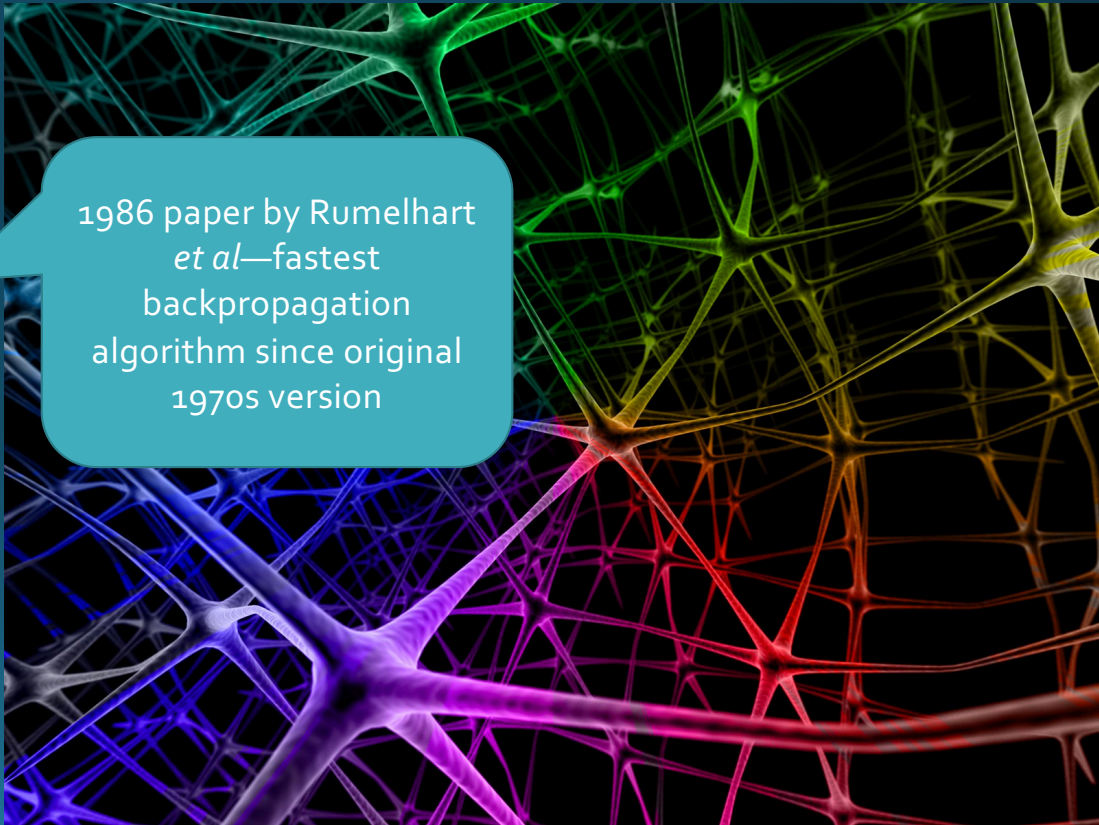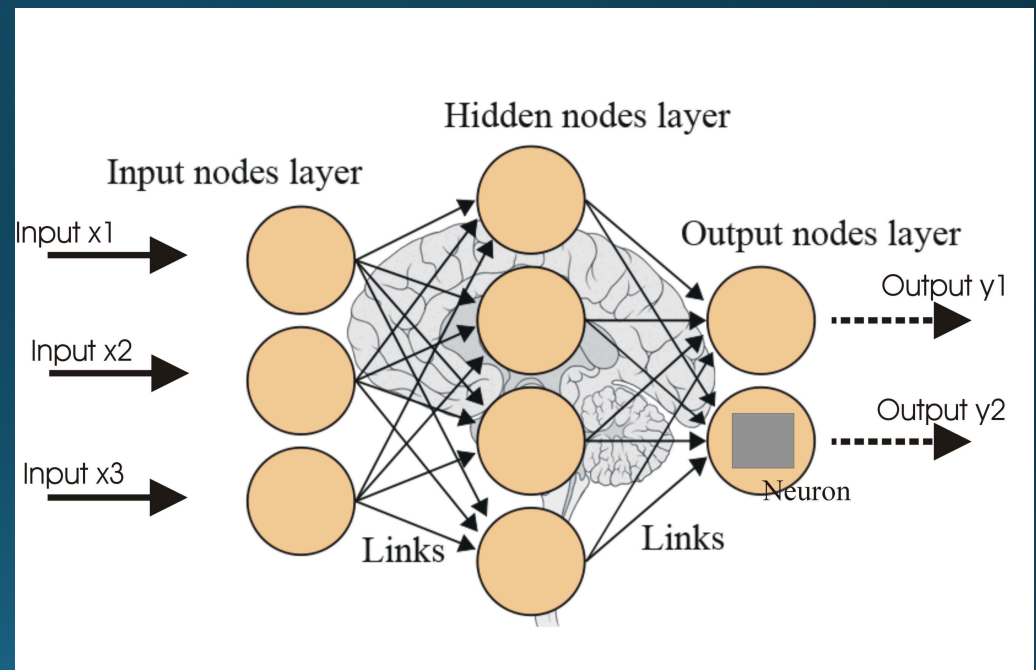# Backpropagation

Artificial Neural Networks

- **Not a new concept!**
  - Roots as far back as 1940s work in unsuperivsed learning
  - Took off in 1980s and 1990s
  - Waned in 2000s
- "Biologically-inspired" computing
  - May or may not be true
- **Shift from rule-based to emergent learning**

1986 paper by Rumelhart *et al*—fastest backpropagation algorithm since original 1970s version
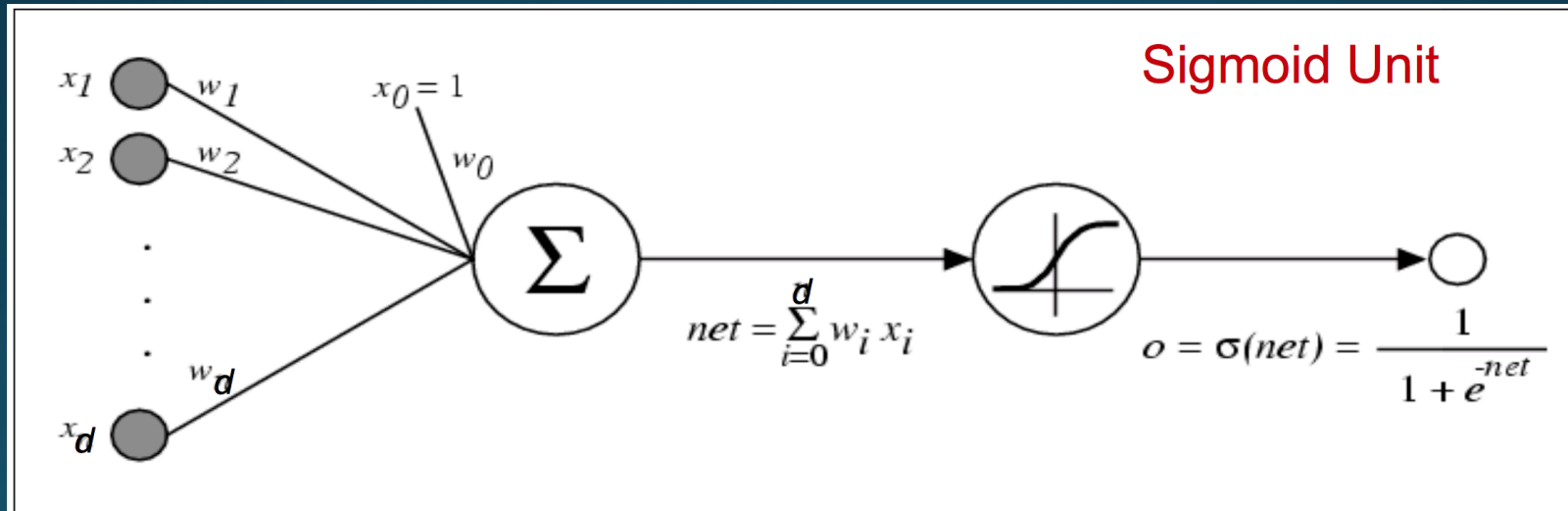
Multilayer networks

- Simplest case: classifier is a multilayer *network* of *logistic units*

- Each *unit* takes some inputs and produces one output using a logistic classifier

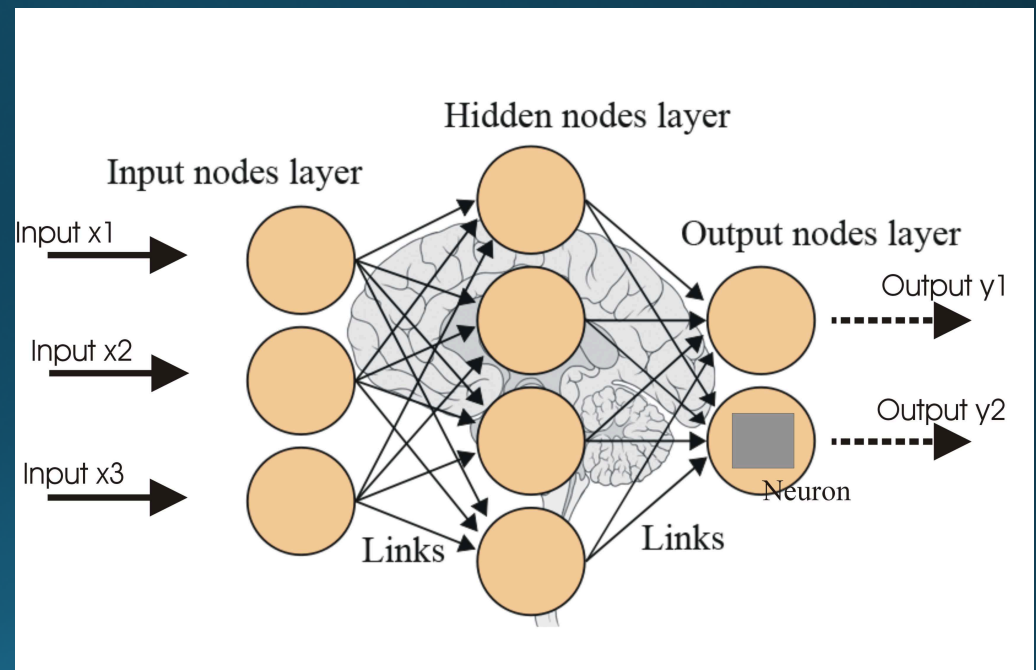- Output of one unit can be the input of other units

LR as a Graph

- Define output *o(x)* =

$$\sigma(w_0 + \sum_i w_i X_i) = \frac{1}{1 + \exp(-(w_0 + \sum_i w_i X_i))}$$

Sigmoid Unit

x_1 w_1 $x_0 = 1$

x_2 w_2 w_0

Σ

w_d

x_d

$net = \sum_{i=0}^{d} w_i x_i$

$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

Multilayer networks

- Simplest case: classifier is a multilayer *network* of ~~logistic~~ *units **that perform some differentiable computation***

- Each *unit* takes some inputs and produces one output ~~using a logistic classifier~~

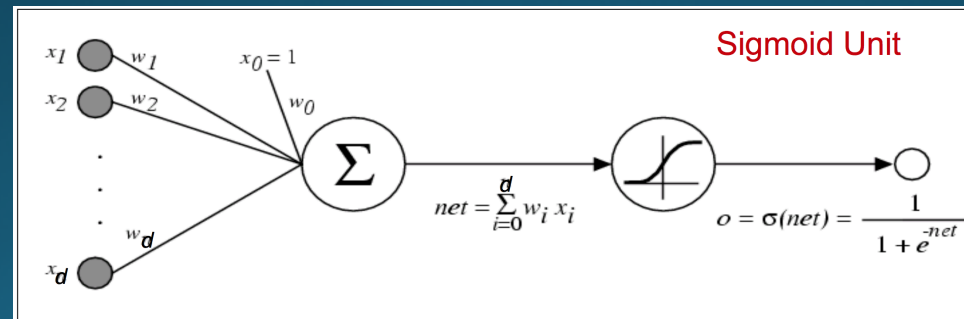- Output of one unit can be the input of other units

Learning a multilayer network

- Define a loss (simplest case: squared error)
  - But over a network of "units" that do simple computations
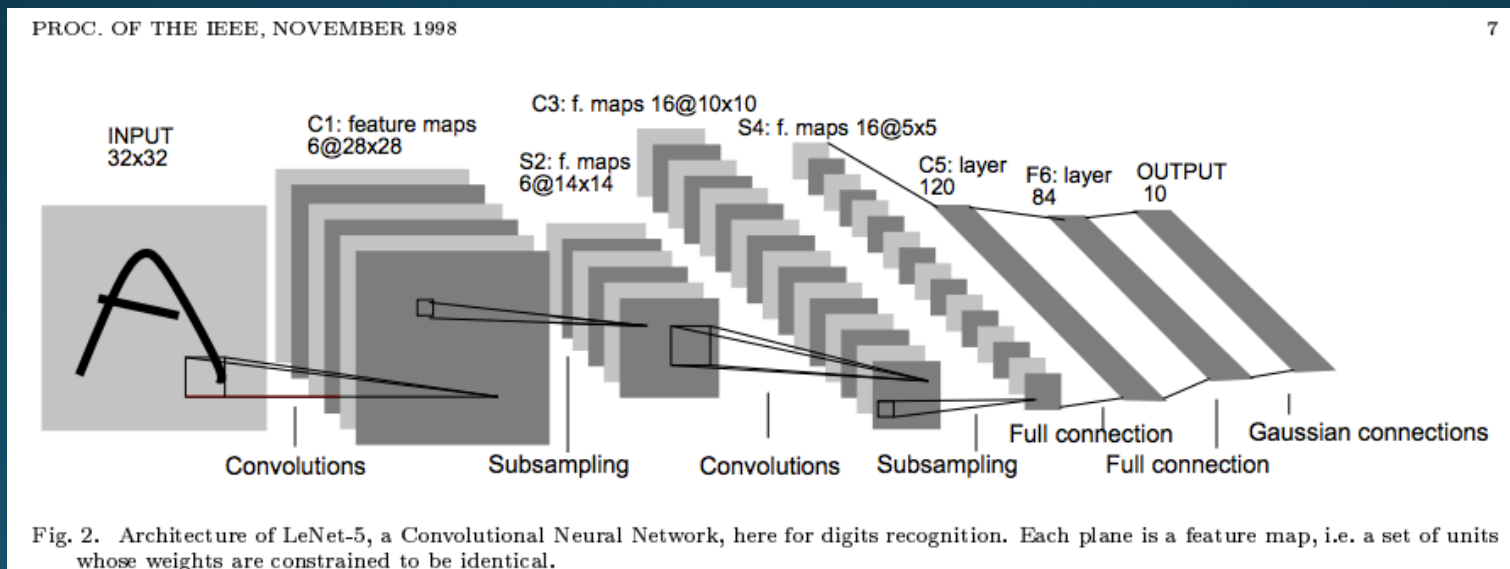
$$J_{X,y}(\vec{w}) = \sum_i (y^i - \hat{y}^i)^2$$

- Minimize loss with gradient descent
  - You can do this over complex networks if you can take the *gradient* of each unit: every computation is *differentiable*
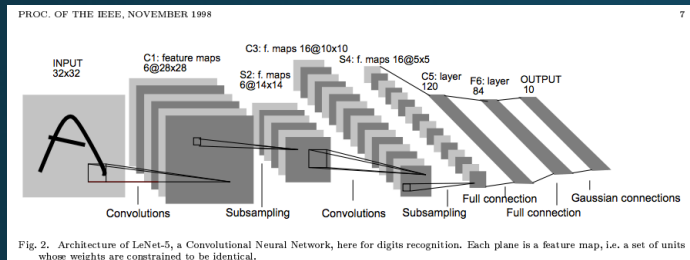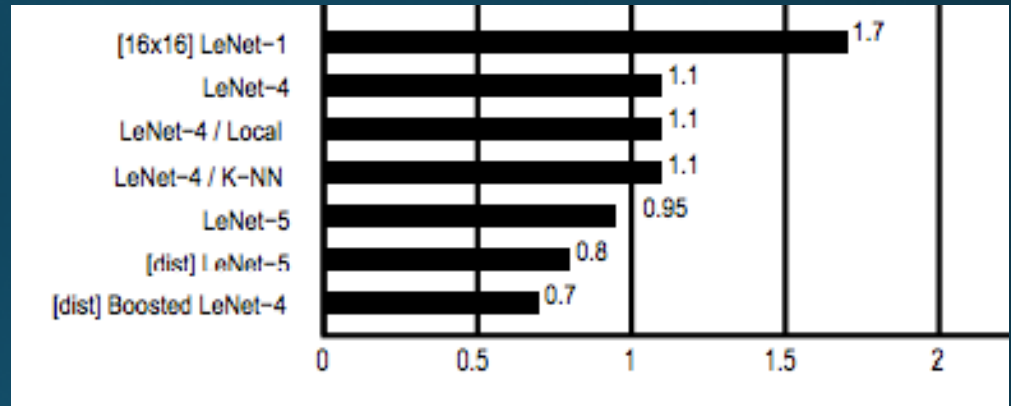

Sigmoid Unit

$net = \sum_{i=0}^{d} w_i x_i$

$o = \sigma(net) = \dfrac{1}{1 + e^{-net}}$

ANNs in the 90s

- In the 90s: mostly 2-layer networks (or specialized "deep" networks that were hand-built)
- Worked well, but training was *slow*

Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

ANNs in the 90's

Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

SVM with polynomial kernel: 98.9 - 99.2% accurate
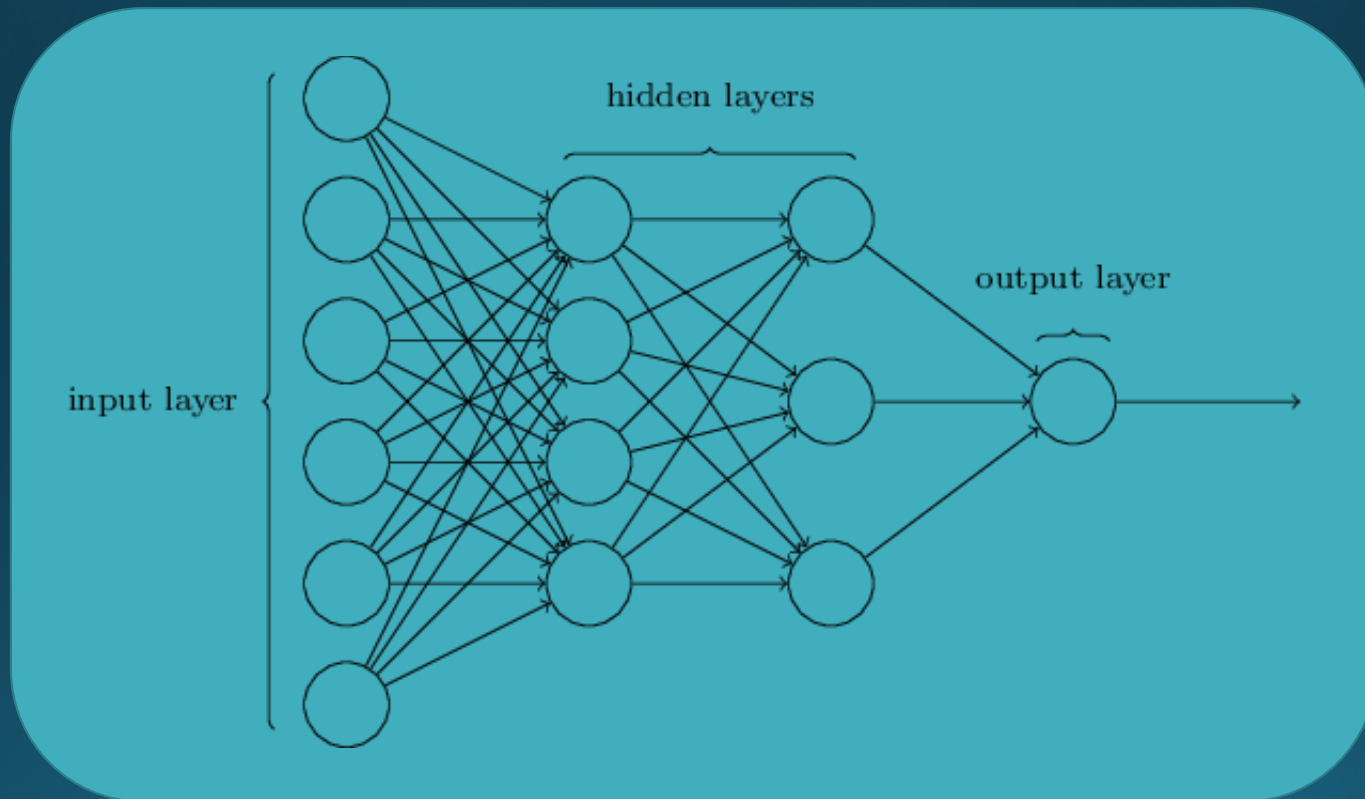

Custom CNN: 98.3 - 99.3% accurate

Nomenclature

- *Backpropagation:* refers **only** to the method for computing the gradient of a function
  - Is NOT specific to multilayer neural networks (in principle, can compute gradients for any function)

- *Stochastic gradient descent:* conducts learning using the derived gradient
  - Hence, you can run SGD on gradients you derive manually, or through backprop

Notation

- "Borrowing" from
    - William Cohen at Carnegie Mellon (author of SSL algorithm you implemented in HW4)
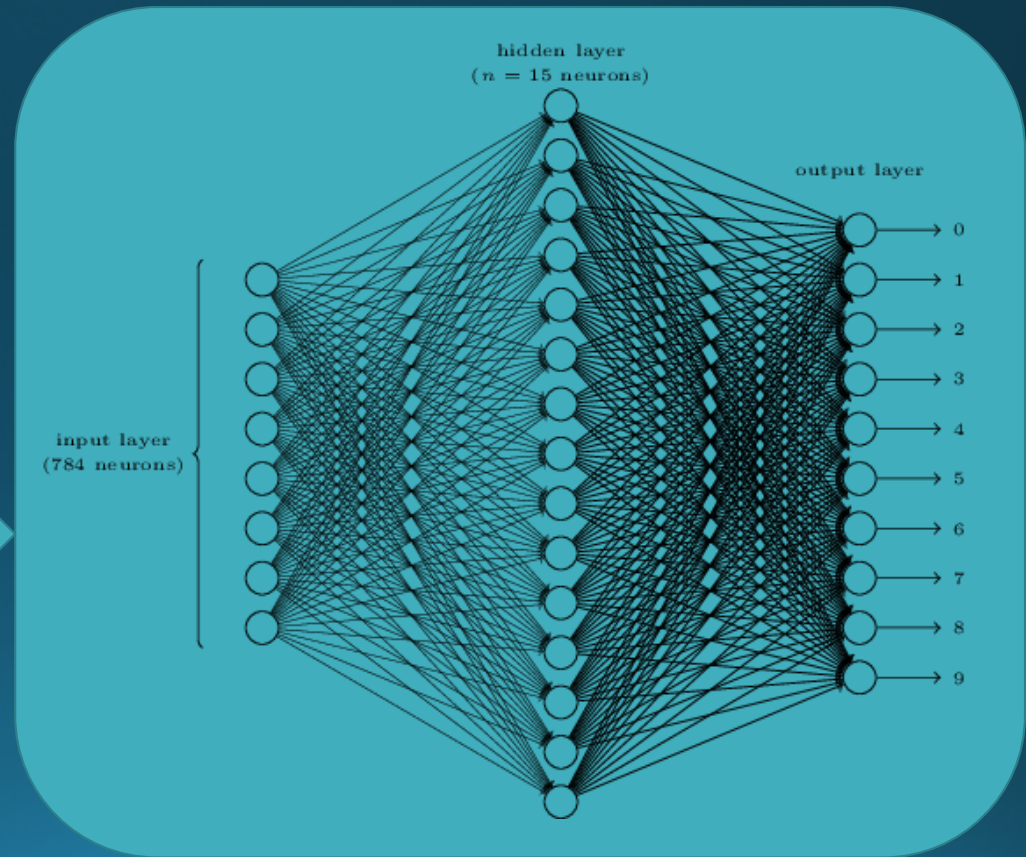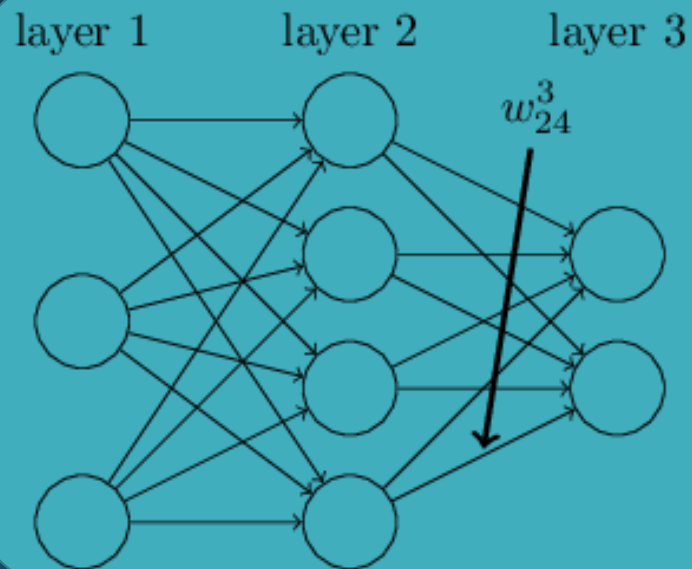   - Michael Nielson of http://neuralnetworksanddeeplearning.com/

Notation

Notation

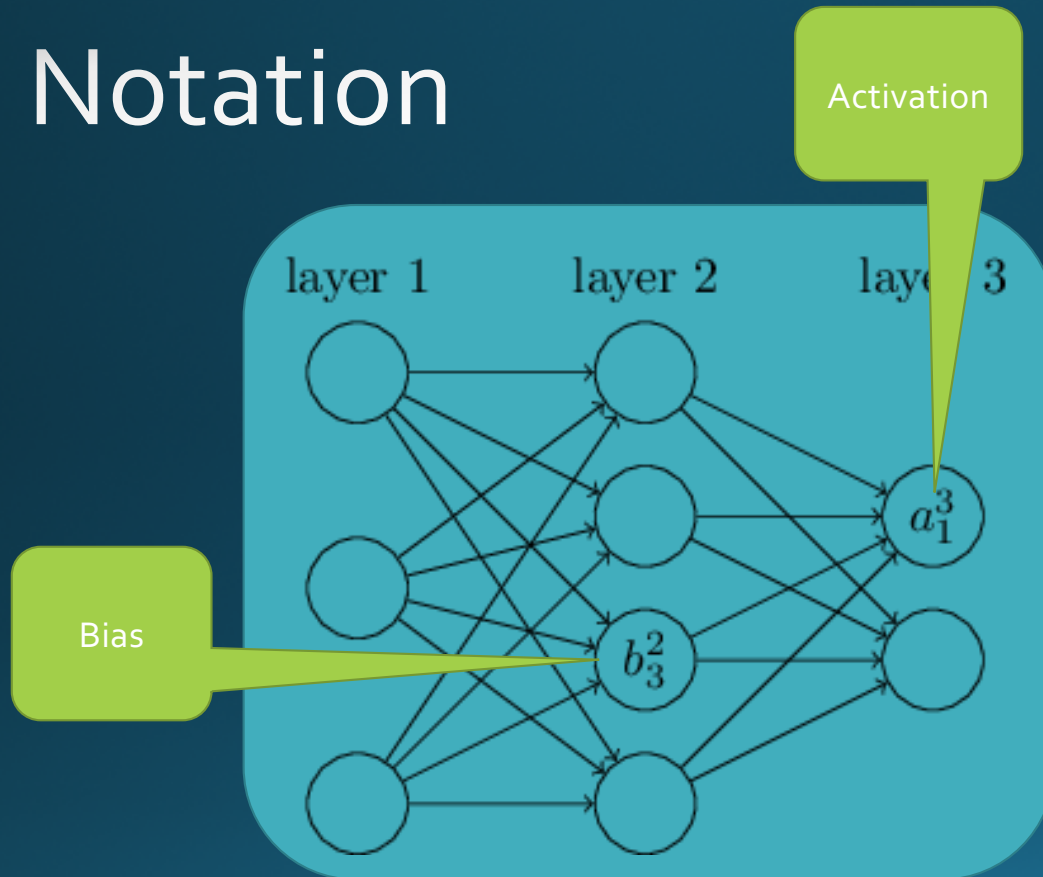- Each digit is 28x28 = 784 dimensions / inputs

Notation

Vectorize: w^l is the weight matrix for layer l

w_{jk}^l is the weight from the kth neuron in the $(l-1)^{\text{th}}$ layer to the jth neuron in the lth layer
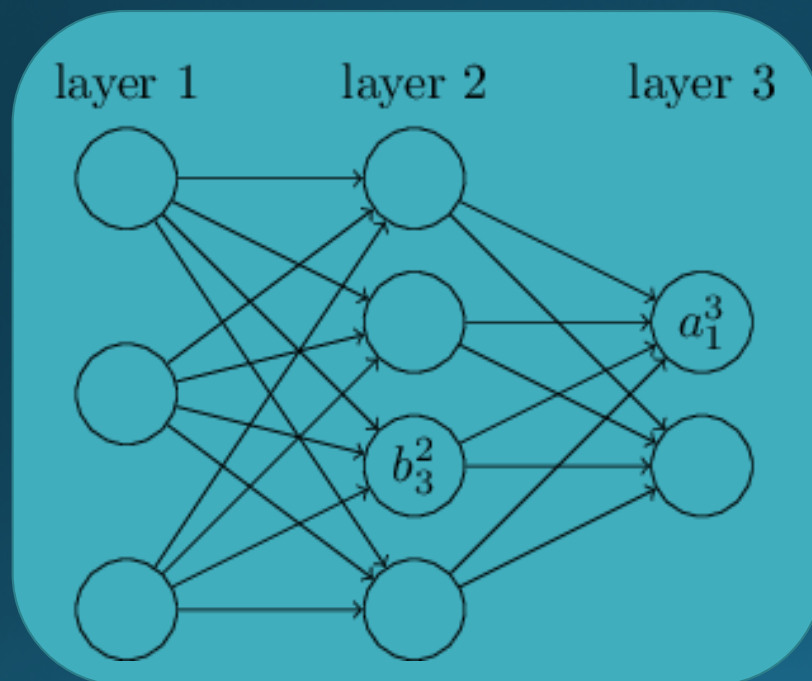
Notation

Activation

Bias

Vectorize: a^l and b^l are activations and bias matrices for layer l

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

Notation
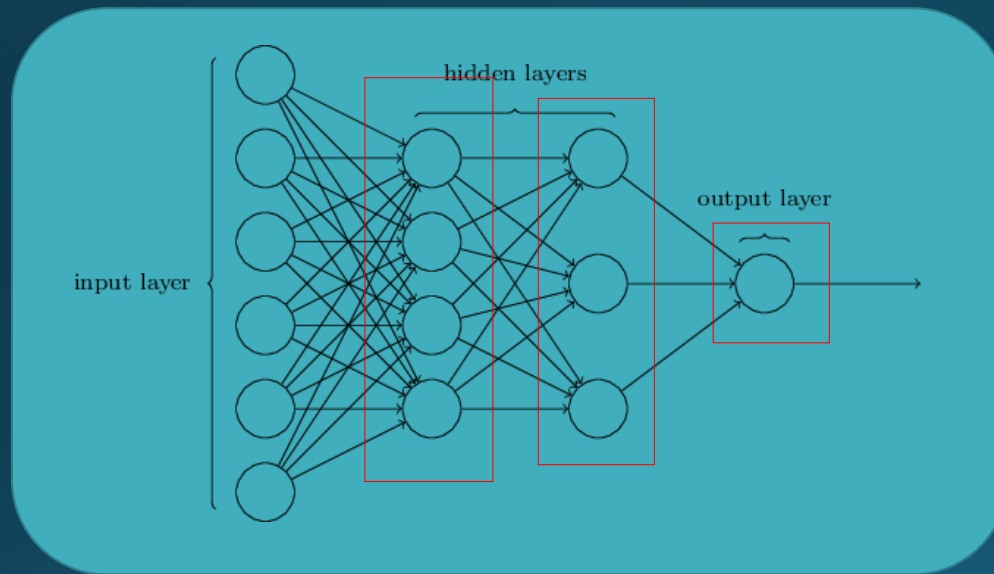

layer 1   layer 2   layer 3

a_1^3

b_3^2

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

$$a^l = \sigma(w^l a^{l-1} + b^l).$$

$$z^l \equiv w^l a^{l-1} + b^l$$

Computation is "feedforward"

for *l=1, 2, ... L:*
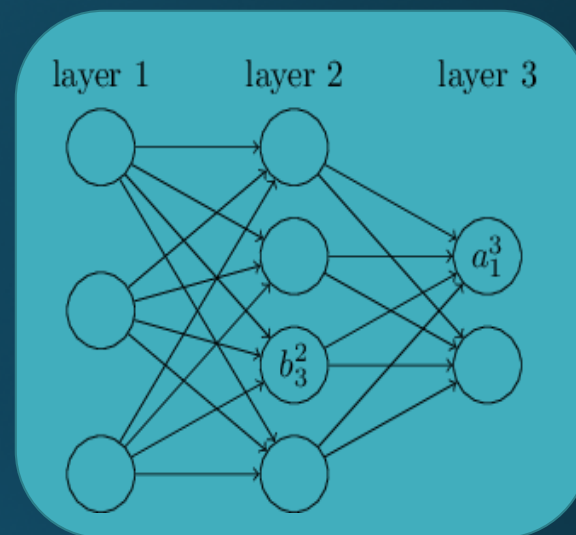
$$a^l = \sigma(w^l a^{l-1} + b^l).$$

Notation

- Set up a cost function, *C*

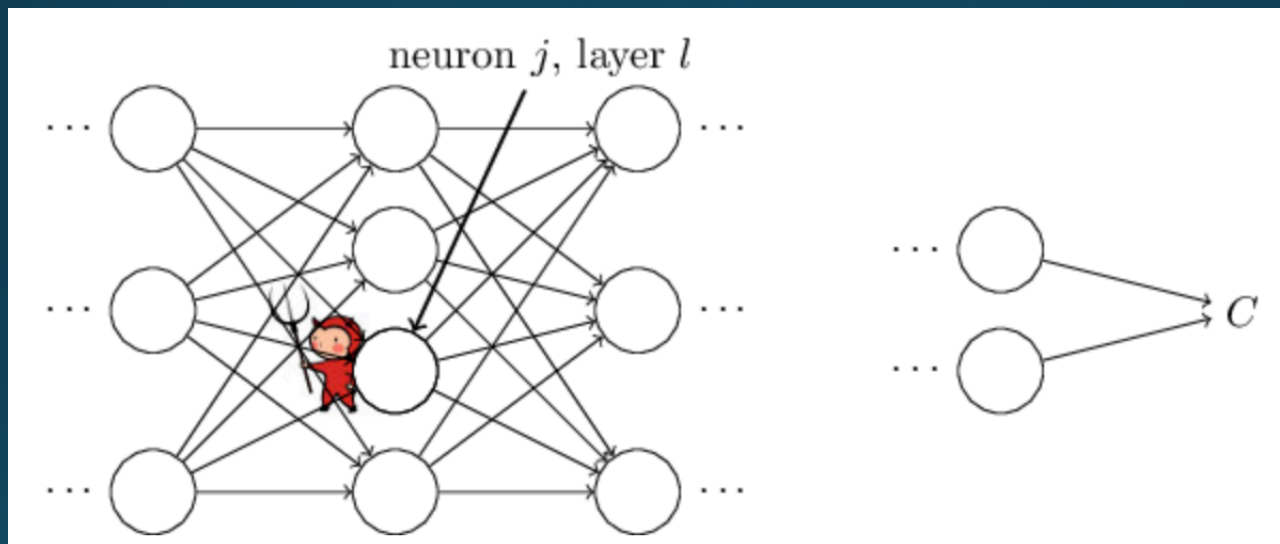$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$

- Rewrite as an average

$$C = \frac{1}{n} \sum_x C_x \qquad \text{where} \qquad C_x = \frac{1}{2}\|y - a^L\|^2$$

Allows us to compute partial derivatives dC_x/dw and dC_x/db for single training examples, then recover dC/dw and dC/db by averaging over training examples.

Notation

neuron j, layer l

Error in j^{th} neuron at the lth layer

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}.$$

BackProp: last layer

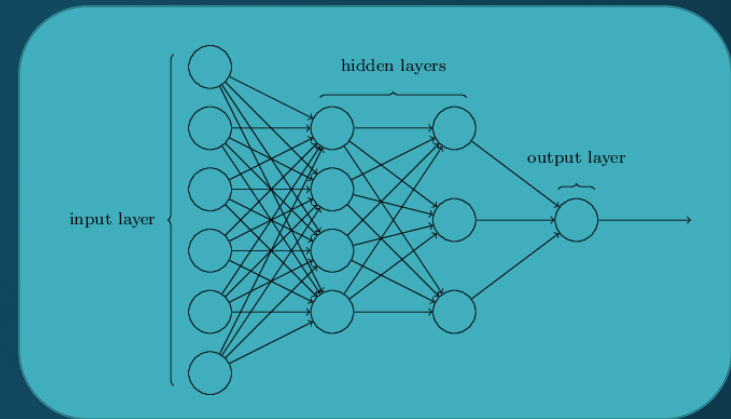$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L).$$

Matrix form:

$$\delta^L = \nabla_a C \odot \sigma'(z^L).$$

components are $\frac{\partial C}{\partial a_j^L}$

components are $\sigma'(z_j^L)$

The Hadamard Product: just element-wise multiplication
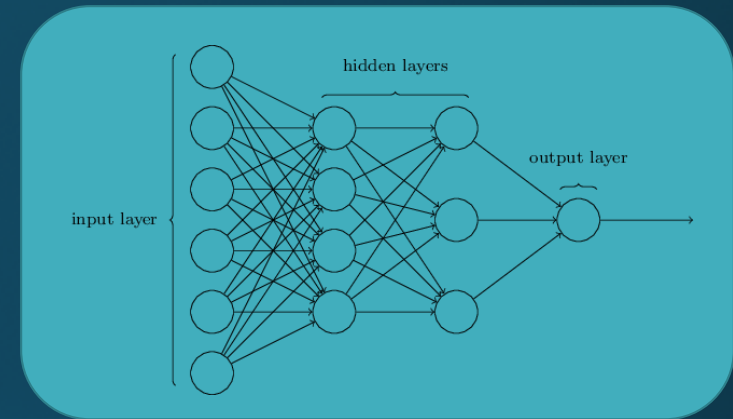

Level *l* for *l=1,...,L*
Matrix: w^l
Vectors:
- bias b^l
- activation a^l
- pre-sigmoid activ: z^l
- target output y
- "local error" δ^l

BackProp: last layer

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L).$$

Matrix form for square loss:

$$\delta^L = (a^L - y) \odot \sigma'(z^L).$$

Level l for $l=1,...,L$
Matrix: w^l
Vectors:
- bias b^l
- activation a^l
- pre-sigmoid activ: z^l
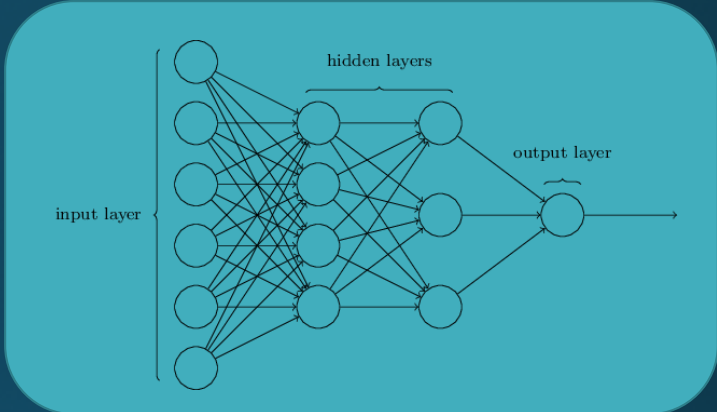- target output y
- "local error" δ^l

BackProp: error at level *l* in terms of error at level *l+1*

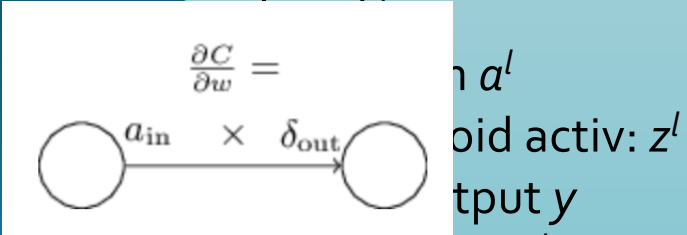$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

which we can use to compute

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \qquad \Longrightarrow \qquad \frac{\partial C}{\partial b} = \delta,$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l. \qquad \Longrightarrow \qquad \frac{\partial C}{\partial w} = a_{in} \delta_{out}$$

Level *l* for *l=1,…,L*
Matrix: w^l
Vectors:

$$\frac{\partial C}{\partial w} =$$

$a_{in} \quad \times \quad \delta_{out}$

n a^l
oid activ: z^l
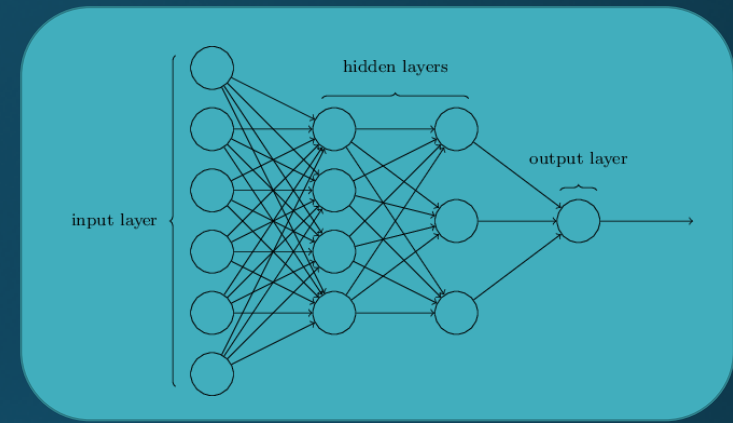tput *y*
- "local error" δ^l

21

BackProp: Summary

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$$

Level l for $l=1,\ldots,L$
Matrix: w^l
Vectors:
- bias b^l
- activation a^l
- pre-sigmoid activ: z^l
- target output y
- "local error" δ^l

22

Full Backpropagation

1. **Input** x**:** Set the corresponding activation a^1 for the input layer.

2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.

3. **Output error** δ^L**:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

4. **Backpropagate the error:** For each $l = L - 1, L - 2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.

5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$ and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$.

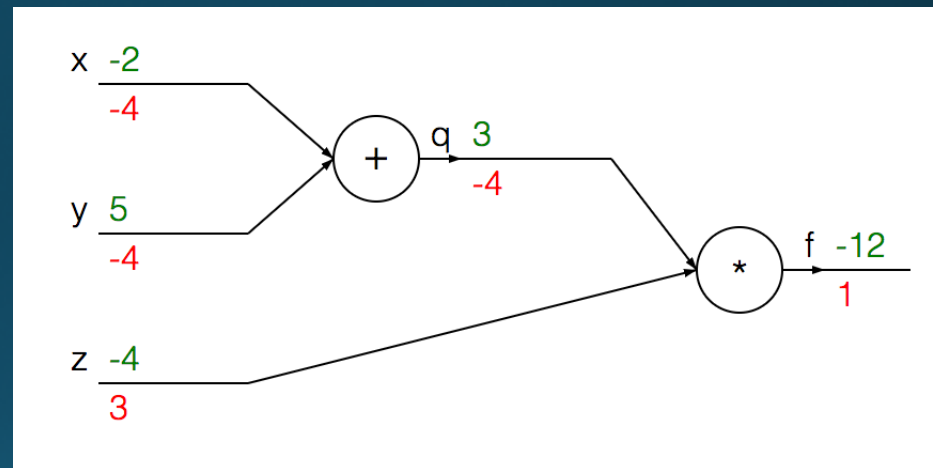Use SGD to update the weights according to the gradients

Example

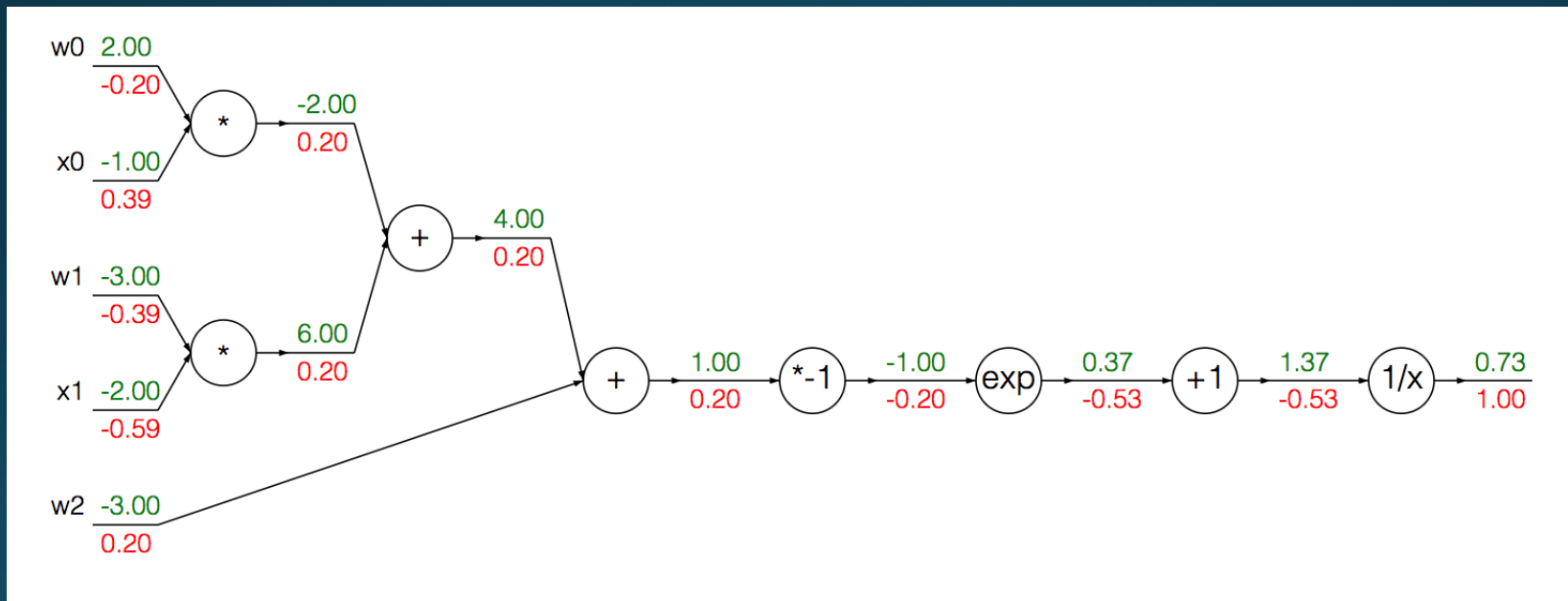- Simple equation
$$f(x, y, z) = (x + y)z$$
- Some example inputs
  - x = -2
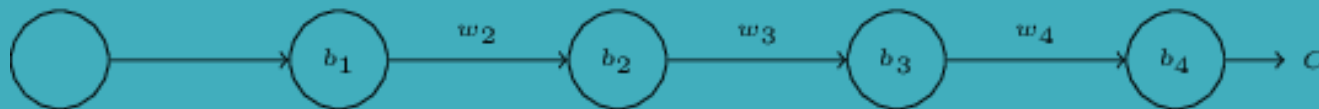 - y = 5
  - z = -4

[slightly less simple] Example

- 2D Logistic Regression, with a bias term $P(Y = 1|X) = \dfrac{1}{1 + \exp(-(w_0 + \sum_i w_i X_i))}$

Weight updates for multilayer ANN

- For nodes *k* in output layer *L:* $\quad \delta_k^L = (t_k - a_k)a_k(1 - a_k)$

- For nodes *j* in hidden layer *h:* $\quad \delta_j^h = \sum_k (\delta_j^{h+1} w_{kj}) a_j (1 - a_j)$

- What happens as the layers get further and further from the output layer?

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$

Gradients are unstable

Max at 1/4

- If weights are usually < 1, and we are multiplying by many, *many* such numbers...

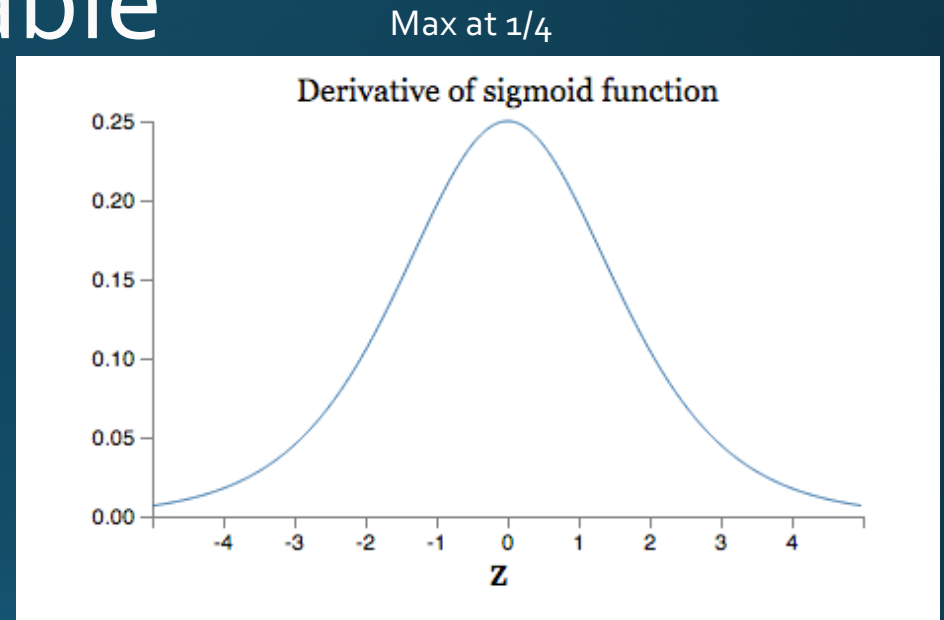The Amazing Vanishing Gradient!

Derivative of sigmoid function

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$
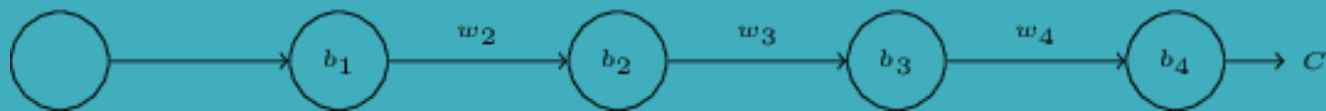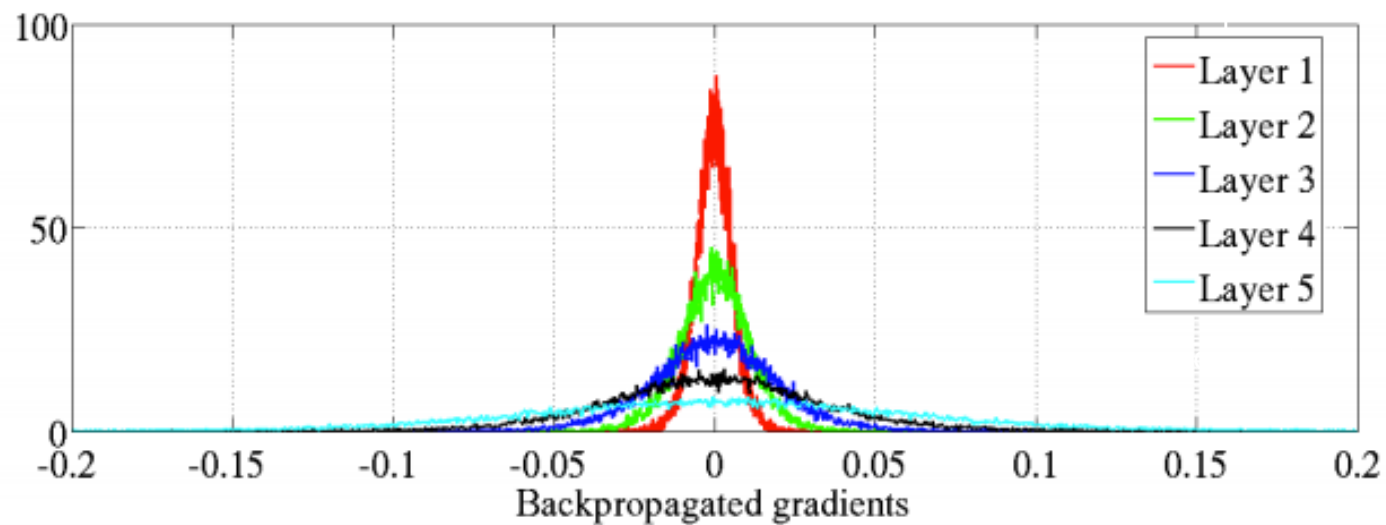
b_1 — w_2 — b_2 — w_3 — b_3 — w_4 — b_4 — C

Histogram of gradients in a 5-layer network for an artificial image recognition task

Understanding the difficulty of training deep feedforward neural networks

It's easy for sigmoid units to saturate

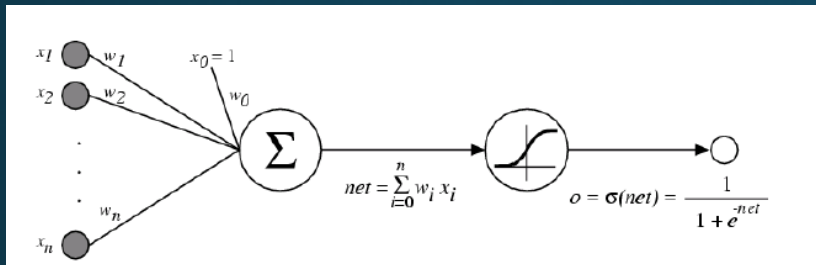$$net = \sum_{i=0}^{n} w_i \, x_i \qquad o = \sigma(net) = \frac{1}{1 + e^{-net}}$$

Learning rate approaches zero, and neuron gets "stuck"

Derivative of sigmoid function

30

It's easy for sigmoid units to saturate

It's easy for sigmoid units to saturate

- If there are 500 non-zero inputs initialized with a Gaussian ~N(0,1) then the SD is $\sqrt{500} \approx 22.4$

It's easy for sigmoid units to saturate

- Saturation visualization from Glorot & Bengio 2010 - - using a smarter initialization scheme

Bottom layer still stuck for first 100 epochs

What's Different About Modern ANNs?

Some key differences

- **Use of softmax and entropic loss instead of quadratic loss**
- Use of alternate non-linearities
  - reLU and hyperbolic tangent
- Better understanding of weight initialization
- Data augmentation
  - Especially for image data
- Ability to explore architectures rapidly

Cross-entropy loss

$$C = -\frac{1}{n} \sum_x \left[y \ln a + (1-y) \ln(1-a) \right]$$

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x x_j (\sigma(z) - y)$$

$$\frac{\partial C}{\partial w} = \sigma(z) - y$$

Cross-entropy loss

Figure 5: Cross entropy (black, surface on top) and quadratic (red, bottom surface) cost as a function of two weights (one at each layer) of a network with two layers, W_1 respectively on the first layer and W_2 on the second, output layer.

Softmax output layer

input layer hidden layer 1 hidden layer 2 hidden layer 3 output layer

Cross-entropy loss after a softmax layer gives a very simple, numerically stable gradient: $(y - a^L)$

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$

$\Delta w_{ij} = (y_i - z_i)y_j$

Network outputs a probability distribution!

Some key differences

- Use of softmax and entropic loss instead of quadratic loss.
  - **Often learning is faster and more stable as well as getting better accuracies in the limit**
- Use of alternate non-linearities
- Better understanding of weight initialization
- Data augmentation
  - Especially for image data
- Ability to explore architectures rapidly

Some key differences

- Use of softmax and entropic loss instead of quadratic loss.
  - Often learning is faster and more stable as well as getting better accuracies in the limit
- **Use of alternate non-linearities**
  - **reLU and hyperbolic tangent**
- Better understanding of weight initialization
- Data augmentation
  - Especially for image data
- Ability to explore architectures rapidly

Alternative non-linearities
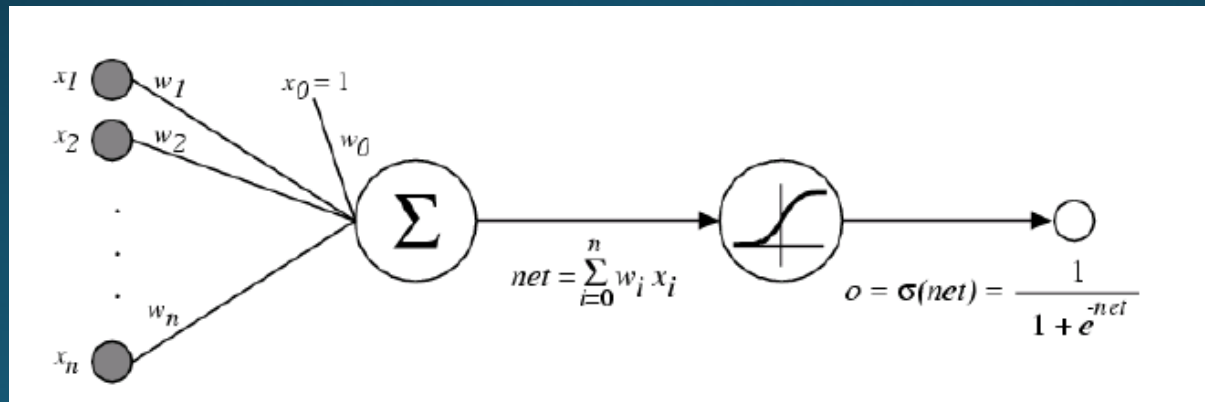
- Changes so far
  - Changed the **loss** from square error to cross-entropy (no effect at test time)
  - Proposed adding another output layer (softmax)
- A new change: modifying the nonlinearity
  - The logistic is not widely used in modern ANNs

Alternative non-linearities

- A new change: modifying the nonlinearity
  - The logistic is not widely used in modern ANNs

- Alternative #1: tanh
  - Like logistic, but shifted to range [-1, +1]

Understanding the difficulty of training deep feedforward neural networks

Alternative non-linearities

- A new change: modifying the nonlinearity
  - The logistic is not widely used in modern ANNs

- Alternative #1: tanh
  - Like logistic, but shifted to range [-1, +1]

- Alternative #2: ReLU
  - Linear with cut-off at zero

- Alternative #2.5: "Soft" ReLU
  - Doesn't saturate (at one end)
  - Sparsifies outputs
  - Helps with vanishing gradient

44

Some key differences

- Use of softmax and entropic loss instead of quadratic loss.
  - Often learning is faster and more stable as well as getting better accuracies in the limit
- Use of alternate non-linearities
  - reLU and hyperbolic tangent
- **Better understanding of weight initialization**
- Data augmentation
  - Especially for image data
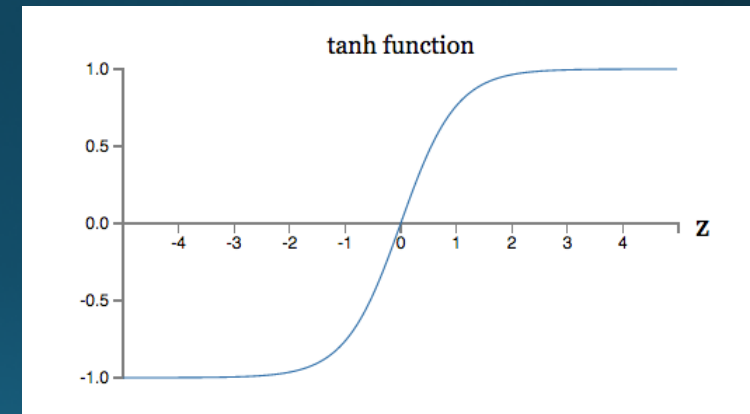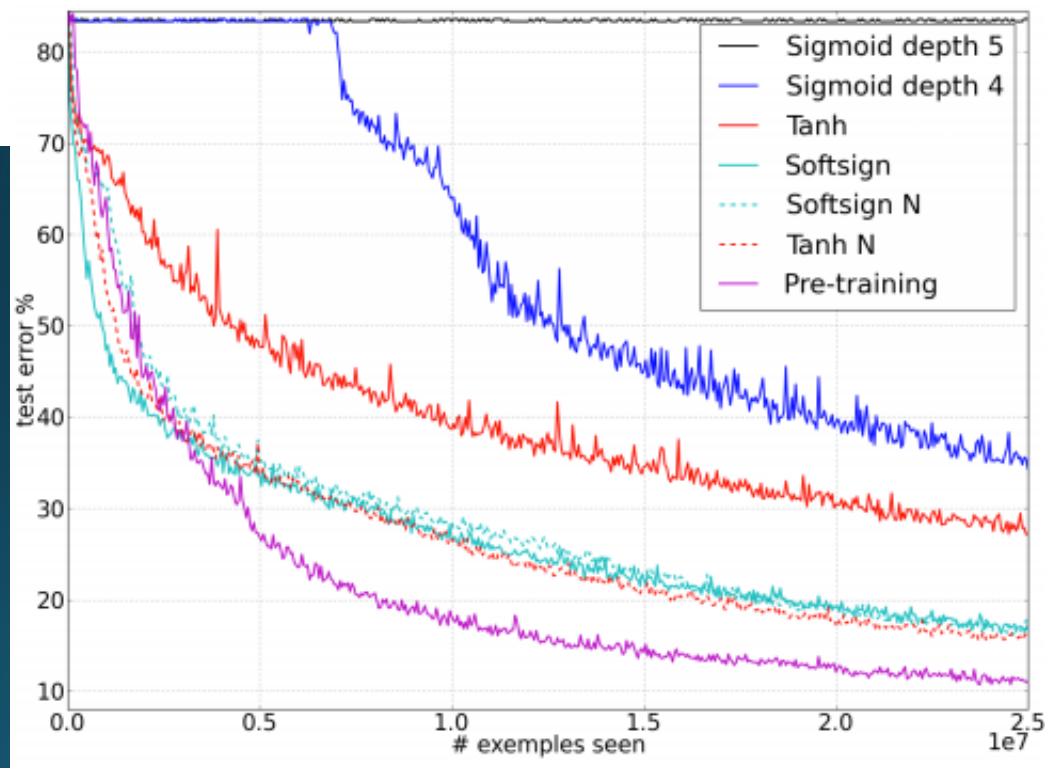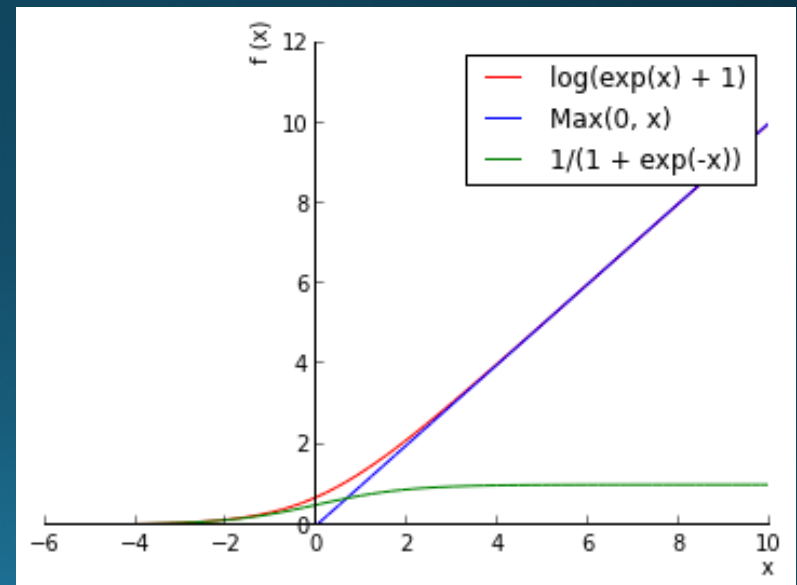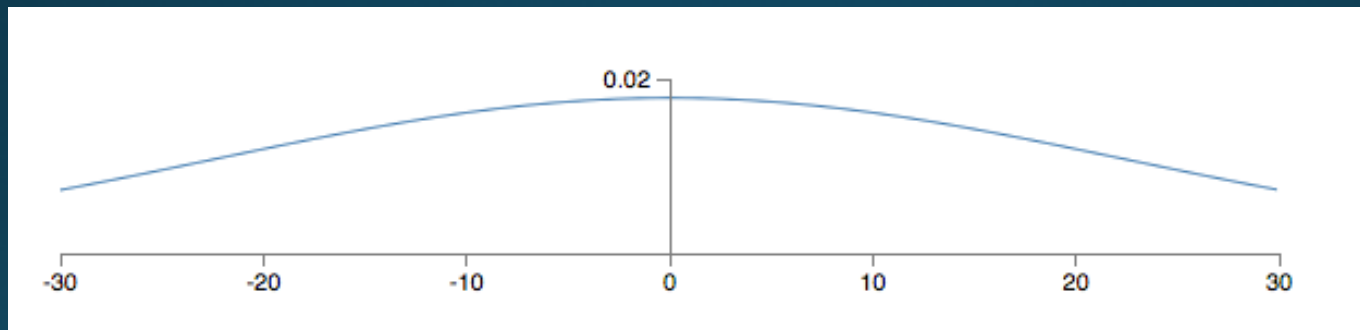- Ability to explore architectures rapidly

It's easy for sigmoid units to saturate

- If there are 500 non-zero inputs initialized with a Gaussian ~N(0,1) then the SD is $\sqrt{500} \approx 22.4$

- Common heuristics for initializing weights

$$N\left(0, \frac{1}{\sqrt{\# \text{ of inputs}}}\right) \quad U\left(\frac{-1}{\sqrt{\# \text{ of inputs}}}, \frac{1}{\sqrt{\# \text{ of inputs}}}\right)$$

Initializing to avoid saturation

- In Glorot and Bengio (2010) they suggest weights if level j (with n_j inputs) from

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$

First breakthrough deep learning results were based on clever pre-training initialization schemes, where deep networks were seeded with weights learned from unsupervised strategies

TYPE	Shapeset	MNIST	CIFAR-10	ImageNet
Softsign	16.27	1.64	55.78	69.14
Softsign N	16.06	1.72	53.8	68.13
Tanh	27.15	1.76	55.9	70.58
Tanh N	15.60	1.64	52.92	68.57

This is not always the solution – but good initialization is very important for deep nets!

Summary

- Backpropagation makes training deep neural networks possible
  - Known since 1970s, understood since 1980s, used since 1990s, tractable since 2010s
- Feed-forward versus backward propagation
  - Feed-forward evaluates the network's current configuration, J()
  - Backpropagation assigns error in J() to individual weights
- Each layer considered a function of its inputs
  - Differentiable activation functions strung together
  - Chain rule of calculus
- Modern deep architectures made possible due to logistical tweaks
  - Vanishing / Exploding gradient and new activation functions

Course Details

- How is Assignment 5 going? **Due next Tuesday!**

- Will update Assignment 5 **later today** with final project teams!

References

- "A gentle introduction to backpropagation", http://numericinsight.com/uploads/A_Gentle_Introduction_to_Backpropagation.pdf

- "Deep Feed-Forward Networks", Chapter 6, *Deep Learning Book*, http://www.deeplearningbook.org/contents/mlp.html

- "Backpropagation, Intuitions", CS231n "CNNs for Visual Recognition", https://cs231n.github.io/optimization-2/

- "How the Backpropagation Algorithm works", Chapter 2, *Neural Networks and Deep Learning*, http://neuralnetworksanddeeplearning.com/