Artificial Neural Networks

Oliver Schulte - CMPT 726

Neural Networks

- Neural networks arise from attempts to model human/animal brains
  - Many models, many claims of biological plausibility
- We will focus on multi-layer perceptrons
  - Mathematical properties rather than plausibility
    - Prof. Hadley CMPT418

Uses of Neural Networks

- Pros
  - Good for continuous input variables.
  - General continuous function approximators.
  - Highly non-linear.
  - Learn feature functions.
  - Good to use in continuous domains with little knowledge:
    - When you don't know good features.
    - You don't know the form of a good functional model.
- Cons
  - Not interpretable, "black box".
  - Learning is slow.
  - Good generalization can require many datapoints.

Applications

There are many, many applications.

- World-Champion Backgammon Player.
  `http://en.wikipedia.org/wiki/TD-Gammon`
  `http://en.wikipedia.org/wiki/Backgammon`
- No Hands Across America Tour.
  `http://www.cs.cmu.edu/afs/cs/usr/tjochem/www/nhaa/nhaa_home_page.html`
- Digit Recognition with 99.26% accuracy.
- ...

Outline

Outline

Feed-forward Networks

- We have looked at generalized linear models of the form:

$$y(\boldsymbol{x}, \boldsymbol{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\boldsymbol{x})\right)$$

  for fixed non-linear basis functions $\phi(\cdot)$
  - We now extend this model by allowing adaptive basis functions, and learning their parameters
- In feed-forward networks (a.k.a. multi-layer perceptrons) we let each basis function be another non-linear function of linear combination of the inputs:

$$\phi_j(\boldsymbol{x}) = f\left(\sum_{j=1}^{M} \ldots\right)$$

Feed-forward Networks

- We have looked at generalized linear models of the form:

$$y(\boldsymbol{x}, \boldsymbol{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\boldsymbol{x})\right)$$

  for fixed non-linear basis functions $\phi(\cdot)$
  - We now extend this model by allowing adaptive basis functions, and learning their parameters
- In feed-forward networks (a.k.a. multi-layer perceptrons) we let each basis function be another non-linear function of linear combination of the inputs:

$$\phi_j(\boldsymbol{x}) = f\left(\sum_{j=1}^{M} \ldots\right)$$

Feed-forward Networks

- Starting with input $\boldsymbol{x} = (x_1, \ldots, x_D)$, construct linear combinations:

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

  These a_j are known as activations
- Pass through an activation function $h(\cdot)$ to get output $z_j = h(a_j)$
  - Model of an individual neuron

Feed-forward Networks

- Starting with input $\boldsymbol{x} = (x_1, \ldots, x_D)$, construct linear combinations:

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

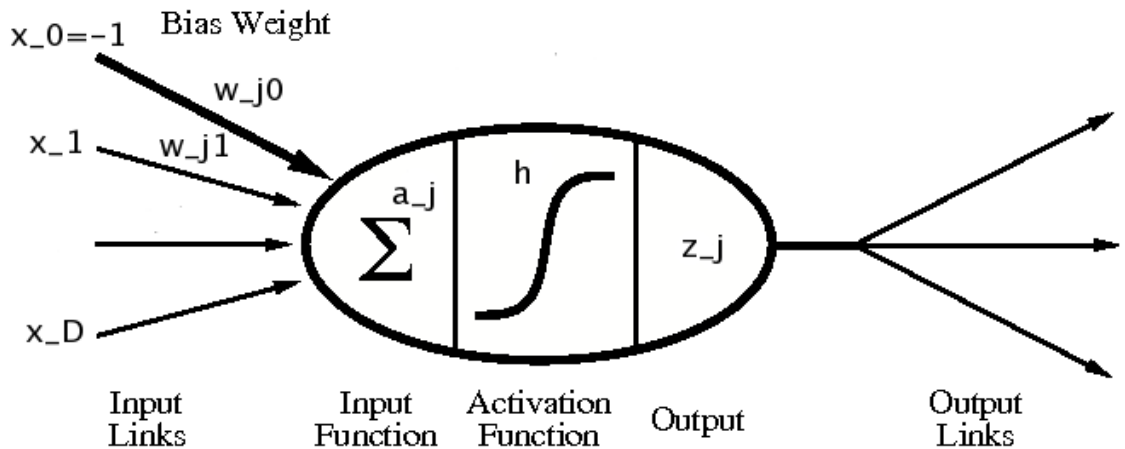
  These a_j are known as activations
- Pass through an activation function $h(\cdot)$ to get output $z_j = h(a_j)$
  - Model of an individual neuron

Feed-forward Networks

- Starting with input $\boldsymbol{x} = (x_1, \ldots, x_D)$, construct linear combinations:

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

  These a_j are known as activations
- Pass through an activation function $h(\cdot)$ to get output $z_j = h(a_j)$
  - Model of an individual neuron

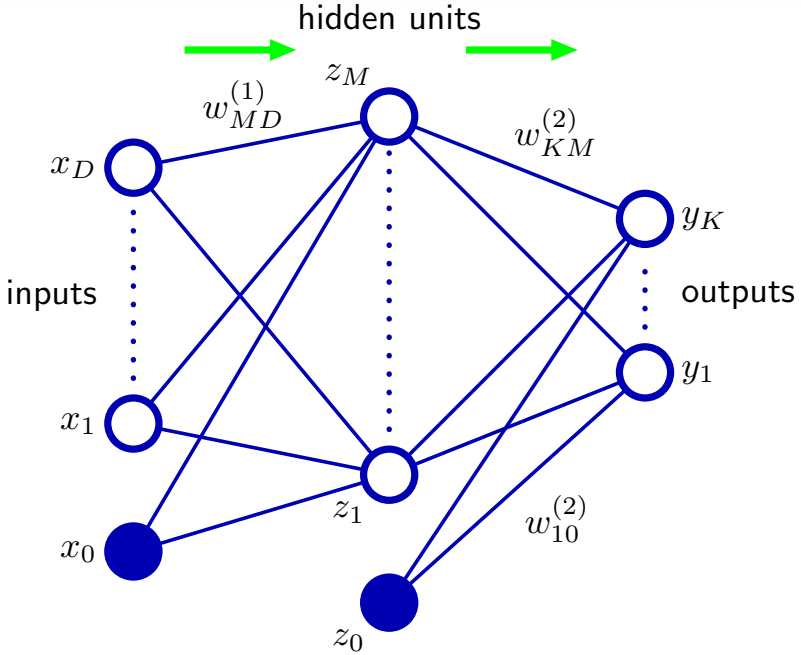


from Russell and Norvig, AIMA2e

Activation Functions

- Can use a variety of activation functions
  - Sigmoidal (S-shaped)
    - Logistic sigmoid $1/(1+\exp(-a))$ (useful for binary classification)
    - Hyperbolic tangent $\tanh$
  - Radial basis function $z_j = \sum_i (x_i - w_{ji})^2$
  - Softmax
    - Useful for multi-class classification
  - Hard Threshold
  - ...
- Should be differentiable for gradient-based learning (later)
- Can use different activation functions in each unit

Feed-forward Networks

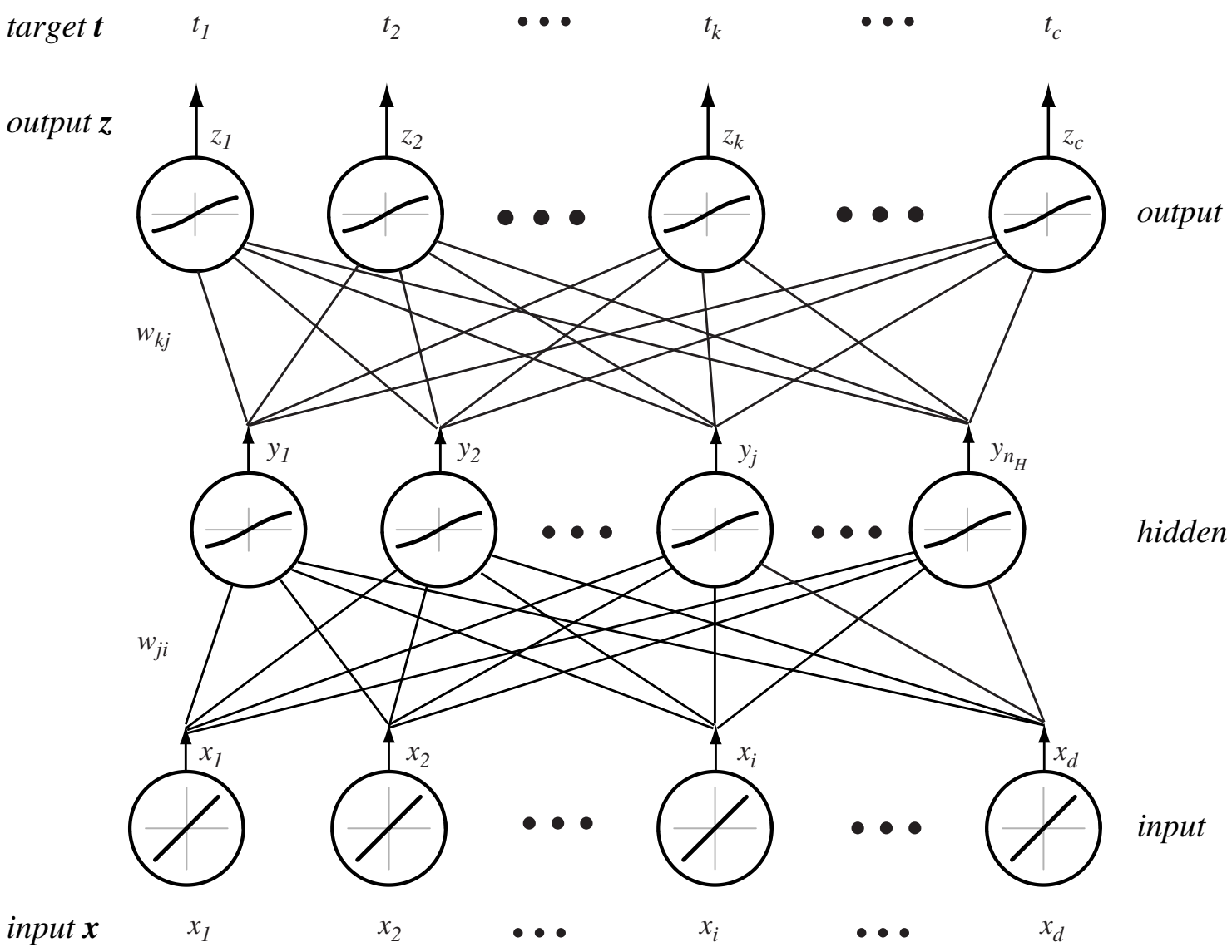


- Connect together a number of these units into a feed-forward network (DAG)
- Above shows a network with one layer of hidden units
- Implements function:

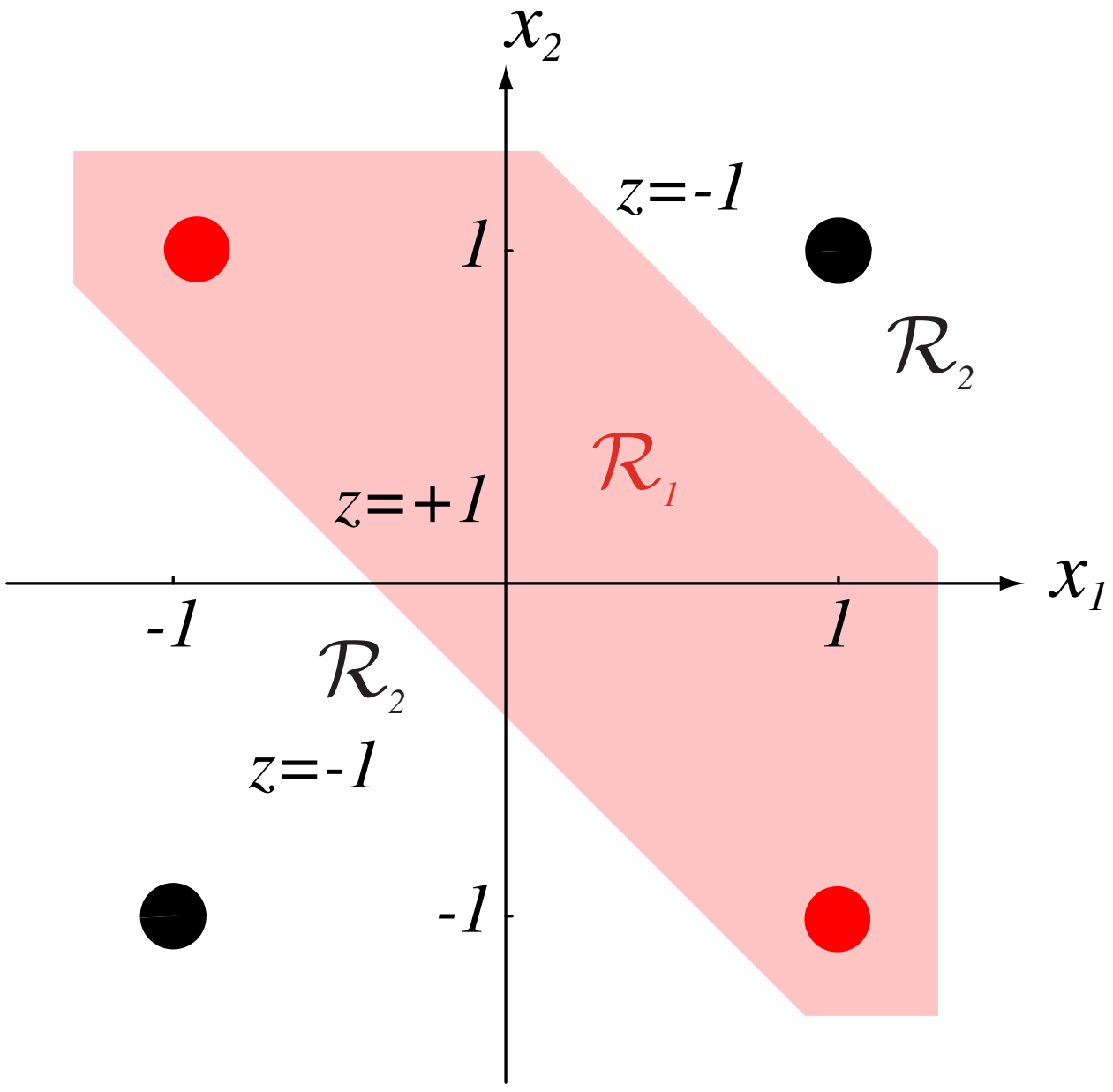
$$y_k(\boldsymbol{x}, \boldsymbol{w}) = h\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

- See `http://aispace.org/neural/`.

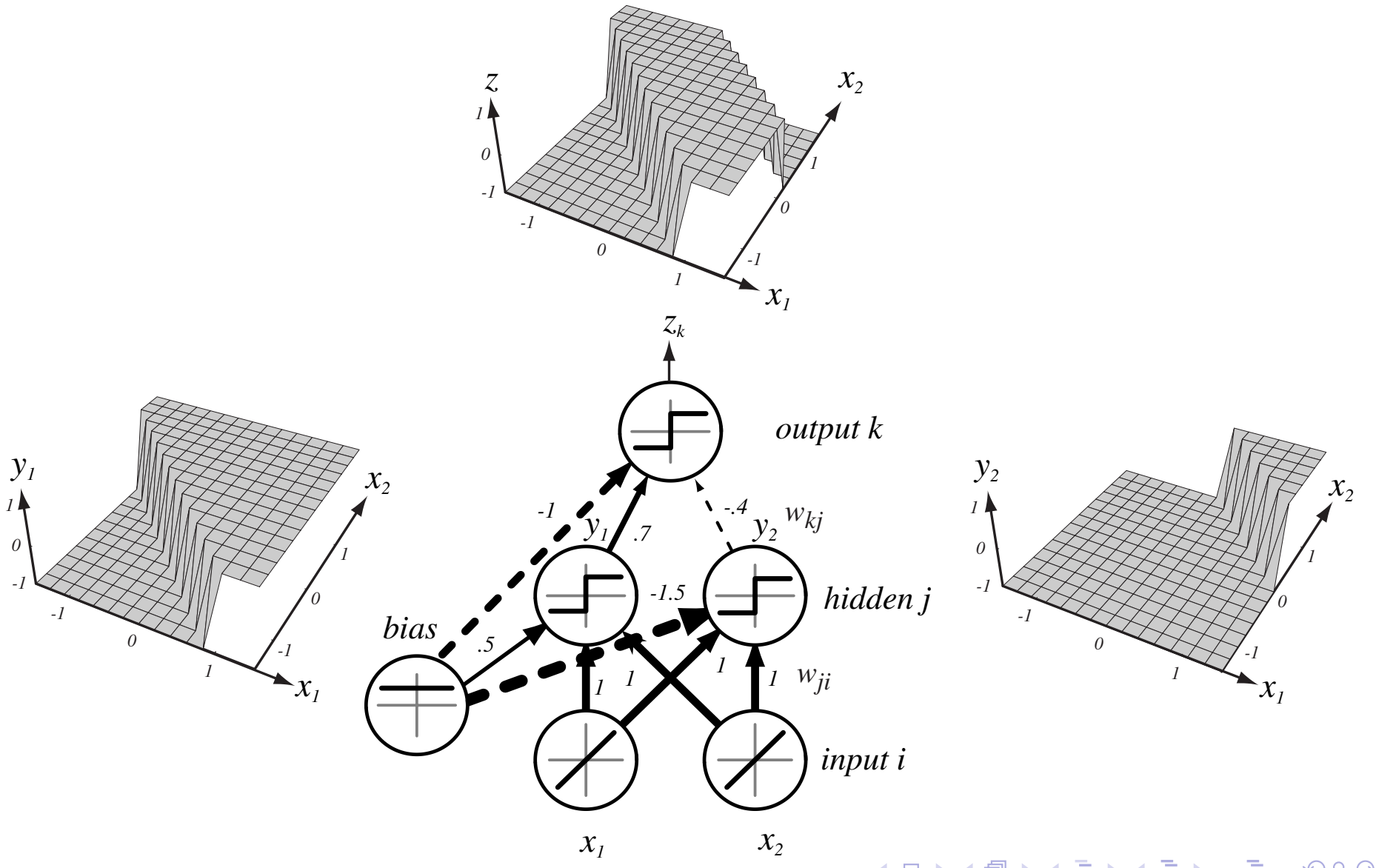
A general network



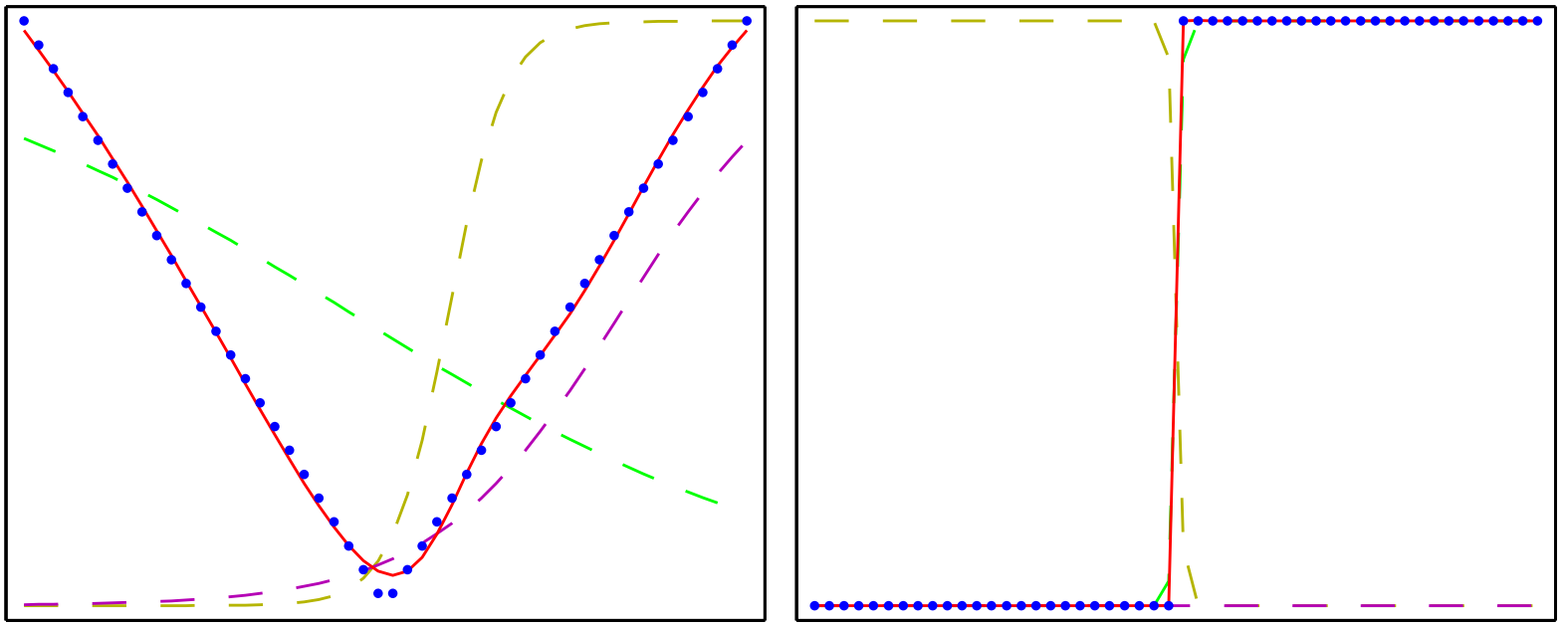
The XOR Problem Revisited



The XOR Problem Solved



Hidden Units Compute Basis Functions

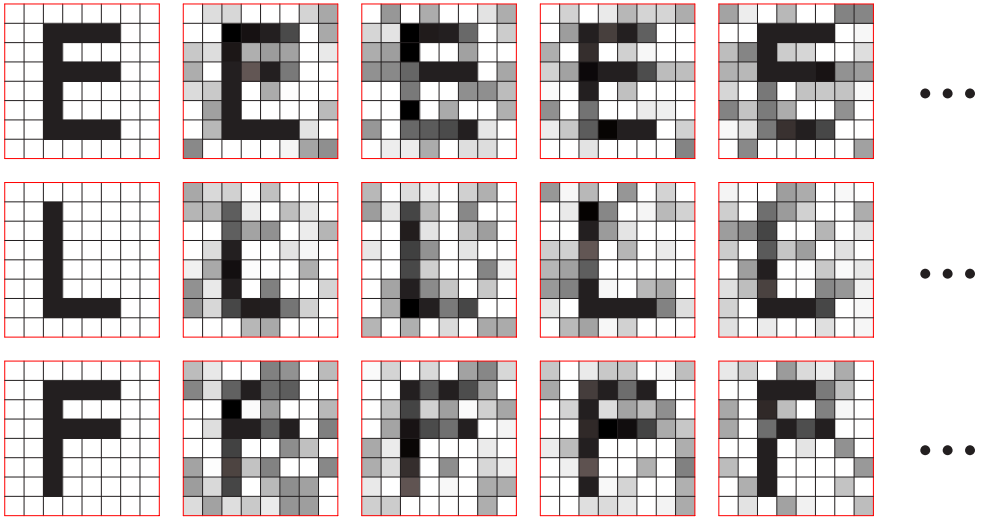


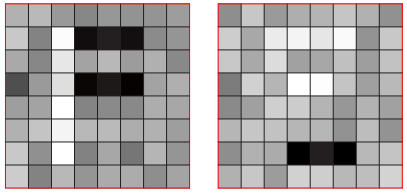
- red dots = network function
- dashed line = hidden unit activation function.
- blue dots = data points

Network function is roughly the sum of activation functions.

Hidden Units As Feature Extractors

sample training patterns





learned input-to-hidden weights

- 64 input nodes
- 2 hidden units
- learned weight matrix at hidden units

Outline

Network Training

- Given a specified network structure, how do we set its parameters (weights)?
  - As usual, we define a criterion to measure how well our network performs, optimize against it
- For regression, training data are $(\boldsymbol{x}_n, t)$, $t_n \in \mathbb{R}$
- Squared error naturally arises:

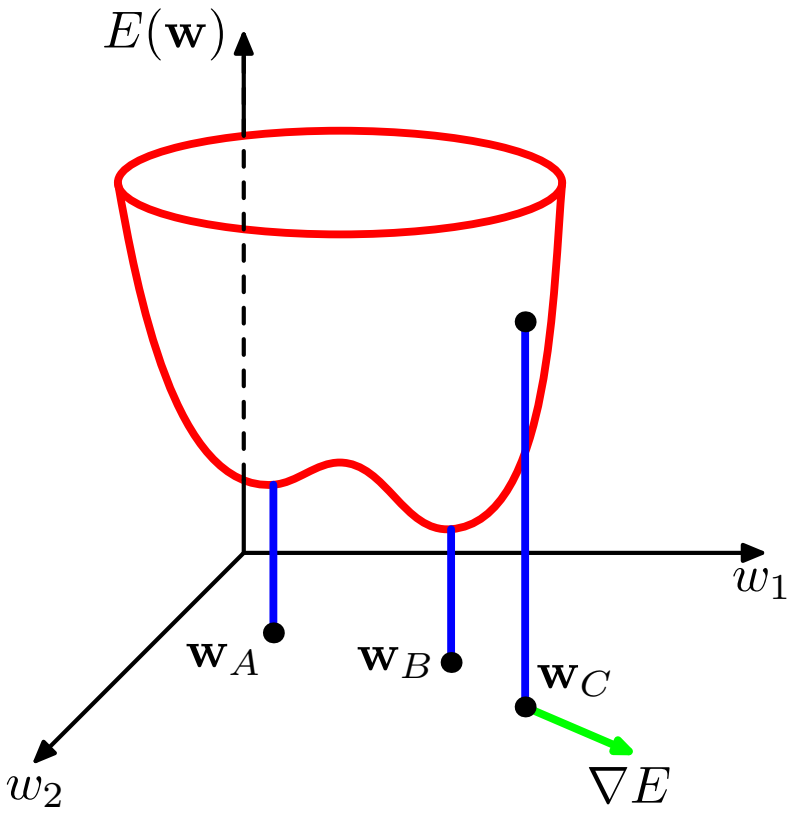
$$E(\boldsymbol{w}) = \sum_{n=1}^{N} \{y(\boldsymbol{x}_n, \boldsymbol{w}) - t_n\}^2$$

Network Training

- Given a specified network structure, how do we set its parameters (weights)?
  - As usual, we define a criterion to measure how well our network performs, optimize against it
- For regression, training data are $(\boldsymbol{x}_n, t)$, $t_n \in \mathbb{R}$
- Squared error naturally arises:

$$E(\boldsymbol{w}) = \sum_{n=1}^{N} \{y(\boldsymbol{x}_n, \boldsymbol{w}) - t_n\}^2$$

Parameter Optimization



- For either of these problems, the error function $E(\boldsymbol{w})$ is nasty
  - Nasty = non-convex
  - Non-convex = has local minima

Descent Methods

- The typical strategy for optimization problems of this sort is a descent method:

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} + \eta \boldsymbol{w}^{(\tau)}$$

- These come in many flavours
  - Gradient descent $\nabla E(\boldsymbol{w}^{(\tau)})$
  - Stochastic gradient descent $\nabla E_n(\boldsymbol{w}^{(\tau)})$
  - Newton-Raphson (second order) ∇^2
- All of these can be used here, stochastic gradient descent is particularly effective
  - Redundancy in training data, escaping local minima

Descent Methods

- The typical strategy for optimization problems of this sort is a descent method:

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} + \eta \boldsymbol{w}^{(\tau)}$$

- These come in many flavours
  - Gradient descent $\nabla E(\boldsymbol{w}^{(\tau)})$
  - Stochastic gradient descent $\nabla E_n(\boldsymbol{w}^{(\tau)})$
  - Newton-Raphson (second order) ∇^2
- All of these can be used here, stochastic gradient descent is particularly effective
  - Redundancy in training data, escaping local minima

Descent Methods

- The typical strategy for optimization problems of this sort is a descent method:

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} + \eta \boldsymbol{w}^{(\tau)}$$

- These come in many flavours
  - Gradient descent $\nabla E(\boldsymbol{w}^{(\tau)})$
  - Stochastic gradient descent $\nabla E_n(\boldsymbol{w}^{(\tau)})$
  - Newton-Raphson (second order) ∇^2
- All of these can be used here, stochastic gradient descent is particularly effective
  - Redundancy in training data, escaping local minima

Computing Gradients

- The function $y(\boldsymbol{x}_n, \boldsymbol{w})$ implemented by a network is complicated
- It isn't obvious how to compute error function derivatives with respect to *hidden* weights.
- The credit assignment problem.

Outline

Error Backpropagation

- Backprop is an efficient method for computing error derivatives $\frac{\partial E_n}{\partial w_{ji}}$ for *all* nodes in the network. Intuition:
  1. Calculating derivatives for weights connected to output nodes is easy.
  2. Treat the derivatives as virtual "error", compute derivative of error for nodes in previous layer.
  3. Repeat until you reach input nodes.
- This procedure propagates backwards the output error signal through the network.

Error at the output nodes

- First, feed training example $\boldsymbol{x}_n$ forward through the network, storing all activations a_j
- Calculating derivatives for weights connected to output nodes is easy
- e.g. For output node with activation $y_k = g(a_k) = g(\sum_i w_{ki} z_i)$:

$$\frac{\partial E_n}{\partial w_{ki}} = \frac{\partial}{\partial w_{ki}} \frac{1}{2}(t_n - y_k)^2 = -(t_n - y_k) g'(a_k) z_i$$

- 0 if no error, or if input z_i from node i is 0.
- Useful notation: $\delta_k \equiv (t_n - y_k) g'(a_k)$.
- Gradient Descent Update:

$$w_{ki} \leftarrow w_{ki} + \eta \delta_k z_i.$$

Error at the output nodes

- First, feed training example $\boldsymbol{x}_n$ forward through the network, storing all activations a_j
- Calculating derivatives for weights connected to output nodes is easy
- e.g. For output node with activation $y_k = g(a_k) = g(\sum_i w_{ki} z_i)$:

$$\frac{\partial E_n}{\partial w_{ki}} = \frac{\partial}{\partial w_{ki}} \frac{1}{2}(t_n - y_k)^2 = -(t_n - y_k) g'(a_k) z_i$$

- 0 if no error, or if input z_i from node i is 0.
- Useful notation: $\delta_k \equiv (t_n - y_k) g'(a_k)$.
- Gradient Descent Update:

$$w_{ki} \leftarrow w_{ki} + \eta \delta_k z_i.$$

Error at the hidden nodes

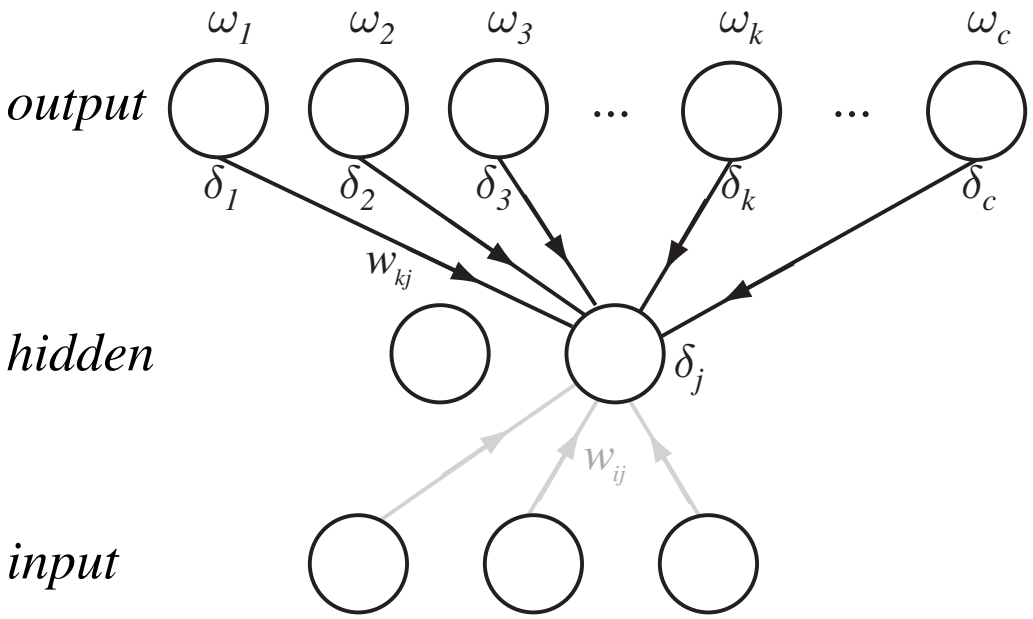
- Consider a hidden node j connected to output nodes.
- Intuition: δ_k is node activation derivative, times output error.
- The error signal δ_j is node activation derivative, times the *weighted sum of contributions to the output errors.*
- In symbols,

$$\delta_j = g'(a_j) \sum_k w_{kj} \delta_k .$$

- Gradient Descent Update:

$$w_{ji} \leftarrow w_{ji} + \eta \delta_j z_i .$$

Backpropagation Picture



The error signal at a hidden unit is proportional to the error signals at the units it influences:

$$\delta_j = g'(a_j) \times \sum_k w_{kj}\delta_k$$

The Backpropagation Algorithm

1. Apply input vector $\boldsymbol{x}_n$ and forward propagate to find all activation levels a_i and output levels z_i.
2. Evaluate the error signals δ_k for all output nodes.
3. Backpropagate the δ_k to obtain error signals δ_j for each hidden node.
4. Perform the gradient descent updates for each weight vector w_{ji}.

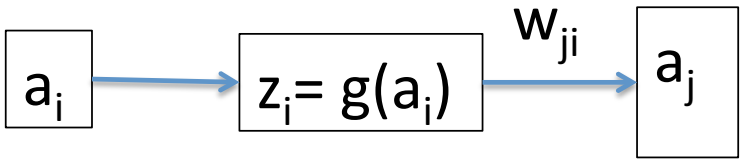
Demo AIspace `http://aispace.org/neural/`.

The Backpropagation Algorithm

1. Apply input vector $\boldsymbol{x}_n$ and forward propgate to find all activation levels a_i and output levels z_i.
2. Evaluate the error signals δ_k for all output nodes.
3. Backpropagate the δ_k to obtain error signals δ_j for each hidden node.
4. Perform the gradient descent updates for each weight vector w_{ji}.

Demo AIspace `http://aispace.org/neural/`.

Correctness Proof for Backpropagation Algorithm.



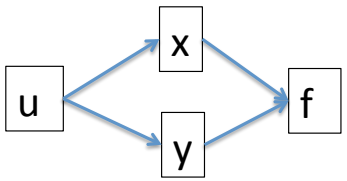
- We need to show that $-\frac{\partial E_n}{w_{ji}} = \delta_j z_i$.
- This follows easily given the following result

Theorem

For each node j, we have $\delta_j = -\frac{\partial E_n}{a_j}$.

- Proof given theorem: $-\frac{\partial E_n}{w_{ji}} = -\frac{\partial E_n}{a_j} \cdot \frac{\partial a_j}{\partial w_{ji}} = \delta_j \cdot z_i$.
- Next we prove the theorem.

Multi-variate Chain Rule



- For $f(x, y)$, with f differentiable wrt x and y, and x and y differentiable wrt u and v:

$$\frac{\partial f}{\partial u} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial u} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial u}$$

and

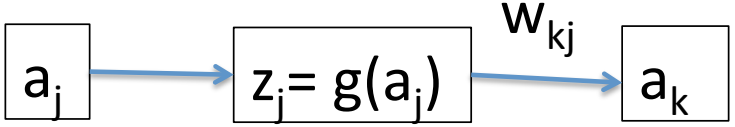
$$\frac{\partial f}{\partial v} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial v} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial v}$$

Proof of Theorem, I

- We want to show that $\delta_j = -\frac{\partial E_n}{a_j}$.
- Think of the error as a function of the activation levels of the nodes *after* node j.
- Formally, we can write $\frac{\partial E_n}{\partial a_j} = \frac{\partial}{\partial a_j} E_n(a_{j_1}, a_{j_2}, \ldots, a_{j_m})$ where $\{j_i\}$ are the indices of the nodes that receive input from j.
- Now using the multi-variate chain rule, we have

$$\frac{\partial E_n}{\partial a_j} = \sum_{k=1}^{m} \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

- It is easy to see that $\frac{\partial a_k}{\partial a_j} = w_{kj} \cdot g'(z_j)$.

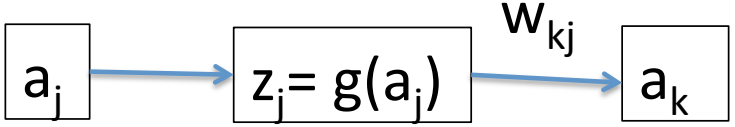


Proof of Theorem, I

- We want to show that $\delta_j = -\frac{\partial E_n}{a_j}$.
- Think of the error as a function of the activation levels of the nodes *after* node j.
- Formally, we can write $\frac{\partial E_n}{\partial a_j} = \frac{\partial}{\partial a_j} E_n(a_{j_1}, a_{j_2}, \ldots, a_{j_m})$ where $\{j_i\}$ are the indices of the nodes that receive input from j.
- Now using the multi-variate chain rule, we have

$$\frac{\partial E_n}{\partial a_j} = \sum_{k=1}^{m} \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

- It is easy to see that $\frac{\partial a_k}{\partial a_j} = w_{kj} \cdot g'(z_j)$.



Proof of Theorem, II

- We want to show that $\delta_j = -\frac{\partial E_n}{a_j}$.
- Proof by backward induction. Easy to see that the claim is true for output nodes. (Exercise).
- Inductive step: Consider node j and suppose that $\delta_k = -\frac{\partial E_n}{a_k}$ for all nodes k that receive input from j.
- Using the multivariate chain rule, we have

$$\begin{aligned} -\frac{\partial E_n}{\partial a_j} &= \sum_{k=1}^{m} -\frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j} \\ &= \sum_{k=1}^{m} \delta_k \frac{\partial a_k}{\partial a_j} = \sum_{k=1}^{m} \delta_k w_{kj} g'(z_j) = \delta_j. \end{aligned}$$

where step 1 applies the inductive hypothesis, step 2 the result from the previous slide, and step 3 the definition of δ_j.

Other Learning Topics

- Regularization: L2-regularizer (weight decay).
- Prune Weights: the Optimal Brain Method.
- Experimenting with Network Architectures is often key.

Outline

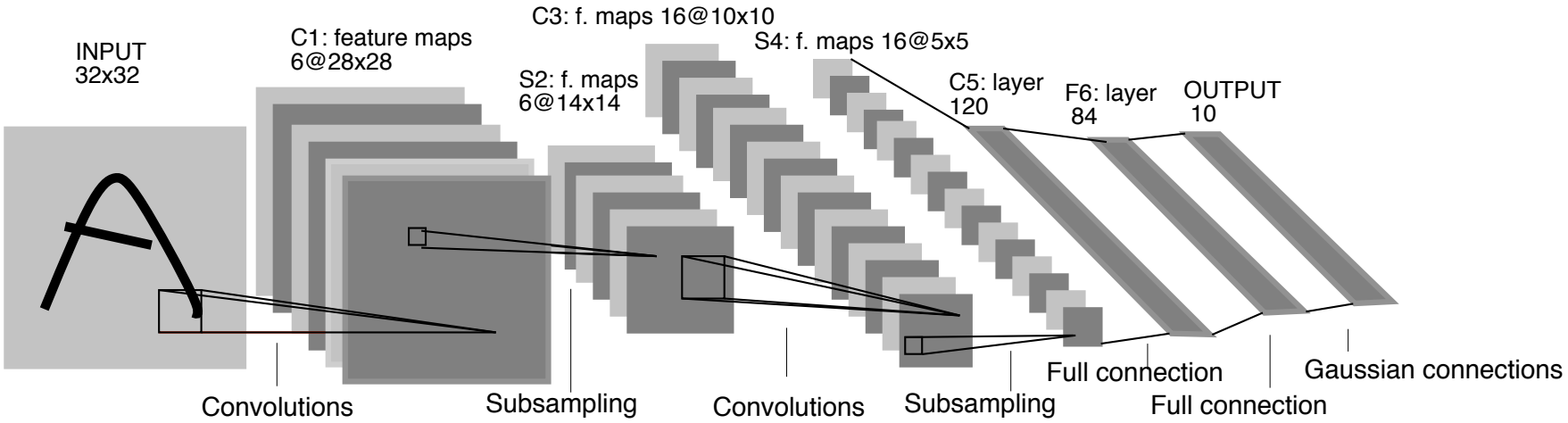
Applications of Neural Networks

- Many success stories for neural networks
  - Credit card fraud detection
  - Hand-written digit recognition
  - Face detection
  - Autonomous driving (CMU ALVINN)

Hand-written Digit Recognition

- MNIST - standard dataset for hand-written digit recognition
  - 60000 training, 10000 test images

LeNet-5



- LeNet developed by Yann LeCun et al.
  - Convolutional neural network
    - Local receptive fields (5x5 connectivity)
    - Subsampling (2x2)
    - Shared weights (reuse same 5x5 “filter”)
    - Breaking symmetry
- See http://www.codeproject.com/KB/library/NeuralNetRecognition.aspx

- The 82 errors made by LeNet5 (0.82% test error rate)

Conclusion

- Feed-forward networks can be used for regression or classification
  - Similar to linear models, except with adaptive non-linear basis functions
  - These allow us to do more than e.g. linear decision boundaries
- Different error functions
- Learning is more difficult, error function not convex
  - Use stochastic gradient descent, obtain (good?) local minimum
- Backpropagation for efficient gradient computation