Lecture 4:

Adaptive Learning

Week of January 23, 2023

University California, Berkeley

Machine Learning Algorithms

MSSE 277B, 3 Units

Spring 2023

Prof. Teresa Head-Gordon

Departments of Chemistry, Bioengineering, Chemical and Biomolecular Engineering

For conjugate gradients we constructed a new set of search directions such that we project out any component from the current gradient that has any overlap with previous line directions

$$d_{i+1}(\vec{x}) = -\nabla f(\vec{x}_{i+1}) + \frac{\nabla f(\vec{x}_{i+1})\nabla f(\vec{x}_{i+1})}{\nabla f(\vec{x}_i)\nabla f(\vec{x}_i)} d_i(\vec{x})$$

And we did this dynamically, updating each new direction sequentially, so that we need not solve for them all at once using Gram-Schmidt orthogonalization.

We completed our disciplined local optimization methods organized under the quadratic approximation to include Newton Methods

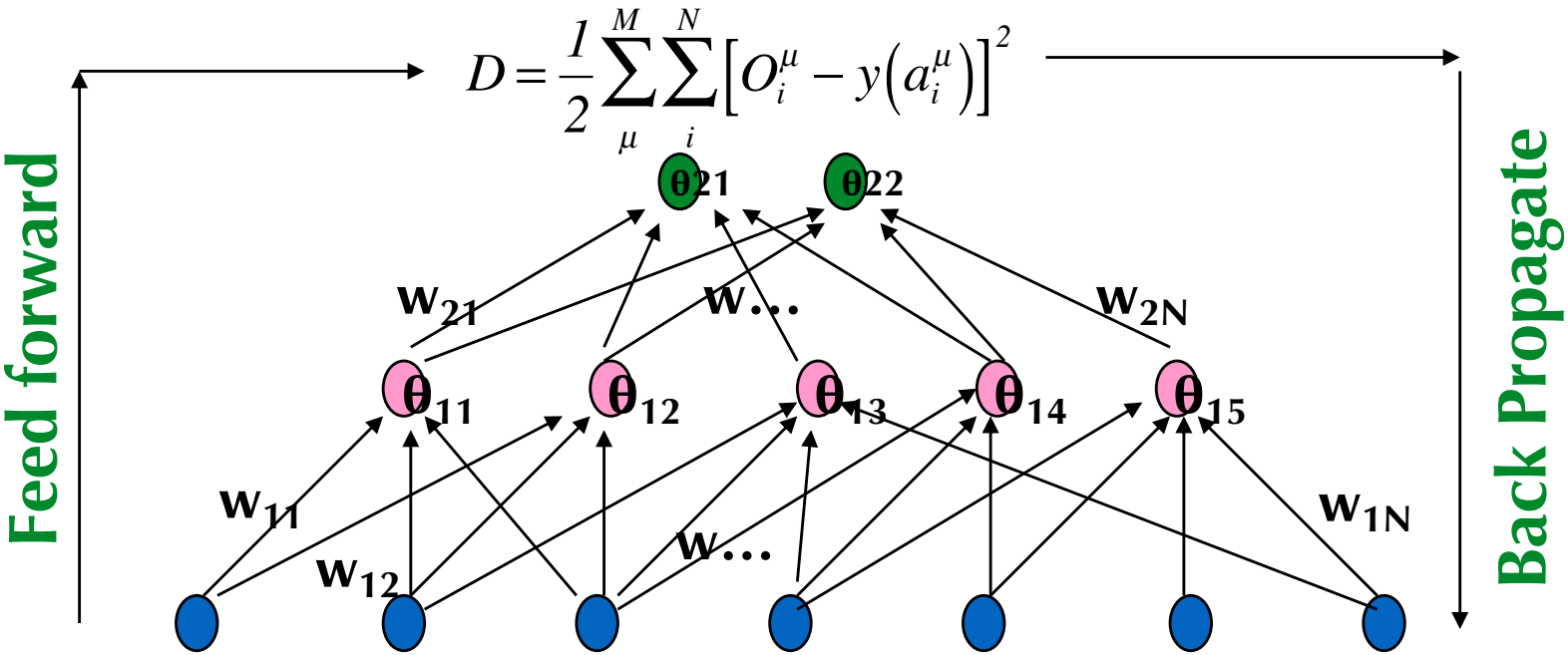
$$\Delta\vec{x} = \vec{x}_{i+1} - \vec{x}_i = -\underline{H}^{-1}(\vec{x}_i)\vec{\nabla} f(\vec{x}_i)$$

$$\vec{x}_{i+1} = \vec{x}_i - \underline{H}^{-1}(\vec{x}_i)\vec{\nabla} f(\vec{x}_i)$$

and Quasi-2nd order (DFP and BFGS) which approximated Hessian updates with function/gradient information. I.e. no explicit H matrix elements (DFP) or matrix inversion (BFGS) making especially the latter one of the most used local optimization method.

Review Lecture 3

Goals for Today's Lecture



Local optimization methods are used in machine learning to train network variables (weights, biases) to best reproduce the training data and to ultimately generalize to test or new data to make new predictions.

$$\delta w_{ij} = -\varepsilon\frac{\partial D}{\partial w_{ij}} = \varepsilon\sum_{\mu}\left[O_i^{\mu} - y\left(a_i^{\mu}\right)\right]\frac{dy}{da_i^{\mu}}\frac{\partial a_i^{\mu}}{\partial w_{ij}}$$

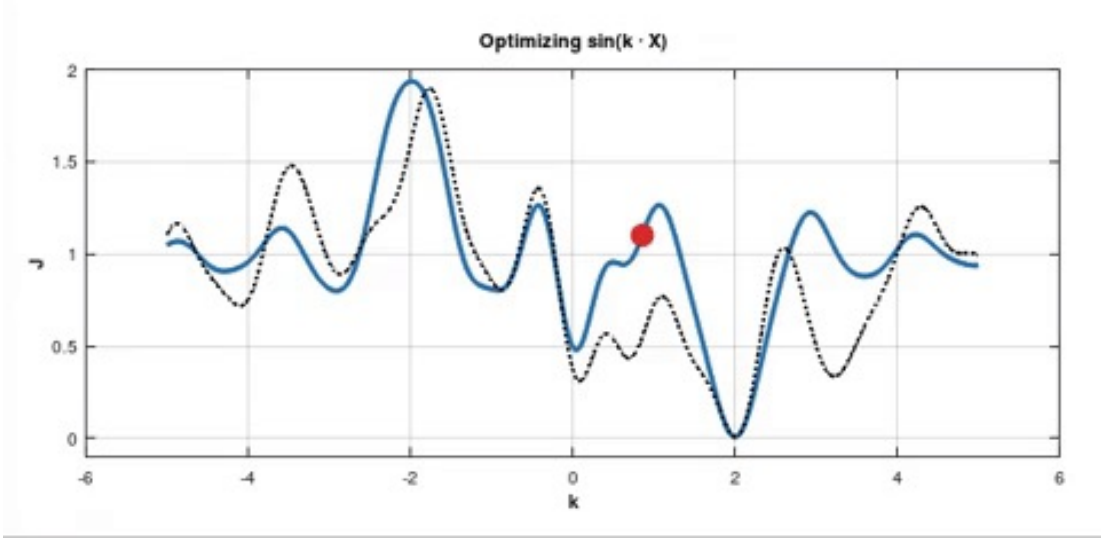
Adjusting weights to minimize the loss function, such as Mean Square Error, often looks like a convex quadratic optimization problem

Goals for Today's Lecture

Lecture 3 Purpose: We will also confront other loss functions that are specific to other types of regression or discrete classification problems, such as mean absolute error, hinge-loss, or cross-entropy.

In all cases the loss function is non-linear in the weights and it can be difficult to optimize. In addition, we optimize loss on training data but we predict on (different) test data

i.e. the objective function is noisy and the loss function has multiple minimum!



Therefore we adapt 1st order methods or consider aspects of (quasi-)2nd-order methods we introduced last lecture.

0th order method: uses function (energy) evaluations only

$$f(\vec{x}_0) \Rightarrow f(\vec{x})$$

Bisection and Golden Section (not often)

1st order method: uses function and derivative evaluations (mostly!)

$$f(\vec{x}_0) \Rightarrow f(\vec{x}), \vec{\nabla} f(\vec{x})$$

Stochastic Gradient Descents,
Momentum GD
conjugate gradients

2nd order method: uses function, derivative, Hessian evaluations

$$f(\vec{x}_0) \Rightarrow f(\vec{x}), \vec{\nabla} f(\vec{x}), \underline{\underline{H}}(\vec{x})$$

Newton Method (rarely but….!)

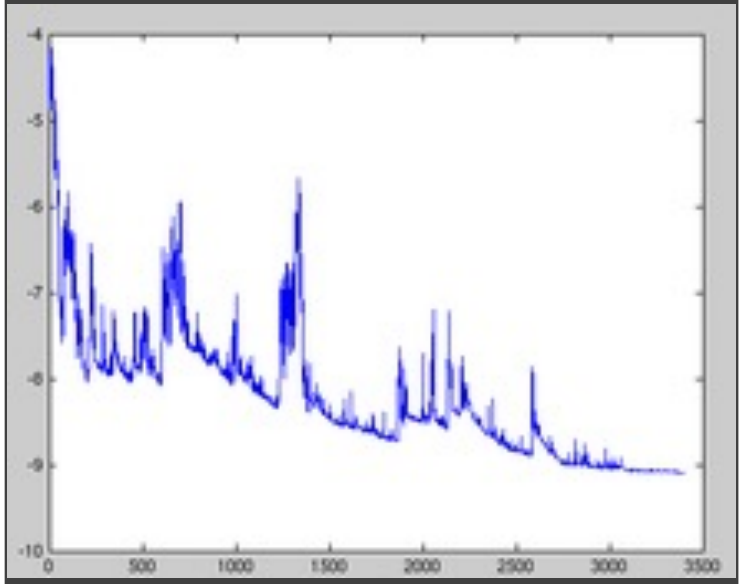
Quasi-2nd order method: uses function, derivative, approximate Hessian

Davis, Fletcher, Powell (DFP)
Broyden, Fletcher Goldstein, Shanno (BFGS – sometimes!)

$$f(\vec{x}_0) \Rightarrow f(\vec{x}), \vec{\nabla} f(\vec{x}), \underline{\underline{\approx H}}(\vec{x})$$

Classification Scheme: Local Optimization

Stochastic Gradient Descent



Machine learning relies on data -i.e. examples!

$$\text{MSE} = -\frac{1}{2}\sum_{i}^{M_{examples}}\sum_{j}^{N_{weight}}\left[O_j^i - y(w_j^i)\right]^2$$

in which the mean squared error (MSE) function requires that all mapping examples be presented to take a proper weight derivative, and to formulate a proper gradient for the search direction.

- Due to nature of ML and practical memory requirements, gradient descent (GD) corresponds to formulating a gradient from just a handful of randomly chosen examples (one or mini-batch) and thus gradient for the total MSE loss function is “noisy”, or is stochastic.

- Stochastic gradient descent (SGD), adding numeical noise to the gradient, reflects the problem structure. SGD’s fluctuations enables it to jump to new and potentially better local minima, and It experimentally improves “generalization” to an example never seen before

- But otherwise SGD is just steepest or gradient descent, often with a fixed and usually not adaptable step size λ which is now called the “learning rate”.

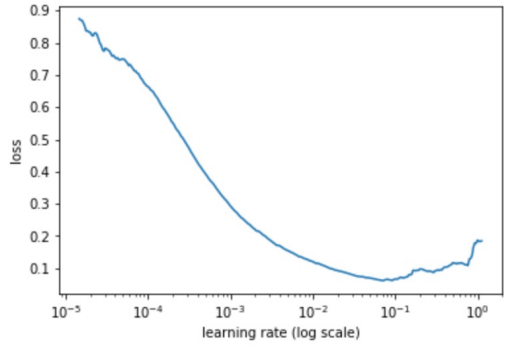
Stochastic Gradient Descent

Let's emphasize that λ is not solving line minimization as we have seen previously by giving it a new symbol

$$\lambda \rightarrow \eta$$

The ML term is "hyper-parameter", and SGD requires an optimization of learning rate η. In this case η is a hyper-parameter that controls how much we are adjusting our network variable with respect to the loss gradient.

Loss function as we vary $\log(\eta)$: the loss function decreases as learning rate increases, until a point where the loss stops decreasing and starts to increase (and we overshoot the minima).



The weight variables are updated via multiple mini-batches and updated "through time" t

$$\vec{w}_{t+1} = \vec{w}_t + \eta d_t(\vec{w})$$

where $d_t(\vec{w})$ is a search direction at time t (gradient direction, conjugate gradient direction, etc)

Improving upon Stochastic Gradient Descent

The question is how can we improve upon SGD? We can do three things:

(1) change the learning rate in some way so that it is adaptable to new information or as "time" proceeds,

(2) include information about previous search directions, or

(3) both.

As before, Newton's method will be our organizing mathematical structure which is based on the quadratic approximation (and continued update of Hessian until convergence for non-quadratic).

When $\eta = 1$ Newton has a faster convergence rate (fewer steps t) at more cost per step (trade-off).

$$\vec{w}_{t+1} = \vec{w}_t - \eta \underline{\underline{H}}^{-1}(\vec{w}_t)\vec{\nabla} f(\vec{w}_t)$$

Methods like BFGS *approximates* $\underline{\underline{H}}^{-1} \sim \underline{\underline{D}}^{-1}$ to avoid cost burden of Hessian and its inverse but gains close to convergence rate of a Newton step.

This suggests a general rule for ML optimization: find a "preconditioner" $\underline{\underline{D}}^{-1}$ that improves performance of gradient descent, but without devoting too much cost to compute it.

$$\vec{w}_{t+1} = \vec{w}_t - (\eta \underline{\underline{D}}^{-1}(\vec{w}_t))\, \vec{\nabla} f(\vec{w}_t)$$

We can see that learning rate can be viewed as "adaptable" where

$$\eta' = \frac{\eta}{\underline{\underline{D}}(\vec{w}_t)}$$

And thus learning rate η' becomes sensitive to individual weights, w_i, through the $\underline{\underline{D}}(\vec{w}_t)$ matrix elements which changes in "time" as we learn.

This introduces the first ML optimization algorithm AdaGrad. AdaGrad uses a different learning rate for each w_{i} at each update by defining a specific $\underline{\underline{D}}(\vec{w}_t)$.

Stochastic Gradient Descent and Adaptable Learning

AdaGrad

Let's define the search direction as the steepest descent direction

$$d_t(w_{\mathrm{t,i}}) = -\nabla_i f(\vec{w}_{t,i})$$

i.e. the gradient of loss function w.r.t. to parameter w_i at time step t.

AdaGrad updates weights according to

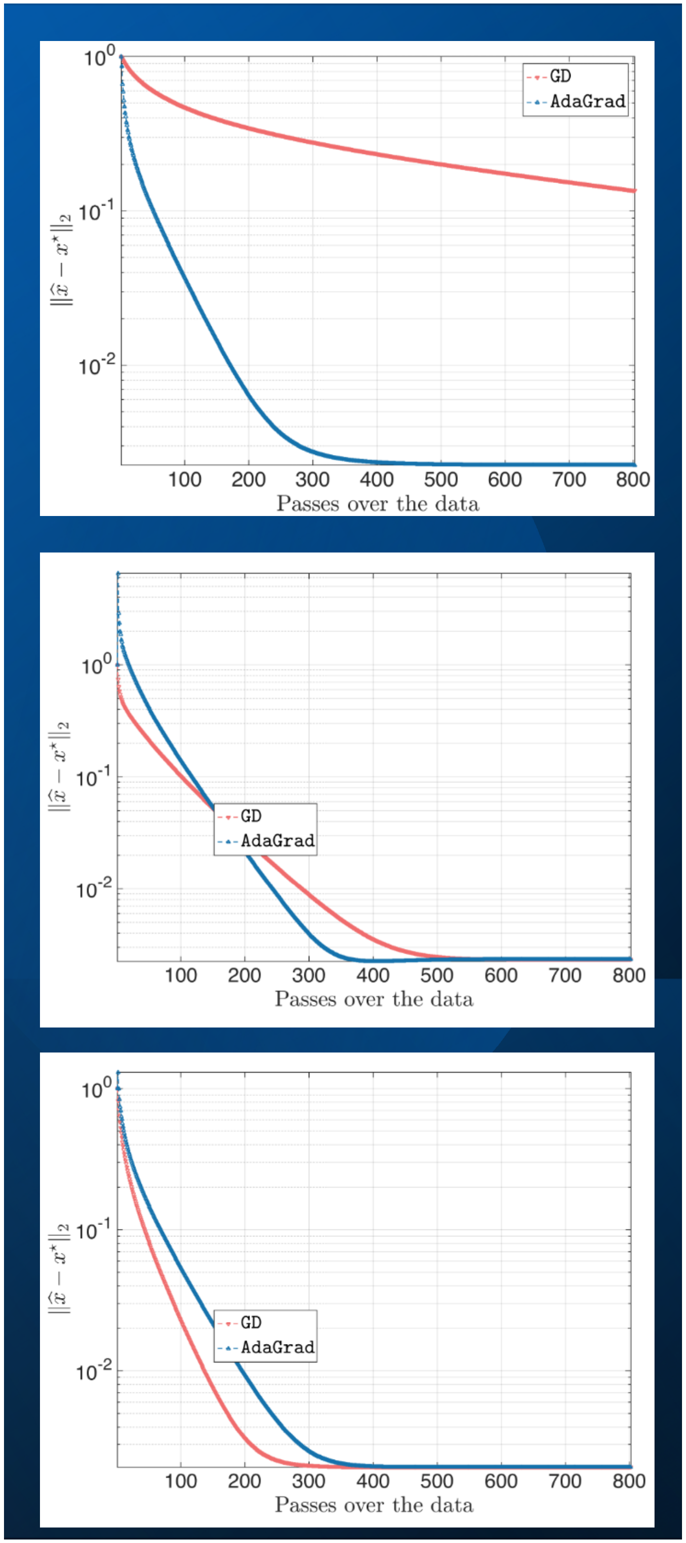
$$w_{\mathrm{t+1,i}} = w_{t,i} + \frac{\eta}{\sqrt{D_{t,ii} + \epsilon}} d_t(w_{\mathrm{t,i}})$$

where

$$D_{t,ii} = Diag\left[\frac{1}{t}\sum_{\tau=1}^{t} \nabla_i f(\vec{w}_\tau) \cdot \nabla_i f(\vec{w}_\tau)^T\right]$$

- In particular, $\underline{D}(\vec{w}_t)$ is selected to be a diagonal preconditioner matrix of the outer product of gradient vectors from all previous time steps.

- I.e $Diag$ will obtain different $\eta' \rightarrow \eta'_{i,t}$ for each $w_{\mathrm{I,t}}$, and ϵ is a smoothing term that avoids division by zero.



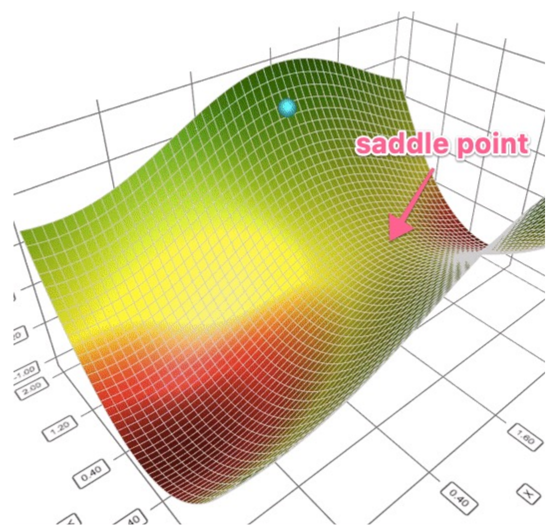
AdaGrad

AdaGrad's strength is making hyperparameter learning rate tuning less sensitive to initial choice (see figure).

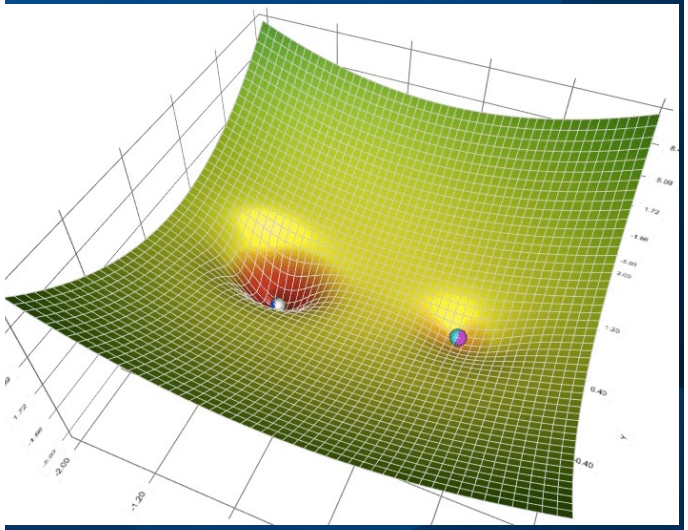
Adagrad's main benefits is that it eliminates the need to manually tune the learning rate as in SGD, and η can be set to almost any small value.

After hyperparameter tuning SGD can be more optimal and as efficient as AdaGrad, but AdaGrad "regularizes" the gradient descent step so that it is always competitive or beats SGD.

$$w_{t+1,i} = w_{t,i} + \frac{\eta}{\sqrt{D_{t,ii} + \epsilon}} d_t(w_i)$$



AdaGrad



AdaGrad's strength is making hyperparameter learning rate tuning less sensitive to choice (see figure).

Adagrad's main benefits is that it eliminates the need to manually tune the learning rate as in SGD, and η can be set to almost any small value.

After hyperparameter tuning SGD can be more optimal and as efficient as AdaGrad, but AdaGrad "regularizes" the gradient descent step so that it is always competitive or beats SGD.

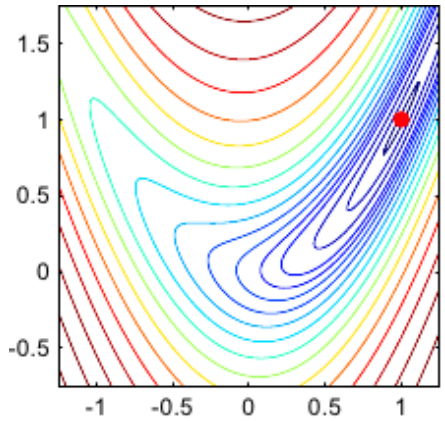
$$w_{t+1,\mathrm{i}} = w_{t,i} + \frac{\eta}{\sqrt{D_{t,ii} + \epsilon}} d_t(w_i)$$

- Adagrad's main weakness is accumulation in the denominator of the adaptive learning rate, which keeps growing during training (squared gradients are positive).

- This causes the effective learning rate η' to shrink to an infinitesimally small value and thus network learning plateaus

Stochastic Gradient Descent with Momentum

SGD and AdaGrad still have the same problems of minimizing along loss function topologies that look like the Rosenbrock function!

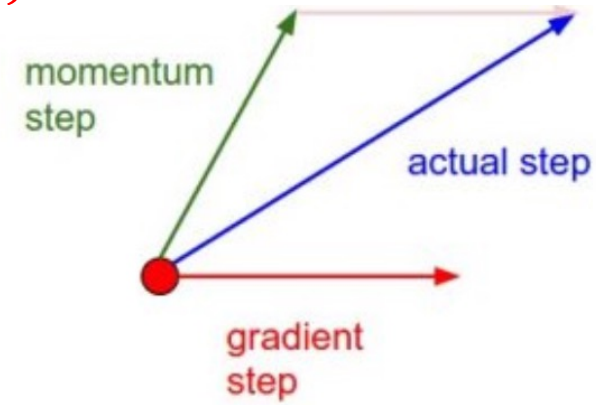


SGD with Momentum (SGDM) is a method that accelerates SGD in better search direction and dampens oscillations along the trough.

It does this by adding a "momentum factor" γ of the direction of the past step to the current direction vector (here the gradient).

$$d_t(\vec{w}) = -\nabla f(\vec{w}_{t-1}) - \gamma d_{t-1}(\vec{w})$$

$$\vec{w}_{t+1} = \vec{w}_t + \eta d_t(\vec{w})$$



It is thus a "poor man's" version of conjugate gradients!

For conjugate gradients we can prove what optimal γ is for quadratic functions (H-orthogonal!).

$$d_{i+1}(\vec{x}) = -\nabla f(\vec{x}_{i+1}) + \gamma\, d_i(\vec{x})$$

$$\gamma = \frac{\nabla f(\vec{x}_{i+1})\nabla f(\vec{x}_{i+1})}{\nabla f(\vec{x}_i)\nabla f(\vec{x}_i)}$$

For SGD with Momentum, γ is yet another useful but heuristic hyperparameter.

$$0 < \gamma < 1$$

It is interesting to see that γ is a memory term which can be tuned to keep previous gradient information in "short to long term memory" compared to CG

$$d_1(\vec{w}) = -\nabla f(\vec{w}_0)$$

$$d_2(\vec{w}) = -\nabla f(\vec{w}_1) - \gamma d_1(\vec{w})$$

$$d_3(\vec{w}) = -\nabla f(\vec{w}_2) - \gamma d_2(\vec{w})$$

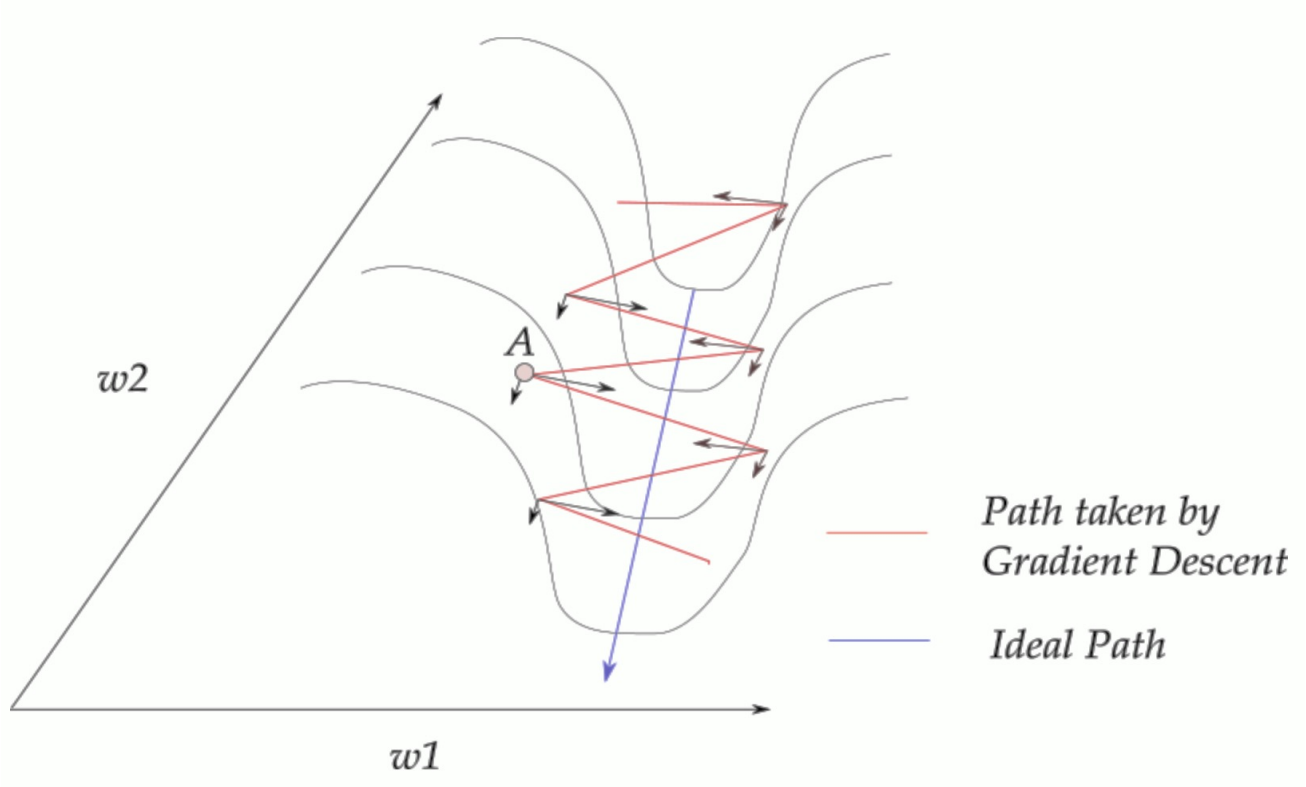
$$= -\nabla f(\vec{w}_2) + \gamma \nabla f(\vec{w}_1) - \gamma^2 \nabla f(\vec{w}_0)$$

Stochastic Gradient Descent with Momentum

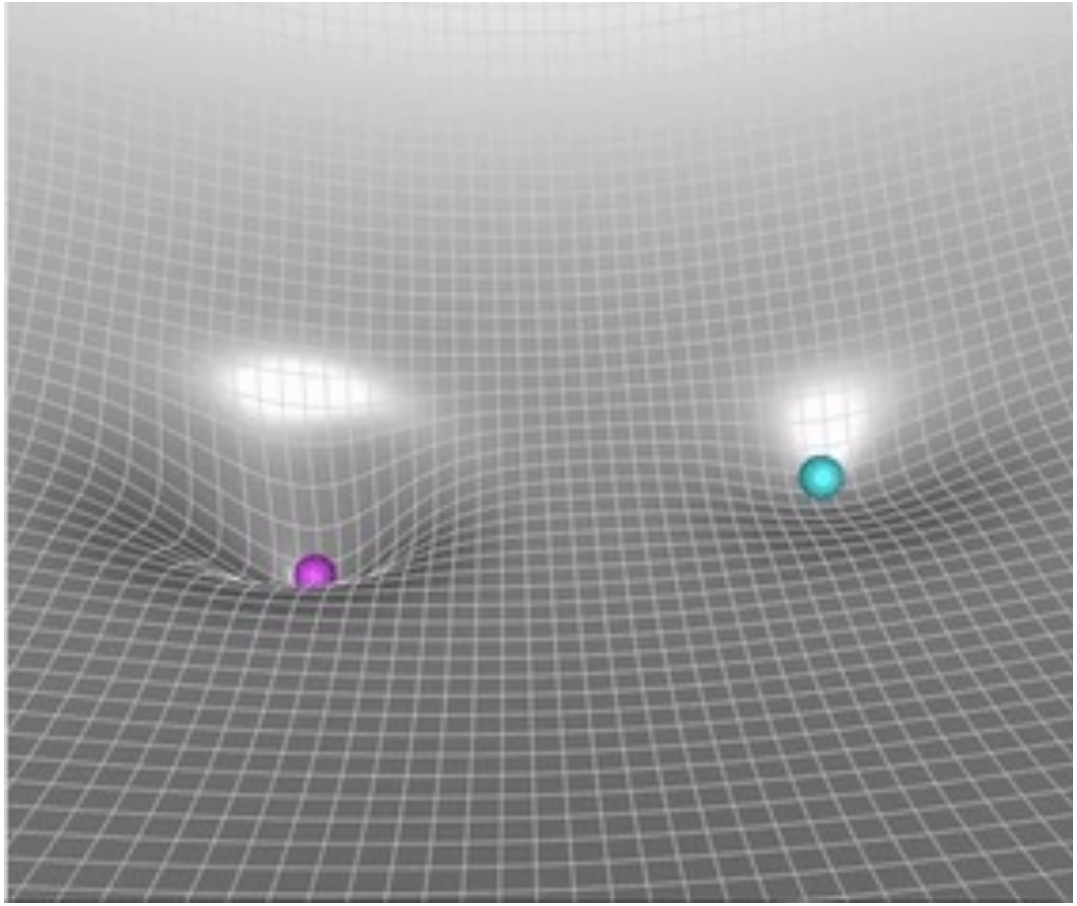
Stochastic Gradient Descent with Momentum

Why is it useful to have memory of previous search directions?

The figure shows that the components of previous search direction vectors that change sign cancel,



while components that move along trough add up – giving you momentum along the ideal path!



Stochastic Gradient Descent with Momentum

1. As momentum accumulates the optimization moves faster

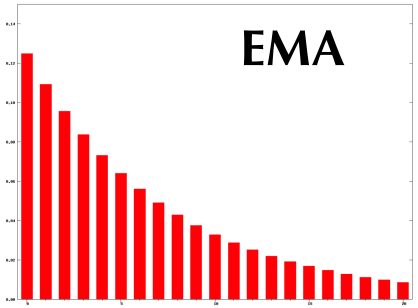
2. Momentum also helps to escape local minima and does not plateau as fast as SGD or AdaGrad.

RMSProp

A similar idea of a gradient history is applied to the preconditioner of AdaGrad. This is important because AdaGrad severely reduces its learning rate in time

$$w_{t+1,\mathrm{i}} = w_{t,i} + \frac{\eta}{\sqrt{D_{t,ii} + \epsilon}} d_t(w_i)$$

I.e. we adapt the preconditioner $\underline{\underline{D}}(\vec{w}_t)$ so that we only keep a finite history of the previous squared gradients through the use of an exponential moving average (EMA)



An EMA is a type of moving average that places a greater weight and significance on the most recent data points.

$$EMA = \alpha(past) + (1 - \alpha)(\text{present})$$

The central idea of RMSprop is keep the EMA of the squared gradients for each weight.

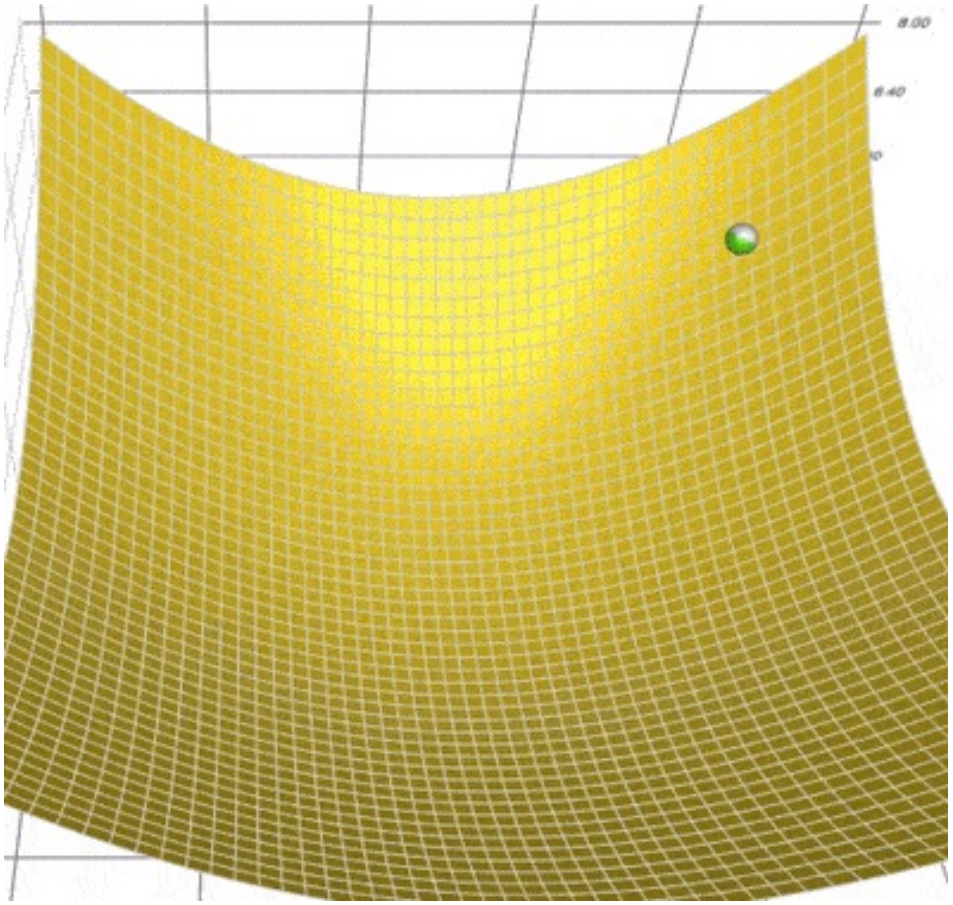
$$D'_{\tau,ii} = \alpha D'_{\tau-1,ii} + (1 - \alpha) diag[\nabla_i f(\vec{w}_{t,i}) \cdot \nabla_i f(\vec{w}_{t,i})^T]$$

where $0 < \alpha < 1$ represent degree of weighting decrease, where lower α discounts older squared gradients faster.

RMSProp

Regularizes weights from growing without bound and stays finite (unlike AdaGrad)

$$w_{t+1,i} = w_{t,i} - \frac{\eta}{\sqrt{D'_{t,ii} + \epsilon}} \overrightarrow{\nabla}_i f(\vec{w}_t)$$



AdaDelta

AdaDelta recognizes a "unit correction" that is not taken into account by RMSProp.

What is a unit correction? If the adjustable parameter has units like distance, the changes to the parameter should be changes in those units as well. LHS has "Δx" units, and RHS units should match.

$$\vec{x}_{t+1} - \vec{x}_t = same\ units$$

Hessian methods define a descent direction that has correct units. For a Newton step $\underline{\underline{H}}^{-1}(\vec{x}_t)$ has "$\Delta x^2/f$" units and $\vec{\nabla} f(\vec{x}_t)$ has "$f/\Delta x$" so $\underline{\underline{H}}^{-1}(\vec{x}_t)\vec{\nabla} f(\vec{x}_t)$ has "Δx" units, i.e. has unit correctness for update, and η stays close to 1!

$$\vec{x}_{t+1} - \vec{x}_t = -\eta \underline{\underline{H}}^{-1}(\vec{x}_t)\vec{\nabla} f(\vec{x}_t)$$

All 1st order methods do not have unit correctness, i.e. LHS has "Δx" units, and RHS has $\propto 1/\Delta x$ units and thus η has to be optimized

$$\vec{x}_{t+1} - \vec{x}_t = -\eta\, \vec{\nabla} f(\vec{x}_t)$$

Inspired by this observation, AdaDelta sets $\eta = 1$ by creating a new EMA to get a "unit correct" method.

AdaDelta

To define this lets rewrite the generic SGD algorithm as

$$\Delta_{t,i} = \eta d_t(w_i)$$

$$\vec{w}_{t+1,i} - \vec{w}_{t,i} = \Delta_{t,i}$$

For SGD, we want to update $\Delta_{t,i}$ so that it has "w units". More to the point we want to chose our adaptive update with w units as an EMA as well, i.e.

$$\Delta^2_{\tau,i} = \alpha\Delta^2_{\tau-1,i} + (1-\alpha)\Delta^2$$

Now update has correct weight units, and eliminates η!

$$\frac{\sqrt{\Delta^2_{\tau,i} + \epsilon}}{\sqrt{D'_{t,ii} + \epsilon}}\vec{\nabla}_i f(\vec{w}_t) \xrightarrow{look\ at\ units} \frac{\sqrt{{w_i}^2}}{\left[\left(\frac{f}{\Delta w}\right)^2\right]^{\frac{1}{2}}}\left(\frac{f}{\Delta w}\right)$$

I.e. RMSProp

$$w_{t+1,i} = w_{t,i} - \frac{\eta}{\sqrt{D'_{t,ii} + \epsilon}}\vec{\nabla}_i f(\vec{w}_t)$$

and AdaDelta

$$w_{t+1,i} = w_{t,i} - \frac{\sqrt{\Delta_{t,i} + \epsilon}}{\sqrt{D'_{t,ii} + \epsilon}}\vec{\nabla}_i f(\vec{w}_t)$$

If we consider that Adagrad and RMSProp all work by adapting the learning rate for SGD, we define Adam as the equivalent adaptive learning method for SGD with Momentum.

In addition to storing an exponentially decaying average of past squared gradients like Adadelta and RMSprop,

$$D'_{t,ii} = \alpha D'_{t-1,ii} + (1-\alpha) diag[\nabla_i f(\vec{w}_\tau) \cdot \nabla_i f(\vec{w}_\tau)^T]$$

Adam also keeps an exponentially decaying average of past gradients or momentum.

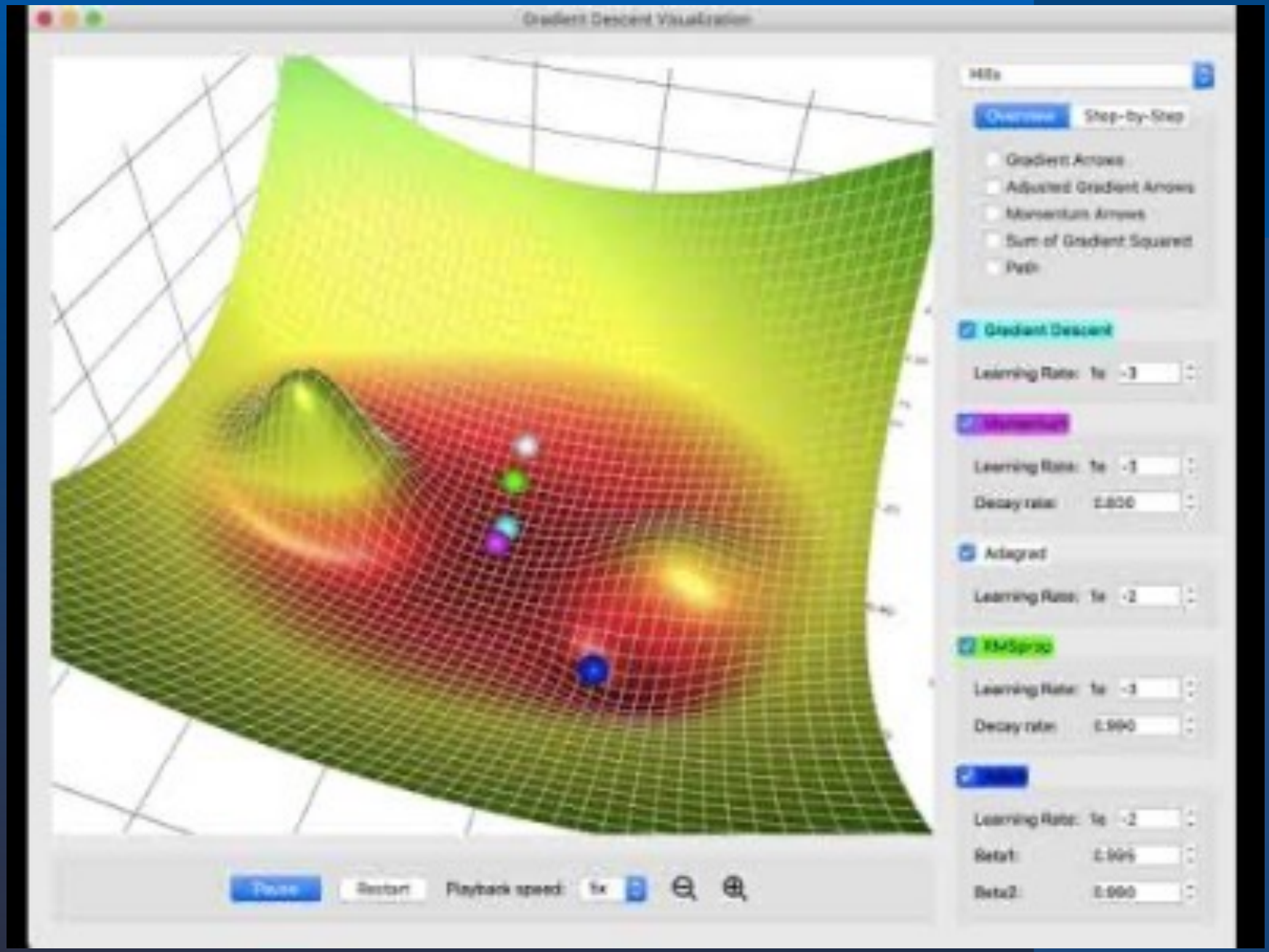
$$d_{t,i} = \gamma\, d_{t-1,i} + (1-\gamma)\, \vec{\nabla_i} f(\vec{w}_t)$$

For greater efficiencies and stabilities $\hat{d}_{t,i} = d_{t,i}/(1-\gamma^t)$ $\widehat{D}_{t,ii} = D_{t,ii}/(1-\alpha^t)$

Final Adam update:

$$w_{t+1,i} = w_{t,i} - \frac{\eta}{\sqrt{\widehat{D}_{t,ii} + \epsilon}} \hat{d}_{t,i}$$

Adam: Adaptive Learning for SGD with Momentum


Gradient Descent Visualization
Overview
Step-by-Step
Gradient Arrows
Adjusted Gradient Arrows
Momentum Arrows
Sum of Gradient Squared
Path
Gradient Descent
Learning Rate: 1e -3
Learning Rate: 1e -3
Decay rate: 0.800
Adagrad
Learning Rate: 1e -2
Learning Rate: 1e -3
Decay rate: 0.900
Learning Rate: 1e -2
Beta1: 0.900
Beta2: 0.990
Pause
Restart
Playback speed: 1x