Beyond SGD...

Challenges of Mini-Batch SGD I

SGD (even with mini-batches) does not guarantee good convergence and presents several challenges:

- **Choosing a proper learning rate (γ):**
  - Too small $\rightarrow$ painfully slow convergence.
  - Too large $\rightarrow$ can hinder convergence, fluctuate around the minimum, or even diverge.

Challenges of Mini-Batch SGD I

SGD (even with mini-batches) does not guarantee good convergence and presents several challenges:

- **Choosing a proper learning rate (γ):**
  - Too small $\rightarrow$ painfully slow convergence.
  - Too large $\rightarrow$ can hinder convergence, fluctuate around the minimum, or even diverge.
- **Learning rate schedules:**
  - Pre-defined schedules (e.g., annealing) must be defined in advance.
  - They are unable to adapt to a dataset's characteristics.

Challenges of Mini-Batch SGD I

SGD (even with mini-batches) does not guarantee good convergence and presents several challenges:

- **Choosing a proper learning rate (γ):**
  - Too small $\rightarrow$ painfully slow convergence.
  - Too large $\rightarrow$ can hinder convergence, fluctuate around the minimum, or even diverge.
- **Learning rate schedules:**
  - Pre-defined schedules (e.g., annealing) must be defined in advance.
  - They are unable to adapt to a dataset's characteristics.
- **Same learning rate for all parameters:**
  - This is problematic for sparse data where features have different frequencies.
  - We might want larger updates for rare, informative features.

Challenges of Mini-Batch SGD II

- **Escaping suboptimal points:**
  - High non-convex error functions (common in NNs) have many suboptimal local minima.
  - The main difficulty may arise from *saddle points* (where one dim slopes up, another down).
  - At saddle points, the gradient is close to zero in all dimensions, making it hard for SGD to escape.

gdv

Momentum I

Idea

Helps accelerate SGD in the relevant direction and dampens oscillations. This is especially useful for navigating areas where the surface curves much more steeply in one dimension than another.
It adds a fraction (μ) of the update vector from the past time step to the current update vector. This simulates a "ball rolling down a hill" and accumulating momentum.

Momentum II

Update Rules

$$\boldsymbol{v}_t = \mu \boldsymbol{v}_{t-1} + \gamma \nabla_{\boldsymbol{w}} J(\boldsymbol{w})$$
$$\boldsymbol{w} = \boldsymbol{w} - \boldsymbol{v}_t$$

- $\boldsymbol{w}$: parameter vector
- γ: learning rate
- μ: momentum term (e.g., 0.9)
- $\boldsymbol{v}_t$: update vector (velocity)

Momentum III

Advantages & Drawbacks

- \+ Gains faster convergence in many scenarios.
- \+ Reduces oscillations.
- – The "ball" is blind. It can roll past the minimum or be led off-track if the slope changes.

Nesterov Accelerated Gradient (NAG) I

Idea

Gives the momentum term a "notion of where it is going." We want a "smarter ball" that knows to slow down before the hill slopes up again. Instead of calculating the gradient at the current position ($\boldsymbol{w}$), it calculates the gradient at the *approximate future position* ($\boldsymbol{w} - \mu \boldsymbol{v}_{t-1}$) after the momentum step is applied.

Update Rules

$$\boldsymbol{v}_t = \mu \boldsymbol{v}_{t-1} + \gamma \nabla_{\boldsymbol{w}} J(\boldsymbol{w} - \mu \boldsymbol{v}_{t-1})$$
$$\boldsymbol{w} = \boldsymbol{w} - \boldsymbol{v}_t$$

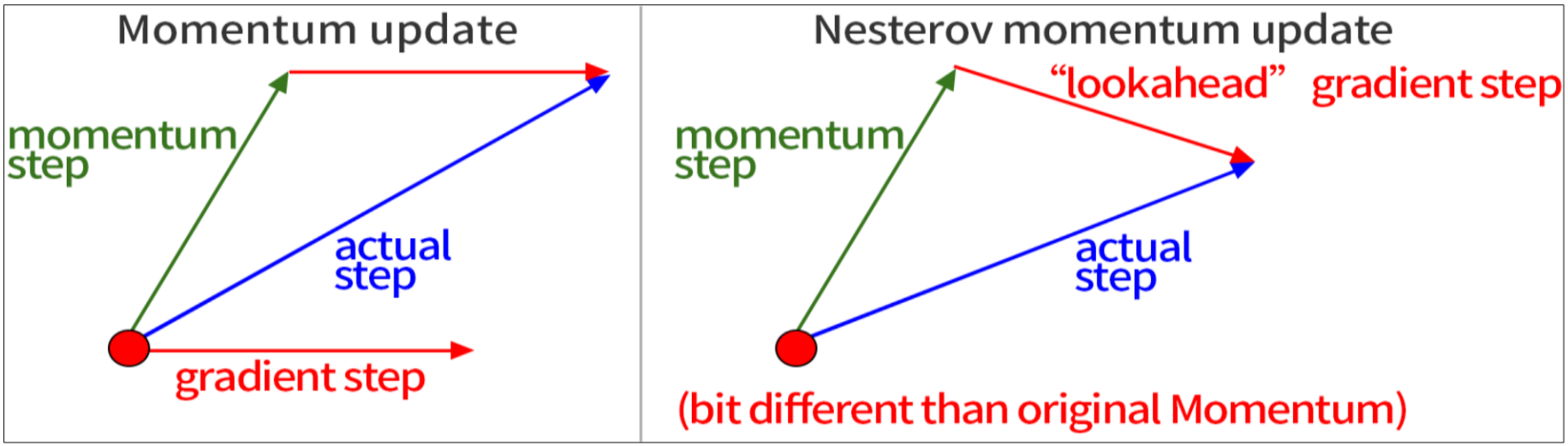
We calculate the gradient not w.r.t. $\boldsymbol{w}$, but w.r.t. the "looked-ahead" position.

Nesterov Accelerated Gradient (NAG) II

Advantages

+ Increased responsiveness and "smarter" updates.
+ Prevents going too fast and overshooting the minimum.
+ Significantly increased the performance of RNNs on many tasks.

NAG Visualization



Adagrad I

Idea

Adapts the learning rate to the parameters on a per-parameter basis.

- **Larger updates** for infrequent parameters.
- **Smaller updates** for frequent parameters.

It is very well-suited for sparse data.

Adagrad II

Update Rules

Let $\boldsymbol{g}_{t,i} = \nabla_{\boldsymbol{w}} J(w_{t,i})$ (gradient of param i at step t). Per-parameter update:

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

- G_t: A diagonal matrix where $G_{t,ii}$ is the **sum of the squares** of the gradients w.r.t. w_i up to time step t.
- ϵ: A smoothing term (e.g., 10^{-8}) to avoid division by zero.

Vectorized form:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\gamma}{\sqrt{G_t + \epsilon}} \odot \boldsymbol{g}_t$$

Adagrad - Advantages & Drawbacks

Advantages

+ Learning rate is adapted for each parameter.
+ Well-suited for sparse data.
+ Eliminates the need to manually tune the learning rate (most use default $\gamma = 0.01$).

Adagrad - Advantages & Drawbacks

Advantages

+ Learning rate is adapted for each parameter.
+ Well-suited for sparse data.
+ Eliminates the need to manually tune the learning rate (most use default $\gamma = 0.01$).

Main Drawback

– **Accumulation of squared gradients in the denominator.**
  - Since every added term is positive, the sum (G_t) keeps growing during training.
  - This causes the learning rate to shrink and eventually become infinitesimally small.
  - At this point, the algorithm effectively stops learning.

Adadelta I

Idea

Resolves Adagrad's aggressive, monotonically decreasing learning rate.

Instead of accumulating *all* past squared gradients, it restricts the window to a fixed size by using a **decaying average** of past squared gradients.

Adadelta II

Update Rules

Running average of squared gradients (using μ as decay term):

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

Parameter update:

$$\Delta \boldsymbol{w}_t = -\frac{RMS[\Delta \boldsymbol{w}]_{t-1}}{RMS[\boldsymbol{g}]_t}\boldsymbol{g}_t$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \Delta \boldsymbol{w}_t$$

Where $RMS[\cdot]_t = \sqrt{E[\cdot^2]_t + \epsilon}$. The update rule units match.

ArXiv Paper

Adadelta III

Advantages

+ Solves Adagrad's decaying learning rate problem.
+ **No learning rate needs to be set.** It has been eliminated from the update rule.

RMSprop I

Idea (Unpublished method by Geoff Hinton)

Developed independently of Adadelta to resolve Adagrad's diminishing learning rates.

Divides the learning rate by an exponentially decaying average of squared gradients.

RMSprop II

Update Rules

(Identical to the first part of Adadelta's derivation, but keeps γ)

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

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\gamma}{\sqrt{E[\boldsymbol{g}^2]_t + \epsilon}}\boldsymbol{g}_t$$

- μ (decay rate) is typically 0.9.
- γ (learning rate) is typically 0.001.

Advantages & Drawbacks

+ Fixes Adagrad's vanishing learning rate.
+ Empirically works very well.
– Still requires a learning rate γ to be chosen (unlike Adadelta).

Adam (Adaptive Moment Estimation) I

Idea

Computes adaptive learning rates for each parameter, and also keeps an exponentially decaying average of past gradients (like momentum).

Combines the ideas of **Momentum** (first moment estimate) and **RMSprop** (second moment estimate).

Adam (Adaptive Moment Estimation) II

Update Rules

Update biased moment estimates:

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t \quad \text{(1st moment, mean)}$$
$$\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2 \quad \text{(2nd moment, variance)}$$

Compute bias-corrected estimates (to counteract 0-initialization):

$$\hat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1-\beta_1^t} \quad \text{and} \quad \hat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1-\beta_2^t}$$

Final update:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\gamma}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon}\hat{\boldsymbol{m}}_t$$

Adam - Advantages & Drawbacks

Advantages

+ Combines benefits of Adagrad/RMSprop (handles sparse gradients) and Momentum (accelerates).
+ Bias-correction helps counteract initial bias towards zero, especially early in training.
+ Works very well in practice and is often the default choice.
+ Default values (e.g., $\gamma = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$) are often effective.

Drawbacks

– More complex, stores two moving averages per parameter ($\boldsymbol{m}_t, \boldsymbol{v}_t$).
– Some studies note it can occasionally fail to converge where SGD+Momentum does (though this is debated).

Which Optimizer to Use?

- **If your input data is sparse:**
  - Use one of the **adaptive learning-rate methods** (Adagrad, Adadelta, RMSprop, Adam).
  - You will likely not need to tune the learning rate.

Which Optimizer to Use?

- **If your input data is sparse:**
  - Use one of the **adaptive learning-rate methods** (Adagrad, Adadelta, RMSprop, Adam).
  - You will likely not need to tune the learning rate.
- **Overall Recommendation:**
  - **Adam** might be the best overall choice.

Which Optimizer to Use?

- **If your input data is sparse:**
  - Use one of the **adaptive learning-rate methods** (Adagrad, Adadelta, RMSprop, Adam).
  - You will likely not need to tune the learning rate.
- **Overall Recommendation:**
  - **Adam** might be the best overall choice.
- **Warning!**
  - Many recent papers use **SGD (with momentum)** and a simple learning rate annealing schedule.
  - SGD *can* find a minimum, but it takes longer, is reliant on good initialization/annealing, and may get stuck in saddle points.

Which Optimizer to Use?

- **If your input data is sparse:**
  - Use one of the **adaptive learning-rate methods** (Adagrad, Adadelta, RMSprop, Adam).
  - You will likely not need to tune the learning rate.
- **Overall Recommendation:**
  - **Adam** might be the best overall choice.
- **Warning!**
  - Many recent papers use **SGD (with momentum)** and a simple learning rate annealing schedule.
  - SGD *can* find a minimum, but it takes longer, is reliant on good initialization/annealing, and may get stuck in saddle points.
- **Final Verdict:** If you care about *fast convergence* and are training a deep or complex network, choose an **adaptive learning rate method**.

gdv