Neural Networks

Oliver Schulte - CMPT 726

Bishop PRML Ch. 5

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.
- No Hands Across America Tour.
- 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:

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

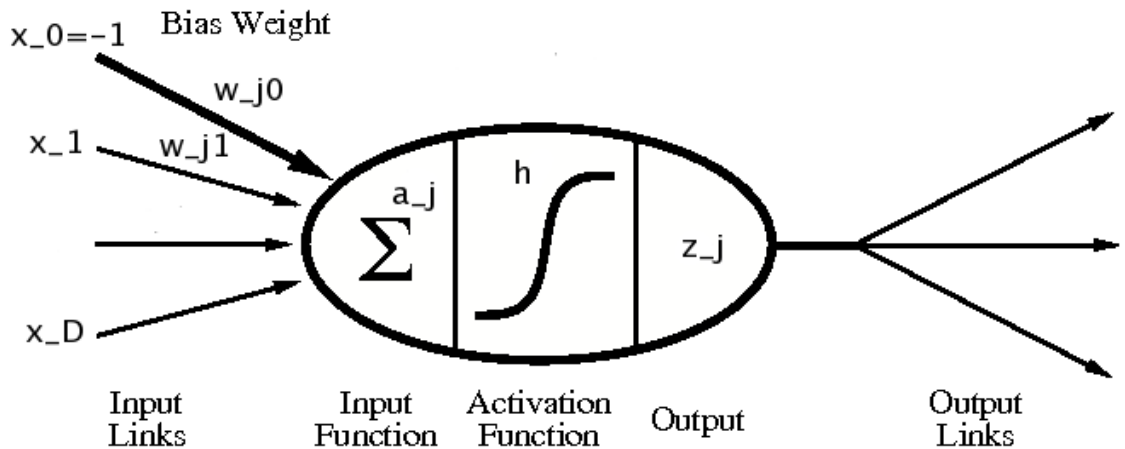
  These a_j are known as activations
- Pass through an activation function $h(\cdot)$ to get output $z_j = h(a_j)$
  - Model of an individual neuron

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

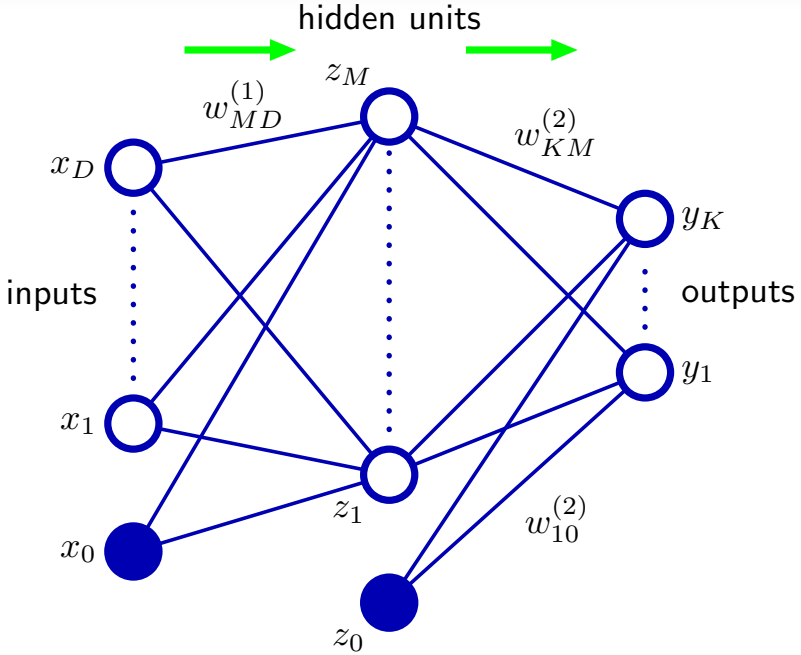


from Russell and Norvig, AIMA2e

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
  - Identity
    - Useful for regression
  - Threshold
  - ...
- Needs to be differentiable for gradient-based learning (later)
- Can use different activation functions in each unit

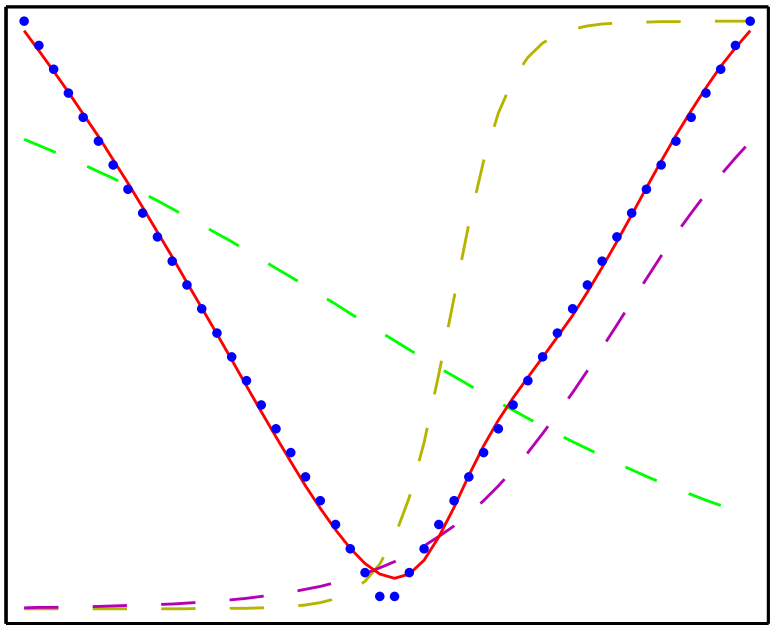
Feed-forward Networks

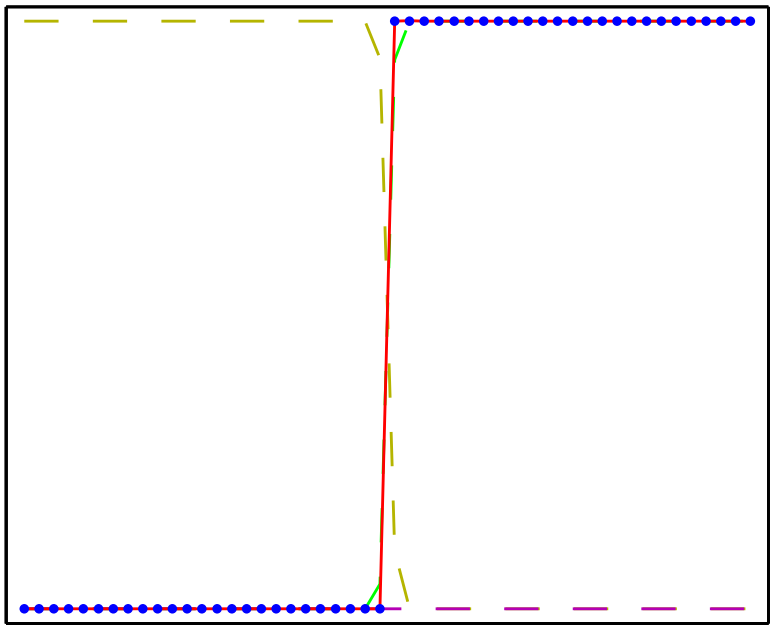


- Connect together a number of these units into a feed-forward network (DAG)
- Above shows a network with one layer of hidden units
- Implements function:

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

Hidden Units Compute Basis Functions





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

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:

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

- For binary classification, this is another discriminative model, ML:

$$p(\boldsymbol{t}|\boldsymbol{w}) = \prod_{n=1}^{N} y_n^{t_n} \{1 - y_n\}^{1-t_n}$$

$$E(\boldsymbol{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\}$$

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:

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

- For binary classification, this is another discriminative model, ML:

$$p(\boldsymbol{t}|\boldsymbol{w}) = \prod_{n=1}^{N} y_n^{t_n} \{1 - y_n\}^{1-t_n}$$

$$E(\boldsymbol{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\}$$

Network Training

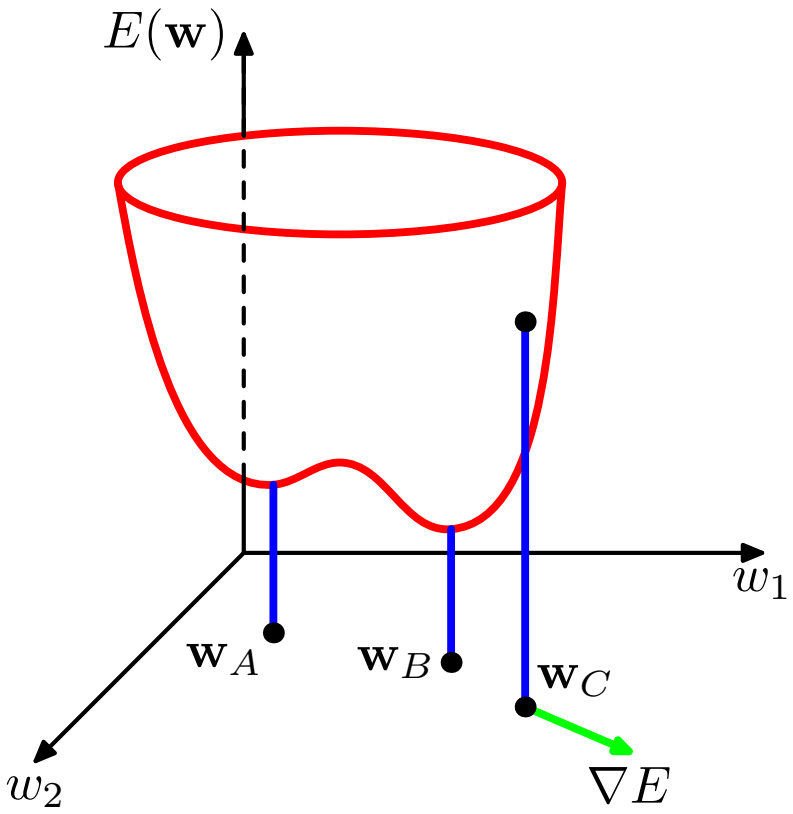
- Given a specified network structure, how do we set its parameters (weights)?
  - As usual, we define a criterion to measure how well our network performs, optimize against it
- For regression, training data are $(\boldsymbol{x}_n, t)$, $t_n \in \mathbb{R}$
  - Squared error naturally arises:

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

- For binary classification, this is another discriminative model, ML:

$$\begin{aligned} p(\boldsymbol{t}|\boldsymbol{w}) &= \prod_{n=1}^{N} y_n^{t_n} \{1 - y_n\}^{1-t_n} \\ E(\boldsymbol{w}) &= -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\} \end{aligned}$$

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:

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

- 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:

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

- 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:

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

- 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 weights
- Numerical method for calculating error derivatives, use finite differences:

$$\frac{\partial E_n}{\partial w_{ji}} \approx \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji} - \epsilon)}{2\epsilon}$$

- How much computation would this take with W weights in the network?
  - $O(W)$ per derivative, $O(W^2)$ total per gradient descent step

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 weights
- Numerical method for calculating error derivatives, use finite differences:

$$\frac{\partial E_n}{\partial w_{ji}} \approx \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji} - \epsilon)}{2\epsilon}$$

- How much computation would this take with W weights in the network?
  - $O(W)$ per derivative, $O(W^2)$ total per gradient descent step

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 weights
- Numerical method for calculating error derivatives, use finite differences:

$$\frac{\partial E_n}{\partial w_{ji}} \approx \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji} - \epsilon)}{2\epsilon}$$

- How much computation would this take with W weights in the network?
  - $O(W)$ per derivative, $O(W^2)$ total per gradient descent step

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 weights
- Numerical method for calculating error derivatives, use finite differences:

$$\frac{\partial E_n}{\partial w_{ji}} \approx \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji} - \epsilon)}{2\epsilon}$$

- How much computation would this take with W weights in the network?
  - $O(W)$ per derivative, $O(W^2)$ total per gradient descent step

Outline

Error Backpropagation

- Backprop is an efficient method for computing error derivatives $\frac{\partial E_n}{\partial w_{ji}}$
  - $O(W)$ to compute derivatives wrt all weights
- 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 linear output nodes $y_k = \sum_i w_{ki} z_i$:

$$\frac{\partial E_n}{\partial w_{ki}} = \frac{\partial}{\partial w_{ki}} \frac{1}{2}(y_n - t_n)^2 = (y_n - t_n) x_{ni}$$

- For hidden layers, propagate error backwards from the output nodes

Error Backpropagation

- Backprop is an efficient method for computing error derivatives $\frac{\partial E_n}{\partial w_{ji}}$
  - $O(W)$ to compute derivatives wrt all weights
- 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 linear output nodes $y_k = \sum_i w_{ki} z_i$:

$$\frac{\partial E_n}{\partial w_{ki}} = \frac{\partial}{\partial w_{ki}} \frac{1}{2}(y_n - t_n)^2 = (y_n - t_n) x_{ni}$$

- For hidden layers, propagate error backwards from the output nodes

Error Backpropagation

- Backprop is an efficient method for computing error derivatives $\frac{\partial E_n}{\partial w_{ji}}$
  - $O(W)$ to compute derivatives wrt all weights
- 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 linear output nodes $y_k = \sum_i w_{ki} z_i$:

$$\frac{\partial E_n}{\partial w_{ki}} = \frac{\partial}{\partial w_{ki}} \frac{1}{2}(y_n - t_n)^2 = (y_n - t_n) x_{ni}$$

- For hidden layers, propagate error backwards from the output nodes

Chain Rule for Partial Derivatives

- A "reminder"
- 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

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

Error Backpropagation

- We can write

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} E_n(a_{j_1}, a_{j_2}, \ldots, a_{j_m})$$

where $\{j_i\}$ are the indices of the nodes in the same layer as node j

- Using the chain rule:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} + \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial w_{ji}}$$

where $\sum_k$ runs over all other nodes k in the same layer as node j.

- Since a_k does not depend on w_{ji}, all terms in the summation go to 0

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$$

Error Backpropagation

- We can write

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} E_n(a_{j_1}, a_{j_2}, \ldots, a_{j_m})$$

where $\{j_i\}$ are the indices of the nodes in the same layer as node j

- Using the chain rule:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} + \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial w_{ji}}$$

where $\sum_k$ runs over all other nodes k in the same layer as node j.

- Since a_k does not depend on w_{ji}, all terms in the summation go to 0

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$$

Error Backpropagation

- We can write

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} E_n(a_{j_1}, a_{j_2}, \ldots, a_{j_m})$$

  where $\{j_i\}$ are the indices of the nodes in the same layer as node j
- Using the chain rule:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} + \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial w_{ji}}$$

  where $\sum_k$ runs over all other nodes k in the same layer as node j.
- Since a_k does not depend on w_{ji}, all terms in the summation go to 0

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$$

Error Backpropagation cont.

- Introduce error $\delta_j \equiv \frac{\partial E_n}{\partial a_j}$

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j \frac{\partial a_j}{\partial w_{ji}}$$

- Other factor is:

$$\frac{\partial a_j}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \sum_k w_{jk} z_k = z_i$$

Error Backpropagation cont.

- Error δ_j can also be computed using chain rule:

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \underbrace{\frac{\partial E_n}{\partial a_k}}_{\delta_k} \frac{\partial a_k}{\partial a_j}$$

  where $\sum_k$ runs over all nodes k in the layer **after** node j.

- Eventually:

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

  - A weighted sum of the later error "caused" by this weight

Error Backpropagation cont.

- Error δ_j can also be computed using chain rule:

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \underbrace{\frac{\partial E_n}{\partial a_k}}_{\delta_k} \frac{\partial a_k}{\partial a_j}$$

  where $\sum_k$ runs over all nodes k in the layer **after** node j.
- Eventually:

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

  - A weighted sum of the later error “caused” by this weight

Outline

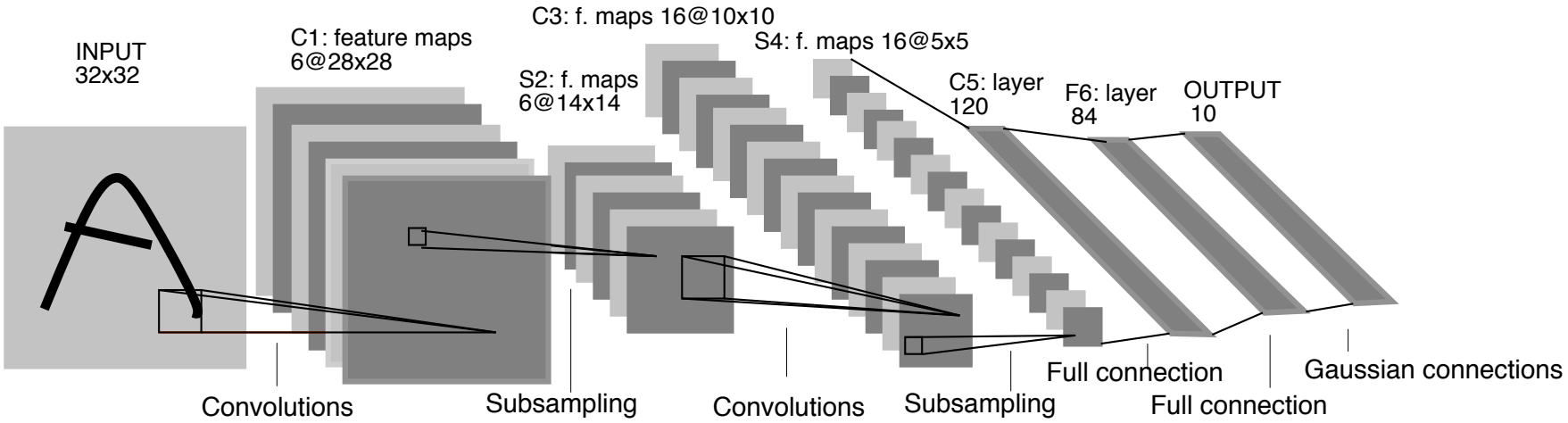
Applications of Neural Networks

- 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

- Readings: Ch. 5.1, 5.2, 5.3
- 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