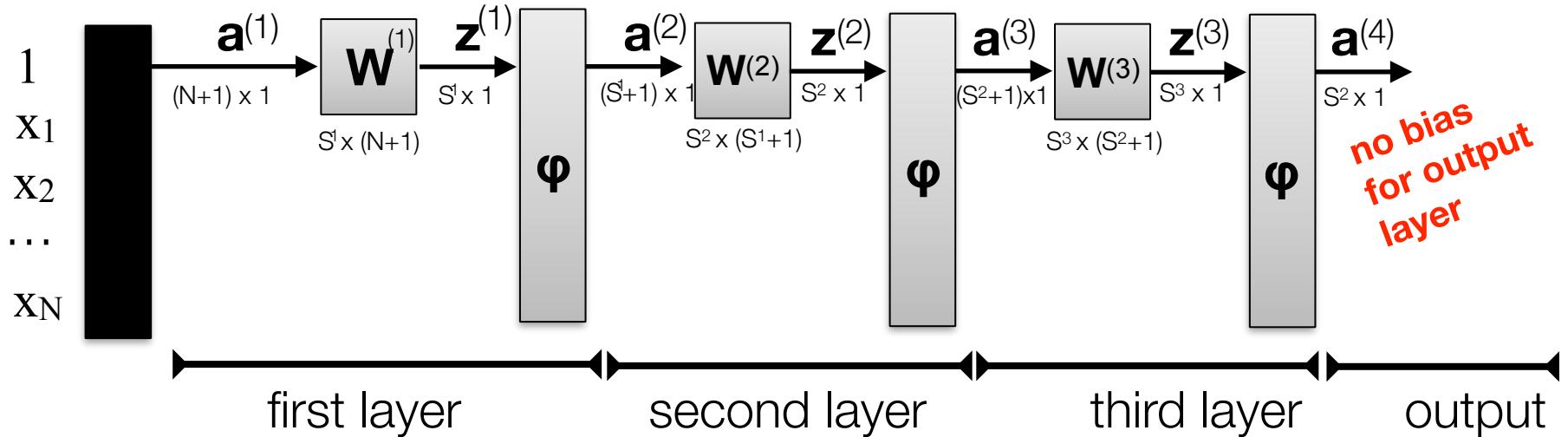
$$\mathbf{a}^{(1)} = \mathbf{x} + \text{concat bias term}$$

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

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

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

Multiple layers notation



$$a^{(L+1)} = \Phi(z^{(L)}) + \text{concat bias term} \quad a^{(4)} \text{ rows=unique classes}$$

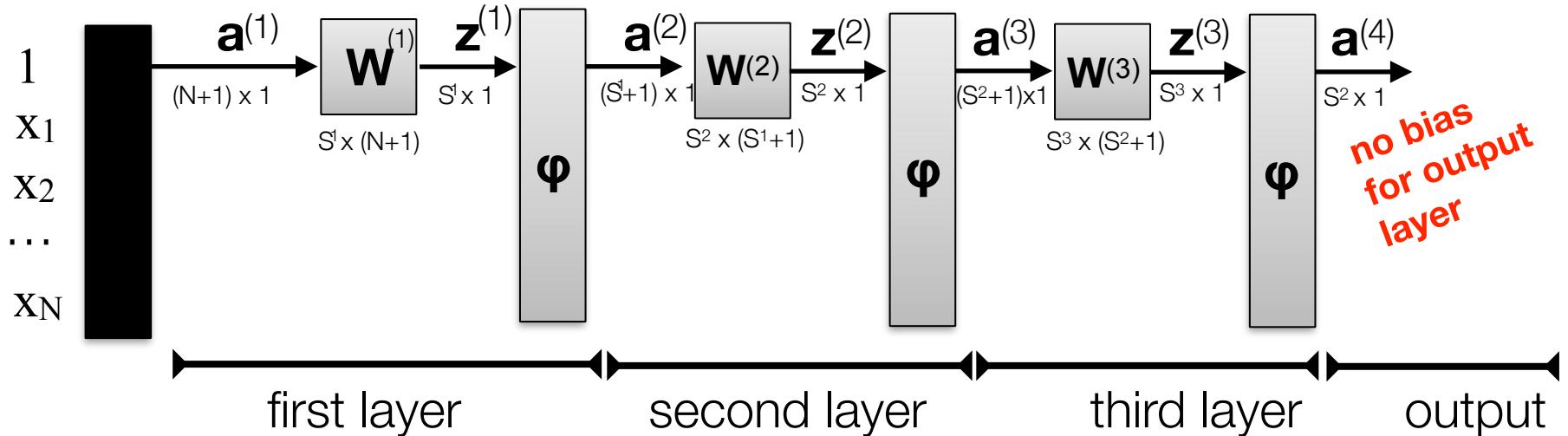
$$z^{(L)} = W^{(L)} a^{(L)}$$

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

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

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

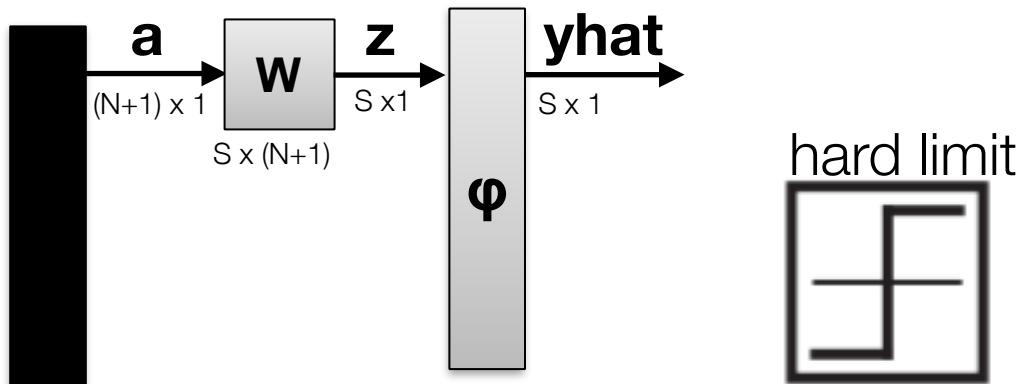
Multiple layers notation



- **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, $z^{(i)}$

Simple Architectures

- Rosenblatt's perceptron, 1957



hard limit



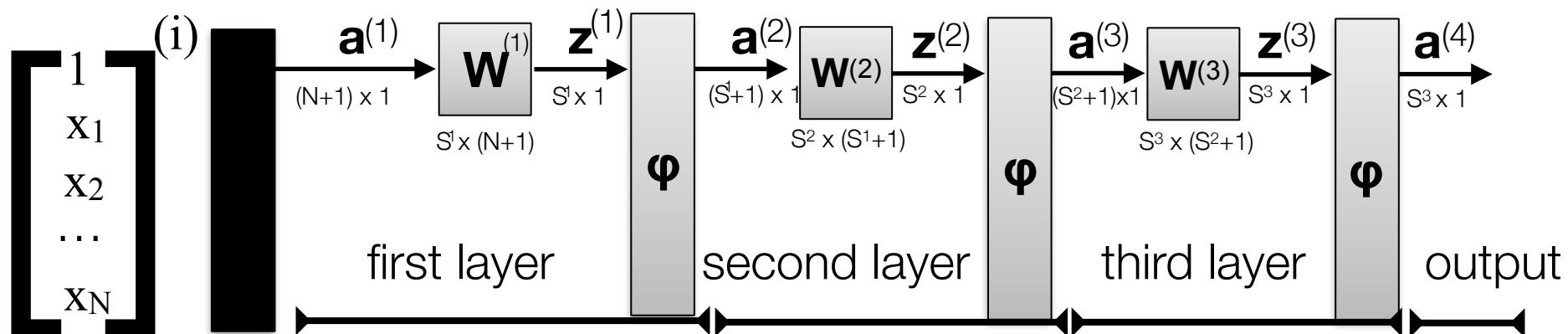
$$\begin{array}{ll} a = -1 & z < 0 \\ a = 1 & z \geq 0 \end{array}$$



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

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

Compact feedforward notation



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

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

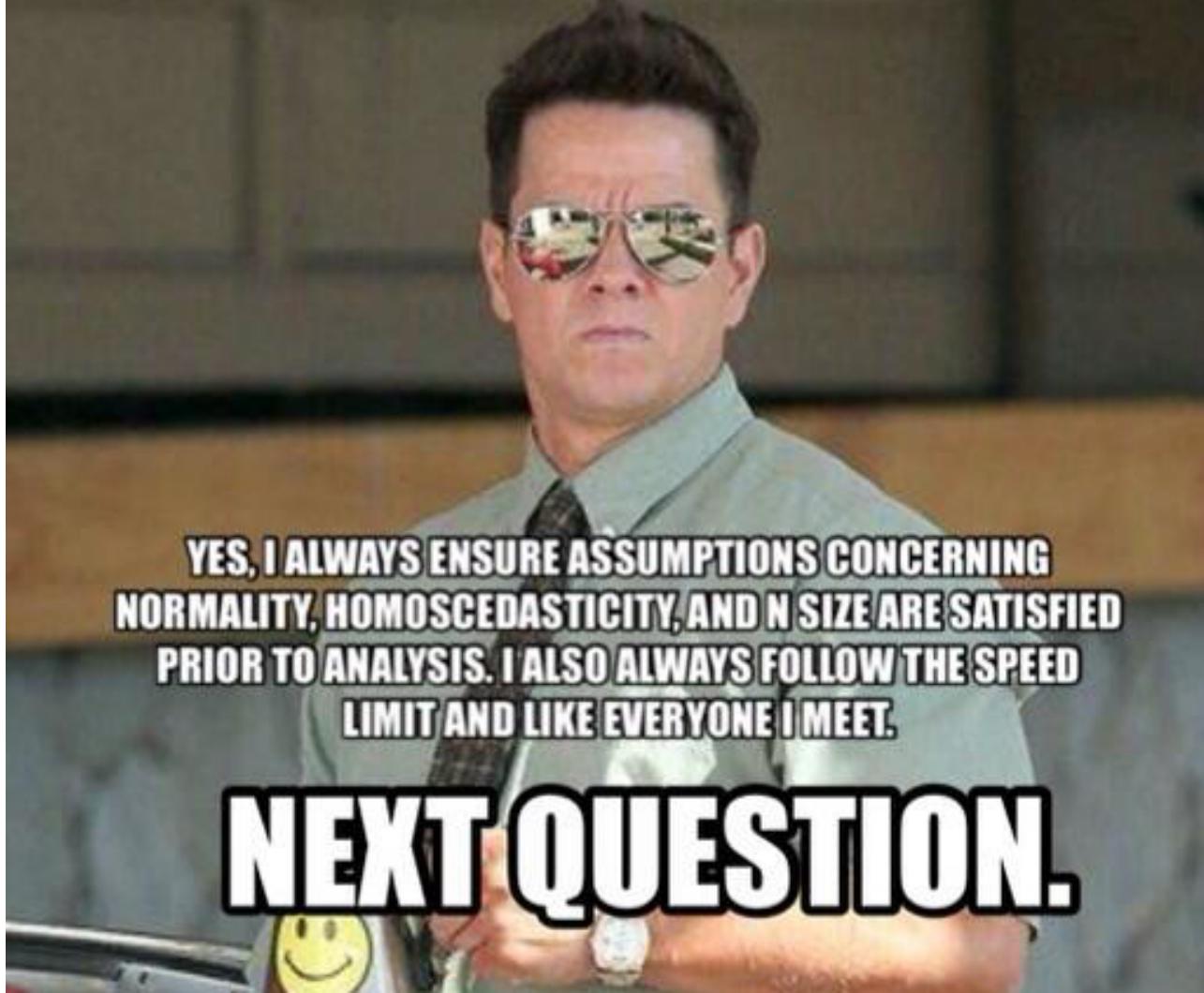
$$\begin{bmatrix} z^{(L)}_1 \\ \vdots \\ z^{(L)}_{S^L} \end{bmatrix}^{(i)} = \begin{bmatrix} w^{(L)}_{1,1} & w^{(L)}_{1,2} & w^{(L)}_{1,3} & \dots & w^{(L)}_{1,S^{L-1}+1} \\ \vdots & \ddots & & & \\ w^{(L)}_{S^L,1} & w^{(L)}_{S^L,2} & \dots & & w^{(L)}_{S^L,S^L-1+1} \end{bmatrix} \begin{bmatrix} a^{(L)}_1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

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

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

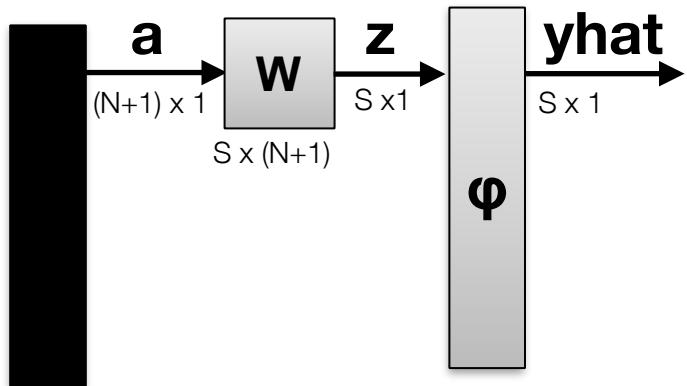
$$\mathbf{Z}^{(L)} = \mathbf{W}^{(L)} \mathbf{\Phi}(\mathbf{W}^{(L-1)} \mathbf{A}^{(L-1)})$$

Training Neural Network Architectures



Simple Architectures

- Rosenblatt's perceptron, 1957

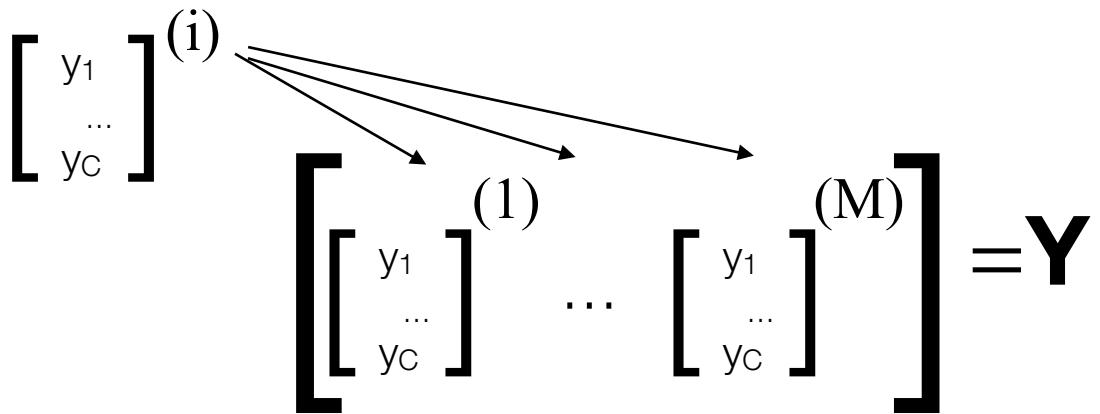


$$\sum_i^M (\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)})^2$$



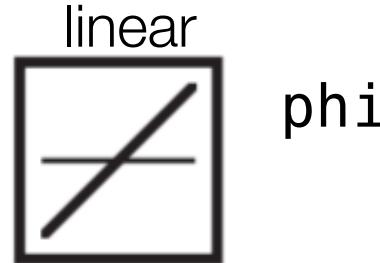
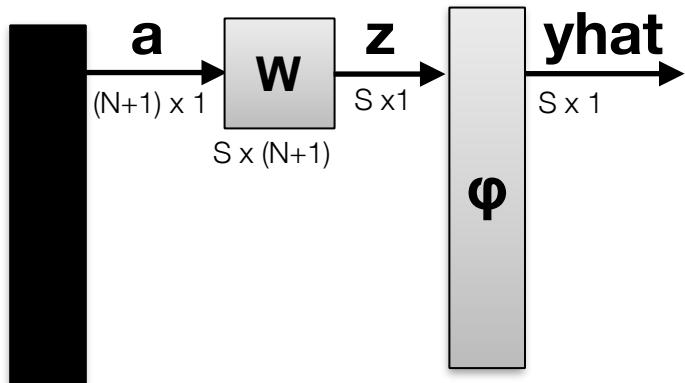
Need objective Function, minimize MSE $J(\mathbf{W}) = ||\mathbf{Y} - \hat{\mathbf{Y}}||^2$

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

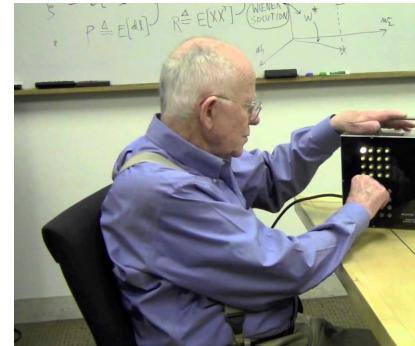


Simple Architectures

- Adaline network, Widrow and Hoff, 1960



Marcian "Ted" Hoff



Bernard Widrow

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

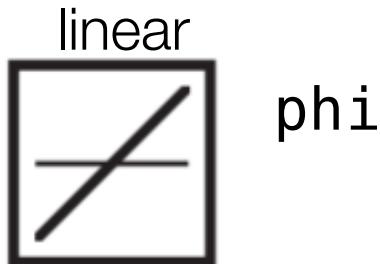
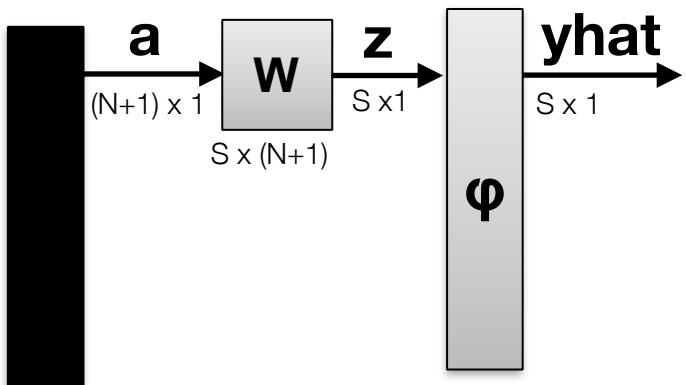
New objective function becomes: $J(\mathbf{W}) = ||\mathbf{Y} - \mathbf{W} \cdot \mathbf{A}||^2$

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 $\mathbf{W} \leftarrow \mathbf{W} + \eta \nabla J(\mathbf{W})$

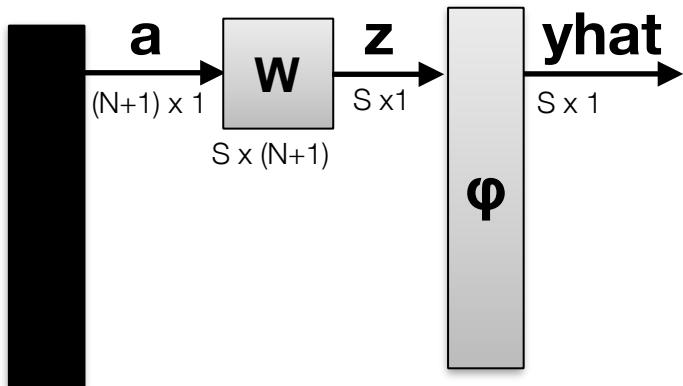
For case $S=1$, \mathbf{W} has only one row, \mathbf{w}
this is just **linear regression...**

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



Simple Architectures

- Modern Perceptron network



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

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

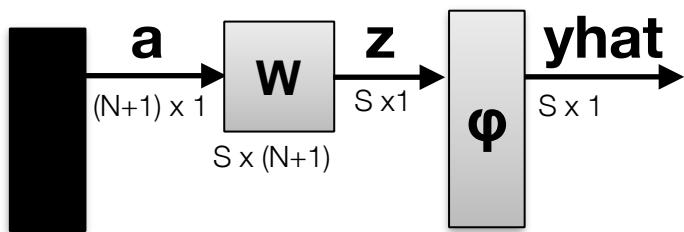
For case $S=1$, this is just **logistic regression...**
and **we have already solved this!**

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



What happens when $S > 1$?

What if we have more than S=1?



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

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

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

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

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



which is one-versus-all!

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

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

...

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

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



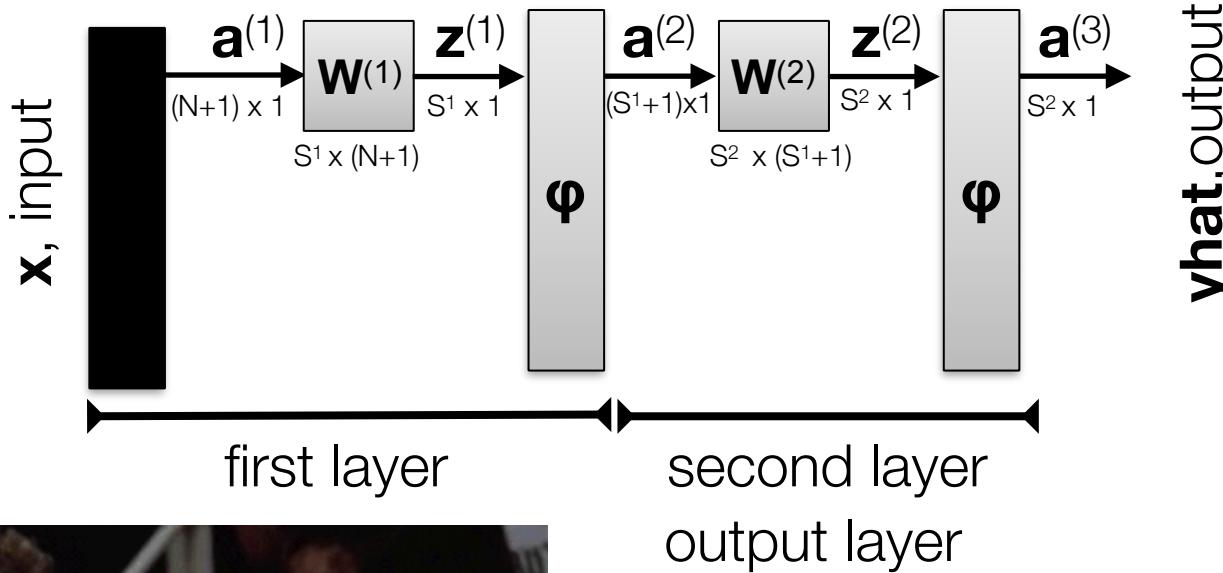
1942-2011

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*)



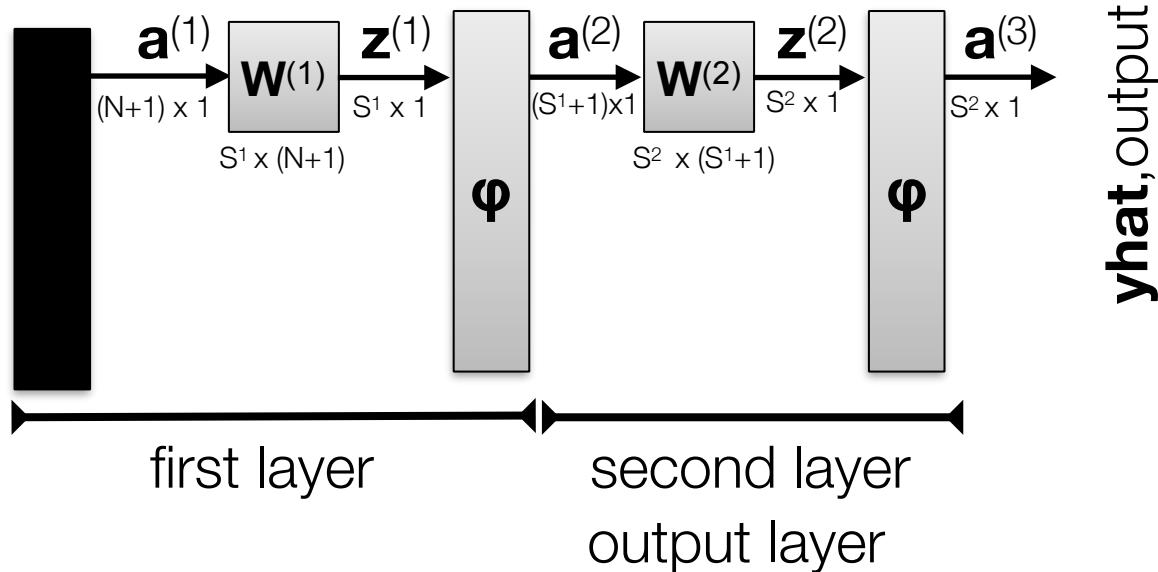
each row of $y\hat{}$ is no longer
independent of the rows in W
so we cannot optimize using
one versus all!!!



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

Back propagation

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

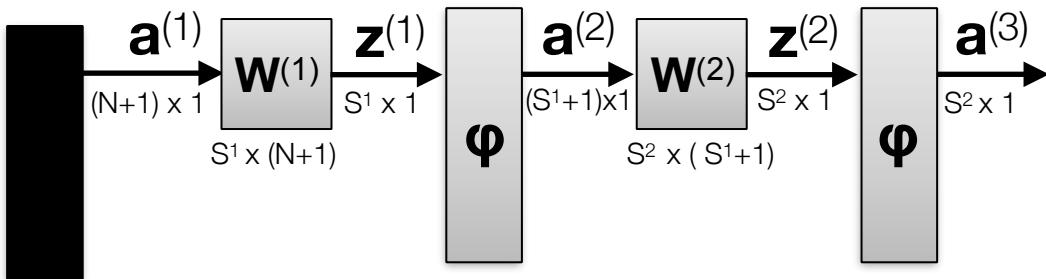


$$J(\mathbf{W}) = \sum_k^M (\hat{\mathbf{y}}^{(k)} - \mathbf{a}^{(L)})^2$$

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



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!

End of Session

- thanks! **Next time is ICA3!!!**

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=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
Multi-Layer Neural Networks

Class Logistics and Agenda

- Agenda:
 - Practical Multi-layer Architectures
 - Programming Examples
- Next Time: No Lecture (out of town)

Last Time: History of Neural Nets

- 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



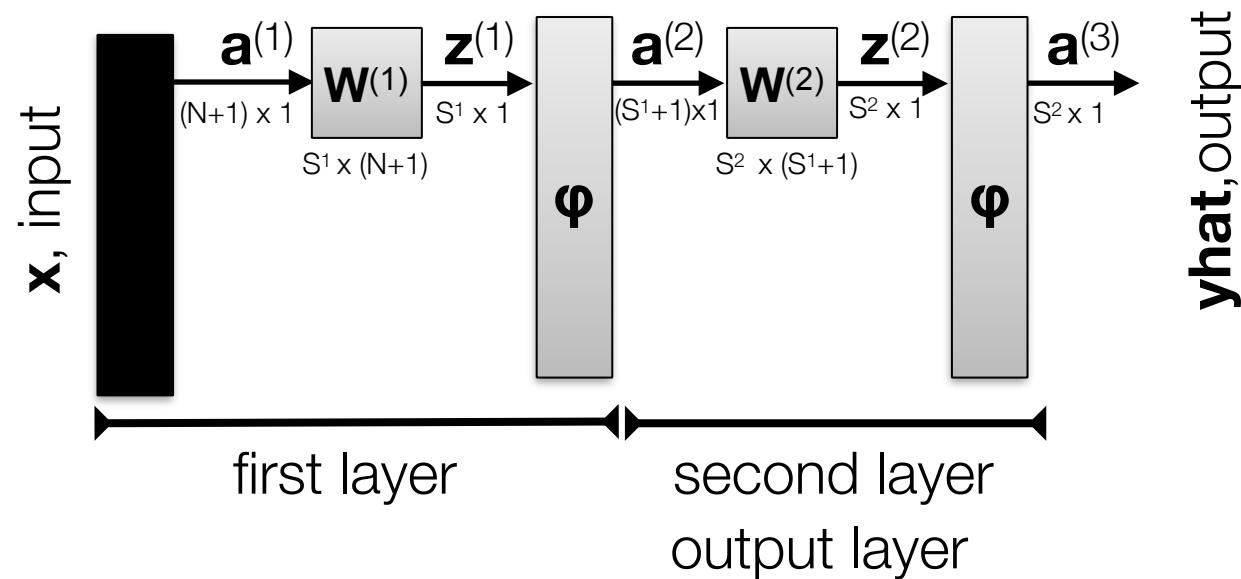
1942-2011

Geoffrey Hinton



Back propagation

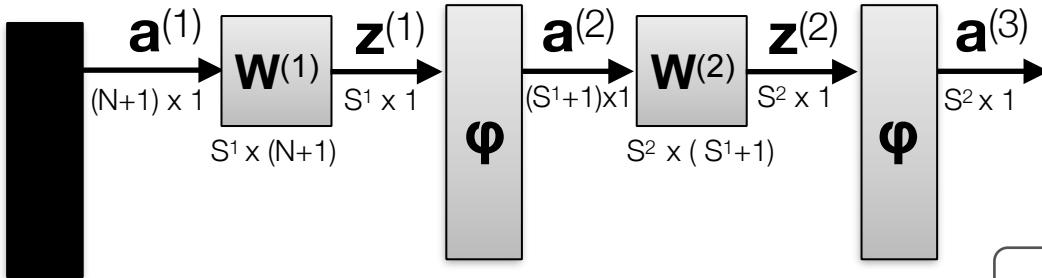
- Steps:
 - propagate weights forward
 - calculate gradient at final layer
 - back propagate gradient for each layer
 - via recurrence relation



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

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

Back propagation summary



1. Forward propagate to get Z, A

2. Get final layer gradient

3. Back propagate sensitivities

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

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

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

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

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

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

Review: Two Layer Perceptron

with regularization
and vectorization

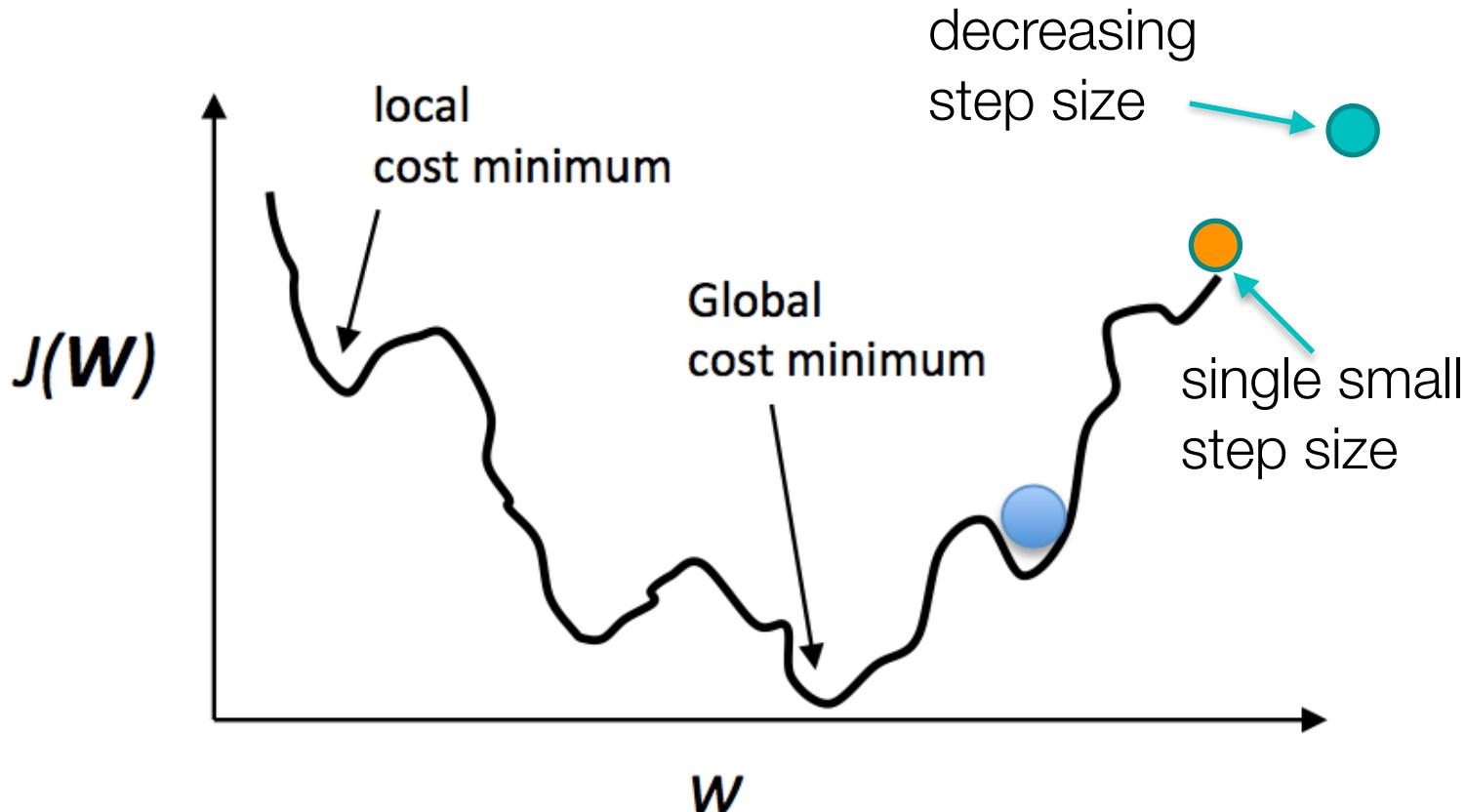


Problems with Advanced Architectures

- 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

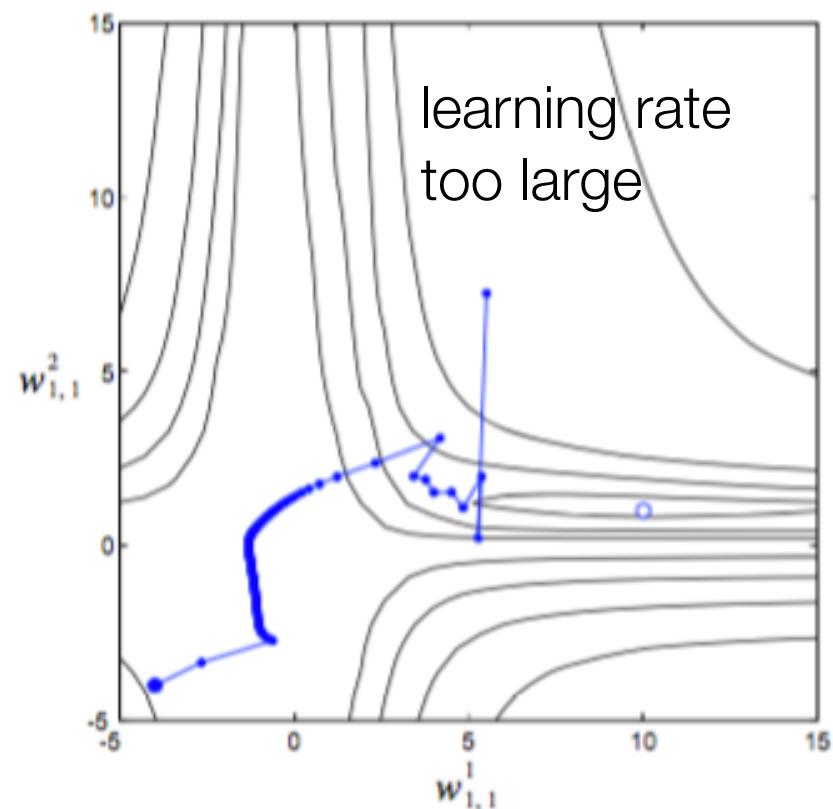
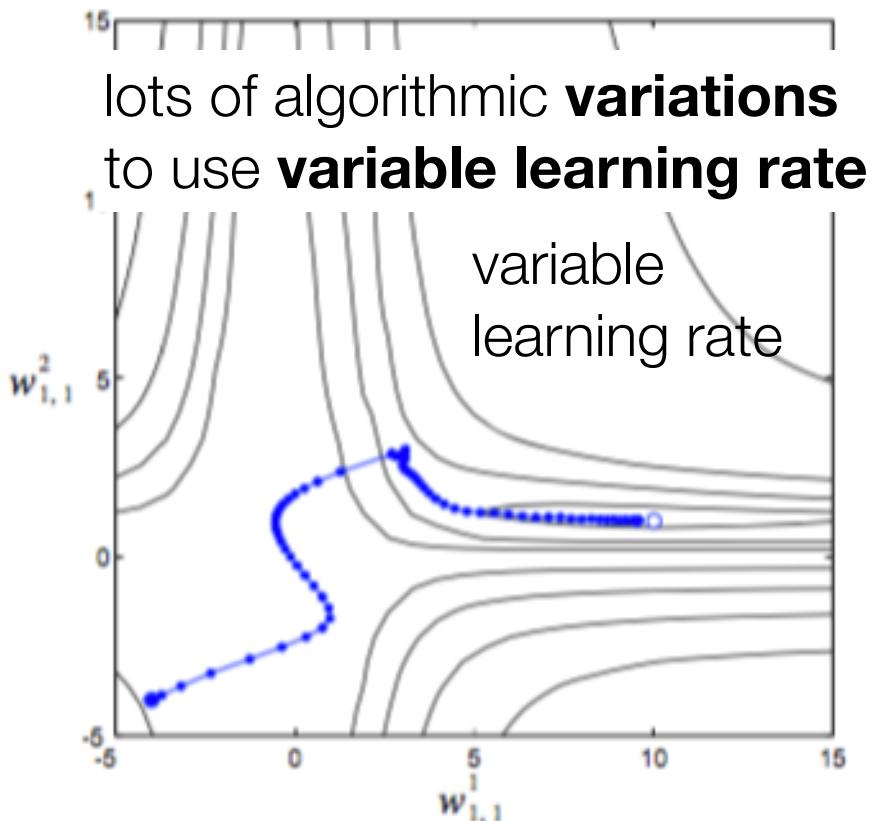
Problems with Advanced Architectures

- 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



Two Layer Perceptron

comparison:

mini-batch

momentum

decreased learning

L-BFGS



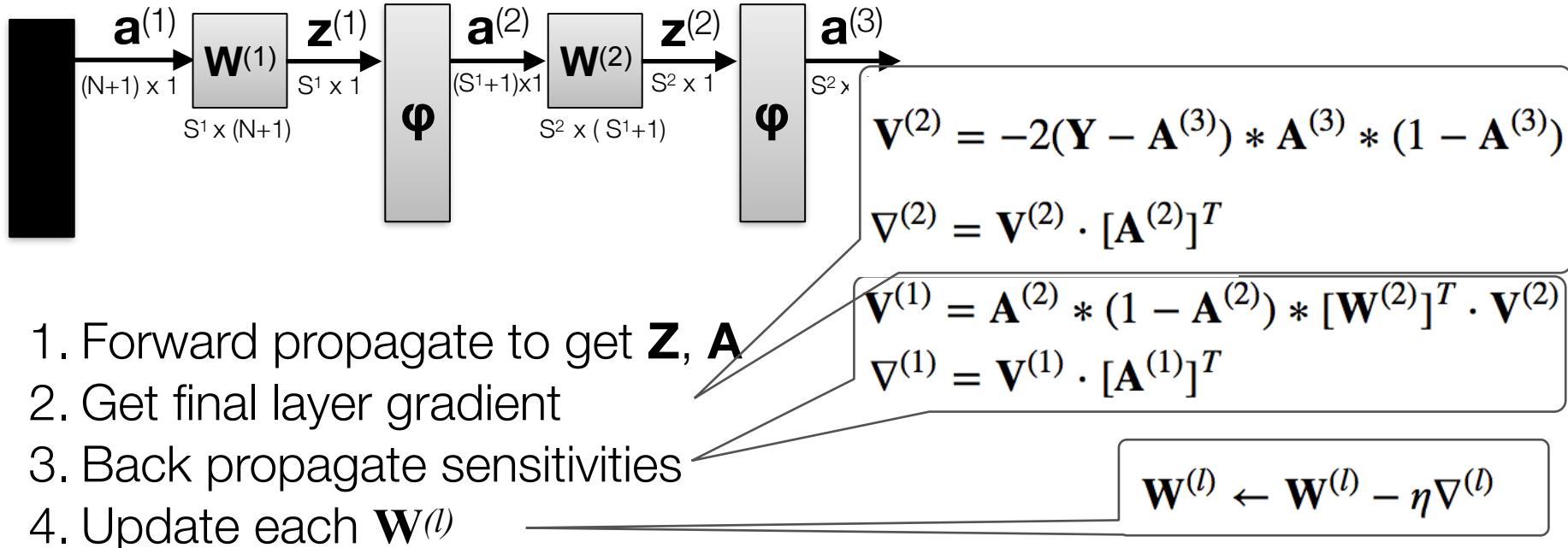
Lecture Notes for Machine Learning in Python

Professor Eric Larson
Multi-Layer Neural Networks

Class Logistics and Agenda

- Agenda:
 - Practical Multi-layer Architectures
 - Programming Examples
 - Town Hall

Review:



- **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.

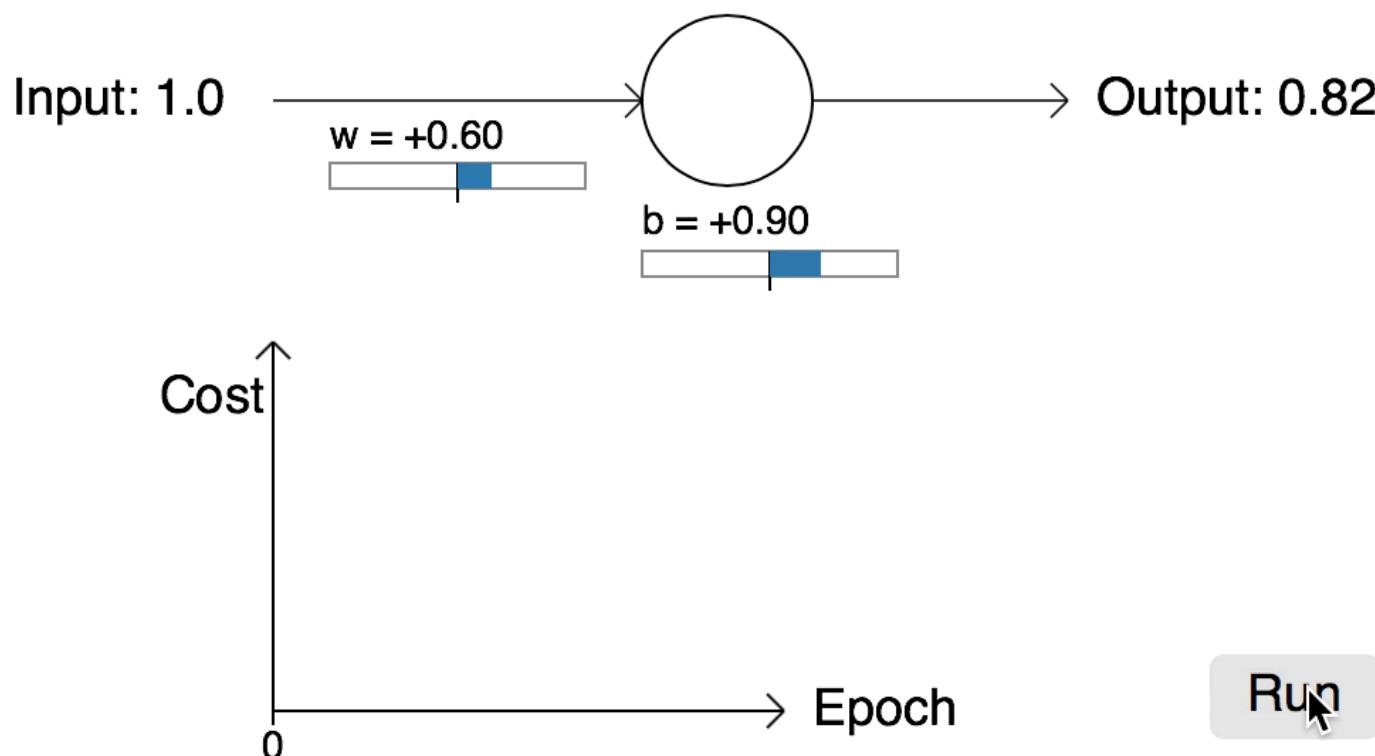
- A. True
- B. False

Practical Implementation of Architectures

- A new cost function?

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

least squares objective,
tends to slow training initially

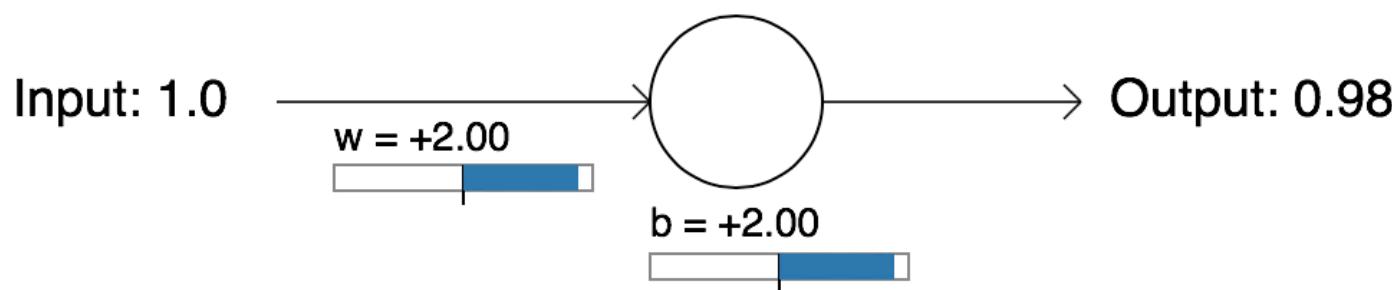


Practical Implementation of Architectures

- A new cost function?

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

least squares objective,
tends to slow training initially



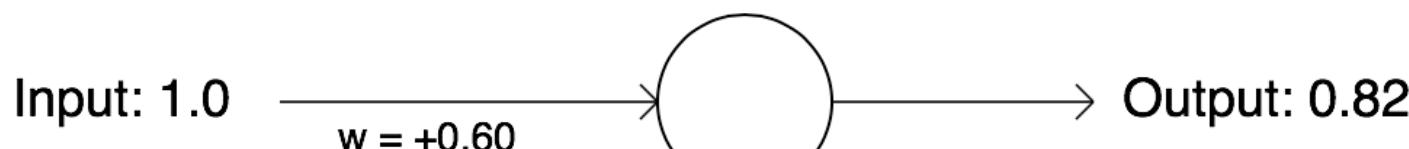
Run

Practical Implementation of Architectures

- A “new” cost function: **Cross entropy**

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

speeds up
initial training



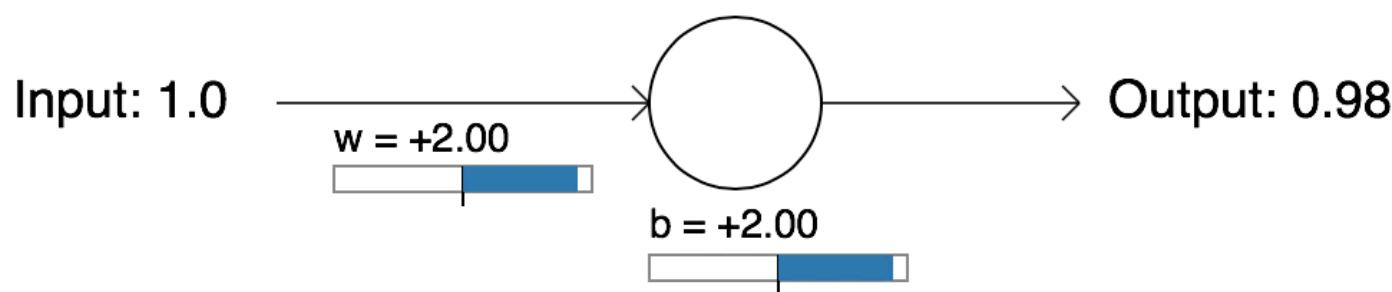
Run

Practical Implementation of Architectures

- A “new” cost function: **Cross entropy**

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

speeds up
initial training



Practical Implementation of Architectures

- A “new” cost function: **Cross entropy**

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

speeds up
initial training

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

bp-5
55

Practical Implementation of Architectures

- A “new” cost function: **Cross entropy**

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

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

56

Two Layer Perceptron

cross entropy

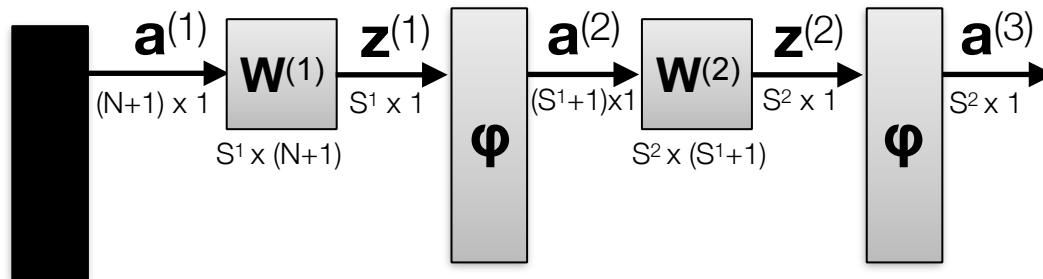


Practical Implementation of Architectures

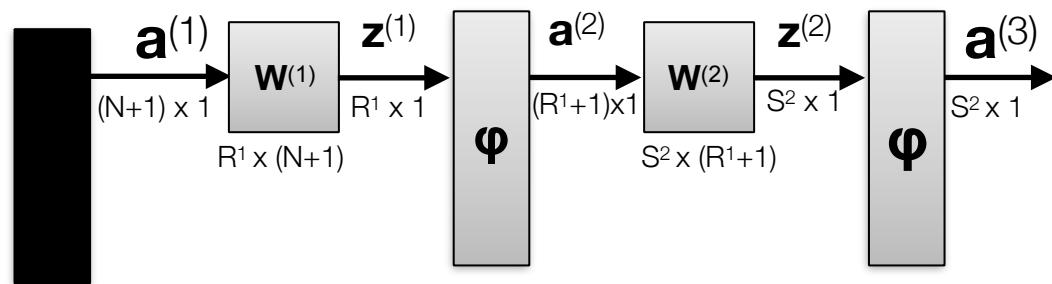
- Beyond L2 regularization
 - dropout
 - expansion (if time)

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

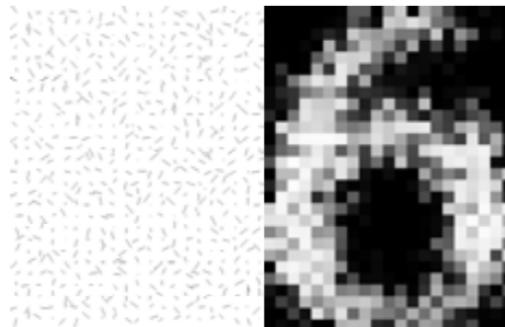
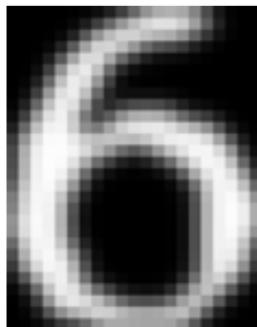
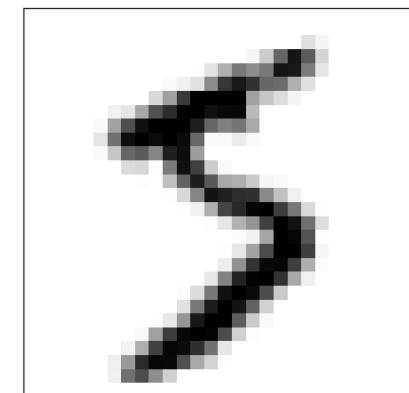
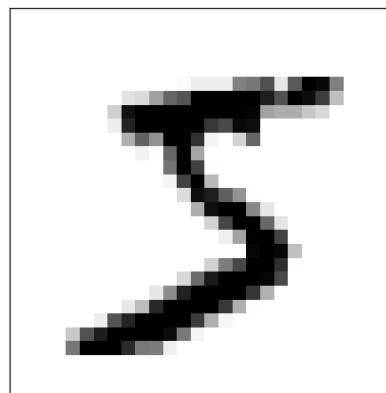


Why does this work?

Practical Implementation of Architectures

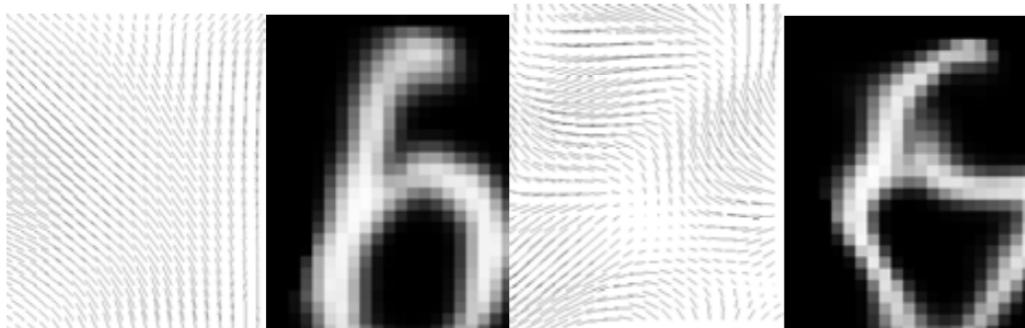
- Expansion

- 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%

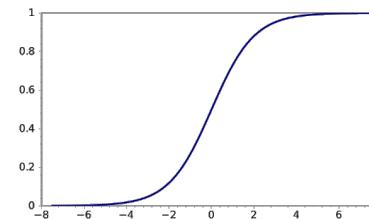
Practical Implementation of Architectures

- Weight initialization
 - uniform distribution makes weights unrealistic
 - try not to **saturate** your neurons right away!

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



each row is sum of layer inputs



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

solution: squash initial weights sizes

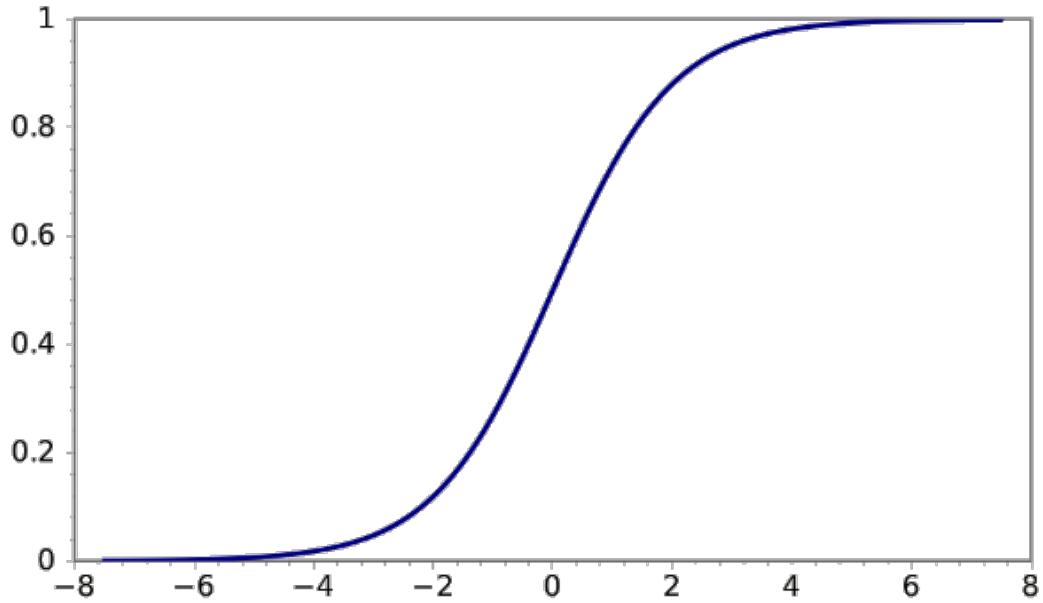
- a nice choice: each element of \mathbf{W} is selected from a Gaussian with zero mean
- for adding Gaussian distributions, variances add together:
 - make each variance $1/\mathbf{W}_{\text{num_elements_in_row}}$
 - which is the same as standard deviation = $1/\sqrt{\mathbf{W}_{\text{num_elements_in_row}}}$

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



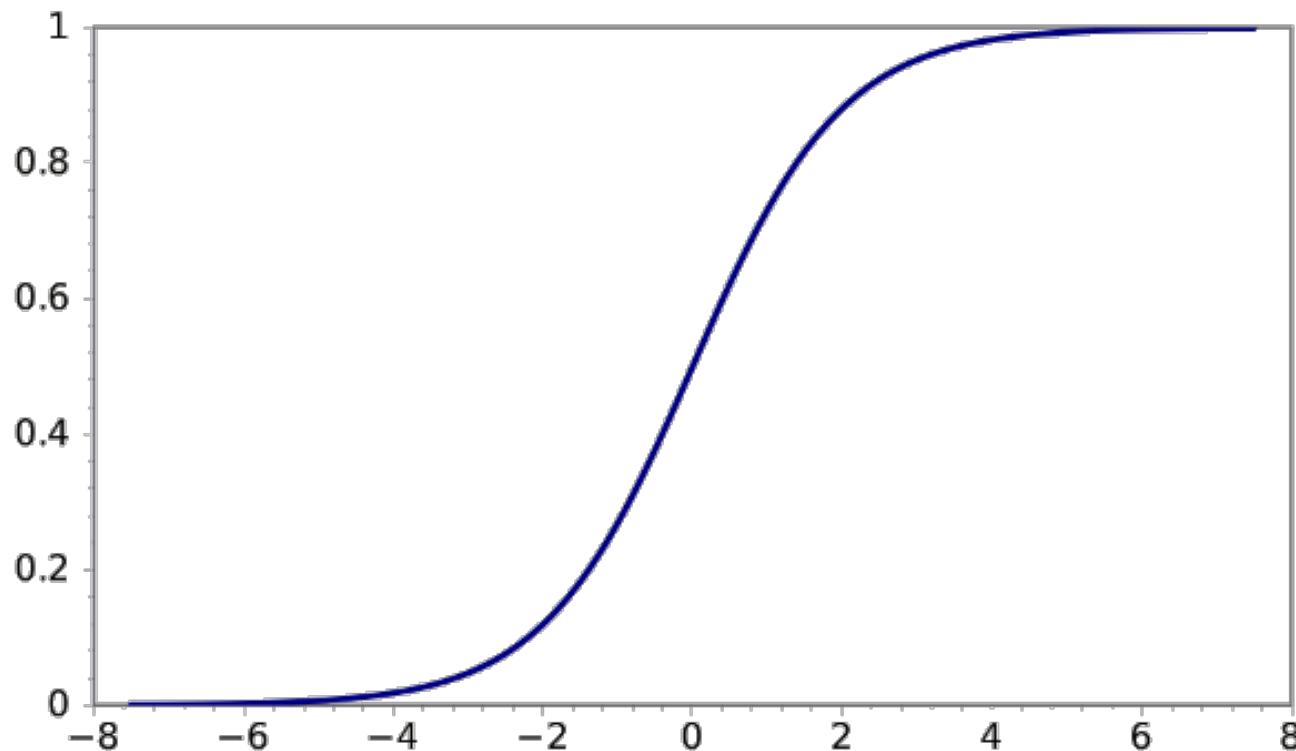
Self Test

- 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



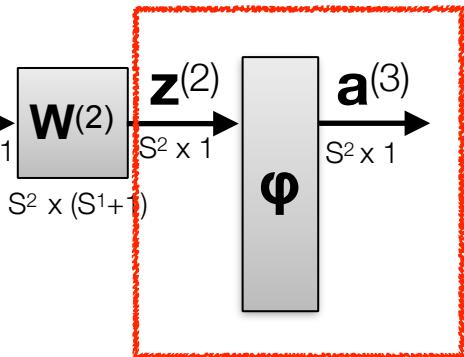
Two Layer Perceptron

Dropout
Smarter Weight Initialization



Practical Implementation of Architectures

- A new nonlinearity: **softmax**



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

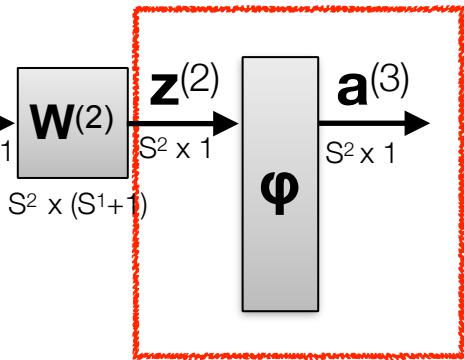
instead of final layer sigmoid!

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

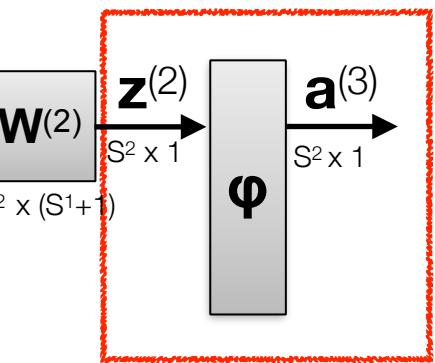
$$\frac{\partial \phi(\mathbf{z}^{(i)})}{\partial \mathbf{z}^{(i)}} = \begin{cases} 1, & \text{if } \mathbf{z}^{(i)} > 0 \\ 0, & \text{else} \end{cases}$$

Two Layer Perceptron

ReLU Nonlinearities



Practical Implementation of Architectures



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

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

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

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

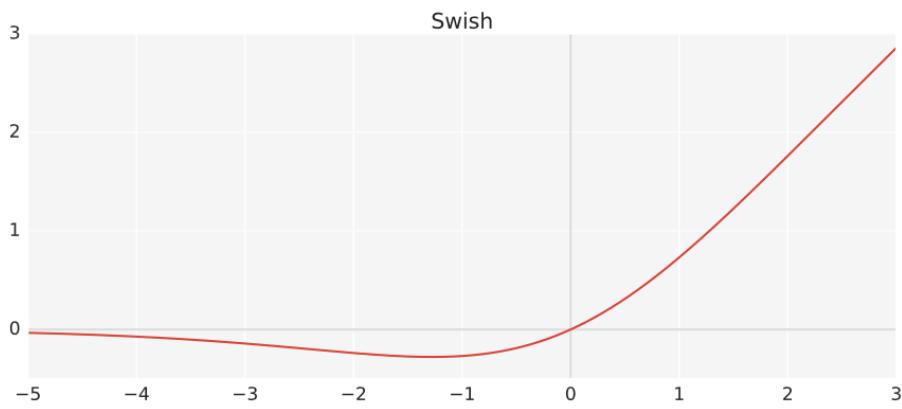


Figure 1: The Swish activation function.

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: Project Work Day
- Next Time: Deep Learning in Keras

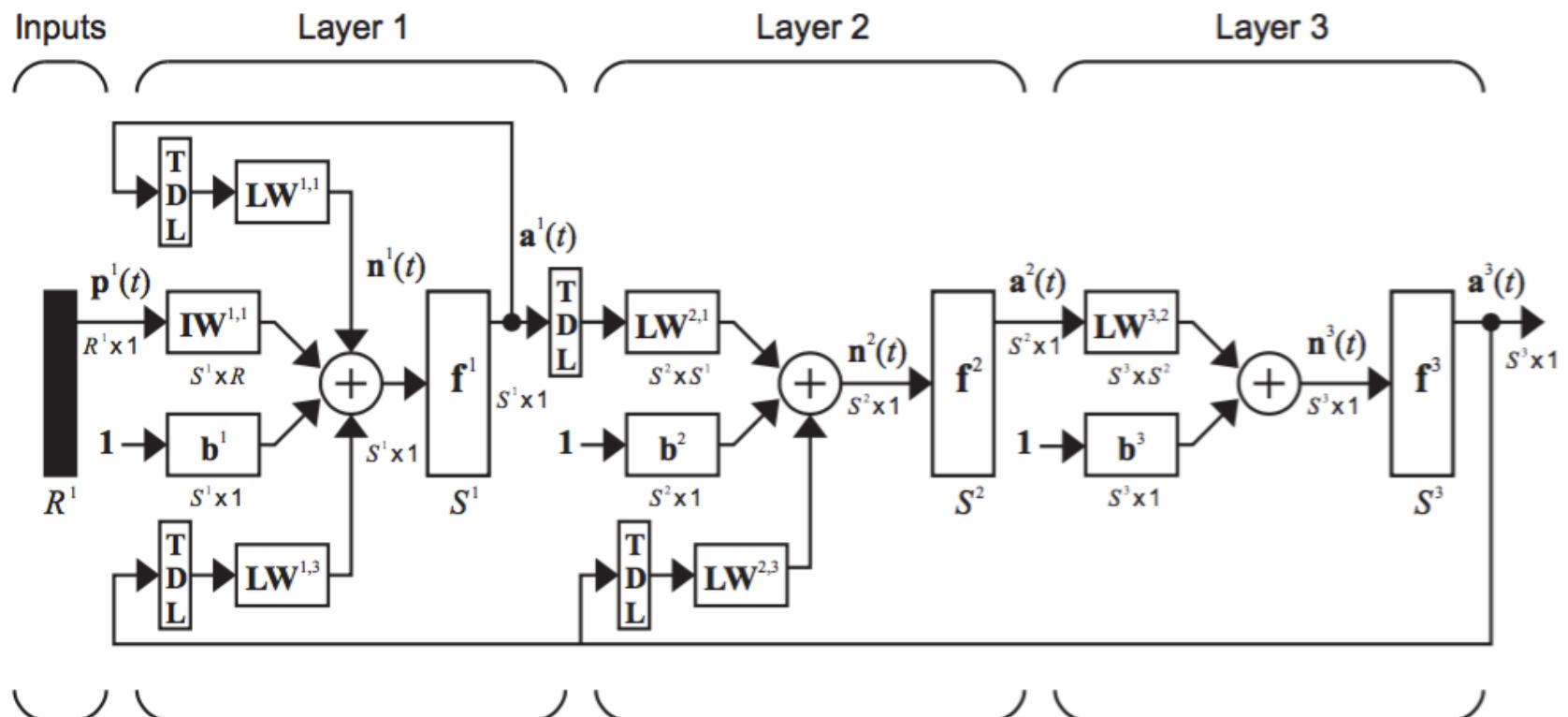
Class Logistics and Agenda

- Logistics
 - Welcome back!
 - End of this week: evaluating MLP
- Two Week Agenda:
 - *SVM Review*
 - *Neural Networks History*
 - *Multi-layer Architectures*
 - Programming Multi-layer training
- **Next Time:** evaluation in-class assignment

Back up slides

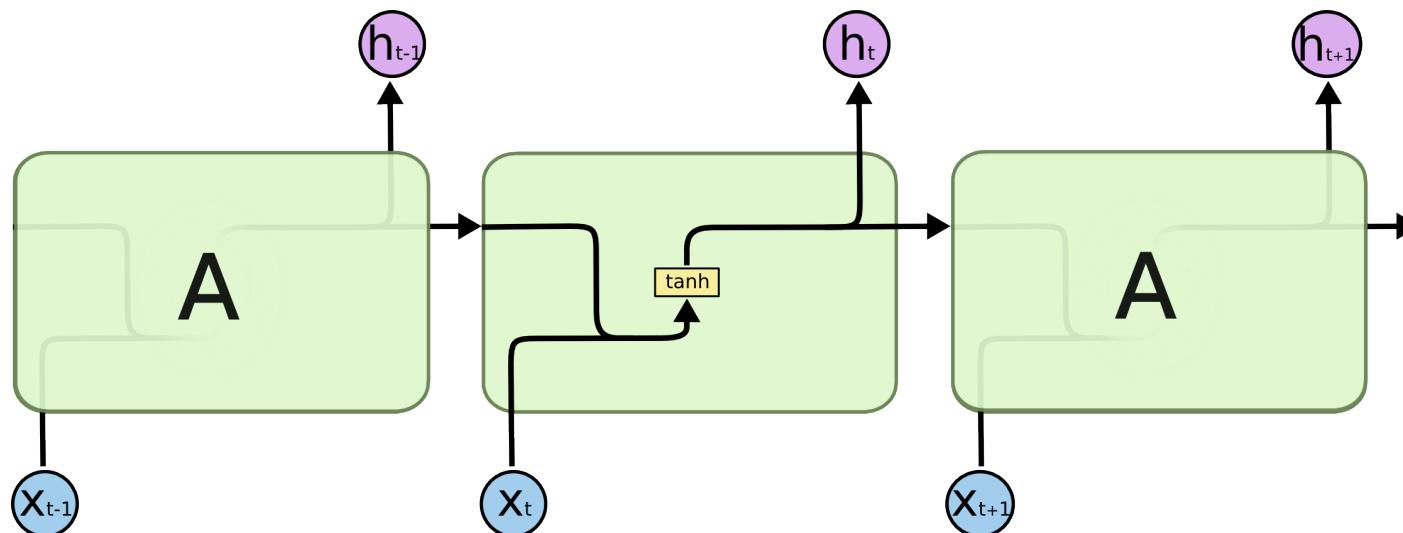
More Advanced Architectures

- 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 memory, **LSTM**



More Advanced Architectures

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



Neural Network
Layer



Pointwise
Operation



Vector
Transfer



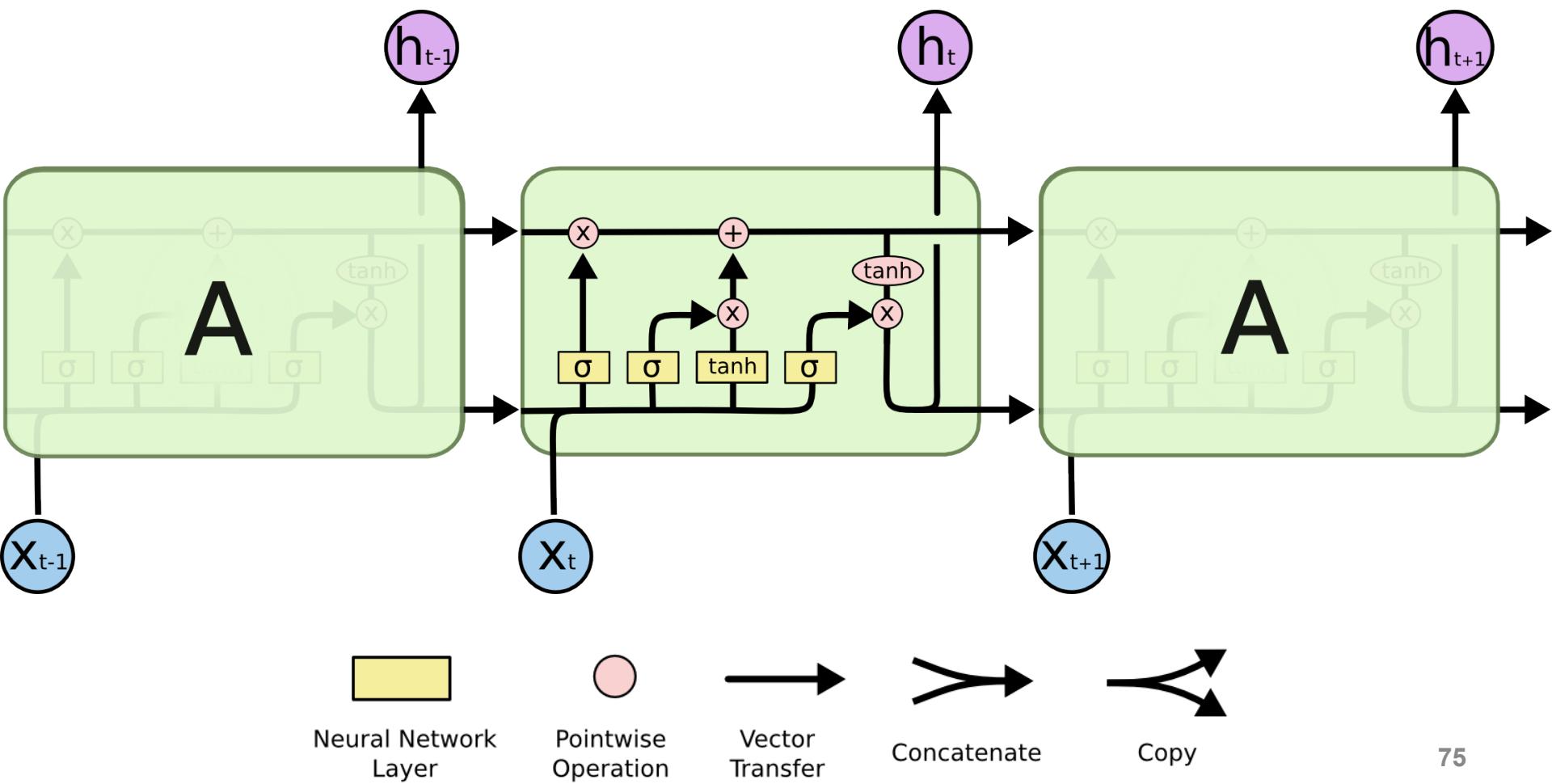
Concatenate



Copy

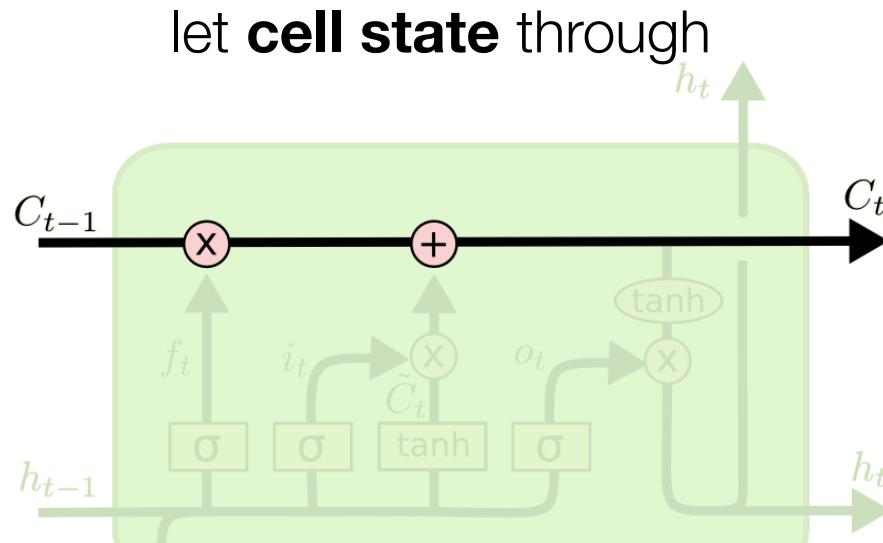
More Advanced Architectures

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

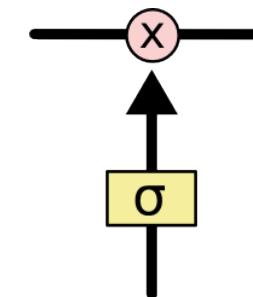


More Advanced Architectures

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

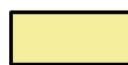


potentially **forget past** inputs via “gate” σ



we will return to this architecture later, for now:

put it in long term memory 😂



Neural Network
Layer



Pointwise
Operation



Vector
Transfer



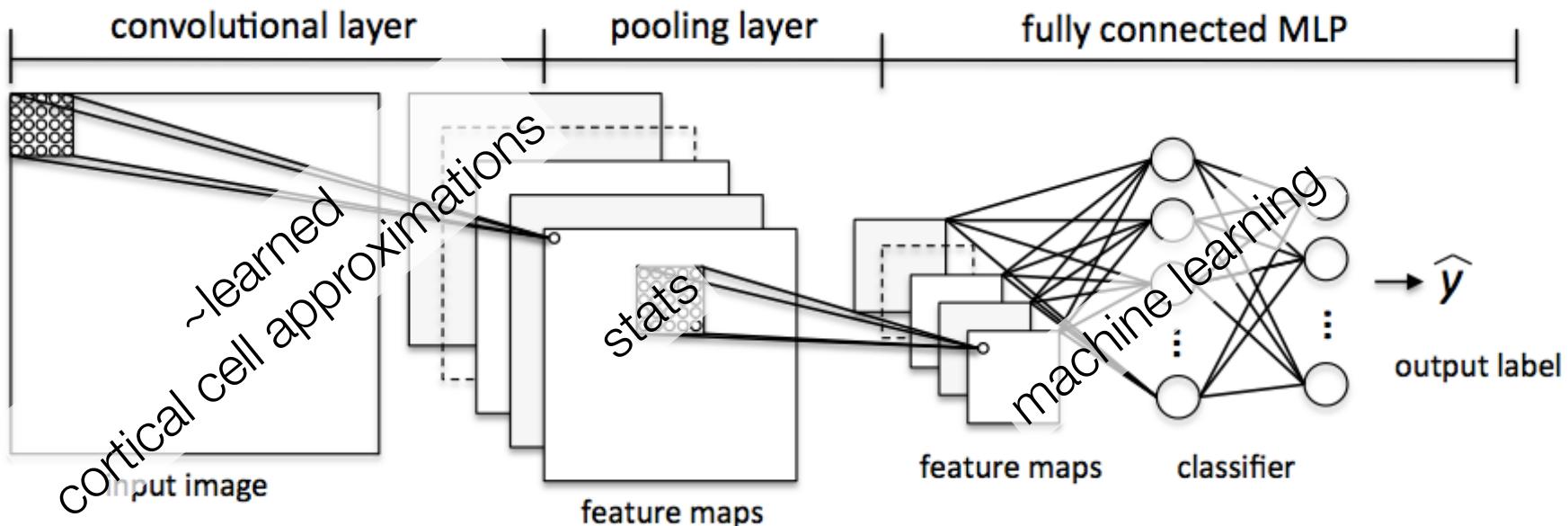
Concatenate



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**