Goals for Today:

Fundamental Properties:

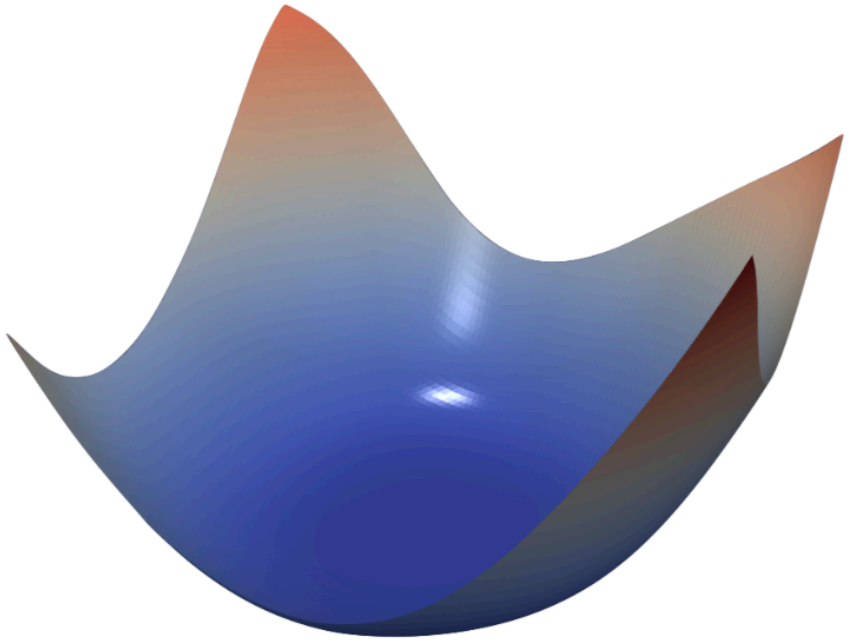
- Convexity
- Smoothness

Gradient Descent:

- Basic implementation
- Obvious drawbacks
- Learning Schedules

Types of Acceleration:

- Stochastic Gradient
- Momentum
- Adaptive Learning



What is optimization?

Find a set of parameters, β, that minimizes f(**x, y**, β): $\arg\min_{\beta} f(\mathbf{x}, \mathbf{y}, \beta)$

For example: in ordinary least squares, f(**x**, **y, β**) would be the mean squared error, **x** would be the set of predictor variables, **y** would be the known values, and **β** would be the set of coefficients and intercept that minimize f(**x, y**, **β**)**.**

For a narrow class of f(**x**, **y**, **β**) we can analytically solve for **β** (ordinary linear least squares and the common variants that we discussed last time), but for non-linear f(**x**,**y**,**β**) we need to use numerical optimization to find **β**.

Optimizers: these are the algorithms that we will use for finding the **β** that minimize our objective function. The most common algorithms involve following gradients, but this is still an active area of research with new specialized algorithms regularly appearing.

Objective function: this is the f(**x**, **y**, **β**) that we are trying to minimize. When this function refers to an error of some kind (e.g., mean squared error) it is sometimes also called a **loss function** and we will use the notation L(**x**, **y**, **β**).

What is optimization?

An important distinction: when discussing optimization, **β** will be reserved for the parameters that we are trying to determine and **x** will usually be considered fixed by the problem (e.g., the sensor values in the Dow dataset). If we were trying to find the minimum of a mathematical function (e.g., $f(x) = x^2 - e^x$) then the "**x**" value that minimizes f(x) would play the role of **β.**

Parameterization (Loss function minimization) :

$$\arg\min_{\beta} f(\mathbf{x}, \mathbf{y}, \beta)$$

vs.

Function minimization :

$$\arg\min_{\mathbf{x}} f(\mathbf{x})$$

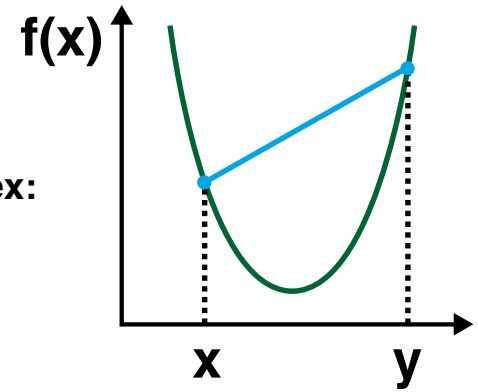
There is no fundamental distinction in the way these two problems are solved, other than that in loss function minimization, only a subset of the parameters are being optimized (**β**).

Convex vs. Non-Convex

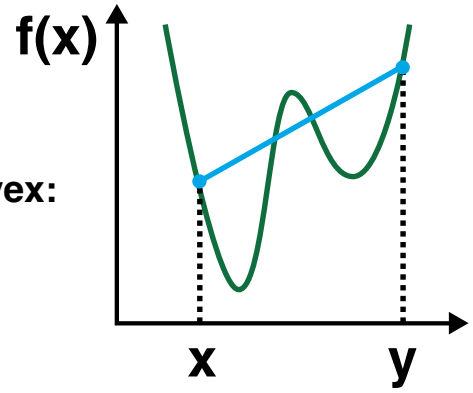
A function, f, is convex if for any **x** and **y** in $\mathbf{R}^n$ (Jensen's Inequality):

$$\forall \lambda \in [0,1],\ f\left(\lambda \mathbf{x} + (1-\lambda)\mathbf{y}\right) \leq \lambda f\left(\mathbf{x}\right) + (1-\lambda) f\left(\mathbf{y}\right)$$

convex:



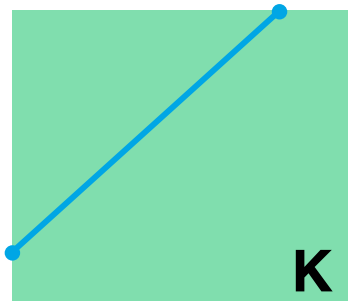
non-convex:



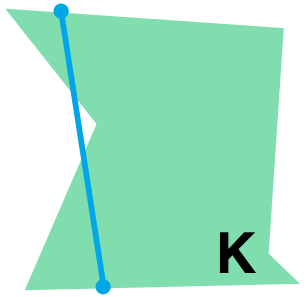
A set, K, is convex if for any **x** and **y** in K:

$$\forall \lambda \in [0,1],\ \lambda \mathbf{x} + (1-\lambda)\mathbf{y} \in K$$

convex:



non-convex:



A function, f, of a random variable **x** is convex if:

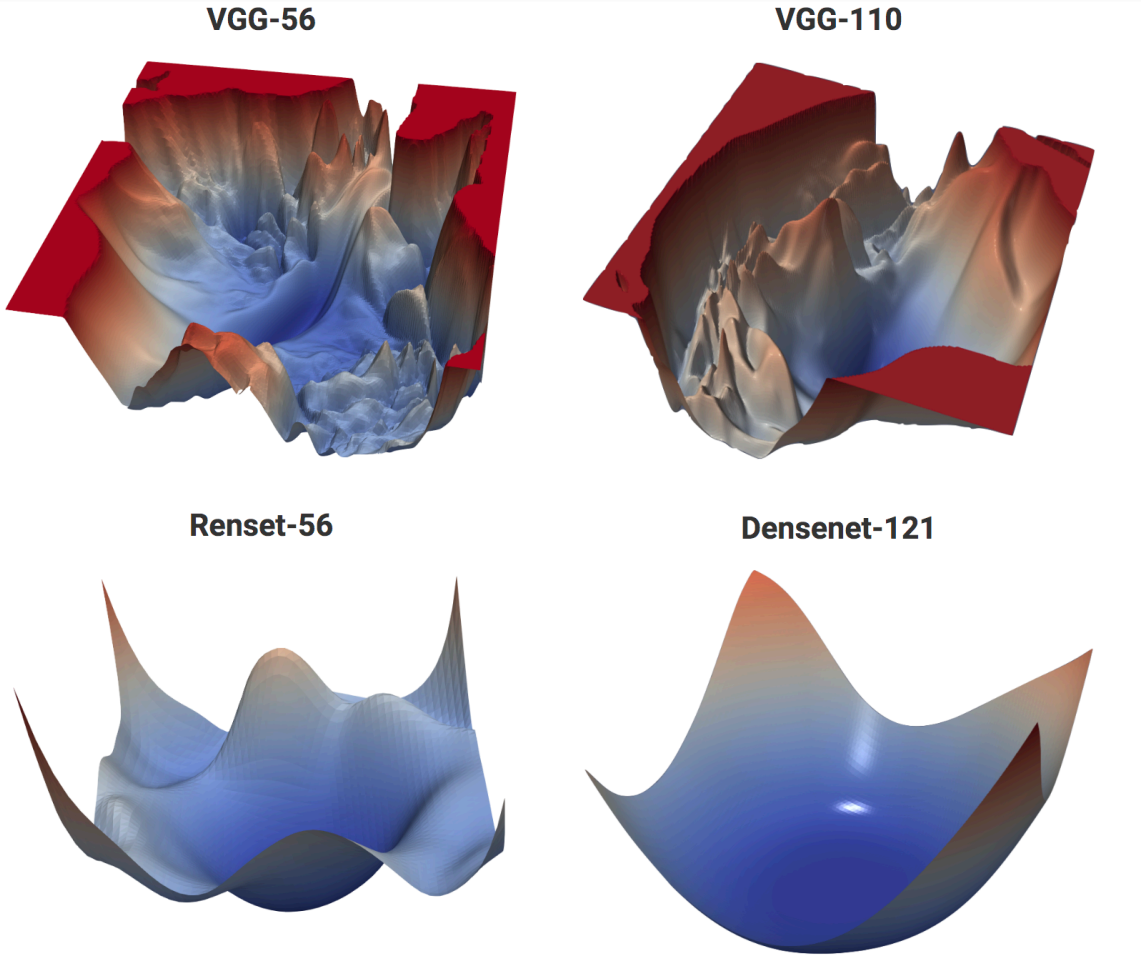
$$f\left(E\left[X\right]\right) \leq E\left(f\left[X\right]\right)$$

Complex models are typically non-convex

Which would you want to optimize?

Which performs the best?

A lot of deep learning research circulates around the fact that the two questions don't necessarily have the same answer.



3D projection of a high dimensional loss surface associated with 4 neural networks for image recognition tasks. The lower two are designed to have relatively better convexity and smoothness. Easier to optimize but not necessarily better. https://www.cs.umd.edu/~tomg/projects/landscapes/

Smooth vs. Non-Smooth

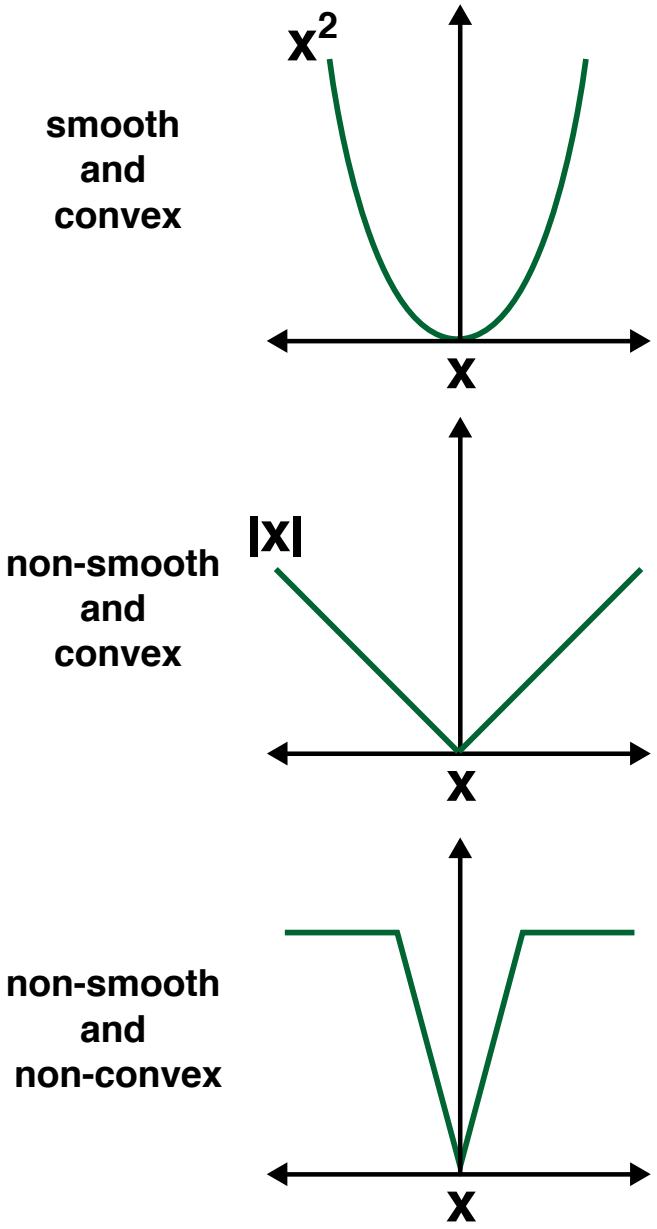
- For **smooth** functions, all differentials are continuous.

- For **non-smooth** functions at least one differential is discontinuous.

- **Lipschitz continuity** generalizes the degree of smoothness to continuous functions.

$$\frac{||f(\mathbf{x}_1) - f(\mathbf{x}_2)||}{||\mathbf{x}_1 - \mathbf{x}_2||} \leq K$$

- Functions that are non-smooth and non-convex are formally hopeless to optimize.



Types of Objective Functions

Convexity and smoothness are determined by both the model and loss function that we are trying to optimize.

We'll use the notation **L**() for a loss function, f() for our model, **x** for our features, **β** for our parameters, and **y** for our training data vector.

$$\mathbf{L}\left[f(\mathbf{x})\right]$$

e.g., function minimization

$$\mathbf{L}\left[f(\mathbf{x}), \mathbf{x}\right]$$

e.g., function minimization with constraints

$$\mathbf{L}\left[f(\mathbf{x}, \beta), \mathbf{y}\right]$$

e.g., least-squares optimization

Least squares with L_2 regularization example:

$$\mathbf{L}\left[f(\mathbf{x}, \beta), \beta, \mathbf{y}\right] = \frac{1}{N}\sum_{i}^{N}\left[f(\mathbf{x}_i, \beta) - \mathbf{y}_i\right]^2 + \alpha||\beta||_2$$

We seek **β*** such that:

$$\mathbf{L}\left[f(\mathbf{x}, \beta^*), \mathbf{y}\right] = \min\left(\mathbf{L}\left[f(\mathbf{x}, \beta), \mathbf{y}\right]\right)$$

How do we know if we have found a minimum?

$$||\nabla\mathbf{L}\left[f(\mathbf{x}, \beta^*), \mathbf{y}\right]|| = 0 \qquad ||\nabla^2\mathbf{L}\left[f(\mathbf{x}, \beta^*), \mathbf{y}\right]|| > 0$$

Numerically,

$$||\nabla\mathbf{L}\left[f(\mathbf{x}, \beta^*), \mathbf{y}\right]|| \leq \epsilon_1 \qquad ||\nabla^2\mathbf{L}\left[f(\mathbf{x}, \beta^*), \mathbf{y}\right]|| \geq \epsilon_2$$

For non-convex problems, **β*** is probably not a global minimum

Formulating an Optimization Problem

- Let's say we have a process with sensor readings for **n** variables, **x**, and binary labelled historical data, **y**, indicating when the process yielded a product that passed inspection (i.e., 1 or -1 for pass/fail). We've collected **t** samples (each containing **x,y**) that we want to use to train a model.

- Our goal is to find a linear separator, $\boldsymbol{\beta} \in \mathbf{R}^n$ (a hyperplane with **β** defining the normal) that divides conditions $\boldsymbol{x}_{pass}$ that yield 1 and $\boldsymbol{x}_{fail}$ that yield -1. If no clean linear separator exists, then we want to minimize the number of misclassifications. **What should we use for our loss function?**

We might use the dot product of the normal and **x**:

$$\mathbf{L}\left[f(\mathbf{x},\beta),\mathbf{y}\right] = \frac{1}{t}\sum_{i}^{t}\delta\left(\text{sign}\left[\beta^{\mathrm{T}}\mathbf{x}\right] \neq y_i\right)$$

This loss captures our definition, but it is **non-smooth** and **non-convex**.

An alternative, (mostly) smooth and convex loss function would be:

Hinge Loss

$$\mathbf{L}\left[f(\mathbf{x},\beta),\mathbf{y}\right] = \frac{1}{t}\sum_{i}^{t}\max\left\{0, 1 - \underbrace{y_i\beta^{\mathrm{T}}\mathbf{x}}\right\}$$

Correct classification yields (+)

Gradient Descent

The vanilla gradient descent algorithm is the parent of the more complicated algorithms that are used in this course:

$$\beta_{i+1} = \beta_i - \alpha \nabla_\beta \mathbf{L}$$

where, **β** is a vector holding the parameters of in the model
α is a scalar called the "learning rate" or "step-size"
L is the loss function
$\nabla_\beta \mathbf{L}$ is gradient of loss function with respect to **β**

For a typical supervised example:

$$\nabla_\beta \mathbf{L}\left[f(\beta, \mathbf{x}), \mathbf{y}\right] = \frac{1}{N}\sum_i^N \nabla_\beta \mathbf{L}\left[f(\beta, \mathbf{x}_i), \mathbf{y}_i\right]$$

Loss compares predictions against specific samples.

For function minimization:

$$\nabla_\beta \mathbf{L}\left[f(\beta, \mathbf{x}), \mathbf{y}\right] \rightarrow \nabla_\mathbf{x} f(\mathbf{x})$$

For root finding/function minimization, there are no samples, f(**x**) is itself the loss function (same distinction as we made earlier).

Problem 1: the gradient evaluation can be expensive for lots of samples

Problem 2: small α leads to slow convergence.

Problem 3: large α means oscillations/instability

Problem 4: neglects obvious sources of info (previous iterations, second derivative)

Stochastic Gradient Descent

Stochastic gradient descent speeds up gradient descent by approximating the gradient evaluation using a **single** randomly chosen sample out of N.

$$\underbrace{\beta_{i+1} = \beta_i - \frac{\alpha}{N}\sum_j^N \nabla_\beta \mathbf{L}\left[\beta_i, \mathbf{x}_j, \mathbf{y}_j\right]}_{\textbf{Gradient Descent}} \rightarrow \underbrace{\beta_{i+1} = \beta_i - \alpha \nabla_\beta \mathbf{L}\left[\beta_i, \mathbf{x}_j, \mathbf{y}_j\right]}_{\textbf{Stochastic Gradient Descent}}$$

- We want a cheap **unbiased estimator** for the gradient. We get the algorithm from the fact that $E(\boldsymbol{\nabla_\beta L}[f(\boldsymbol{\beta}, \mathbf{x}_j), \mathbf{y}_j]) = \boldsymbol{\nabla_\beta L}[f(\boldsymbol{\beta}, \mathbf{x}), \mathbf{y}]$ when j is chosen from a random uniform distribution across N samples.

- Further generalizing:

$$\underbrace{\beta_{i+1} = \beta_i - \frac{\alpha}{N}\sum_j^N \nabla_\beta \mathbf{L}\left[\beta_i, \mathbf{x}_j, \mathbf{y}_j\right]}_{\textbf{Gradient Descent}} \rightarrow \underbrace{\beta_{i+1} = \beta_i - \frac{\alpha}{M}\sum_j^{M<N} \nabla_\beta \mathbf{L}\left[\beta_i, \mathbf{x}_j, \mathbf{y}_j\right]}_{\textbf{Mini-Batch Gradient Descent}}$$

Mini-Batch Gradient Descent
(m=1, we recover SGD)

- SGD and mini-batch GD largely solve the issue of evaluating the gradient with respect to large numbers of samples and are still the default in many deep learning applications.

Problem 1: Still haven't solved the issue with choosing α.

Problem 2: Still neglects potential info (previous iterations, second derivative)

Learning Schedules

Take a moment and ask yourself what you might want α to do in a generic problem

Start high and end low?

This idea is commonly referred to as "Learning Rate Annealing" and has many flavors

1) 1/t Decay:

$$\alpha_t = \frac{\alpha_0}{1 + rt}$$

2) Exponential Decay:

$$\alpha_t = \alpha_0 e^{-rt}$$

3) Step Decay:
(0-indexing)

$$\alpha_t = \alpha_0 r^{\text{floor}\left(\frac{1+t}{d}\right)}$$

where, α_0 is an initial learning rate
r is a decay rate
d is decay interval in steps

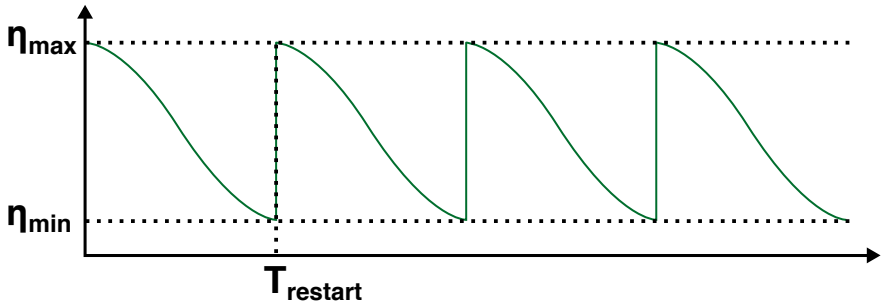
All achieve the qualitative high→low behavior necessary to heuristically increase convergence while permitting discovery of "deep" minima.

4) Warm Restarting:

$$\alpha_t = \alpha_{\min} + \frac{1}{2}\left(\alpha_{\max} - \alpha_{\min}\right)\left(1 + \cos\left[\frac{t_{i-0}\pi}{t_i}\right]\right)$$

t_{i-0} intervals since las restart

Heuristic for finding deeper and broader minima



Momentum Variants

The intuition behind momentum is that the previous gradient evaluation(s) potentially give us good information about the direction of the minimum. The analogy comes from the physics of an object rolling down the hill. It will start slowly, but if it is consistently downhill in the same direction, then the object "accelerates" toward the minimum. The change in algorithm is tiny but powerful:

$$\mathbf{m}_{i+1} = \eta \mathbf{m}_i - \alpha \nabla_\beta \mathbf{L}\left[\beta_i\right]^*$$
$$\beta_{i+1} = \beta_i + \mathbf{m}_{i+1}$$

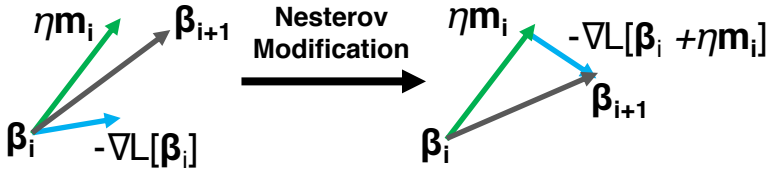
where, η is a new hyperparameter and we recover GD/SGD when η=0. *(typical values [0.5,0.9])*

*not showing x and y dependence

An even better variant developed by Nesterov calculates the gradient at $\boldsymbol{\beta}_i + \eta \mathbf{m}_i$ (i.e., *after* applying the momentum displacement). That is, on step i+1 we know the update from momentum alone ($\beta \mathbf{m}_i$) before we calculate the gradient–so why not calculate the gradient at that point, $\boldsymbol{\beta}_i + \eta \mathbf{m}_i$, since it is probably closer to $\boldsymbol{\beta}_{i+1}$ than $\boldsymbol{\beta}_i$ is?

$$\mathbf{m}_{i+1} = \eta \mathbf{m}_i - \alpha \nabla_\beta \mathbf{L}\left[\beta_i + \eta \mathbf{m}_i\right]$$
$$\beta_{i+1} = \beta_i + \mathbf{m}_{i+1}$$

Nesterov Momentum



Adaptive Learning Rates

"Adaptive" meaning parameter-specific learning rates that are dynamically updated during the optimization depending on the optimization surface.

AdaGrad (adaptive gradient algorithm, diagonal version)

$$\mathbf{A}_{i+1} = \mathbf{A}_i + \text{diag}\left(\nabla_\beta \mathbf{L}\right)^2$$

Diagonal matrix with cumulative square of the update derivatives

$$\beta_{i+1} = \beta_i - \frac{\alpha}{\sqrt{\mathbf{A}_{i+1}} + \text{diag}(\varepsilon)} \nabla_\beta \mathbf{L}$$

Large gradients get damped and small get accelerated

Note 1: Monotonic decrease in the learning rate makes it susceptible to early termination.

Note 2: eps is a small fixed value (e.g., 1E-4) to avoid division by zero.

Note 3: sqrt is actually important. What would be the problem with full matrix AdaGrad?

RMSProp (root mean square propagation)

$$\mathbf{A}_{i+1} = \eta \mathbf{A}_i + (1-\eta)\,\text{diag}\left(\nabla_\beta \mathbf{L}\right)^2$$

mixes in a fraction 1-η of the current gradients

$$\beta_{i+1} = \beta_i - \frac{\alpha}{\sqrt{\mathbf{A}_{i+1}} + \text{diag}(\varepsilon)} \nabla_\beta \mathbf{L}$$

Note 1: Averaging of the previous gradients addresses early stopping issues.

Note 2: η is a new hyperparameter that is typically fixed (e.g., 0.9,0.99,0.999 etc.)

Note 3: In principle, this addresses the early stopping issues of AdaGrad.

Extensions Under Active Research

AdaDelta: Uses an exponential average of past gradients for the update step.

Adam: Adds a sliding window to AdaGrad and a form of momentum

Shampoo: Full matrix AdaGrad is in principle better than the diagonal version, but square roots of non-diagonal matrices are expensive. Shampoo interpolates between full-matrix and diagonal AdaGrad.

AdaFactor: Shampoo + reduced memory requirements (good for really large neural networks)

GGT: Uses recent gradients to approximate full matrix AdaGrad

Extreme Tensoring: Approximates full-matrix AdaGrad using a low-rank tensor

Recommended Reading:

Introductory Lectures on Convex Optimization**, Yurii Nestorov** (2004)