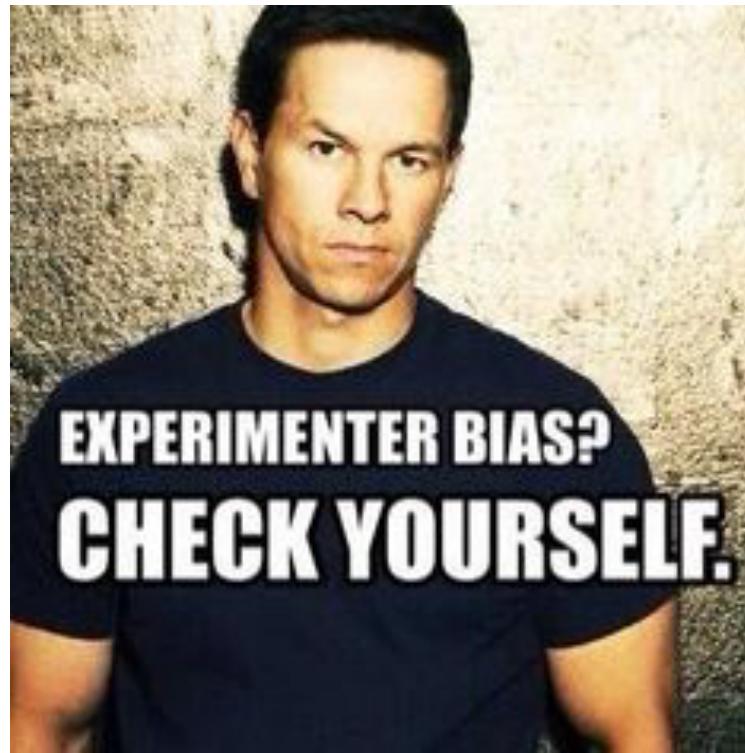

Lecture Notes for Machine Learning in Python

Professor Eric Larson
SVM Review and Neural Network Notation

Class Logistics and Agenda

- Logistics:
 - Nothing due this week!
 - But there is a flipped lecture assignment over Spring Break - Yay!
- 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

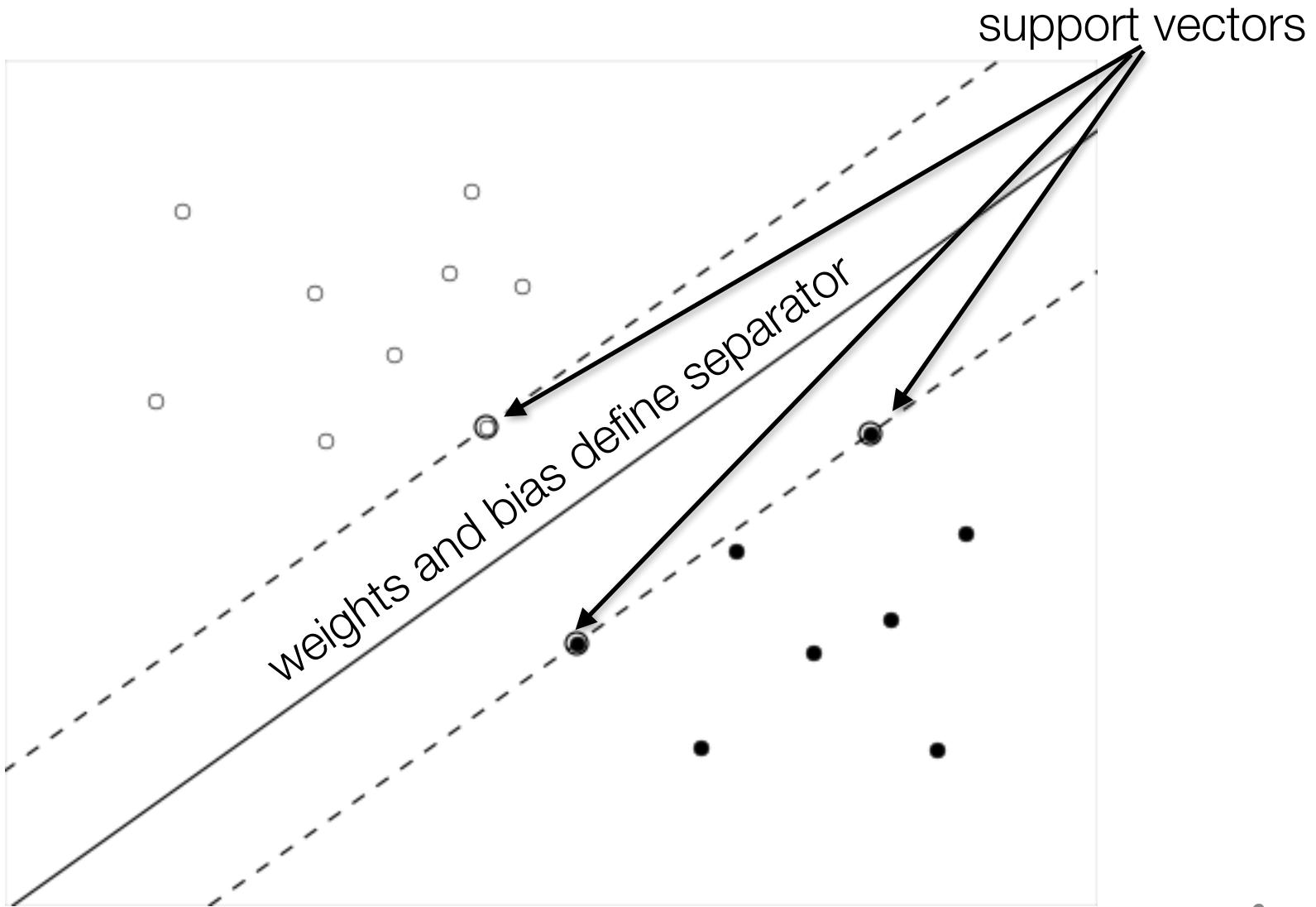
- Linear SVMs
 - Architecture near identical to logistic regression, except:
 - maximize margin
 - constrained optimization : to $y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i$,
 $\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^n \zeta_i$
 - **Self Test:** What are the trained parameters for the linear SVM?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

SVMs Summary: Linear

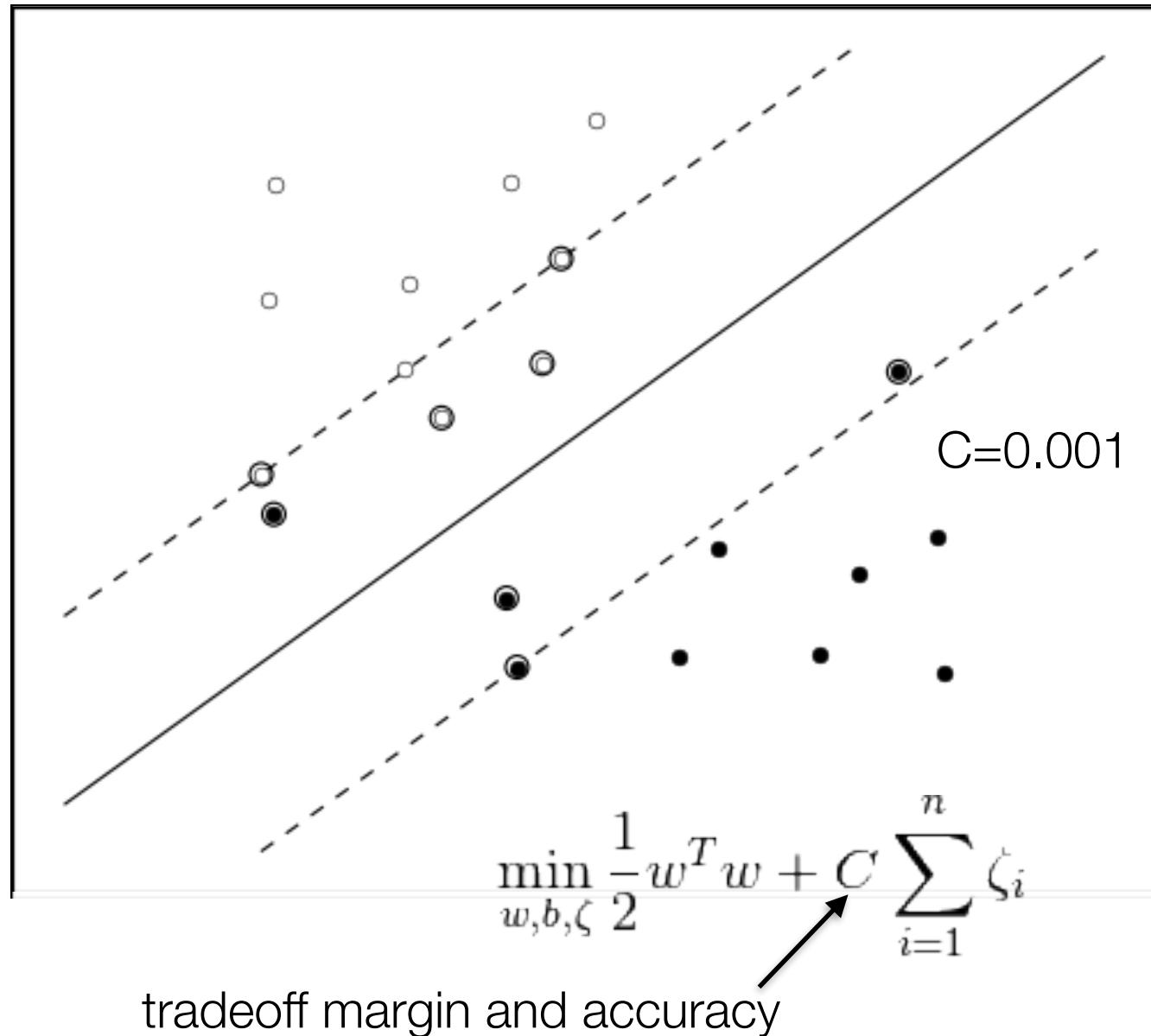
- Linear SVMs
 - Architecture near identical to logistic regression, except:
 - maximize margin
 - constrained optimization
 - Trained Parameters:
 - bias and weights for each class
 - support vectors chosen for margin calculation
 - slack variables for inseparable cases

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

SVMs Summary: Linear



SVMs Summary: Linear



SVMs Summary: non-linear

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

SVMs Summary: non-linear

- Popular Kernels

- polynomial

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

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

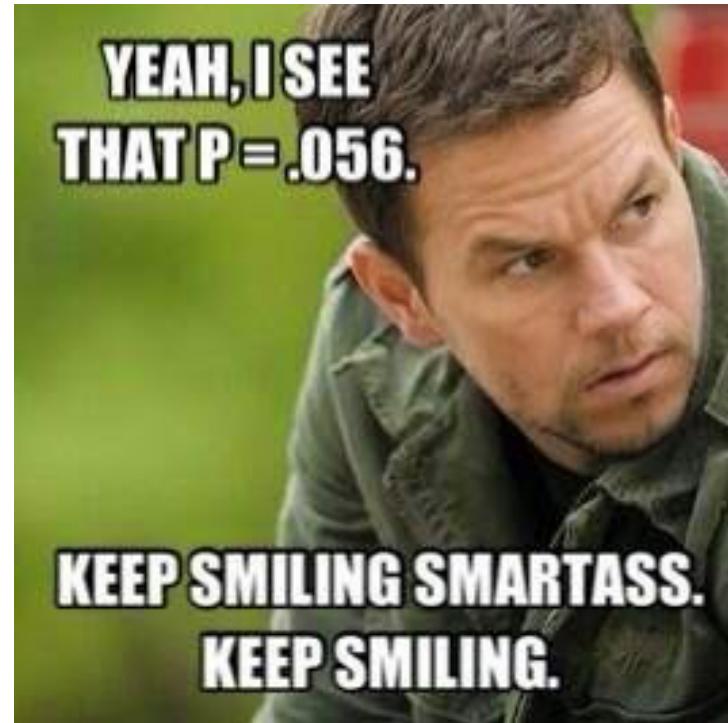
SVM parameterization

svm_gui.py



The End of SVMs

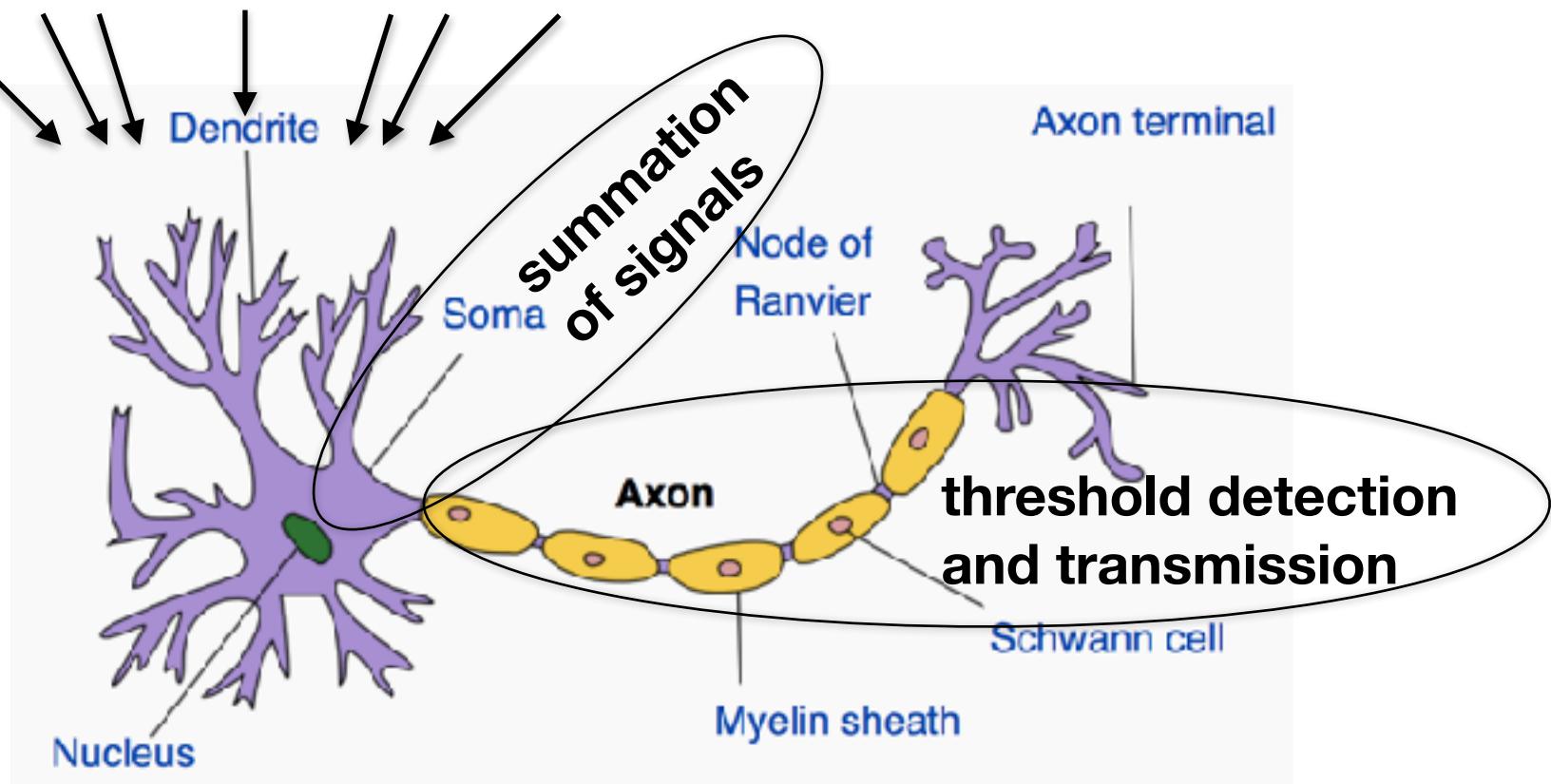
A history of Neural Networks



Neurons

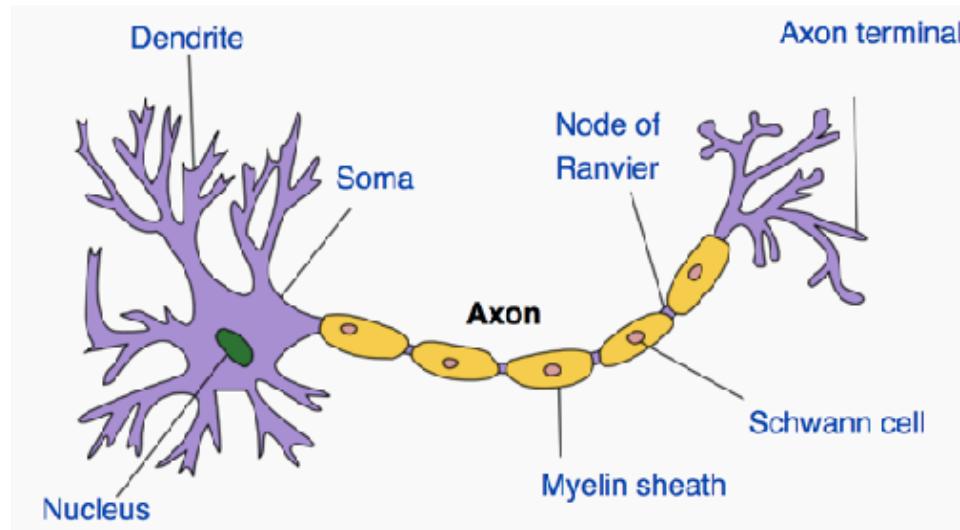
- From biology to modeling:

input from neighboring neurons

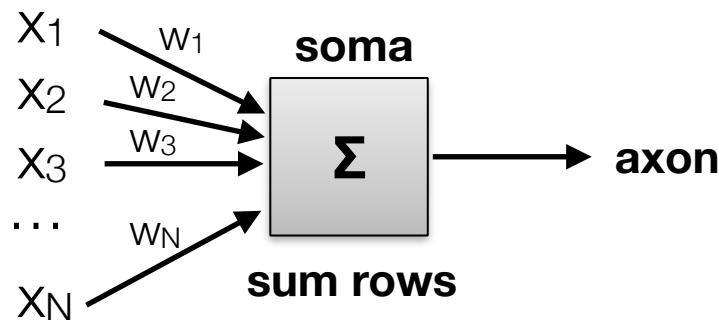


Neurons

- McCulloch and Pitts, 1943



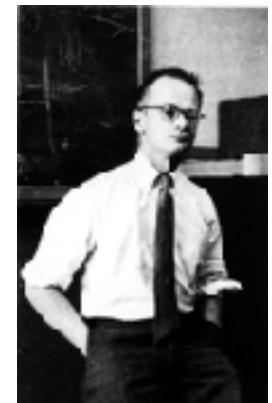
dendrite



input



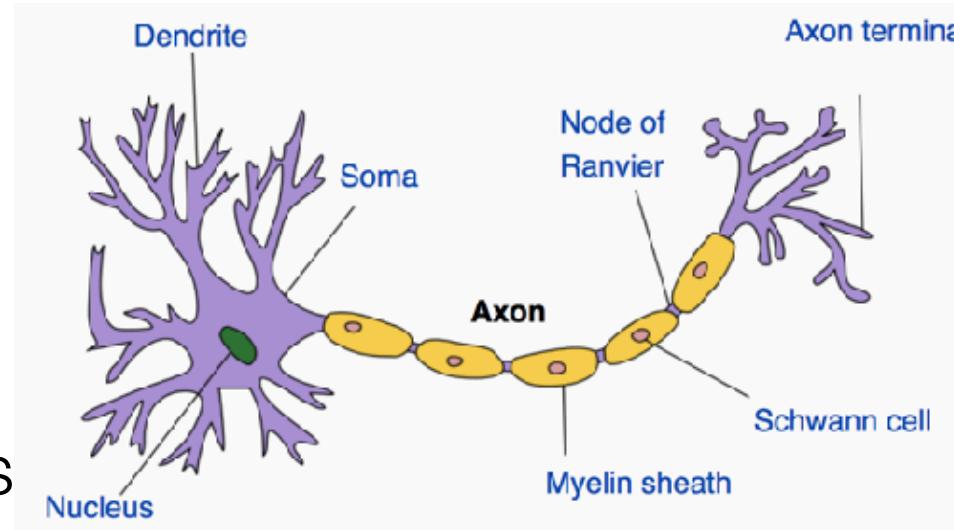
Warren McCulloch



Walter Pitts

Neurons

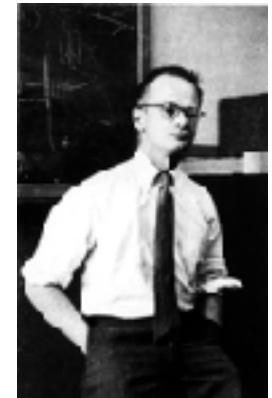
- McCulloch and Pitts, 1943
- Donald Hebb, 1949
 - Hebb's Law: close neurons fire together
 - synaptic transmission
 - basis of neural pathways



Donald O. Hebb



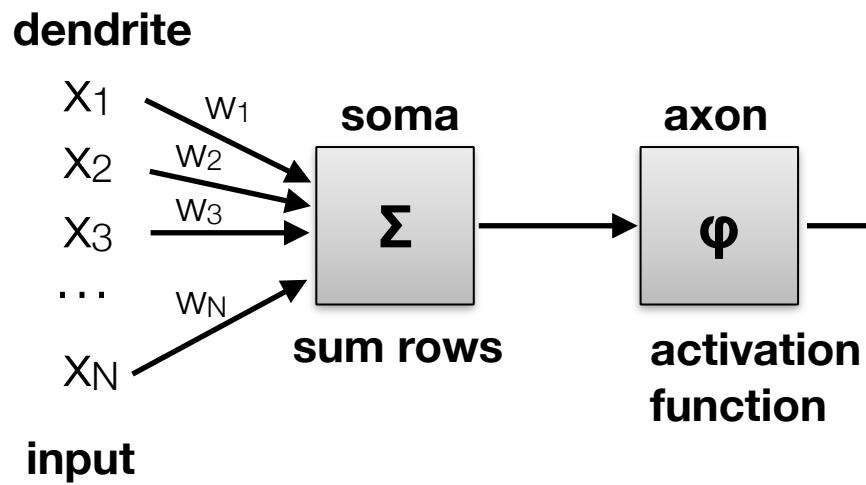
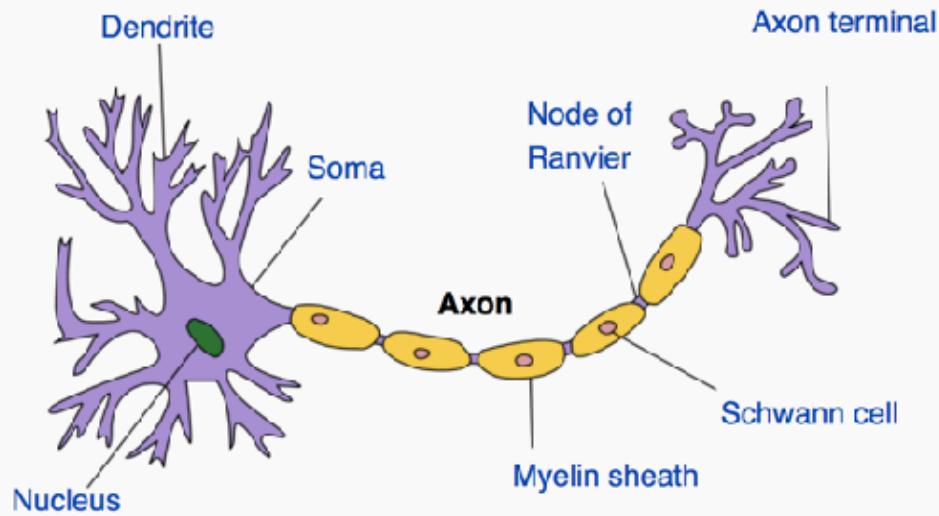
Warren McCulloch



Walter Pitts

Neurons

- Rosenblatt's perceptron, 1957



hard limit



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

linear



$$a = z$$

sigmoid

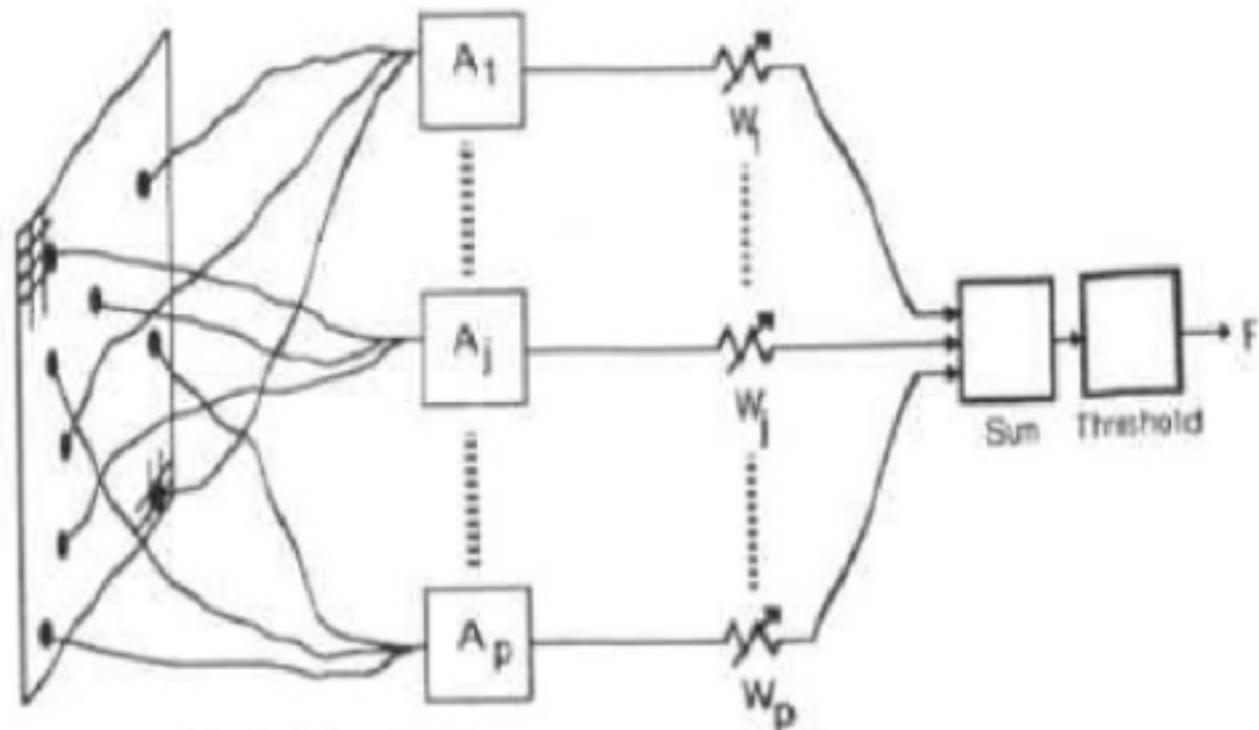


$$a = \frac{1}{1+\exp(-z)}$$



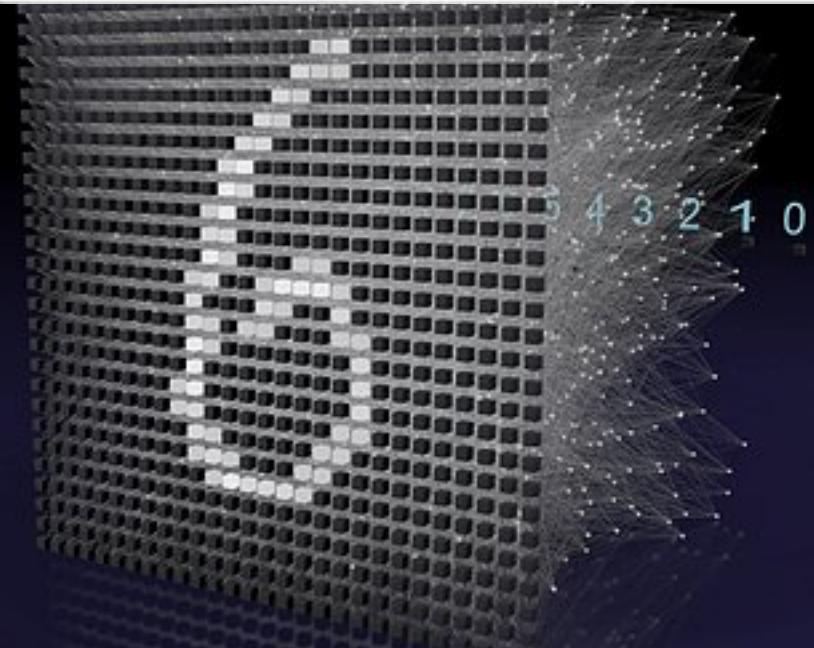
Frank Rosenblatt

The Mark 1 |



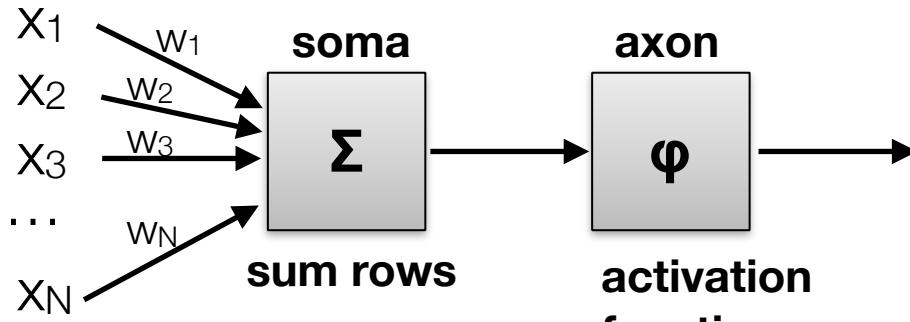
PERCEPTRON

Perceptron Learning Rule:
~Stochastic Gradient Descent

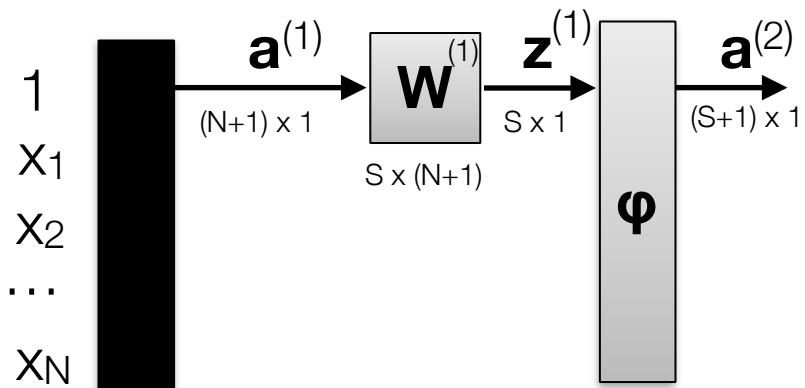


Neurons

dendrite



input



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

$$z = Wa^{(1)}$$

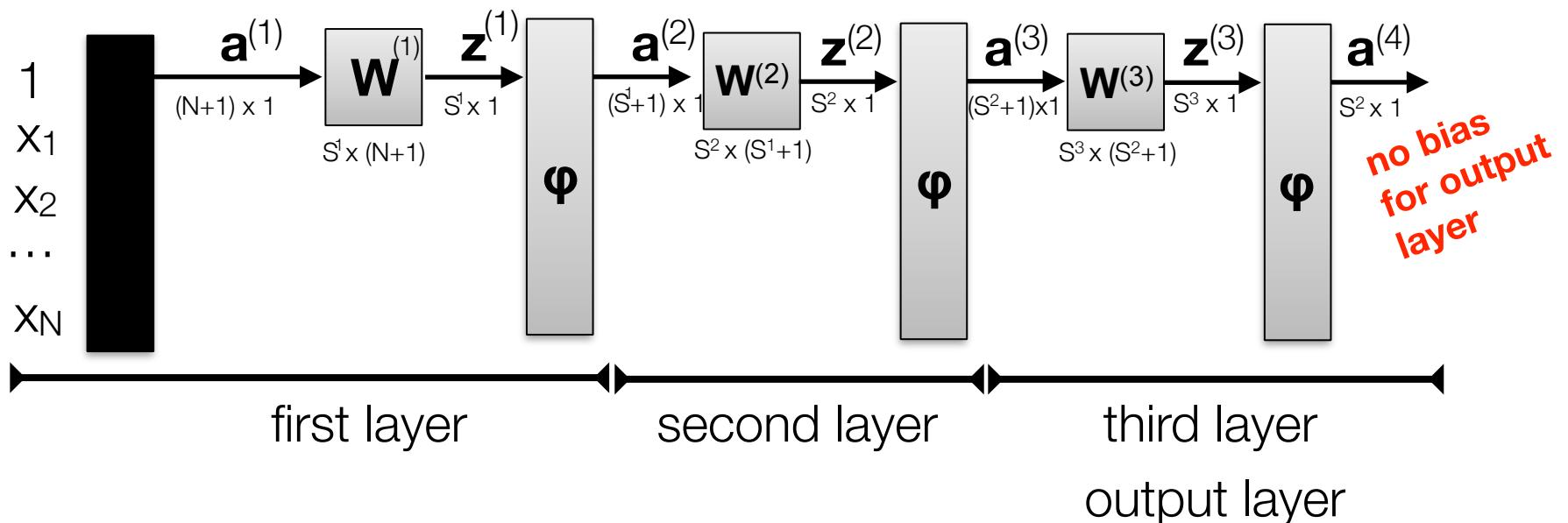
$$W = \begin{bmatrix} W_{1,2} & W_{1,3} & W_{1,4} & \dots & W_{1,N+1} \\ \vdots & & & & \\ W_{S,2} & W_{S,3} & W_{S,4} & \dots & W_{S,N+1} \end{bmatrix}$$

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

- **Some notation**

- need bias term
- matrix representation
- multiple layers

Multiple layers notation



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

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

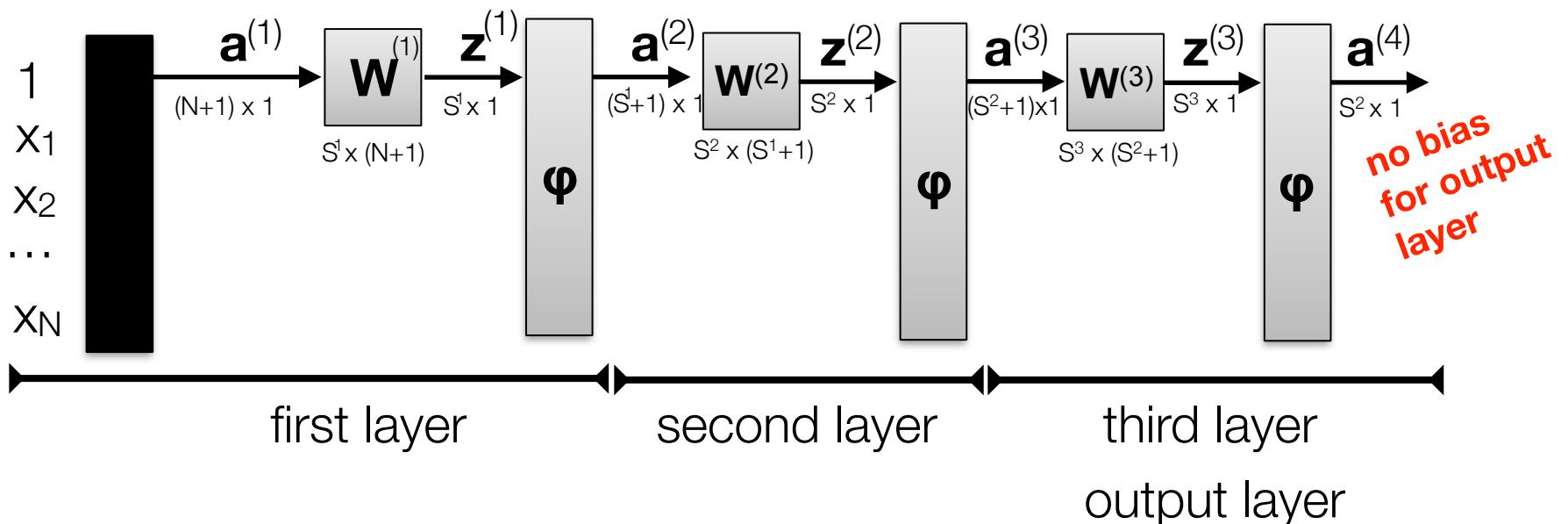
$$\mathbf{z}^{(L)} = \mathbf{W}^{(L)} \mathbf{a}^{(L)}$$

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

Google Alert:

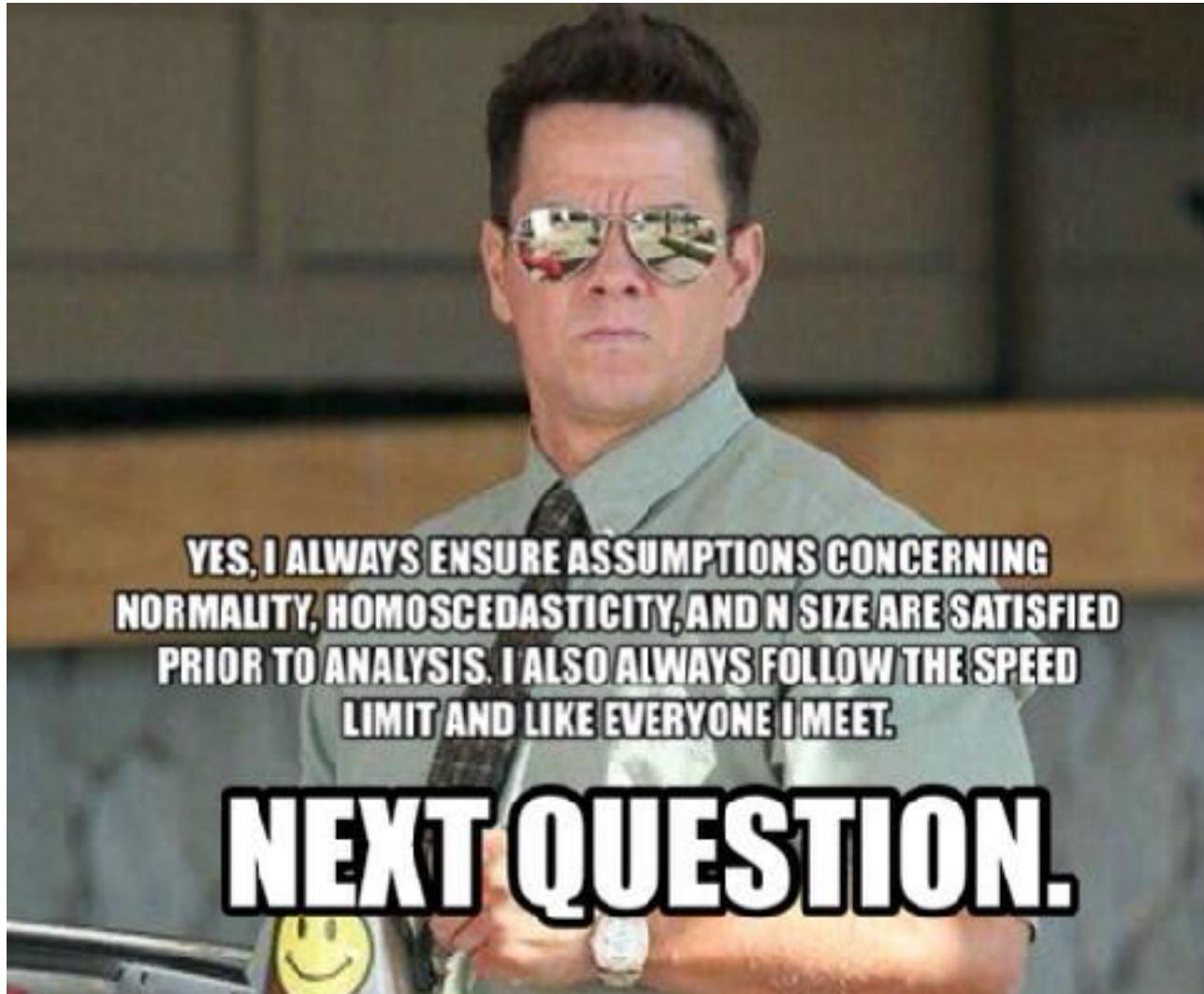
you will also see implementations where \mathbf{W} is a column vector, \mathbf{w}

Multiple layers notation



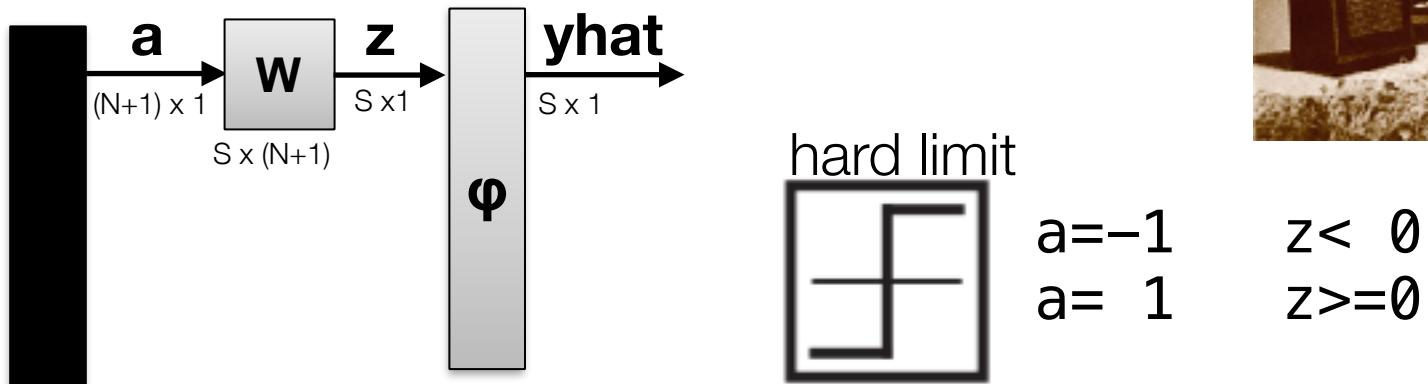
- **Self test:** How many parameters need to be trained in the above network?
 - A. $[(N+1) \times S^1] + [(S^1+1) \times S^2] + [(S^2+1) \times S^3]$
 - B. $|W^{(1)}| + |W^{(2)}| + |W^{(3)}|$
 - C. can't determine from diagram
 - D. it depends on the sizes of intermediate variables, $z^{(i)}$

Training Neural Network Architectures



Simple Architectures

- Rosenblatt's perceptron, 1957

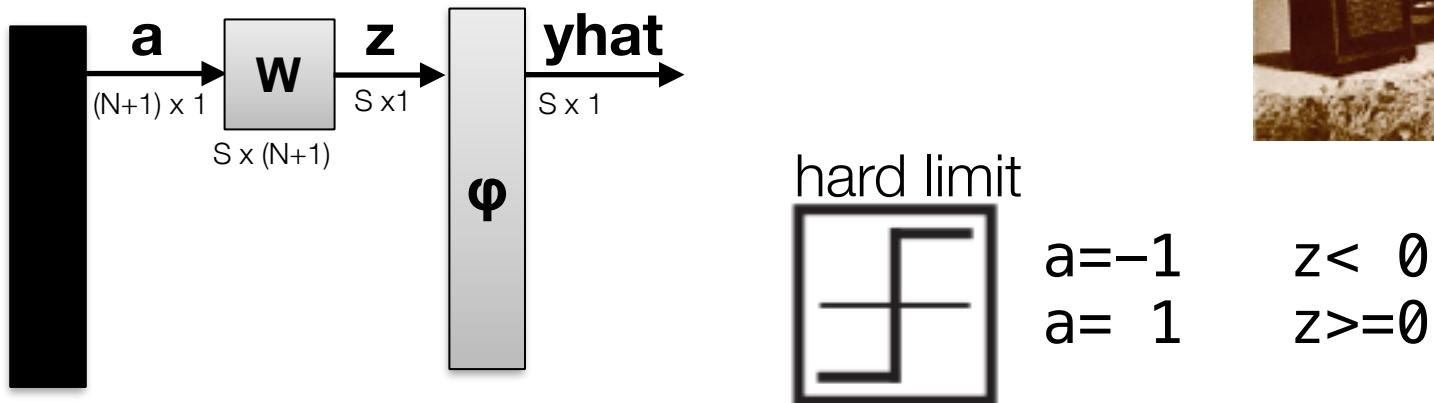


Self Test - If this is a binary classification problem, how large is S ?

- Can't determine
- 2
- 1
- N

Simple Architectures

- Rosenblatt's perceptron, 1957



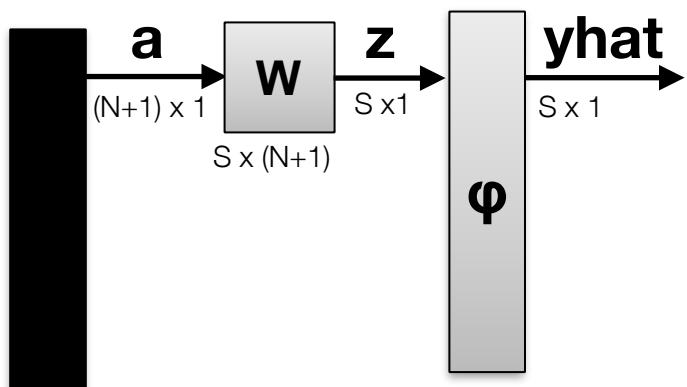
Need objective Function, minimize MSE

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

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

Simple Architectures

- Adaline network, Widrow and Hoff, 1960



$$a=z$$

Marcian “Ted” Hoff



Bernard Widrow

Objective Function, minimize MSE

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

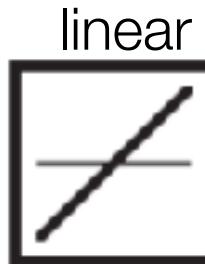
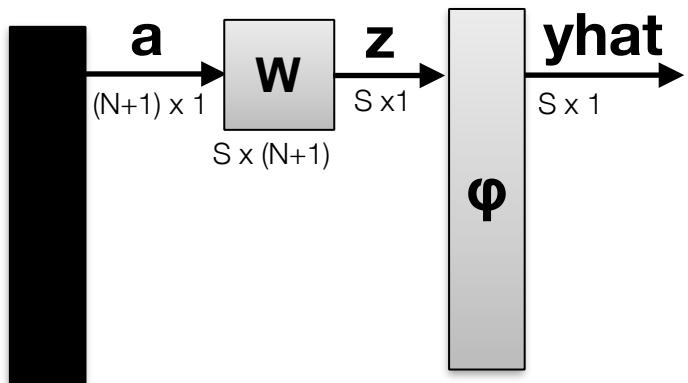
new objective function becomes: $J(W) = \sum_i^M (\mathbf{y}^{(i)} - \mathbf{W} \cdot \mathbf{x}^{(i)})^2$

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

- Adaline network, Widrow and Hoff, 1960



$$a = z$$

Marcian “Ted” Hoff



Bernard Widrow

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

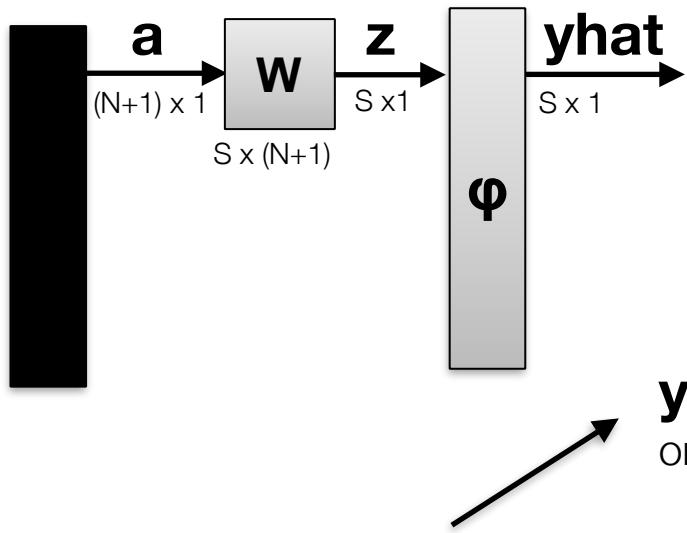
For case $S=1$, this is just solving **linear regression**
and **we have already solved this!**

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

for \mathbf{W} , each row can be solved independently!



Simple Architectures



$$yhat^{(i)} = \Phi(wa^{(i)})$$

output for instance i

$$yhat^{(i)} = \begin{bmatrix} \Phi(w_{row=1}x^{(i)}) \\ \Phi(w_{row=2}x^{(i)}) \\ \Phi(w_{row=3}x^{(i)}) \\ \vdots \\ \Phi(w_{row=C}x^{(i)}) \end{bmatrix}$$

Each class is independently trained!

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

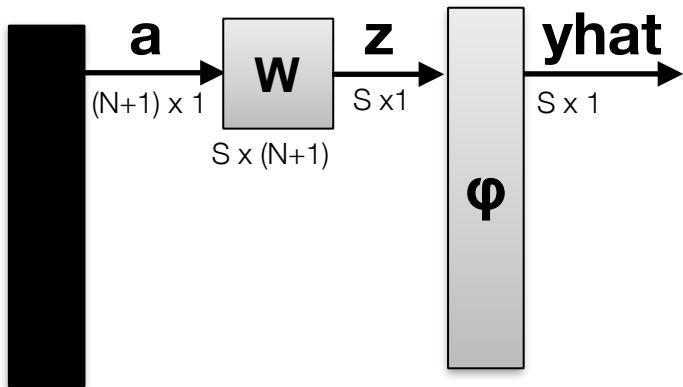
$$w_{row} \leftarrow w_{row} + \eta [X * (y_{row} - \hat{y}_{row})]$$

which is one-versus-all!
we have already solved this!



Simple Architectures

- Modern Perceptron network



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

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 \mathbf{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 $\mathbf{w}_{\text{class}}$ 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



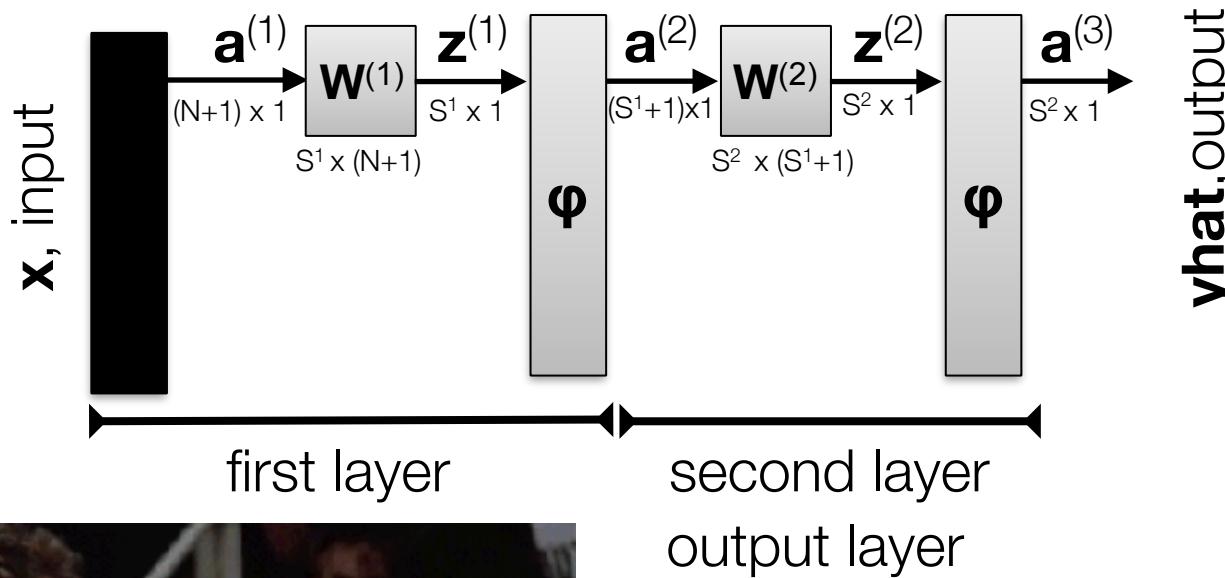
1942-2011

Geoffrey Hinton



More Advanced Architectures: MLP

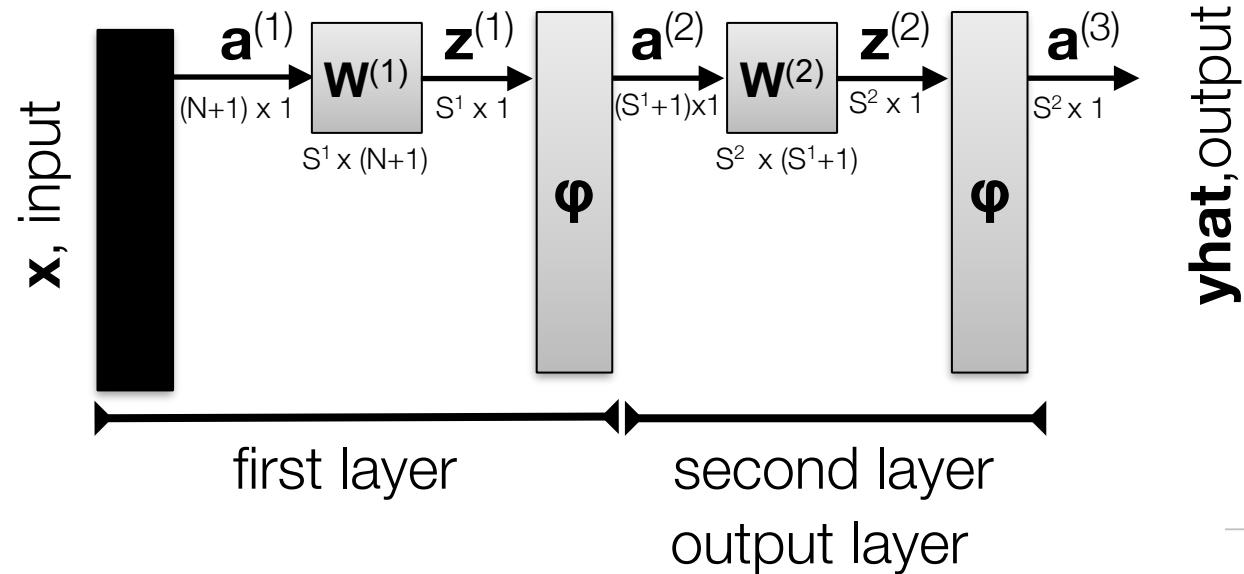
- The multi-layer perceptron (MLP):
 - two layers shown, but could be arbitrarily many layers
 - algorithm is agnostic to number of layers (*kinda*)



each row of $y\hat{}$ is no longer independent of the rows in \mathbf{W}

Back propagation

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



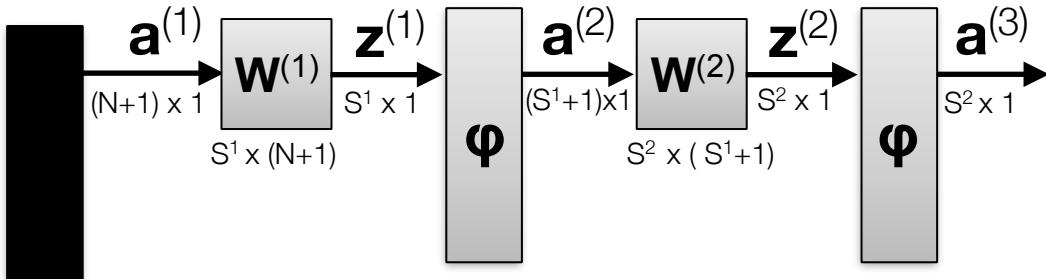
$$J(\mathbf{W}) = \sum_k^M (\mathbf{y}^{(k)} - \mathbf{a}^{(L)})^2$$

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



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 next time!

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=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

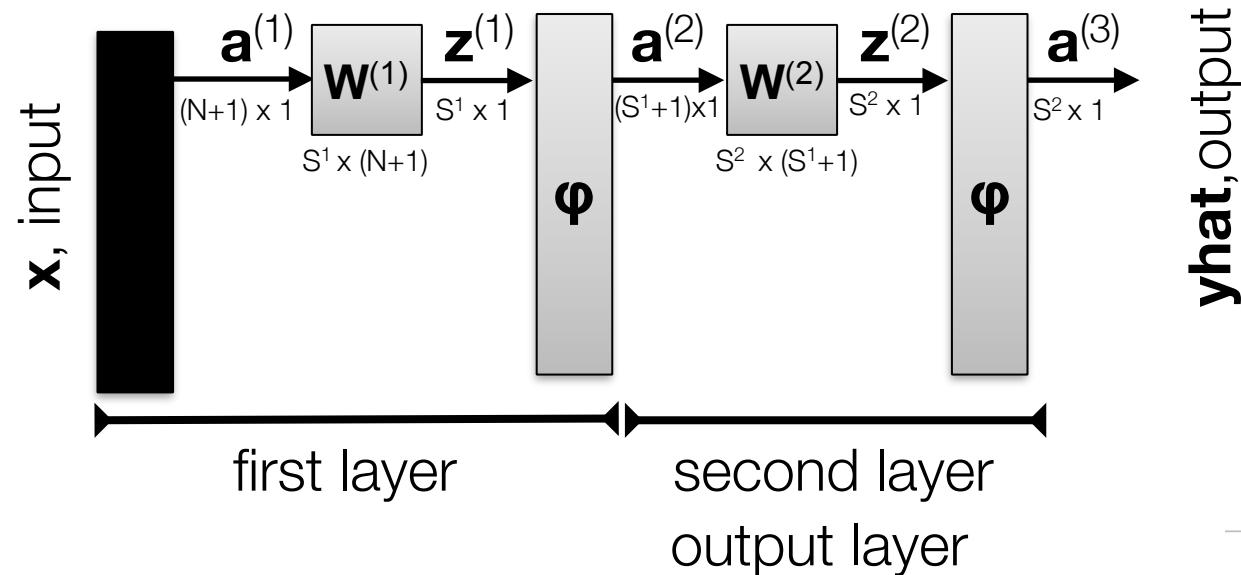
Professor Eric Larson
Multi-Layer Neural Networks

Class Logistics and Agenda

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

Back propagation

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

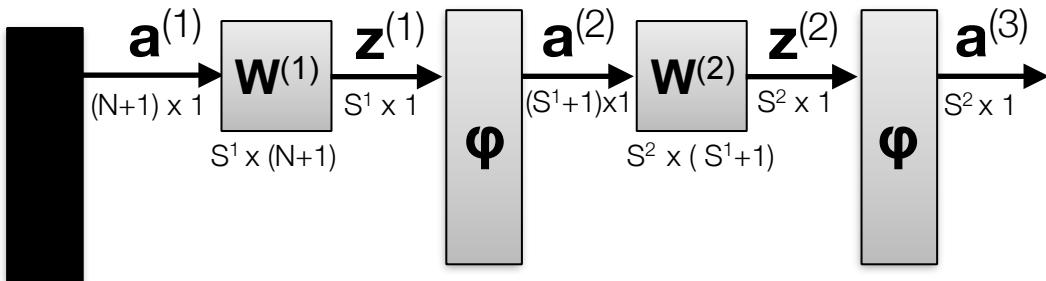


$$J(\mathbf{W}) = \sum_k^M (\mathbf{y}^{(k)} - \mathbf{a}^{(L)})^2$$

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

Back propagation

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

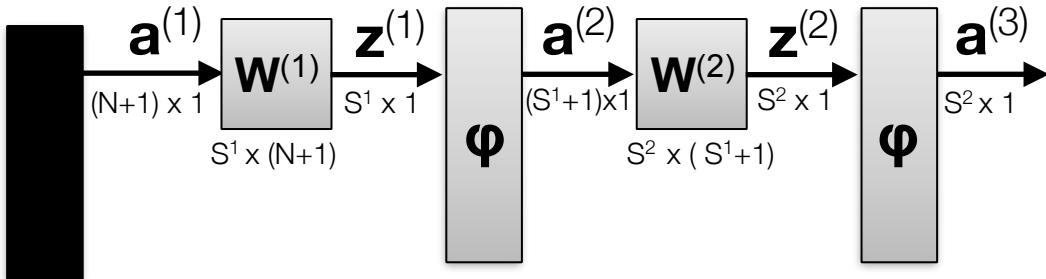


$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta \frac{\partial J(\mathbf{W})}{\partial w_{i,j}^{(l)}}$$

use chain rule:

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

Back propagation



can we use
chain rule
again?

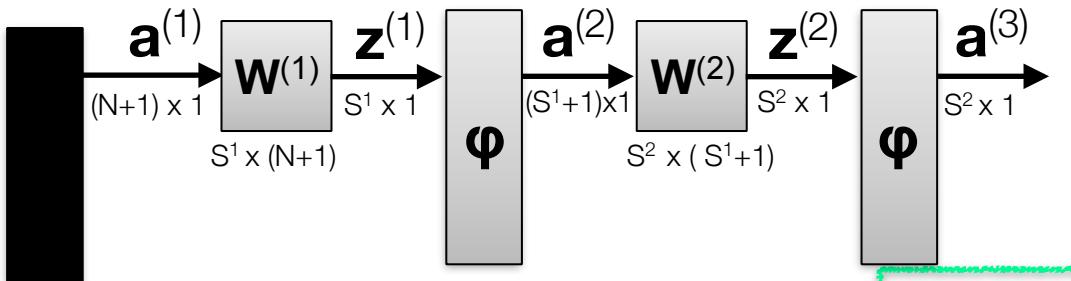
$$\frac{\partial J(\mathbf{W})}{\partial w_{i,j}^{(l)}} = \boxed{\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}}} a_j^{(l)}$$

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

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

Back propagation

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



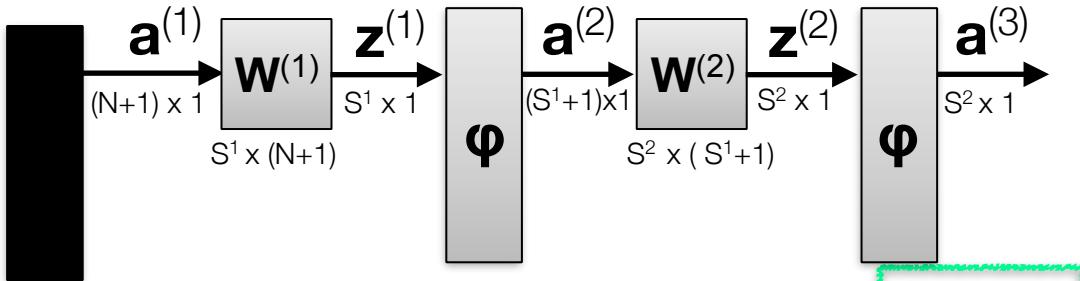
can we get this
gradient?

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

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

Back propagation

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



use chain rule:

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

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

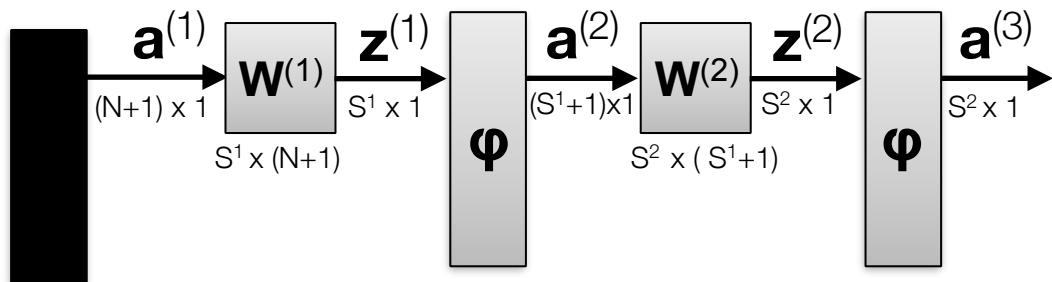
$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}} = \text{diag}[\mathbf{a}^{(l+1)} * (1 - \mathbf{a}^{(l+1)})] \cdot \mathbf{W}^{(l+1)} \frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l+1)}}$$

recurrence relation

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

Back propagation

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



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

one more step

need last layer
gradient:

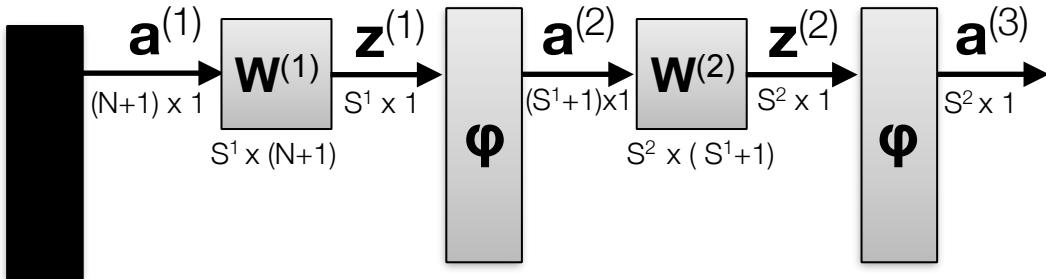
$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(2)}} = \frac{\partial}{\partial \mathbf{z}^{(2)}} (\mathbf{y}^{(k)} - \phi(\mathbf{z}^{(2)}))^2$$

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

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

Back propagation summary

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



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

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

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

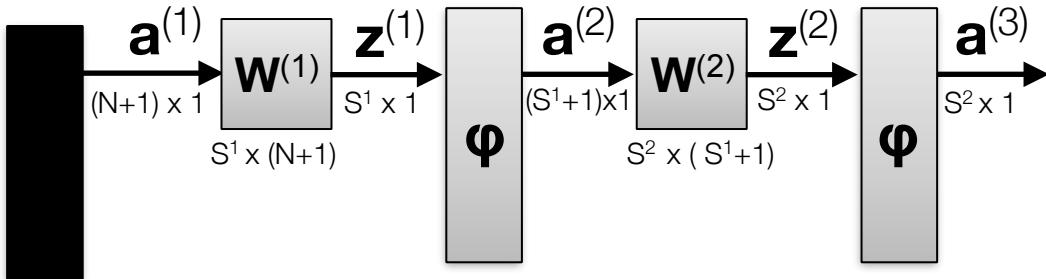
$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(2)}} = -2(\mathbf{y}^{(k)} - \mathbf{a}^{(3)}) * \mathbf{a}^{(3)} * (1 - \mathbf{a}^{(3)})$$

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

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

Back propagation summary

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



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

- **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.

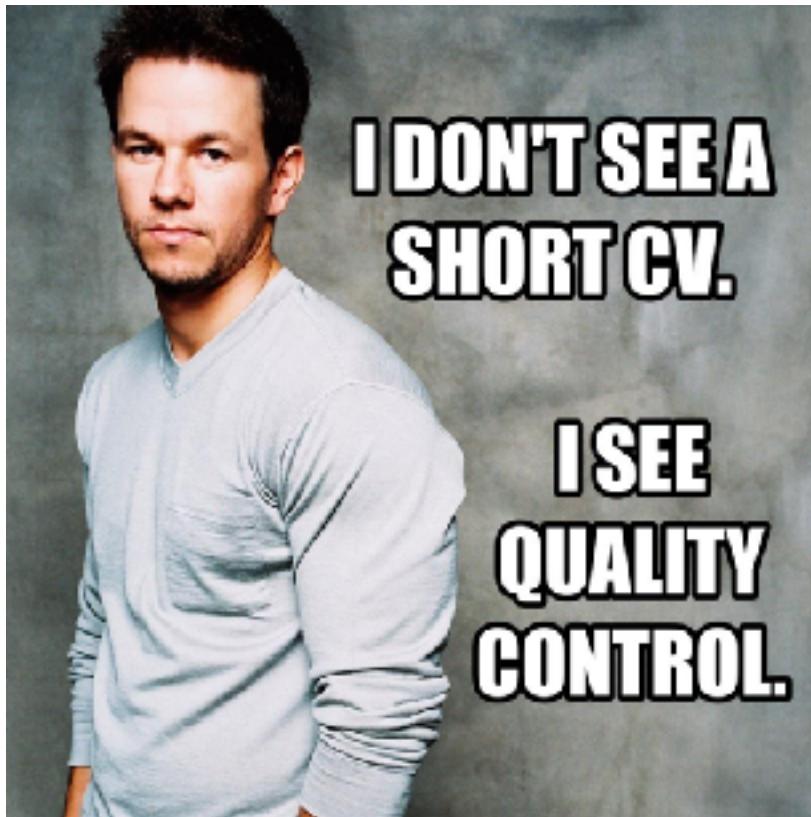
- A. True
- B. False

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

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

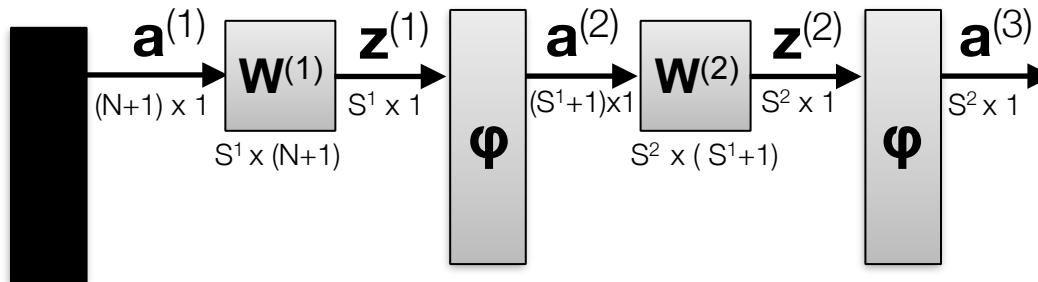
$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \frac{\partial J(\mathbf{W}^{(l)})}{\partial \mathbf{z}^{(l)}} \cdot \mathbf{a}^{(l)}$$

Programming Multi-layer Neural Networks



Guided Example

Back propagation implementation



1. Forward propagate to get \mathbf{z} , \mathbf{a} for all layers

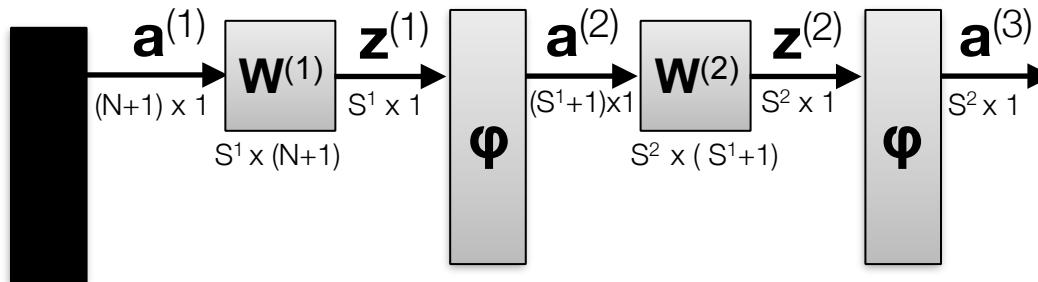
```
# feedforward all instances
A1, Z1, A2, Z2, A3 = self._feedforward(X_data, self.W1, self.W2)
```



```
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

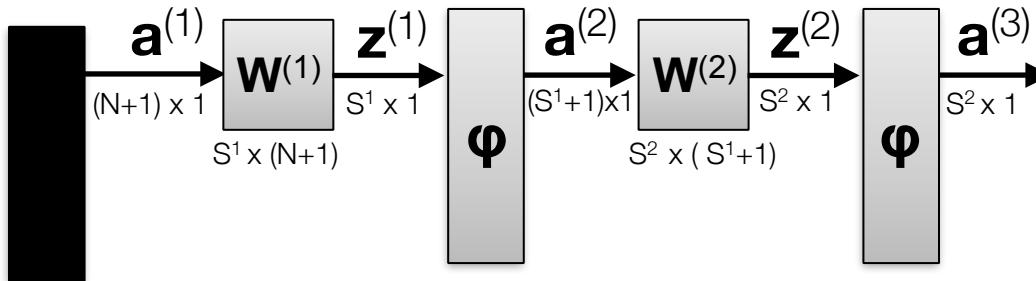


1. Forward propagate to get \mathbf{z} , \mathbf{a} for all layers for each $\mathbf{y}^{(k)}$
2. Get final layer gradient $\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(2)}} = -2(\mathbf{y}^{(k)} - \mathbf{a}^{(3)}) * \mathbf{a}^{(3)} * (1 - \mathbf{a}^{(3)})$
3. Update back propagation variables $\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}} = \text{diag}[\mathbf{a}^{(l+1)} * (1 - \mathbf{a}^{(l+1)})] \cdot \mathbf{W}^{(l+1)} \frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l+1)}}$

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 \mathbf{z} , \mathbf{a} for all layers
2. Get final layer gradient
3. Update back propagation variables

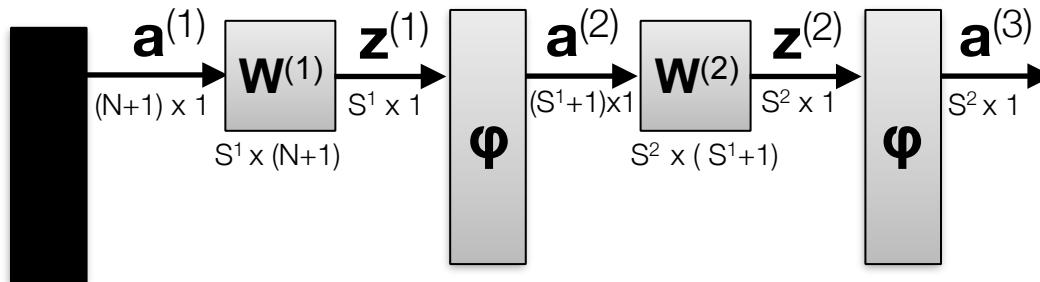
$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \frac{\partial J(\mathbf{W}^{(l)})}{\partial \mathbf{z}^{(l)}} \cdot \mathbf{a}^{(l)}$$

```
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 \mathbf{z} , \mathbf{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)

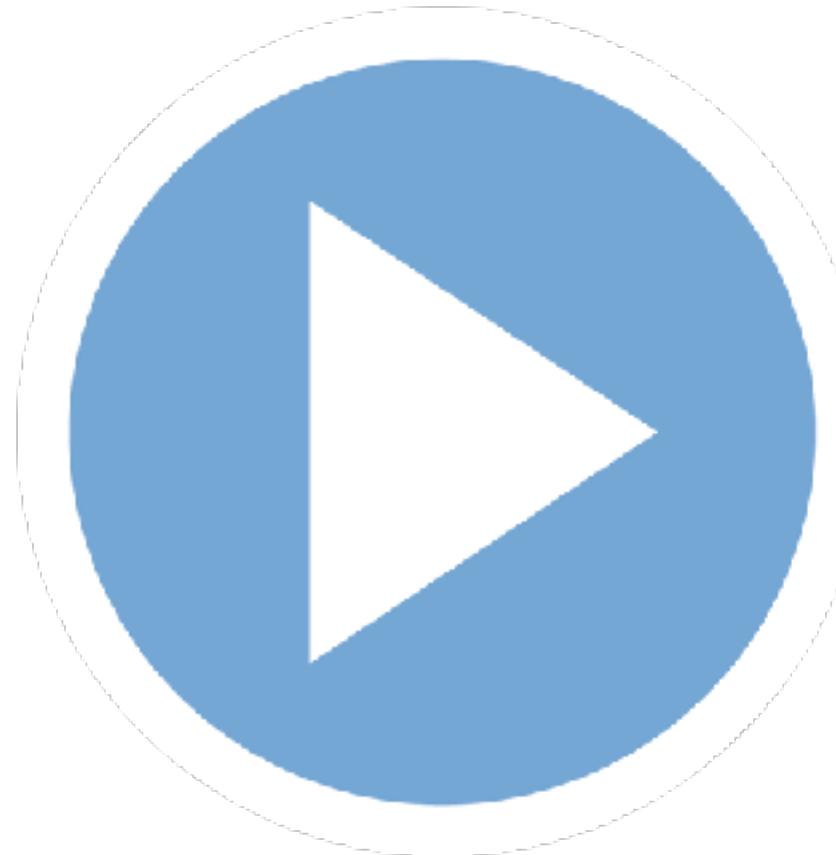
# compute gradient via backpropagation
grad1, grad2 = self._get_gradient(A1=A1, A2=A2,
                                    A3=A3, z1=z1,
                                    Y_enc=Y_enc,
                                    W1=self.W1, W2=self.W2)

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

putting it all together

Two Layer Perceptron

with regularization
vectorization

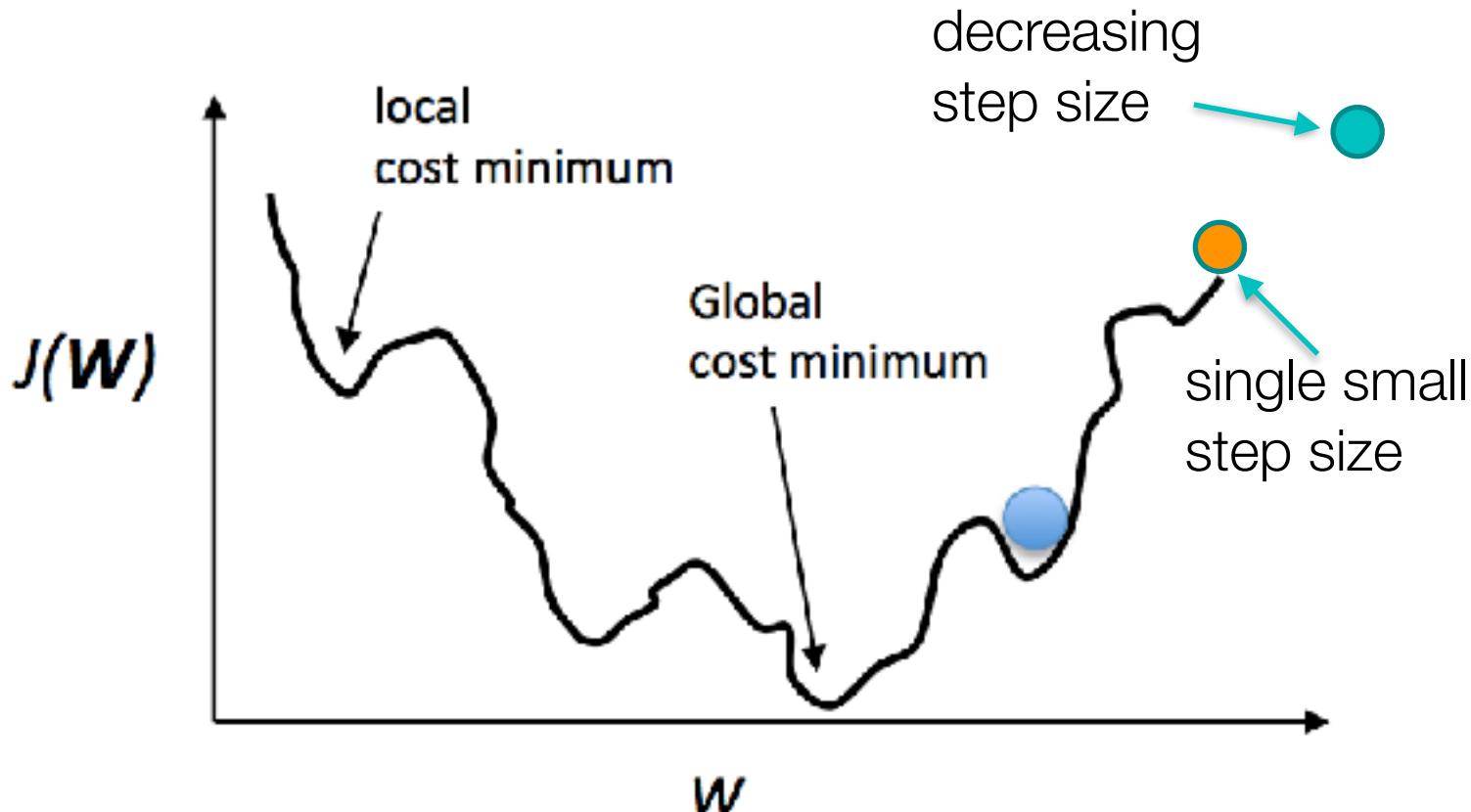


Problems with Advanced Architectures

- 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

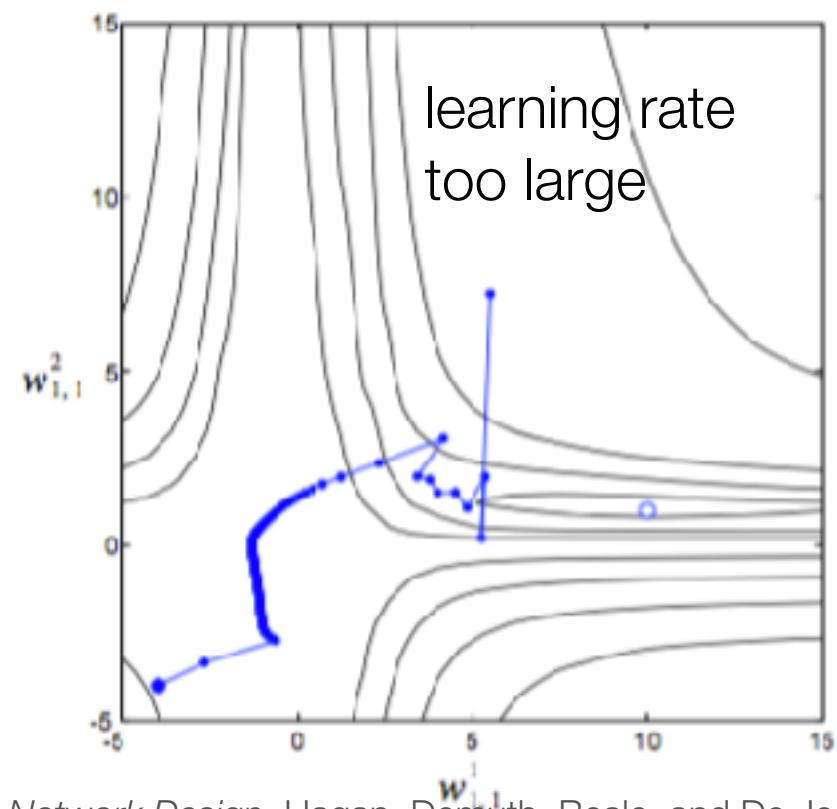
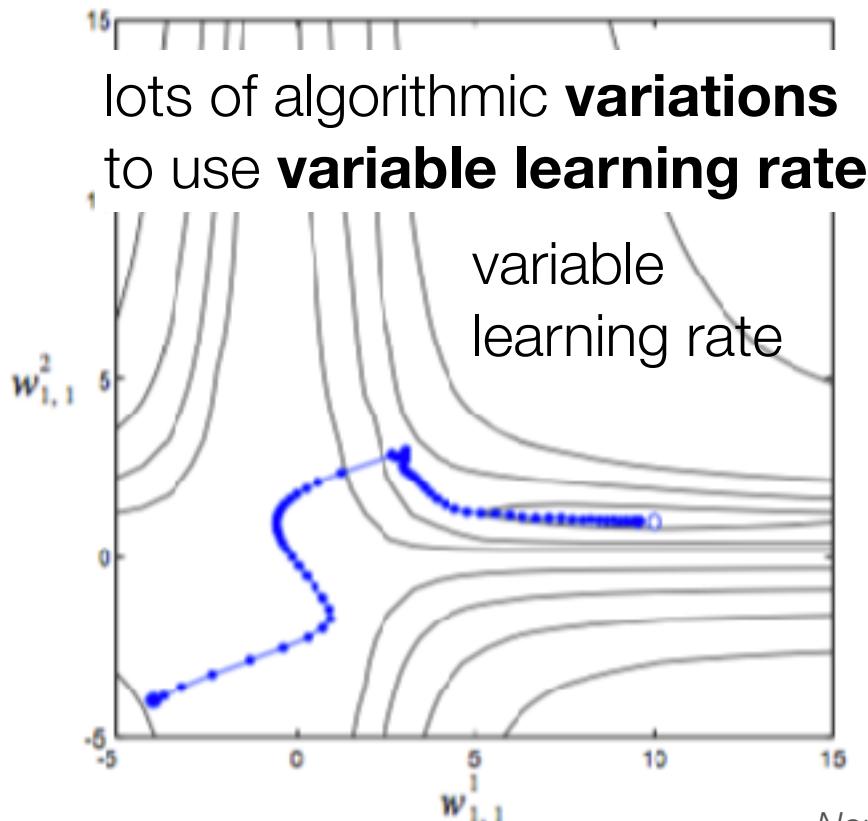
Problems with Advanced Architectures

- 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

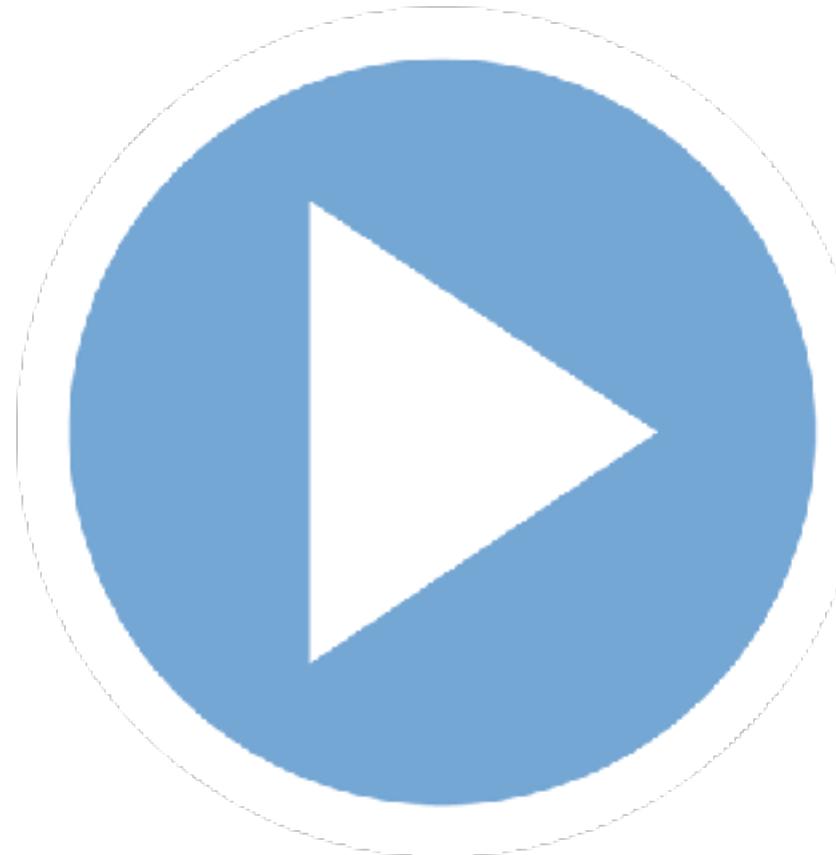
comparison:

mini-batch

momentum

decreased learning

L-BFGS



End of Session

- Have a good spring break
- Don't forget about the final flipped assignment!!!

Lecture Notes for Machine Learning in Python

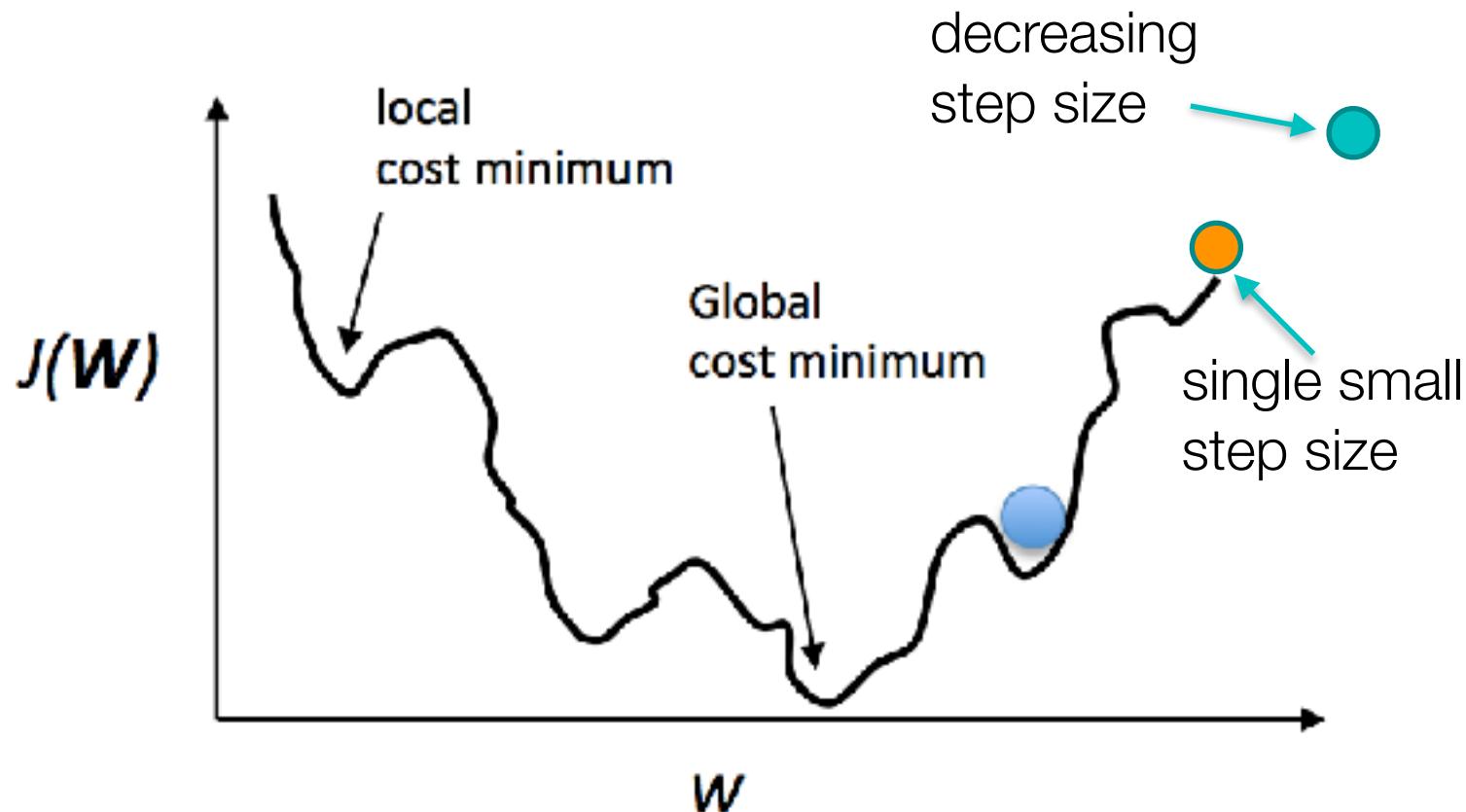
Professor Eric Larson
More Practical MLP

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:** end of NN, start ensemble classifiers

Last Time: Problems with Advanced Architectures

- Space is no longer convex
 - **Two solutions:**
 - “cool down” by decreasing step size for higher iterations
 - keep going in direction of previous gradient (to some degree)



Two Layer Perceptron

comparison:

mini-batch

momentum

decreased learning

L-BFGS

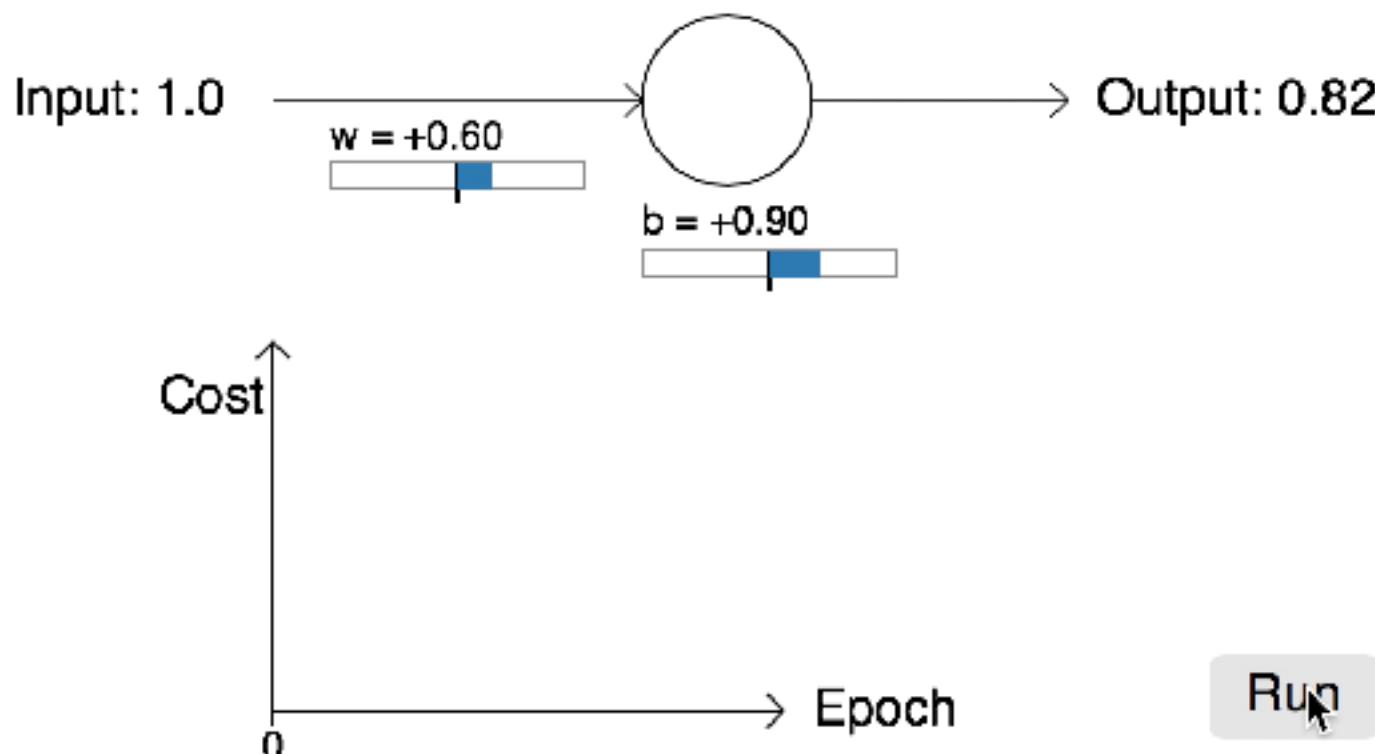


Practical Implementation of Architectures

- 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

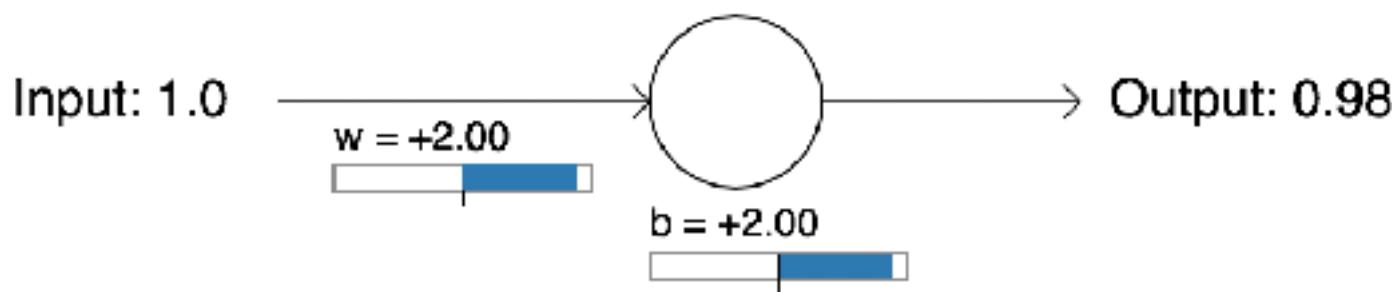


Practical Implementation of Architectures

- 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



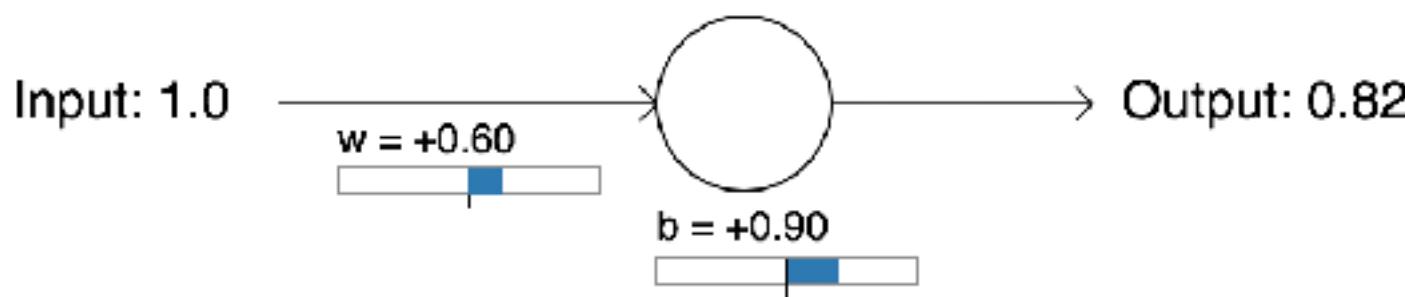
Run

Practical Implementation of Architectures

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

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

speeds up
initial training



Run

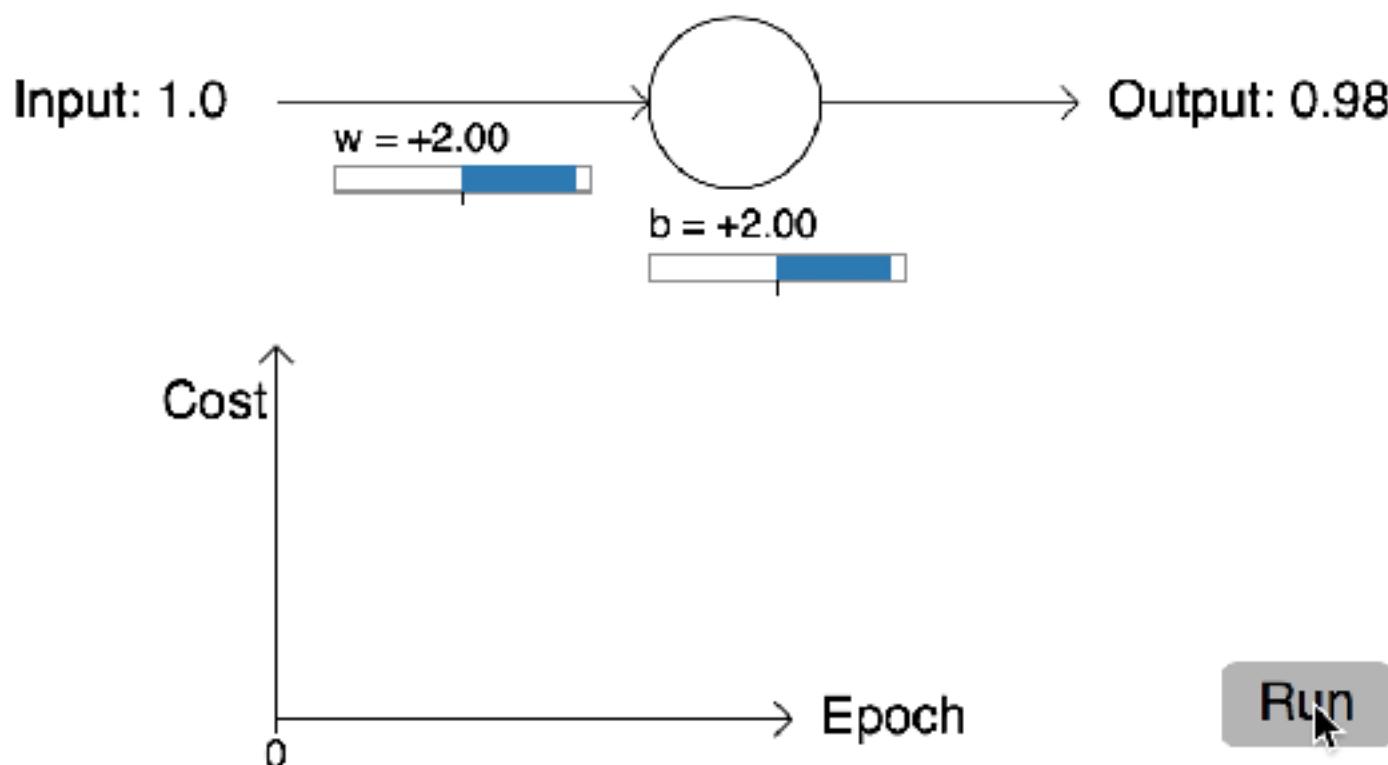
61

Practical Implementation of Architectures

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

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

speeds up
initial training



Practical Implementation of Architectures

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

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

speeds up
initial training

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

bp-5
63

Practical Implementation of Architectures

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

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

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

```
# vectorized backpropagation
sigma3 = (A3-Y_enc) # <- this is only line
sigma2 = (W2.T @ sigma3)*A2*(1-A2)
```

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

```
grad1 = sigma2[1:,:]*A1
grad2 = sigma3 @ A2.T
```

new update

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

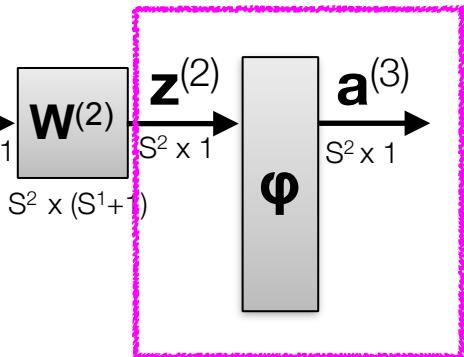
```
# vectorized backpropagation
sigma3 = -2*(Y_enc-A3)*A3*(1-A3)
sigma2 = (W2.T @ sigma3)*A2*(1-A2)
```

```
grad1 = sigma2[1:,:]*A1
grad2 = sigma3 @ A2.T
```

bp-5
64

Practical Implementation of Architectures

- A new nonlinearity: **softmax**



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

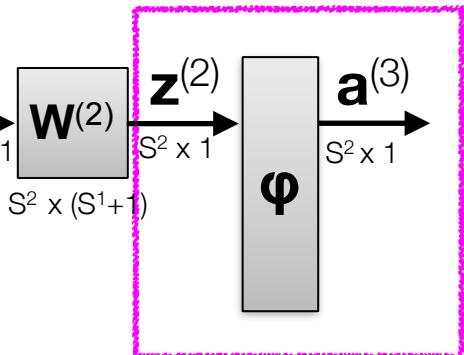
instead of final layer sigmoid!

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

if $J(\mathbf{W}) = -\sum y_j \ln(a_j^{(L)})$, then update equations are **identical** to that of Cross Entropy.

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

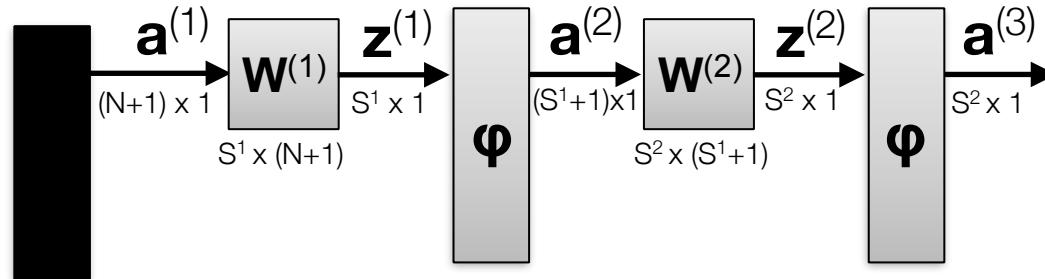
$$\frac{\partial \phi(\mathbf{z}^{(i)})}{\partial \mathbf{z}^{(i)}} = \begin{cases} 1, & \text{if } \mathbf{z}^{(i)} > 0 \\ 0, & \text{else} \end{cases}$$

Practical Implementation of Architectures

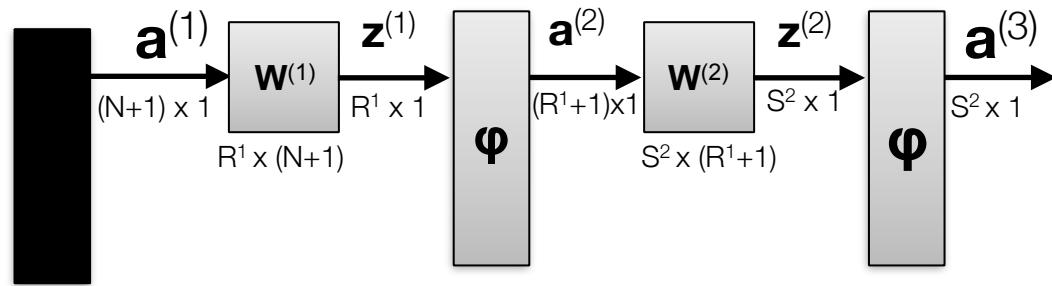
- Beyond L2 regularization
 - dropout
 - expansion

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

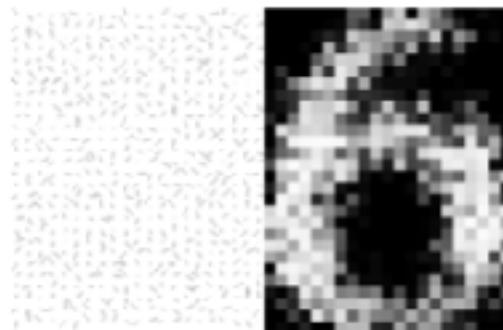
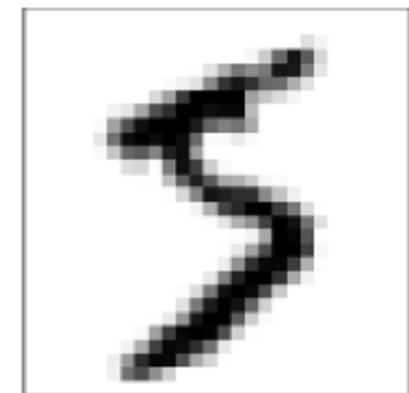
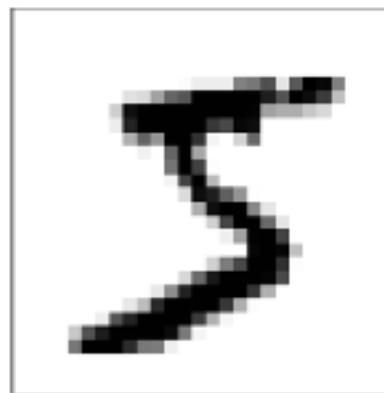


Why does this work?

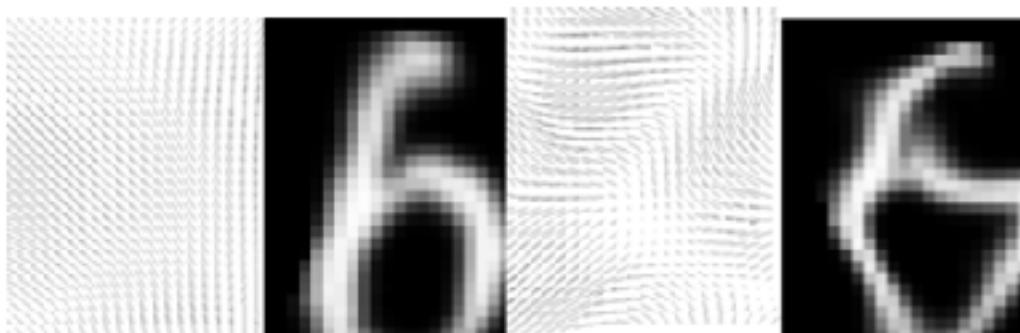
Practical Implementation of Architectures

- Expansion

- get more data
- perturb your data



Neural Networks and Deep Learning,
Michael Nielson, 2015



98.4% → 99.3%

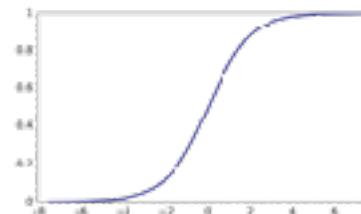
Practical Implementation of Architectures

- 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 $\varepsilon < \sum < 1 - \varepsilon$ 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 = $1/\sqrt{\mathbf{W}_{\text{num_elements_in_row}}}$

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 hidden layer can solve any problem with enough data
 - but... it might be better to have even more layers for decreased computation and generalizability

Two Layer Perceptron

Practical implementations



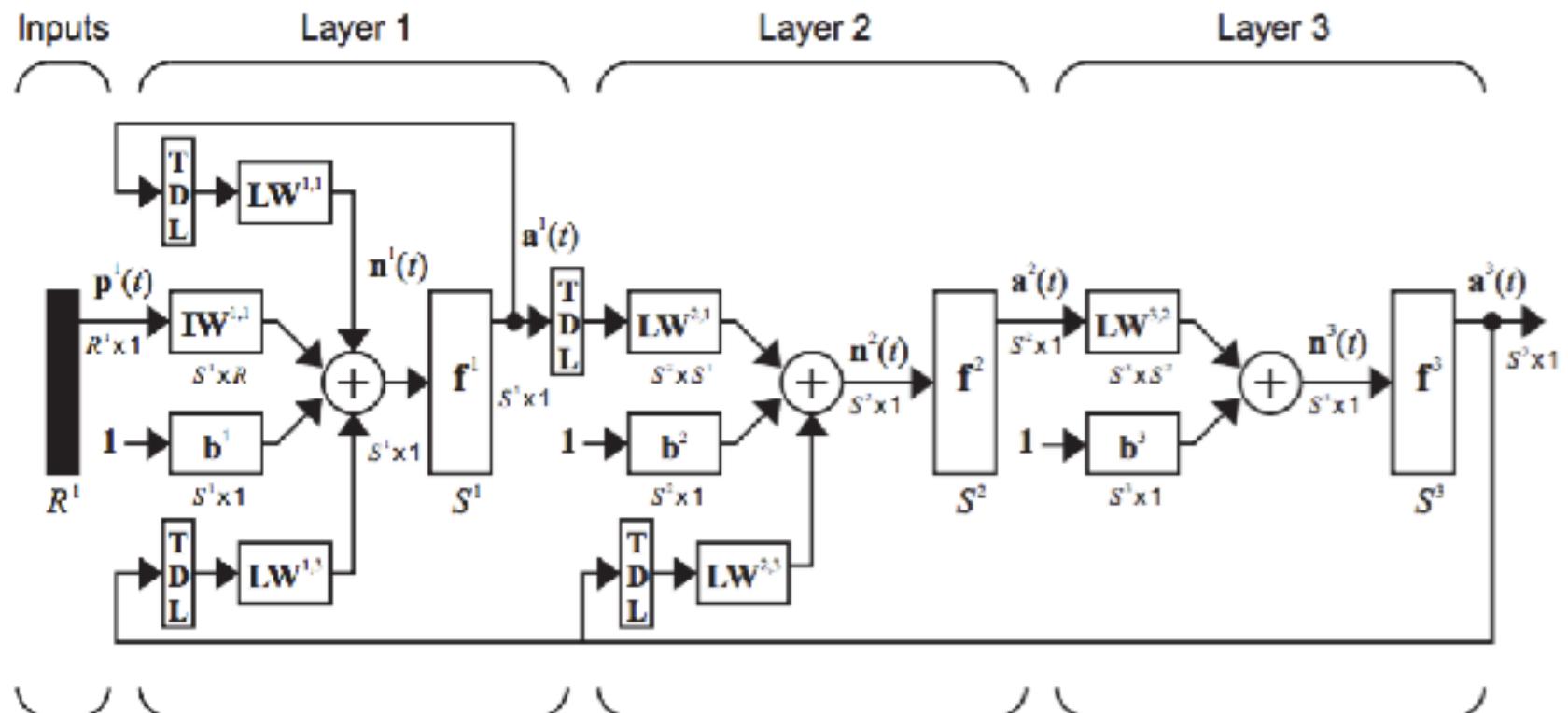
End of Session

- End of neural networks for a week
- Then we return with a vengeance

Back up slides

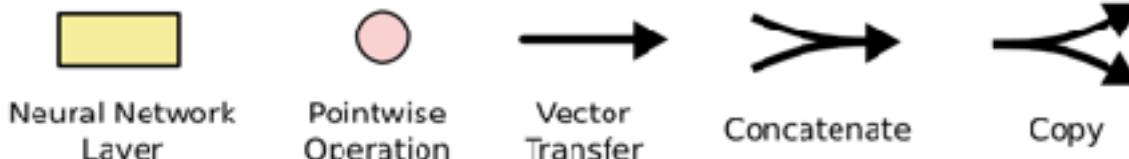
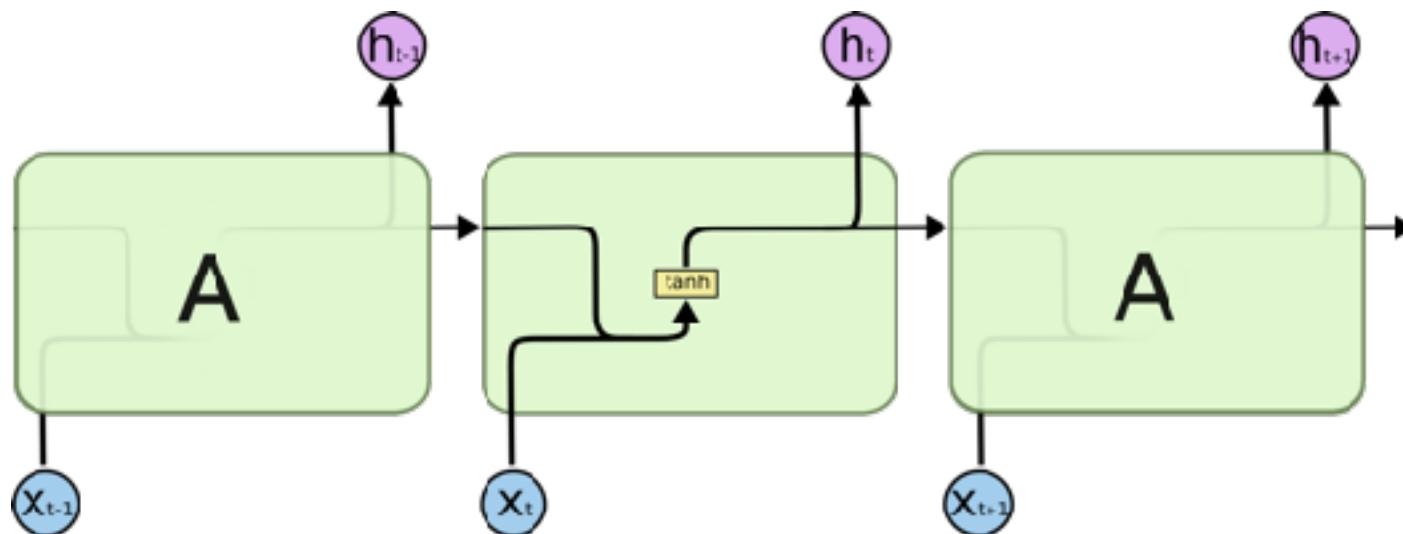
More Advanced Architectures

- 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**



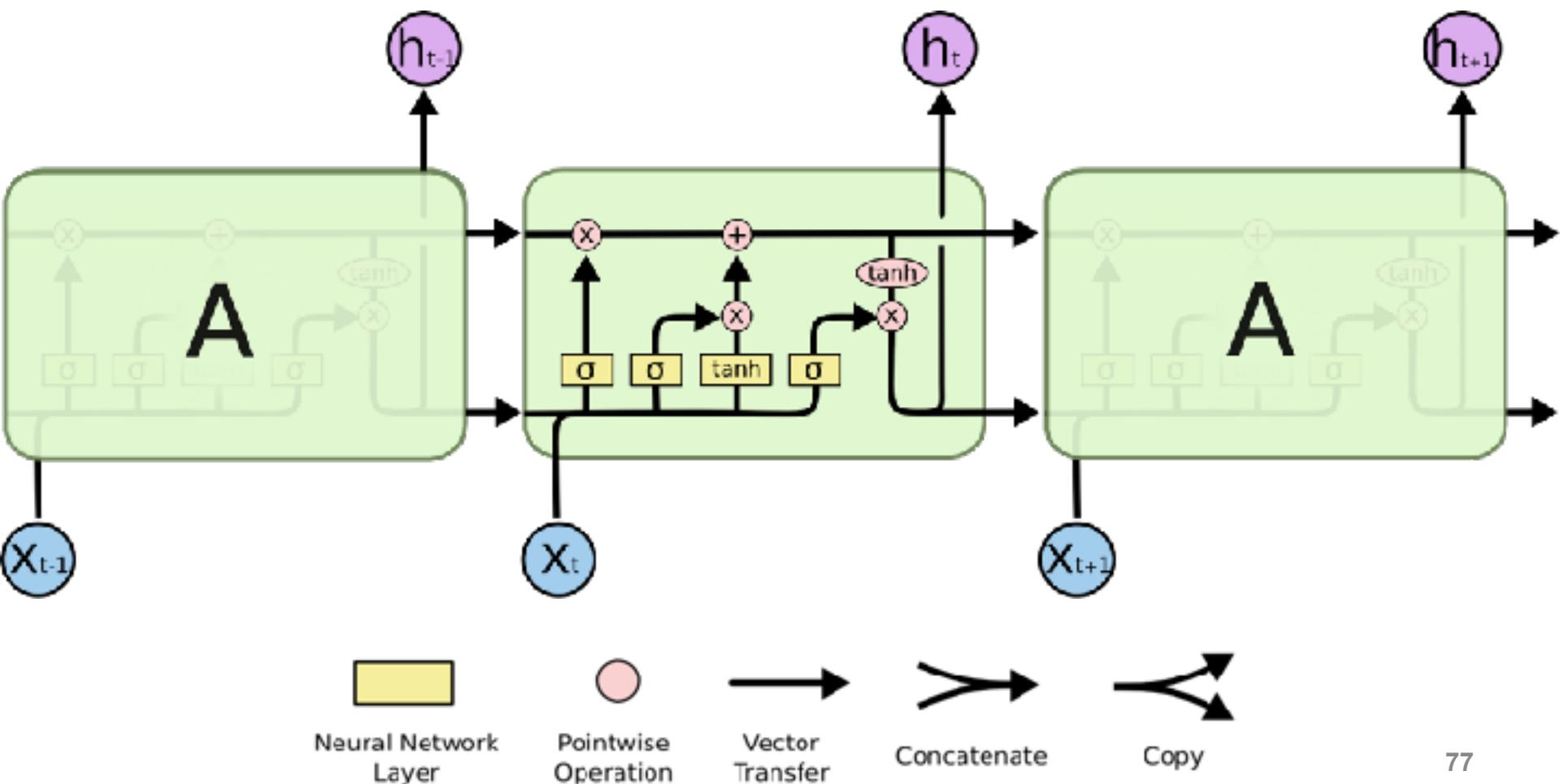
More Advanced Architectures

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



More Advanced Architectures

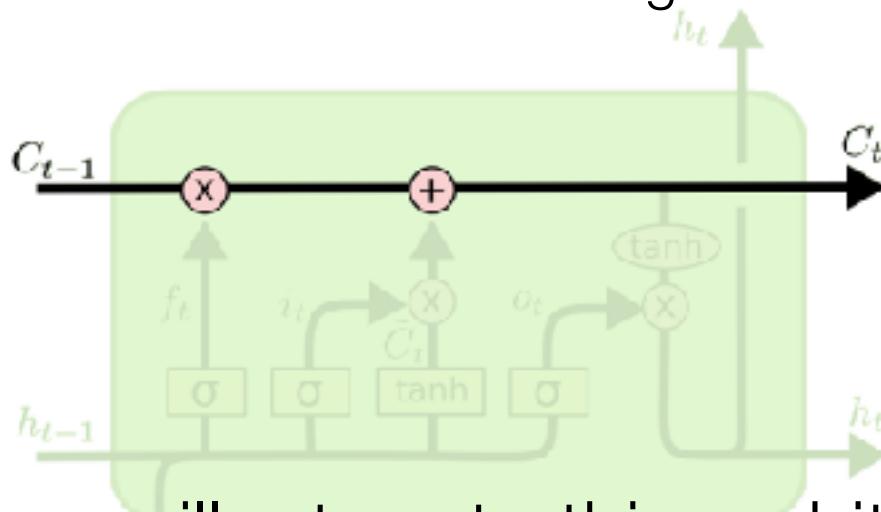
- **LSTM key idea:** limit how past data can affect output



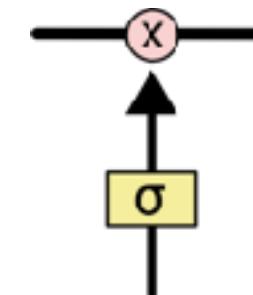
More Advanced Architectures

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

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 😂



Neural Network
Layer



Pointwise
Operation



Vector
Transfer



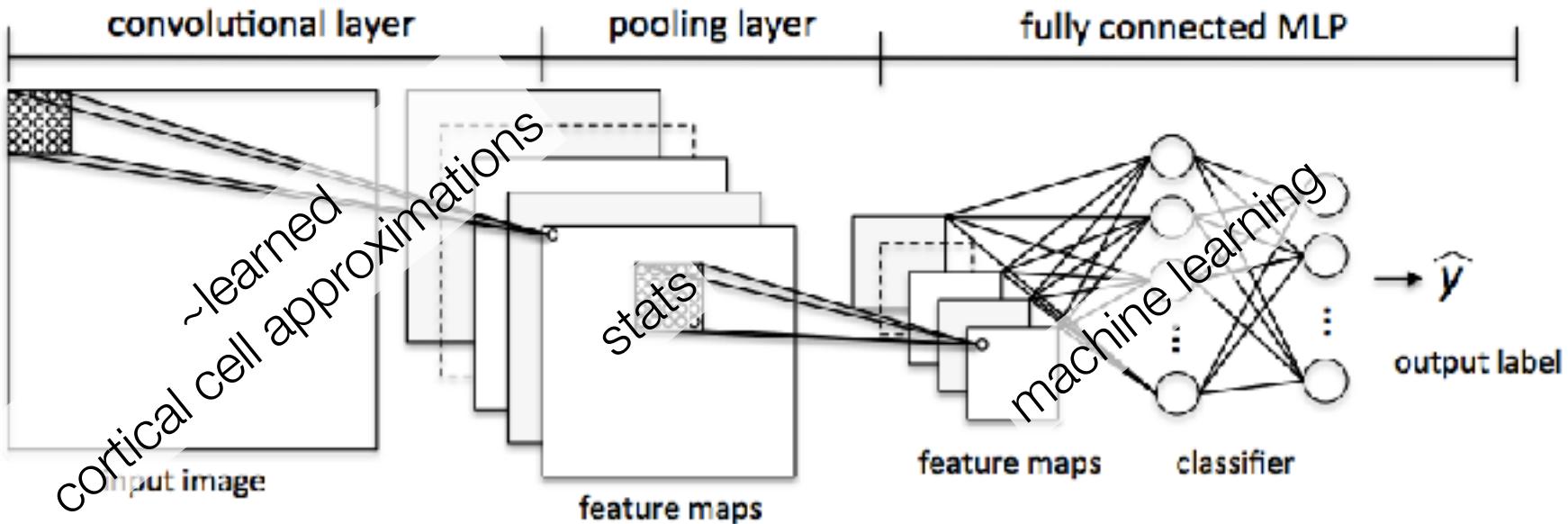
Concatenate



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**

End of Neural Networks Introduction

- Briefly step away from Neural Networks
- **Next time: In class assignment:**
 - evaluation and cross validation
- **Next Next time:** Ensembles

these will relate
back to neural networks

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

Michael Nielsen

<http://neuralnetworksanddeeplearning.com>

Watch videos before class!

- Next time is an in-class-assignment
 - cross validation!