CSCI567 Machine Learning (Fall 2014)

Drs. Sha & Liu

{feisha,yanliu.cs}@usc.edu

October 28, 2014
Outline

1. Review of last lecture
   - AdaBoost
   - Boosting as learning nonlinear basis

2. Neural networks

3. Summary
How Boosting algorithm works?

- Given: $N$ samples $\{x_n, y_n\}$, 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 $h_t(x)$ based on the current weight $w_t(n)$, by minimizing the weighted classification error
     \[ \epsilon_t = \sum_n w_t(n) \mathbb{I}[y_n \neq h_t(x_n)] \]
  2. Calculate weights for combining classifiers $\beta_t = \frac{1}{2} \log \frac{1-\epsilon_t}{\epsilon_t}$
  3. Update weights
     \[ w_{t+1}(n) \propto w_t(n) e^{-\beta_t y_n h_t(x_n)} \]
     and normalize them such that $\sum_n w_{t+1}(n) = 1$.
- Output the final classifier
  \[ h[x] = \text{sign} \left[ \sum_{t=1}^{T} \beta_t h_t(x) \right] \]
**Derivation of the AdaBoost**

**Minimize exponential loss**

\[
\ell^{\text{EXP}}(h(\mathbf{x}), y) = e^{-yf(\mathbf{x})}
\]

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

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

\[
f(\mathbf{x}) = f_{t-1}(\mathbf{x}) + \beta_t h_t(\mathbf{x})
\]

\[
(h^*_t(\mathbf{x}), \beta^*_t) = \arg\min_{(h_t(\mathbf{x}), \beta_t)} \sum_{n} e^{-y_nf(\mathbf{x}_n)}
\]

\[
= \arg\min_{(h_t(\mathbf{x}), \beta_t)} \sum_{n} e^{-y_n[f_{t-1}(\mathbf{x}_n)+\beta_th_t(\mathbf{x}_n)]}
\]
Nonlinear basis learned by boosting

Two-stage process

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

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

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

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

1 Review of last lecture

2 Neural networks
   - Algorithm
   - Deep Neural Networks (DNNs)

3 Summary
Basic idea

**Use nonlinear basis functions**

Transform the input feature with nonlinear function

Original input/feature $\mathbf{x} \in \mathbb{R}^D$

new features $\Phi(\mathbf{x}) \in \mathbb{R}^M$

$x_1$ $\rightarrow$ $\phi_1(\mathbf{x})$

$x_2$ $\rightarrow$ $\phi_2(\mathbf{x})$

$\ldots$

$x_m$ $\rightarrow$ $\phi_m(\mathbf{x})$

$x_D$ $\rightarrow$ $\phi_M(\mathbf{x})$

Linear regression

$y = \mathbf{w}^T \Phi(\mathbf{x}) + w_0$

Linear classification

$y = \text{sgn}(\mathbf{w}^T \Phi(\mathbf{x}) + w_0)$
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
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

\[ z_1 = h(0.5x + 1) \]
\[ z_2 = h(x + 1) \]
\[ z_3 = h(10x) \]

\[ y = -z_1 + z_2 + 0.5 \cdot z_3 + 0.5 \]
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)

\[ z_k = \sum_k w_{kj}^{(2)} h \left( \sum_i w_{ji}^{(1)} x_i \right) \]

\[ y_k = \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}} \]
Can have multiple (ie, deep) layers

Implements highly complicated nonlinear mapping

\[ y = f(x) \]
How to learn the parameters?

Choose the right loss function

Regression: least-square loss

\[ \min \sum_{n} (f(x_n) - y_n)^2 \]

Classification: cross-entropy loss

\[ \min - \sum_{n} \sum_{k} y_{nk} \log f_k(x_n) \]

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,

\[
g = \frac{\partial \left[f(x_n) - y_n\right]^2}{\partial w}
\]

Update the parameter right away

\[
w \leftarrow w - \eta g
\]

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 rate 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

Illustrative example

Derivation on Blackboard

Details can be found on the textbooks as well as many online tutorials.
Key steps (essentially, chain rule in calculus)

To compute

$$\frac{\partial \ell}{\partial w_{ji}}$$

we compute

$$\frac{\partial \ell}{\partial w_{ji}} = z_i \frac{\partial \ell}{\partial a_j}$$

as $w_{ji}$ affects only $a_j$

Nonlinear passthru

$$\frac{\partial \ell}{\partial a_j} = \frac{\partial \ell}{\partial z_j} \frac{\partial z_j}{\partial a_j} = h'(a_j) \frac{\partial \ell}{\partial z_j}$$

Recursion

$$\frac{\partial \ell}{\partial z_j} = \sum_k \frac{\partial \ell}{\partial a_k} \frac{\partial a_k}{\partial z_j}$$
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-backpropagation 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: Theanos, cuDNN, Caffe, 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

1. Review of last lecture
2. Neural networks
3. Summary
   - Supervised learning
Summary of the course so far: a short list of important concepts

**Supervised learning** has been our focus

- Setup: given a training dataset \( \{x_n, y_n\}_{n=1}^N \), we learn a function \( h(x) \) to predict \( x \)'s true value \( y \) (i.e., regression or classification)

- Linear vs. nonlinear features
  1. Linear: \( h(x) \) depends on \( w^T x \)
  2. Nonlinear: \( h(x) \) depends on \( w^T \phi(x) \), which in terms depends on a kernel function \( k(x_m, x_n) = \phi(x_m)^T \phi(x_n) \),

- 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