



# Neural Networks

Rishabh Iyer

University of Texas at Dallas

Thanks to Nick Rouzzi for sharing these slides

## Announcements

① Homework 4 will be out soon

→ Due date: end of Nov 2020

② Finals on Dec 2<sup>nd</sup> 2020

→ Let me know ASAP if doesn't work

# Handwritten Digit Recognition

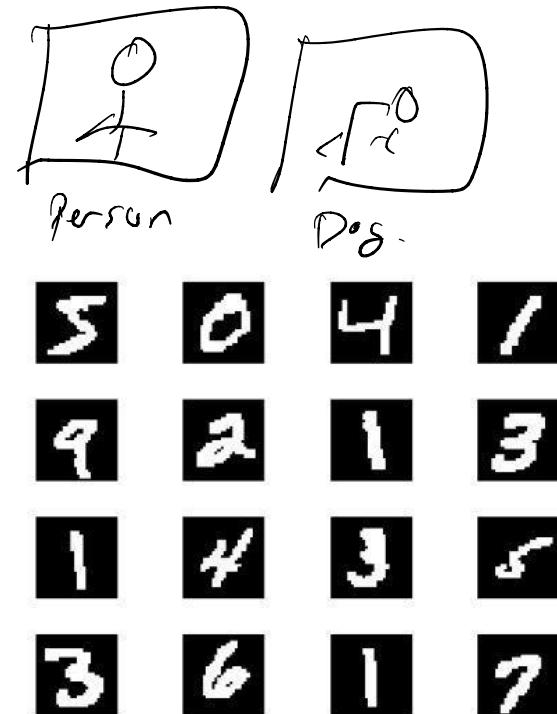


- Given a collection of handwritten digits and their corresponding labels, we'd like to be able to correctly classify handwritten digits

- A simple algorithmic technique can solve this problem with 95% accuracy

$\xrightarrow{\text{Logistic Regression}}$

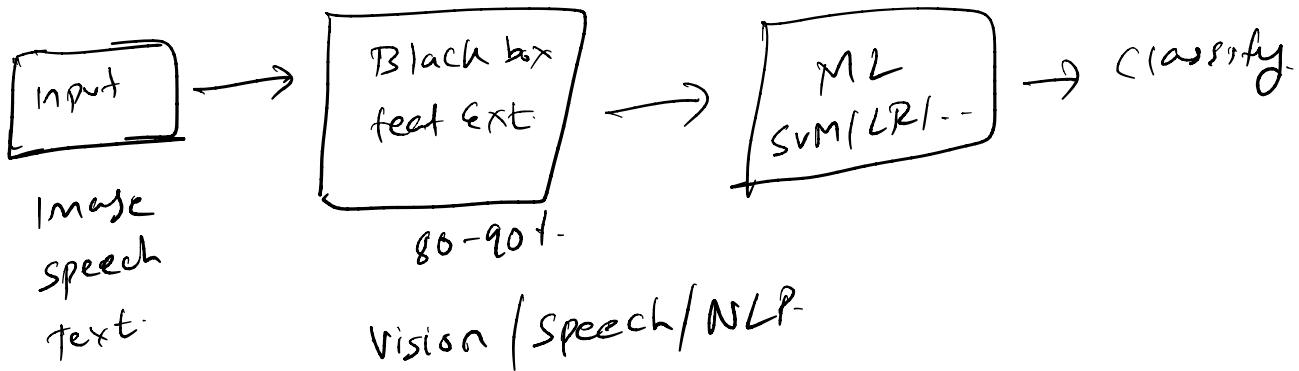
- State-of-the-art methods can achieve near 99% accuracy
- (you've probably seen these in action if you've deposited a check recently)



Digits from the MNIST data set

CIFAR-10  
 $10 \rightarrow$  Random classifier

## Pre Deep Learning,



# Neural Networks

---

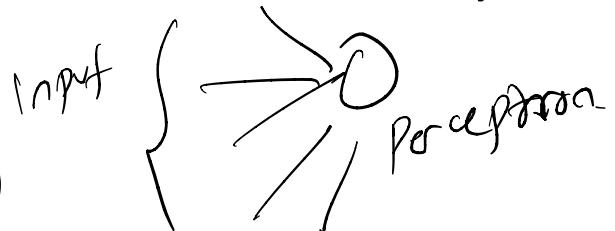


- The basis of neural networks was developed in the 1940s -1960s
  - The idea was to build mathematical models that might “compute” in the same way that neurons in the brain do
  - As a result, neural networks are biologically inspired, though many of the algorithms developed for them are not biologically plausible
  - Perform surprisingly well for the handwritten digit recognition task (and many others)

# Neural Networks

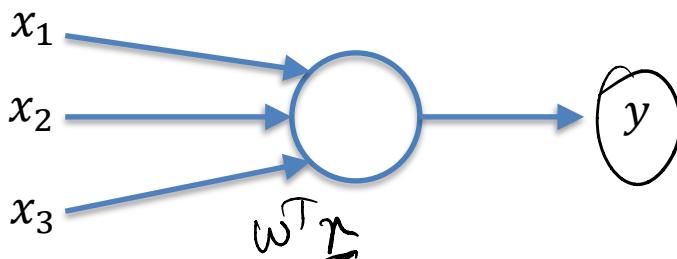


- Neural networks consist of a collection of artificial neurons
- There are different types of neuron models that are commonly studied
  - The perceptron (one of the first studied)
  - The sigmoid neuron (one of the most common, but many more)
  - Rectified linear units (ReLU)
- A neural network is a directed graph consisting of a collection of neurons (the nodes), directed edges (each with an associated weight), and a collection of fixed binary inputs



# The Perceptron

- A perceptron is an artificial neuron that takes a collection of **binary** inputs and produces a binary output
  - The output of the perceptron is determined by summing up the weighted inputs and thresholding the result: if the weighted sum is larger than the threshold, the output is one (and zero otherwise)

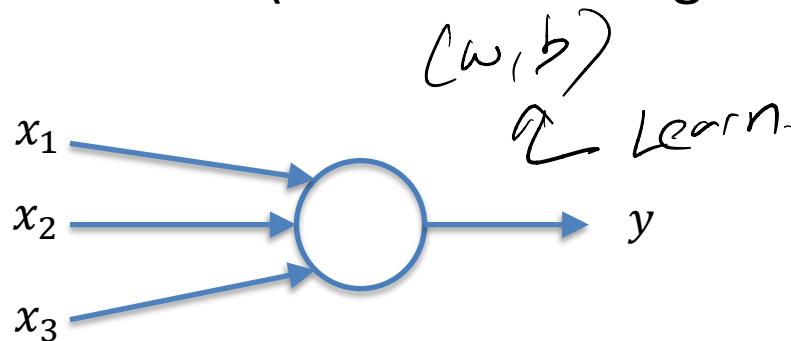


$$y = \begin{cases} 1 & \underline{w_1}x_1 + \underline{w_2}x_2 + \underline{w_3}x_3 > \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

$$\underline{w^T n} + b > 0$$

# Perceptrons

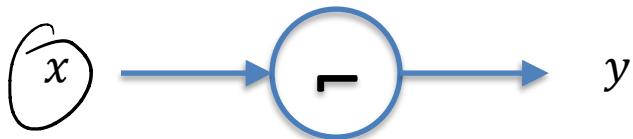
- Perceptrons are usually expressed in terms of a collection of input weights and a bias  $b$  (which is the negative threshold)



$$y = \begin{cases} 1 & w_1x_1 + w_2x_2 + w_3x_3 + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

- A single node perceptron is just a linear classifier
- This is actually where the “perceptron algorithm” comes from

# Perceptron for NOT



- Choose  $w = -1$ , threshold =  $-.5$

- $y = \begin{cases} 1 & -x > -.5 \\ 0 & -x \leq -.5 \end{cases}$

$\leftarrow x < \overline{0.5}$   
 $\leftarrow x \geq 0.5$

$$\begin{aligned} w^T m + b &> 0 \\ -x + 0.5 &> 0 \\ w = -1 & \\ b = 0.5 & \end{aligned}$$

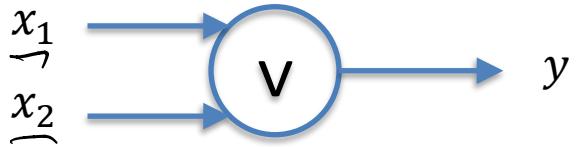
$$x \in \{0, 1\}$$

# Perceptron for OR

---



# Perceptron for OR



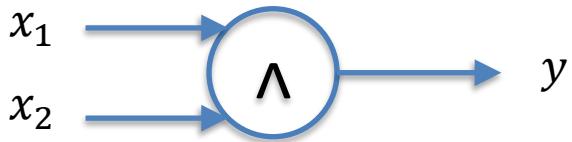
- Choose  $w_1 = w_2 = 1$ , threshold = 0
- $y = \begin{cases} 1 & x_1 + x_2 > 0 \\ 0 & x_1 + x_2 \leq 0 \end{cases}$

# Perceptron for AND

---



# Perceptron for AND



- Choose  $w_1 = w_2 = 1$ , threshold = 1.5

$$\bullet \quad y = \begin{cases} 1 & x_1 + x_2 > 1.5 \\ 0 & x_1 + x_2 \leq 1.5 \end{cases}$$

# Perceptron for XOR

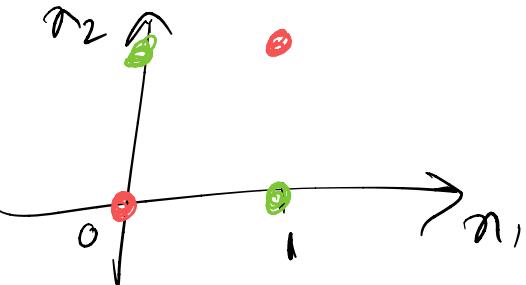
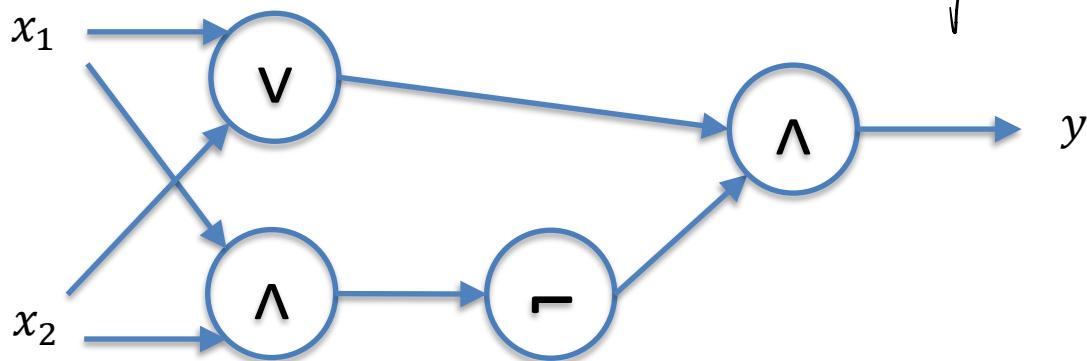
---



# Perceptron for XOR



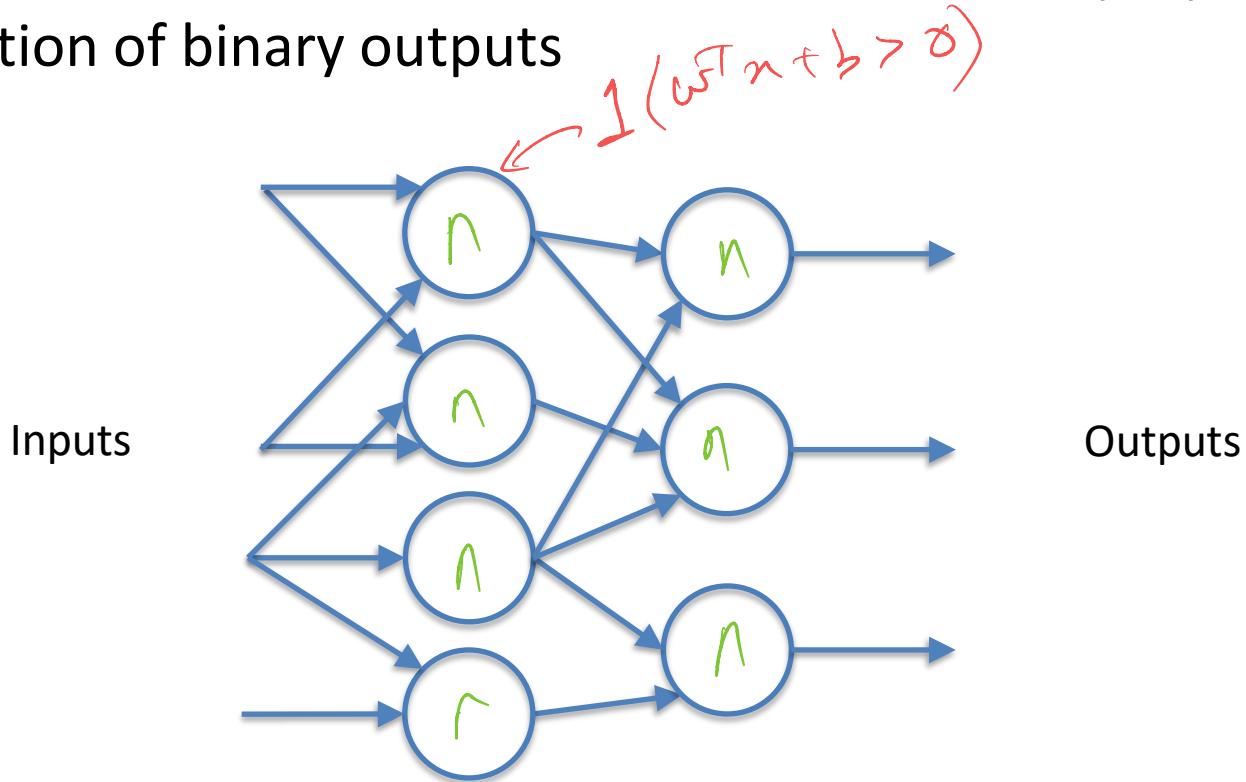
- Need more than one perceptron!



- Weights for incoming edges are chosen as before
- Networks of perceptrons can encode any circuit!

# Neural Networks

- Gluing a bunch of perceptrons together gives us a neural network
- In general, neural nets have a collection of binary inputs and a collection of binary outputs



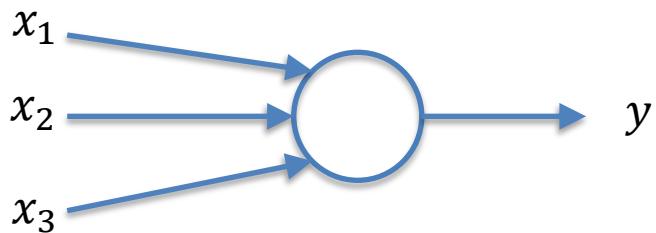
# Beyond Perceptrons



- Given a collection of input-output pairs, we'd like to learn the weights of the neural network so that we can correctly predict the output of an unseen input
  - We could try learning via gradient descent (e.g., by minimizing the Hamming loss)  
← 0/1 Loss
    - This approach doesn't work so well: small changes in the weights can cause dramatic changes in the output
    - This is a consequence of the discontinuity of sharp thresholding (same problem we saw with perceptron alg.)

# The Sigmoid Neuron

- A sigmoid neuron is an artificial neuron that takes a collection of real inputs and produces an output in the interval  $[0,1]$ 
  - The output is determined by summing up the weighted inputs plus the bias and applying the sigmoid function to the result



$$y = \sigma(w_1x_1 + w_2x_2 + w_3x_3 + b)$$

where  $\sigma$  is the sigmoid function

# The Sigmoid Function

- The sigmoid function is a continuous function that approximates a step function

$$(1 - G(z)) \rightarrow$$

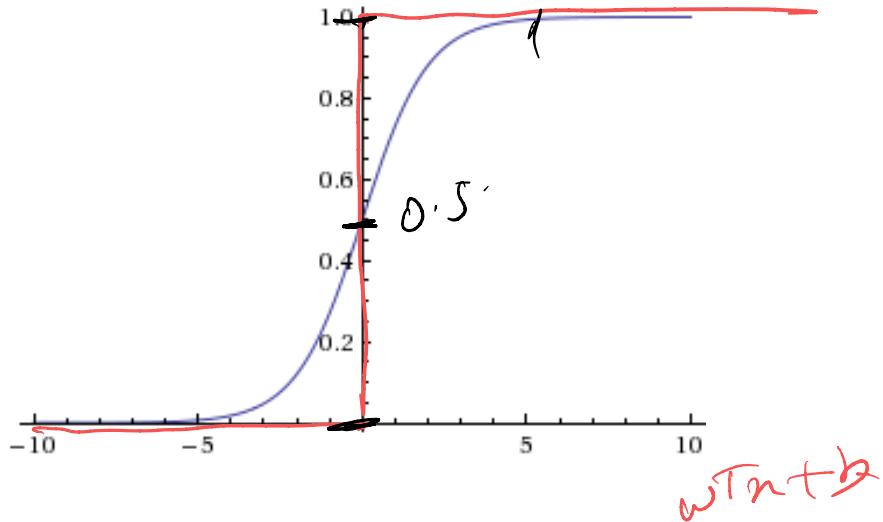
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$P(y=1|x) = G(w^T x + b)$$

$$= \frac{1}{1 + e^{-(w^T x + b)}}$$

$\rightarrow 1(w^T x + b)$

$\rightarrow G(w^T x + b)$

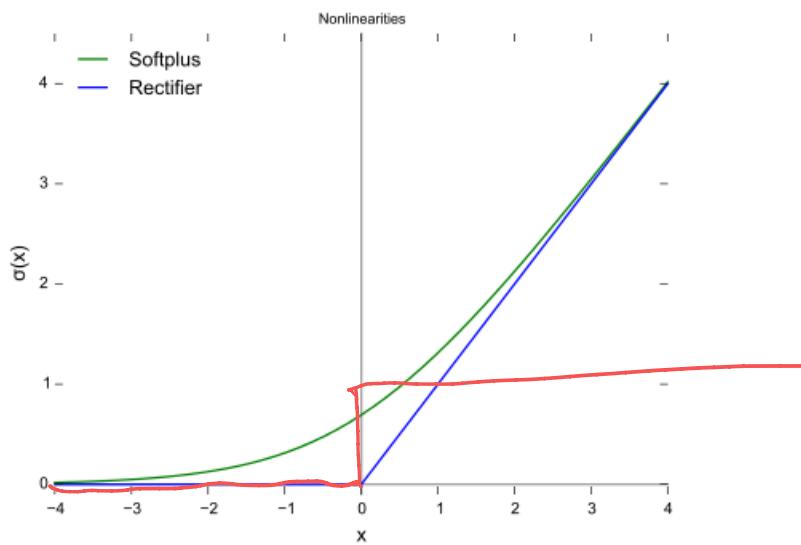


$$P(y=1|x) > 0.5$$

$$\Rightarrow w^T x + b > 0$$

# Rectified Linear Units

- The sigmoid neuron approximates a step function as a smooth function
- The relu is given by  $\max(0, x)$  which can be approximated as a smooth continuous function  $\underbrace{\ln(1 + e^x)}$



# Softmax

- The softmax function maps a vector of real numbers to a vector of probabilities as

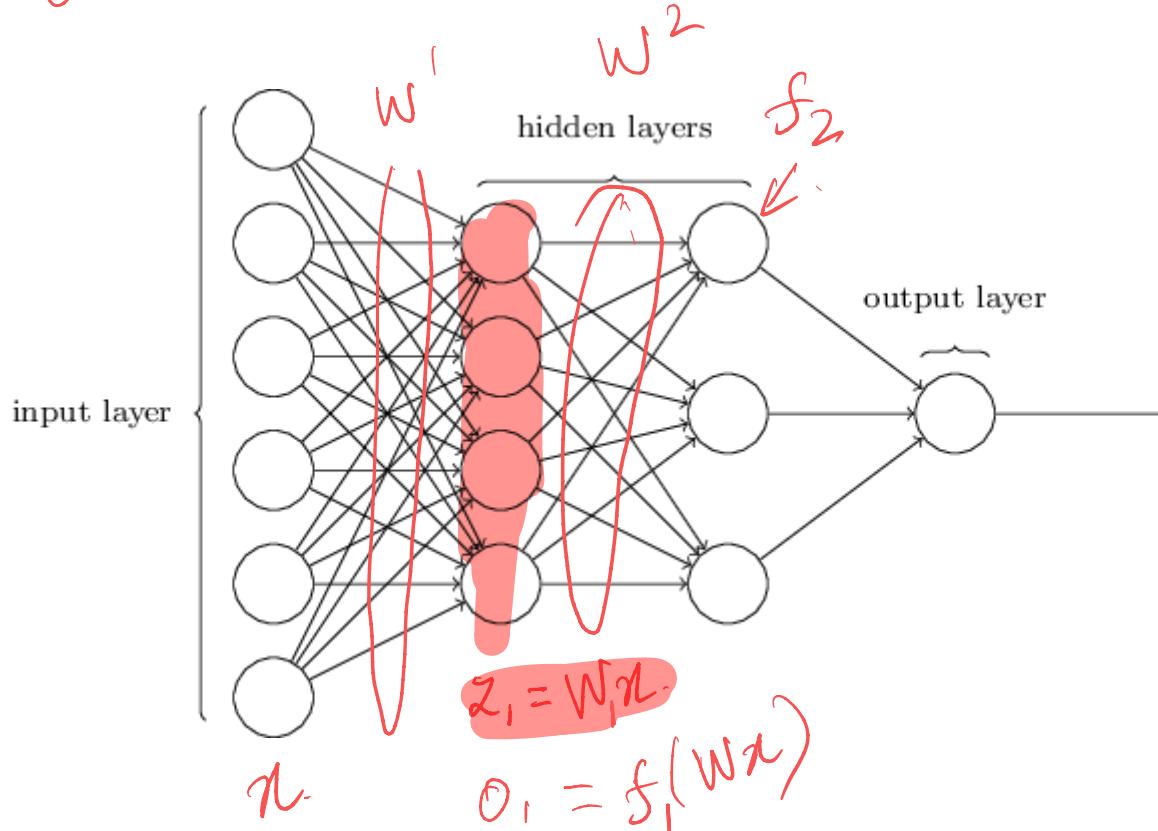
$$\text{softmax}(z)_j = \frac{e^{z_j}}{\sum_k e^{z_k}} \approx 1 \left( j = \arg \max_k z_k \right)$$

- If there is a dominant value in  $z$ , then it will become one under the softmax
- Often used as the final layer of a neural network

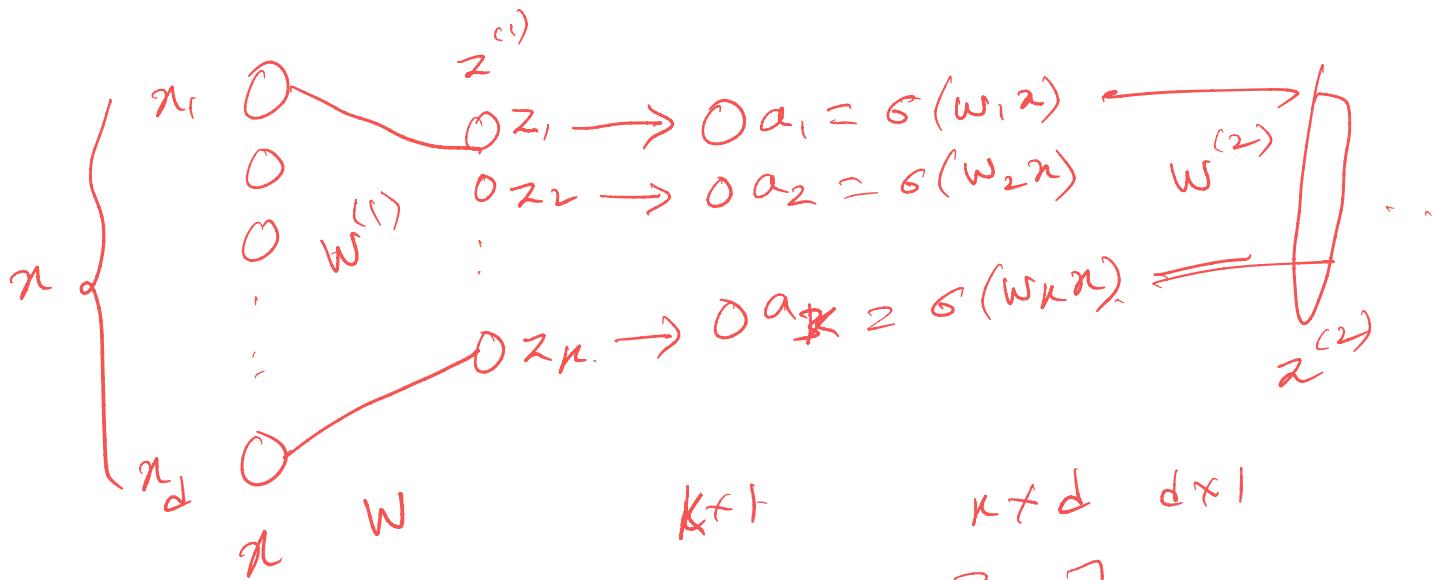
# Multilayer Neural Networks



$$g(a) = \underline{f_2}(\underline{w^2} \underline{f_1}(w^1 a))$$



from Neural Networks and Deep Learning by Michael Nielson



$$z_1 = w_1 x$$

$$z_2 = w_2 x.$$

$$\vdots$$

$$z_k = w_k x.$$

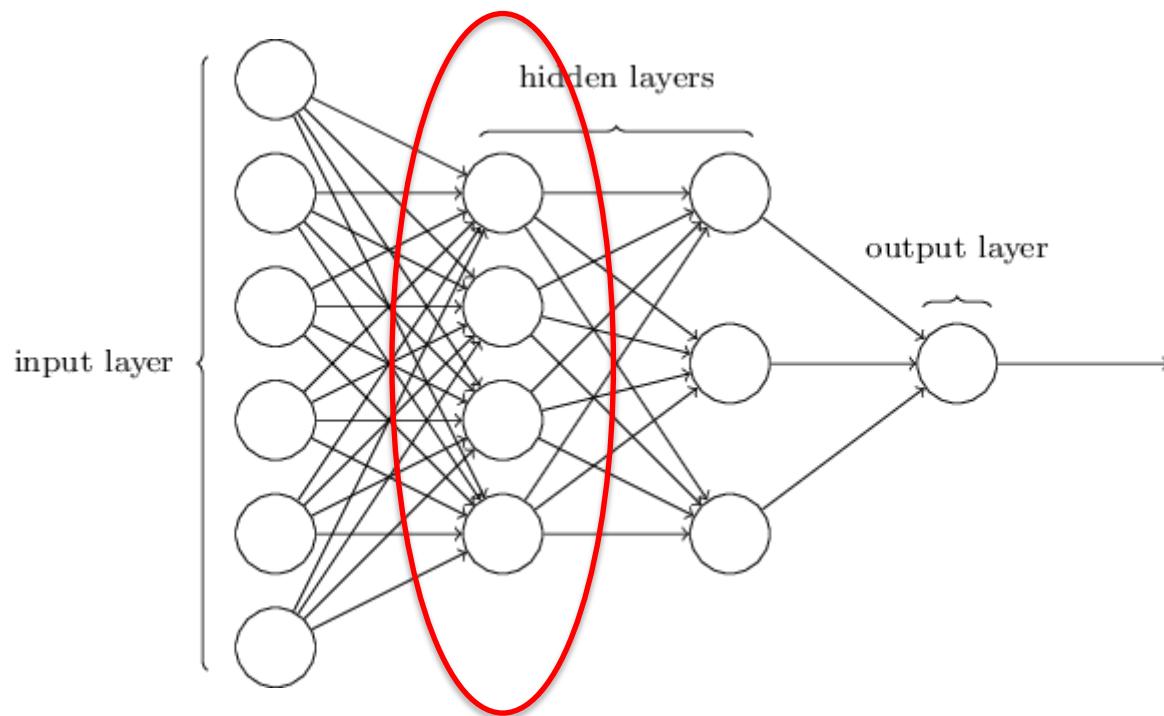
$$\begin{bmatrix} z \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_k \end{bmatrix} x.$$

$k+1$        $k+d$      $d \times 1$

# Multilayer Neural Networks

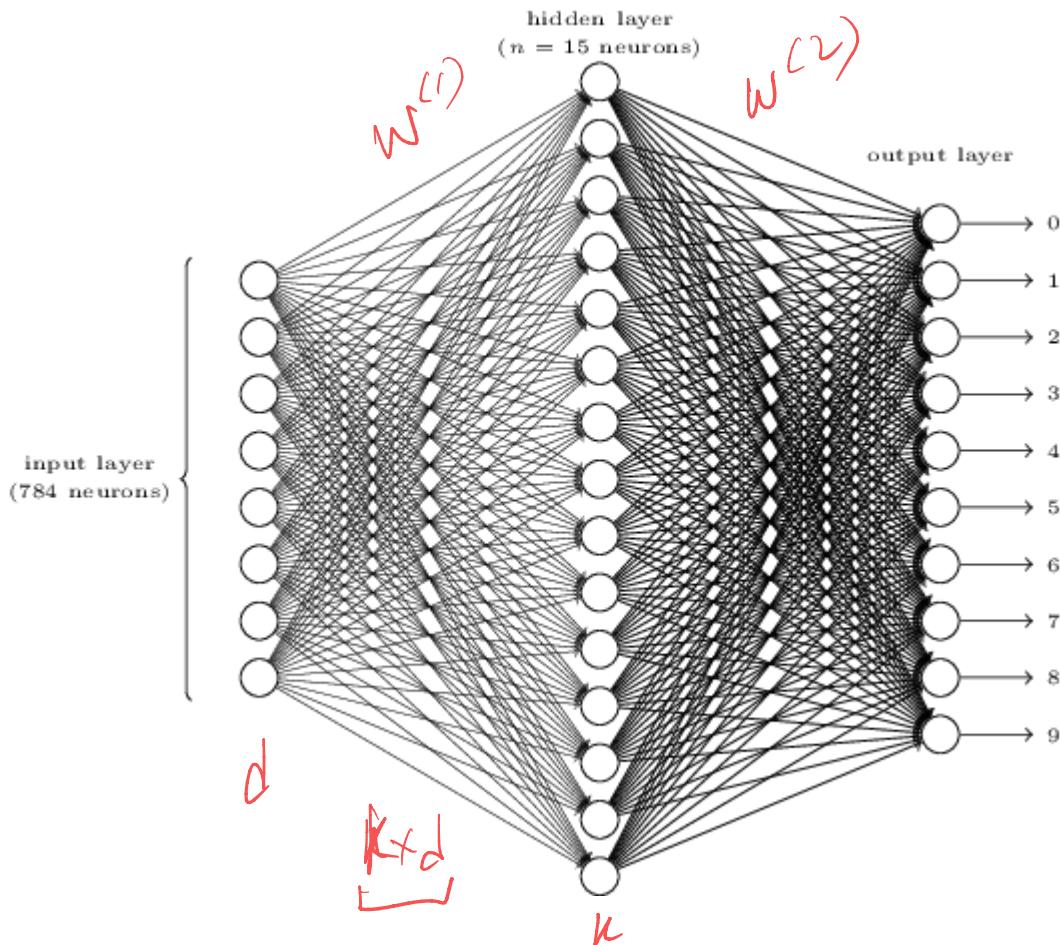


NO intralayer connections



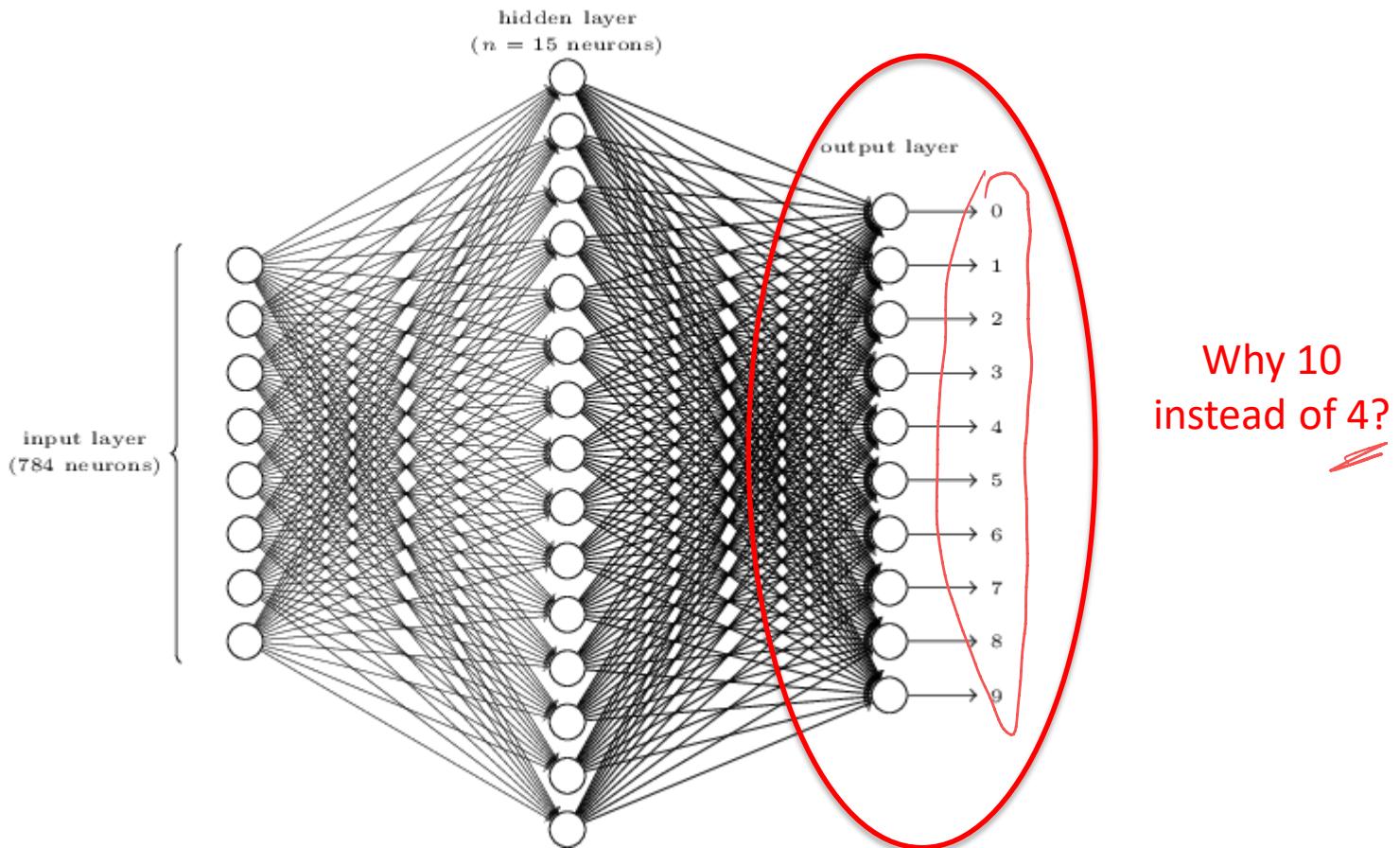
from Neural Networks and Deep Learning by Michael Nielson

# Neural Network for Digit Classification



from Neural Networks and Deep Learning by Michael Nielson

# Neural Network for Digit Classification



from Neural Networks and Deep Learning by Michael Nielson

# Expressiveness of NNs

- Boolean functions
  - Every Boolean function can be represented by a network with a single hidden layer consisting of possibly exponentially many hidden units
- Continuous functions
  - Every bounded continuous function can be approximated up to arbitrarily small error by a network with one hidden layer
  - Any function can be approximated to arbitrary accuracy with two hidden layers

$\text{poly}(\frac{1}{\epsilon})$

$\epsilon$

# Expressiveness of NNs

- **Theorem [Zhang et al. 2016]:** There exists a two-layer neural network with ReLU activations and  $2n + d$  weights that can represent any function on a sample of size  $n$  in  $d$  dimensions
- This should mean that it is very easy to overfit with neural networks
- Generalization performance of networks is difficult to assess theoretically

# Training Neural Networks

- To do the learning, we first need to define a loss function to minimize

$$C(w, b) = \frac{1}{2M} \sum_m \|y^m - a(x^m, w, b)\|^2$$

cross-entropy, Logistic Loss

$w, b$

- The training data consists of input output pairs  $(x^1, y^1), \dots, (x^M, y^M)$

$$x \rightarrow \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \dots \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} a(x, w, b)$$

- $a(x^m, w, b)$  is the output of the neural network for the  $m^{th}$  sample
- $w$  and  $b$  are the weights and biases

$$\frac{1}{2M} \left( y^{(m)} - a(x^{(m)}, w, b) \right)^2$$

# Gradient of the Loss

- The derivative of the loss function is calculated as follows

$$\frac{\partial C(w, b)}{\partial w_k} = \frac{1}{M} \sum_m [y^m - a(x^m, w, b)] \frac{\partial a(x^m, w, b)}{\partial w_k}$$

- To compute the derivative of  $a$ , use the chain rule and the derivative of the sigmoid function

$$\frac{d\sigma(z)}{dz} = \sigma(z) \cdot (1 - \sigma(z))$$

- This gets complicated quickly with lots of layers of neurons

# Stochastic Gradient Descent

- To make the training more practical, stochastic gradient descent is used instead of standard gradient descent
- Recall, the idea of stochastic gradient descent is to approximate the gradient of a sum by sampling a few indices and averaging

$$\nabla_x \sum_{i=1}^n f_i(x) \approx \frac{1}{K} \sum_{k=1}^K \nabla_x f_{i^k}(x)$$

minibatch of  $K$   
examples  
 $i^k, k=1:K$

here, for example, each  $i^k$  is sampled uniformly at random from  $\{1, \dots, n\}$

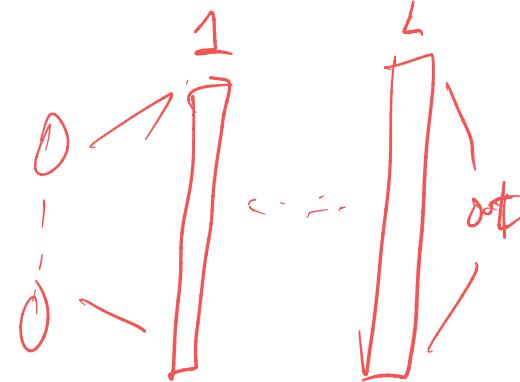
# Computing the Gradient

- We'll compute the gradient for a single sample

$$C(w, b) = \frac{1}{2} \|y - a(x, w, b)\|^2$$

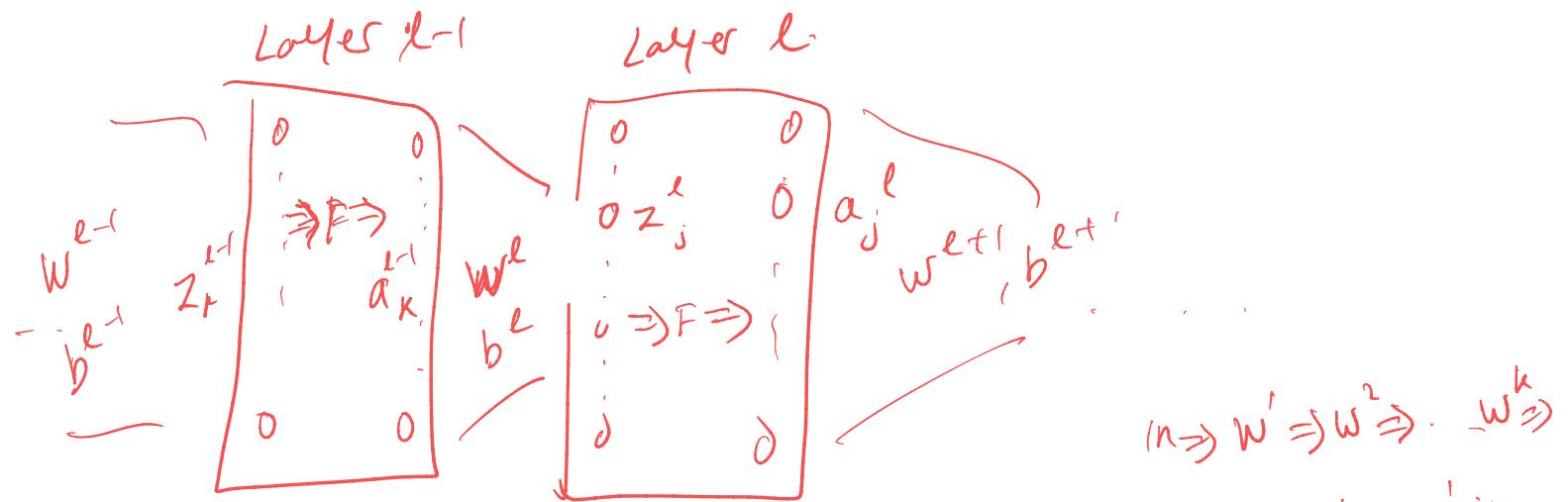
- Some definitions:

- $L$  is the number of layers
- $a_j^l$  is the output of the  $j^{th}$  neuron on the  $l^{th}$  layer
- $z_j^l$  is the weighted input of the  $j^{th}$  neuron on the  $l^{th}$  layer



$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$$

- $\delta_j^l$  is defined to be  $\frac{\partial C}{\partial z_j^l}$



$$z_j^l = \sum_k w_{jk}^l a_{k.}^{l-1} + b_j^l, \quad \forall j$$

( $\Rightarrow w^1 \Rightarrow w^2 \Rightarrow \dots \Rightarrow w^k$ )

$$\theta = \frac{w^k w^{k-1} \dots w^1}{w \text{ in}}$$

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

vector  $a^l = f(z^l)$ ,  $\Rightarrow$

$a_j^l = f(z_j^l)$

$\theta$   $\hookrightarrow$  per  $j$

# Computing the Gradient

For the output layer, we have the following partial derivative

$$\begin{aligned}
 \frac{\partial C}{\partial z_j^L} &= -(\underbrace{y_j - a_j^L}_{\text{Error}}) \frac{\partial a_j^L}{\partial z_j^L} \\
 &= -(y_j - a_j^L) \frac{\partial \sigma(z_j^L)}{\partial z_j^L} \\
 &= -(y_j - a_j^L) \sigma(z_j^L) (1 - \sigma(z_j^L)) \\
 &= \delta_j^L
 \end{aligned}$$

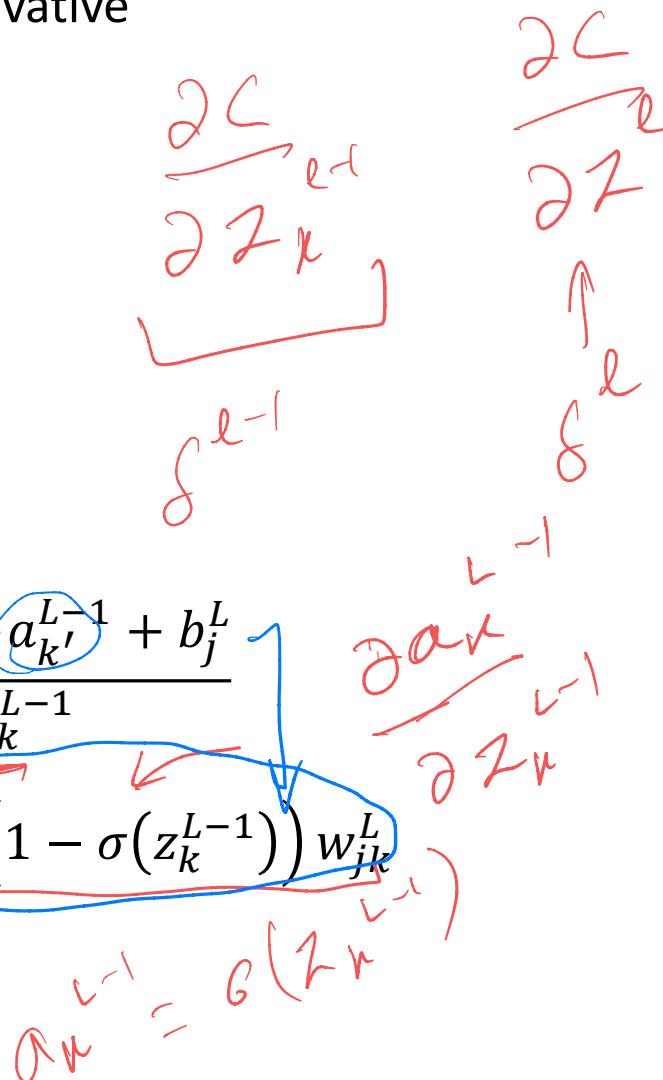
$$\text{Error} = [y - \underline{a(w, b, x)}]^2$$

- For simplicity, we will denote the vector of all such partials for each node in the  $l^{th}$  layer as  $\delta^l$

# Computing the Gradient

For the  $L - 1$  layer, we have the following partial derivative

$$\begin{aligned}
 \frac{\partial C}{\partial z_k^{L-1}} &= \sum_j (a_j^L - y_j) \frac{\partial a_j^L}{\partial z_k^{L-1}} \\
 &= \sum_j (a_j^L - y_j) \frac{\partial \sigma(z_j^L)}{\partial z_k^{L-1}} \\
 &= \sum_j (a_j^L - y_j) \sigma(z_j^L) (1 - \sigma(z_j^L)) \frac{\partial z_j^L}{\partial z_k^{L-1}} \\
 &= \sum_j (a_j^L - y_j) \sigma(z_j^L) (1 - \sigma(z_j^L)) \frac{\partial \sum_{k'} w_{jk}^L a_{k'}^{L-1} + b_j^L}{\partial z_k^{L-1}} \\
 &= \sum_j (a_j^L - y_j) \sigma(z_j^L) (1 - \sigma(z_j^L)) \sigma(z_k^{L-1}) (1 - \sigma(z_k^{L-1})) w_{jk}^L \\
 &= ((\delta^L)^T w_{*k}^L) (1 - \sigma(z_k^{L-1})) \sigma(z_k^{L-1})
 \end{aligned}$$



# Computing the Gradient

- We can think of  $w^l$  as a matrix
- This allows us to write

$$(w^L)^T \delta^L \quad DF_{z^{L-1}}(z^{L-1})$$

$$\delta^{L-1} = ((\delta^L)^T w^L)^T \circ (1 - \sigma(z^{L-1})) \circ \sigma(z^{L-1})$$

where  $\sigma(z^{L-1})$  is the vector whose  $k^{th}$  component is  $\sigma(z_k^{L-1})$

- Applying the same strategy, for  $l < L$

$$\delta^l = ((\delta^{l+1})^T w^{l+1})^T \circ (1 - \sigma(z^l)) \circ \sigma(z^l)$$

$$Wx$$

$$x \cdot y = \sum_i x_i y_i$$

$$x^T y = [x_i y_i]$$

# Computing the Gradient

- Now, for the partial derivatives that we care about

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

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

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

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$$

- We can compute these derivatives one layer at a time!

# Backpropagation

- Compute the inputs/outputs for each layer by starting at the input layer and applying the sigmoid functions
- Compute  $\delta^L$  for the output layer

$$\delta^L, \delta^{L-1}, \dots, \delta^1$$

$$\underbrace{\delta_j^L = -(y_j - a_j^L) \sigma(z_j^L) (1 - \sigma(z_j^L))}_{\text{for } l=L}$$

- Starting from  $l = L - 1$  and working backwards, compute

$$\underbrace{\delta^l = ((\delta^{l+1})^T w^{l+1})^T \circ \sigma(z^l) \circ (1 - \sigma(z^l))}_{\text{for } l=L-1, \dots, 1}$$

- Perform gradient descent

$$\begin{aligned} b_j^l &= b_j^l - \gamma \cdot \delta_j^l \\ w_{jk}^l &= w_{jk}^l - \gamma \cdot \delta_j^l a_k^{l-1} \end{aligned} \quad \left. \begin{array}{c} \downarrow \\ \forall l \end{array} \right.$$

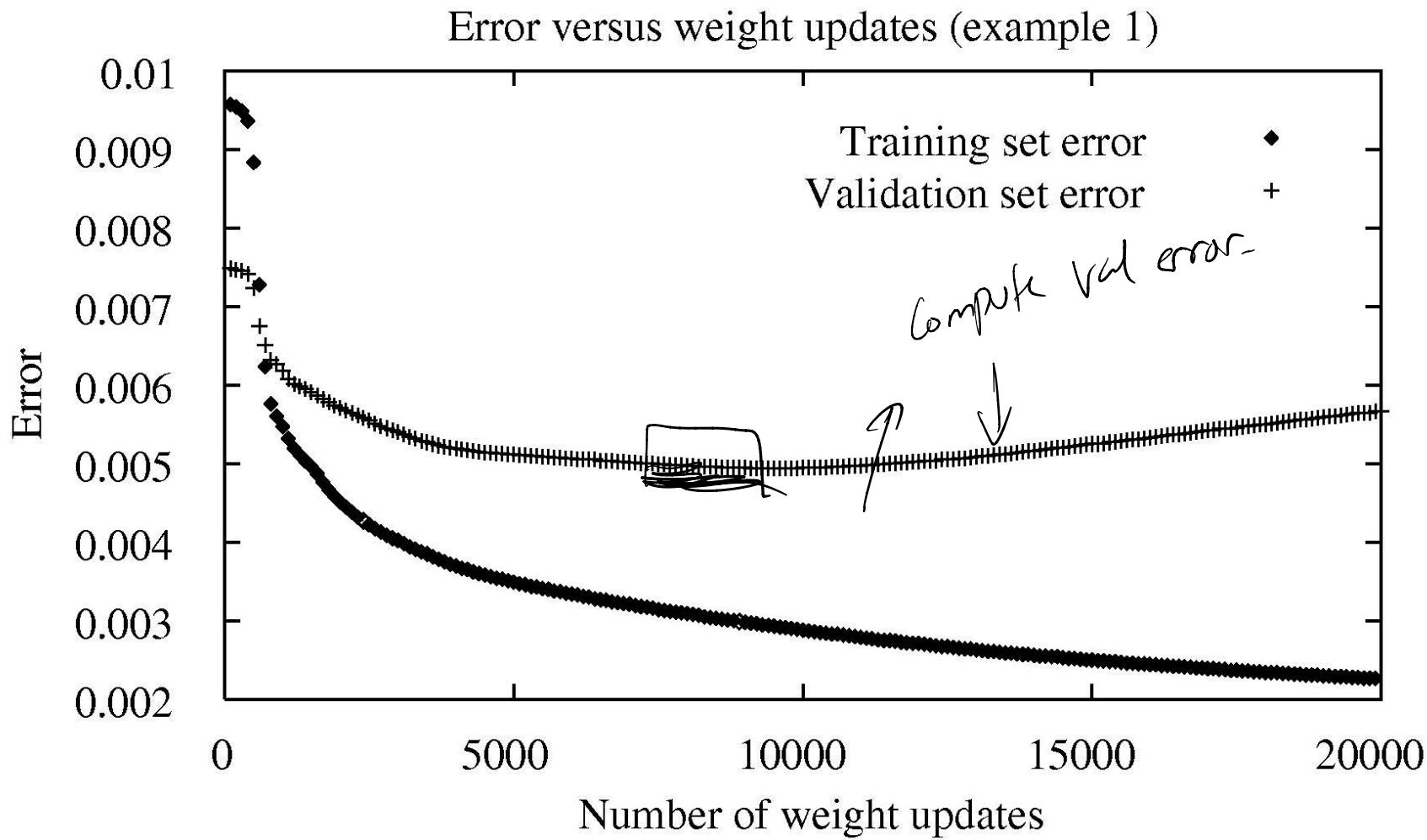
# Backpropagation

Computing gradient -

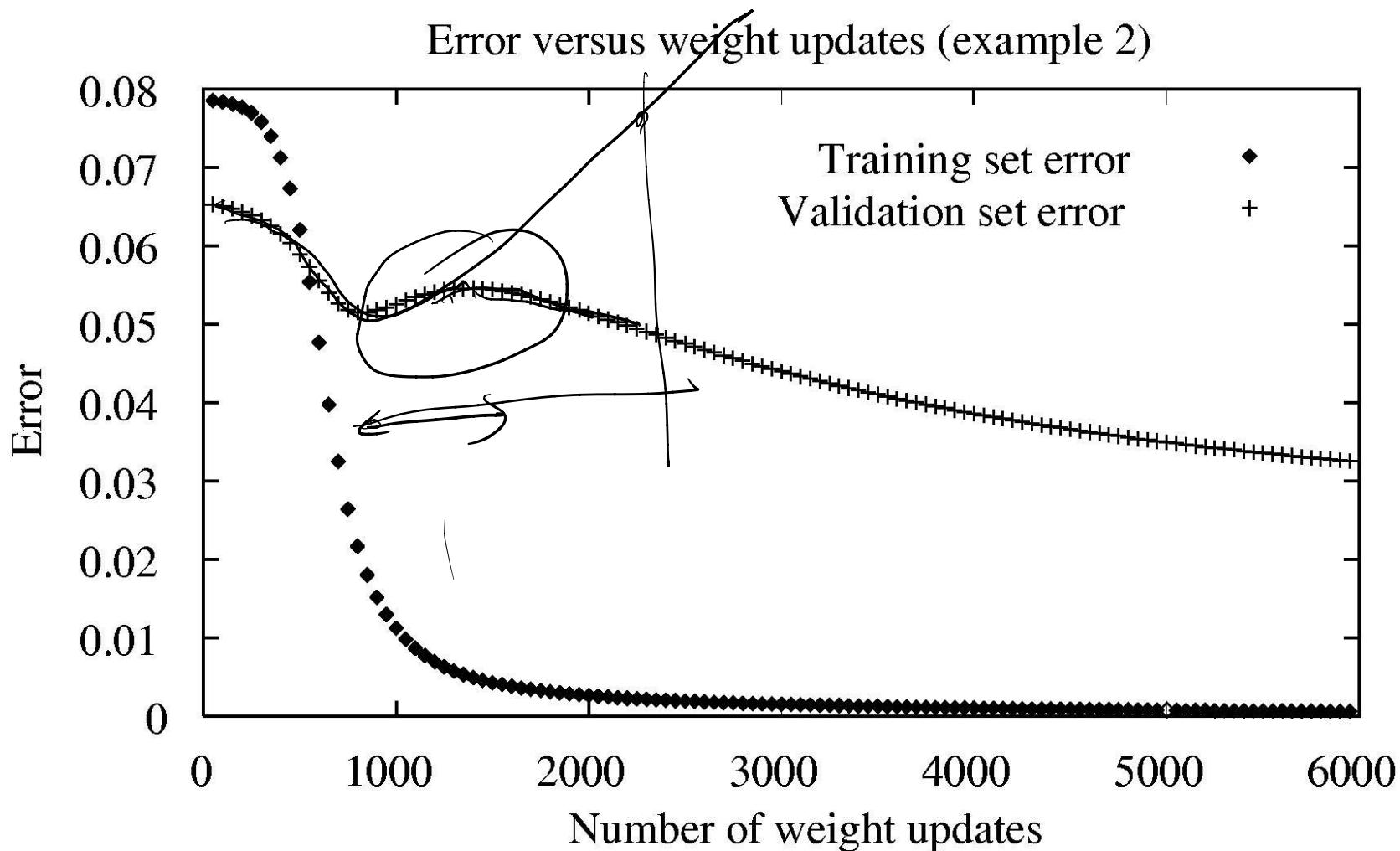
- Backpropagation converges to a local minimum (loss is not convex in the weights and biases)
  - Like EM, can just run it several times with different initializations
  - Training can take a very long time (even with stochastic gradient descent) ← h̄f̄o
  - Prediction after learning is fast
  - Sometimes include a **momentum** term  $\alpha$  in the gradient update

$$w(t) = w(t-1) - \gamma \cdot \nabla_w C(t-1) + \alpha(-\gamma \cdot \nabla_w C(t-2))$$

# Overfitting



# Overfitting



# Neural Networks in Practice



- Many ways to improve weight learning in NNs
  - Use a regularizer! (better generalization?)
  - Try other loss functions, e.g., the cross entropy
$$-y \log a(x, w, b) - (1 - y) \log(1 - a(x, w, b))$$
  - Initialize the weights of the network more cleverly
    - Random initializations are likely to be far from optimal
  - The learning procedure can have numerical difficulties if there are a large number of layers

$\Downarrow$   
Sigmoid  $\Rightarrow$  ReLU  
today -

# Regularized Loss

- Penalize learning large weights

$$C'(w, b) = \frac{1}{2M} \sum_m \|y^m - a(x^m, w, b)\|^2 + \frac{\lambda}{2} \|w\|_2^2$$

- Can still use the backpropagation algorithm in this setting
- $\ell_1$  regularization can also be useful
- Regularization can help with convergence,  $\lambda$  should be chosen with a validation set

# Dropout

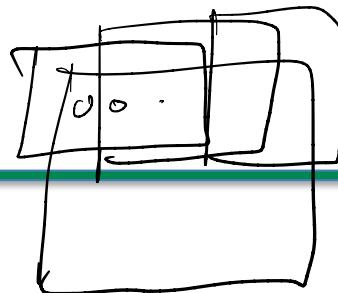
---

- A heuristic bagging-style approach applied to neural networks to counteract overfitting
  - Randomly remove a certain percentage of the neurons from the network and then train only on the remaining neurons
  - The networks are recombined using an approximate averaging technique (keeping around too many networks and doing proper bagging can be costly in practice)

# Other Techniques

- Early stopping
  - Stop the learning early in the hopes that this prevents overfitting ← Val set error
- Parameter tying
  - Assume some of the weights in the model are the same to reduce the dimensionality of the learning problem
  - Also a way to learn “simpler” models
  - Can lead to significant compression in neural networks (i.e., >90%)

# Other Ideas



- Convolutional neural networks
  - Instead of the output of every neuron at layer  $l$  being used as an input to every neuron at layer  $l + 1$ , the edges between layers are chosen more locally
  - Many tied weights and biases (i.e., convolution nets apply the same process to many different local chunks of neurons)
  - Often combined with pooling layers (i.e., layers that, say, half the number of neurons by replacing small regions of neurons with their maximum output)
  - Used extensively for image classification tasks