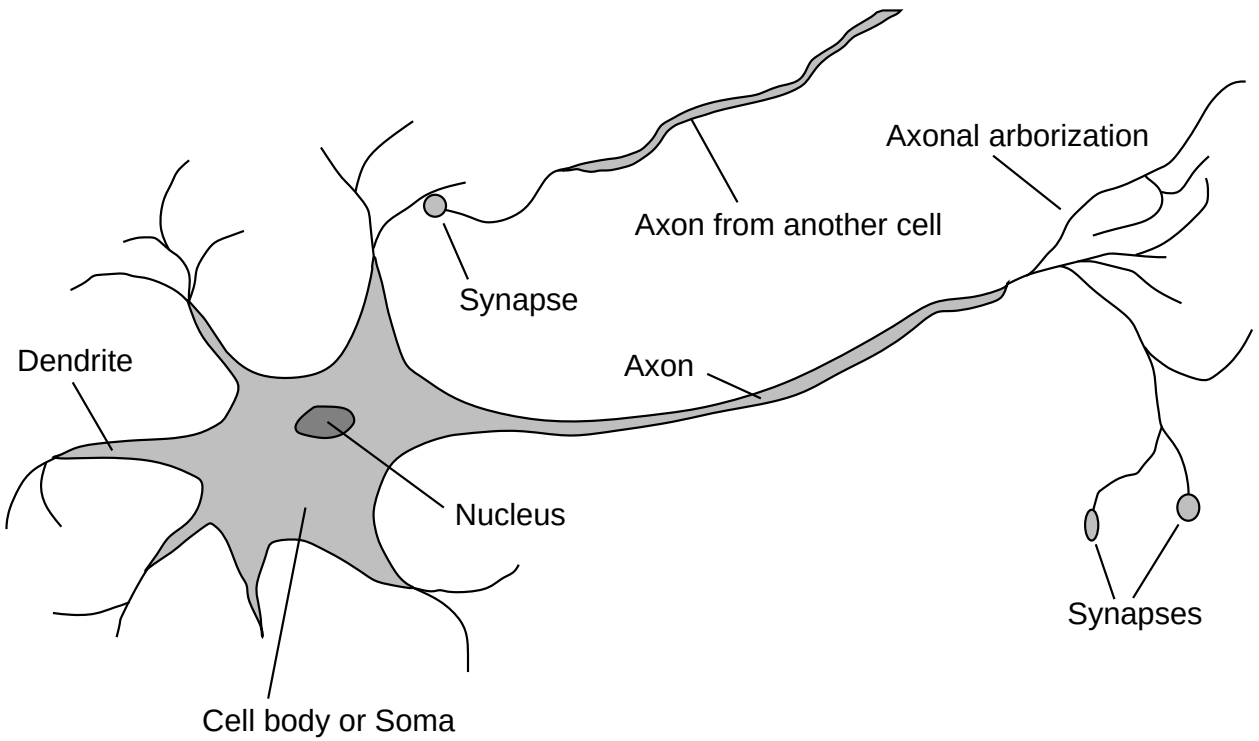
COMS 4771 Lecture 24

1. Neural networks
2. Practical techniques for training neural networks

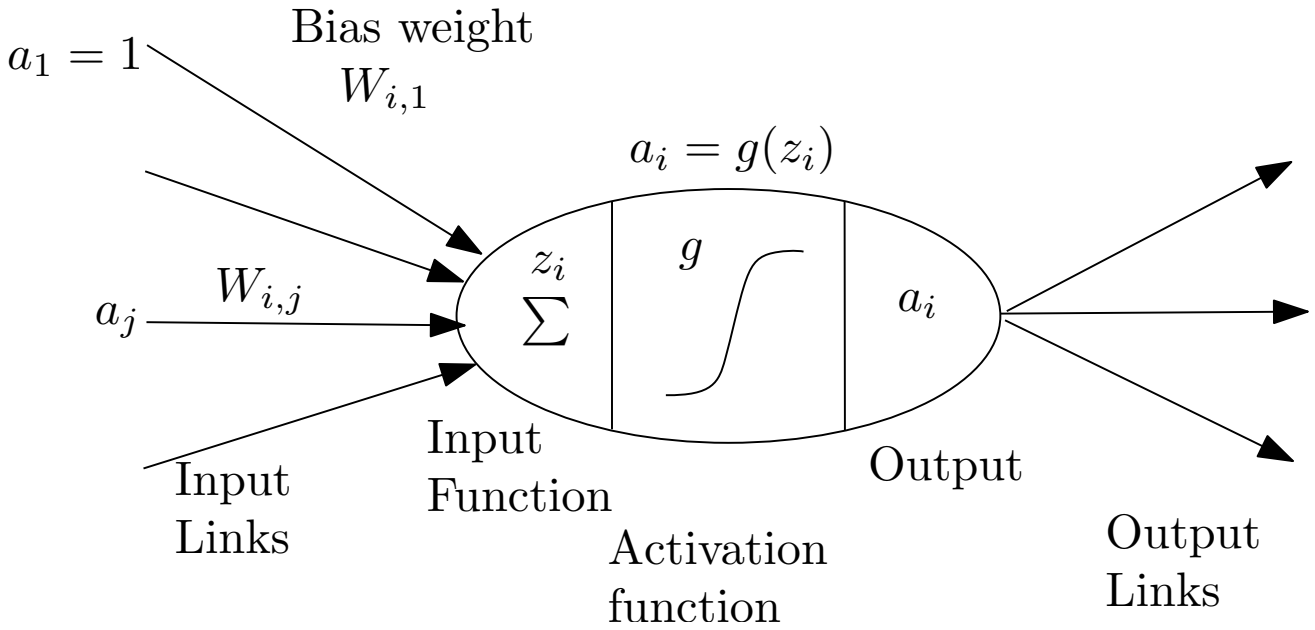
Neural networks

Brains

10^{11} neurons of > 20 types, 10^{14} synapses, 1ms–10ms cycle time
Signals are noisy "spike trains" of electrical potential



McCulloch–Pitts "unit"

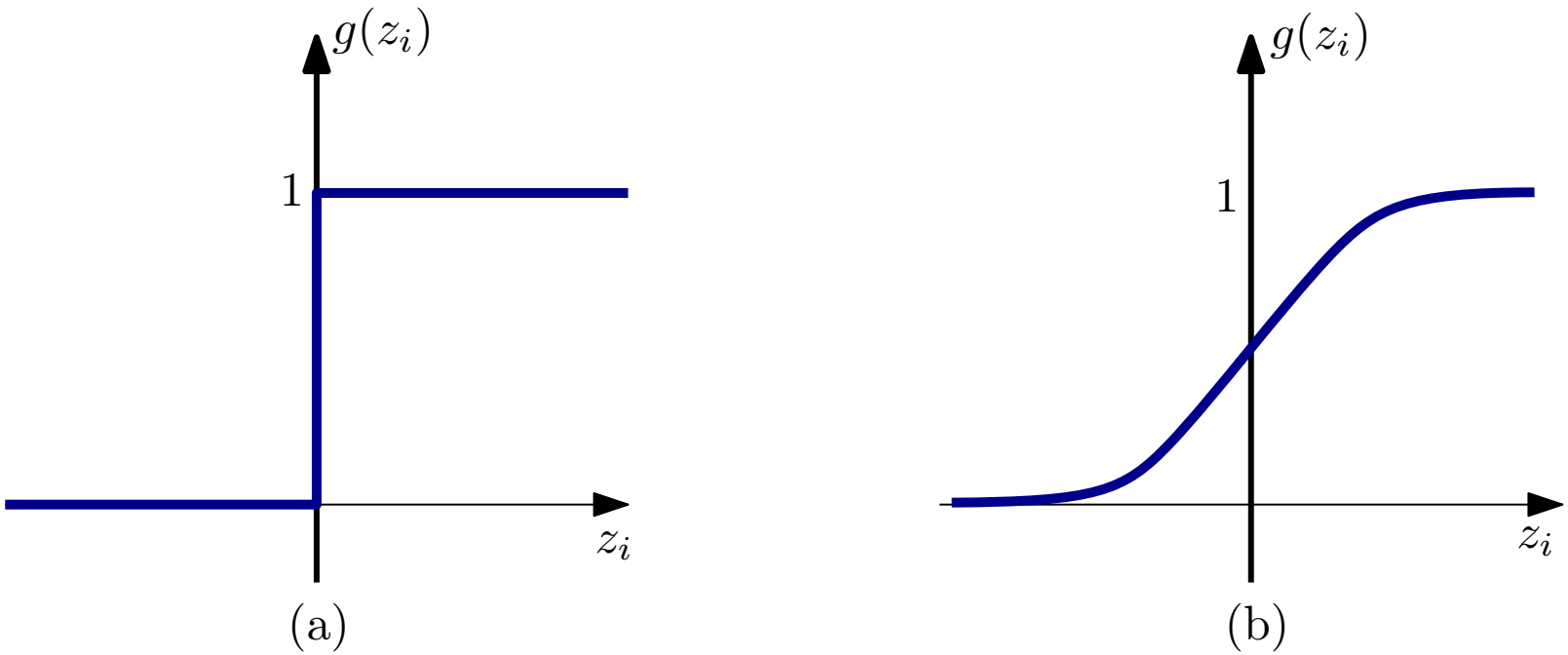


Output is a "squashed" linear function of the inputs:

$$a_i \leftarrow g(z_i) = g\left(\sum_j W_{i,j} a_j\right)$$

A gross oversimplification of real neurons, but its purpose is
to develop understanding of what networks of simple units can do

Activation functions



(a) is a *step function* or *threshold function*
(b) is a *sigmoid* function $1/(1+e^{-x})$
Changing the bias weight $W_{i,1}$ moves the threshold location

Network structures

Feed-forward networks:

- single-layer perceptrons
- multi-layer perceptrons

Feed-forward networks implement functions, have no internal state

Network structures

Feed-forward networks:

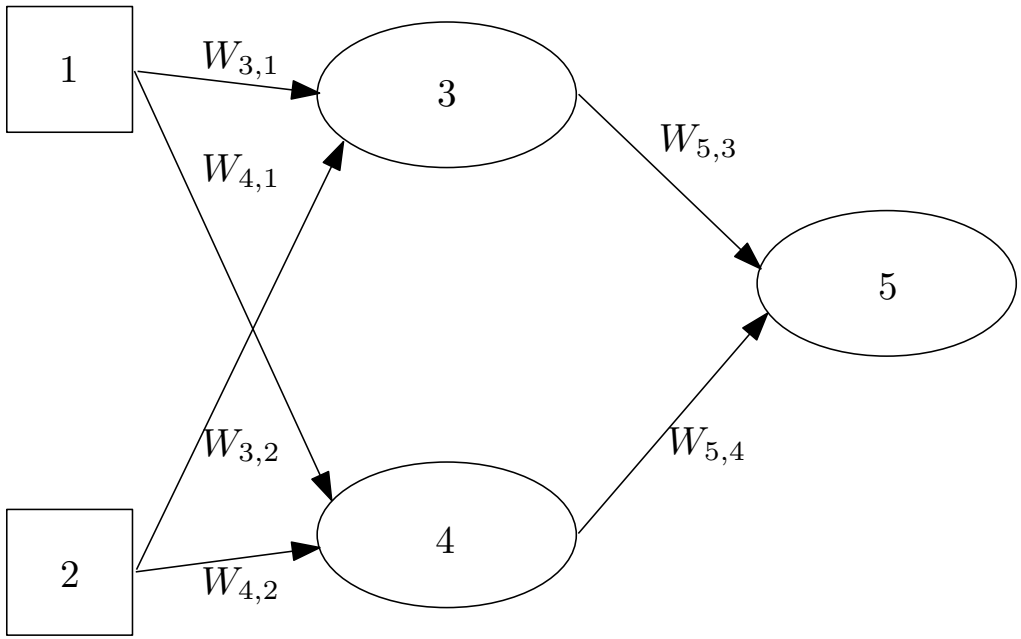
- single-layer perceptrons
- multi-layer perceptrons

Feed-forward networks implement functions, have no internal state

Recurrent networks:

- Hopfield networks have symmetric weights ($W_{i,j} = W_{j,i}$)
  $g(x) = \mathrm{sign}(x)$, $a_i = \pm 1$; *holographic associative memory*
- Boltzmann machines use stochastic activation functions
- Recurrent neural nets have directed cycles with delays
  $\Rightarrow$ have internal state (like flip-flops), can oscillate etc.

Feed-forward example

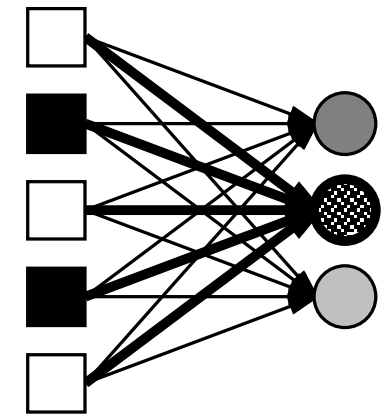


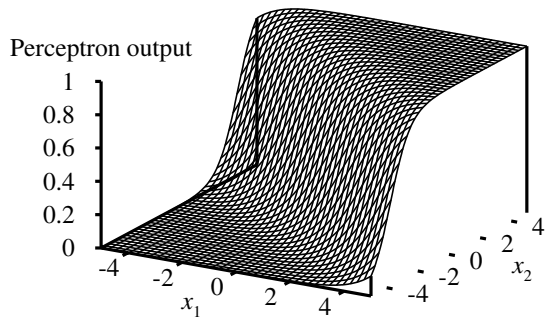
Feed-forward network = a parameterized family of nonlinear functions:

$$\begin{aligned} a_5 &= g(W_{5,3} \cdot a_3 + W_{5,4} \cdot a_4) \\ &= g(W_{5,3} \cdot g(W_{3,1} \cdot a_1 + W_{3,2} \cdot a_2) + W_{5,4} \cdot g(W_{4,1} \cdot a_1 + W_{4,2} \cdot a_2)) \end{aligned}$$

Adjusting weights changes the function: do learning this way!

Single-layer perceptrons





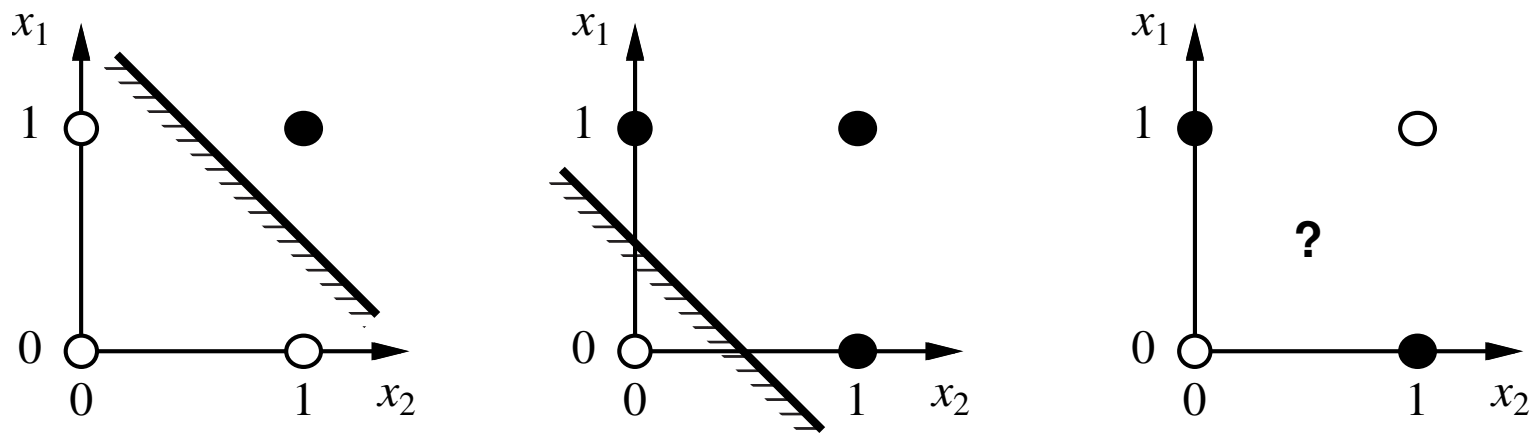
Output units all operate separately—no shared weights

Adjusting weights moves the location, orientation, and steepness of cliff

Expressiveness of perceptrons

Consider a perceptron with $g =$ step function (Rosenblatt, 1957, 1960)
Can represent AND, OR, NOT, majority, etc., but not XOR
Represents a *linear separator* in input space:

$$\sum_j W_j x_j > 0 \quad \text{or} \quad \langle \boldsymbol{W}, \boldsymbol{x} \rangle > 0$$



(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_2

Minsky & Papert (1969) pricked the neural network balloon

Perceptron learning

Given: training set $S = \{(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots, (\boldsymbol{x}^{(n)}, y^{(n)}) \in \mathbb{R}^d \times \mathbb{R}\}$

Suppose we measure error in predicting $\hat{y}$ for $(\boldsymbol{x}, y)$ using a loss function $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ as $\ell(y, \hat{y})$.

Example: square loss, $\ell(y, \hat{y}) = (y - \hat{y})^2$.

Perceptron learning

Given: training set $S = \{(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots, (\boldsymbol{x}^{(n)}, y^{(n)}) \in \mathbb{R}^d \times \mathbb{R}\}$

Suppose we measure error in predicting $\hat{y}$ for $(\boldsymbol{x}, y)$ using a loss function $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ as $\ell(y, \hat{y})$.

Example: square loss, $\ell(y, \hat{y}) = (y - \hat{y})^2$.

Goal: learn the weights of a single layer perceptron with activation function g by minimizing squared loss on the training set:

$$L(\boldsymbol{W}) \;:=\; \frac{1}{n} \sum_{i=1}^{n} \ell(y^{(i)}, g(\langle \boldsymbol{W}, \boldsymbol{x}^{(i)} \rangle)).$$

Perceptron learning

Given: training set $S = \{(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots, (\boldsymbol{x}^{(n)}, y^{(n)}) \in \mathbb{R}^d \times \mathbb{R}\}$

Suppose we measure error in predicting $\hat{y}$ for $(\boldsymbol{x}, y)$ using a loss function $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ as $\ell(y, \hat{y})$.

Example: square loss, $\ell(y, \hat{y}) = (y - \hat{y})^2$.

Goal: learn the weights of a single layer perceptron with activation function g by minimizing squared loss on the training set:

$$L(\boldsymbol{W}) \;:=\; \frac{1}{n} \sum_{i=1}^{n} \ell(y^{(i)}, g(\langle \boldsymbol{W}, \boldsymbol{x}^{(i)} \rangle)).$$

This is usually a *non-convex* problem, even if ℓ is convex.

Perceptron learning

Given: training set $S = \{(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots, (\boldsymbol{x}^{(n)}, y^{(n)}) \in \mathbb{R}^d \times \mathbb{R}\}$

Suppose we measure error in predicting $\hat{y}$ for $(\boldsymbol{x}, y)$ using a loss function $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ as $\ell(y, \hat{y})$.

Example: square loss, $\ell(y, \hat{y}) = (y - \hat{y})^2$.

Goal: learn the weights of a single layer perceptron with activation function g by minimizing squared loss on the training set:

$$L(\boldsymbol{W}) \;:=\; \frac{1}{n} \sum_{i=1}^{n} \ell(y^{(i)}, g(\langle \boldsymbol{W}, \boldsymbol{x}^{(i)} \rangle)).$$

This is usually a *non-convex* problem, even if ℓ is convex.

Gradient descent is used to find weights corresponding to a local minimum.

Assumption: ℓ and g are differentiable. Can also work with subgradients of ℓ.

Gradient Descent

To compute gradient, split loss function over training examples:

$$L(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} L_i(\boldsymbol{W}), \text{ where } L_i(\boldsymbol{W}) = \ell(y^{(i)}, g(\langle \boldsymbol{W}, \boldsymbol{x}^{(i)} \rangle)).$$

Compute the gradients for each $L_i(\boldsymbol{W})$ and add them up.

Gradient Descent

To compute gradient, split loss function over training examples:

$$L(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} L_i(\boldsymbol{W}), \text{ where } L_i(\boldsymbol{W}) = \ell(y^{(i)}, g(\langle \boldsymbol{W}, \boldsymbol{x}^{(i)} \rangle)).$$

Compute the gradients for each $L_i(\boldsymbol{W})$ and add them up.

Using chain rule for derivatives, we have

$$\begin{aligned} \frac{\partial L_i}{\partial W_j} &= \ell'(y^{(i)}, g(\langle \boldsymbol{W}, \boldsymbol{x}^{(i)} \rangle)) \cdot g'(\langle \boldsymbol{W}, \boldsymbol{x}^{(i)} \rangle) \cdot x_j^{(i)} \\ &= \ell'(y^{(i)}, \hat{y}^{(i)}) \cdot g'(z_i) \cdot x_j^{(i)} \end{aligned}$$

Here, $z_i = \langle \boldsymbol{W}, \boldsymbol{x}^{(i)} \rangle$ is the input to the output node, and $\hat{y}^{(i)} = g(z_i)$ is its output (also the prediction).

Gradient Descent

To compute gradient, split loss function over training examples:

$$L(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} L_i(\boldsymbol{W}), \text{ where } L_i(\boldsymbol{W}) = \ell(y^{(i)}, g(\langle \boldsymbol{W}, \boldsymbol{x}^{(i)} \rangle)).$$

Compute the gradients for each $L_i(\boldsymbol{W})$ and add them up.

Using chain rule for derivatives, we have

$$\begin{aligned} \frac{\partial L_i}{\partial W_j} &= \ell'(y^{(i)}, g(\langle \boldsymbol{W}, \boldsymbol{x}^{(i)} \rangle)) \cdot g'(\langle \boldsymbol{W}, \boldsymbol{x}^{(i)} \rangle) \cdot x_j^{(i)} \\ &= \ell'(y^{(i)}, \hat{y}^{(i)}) \cdot g'(z_i) \cdot x_j^{(i)} \end{aligned}$$

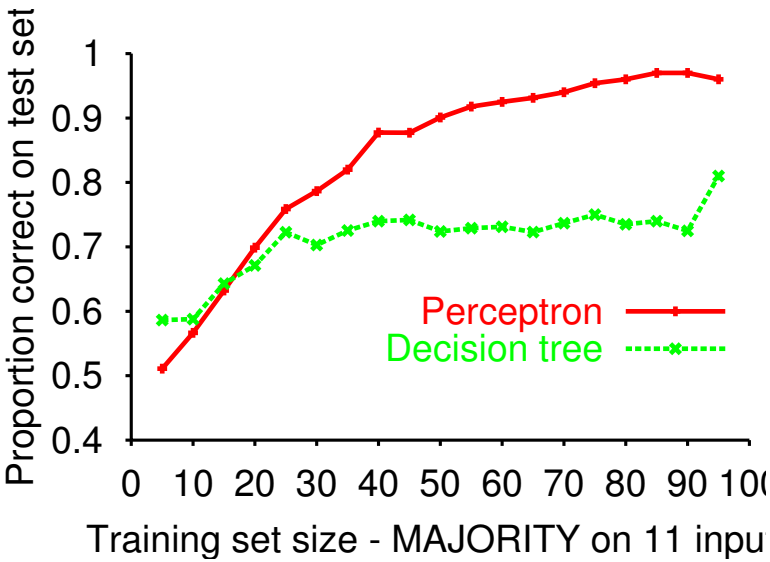
Here, $z_i = \langle \boldsymbol{W}, \boldsymbol{x}^{(i)} \rangle$ is the input to the output node, and $\hat{y}^{(i)} = g(z_i)$ is its output (also the prediction).

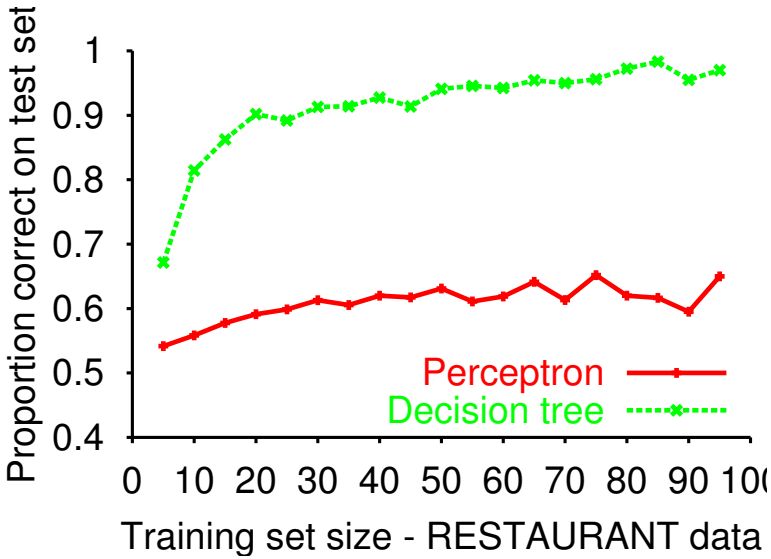
Gradient descent weight update rule:

$$W_j \leftarrow W_j - \eta_t \cdot \frac{1}{n}\sum_{i=1}^{n} \ell'(y^{(i)}, \hat{y}^{(i)}) \cdot g'(z_i) \cdot x_j^{(i)}$$

Perceptron learning contd.

Perceptron learning rule converges to a consistent function
for any linearly separable data set



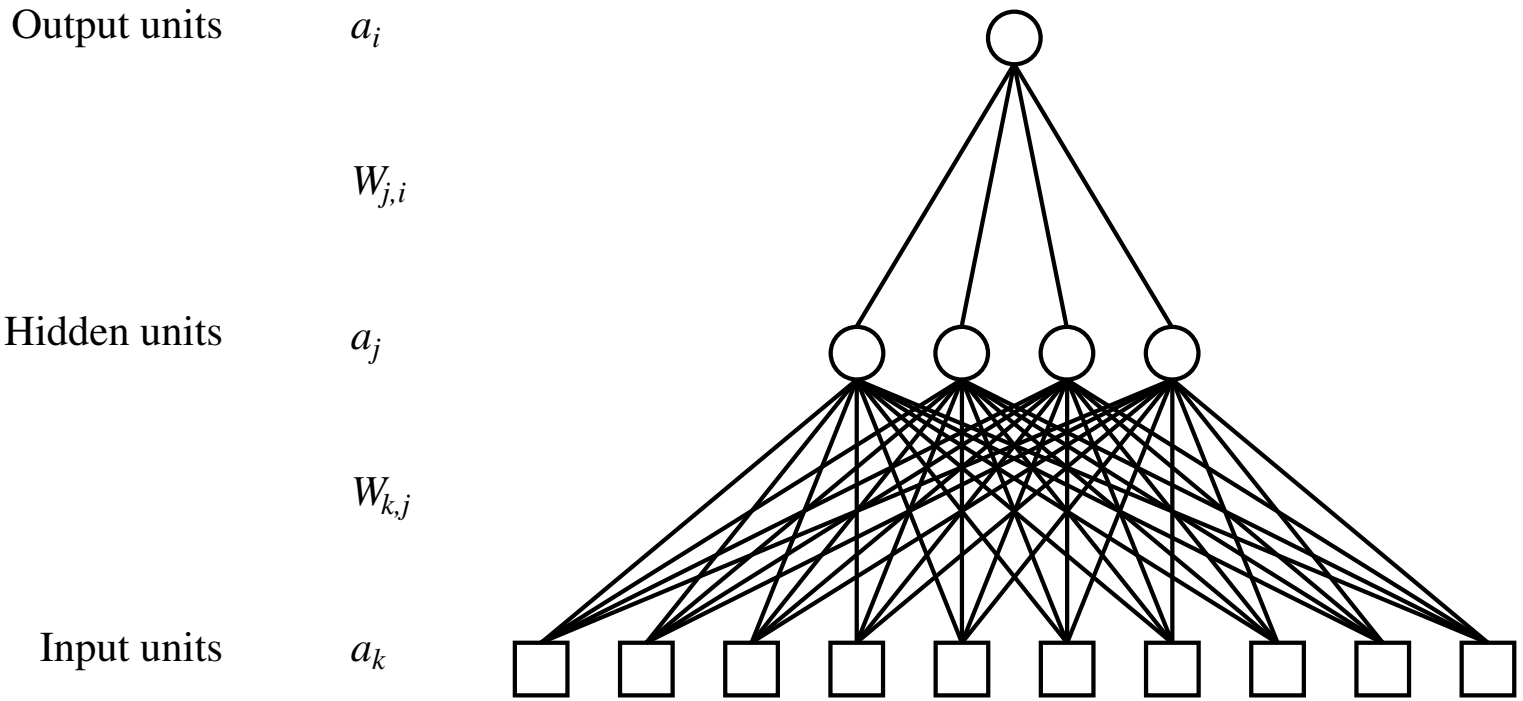


Perceptron learns majority function easily, DTL is hopeless

DTL learns restaurant function easily, perceptron cannot represent it

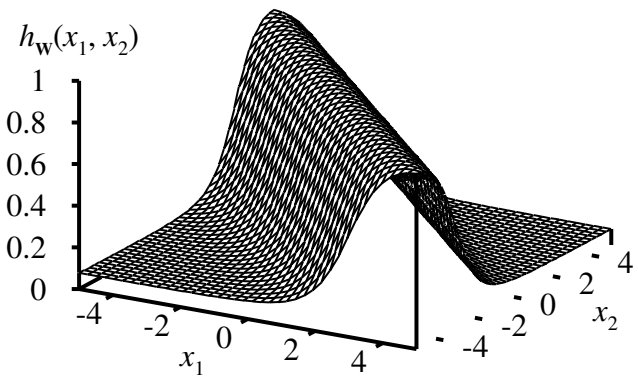
Multilayer perceptrons

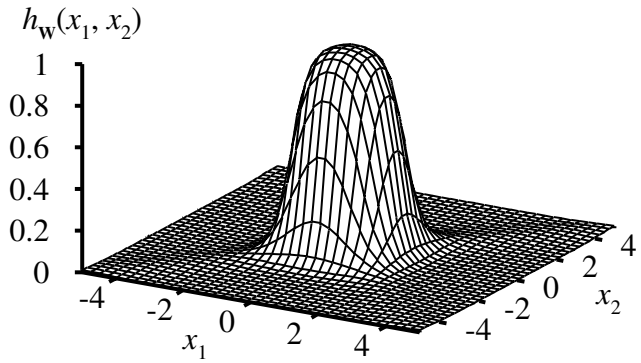
Layers are usually fully connected;
numbers of *hidden units* typically chosen by hand



Expressiveness of MLPs

All continuous functions with 2 layers, all functions with 3 layers

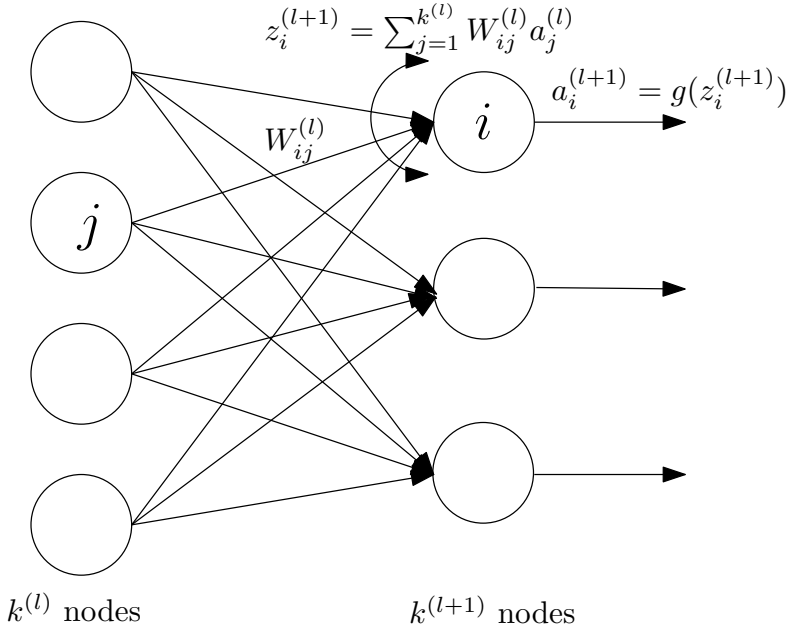




- Combine two opposite-facing threshold functions to make a ridge
- Combine two perpendicular ridges to make a bump
- Add bumps of various sizes and locations to fit any surface
- Proof requires exponentially many hidden units

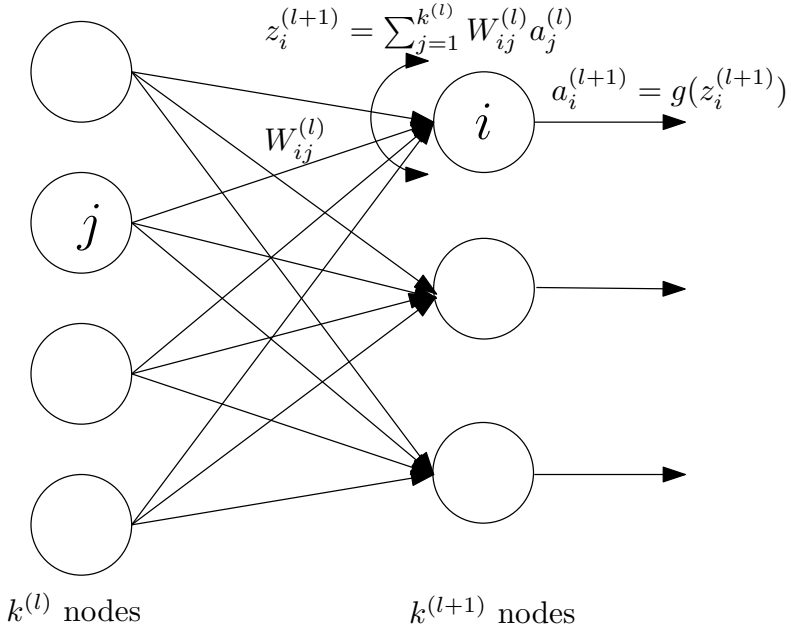
Some mathematical notation

- Consider a H level neural network.
- Index the layers by $l = 0, 1, 2, \ldots, H$.
- 0 indicates the layer of the input nodes.
- Let $k^{(l)}$ denote the number of nodes in layer l.



Some mathematical notation

- Consider a H level neural network.
- Index the layers by $l = 0, 1, 2, \ldots, H$.
- 0 indicates the layer of the input nodes.
- Let $k^{(l)}$ denote the number of nodes in layer l.

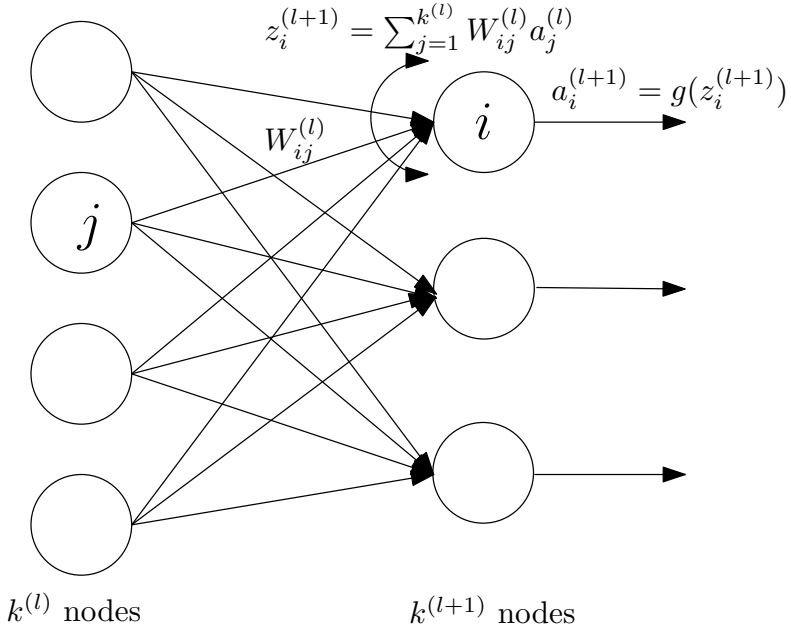


Above figure shows the connections between layers l and $l+1$. In this figure,

- j indexes $k^{(l)}$ nodes in layer l, and i indexes the $k^{(l+1)}$ nodes in layer $l+1$.

Some mathematical notation

- Consider a H level neural network.
- Index the layers by $l = 0, 1, 2, \ldots, H$.
- 0 indicates the layer of the input nodes.
- Let $k^{(l)}$ denote the number of nodes in layer l.

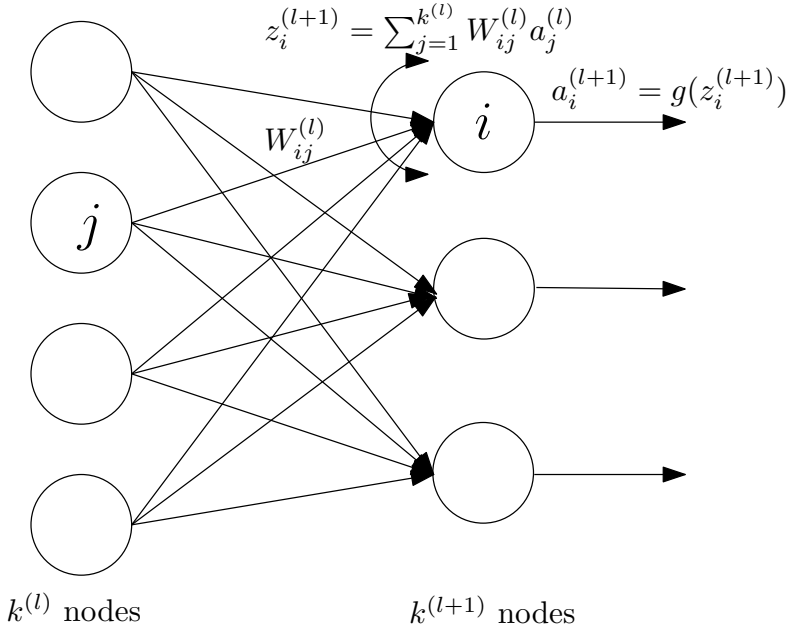


Above figure shows the connections between layers l and $l+1$. In this figure,

- j indexes $k^{(l)}$ nodes in layer l, and i indexes the $k^{(l+1)}$ nodes in layer $l+1$.
- $a_j^{(l)}$ is the output of the node j in layer l, and $\boldsymbol{a}^{(l)}$ is the $k^{(l)}$ vector of these values.

Some mathematical notation

- Consider a H level neural network.
- Index the layers by $l = 0, 1, 2, \ldots, H$.
- 0 indicates the layer of the input nodes.
- Let $k^{(l)}$ denote the number of nodes in layer l.

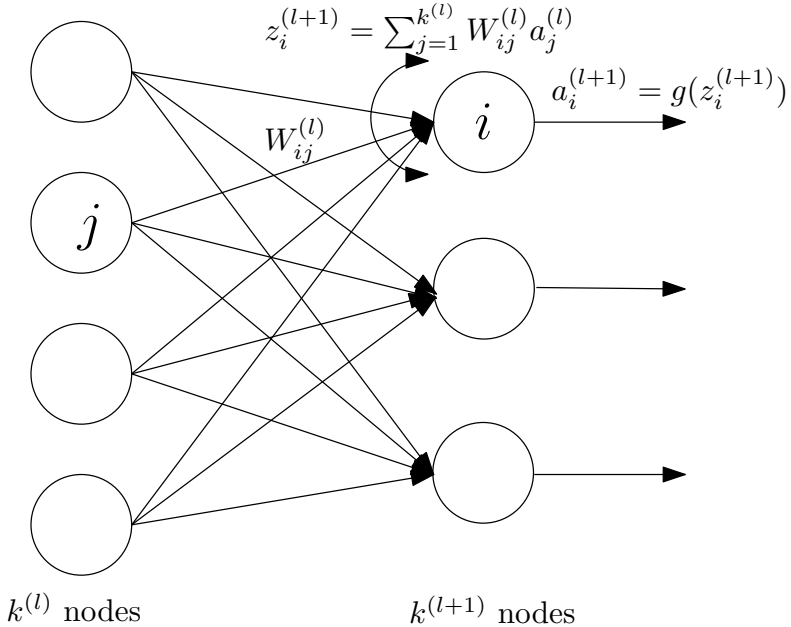


Above figure shows the connections between layers l and $l+1$. In this figure,

- j indexes $k^{(l)}$ nodes in layer l, and i indexes the $k^{(l+1)}$ nodes in layer $l+1$.
- $a_j^{(l)}$ is the output of the node j in layer l, and $\boldsymbol{a}^{(l)}$ is the $k^{(l)}$ vector of these values.
- $W_{ij}^{(l)}$ is the weight of the connection between node j in layer l and node i in layer $l+1$, and $\boldsymbol{W}^{(l)}$ is the $k^{(l+1)} \times k^{(l)}$ matrix of these weights.

Some mathematical notation

- Consider a H level neural network.
- Index the layers by $l = 0, 1, 2, \ldots, H$.
- 0 indicates the layer of the input nodes.
- Let $k^{(l)}$ denote the number of nodes in layer l.



Above figure shows the connections between layers l and $l+1$. In this figure,

- j indexes $k^{(l)}$ nodes in layer l, and i indexes the $k^{(l+1)}$ nodes in layer $l+1$.
- $a_j^{(l)}$ is the output of the node j in layer l, and $\boldsymbol{a}^{(l)}$ is the $k^{(l)}$ vector of these values.
- $W_{ij}^{(l)}$ is the weight of the connection between node j in layer l and node i in layer $l+1$, and $\boldsymbol{W}^{(l)}$ is the $k^{(l+1)} \times k^{(l)}$ matrix of these weights.
- $z_i^{(l+1)} = \sum_{j=1}^{k^{(l)}} W_{ij}^{(l)} a_i^{(l)}$ is the input to node i of layer $l+1$, and $\boldsymbol{z}^{(l+1)}$ is the $k^{(l+1)}$ vector of these values.

Prediction: Forward Propagation

With all that notation, we can describe the operation of a neural network succinctly as follows.

1: Input: data point $\boldsymbol{x}$.
2: Set $\boldsymbol{a}^{(0)} = \boldsymbol{x}$.
3: **for** $l = 0, 1, \ldots, H-1$ **do**
4: Compute $\boldsymbol{z}^{(l+1)} = \boldsymbol{W}^{(l)} \boldsymbol{a}^{(l)}$.
5: Compute $a_i^{(l+1)} = g(z_i^{(l+1)})$ for $i = 1, 2, \ldots, k^{(l+1)}$.
6: **end for**
7: Output $\hat{y} = \boldsymbol{a}^{(H)}$.

This is called the "Forward Propagation" algorithm.

Multilayer neural network learning

As in the case of a single layer perceptron, weights updated using gradient descent to minimize training error.

Directly computing the partial derivatives w.r.t. weights $W_{ij}^{(l)}$ becomes complicated.

Multilayer neural network learning

As in the case of a single layer perceptron, weights updated using gradient descent to minimize training error.

Directly computing the partial derivatives w.r.t. weights $W_{ij}^{(l)}$ becomes complicated.

Instead, a **Backward Propagation** algorithm (*backprop* for short) computes the partial derivatives.

Again, gradients are computed example by example, and averaged in the end.

Backward Propagation algorithm

Let $\tilde{W}_{ij}^{(l)}$, $\tilde{a}_i^{(l)}$, $\tilde{z}_j^{(l)}$ denote partial derivatives of the loss w.r.t. $W_{ij}^{(l)}$, $a_i^{(l)}$, $z_j^{(l)}$

Let $\tilde{\boldsymbol{W}}^{(l)}$, $\tilde{\boldsymbol{a}}^{(l)}$ and $\tilde{\boldsymbol{z}}^{(l)}$ be corresponding matrix and vectors

Backward Propagation algorithm

Let $\tilde{W}_{ij}^{(l)}$, $\tilde{a}_i^{(l)}$, $\tilde{z}_j^{(l)}$ denote partial derivatives of the loss w.r.t. $W_{ij}^{(l)}$, $a_i^{(l)}$, $z_j^{(l)}$

Let $\tilde{\boldsymbol{W}}^{(l)}$, $\tilde{\boldsymbol{a}}^{(l)}$ and $\tilde{\boldsymbol{z}}^{(l)}$ be corresponding matrix and vectors

1: Input: training example $(\boldsymbol{x}, y)$
2: Set $\boldsymbol{a}^{(0)} = \boldsymbol{x}$, and run the Forward Propagation algorithm to compute $\boldsymbol{a}^{(l)}$ and $\boldsymbol{z}^{(l)}$ for $l = 1, 2, \ldots, H$.
3: Compute $\tilde{\boldsymbol{a}}^{(H)} = \ell'(y, \boldsymbol{a}^{(H)})$.
4: **for** $l = H - 1$ down to 0 **do**
5: Compute $\tilde{z}_i^{(l+1)} = g'(z_i^{l+1}) \cdot \tilde{a}_i^{(l+1)}$ for $i = 1, 2, \ldots, k^{(l+1)}$.
6: Compute $\tilde{\boldsymbol{W}}^{(l)} = \tilde{\boldsymbol{z}}^{(l+1)} \boldsymbol{a}^{(l)\top}$.
7: Compute $\tilde{\boldsymbol{a}}^{(l)} = \boldsymbol{W}^{(l)\top} \tilde{\boldsymbol{z}}^{(l+1)}$.
8: **end for**
9: Output $\tilde{\boldsymbol{W}}^{(l)}$ for $l = 0, 1, \ldots, H - 1$.

Deriving backprop updates

- $z_i^{(l+1)}$: treat loss as a function of $\boldsymbol{a}^{(l+1)}$ and apply chain rule:

$$\tilde{z}_i^{(l+1)} = \left\langle \tilde{\boldsymbol{a}}^{(l+1)}, \left\langle \frac{\partial a_1^{(l+1)}}{\partial z_i^{(l+1)}}, \frac{\partial a_2^{(l+1)}}{\partial z_i^{(l+1)}}, \cdots, \frac{\partial a_{k(l+1)}^{(l+1)}}{\partial z_i^{(l+1)}} \right\rangle \right\rangle = \tilde{a}_i^{(l+1)} \cdot g'(z_i^{(l+1)}).$$

Deriving backprop updates

- $z_i^{(l+1)}$: treat loss as a function of $\boldsymbol{a}^{(l+1)}$ and apply chain rule:

$$\tilde{z}_i^{(l+1)} = \left\langle \tilde{\boldsymbol{a}}^{(l+1)}, \left\langle \frac{\partial a_1^{(l+1)}}{\partial z_i^{(l+1)}}, \frac{\partial a_2^{(l+1)}}{\partial z_i^{(l+1)}}, \cdots, \frac{\partial a_{k^{(l+1)}}^{(l+1)}}{\partial z_i^{(l+1)}} \right\rangle \right\rangle = \tilde{a}_i^{(l+1)} \cdot g'(z_i^{(l+1)}).$$

- $W_{ij}^{(l)}$: treat loss as a function of $\boldsymbol{z}^{(l+1)}$ and apply chain rule:

$$\tilde{W}_{ij}^{(l)} = \left\langle \tilde{\boldsymbol{z}}^{(l+1)}, \left\langle \frac{\partial z_1^{(l+1)}}{\partial W_{ij}^{(l)}}, \frac{\partial z_2^{(l+1)}}{\partial W_{ij}^{(l)}}, \cdots, \frac{\partial z_{k^{(l+1)}}^{(l+1)}}{\partial W_{ij}^{(l)}} \right\rangle \right\rangle = \tilde{z}_i^{(l+1)} \cdot a_j^{(l)}.$$

Deriving backprop updates

- $z_i^{(l+1)}$: treat loss as a function of $\boldsymbol{a}^{(l+1)}$ and apply chain rule:

$$\tilde{z}_i^{(l+1)} = \left\langle \tilde{\boldsymbol{a}}^{(l+1)}, \left\langle \frac{\partial a_1^{(l+1)}}{\partial z_i^{(l+1)}}, \frac{\partial a_2^{(l+1)}}{\partial z_i^{(l+1)}}, \cdots, \frac{\partial a_{k^{(l+1)}}^{(l+1)}}{\partial z_i^{(l+1)}} \right\rangle \right\rangle = \tilde{a}_i^{(l+1)} \cdot g'(z_i^{(l+1)}).$$

- $W_{ij}^{(l)}$: treat loss as a function of $\boldsymbol{z}^{(l+1)}$ and apply chain rule:

$$\tilde{W}_{ij}^{(l)} = \left\langle \tilde{\boldsymbol{z}}^{(l+1)}, \left\langle \frac{\partial z_1^{(l+1)}}{\partial W_{ij}^{(l)}}, \frac{\partial z_2^{(l+1)}}{\partial W_{ij}^{(l)}}, \cdots, \frac{\partial z_{k^{(l+1)}}^{(l+1)}}{\partial W_{ij}^{(l)}} \right\rangle \right\rangle = \tilde{z}_i^{(l+1)} \cdot a_j^{(l)}.$$

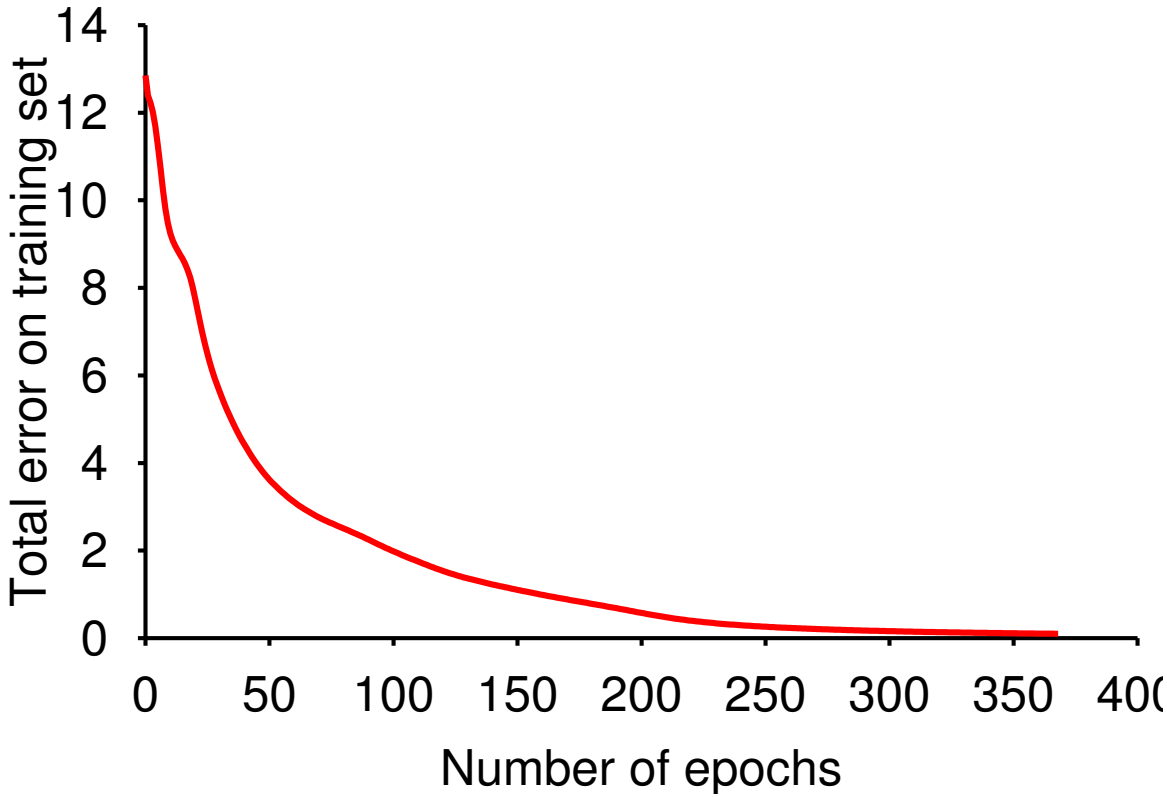
- $a_j^{(l)}$: treat loss as a function of $\boldsymbol{z}^{(l+1)}$ and apply chain rule:

$$\tilde{a}_j^{(l)} = \left\langle \tilde{\boldsymbol{z}}^{(l+1)}, \left\langle \frac{\partial z_1^{(l+1)}}{\partial a_j^{(l)}}, \frac{\partial z_2^{(l+1)}}{\partial a_j^{(l)}}, \cdots, \frac{\partial z_{k^{(l+1)}}^{(l+1)}}{\partial a_j^{(l)}} \right\rangle \right\rangle = \langle \tilde{\boldsymbol{z}}^{(l+1)}, \boldsymbol{W}_{\star j}^{(l)} \rangle,$$

  where $\boldsymbol{W}_{\star j}^{(l)}$ is the j-th column of $\boldsymbol{W}^{(l)}$.

Backprop learning experiments

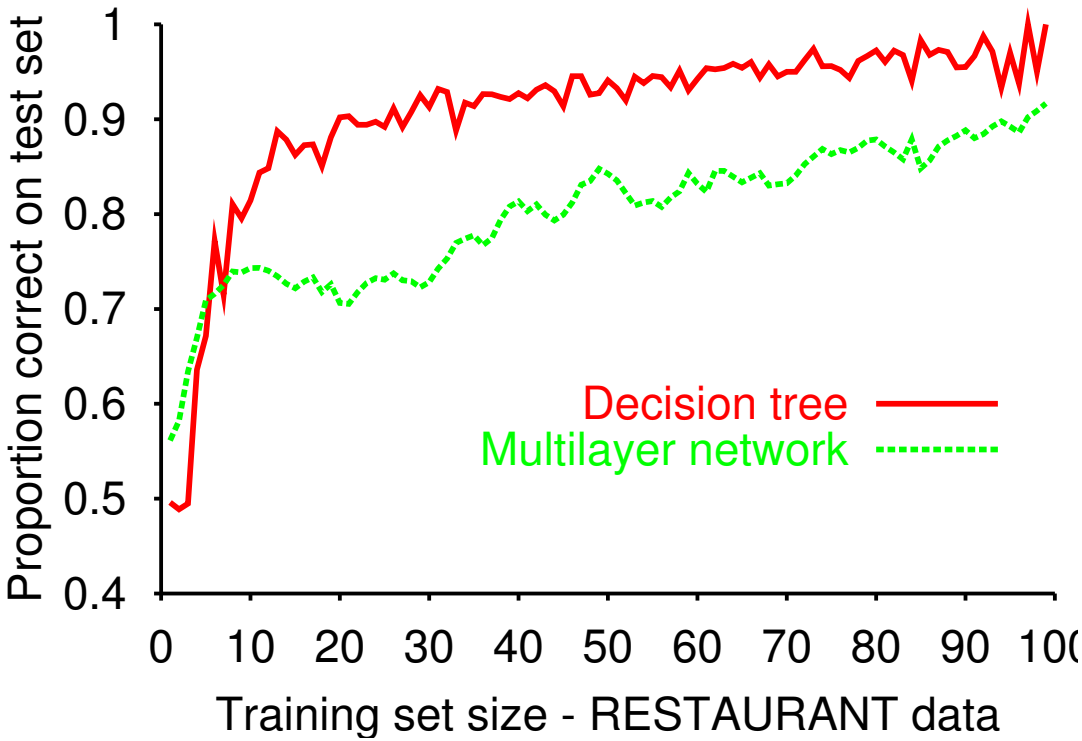
Training curve for 100 restaurant examples: finds exact fit



Typical problems: slow convergence, local minima

Backprop learning experiments

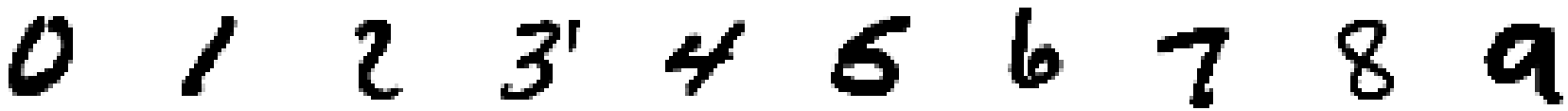
Learning curve for MLP with 4 hidden units:



MLPs are quite good for complex pattern recognition tasks, but resulting hypotheses cannot be understood easily

MNIST handwritten digit recognition

Handwritten digits data: grayscale 28×28 images, treated as vectors in $\mathbb{R}^{784}$, with labels indicating the digit they represent.



3-nearest-neighbor $=$ 2.4% error

400–300–10 unit MLP $=$ 1.6% error

Current best ("Multi-column Deep Convolutional Neural Networks") $\approx$ 0.2% error

Practical techniques for training neural networks

Online Training

- Implementing each step of gradient descent using backprop involves one pass over the entire training set
- Instead, can also train example-by-example exactly like Online Perceptron Method

Online Training

- Implementing each step of gradient descent using backprop involves one pass over the entire training set
- Instead, can also train example-by-example exactly like Online Perceptron Method
- Theory: this works as long as training set is randomly shuffled in the beginning, or we have a continuous stream of i.i.d. examples.

Online Training

- Implementing each step of gradient descent using backprop involves one pass over the entire training set
- Instead, can also train example-by-example exactly like Online Perceptron Method
- Theory: this works as long as training set is randomly shuffled in the beginning, or we have a continuous stream of i.i.d. examples.
- Per-example gradients can be thought of as true gradient + zero-mean noise.
- Hence, this is called **stochastic gradient descent** (SGD for short)

SGD for training Neural Networks

1: Input: training set S as an *input stream*
2: Choose learning rates η_t for $t = 1, 2, \ldots$
3: Initialize weights $W_{ij}^{(l)}$ randomly.
4: **for** each new training example $(\boldsymbol{x}, y)$ **do**
5: Compute partial derivatives $\tilde{W}_{ij}^{(l)}$ using backprop for example $(\boldsymbol{x}, y)$
6: Update weights

$$W_{ij}^{(l)} \leftarrow W_{ij}^{(l)} - \eta_t \tilde{W}_{ij}^{(l)}.$$

7: **end for**

Note: The pseudocode does one pass over the training set. In practice, typically do multiple passes over the training set.

Improving SGD

The following techniques improve SGD generally (not just for training NN):

- SGD works best when data are i.i.d.

Improving SGD

The following techniques improve SGD generally (not just for training NN):

- SGD works best when data are i.i.d.
- If data are in a batch, *randomly shuffle* examples and use that order
- Shuffling *mixes* examples from different classes and accelerates learning by increasing information provided by each new example.

Improving SGD

The following techniques improve SGD generally (not just for training NN):

- SGD works best when data are i.i.d.
- If data are in a batch, *randomly shuffle* examples and use that order
- Shuffling *mixes* examples from different classes and accelerates learning by increasing information provided by each new example.
- In theory, *averaged SGD* (AGSD) defined below has optimal convergence rate:

$$\bar{\boldsymbol{W}}_{\text{final}} := \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{W}_t.$$

  Here, $\boldsymbol{W}_t$ is the vector of all weights (in all layers) computed in the t-th step of SGD, and $\bar{\boldsymbol{W}}_{\text{final}}$ is the final output of the averaged SGD.

Improving SGD

The following techniques improve SGD generally (not just for training NN):

- SGD works best when data are i.i.d.
- If data are in a batch, *randomly shuffle* examples and use that order
- Shuffling *mixes* examples from different classes and accelerates learning by increasing information provided by each new example.
- In theory, *averaged SGD* (AGSD) defined below has optimal convergence rate:

$$\bar{\boldsymbol{W}}_{\text{final}} := \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{W}_t.$$

  Here, $\boldsymbol{W}_t$ is the vector of all weights (in all layers) computed in the t-th step of SGD, and $\bar{\boldsymbol{W}}_{\text{final}}$ is the final output of the averaged SGD.
- In practice, AGSD may converge slowly; problem can be alleviated by averaging only the second half of SGD iterates.

Normalizing input

- Centering inputs so that the mean of each feature is 0 is advantageous:

$$\boldsymbol{x}^{(i)} \leftarrow \boldsymbol{x}^{(i)} - \mu,$$

  here μ is the mean of all the feature vectors in the training set.
- Reason: in backprop, partial derivatives are computed by multiplying features with derivatives

Normalizing input

- Centering inputs so that the mean of each feature is 0 is advantageous:

$$\boldsymbol{x}^{(i)} \leftarrow \boldsymbol{x}^{(i)} - \mu,$$

  here μ is the mean of all the feature vectors in the training set.

- Reason: in backprop, partial derivatives are computed by multiplying features with derivatives

  If all features are large and positive, say, updates of all weights feeding into a given node are in sync, which slows convergence.

Normalizing input

- Centering inputs so that the mean of each feature is 0 is advantageous:

$$\boldsymbol{x}^{(i)} \leftarrow \boldsymbol{x}^{(i)} - \mu,$$

  here μ is the mean of all the feature vectors in the training set.

- Reason: in backprop, partial derivatives are computed by multiplying features with derivatives

  If all features are large and positive, say, updates of all weights feeding into a given node are in sync, which slows convergence.

- After centering, scaling inputs to have variance 1 in each feature is useful:

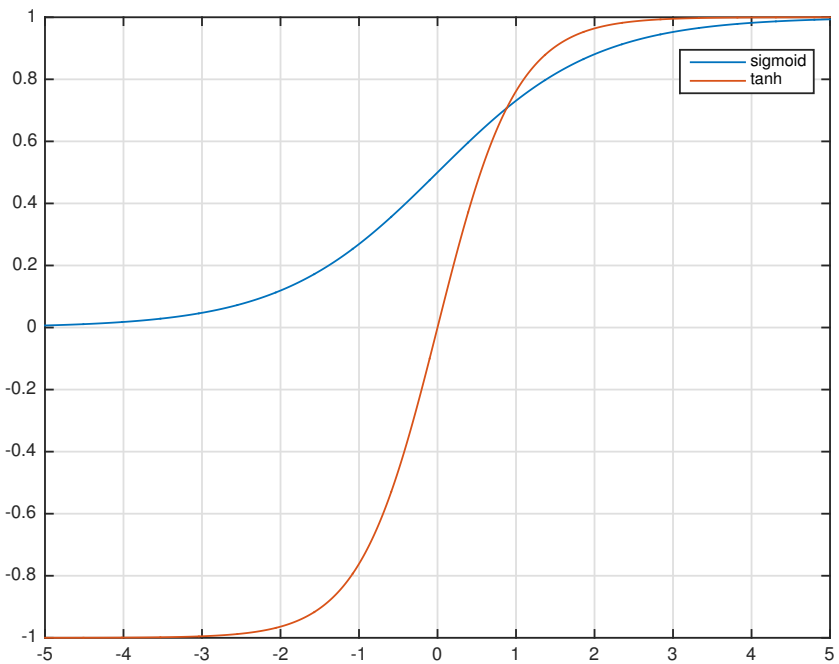
$$x_j^{(i)} \leftarrow \frac{1}{\sigma_j} x_j^{(i)}$$

  where σ_j is the standard deviation of feature j in the training set.

- Reason: have same "scale" features puts weights on the same scale as well, balancing out learning of different weights.

Activation Function

- Typically, hyperbolic tangent $g(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ works better than the sigmoid, $g(x) = \frac{1}{1+e^{-x}}$
- Reason: tanh ensures outputs are 0-mean as long as weights are sufficiently small.



Initializing weights

- Weights typically initialized randomly

Initializing weights

- Weights typically initialized randomly
- Large weights can cause outputs to *saturate*: i.e. become close to 1 or -1
- At saturation, derivatives are very small, causing learning process to slow

Initializing weights

- Weights typically initialized randomly
- Large weights can cause outputs to *saturate*: i.e. become close to 1 or -1
- At saturation, derivatives are very small, causing learning process to slow
- On the other hand, very small weights also lead to very small derivatives

Initializing weights

- Weights typically initialized randomly
- Large weights can cause outputs to *saturate*: i.e. become close to 1 or -1
- At saturation, derivatives are very small, causing learning process to slow
- On the other hand, very small weights also lead to very small derivatives
- Thus, weights should be chosen so that they're not too large or too small

Initializing weights

- Weights typically initialized randomly
- Large weights can cause outputs to *saturate*: i.e. become close to 1 or -1
- At saturation, derivatives are very small, causing learning process to slow
- On the other hand, very small weights also lead to very small derivatives
- Thus, weights should be chosen so that they're not too large or too small
- Good choice (assuming features are normalized): for a unit i with k inputs, choose weight initial weights W_{ij} from a distribution with mean 0 and variance $1/k$, e.g.
  1. Uniform over $\left[-\sqrt{\frac{3}{k}}, \sqrt{\frac{3}{k}}\right]$.
  2. $\mathcal{N}(0, \frac{1}{k})$.

Summary

- Most brains have lots of neurons; each neuron $\approx$ linear–threshold unit (?)
- Perceptrons (one-layer networks) insufficiently expressive
- Multi-layer networks are sufficiently expressive; can be trained by gradient descent via back-propagation
- Doing backprop correctly is a delicate art: some useful techniques are SGD, feature normalization, and using tanh rather than sigmoid, and initializing weights randomly
- Many applications: speech, driving, handwriting, fraud detection, etc.