# Lecture Notes for Machine Learning in Python

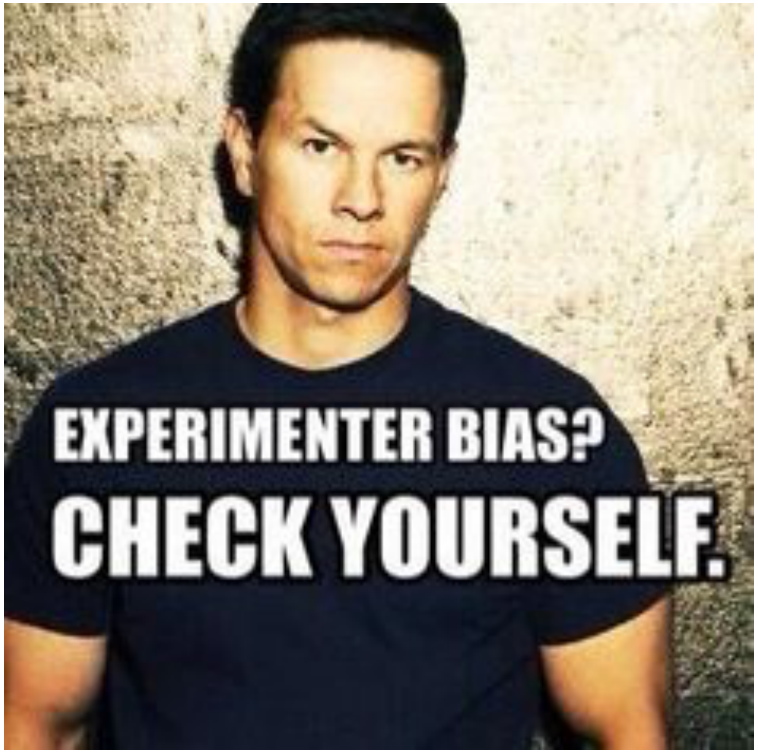
Professor Eric Larson

Week Seven A

# Class Logistics and Agenda

- Grades Coming Soon
- A2 will be about using linear classifiers to perform classifications tasks
  - extending?
- Two Week Agenda:
  - SVM Review
  - Neural Networks History
  - Multi-layer Architectures
  - Programming Multi-layer training

# Support Vector Machine Review



# SVMs Summary: Linear

- Linear SVMs
  - Architecture near identical to logistic regression, except:
    - maximize margin
    - constrained optimization

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

$$\text{to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i,$$

$$\zeta_i \geq 0, i = 1, ..., n$$

  - **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

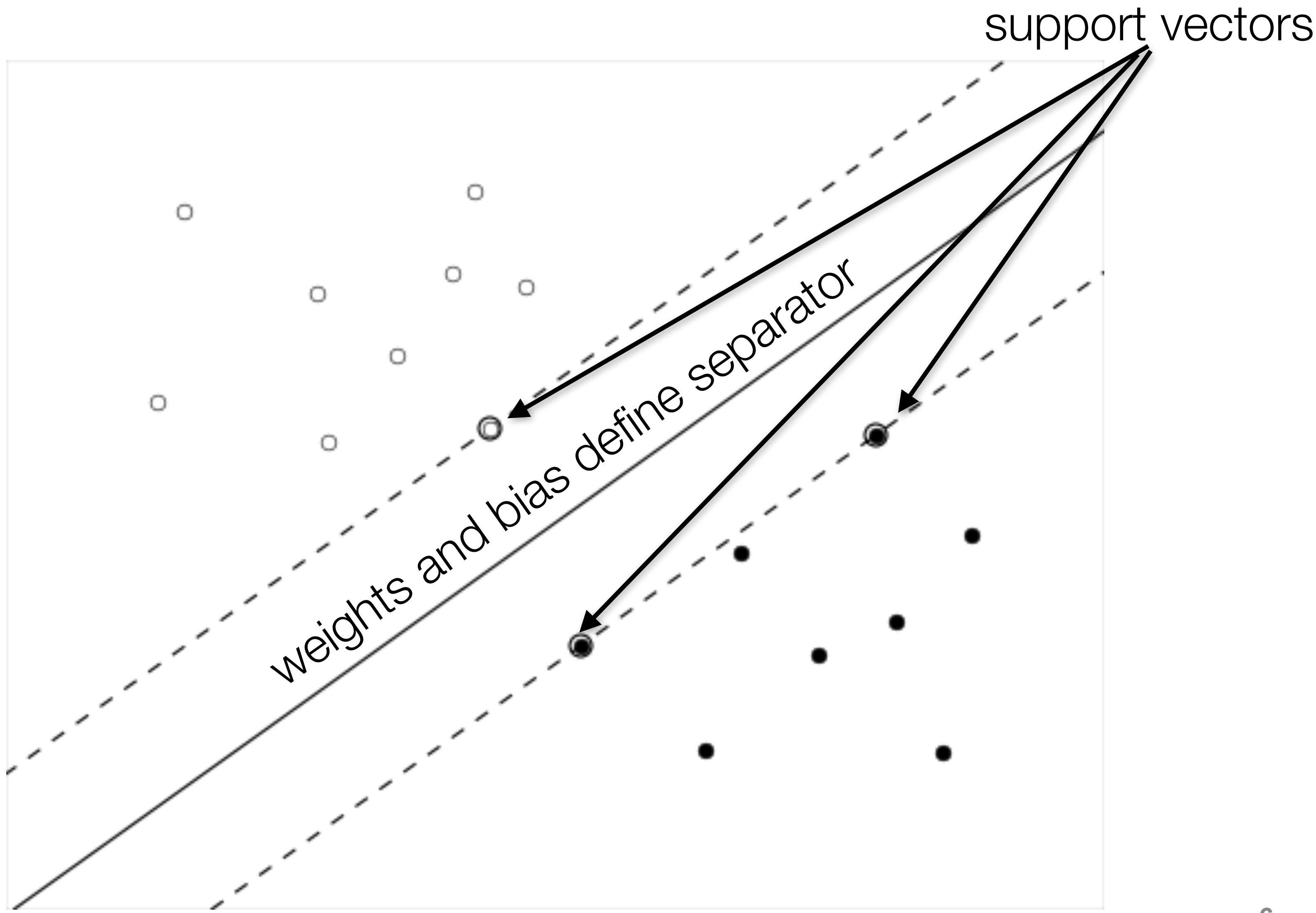
$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i$$

$$\text{to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i,$$

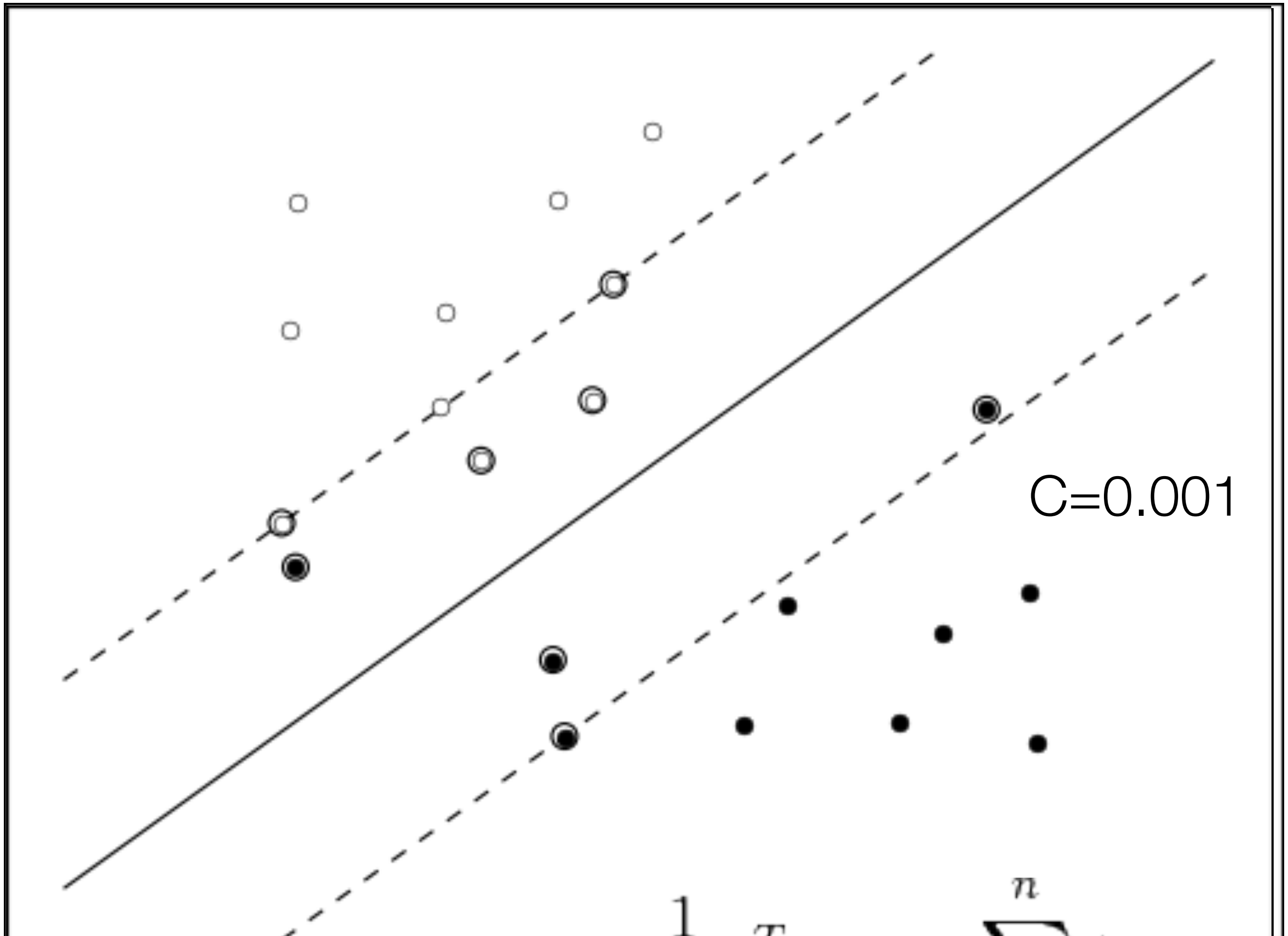
$$\zeta_i \geq 0, i = 1, ..., n$$

  - Trained Parameters:
    - bias and weights for each class
    - support vectors chosen for margin calculation
    - slack variables for inseparable cases

# SVMs Summary: Linear



# SVMs Summary: Linear




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

tradeoff margin and accuracy

# SVMs Summary: non-linear

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

# SVMs Summary: non-linear

- Popular Kernels
  - polynomial

    gamma coef0 degree

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

  - radial basis function

    gamma

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

  - sigmoid

    gamma coef0

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

# Demo

## SVM parameterization

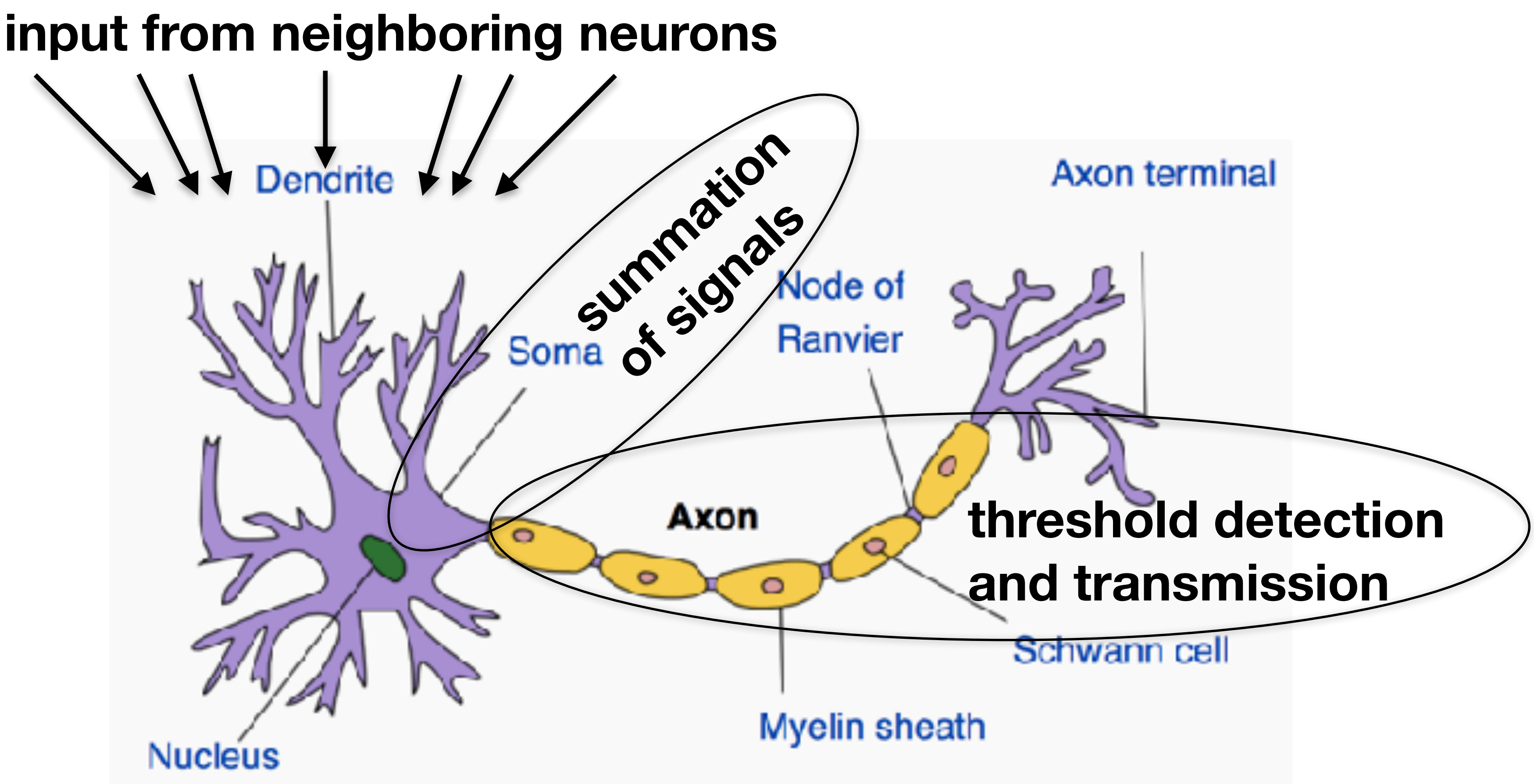
svm_gui.py

The End of SVMs

# A history of Neural Networks

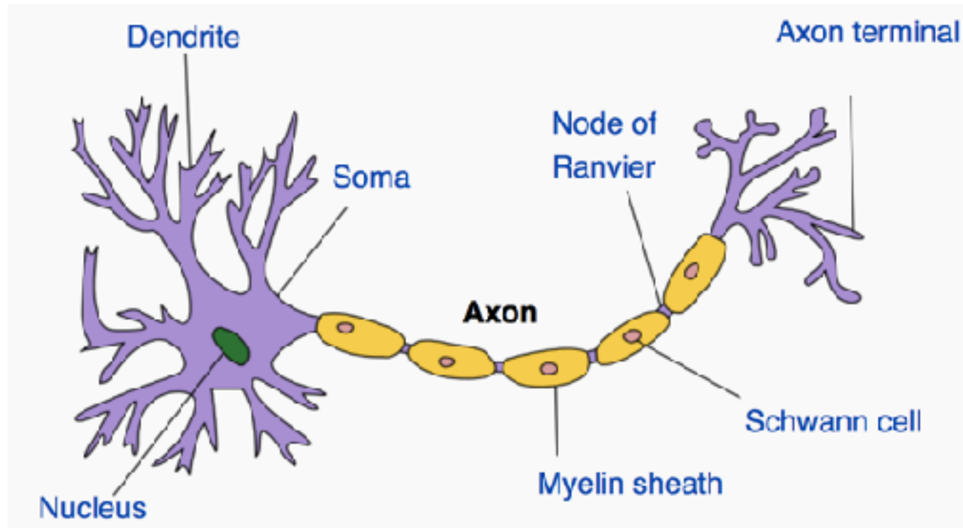
# Neurons

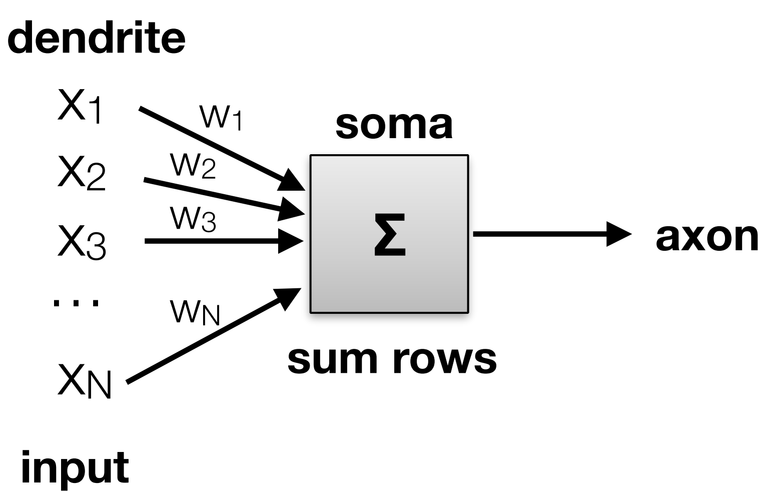
- From biology to modeling:



# Neurons

- McCulloch and Pitts, 1943



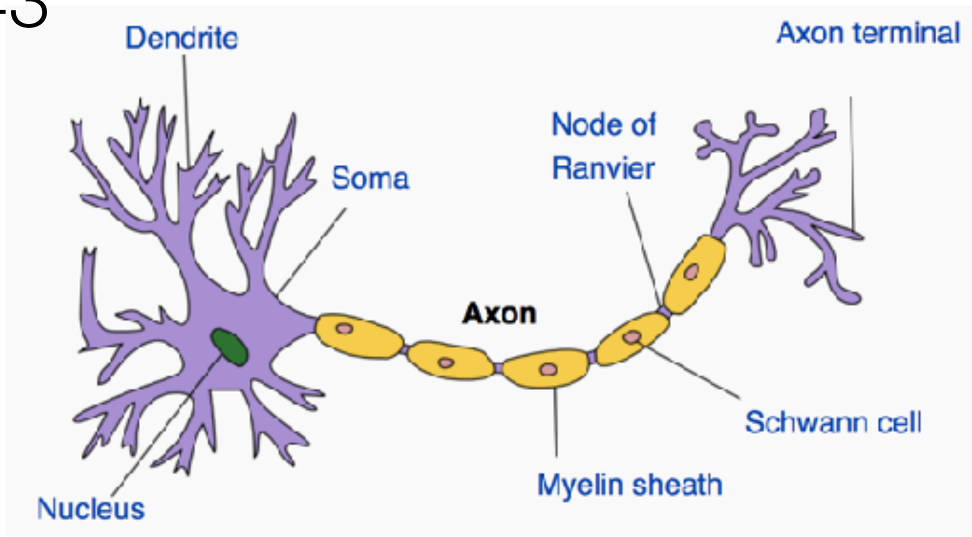


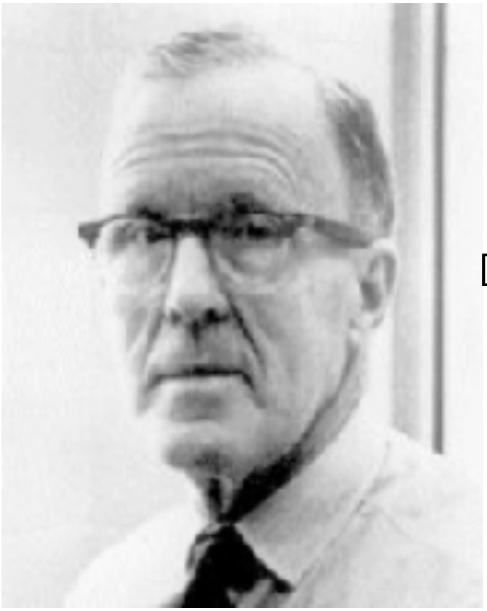
Warren McCulloch


Walter Pitts

# Neurons

- McCulloch and Pitts, 1943
- Donald Hebb, 1949
  - close neurons fire together
  - basis of memory
  - and neural pathways



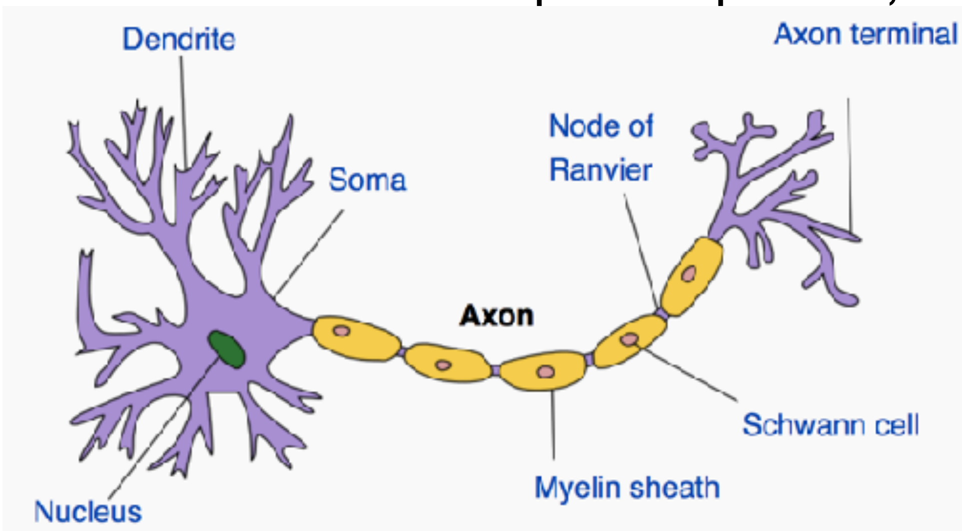
Donald O. Hebb

Warren McCulloch

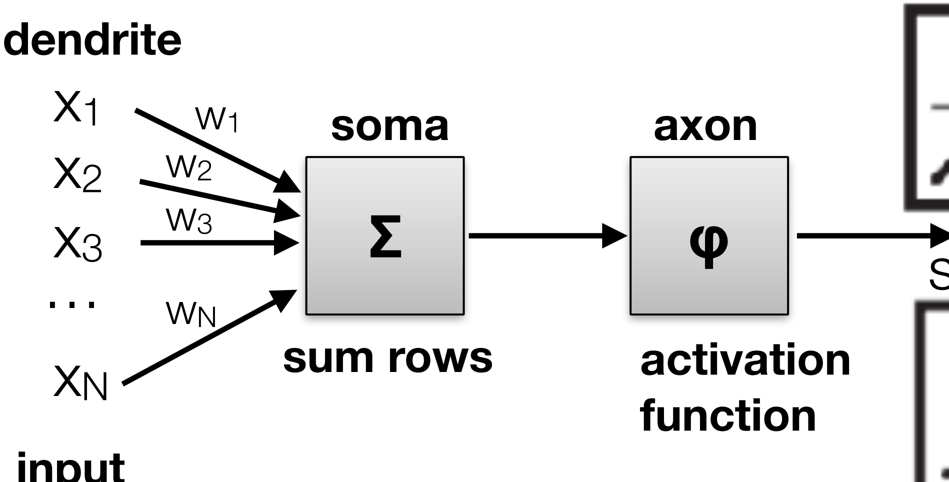
Walter Pitts

# Neurons

- Rosenblatt's perceptron, 1957




Frank Rosenblatt




hard limit

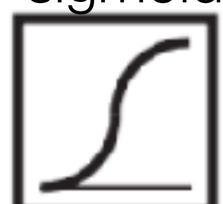
$a=-1 \quad z<0$

$a=1 \quad z>=0$

linear

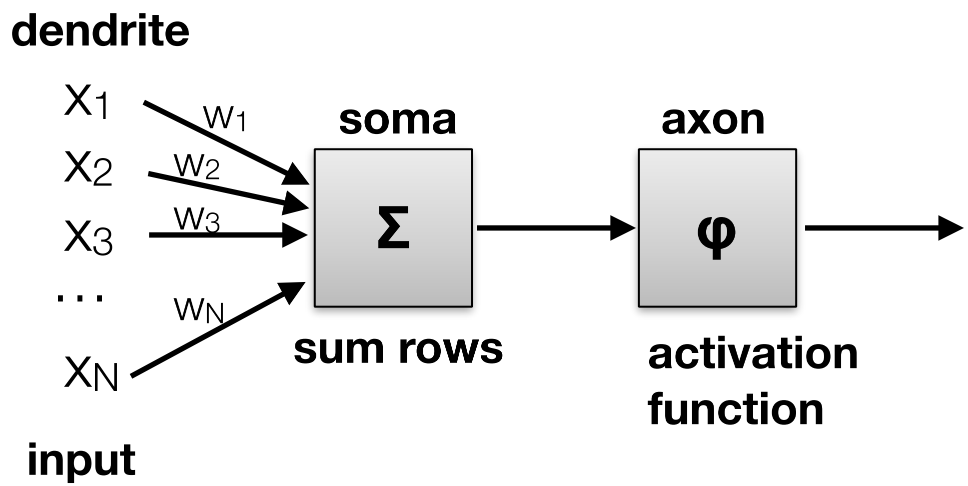
$a=z$

sigmoid

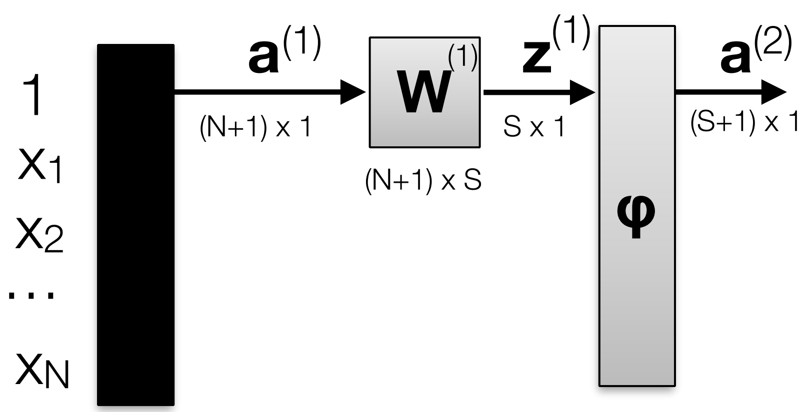


$$a=\frac{1}{1+\exp(-z)}$$

# Neurons




- **Some notation**
  - need bias term
  - matrix representation
  - multiple layers




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

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

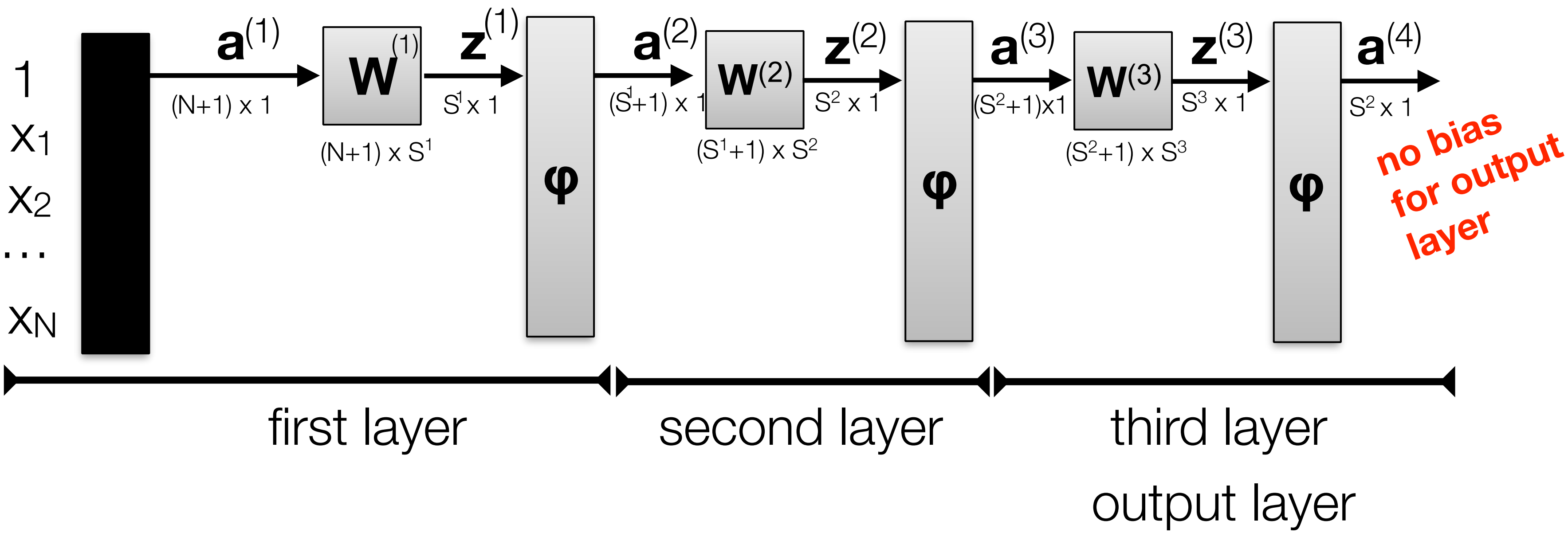
$$\mathbf{W}=\begin{bmatrix} w_{1,2} & w_{1,3} & w_{1,4} & \dots & w_{1,N+1} \\ & & \dots & & \\ w_{S,2} & w_{S,3} & w_{S,4} & \dots & w_{S,N+1} \end{bmatrix}$$

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



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

# Multiple layers notation




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

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

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

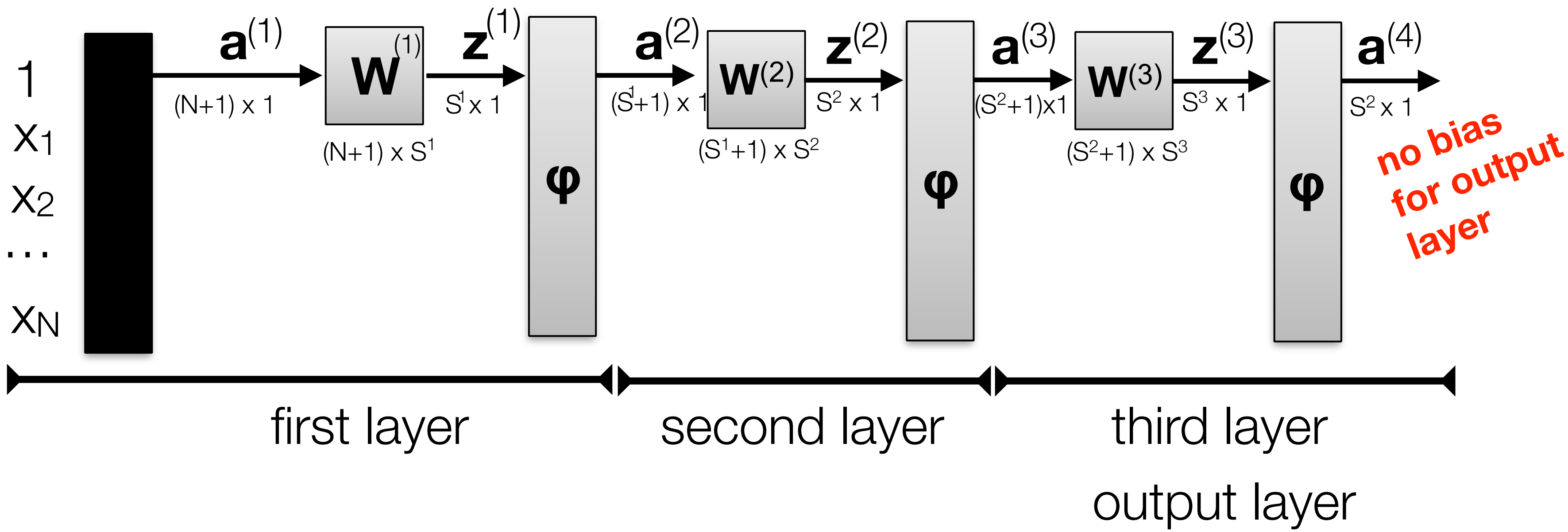
$$\mathbf{W}^{(L)} = \begin{bmatrix} w^{(L)}_{0,2} & w^{(L)}_{0,3} & w^{(L)}_{0,4} & \dots & w^{(L)}_{0,S^L} \\ & & \dots & & \\ w^{(L)}_{S^{L-1},2} & w^{(L)}_{S^{L-1},3} & w^{(L)}_{S^{L-1},4} & \dots & w^{(L)}_{S^{L-1},S^L} \end{bmatrix}$$

***Google Alert:***
you will also see implementations where **W** is a column vector, **w**



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

# Multiple layers notation

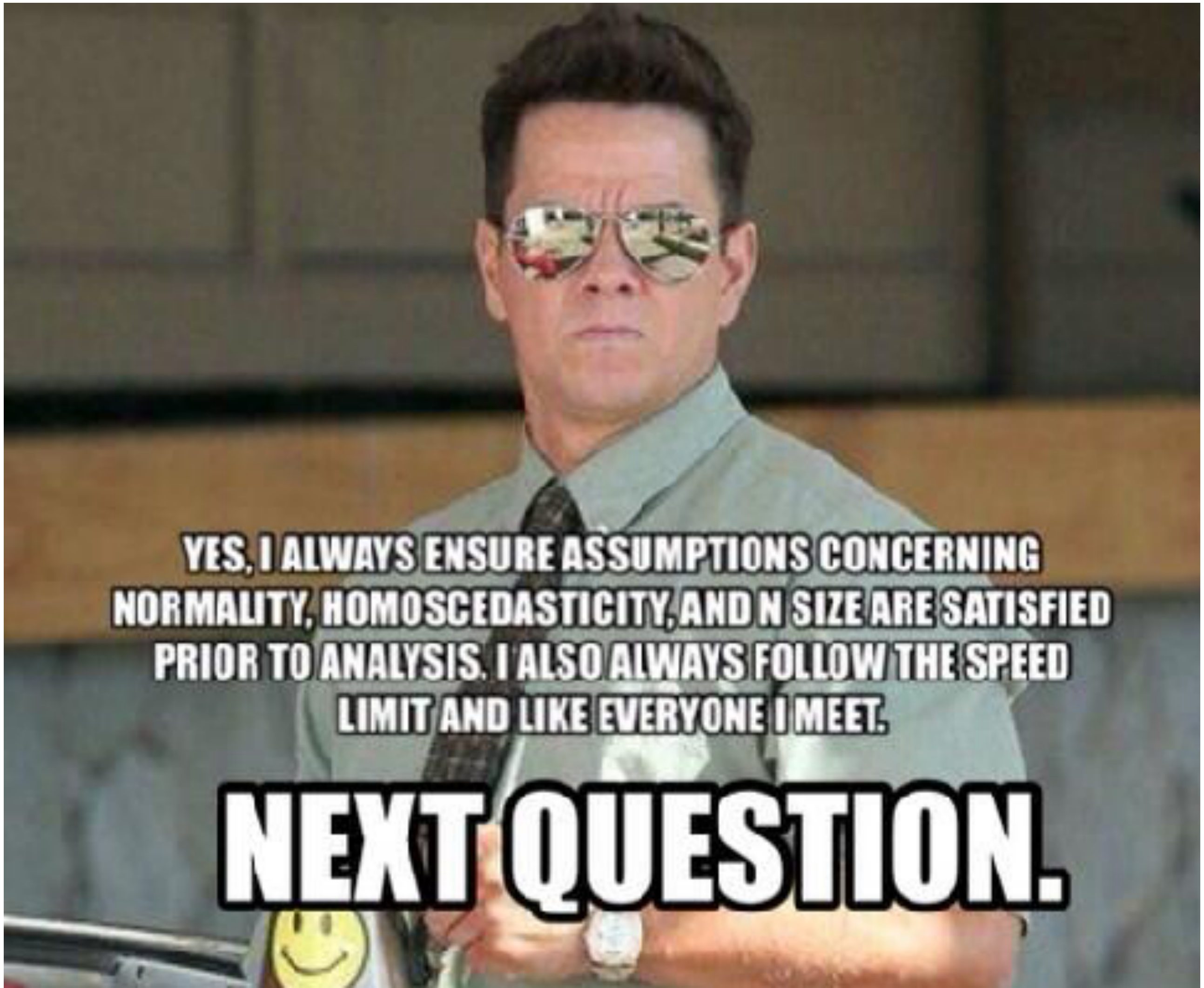



- **Self test**: How many parameters need to be trained in the above network?
  - A. $(N+1) \times S^1 + (S^1+1) \times S^2 + (S^2+1) \times S^3$
  - B. $|\mathbf{W}^{(1)}| + |\mathbf{W}^{(2)}| + |\mathbf{W}^{(3)}|$
  - C. can't determine from diagram
  - D. it depends on the sizes of intermediate variables,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



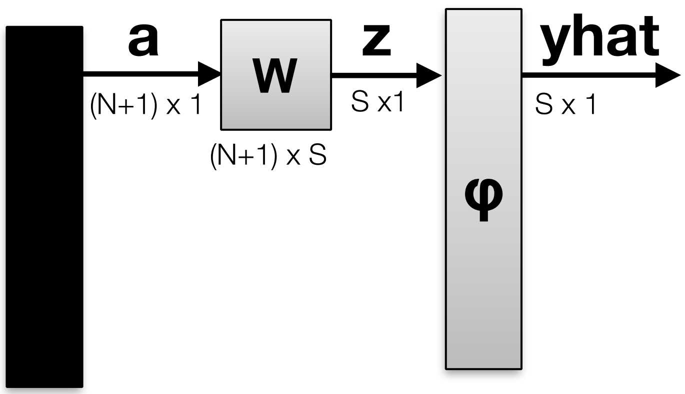
notation adapted from *Neural Network Design*, Hagan, Demuth, Beale, and De Jesus

# Training Neural Network Architectures



# Simple Architectures


- Rosenblatt’s perceptron, 1957




hard limit

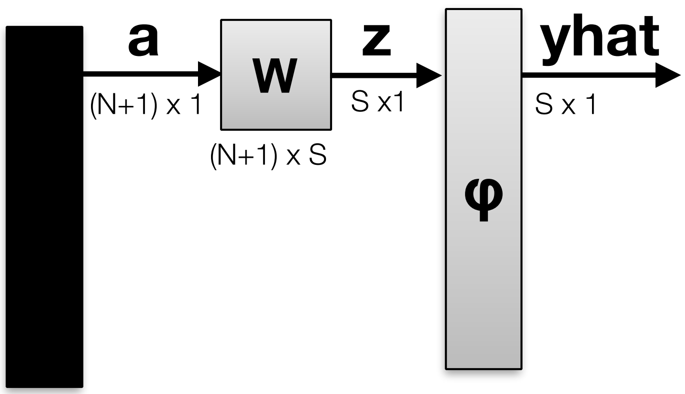
```
a=-1    z< 0
a= 1    z>=0
```

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

A. Can’t determine
B. 2
C. 1
D. N

# Simple Architectures


- Rosenblatt's perceptron, 1957




hard limit

```
a=-1    z< 0
a= 1    z>=0
```

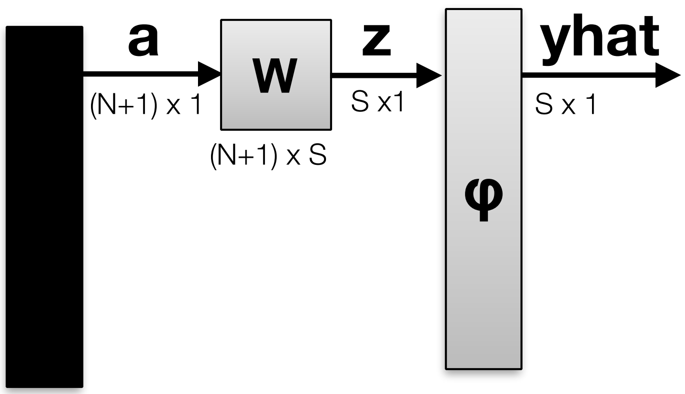
Need objective Function, minimize MSE

$$\sum_{i}^{M}(\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)})^2 \qquad J(W) = \sum_{i}^{M}(\mathbf{y}^{(i)} - \phi(\mathbf{W}\cdot\mathbf{x}^{(i)}))^2$$

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

# Simple Architectures

- Adaline network, Widrow and Hoff, 1960




linear

a=z

Marcian "Ted" Hoff

Bernard Widrow

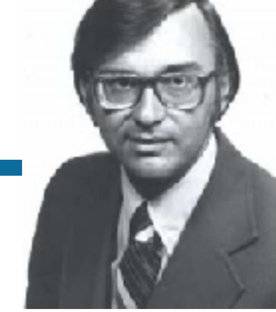
Objective Function, minimize MSE $\sum_{i}^{M}(\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)})^2$

new objective function becomes: $J(W) = \sum_{i}^{M}(\mathbf{y}^{(i)} - \mathbf{W}\cdot\mathbf{x}^{(i)})^2$

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

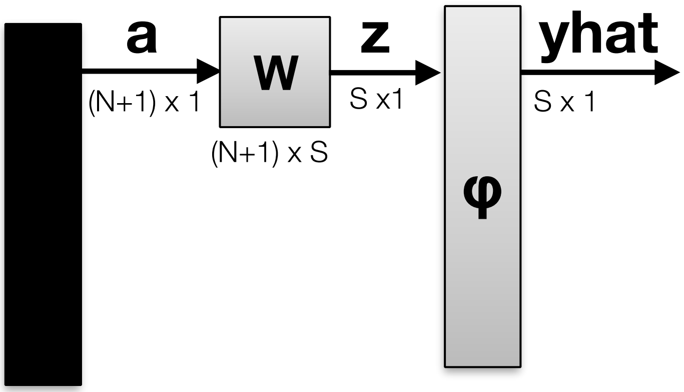
We have been using the **Widrow-Hoff Learning Rule**

# Simple Architectures



Marcian “Ted” Hoff

- Adaline network, Widrow and Hoff, 1960




linear

a=z

Bernard Widrow

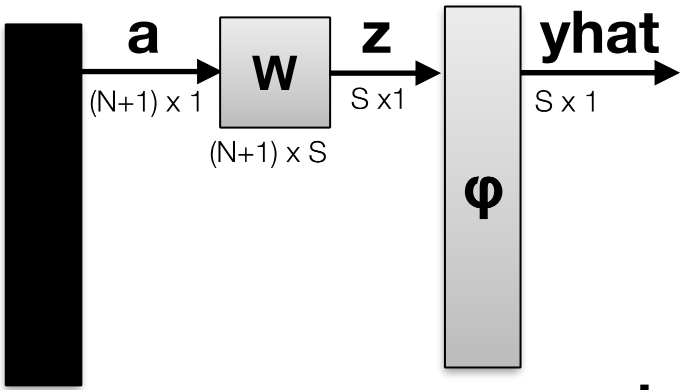
need gradient $\nabla J(\mathbf{W})$ for update equation $\mathbf{W} \leftarrow \mathbf{W} + \eta \nabla J(\mathbf{W})$

For case S=1, this is just solving **linear regression** and **we have already solved this**!

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

for **W**, each row can be solved independently!

# Simple Architectures




$\textbf{yhat}^{(i)} = \boldsymbol{\phi}(\mathbf{W}\mathbf{a}^{(i)})$

output for instance i

$$\textbf{yhat}^{(i)} = \begin{bmatrix} \phi(\mathbf{w}_{row=1}\mathbf{x}^{(i)}) \\ \phi(\mathbf{w}_{row=2}\mathbf{x}^{(i)}) \\ \phi(\mathbf{w}_{row=3}\mathbf{x}^{(i)}) \\ \dots \\ \phi(\mathbf{w}_{row=C}\mathbf{x}^{(i)}) \end{bmatrix}$$

one hot

Each class is independently trained!

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

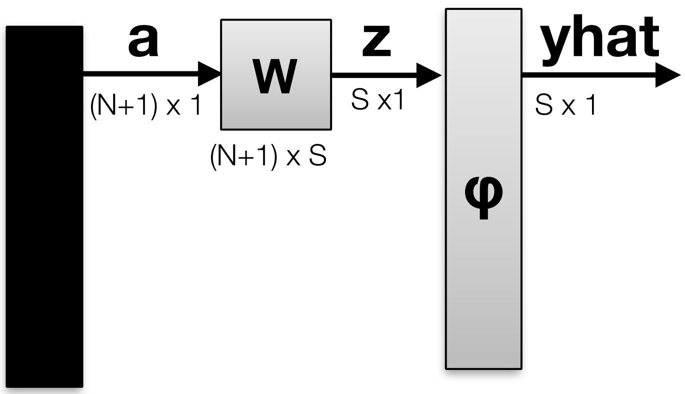
$$\mathbf{w}_{row} \leftarrow \mathbf{w}_{row} + \eta[\mathbf{X} * (\mathbf{y}_{row} - \hat{\mathbf{y}}_{row})]$$

which is one-versus-all!

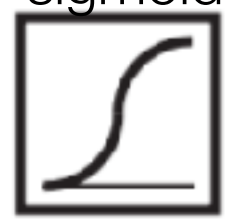
we have already solved this!

# Simple Architectures

- Modern Perceptron network




sigmoid



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

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 solving **logistic regression** and **we have already solved this**!

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

for **W**, each row can be solved independently!

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

which is one-versus-all!

# Simple Architectures: summary

- Adaline network, Widrow and Hoff, 1960
  - linear regression with classifier
- Perceptron
  - *with sigmoid activation*: logistic regression
- One-versus-all implementation is the same as having $\mathbf{w}_{class}$ 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

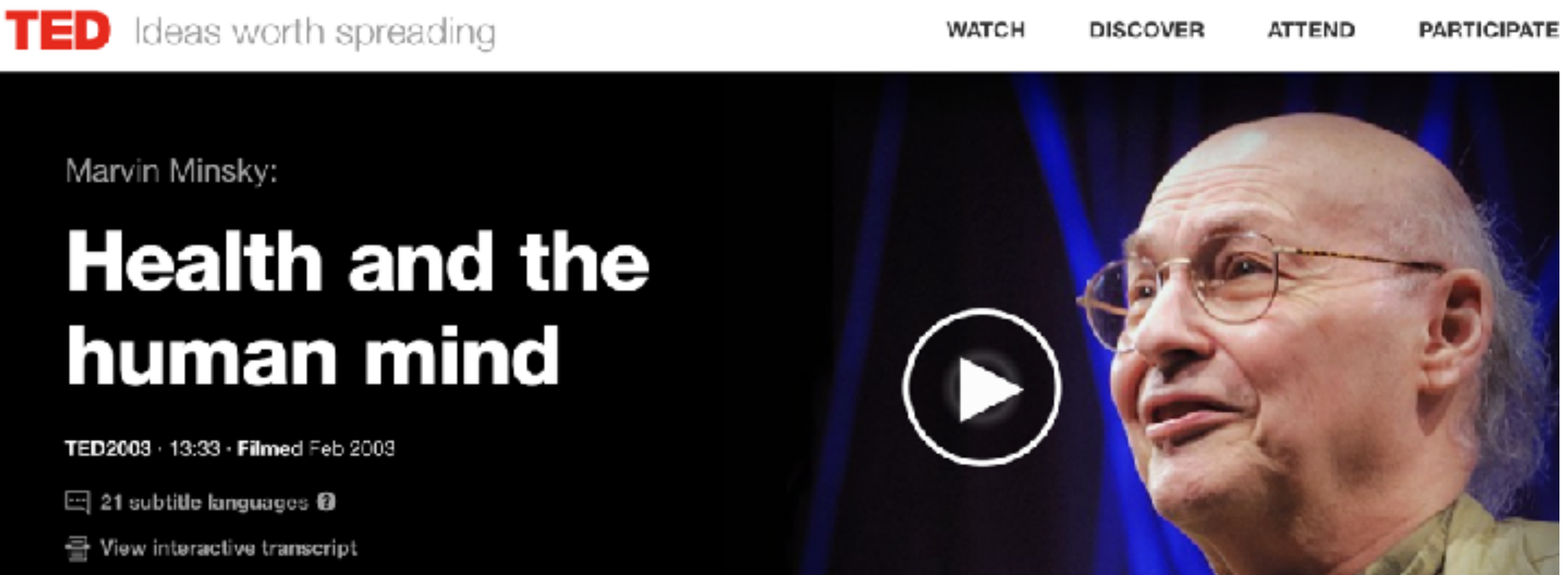
**why were they not used?**

# 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:



# 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**

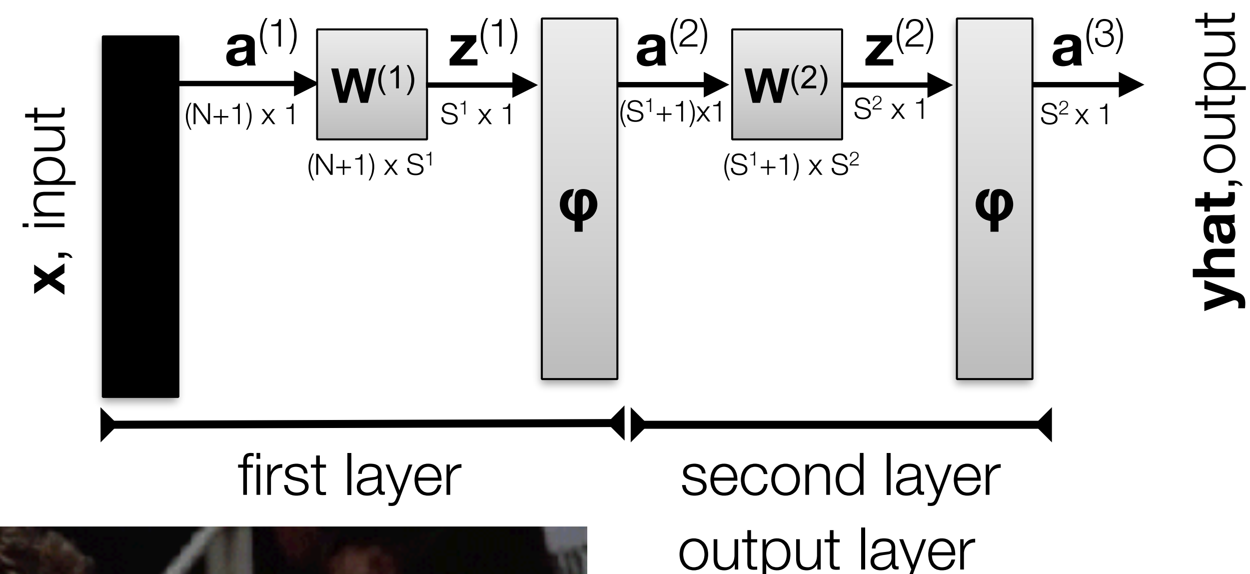
David Rumelhart

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*)

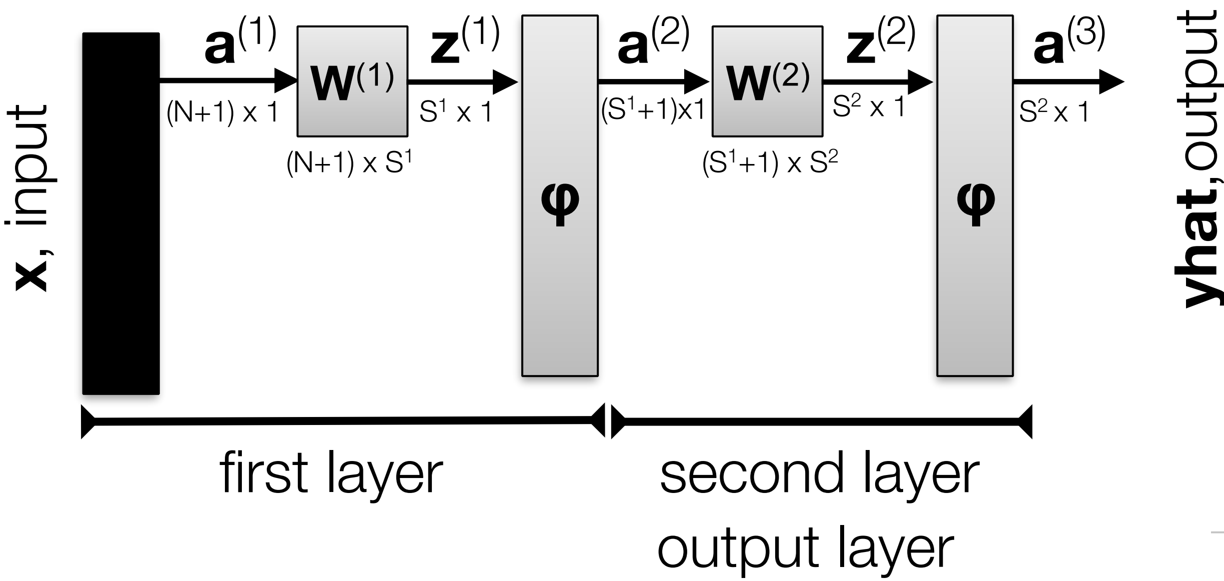




each row of **yhat** is no longer independent of the rows in **W**

# Back propagation

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

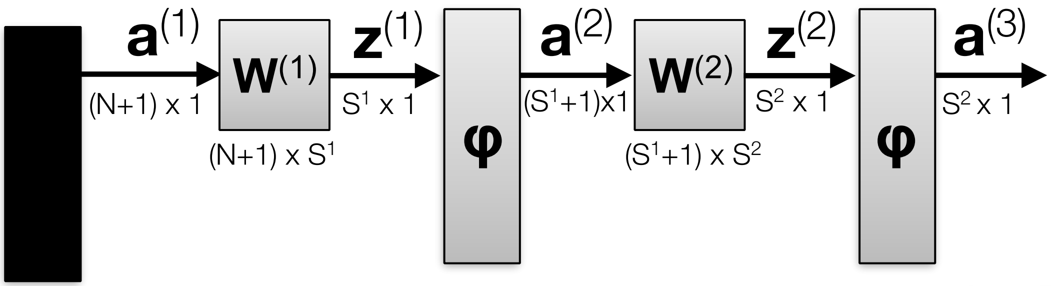



$$J(\mathbf{W}) = \sum_{k}^{M} (\mathbf{y}^{(k)} - \mathbf{a}^{(L)})^2$$

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

# Back propagation

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




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

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

Solve this next time!

# End of Session

- thanks!

**More help on neural networks:**

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=0ahUKEwioprvn27fPAhWMx4MKHYbwDIwQFggeMAA&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

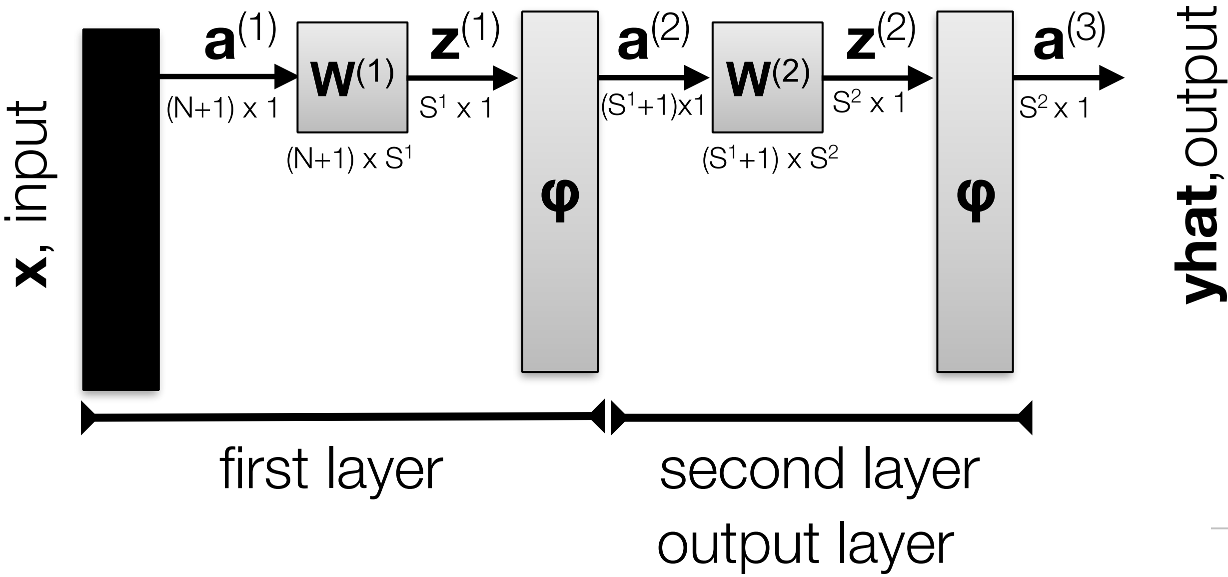
Week Seven B

# Class Logistics and Agenda

- Grades Coming Soon, but slowly
- A2 posted, schedule revised
- Two Week Agenda:
  - *SVM Review*
  - *Neural Networks History*
  - Multi-layer Architectures
  - Programming Multi-layer training
- Next Time: fall break

# Back propagation

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

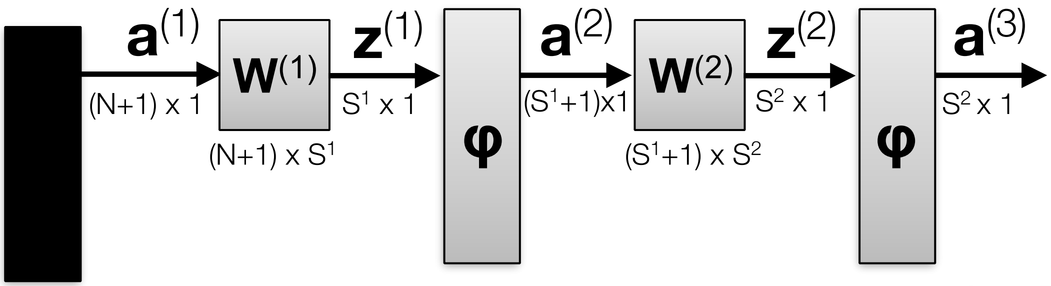



$$J(\mathbf{W}) = \sum_{k}^{M} (\mathbf{y}^{(k)} - \mathbf{a}^{(L)})^2$$

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

# Back propagation

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

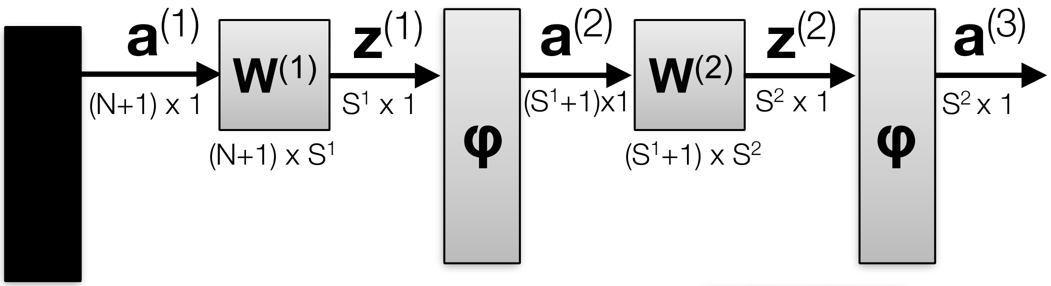



$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta \boxed{\frac{\partial J(\mathbf{W})}{\partial w_{i,j}^{(l)}}}$$

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

# Back propagation

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




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

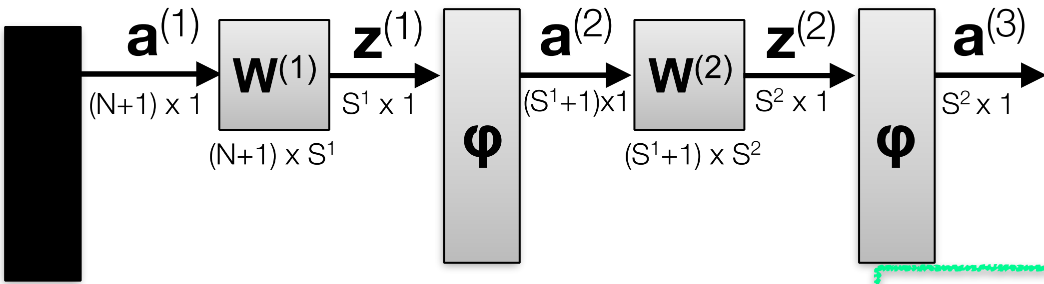
can we use chain rule again?

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

# Back propagation

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

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




can we get this gradient?

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

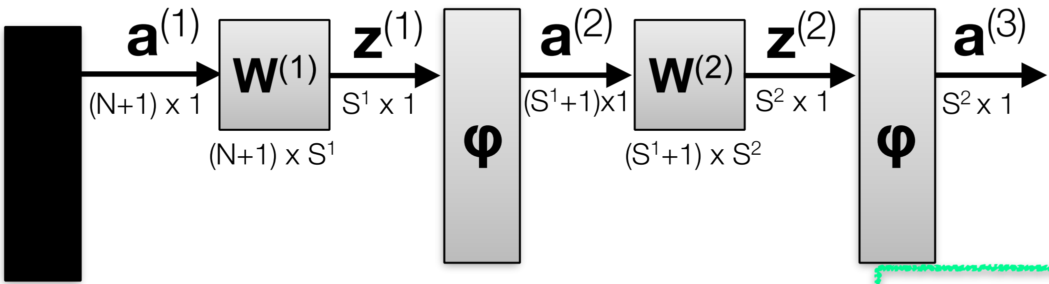


bp-3

# Back propagation

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

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




use chain rule: $\displaystyle \frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}} = \frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l+1)}} \boxed{\frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{z}^{(l)}}}$

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

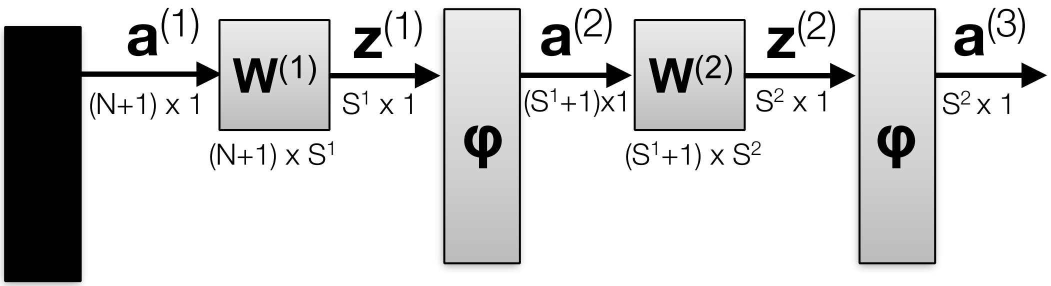
$$\boxed{\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}}} = \mathrm{diag}[\mathbf{a}^{(l+1)} * (1 - \mathbf{a}^{(l+1)})] \cdot \mathbf{W}^{(l+1)} \boxed{\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l+1)}}}$$

recurrence relation

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

# Back propagation

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




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

**one more step**
need last layer gradient:

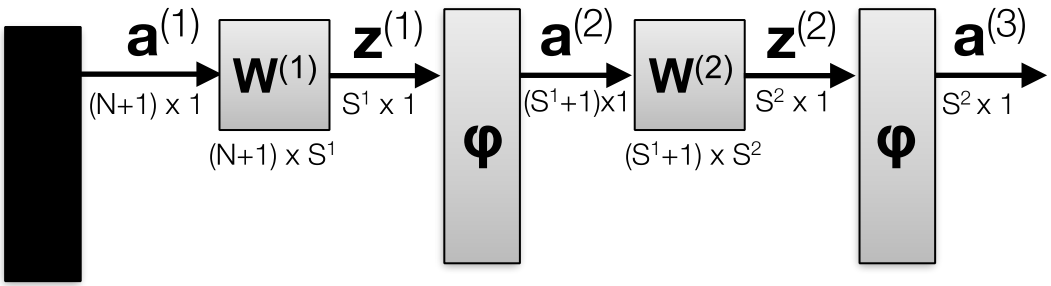
$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(2)}} = \frac{\partial}{\partial \mathbf{z}^{(2)}} (\mathbf{y}^{(k)} - \phi(\mathbf{z}^{(2)}))^2$$

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

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

# Back propagation summary

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




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

1. Forward propagate to get **z**, **a** for all layers
2. Get final layer gradient
3. Update back propagation variables
4. Update each $\mathbf{W}^{(l)}$

for each $\mathbf{y}^{(k)}$

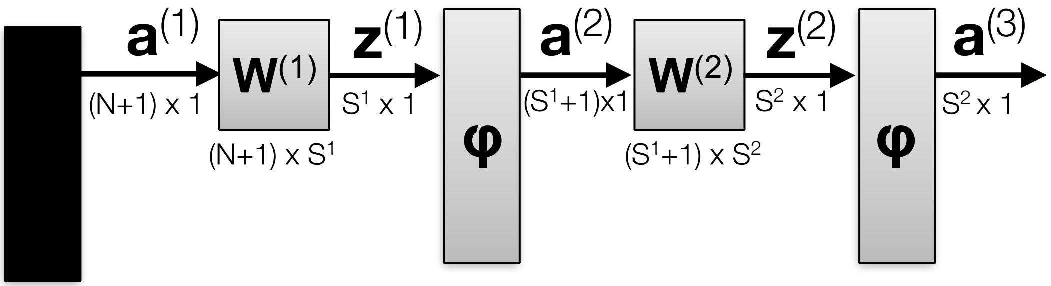
$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(2)}} = -2(\mathbf{y}^{(k)} - \mathbf{a}^{(3)}) * \mathbf{a}^{(3)} * (1 - \mathbf{a}^{(3)})$$

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

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \frac{\partial J(\mathbf{W}^{(l)})}{\partial \mathbf{z}^{(l)}} \cdot \mathbf{a}^{(l)}$$

# Back propagation summary

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




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

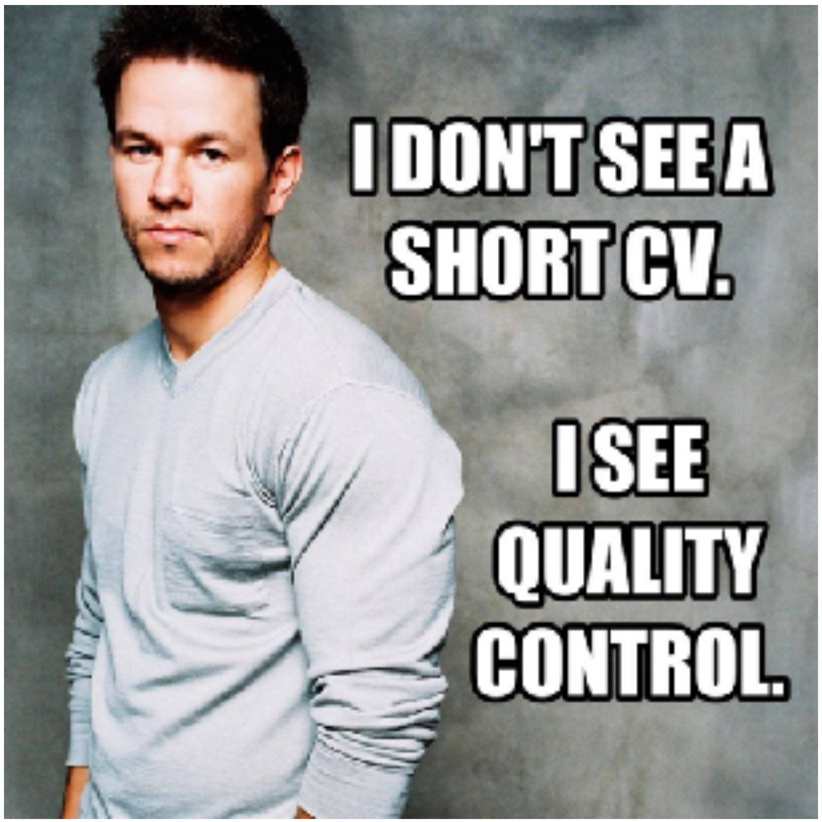
- **Self Test**:
  **True or False**: If we change the cost function, $J(\mathbf{W})$, we only need to update the final layer calculation of the back propagation steps. The remainder of the algorithm is unchanged.
  - A. True
  - B. False

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

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

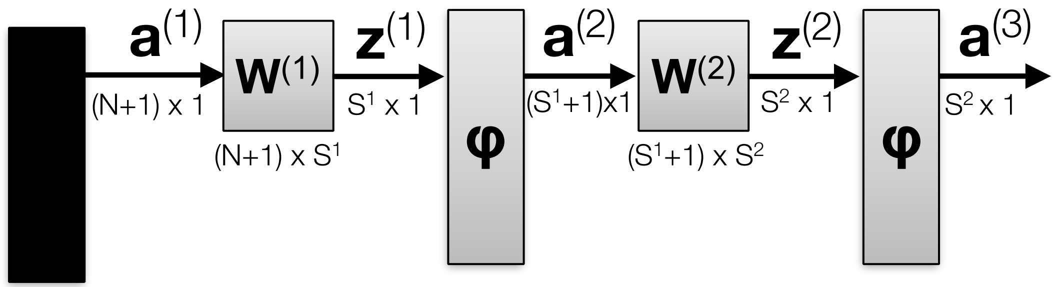
$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \frac{\partial J(\mathbf{W}^{(l)})}{\partial \mathbf{z}^{(l)}} \cdot \mathbf{a}^{(l)}$$

# Programming Multi-layer Neural Networks




Guided Example

# Back propagation implementation




1. Forward propagate to get **z**, **a** for all layers

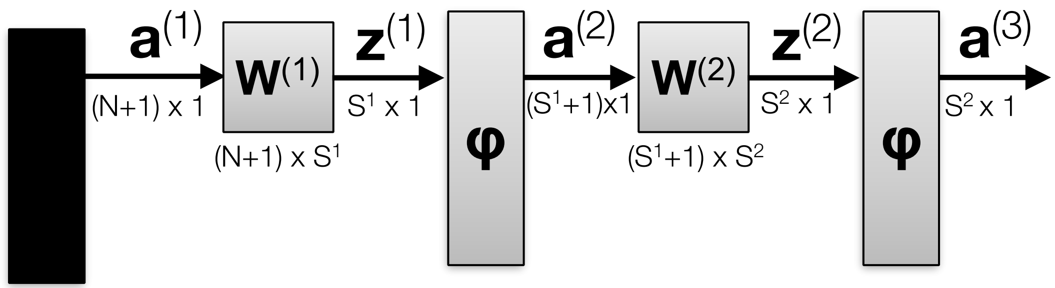
```
# feedforward all instances
A1, Z1, A2, Z2, A3 = self._feedforward(X_data,self.W1,self.W2)
```

```
def _feedforward(self, X, W1, W2):

    A1 = self._add_bias_unit(X, how='column')
    Z1 = W1 @ A1.T
    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




1. Forward propagate to get **z**, **a** for all layers
2. Get final layer gradient
3. Update back propagation variables

for each $\mathbf{y}^{(k)}$

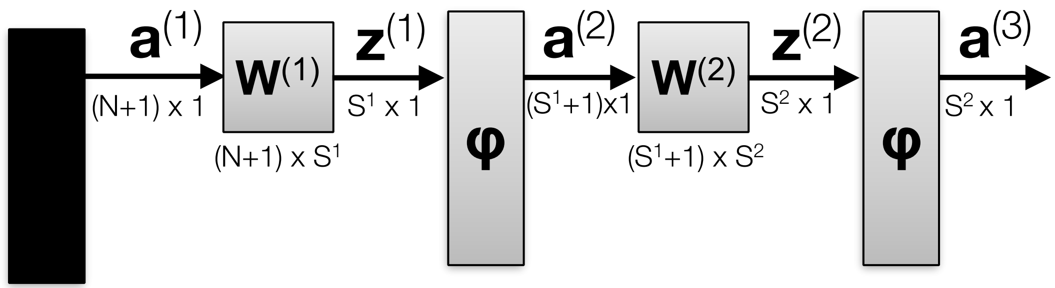
$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(2)}} = -2(\mathbf{y}^{(k)} - \mathbf{a}^{(3)}) * \mathbf{a}^{(3)} * (1 - \mathbf{a}^{(3)})$$

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

**separate** activations for **each** instance

```
# for each instance's activations
for (a1,a2,a3,y) in zip(A1,A2.T,A3.T,Y_enc.T):
    dJ_dz2 = -2*(y - a3)*a3*(1-a3)
    dJ_dz1 = dJ_dz2 @ W2 @ np.diag(a2*(1-a2))
```

# Back propagation implementation




1. Forward propagate to get **z**, **a** for all layers
2. Get final layer gradient
3. Update back propagation variables

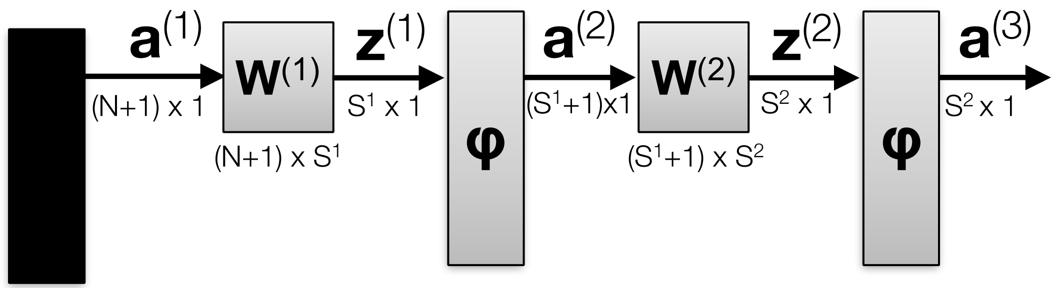
$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \boxed{\frac{\partial J(\mathbf{W}^{(l)})}{\partial \mathbf{z}^{(l)}} \cdot \mathbf{a}^{(l)}}$$

```
grad1 = np.zeros(W1.shape)
grad2 = np.zeros(W2.shape)

# for each instance's activations
for (a1,a2,a3,y) in zip(A1,A2.T,A3.T,Y_enc.T):
    dJ_dz2 = -2*(y - a3)*a3*(1-a3)
    dJ_dz1 = dJ_dz2 @ W2 @ np.diag(a2*(1-a2))

    grad2 += dJ_dz2[:,np.newaxis]  @ a2[np.newaxis,:]
    grad1 += dJ_dz1[1:,np.newaxis] @ a1[np.newaxis,:] # don't incorporate bias
```

# Back propagation implementation




1. Forward propagate to get **z**, **a** for all layers
2. Get final layer gradient
3. Update back propagation variables
4. Update each $\mathbf{W}^{(l)}$

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \frac{\partial J(\mathbf{W}^{(l)})}{\partial \mathbf{z}^{(l)}} \cdot \mathbf{a}^{(l)}$$

```
# 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
```

putting it all together

# Demo

## Two Layer Perceptron

with regularization
vectorization

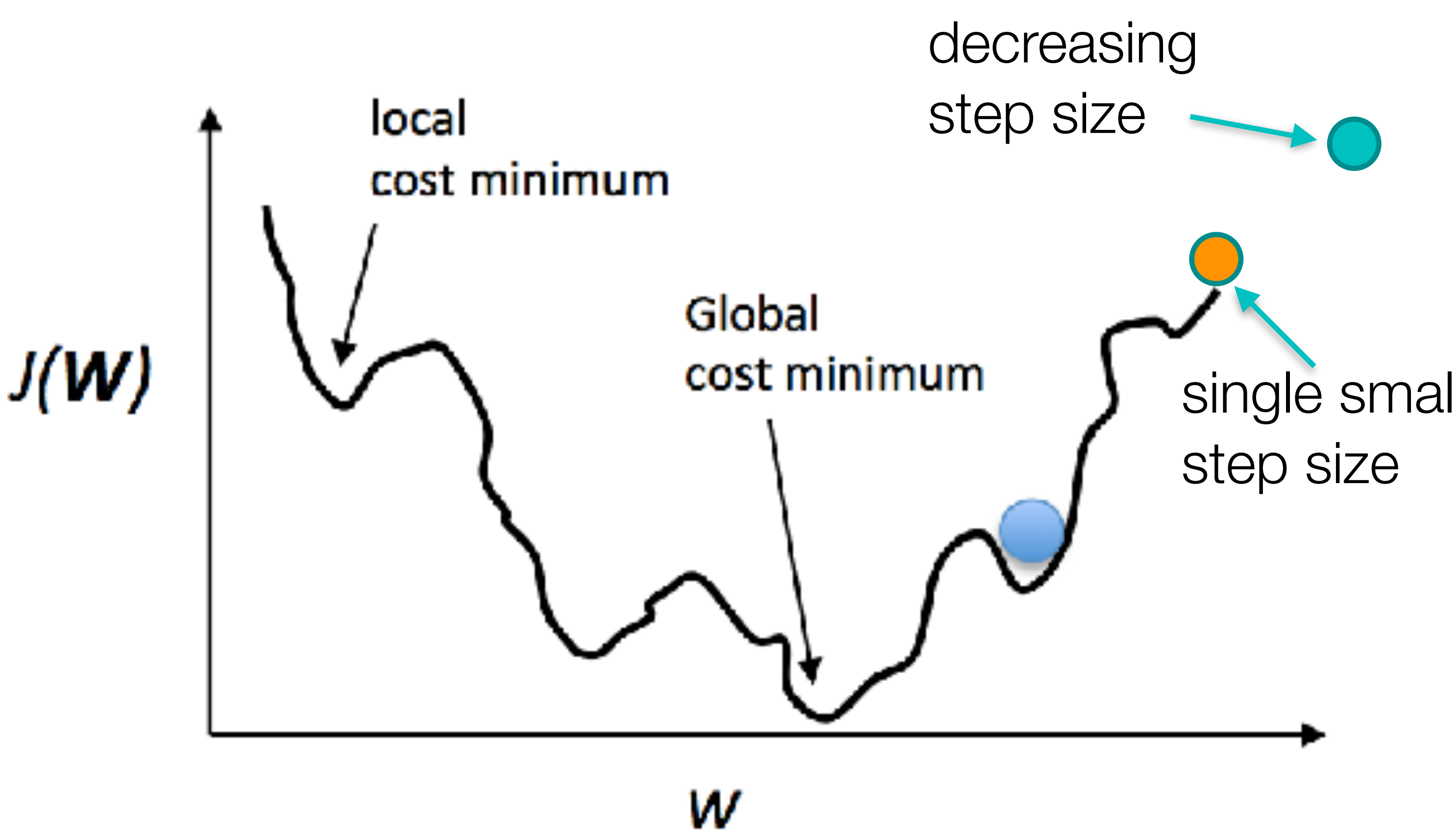
# Problems with Advanced Architectures

- 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

**Self Test**:

# Problems with Advanced Architectures

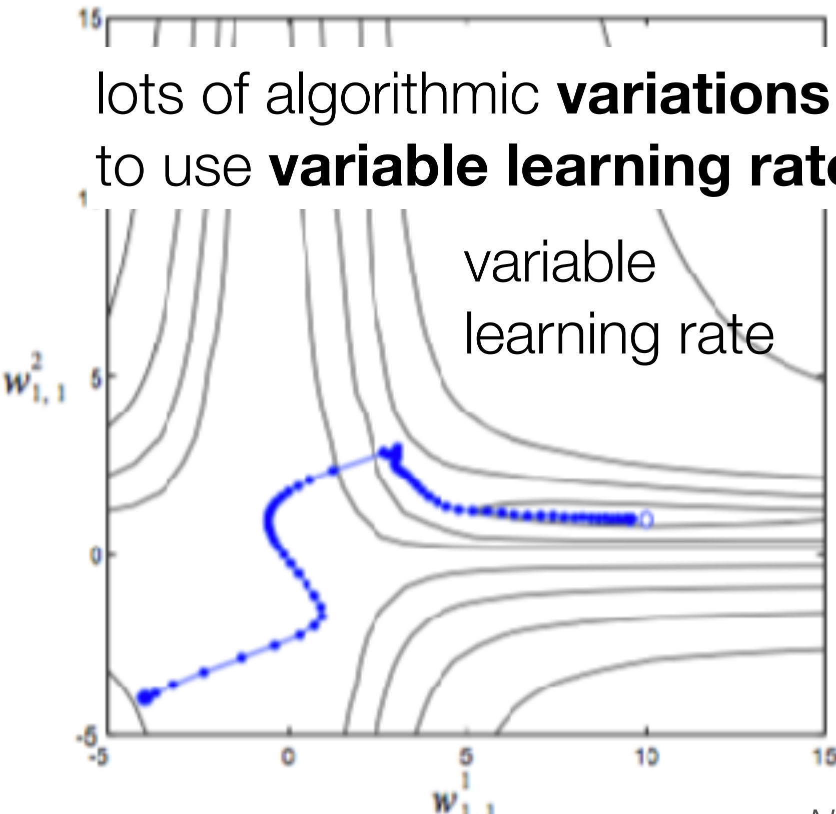
- Space is no longer convex
  - **One solution**:
    - start with large step size
    - "cool down" by decreasing step size for higher iterations

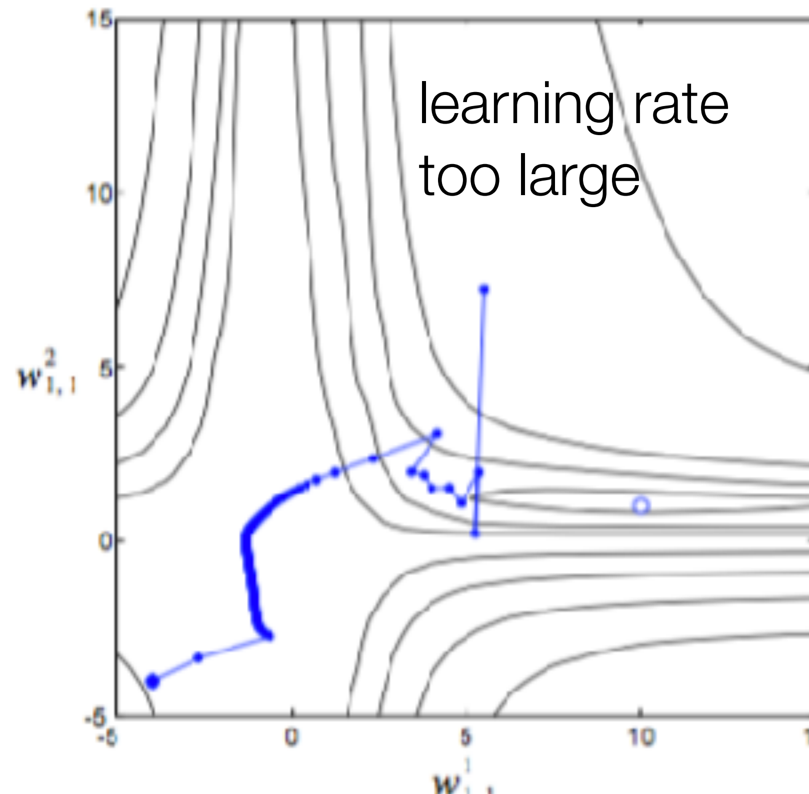


# Problems with Advanced Architectures

- 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**




*Neural Network Design*, Hagan, Demuth, Beale, and De Jesus

# Demo

## Two Layer Perceptron

**comparison**:

mini-batch
- momentum
- decreased learning

L-BFGS