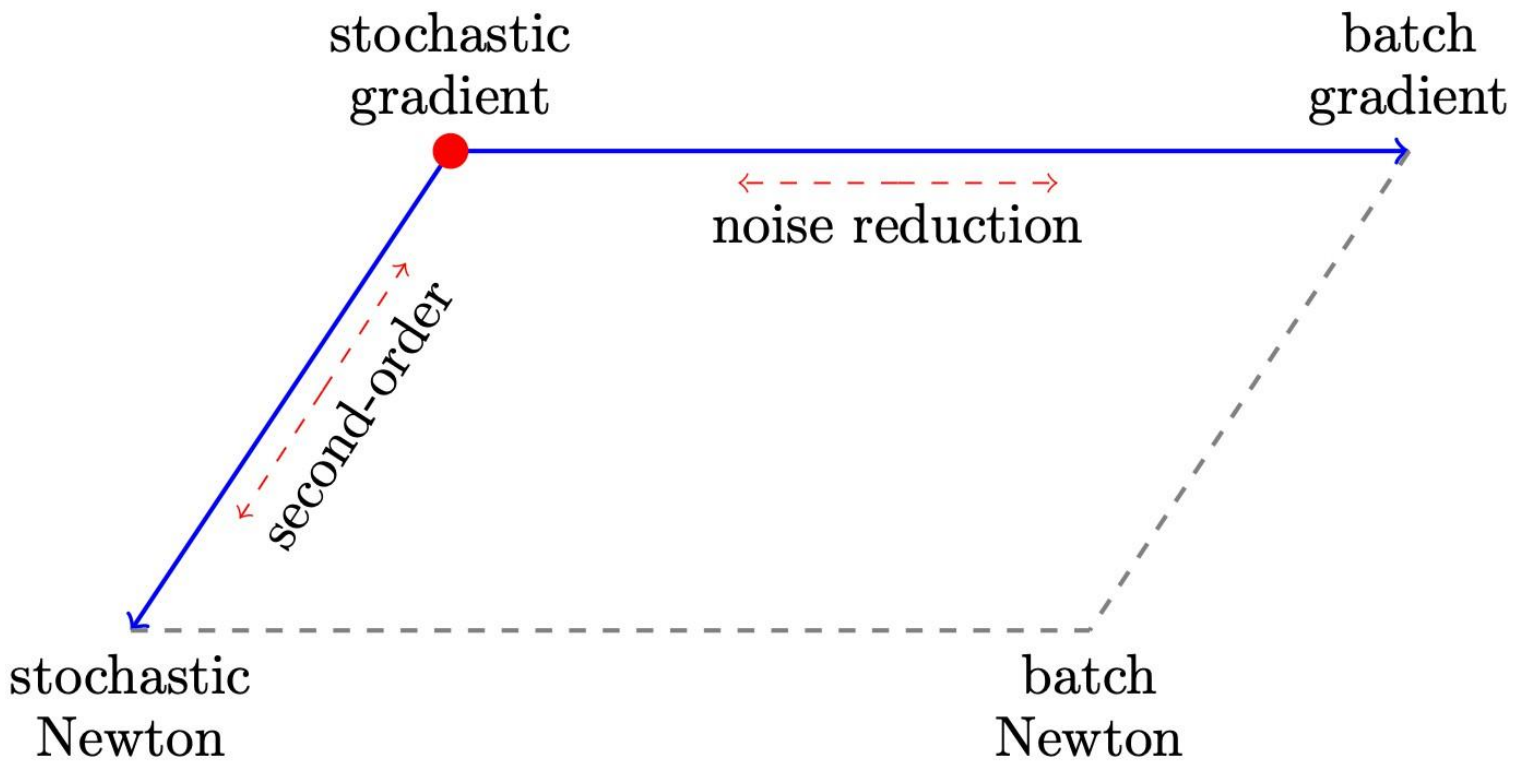
# IMPROVING SGD




Slides taken from Jorge Nocedal

# STOCHASTIC AVERAGED GRADIENT

- Basic SAG algorithm:
  - while(1)
  - Sample $i$ from $\{1, 2, \ldots, N\}$.
  - Compute $f_i'(x)$.
  - $d = d - y_i + f_i'(x)$.
  - $y_i = f_i'(x)$.
  - $x = x - \frac{\alpha}{N} d$.

# STOCHASTIC AVERAGED GRADIENT

## The Core Idea: Averaging Past Gradients

The insight behind SAG is elegantly simple: instead of using just the gradient from the current mini-batch (or single data point in the simplest case), why not leverage information from gradients computed in previous iterations? SAG maintains a memory of the *most recently computed gradient* for *each individual data point* in the training set.

Let the objective function be the average loss over $N$ data points: $L(\theta) = \frac{1}{N} \sum_{i=1}^{N} L_i(\theta)$ where $L_i(\theta)$ is the loss associated with the $i$-th data point and $\theta$ represents the model parameters.

Full batch gradient descent uses the gradient $\nabla L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla L_i(\theta)$. SGD, in its simplest form (batch size 1), picks a random index $i_k$ at iteration $k$ and updates using $\nabla L_{i_k}(\theta_k)$.

SAG works differently. It maintains a table containing $N$ gradient vectors, $g_1, g_2, \ldots, g_N$. Each $g_i$ stores the most recent gradient $\nabla L_i(\theta)$ computed for data point $i$ at some past iteration when $i$ was selected.

Slides taken from Mark Schmidt


# STOCHASTIC AVERAGED GRADIENT

## The SAG Update Rule

At each iteration $k+1$:

1. **Sample:** Randomly select an index $i_k$ from $\{1, 2, \ldots, N\}$.
2. **Compute Current Gradient:** Calculate the gradient for the selected data point using the current parameters $\theta_k$: $v_{i_k} = \nabla L_{i_k}(\theta_k)$ .
3. **Update Average Gradient Estimate:** The core of SAG is using an average of the stored gradients as the update direction. This average can be updated efficiently. Let $G_k = \frac{1}{N}\sum_{j=1}^{N} g_j$ be the average of gradients stored up to iteration $k$. The new average $G_{k+1}$ is calculated by effectively replacing the old stored gradient $g_{i_k}$ with the newly computed gradient $v_{i_k}$ in the sum: $G_{k+1} = G_k + \frac{1}{N}(v_{i_k} - g_{i_k})$ This avoids re-summing all $N$ stored gradients at every step.
4. **Update Parameters:** Update the parameters using this less noisy average gradient estimate: $\theta_{k+1} = \theta_k - \eta G_{k+1}$ where $\eta$ is the learning rate.
5. **Update Memory:** Store the newly computed gradient $v_{i_k}$ in the memory table, replacing the previous value for $g_{i_k}$: $g_{i_k} \leftarrow v_{i_k}$

Initially, the stored gradients $g_i$ might be initialized to zero vectors or computed from an initial parameter guess.

Slides taken from Mark Schmidt

# STOCHASTIC AVERAGED GRADIENT

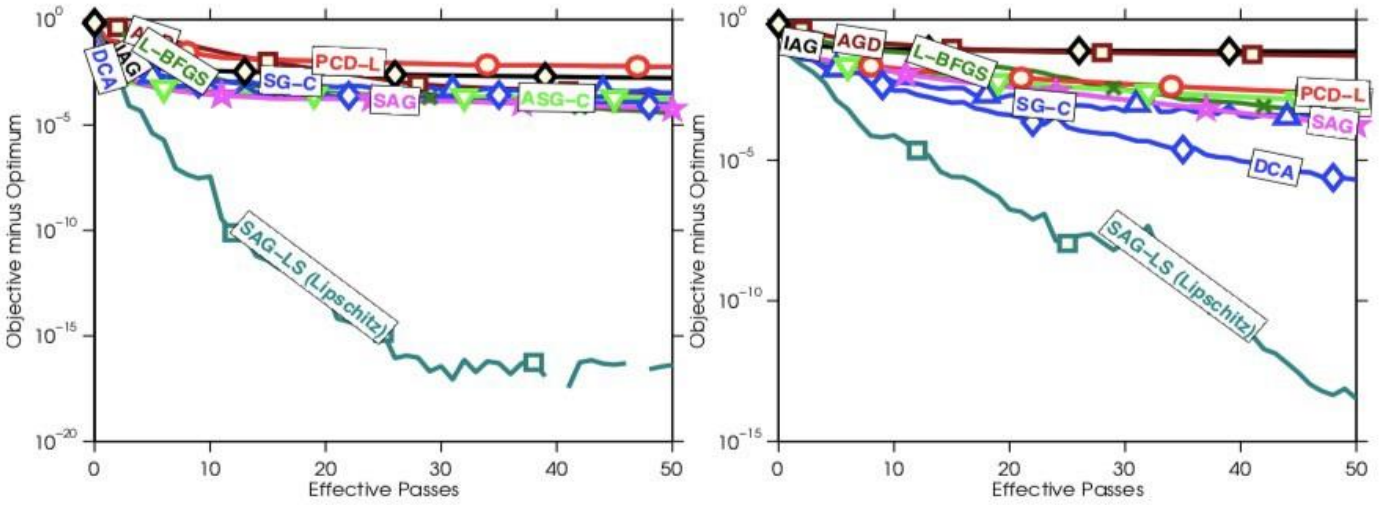
## Convergence Properties

One of the main advantages of SAG is its convergence rate. For strongly convex and smooth objective functions, SAG achieves a *linear* convergence rate. This is a significant improvement over the typically *sublinear* rate of standard SGD for such problems. In essence, SAG converges much faster in terms of the number of iterations required to reach a certain accuracy, approaching the desirable properties of full batch gradient descent. The convergence does not degrade significantly with increasing dataset size $N$, unlike batch gradient descent whose cost per iteration scales linearly with $N$.

Slides taken from Mark Schmidt

# SAG with Non-Uniform Sampling

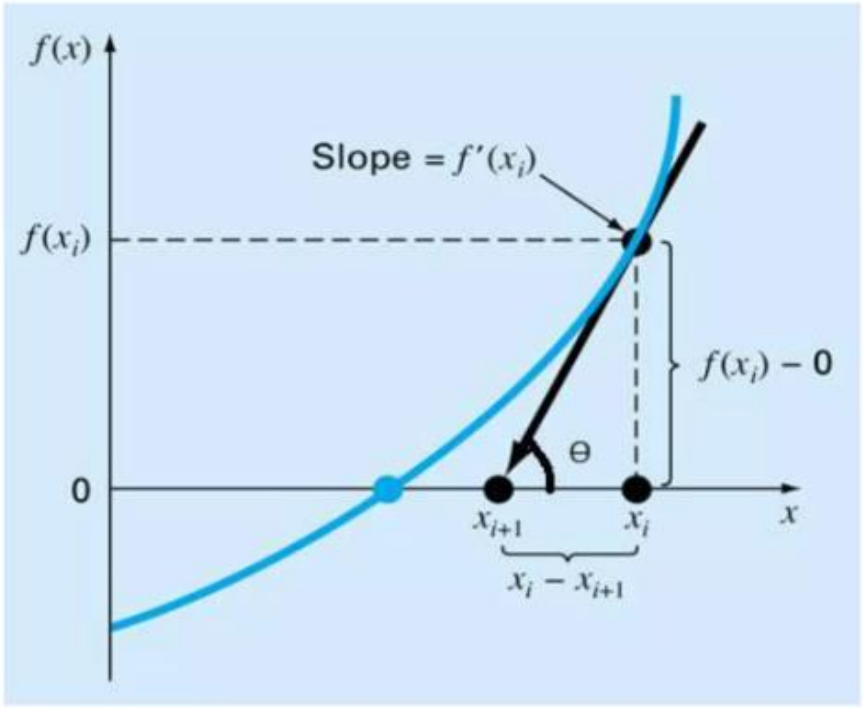
- protein ($n = 145751$, $p = 74$) and sido ($n = 12678$, $p = 4932$)




- Adaptive non-uniform sampling helps a lot.

# Newton's Method of Optimization

Let us consider an equation f(x)=0 having graphical representation as




Slides taken from Mark Schmidt

# Newton's Method of Optimization

- f(x) =0 ,is an given equation
- Starting from an initial point $x_0$
- Determine the slope of f(x) at x=$x_0$ .Call it f'($x_0$).

Slope =tanΘ=$\frac{f(x_i)-0}{x_i-x_{i+1}} = f'(x_i)$.

From here we get

- $f'(x_i) = \frac{f(x_i)}{x_i-x_{i+1}}$ .

Hence ;

- $\mathbf{x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}}$

midt

# Multivariate Newton's Method

The second-order Taylor expansion of a function $f(x)$ around a point $x_t$ is given by

$$f(x) \approx f(x_t) + \langle \nabla f(x_t), x - x_t \rangle + \frac{1}{2} \langle x - x_t, \nabla^2 f(x_t)(x - x_t) \rangle$$

where $\nabla^2 f(x_t)$ is the Hessian matrix of $f$ at $x_t$. The minimum of $f(x)$ can be found in closed form by setting the gradient (with respect to $x$) of the above expression to zero, which gives

$$\nabla f(x_t) + \nabla^2 f(x_t)(x - x_t) = 0 \quad \Longrightarrow \quad x = x_t - [\nabla^2 f(x_t)]^{-1} \nabla f(x_t).$$

So, we find that by moving from first-order to second-order Taylor approximation, and assuming that $\nabla^2 f(x_t)$ is invertible, the natural direction of descent changes from

$$\underbrace{d = -\nabla f(x_t)}_{\substack{\text{using first-order} \\ \text{Taylor approximation}}} \quad \text{to} \quad \underbrace{d = -[\nabla^2 f(x_t)]^{-1} \nabla f(x_t)}_{\text{using second-order Taylor approximation}} .$$

Slides taken from Mark Schmidt


# Multivariate Newton's Method

Having established the second-order direction of descent, we can define the natural generalization of the gradient descent algorithm to the second-order setting. This algorithm, which takes the name *damped Newton's method*, is given by the update rule

$$\boxed{x_{t+1} = x_t - \eta[\nabla^2 f(x_t)]^{-1} \nabla f(x_t)} \,. \tag{1}$$

For the choice $\eta = 1$, which corresponds to setting $x_{t+1}$ to be the minimum of the second-order approximation centered at $x_t$ at each iteration, this algorithm is known simply as *Newton's method*.

Slides taken from Mark Schmidt


# SGD vs Newton's Method

**Derivative Order**

- **Gradient Descent:** Uses first-order derivatives (the gradient) to find the direction of steepest descent.
- **Newton's Method:** Uses first and second-order derivatives (the gradient and the Hessian matrix) to account for the local curvature of the function.

Slides taken from Mark Schmidt

# SGD vs Newton's Method

**Convergence Rate**

- **Gradient Descent:** Linear convergence; requires more iterations but each step is computationally fast.
- **Newton's Method:** Quadratic convergence; reaches the minimum in far fewer iterations.

Slides taken from Mark Schmidt

# SGD vs Newton's Method

**Computational Cost per Step**

- **Gradient Descent:** Low cost, scaling linearly with the number of dimensions O(d).
- **Newton's Method:** High cost, requiring the computation and inversion of an n × n Hessian matrix, which takes $O(d^3)$ operations

Slides taken from Mark Schmidt

# SGD vs Newton's Method

**Hyperparameters**

- **Gradient Descent:** Requires careful tuning of the learning rate (step size).
- **Newton's Method:** Generally parameter-free regarding step size, making direct optimal steps on quadratic functions.

Slides taken from Mark Schmidt

# Quasi Newton Methods

- Newton's method performs the below iteration to compute the next point on the gradient: $x_{t+1} = x_t - \eta[\nabla^2 f(x_t)]^{-1} \nabla f(x_t)$
- **Broyden–Fletcher–Goldfarb–Shanno (BFGS)** algorithm is a **quasi-Newton method**.
- Quasi-Newton methods replace the expensive operation of the computation of inverse of Hessian matrix with an approximation (e.g. a positive definite matrix).
- However, BFGS still uses a n x n memory to store the approximation matrix. If the machine learning model has large number of parameters (hundreds of thousands to millions), this will be very large to fit in memory.

Slides taken from Mark Schmidt


# Quasi Newton Methods

- **Limited Memory BFGS (L-BFGS)** is an improvement over BFGS to optimize memory usage. L-BFGS uses the approximation to Hessian matrix approach. But instead of storing a large n x n matrix as approximation to Hessian, it stores a set of vectors. This reduces memory usage significantly and it can used as optimization algorithm to build large machine learning models.

Slides taken from Mark Schmidt

# SGD vs Quasi Newton Methods

- **Curvature Awareness**: Gradient descent is "geometrically blind". If a function features a steep cliff in one direction but a gentle slope in another, GD will violently zigzag back and forth. L-BFGS uses partial second-derivatives to compute how the surface bends, automatically slowing down where it is steep and accelerating down gentle slopes.

- **No Learning Rate Tweak**: Standard gradient descent requires strict optimization of a single learning rate. L-BFGS uses a line-search method to automatically calculate the ideal step size during every iteration.

Slides taken from Mark Schmidt

# SGD vs Quasi Newton Methods

- **The "Limited-Memory" Trick**: Newton methods require calculating an n × n Hessian matrix, which is computationally impossible for millions of parameters. Instead of storing a full matrix, L-BFGS utilizes a **two-loop recursion** algorithm that dynamically computes the matrix updates using only the history of the last m (usually 5 to 20) gradient vectors

Slides taken from Mark Schmidt


# Batch Newton Methods

Sample a batch of training samples and follow a quasi-newton method to update model parameters.

Slides taken from Mark Schmidt

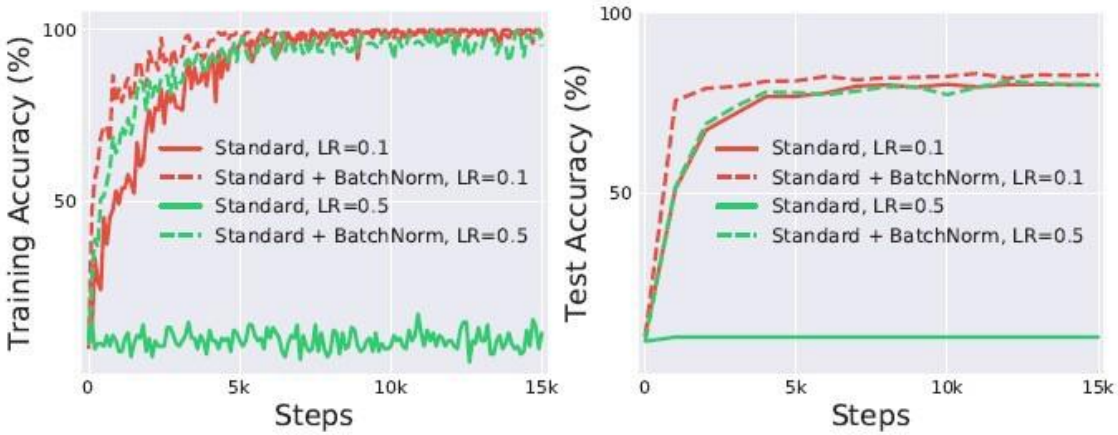

# BATCH NORMALIZATION

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

# BATCH NORMALIZATION: OTHER BENEFITS IN PRACTICE

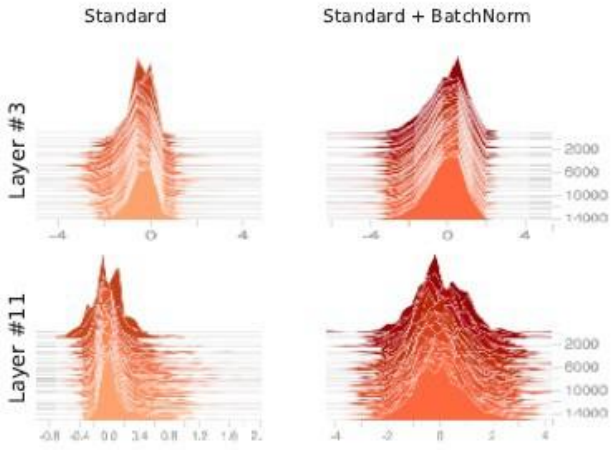
- BN reduces training times. (Because of less Covariate Shift, less exploding/vanishing gradients.)
- BN reduces demand for regularization, e.g. dropout or L2 norm.
  - Because the means and variances are calculated over batches and therefore every normalized value depends on the current batch. I.e. 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"

# DOING IT THE "CORRECT WAY" IS TOO EXPENSIVE!

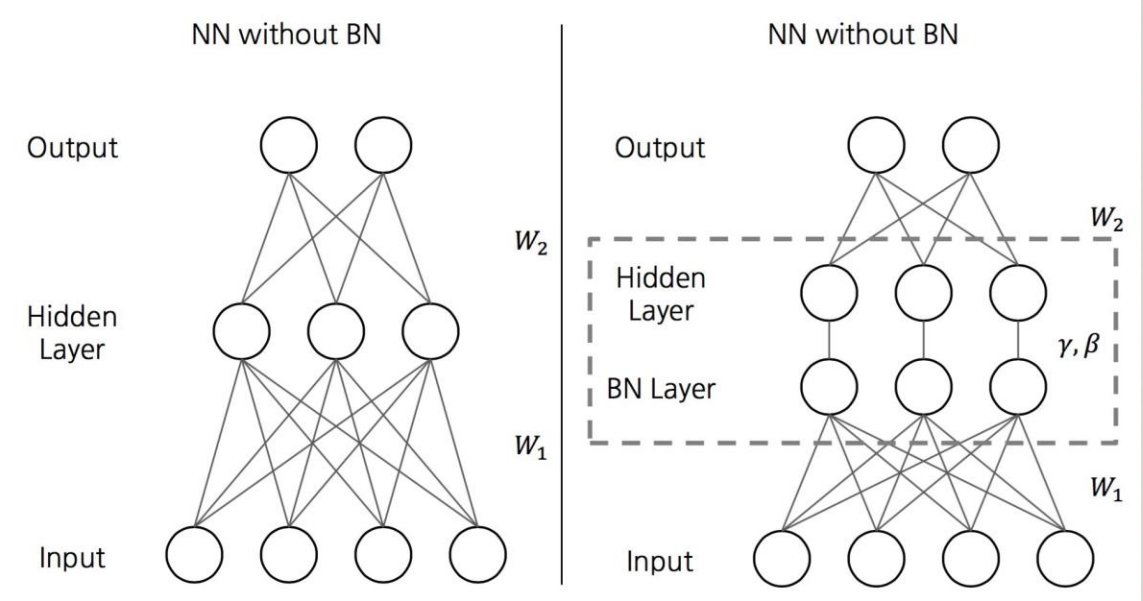
- A proper method has to include the current example batch *and* somehow all previous batches ( all examples) in the normalization step.

- This leads to calculating in covariance matrix and its inverse square root. That's expensive. The authors found a faster way!

# THE PROPOSED SOLUTION: REGULARIZATION EXTRA 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: $\gamma$, $\beta$

**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 $\Theta$; 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:

$$\text{E}[x] \leftarrow \text{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$

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

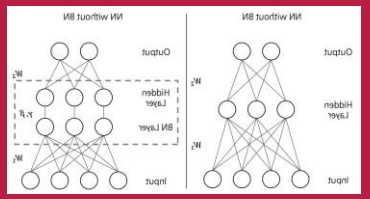
11: In $N_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma,\beta}(x)$ with $y = \frac{\gamma}{\sqrt{\text{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\text{E}[x]}{\sqrt{\text{Var}[x]+\epsilon}}\right)$

12: **end for**

**Algorithm 2:** Training a Batch-Normalized Network

Alg 1 (previous slide)

Architecture modification



**Note that BN(x) is different during test…**

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

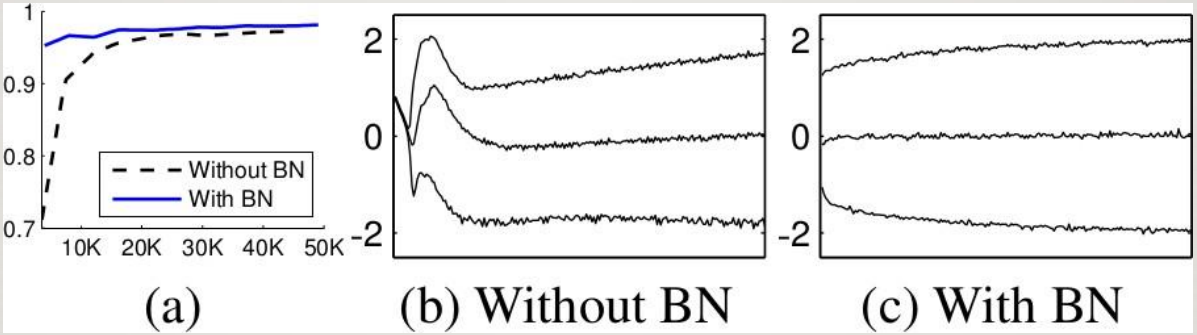
**Vs.**

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

# 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.




*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.*


# THANKS

# QUESTIONS?

Email: sourangshu@cse.iitkgp.ac.in