Lec 22: Deep Learning

Ailin Zhang

Recap: AdaBoost, Gradient Boost

- In AdaBoost, the loss function is in the exponential form:

$$loss(\beta) = \sum_{i=1}^{n} \exp(-y_i \sum_{k=1}^{d} \beta_k h_k(x_i))$$

- When training AdaBoost classifier, we sequentially add members to the committee:

$$\text{current committee:} \quad \sum_{k=1}^{m} \beta_k h_k(X_i),$$

$$\text{add a new member:} \quad \sum_{k=1}^{m} \beta_k h_k(X_i) + \beta_{\text{new}} h_{\text{new}}(X_i).$$

Suppose the current committee has m classifiers, and we want to add a new member h_{new}.

Recap: AdaBoost, Gradient Boost

- $$loss(\beta_{\text{new}}, h_{\text{new}}) = \sum_{i=1}^{n} e^{-y_i\left(\sum_{k=1}^{m} \beta_k h_k(X_i) + \beta_{\text{new}} h_{\text{new}}(X_i)\right)}$$

  where we assume the current members and their weights of votes are fixed.
- Take derivative

$$\frac{\partial loss}{\partial \beta_{\text{new}}} = \sum_{i=1}^{n} e^{-y_i\left(\sum_{k=1}^{m} \beta_k h_k(X_i) + \beta_{\text{new}} h_{\text{new}}(X_i)\right)} \cdot \left(-y_i h_{\text{new}}(X_i)\right).$$

- For the current committee without adding a new member, let

$$w_i = e^{-y_i\left(\sum_{k=1}^{m} \beta_k h_k(x_i)\right)}.$$

Recap: AdaBoost, Gradient Boost

- Normalize $w_i \leftarrow w_i / \sum_{i=1}^{n} w_i$ to make it a distribution. This distribution focuses on those examples that are not well classified by the current committee.
- Choose the weak classifier h_{new} by maximizing $\sum_{i=1}^{n} w_i y_i h_{\text{new}}(x_i)$ for the steepest drop in loss.
- We then solve β_{new} by setting the derivative to 0,

$$\sum_{i=1}^{n} w_i e^{-y_i \beta_{\text{new}} h_{\text{new}}(x_i)} \cdot y_i h_{\text{new}}(x_i) = 0,$$

$$\sum_{i \in \text{correct}} w_i e^{-\beta_{\text{new}}} = \sum_{i \in \text{wrong}} w_i e^{\beta_{\text{new}}},$$

$$\sum_{i \in \text{correct}} w_i = \sum_{i \in \text{wrong}} w_i e^{2\beta_{\text{new}}}.$$

Recap: AdaBoost, Gradient Boost

- If we define error rate as

$$\epsilon = \frac{\sum_{i \in \text{wrong}} w_i}{\sum_i w_i},$$

  β_{new} can be obtained as

$$\beta_{\text{new}} = \frac{1}{2} \log \frac{1-\epsilon}{\epsilon}.$$

- It says that the weight is determined by how much error h_{new} made on the weighted data. This explains the name of adaBoost, where ''ada" means adapative.

Gradient Boosting

- Consider a more general minimization problem: $min_f \{\sum_{i=1}^{n} L(y_i, f(x_i))\}$, for any loss function L.
- The gradient boosting algorithm solves this problem by iteratively changing $f(x)$.
  - Take any given prediction (candidate for minimization) and call it $\hat{f}(x)$.
  - With gradient boosting, to minimize the objective function, we would like to let $(\Delta f_i)_{i=1...n} \propto -\left(\frac{\partial L}{\partial f_i}\right)_{i=1...n}$
  - We choose β and h by minimizing a squared-loss problem:

$$min \left\{ \sum_{i=1}^{n} (\tilde{y}_i - \beta h(x_i))^2 \right\}$$

  - Note that the loss we want to minimize is actually L, which may not be squared loss. Thus we reestimate β by minimizing L

Algorithm

Suppose we want to minimize a general loss:

$$\min \sum_{i=1}^{n} L(y_i, f_i).$$

Then gradient boosting consists of the following steps

- *Step 1*: Set $f_0(x_i) = \frac{1}{n}\sum_1^n y_i$, $m = 1$.
- *Step 2*: Compute residuals $y_i^m = y_i - f_{m-1}(x_i)$.
- *Step 3*: Choose h^* by minimizing $\sum_{i=1}^n (y_i^m - \beta h(x_i))^2$, and set $h_m = h^*$.
- *Step 4*: Reestimate β by $\beta^* = \arg\min_\beta \sum_{i=1}^n L(y_i, f)$
- *Step 5*: $m \leftarrow m + 1$; repeat Step 2.

XGB - Extreme Gradient Boosting

- XGB is a boosting algorithm that relies on the Newton-Rhapson method.

XGB - Extreme Gradient Boosting

- XGB is a boosting algorithm that relies on the Newton-Rhapson method.
- Consider a second-order aproximation for any loss function at a given point $\hat{\theta}$. We can write it as
$L(\theta) = L(\hat{\theta}) + L^{'}(\hat{\theta})(\theta - \hat{\theta}) + \frac{1}{2}L^{''}(\hat{\theta})(\theta - \hat{\theta})^2$.
- Then, an approximation for the total loss, around the current $\hat{f}(x)$ is:

$$\sum_{i=1}^{n} \left\{ L(y_i, \hat{f}(x_i)) + L^{'}(y_i, \hat{f}(x_i))\Delta f(x_i) + \frac{1}{2}L^{''}(y_i, \hat{f}(x_i))\Delta f(x_i)^2 \right\}.$$

- The central question XGB tries to answer is what are the $\Delta f(x_i)$ that we need to pick at every step.

XGB - Extreme Gradient Boosting

- Let $\hat{f}(x_i) = \hat{f}_i$, $g_i = L^{'}(y_i, \hat{f}_i)$ and $a_i = L^{''}(y_i, \hat{f}_i)$. And assume we want to use weak learners (trees) as the Δf_i. Then, at every iteration, we want to find $T(x) = \sum_{m=1}^{M} c_m 1(x \in R_m)$ that minimizes $\sum_{i=1}^{n} g_i T(x_i) + \frac{1}{2} a_i T(x_i)^2$.
- This is equal to $\sum_{m=1}^{M} c_m \sum_{i:x_i \in R_m} g_i + \frac{1}{2} c_m^2 \sum_{i:x_i \in R_m} a_i$. Let $\sum_{i:x_i \in R_m} g_i = G_m$, and $\sum_{i:x_i \in R_m} a_i = A_m$.
- To find the c_m for $m = 1...M$, M, and the regions R_m that minimize $\sum_{i=1}^{M} c_m G_m + \frac{1}{2} c_m^2 A_m$.

XGB - Extreme Gradient Boosting (Regularization)

- Also, in XGB, we explicitly penalize excess complexity, adding a penalty of the type $\gamma M + \frac{1}{2}\lambda \sum_{m=1}^{M} c_m^2$ to this problem (at each iteration). For any given M, the solution for each c_m is independent than that of the others. So we can find c_m by fixing M and solving:

$$c_m = argmin\{c_m G_m + \frac{1}{2} c_m^2 (A_m + \lambda)\}$$

XGB - Extreme Gradient Boosting (Computation)

- Taking the derivative and setting it to 0, we get:

$$G_m + c_m(A_m + \lambda) = 0$$

$$\Rightarrow c_m = -\frac{G_m}{A_m + \lambda}$$

- Plugging this into the last objective function mentioned above, we get that at the optimal, this function takes the value $-\frac{G_m^2}{A_m+\lambda} + \frac{G_m^2}{2(A_m+\lambda)} = \frac{-G_m^2}{2(A_m+\lambda)}$.
- At each iteration, we grow a tree to minimize $\sum_{m=1}^{M} -\frac{1}{2}\frac{G_m^2}{A_m+\lambda} + \gamma M$.

Summary for regularized learning

Summary for regularized learning:

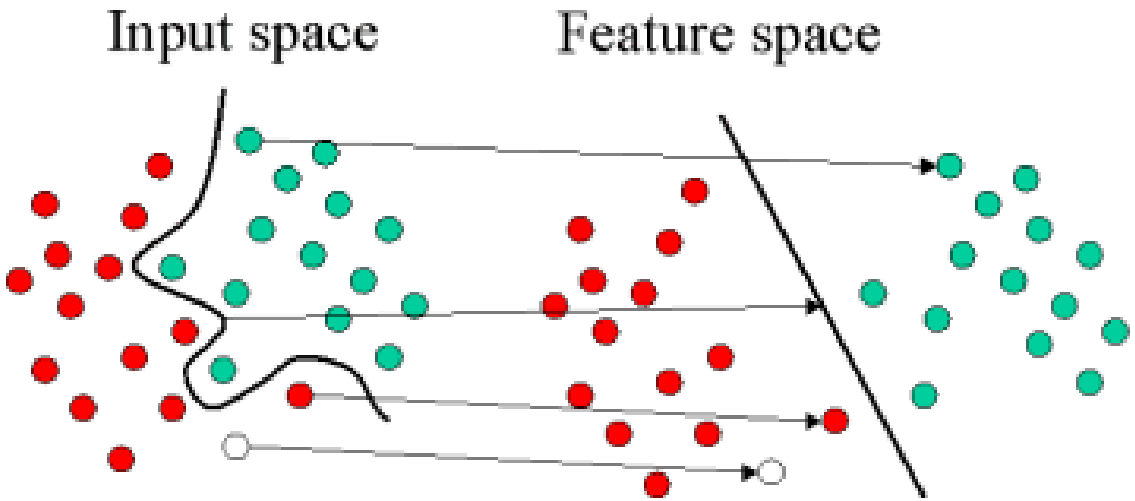
- Ridge regression
- Lasso regression
- Coordinate descent
- Spline regression
- Least angle regression
- Stagewise regression
- Bayesian regression
- Perceptron
- SVM
- Boosting

Roadmap for deep learning

- Neural network: multi-layer perceptrons
- Back-propagation: chain-rule calculation
- Rectified linear units and linear spline
- Stochastic gradient descent
- Convolutional network
- Residual and recurrent networks
- Long short term memory (LSTM)
- Transformer
- Supervised and unsupervised learning

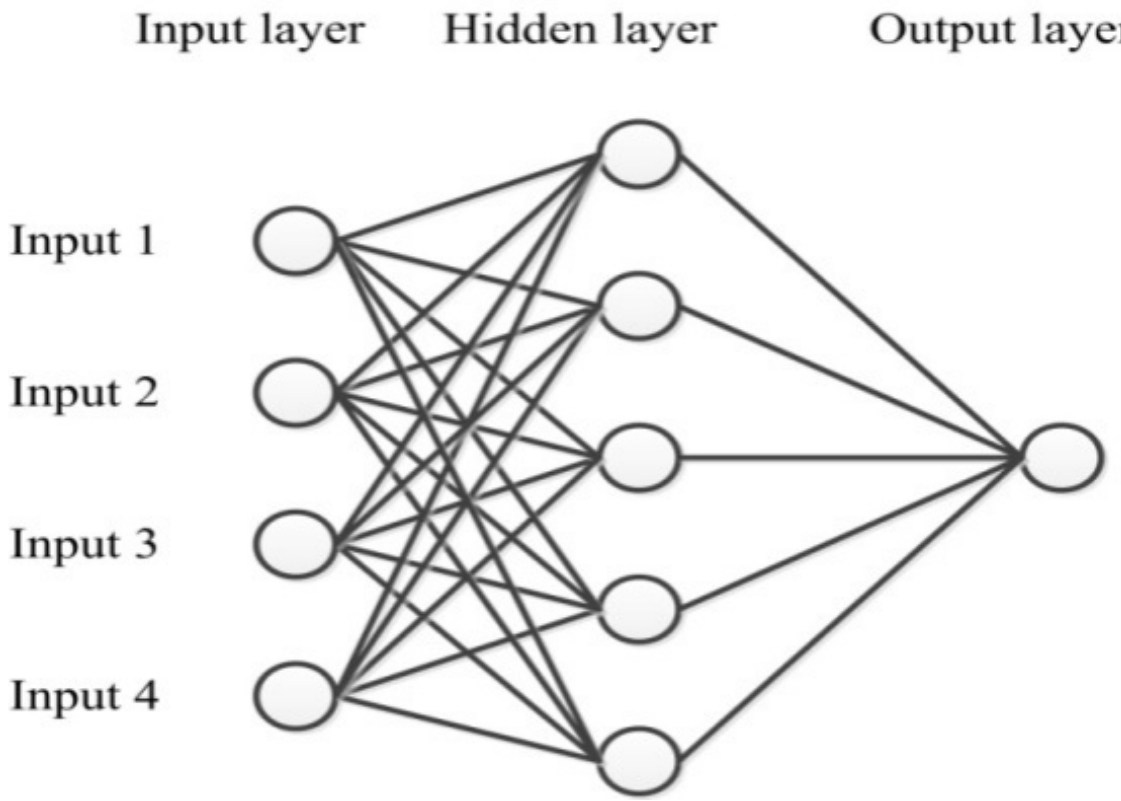
Perceptron

- A perceptron seeks to separate the positive examples and negative examples by projecting them onto β.
- If the data are not linearly separable, a perceptron cannot work. We may need to transform the original variables into some features so that they can be linearly separated.



Neural network: multi-layer perceptrons

One way to solve this problem is to generalize the perceptron into multi-layer perceptrons. This structure is also called feedforward neural network.



SVM/ Adaboost/ Tree can all be considered within this framework

Multi-layer perceptron

The neural network is logistic regression on top of logistic regressions. y_i follows a logistic regression on $h_i = (h_{ik}, k = 1, ..., d)^\top$, and each h_{ik} follows a logistic regression on $X_i = (x_{ij}, j = 1, ..., p)^\top$,

$$y_i \sim \text{Bernoulli}(p_i),$$

$$p_i = \sigma(h_i^\top \beta) = \sigma(\sum_{k=1}^{d} \beta_k h_{ik}),$$

$$h_{ik} = \sigma(X_i^\top \alpha_k) = \sigma(\sum_{j=1}^{p} \alpha_{kj} x_{ij}).$$

Multi-layer perceptron

obs	input	hidden	output
1	$X_1^\top$	$h_1^\top$	y_1
2	$X_2^\top$	$h_2^\top$	y_2
...			
n	$X_n^\top$	$h_n^\top$	y_n

Back propagation and chain rule

The log-likelihood is

$$l(\beta, \alpha) = \sum_{i=1}^{n} \left[y_i \sum_{k=1}^{d} \beta_k h_{ik} - \log[1 + \exp(\sum_{k=1}^{d} \beta_k h_{ik})] \right].$$

Back propagation and chain rule

The log-likelihood is

$$l(\beta, \alpha) = \sum_{i=1}^{n} \left[y_i \sum_{k=1}^{d} \beta_k h_{ik} - \log[1 + \exp(\sum_{k=1}^{d} \beta_k h_{ik})] \right].$$

The gradient is

$$\frac{\partial l}{\partial \beta} = \sum_{i=1}^{n} (y_i - p_i) h_i,$$

$$\frac{\partial l}{\partial \alpha_k} = \frac{\partial l}{\partial h_k} \frac{\partial h_k}{\partial \alpha_k} = \sum_{i=1}^{n} (y_i - p_i) \beta_k h_{ik} (1 - h_{ik}) X_i$$

$\partial l / \partial \alpha_k$ is calculated by chain rule. The gradient descent learning algorithm again learns from mistake or error $y_i - p_i$. The chain rule back-propagates the error in order for β and α to update.