Stochastic Gradient Descent

Shao Tang

February 24, 2017

Motivation

Kaggle: Avito Context Ad Clicks

- Objective: predict if context ads will earn a user's click.
- Response (binary): if there was a click on this ad.
- Features (numerical and categorical).

Main Challenges:

- The data set (>50GB) contains over 200 millions of observations.
- The number of categories for each categorical variable is not known in advance.
- large n and large p.

Problem Formulation

- Cost function over the data-generating distribution p_{data}:

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y).$$

- Cost function over the training set:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim \hat{p}_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y) = \frac{1}{n}\sum_{i=1}^{n} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}),$$

where L is the per-example loss function, $f(\boldsymbol{x};\boldsymbol{\theta})$ and y are the predicted and observed output given input $\boldsymbol{x}$, respectively, and $\hat{p}_{data}$ is the *empirical distribution.*

Stochastic Gradient Descent (SGD)

- Exact gradient:

$$\begin{aligned}\boldsymbol{g} &= \nabla_{\boldsymbol{\theta}} J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{data}} \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y) \\ &= \sum_{\boldsymbol{x}} \sum_{y} p_{data}(\boldsymbol{x}, y) \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y).\end{aligned}$$

- An *unbiased* estimator of the exact gradient of the generalization error: (iid $\boldsymbol{x}$)

$$\hat{\boldsymbol{g}} = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}),$$

where $\{(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$ is a minibatch of examples sampled from the data-generating distribution p_{data} with m as the sample size.

Examples

- Linear regression:

$$\hat{\boldsymbol{g}}_i = [\boldsymbol{x}^{(i)T}\boldsymbol{\theta} - y^{(i)}]\boldsymbol{x}^{(i)},$$

we can verify that the gradient obtained from all samples can be summed up to the full gradient:

$$\sum_{i=1}^{n} \hat{\boldsymbol{g}}_i = \sum_{i=1}^{n} (\boldsymbol{x}^{(i)}\boldsymbol{x}^{(i)T}\boldsymbol{\theta} - y^{(i)}\boldsymbol{x}^{(i)}) = \boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}).$$

- This holds for any **additive** loss function, i.e.,

$$L(f(\boldsymbol{x};\boldsymbol{\theta}), y) = \sum_{i=1}^{n} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}).$$

Basic SGD

Algorithm 1 Basic SGD. (random shuffle all samples before the start of each epoch)

Require: Learning rate ε_k

Require: Initial parameter θ

1: **while** not converged **do**

2: Sample a minibatch of m examples from the training set as $\{(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$.

3: Compute gradient estimate:

$$\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}).$$

4: Update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon_k \hat{\boldsymbol{g}}$.

5: **end while**

Why SGD?

- If the dataset is highly redundant ($N \gg p$):
  - Rather than computing the full gradient, it is better to update the coefficients based on a sub-sample of original observations.
  - The extreme version of this approach updates coefficients after each observation (so called "online").

- Mini-batches are usually better than online.
  - Computing the gradient for many cases uses matrix-matrix multipliers are very efficient, especially on GPUs.

Randomized coordinate descent

- Instead of sampling from the n observations, we can sample the coordinates of the whole gradient vector $\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y) \in \mathbb{R}^p$ when p is extremely large.

- In the situation of large n and large p, we can do a two-way sampling, to reduce the computational burden.

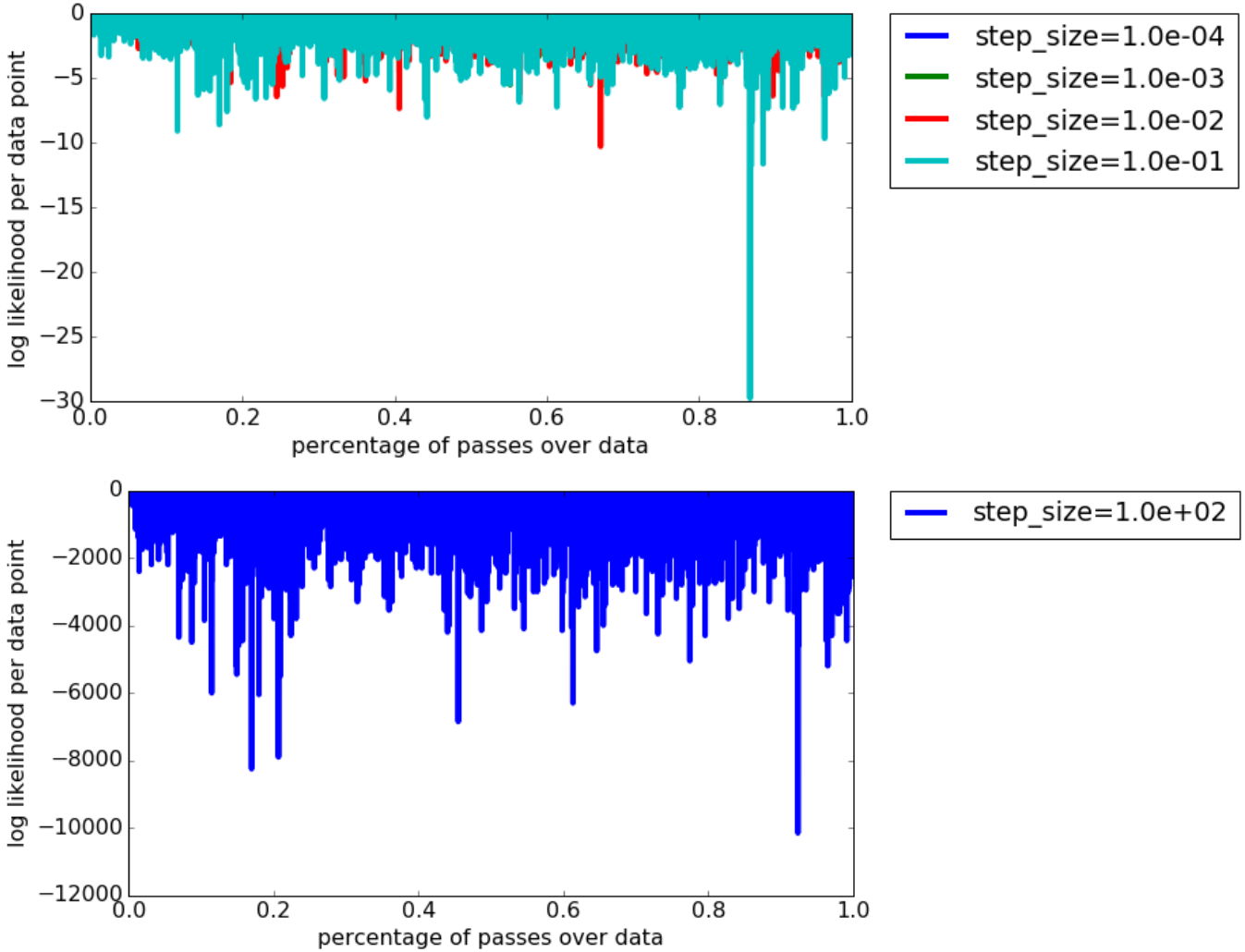
Effects of the Step Size

- SGD gradient estimator introduces a source of **noise** that does *NOT* vanish even at a minimum.

- "The learning rate(step size) may be chosen by trial and error, but it is usually best to choose it by monitoring learning curves that plot the objective function as a function of time(number of iterations). This is more of an **art** than a science, and most guidance on this subject should be regarded with some skepticism."

Effects of the Step Size

Logistic regression on a real world data: amazon review prediction. ($n \sim 50000$, $p \sim 200$)



Effects of the Step Size

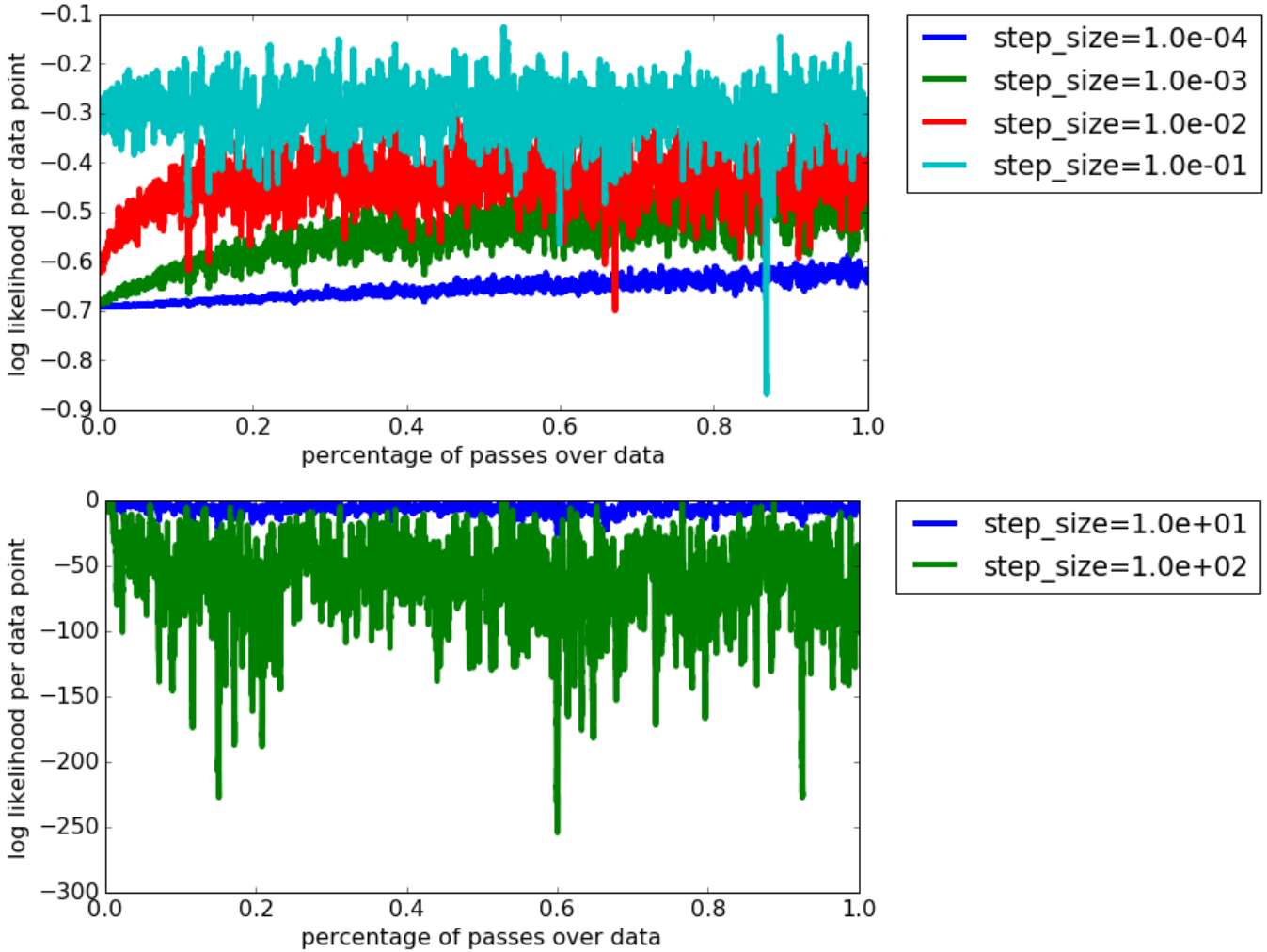


Figure: Batch size $m = 1$, smoothing width 100. (Smoothing is for data visualization only.)

Effects of the Step Size

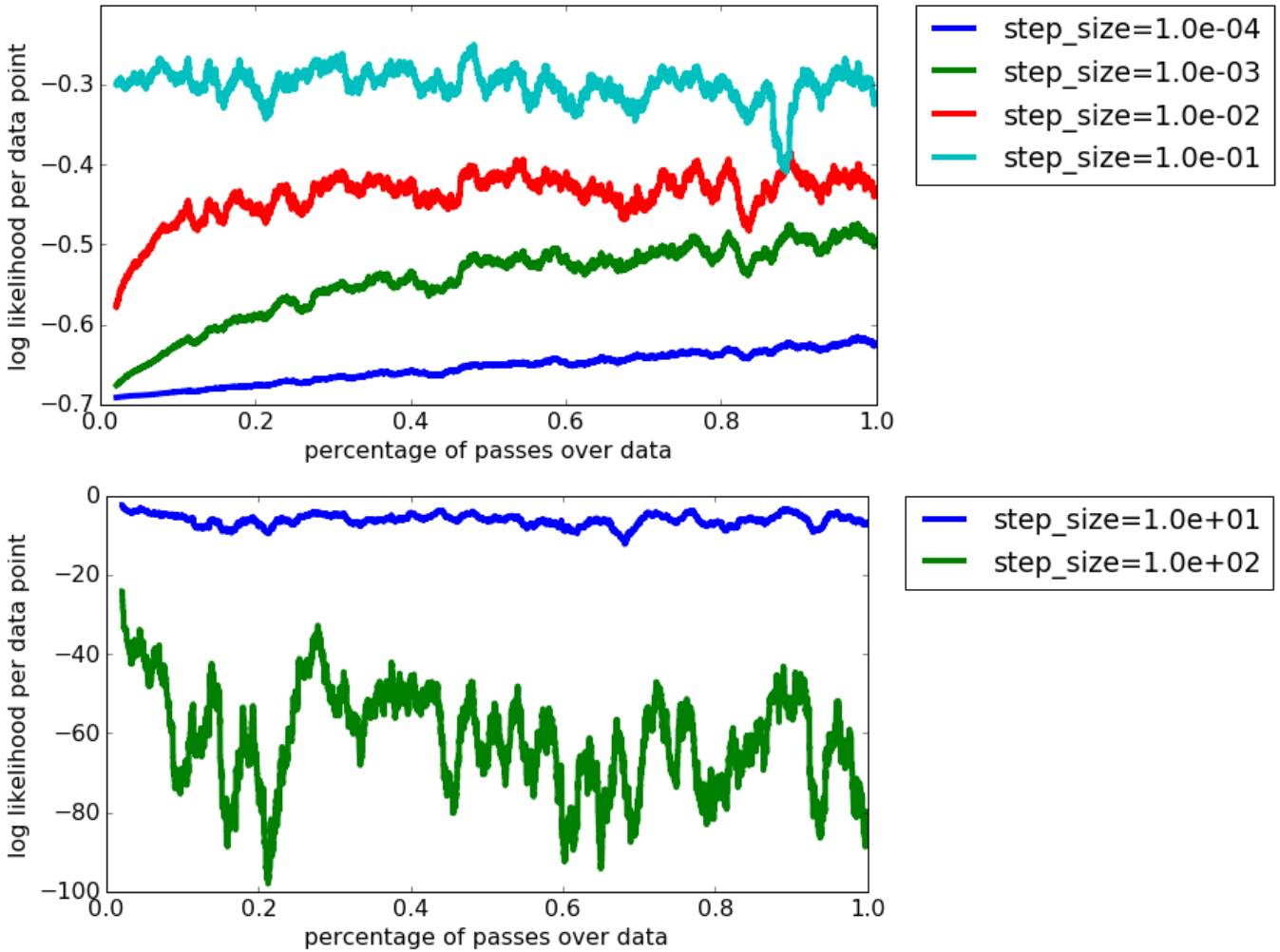


Figure: Batch size $m = 1$, smoothing width 1000.

MiniBatch SGD Step Size

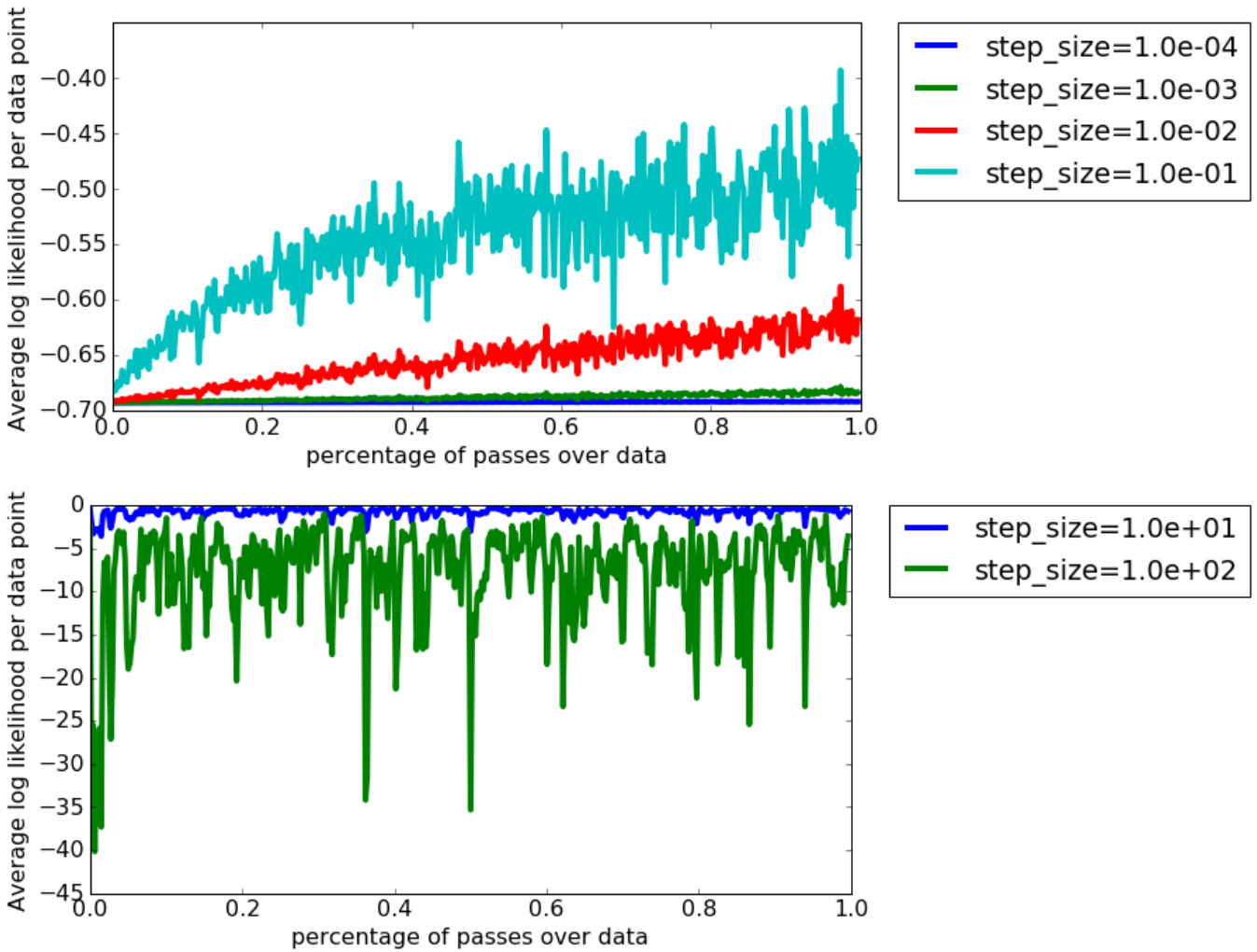


Figure: Batch size $m = 100$, non-smoothing.

MiniBatch SGD and GD

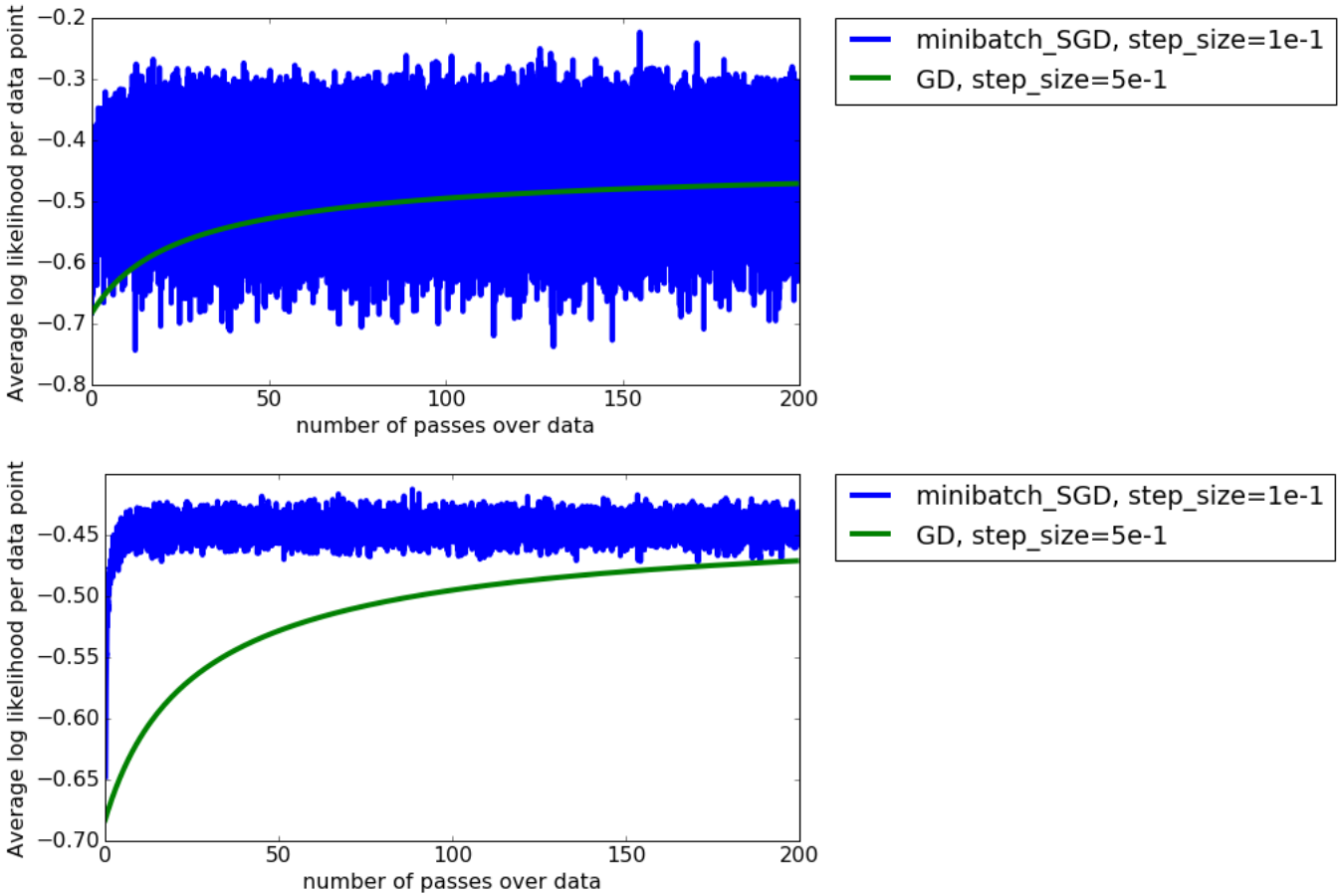


Figure: Batch size $m = 1$ and $m = n$, smoothing width 1 (top) and 50 (bottom).

Empirical Conclusions

- The convergence is strongly affected by the *pre-fixed* step size.
- Increasing the batch size helps function value convergence.
- The minibatch SGD often improves function value faster than the Gradient Descent (GD) method.
- SGD can be directly used on the streamed data (online learning).
- "Don't trust the last coefficient estimate".

SGD with momentum

Algorithm 2 SGD with momentum.

Require: Learning rate ε, momentum parameter α

Require: Initial parameter θ, initial velocity $\boldsymbol{v}$

1: **while** not converged **do**

2: Sample a minibatch of m examples from the training set as $\{(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$.

3: Compute gradient estimate:

$$\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}).$$

4: Velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \varepsilon \hat{\boldsymbol{g}}$.

5: Update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

6: **end while**

Intuition of the momentum

Imagine a ball on the error surface $E(x, y)$. The location of the ball (x, y) represents the coefficient vector.

- Initially, the ball follows the gradient. However, it no longer does steepest descent when it has the velocity.
- Its momentum keeps it going in the previous direction.
- The momentum damps oscillations in directions of high curvature by combining gradients with opposite signs.
- It helps build up speed in directions with a consistent gradient.

SGD with Nesterov momentum

Algorithm 3 SGD with Nesterov momentum.

Require: Learning rate ε, momentum parameter α

Require: Initial parameter θ, initial velocity $\boldsymbol{v}$

1: **while** not converged **do**

2: Sample a minibatch of m examples from the training set as $\{(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$.

3: Update :$\widetilde{\theta} \leftarrow \theta + \alpha \boldsymbol{v}$.

4: Compute gradient estimate:

$$\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_{\widetilde{\theta}} L(f(\boldsymbol{x}^{(i)}; \widetilde{\theta}), y^{(i)}).$$

5: Velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \varepsilon \hat{\boldsymbol{g}}$.

6: Update: $\theta \leftarrow \theta + \boldsymbol{v}$.

7: **end while**

SGD with Adaptive Learning Rates

- AdaGrad: individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all of their historical squared values.

- Adadelta: is an extension of AdaGrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w.

SGD with Adaptive Learning Rates

- RMSProp: modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average.

- Adam (adaptive moments): In addition to storing an exponentially decaying average of past squared gradients like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients.

Follow-the-Regularized-Leader

- However, the online learning is not particularly effective at producing sparse models.
- Let $L_i(\boldsymbol{\theta}) \triangleq L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$, and $\boldsymbol{g}_i = \nabla_{\boldsymbol{\theta}} L_i(\boldsymbol{\theta}_i)$. Considering

$$\boldsymbol{\theta}_t = \operatorname{argmin}_{\boldsymbol{\theta}} \sum_{i=1}^{t-1} \langle \boldsymbol{g}_i, \boldsymbol{\theta} \rangle + \frac{1}{2\varepsilon} \|\boldsymbol{\theta}\|_2^2, \qquad (\star)$$

we have

$$\boldsymbol{\theta}_t = -\varepsilon \sum_{i=1}^{t-1} \boldsymbol{g}_i.$$

It is easy to see that

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \varepsilon \boldsymbol{g}_i,$$

which is the online learning and can be viewed as optimization with the linearization trick.

Follow-the-Regularized-Leader

- With $u_i(\theta) = \langle \boldsymbol{g}_i, \theta \rangle$ and $R(\theta) = \frac{1}{2\varepsilon}\|\theta\|_2^2$, $(\star)$ belongs to the Follow-the-Regularized-Leader algorithm:

$$\theta_t = \text{argmin}_\theta \sum_{i=1}^{t-1} u_i(\theta) + R(\theta).$$

- Adding the ℓ_1 penalty, $R(\theta) = \frac{1}{2\varepsilon}\|\theta\|_2^2 + \frac{\lambda}{2}\|\theta\|_1$, we have

$$\theta_t = \text{argmin}_\theta \langle \sum_{i=1}^{t-1} \boldsymbol{g}_i, \theta \rangle + \frac{1}{2\varepsilon}\|\theta\|_2^2 + \frac{\lambda}{2}\|\theta\|_1.$$

$$\theta_t = \text{argmin}_\theta \|\theta + \varepsilon \boldsymbol{g}_i\|_2^2 + \lambda\varepsilon\|\theta\|_1,$$

which can be solved by soft-thresholding.

Follow-the-Regularized-Leader

- We have

$$\theta_{t,j} = \begin{cases} 0 & |\sum_{i=1}^{t-1} g_{i,j}| < \lambda \\ -\varepsilon[\sum_{i=1}^{t-1} g_{i,j} - \lambda \text{sign}(\sum_{i=1}^{t-1} g_{i,j})] & \text{otherwise} \end{cases}$$

- In each step, we only need to compute $\boldsymbol{g}_i$ and update an auxiliary vector $\boldsymbol{z}_t = \sum_{i=1}^{t-1} \boldsymbol{g}_i$ to obtain the coefficients update.

- FTRL can be easily extended to FTRL-Proximal algorithm with coordinate-wise learning rate.