

Training Neural Networks: Optimization

Intro to Deep Learning, Spring 2021

Recap

- Neural networks are universal approximators
- We must *train* them to approximate any function
- Networks are trained to minimize total “error” on a training set
 - We do so through empirical risk minimization
- We use variants of gradient descent to do so
 - Gradients are computed through backpropagation

Recap

- Vanilla gradient descent may be too slow or unstable
- Better convergence can be obtained through
 - Second order methods that normalize the variation across dimensions
 - Adaptive or decaying learning rates that can improve convergence
 - Methods like Rprop that decouple the dimensions can improve convergence
 - Momentum methods which emphasize directions of steady improvement and deemphasize unstable directions

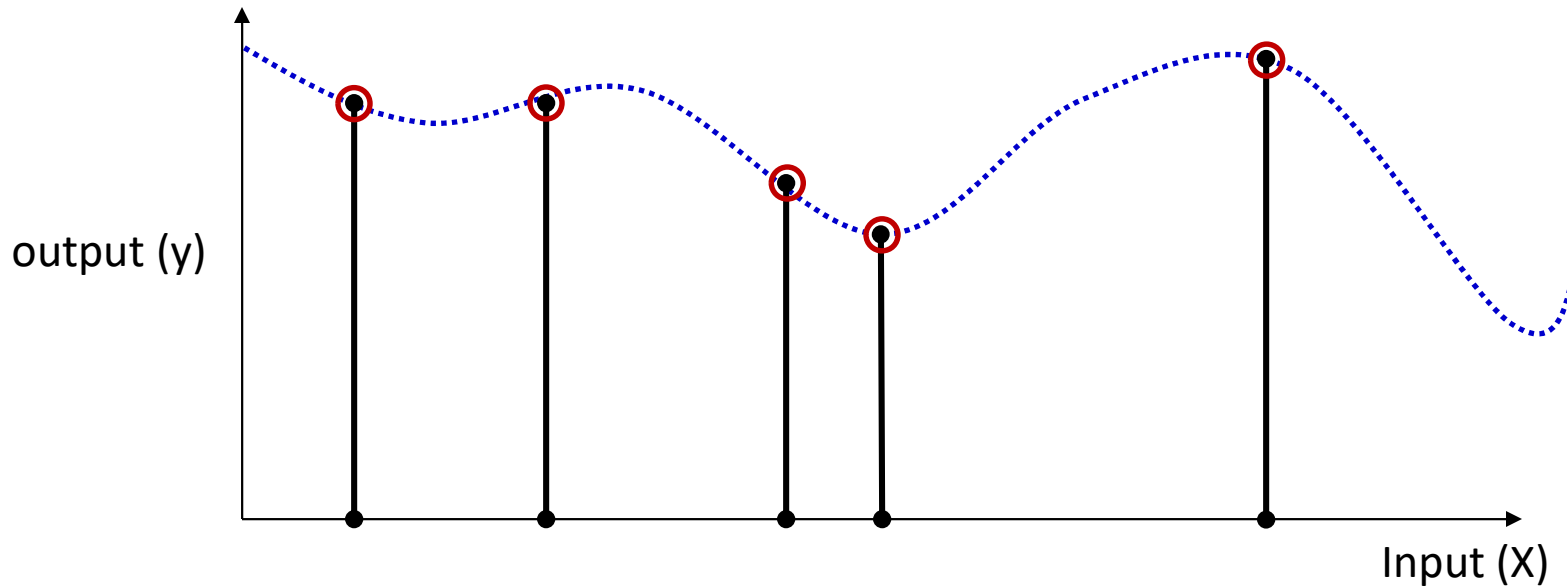
Moving on...

- Incremental updates
- Revisiting “trend” algorithms
- Generalization
- Tricks of the trade
 - Divergences..
 - Activations
 - Normalizations

Moving on: Topics for the day

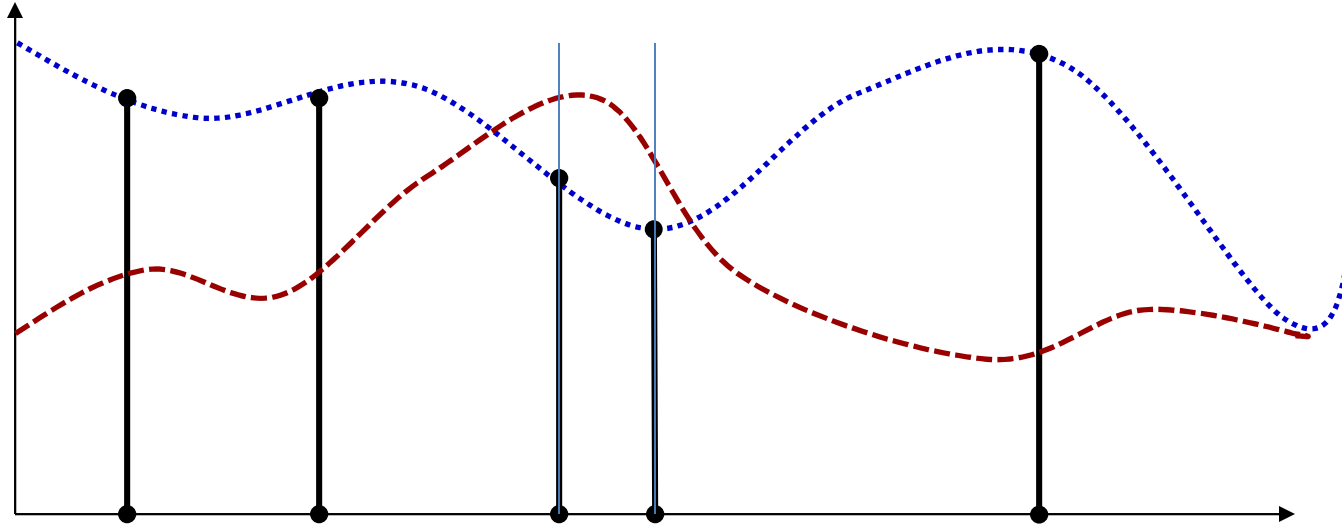
- Incremental updates
- Revisiting “trend” algorithms
- Generalization
- Tricks of the trade
 - Divergences..
 - Activations
 - Normalizations

The training formulation



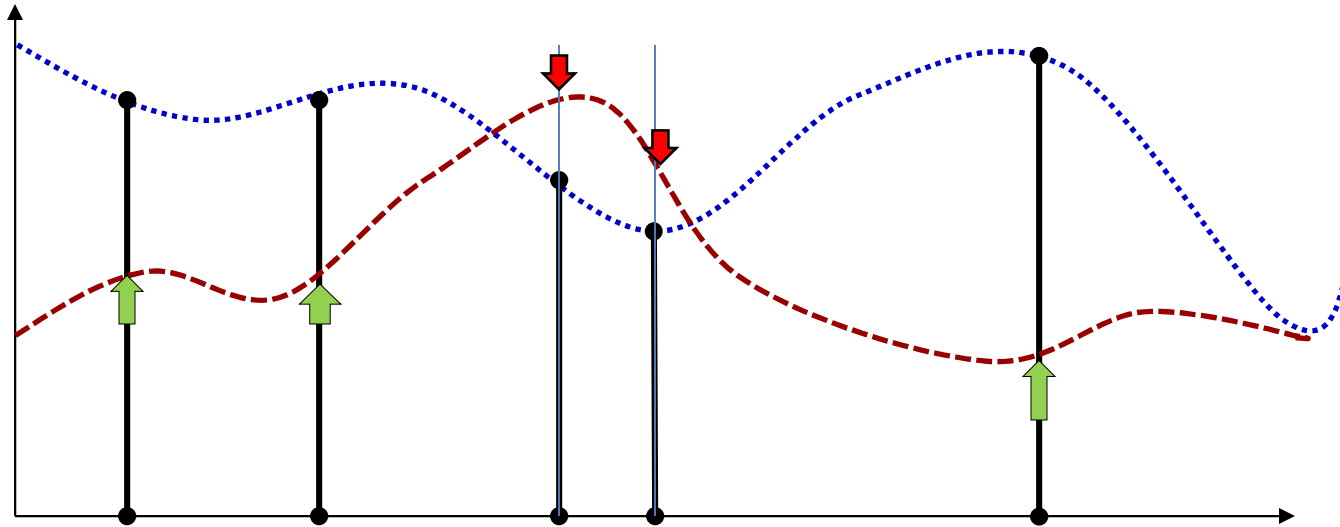
- Given input output pairs at a number of locations, estimate the entire function

Gradient descent



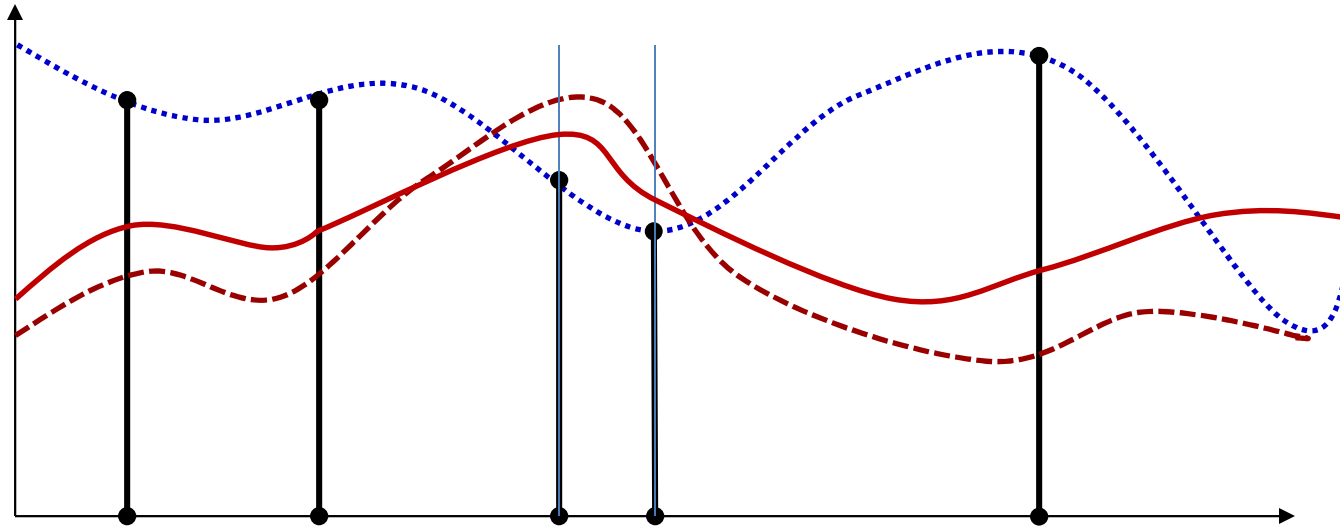
- Start with an initial function

Gradient descent



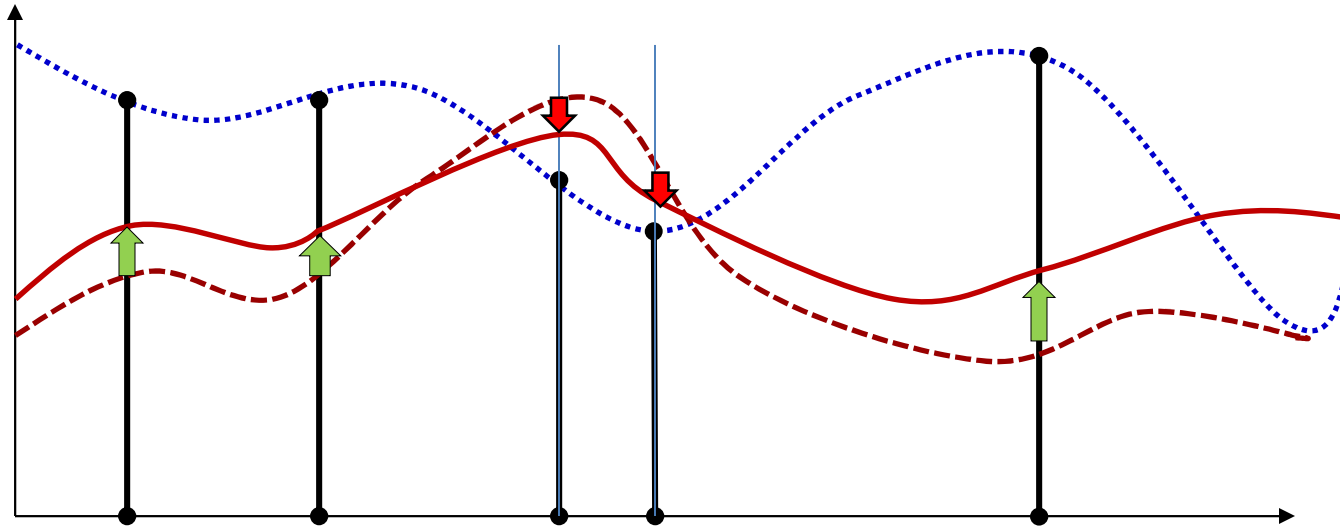
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
 - Gradient descent adjusts parameters to adjust the function value at *all* points
 - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



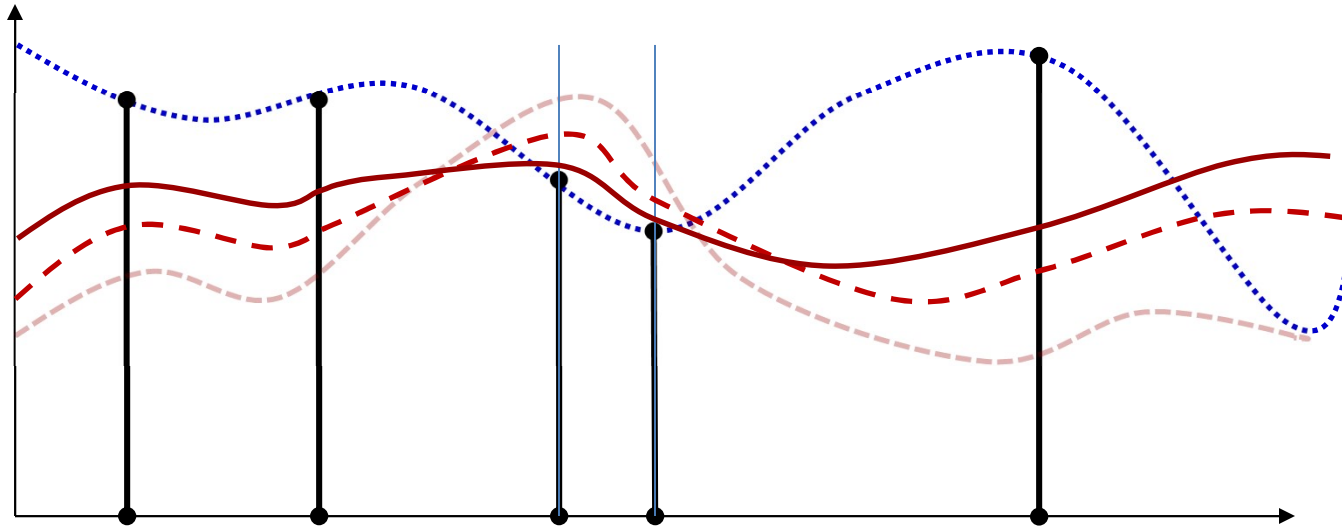
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
 - Gradient descent adjusts parameters to adjust the function value at *all* points
 - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



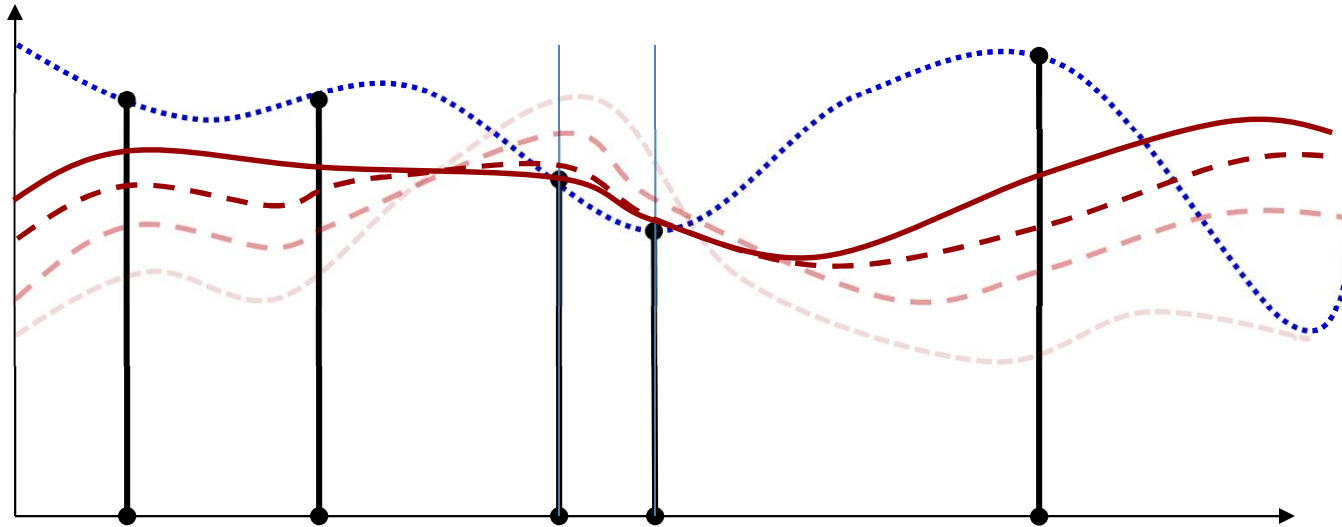
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
 - Gradient descent adjusts parameters to adjust the function value at *all* points
 - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



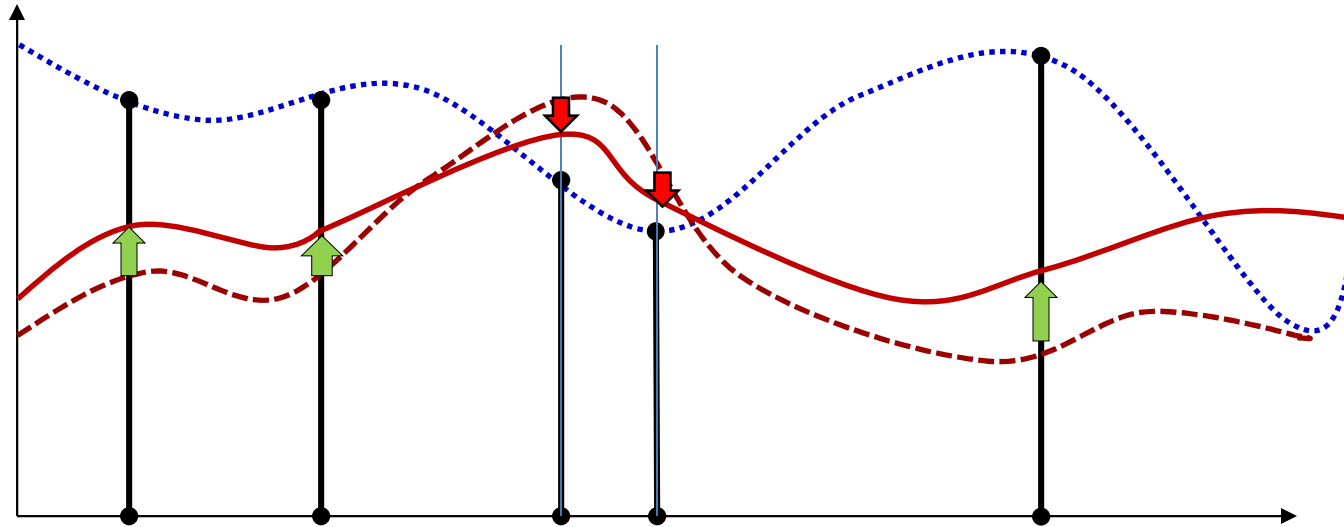
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
 - Gradient descent adjusts parameters to adjust the function value at *all* points
 - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



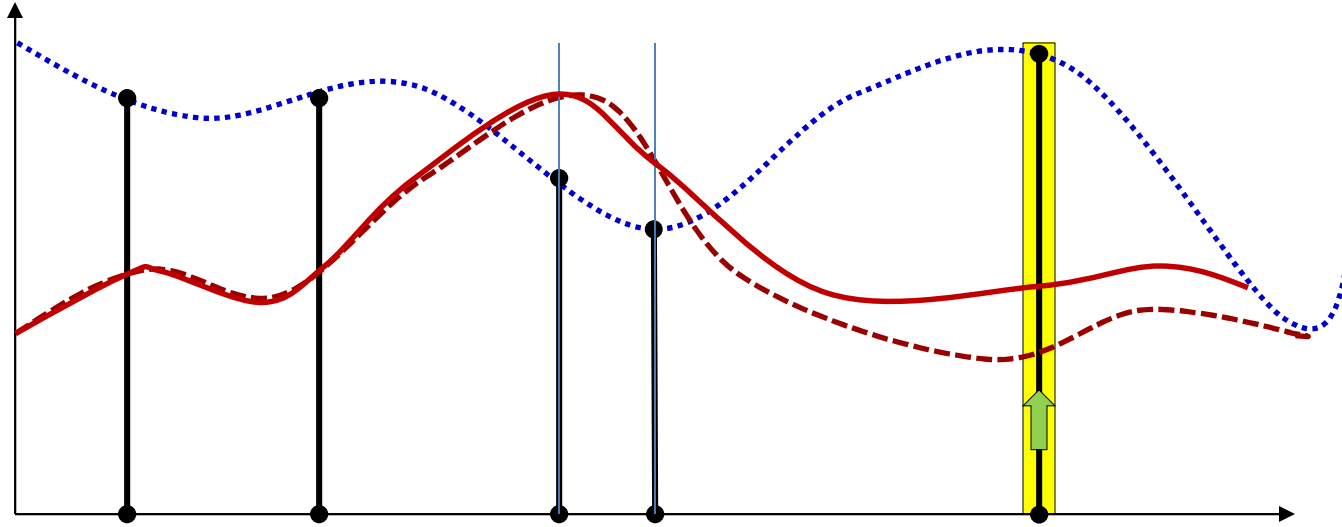
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
 - Gradient descent adjusts parameters to adjust the function value at *all* points
 - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Effect of number of samples



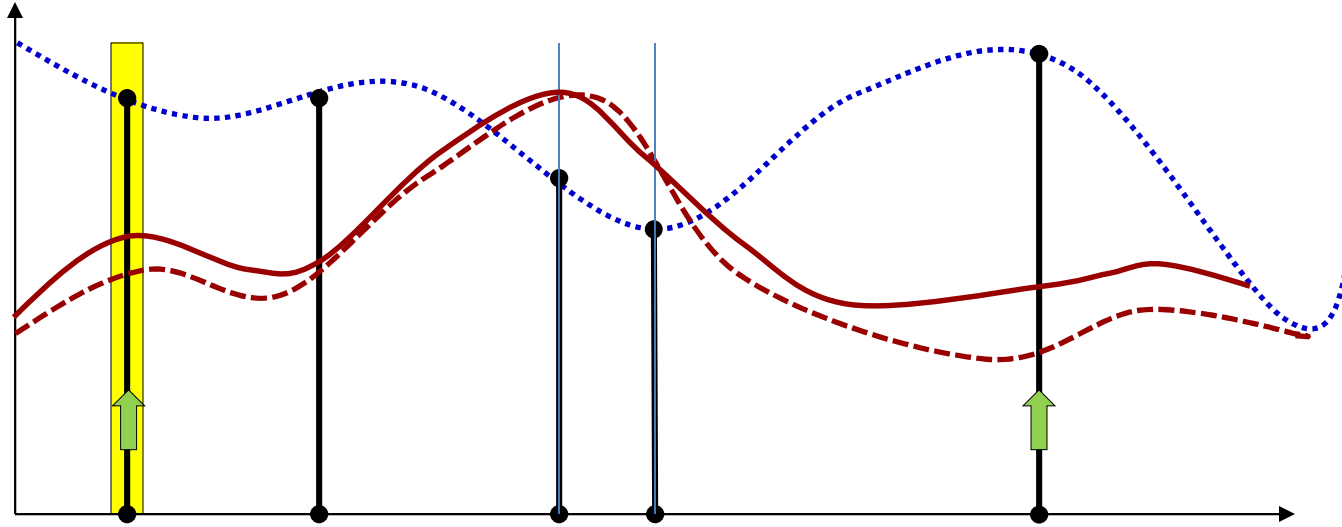
- Problem with conventional gradient descent: we try to simultaneously adjust the function at *all* training points
 - We must process *all* training points before making a single adjustment
 - **“Batch”** update

Alternative: Incremental update



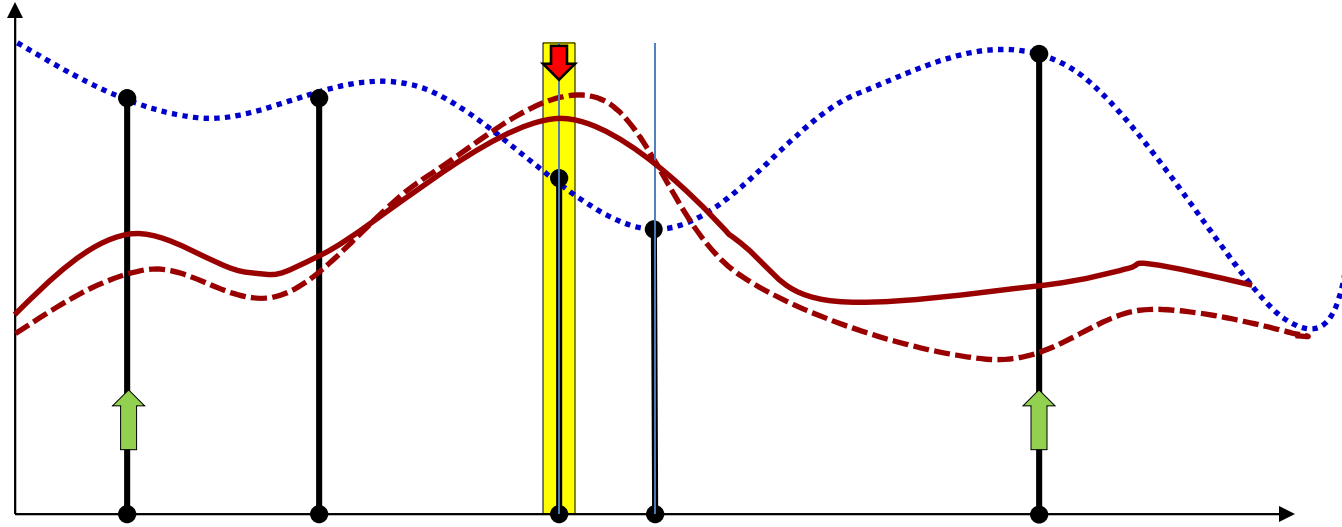
- Alternative: adjust the function at one training point at a time
 - Keep adjustments small

Alternative: Incremental update



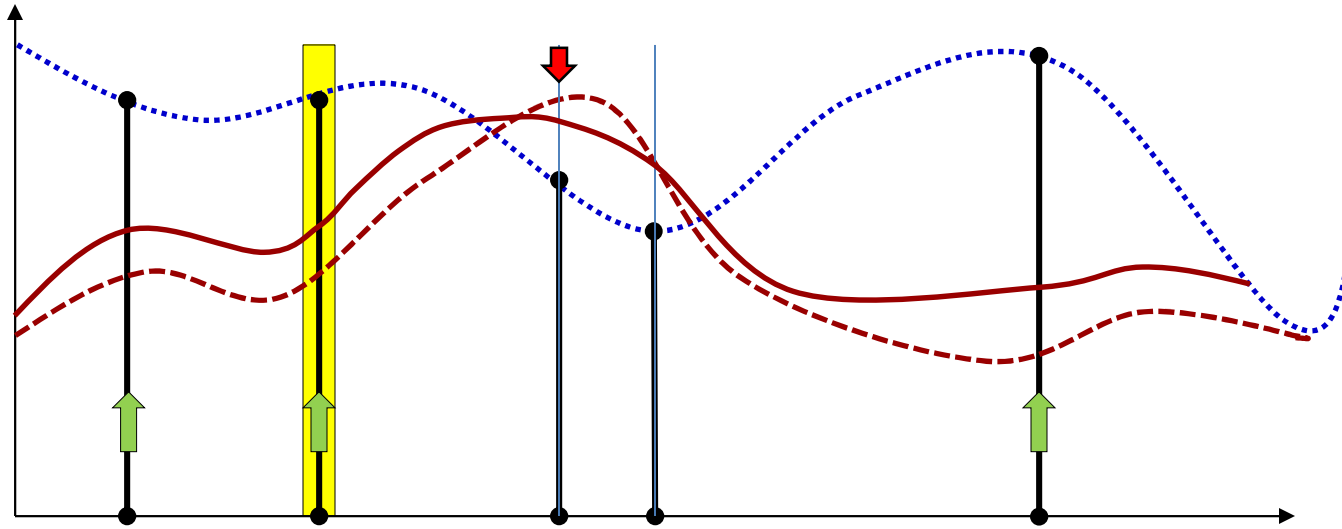
- Alternative: adjust the function at one training point at a time
 - Keep adjustments small

Alternative: Incremental update



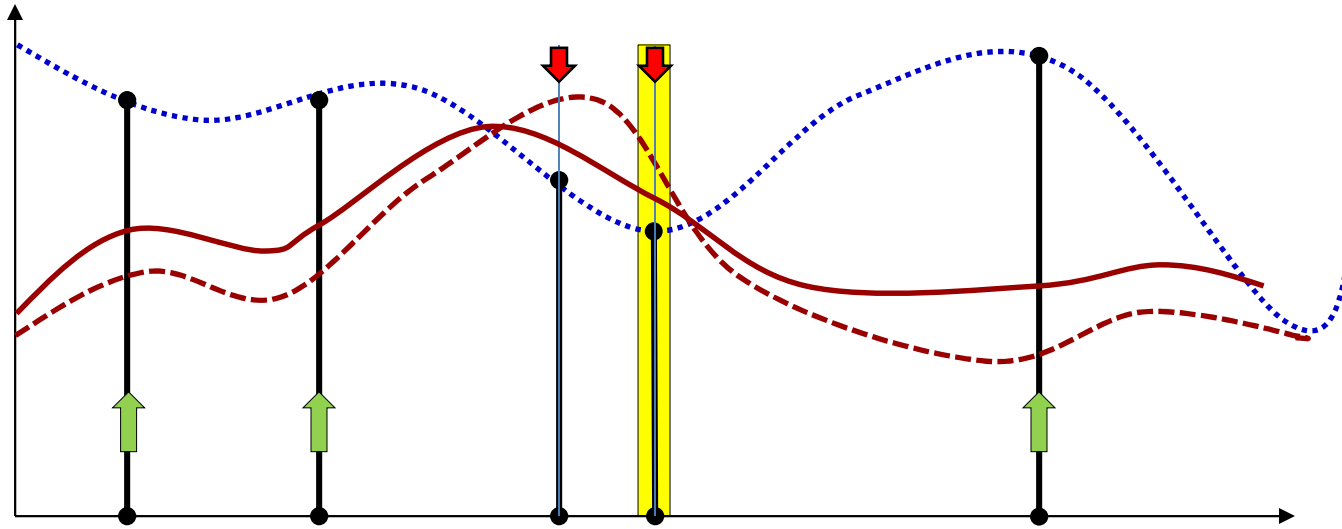
- Alternative: adjust the function at one training point at a time
 - Keep adjustments small

Alternative: Incremental update



- Alternative: adjust the function at one training point at a time
 - Keep adjustments small

Alternative: Incremental update



- Alternative: adjust the function at one training point at a time
 - Keep adjustments small
 - Eventually, when we have processed all the training points, we will have adjusted the entire function
 - With *greater* overall adjustment than we would if we made a single “Batch” update

Incremental Update

- Given $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights W_1, W_2, \dots, W_K
- Do:
 - For all $t = 1:T$
 - For every layer k :
 - Compute $\nabla_{W_k} \text{Div}(\mathbf{Y}_t, \mathbf{d}_t)$
 - Update
$$W_k = W_k - \eta \nabla_{W_k} \text{Div}(\mathbf{Y}_t, \mathbf{d}_t)^T$$
- Until *Loss* has converged

Incremental Updates

- The iterations can make multiple passes over the data
- A single pass through the entire training data is called an “epoch”
 - An epoch over a training set with T samples results in T updates of parameters

Incremental Update

- Given $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights W_1, W_2, \dots, W_K

- Do: 

– For all $t = 1:T$

- For every layer k :

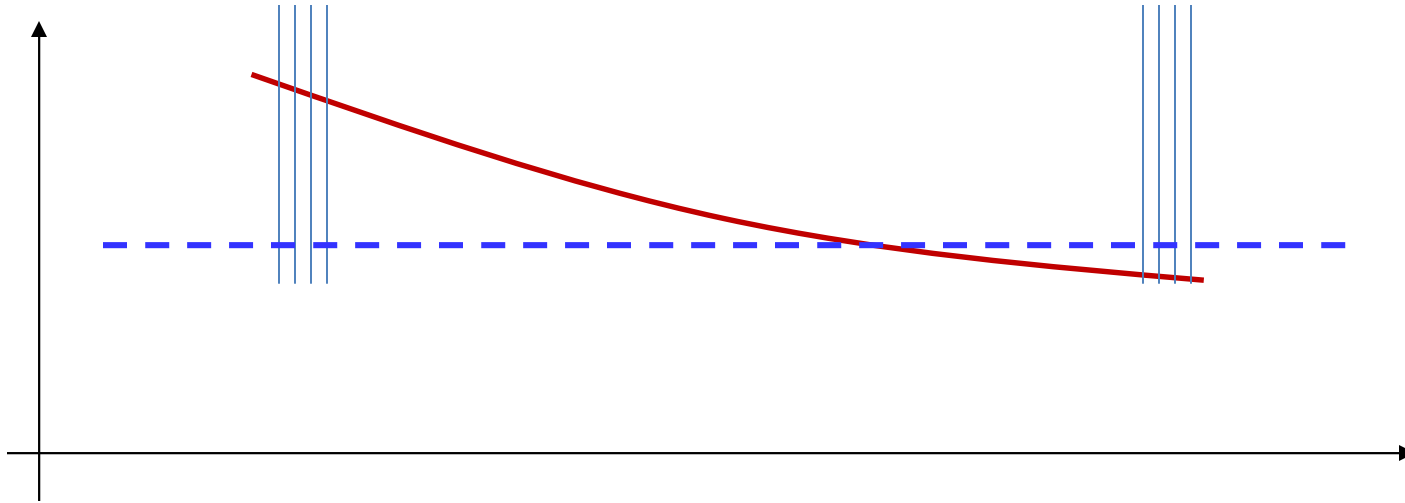
– Compute $\nabla_{W_k} \text{Div}(Y_t, d_t)$

– Update

$$W_k = W_k - \eta \nabla_{W_k} \text{Div}(Y_t, d_t)^T \leftarrow \text{One update}$$

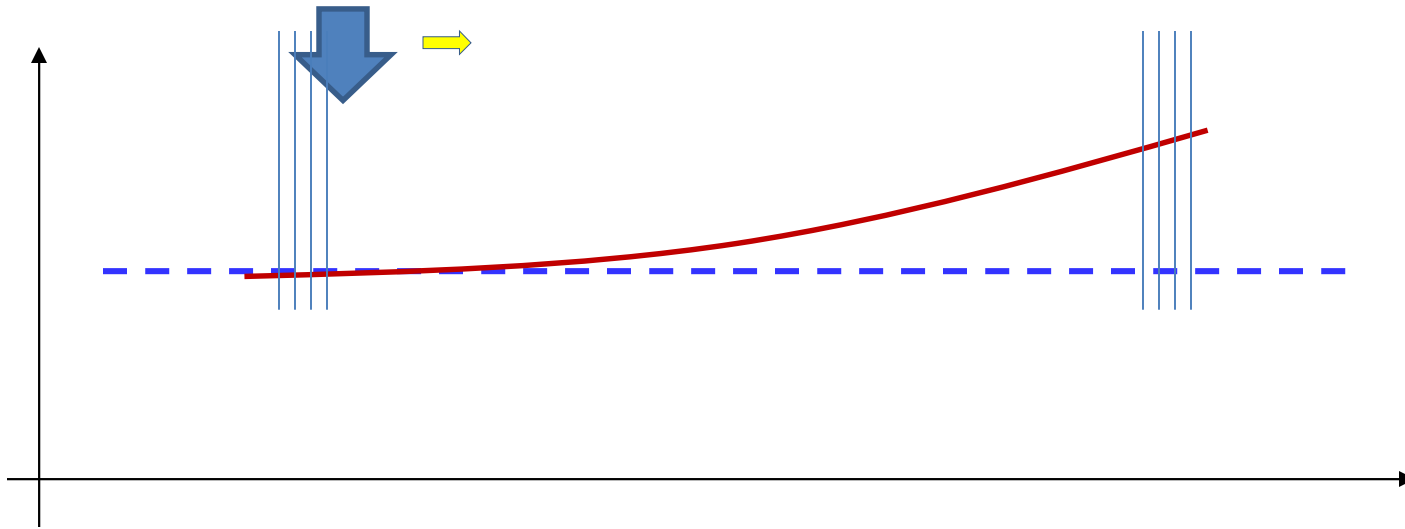
- Until *Loss* has converged

Caveats: order of presentation



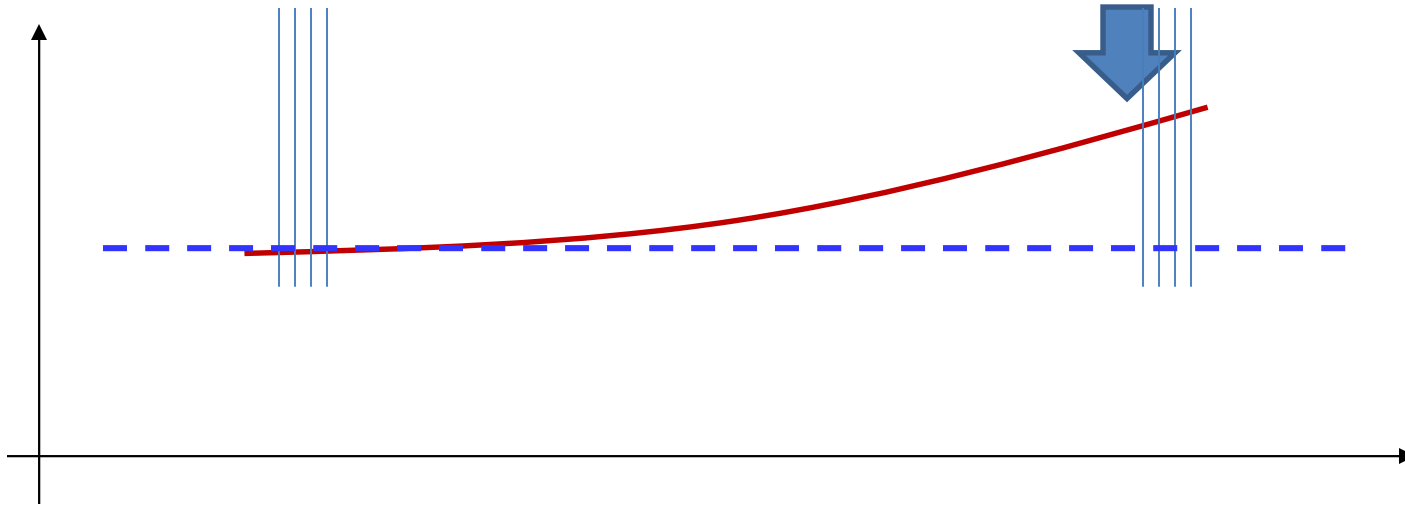
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



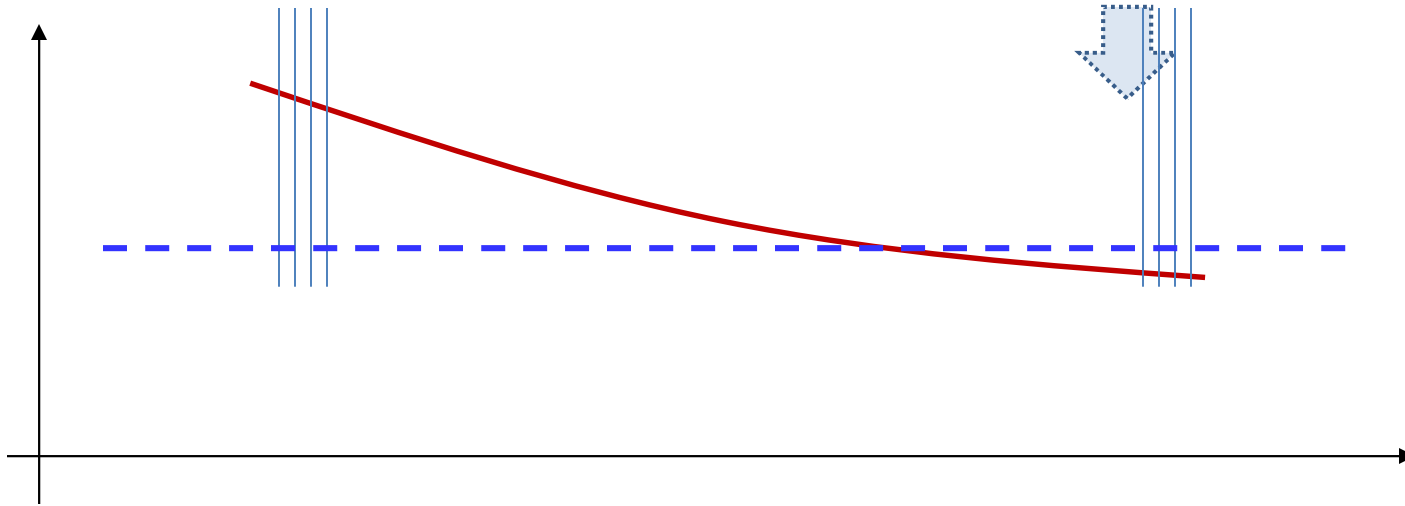
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



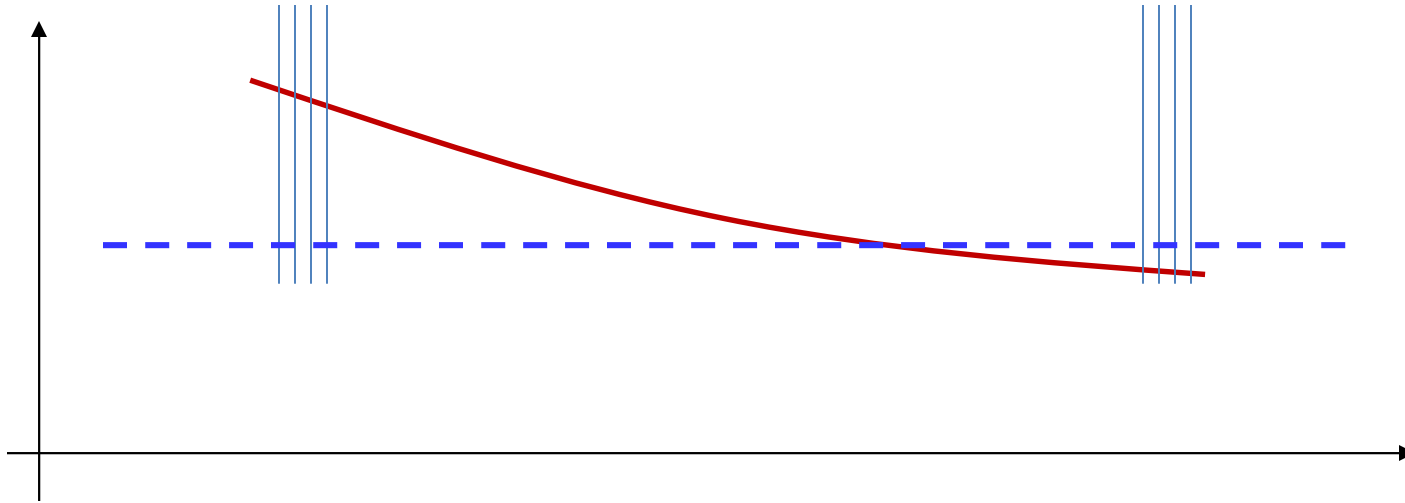
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



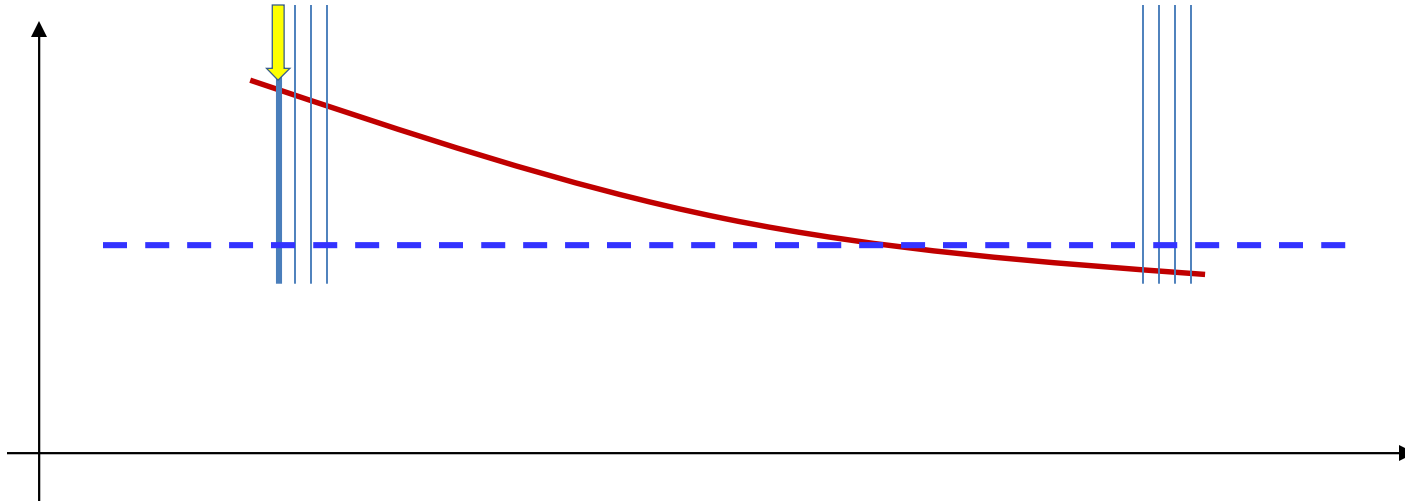
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



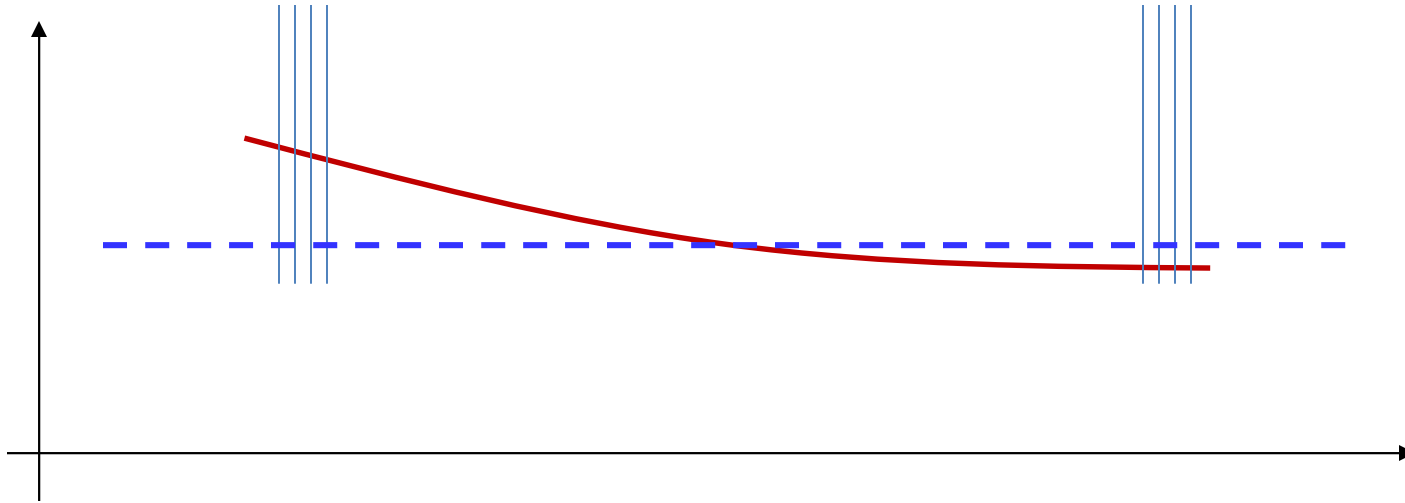
- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly* to get more convergent behavior

Caveats: order of presentation



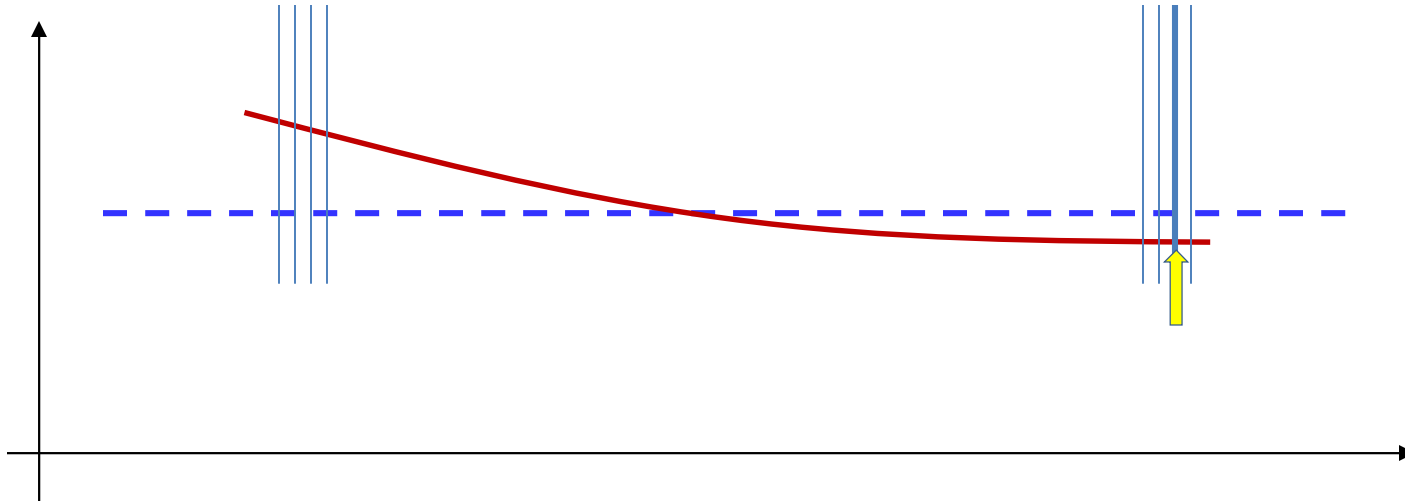
- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly* to get more convergent behavior

Caveats: order of presentation



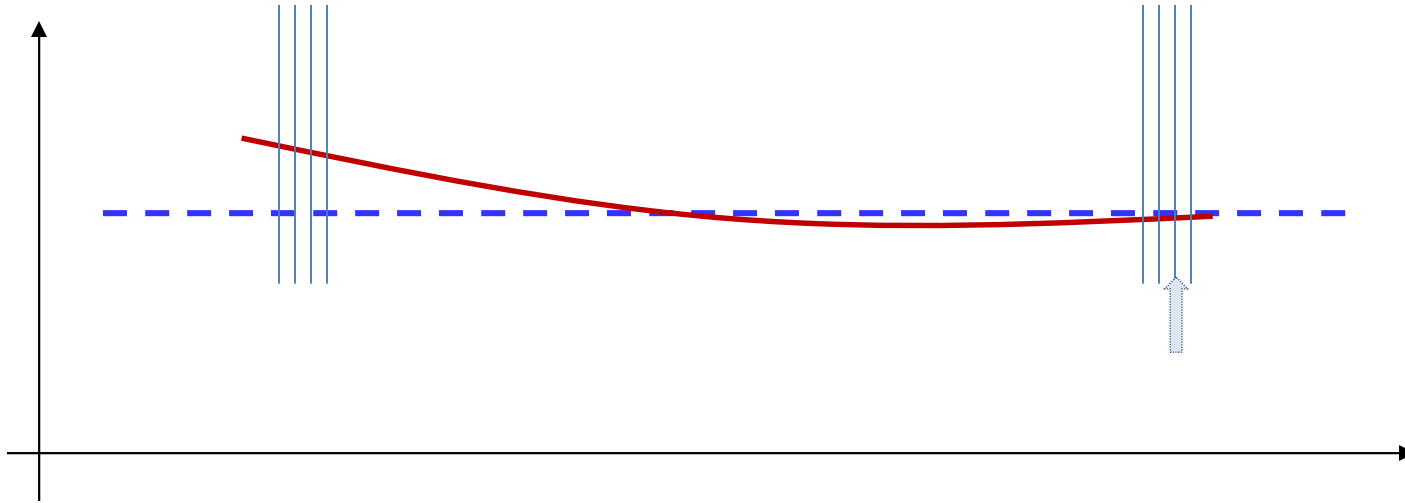
- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly* to get more convergent behavior

Caveats: order of presentation



- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly* to get more convergent behavior

Caveats: order of presentation



- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly* to get more convergent behavior

Incremental Update: Stochastic Gradient Descent

- Given $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights W_1, W_2, \dots, W_K
- Do:
 - Randomly permute $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
 - For all $t = 1:T$
 - For every layer k :
 - Compute $\nabla_{W_k} \text{Div}(\mathbf{Y}_t, \mathbf{d}_t)$
 - Update
$$W_k = W_k - \eta \nabla_{W_k} \text{Div}(\mathbf{Y}_t, \mathbf{d}_t)^T$$
- Until *Loss* has converged

Story so far

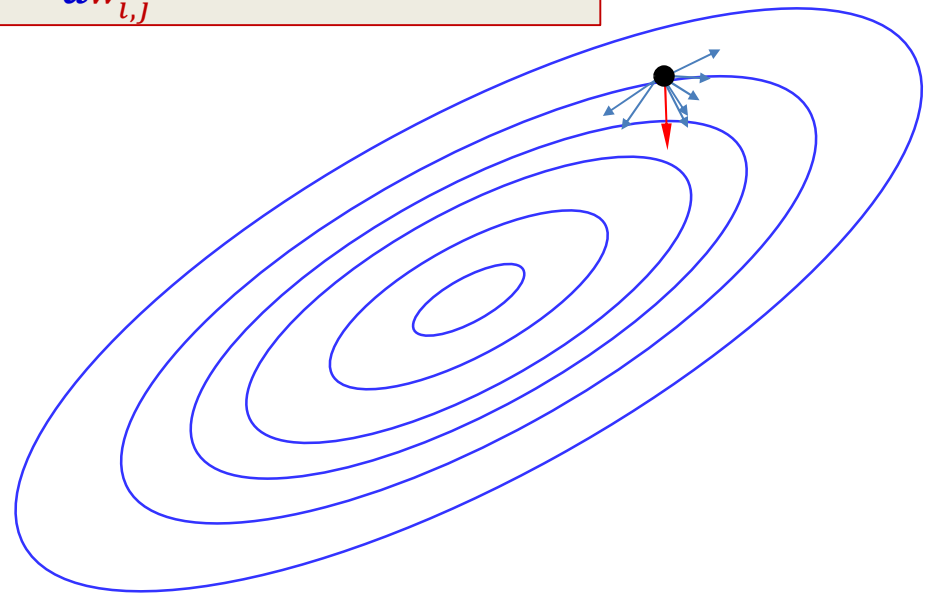
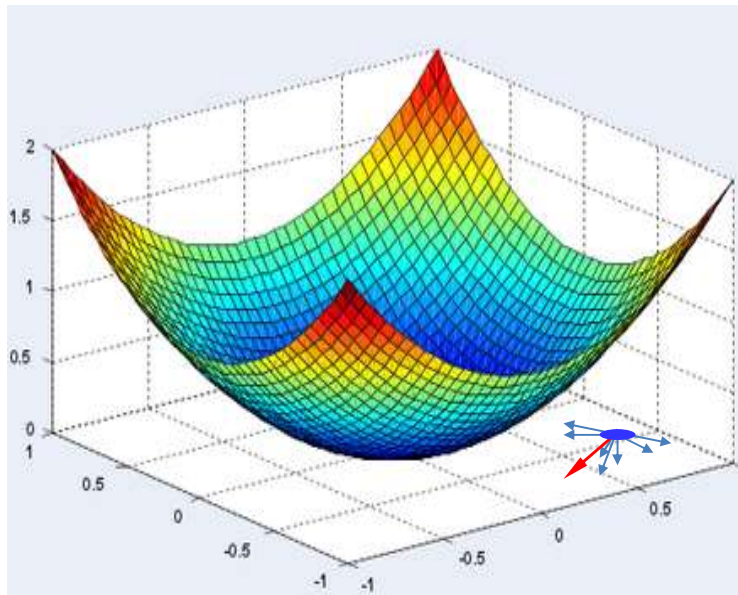
- In any gradient descent optimization problem, presenting training instances incrementally can be more effective than presenting them all at once
 - Provided training instances are provided in random order
 - “Stochastic Gradient Descent”
- This also holds for training neural networks

Explanations and restrictions

- So why does this process of incremental updates work?
- Under what conditions?
- For “why”: first consider a simplistic explanation that’s often given
 - Look at an extreme example

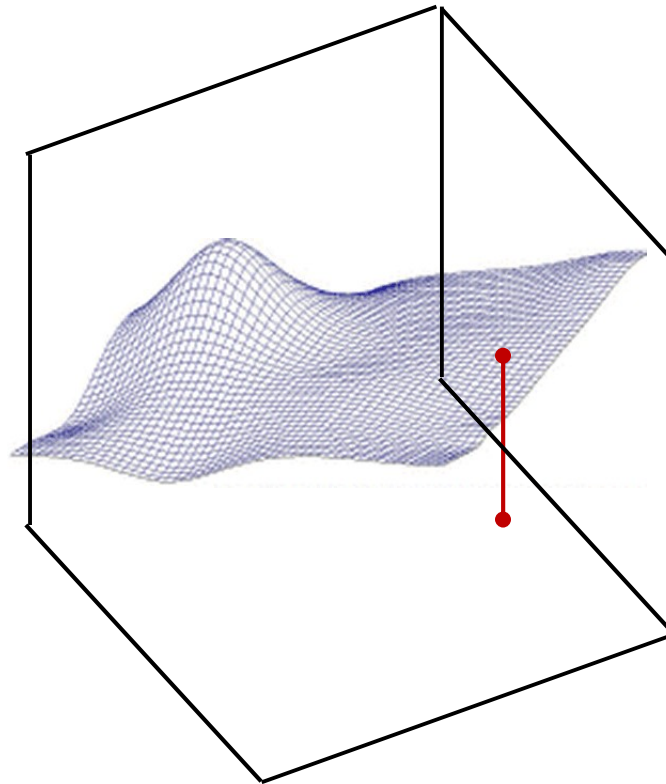
The expected behavior of the gradient

$$\frac{dE(W^{(1)}, W^{(2)}, \dots, W^{(K)})}{dw_{i,j}^{(k)}} = \frac{1}{T} \sum_i \frac{dDiv(Y(X_i), d_i; W^{(1)}, W^{(2)}, \dots, W^{(K)})}{dw_{i,j}^{(k)}}$$



- The individual training instances contribute different directions to the overall gradient
 - The final gradient points is the average of individual gradients
 - It points towards the *net* direction

Extreme example

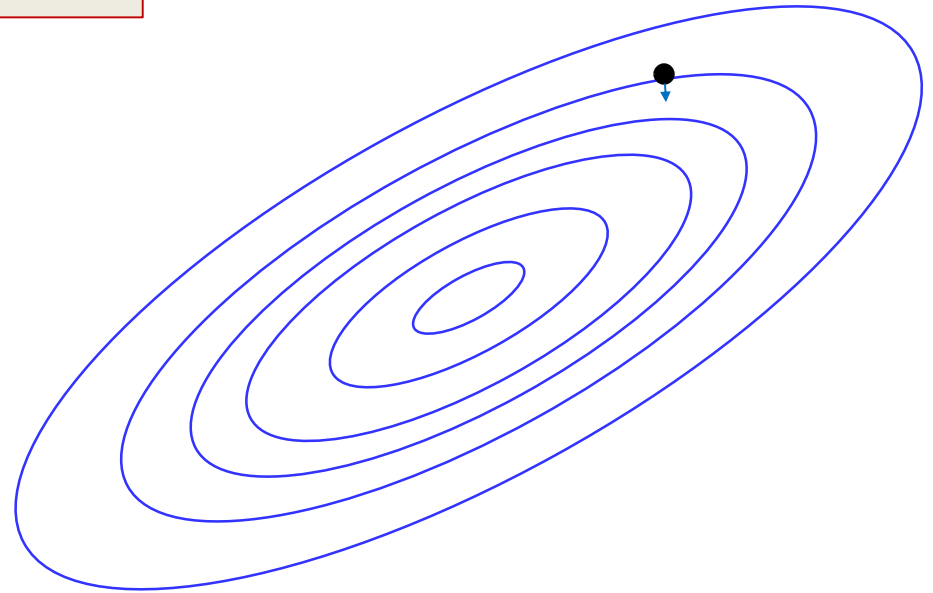
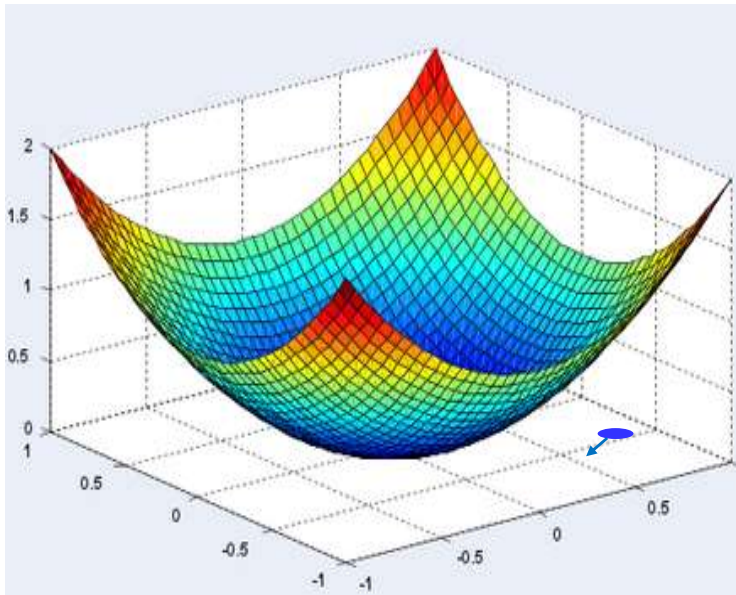


$$X_1 = X_2 = \dots = X_T$$

- Extreme instance of data clotting: all the training instances are exactly the same

The expected behavior of the gradient

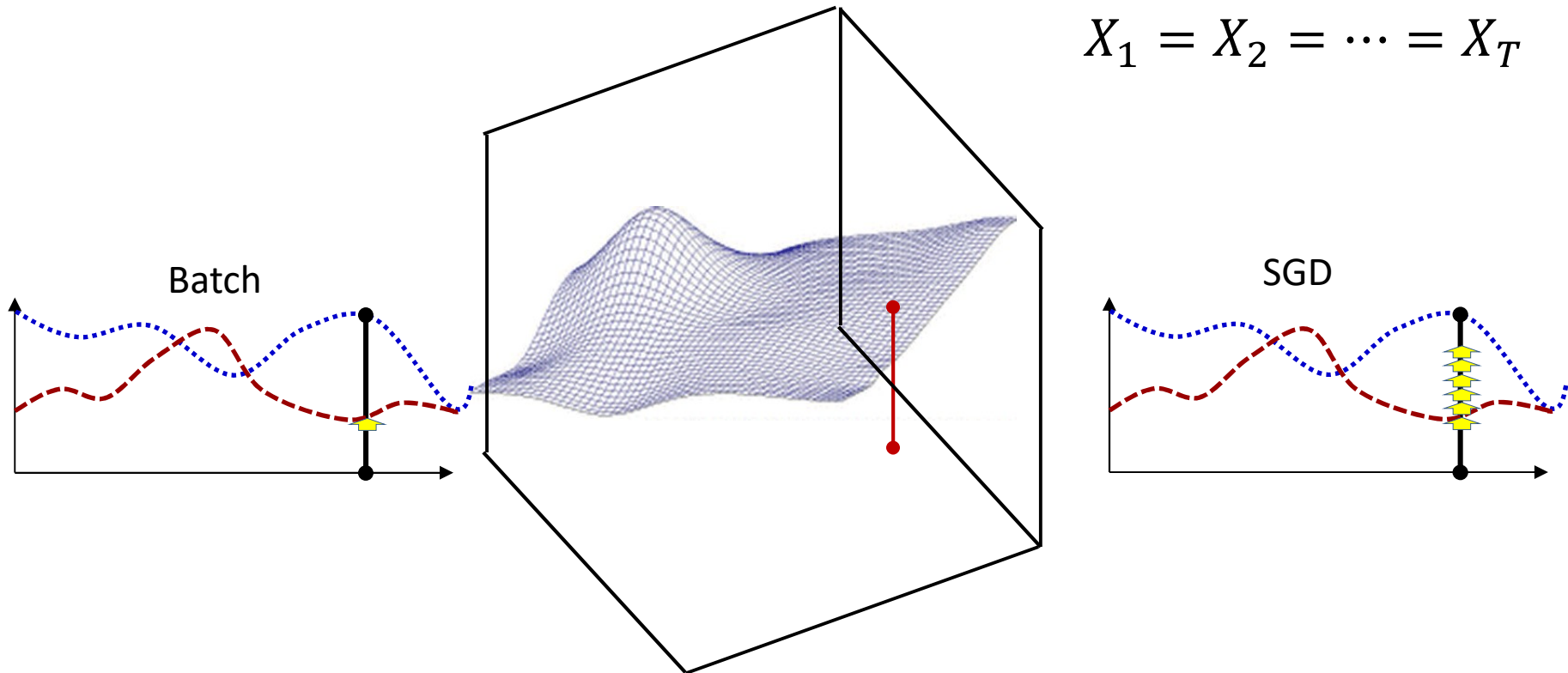
$$\frac{dE}{dw_{i,j}^{(k)}} = \frac{1}{T} \sum_i \frac{d\text{Div}(\mathbf{Y}(\mathbf{X}_i), d_i)}{dw_{i,j}^{(k)}} = \frac{d\text{Div}(\mathbf{Y}(\mathbf{X}_i), d_i)}{dw_{i,j}^{(k)}}$$



- The individual training instance contribute identical directions to the overall gradient
 - The final gradient points is simply the gradient for an individual instance

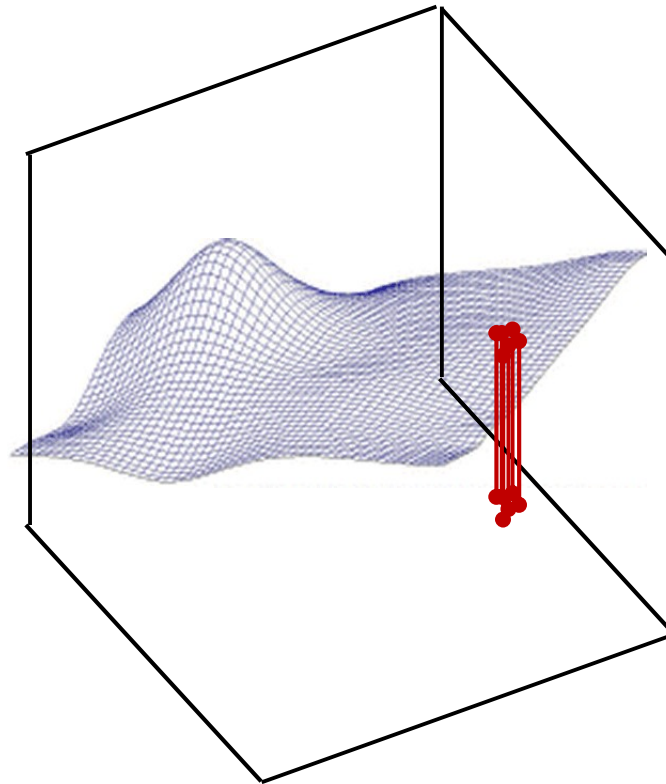
Batch vs SGD

$$X_1 = X_2 = \dots = X_T$$



- Batch gradient descent operates over T training instances to get a *single* update
- SGD gets T updates for the same computation

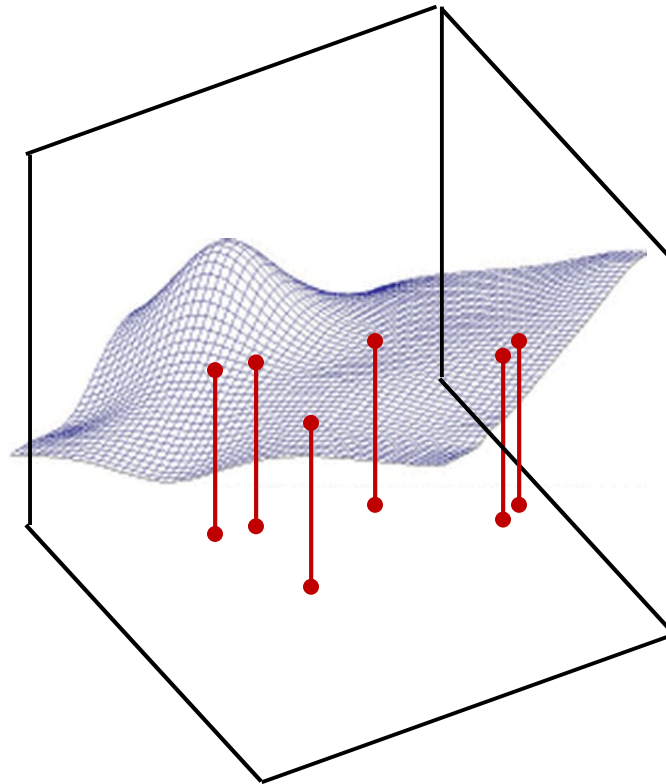
Clumpy data..



$$X_1 \approx X_2 \approx \dots \approx X_T$$

- Also holds if all the data are not identical, but are tightly clumped together

Clumpy data..

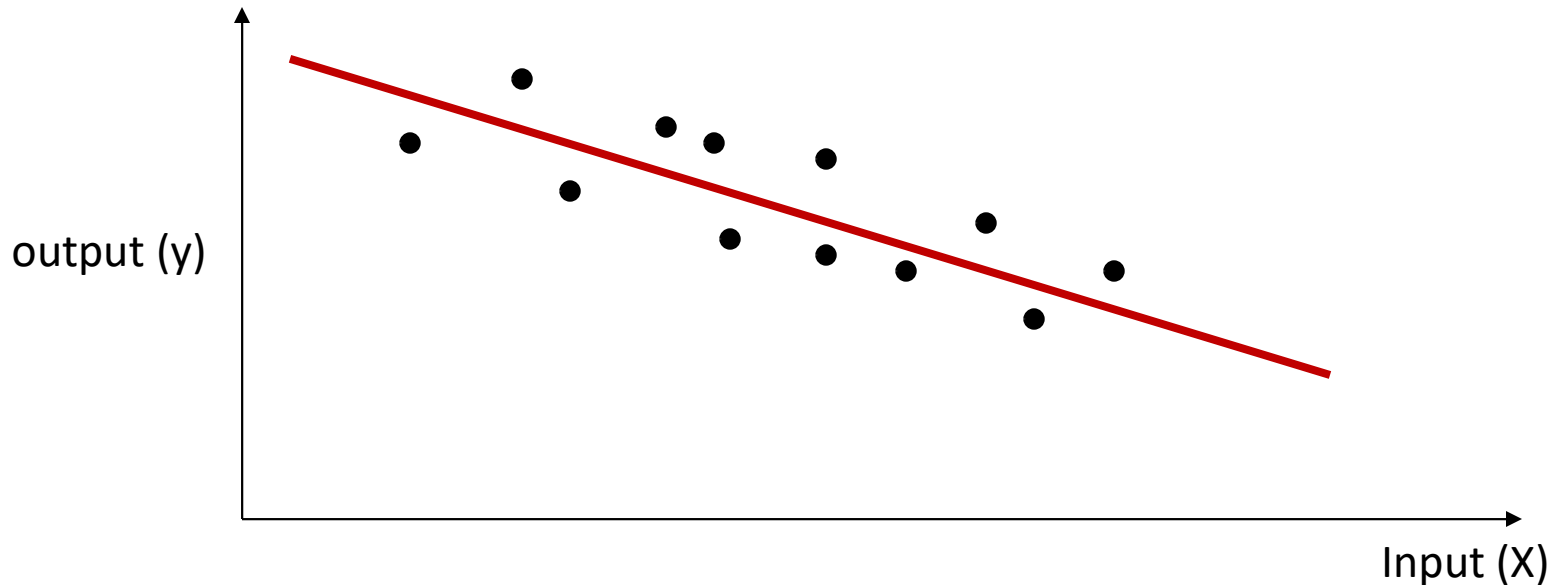


- As data get increasingly diverse, the benefits of incremental updates decrease, but do not entirely vanish

When does it work

- What are the considerations?
- And how well does it work?

Caveats: learning rate



- Except in the case of a perfect fit, even an optimal overall fit will look incorrect to *individual* instances
 - Correcting the function for individual instances will lead to never-ending, non-convergent updates
 - We must *shrink* the learning rate with iterations to prevent this
 - Correction for individual instances with the eventual miniscule learning rates will not modify the function

Incremental Update: Stochastic Gradient Descent

- Given $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights W_1, W_2, \dots, W_K ; $j = 0$
- Do:
 - Randomly permute $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
 - For all $t = 1:T$
 - $j = j + 1$
 - For every layer k :
 - Compute $\nabla_{W_k} \text{Div}(\mathbf{Y}_t, \mathbf{d}_t)$
 - Update
$$W_k = W_k - \eta_j \nabla_{W_k} \text{Div}(\mathbf{Y}_t, \mathbf{d}_t)^T$$
- Until *Loss* has converged

Incremental Update: Stochastic Gradient Descent

- Given $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights W_1, W_2, \dots, W_K ; $j = 0$
- Do:

– Randomly permute $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$

– For all $t = 1:T$

• $j = j + 1$

• For every layer k :

– Compute $\nabla_{W_k} \text{Div}(\mathbf{Y}_t, \mathbf{d}_t)$

– Update

$$W_k = W_k - \eta_j \nabla_{W_k} \text{Div}(\mathbf{Y}_t, \mathbf{d}_t)^T$$

- Until *Loss* has converged

Randomize input order

Learning rate reduces with j

SGD convergence

- SGD converges “almost surely” to a global or local minimum for most functions
 - Sufficient condition: step sizes follow the following conditions (Robbins and Munro 1951)

$$\sum_k \eta_k = \infty$$

- Eventually the entire parameter space can be searched

$$\sum_k \eta_k^2 < \infty$$

- The steps shrink
 - The fastest converging series that satisfies both above requirements is

$$\eta_k \propto \frac{1}{k}$$

- This is the optimal rate of shrinking the step size for strongly convex functions
 - More generally, the learning rates are heuristically determined
- If the loss is convex, SGD converges to the optimal solution
- For non-convex losses SGD converges to a local minimum

SGD convergence

- We will define convergence in terms of the number of iterations taken to get within ϵ of the optimal solution

- $|f(W^{(k)}) - f(W^*)| < \epsilon$
- Note: $f(W)$ here is the optimization objective on the *entire* training data, although SGD itself updates after every training instance

- Using the optimal learning rate $1/k$, for *strongly convex* functions,

$$|f(W^{(k)}) - f(W^*)| < \frac{1}{k} |f(W^{(0)}) - f(W^*)|$$

- Strongly convex \rightarrow Can be placed inside a quadratic bowl, touching at any point
 - Giving us the iterations to ϵ convergence as $O\left(\frac{1}{\epsilon}\right)$
- For generically convex (but not strongly convex) function, various proofs report an ϵ convergence of $\frac{1}{\sqrt{k}}$ using a learning rate of $\frac{1}{\sqrt{k}}$.

Batch gradient convergence

- In contrast, using the batch update method, for *strongly convex* functions,

$$|f(W^{(k)}) - f(W^*)| < c^k |f(W^{(0)}) - f(W^*)|$$

– Giving us the iterations to ϵ convergence as $O\left(\log\left(\frac{1}{\epsilon}\right)\right)$

- For generic convex functions, iterations to ϵ convergence is $O\left(\frac{1}{\epsilon}\right)$
- Batch gradients converge “faster”
 - But SGD performs T updates for every batch update

SGD Convergence: Loss value

If:

- f is λ -strongly convex, and
- at step t we have a noisy estimate of the subgradient \hat{g}_t with $\mathbb{E}[\|\hat{g}_t\|^2] \leq G^2$ for all t ,
- and we use step size $\eta_t = 1/\lambda t$

Then for any $T > 1$:

$$\mathbb{E}[f(w_T) - f(w^*)] \leq \frac{17G^2(1 + \log(T))}{\lambda T}$$

SGD Convergence

- We can bound the expected difference between the loss over our data using the optimal weights w^* and the weights w_T at **any single iteration** to $\mathcal{O}\left(\frac{\log(T)}{T}\right)$ for strongly convex loss or $\mathcal{O}\left(\frac{\log(T)}{\sqrt{T}}\right)$ for convex loss
- Averaging schemes can improve the bound to $\mathcal{O}\left(\frac{1}{T}\right)$ and $\mathcal{O}\left(\frac{1}{\sqrt{T}}\right)$
- **Smoothness** of the loss is **not required**

SGD Convergence and weight averaging

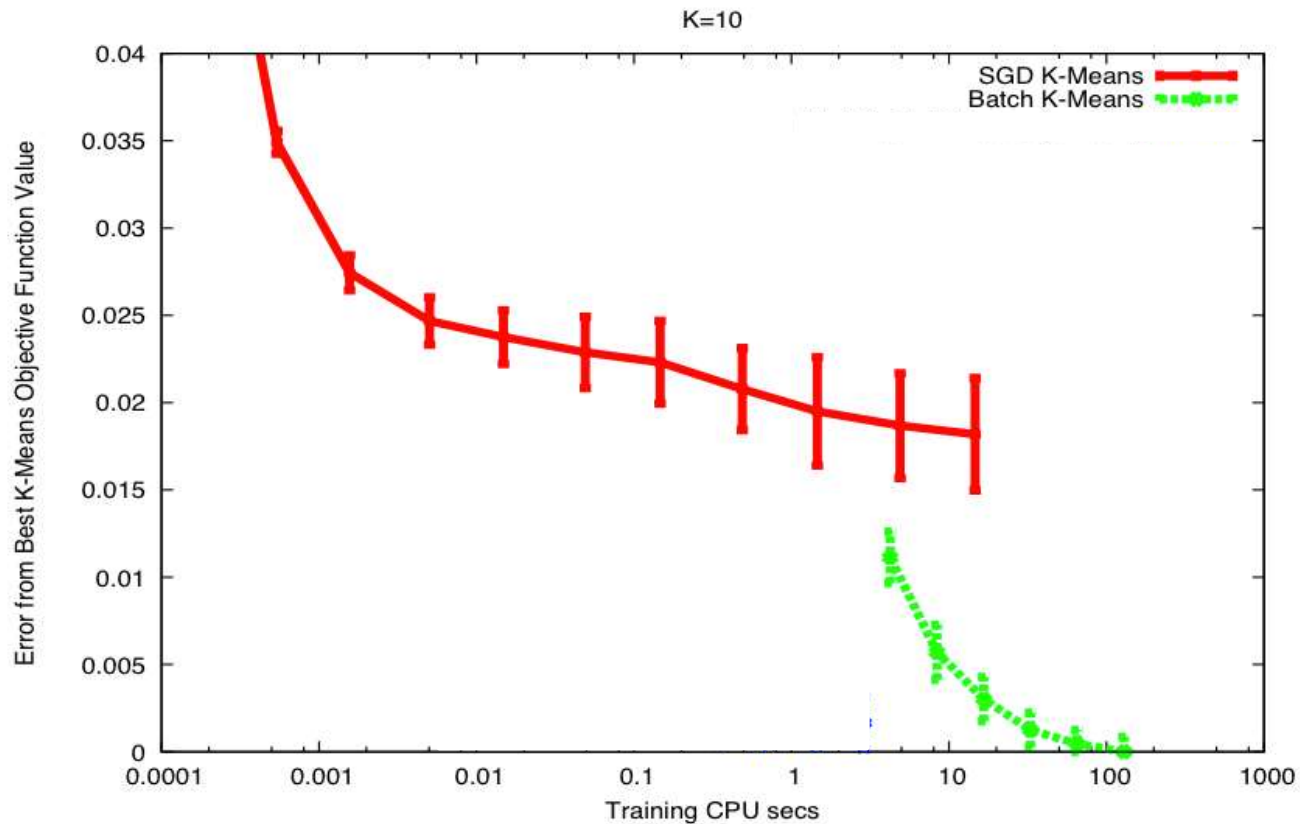
Polynomial Decay Averaging:

$$\bar{w}_t^\gamma = \left(1 - \frac{\gamma + 1}{t + \gamma}\right) \bar{w}_{t-1}^\gamma + \frac{\gamma + 1}{t + \gamma} w_t$$

With γ some small positive constant, e.g. $\gamma = 3$

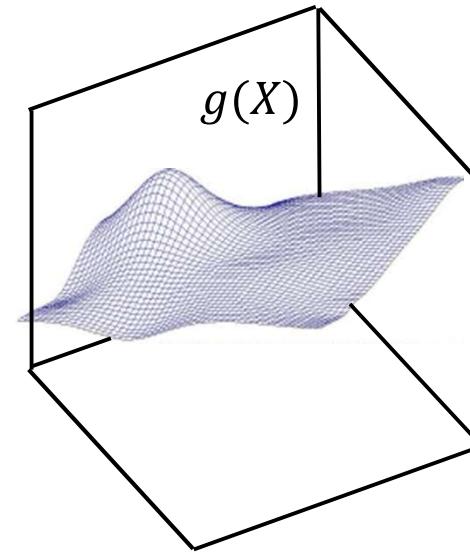
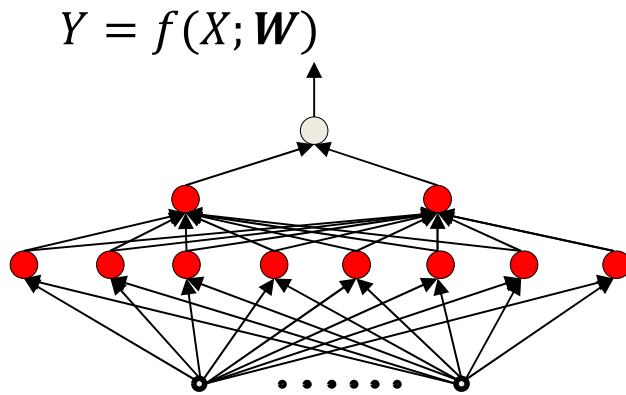
Achieves $\mathcal{O}\left(\frac{1}{T}\right)$ (strongly convex) and $\mathcal{O}\left(\frac{1}{\sqrt{T}}\right)$ (convex) convergence

SGD example



- A simpler problem: K-means
- Note: SGD converges faster
 - But to a poorer minimum
- Also note the rather large variation between runs
 - Let's try to understand these results..

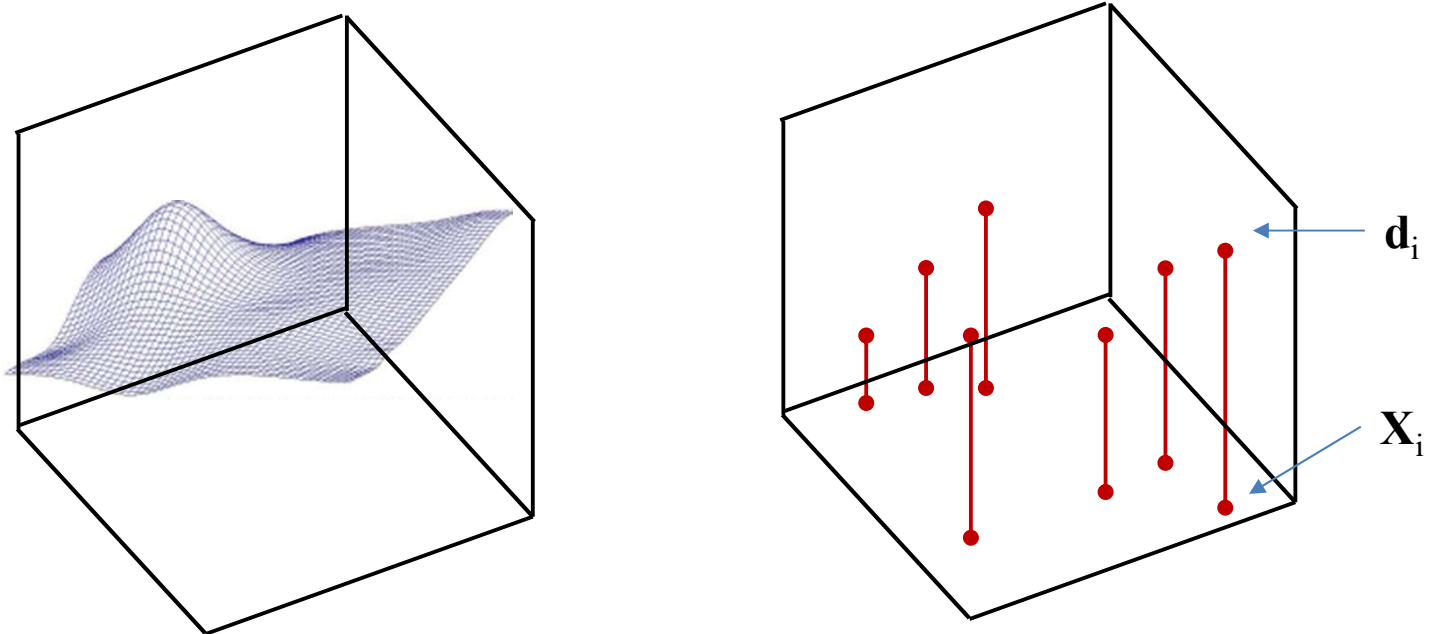
Recall: Modelling a function



- To learn a network $f(X; \mathbf{W})$ to model a function $g(X)$ we minimize the *expected divergence*

$$\begin{aligned}\widehat{\mathbf{W}} &= \operatorname{argmin}_{\mathbf{W}} \int_X \operatorname{div}(f(X; \mathbf{W}), g(X)) P(X) dX \\ &= \operatorname{argmin}_{\mathbf{W}} E[\operatorname{div}(f(X; \mathbf{W}), g(X))]\end{aligned}$$

Recall: The *Empirical* risk



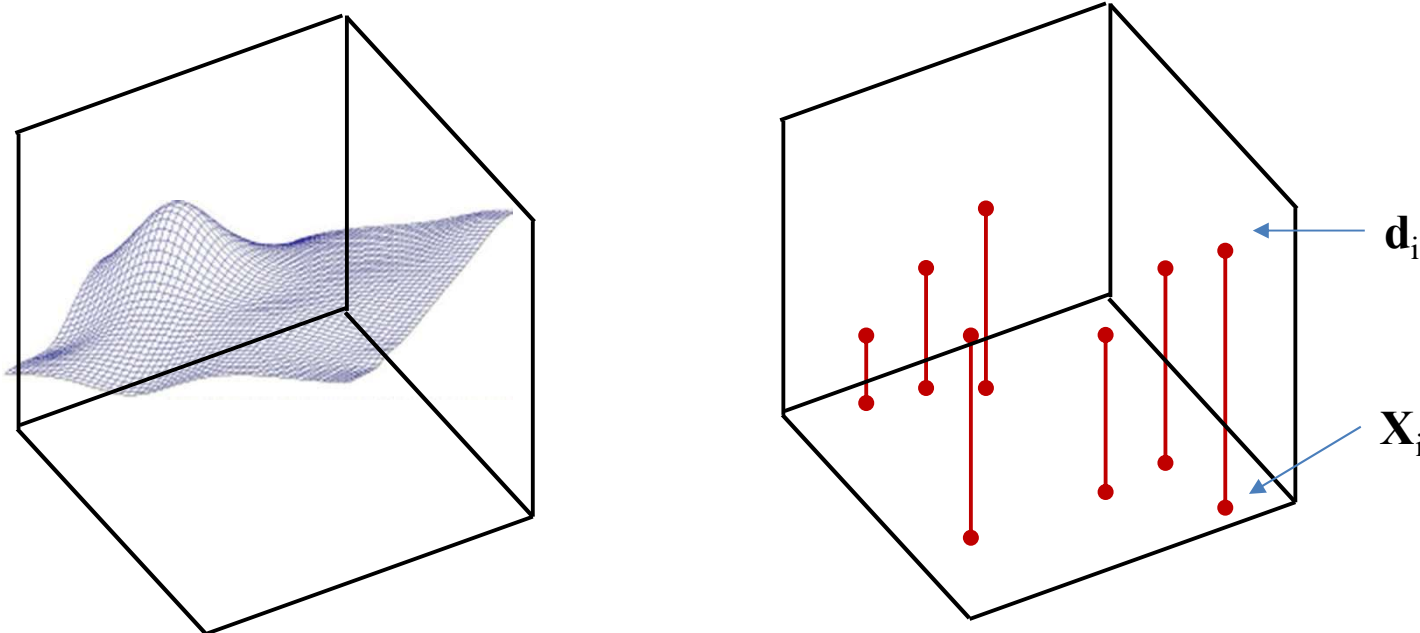
- In practice, we minimize the *empirical risk (or loss)*

$$Loss(W) = \frac{1}{N} \sum_{i=1}^N div(f(X_i; W), d_i)$$
$$\hat{W} = \underset{W}{\operatorname{argmin}} Loss(W)$$

- The *expected value* of the *empirical risk* is actually the *expected divergence*

$$E[Loss(W)] = E[div(f(X; W), g(X))]$$

Recall: The *Empirical* risk



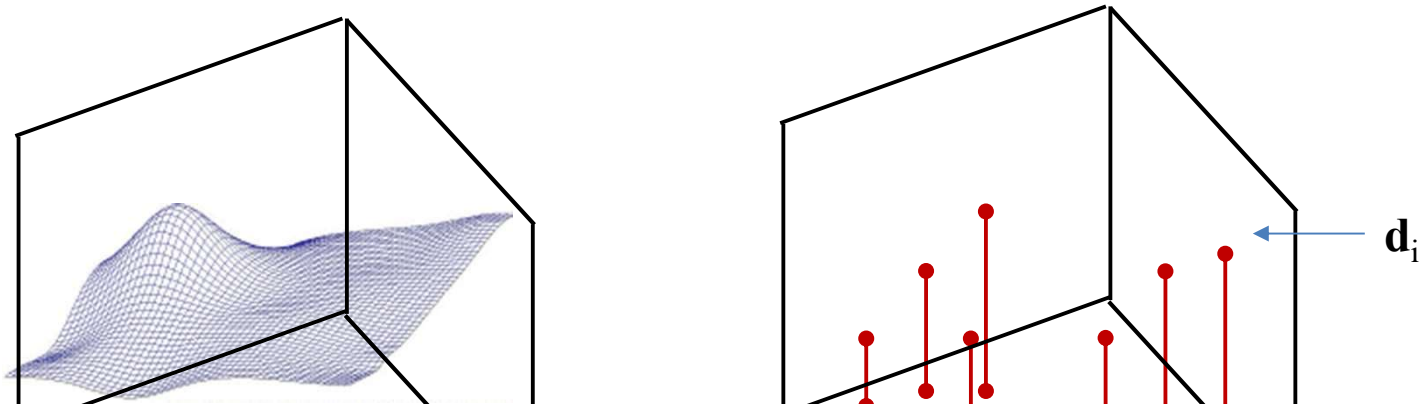
- In practice, we minimize the *empirical risk* (or *loss*)

$$Loss(W) = \frac{1}{N} \sum_{i=1}^N div(f(X_i; W), d_i)$$

The empirical risk is an *unbiased* estimate of the expected divergence
Though there is no guarantee that minimizing it will minimize the
expected divergence

$$E[Loss(W)] = E[div(f(X; W), g(X))]$$

Recall: The *Empirical* risk



The variance of the empirical risk: $\text{var}(\text{Loss}) = 1/N \text{ var}(\text{div})$

The variance of the estimator is proportional to $1/N$

The larger this variance, the greater the likelihood that the W that minimizes the empirical risk will differ significantly from the W that minimizes the expected divergence

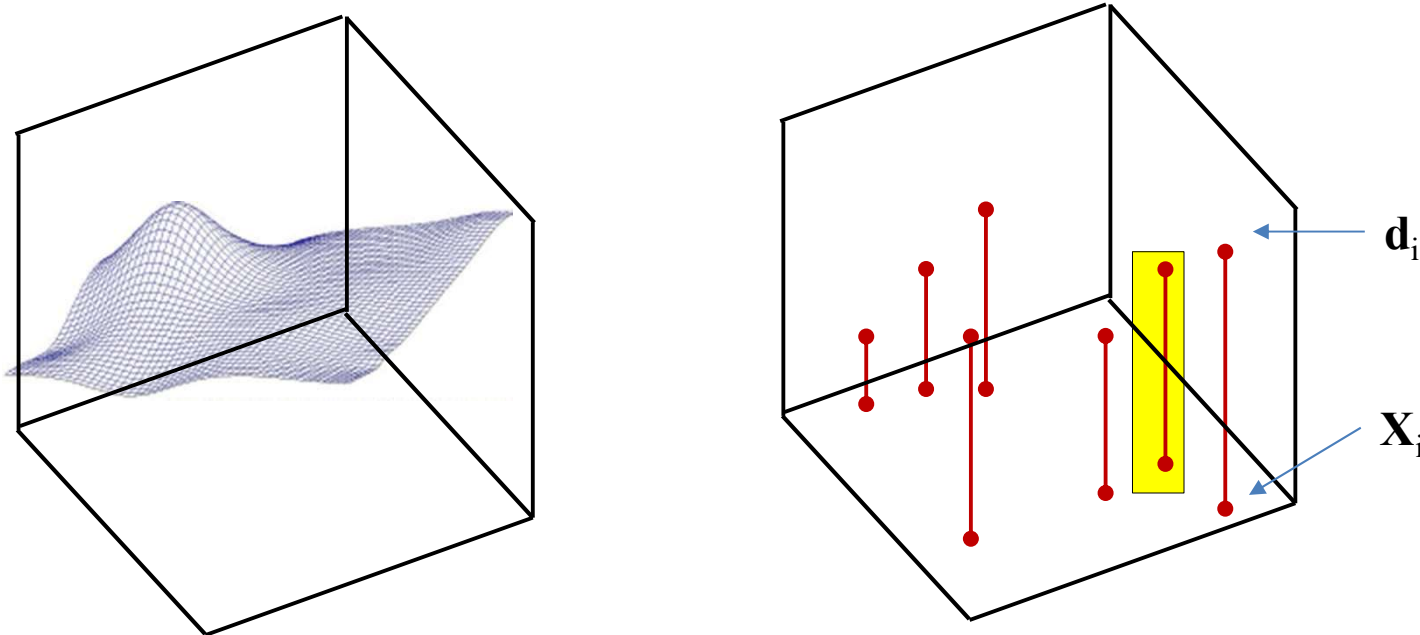
$$\text{Loss}(W) = \frac{1}{N} \sum_{i=1}^N \text{div}(f(X_i; W), d_i)$$

The empirical risk is an *unbiased* estimate of the expected divergence

Though there is no guarantee that minimizing it will minimize the expected divergence

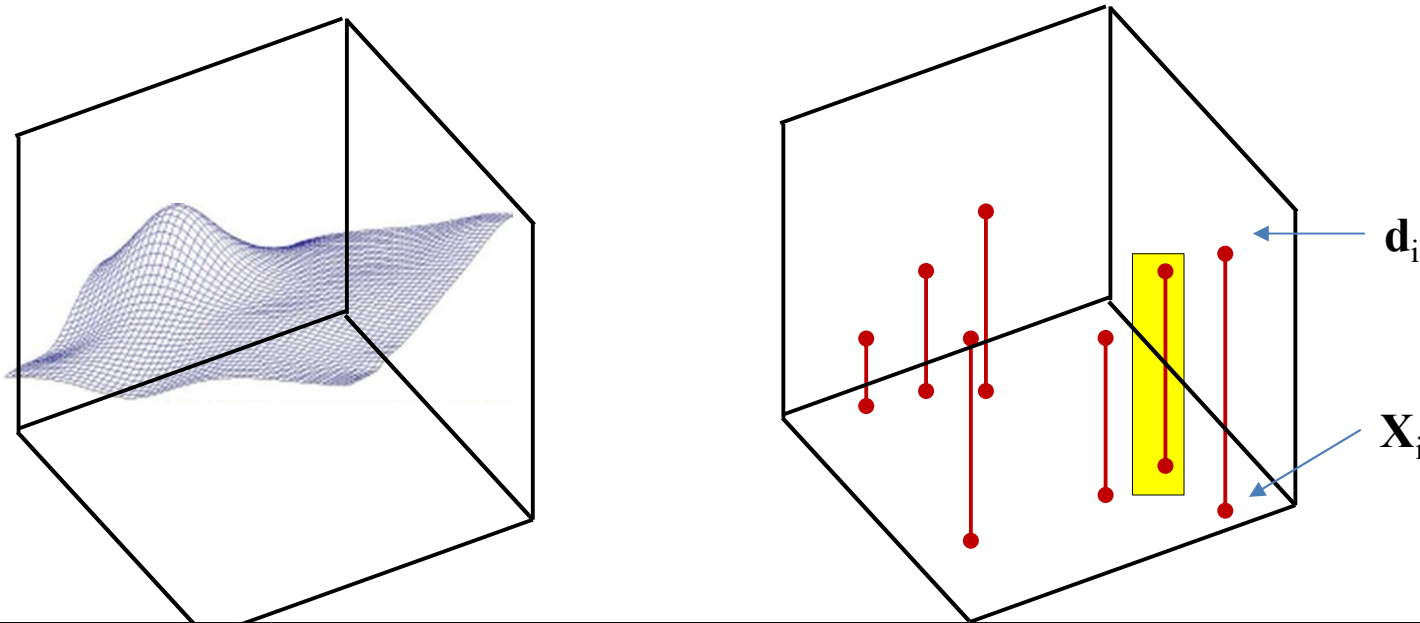
$$E[\text{Loss}(W)] = E[\text{div}(f(X; W), g(X))]$$

SGD



- At each iteration, **SGD** focuses on the divergence of a **single** sample $div(f(X_i; W), d_i)$
- The *expected value* of the *sample error* is **still** the *expected divergence* $E[div(f(X; W), g(X))]$

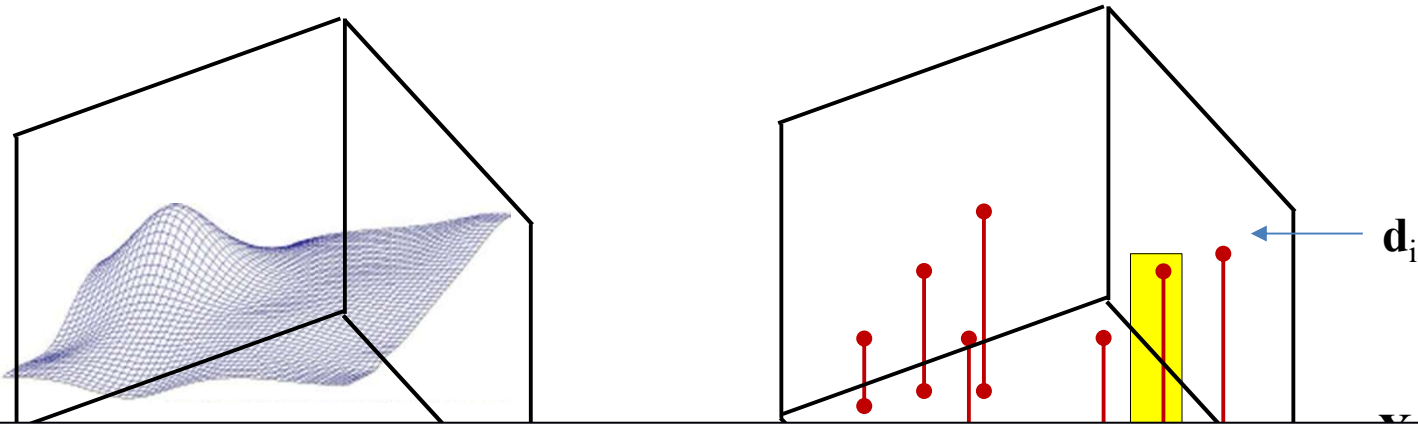
SGD



The sample divergence is also an *unbiased* estimate of the expected error

- At each iteration, **SGD** focuses on the divergence of a **single** sample $div(f(X_i; W), d_i)$
- The *expected value* of the *sample error* is **still** the *expected divergence* $E[div(f(X; W), g(X))]$

SGD

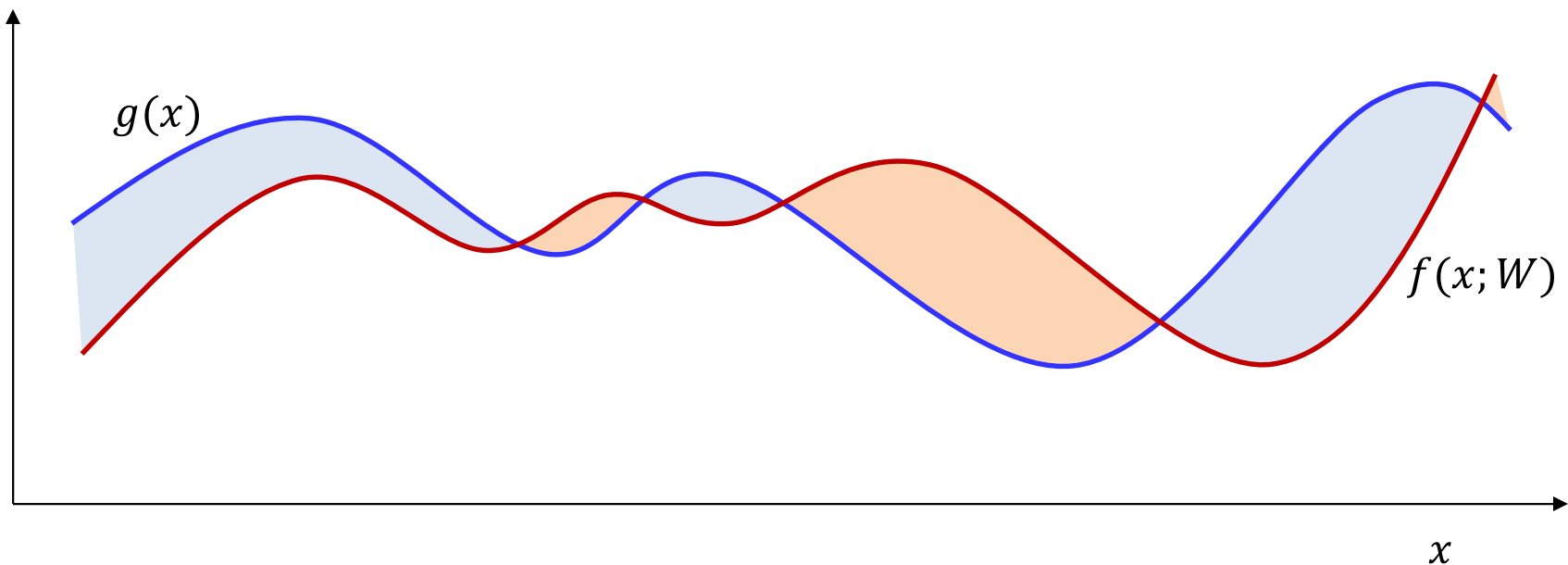


The variance of the sample divergence is the variance of the divergence itself: $\text{var}(\text{div})$. This is N times the variance of the empirical average minimized by batch update

The sample divergence is also an *unbiased* estimate of the expected error

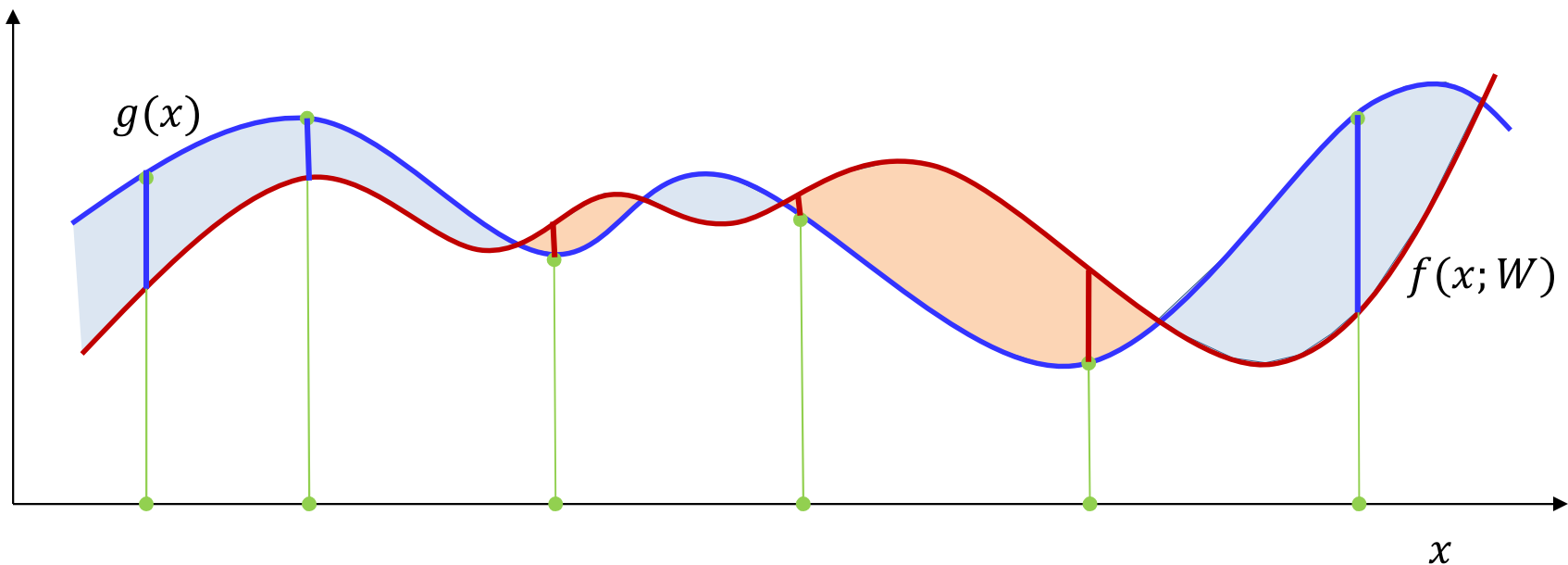
- At each iteration, **SGD** focuses on the divergence of a **single** sample $\text{div}(f(X_i; W), d_i)$
- The *expected value* of the *sample error* is **still** the *expected divergence* $E[\text{div}(f(X; W), g(X))]$

Explaining the variance



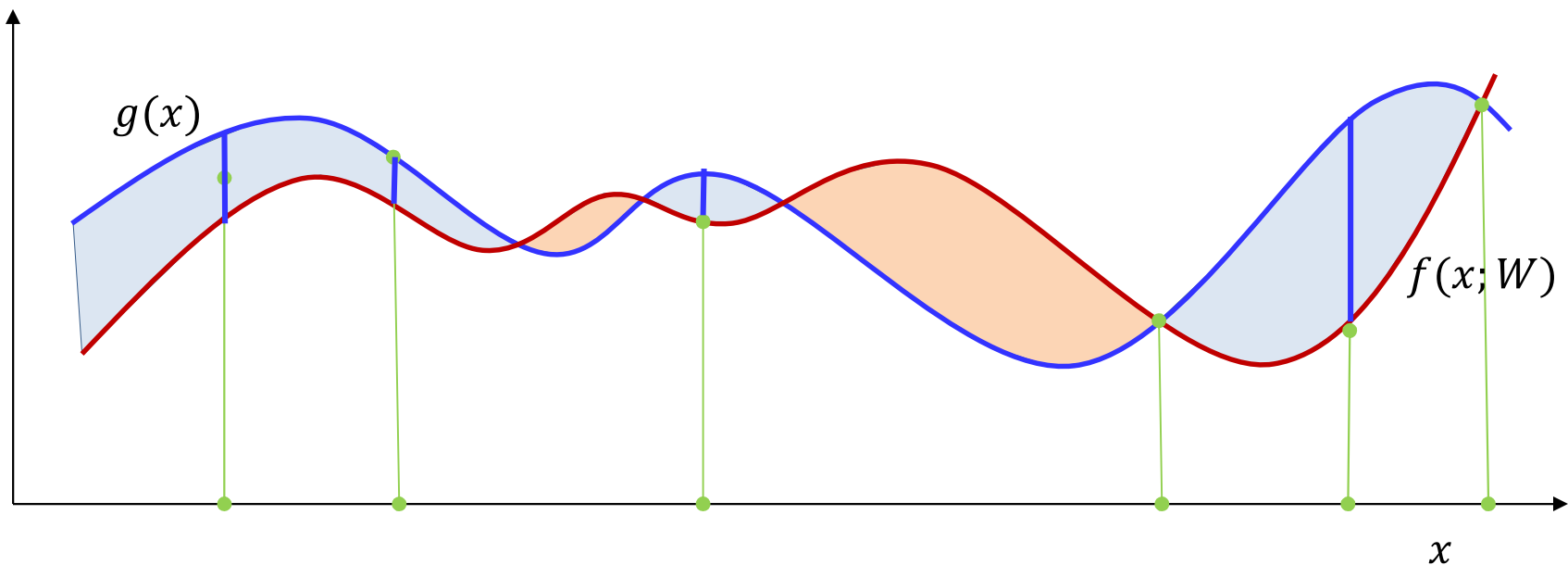
- The blue curve is the function being approximated
- The red curve is the approximation by the model at a given W
- The heights of the shaded regions represent the point-by-point error
 - The divergence is a function of the error
 - We want to find the W that minimizes the average divergence

Explaining the variance



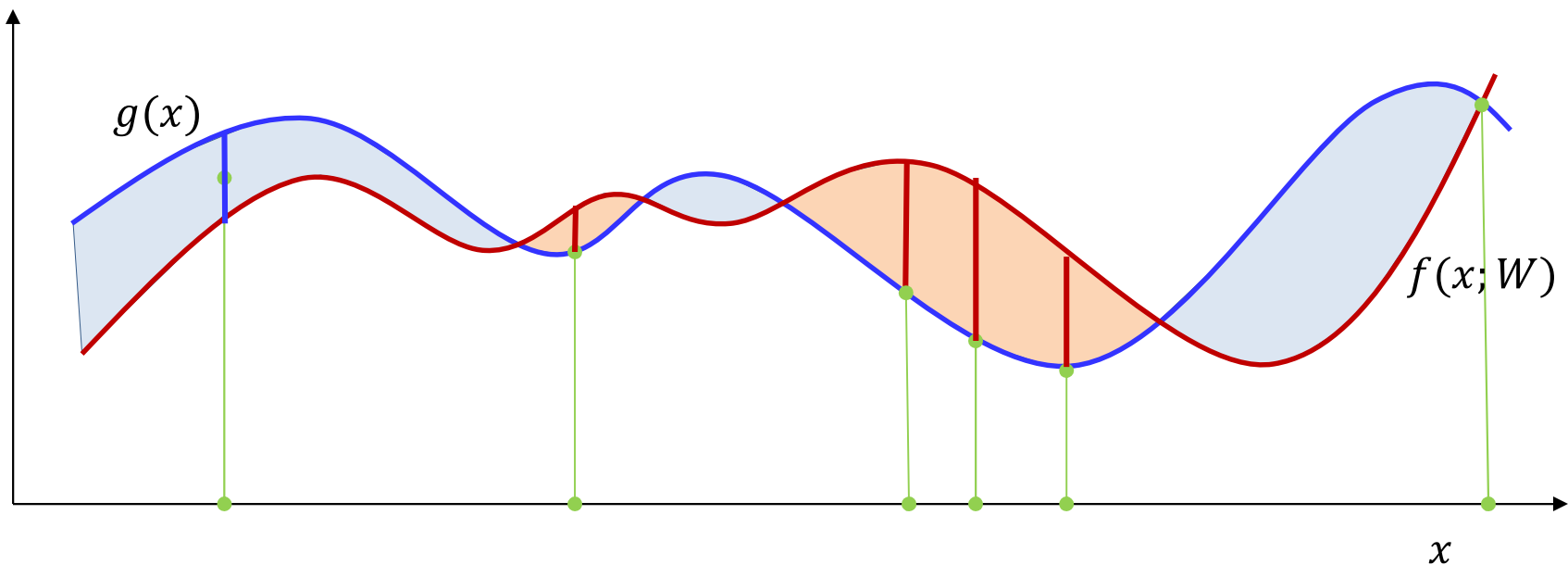
- Sample estimate approximates the shaded area with the average length of the lines

Explaining the variance



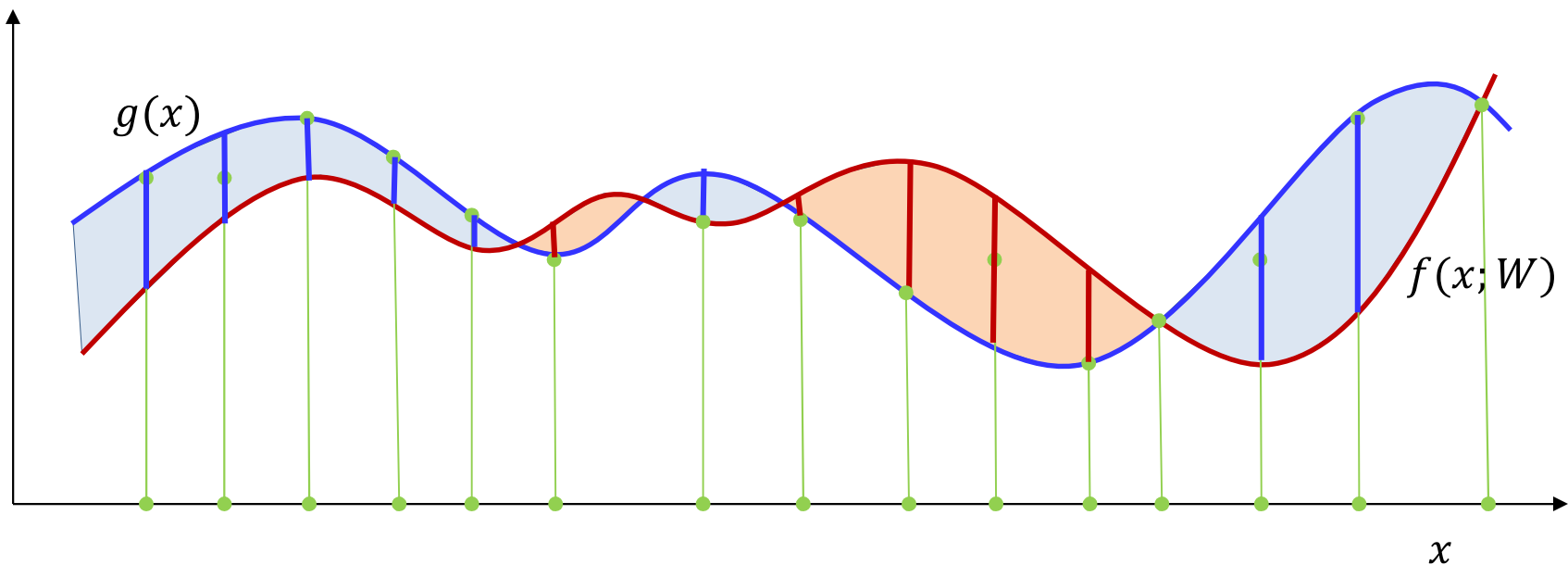
- Sample estimate approximates the shaded area with the average length of the lines
- This average length will change with position of the samples

Explaining the variance



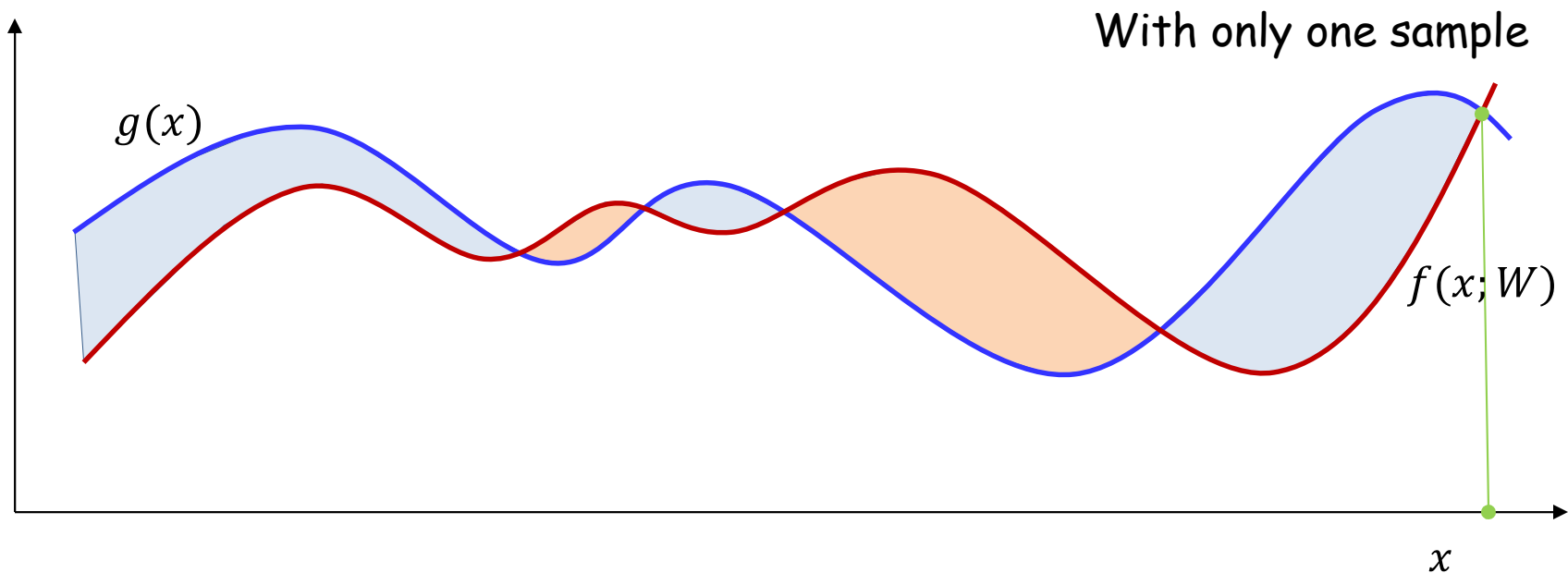
- Sample estimate approximates the shaded area with the average length of the lines
- This average length will change with position of the samples

Explaining the variance



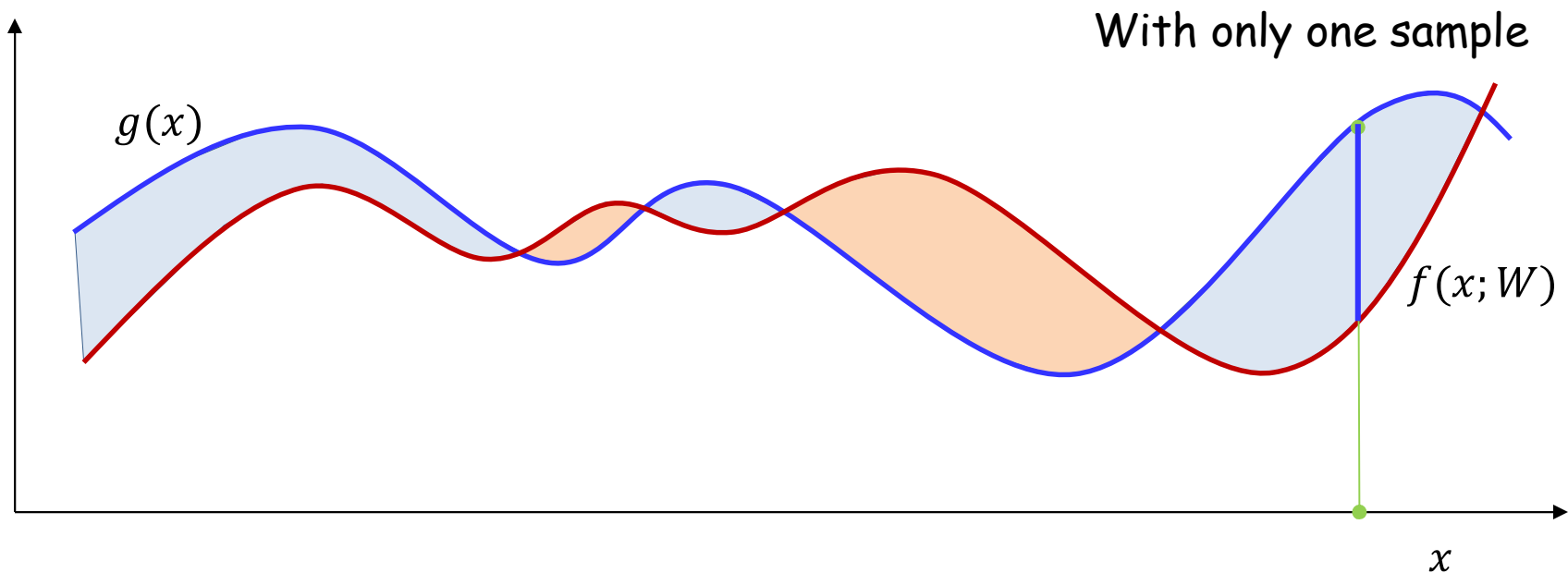
- Having more samples makes the estimate more robust to changes in the position of samples
 - The variance of the estimate is smaller

Explaining the variance



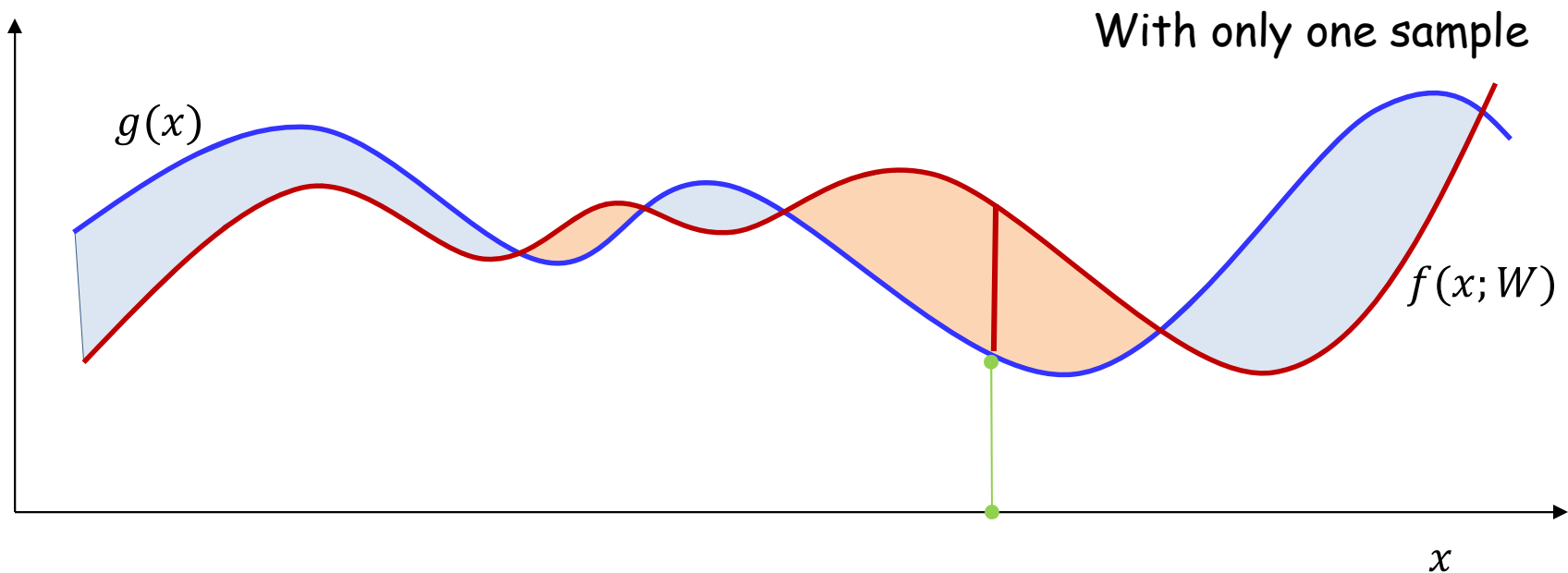
- Having very few samples makes the estimate swing wildly with the sample position
 - Since our estimator learns the W to minimize this estimate, the learned W too can swing wildly

Explaining the variance



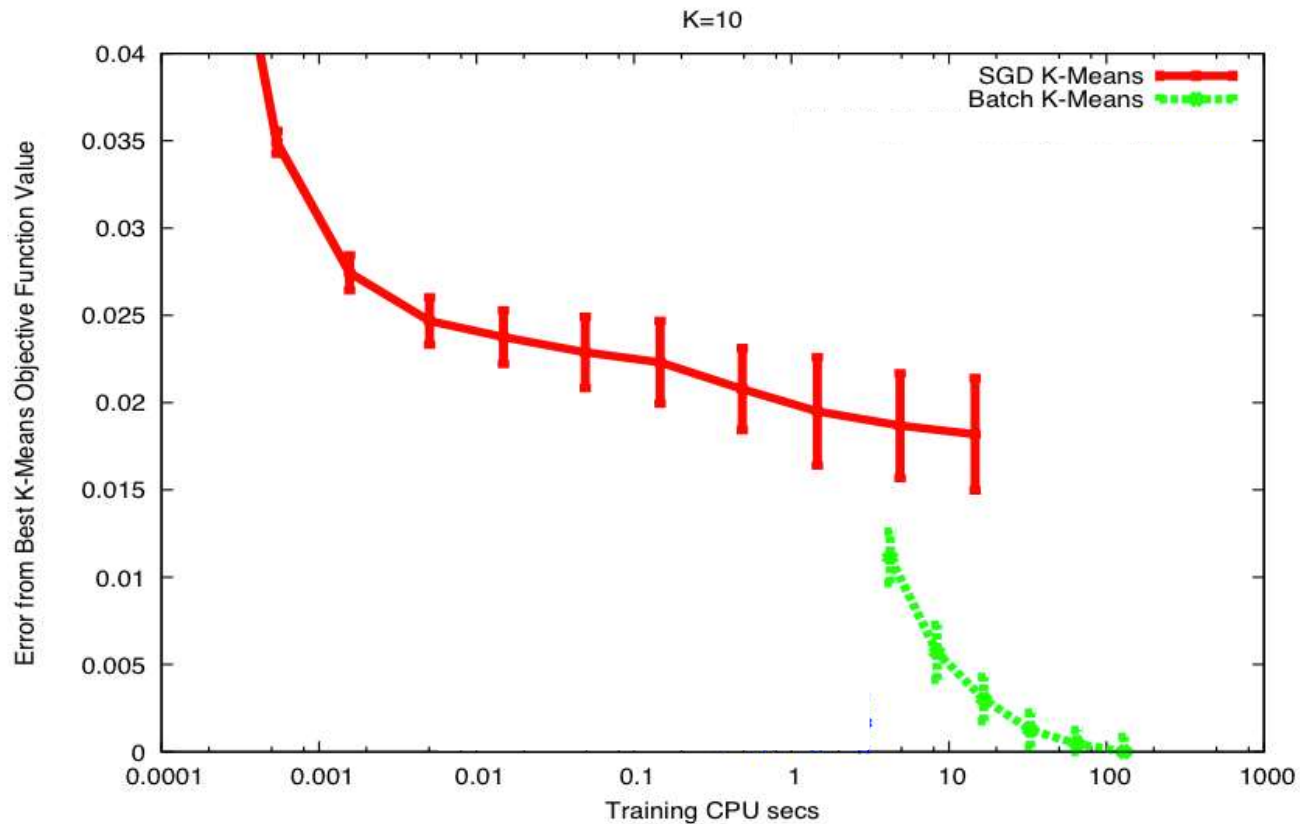
- Having very few samples makes the estimate swing wildly with the sample position
 - Since our estimator learns the W to minimize this estimate, the learned W too can swing wildly

Explaining the variance



- Having very few samples makes the estimate swing wildly with the sample position
 - Since our estimator learns the W to minimize this estimate, the learned W too can swing wildly

SGD example

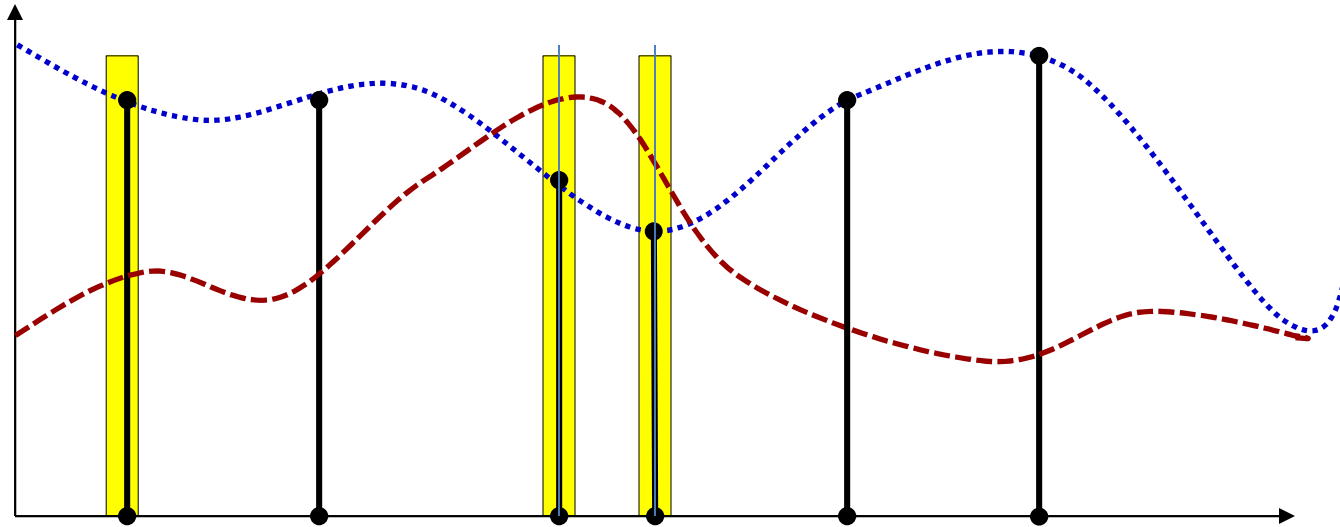


- A simpler problem: K-means
- Note: SGD converges faster
- But also has large variation between runs

SGD vs batch

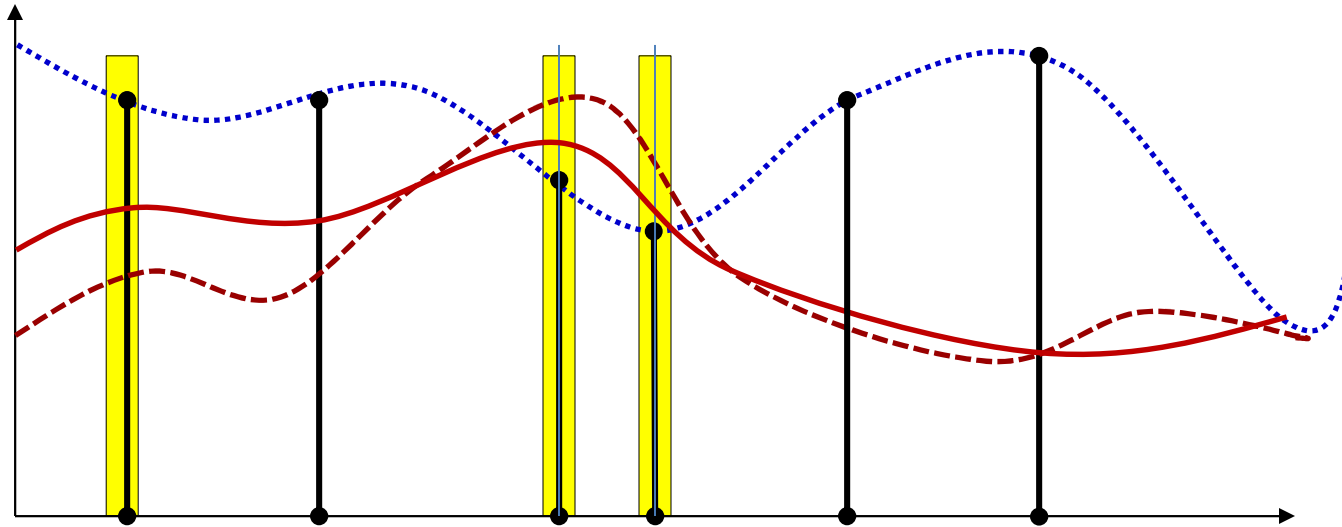
- SGD uses the gradient from only one sample at a time, and is consequently high variance
- But also provides significantly quicker updates than batch
- Is there a good medium?

Alternative: Mini-batch update



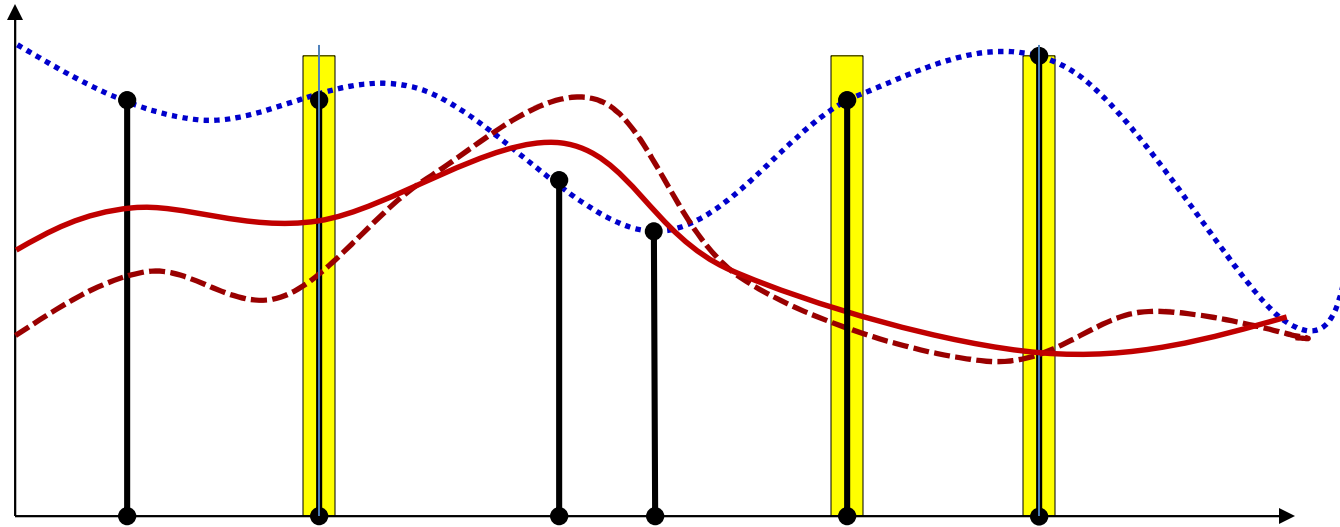
- Alternative: adjust the function at a small, randomly chosen subset of points
 - Keep adjustments small
 - If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data

Alternative: Mini-batch update



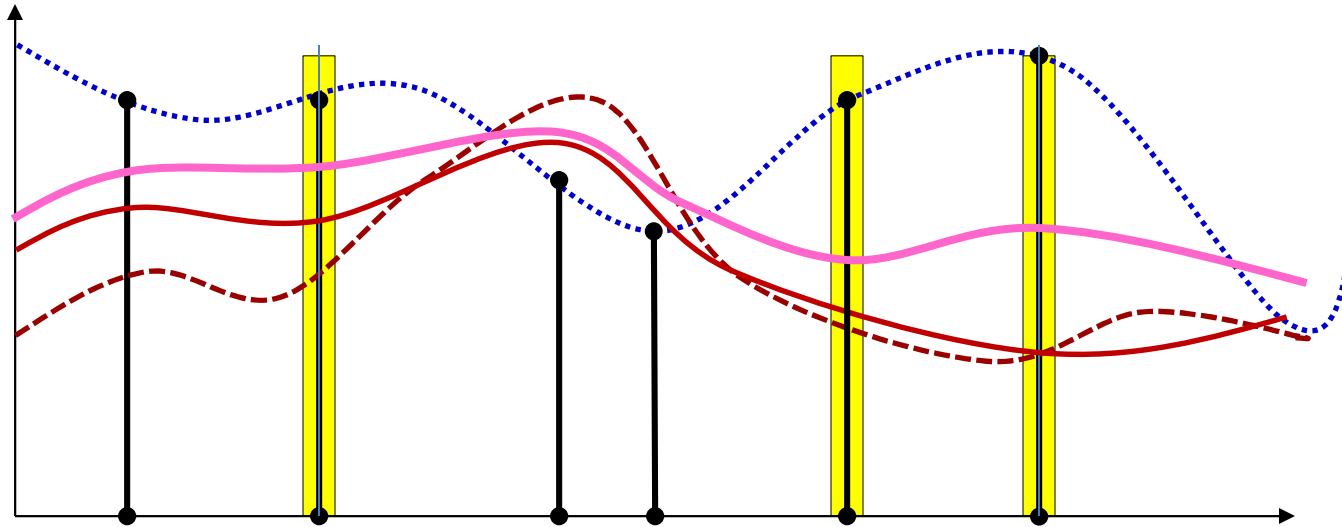
- Alternative: adjust the function at a small, randomly chosen subset of points
 - Keep adjustments small
 - If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data

Alternative: Mini-batch update



- Alternative: adjust the function at a small, randomly chosen subset of points
 - Keep adjustments small
 - If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data

Alternative: Mini-batch update



- Alternative: adjust the function at a small, randomly chosen subset of points
 - Keep adjustments small
 - If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data

Incremental Update: Mini-batch update

- Given $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights $W_1, W_2, \dots, W_K; j = 0$
- Do:
 - Randomly permute $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
 - For $t = 1:b:T$
 - $j = j + 1$
 - For every layer k :
 - $\Delta W_k = 0$
 - For $t' = t : t+b-1$
 - For every layer k :
 - » Compute $\nabla_{W_k} \text{Div}(Y_{t'}, d_{t'})$
 - » $\Delta W_k = \Delta W_k + \frac{1}{b} \nabla_{W_k} \text{Div}(Y_{t'}, d_{t'})^T$
 - Update
 - For every layer k :
$$W_k = W_k - \eta_j \Delta W_k$$
- Until *Err* has converged

Incremental Update: Mini-batch update

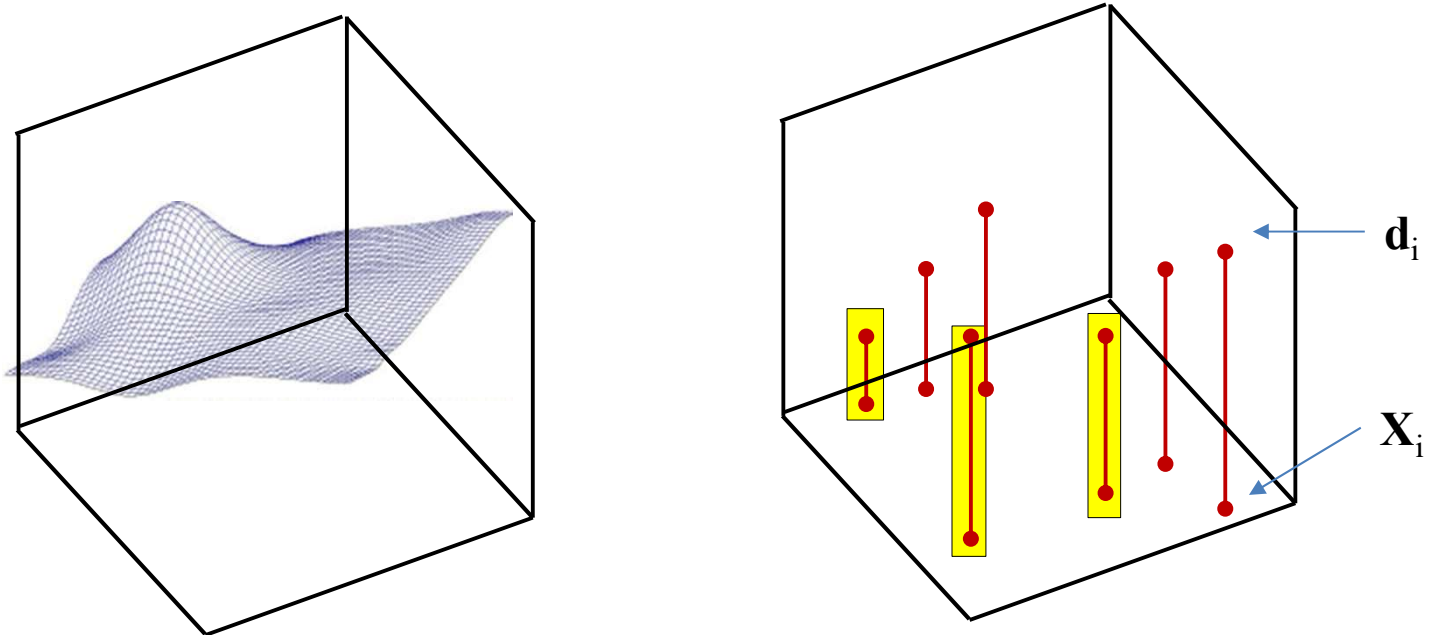
- Given $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights $W_1, W_2, \dots, W_K; j = 0$
- Do:
 - Randomly permute $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
 - For $t = 1:b:T$
 - $j = j + 1$
 - For every layer k :
 - $\Delta W_k = 0$
 - For $t' = t : t+b-1$
 - For every layer k :
 - » Compute $\nabla_{W_k} \text{Div}(Y_{t'}, d_{t'})$
 - » $\Delta W_k = \Delta W_k + \frac{1}{b} \nabla_{W_k} \text{Div}(Y_{t'}, d_{t'})^T$
 - Update
 - For every layer k :

$$W_k = W_k - \eta_j \Delta W_k$$
- Until *Err* has converged

Mini-batch size

Shrinking step size

Mini Batches



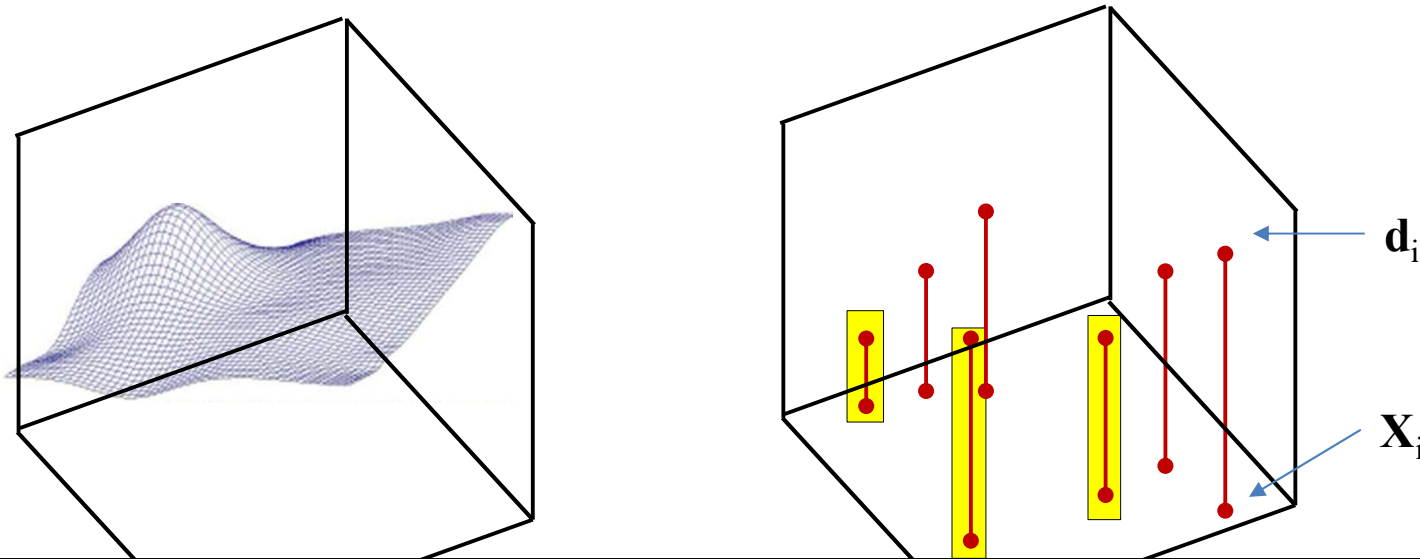
- Mini-batch updates compute and minimize a *batch loss*

$$\text{MiniBatchLoss}(W) = \frac{1}{b} \sum_{i=1}^b \text{div}(f(X_i; W), d_i)$$

- The *expected value* of the *batch loss* is also the *expected divergence*

$$E[\text{MiniBatchLoss}(W)] = E[\text{div}(f(X; W), g(X))]$$

Mini Batches



The minibatch loss is also an unbiased estimate of the expected loss

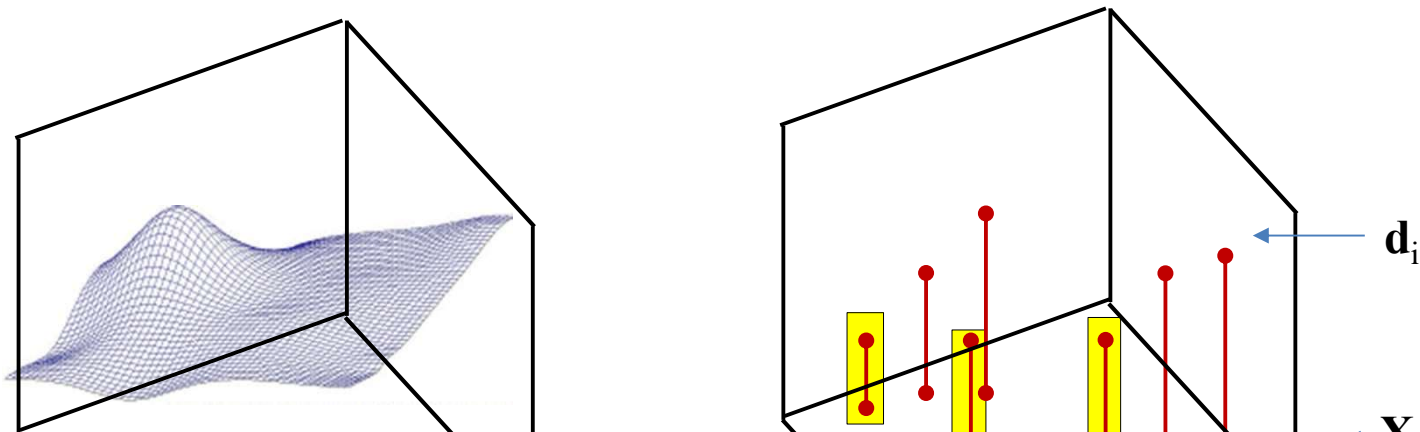
- Mini-batch updates compute and minimize a *batch loss*

$$\text{MiniBatchLoss}(W) = \frac{1}{b} \sum_{i=1}^b \text{div}(f(X_i; W), d_i)$$

- The *expected value* of the *batch loss* is also the *expected divergence*

$$E[\text{MiniBatchLoss}(W)] = E[\text{div}(f(X; W), g(X))]$$

Mini Batches



The variance of the minibatch loss: $\text{var}(\text{BatchLoss}) = 1/b \text{ var}(\text{div})$
This will be much smaller than the variance of the sample error in SGD

The minibatch loss is also an unbiased estimate of the expected error

- Mini-batch updates compute and minimize a *batch loss*

$$\text{MiniBatchLoss}(W) = \frac{1}{b} \sum_{i=1}^b \text{div}(f(X_i; W), d_i)$$

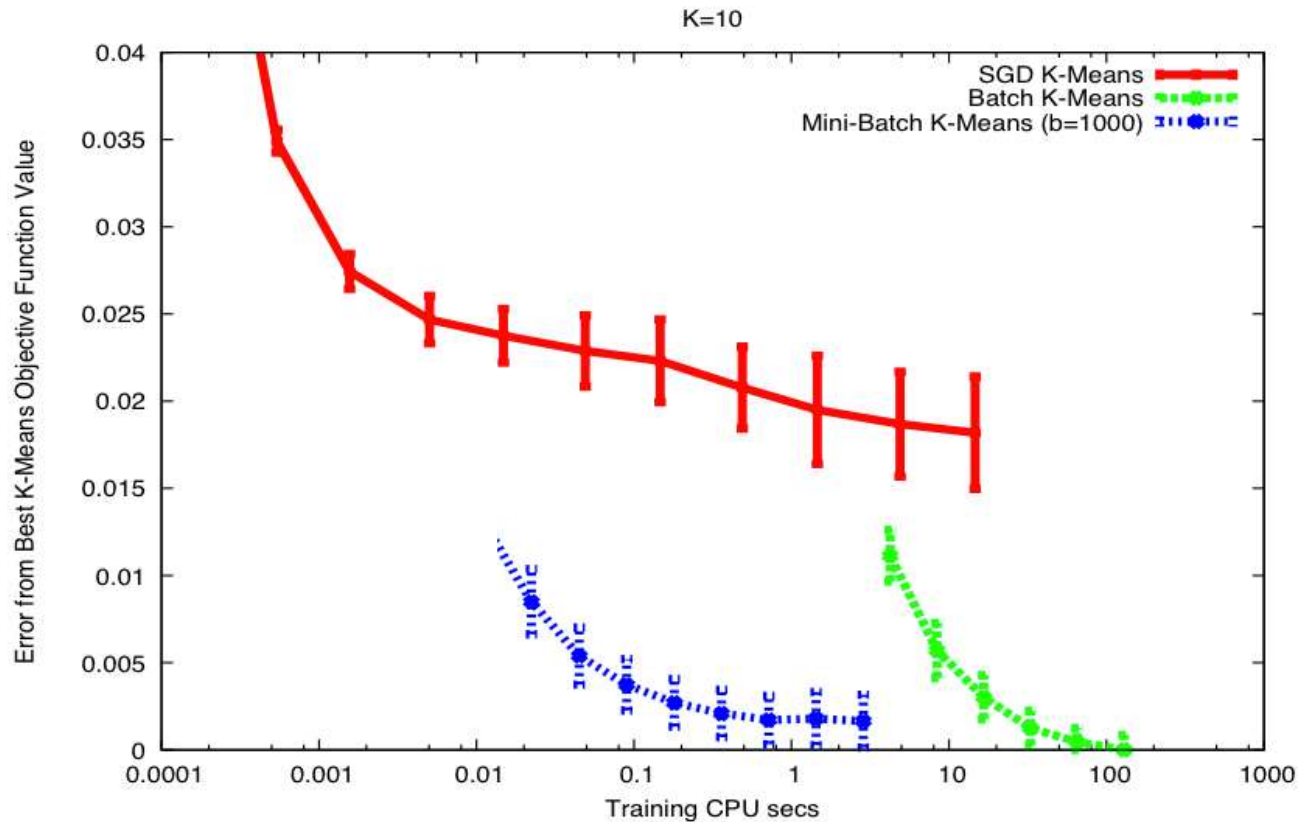
- The *expected value* of the *batch loss* is also the *expected divergence*

$$E[\text{MiniBatchLoss}(W)] = E[\text{div}(f(X; W), g(X))]$$

Minibatch convergence

- For convex functions, convergence rate for SGD is $\mathcal{O}\left(\frac{1}{\sqrt{k}}\right)$.
- For *mini-batch* updates with batches of size b , the convergence rate is $\mathcal{O}\left(\frac{1}{\sqrt{bk}} + \frac{1}{k}\right)$
 - Apparently an improvement of \sqrt{b} over SGD
 - But since the batch size is b , we perform b times as many computations per iteration as SGD
 - We actually get a *degradation* of \sqrt{b}
- However, in practice
 - The objectives are generally not convex; mini-batches are more effective with the right learning rates
 - We also get additional benefits of vector processing

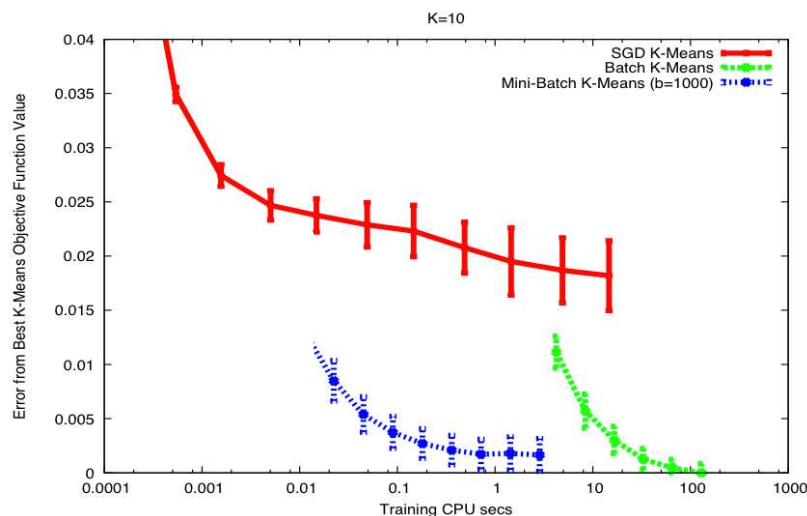
SGD example



- Mini-batch performs comparably to batch training on this simple problem
 - But converges orders of magnitude faster

Measuring Loss

- Convergence is generally defined in terms of the *overall training loss*
 - Not sample or batch loss
- Infeasible to actually measure the overall training loss after each iteration
- More typically, we estimate is as
 - Divergence or classification error on a held-out set
 - Average sample/batch loss over the past N samples/batches



Training and minibatches

- In practice, training is usually performed using mini-batches
 - The mini-batch size is a hyper parameter to be optimized
- Convergence depends on learning rate
 - Simple technique: fix learning rate until the error plateaus, then reduce learning rate by a fixed factor (e.g. 10)
 - ***Advanced methods***: Adaptive updates, where the learning rate is itself determined as part of the estimation

Story so far

- SGD: Presenting training instances one-at-a-time can be more effective than full-batch training
 - Provided they are provided in random order
- For SGD to converge, the learning rate must shrink sufficiently rapidly with iterations
 - Otherwise the learning will continuously “chase” the latest sample
- SGD estimates have higher variance than batch estimates
- Minibatch updates operate on *batches* of instances at a time
 - Estimates have lower variance than SGD
 - Convergence rate is theoretically worse than SGD
 - But we compensate by being able to perform batch processing

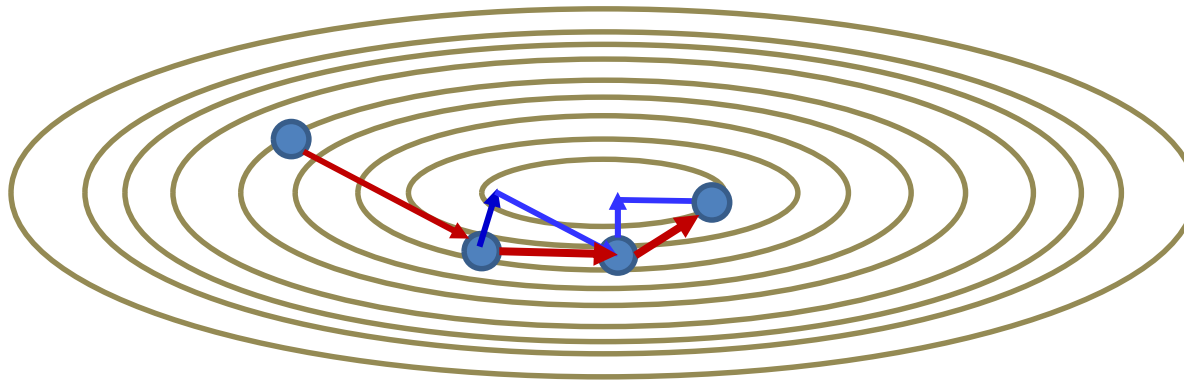
Training and minibatches

- Convergence depends on learning rate
 - Simple technique: fix learning rate until the error plateaus, then reduce learning rate by a fixed factor (e.g. 10)
 - ***Advanced methods:*** Adaptive updates, where the learning rate is itself determined as part of the estimation

Moving on: Topics for the day

- Incremental updates
- Revisiting “trend” algorithms
- Generalization
- Tricks of the trade
 - Divergences..
 - Activations
 - Normalizations

Recall: Momentum

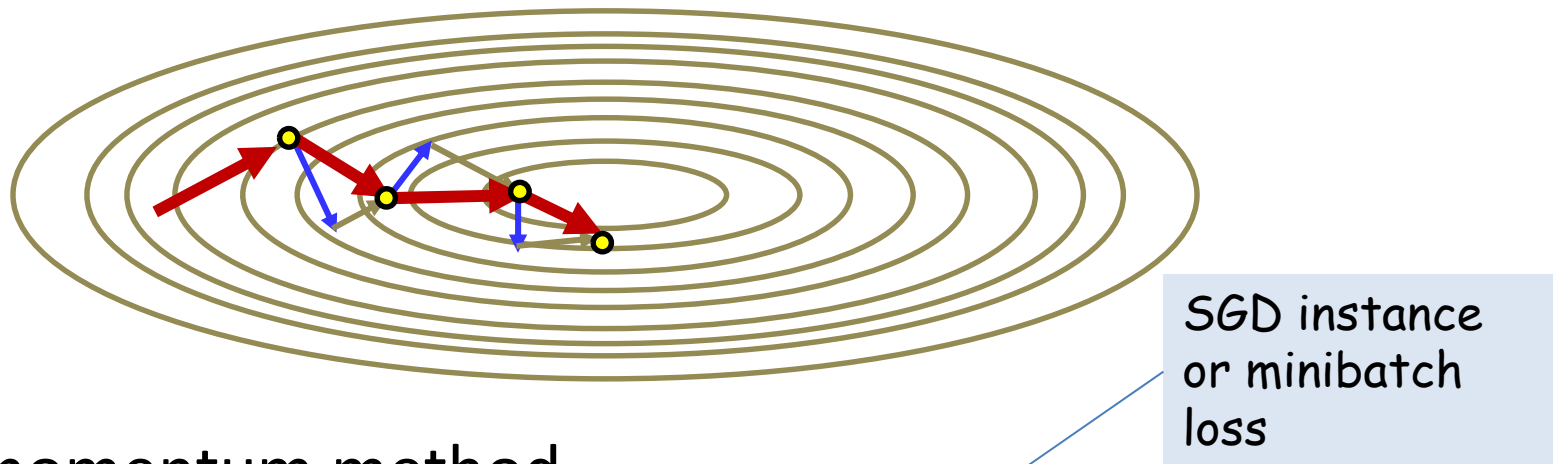


- The momentum method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Loss}(W^{(k-1)})$$

- Updates using a running average of the gradient

Momentum and incremental updates



- The momentum method

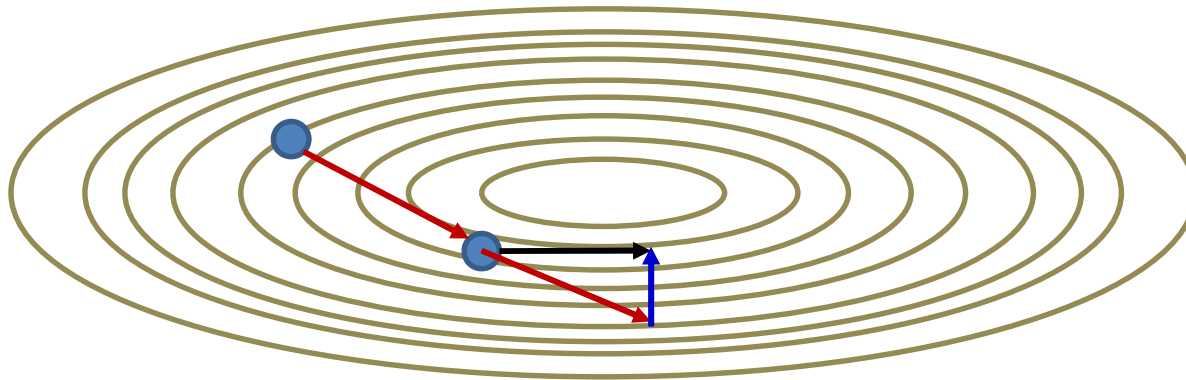
$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Loss}(W^{(k-1)})^T$$

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

Momentum: Mini-batch update

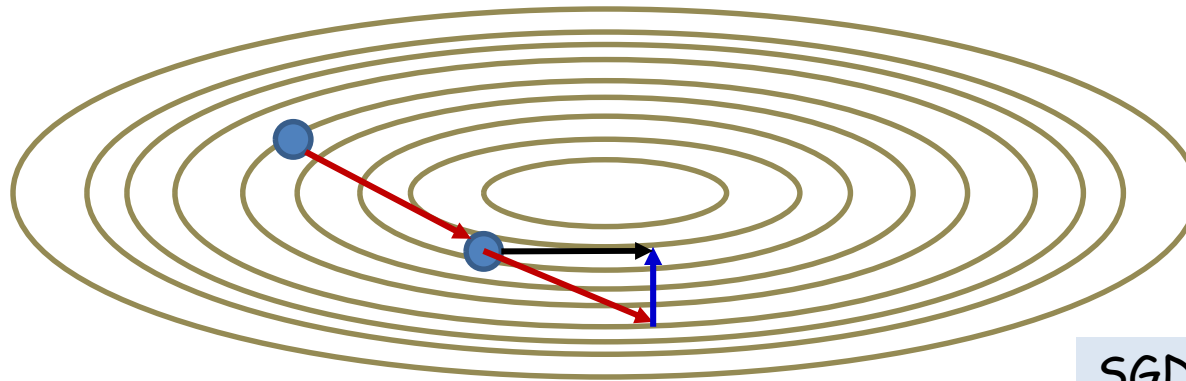
- Given $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights W_1, W_2, \dots, W_K ; $j = 0, \Delta W_k = 0$
- Do:
 - Randomly permute $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
 - For $t = 1:b:T$
 - $j = j + 1$
 - For every layer k :
 - $\nabla_{W_k} Loss = 0$
 - For $t' = t : t+b-1$
 - For every layer k :
 - » Compute $\nabla_{W_k} Div(Y_{t'}, d_{t'})$
 - » $\nabla_{W_k} Loss += \frac{1}{b} \nabla_{W_k} Div(Y_{t'}, d_{t'})$
 - Update
 - For every layer k :
$$\Delta W_k = \beta \Delta W_k - \eta_j (\nabla_{W_k} Loss)^T$$
$$W_k = W_k + \Delta W_k$$
- Until $Loss$ has converged

Nestorov's Accelerated Gradient



- At any iteration, to compute the current step:
 - First extend the previous step
 - Then compute the gradient at the resultant position
 - Add the two to obtain the final step
- This also applies directly to incremental update methods
 - The accelerated gradient smooths out the variance in the gradients

Nestorov's Accelerated Gradient



- Nestorov's method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Loss}(W^{(k-1)} + \beta \Delta W^{(k-1)})^T$$

$$W^{(k)} = W^{(k-1)} + \Delta W^{(k)}$$

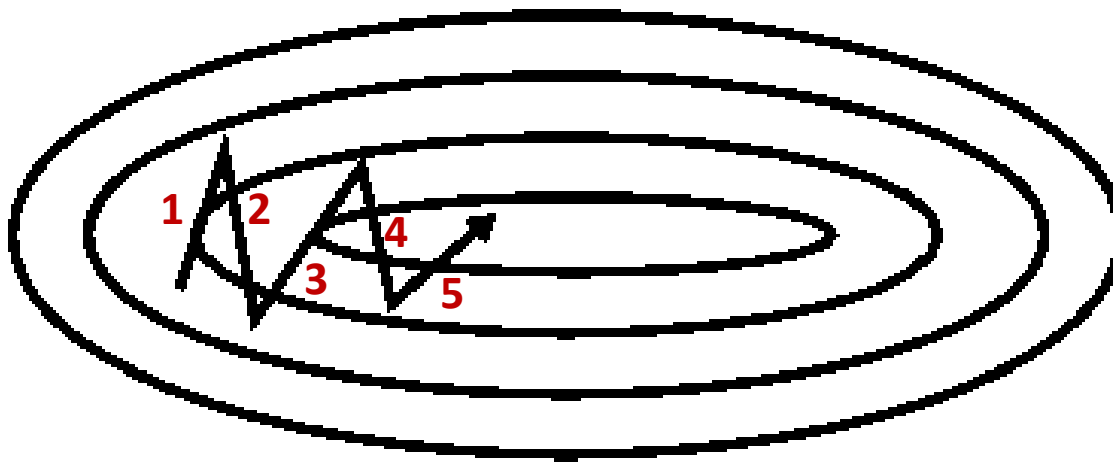
Nestorov: Mini-batch update

- Given $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights W_1, W_2, \dots, W_K ; $j = 0, \Delta W_k = 0$
- Do:
 - Randomly permute $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
 - For $t = 1:b:T$
 - $j = j + 1$
 - For every layer k :
 - $W_k = W_k + \beta \Delta W_k$
 - $\nabla_{W_k} Loss = 0$
 - For $t' = t : t+b-1$
 - For every layer k :
 - » Compute $\nabla_{W_k} Div(Y_{t'}, d_{t'})$
 - » $\nabla_{W_k} Loss += \frac{1}{b} \nabla_{W_k} Div(Y_{t'}, d_{t'})$
 - Update
 - For every layer k :
 - $W_k = W_k - \eta_j \nabla_{W_k} Loss^T$
 - $\Delta W_k = \beta \Delta W_k - \eta_j \nabla_{W_k} Loss^T$
- Until $Loss$ has converged

Still higher-order 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**
 - ...
- All roughly equivalent in performance

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 Nestorov are used
- Improvement: Dampen step size in directions with high motion
 - *Second order term*

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...

RMS Prop

- Notation:
 - Updates are *by parameter*
 - Derivative of loss w.r.t any individual parameter w is shown as $\partial_w D$
 - Batch or minibatch loss, or individual divergence for batch/minibatch/SGD
 - The *squared* derivative is $\partial_w^2 D = (\partial_w D)^2$
 - 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 $E[\partial_w^2 D]$
- Modified update rule: We want to
 - scale down updates with large mean squared derivatives
 - scale up updates with small mean squared derivatives

RMS Prop

- This is a variant on the *basic* mini-batch SGD algorithm
- **Procedure:**
 - Maintain a running estimate of the mean squared value of derivatives for each parameter
 - Scale update of the parameter by the *inverse* of the *root mean squared* derivative

$$E[\partial_w^2 D]_k = \gamma E[\partial_w^2 D]_{k-1} + (1 - \gamma)(\partial_w^2 D)_k$$

$$w_{k+1} = w_k - \frac{\eta}{\sqrt{E[\partial_w^2 D]_k + \epsilon}} \partial_w D$$

RMS Prop

- This is a variant on the *basic* mini-batch SGD algorithm
- **Procedure:**
 - Maintain a running estimate of the mean squared value of derivatives for each parameter
 - Scale update of the parameter by the *inverse* of the *root mean squared* derivative

$$E[\partial_w^2 D]_k = \gamma E[\partial_w^2 D]_{k-1} + (1 - \gamma)(\partial_w^2 D)_k$$

$$w_{k+1} = w_k - \frac{\eta}{\sqrt{E[\partial_w^2 D]_k + \epsilon}} \partial_w D$$

Note similarity to RPROP

The magnitude of the derivative is being normalized out

RMS Prop (updates are for each weight of each layer)

- Do:
 - Randomly shuffle inputs to change their order
 - Initialize: $k = 1$; for all weights w in all layers, $E[\partial_w^2 D]_k = 0$
 - For all $t = 1:B:T$ (incrementing in blocks of B inputs)
 - For all weights in all layers initialize $(\partial_w D)_k = 0$
 - For $b = 0:B-1$
 - Compute
 - » Output $Y(X_{t+b})$
 - » Compute gradient $\frac{dDiv(Y(X_{t+b}), d_{t+b})}{dw}$
 - » Compute $(\partial_w D)_k += \frac{1}{B} \frac{dDiv(Y(X_{t+b}), d_{t+b})}{dw}$
 - update: for all $w \in \{w_{\{ij\}}^k \forall i, j, k\}$

$$E[\partial_w^2 D]_k = \gamma E[\partial_w^2 D]_{k-1} + (1 - \gamma)(\partial_w^2 D)_k$$
$$w_{k+1} = w_k - \frac{\eta}{\sqrt{E[\partial_w^2 D]_k + \epsilon}} \partial_w D$$
 - $k = k + 1$
- Until loss has converged

Typical values:

$$\gamma = 0.9$$

$$\eta = 0.001$$

ADAM: RMSprop with 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
- **Procedure:**
 - Maintain a running estimate of the mean derivative for each parameter
 - Maintain a running estimate of the mean squared value of derivatives for each parameter
 - Scale update of the parameter by the *inverse* of the *root mean squared* derivative

$$m_k = \delta m_{k-1} + (1 - \delta)(\partial_w D)_k$$

$$v_k = \gamma v_{k-1} + (1 - \gamma)(\partial_w^2 D)_k$$

$$\hat{m}_k = \frac{m_k}{1 - \delta^k}, \quad \hat{v}_k = \frac{v_k}{1 - \gamma^k}$$

$$w_{k+1} = w_k - \frac{\eta}{\sqrt{\hat{v}_k + \epsilon}} \hat{m}_k$$

ADAM: RMSprop with momentum

- RMS prop only considers a second-moment normalized version of the current gradient
- ADAM utilizes a smoothed version of the *momentum-augmented* gradient
- **Procedure:**
 - Maintain a running estimate of the mean derivative for each parameter
 - Maintain a running estimate of the mean squared value for each parameter
 - Scale update of the parameter by the *inverse* of the derivative

$$m_k = \delta m_{k-1} + (1 - \delta)(\partial_w D)_k$$

$$v_k = \gamma v_{k-1} + (1 - \gamma)(\partial_w^2 D)_k$$

$$\hat{m}_k = \frac{m_k}{1 - \delta^k}, \quad \hat{v}_k = \frac{v_k}{1 - \gamma^k}$$

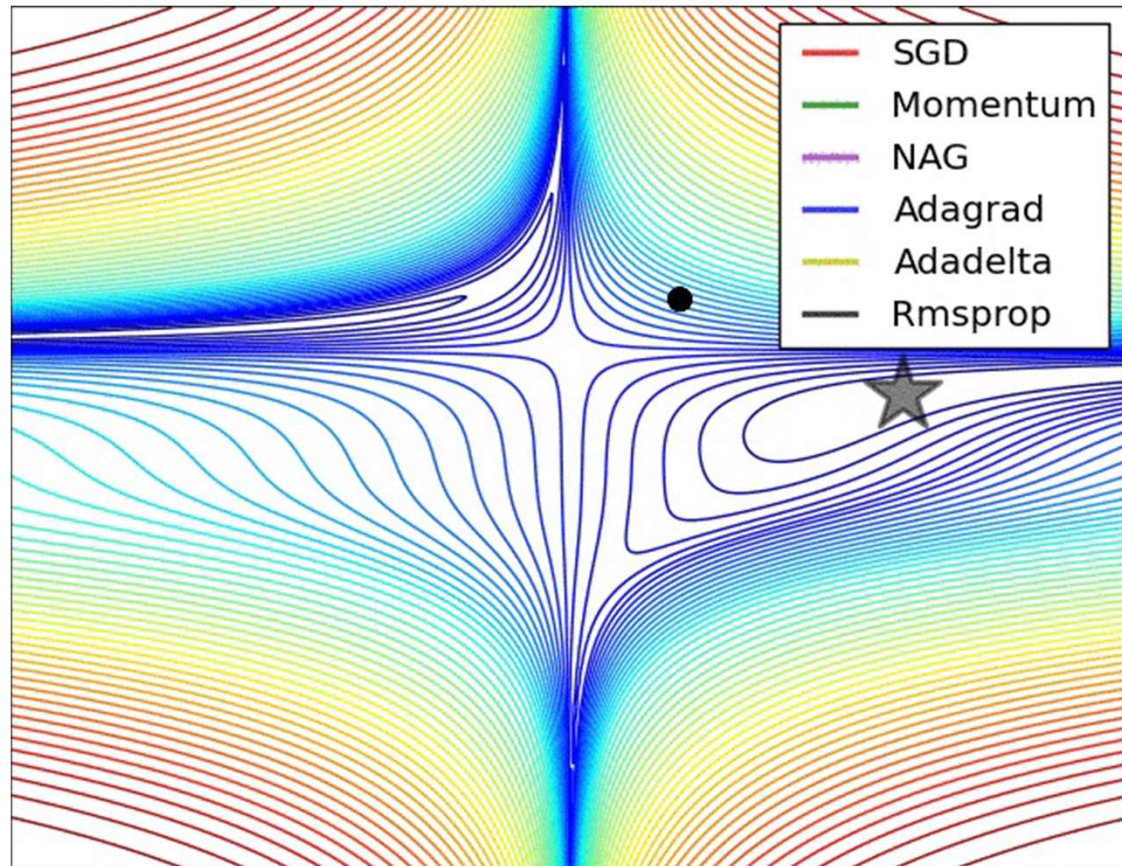
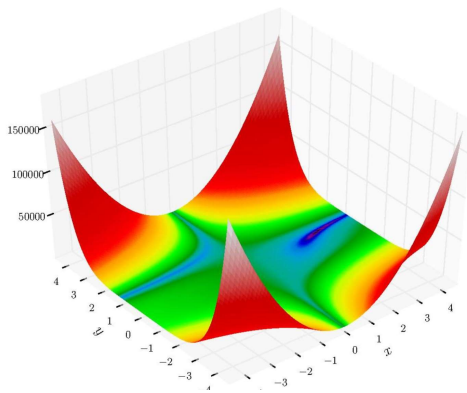
$$w_{k+1} = w_k - \frac{\eta}{\sqrt{\hat{v}_k + \epsilon}} \hat{m}_k$$

Ensures that the δ and γ terms do not dominate in early iterations

Other variants of the same theme

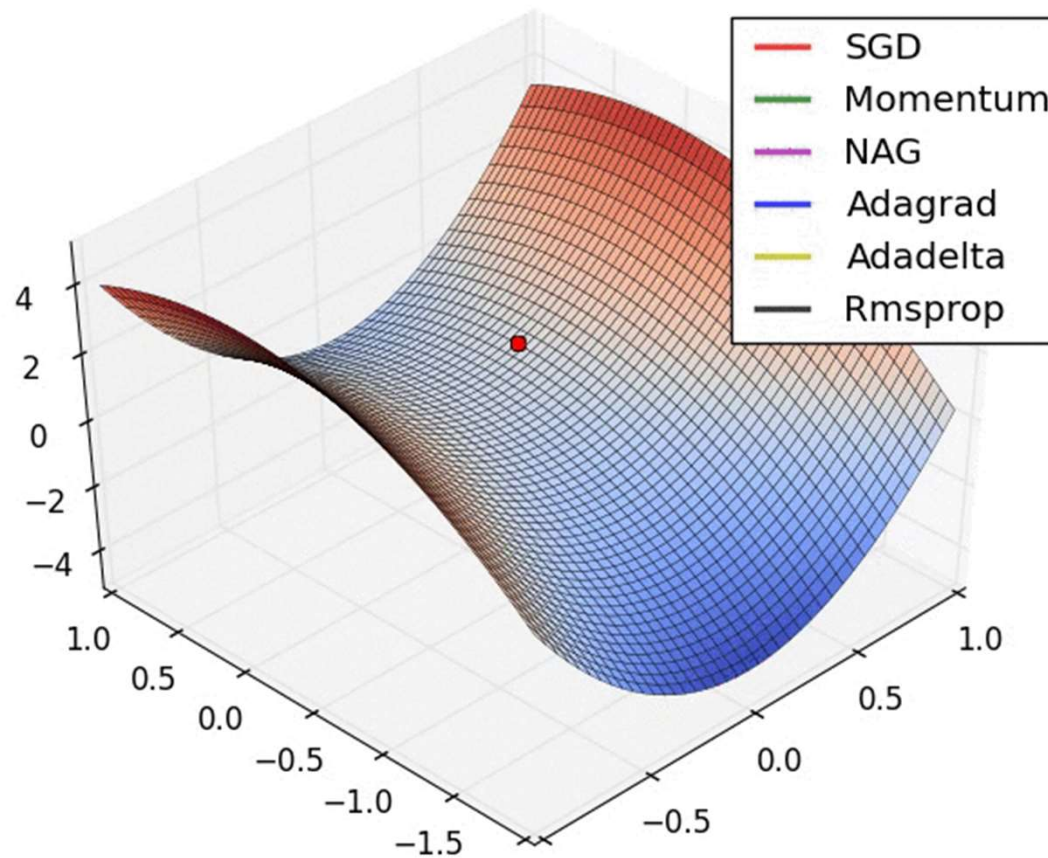
- Many:
 - Adagrad
 - AdaDelta
 - AdaMax
 - ...
- Generally no explicit learning rate to optimize
 - But come with other hyper parameters to be optimized
 - Typical params:
 - RMSProp: $\eta = 0.001, \gamma = 0.9$
 - ADAM: $\eta = 0.001, \delta = 0.9, \gamma = 0.999$

Visualizing the optimizers: Beale's Function



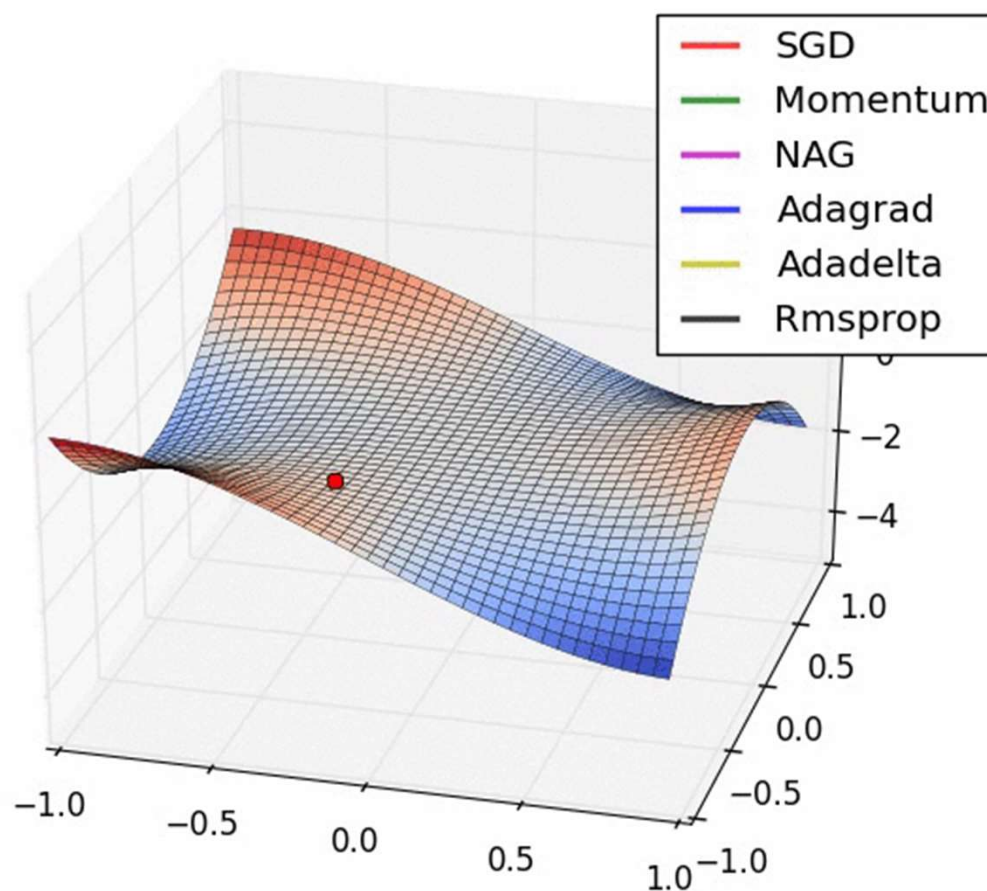
- <http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html>

Visualizing the optimizers: Long Valley



- <http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html>

Visualizing the optimizers: Saddle Point



- <http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html>

Story so far

- 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
 - RMSprop and more advanced techniques