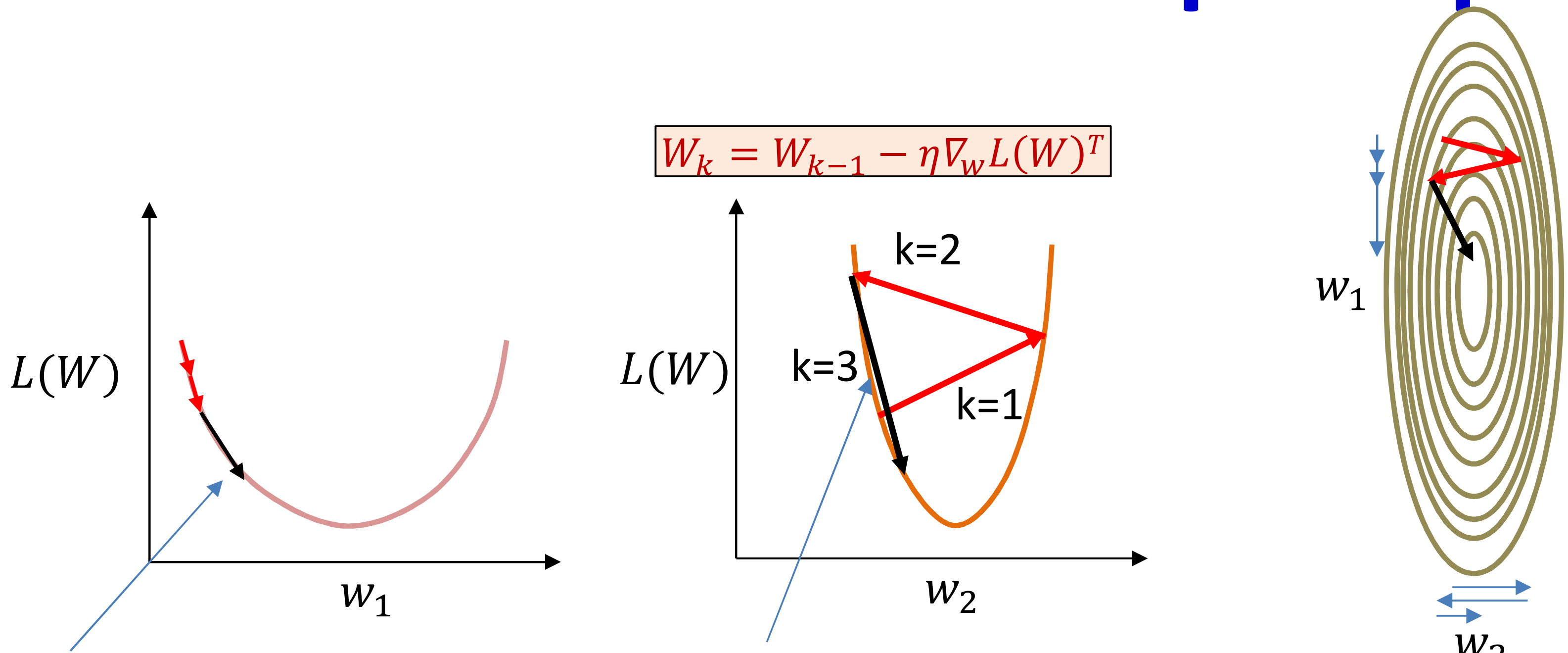
Momentum methods: principle



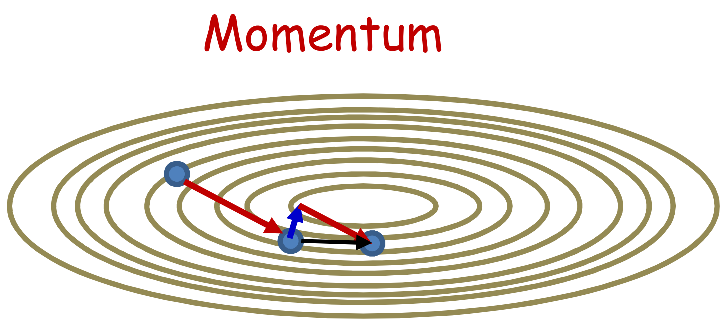
Increase stepsize because previous updates consistently moved weight right

Decrease stepsize because previous updates kept changing direction

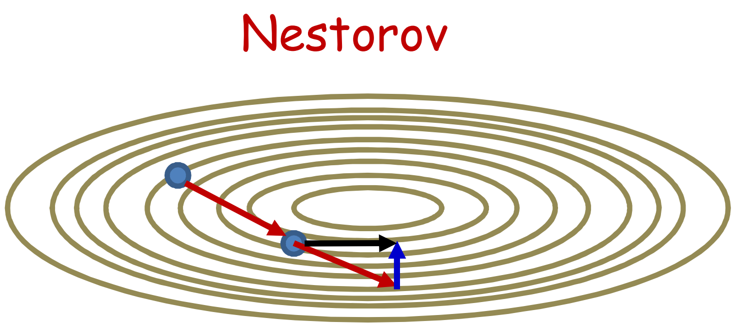
Stepsize shrinks along w2 but increases along w1

- Ideally: Have component-specific step size
  - But the resulting updates will not be against the gradient and do not guarantee descent
- Adaptive solution: Start with a common step size
  - *Shrink* step size in directions where the weight oscillates
  - *Expand* step size in directions where the weight moves consistently in one direction

Quick recap: Momentum methods



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



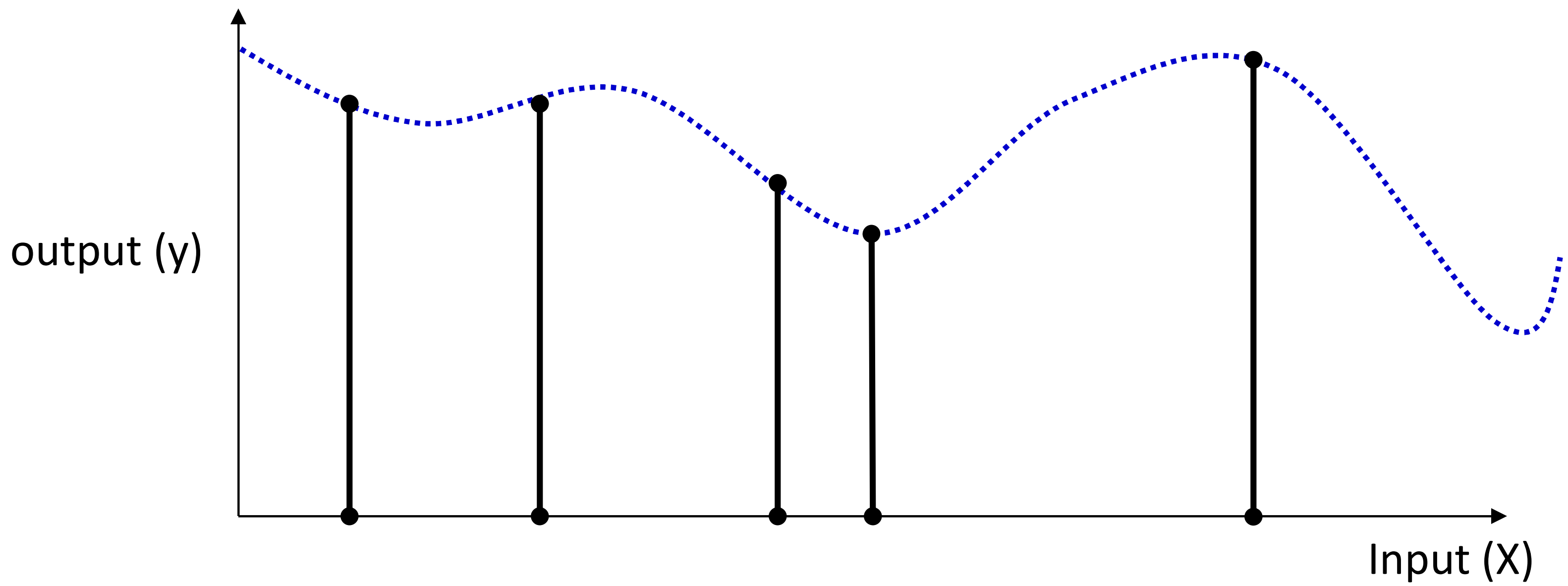
$$W_{extend}^{(k)} = W^{(k-1)} + \beta \Delta W^{(k-1)}$$

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Loss\left(W_{extend}^{(k)}\right)^T$$

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

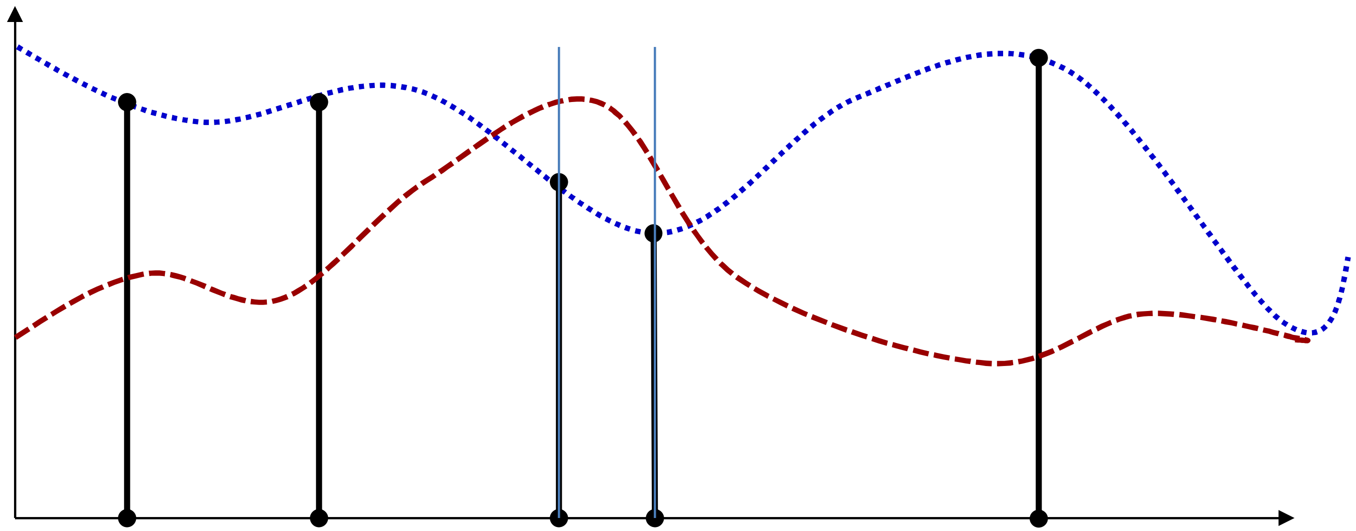
- Momentum: Retain gradient value, but *smooth out* gradients by maintaining a running average
  - Cancels out steps in directions where the weight value oscillates
  - Adaptively increases step size in directions of consistent change

The training formulation



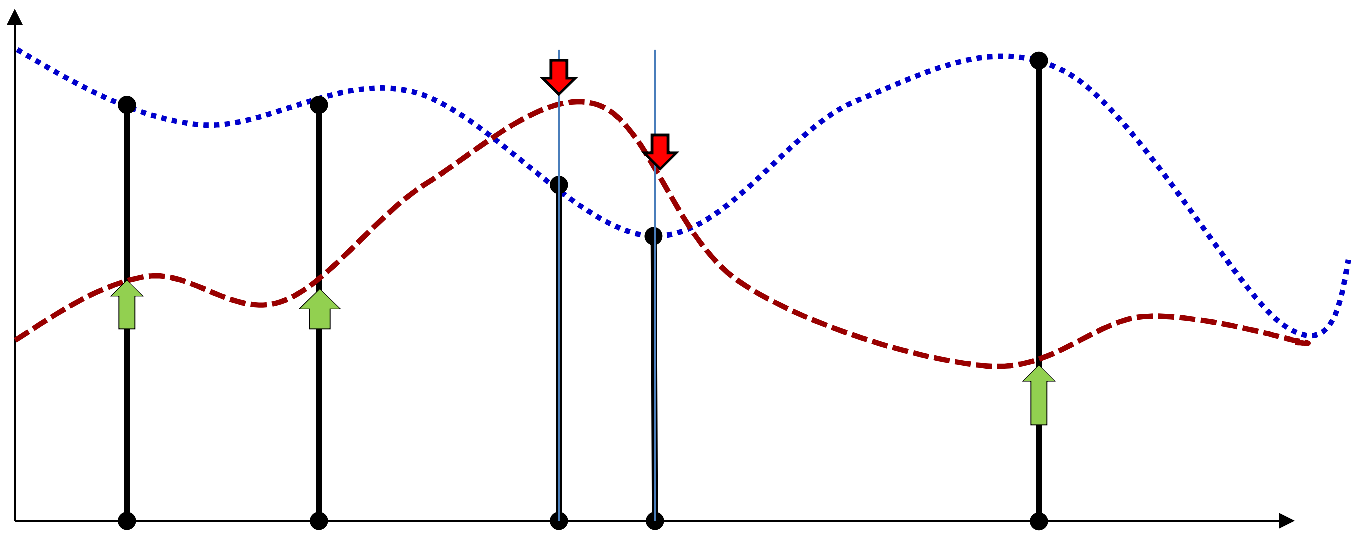
- Given input output pairs at a number of locations, estimate the entire function

Gradient descent



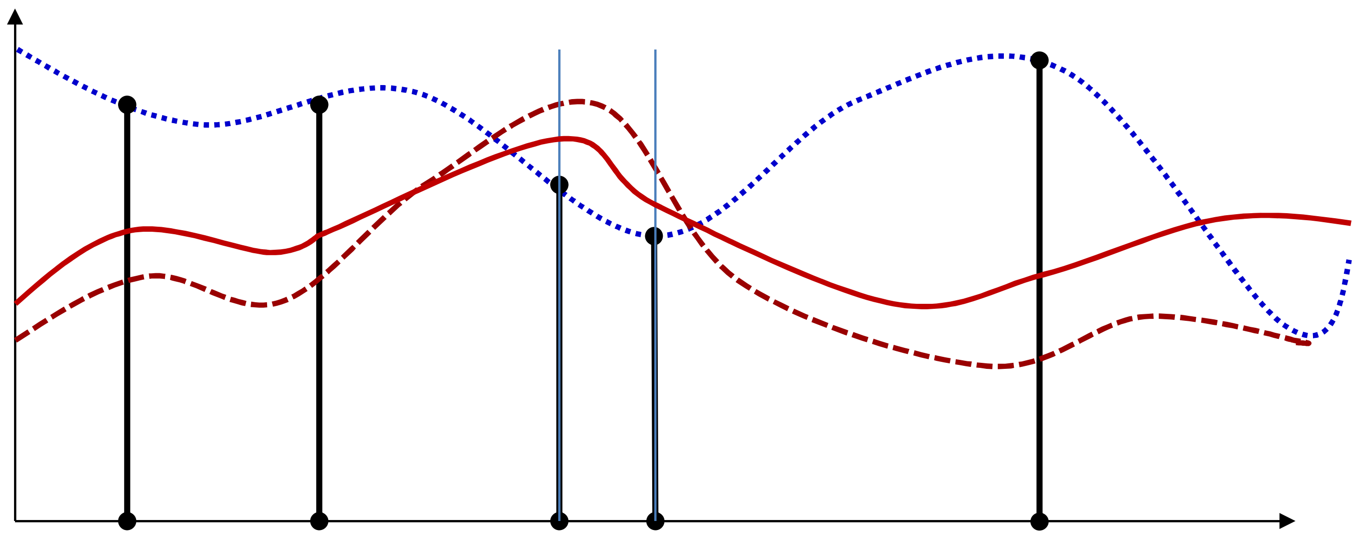
- Start with an initial function

Gradient descent



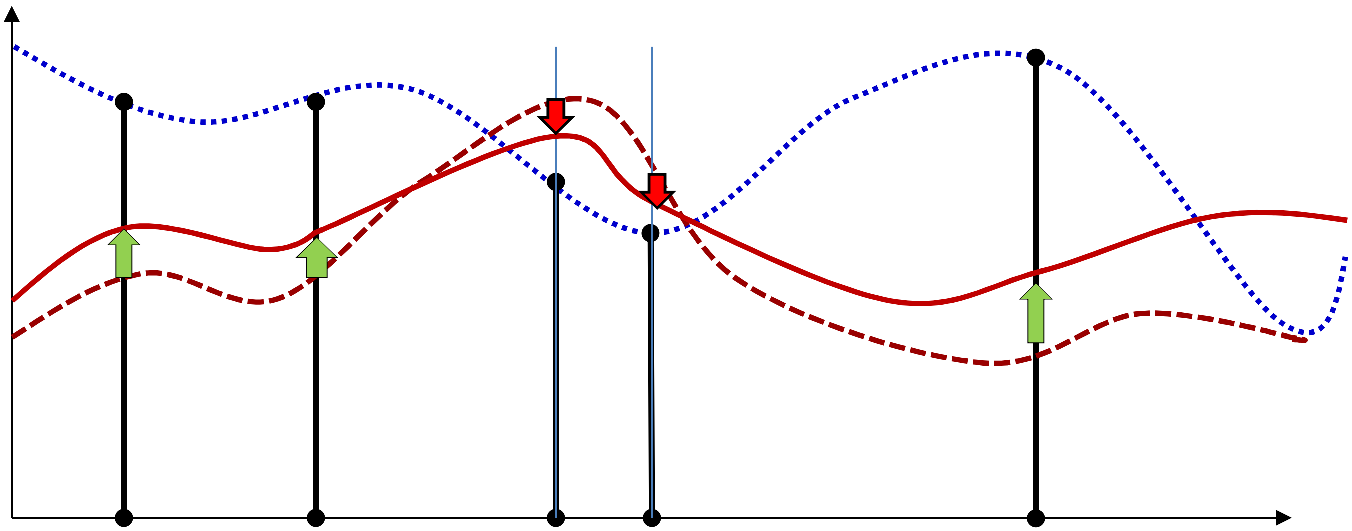
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



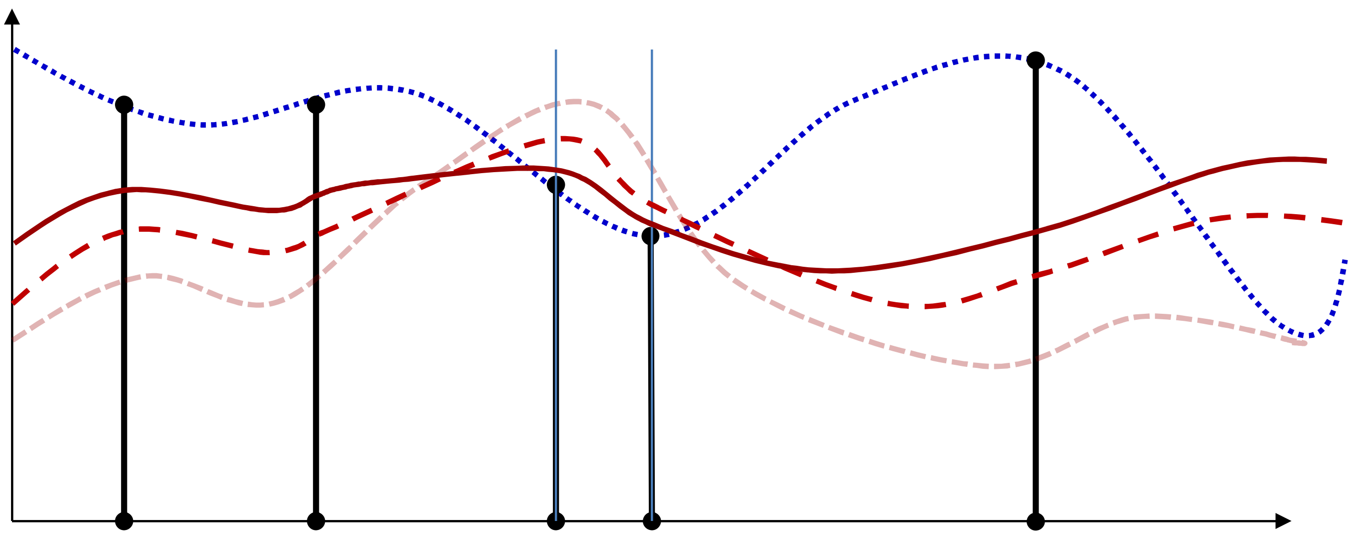
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



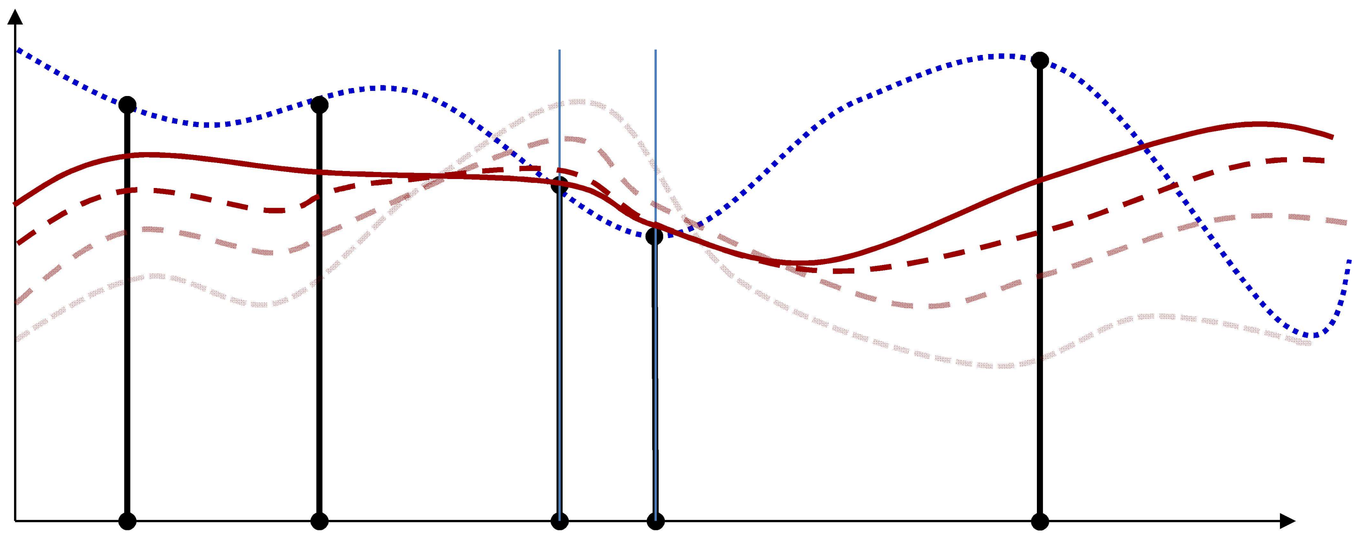
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



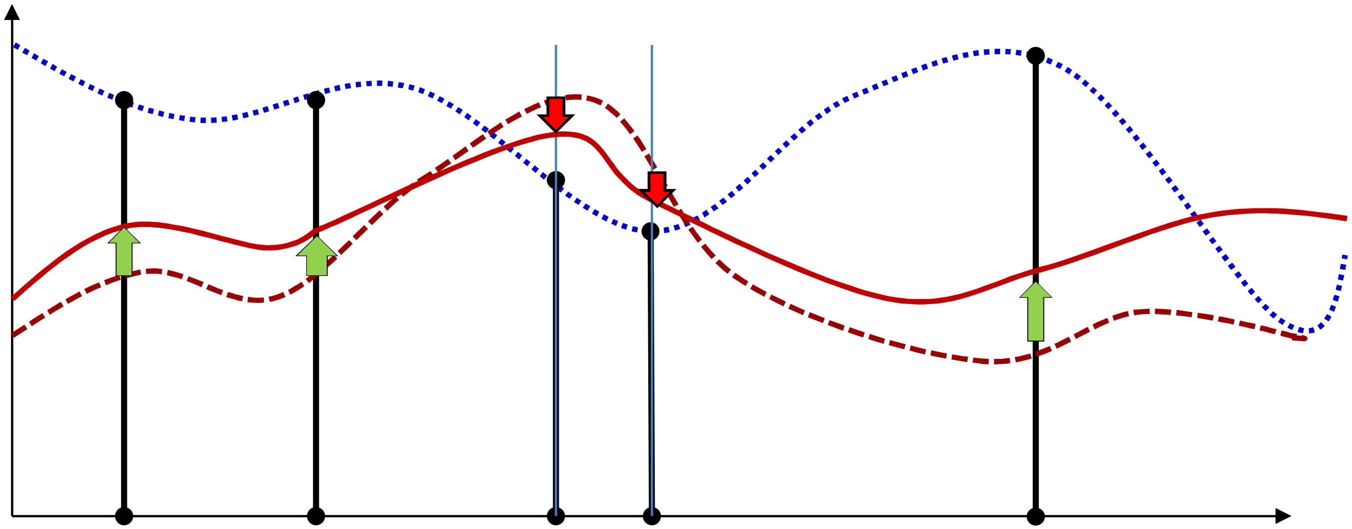
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



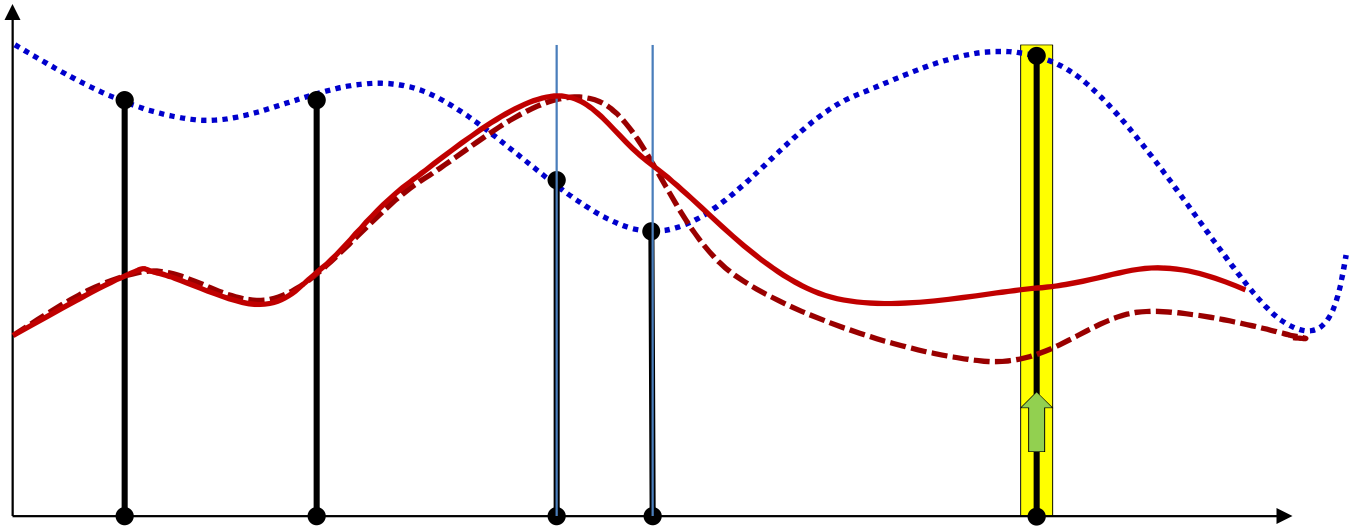
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Effect of number of samples



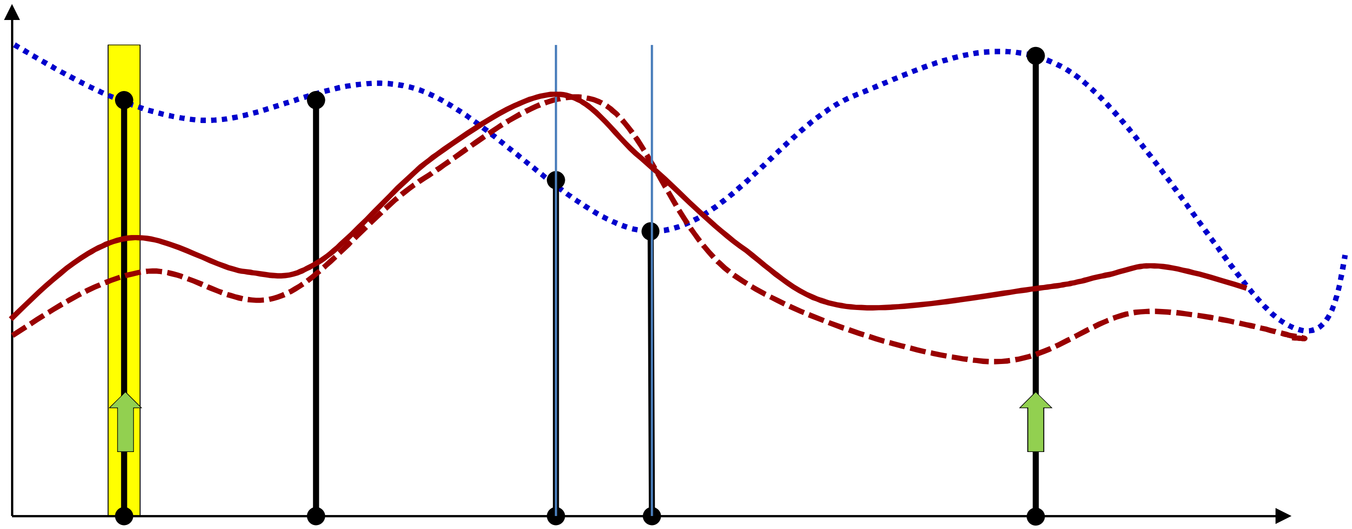
- Problem with conventional gradient descent: we try to simultaneously adjust the function at *all* training points
  - We must process *all* training points before making a single adjustment
  - **"Batch"** update

Alternative: Incremental update



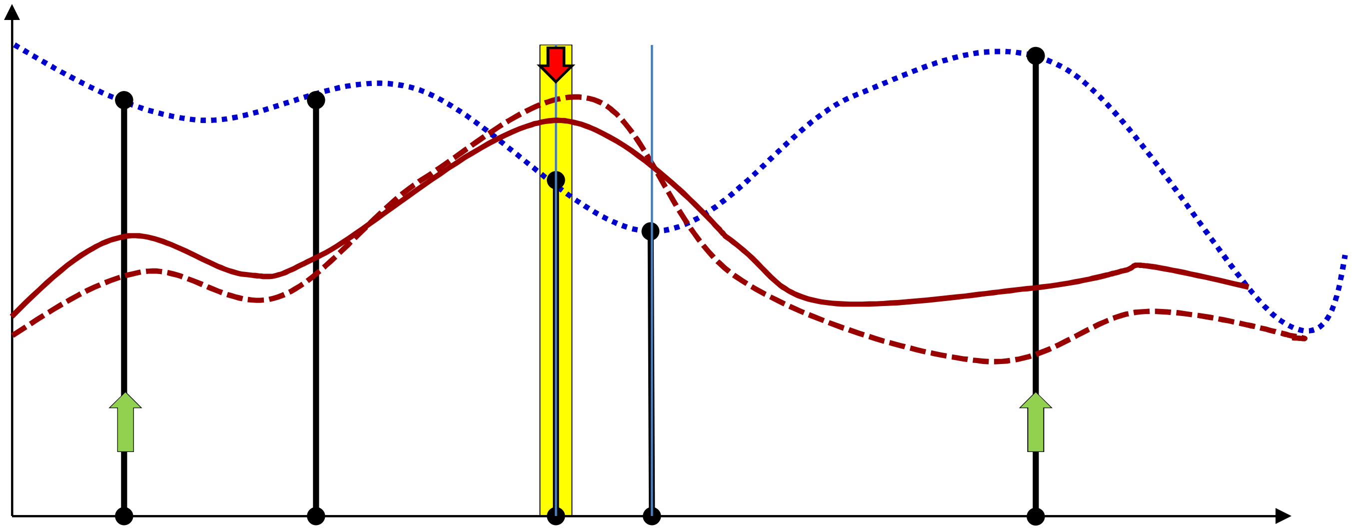
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

Alternative: Incremental update



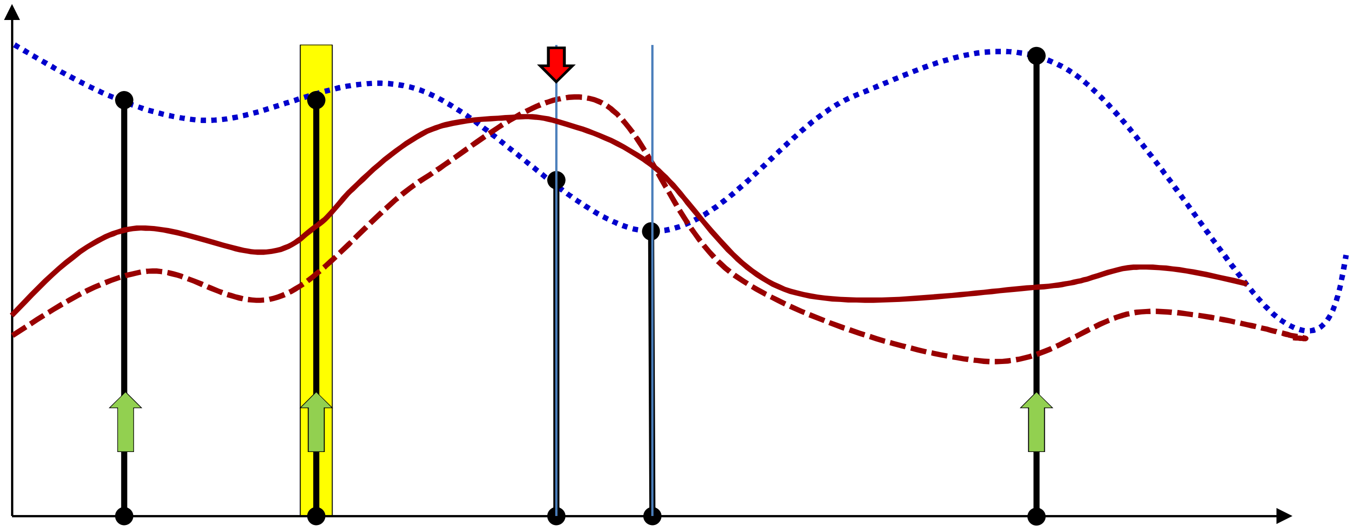
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

Alternative: Incremental update



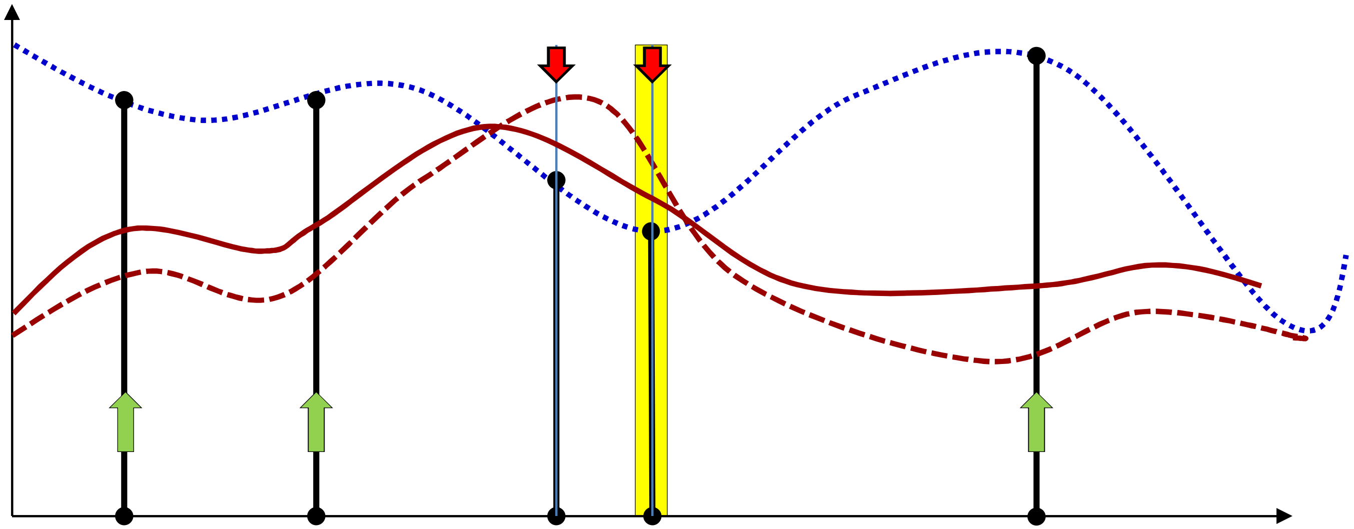
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

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: Stochastic Gradient Descent

- Given $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$
- Initialize all weights $W_1, W_2, \ldots, W_K$
- Do:
  - For all $t = 1 : T$
    - For every layer k:
      - Compute $\nabla_{W_k} \boldsymbol{Div}(\boldsymbol{Y}_t, \boldsymbol{d}_t)$
      - Update

$$W_k = W_k - \eta \nabla_{W_k} \boldsymbol{Div}(\boldsymbol{Y}_t, \boldsymbol{d}_t)^T$$

- Until $Loss$ has converged

Stochastic Gradient Descent

- 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: Stochastic Gradient Descent

- Given $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$
- Initialize all weights $W_1, W_2, \ldots, W_K$
- Do:
  - Randomly permute $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$
  - For all $t = 1:T$
    - For every layer k:
      - Compute $\nabla_{W_k} \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})$
      - Update

$$W_k = W_k - \eta \nabla_{W_k} \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})^T$$

- Until $Loss$ has converged

SGD convergence

- SGD converges "almost surely" to a global or local minimum for most functions
  - Sufficient condition: step sizes follow the following conditions

$$\sum_k \eta_k = \infty$$

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

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

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

  – Strongly convex → 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,

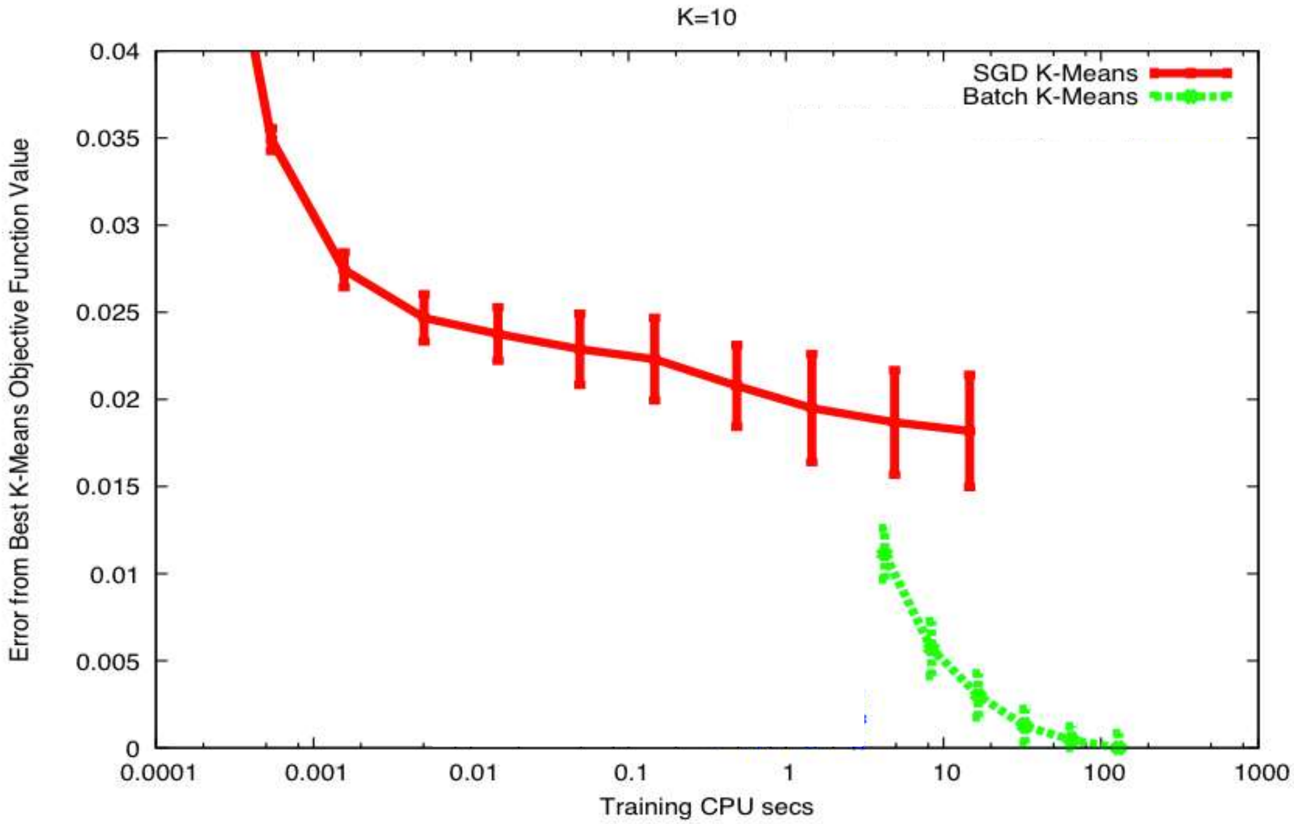
$$\left|W^{(k)} - W^*\right| < c^k \left|W^{(0)} - W^*\right|$$

  – 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 $O\left(\frac{1}{\epsilon}\right)$

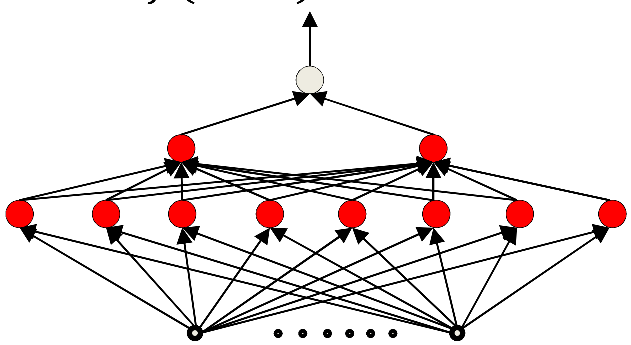
- Batch gradients converge "faster"
  – But SGD performs T updates for every batch update

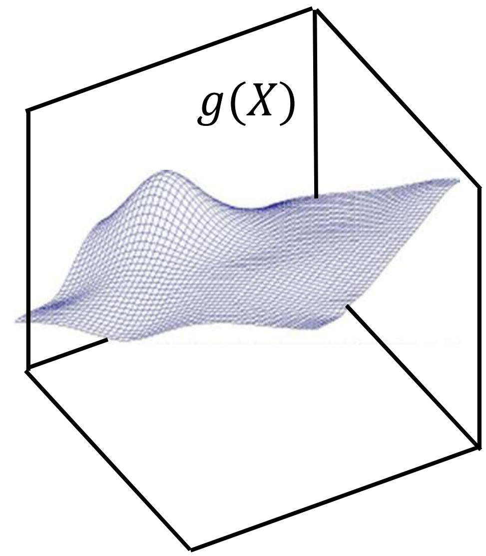
SGD example



- A simpler problem: K-means
- Note: SGD converges slower
- Also note the rather large variation between runs
  - Lets try to understand these results..

Recall: Modelling a function



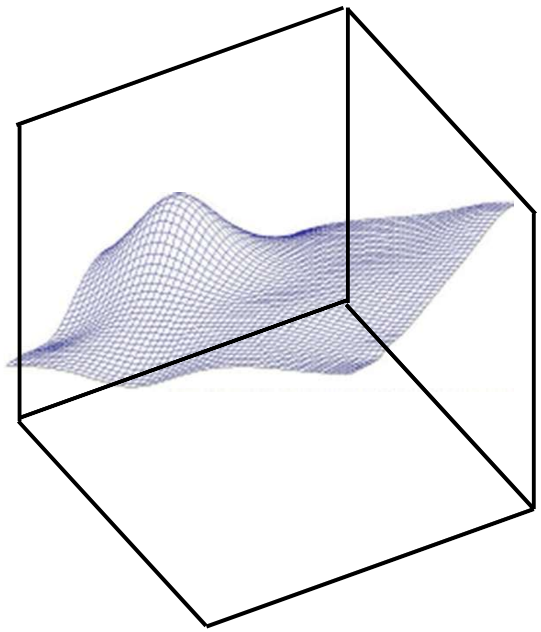


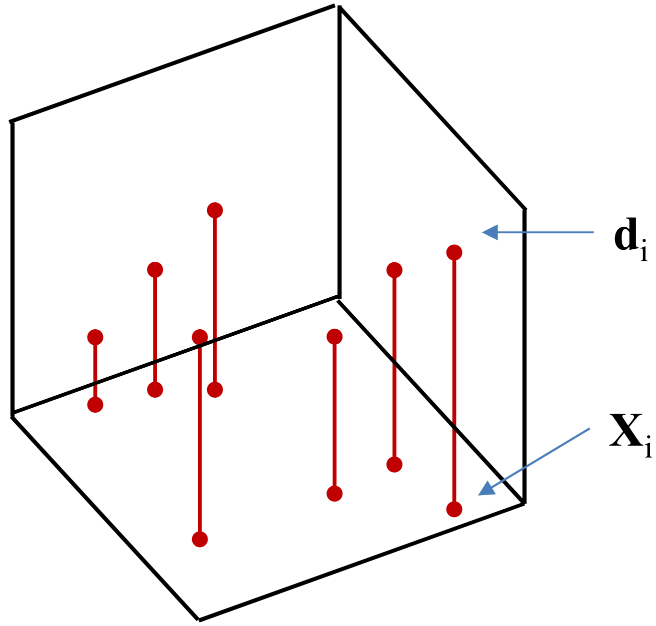
- To learn a network $f(X; \boldsymbol{W})$ to model a function $g(X)$ we minimize the *expected divergence*

$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}} \int_X div\big(f(X; W), g(X)\big)P(X)dX$$

$$= \underset{W}{\operatorname{argmin}}\, E\Big[div\big(f(X; W), g(X)\big)\Big]$$

Recall: The *Empirical* risk





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

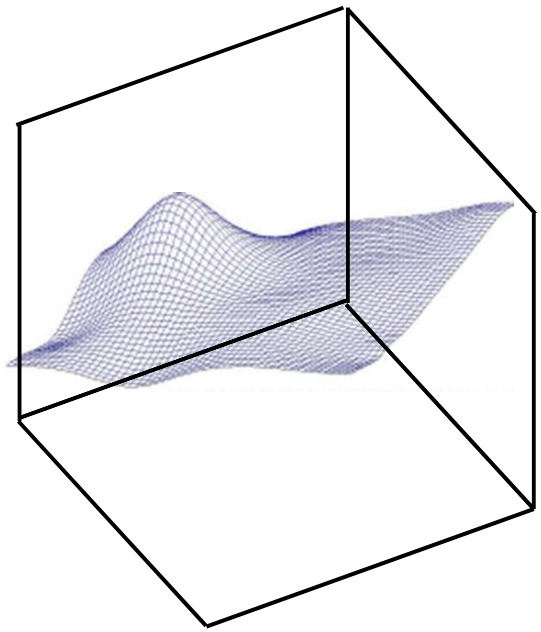
$$Loss\big(f(X;W), g(X)\big) = \frac{1}{N}\sum_{i=1}^{N} div(f(X_i;W), d_i)$$

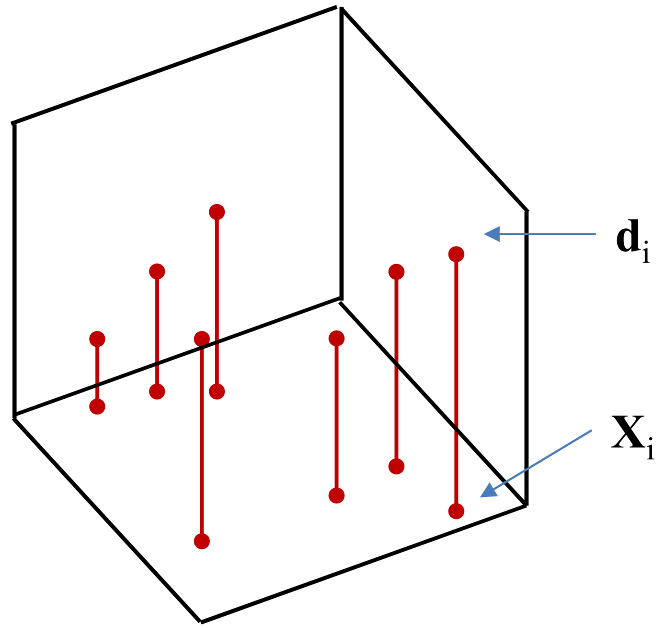
$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}}\, Loss\big(f(X;W), g(X)\big)$$

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

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

Recall: The *Empirical* risk





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

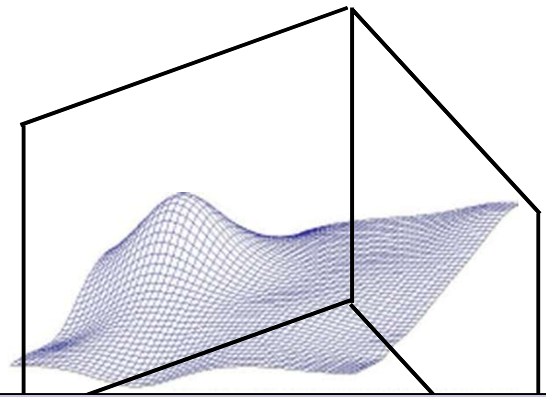
$$Loss(f(X;W), g(X)) = \frac{1}{N}\sum_{i=1}^{N} div(f(X_i;W), d_i)$$

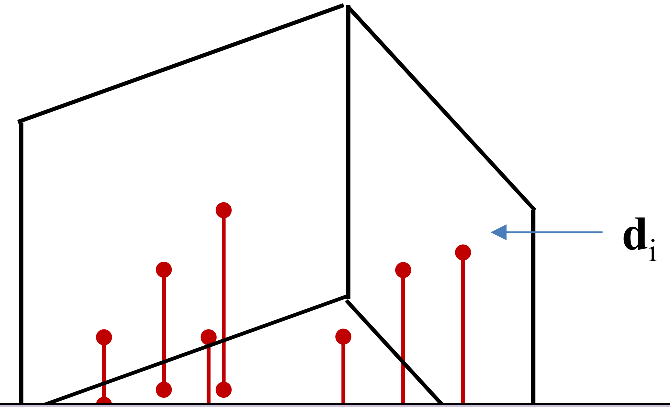
The empirical risk is an *unbiased* estimate of the expected loss

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

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

Recall: The *Empirical* risk





The variance of the empirical risk: var(Loss) = 1/N var(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 loss

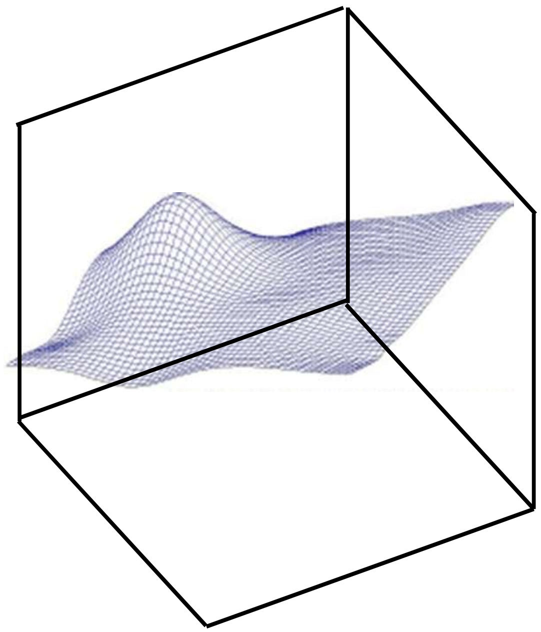
$$Loss\big(f(X;W), g(X)\big) = \frac{1}{N}\sum_{i=1}^{N} div(f(X_i;W), d_i)$$

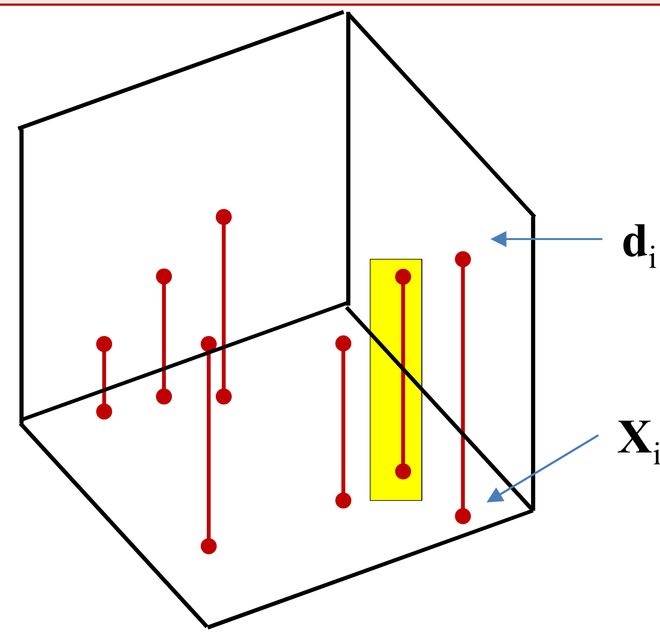
The empirical risk is an *unbiased* estimate of the expected loss

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

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

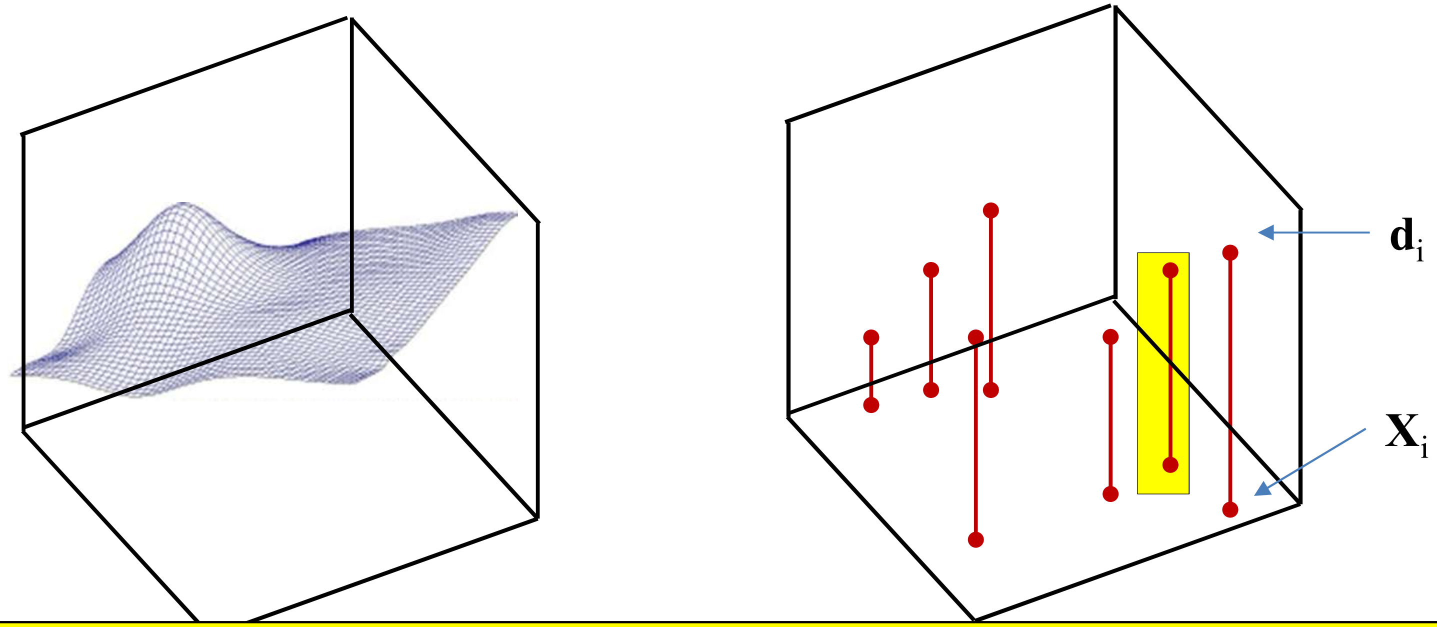
SGD





- 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* $E\left[div\left(f(X; W), g(X)\right)\right]$

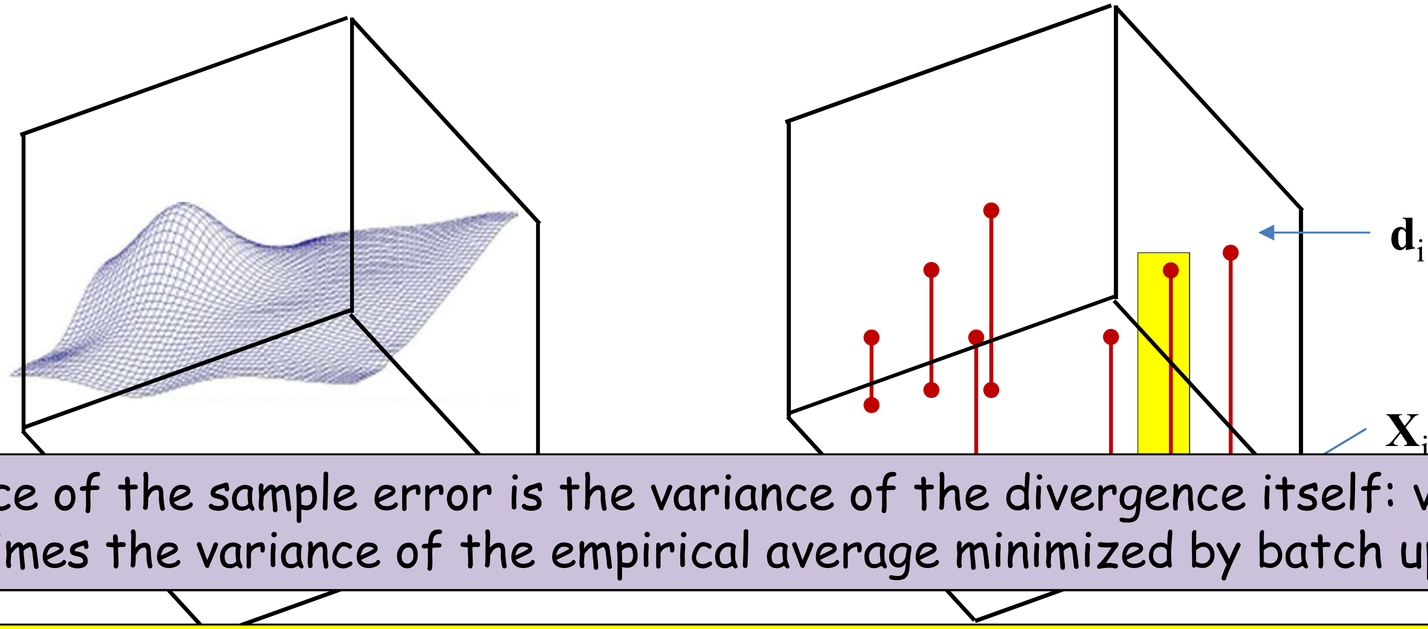
SGD



The sample error 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* $E\left[div\left(f(X; W), g(X)\right)\right]$

SGD

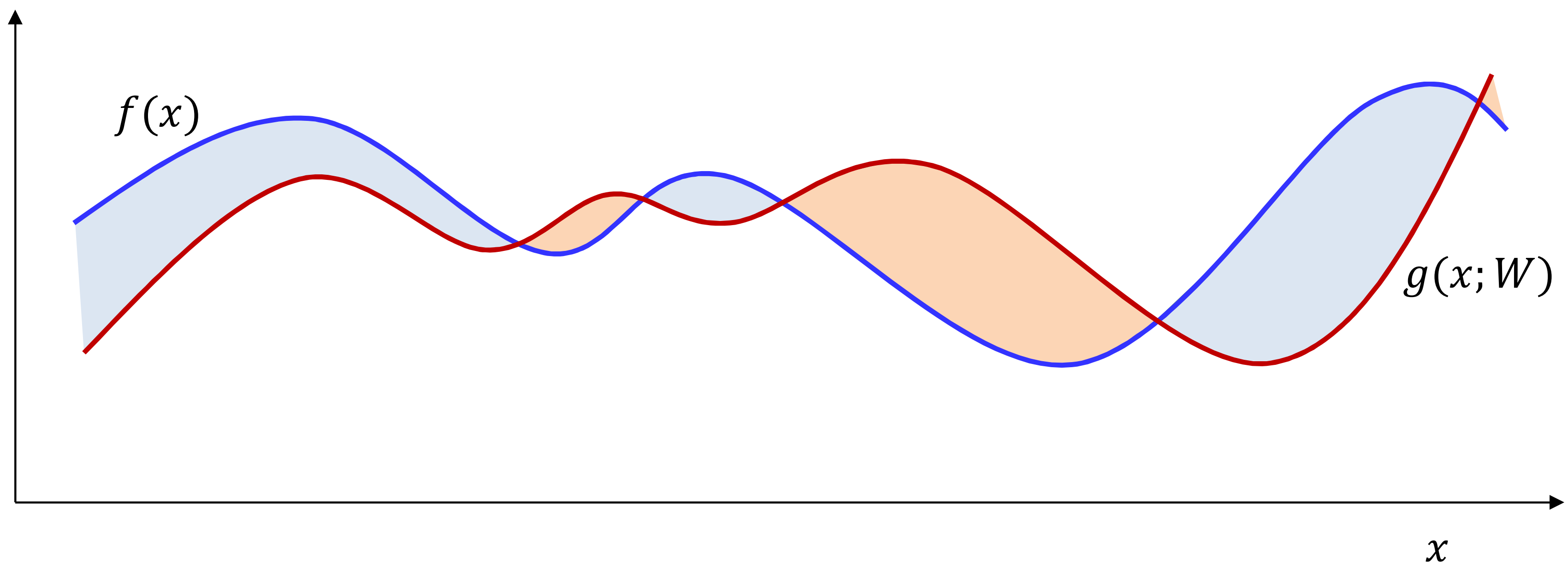


The variance of the sample error is the variance of the divergence itself: var(div)
This is N times the variance of the empirical average minimized by batch update

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

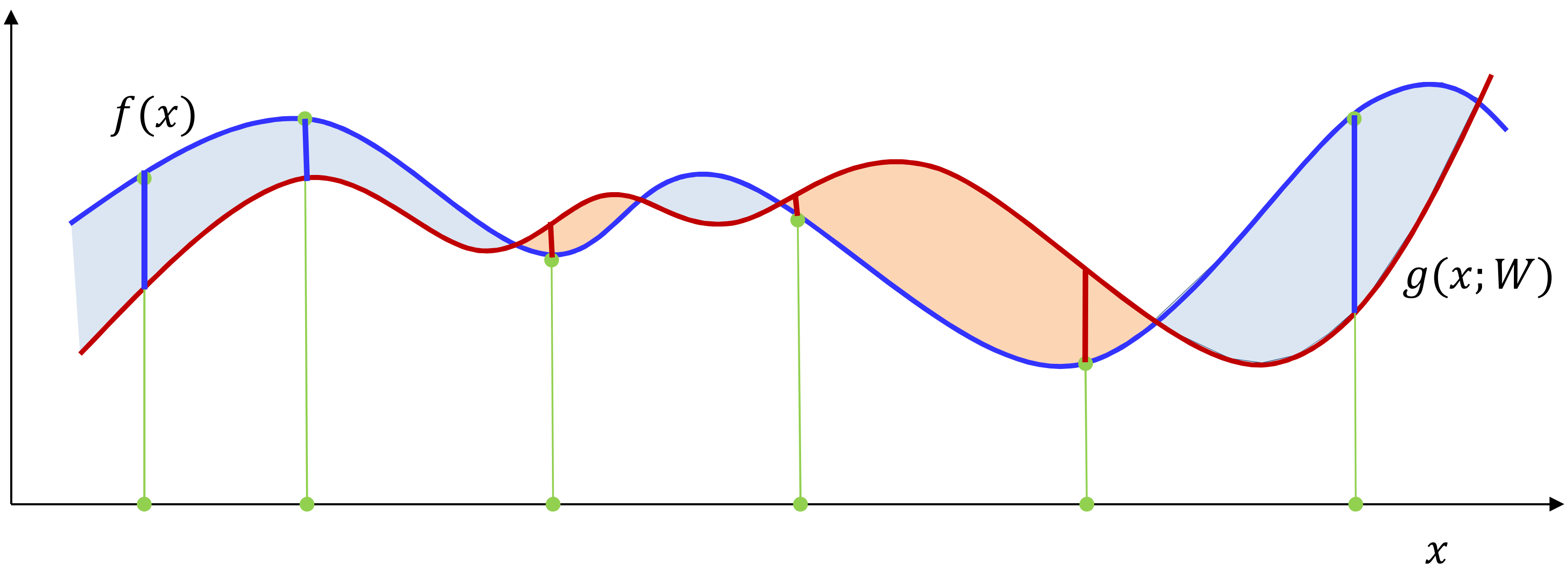
- At each iteration, **SGD** focuses on the divergence of a ***single*** sample $div(f(X_i; W), d_i)$
- The *expected value* of the *sample error* is ***still*** the *expected divergence* $E\left[div\left(f(X;W), g(X)\right)\right]$

Explaining the variance



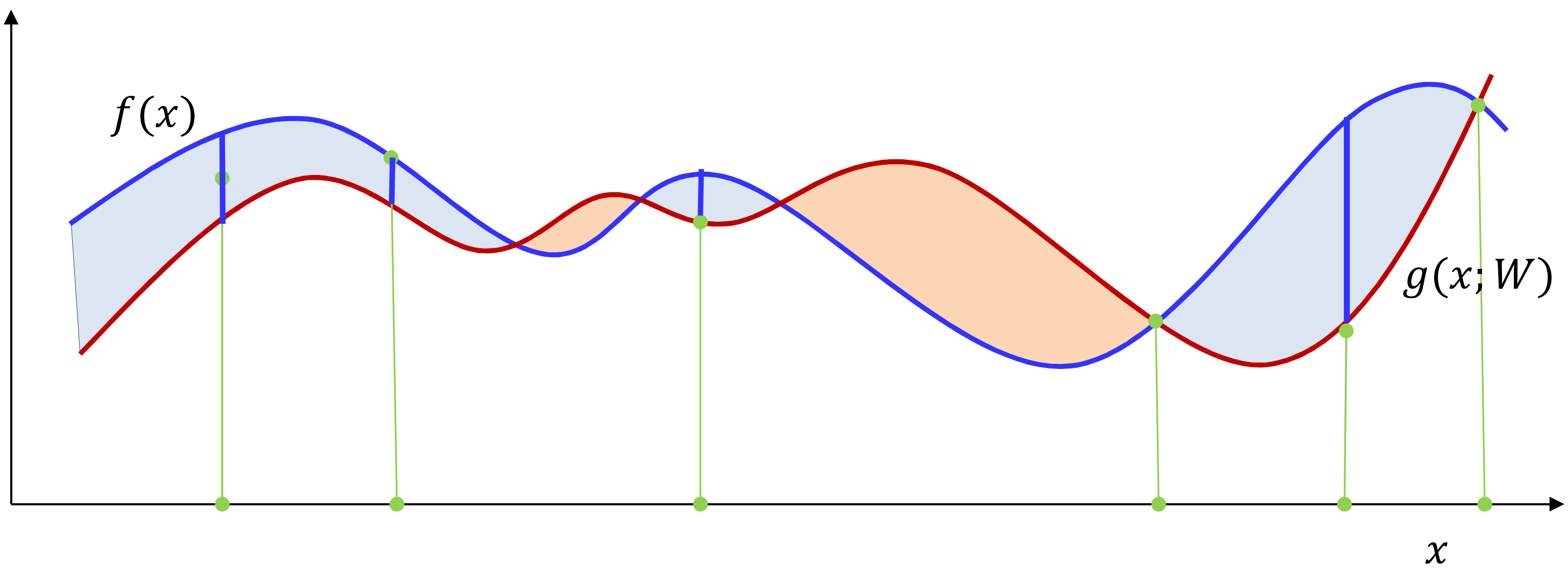
- The blue curve is the function being approximated
- The red curve is the approximation by the model at a given W
- The heights of the shaded regions represent the point-by-point error
  - The divergence is a function of the error
  - We want to find the W that minimizes the average divergence

Explaining the variance



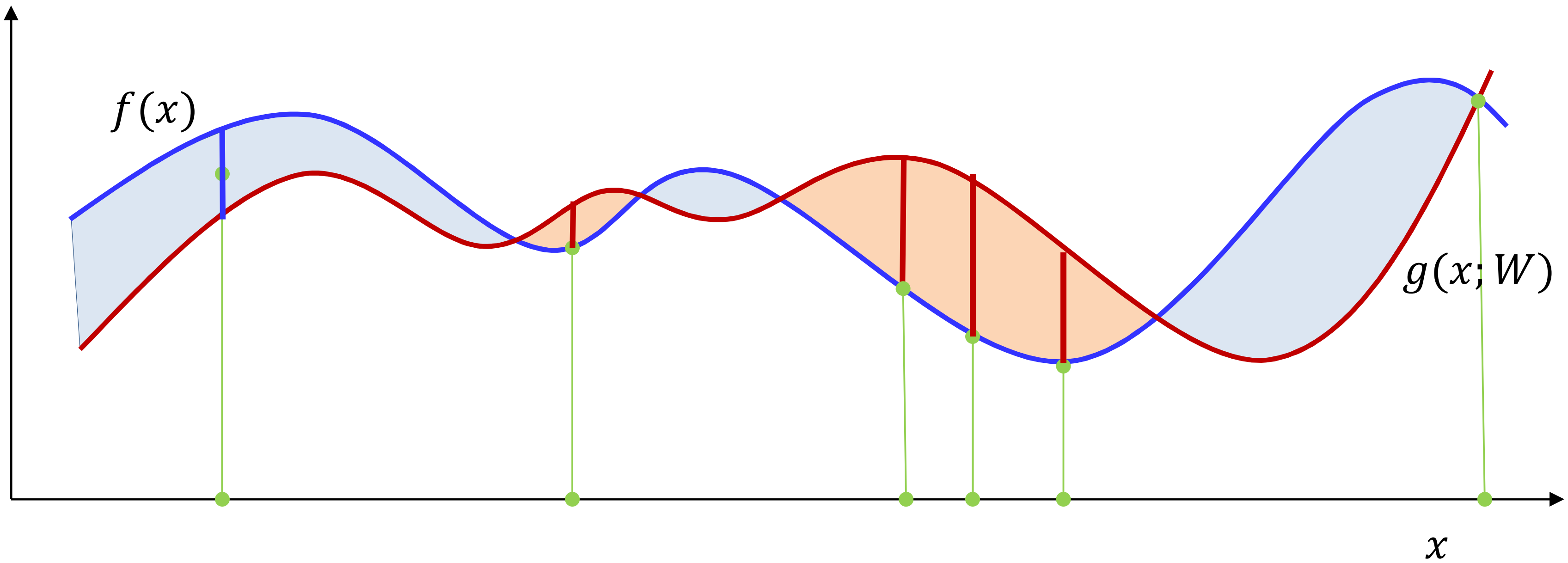
- Sample estimate approximates the shaded area with the average length of the lines

Explaining the variance



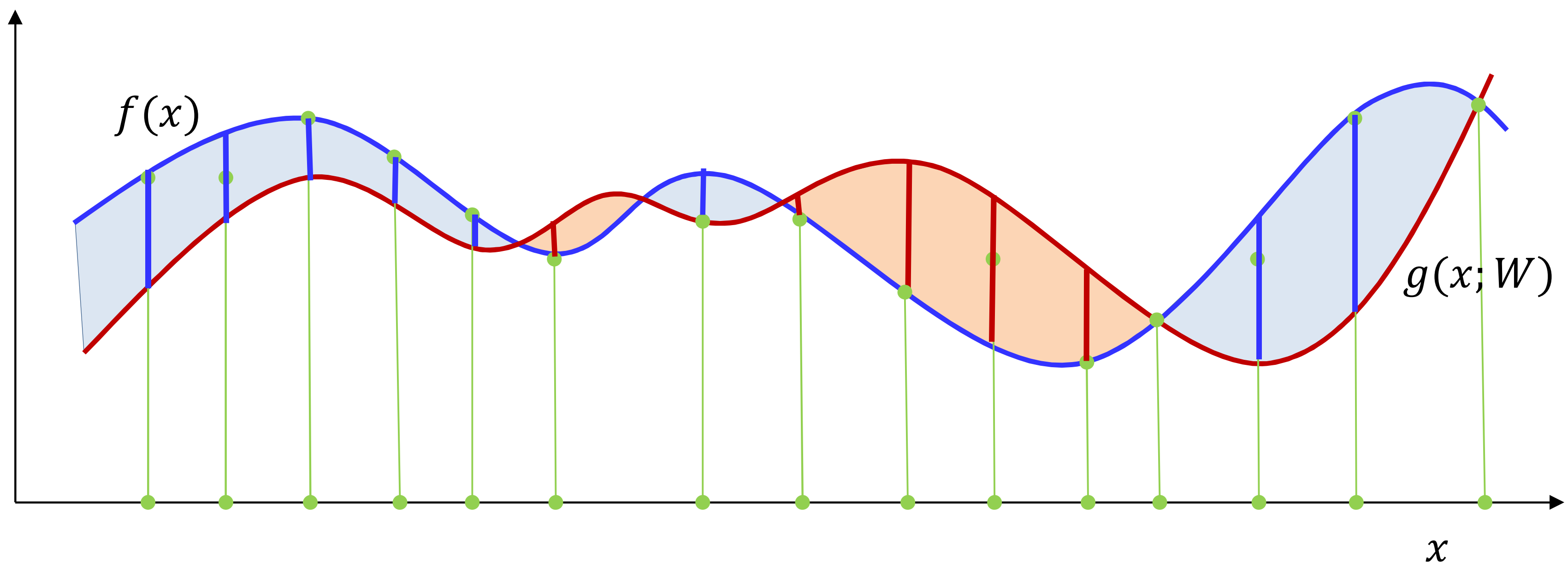
- Sample estimate approximates the shaded area with the average length of the lines
- This average length will change with position of the samples

Explaining the variance



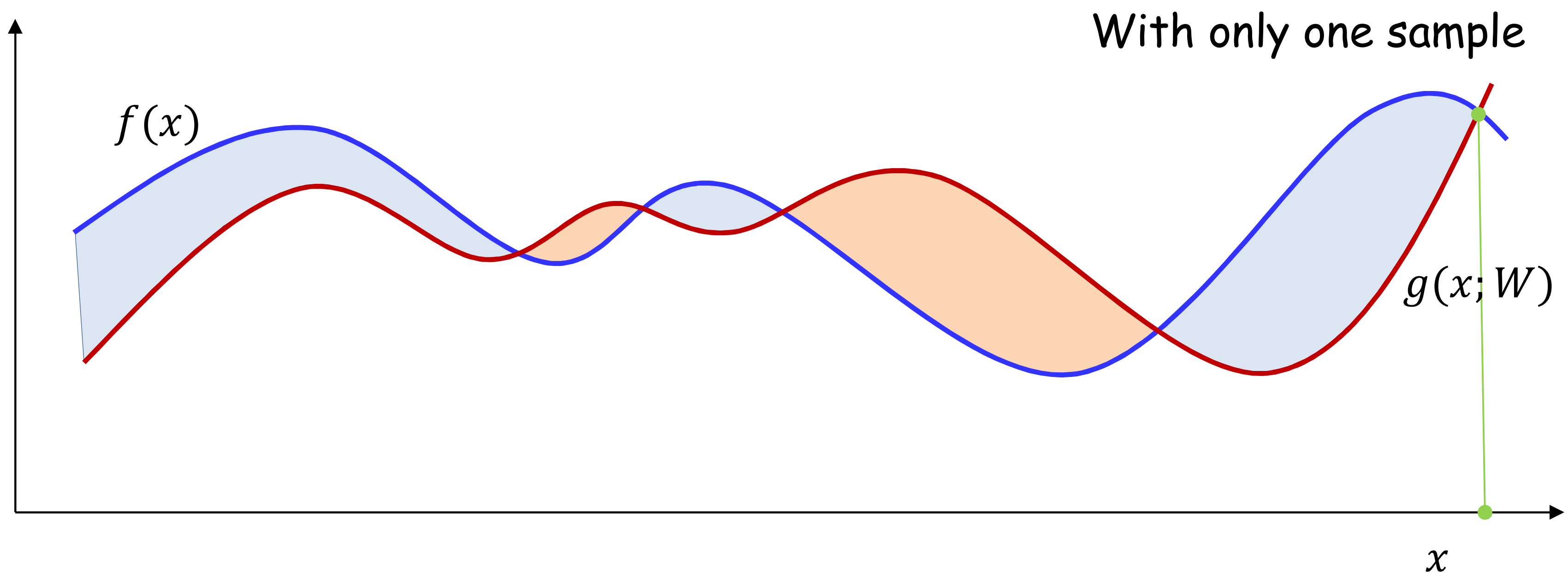
- Sample estimate approximates the shaded area with the average length of the lines
- This average length will change with position of the samples

Explaining the variance



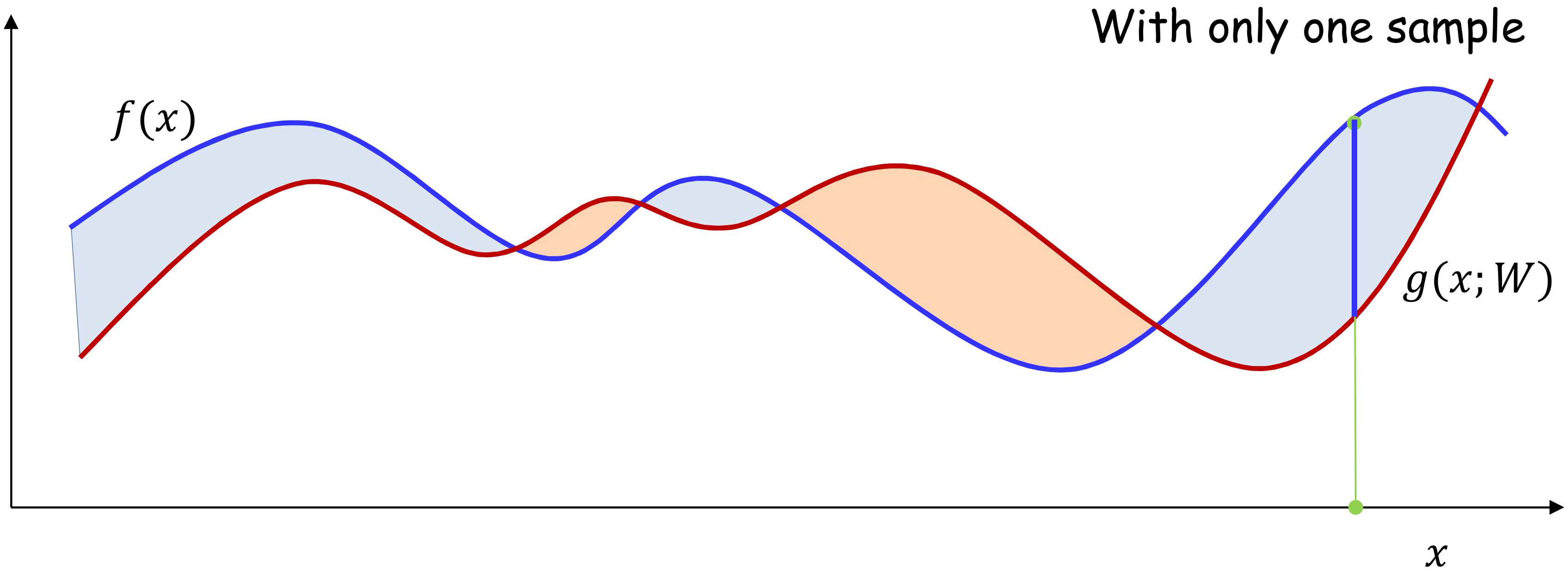
- Having more samples makes the estimate more robust to changes in the position of samples
  - The variance of the estimate is smaller

Explaining the variance



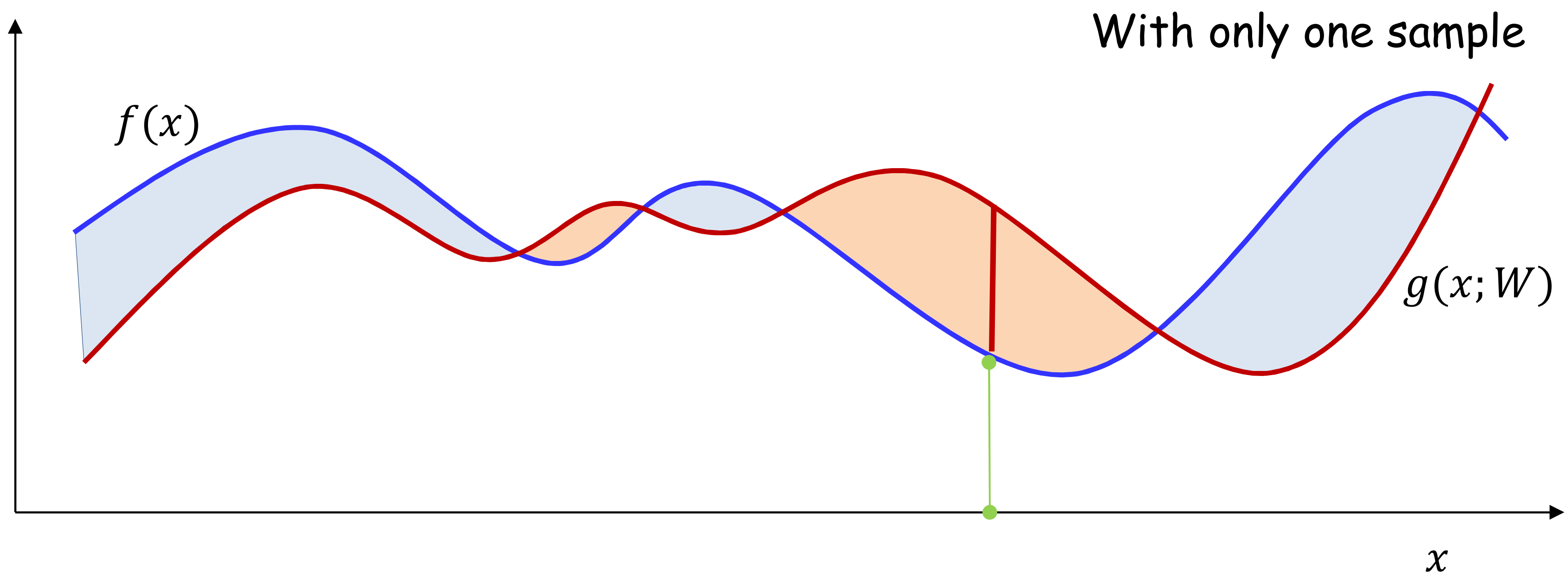
- Having very few samples makes the estimate swing wildly with the sample position
  - Since our estimator learns the W to minimize this estimate, the learned W too can swing wildly

Explaining the variance



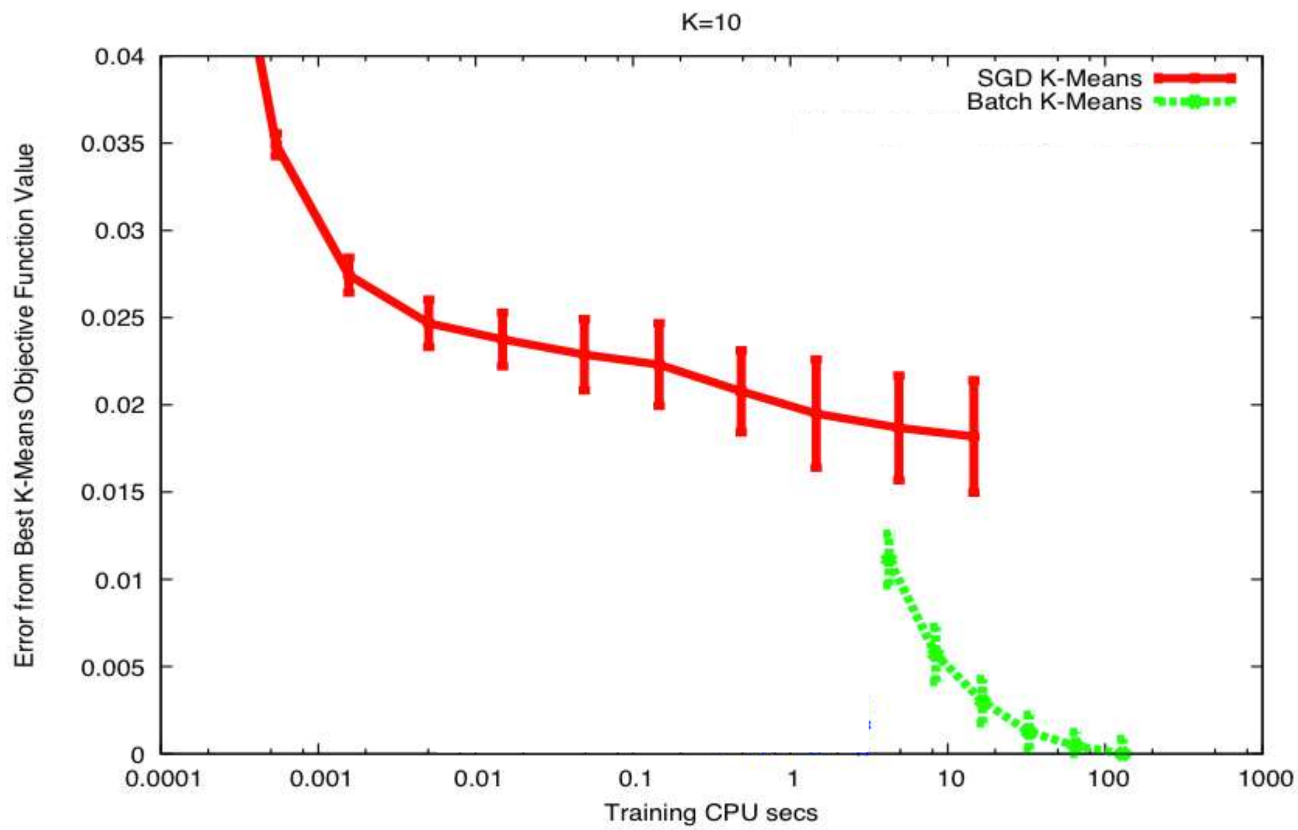
- Having very few samples makes the estimate swing wildly with the sample position
  - Since our estimator learns the W to minimize this estimate, the learned W too can swing wildly

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



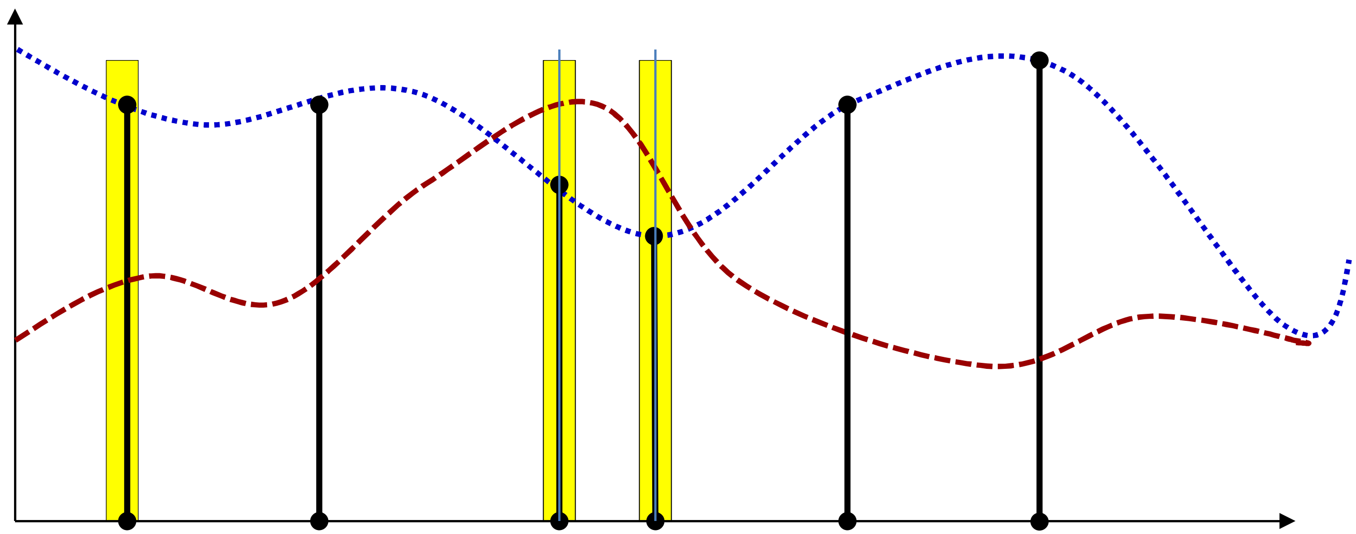
- A simpler problem: K-means
- Note: SGD converges slower
- Also has large variation between runs

SGD vs batch

- SGD uses the gradient from only one sample at a time, and is consequently high variance

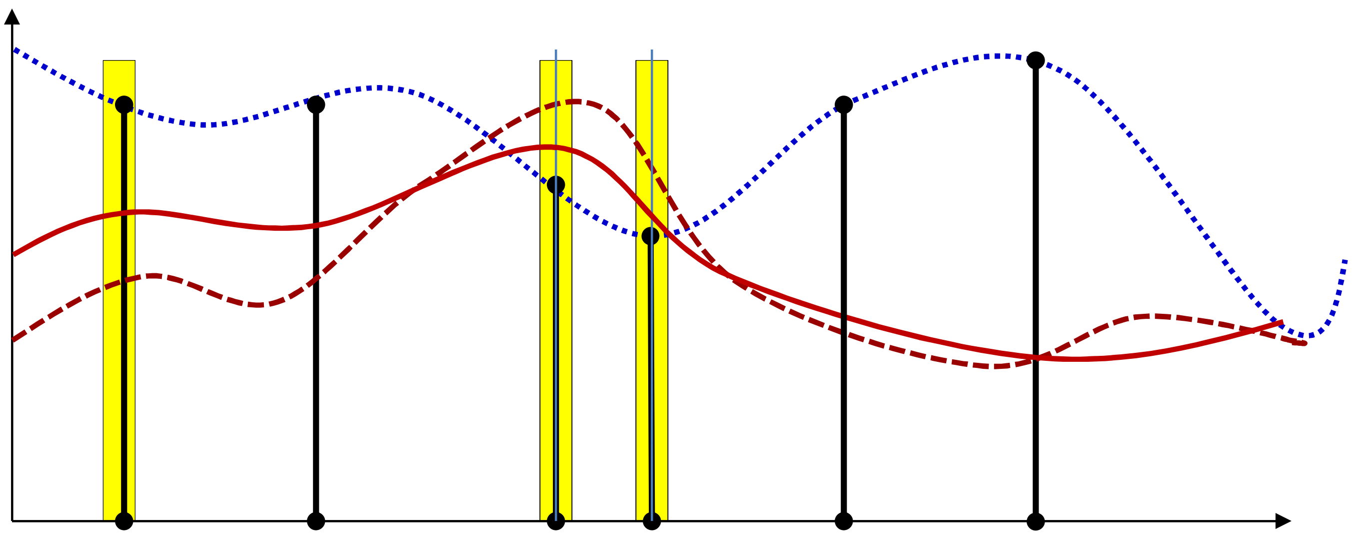
- But also provides significantly quicker updates than batch
- Is there a good medium?

Alternative: Mini-batch update



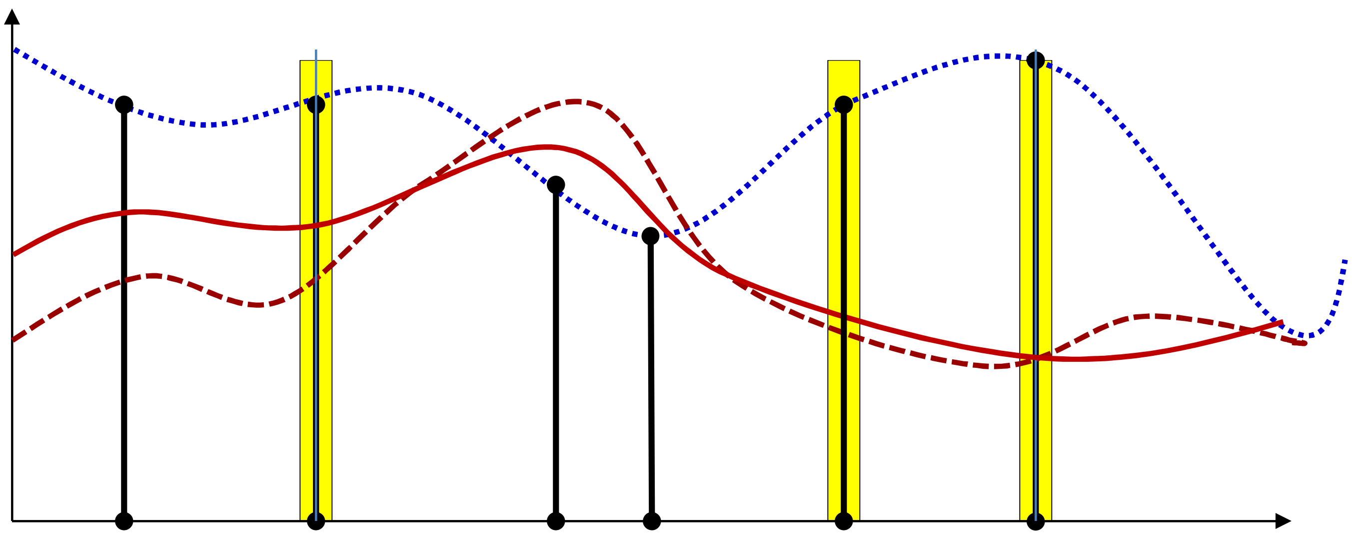
- Alternative: adjust the function at a small, randomly chosen subset of points
  - Keep adjustments small
  - If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data

Alternative: Mini-batch update



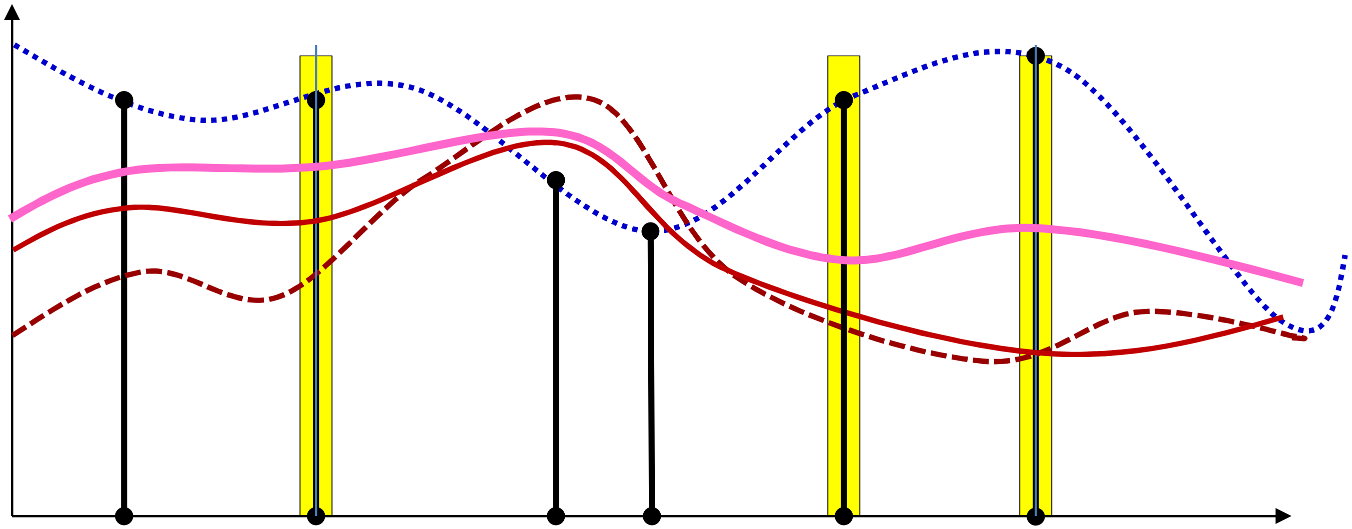
- Alternative: adjust the function at a small, randomly chosen subset of points
  – Keep adjustments small
  – If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data

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),…, (X_T, d_T)
- Initialize all weights $W_1, W_2, \ldots, W_K$; $j = 0$
- Do:
  - Randomly permute (X_1, d_1), (X_2, d_2),…, (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 $\nabla_{W_k} Div(Y_t, d_t)$
        - » $\Delta W_k = \Delta W_k + \frac{1}{b} \nabla_{W_k} Div(Y_t, d_t)^T$
    - 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),…, (X_T, d_T)
- Initialize all weights $W_1, W_2, \ldots, W_K$; $j = 0$
- Do:
  - Randomly permute (X_1, d_1), (X_2, d_2),…, (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 $\nabla_{W_k} Div(Y_t, d_t)$
        - » $\Delta W_k = \Delta W_k + \frac{1}{b} \nabla_{W_k} Div(Y_t, d_t)^T$
    - Update
      - For every layer k:

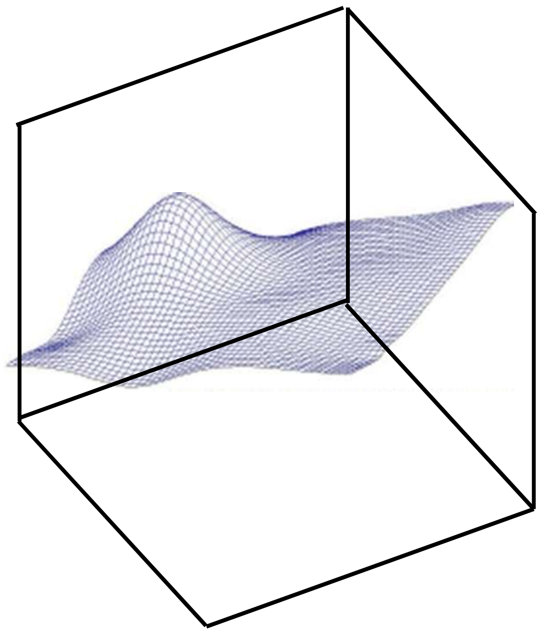
$$W_k = W_k - \eta_j \Delta W_k$$

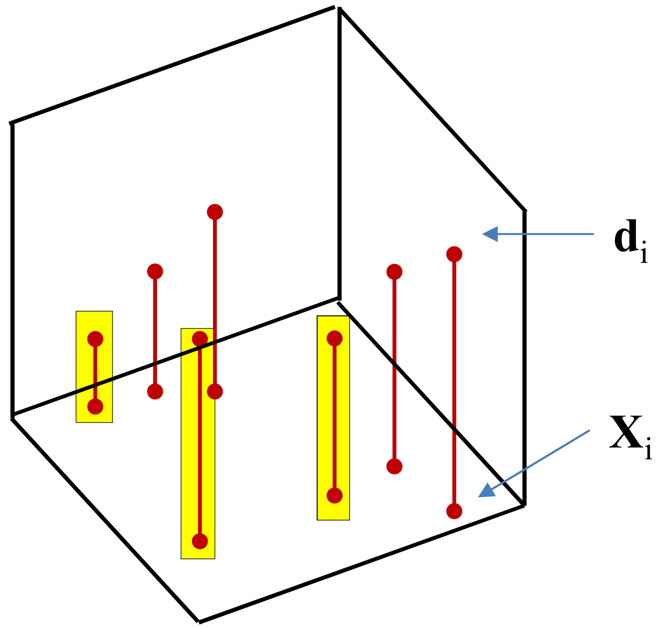
- Until Err has converged

Mini-batch size

Shrinking step size

Mini Batches





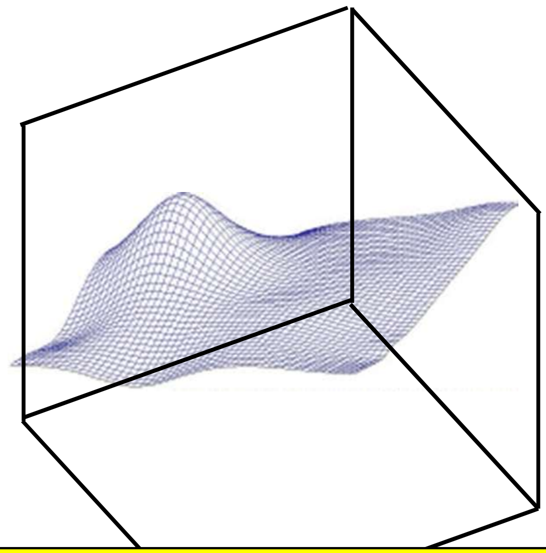
- Mini-batch updates compute and minimize a *batch loss*

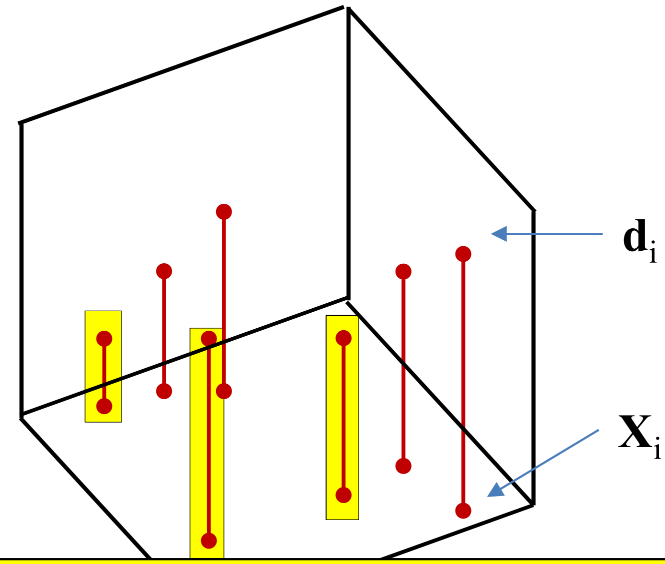
$$BatchLoss\big(f(X;W), g(X)\big) = \frac{1}{b}\sum_{i=1}^{b} div(f(X_i;W), d_i)$$

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

$$E\Big[BatchLoss\big(f(X;W), g(X)\big)\Big] = E\Big[div\big(f(X;W), g(X)\big)\Big]$$

Mini Batches





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

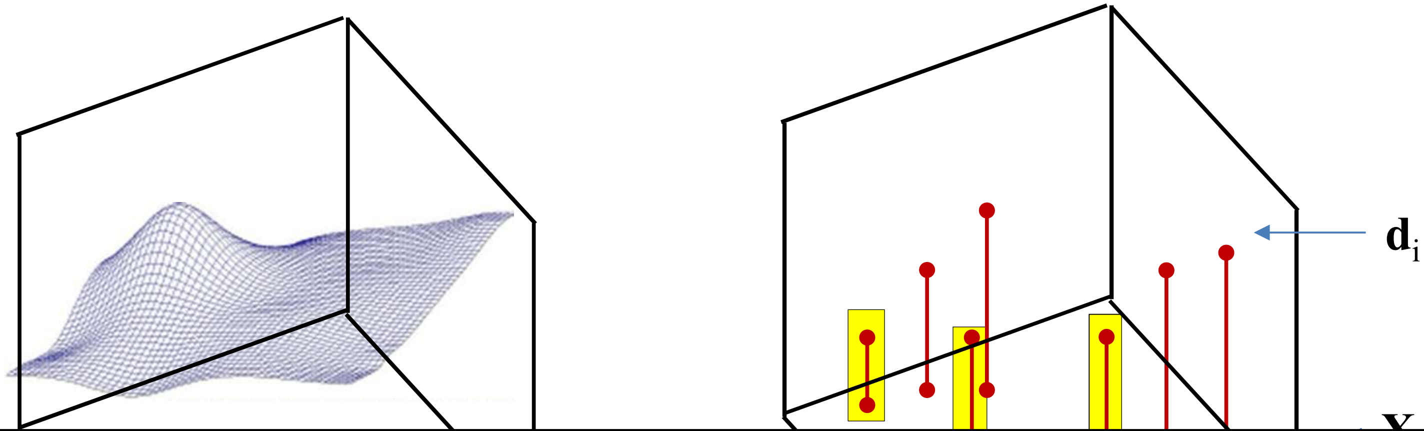
- Mini-batch updates compute and minimize a *batch loss*

$$BatchLoss\big(f(X;W), g(X)\big) = \frac{1}{b}\sum_{i=1}^{b} div(f(X_i;W), d_i)$$

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

$$E\Big[BatchLoss\big(f(X;W), g(X)\big)\Big] = E\Big[div\big(f(X;W), g(X)\big)\Big]$$

Mini Batches



The variance of the batch loss: var(BatchLoss) = 1/b var(div)
This will be much smaller than the variance of the sample error in SGD

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

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

$$BatchLoss\big(f(X;W),g(X)\big) = \frac{1}{b}\sum_{i=1}^{b} div(f(X_i;W),d_i)$$

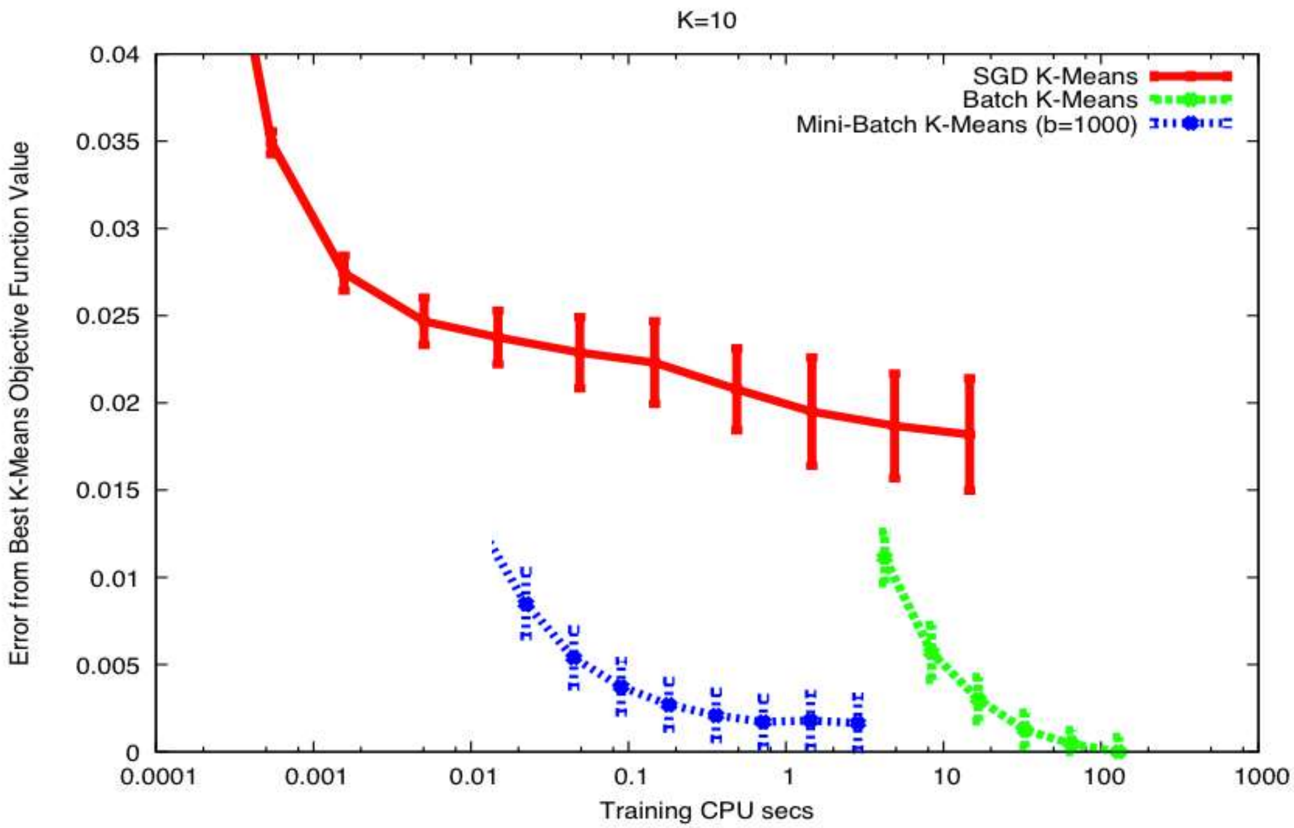
- The *expected value* of the *batch loss* is also the *expected divergence*

$$E\big[BatchLoss\big(f(X;W),g(X)\big)\big] = E\big[div\big(f(X;W),g(X)\big)\big]$$

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

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