DATA SCIENCE II: Machine Learning MTH 9899 Baruch College

Lecture 5: Genetic Algorithms and Optimization

Adrian Sisser

May 3, 2017

Outline

1 Optimization
- Genetic Algorithms

2 Neural Networks
- Optimizers
- Activation Functions

Optimization

Optimization is an important part of ML.

- Many of the techniques we've talked involve iterative algorithms to solve: k-Means, Regression Trees, Forward/Backward Selection, Neural Networks, etc.
- Regression Trees (at least the recursive partitioning approach that we've talked about) take a greedy approach to solving the problem of splitting - this isn't globally optimal.
- k-Means and EM are both sensitive to starting points.

Are there any ways to try to find globally optimal solutions?

Outline

1 Optimization
- Genetic Algorithms

2 Neural Networks
- Optimizers
- Activation Functions

GAs - An Overview

Genetic Algorithms (GAs), are an attempt to mimic a biological system. They borrow heavily from biology for terminology and concepts.

Chromosome	A representation of a potential solution.
Gene	A particular parameter value on a chromosome.
Genotype	A candidate solution's underlying chromosomes.
Phenotype	A candidate solution's 'appearance'. Often the same as the genotype.
Mutation	A 'random' change to a candidate solution that might or might not improve it.
Fitness	A measure of a candidate solution's performance, ie R^2 for regression.
Generation	A set of candidate solutions that exist and 'compete' together.

High-level Algorithm

So what do we do? A very rough overview:

- Create an initial generation of n, possible solutions.
- Test the fitness of these candidate solutions, and pick the best j.
- Take these j solutions, and create many more variants, by 'mutating' them.
- Go back to the 'fitness test', and repeat this process until you have a 'good' solution.

Mutations

So how do we mutate things?

Crossover 'Swap' parts of two chromosomes by switching adjoining sections of 2 chromosomes.

Point Mutation Change a given gene's value randomly, in hopes it will produce a better solution.

A Practical Application

One simple example of a GA in practice is variable selection in regression. We've looked at a few techniques for solving this:

- Forward Selection - Start with an empty set and add in the most relevant variables
- Backward Selection - Start with a full set and remove the least relevant variables
- Lasso - Penalize the norm of β to reduce the complexity.

In practice, these all have significant issues. Can we use a GA to try and solve this?

Modeling Variable Selection

What's a reasonable way to model this problem? We can create a simple bit-string, representing whether or not each variable is included in the regression.

$$\begin{aligned} C_i &= b_0, b_1, ..., b_F \\ \text{where} \quad & b_i \in 0, 1 \\ & \text{and } b_i = 1 \text{ means feature } i \text{ is turned on} \end{aligned}$$

The Parameters

There are a lot of ’knobs’ to turn in a Genetic Algorithm:

- Mutation Probabilities - the probability of a point mutation or a cross-over
- Population Sizes - A larger population allows weaker members to stay around, and might improve overall results
- Mutation Types - The ’geometry’ of cross-over mutations
- Generaion Selection - In the example, we went with a simple top n selection. More advanced techniques do things like limit the number of children that can survive.

Applications

There are a lot of novel GA based applications out there:

- **NEAT** - NeuroEvolution of Augmenting Topologies - A technique to try and evolve NN topologies
- Neural Networks - Can use GA for training rather than backpropagation
- KMeans - We can try crossing over various set memberships to see if we can get more stable long-term solutions.
- Regression Trees - The evtree package in R, implements GA evolution of a regression tree

Outline

1 Optimization
- Genetic Algorithms

2 Neural Networks
- Optimizers
- Activation Functions

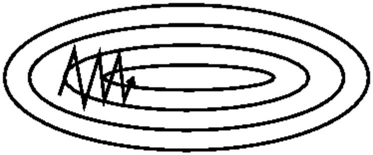
Our original update rule was:

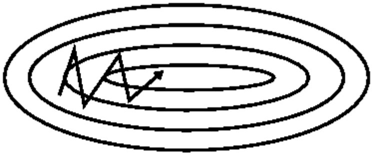
$$\theta_{i+1} = \theta_i - \eta \frac{\partial E}{\partial \theta_i}$$

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.





[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:

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

AdaGrad

AdaGrad is another optimization technique, based on the idea that we don't update all parameters at the same rate. It also accounts for the fact that we want the learning rate to decrease over time.
First, let's define the sum of the squared gradients over time, G_i will be a diaganol matrix:

$$G_T = \operatorname{diag} \sum_{j=1}^{T} (\frac{\partial E}{\partial \theta_j})^2$$

Now, we learn with this, where ϵ is a small constant to avoid dividing by zero:

$$\theta_{i+1} = \theta_i - \frac{\eta}{\sqrt{G_T + \epsilon}} \odot \frac{\partial E}{\partial \theta_i}$$

What are the drawbacks?

RMSProp

RMSProp is a variant of AdaGrad (similar to AdaDelta, which we will omit for time considerations) developed by Hinton. We basically replace the sum of squared gradients from AdaGrad with an EMA (Exponentially Moving Average) version.

$$G_i = \lambda G_{i-1} + (1-\lambda)(\frac{\partial E}{\partial \theta_i})^2$$

$$\theta_{i+1} = \theta_i - \frac{\eta}{\sqrt{G_i + \epsilon}} \circ \frac{\partial E}{\partial \theta_i}$$

Adam

Adam is short for Adaptive Momentum Estimation that extends RMSProp and momentum. Remember that momentum kept a simple EMA of gradient while RMS Prop kept an EMA of the gradient squared.

$$m_T = \lambda_m m_{T-1} + (1-\lambda_m)\frac{\partial E}{\partial \theta_T}$$

$$G_T = \lambda_g G_{T-1} + (1-\lambda_g)(\frac{\partial E}{\partial \theta_T})^2$$

These exponential moving averages are very biased initially, so Adam corrects them:

$$m_T^* = \lambda_m m_{T-1} + (1-\lambda_m)\frac{\partial E}{\partial \theta_T}$$

$$G_T^* = \lambda_g G_{T-1} + (1-\lambda_g)(\frac{\partial E}{\partial \theta_T})^2$$

Adam takes these two updates, and combines them similarly to AdaGrad:

$$\theta_{i+1} = \theta_i - \frac{\eta}{\sqrt{G_i} + \epsilon} m_i$$

Outline

1 Optimization
- Genetic Algorithms

2 Neural Networks
- Optimizers
- Activation Functions

So far, we've mostly spoken about the sigmoid activation function, as it's the 'original' and has nice derivative properties:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$
$$\frac{d\sigma}{dx} = \sigma(x)(1-\sigma(x))$$

Output Activation Functions For Classification

For classification outputs, we need a special activation function. The most popular choice is the Softmax activation:

$$\mathcal{A}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$$

This converts the output to a PDF across the different class of outputs.

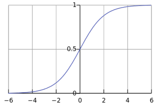
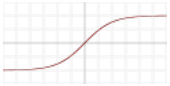
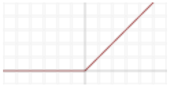
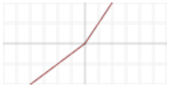
What are the requirements for a good activation function:

- Non-Linear - otherwise we're just fitting a linear model.
- Differentiable? - Not exactly - it just needs to be subdifferentiable
- Continuous? - Not needed either.
- Not easily saturated..

Does a step function work?

$$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$$

No! It is subdifferentiable, but the gradient is 0 for most of it, so we can't make any progress with Gradient Descent.

Name	$\mathcal{A}(x)$	$\frac{d\mathcal{A}}{dx}$	
Sigmoid	$\sigma(x)$	$\sigma(x)(1-\sigma(x))$	
Tanh	$\tanh(x)$	$1-\tanh^2(x)$	
Rectified Linear Unit (ReLU)	$\max(x,0)$	$\begin{cases} 0 & \text{for} \quad x<0 \\ 1 & \text{for} \quad x\geq 0 \end{cases}$	
Parameterized ReLU	$\begin{cases} \alpha x & \text{for} \quad x<0 \\ x & \text{for} \quad x\geq 0 \end{cases}$	$\begin{cases} \alpha & \text{for} \quad x<0 \\ 1 & \text{for} \quad x\geq 0 \end{cases}$	

So why do we have 2 versions of ReLU?

ReLU

ReLU (and it's variants) have become extremely popular:

- PReLu helps the fact that we have a large area with 0 gradient in 'vanilla' ReLU - this means that we can get 'stuck' in this region and the neuron becomes useless.
- ReLu has a gradient of 1 in it's 'active' area. This means that if we build very deep networks, we don't have to worry about the gradient vanishing.

Vanishing Gradient

The “Vanishing Gradient Problem” is a well known one in training deep neural networks. The basic idea is that when you use a sigmoid activation function, for a very large area, the gradient is nearly 0. When we have a deep network, with many layers and we apply back propagation, by the time we get back to the early layers, the gradient is extremely small.

ReLU fixes this because the derivative when you are in the ’active’ region of the neuron is 1.

This has been a fundamental change in NN and enabled much deeper networks.

Vanishing Gradient

There are other ways to fix the Vanishing Gradient Problem.

- Other activation functions that don't become saturated.
- Training methods - we can train a network layer by layer in an unsupervised manner.
- Residual Network (ResNet) - We feed the original input plus the output of the previous hidden layer to each layer.