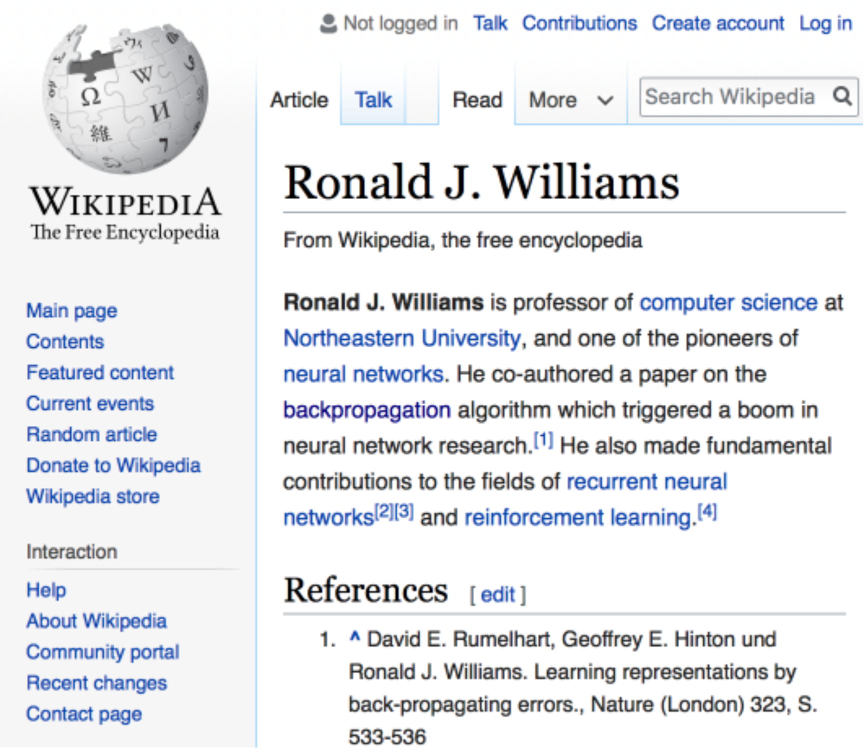
# Backpropagation

CS 510

Lecture #12

March 30th, 2020

# A Rebirth, one of several



Not logged in Talk Contributions Create account Log in

Article Talk Read More Search Wikipedia

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact page

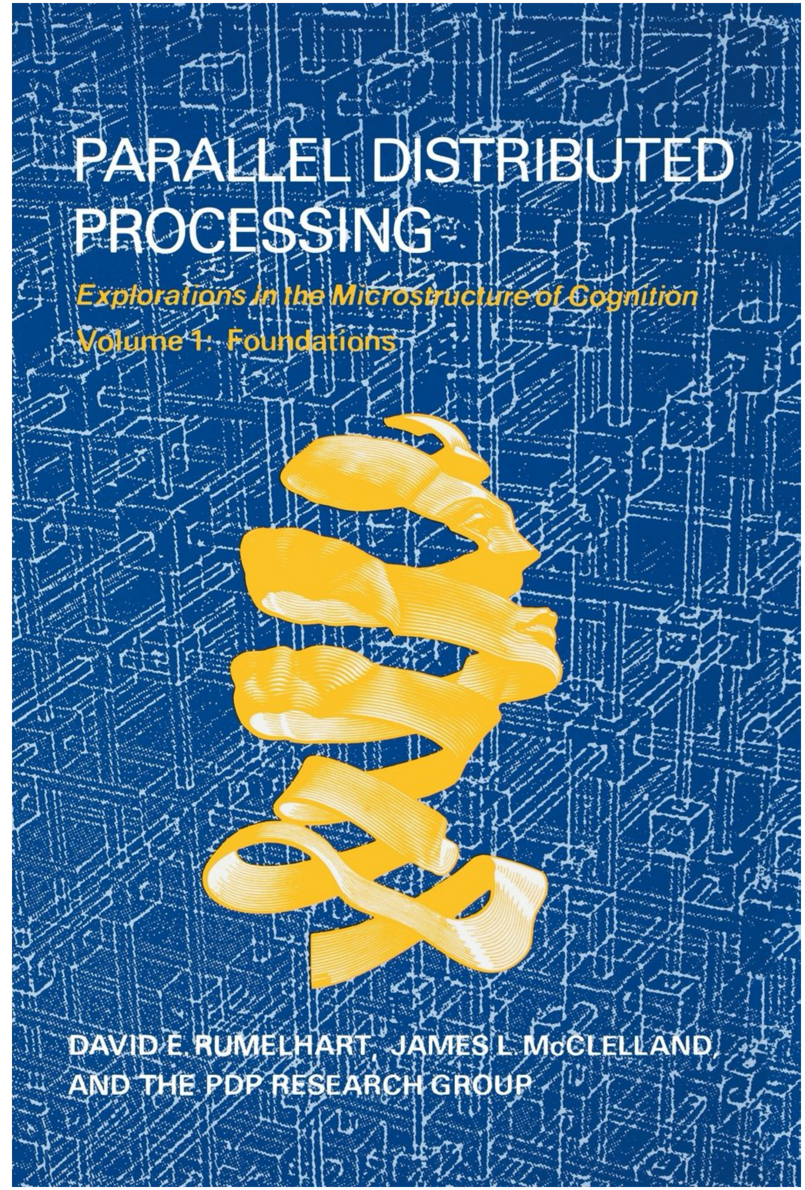
## Ronald J. Williams

From Wikipedia, the free encyclopedia

**Ronald J. Williams** is professor of computer science at Northeastern University, and one of the pioneers of neural networks. He co-authored a paper on the backpropagation algorithm which triggered a boom in neural network research.[1] He also made fundamental contributions to the fields of recurrent neural networks[2][3] and reinforcement learning.[4]

## References [ edit ]


1. ^ David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams. Learning representations by back-propagating errors., Nature (London) 323, S. 533-536

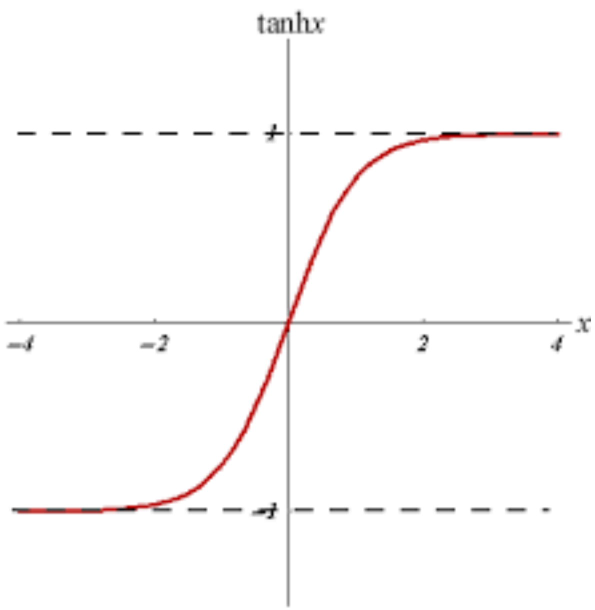



# Multi-layer Perceptrons II

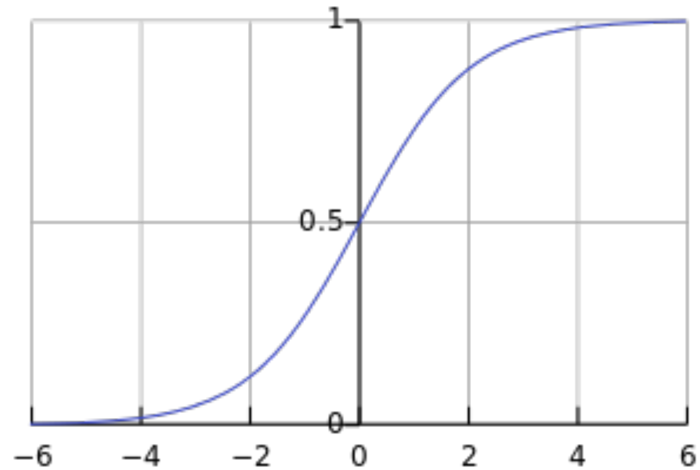
- Classic perceptrons threshold linear functions
  - $f(x) = h(w \cdot x + b)$
  - h() is a threshold-based *activation function*
  - Converts activations into decisions
- But if we want to combine perceptrons?
  - Thresholding individual perceptrons is not useful
  - Replacing h() with identity would allow us to sum linear responses
  - But a sum of linear responses is just another linear response

# Sigmoid Activation Functions

- $f(x) = s(w \cdot x + b)$




$y = tanh(x)$




$y = (1 + e^{-x})^{-1}$

# Activation Function Properties

- Activation functions <u>must</u>
  - Be non-linear
- Activation functions <u>may</u>
  - map an infinite domain to a finite range
    - Like [-1, 1] (for tanh) or [0, 1] (for logistic)
    - Keeps values from growing too large/small
    - Sometimes called "squashing"
  - Have non-zero derivatives everywhere
    - Useful for training

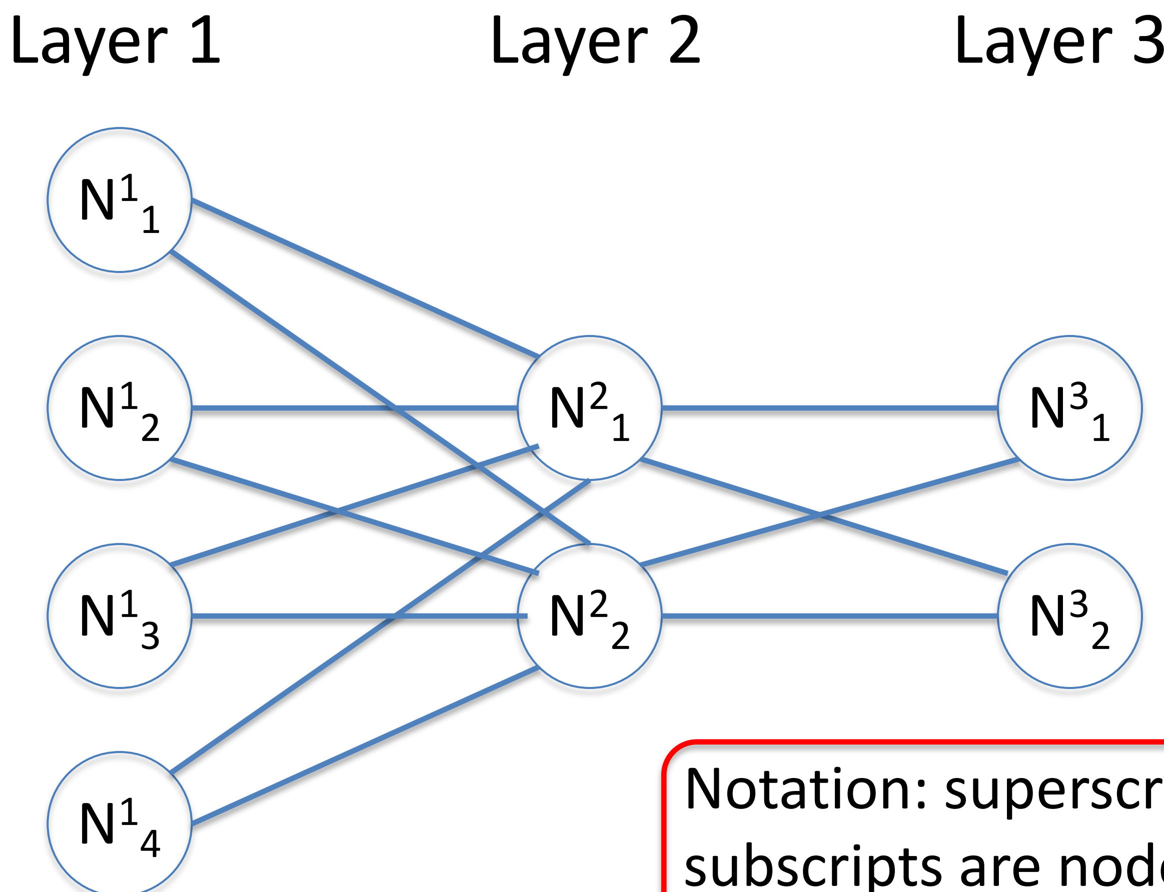
# Backpropagation

- Backpropagation is the algorithm that describes how we update weights in a network, given
  - Training samples
  - Training labels
  - A cost function
- Its used for (almost) all networks
- Network nodes may be
  - Non-linear perceptrons (the most common)
  - Convolutional units
  - Pooling units
  - Batch normalization units
  - …

# Goals For Today

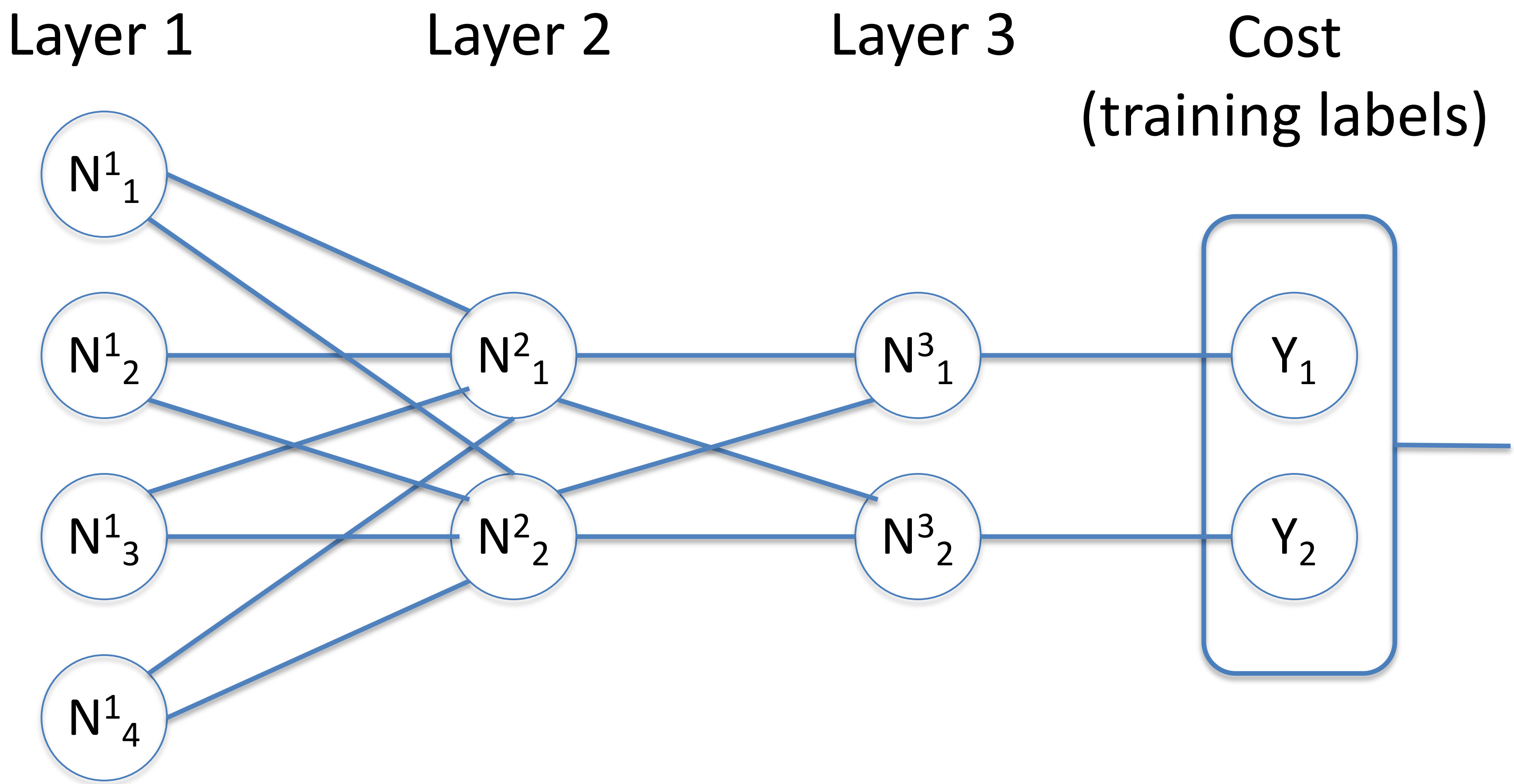
- Walk you through the math of backpropagation
  - Complicated, but just calculus
  - Almost universal : modifiable for different node types (see previous slide)
- Today's derivation assumes multi-layer perceptrons
  - $z(x) = wx + b$
  - $a(x) = h\big(z(x)\big) = h(wx + b)$
- Remember the chain rule from calculus:
  - $f(x) = g\big(h(x)\big) \rightarrow f'(x) = g'\big(h(x)\big)h'(x)$

# Simple Neural Network




Notation: superscripts are layers, subscripts are node numbers

# Setup for Training: Cost



# Cost Functions

- Cost functions measure the gap between the network output and the ideal output
- Two necessary properties

*Allows us to optimize per sample*

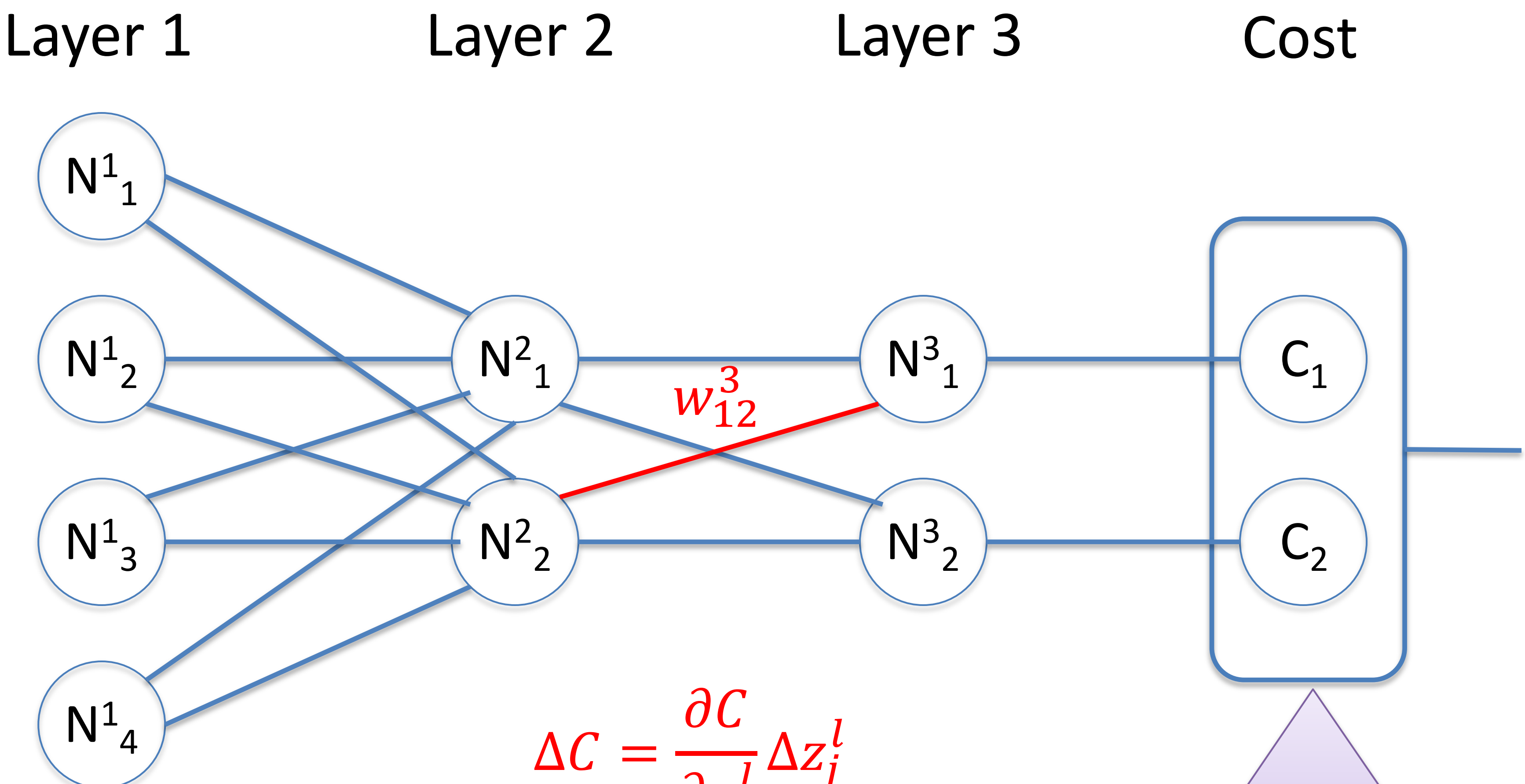
  1. An average over samples: $C = \frac{1}{n}\sum_x C_x$
  2. Function of output activations: $C = C(a_l)$

- Example: mean squared error

$$C = \frac{1}{2n}\sum_x \|y(x) - a^L(x)\|^2$$

*Allows us to initialize the partial derivative computations*

# δs : local derivatives as error measures




$$\Delta C = \frac{\partial C}{\partial z^l_j} \Delta z^l_j$$

Subtle change from previous diagram, now showing cost C not training label.

# Partial derivatives as error measures

- Imagine you want to change the output $z^l_{j,}$ by $\Delta z^l_j$
- Then $\Delta C = \frac{\partial C}{\partial z_j^l} \Delta z_j^l$
- If $\left|\frac{\partial C}{\partial z_j^l}\right|$ is large, then C becomes smaller by giving $\Delta z_j^l$ the opposite sign
- But if $\left|\frac{\partial C}{\partial z_j^l}\right|$ is near zero, then $\Delta z_j^l$ doesn't matter.
  - $\frac{\partial C}{\partial z_j^l}$ is already optimal!
  - $\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$

# Recap - where are we?

- We can optimize on a per-sample basis
  - Because the cost function is an average
- Minimizing the δs optimizes the net
  - The δs depend on the data samples
- But how do we minimize the δs?

- We will assume that nodes have non-linear functions, so $a_j^l = h(z_j^l)$

# Output Layer

- $\delta_j^L = \frac{\partial C}{\partial a_j^L} h'\left(z_j^L\right)$ by the chain rule
- $\frac{\partial C}{\partial a_j^L}$ is the partial derivative of C with respect the activation of output unit j
  - If C is LMS (slide #6)
    - $\frac{\partial C}{\partial a_j^L} = a_j^L(x) - y(x)$
    - The difference between the output & desired output

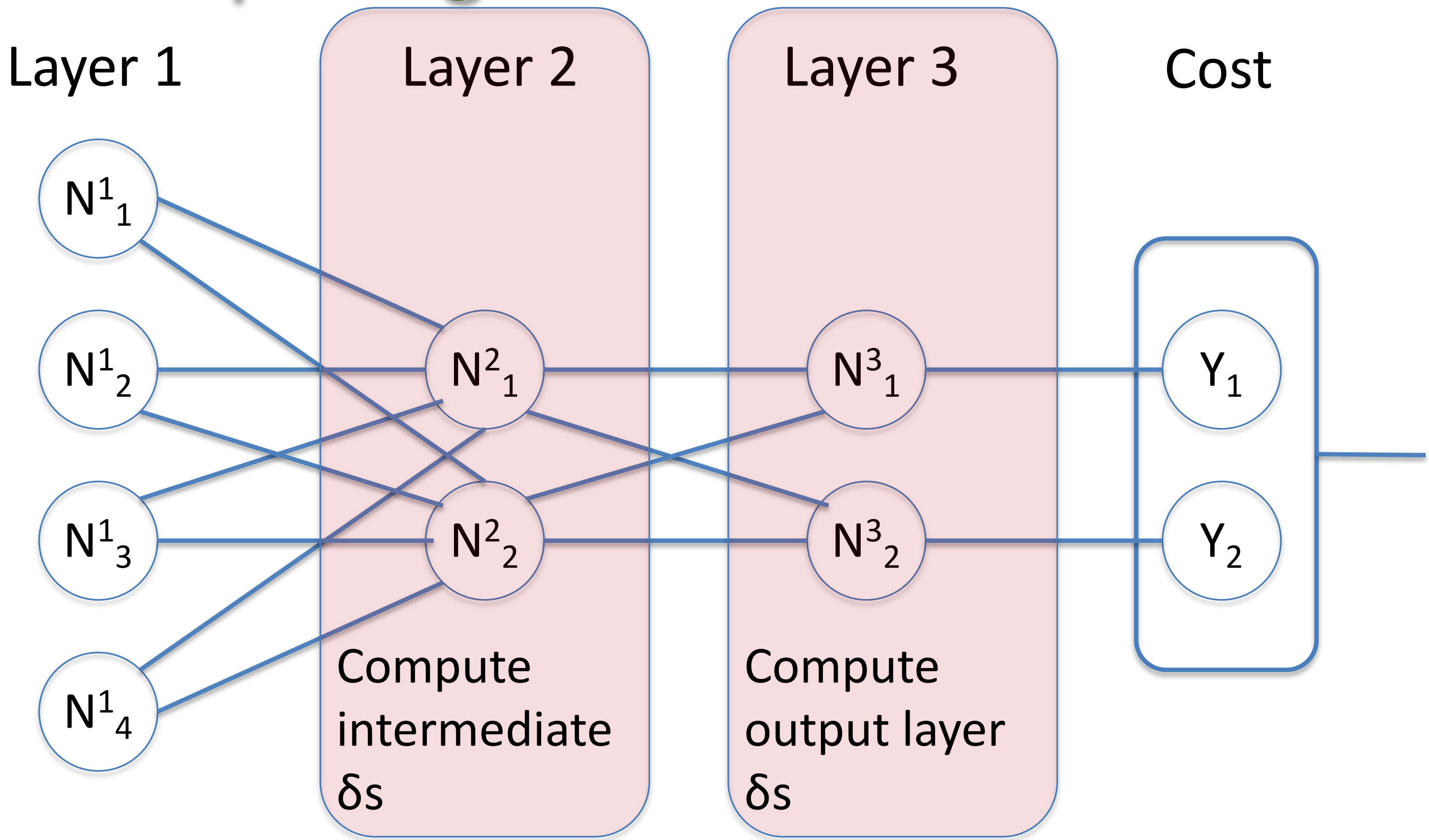
# Output Layer (cont.)

- $h'\left(z_j^L\right)$ is the derivative of the non-linear transfer function at $z_j^L$
- If $h(x) = tanh(x)$, $\sigma'(x) = 1 - tanh^2(x)$
- If $h(x) = (1 + e^{-x})^{-1}$,
$$\sigma'(x) = \sigma(x)\big(1 - \sigma(x)\big)$$
- $\delta_j^L = \left(a_j^L(x) - y(x)\right)\left(1 - tanh^2\left(z_j^L(x)\right)\right)$ <u>or</u>
- $\delta_j^L = \left(a_j^L(x) - y(x)\right)\left(a_j^L(x)\left(1 - a_j^L(x)\right)\right)$

# $\delta^{L}$ given $\delta^{L+1}$

- $\delta_j^l = \sigma'\left(z_j^l\right) \sum_k w_{kj}^{l+1} \delta_k^{l+1}$
- σ' is computed as on previous slide
- The RHS is just the sum of the impacts
- This is where *backpropagation* comes from
  - Calculate δs for output layer
  - Then recursively compute δs for previous layers

# Computing δs…



# So…

- Given an input x and output y:
  - We can compute $\delta^l_j$ for every node j at every level l
  - Minimizing the δs will optimize the network
    - Relative to this sample
  - So we need to adjust the weights $w_i$ and b to reduce the δs
    - But just a little for each input/output pair
    - So we can optimize across all samples

# Adjusting b

- Remember that $\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$ (slide #8)
- And that $z_j^l = w_j^l x + b$
- So $\frac{\partial C}{\partial b_j^l} = \delta_j^l$
- So $b_j^l \leftarrow (1 - \alpha) b_j^l - \alpha \delta_j^l$
  - Where α is a learning rate
  - Regulates how much you react to each sample

# Adjusting w's

- $\frac{\partial C}{\partial w_{jk}^{l}} = a_k^{l-1}\delta_j^l$
- So $w_{jk}^{l} \leftarrow (1-\alpha)w_{jk}^{l} - \alpha a_k^{l-1}\delta_j^l$
  - Where α is the same learning rate as before
  - We are collectively minimizing the deltas by heading downhill in the k+1 dimensional space defined by w & b

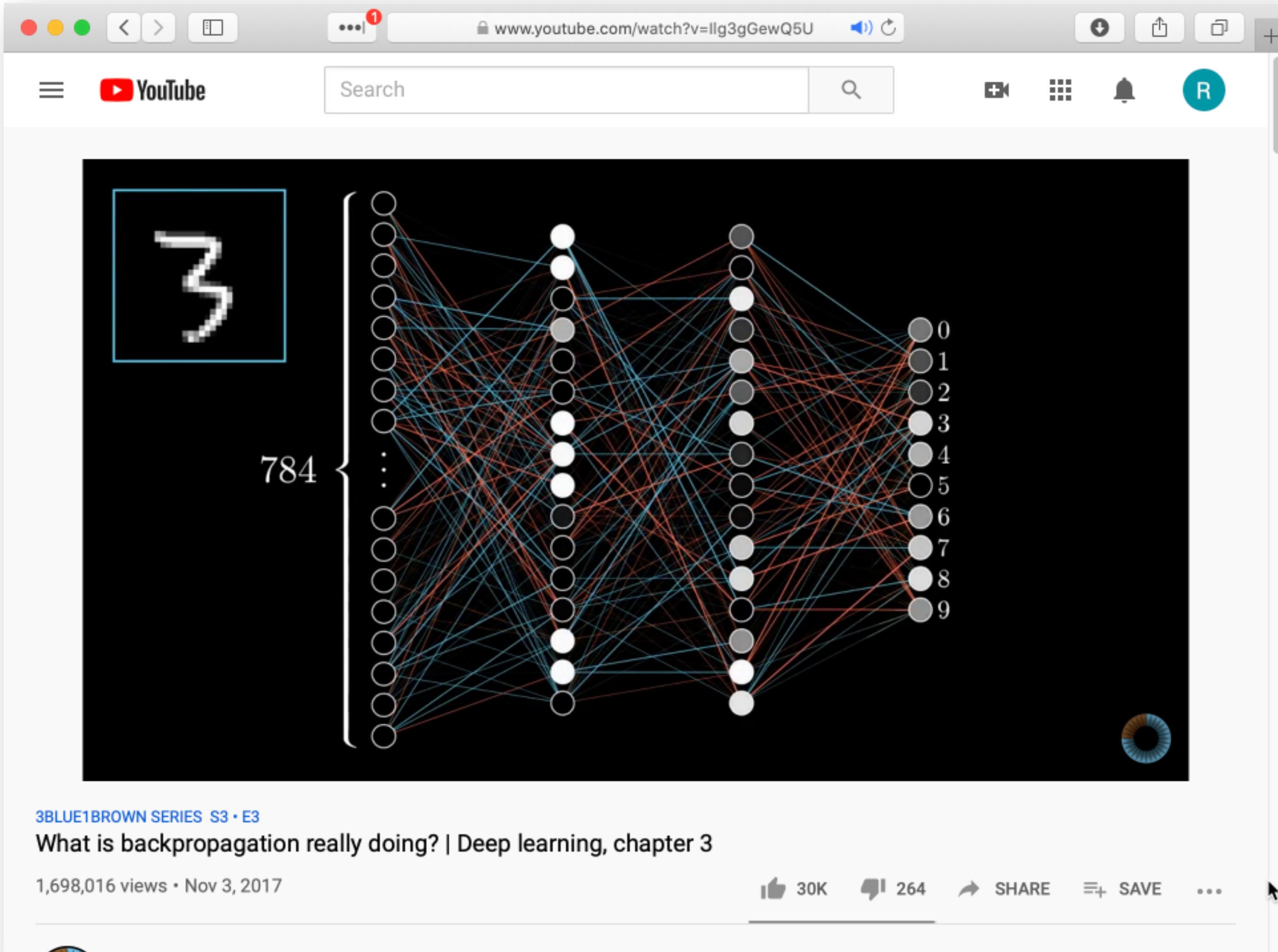
# Backpropagation (redux)

- Backpropagation updates weights in a network, given
  - Training samples
  - Training labels
  - A cost function
- Network nodes may be
  - Non-linear perceptrons (the most common)
  - Convolutional units
  - Pooling units
  - Batch normalization units
  - …

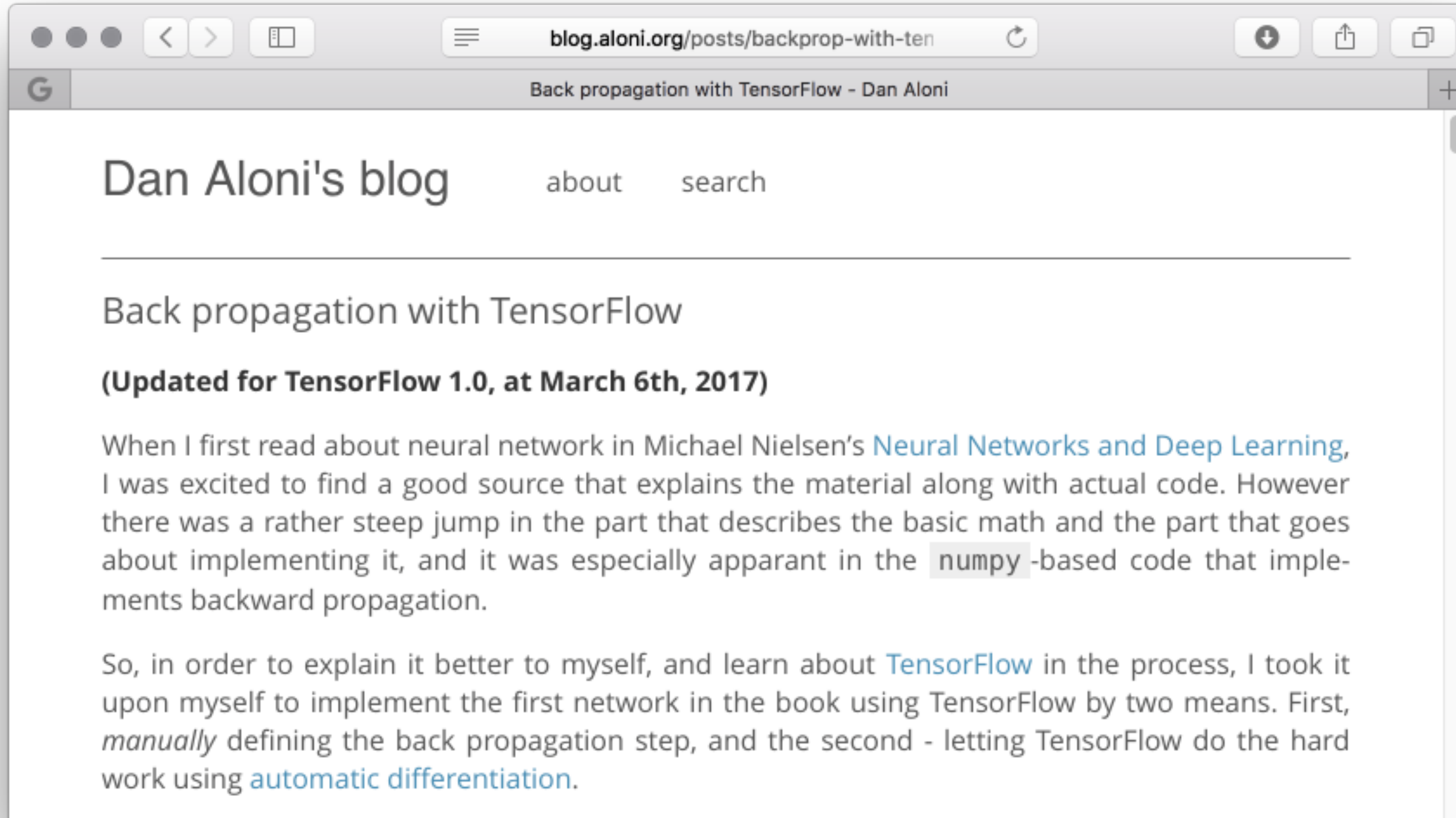
# Step Back!  Other Resources

- Modulo some notation ambiguity the previous formula-based presentation if fine, but for some of us unsatisfying
- It is best to approach the task of understanding backpropogation simultaneously from three angles.
    1. Mathematical formulas (just finished)
    2. Develop an internal visualization
    3. Running code

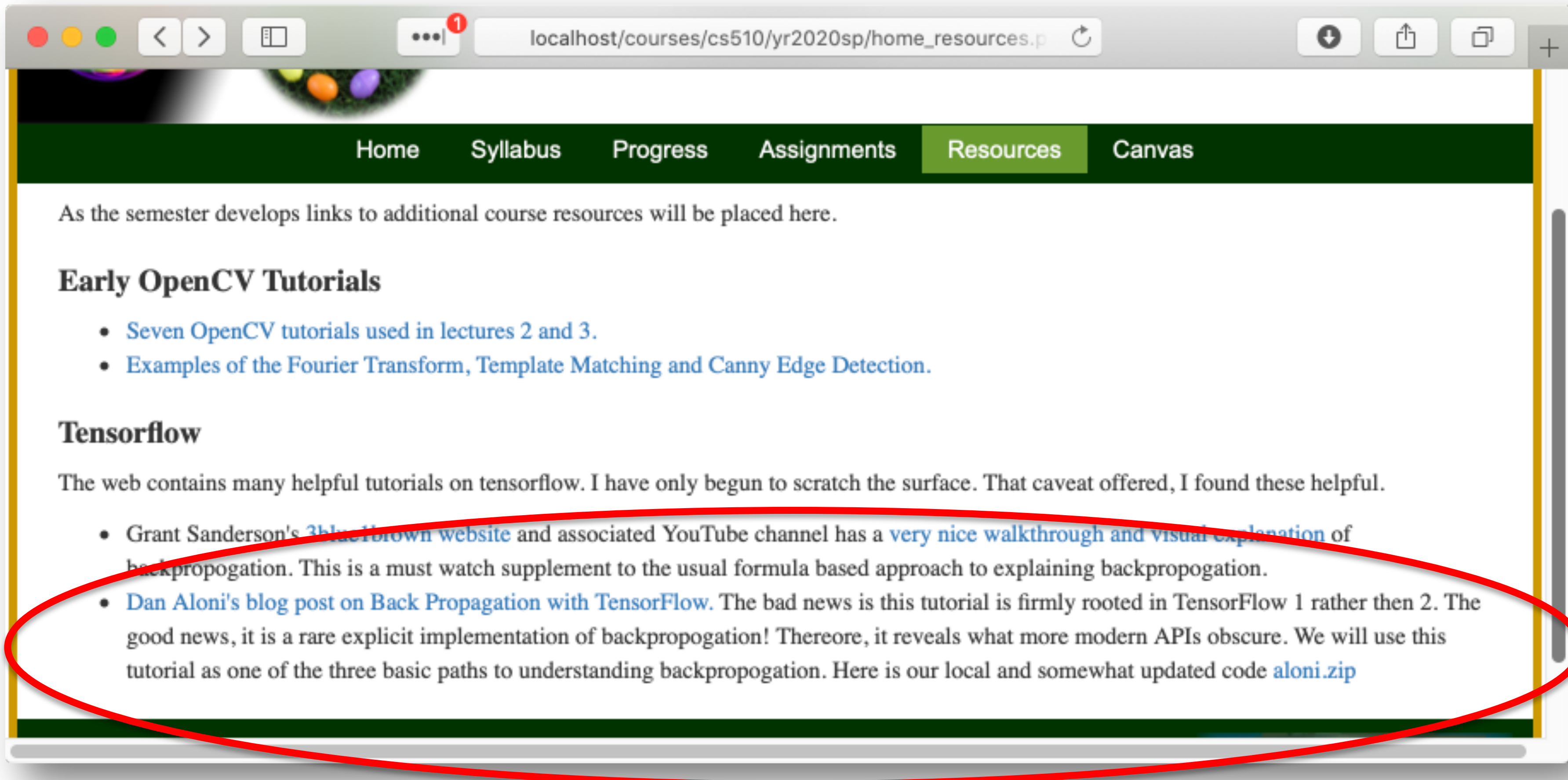
# May I Recommend



# And Also


blog.aloni.org/posts/backprop-with-ten
Back propagation with TensorFlow - Dan Aloni
Dan Aloni's blog
about
search
Back propagation with TensorFlow
(Updated for TensorFlow 1.0, at March 6th, 2017)
When I first read about neural network in Michael Nielsen's Neural Networks and Deep Learning, I was excited to find a good source that explains the material along with actual code. However there was a rather steep jump in the part that describes the basic math and the part that goes about implementing it, and it was especially apparant in the numpy -based code that implements backward propagation.
So, in order to explain it better to myself, and learn about TensorFlow in the process, I took it upon myself to implement the first network in the book using TensorFlow by two means. First, manually defining the back propagation step, and the second - letting TensorFlow do the hard work using automatic differentiation.

# Please Run and Play With …




As we will discuss in lecture today, I expect everyone to setup a TF 1.14 environment and play with this code.