



# Neural Networks

Rishabh Iyer

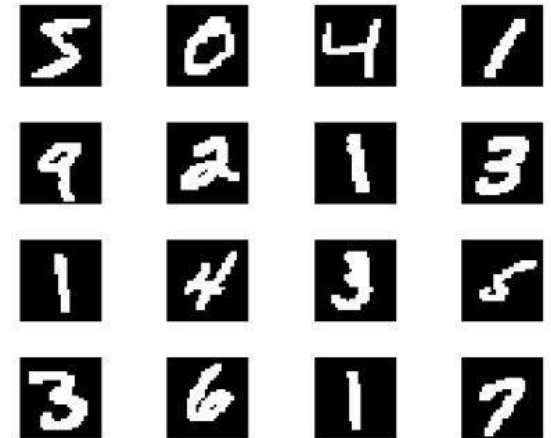
University of Texas at Dallas

Thanks to Nick Rouzzi for sharing these slides

# 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
  - 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

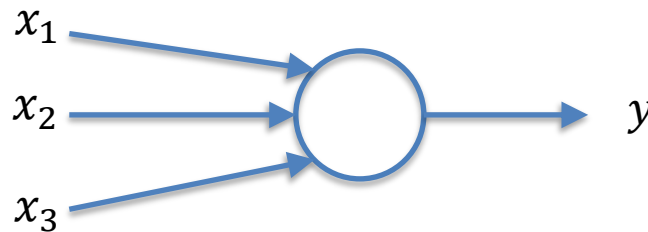
- The basis of neural networks was developed in the 1940s -1960s
  - The idea was to build mathematical models that might “compute” in the same way that neurons in the brain do
  - As a result, neural networks are biologically inspired, though many of the algorithms developed for them are not biologically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

- Neural networks 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
- 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

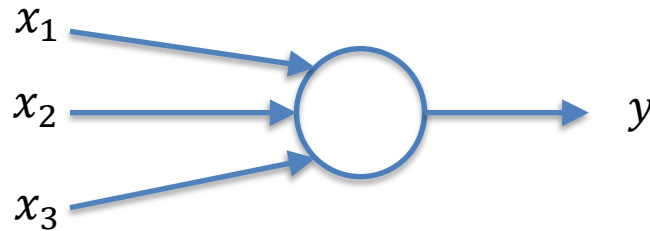


- A perceptron is an artificial neuron that takes a collection of binary inputs and produces a binary output
- The output of the perceptron is determined by summing up the weighted inputs and thresholding the result: if the weighted sum is larger than the threshold, the output is one (and zero otherwise)



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

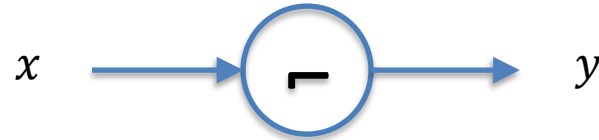
- Perceptrons are usually expressed in terms of a collection of input weights and a bias  $b$  (which is the negative threshold)



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

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

# Perceptron for NOT



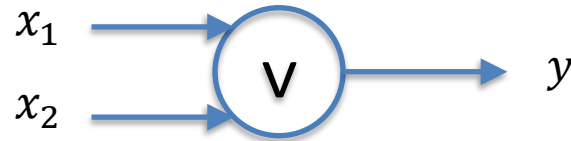
- Choose  $w = -1$ , threshold  $= -.5$
- $y = \begin{cases} 1 & -x > -.5 \\ 0 & -x \leq -.5 \end{cases}$

# Perceptron for OR





# Perceptron for OR

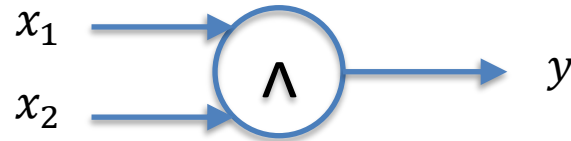


- Choose  $w_1 = w_2 = 1$ , threshold = 0
- $$y = \begin{cases} 1 & x_1 + x_2 > 0 \\ 0 & x_1 + x_2 \leq 0 \end{cases}$$

# Perceptron for AND



# Perceptron for AND



- Choose  $w_1 = w_2 = 1$ , threshold = 1.5
- $$y = \begin{cases} 1 & x_1 + x_2 > 1.5 \\ 0 & x_1 + x_2 \leq 1.5 \end{cases}$$

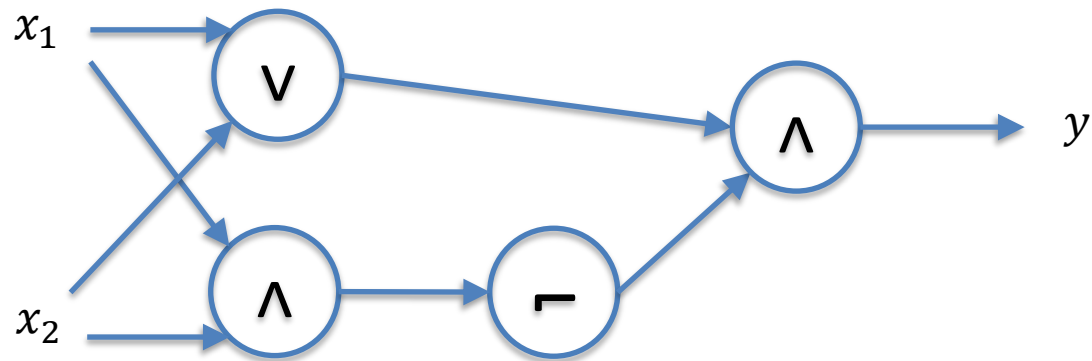
# Perceptron for XOR



# Perceptron for XOR



- Need more than one perceptron!

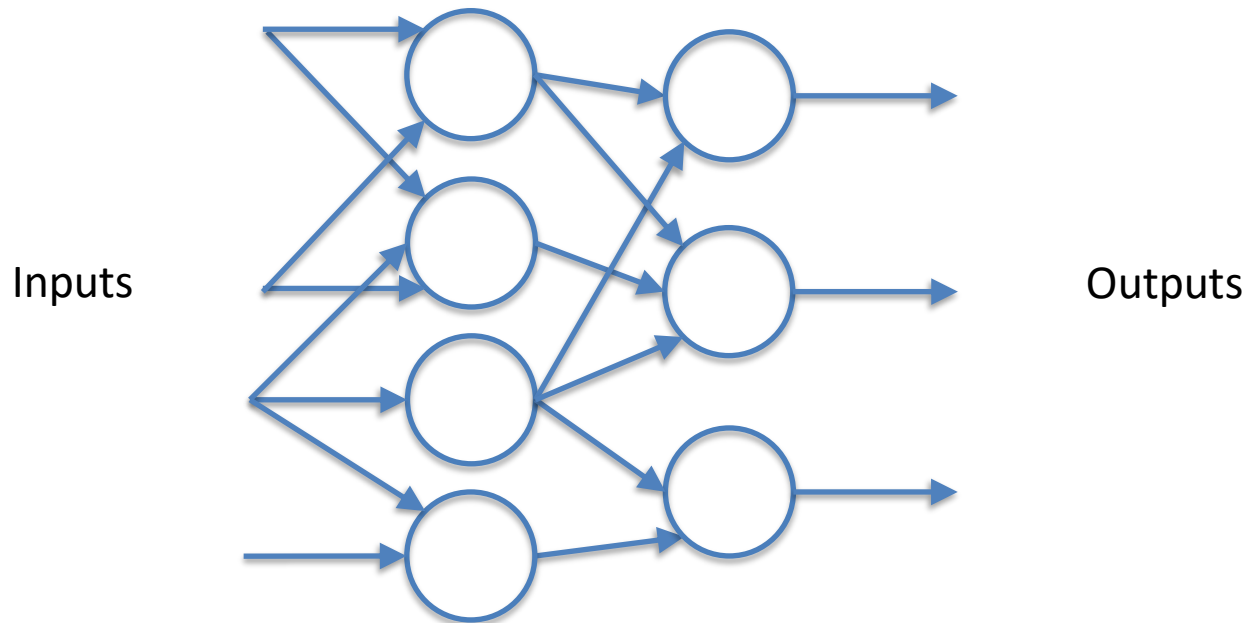


- Weights for incoming edges are chosen as before
- Networks of perceptrons can encode any circuit!

# 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

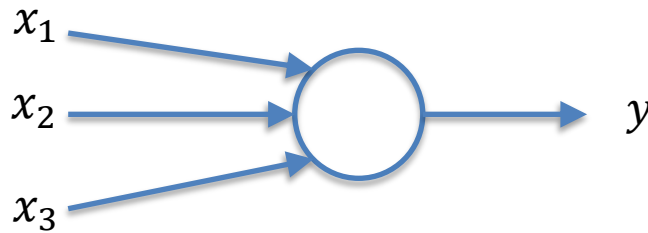


- Given a collection of input-output pairs, we'd like to learn the weights of the neural network so that we can correctly predict the output of an unseen input
  - We could try learning via gradient descent (e.g., by minimizing the Hamming loss)
    - This approach doesn't work so well: small changes 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**

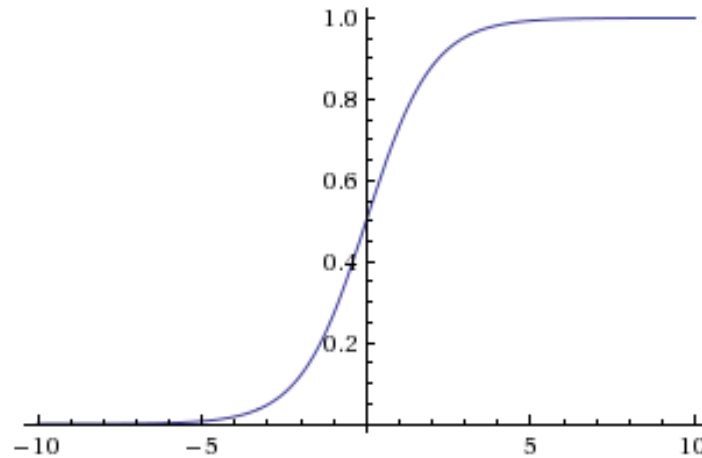


# The Sigmoid Function



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

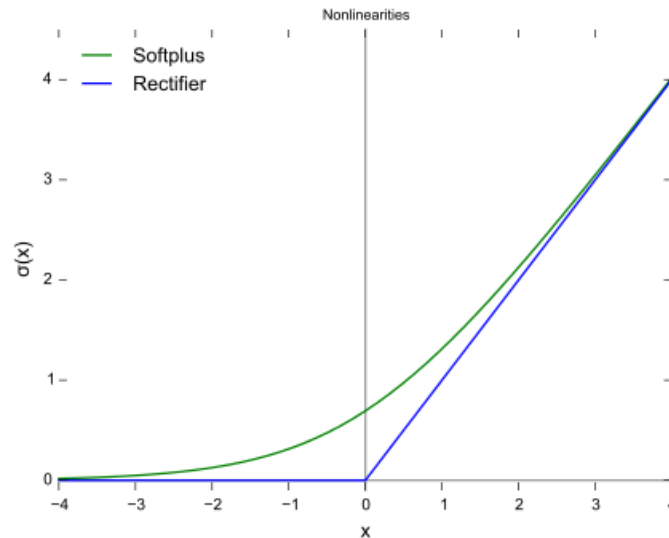
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



# 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  $\ln(1 + e^x)$

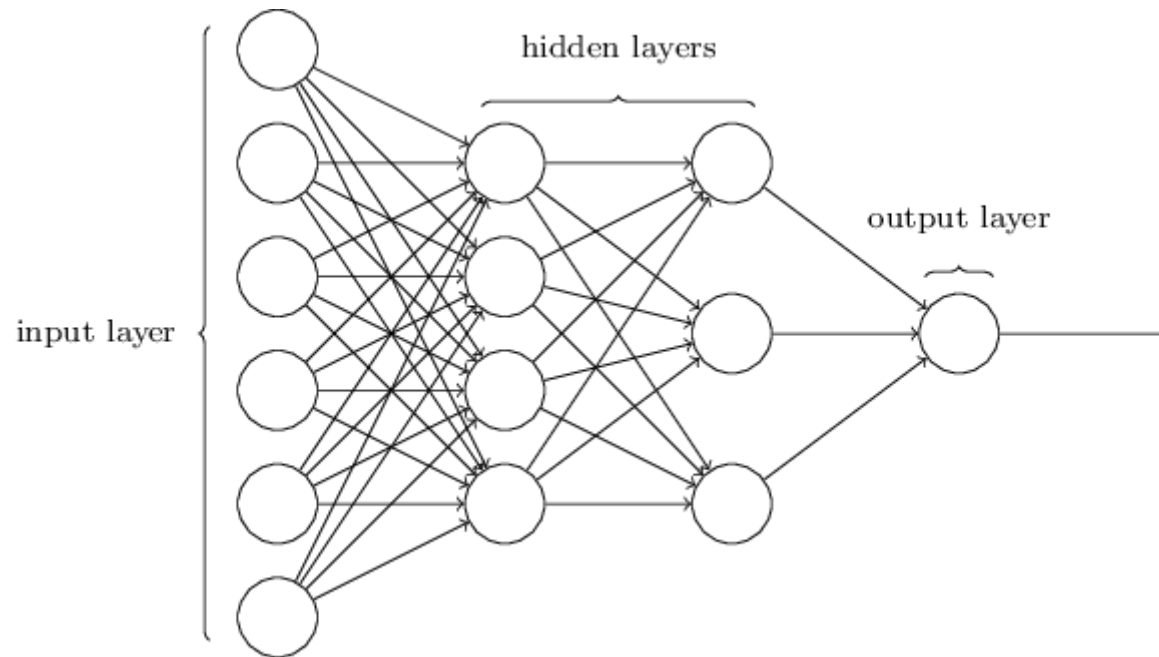


- The softmax function maps a vector of real numbers to a vector of probabilities as

$$\text{softmax}(z)_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

- 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

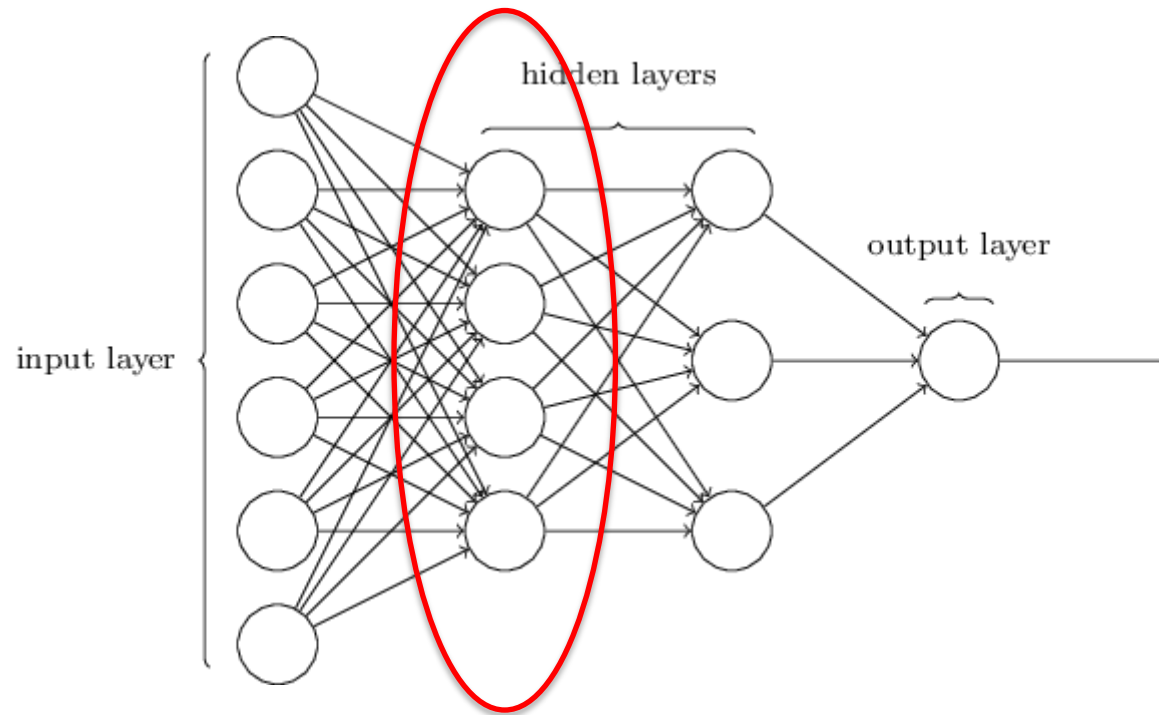


from Neural Networks and Deep Learning by Michael Nielson

# Multilayer Neural Networks

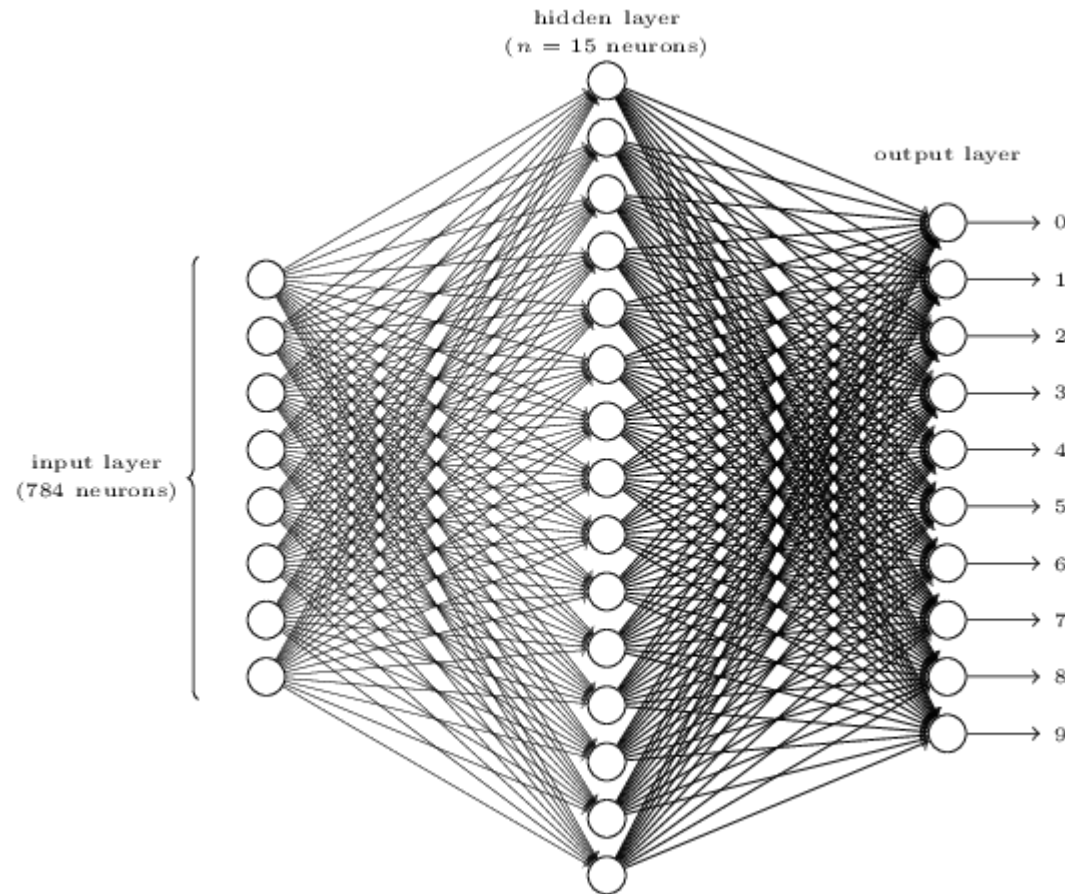


NO intralayer connections



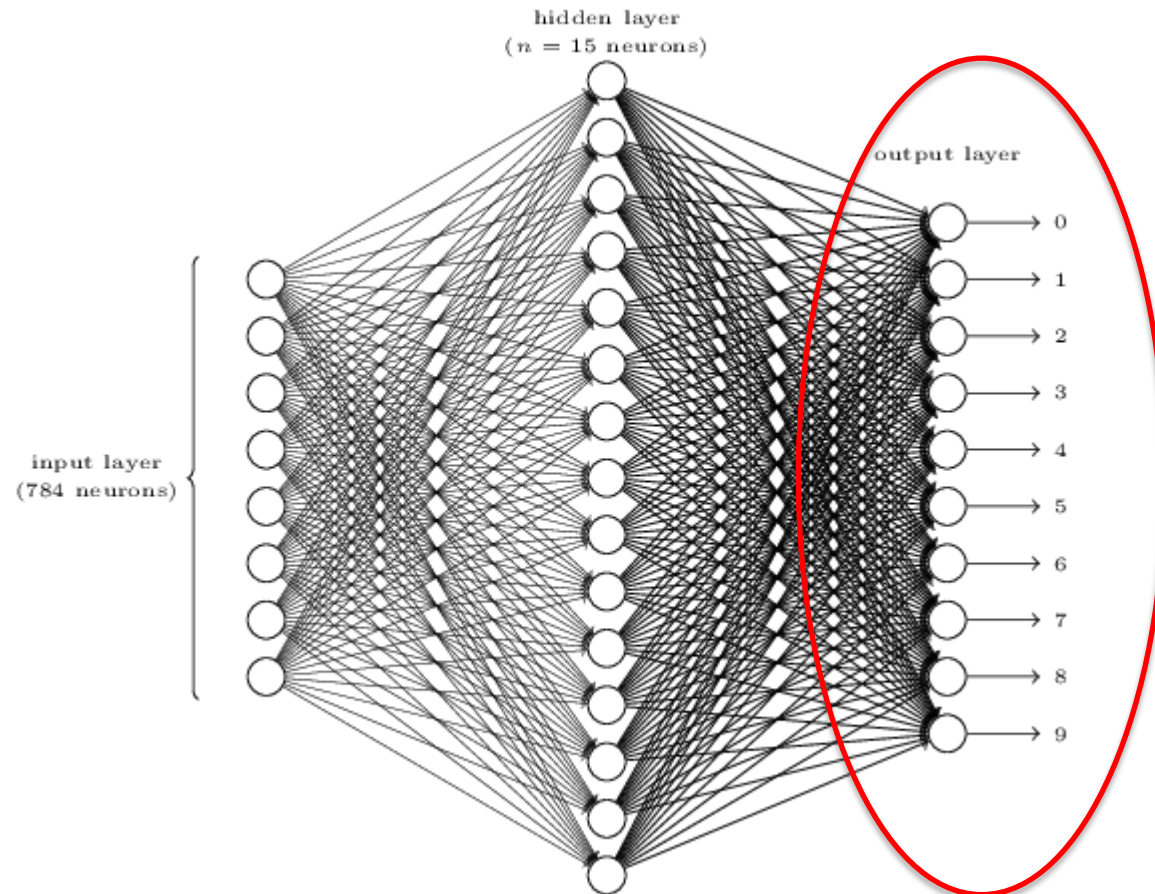
from Neural Networks and Deep Learning by Michael Nielson

# Neural Network for Digit Classification



from Neural Networks and Deep Learning by Michael Nielson

# Neural Network for Digit Classification



Why 10  
instead of 4?

from Neural Networks and Deep Learning by Michael Nielson

- 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



- **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

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

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

- The training data consists of input output pairs  $(x^1, y^1), \dots, (x^M, y^M)$
- $a(x^m, w, b)$  is the output of the neural network for the  $m^{th}$  sample
- $w$  and  $b$  are the weights and biases

- 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

- 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

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

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  $\frac{\partial C}{\partial z_j^l}$

# Computing the Gradient



For the output layer, we have the following partial derivative

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

- 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

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

# Computing the Gradient



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

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

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

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

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



# Computing the Gradient



- Now, for the partial derivatives that we care about

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

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

- 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

$$\delta_j^L = -(y_j - a_j^L) \sigma(z_j^L) (1 - \sigma(z_j^L))$$

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

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

- Perform gradient descent

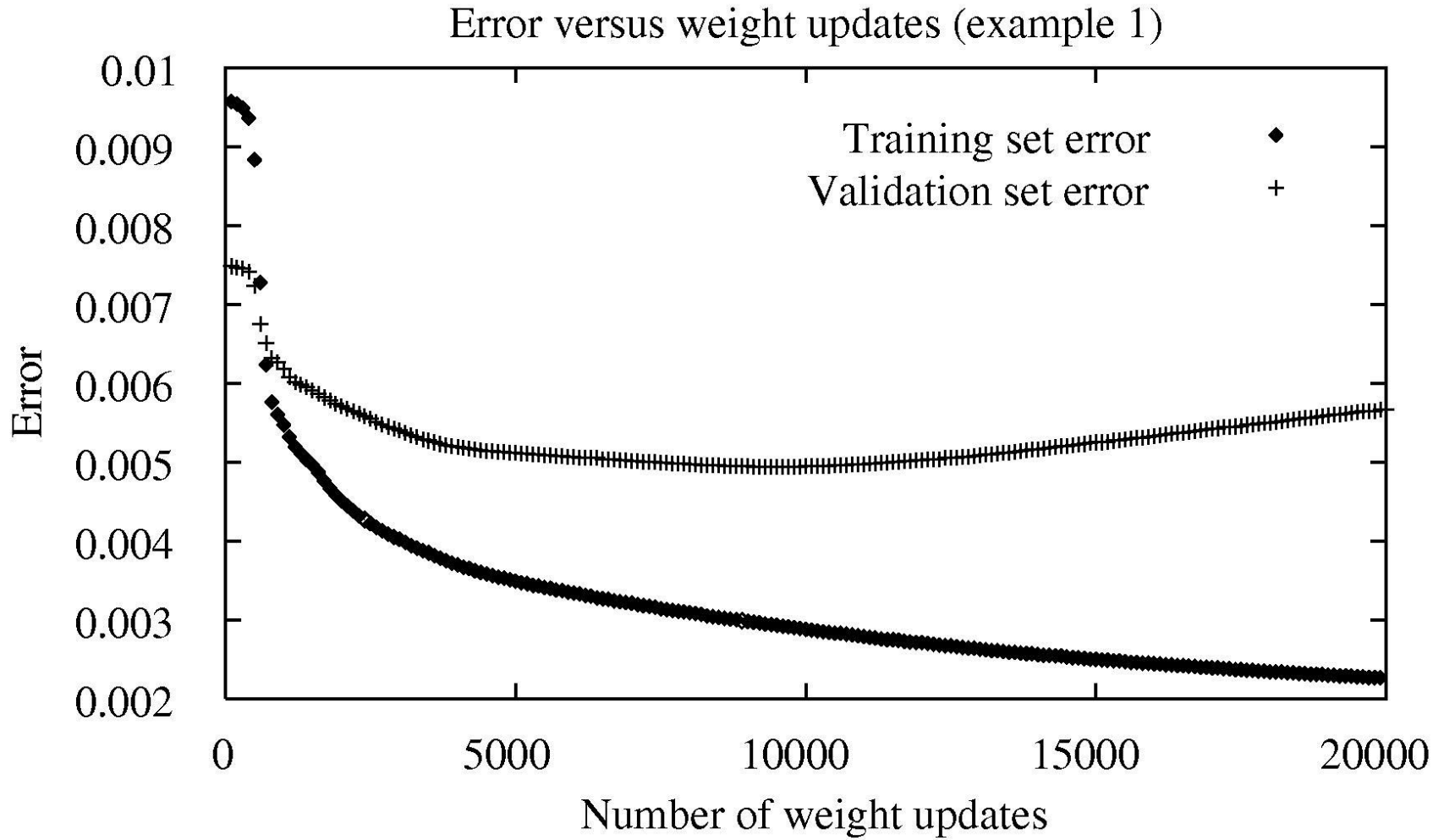
$$b_j^l = b_j^l - \gamma \cdot \delta_j^l$$

$$w_{jk}^l = w_{jk}^l - \gamma \cdot \delta_j^l a_k^{l-1}$$

- 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)
  - Prediction after learning is fast
  - Sometimes include a **momentum** term  $\alpha$  in the gradient update

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

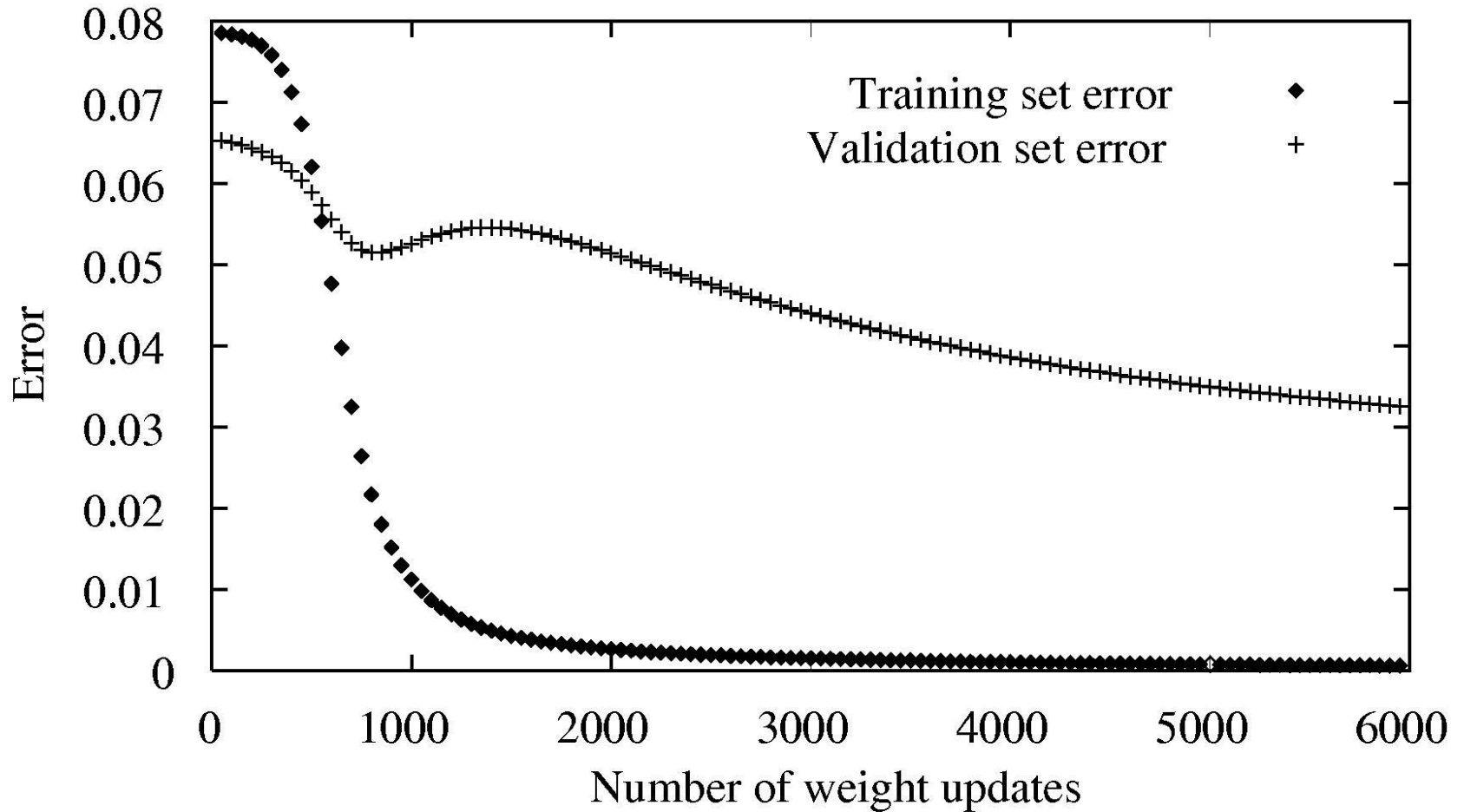
# Overfitting



# Overfitting



Error versus weight updates (example 2)



- 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

- 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

- 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)



- Early stopping
  - Stop the learning early in the hopes that this prevents overfitting
- 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%)

- 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