165B
Machine Learning
Optimization Methods

Lei Li (leili@cs)
UCSB

Acknowledgement: Slides borrowed from Bhiksha Raj's 11485 and Mu Li & Alex Smola's 157 courses on Deep Learning, with modification

Change of Office Hour

- Starting Feb 7. moving to Monday 4-5pm.
- On zoom or in person (HFH 2121)

Convergence of Gradient Descent

Gradient Descent

- Finding the parameter θ to minimize the empirical risk over training data $D = \{(x_n, y_n)\}_{n=1}^N$

$$\hat{\theta} \leftarrow \arg\min_{\theta} L(\theta) = \frac{1}{N}\sum_{n} \ell(y_n, f(x_n; \theta))$$

- Start from initial value
- Update rule: $\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t)$

Convergence Rate

- Assume *f* is convex, and its gradient is Lipschitz continuous with constant *L*

$$\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\| \leq L\|\mathbf{x} - \mathbf{y}\|$$

- If use learning rate $\eta \leq 1/L$, after *T* steps

$$f(\mathbf{x}_T) - f(\mathbf{x}^*) \leq \frac{\|\mathbf{x}_0 - \mathbf{x}^*\|^2}{2\eta T}$$

  - Convergence rate $O(1/T)$
  - To get $f(\mathbf{x}_T) - f(\mathbf{x}^*) \leq \epsilon$, needs $O(1/\epsilon)$ iterations

Proof

- Gradient L-Lipschitz means

$$f(\mathbf{y}) \le f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y}-\mathbf{x}) + \frac{L}{2}\|\mathbf{y}-\mathbf{x}\|^2$$

- Plug in $\mathbf{y} = \mathbf{x} - \eta \nabla f(\mathbf{x})$

$$f(\mathbf{y}) \le f(\mathbf{x}) - \left(1 - \frac{L\eta}{2}\right)\eta\|\nabla f(\mathbf{x})\|^2$$

- Take $0 < \eta \le 1/L$

$$f(\mathbf{y}) \le f(\mathbf{x}) - \frac{\eta}{2}\|\nabla f(\mathbf{x})\|^2$$

f decreases every time

Proof II

- By the convexity: $f(\mathbf{x}) \le f(\mathbf{x}^*) + \nabla f(\mathbf{x})^T(\mathbf{x} - \mathbf{x}^*)$
- Plug in to $f(\mathbf{y}) \le f(\mathbf{x}) - \frac{\eta}{2}\|\nabla f(\mathbf{x})\|^2$

$$f(\mathbf{y}) \le f(\mathbf{x}^*) + \nabla f(\mathbf{x})^T(\mathbf{x} - \mathbf{x}^*) - \frac{\eta}{2}\|\nabla f(\mathbf{x})\|^2$$

$$f(\mathbf{y}) - f(\mathbf{x}^*) \le \left(2\eta\nabla f(\mathbf{x})^T(\mathbf{x} - \mathbf{x}^*) - \eta^2\|\nabla f(\mathbf{x})\|^2\right)/2\eta$$

$$= \left(\underline{\|\mathbf{x} - \mathbf{x}^*}\|^2 + 2\eta\nabla f(\mathbf{x})^T(\mathbf{x} - \mathbf{x}^*) - \eta^2\|\nabla f(\mathbf{x})\|^2 - \underline{\|\mathbf{x} - \mathbf{x}^*}\|^2\right)/2\eta$$

$$= \left(\|\mathbf{x} - \mathbf{x}^*\|^2 - \underline{\|\mathbf{x} - \eta\nabla f(\mathbf{x}) - \mathbf{x}^*\|^2}\right)/2\eta$$

$$= \left(\|\mathbf{x} - \mathbf{x}^*\|^2 - \underline{\|\mathbf{y} - \mathbf{x}^*\|^2}\right)/2\eta$$

Proof III

- Sum all *T* steps

$$\sum_{t=1}^{T} f(\mathbf{x}_t) - f(\mathbf{x}^*) \le \sum_{t=1}^{T} \left(\|\mathbf{x}_{t-1} - \mathbf{x}^*\|^2 - \|\mathbf{x}_t - \mathbf{x}^*\|^2 \right) / 2\eta$$

$$= \left(\|\mathbf{x}_0 - \mathbf{x}^*\|^2 - \|\mathbf{x}_T - \mathbf{x}^*\|^2 \right) / 2\eta \le \|\mathbf{x}_0 - \mathbf{x}^*\|^2 / 2\eta$$

- *f* is decreasing every time:

$$f(\mathbf{x}_T) - f(\mathbf{x}^*) \le \frac{1}{T} \sum_{t=1}^{T} f(\mathbf{x}_t) - f(\mathbf{x}^*) \le \frac{\|\mathbf{x}_0 - \mathbf{x}^*\|^2}{2\eta T}$$

Apply to Deep Learning

- *f* is the sum of loss over all training data, **x** is the learnable parameters

$$f(\mathbf{x}) = \frac{1}{n}\sum_{i=0}^{n}\ell_i(\mathbf{x})$$

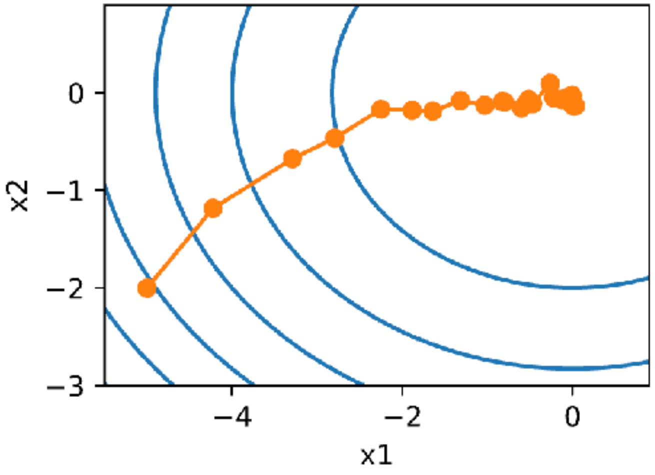
$\ell_i(\mathbf{x})$ the loss for the *i*-th example

- *f* is often not convex, so the convergence analysis before cannot be applied

Stochastic Gradient Descent

- Instead of compute the full gradient, at each step, randomly select a sample t_i

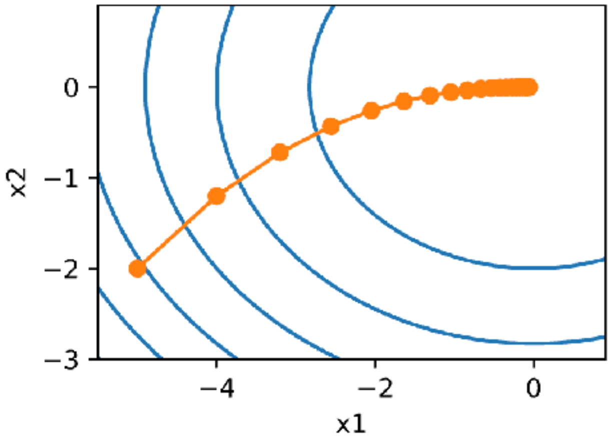
$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta_t \nabla \ell_{t_i}(\mathbf{x}_{t-1})$$



- Compare to gradient descent

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

$$f(\mathbf{x}) = \frac{1}{n} \sum_{i=0}^{n} \ell_i(\mathbf{x})$$



Minibatch Stochastic Gradient Descent

- Instead of full gradient, evaluate and update on random minibatch of data samples B_t

$$x_{t+1} = x_t - \frac{\eta}{|B_t|} \sum_{t_n \in B_t} \nabla \ell_{t_n}(x_t)$$

Stochastic Gradient Descents

- Benefits:
  - Pre-step cost is smaller (and independent of sample size)
  - only need to compute one/batch gradient at a time, smaller memory consumption
- Note stochastic gradient is unbiased estimate of the full gradient at each step

$$E[\nabla \ell_{t_n}(\theta)] = \nabla \ell(\theta)$$

Learning rate

- SGD typically use diminishing step sizes, e.g. $\eta_t = 1/t$
- Why not fixed learning rate?

Convergence Rate

- Assume *f* is convex with a diminishing learning rate $\eta_t = 1/t$, e.g.

$$\mathbb{E}[f(\mathbf{x}_T)] - f(\mathbf{x}^*) = O(1/\sqrt{T})$$

- Under the same assumption, for gradient descent

$$f(\mathbf{x}_T) - f(\mathbf{x}^*) = O(1/\sqrt{T})$$

- Assume gradient L-Lipschitz and fixed η

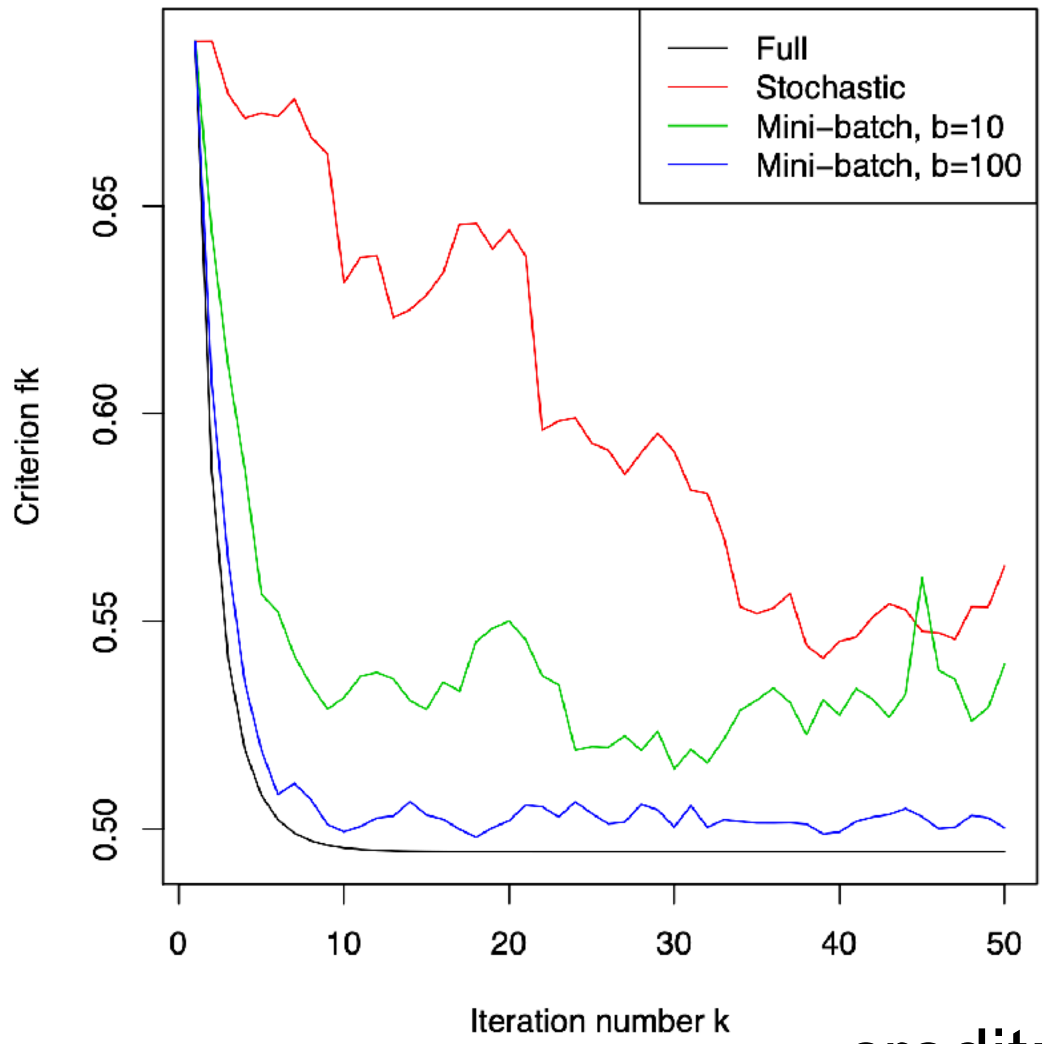
$$f(\mathbf{x}_T) - f(\mathbf{x}^*) = O(1/T)$$

  – But does not improve for SGD

In Practice

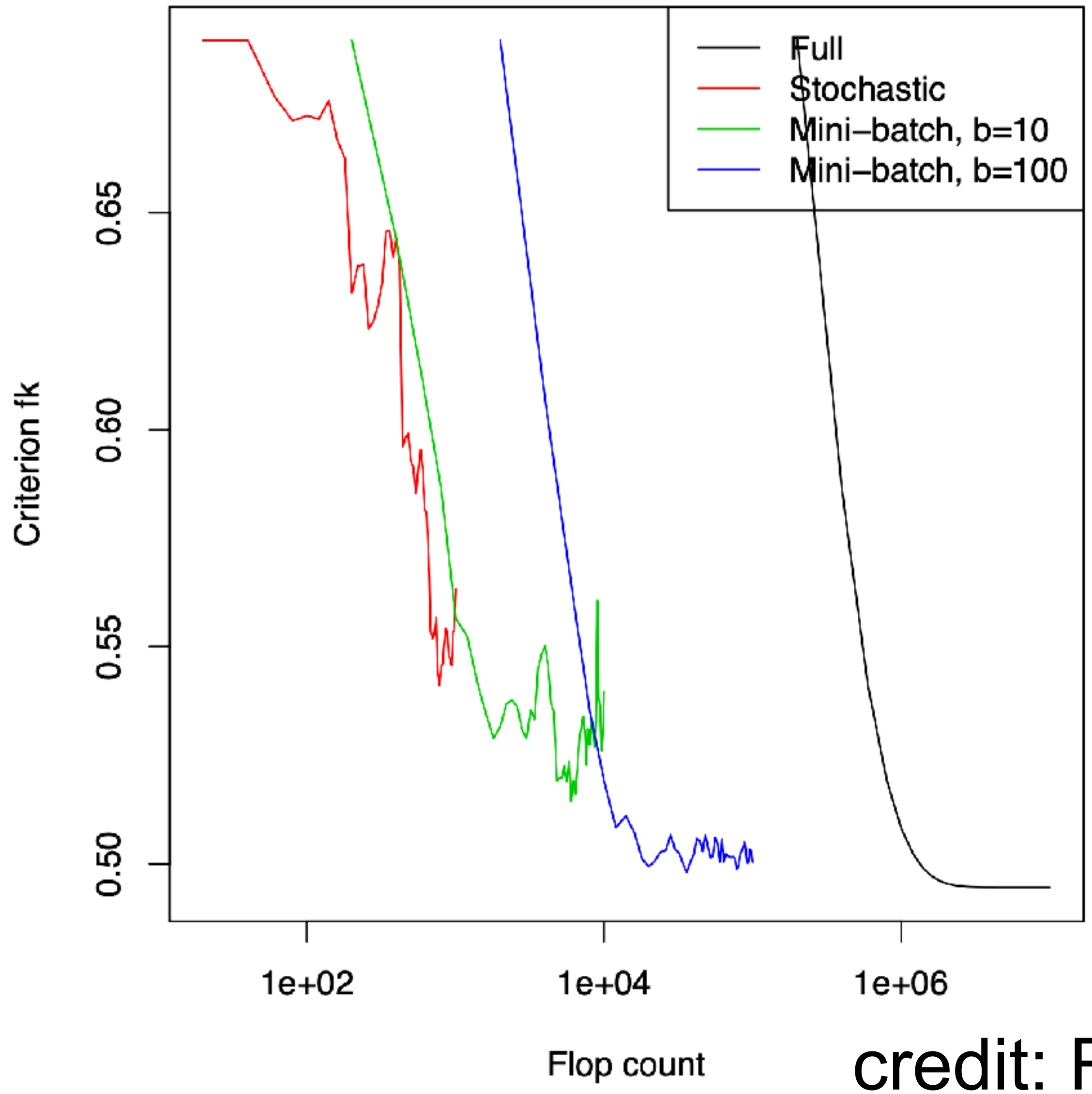
- Does not diminish the learning rate so dramatically
  - We don’t care about optimizing to high accuracy
- Despite converging slower, SGD is way faster on computing the gradient than GD in each iteration
  - Specially for deep learning with complex models and large-scale datasets

Example: Logistic Regression



credit: R. Tibshirani

Convergence in terms of computation

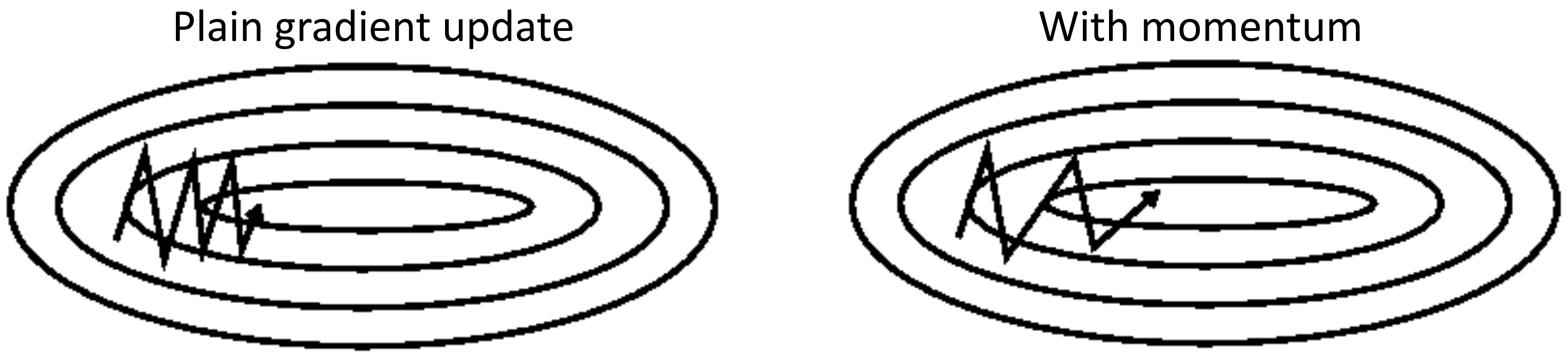


credit: R. Tibshirani

Summary

- SGD is effective in terms of per-iteration cost/memory
- but SGD is slow to converge for strongly convex functions
- New wave of “variance reduction” techniques show modified SGD can converge much faster for finite sums
  - e.g. SVRG

Momentum Method

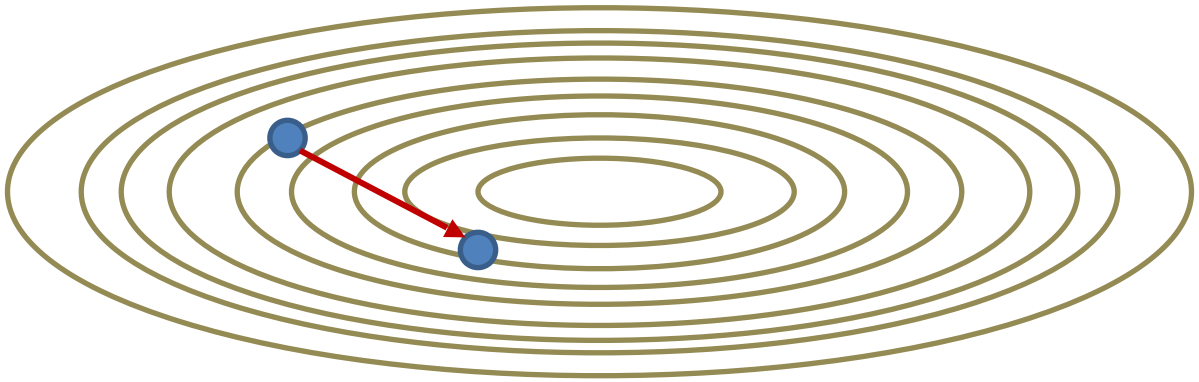


- The momentum method maintains a running average of all gradients until the *current* step

$$v_{t+1} = \beta v_t - \eta \nabla \ell(x_t)$$
$$x_{t+1} = x_t + v_t$$

  – Typical β value is 0.9
- The running average steps
  – Get longer in directions where gradient retains the same sign
  – Become shorter in directions where the sign keeps flipping

Momentum Method

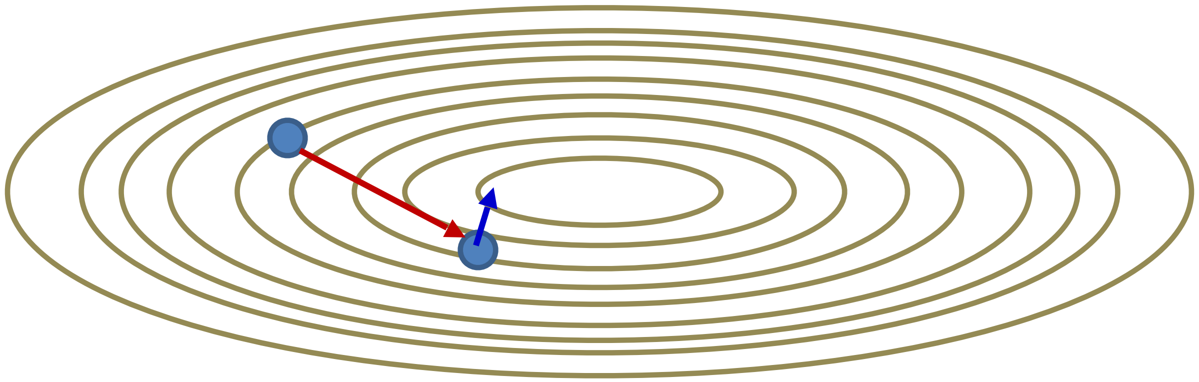


- The momentum method

$$v_{t+1} = \beta v_t - \eta \nabla \ell(x_t)$$

- At any iteration, to compute the current step:

Momentum Method

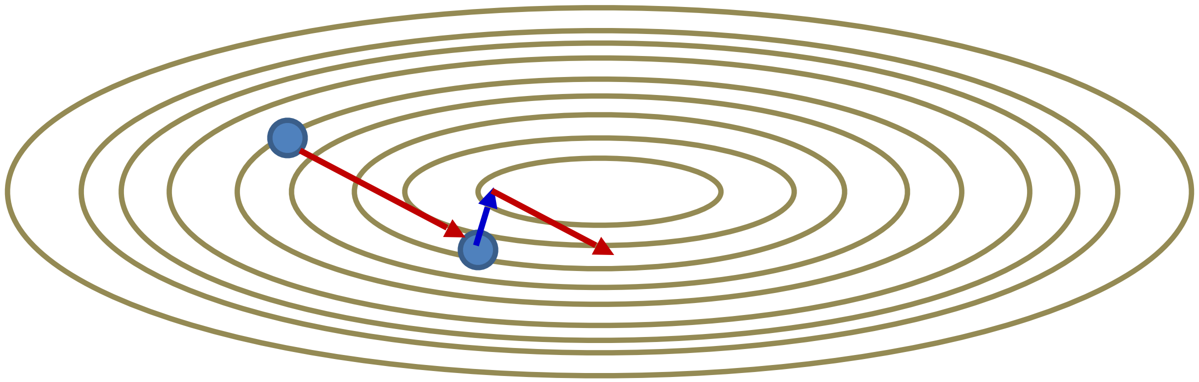


- The momentum method

$$v_{t+1} = \beta v_t - \eta \nabla \ell(x_t)$$

- At any iteration, to compute the current step:
  - First computes the gradient step at the current location

Momentum Method



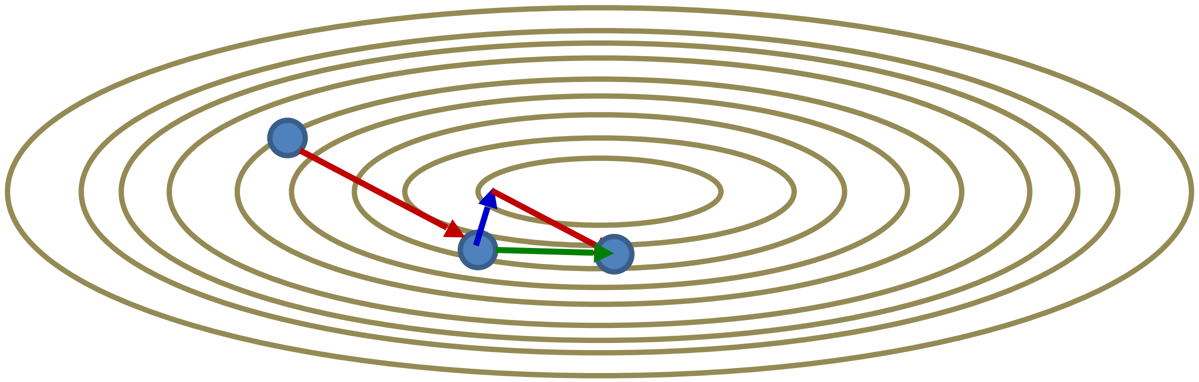
- The momentum method

$$v_{t+1} = \beta v_t - \eta \nabla \ell(x_t)$$

$$x_{t+1} = x_t + v_t$$

- At any iteration, to compute the current step:
  - First computes the gradient step at the current location
  - Then adds in the historical average step
    - which is a running average

Momentum Method



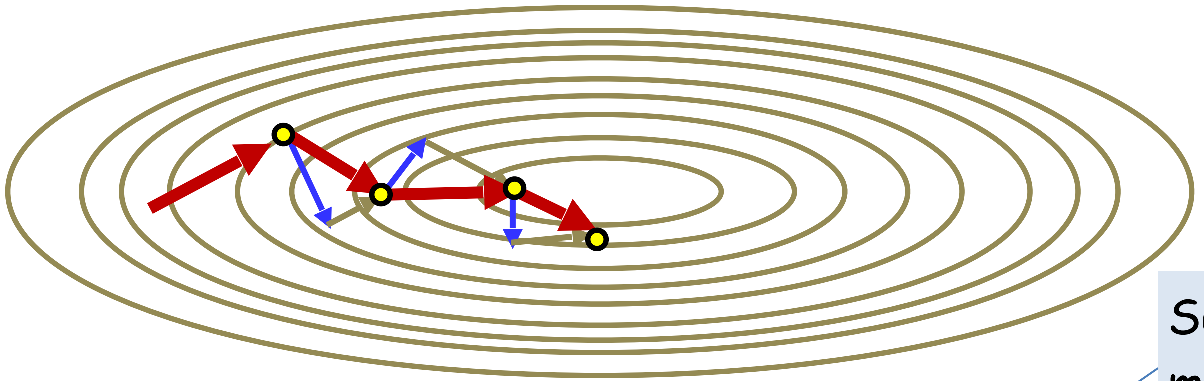
- The momentum method

$$v_{t+1} = \beta v_t - \eta \nabla \ell(x_t)$$

$$x_{t+1} = x_t + v_t$$

- At any iteration, to compute the current step:
  - First computes the gradient step at the current location
  - Then adds in the historical average step
    - which is a running average

SGD with Momentum Updates



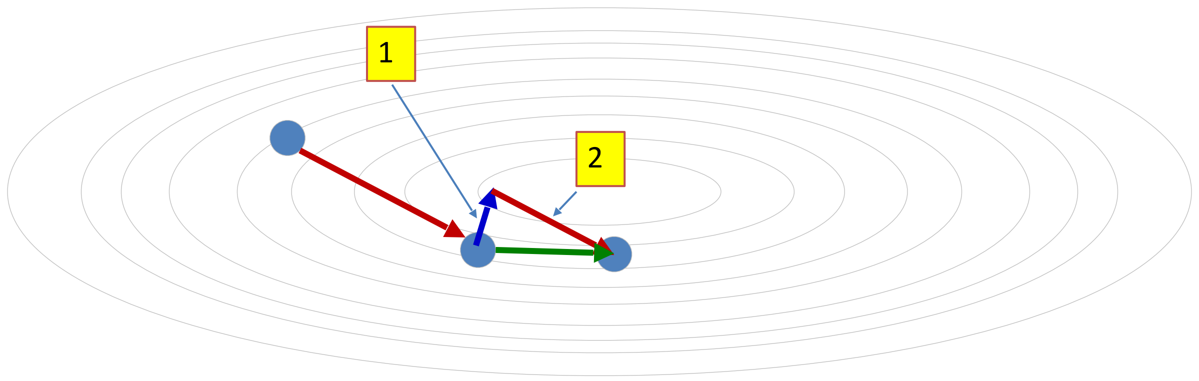
SGD instance or minibatch loss

- The momentum method

$$v_{t+1} = \beta v_t - \eta \nabla \ell(x_t)$$

- Incremental SGD and mini-batch gradients tend to have high variance
- Momentum smooths out the variations
  - Smoother and faster convergence

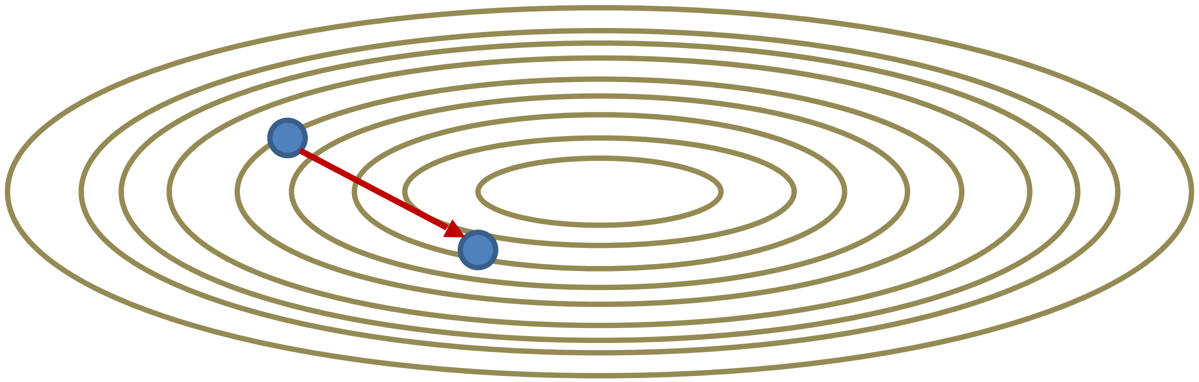
Momentum Method



- Momentum update steps are actually computed in two stages
  - First: We take a step against the gradient at the current location
  - Second: Then we add a scaled version of the previous step

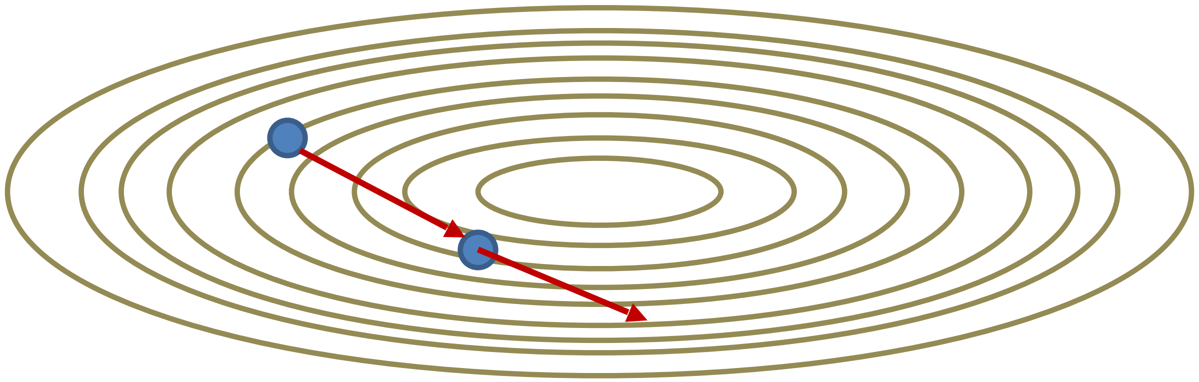
- The procedure can be made more optimal by reversing the order of operations..

Nestorov's Accelerated Gradient



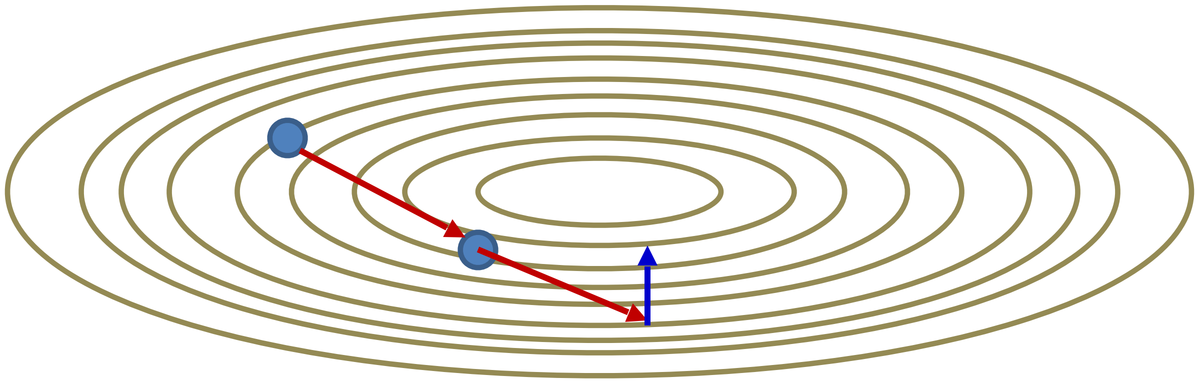
- Change the order of operations
- At any iteration, to compute the current step:

Nestorov's Accelerated Gradient



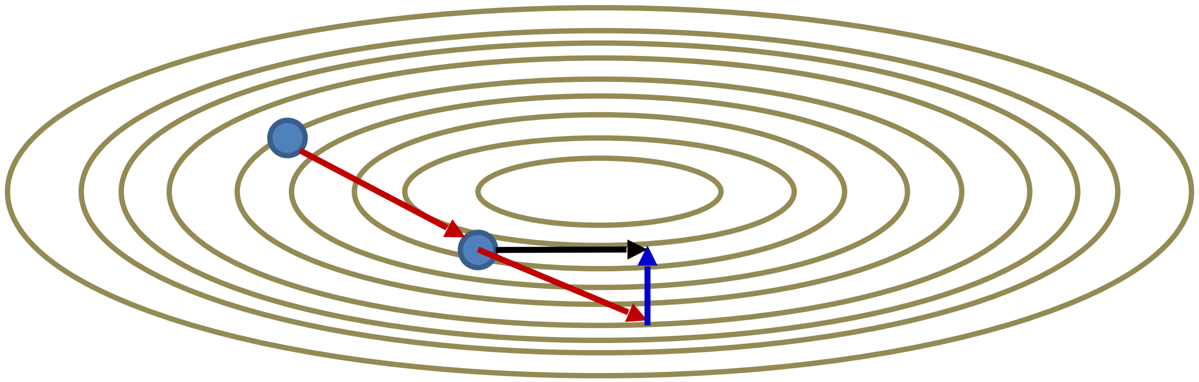
- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step

Nestorov's Accelerated Gradient



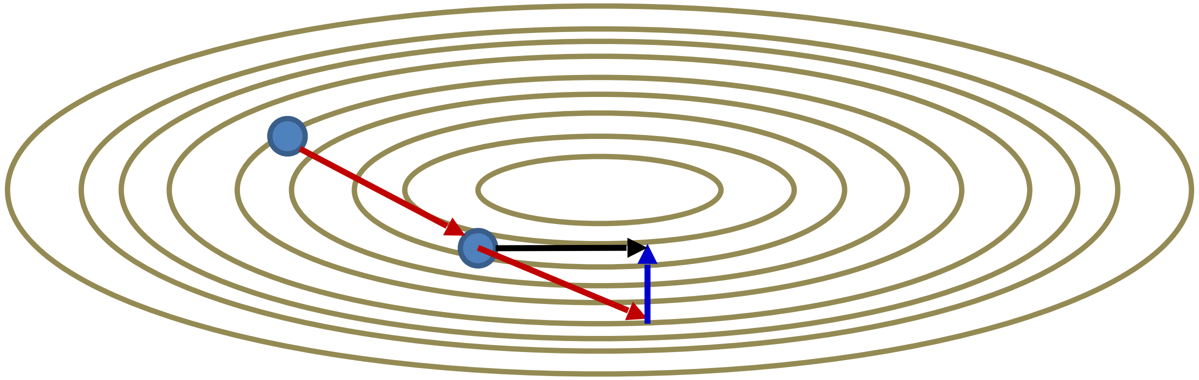
- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step
  - Then compute the gradient step at the resultant position

Nestorov's Accelerated Gradient



- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step
  - Then compute the gradient step at the resultant position
  - Add the two to obtain the final step

Nestorov's Accelerated Gradient

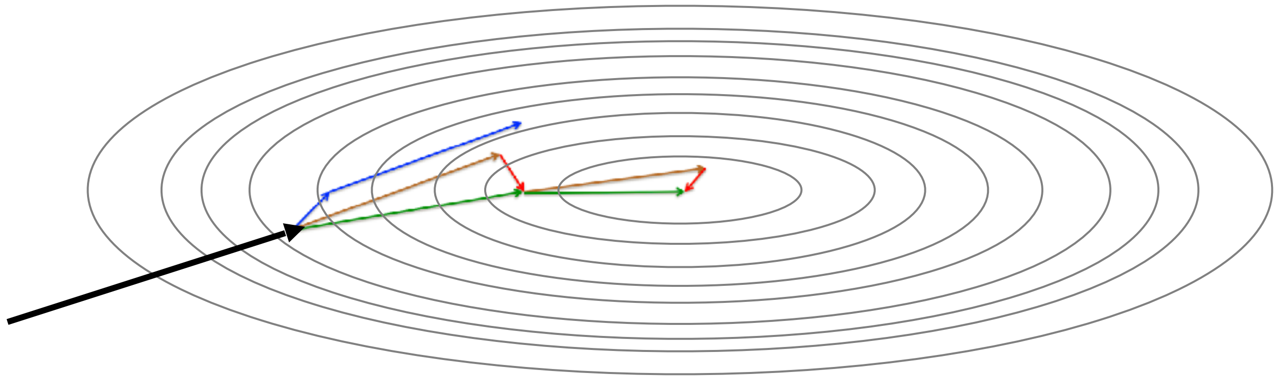


$$x'_{t+1} = x_t + \beta v_t$$

$$v_{t+1} = \beta v_t - \eta \nabla \ell(x'_{t+1})$$

$$x_{t+1} = x_t + v_t$$

Nestorov's Accelerated Gradient

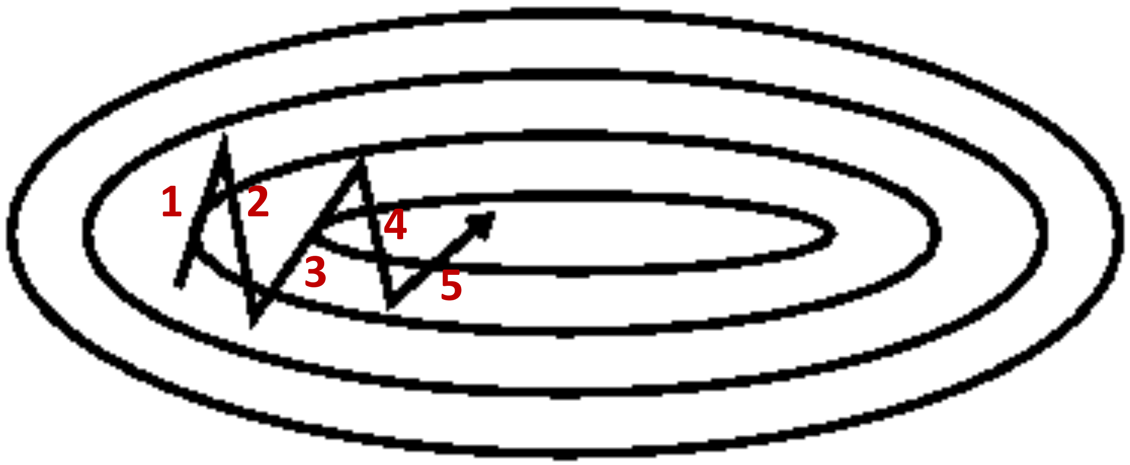


- Comparison with momentum (example from Hinton)
- Converges much faster

Adaptive Gradient Methods

- Momentum and Nestorov’s method improve convergence by normalizing the *mean* of the derivatives
- More recent methods take this one step further by also considering their variance
  - RMS Prop
  - Adagrad
  - AdaDelta
  - ADAM: very popular in practice
  - …

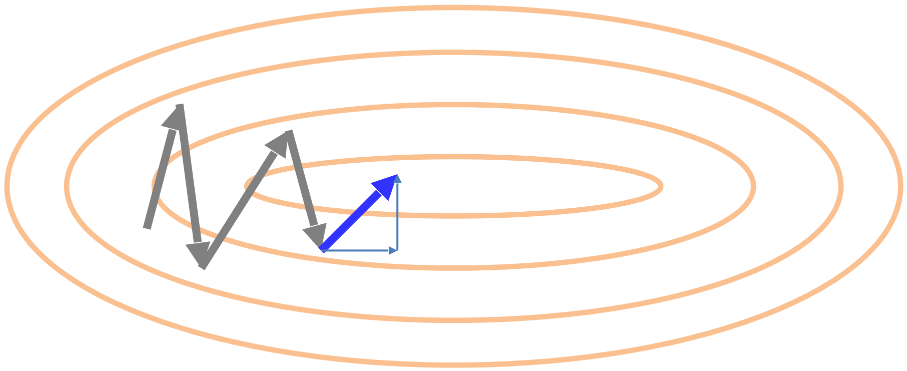
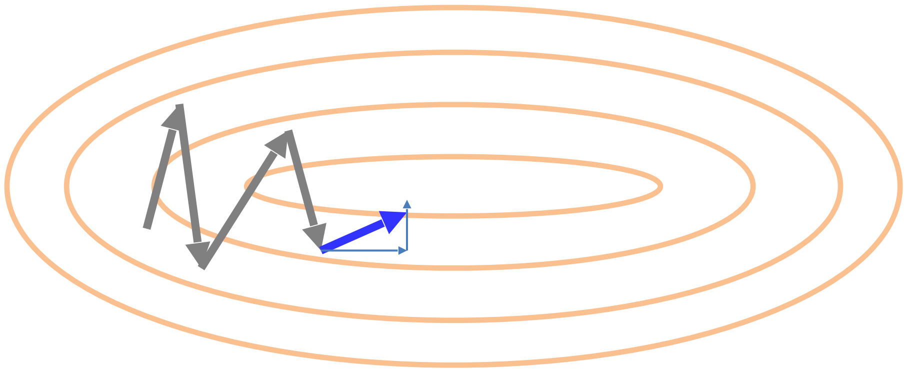
Smoothing the trajectory



Step	X component	Y component
1	1	+2.5
2	1	-3
3	2	+2.5
4	1	-2
5	1.5	1.5

- Observation: Steps in "oscillatory" directions show large total movement
  - In the example, total motion in the vertical direction is much greater than in the horizontal direction
  - Can happen even when momentum or Nesterov are used
- Improvement: Dampen step size in directions with high motion
  - ***Second order moments***

Normalizing steps by second moment

- Modify usual gradient-based update:
  - Scale updates in every component in inverse proportion to the total movement of that component in recent past
    - *According to their variation (not just their average)*
- This will change the relative update sizes for the individual components
  - In the above example it would scale *down* Y component
  - And scale *up* X component (in comparison)
- We will see two popular methods that embody this principle…

Adaptive Gradient

- Notation:
  - Updates are ***by parameter***
  - Derivative of loss w.r.t any individual parameter x is shown as g
    - Batch or minibatch loss, or individual divergence for batch/minibatch/ SGD
  - The ***squared*** derivative is $g^2 = (\nabla \ell(x))^2$
    - Short-hand notation represents the squared derivative, not the second derivative
  - The ***mean squared*** derivative is a running estimate of the average squared derivative. We will show this as $E[g^2]$
- Modified update rule: We want to
  - scale down updates with large mean squared derivatives
  - scale up updates with small mean squared derivatives

AdaGrad

- AdaGrad (Duchi, Hazan, and Singer 2010) very popular adaptive method.

$$G_{t+1} = G_t + \nabla \ell(x_t)^2$$

$$x_{t+1} = x_t - \eta \frac{1}{\sqrt{G_{t+1} + \epsilon}} \nabla \ell(x_t)$$

- Element-wise computation

AdaGrad

- AdaGrad (Duchi, Hazan, and Singer 2010) very popular adaptive method.

element-wise

$$G_{t+1} = G_t + \nabla \ell(x_t)^2$$

$$x_{t+1} = x_t - \eta \frac{1}{\sqrt{G_{t+1} + \epsilon}} \nabla \ell(x_t)$$

- Benefits:
  - AdaGrad does not require tuning learning rate η
  - Actual learning rate will decrease
  - Can drastically improve over SGD

Quiz

- https://edstem.org/us/courses/16390/lessons/29666/slides/170130

RMSProp

- Similar to AdaGrad, accumulate the squared gradients, but with running average
  - Adagrad denominator monotonically increase ==> diminishing updates for parameters
  - why not decay the denominator

$$G_{t+1} = \beta G_t + (1-\beta)\nabla \ell(x_t)^2$$

$$x_{t+1} = x_t - \eta \frac{1}{\sqrt{G_{t+1}+\epsilon}} \nabla \ell(x_t)$$

element-wise

-

ADAM: RMSprop + Momentum

- RMS prop only considers a second-moment normalized version of the current gradient
- ADAM utilizes a smoothed version of the *momentum-augmented* gradient
  - Considers both first and second moments

$$m_{t+1} = \beta_1 m_t - (1 - \beta_1) \nabla \ell(x_t)$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2)(\nabla \ell(x_t))^2$$

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1 - \beta_1^{t+1}}$$

$$\hat{v}_{t+1} = \frac{v_{t+1}}{1 - \beta_2^{t+1}}$$

$$x_{t+1} = x_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}} + \epsilon} \hat{m}_{t+1}$$

ADAM: RMSprop + Momentum

- RMS prop only considers a second-moment normalized version of the current gradient
- ADAM utilizes a smoothed version of the *momentum-augmented* gradient
  - Considers both first and second moments

$$m_{t+1} = \beta_1 m_t - (1 - \beta_1) \nabla \ell(x_t)$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2)(\nabla \ell(x_t))^2$$

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1 - \beta_1^{t+1}}$$

$$\hat{v}_{t+1} = \frac{v_{t+1}}{1 - \beta_2^{t+1}}$$

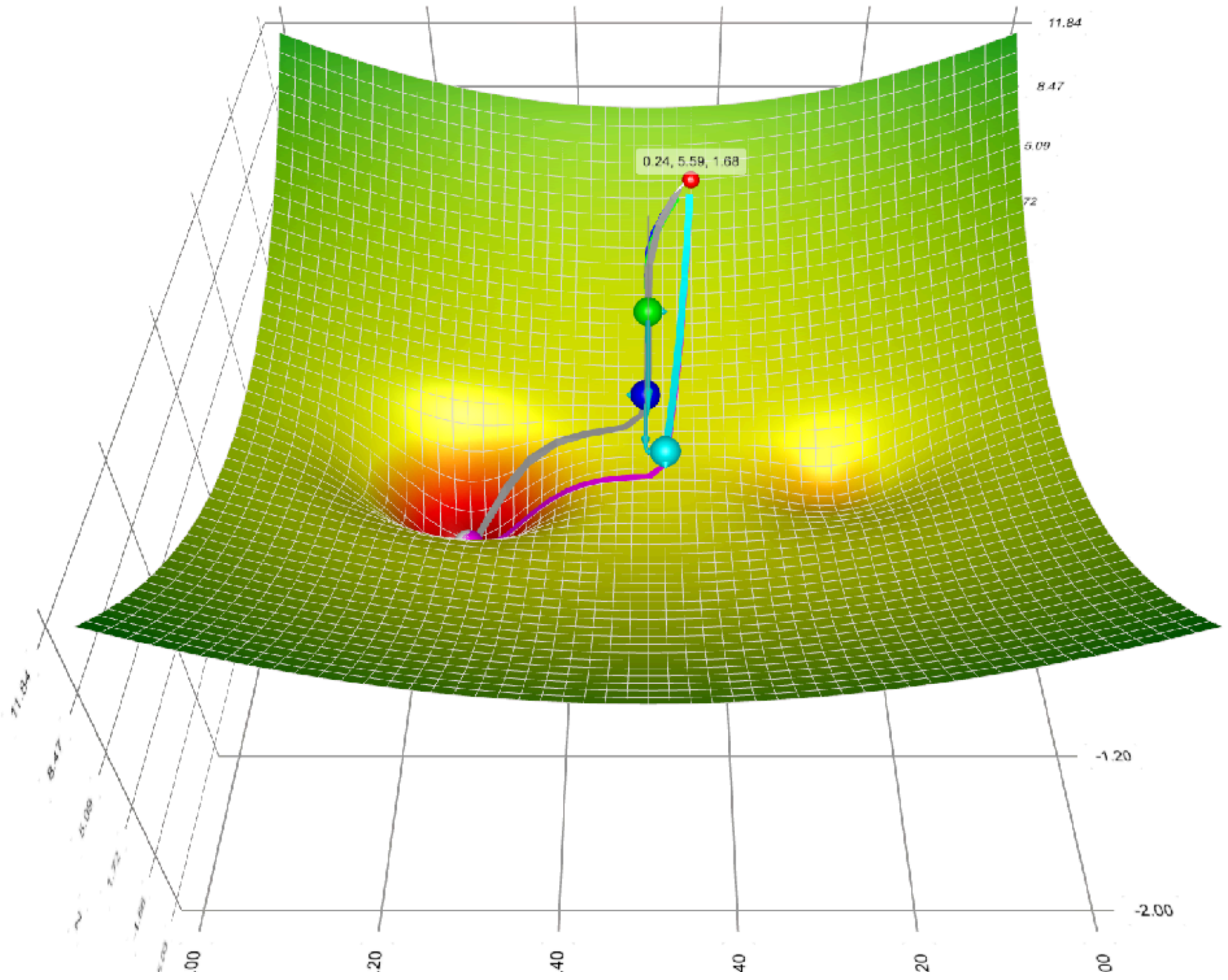
Why?

$$x_{t+1} = x_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}} + \epsilon} \hat{m}_{t+1}$$

Other variants of the same theme

- Many:
  - AdaDelta
  - AdaMax
  - …
- Generally no explicit learning rate to optimize
  - But come with other hyper parameters to be optimized
  - Typical params:
    - AdaGrad: $\eta = 0.001,$
    - RMSProp: $\eta = 0.001, \beta = 0.9$
    - ADAM: $\eta = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$

Visualization



https://github.com/lilipads/gradient_descent_viz

Newton's Method

- Second-order method
- $f(x_t + \Delta x) \approx f(x_t) + \Delta x^T \nabla f|_{x_t} + \frac{1}{2} \Delta x^T \nabla^2 f|_{x_t} \Delta x$
- Let gradient $g_t = \nabla f|_{x_t}$, Hessian $H_t = \nabla^2 f|_{x_t}$
- Let $\frac{\partial f(x_t + \Delta x)}{\partial \Delta x} = 0$

$$x_{t+1} = x_t - \eta \cdot H_t^{-1} \cdot g_t$$

- updated on stochastic minibatch for large data

Newton's method

- Faster convergence
- Higher per-iteration cost. O(d^3)
  - also needs memory O(d^2)
-

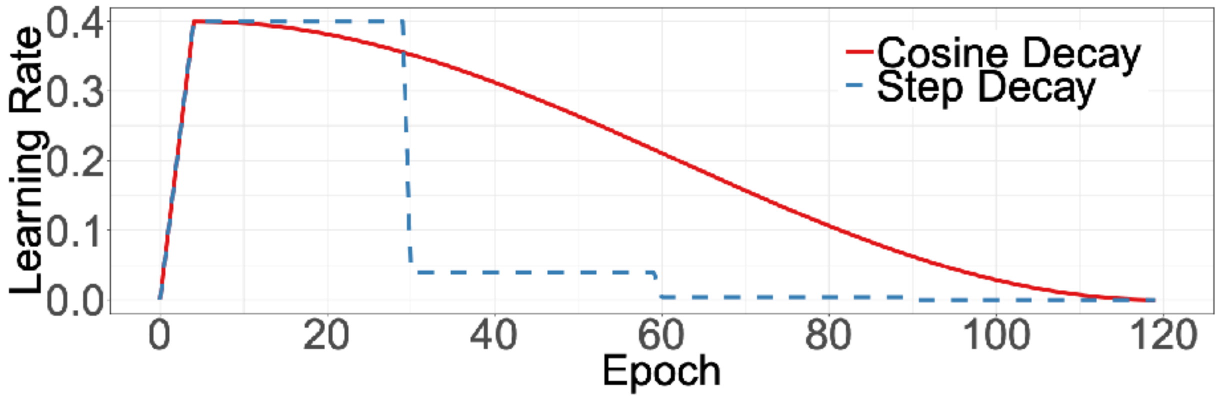
Tricks for Training

Learning Rate Warmup

- A large learning rate for randomly initialized parameters may cause numerical issue
- The warmup trick uses a small learning rate at beginning and then increases it to the initial value. For example:
  - If we choose the initial learning rate to be 0.1 and use 5 epochs for warmup
  - Start the learning rate with 0, linearly increases it to 0.1 in the first 5 epochs

Cosine Decay

- We need to decrease learning rate for SGD to converge
  - E.g. decreasing by 10x at epoch 30, 60, and 90
- Assume in total *T* iterations (batches), the cosine decay computes learning rate at iteration *t* by $\eta_t = 1/2\left(1 + \cos\left(t\pi/T\right)\right)\eta$

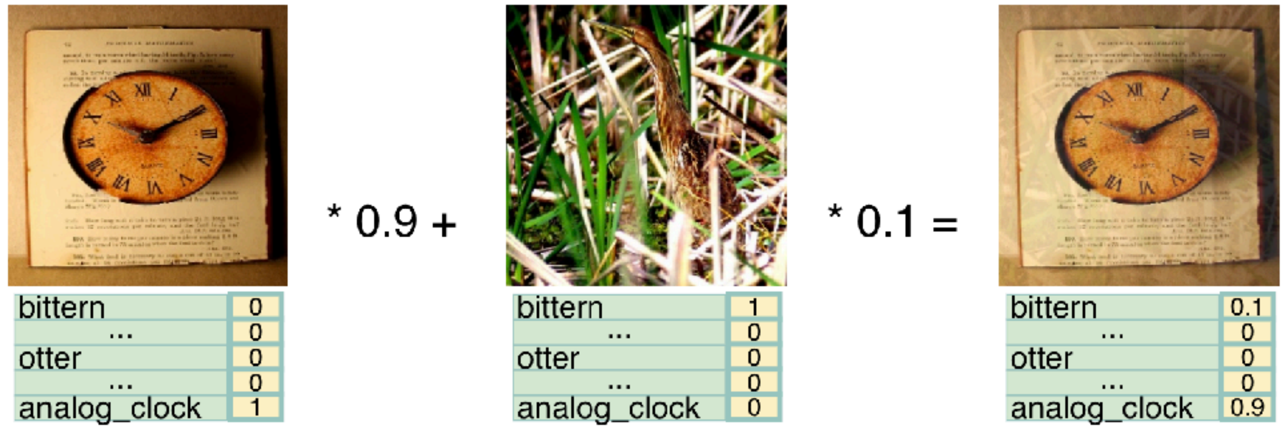


Mixup Training Example

- Randomly select two examples *i* and *j*, sample a random number $\lambda \in [0,1]$
- Compute the mixed new example

$$x = \lambda x_i + (1-\lambda)x_j \qquad y = \lambda y_i + (1-\lambda)y_j$$

- Train on mixed examples



Label Smoothing

- Assume $\mathbf{y} \in \mathbb{R}^n$ is the one-hot encoding of label

$$y_i = \begin{cases} 1 & \text{if belongs to class } i \\ 0 & \text{otherwise} \end{cases}$$

- Approximating 0/1 values with softmax is hard
- The smoothed version

$$y_i = \begin{cases} 1 - \epsilon & \text{if belongs to class } i \\ \epsilon/(n-1) & \text{otherwise} \end{cases}$$

  – Commonly use $\epsilon = 0.1$

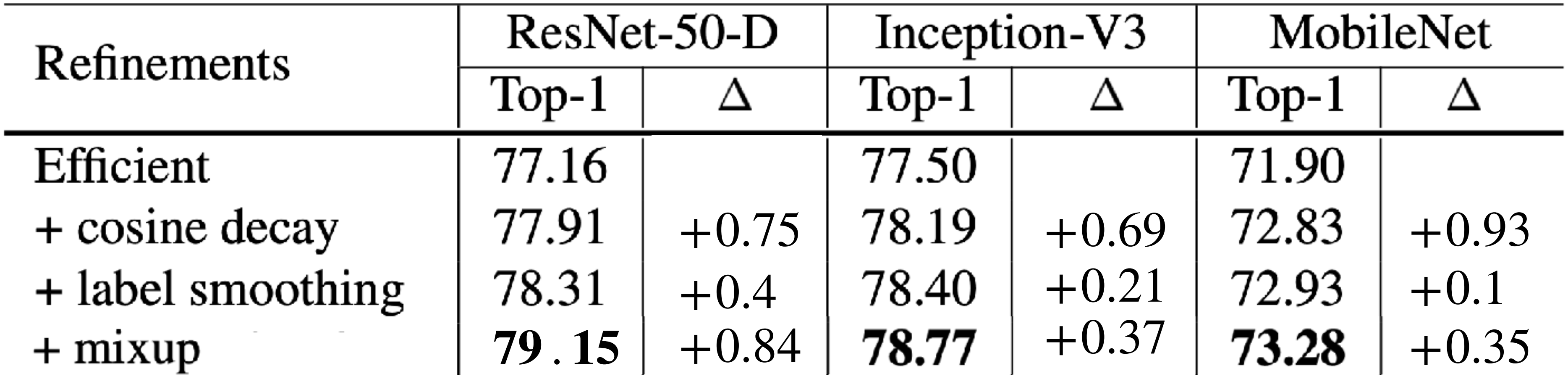
Synchronized Batch Normalization

- BatchNorm needs a large batch size for reliable statistics
- Object detection tasks may allow a small batch size due to GPU memory constraints, e.g. 1 image per GPU
- In multi-GPU training, each GPU computes mean/variance separately
- Synchronized BatchNorm computes statistics over all GPUs

Random Batch Shapes

- Images are resized to same shape in a batch, e.g. 224 width and 224 height
- We can vary this shape:
  - For each batch, choose a random width/height from 224 (7x32), 256 (8x32), 228 (9x32), …
  - Resize all images into this shape

Image Classification



Refinements	ResNet-50-D		Inception-V3		MobileNet	
	Top-1	Δ	Top-1	Δ	Top-1	Δ
Efficient	77.16		77.50		71.90	
+ cosine decay	77.91	+0.75	78.19	+0.69	72.83	+0.93
+ label smoothing	78.31	+0.4	78.40	+0.21	72.93	+0.1
+ mixup	**79 . 15**	+0.84	**78.77**	+0.37	**73.28**	+0.35

Hang et.al *Bag of Tricks for Image Classification with Convolutional Neural Networks*

Summary

- Gradient descent can be sped up by incremental updates
  - Convergence is guaranteed under most conditions
    - Learning rate must shrink with time for convergence
  - Stochastic gradient descent: update after each observation. Can be much faster than batch learning
  - Mini-batch updates: update after batches. Can be more efficient than SGD

- Convergence can be improved using smoothed updates
  - AdaGrad, RMSprop, Adam and more advanced techniques

Next Up

- Detecting objects in images