# DATA SCIENCE II: Machine Learning MTH 9899 Baruch College

## Lecture 4: Boosting, Neural Network Gradient Descent Learning

Adrian Sisser

April 26, 2017

## Outline

## Outline

## Boosting

Boosting is the idea of combining a set of weak learners to form a stronger learner.
It a type of ensemble, since it’s a collection of learners, but is different in many important aspects.
Boosting differs because it trains the learners in sequence, with a focus on errors made by the prior learners.

## AdaBoost

AdaBoost was the first successful implementation of boosting. It applies to a binary classification problem:

$$Y = -1, +1$$

We will learn weak learners, with a goal of combining them all into a final learner:

$$\begin{aligned} f_j &: X \rightarrow -1, +1 \\ F_i(x) &= \sum_{j=1}^{i} \beta_j f_j(x) \end{aligned}$$

## AdaBoost

```
function TrainAdaBoost($(X, Y)$)
    $W_1(j) \leftarrow \frac{1}{N}$                          ▷ $N$ is the number of points in $X$
    $F_0(X) = 0$
    for $i \in 1..T$ do                                  ▷ $i$ indicates the Boosting Round
        $f_i \leftarrow$ TrainWeakLearner($(X, Y, W_i)$)
        $\epsilon_i \leftarrow \sum_j W_i(j) I(Y_j \neq f_i(x_j))$
        $\alpha_i \leftarrow \frac{1}{2} \ln \frac{1-\epsilon_i}{\epsilon_i}$
        $W_{i+1}(j) \leftarrow W_i(j) \exp(-\alpha_i y_j f_i(x_j))$
        $W_{i+1}(j) \leftarrow \frac{W_i(j)}{\sum_k W_i(k)}$   ▷ Normalize the weights to sum to 1
        $F_i(X) \leftarrow F_{i-1}(X) + \alpha_i f_i(X)$
    end for
    return $F_T$
end function
```

So now we ask, where did those update rules come from? Let's define an error function for $F_j$:

$$\begin{aligned}
E &= \sum_{i=1}^{N} e^{-y_i F_j(x_i)} && \text{Exponential Loss} \\
&= \sum_{i=1}^{N} e^{-y_i (F_{j-1}(x_i) + \alpha_j f_j(x_i))} && \text{Using the def of } F_j \\
&= \sum_{i=1}^{N} e^{-y_i F_{j-1}(x_i)} e^{-y_i \alpha_j f_j(x_i)} \\
&= \sum_{f_j(x_i)=y_i} e^{-y_i F_{j-1}(x_i)} e^{-\alpha_j} + \sum_{f_j(x_i)\neq y_i} e^{-y_i F_{j-1}(x_i)} e^{\alpha_j} && \text{Split up into right/wrong examples} \\
&= \sum_{i=1}^{N} e^{-y_i F_{j-1}(x_i)} e^{-\alpha_j} + \sum_{f_j(x_i)\neq y_i} e^{-y_i F_{j-1}(x_i)} \left(e^{\alpha_j} - e^{-\alpha_j}\right)
\end{aligned}$$

Now, we want to minimize the expression from before:

$$E = \sum_{i=1}^{N} e^{-y_i F_{j-1}(x_i)} e^{-\alpha_j} + \sum_{f_j(x_i) \neq y_i} e^{-y_i F_{j-1}(x_i)} \left(e^{\alpha_j} - e^{-\alpha_j}\right)$$

In the algo, we said:

$$\begin{aligned} W_{j+1}(i) &= W_j(i) e^{-\alpha_j y_i f_j(x_i)} \\ &= W_{j-1}(i) e^{-\alpha_{j-1} y_i f_{j-1}(x_i)} e^{-\alpha_j y_i f_j(x_i)} \\ &= e^{-y_i F_{j-1}(x_i)} \end{aligned}$$

By telescoping the prior statement ...

So, we can see that the weight is the same as the only term in the equation above that is dependent on the learner we're trying to optimize in this iteration, $f_j$. Given that, we try to minimize the sum of the weights of the incorrect points.

Let's look at how we choose $\alpha$, the weight we use when we add the new learner $f_j$ to the ensemble. We'll start with an earlier vesion of the equation, since it will make solving it easier.

$$
\begin{aligned}
E &= \sum_{f_j(x_i)=y_i} e^{-y_i F_{j-1}(x_i)} e^{-\alpha_j} + \sum_{f_j(x_i)\neq y_i} e^{-y_i F_{j-1}(x_i)} e^{\alpha_j} \\
\frac{\partial E}{\partial \alpha_j} &= -\sum_{f_j(x_i)=y_i} e^{-y_i F_{j-1}(x_i)} e^{-\alpha_j} + \sum_{f_j(x_i)\neq y_i} e^{-y_i F_{j-1}(x_i)} e^{\alpha_j} \\
0 &= -\sum_{f_j(x_i)=y_i} e^{-y_i F_{j-1}(x_i)} + e^{2\alpha_j} \sum_{f_j(x_i)\neq y_i} e^{-y_i F_{j-1}(x_i)} \\
\alpha_j &= \frac{1}{2} \ln \frac{\sum_{f_j(x_i)=y_i} e^{-y_i F_{j-1}(x_i)}}{\sum_{f_j(x_i)\neq y_i} e^{-y_i F_{j-1}(x_i)}} \\
\alpha_j &= \frac{1}{2} \ln \frac{1-\epsilon_j}{\epsilon_j}
\end{aligned}
$$

We’ve now seen what AdaBoost is really doing. It’s a greedy optimization at each iteration - where we train a learner to best fit the weighted points, and the weight of points increases as they are misclassified.

## Outline

Gradient Boosting is another boosting technique. It's applicable to problems like regression. We'll take the same approach as we did with AdaBoost. At each step, we'll add a new weak learner into the overall predictor, and find the optimal weight for it.

- You can use any weak learner, typically, we use Regression Trees
- Like other boosting algorithms, it can be slower than a normal Random Forest, because it can't be parallelized.

```
function TRAINGB($X, Y$)
    $F_0(X) = 0$
    for $i \in 1..T$ do
        $r_i \leftarrow -\frac{\partial E}{\partial F_{i-1}}$
        $f_i \leftarrow$ TRAINWEAKLEARNER($X, r_i$)
        $\alpha_i \leftarrow \arg\min_\alpha E$
        $F_i(X) = F_{i-1}(X) + \alpha_i f_i(X)$
    end for
    return $F_T$
end function
```

Let's look in more detail how this is the same as AdaBoost. In AdaBoost, our loss function was $e^{-y_i F(x_i)}$ and $E = \sum_{i=1}^{N} e^{-y_i F_j(x_i)}$. Since

$$r_j = -\frac{\partial E}{\partial F_{j-1}}$$

$$r_j = \sum_{i=1}^{N} y_i e^{-y_i F_{j-1}(x_i)}$$

$$r_j = \sum_{i=1}^{N} y_i W_j(i)$$

From earlier slide

So, we can see that $r_j$ is just the weighted inputs from earlier.

What happens in the case of regression? Let's define an error function:

$$E = \sum_{i=1}^{N} \frac{1}{2}(y_i - F_j(i))^2$$

Now let's calculate $r_j$:

$$\begin{aligned} r_j &= -\frac{\partial E}{\partial F_{j-1}} \\ &= (y_i - F_j(i)) \end{aligned}$$

For GB

So for regression, $r_j$ is just the residual!! Now, we just have to fit learners at each iteration to our residual, and add them in.

To add in a new residual, we need to find $\alpha$, the weight.

$$E = \sum_{i=1}^{N} \frac{1}{2}(y_i - F_{j-1}(i) - \alpha_j f_j(i))^2$$
$$\frac{\partial E}{\partial \alpha_j} = \sum -f_j(i)(y_i - F_{j-1}(i) - \alpha_j f_j(i))$$

There is no easy way to solve this. Instead, in a general Gradient Boosting algorith, the $\alpha$ value is effectively a a learning rate - another hyperparameter that we have to tune.

## Outline

Last week, we talked about Extra Trees as one of the best 'off the shelf' modelling options. **XGBoost** is an open source package that is also, arguably, the best stock ML algo. XGBoost is short for Extreme Gradient Boosting and has a few interesting features:

- Very well implemented - FAST
- Distributed - It can be split up across a cluster.
- Custom Loss Functions

XGBoost takes care to reduce overfitting by regularizing - they add a penalty to avoid growing overly complex trees.

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

Here $\gamma$ and $\lambda$ are hyper parameters, $T$ is the number of leaves, and $w_j$ is the value in each leaf of the tree. The first component simple penalizes tree size. The second part is more interesting: We can think of it as penalizing us for vastly different values in the leaves - so we need a significant improvement in loss to justify it.

The next interesting technique in XGBoost is how they model the loss given a new function $f_j$ to add in. We can take a taylor expansion of the loss:

$$\mathcal{L}(Y, Y_j) = \mathcal{L}(Y, Y_{j-1}) + \frac{\partial \mathcal{L}}{\partial Y_{j-1}} + \frac{1}{2}\frac{\partial^2 \mathcal{L}}{\partial Y_{j-1}^2}^2$$

$$\mathcal{O} = \mathcal{L}(Y, Y_{j-1}) + \frac{\partial \mathcal{L}}{\partial Y_{j-1}} + \frac{1}{2}\frac{\partial^2 \mathcal{L}}{\partial Y_{j-1}^2}^2 + \Omega$$

Now, when we want to determine if we should add a new layer to the tree, we can see if the new split, will increase or decrease the overall objective based on just the first and second derivatives.

## Basic of Optimization

It's such an important topic, let's talk about opimization again. We are trying to find the parameter set, $\Theta$ (including weights, biases, any other 'meta params' you might define), that minimizes the total error. Our error surface is FAR from convex. This means we are usually going to finding **local** optima - we have to be wary of optimization techniques that don't introduce enough variance to help find better local optima.

## Basic Gradient Descent

The most basic form of Gradient Descent involves calculating the gradient of the error with respect to the entire dataset, but this has a few drawbacks:

- If your datasize is too big, it might not work at all, ie won't fit in memory.
- It can be incredibly inefficient. Your using all of your points to calculate a gradient, but you might have been able to figure out the step direction much faster.
- By using all the data to calculate each step, we have very little variance in our step directions. This makes finding a near global optimum harder.

## Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) attempts to avoid the problems from before by calculating the gradient and updating the params on a row by row. We will choose the order randomly, thus the term stochastic.

- By using one row at a time, an outlier can cause a step in a wildly different direction than you want to be going in.
- Convergence take a long time
- The high variance of the step directions makes it easier to get out of local optima
- It's inefficient - computers are very good at performing bulk computations (especially GPUs) - operating on 1 row at a time is inherently slow.

## Minibatches

Let’s do the the obvious choice: Mix the 2.
Minibatches refers to training on randomly batches of the randomly shuffled data.

- By choosing a good batch size we can manage a trade-off between variance (ie the ability to get out of local minima) and the speed of training
- Choosing batch size isn’t easy, usually we use a few hundred examples at a time.

Note that in practice, when people say SGD they usually mean SGD with Minibatches.

## Outline

## Overview

A wide variety of trick and techniques have been developed to help speed up convergence in a few ways:

- Managing the learning rate
- Introducing a ’Memory’ of recent steps
- Hetergenous steps across different features.

Our original update rule was:

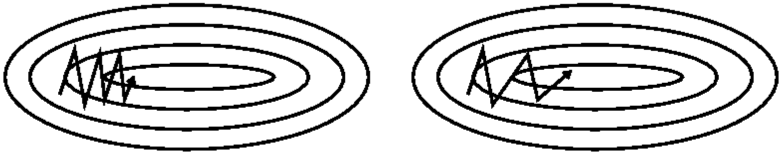
$$\theta_{i+1} = \theta_i - \eta \frac{\partial E}{\partial \theta_i}$$

With Momentum, we blend the update for this step with the update from a previous step:

$$\theta_{i+1} = \theta_i - (\eta \frac{\partial E}{\partial \theta_i} + \gamma(\theta_i - \theta_{i-1}))$$

By doing this, we avoid the unnecessary variance induced by asymmetric error surfaces.

## Algorithms Built on Gradient Descent



[1] Source:
http://sebastianruder.com/optimizing-gradient-descent/index.html

## Nesterov Momentum

Nesterov Momentum is based off a simple observation about the basic momentum algorithm. From basic momentum, we know:

$$\theta_{i+1} = \theta_i - (\eta \frac{\partial E}{\partial \theta_i} + \gamma(\theta_i - \theta_{i-1}))$$

This means that we already know a large part of the update for the next step, so we can calculate the gradient starting from there, instead of the current weights:

$$\theta_{i+1} = \theta_i - (\eta \frac{\partial E(\theta_i - \gamma(\theta_i - \theta_{i-1}))}{\partial \theta_i} + \gamma(\theta_i - \theta_{i-1}))$$