CS60021: Scalable Data Mining

Large Scale Machine Learning

Sourangshu Bhattacharya

Much of ML is optimization

Linear Classification

$$\arg\min_{w} \sum_{i=1}^{n} ||w||^2 + C \sum_{i=1}^{n} \xi_i$$
$$\text{s.t. } 1 - y_i x_i^T w \leq \xi_i$$
$$\xi_i \geq 0$$

Maximum Likelihood

$$\arg\max_{\theta} \sum_{i=1}^{n} \log\, p_\theta(x_i)$$

K-Means

$$\arg \min_{\mu_1, \mu_2, \ldots, \mu_k} J(\mu) = \sum_{j=1}^{k} \sum_{i \in C_j} ||x_i - \mu_j||^2$$

Stochastic optimization

- Goal of machine learning :
  - Minimize expected loss

$$\min_h L(h) = \mathbf{E}\left[\text{loss}(h(x), y)\right]$$

given samples $(x_i, y_i)\ i = 1, 2...m$

- This is Stochastic Optimization
  - Assume loss function is convex

Batch (sub)gradient descent for ML

- Process all examples together in each step

$$w^{(k+1)} \leftarrow w^{(k)} - \eta_t \left(\frac{1}{n} \sum_{i=1}^{n} \frac{\partial L(w, x_i, y_i)}{\partial w} \right)$$

where L is the regularized loss function

- Entire training set examined at each step
- Very slow when *n* is very large

Stochastic (sub)gradient descent

- "Optimize" one example at a time
- Choose examples randomly (or reorder and choose in order)
  - Learning representative of example distribution

for $i = 1$ to n:

$$w^{(k+1)} \leftarrow w^{(k)} - \eta_t \frac{\partial L(w, x_i, y_i)}{\partial w}$$

where L is the regularized loss function

Stochastic (sub)gradient descent

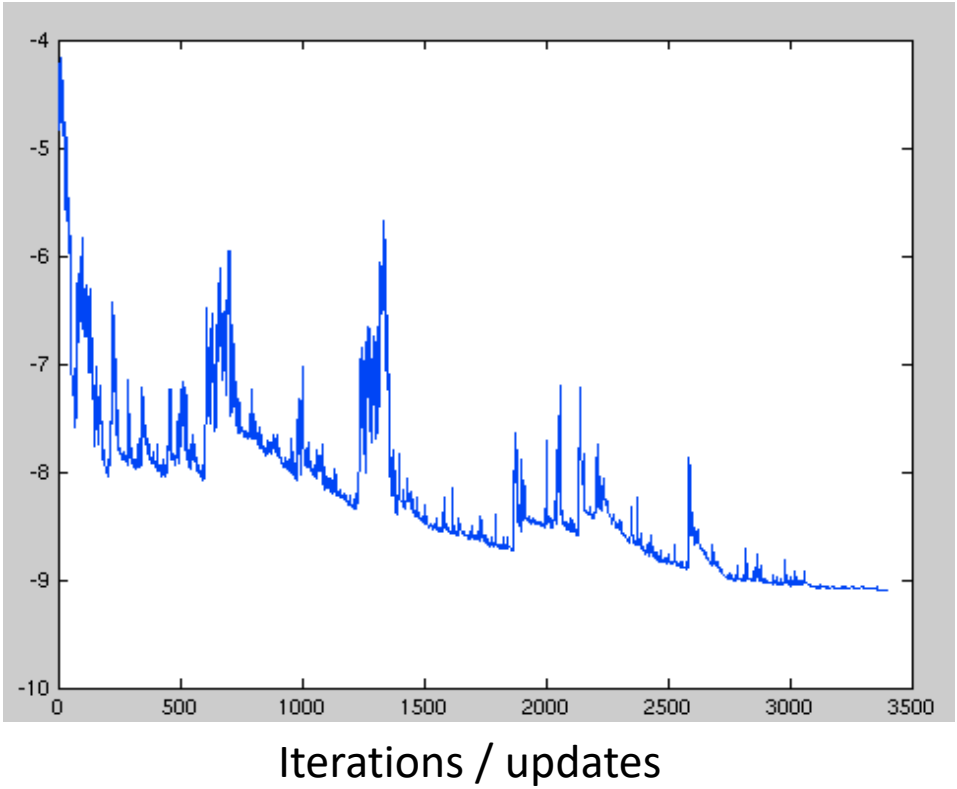
for $i = 1$ to n:

$$w^{(k+1)} \leftarrow w^{(k)} - \eta_t \frac{\partial L(w, x_i, y_i)}{\partial w}$$

where L is the regularized loss function

- Equivalent to online learning (the weight vector *w* changes with every example)
- Convergence guaranteed for convex functions (to local minimum)

SGD convergence



Stochastic gradient descent

- Given dataset $D = \{(x_1, y_1), \dots, (x_m, y_m)\}$
- Loss function: $L(\theta, D) = \frac{1}{N}\sum_{i=1}^{N} l(\theta; x_i, y_i)$
- For linear models: $l(\theta; x_i, y_i) = l(y_i, \theta^T \phi(x_i))$
- Assumption D is drawn IID from some distribution $\mathcal{P}$.
- Problem:

$$\min_{\theta} L(\theta, D)$$

Stochastic gradient descent

- Input: D
- Output: $\bar{\theta}$

Algorithm:

- Initialize θ^0
- For $t = 1, \ldots, T$

$$\theta^{t+1} = \theta^t - \eta_t \nabla_\theta l(y_t, \theta^T \phi(x_t))$$

- $\bar{\theta} = \frac{\sum_{t=1}^{T} \eta_t \theta^t}{\sum_{t=1}^{T} \eta_t}$.

SGD convergence

- Expected loss: $s(\theta) = E_{\mathcal{P}}[l(y, \theta^T \phi(x)]$
- Optimal Expected loss: $s^* = s(\theta^*) = \min_{\theta} s(\theta)$
- Convergence:

$$E_{\bar{\theta}}[s(\bar{\theta})] - s^* \leq \frac{R^2 + L^2 \sum_{t=1}^{T} \eta_t^2}{2 \sum_{t=1}^{T} \eta_t}$$

- Where: $R = \|\theta^0 - \theta^*\|$
- $L = \max \nabla l(y, \theta^T \phi(x))$

SGD convergence proof

- Define $r_t = \|\theta^t - \theta^*\|$ and $g_t = \nabla_\theta l\big(y_t, \theta^T \phi(x_t)\big)$
- $r_{t+1}^2 = r_t^2 + \eta_t^2 \|g_t\|^2 - 2\eta_t(\theta^t - \theta^*)^T g_t$
- Taking expectation w.r.t $\mathcal{P}, \bar{\theta}$ and using $s^* - s(\theta^t) \geq g_t^T(\theta^* - \theta^t)$, we get:
  $E_{\bar{\theta}}[r_{t+1}^2 - r_t^2] \leq \eta_t^2 L^2 + 2\eta_t(s^* - E_{\bar{\theta}}[s(\theta^t)])$

- Taking sum over $t = 1, \ldots, T$ and using
  $$E_{\bar{\theta}}[r_{t+1}^2 - r_0^2] \leq L^2 \sum_{t=0}^{T-1} \eta_t^2 + 2 \sum_{t=0}^{T-1} \eta_t (s^* - E_{\bar{\theta}}[s(\theta^t)])$$

SGD convergence proof

- Using convexity of s:

$$\left(\sum_{t=0}^{T-1}\eta_t\right)E_{\bar{\theta}}[s(\bar{\theta})] \leq E_{\bar{\theta}}[\sum_{t=0}^{T-1}\eta_t s(\theta^t)]$$

- Substituting in the expression from previous slide:

$$E_{\bar{\theta}}[r_{t+1}^2 - r_0^2] \leq L^2\sum_{t=0}^{T-1}\eta_t^2 + 2\sum_{t=0}^{T-1}\eta_t(s^* - E_{\bar{\theta}}[s(\bar{\theta})])$$

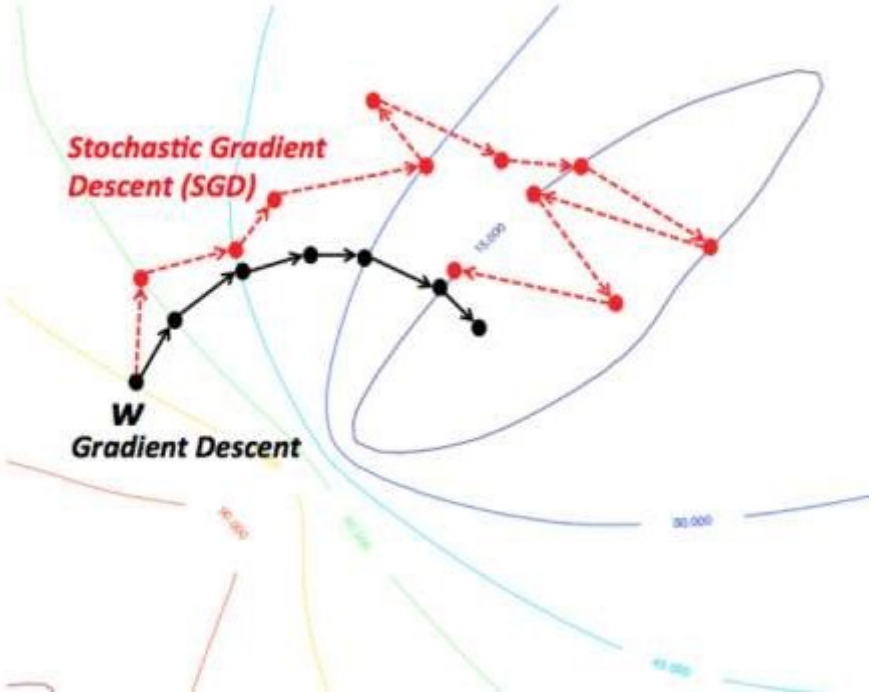
- Rearranging the terms proves the result.

SGD - Issues

- Convergence very sensitive to learning rate

  (η_t) (oscillations near solution due to probabilistic nature of sampling)
  - Might need to decrease with time to ensure the algorithm converges eventually

- Basically – SGD good for machine learning with large data sets!



Mini-batch SGD

- Stochastic – 1 example per iteration
- Batch – All the examples!
- Mini-batch SGD:
  - Sample *m* examples at each step and perform SGD on them
- Allows for parallelization, but choice of *m* based on heuristics

Example: Text categorization

- **Example by Leon Bottou:**
  - **Reuters RCV1** document corpus
    - Predict a category of a document
      - One **vs.** the rest classification
  - ***n* = 781,000** training examples (documents)
  - 23,000 test examples
  - ***d* = 50,000** features
    - One feature per word
    - Remove stop-words
    - Remove low frequency words

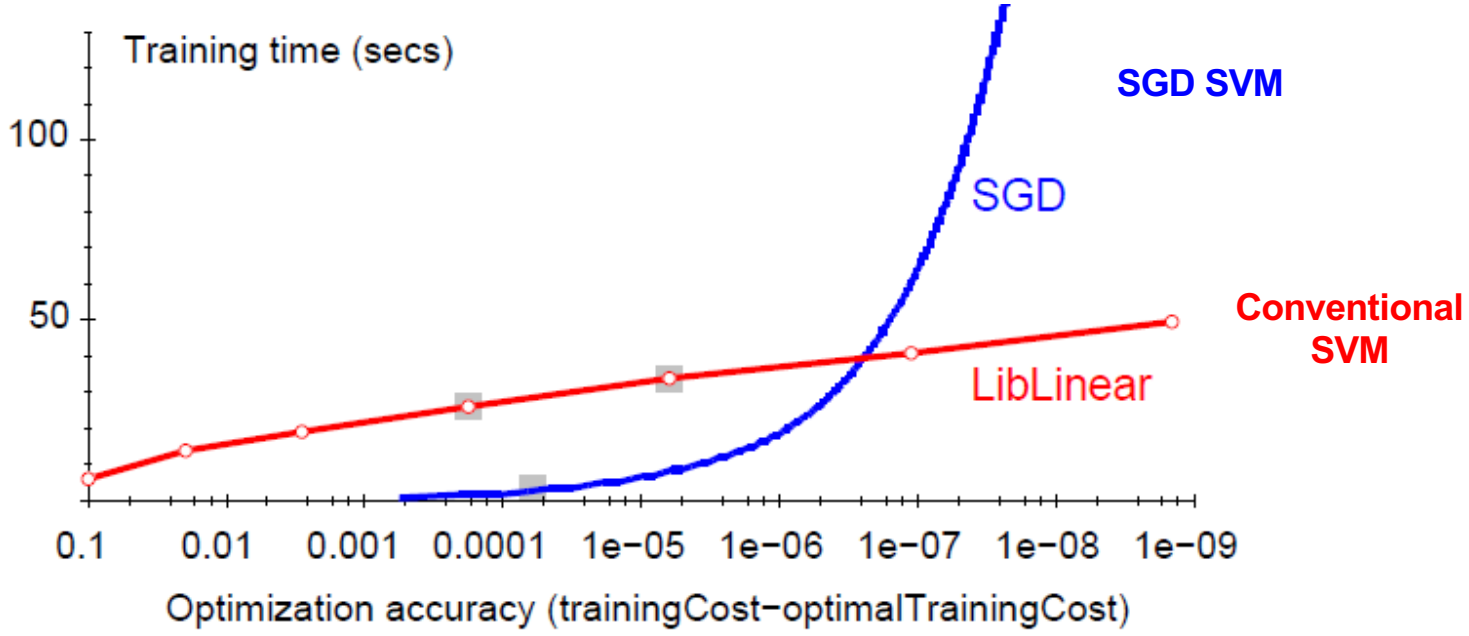
Example: Text categorization

- **Questions:**
  - **(1)** Is **SGD** successful at minimizing ***f(w,b)***?
  - **(2)** How quickly does **SGD** find the min of ***f(w,b)***?
  - **(3)** What is the error on a test set?

	Training time	*Value of f(w,b)*	*Test error*
Standard SVM	23,642 secs	*0.2275*	*6.02%*
"Fast SVM"	66 secs	0.2278	6.03%
SGD SVM	1.4 secs	*0.2275*	*6.02%*

(1) SGD-SVM is successful at minimizing the value of ***f(w,b)***
(2) SGD-SVM is super fast
(3) SGD-SVM test set error is comparable

Optimization "Accuracy"



Optimization quality: $| f(w,b) - f(w^{opt},b^{opt}) |$

For optimizing *f(w,b) within reasonable* quality *SGD-SVM* is super fast

Learning Rate / Step-size schedule

- **Need to choose learning rate η and t_0**

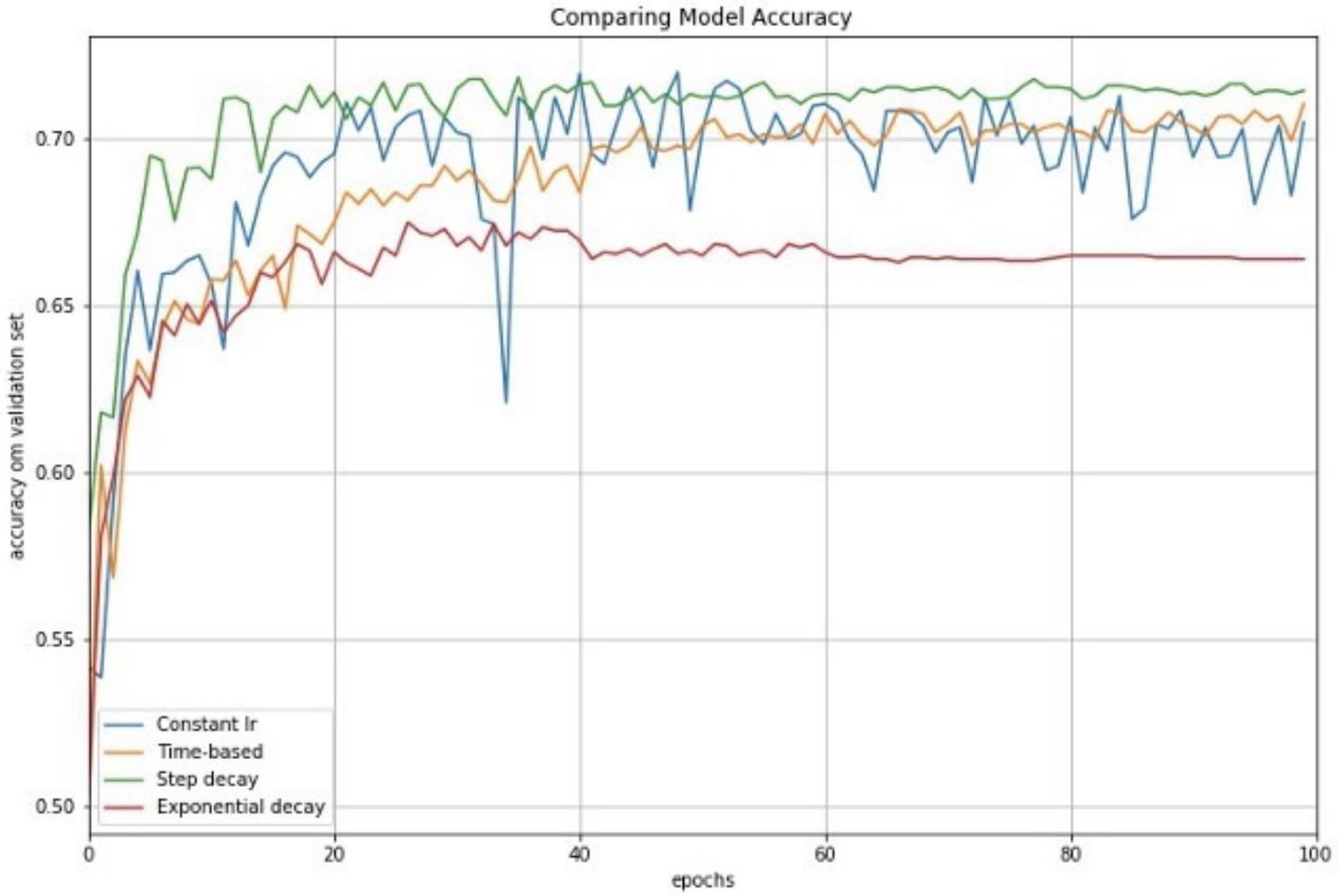
$$w_{t+1} \leftarrow w_t - \frac{\eta_0}{t+t_0}\left(\frac{\partial L(x_i, y_i)}{\partial w}\right); \quad \eta = \frac{\eta_0}{t+t_0}$$

- **Leon suggests:**
  - Choose $\mathbf{t_0}$ so that the expected initial updates are comparable with the expected size of the weights
  - Choose η_0:
    - Select a **small subsample**
    - Try various rates η_0 (e.g., 10, 1, 0.1, 0.01, …)
    - Pick the one that most reduces the cost
    - Use η for next 100k iterations on the full dataset
  - Alternative form:

$$\eta = \frac{\eta_0}{1 + (decay * t)}$$

- Step decay schedule:
  - Drop the learning rate by half every 10 epochs.
    - $\eta = \eta_0 * (drop)^{floor(\frac{t}{t_{drop}})}$

Learning rate comparison



ACCELERATED GRADIENT DESCENT

Stochastic gradient descent

Idea: Perform a parameter update for each training example *x(i)* and label *y(i)*

Update: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x(i), y(i))$

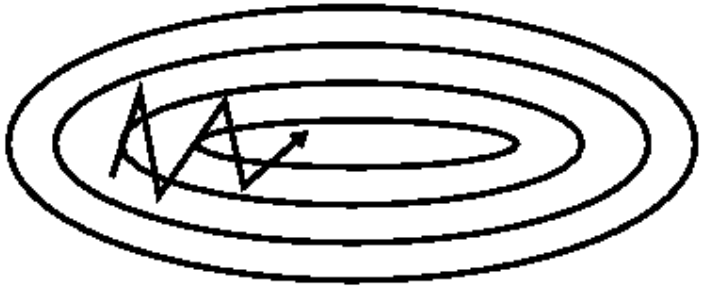
Performs redundant computations for large datasets

Momentum gradient descent

- Idea: Overcome ravine oscillations by momentum
- Update:
  - $v_t = \gamma v_{t-1} + \eta \cdot \nabla_\theta J(\theta)$
  - $\theta = \theta - v_t$

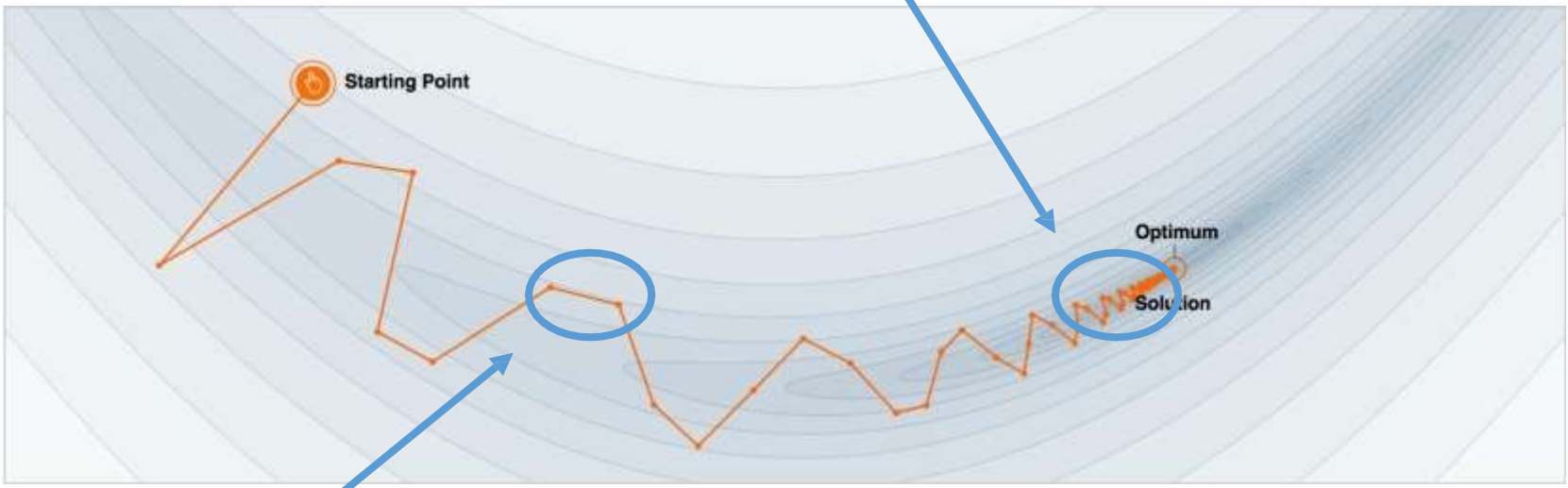
SGD

SGD with momentum



Why Momentum Really Works

The momentum term **reduces updates for dimensions whose gradients change directions**.



The momentum term **increases for dimensions whose gradients point in the same directions.**

Demo : http://distill.pub/2017/momentum/

Nesterov accelerated gradient

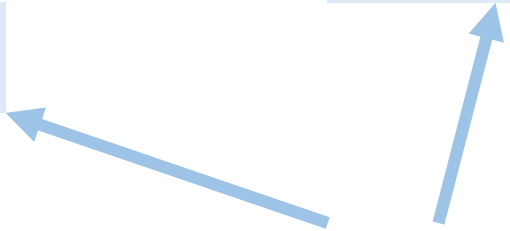
- However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory.

- We would like to have a smarter ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.

- **Nesterov accelerated gradient** gives us a way of it.

Nesterov accelerated gradient

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$



Approximation of the next position of the parameters(predict)

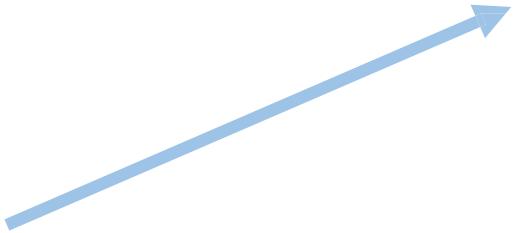
Nesterov accelerated gradient

Approximation of the next position of the parameters' gradient(correction)

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Approximation of the next position of the parameters(predict)

Nesterov accelerated gradient



Blue line : predict

Red line : correction

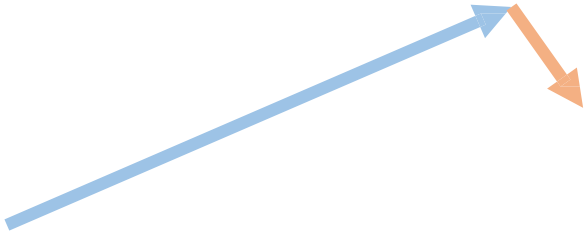
Green line :accumulated gradient

Approximation of the next position of the parameters' gradient(correction)

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Approximation of the next position of the parameters(predict)

Nesterov accelerated gradient



Blue line : predict

Red line : correction

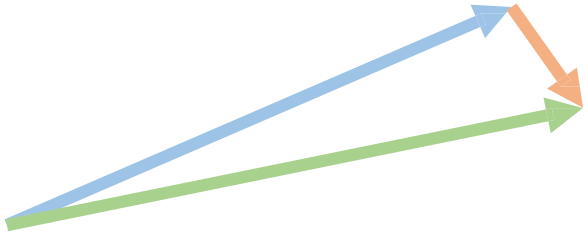
Green line :accumulated gradient

Approximation of the next position of the parameters' gradient(correction)

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Approximation of the next position of the parameters(predict)

Nesterov accelerated gradient



Blue line : predict

Red line : correction

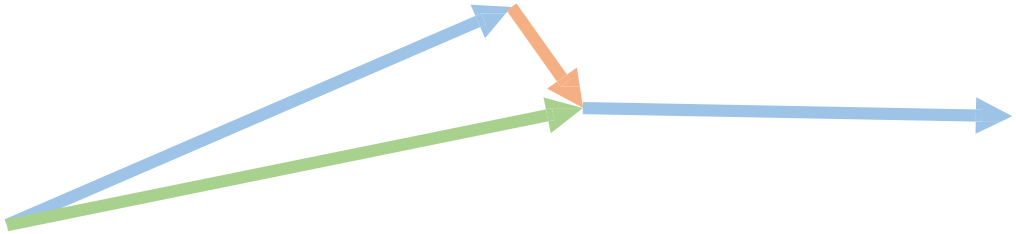
Green line :accumulated gradient

Approximation ofthe next position of the parameters' gradient(correction)

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Approximation of the next position of the parameters(predict)

Nesterov accelerated gradient



Blue line : predict

Red line : correction

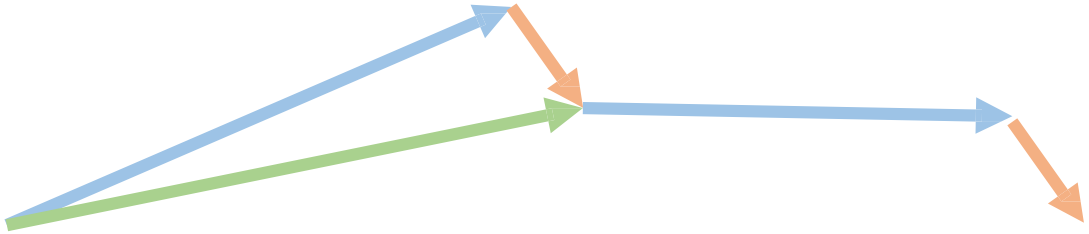
Green line :accumulated gradient

Approximation of the next position of the parameters' gradient(correction)

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Approximation of the next position of the parameters(predict)

Nesterov accelerated gradient



Blue line : predict

Red line : correction

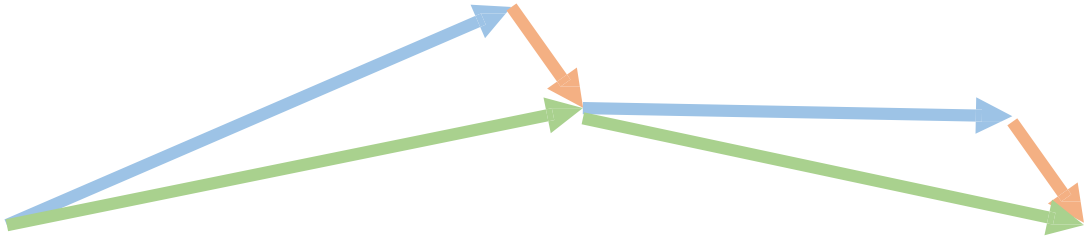
Green line :accumulated gradient

Approximation of the next position of the parameters' gradient(correction)

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Approximation of the next position of the parameters(predict)

Nesterov accelerated gradient



Blue line : predict

Red line : correction

Green line :accumulated gradient

Approximation of the next position of the parameters' gradient(correction)

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Approximation of the next position of the parameters(predict)

Nesterov accelerated gradient

- This anticipatory update **prevents** us from **going too fast** and **results in increased responsiveness**.

- Now , we can adapt our updates to the slope of our error function and **speed up SGD** in turn.

What's next…?

- We also want to adapt our updates to each individual parameter to perform larger or smaller updates **depending on their importance**.
  - Adagrad
  - Adadelta
  - RMSprop
  - Adam

Adagrad

- Adagrad adapts the learning rate to the parameters
  - Performing larger updates for infrequent
  - Performing smaller updates for frequent parameters.

- Ex.
  - Training large-scale neural nets at Google that learned to recognize cats in Youtube videos.

Different learning rate for every parameter

- Previous methods :
  - **we used the same learning rate η for all parameters θ**

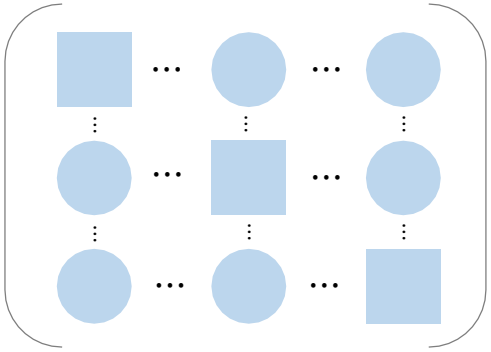
- Adagrad :
  - It uses a different learning rate for every parameter θ_i at every time step t

Adagrad

SGD

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

$G_t \in \mathbb{R}^{d \times d}$ =



Adagrad modifies the general learning rate η based on the past gradients that have been computed for θ_i

Adagrad

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$$g_{t,i} = \nabla_\theta J(\theta_i)$$

Vectorize

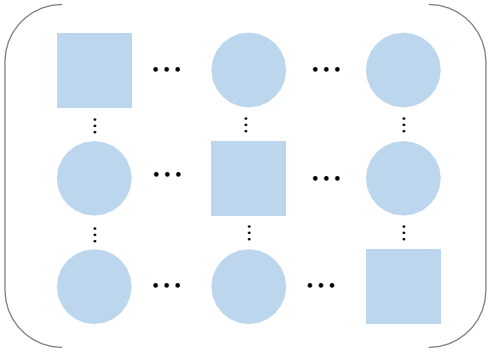
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

Adagrad

SGD

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

$G_t \in \mathbb{R}^{d \times d}$ =



G_t is a diagonal matrix where each diagonal element (i,i) is the sum of the squares of the gradients θ_i up to time step t.

$$G_{t,ii} = \sum_{k=1}^{t} g_{t,i}^2$$

$$g_{t,i} = \nabla_\theta J(\theta_i)$$

Adagrad

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

Vectorize

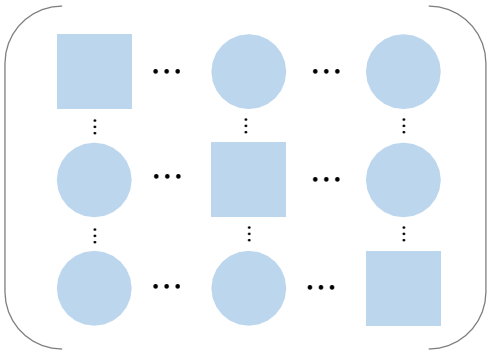
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

Adagrad

SGD

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

$G_t \in \mathbb{R}^{d \times d}$ =



ε is a smoothing term that avoids division by zero (usually on the order of 1e − 8).

Adagrad

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$$g_{t,i} = \nabla_\theta J(\theta_i)$$

Vectorize

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

Adagrad's advantages

- Advantages :
  - It is well-suited for dealing with sparse data.
  - It greatly improved the robustness of SGD.
  - It eliminates the need to manually tune the learning rate.

Adagrad's disadvantage

- Disadvantage :
  - Main weakness is its accumulation of the squared gradients in the denominator.

Adagrad's disadvantage

- The disadvantage causes the learning rate to shrink and become infinitesimally small. The algorithm can no longer acquire additional knowledge.

- The following algorithms aim to resolve this flaw.
  - Adadelta
  - RMSprop
  - Adam

Adadelta

- The expected square sum of gradients is recursively defined as a decaying average of all past squared gradients.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

  - $E[g^2]_t$: The running average at timestep t.
  - γ : A fraction similarly to the Momentum term, around 0.9

Adadelta

Adagrad

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

SGD

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Adadelta

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Adadelta

Adagrad

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

SGD

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Replace the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$

Adadelta

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Adadelta

Adagrad

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

SGD

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Replace the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$

Adadelta

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Adadelta

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$

Update units should have the same hypothetical units

- The units in this update do not match and the update should have the same hypothetical units as the parameter.
  - As well as in SGD, Momentum, or Adagrad

- To realize this, first defining another exponentially decaying average

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2$$

Adadelta update rule

- Replacing the learning rate η in the previous update rule with $RMS[\Delta\theta]_{t-1}$ finally yields the Adadelta update rule:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

- Note : **we do not even need to set a default learning rate**

RMSprop

RMSprop and Adadelta have both been developed independently around the same time to resolve Adagrad's radically diminishing learning rates.

RMSprop

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

RMSprop

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients.

RMSprop

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

Adam

- Adam's feature :
  - Storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop
  - Keeping an exponentially decaying average of past gradients m_t, similar to momentum.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$ → The first moment (the mean)

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$ → The second moment (the uncentered variance)

Adam

- As m_t and v_t are initialized as vectors of 0's, they are biased towards zero.
  - Especially during the initial time steps
  - Especially when the decay rates are small
    - (i.e. β1 and β2 are close to 1).

- Counteracting these biases in Adam

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

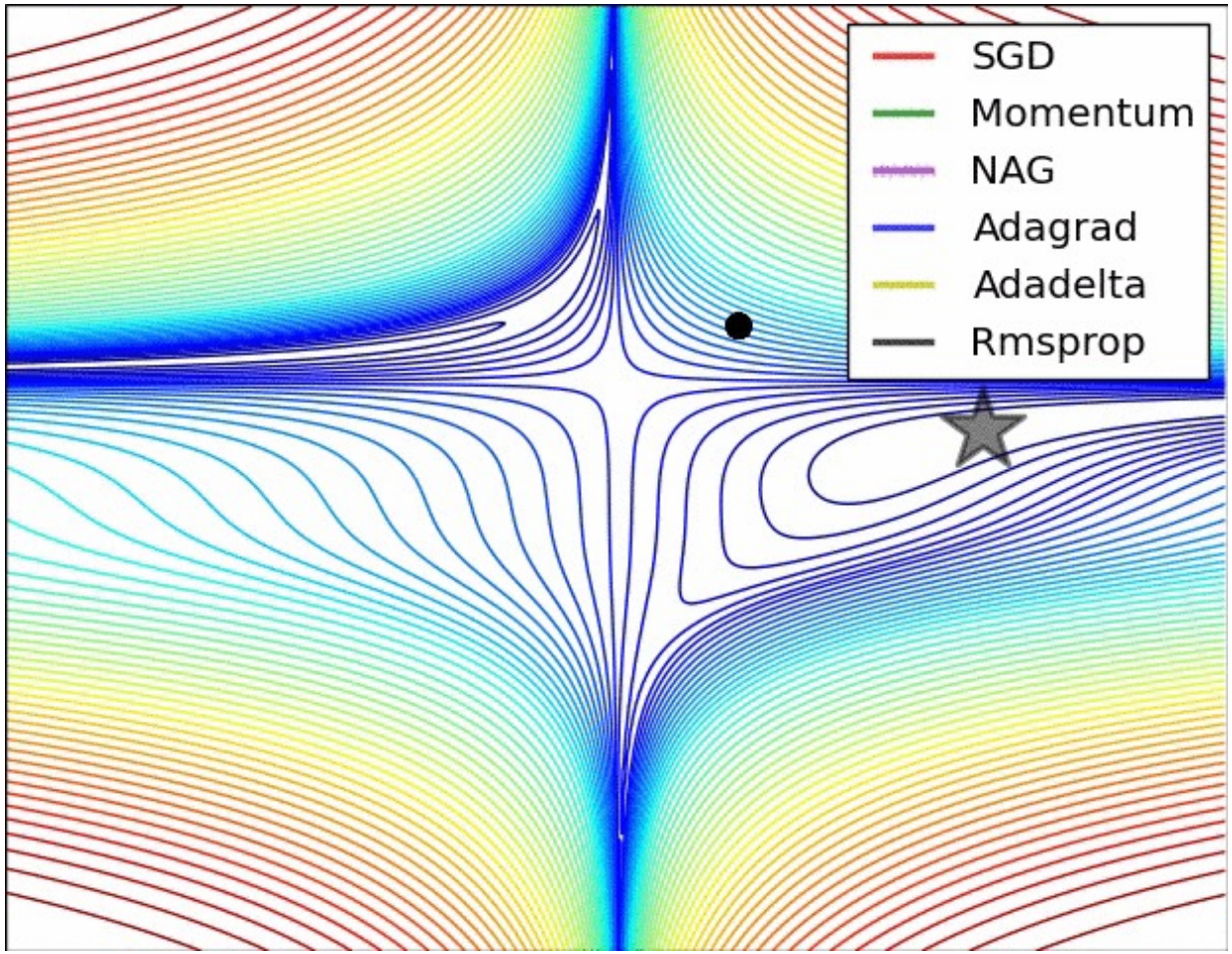
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Adam

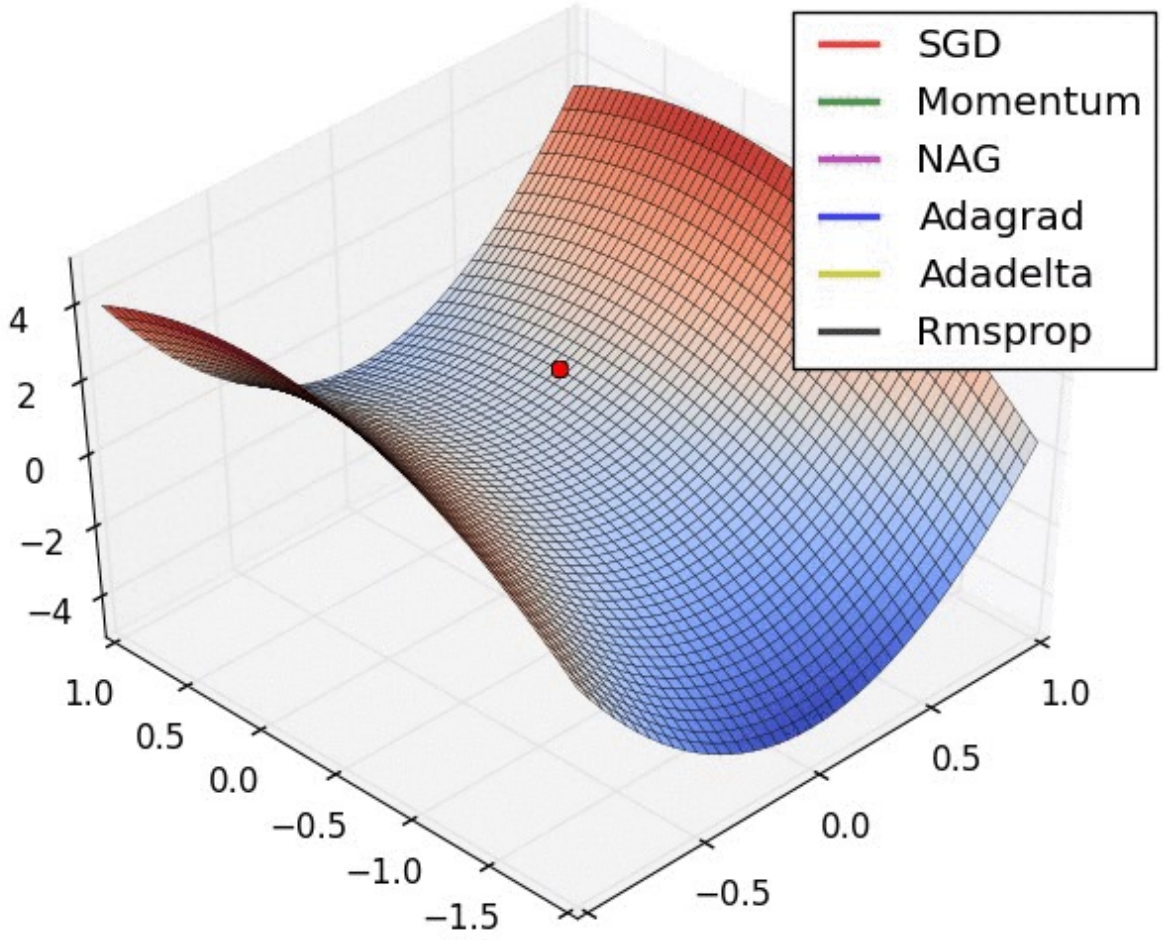
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

Note : default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ε

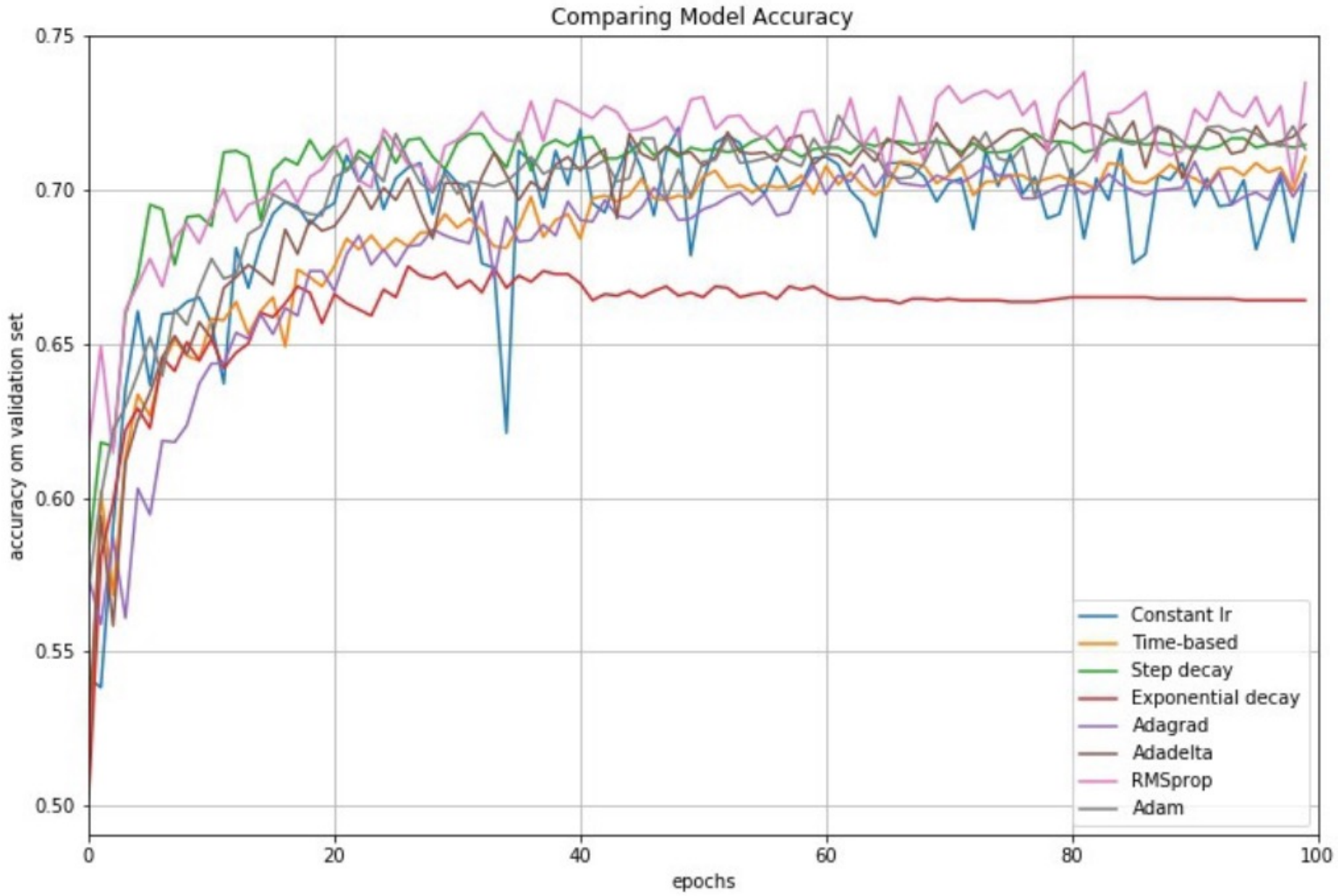
Visualization



Visualization



Enhancements comparison



Summary

- There are two main ideas at play:
  - Momentum : Provide consistency in update directions by incorporating past update directions.
  - Adaptive gradient : Scale the scale updates to individual variables using the second moment in that direction.
  - This also relates to adaptively altering step length for each direction.

THEORETICAL GUARRANTEES

Convergence Rate and Assumptions

A sequence $\{x^k\}$ is said to convergence at the rate γ^k, if:

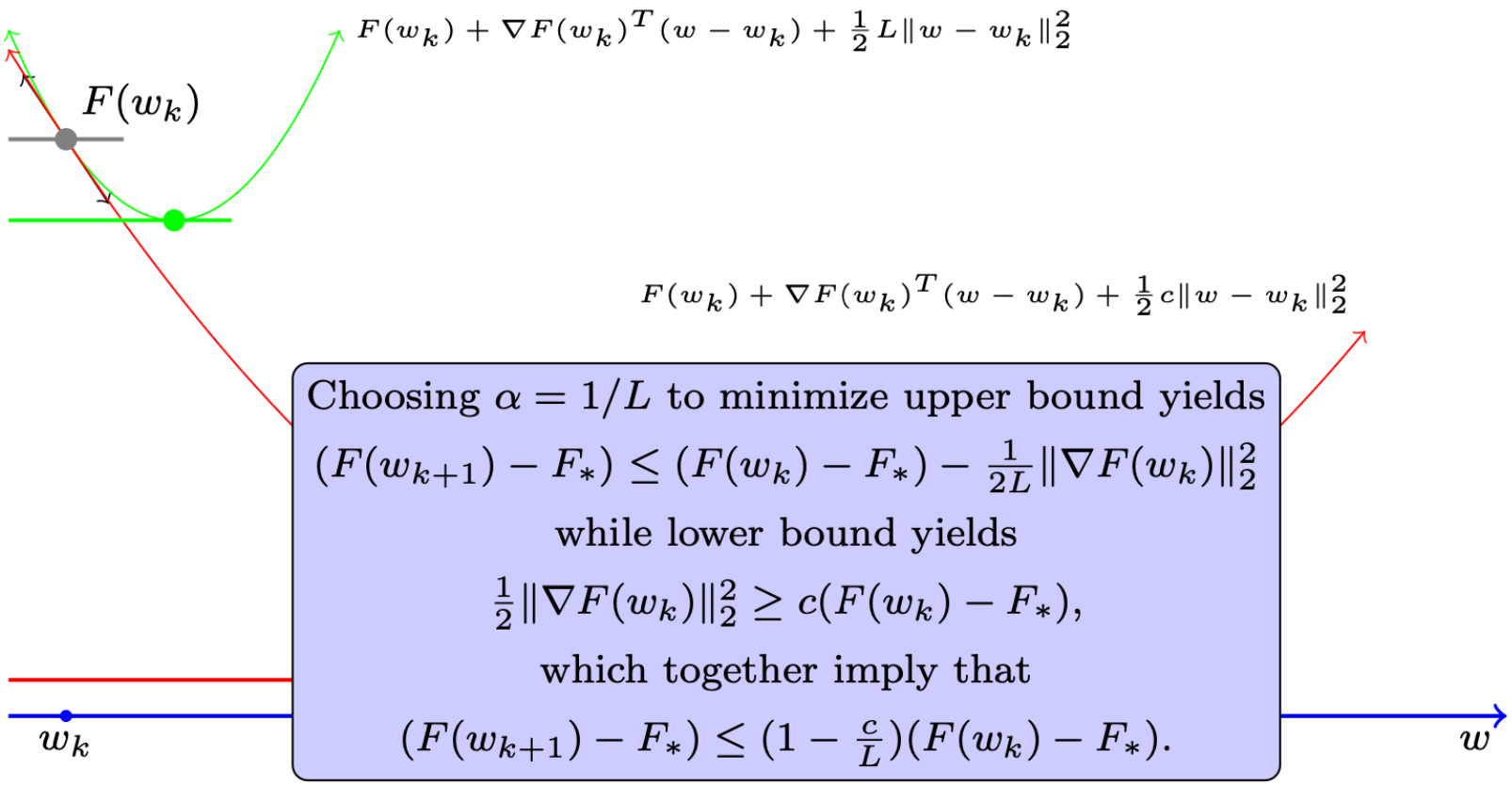
$$\|\mathbf{x}^{k+1} - \mathbf{x}^*\| \leq \gamma \|\mathbf{x}^k - \mathbf{x}^*\| \quad (\Rightarrow \|\mathbf{x}^k - \mathbf{x}^*\| \leq \gamma^k \|\mathbf{x}^0 - \mathbf{x}^*\|),$$

Assumption $\langle L/c \rangle$

The objective function $F : \mathbb{R}^d \to \mathbb{R}$ is

- *c-strongly convex ($\Rightarrow$ unique minimizer) and*
- *L-smooth (i.e., ∇F is Lipschitz continuous with constant L).*

Gradient Descent Convergence



Convergence Rate and Computational Complexity

Overall Complexity (ϵ) = Convergence Rate^{-1}(ϵ) $*$ Complexity of each iteration

	Strongly Convex + Smooth			Convex + Smooth		
	Convergence Rate	Complexity of each iteration	Overall Complexity	Convergence Rate	Complexity of each iteration	Overall Complexity
GD	$O\left(\exp\left(-\frac{t}{Q}\right)\right)$	$O(n \cdot d)$	$O\left(nd \cdot Q \cdot \log\left(\frac{1}{\epsilon}\right)\right)$	$O\left(\frac{\beta}{t}\right)$	$O(n \cdot d)$	$O\left(nd \cdot \beta \cdot \left(\frac{1}{\epsilon}\right)\right)$
SGD	$O\left(\frac{1}{t}\right)$	$O(d)$	$O\left(\frac{d}{\epsilon}\right)$	$O\left(\frac{1}{\sqrt{t}}\right)$	$O(d)$	$O\left(\frac{d}{\epsilon^2}\right)$

SGD Analysis

THEOREM 14.8 *Let $B, \rho > 0$. Let f be a convex function and let $\mathbf{w}^\star \in \operatorname{argmin}_{\mathbf{w}:\|\mathbf{w}\|\leq B} f(\mathbf{w})$. Assume that SGD is run for T iterations with $\eta = \sqrt{\frac{B^2}{\rho^2 T}}$. Assume also that for all t, $\|\mathbf{v}_t\| \leq \rho$ with probability 1. Then,*

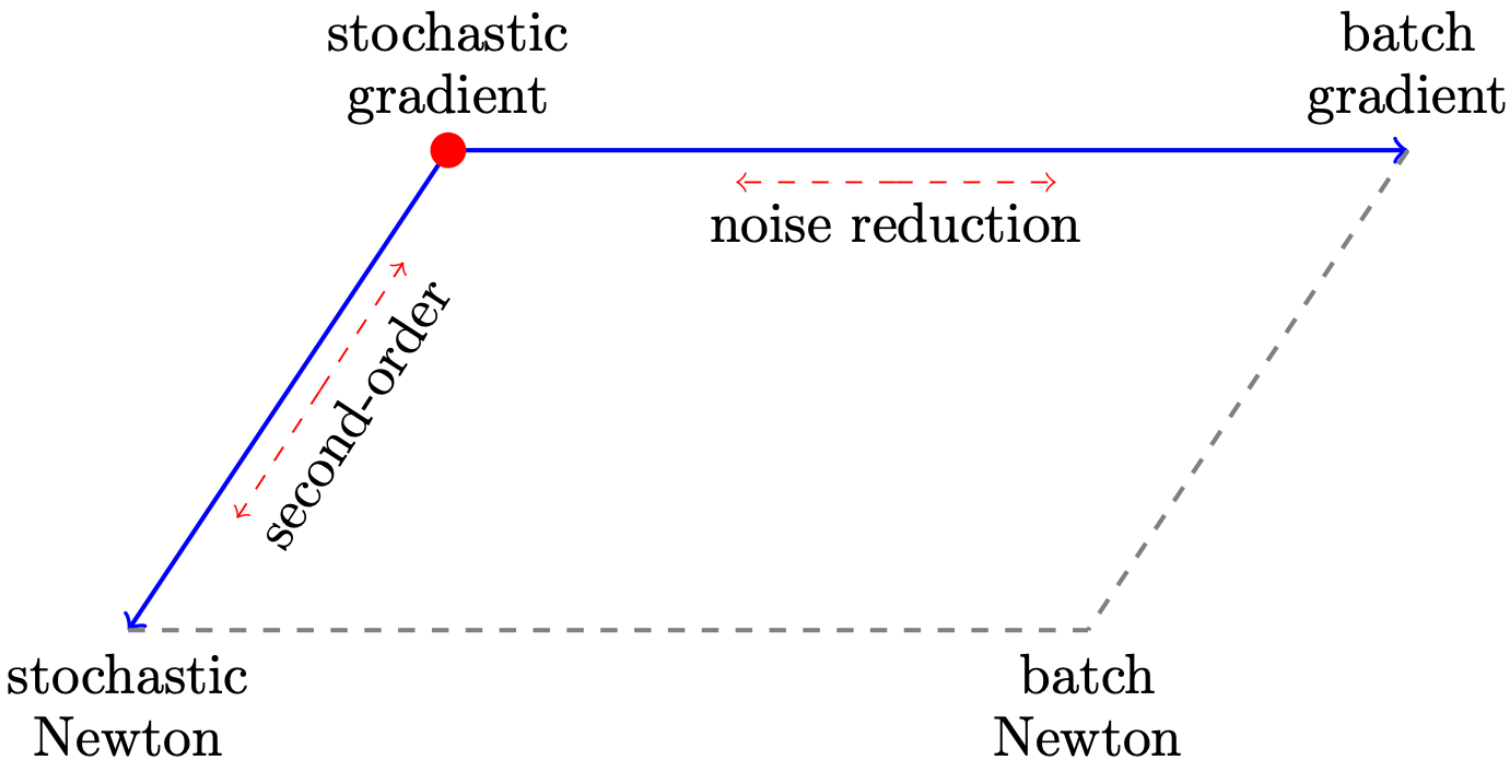
$$\mathbb{E}\left[f(\bar{\mathbf{w}})\right] - f(\mathbf{w}^\star) \leq \frac{B\rho}{\sqrt{T}}.$$

Therefore, for any $\epsilon > 0$, to achieve $\mathbb{E}[f(\bar{\mathbf{w}})] - f(\mathbf{w}^\star) \leq \epsilon$, it suffices to run the SGD algorithm for a number of iterations that satisfies

$$T \geq \frac{B^2\rho^2}{\epsilon^2}.$$

LINEAR RATE METHODS

Improving SGD



Slides taken from Jorge Nocedal

Stochastic Averaged Gradient

- **Can we have a rate of $O(\rho^t)$ with only 1 gradient evaluation per iteration?**
  - YES! The stochastic average gradient (SAG) algorithm:
    - Randomly select i_t from $\{1, 2, \ldots, N\}$ and compute $f'_{i_t}(x^t)$.

$$x^{t+1} = x^t - \frac{\alpha^t}{N} \sum_{i=1}^{N} y_i^t$$

    - **Memory**: $y_i^t = \nabla f_i(x^t)$ from the last t where i was selected.
      [Le Roux et al., 2012]
  - Stochastic variant of increment average gradient (IAG).
    [Blatt et al., 2007]
  - Assumes gradients of non-selected examples don't change.
  - Assumption becomes accurate as $||x^{t+1} - x^t|| \to 0$.

Slides taken from Mark Schmidt

SAG Convergence Rate

- If each f_i' is $L-$continuous and f is strongly-convex, with $\alpha_t = 1/16L$ SAG has

$$\mathbb{E}[f(x^t) - f(x^*)] \leqslant \left(1 - \min\left\{\frac{\mu}{16L}, \frac{1}{8N}\right\}\right)^t C,$$

where

$$C = [f(x^0) - f(x^*)] + \frac{4L}{N}\|x^0 - x^*\|^2 + \frac{\sigma^2}{16L}.$$

- Linear convergence rate but only 1 gradient per iteration.
  - For well-conditioned problems, constant reduction per pass:

$$\left(1 - \frac{1}{8N}\right)^N \leq \exp\left(-\frac{1}{8}\right) = 0.8825.$$

  - For ill-conditioned problems, almost same as deterministic method (but N times faster).

SAG Implementation

- Basic SAG algorithm:
  - while(1)
  - Sample i from $\{1, 2, \ldots, N\}$.
  - Compute $f_i'(x)$.
  - $d = d - y_i + f_i'(x)$.
  - $y_i = f_i'(x)$.
  - $x = x - \frac{\alpha}{N} d$.
- Practical variants of the basic algorithm allow:
  - Regularization.
  - Sparse gradients.
  - Automatic step-size selection.
    - Common to use adaptive step-size procedure to estimate L.
  - Termination criterion.
    - Can use $\|x^{t+1} - x^t\|/\alpha = \frac{1}{n} d \approx \|\nabla f(x^t)\|$ to decide when to stop.
  - Acceleration [Lin et al., 2015].
  - Adaptive non-uniform sampling [Schmidt et al., 2013].

SAG Implementation

- Does re-shuffling and doing full passes work better?
  - For classic SG: Maybe?
    - Noncommutative arithmetic-geometric mean inequality conjecture.

      [Recht & Ré, 2012]
  - For SAG: NO.
  - Performance is intermediate between IAG and SAG.
- Can non-uniform sampling help?
  - For classic SG methods, can only improve constants.
  - For SAG, bias sampling towards Lipschitz constants L_i,

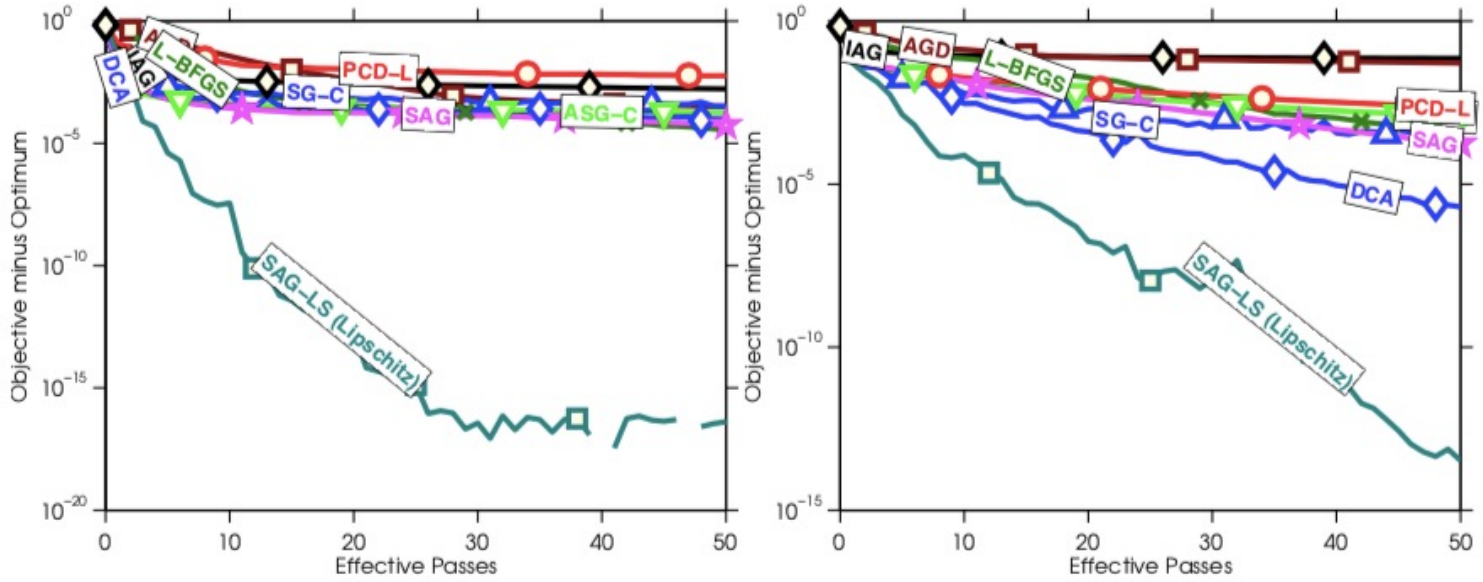
    $$\|\nabla f_i(x) - \nabla f_i(y)\| \leq L_i \|x - y\|.$$

    improves rate to depend on L_{mean} instead of L_{max}.

    (with bigger step size)
  - Adaptively estimate L_i as you go. (see paper/code).
  - Slowly learns to ignore well-classified examples.

SAG with Non-Uniform Sampling

- protein ($n = 145751$, $p = 74$) and sido ($n = 12678$, $p = 4932$)



- Adaptive non-uniform sampling helps a lot.

Stochastic Variance Reduced GD

SVRG algorithm:

- Start with x_0
- for $s = 0, 1, 2 \ldots$
  - $d_s = \frac{1}{N} \sum_{i=1}^{N} f_i'(x_s)$
  - $x^0 = x_s$
  - for $t = 1, 2, \ldots m$
    - Randomly pick $i_t \in \{1, 2, \ldots, N\}$
    - $x^t = x^{t-1} - \alpha_t (f_{i_t}'(x^{t-1}) - f_{i_t}'(x_s) + d_s)$.
  - $x_{s+1} = x^t$ for random $t \in \{1, 2, \ldots, m\}$.

Requires 2 gradients per iteration and occasional full passes,
but only requires storing d_s and x_s.

Practical issues similar to SAG (acceleration versions, automatic step-size/termination, handles sparsity/regularization, non-uniform sampling, mini-batches).

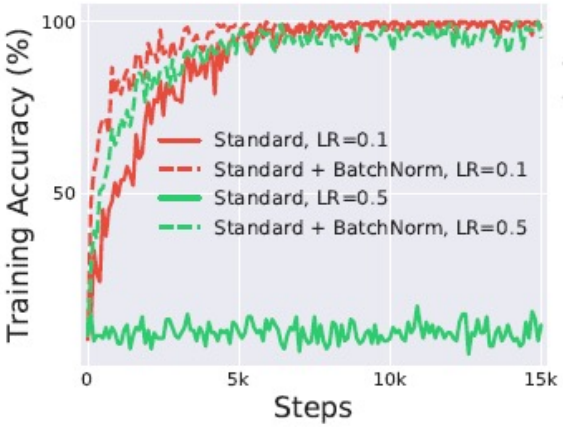
BATCH NORMALIZATION

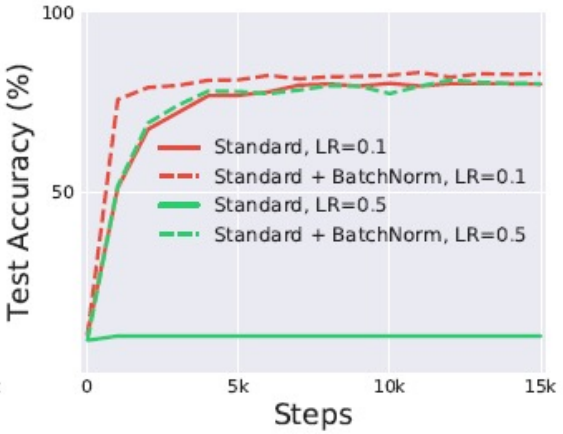
Slides taken from Jude Shavlik: http://pages.cs.wisc.edu/~shavlik/cs638_cs838.html

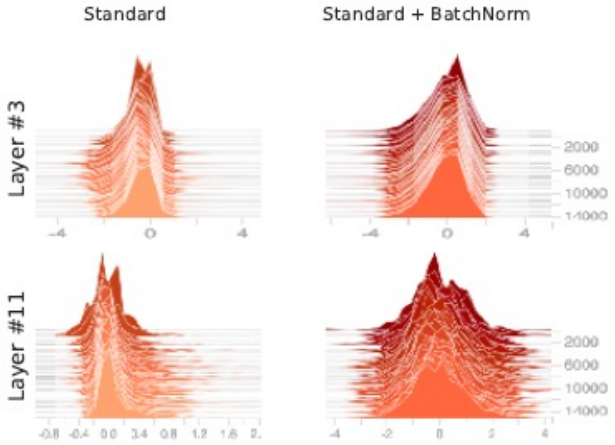
Batch normalization: Other benefits in practice

- BN reduces training times. (Because of less Covariate Shift, less exploding/vanishing gradients.)
- BN reduces demand for regularization, e.g. dropout or L2 norm.
  - Because the means and variances are calculated over batches and therefore every normalized value depends on the current batch. I.e. the network can no longer just memorize values and their correct answers.)
- BN allows higher learning rates. (Because of less danger of exploding/vanishing gradients.)
- BN enables training with saturating nonlinearities in deep networks, e.g. sigmoid. (Because the normalization prevents them from getting stuck in saturating ranges, e.g. very high/low values for sigmoid.)

Internal Covariate Shift







Why the naïve approach Does not work?

- Normalizes layer inputs to zero mean and unit variance. *whitening*.
- Naive method: Train on a batch. Update model parameters. Then normalize. Doesn't work: Leads to exploding biases while distribution parameters (mean, variance) don't change.

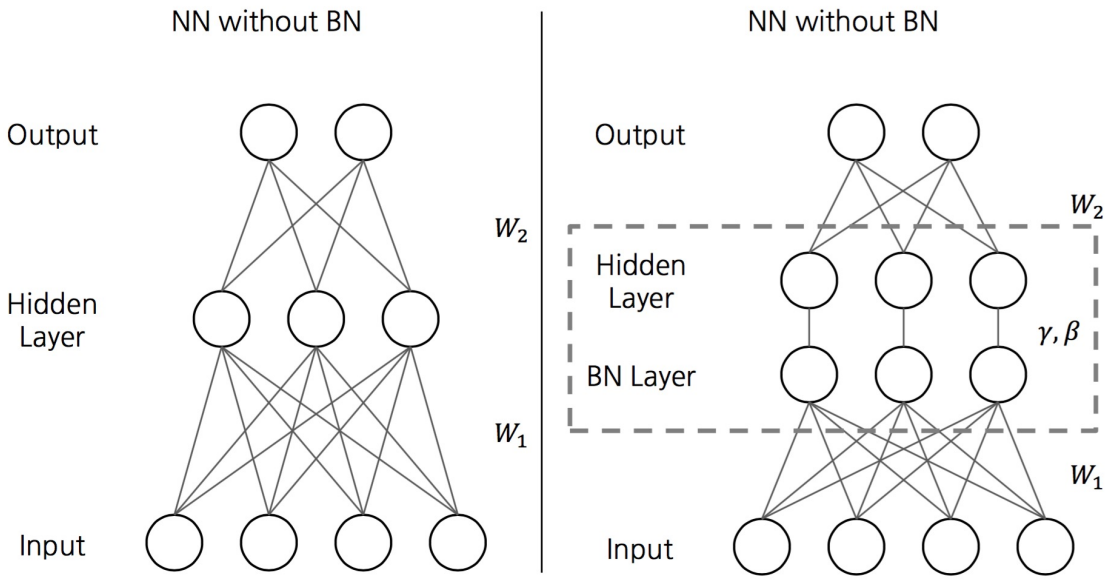
  - If we do it this way gradient always ignores the effect that the normalization for the next batch would have
  - i.e. : "The issue with the above approach is that the gradient descent optimization does not take into account the fact that the normalization takes place"

The proposed solution:

To add an extra regularization layer

we introduce, for each activation $x^{(k)}$, a pair of parameters $\gamma^{(k)}, \beta^{(k)}$, which scale and shift the normalized value:

$$y^{(k)} = \gamma^{(k)} \widehat{x}^{(k)} + \beta^{(k)}.$$



A new layer is added so the gradient can “see” the normalization and make adjustments if needed.

Algorithm Summary: Normalization via Mini-Batch Statistics

- Each feature (component) is normalized individually
- Normalization according to:
  - componentNormalizedValue = (componentOldValue - E[component]) / sqrt(Var(component))
- A new layer is added so the gradient can "see" the normalization and made adjustments if needed.
  - The new layer has the power to learn the identity function to de-normalize the features if necessary!
  - Full formula: newValue = gamma * componentNormalizedValue + beta (gamma and beta learned per component)
- E and Var are estimated for each mini batch.
- BN is fully differentiable.

The Batch Transformation: formally from the paper.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

The full algorithm as proposed in the paper

Input: Network N with trainable parameters Θ;
subset of activations $\{x^{(k)}\}_{k=1}^{K}$

Output: Batch-normalized network for inference, $N_{\text{BN}}^{\text{inf}}$

1: $N_{\text{BN}}^{\text{tr}} \leftarrow N$ // Training BN network
2: **for** $k = 1 \ldots K$ **do**
3: Add transformation $y^{(k)} = \text{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to $N_{\text{BN}}^{\text{tr}}$ (Alg. 1)
4: Modify each layer in $N_{\text{BN}}^{\text{tr}}$ with input $x^{(k)}$ to take $y^{(k)}$ instead
5: **end for**
6: Train $N_{\text{BN}}^{\text{tr}}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$
7: $N_{\text{BN}}^{\text{inf}} \leftarrow N_{\text{BN}}^{\text{tr}}$ // Inference BN network with frozen // parameters
8: **for** $k = 1 \ldots K$ **do**
9: // For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.
10: Process multiple training mini-batches $\mathcal{B}$, each of size m, and average over them:

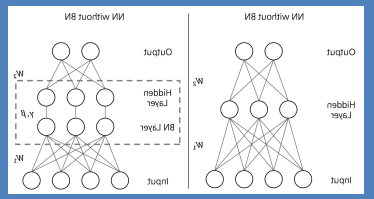
$$\text{E}[x] \leftarrow \text{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$
$$\text{Var}[x] \leftarrow \frac{m}{m-1}\text{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

11: In $N_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma,\beta}(x)$ with $y = \frac{\gamma}{\sqrt{\text{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\text{E}[x]}{\sqrt{\text{Var}[x]+\epsilon}}\right)$
12: **end for**

Algorithm 2: Training a Batch-Normalized Network

Alg 1 (previous slide)

Architecture modification



Note that BN(x) is different during test…

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

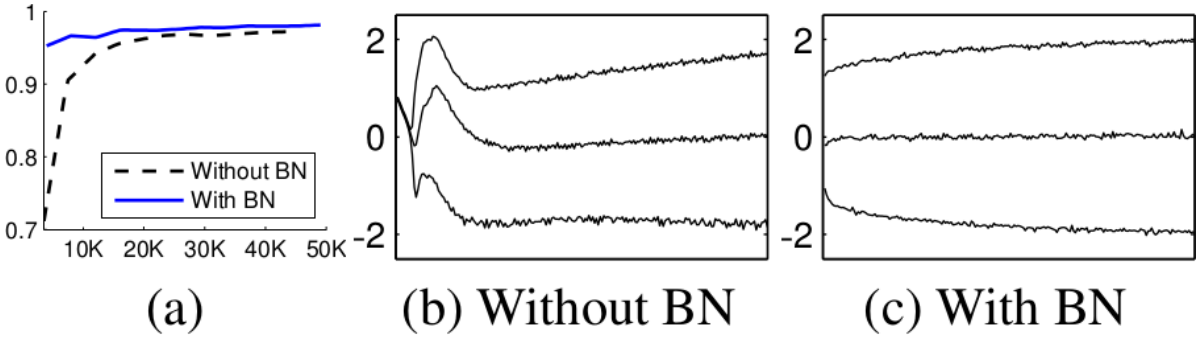
Vs.

$$\text{Var}[x] \leftarrow \frac{m}{m-1}\text{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

Batch normalization: Other benefits in practice

- BN reduces training times. (Because of less Covariate Shift, less exploding/vanishing gradients.)
- BN reduces demand for regularization, e.g. dropout or L2 norm.
  - Because the means and variances are calculated over batches and therefore every normalized value depends on the current batch. I.e. the network can no longer just memorize values and their correct answers.)
- BN allows higher learning rates. (Because of less danger of exploding/vanishing gradients.)
- BN enables training with saturating nonlinearities in deep networks, e.g. sigmoid. (Because the normalization prevents them from getting stuck in saturating ranges, e.g. very high/low values for sigmoid.)

Batch normalization: Better accuracy , faster.



(a) (b) Without BN (c) With BN

BN applied to MNIST (a), and activations of a randomly selected neuron over time (b, c), where the middle line is the median activation, the top line is the 15th percentile and the bottom line is the 85th percentile.

References:

- SGD proof by Yuri Nesterov.
- MMDS http://www.mmds.org/
- *Blog of Sebastian Ruder http://ruder.io/optimizing-gradient-descent/*
- *Learning rate comparison https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1*