Lecture Notes for Machine Learning in Python

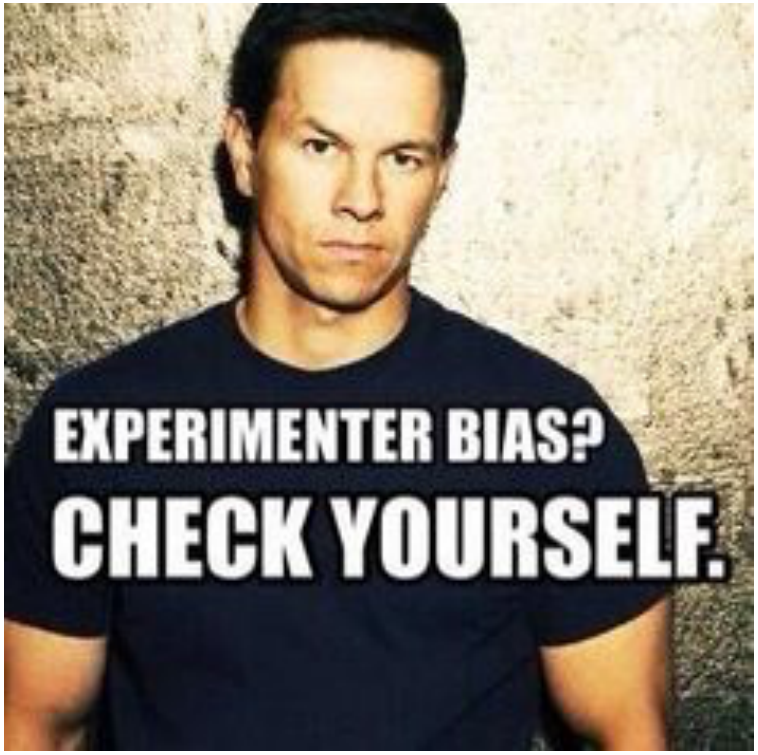
Professor Eric Larson

SVM Review and Neural Network Notation

Class Logistics and Agenda

- Logistics:
  - Nothing due this week!
  - Lab 4 due the week after spring break
  - ICA next time (back propagation)
  - ICA over the break (validation)
- Multi Week Agenda:
  - SVM Review
  - Neural Networks History, up to 1980
  - 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
  - **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

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

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

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

SVMs Summary: Linear

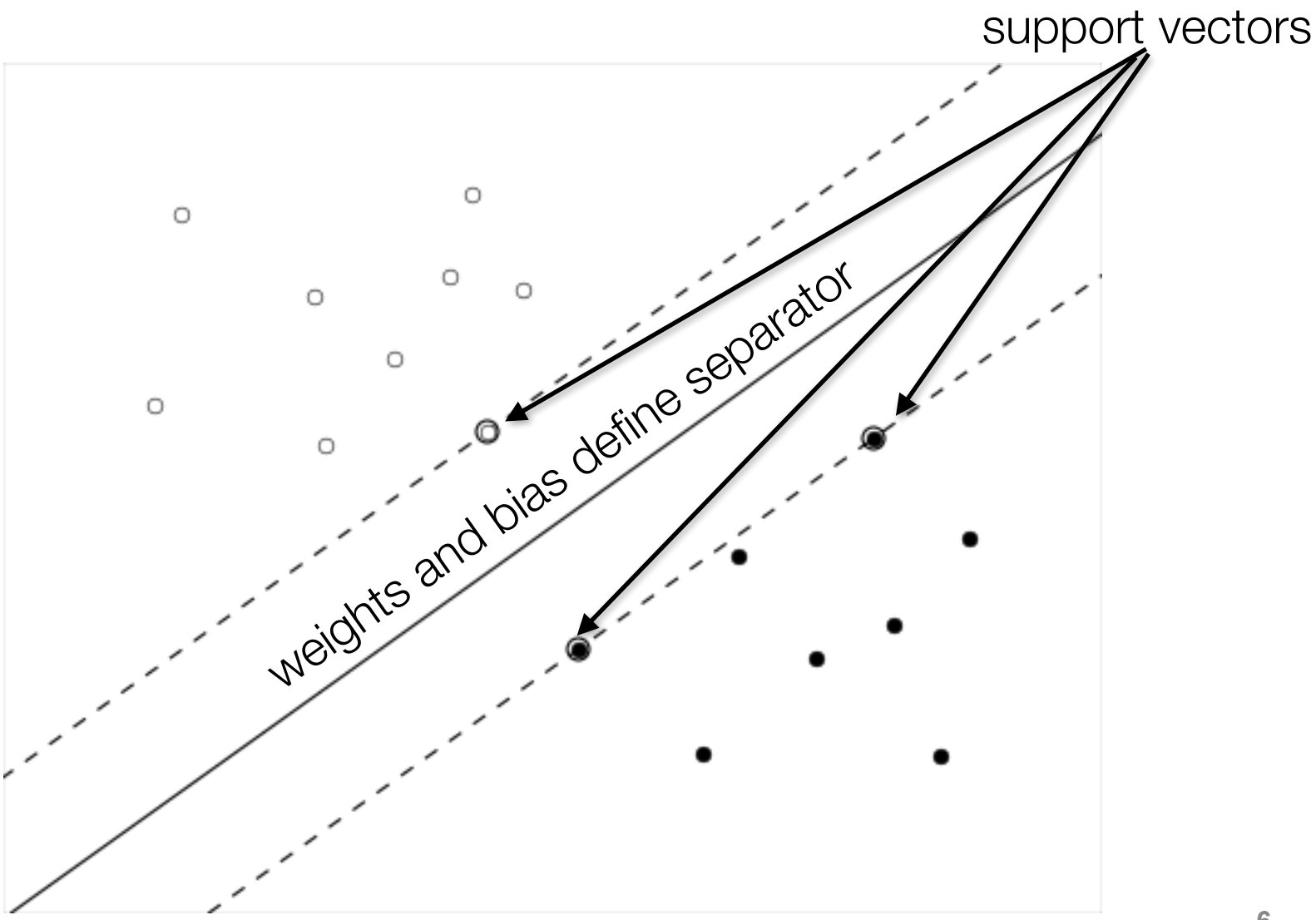
- Linear SVMs
  - Architecture near identical to logistic regression, except:
    - maximize margin
    - constrained optimization
  - Trained Parameters:
    - bias and weights for each class
    - support vectors chosen for margin calculation
    - slack variables for inseparable cases

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

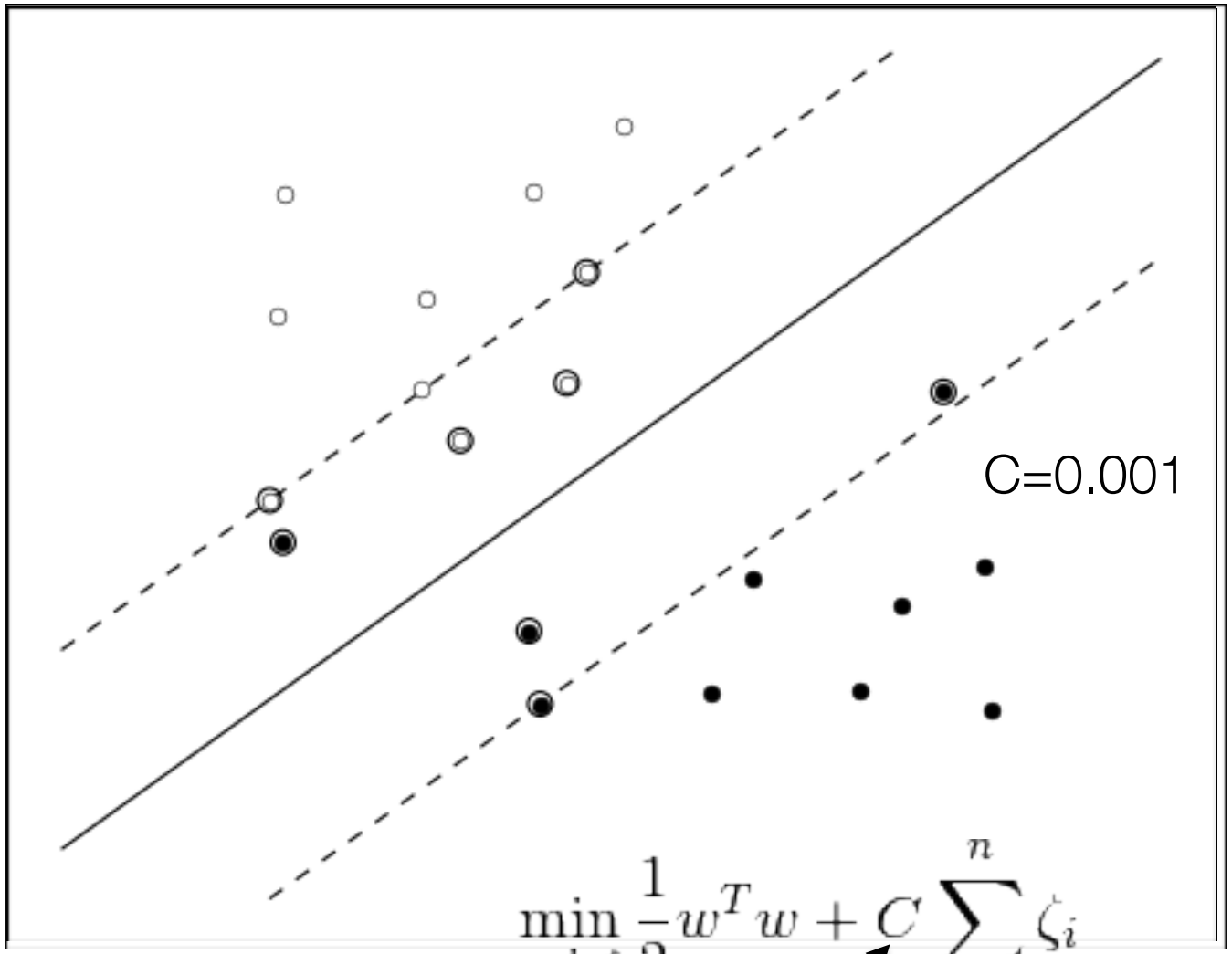
$$\text{to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i,$$

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

SVMs Summary: Linear



SVMs Summary: Linear



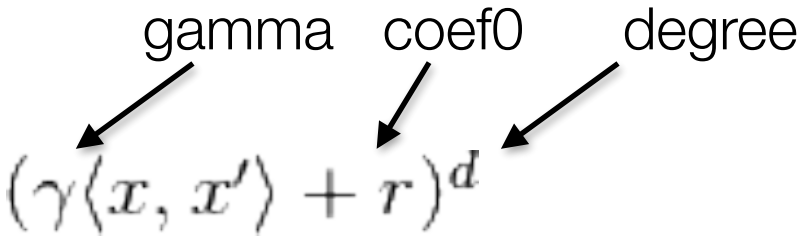
tradeoff margin and accuracy

SVMs Summary: non-linear

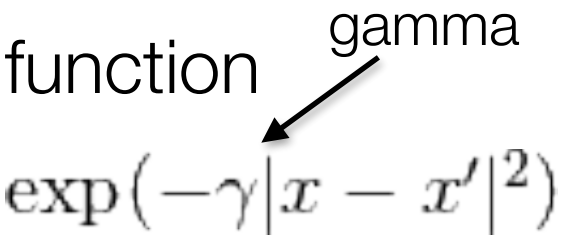
- Non-linear SVMs
  - Architecture not like logistic regression
    - kernels == high dimensional dot-product
    - impossible to store weights
      - use kernel trick so no need to store them!
  - Parameters
    - biases?
    - selected support vectors? (non-zero alphas)
    - slack variables?
   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

SVMs Summary: non-linear

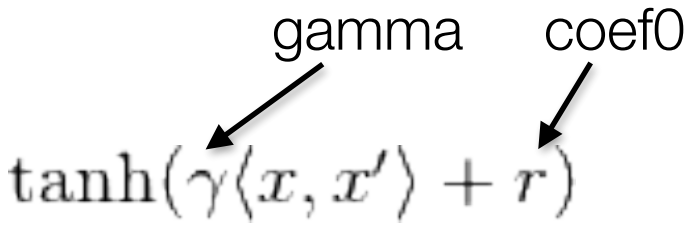
- Popular Kernels
  - polynomial



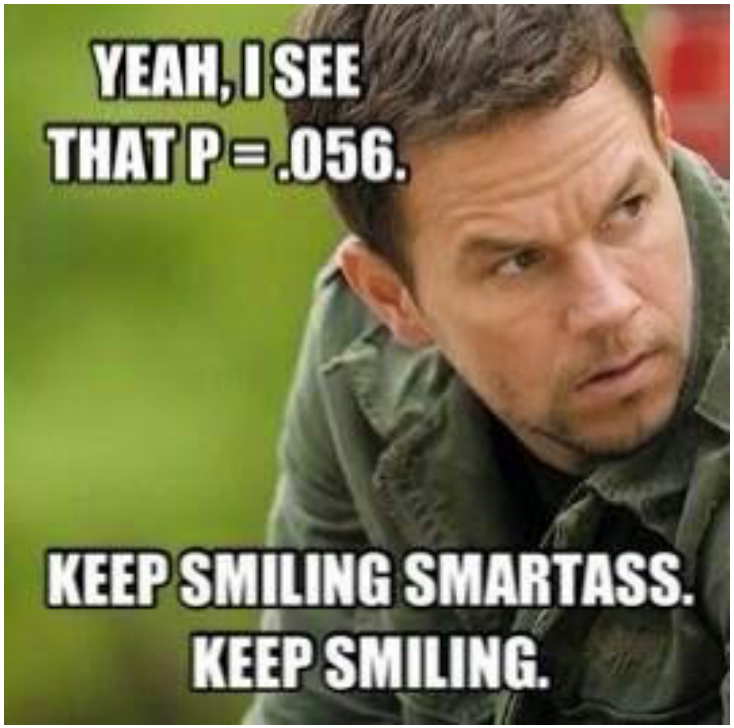
  - radial basis function



  - sigmoid

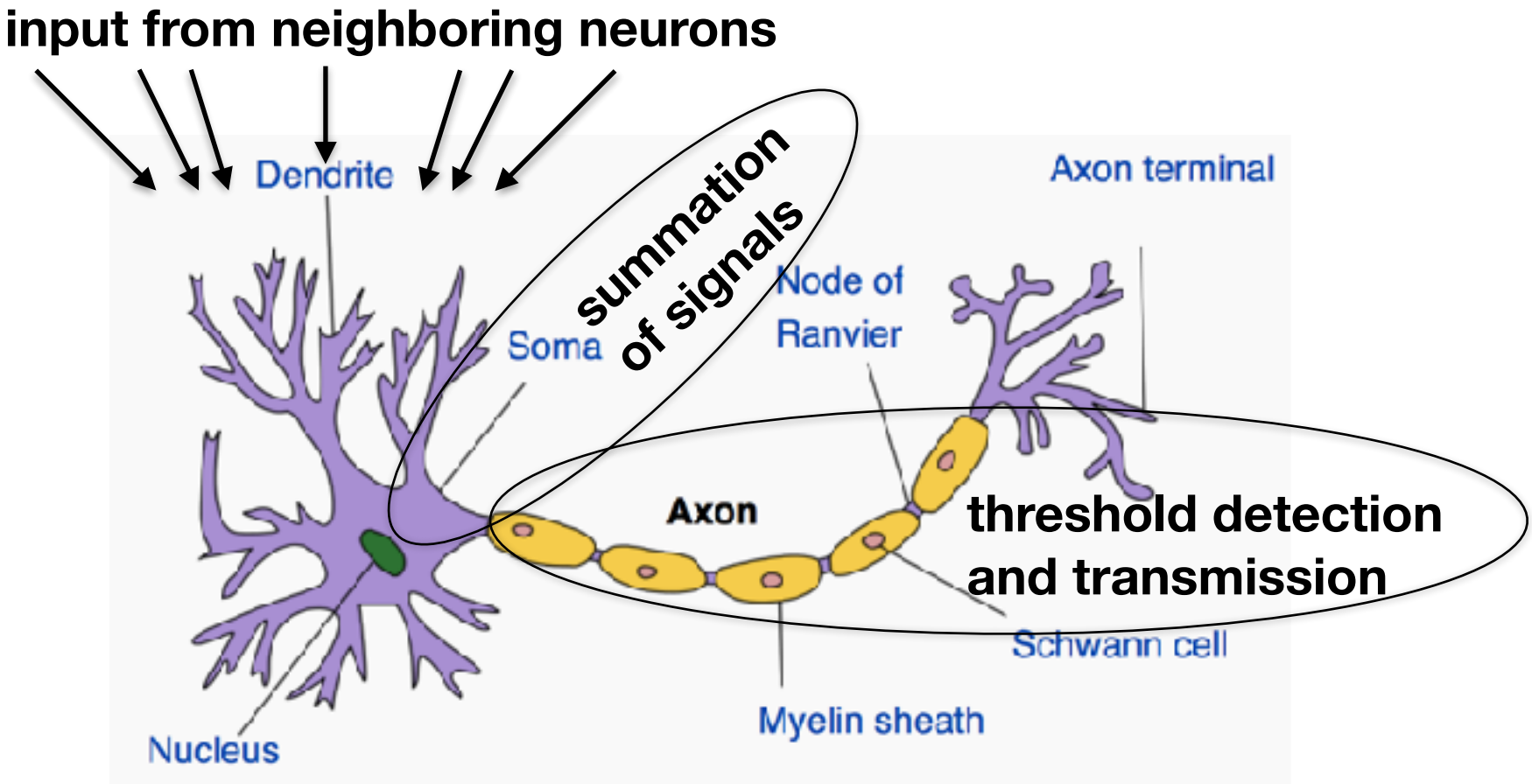


A history of Neural Networks



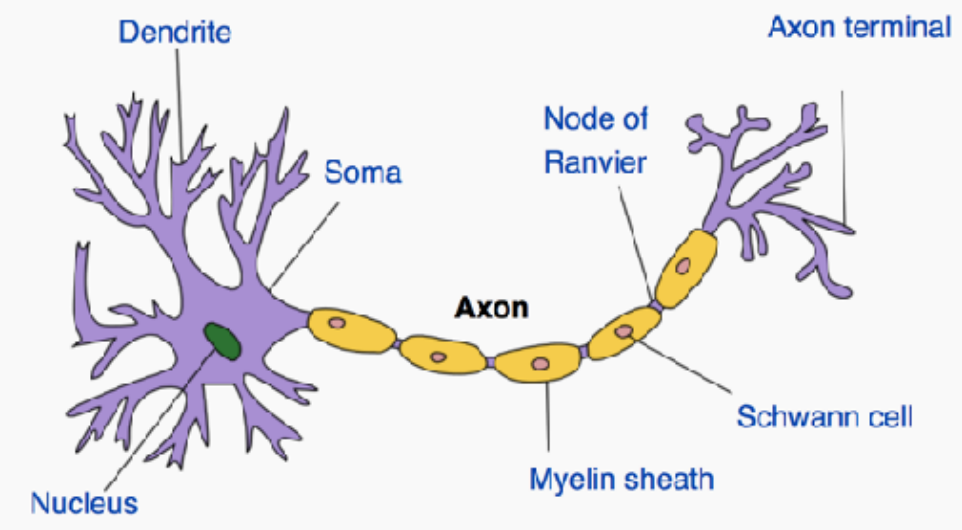
Neurons

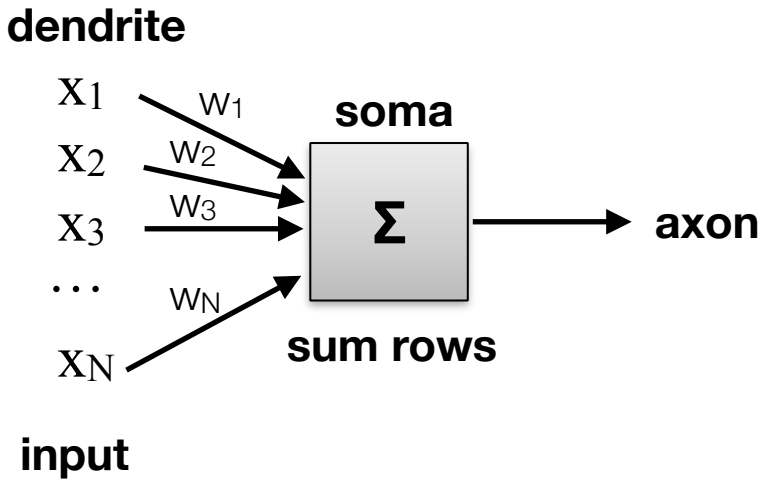
- From biology to modeling:



Neurons

- McCulloch and Pitts, 1943





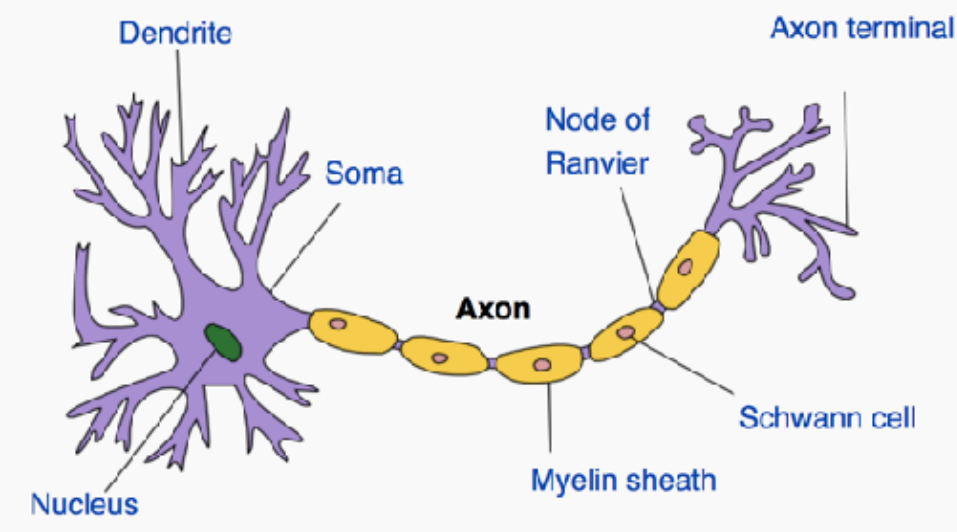
logic gates of the mind

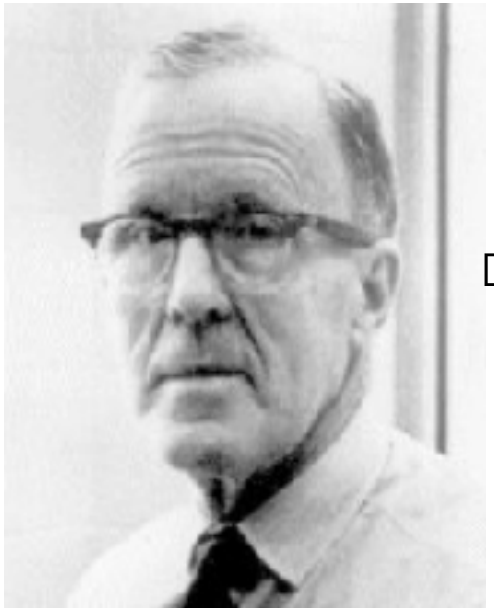
Warren McCulloch

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





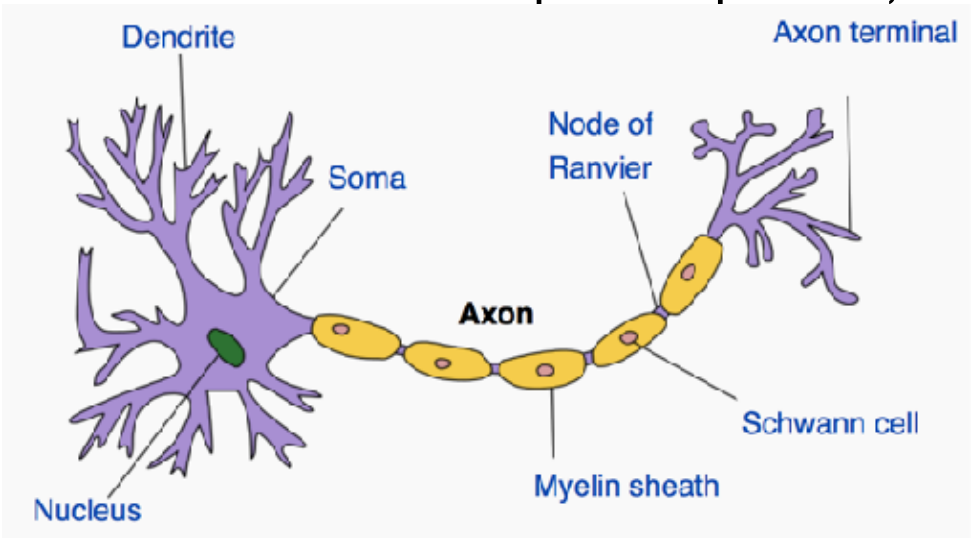
Donald O. Hebb

Warren McCulloch

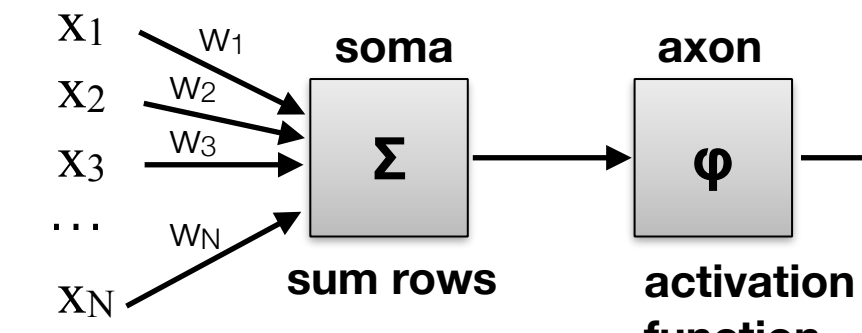
Walter Pitts

Neurons

- Rosenblatt's perceptron, 1957



Frank Rosenblatt



hard limit

a=−1 z< 0
a= 1 z>=0

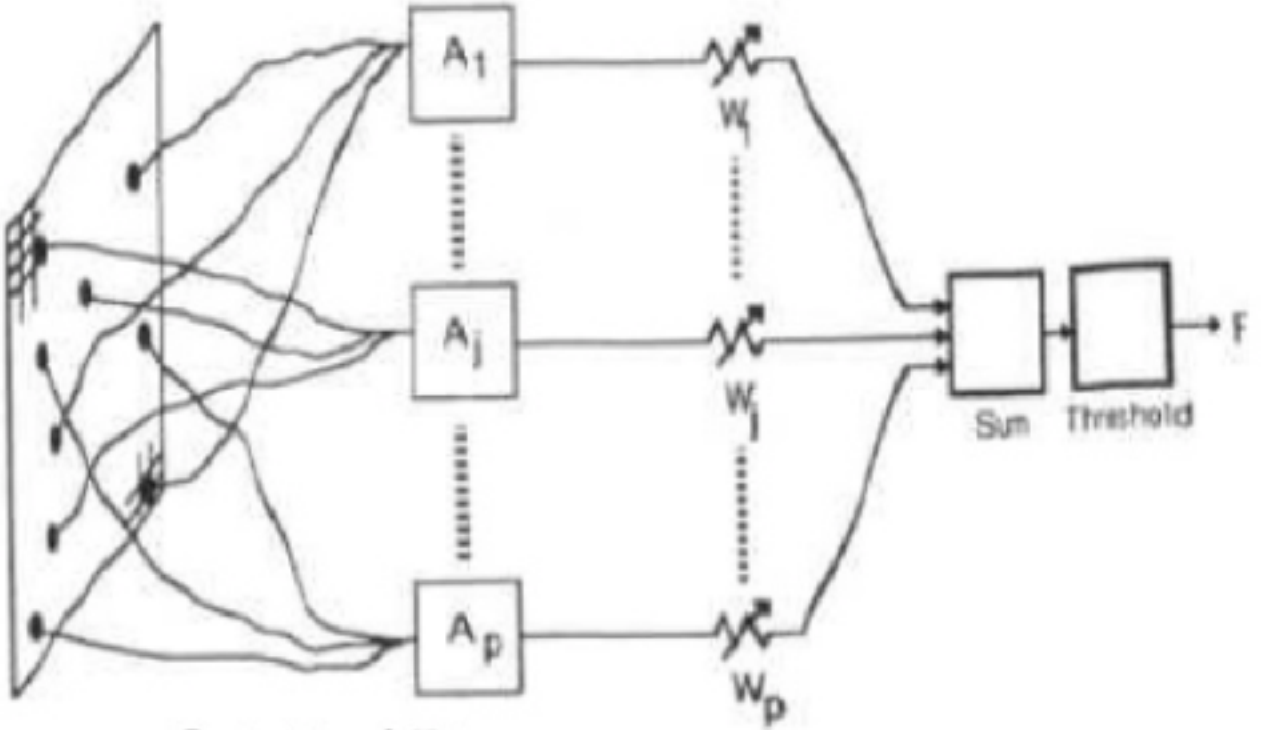
linear

a=z

sigmoid

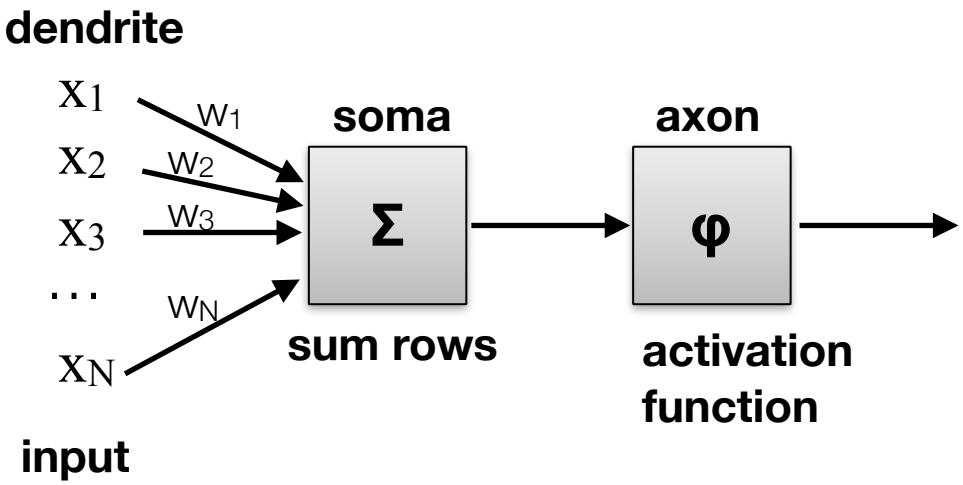
$$a=\frac{1}{1+\exp(-z)}$$

The Mark 1 I

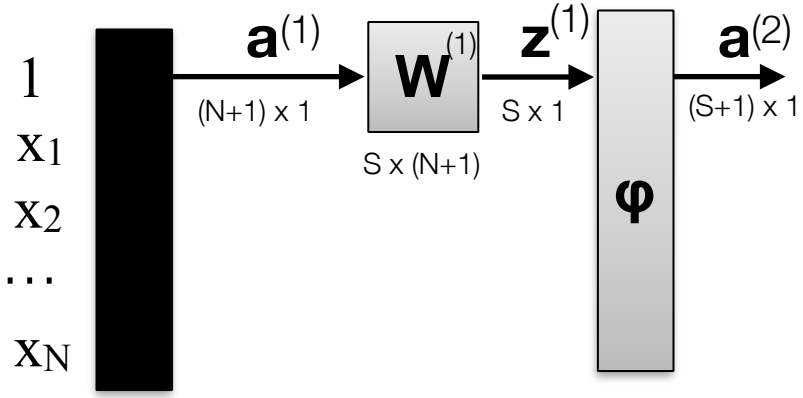


Perceptron Learning Rule:
~Stochastic Gradient Descent

Neurons



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



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

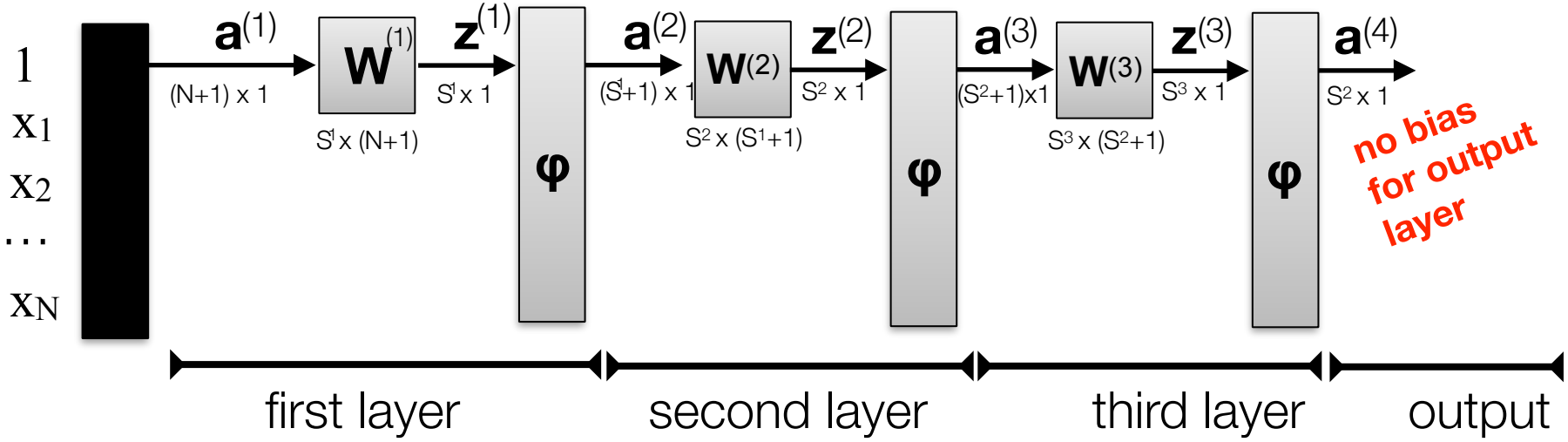
$\mathbf{z}=\mathbf{W}\mathbf{a}^{(1)}$

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

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

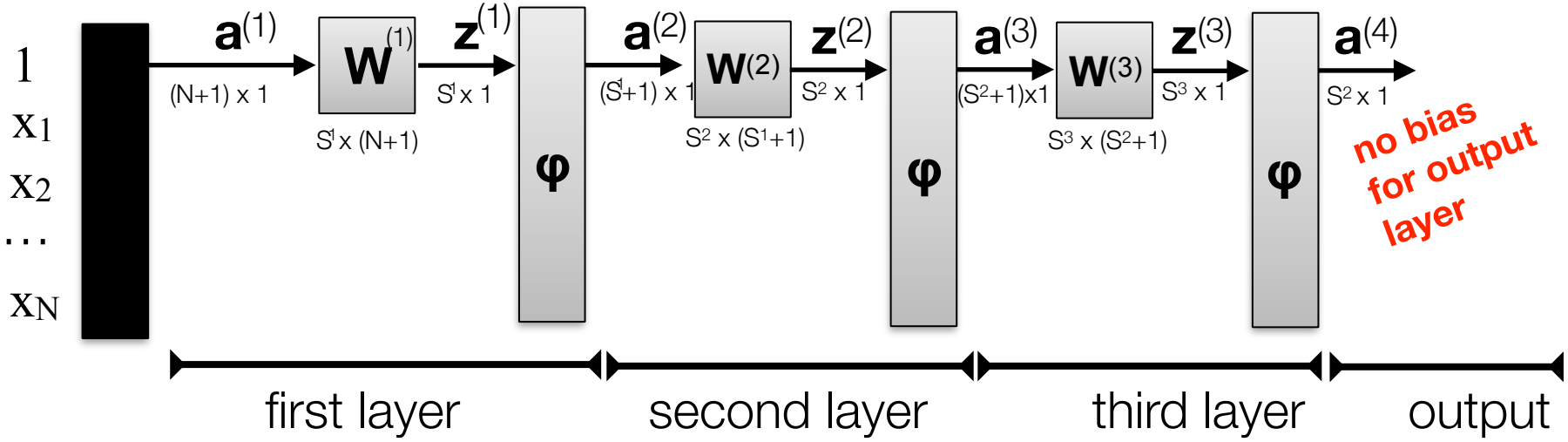
$\mathbf{z}^{(L)} = \mathbf{W}^{(L)}\mathbf{a}^{(L)}$

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

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

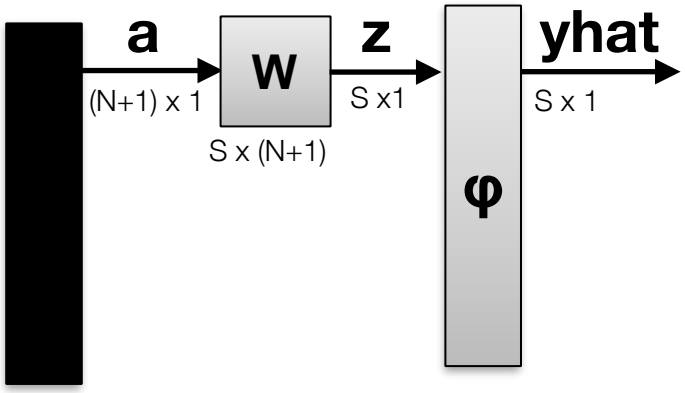


- **Self test**: How many parameters need to be trained in the above network?
  - A. $[(N+1) \times S^1] + [(S^1+1) \times S^2] + [(S^2+1) \times S^3]$
  - B. $|\mathbf{W}^{(1)}| + |\mathbf{W}^{(2)}| + |\mathbf{W}^{(3)}|$
  - C. can't determine from diagram
  - D. it depends on the sizes of intermediate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

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

Simple Architectures

- Rosenblatt's perceptron, 1957



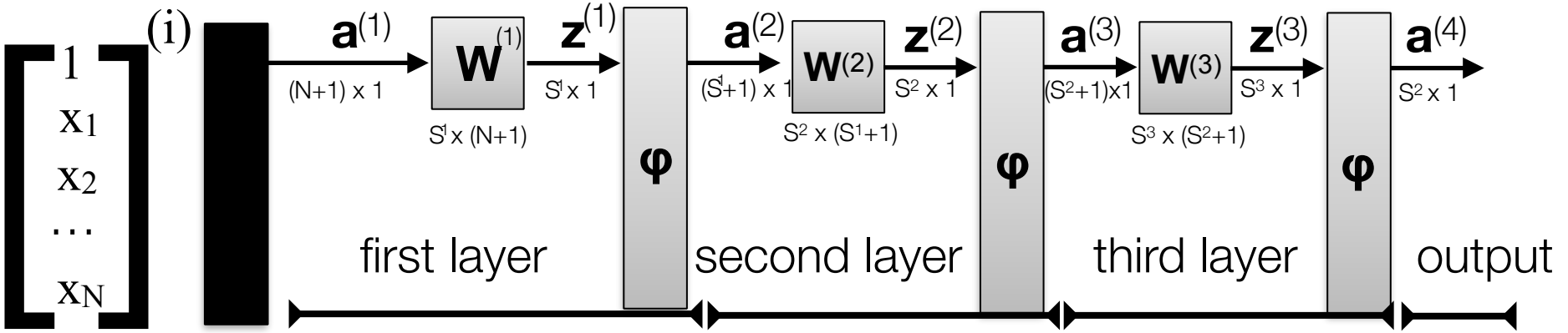
hard limit

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

Compact feedforward notation



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

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

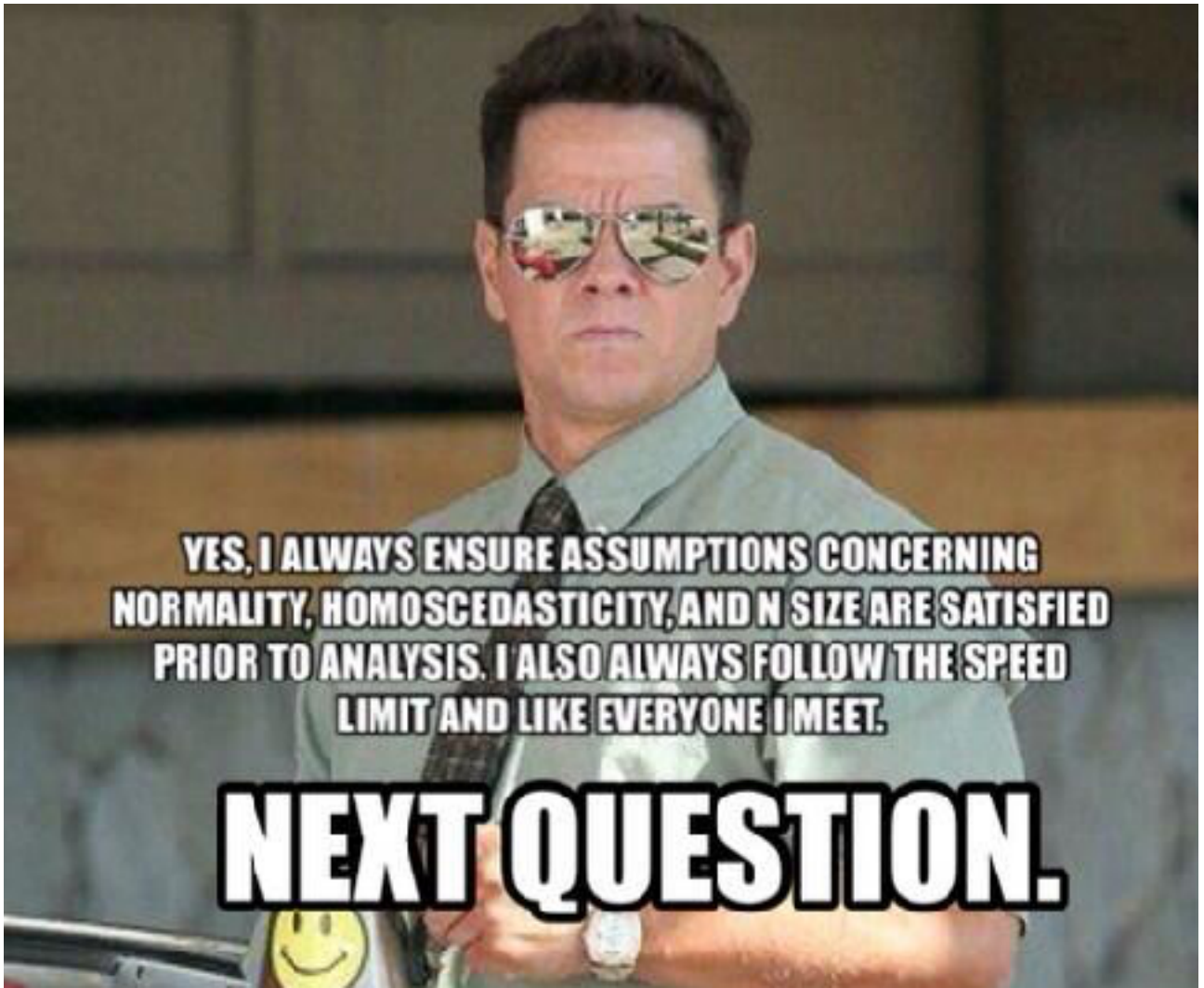
$$\begin{bmatrix} z^{(L)}_1 \\ \dots \\ z^{(L)}_{S^L} \end{bmatrix}^{(i)} = \begin{bmatrix} w^{(L)}_{1,1} & w^{(L)}_{1,2} & w^{(L)}_{1,3} & \dots & w^{(L)}_{1,S^{L-1}+1} \\ & \dots & & & \\ w^{(L)}_{S^L,1} & w^{(L)}_{S^L,2} & \dots & & w^{(L)}_{S^L,S^{L-1}+1} \end{bmatrix} \begin{bmatrix} a^{(L)}_1 \\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

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

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

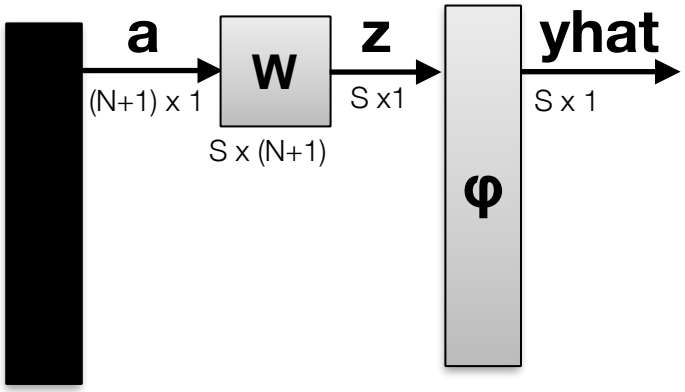
notation adapted from *Neural Network Design*, Hagan, Demuth, Beale, and De Jesus

Training Neural Network Architectures



Simple Architectures

- Rosenblatt's perceptron, 1957



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

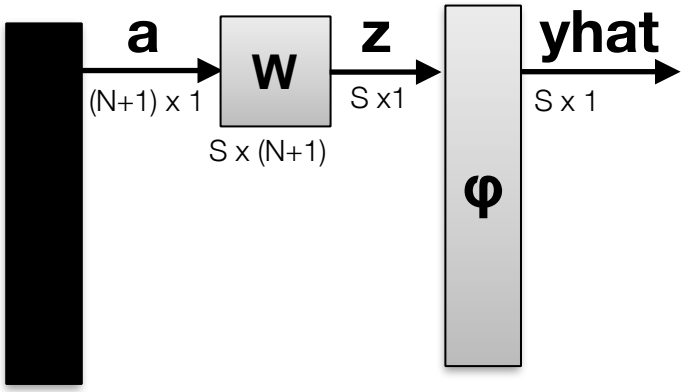
Need objective Function, minimize MSE $J(\mathbf{W}) = ||\mathbf{Y} - \hat{\mathbf{Y}}||^2$

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

$$\begin{bmatrix} y_1 \\ \dots \\ y_C \end{bmatrix}^{(i)} \Rightarrow \left[\begin{bmatrix} y_1 \\ \dots \\ y_C \end{bmatrix}^{(1)} \dots \begin{bmatrix} y_1 \\ \dots \\ y_C \end{bmatrix}^{(M)} \right] = \mathbf{Y}$$

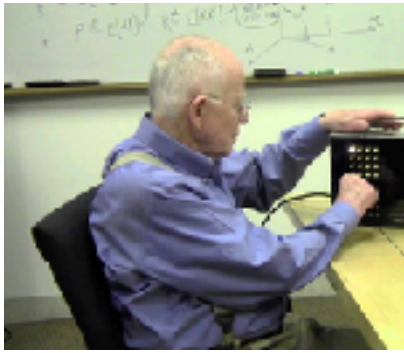
Simple Architectures

- Adaline network, Widrow and Hoff, 1960



a=z

Marcian "Ted" Hoff



Bernard Widrow

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

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

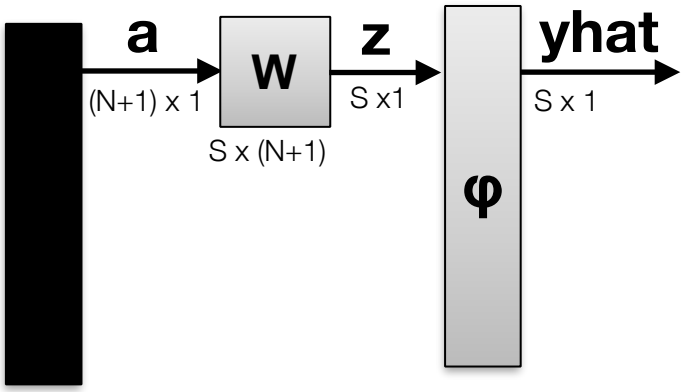
Need gradient $\nabla J(\mathbf{W})$ for update equation $\mathbf{W} \leftarrow \mathbf{W} + \eta \nabla J(\mathbf{W})$

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

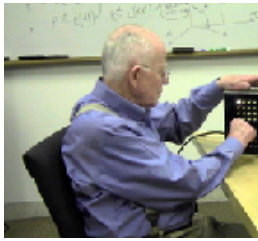
Simple Architectures

Marcian “Ted” Hoff

- Adaline network, Widrow and Hoff, 1960



a=z



Bernard Widrow

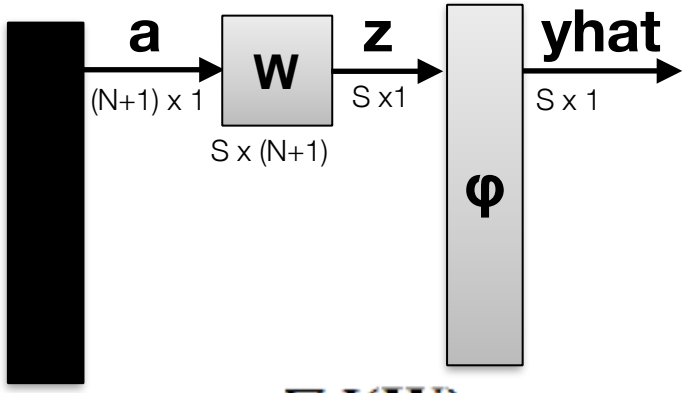
need gradient $\nabla J(\mathbf{W})$ for update equation $\mathbf{W} \leftarrow \mathbf{W} + \eta \nabla J(\mathbf{W})$

For case S=1, $\mathbf{W}$ 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



sigmoid

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

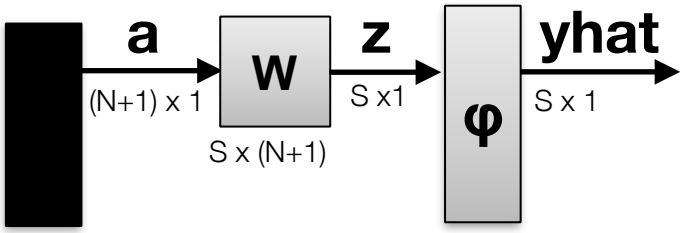
need gradient $\nabla J(\mathbf{W})$ for update equation $\mathbf{W} \leftarrow \mathbf{W} + \eta \nabla J(\mathbf{W})$

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

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

What happens when S > 1 ?

What if we have more than S=1?



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

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

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

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

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

which is one-versus-all!

$$J({}_1\mathbf{w}) = \sum_{i=1} [y_1^{(i)} - \varphi({}_1\mathbf{w} \cdot \mathbf{x}^{(i)})]^2$$

$$J({}_2\mathbf{w}) = \sum_{i=1} [y_2^{(i)} - \varphi({}_2\mathbf{w} \cdot \mathbf{x}^{(i)})]^2$$

$$\dots$$

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

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

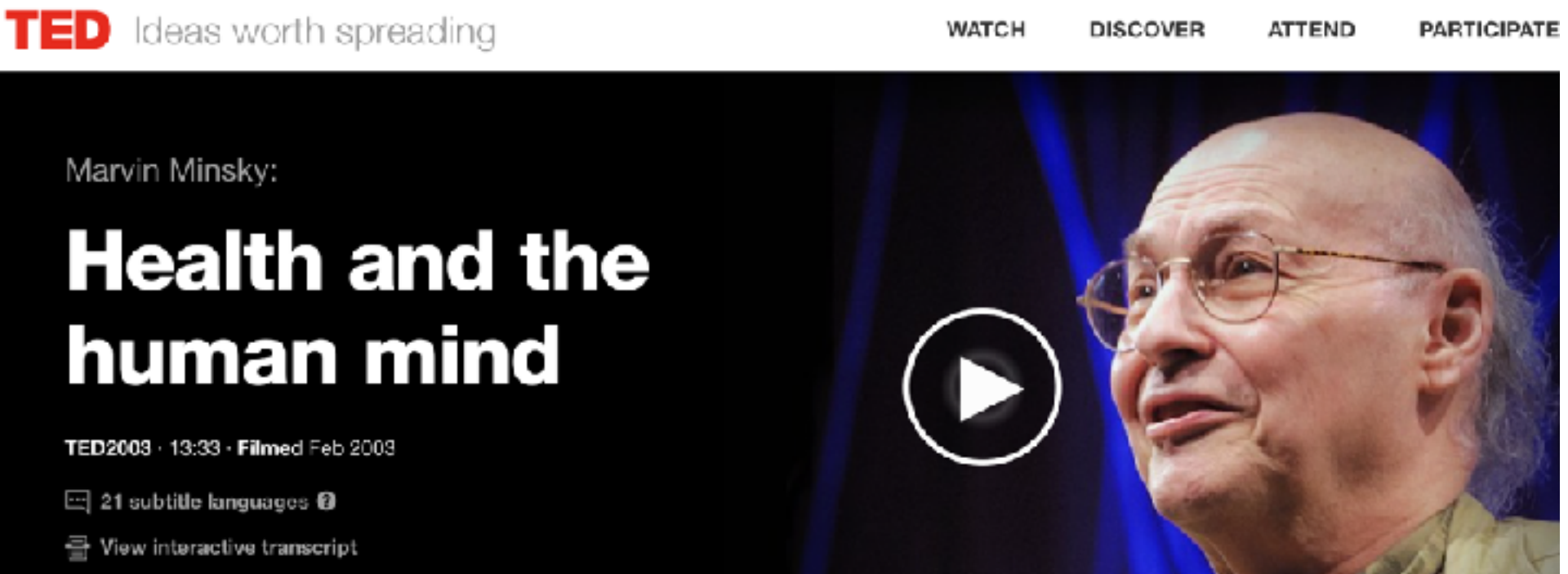
these networks were created in the 50's and 60's
but were abandoned
why were they not used?

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…"

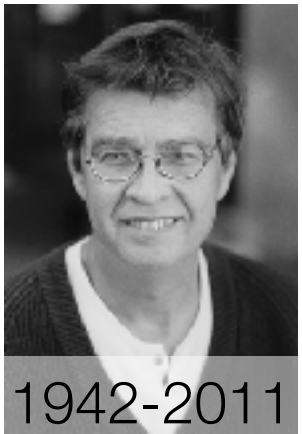
- Minsky and Papert publish limitations paper, 1969:



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

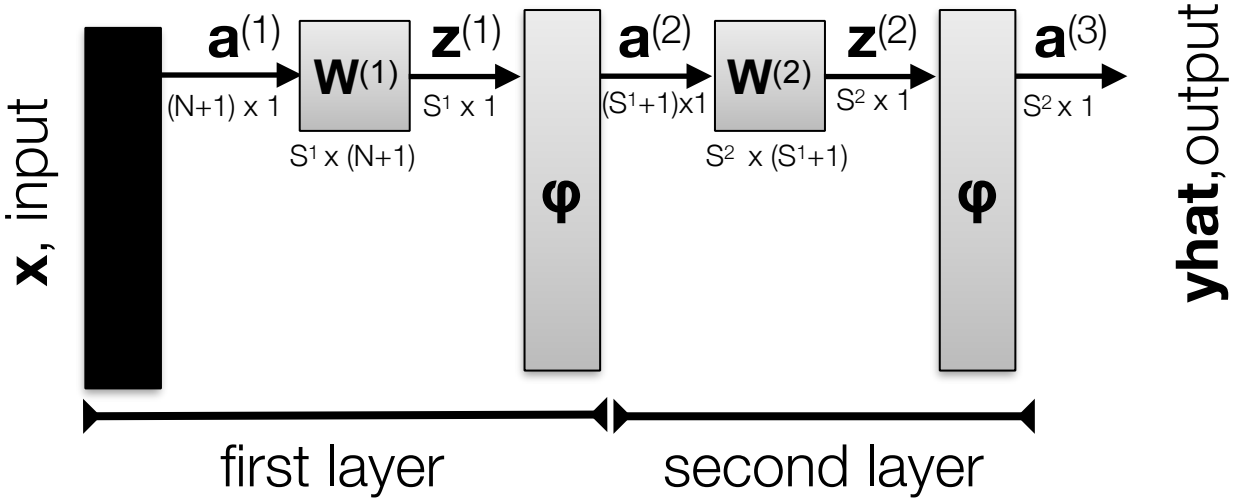


1942-2011

Geoffrey Hinton

More Advanced Architectures: MLP

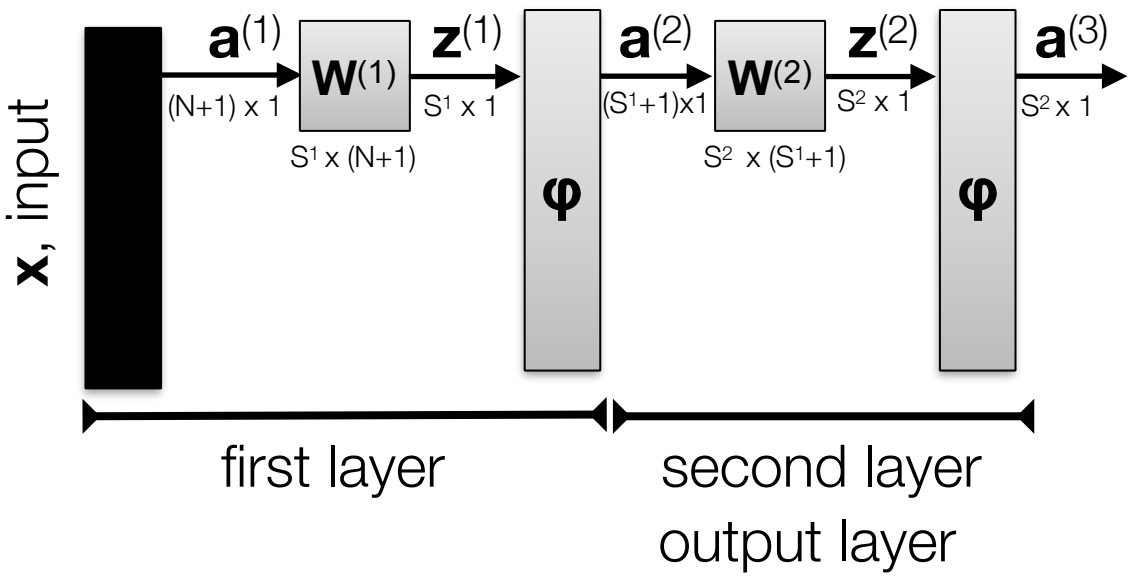
- The multi-layer perceptron (MLP):
  - two layers shown, but could be arbitrarily many layers
  - algorithm is agnostic to number of layers (*kinda*)



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

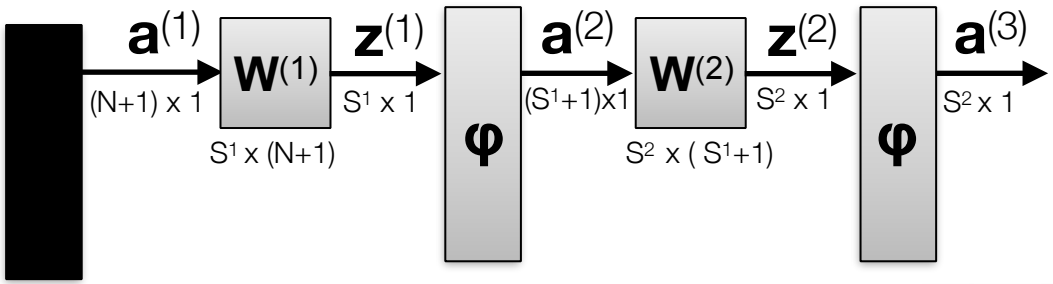


$$J(\mathbf{W}) = \sum_{k}^{M} (\mathbf{y}^{(k)} - \mathbf{a}^{(L)})^2$$

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

Back propagation

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



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

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!

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=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

Multi-Layer Neural Networks

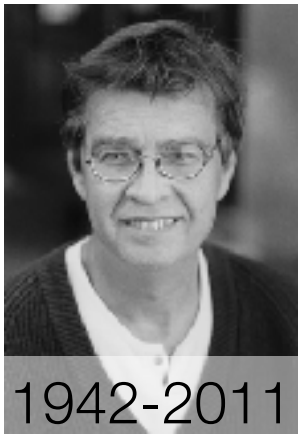
Class Logistics and Agenda

- Multi Week Agenda:
  - *SVM Review*
  - *Neural Networks History, up to 1980*
  - Multi-layer Architectures
  - Programming Multi-layer training
- Next Time: Break before Lab 4!

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

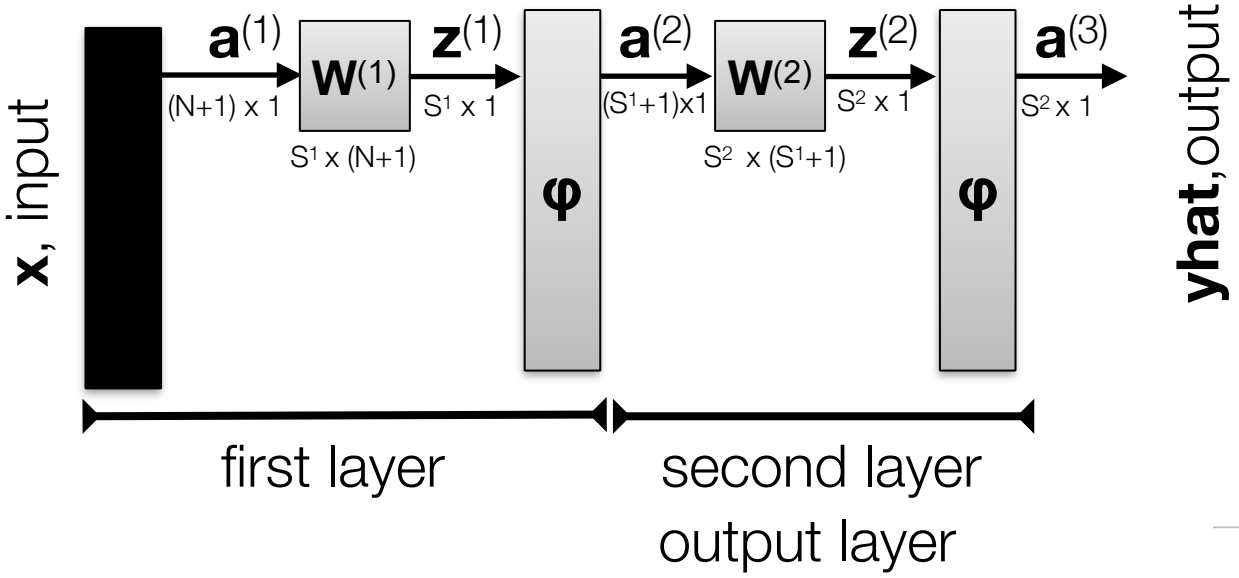


1942-2011

Geoffrey Hinton

Back propagation

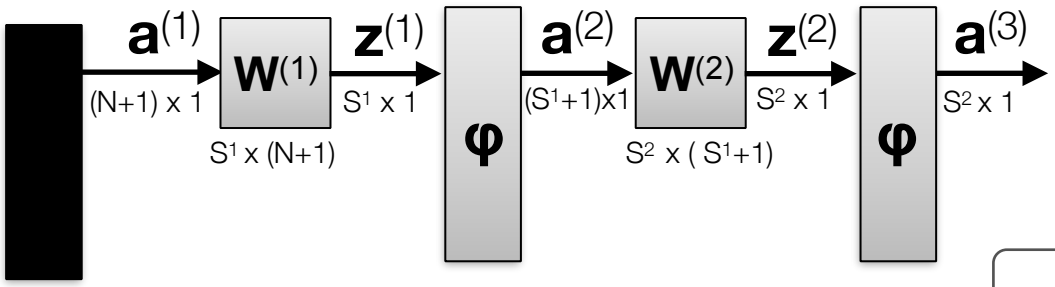
- Steps:
  - propagate weights forward
  - calculate gradient at final layer
  - back propagate gradient for each layer
    - via recurrence relation



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

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

Back propagation summary



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

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

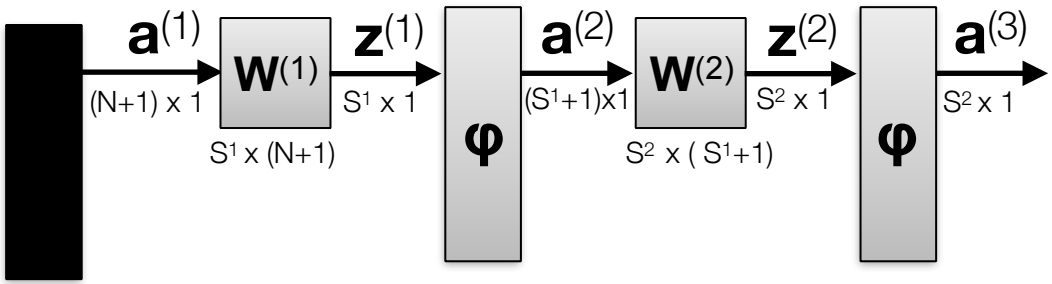
$$\nabla^{(2)} = \mathbf{V}^{(2)} \cdot [\mathbf{A}^{(2)}]^T$$

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

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

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

Back propagation summary

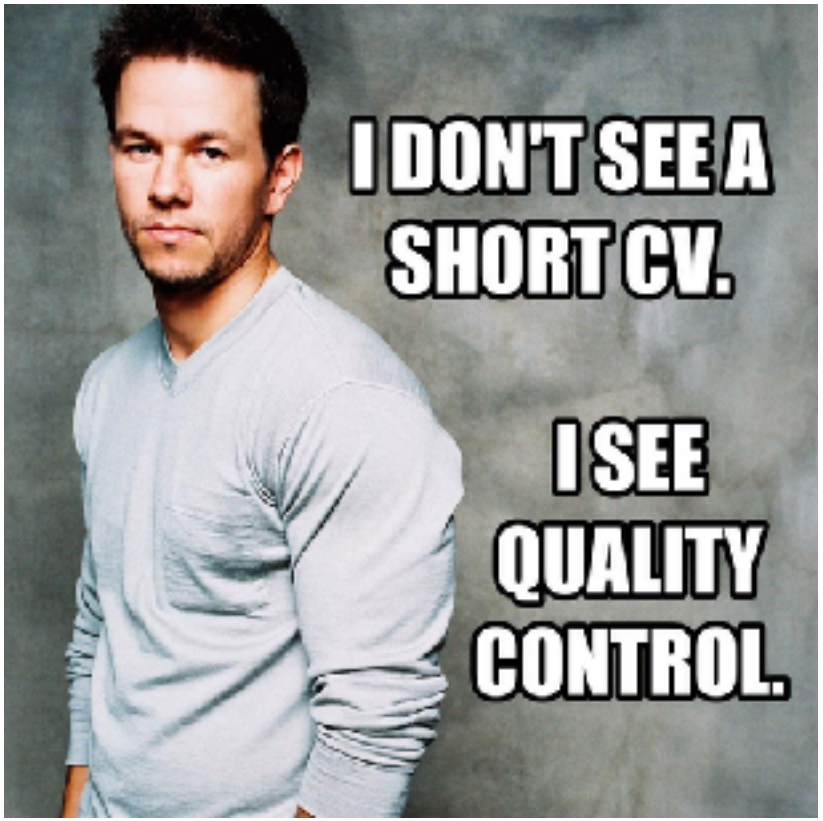


- **Self Test**:
  True or False: If we change the cost function, $J(\mathbf{W})$, we only need to update the final layer calculation of the back propagation steps. The remainder of the algorithm is unchanged.
  - A. True
  - B. False

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

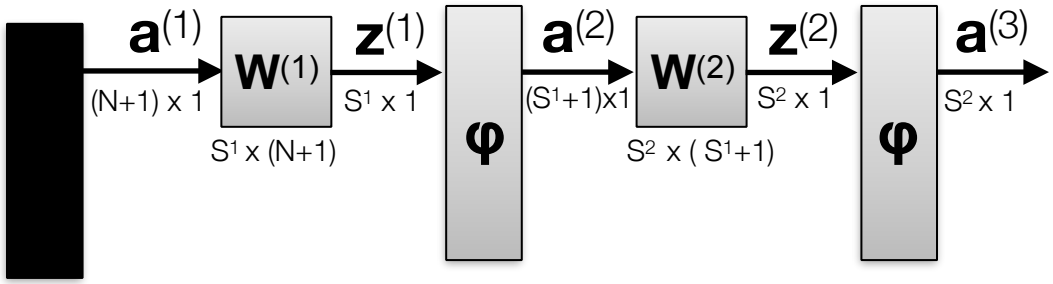
$$\nabla^{(2)} = \mathbf{V}^{(2)} \cdot [\mathbf{A}^{(2)}]^T$$

Programming Multi-layer Neural Networks



Guided Example

Recall from the in-class asisgnment



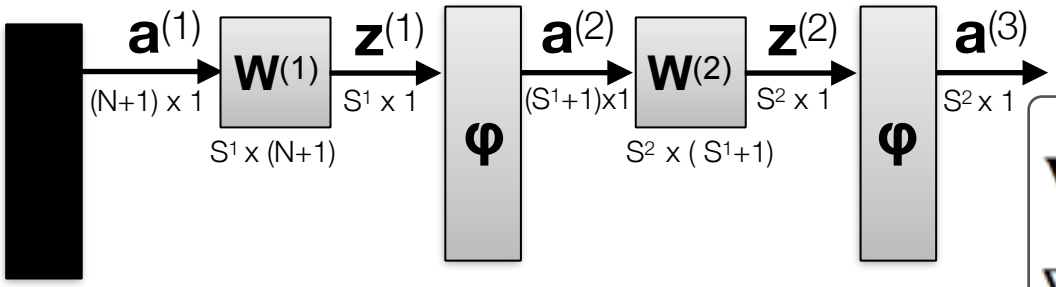
1. Forward propagate to get **Z**, **A**

```
# 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



1. Forward propagate to get **Z**, **A**
2. Get final layer gradient
3. Back propagate sensitivities

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

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

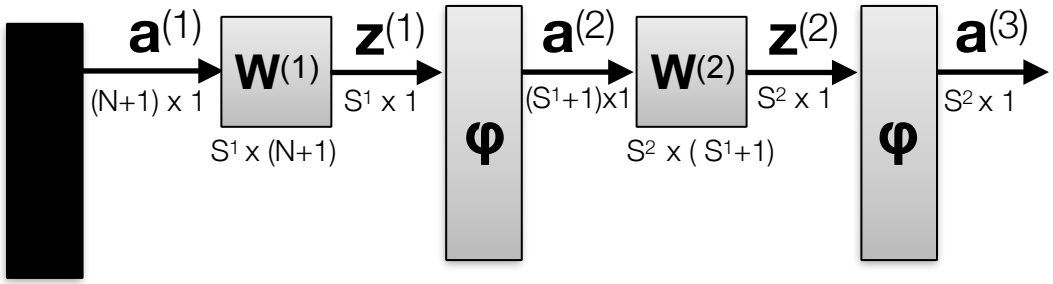
$$\mathbf{V}^{(1)} = \mathbf{A}^{(2)} * (1 - \mathbf{A}^{(2)}) * [\mathbf{W}^{(2)}]^T \cdot \mathbf{V}^{(2)}$$

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

```
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 **Z**, **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
```

Demo

Two Layer Perceptron

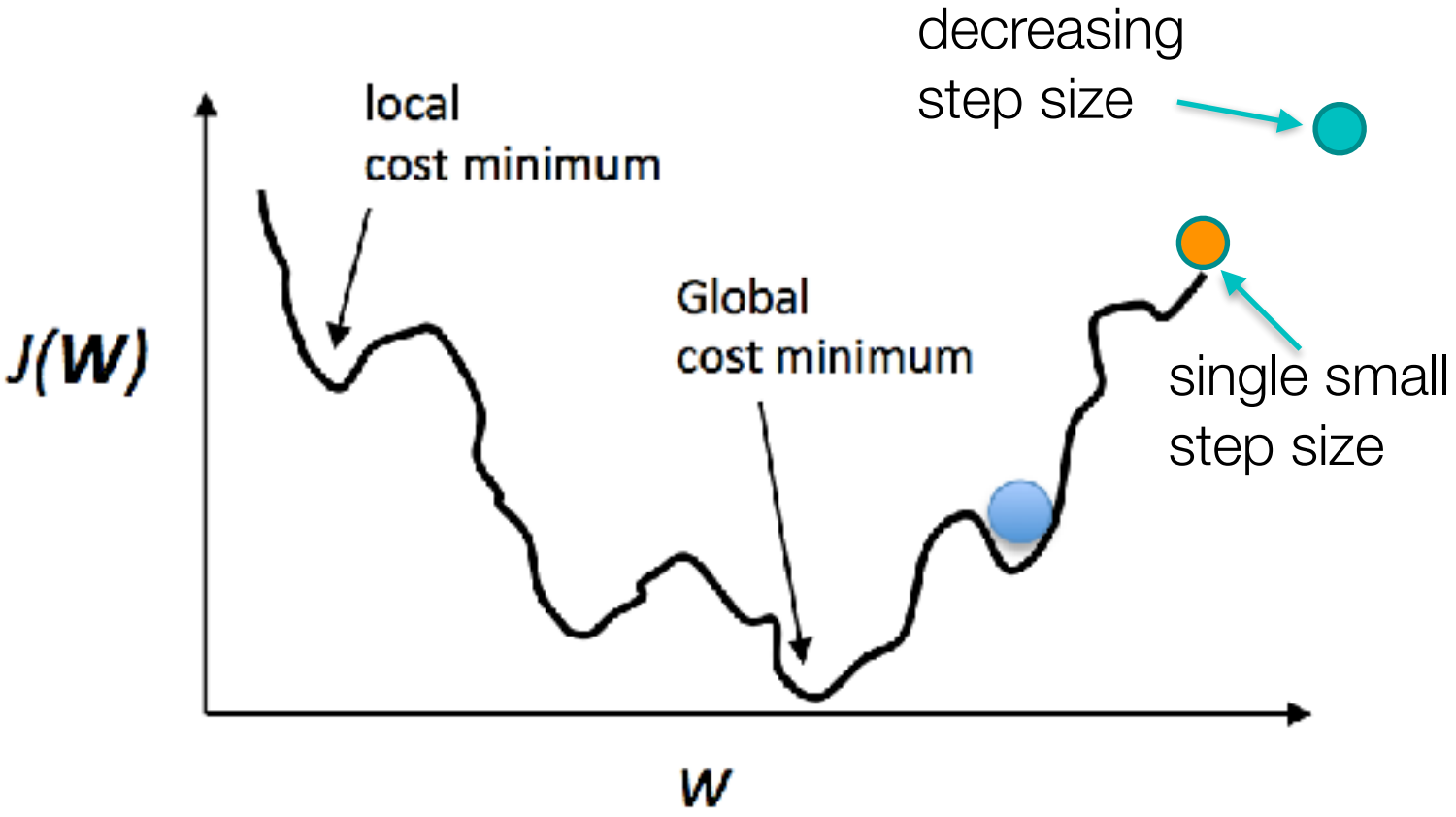
with regularization
and vectorization

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

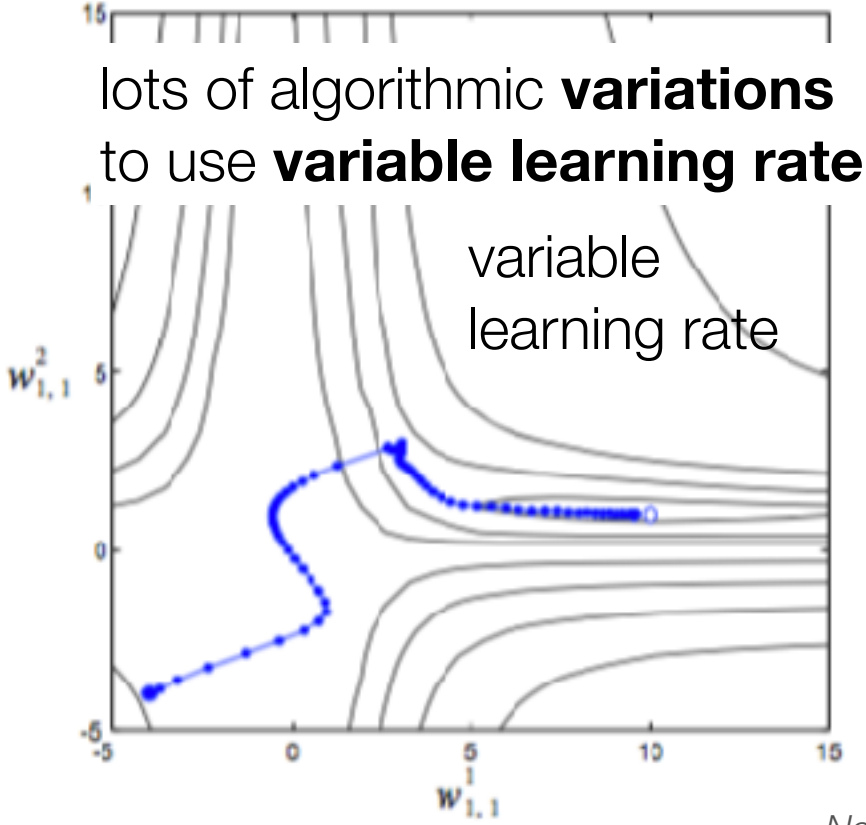
- Space is no longer convex
  - **One solution**:
    - start with large step size
    - "cool down" by decreasing step size for higher iterations

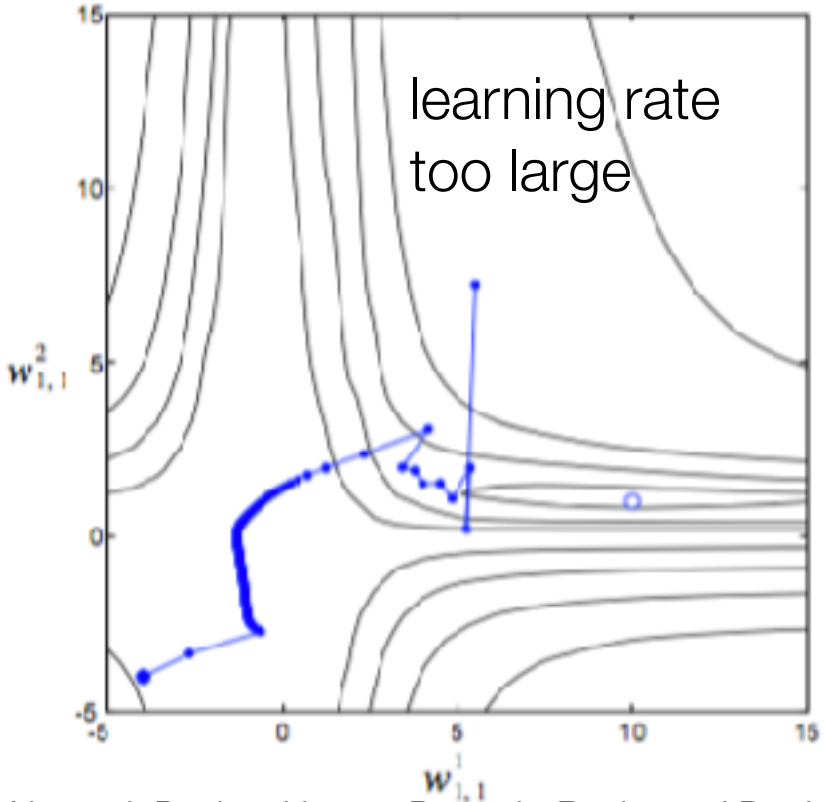


Problems with Advanced Architectures

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

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

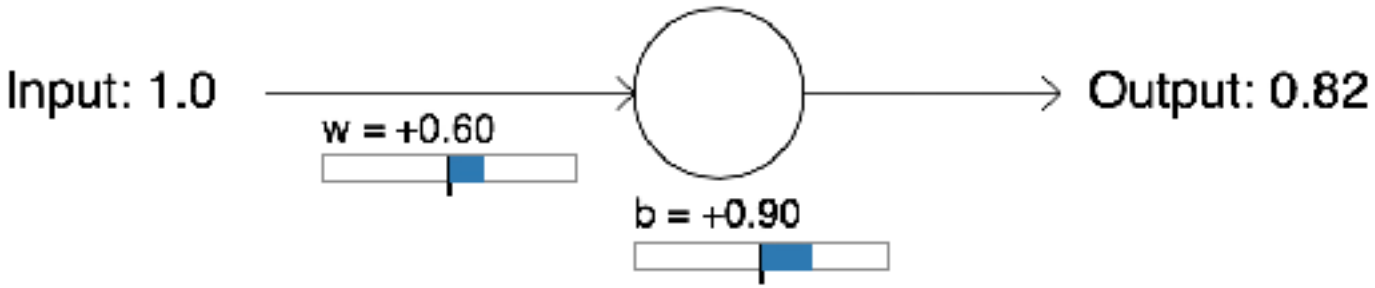
L-BFGS

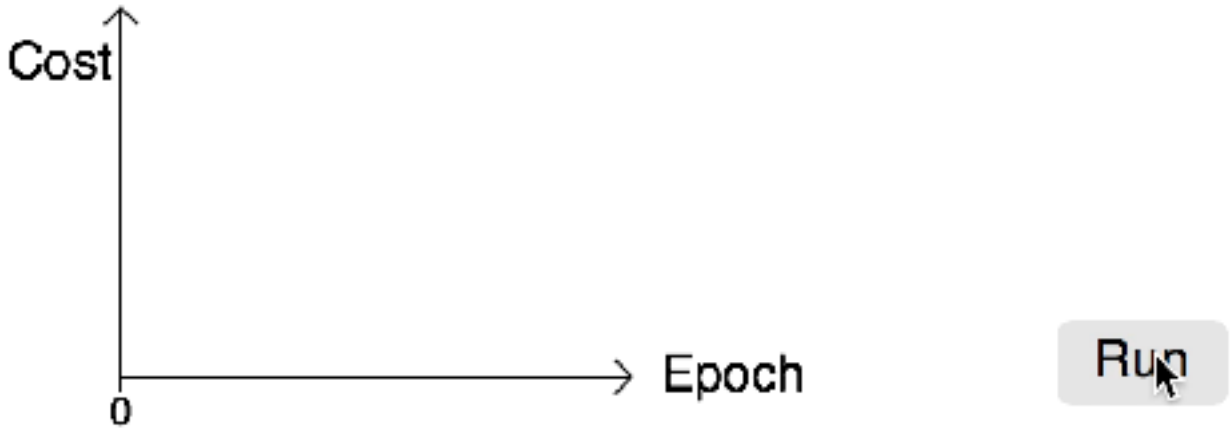
Practical Implementation of Architectures

- 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





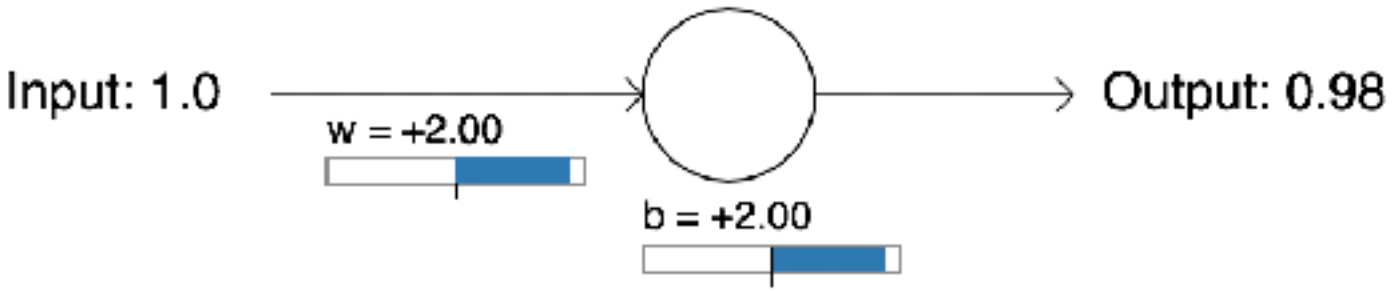
Neural Networks and Deep Learning, Michael Nielson, 2015

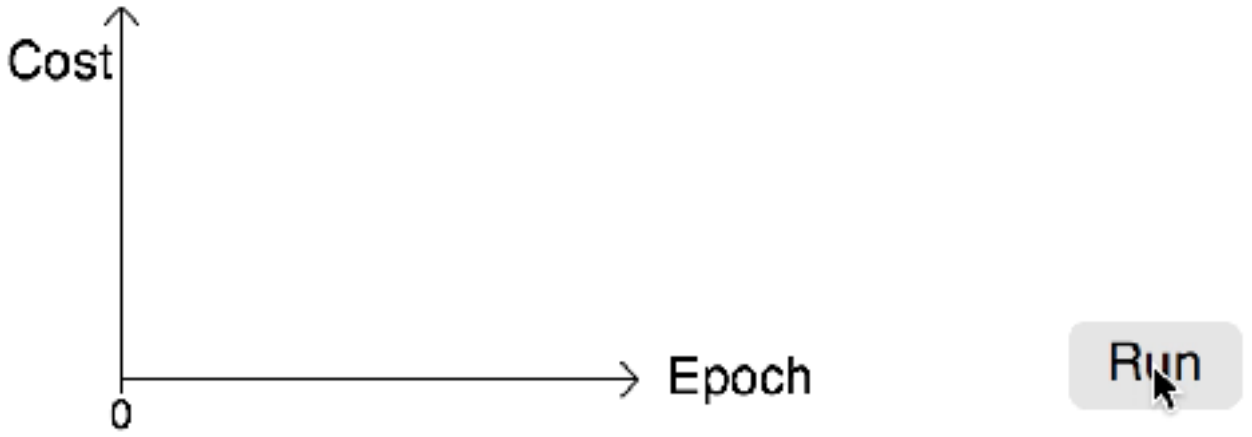
Practical Implementation of Architectures

- 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



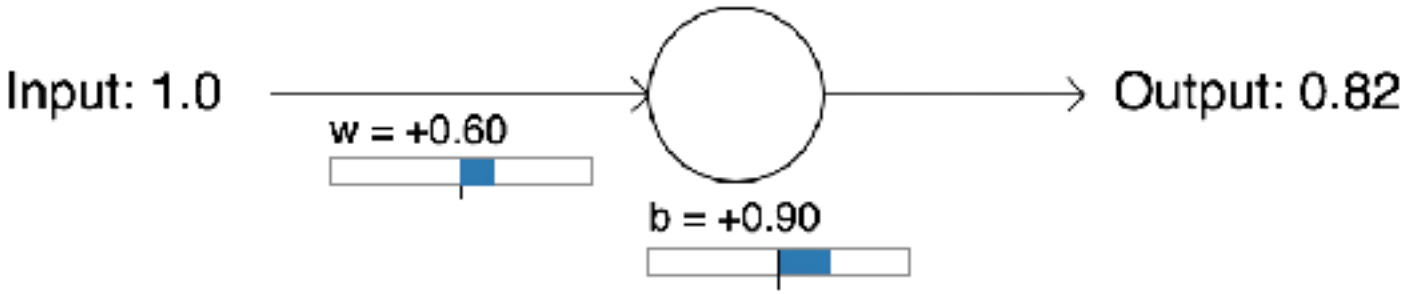


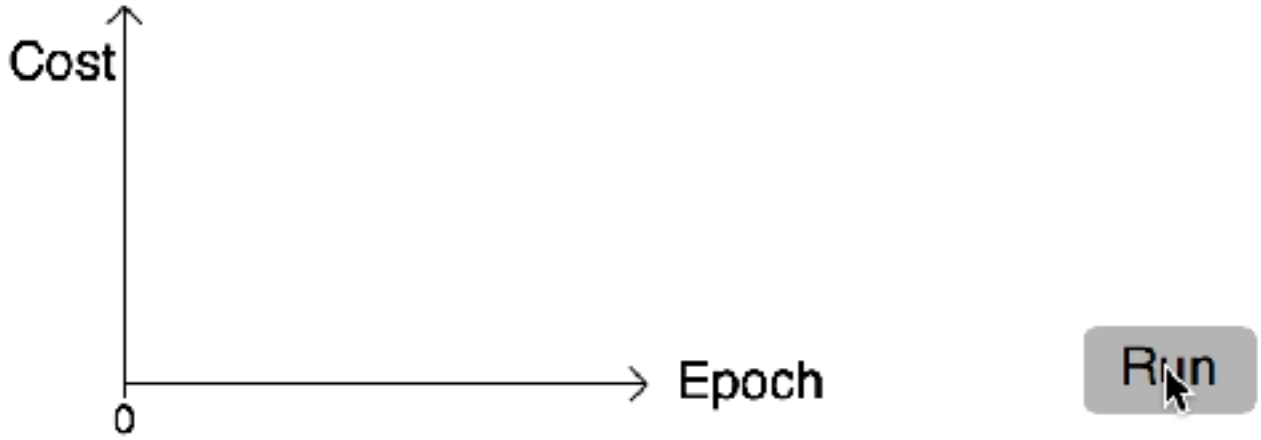
Practical Implementation of Architectures

- A new cost function: **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





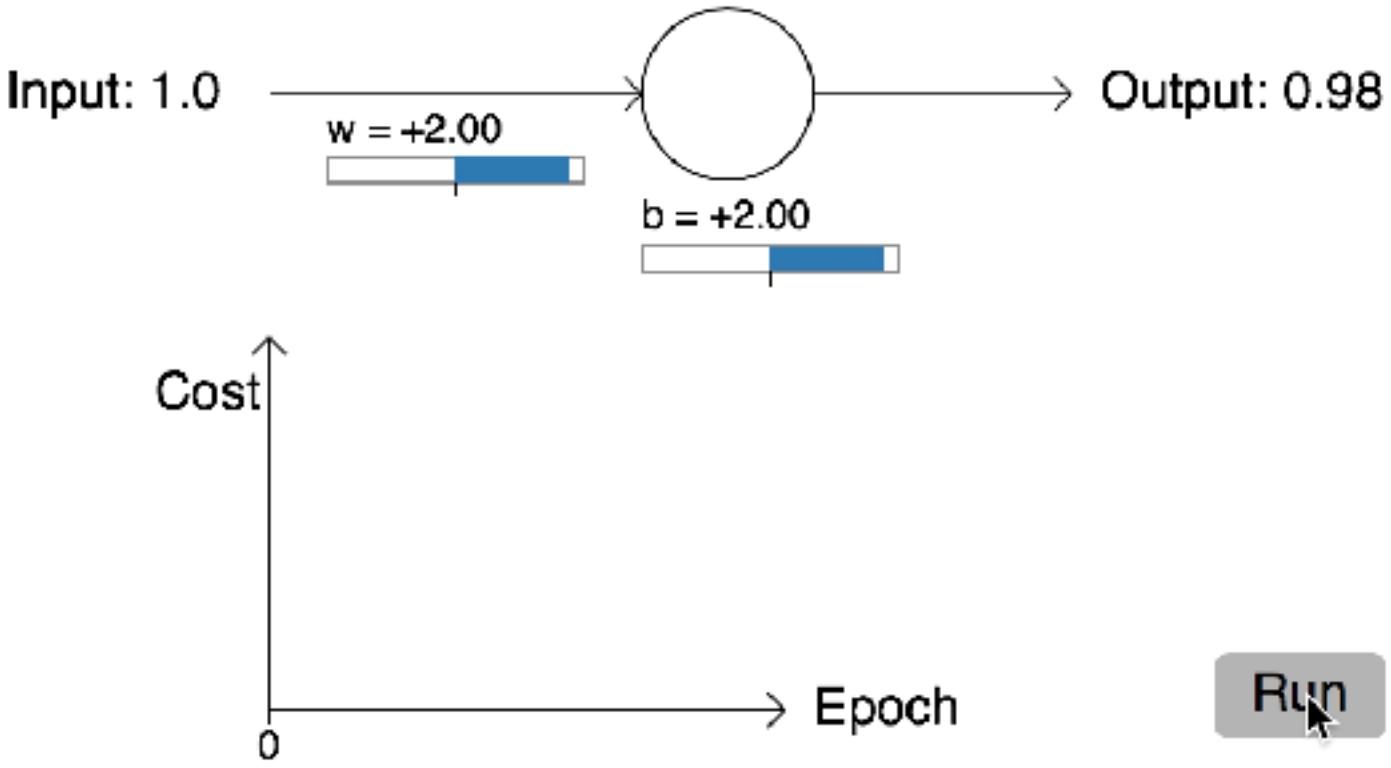
Neural Networks and Deep Learning, Michael Nielson, 2015

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

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

bp-5

Neural Networks and Deep Learning, Michael Nielson, 2015

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

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

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

```
# 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
```

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

new update

bp-5

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

Neural Networks and Deep Learning, Michael Nielson, 2015

Demo

Two Layer Perceptron

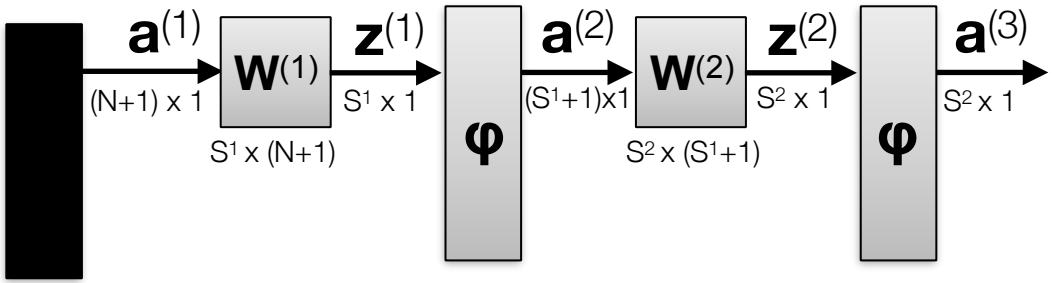
cross entropy

Practical Implementation of Architectures

- Beyond L2 regularization
  - dropout
  - expansion

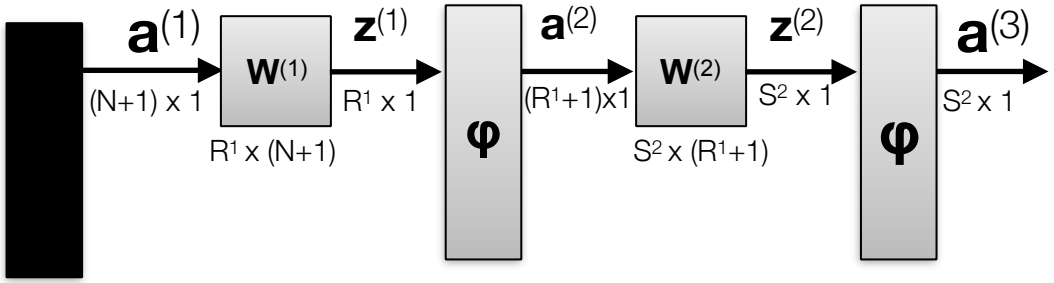
Practical Implementation of Architectures

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



For each mini-batch,

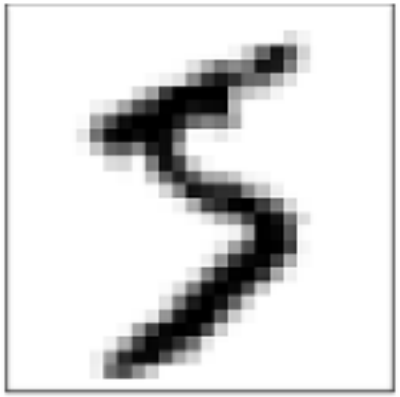
R^1 is subset of S^1



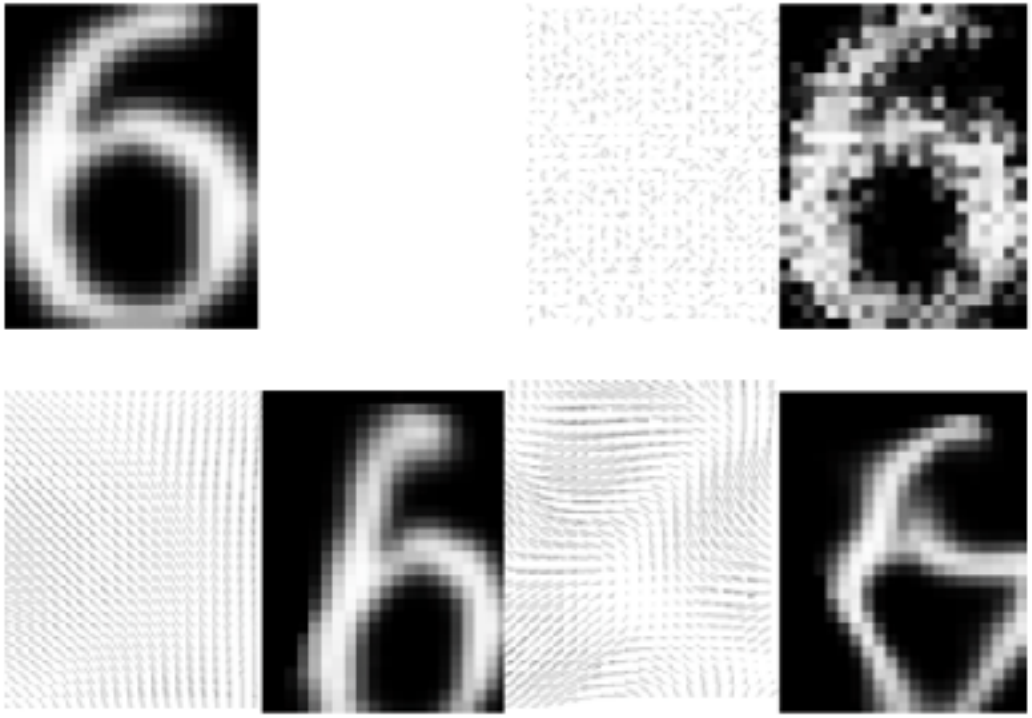
Why does this work?

Practical Implementation of Architectures

- Expansion
  - get more data
  - perturb your data



Neural Networks and Deep Learning, Michael Nielson, 2015



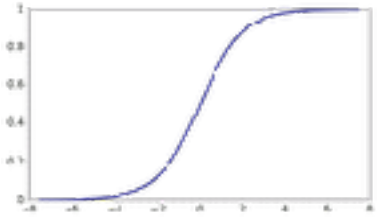
Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis, by Patrice Simard, Dave Steinkraus, and John Platt (2003)

98.4%→99.3%

Practical Implementation of Architectures

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

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



each row is sum of layer inputs

want sum to be between $-\varepsilon<\Sigma<\varepsilon$ for no saturation
solution: squash initial weights sizes

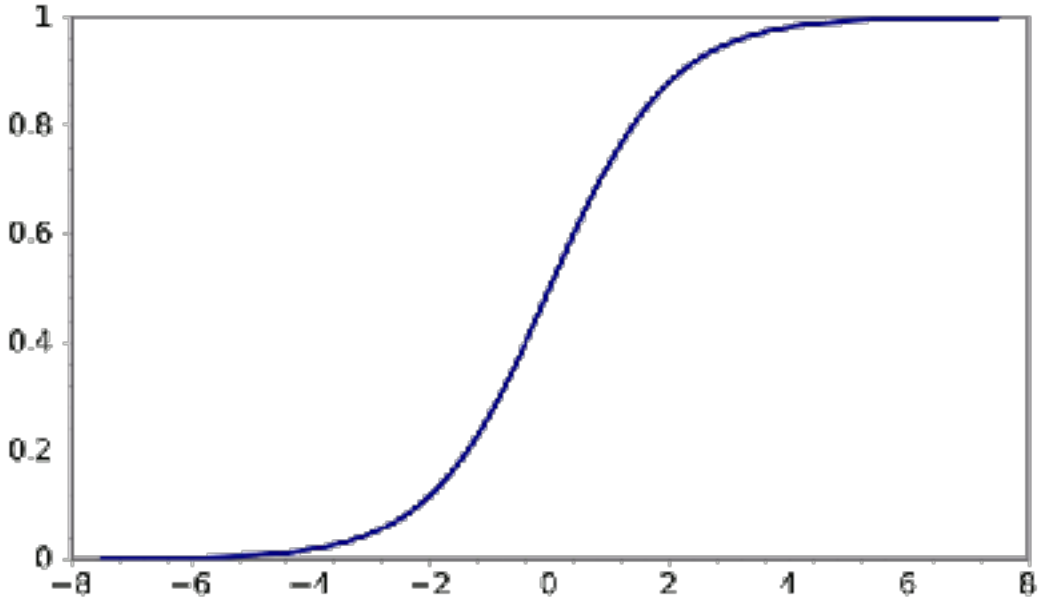
- a nice choice: each element of **W** is selected from a Gaussian with zero mean
- for adding Gaussian distributions, variances add together:
  - make each variance $1/\mathbf{W}_{\text{num\_elements\_in\_row}}$
  - which is the same as standard deviation = $1/sqrt(\mathbf{W}_{\text{num\_elements\_in\_row}})$

Self Test

- for adding Gaussian distributions, variances add together

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

- If you initialized the weights with too large variance, you would expect the output of the neuron, $\boldsymbol{\phi}$, to be:
  - A. saturated to “1”
  - B. saturated to “0”
  - C. could either be saturated to “0” or “1”
  - D. would not be saturated

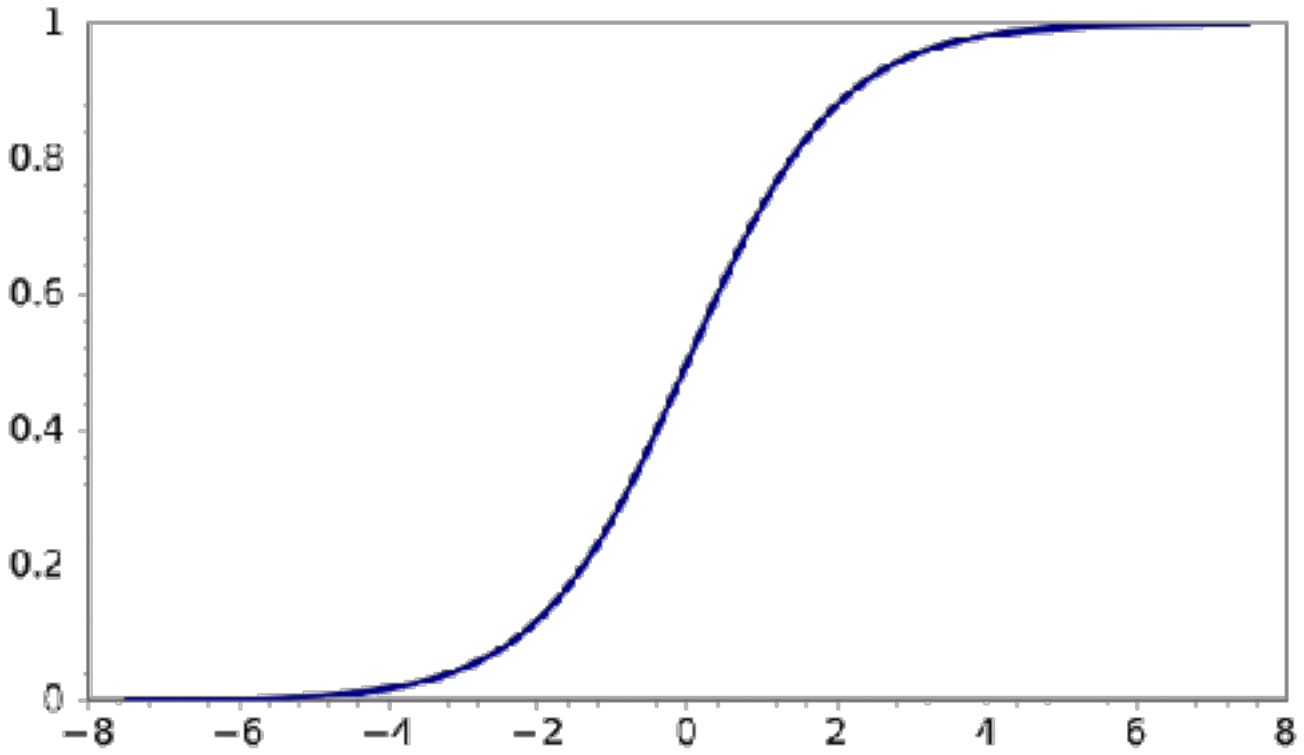


Self Test

- for adding Gaussian distributions, variances add together

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

- What is the derivative of a saturated sigmoid neuron?
  - A. zero
  - B. one
  - C. a * (1-a)
  - D. it depends



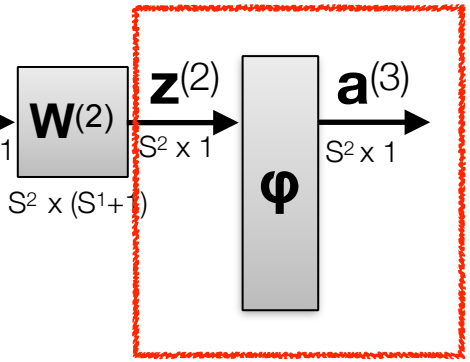
Demo

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!

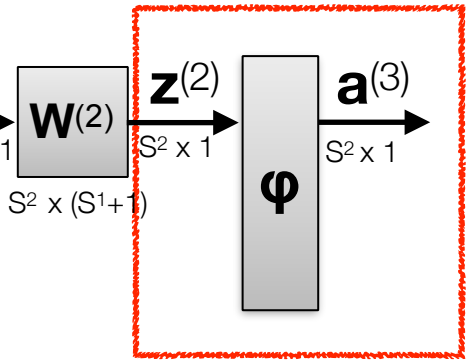
it has many of the same properties as Cross Entropy
but also the advantage of **interpretation as a probability**

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.

Neural Networks and Deep Learning, Michael Nielson, 2015

Practical Implementation of Architectures

- A new nonlinearity: **recitifed 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

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

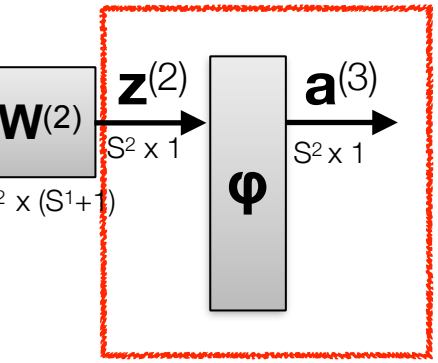
Neural Networks and Deep Learning, Michael Nielson, 2015

Demo

Two Layer Perceptron

ReLU Nonlinearities

Practical Implementation of Architectures



- ReLU not the only way!
- **Swish** (GoogleBrain 2017)
- Mixing of sigmoid, σ, and ReLU

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

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

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

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

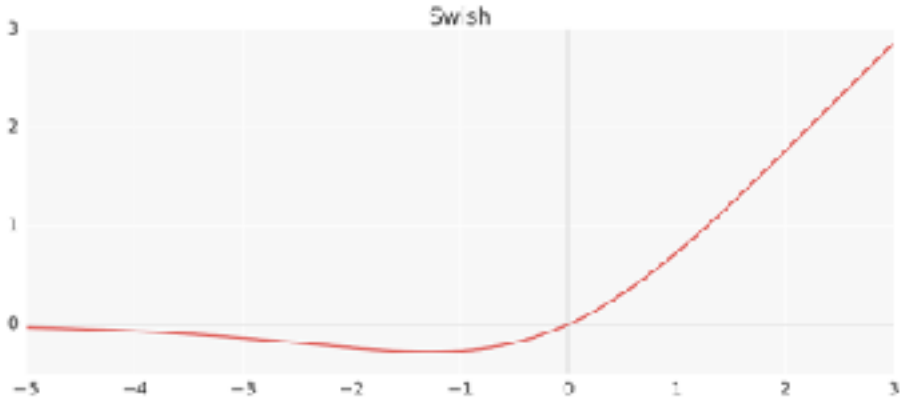


Figure 1: The Swish activation function.

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 solve any problem with enough data
  - but… 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

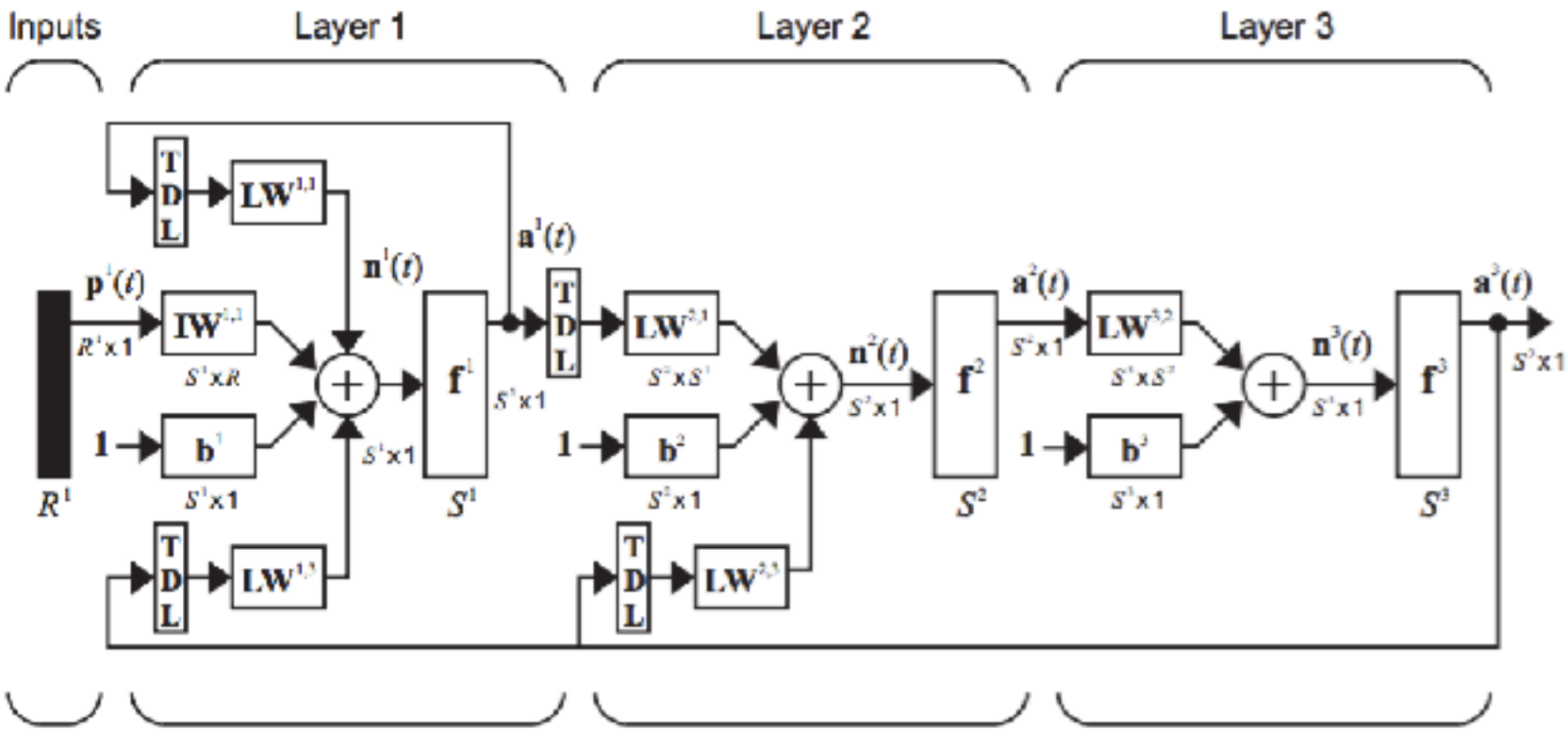
Class Logistics and Agenda

- 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

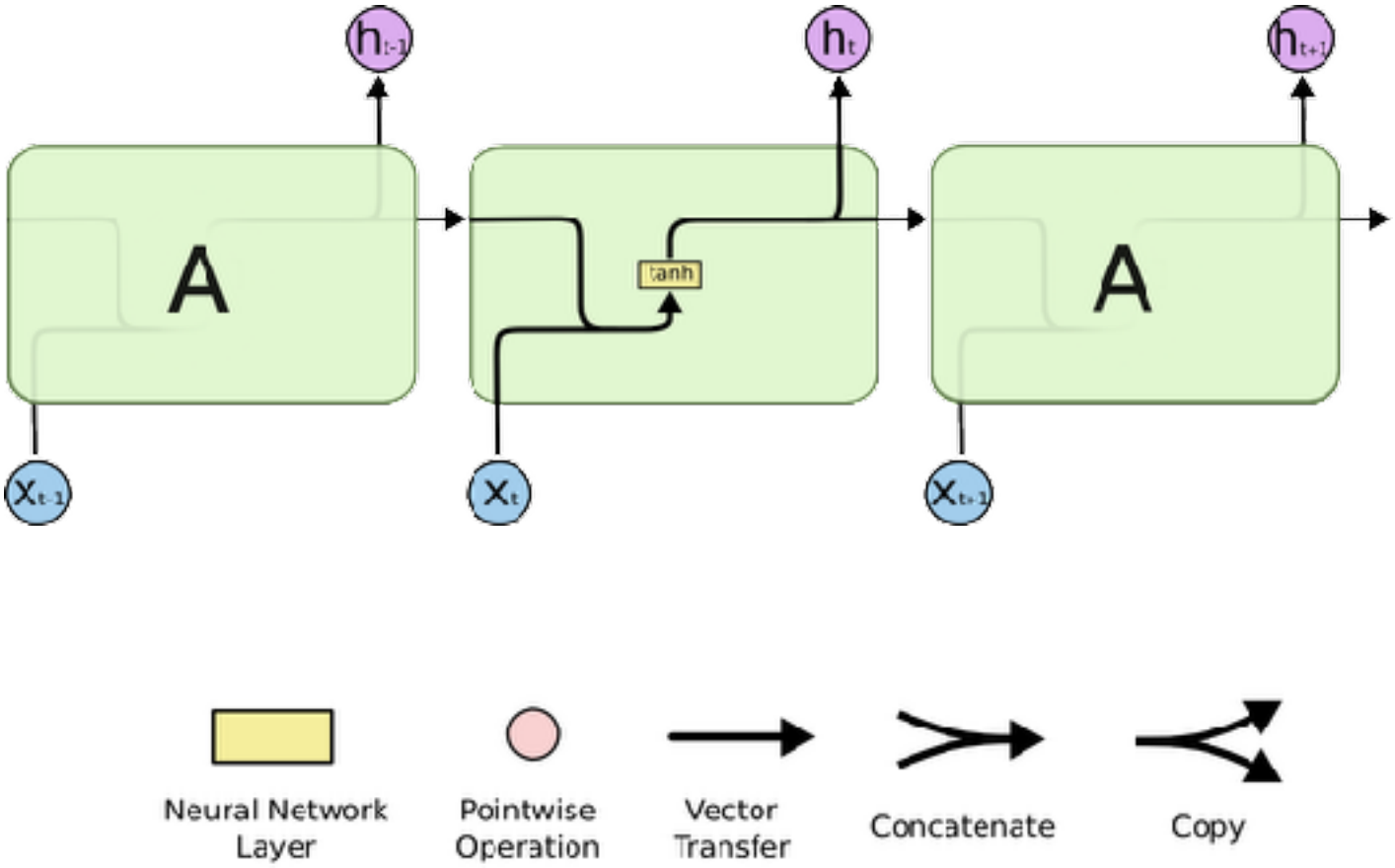
- Dynamic Networks (recurrent networks)
  - can use current and previous inputs, in time
  - still popular, but ultimately extremely hard to train
  - **highly successful variant**: long short term memory, **LSTM**



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

More Advanced Architectures

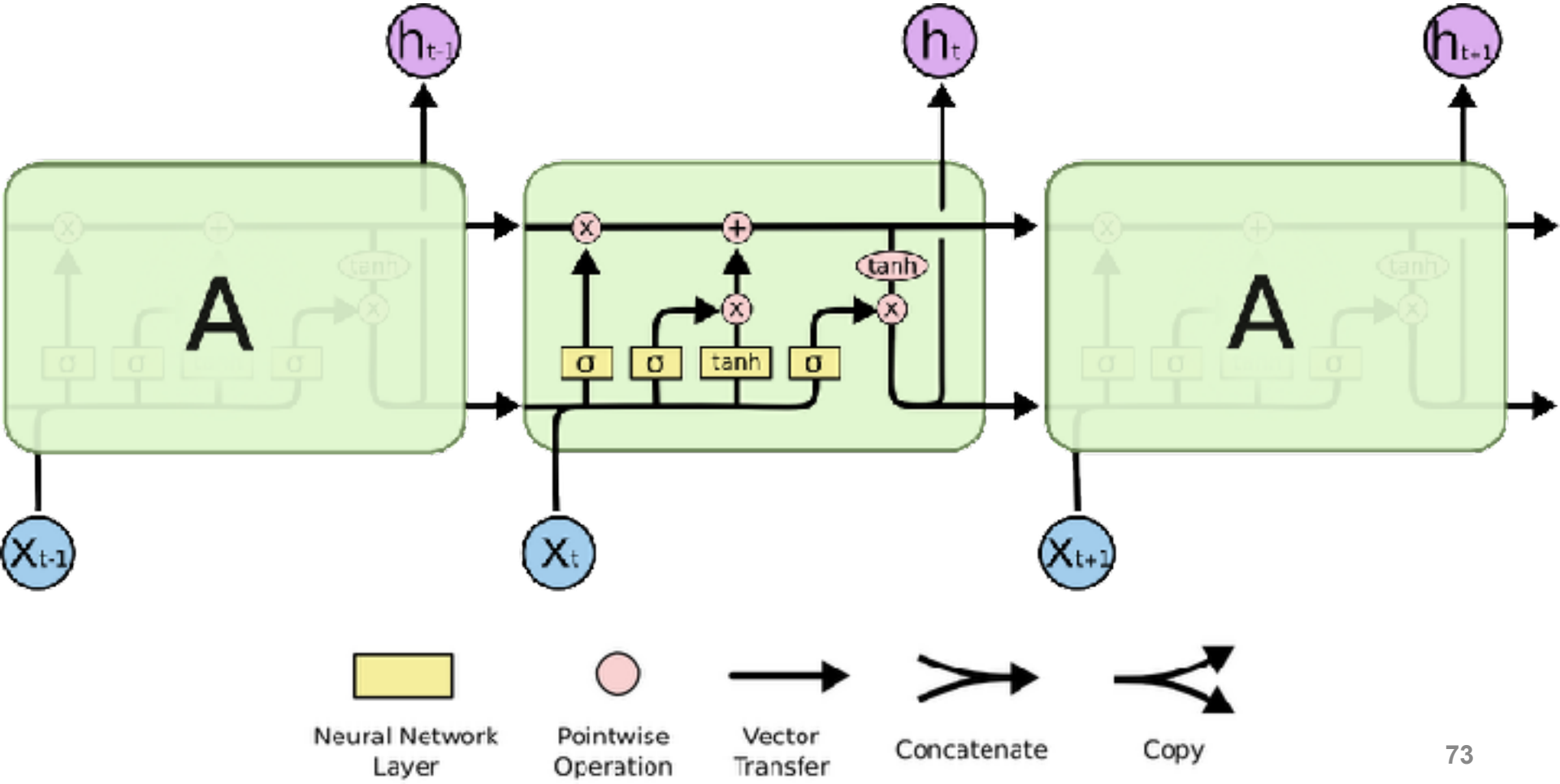
- **LSTM key idea**: limit how past data can affect output



http://colah.github.io/posts/2015-08-Understanding-LSTMs/

More Advanced Architectures

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

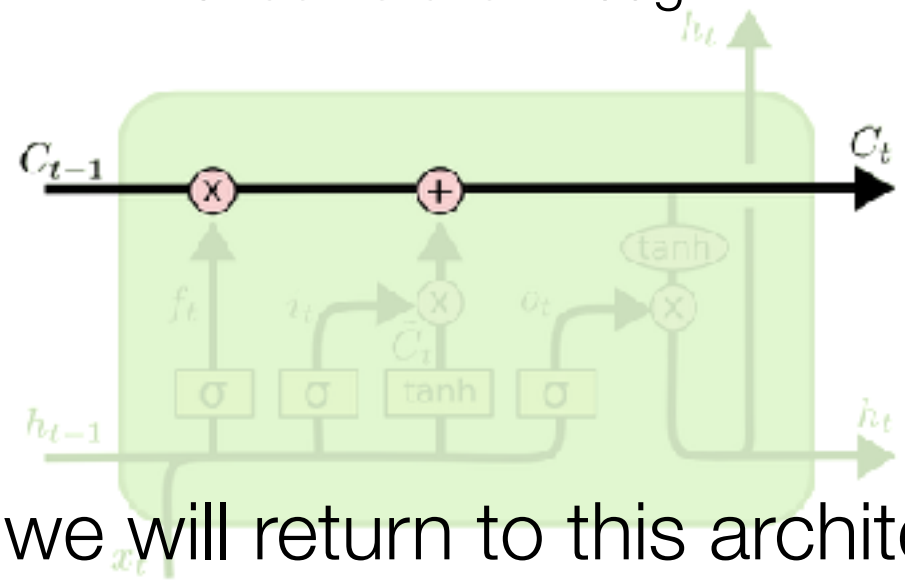


http://colah.github.io/posts/2015-08-Understanding-LSTMs/

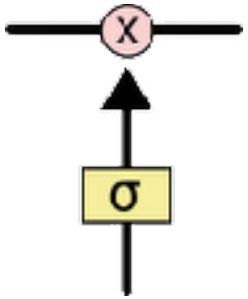
More Advanced Architectures

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

let **cell state** through

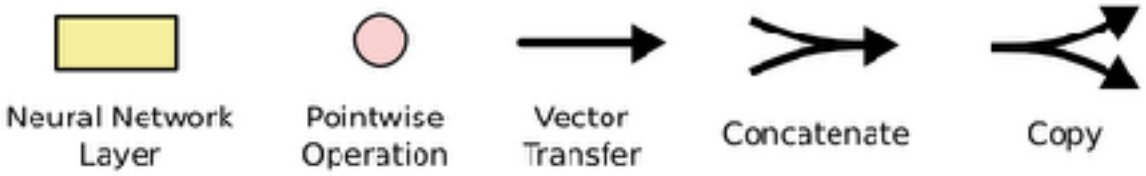


potentially **forget past** inputs via "gate" σ



we will return to this architecture later, for now:

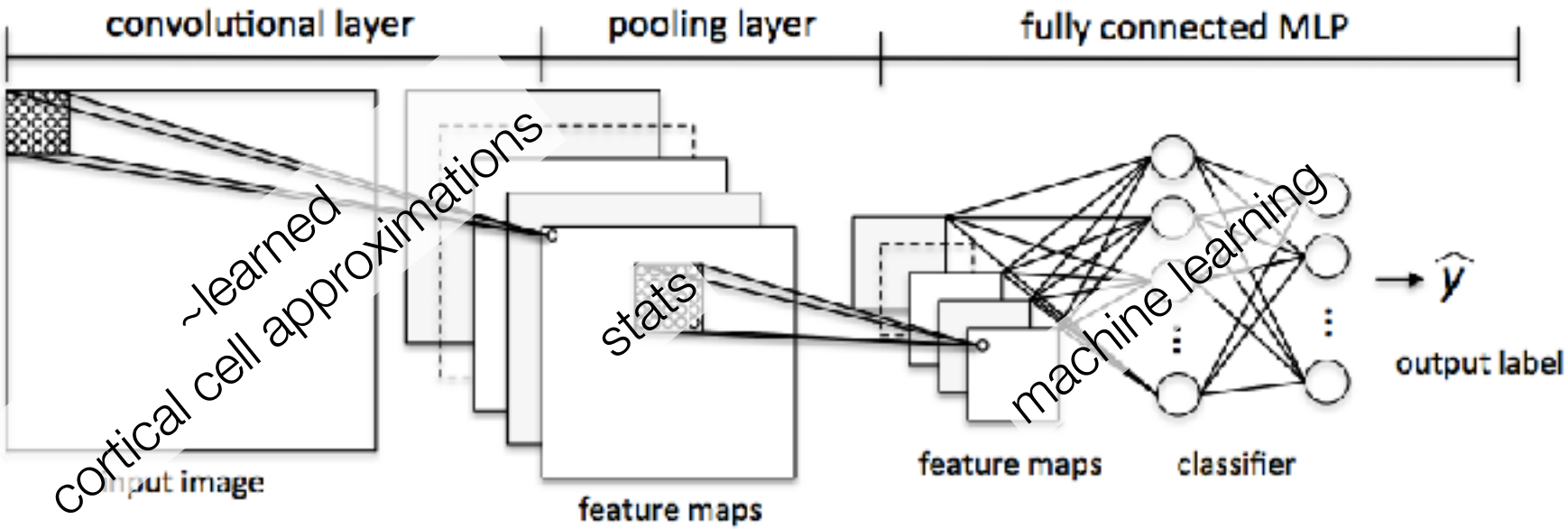
put it in long term memory 😂



http://colah.github.io/posts/2015-08-Understanding-LSTMs/

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**