Neural Networks and Deep Learning

Professor Ameet Talwalkar

November 12, 2015

Outline

How Boosting algorithm works?

- Given: N samples $\{\boldsymbol{x}_n, y_n\}$, where $y_n \in \{+1, -1\}$, and some ways of constructing weak (or base) classifiers
- Initialize weights $w_1(n) = \frac{1}{N}$ for every training sample.
- For t= 1 to T
  1. Train a weak classifier $h_t(\boldsymbol{x})$ based on the current weight $w_t(n)$, by minimizing the weighted classification error

$$\epsilon_t = \sum_n w_t(n)\mathbb{I}[y_n \neq h_t(\boldsymbol{x}_n)]$$

  2. Calculate weights for combining classifiers $\beta_t = \frac{1}{2}\log\frac{1-\epsilon_t}{\epsilon_t}$
  3. Update weights

$$w_{t+1}(n) \propto w_t(n)e^{-\beta_t y_n h_t(\boldsymbol{x}_n)}$$

     and normalize them such that $\sum_n w_{t+1}(n) = 1$.
- Output the final classifier

$$h[\boldsymbol{x}] = \text{sign}\left[\sum_{t=1}^{T}\beta_t h_t(\boldsymbol{x})\right]$$

Derivation of the AdaBoost

Minimize exponential loss

$$\ell^{\text{EXP}}(h(\boldsymbol{x}), y) = e^{-yf(\boldsymbol{x})}$$

Greedily (sequentially) find the best classifier to optimize the loss

A classifier $f_{t-1}(\boldsymbol{x})$ is improved by adding a new classifier $h_t(\boldsymbol{x})$

$$f(\boldsymbol{x}) = f_{t-1}(\boldsymbol{x}) + \beta_t h_t(\boldsymbol{x})$$

$$\begin{aligned}(h_t^*(\boldsymbol{x}), \beta_t^*) &= \arg\min_{(h_t(\boldsymbol{x}),\beta_t)} \sum_n e^{-y_n f(\boldsymbol{x}_n)} \\ &= \arg\min_{(h_t(\boldsymbol{x}),\beta_t)} \sum_n e^{-y_n [f_{t-1}(\boldsymbol{x}_n) + \beta_t h_t(\boldsymbol{x}_n)]}\end{aligned}$$

Nonlinear basis learned by boosting

Two-stage process

- Get $\text{SIGN}[f_1(\boldsymbol{x})]$, $\text{SIGN}[f_2(\boldsymbol{x})]$,$\cdots$,
- Combine into a linear classification model

$$y = \text{SIGN}\left\{\sum_t \beta_t \text{SIGN}[f_t(\boldsymbol{x})]\right\}$$

Equivalently, each stage learns a nonlinear basis $\phi_t(\boldsymbol{x}) = \text{SIGN}[f_t(\boldsymbol{x})]$.

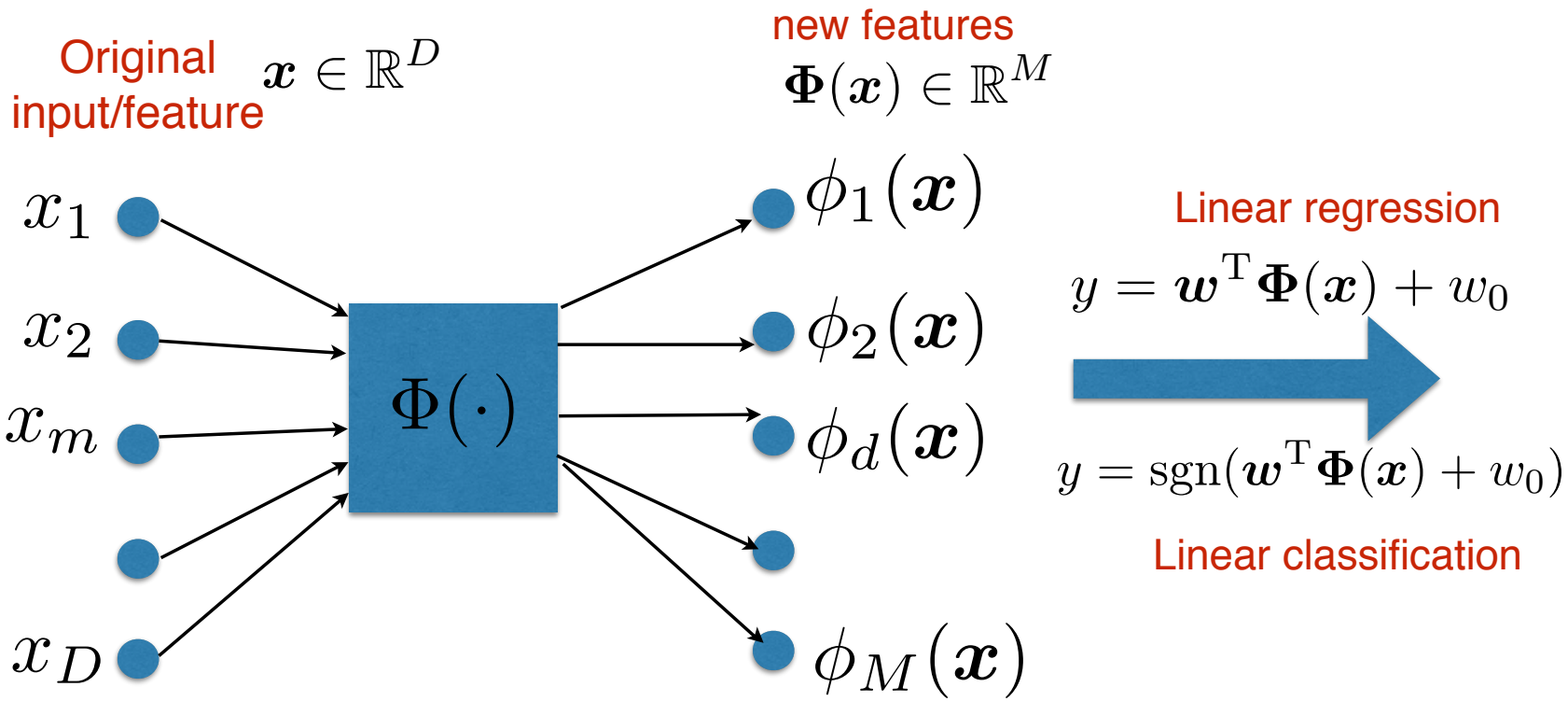
One thought is then, why not learning the basis functions and the classifier at the same time?

Outline

Basic idea

Use nonlinear basis functions

Transform the input feature with nonlinear function



Nonlinear basis as two-layer network

Layered architecture of "neurons"

Input layer: features

hidden layer: nonlinear transformation

Output layer: targets

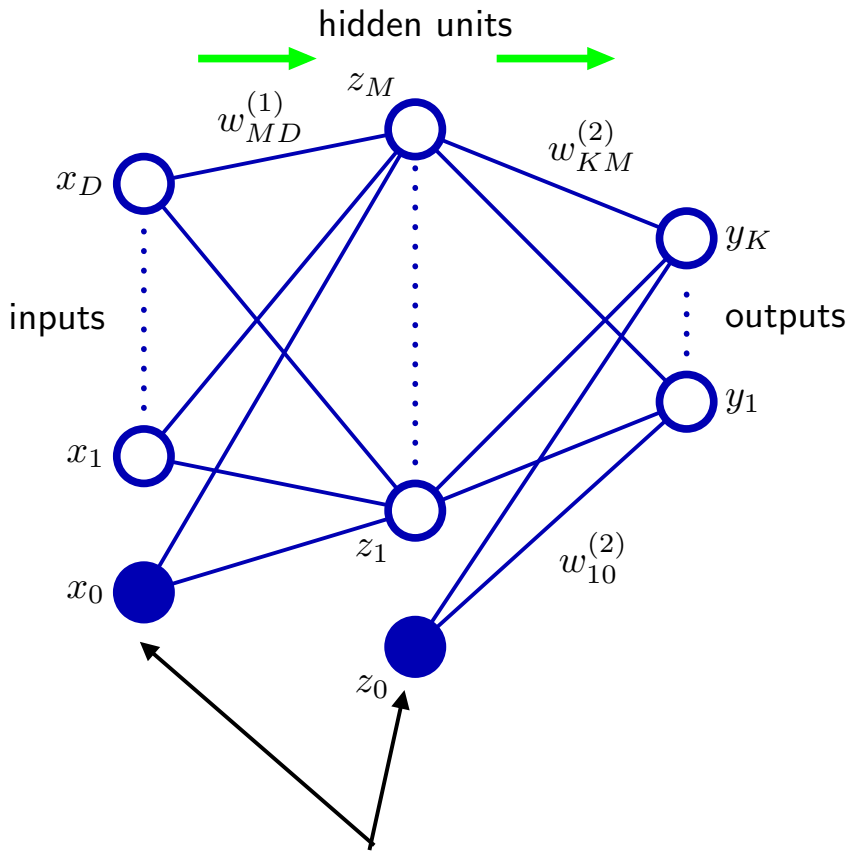
Feedforward computation

hidden layer output:

$$z_j = h(a_j) = h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)$$

Output layer output

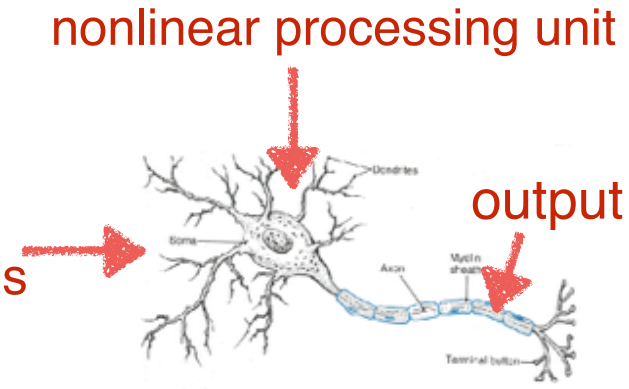
$$y_k = g\left(\sum_{j=0}^{M} w_{kj}^{(2)} z_j\right)$$



we often set these two have a constant value of 1, thus "bias"

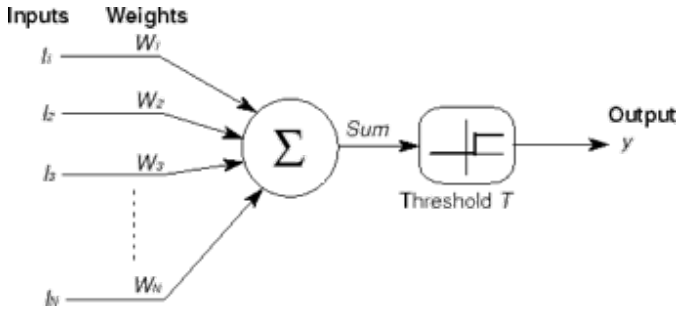
A very concise history

1943 McCulloch-Pitts model of single neurons



1960's Rosenblatt's perceptron learning

1969 Minsky and Papert's perceptron



1985 Hopfield neural nets

1986 Parallel and Distributed Processing (PDP book) and Connectionisms

2006 Deep nets

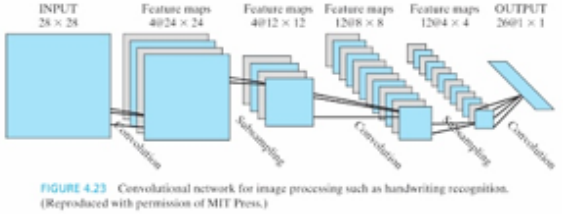
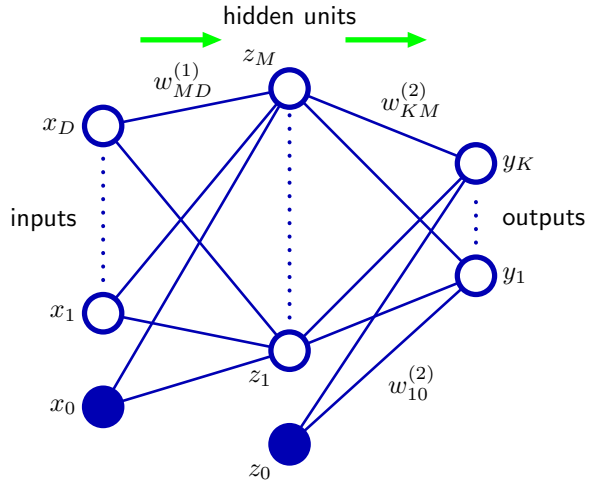


FIGURE 4.23 Convolutional network for image processing such as handwriting recognition. (Reproduced with permission of MIT Press.)

Neural networks are very powerful

Sufficient

Universal approximator: with sufficient number of nonlinear hidden units, linear output unit can approximate any continuous functions



Transfer function for the neurons

sigmoid function

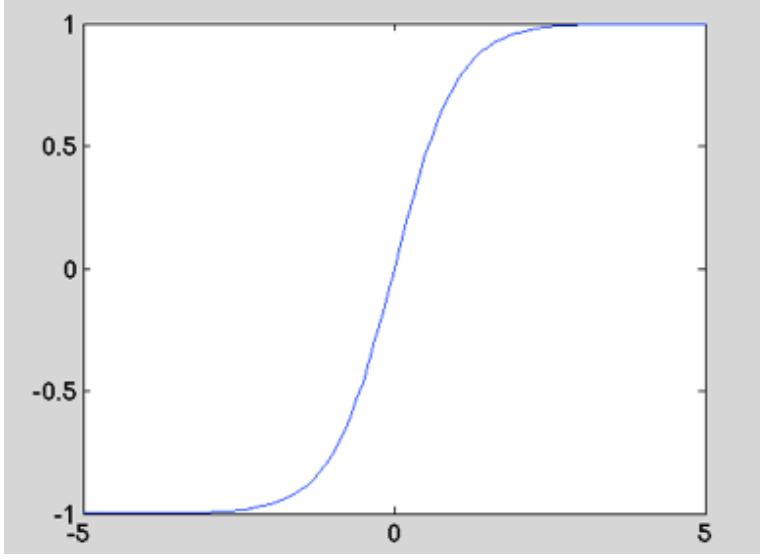
$$h(z) = \frac{1}{1 + e^{-z}}$$

tanh function:

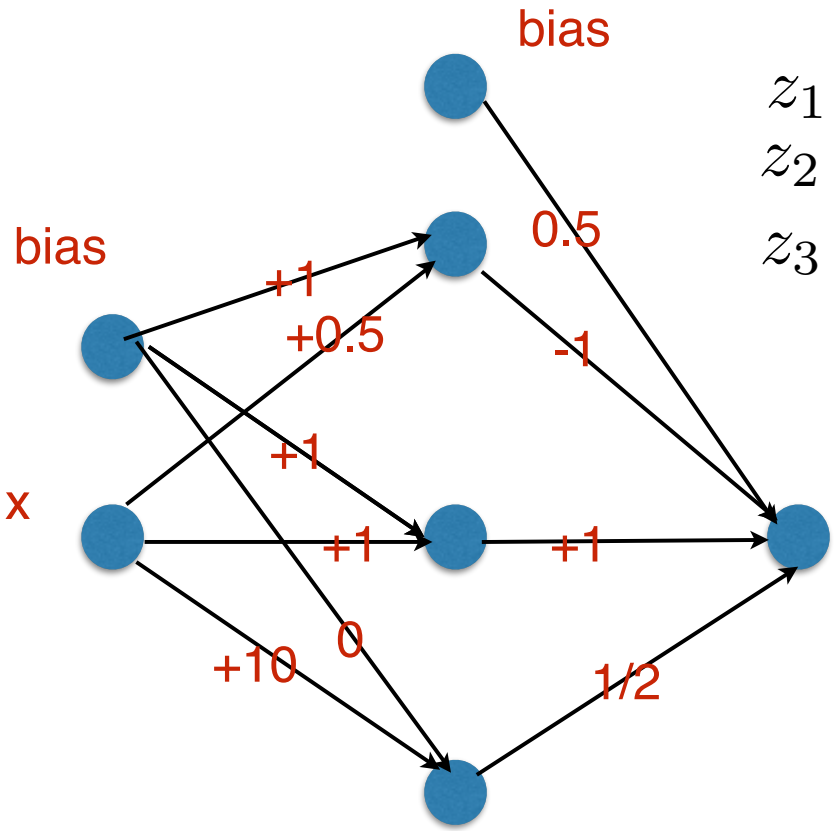
$$h(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

piecewise linear

$$h(z) = \max(0, z)$$



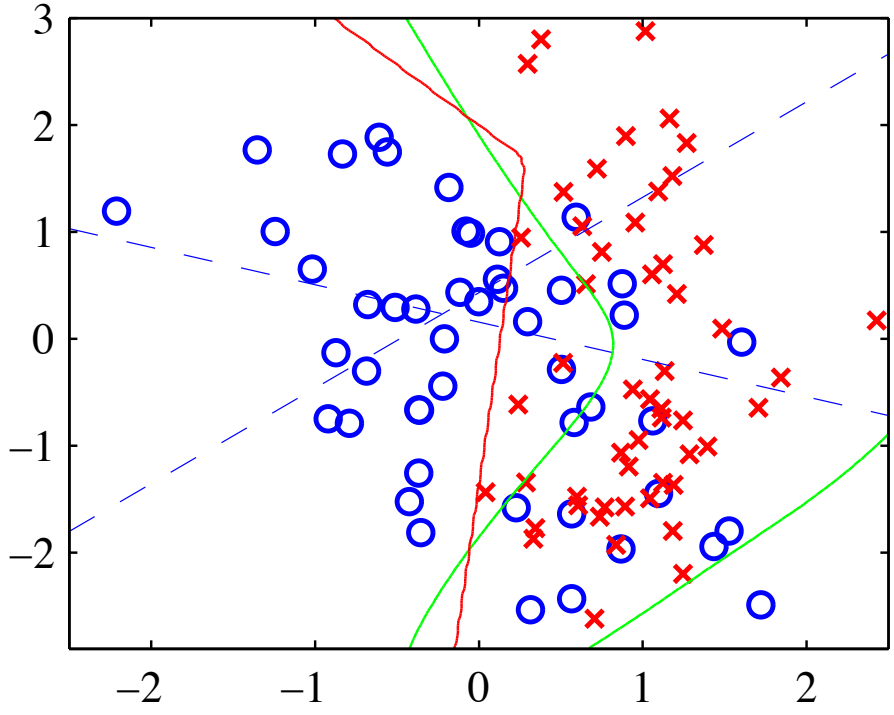
Ex: computing highly nonlinear function



$$z_1 = h(0.5x + 1)$$
$$z_2 = h(x + 1)$$
$$z_3 = h(10x)$$

$$y = -z_1 + z_2 + 0.5 * z3 + 0.5$$

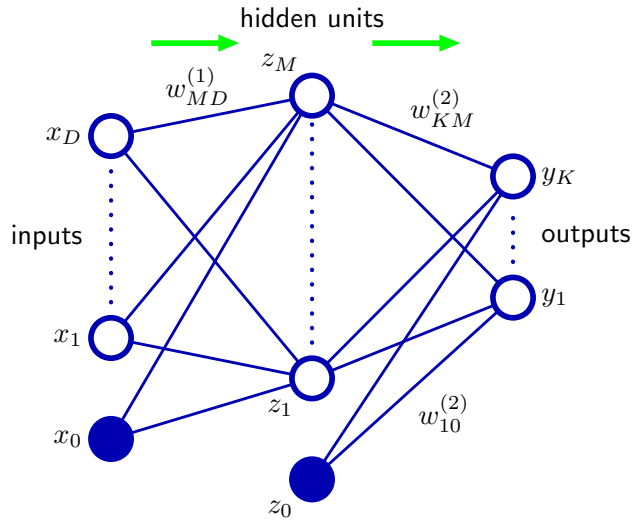
Complicated decision boundaries



Choice of output nodes

Regression

Linear output $$y_k = \sum_k w_{kj}^{(2)} h\left(\sum_i w_{ji}^{(1)} x_i\right)$$



Classification

sigmoid (for binary classification)

$$y = \sigma\left(\sum_k w_{kj}^{(2)} h\left(\sum_i w_{ji}^{(1)} x_i\right)\right)$$

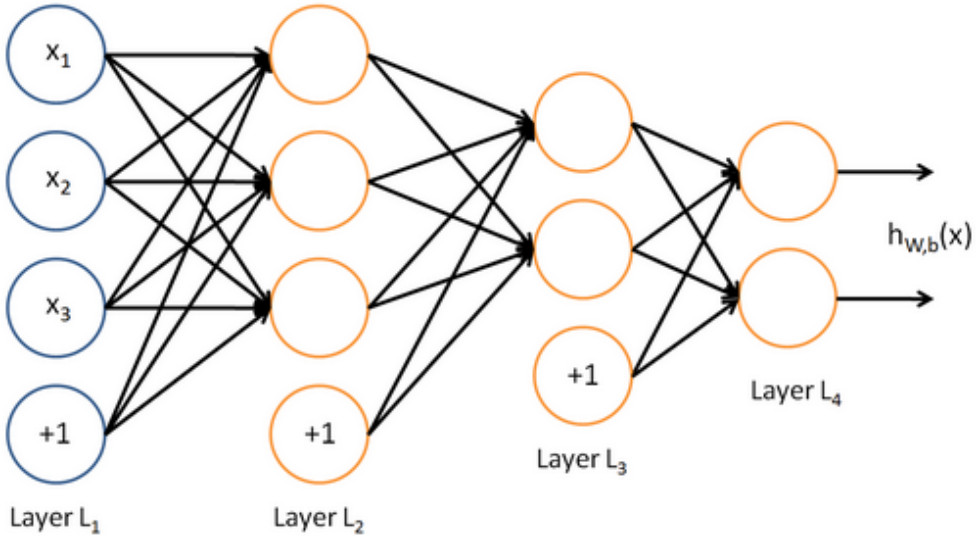
softmax (for multiclass classification)

$$z_k = \sum_k w_{kj}^{(2)} h\left(\sum_i w_{ji}^{(1)} x_i\right) \qquad y_k = \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}}$$

Can have multiple (ie, deep) layers

Implements highly complicated nonlinear mapping

$$\boldsymbol{y} = f(\boldsymbol{x})$$



How to learn the parameters?

Choose the right loss function

Regression: least-square loss

$$\min \sum_n (f(\boldsymbol{x}_n) - y_n)^2$$

Classification: cross-entropy loss

$$\min -\sum_n \sum_k y_{nk} \log f_k(\boldsymbol{x}_n)$$

Very hard optimization problem

Stochastic gradient descent is commonly used

Many optimization tricks are applied

Stochastic gradient descent

High-level idea

Randomly pick a data point (x_n, y_n)

Compute the gradient using only this data point, for example,

$$g = \frac{\partial [f(\boldsymbol{x}_n) - y_n)^2]}{\partial \boldsymbol{w}}$$

Update the parameter right away

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta g$$

Iterate the process until some stop criteria

There are many possible improvements to this simple procedure
(in practice, this procedure works pretty well in many cases, though!)

Several common tricks

Initialization is very important

We are solving a very difficult optimization problem.

There are several heuristics on how to select your starting points wisely.

Learning rate decay

Step size can be big in the begin but should be tuned down later, for example

$$\eta \leftarrow \eta - t\delta_\eta$$

As the iteration t goes up, the learning grate becomes smaller.

Minibatch

Use small batch of data points (instead just one) to estimate gradients more robustly.

Momentum

Remembering the good direction in previous iterations that you have changed the parameters

....

Heavy tuning

In practice

Many tricks require experimenting, and tweaking to obtain the best results

Additionally, other hyperparameters need to be tuned too

- Number of hidden layers?
- Number of hidden units in each layers?
- …

But all those pay off

Deep neural networks attains the best results in automatic speech recognition.

Deep neural networks attains the best results in image recognition.

Deep neural networks attains the best results in recognizing faces

Deep neural networks attains the best results in recognizing poses.

How to compute the gradient?

Even for very complicated nonlinear functions

Computing the gradient is surprisingly simple to implement

The idea behind it is called error back propagation.

It employs the simple chain-rule for taking derivative.

Implemented in many sophisticated packages

Theano

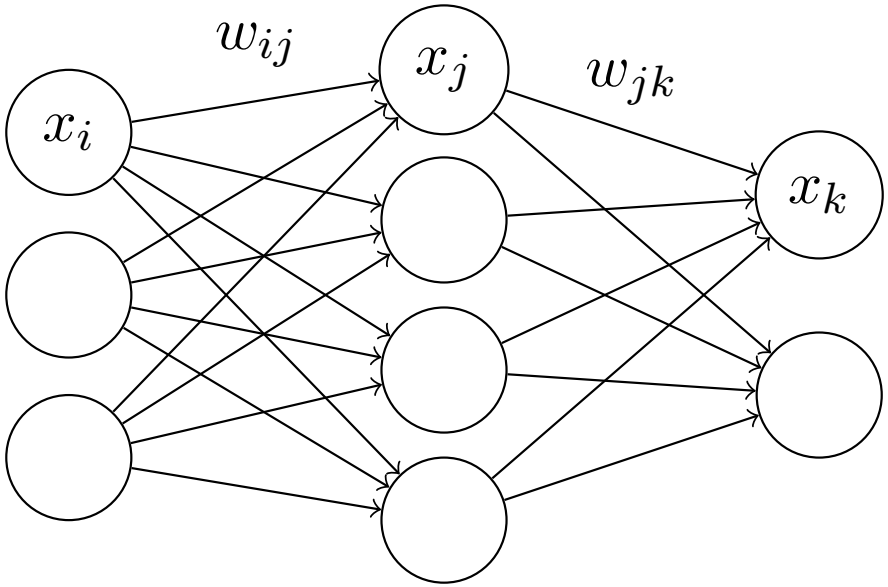
cuDNN

…

Derivation of the error-backpropagation

- Calculate the feed-forward signals from the input to the output.
- Calculate output error E based on the predictions x_k and the target t_k.
- Backpropagate the error by weighting it by the gradients of the associated activation functions and the weights in previous layers.
- Calculating the gradients $\frac{\partial E}{\partial w}$ for the parameters based on the backpropagated error signal and the feedforward signals from the inputs.
- Update the parameters using the calculated gradients $w \leftarrow w - \eta \frac{\partial E}{\partial w}$.

Illustrative example



- w_{ij}: weights connecting node i in layer $(\ell - 1)$ to node j in layer ℓ.
- b_j: bias for node j.
- z_j: input to node j (where $z_j = b_j + \sum_i x_i w_{ij}$).
- g_j: activation function for node j (applied to z_j).
- $x_j = g_j(z_j)$: ouput/activation of node j.
- t_k: target value for node k in the output layer.

Illustrative example (cont'd)

- Network output

$$x_k = g_k(b_k + \sum_j g_j(b_j + \sum_i x_i w_{ij}) w_{jk})$$

- Let's assume that the error function is the sum of the squared difference between the target values t_k and the network output x_k

$$E = \frac{1}{2} \sum_{k \in K} (x_k - t_k)^2$$

- Gradients for the output layer

$$\begin{aligned} \frac{\partial E}{\partial w_{jk}} &= (x_k - t_k) \frac{\partial}{\partial w_{jk}} (x_k - t_k) = (x_k - t_k) g_k'(z_k) \frac{\partial}{\partial w_{jk}} z_k \\ &= (x_k - t_k) g_k'(z_k) x_j = \delta_k x_j \end{aligned}$$

where δ_k is the output error through the top activation layer.

Illustrative example (cont'd)

- Gradients for the hidden layer

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}} &= \sum_{k\in K}(x_k - t_k)\frac{\partial}{\partial w_{ij}}(x_k - t_k) = \sum_{k\in K}(x_k - t_k)g_k'(z_k)\frac{\partial}{\partial w_{ij}}z_k \\ &= \sum_{k\in K}(x_k - t_k)g_k'(z_k)w_{jk}g_j'(z_j)x_i \\ &= x_i g_j'(z_j)\sum_{k\in K}(x_k - t_k)g_k'(z_k)w_{jk} = \delta_j x_i\end{aligned}$$

  where we substituted $z_k = b_k + \sum_j g_j(b_i + \sum_i x_i w_{ij})w_{jk}$ and $\frac{z_k}{w_{ij}} = \frac{z_k}{x_j}\frac{x_j}{w_{ij}} = w_{jk}g_j'(z_j)x_i$.

- The gradients with respect to the biases are respectively:

$$\frac{E}{\partial b_k} = \delta_k, \quad \frac{E}{\partial b_i} = \delta_j$$

Basic idea behind DNNs

Architecturally, a big neural networks (with a lot of variants)

- in depth: 4-5 layers are commonly (Google LeNet uses more than 20)
- in width: the number of hidden units in each layer can be a few thousands
- the number of parameters: hundreds of millions, even billions

Algorithmically, many new things

- Pre-training: do not do error-backprogation right away
- Layer-wise greedy: train one layer at a time
- ...

Computing

- Heavy computing: in both speed in computation and coping with a lot of data
- Ex: fast Graphics Processing Unit (GPUs) are almost indispensable

Good references

- Easy to find as DNNs are very popular these days
- Many, many online video tutorials
- Good open-source packages: Theano, Caffe, MatConvNet, TensorFlow, etc
- Examples:
  - Wikipedia entry on "Deep Learning" `http://en.wikipedia.org/wiki/Deep_learning` provides a decent portal to many things including deep belief networks, convolution nets
  - A collection of tutorials and codes for implementing them in Python `http://www.deeplearning.net/tutorial/`

Outline

Summary of the course so far: a short list of important concepts

Supervised learning has been our focus

- Setup: given a training dataset $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$, we learn a function $h(\boldsymbol{x})$ to predict $\boldsymbol{x}$'s true value y (i.e., regression or classification)
- Linear vs. nonlinear features
  1. Linear: $h(\boldsymbol{x})$ depends on $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$
  2. Nonlinear: $h(\boldsymbol{x})$ depends on $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x})$, which in terms depends on a kernel function $k(\boldsymbol{x}_m, \boldsymbol{x}_n) = \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n)$,
- Loss function
  1. Squared loss: least square for regression (minimizing residual sum of errors)
  2. Logistic loss: logistic regression
  3. Exponential loss: AdaBoost
  4. Margin-based loss: support vector machines
- Principles of estimation
  1. Point estimate: maximum likelihood, regularized likelihood

cont'd

- Optimization
  1. Methods: gradient descent, Newton method
  2. Convex optimization: global optimum vs. local optimum
  3. Lagrange duality: primal and dual formulation
- Learning theory
  1. Difference between training error and generalization error
  2. Overfitting, bias and variance tradeoff
  3. Regularization: various regularized models