Train Your Model: Optimization 2

CSE 849 Deep Learning

Spring 2025

Zijun Cui

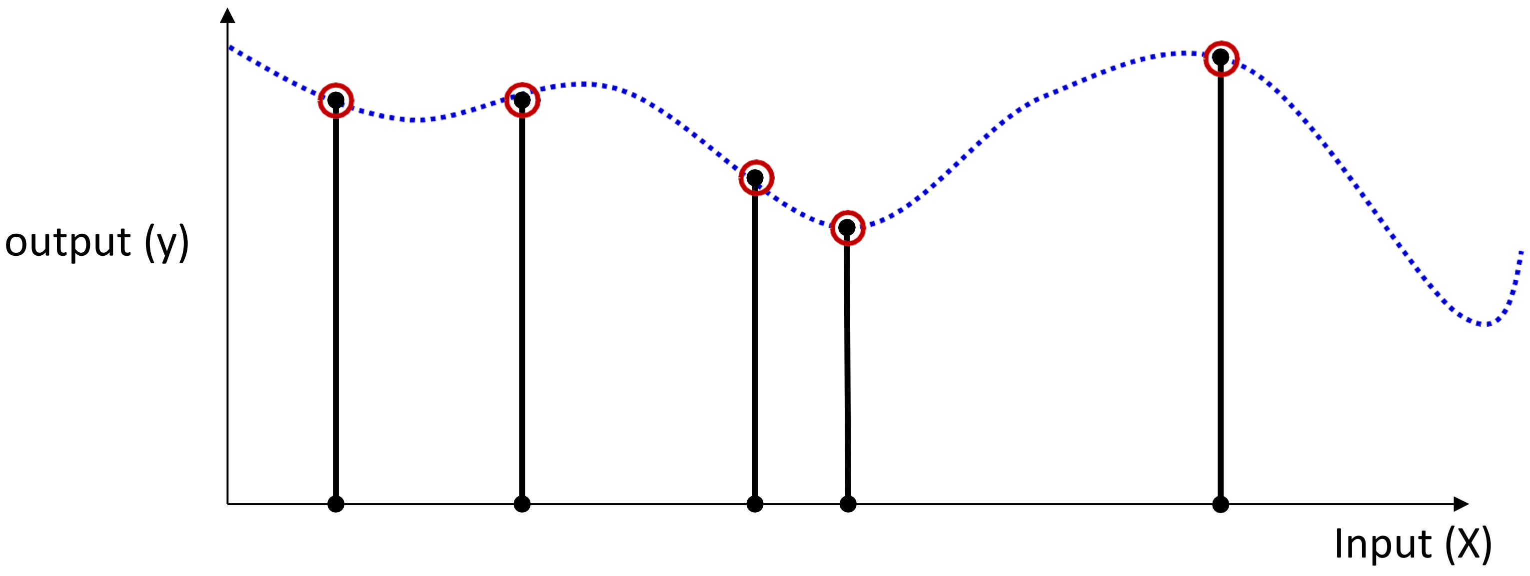
Partial slides are adapted from CMU 11-785 Introduction to Deep Learning by Bhiksha Raj

Today’s Topic

- Incremental updates
  - Batch
  - Stochastic Gradient Descent
  - Mini-Batch

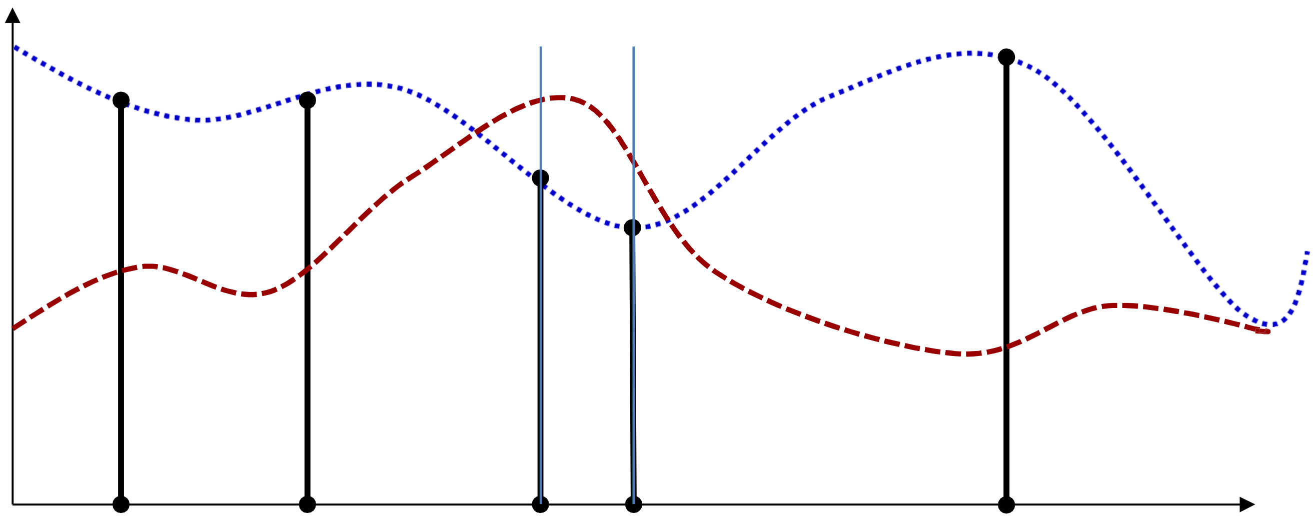
- Revisiting optimization algorithms with incremental updates
  - Advanced methods for training with mini-batch
  - Adam Optimization

The training formulation



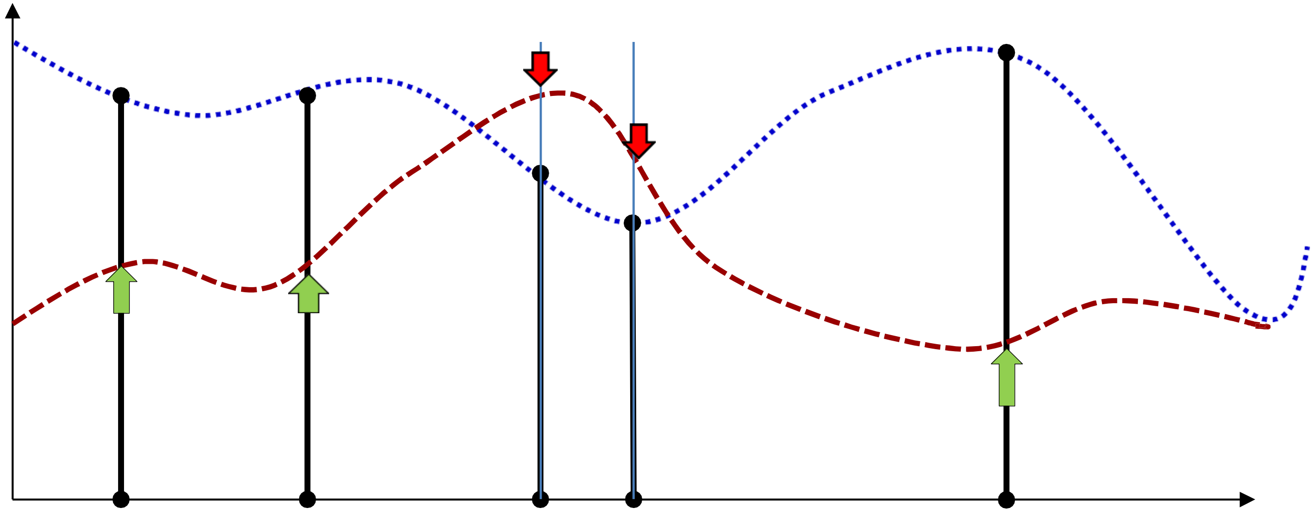
- Given input output pairs at a number of locations, estimate the entire function

Gradient descent



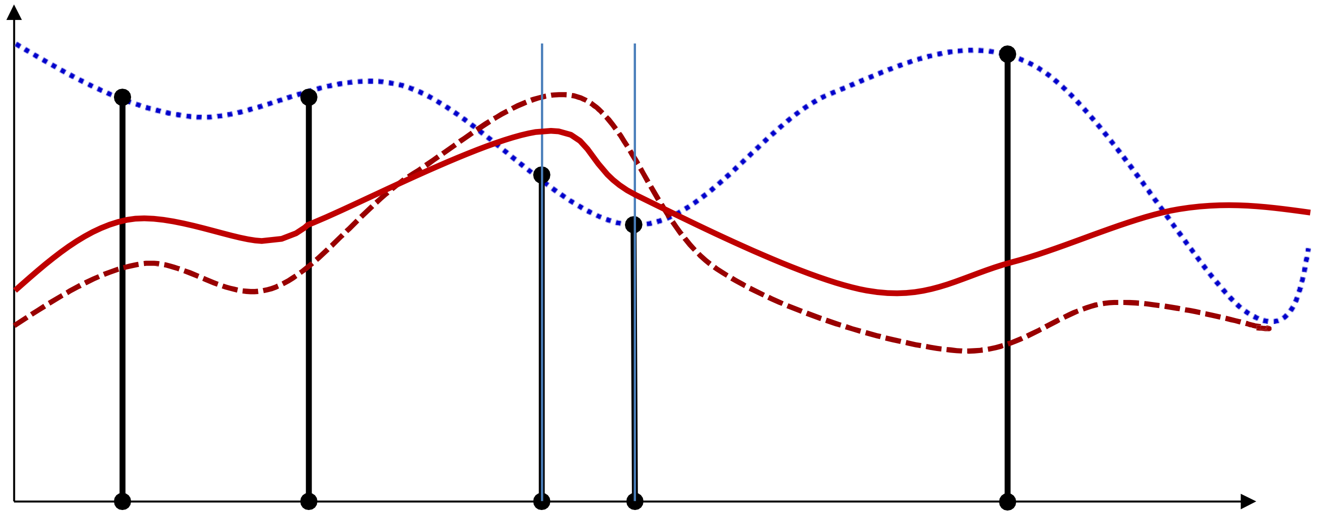
- Start with an initial function

Gradient descent

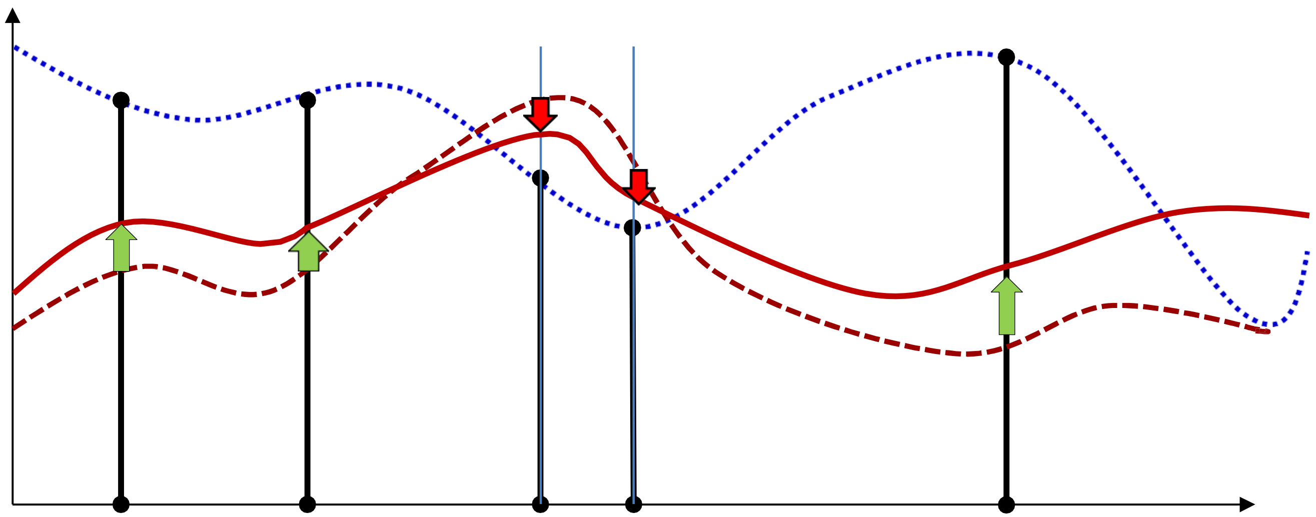


- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

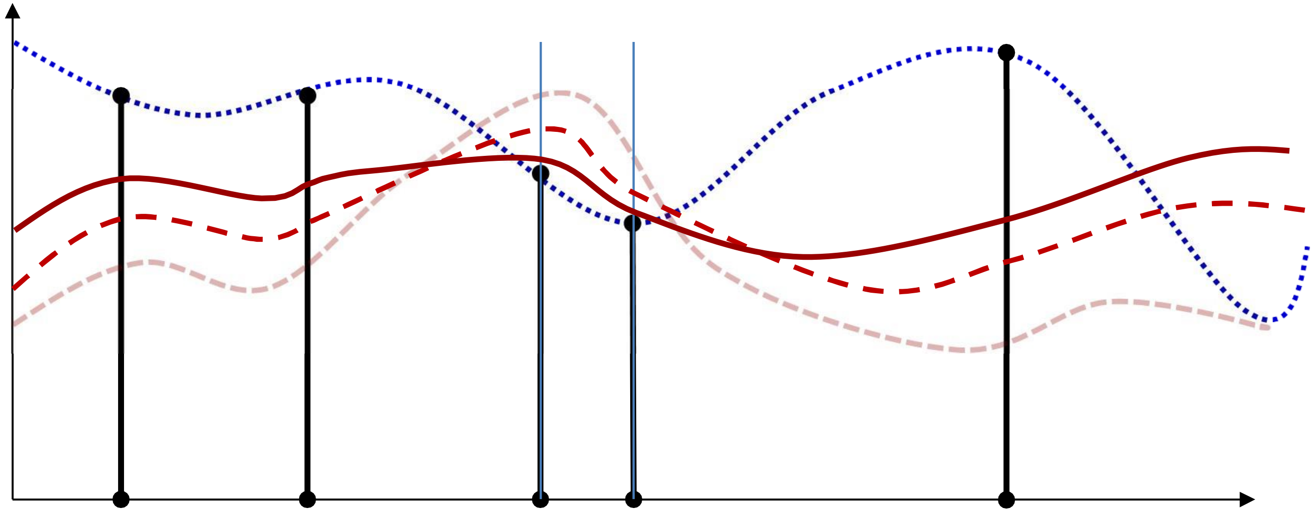
Gradient descent



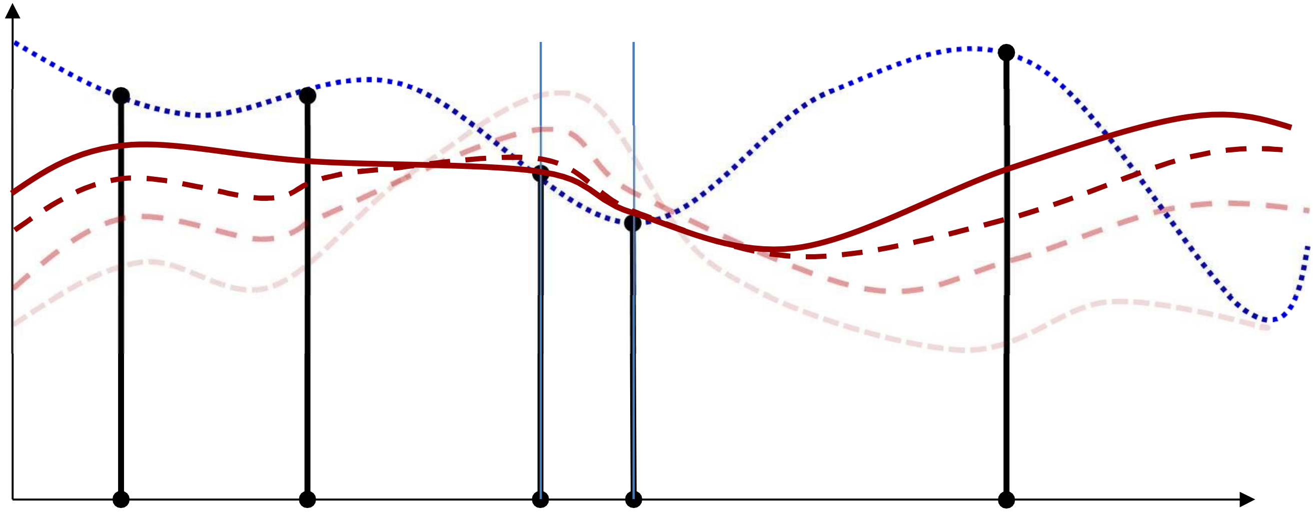
Gradient descent



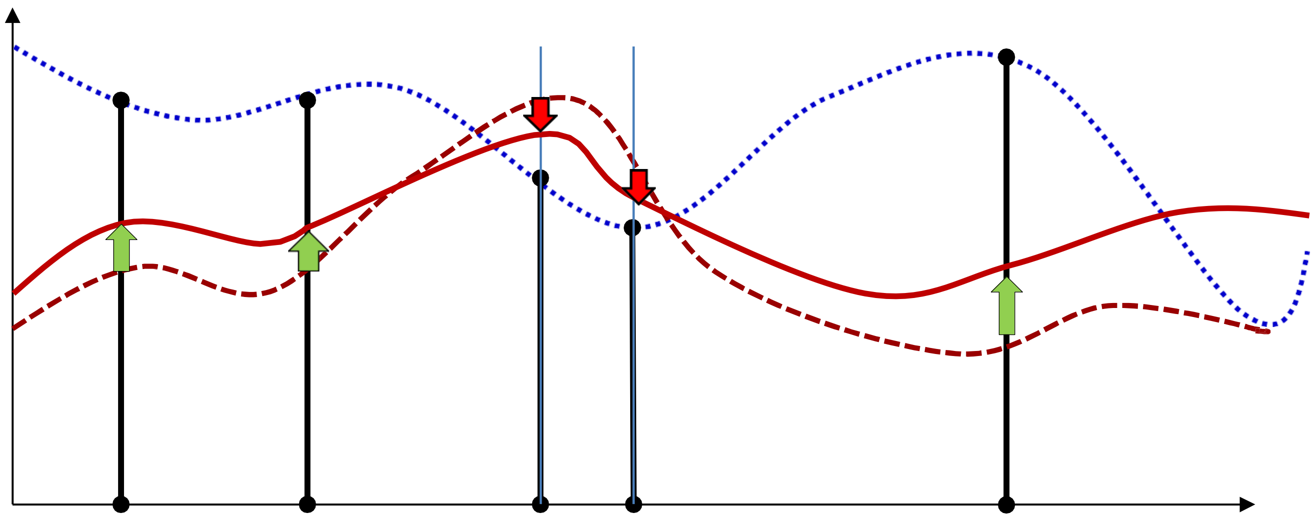
Gradient descent



Gradient descent

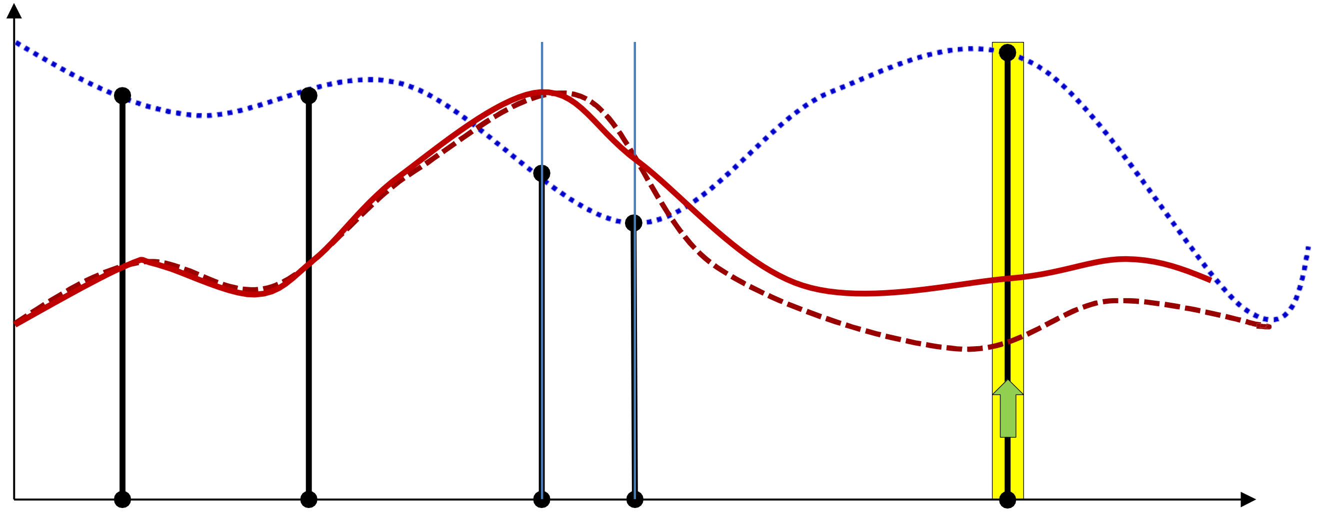


Effect of number of samples



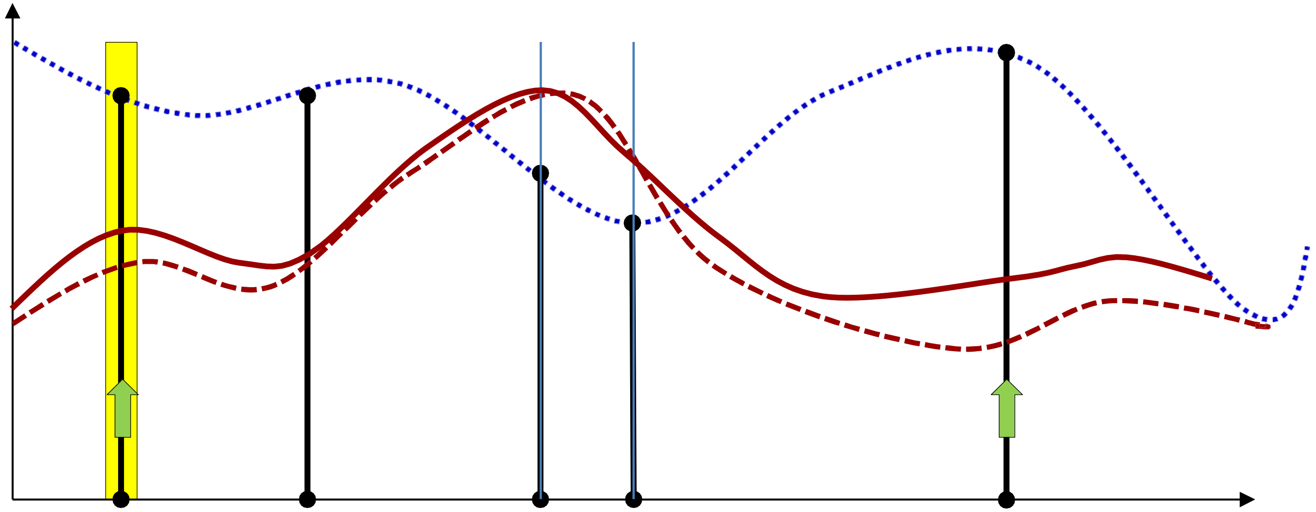
- Problem with conventional gradient descent: we try to simultaneously adjust the function at *all* training points
  - We must process *all* training points before making a single adjustment
  - "**Batch**" update

Alternative: Incremental update

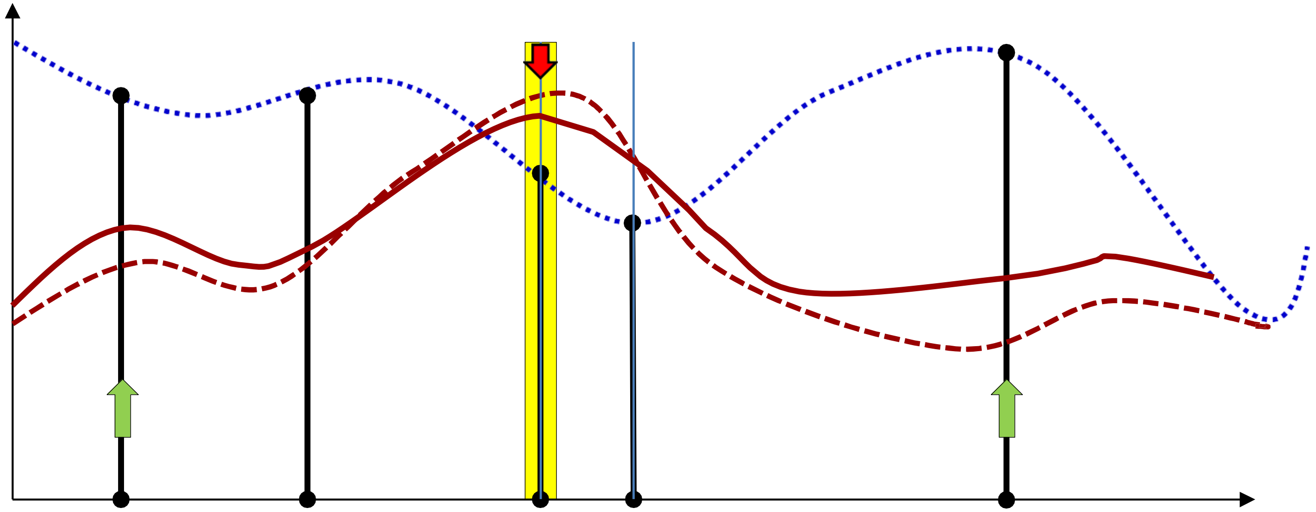


- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

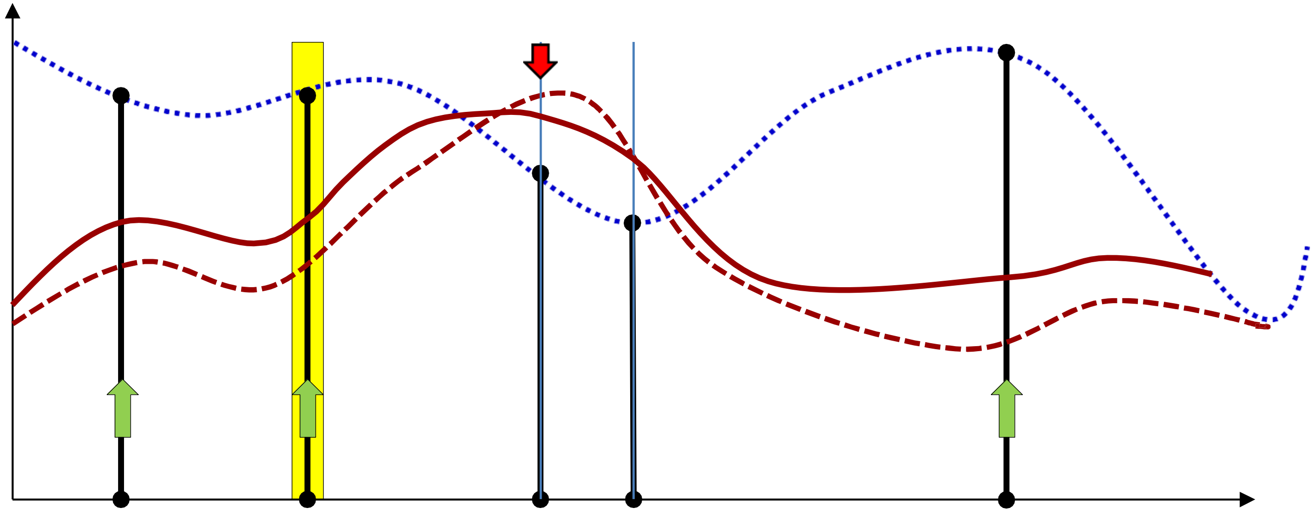
Alternative: Incremental update



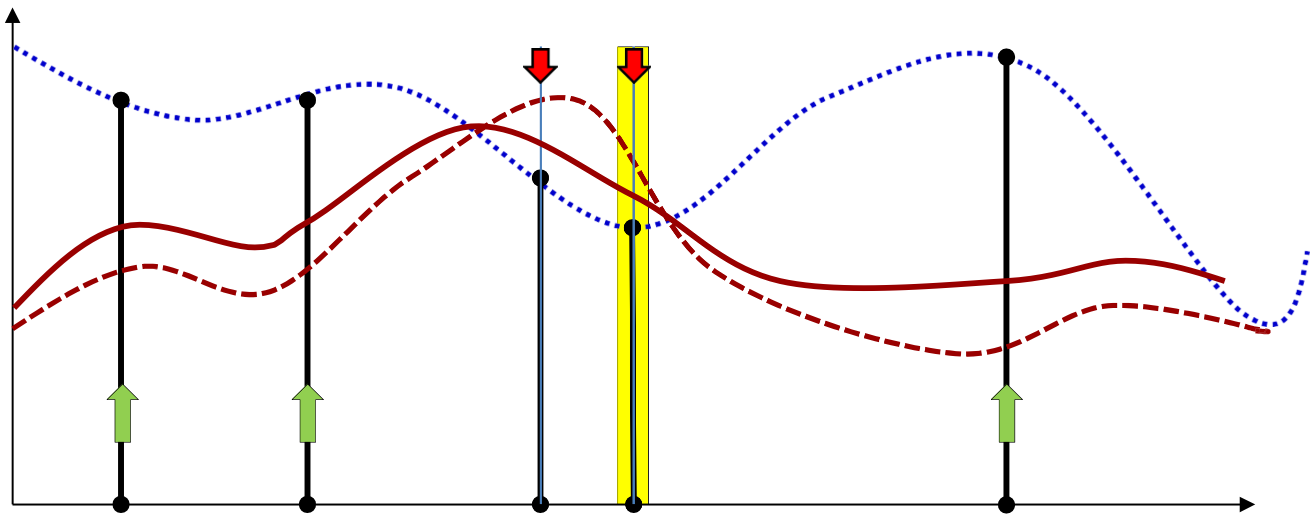
Alternative: Incremental update



Alternative: Incremental update



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 $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$
- 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

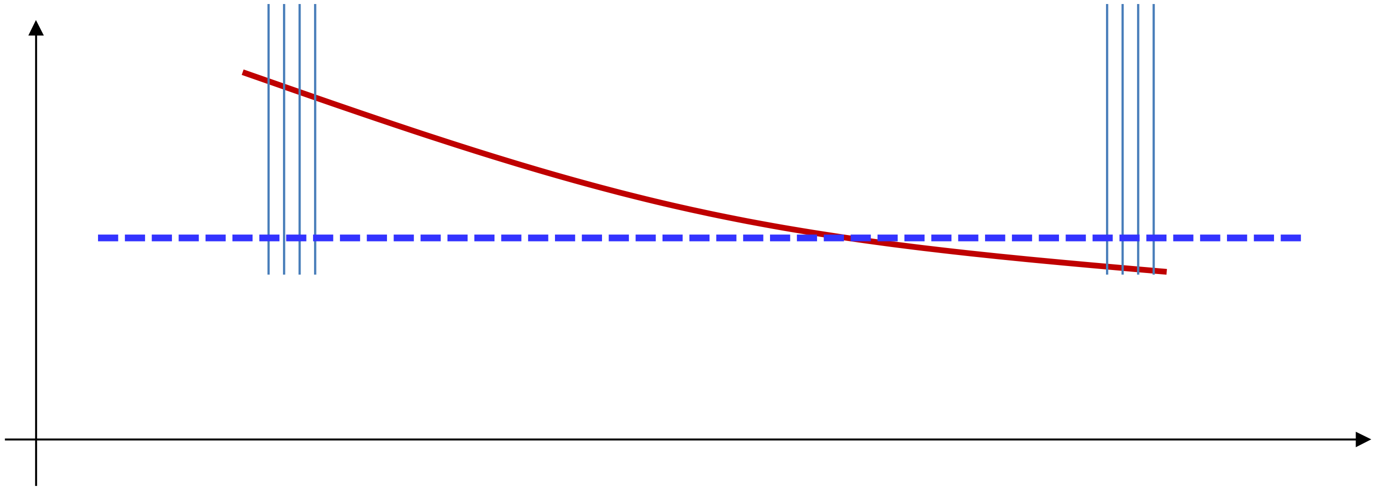
- The iterations can make multiple passes over the data
- A single pass through the entire training data is called an "epoch"
  - An epoch over a training set with T samples results in T updates of parameters

Incremental Update

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

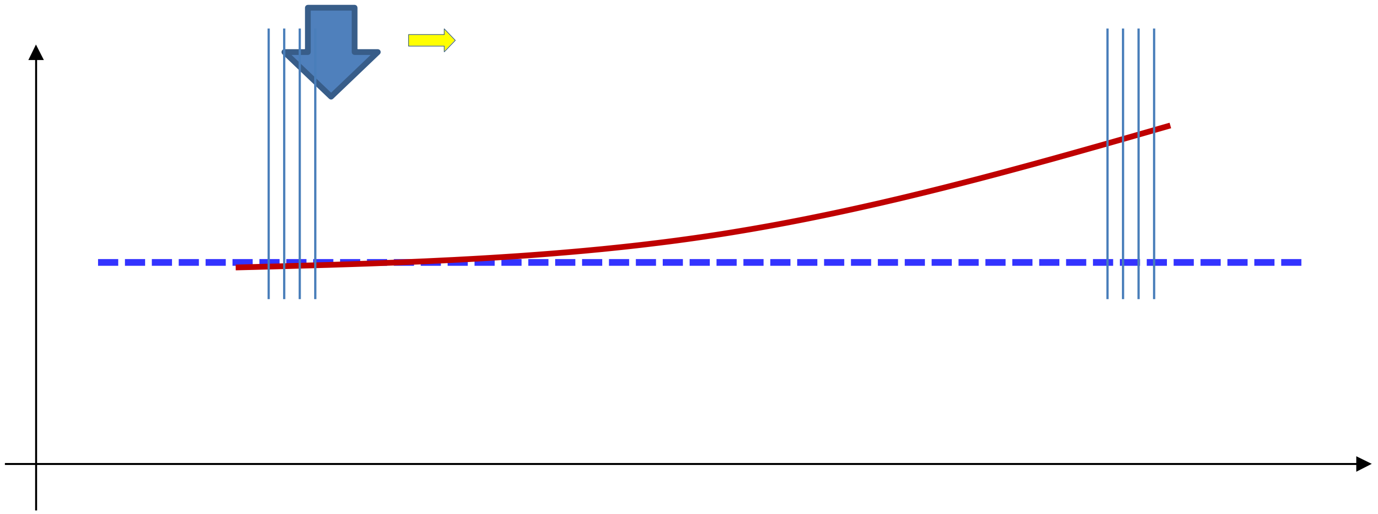
        $W_k = W_k - \eta \nabla_{W_k} \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})^T$ One update
- Until $Loss$ has converged

Caveats: order of presentation

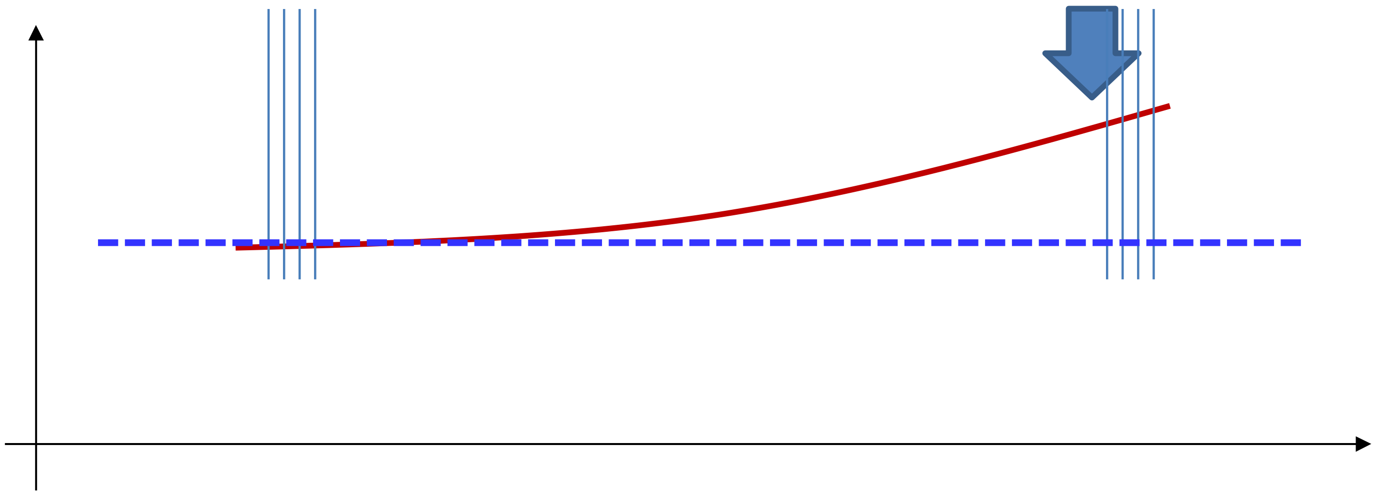


- If we loop through the samples in the same order, we may get *cyclic* behavior

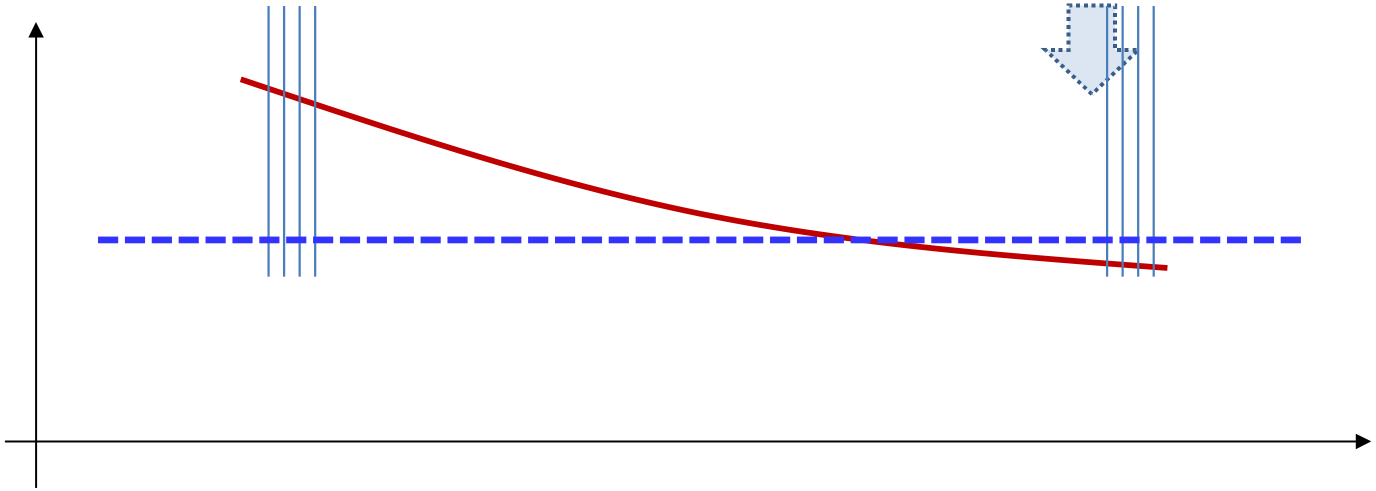
Caveats: order of presentation



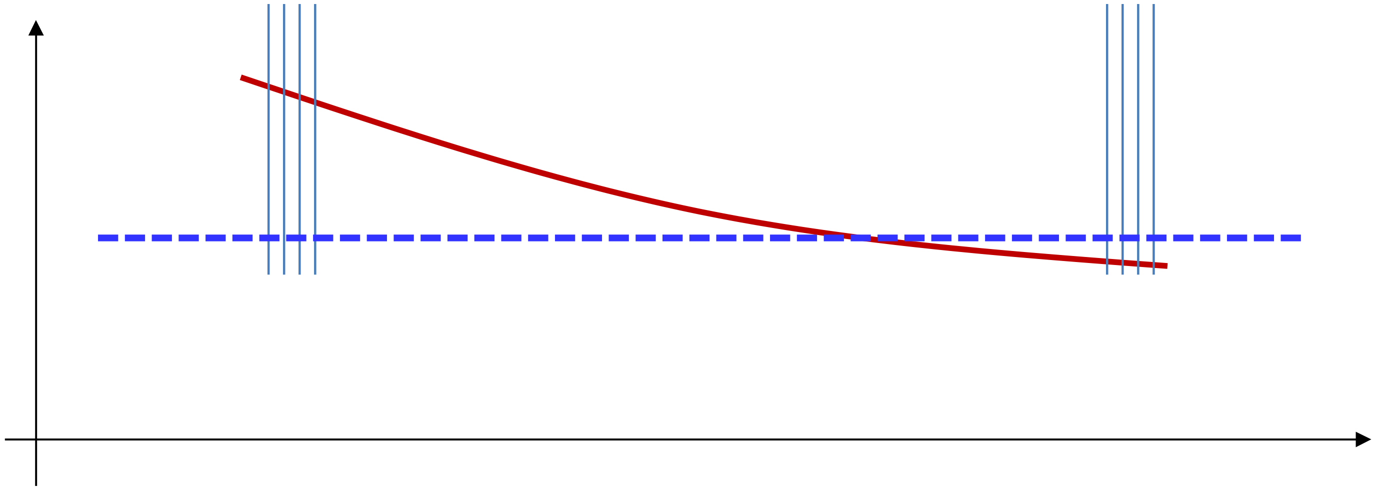
Caveats: order of presentation



Caveats: order of presentation

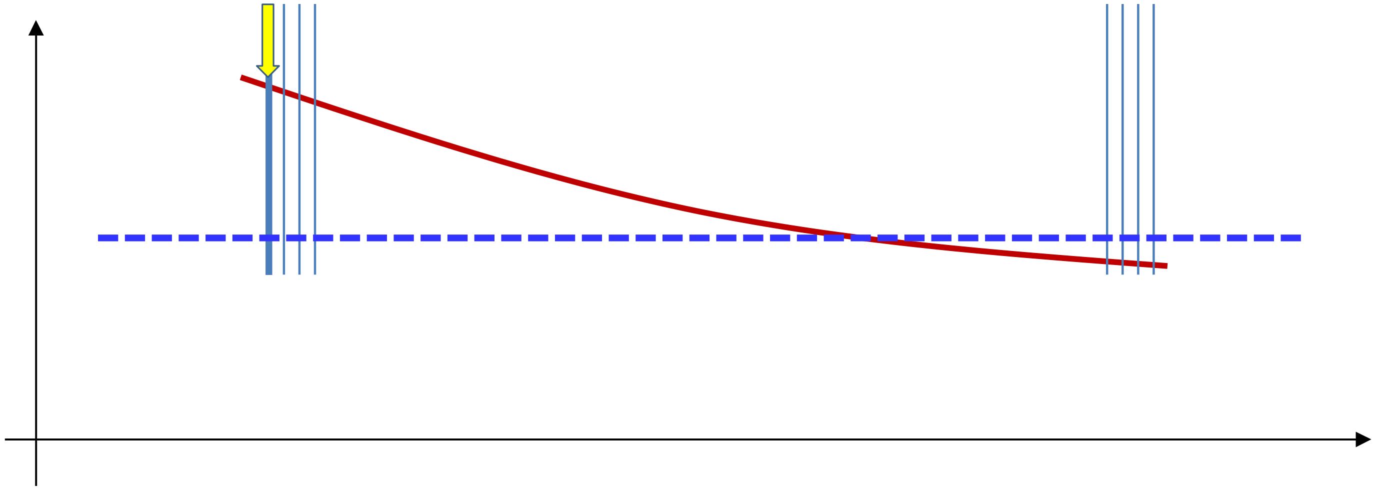


Caveats: order of presentation



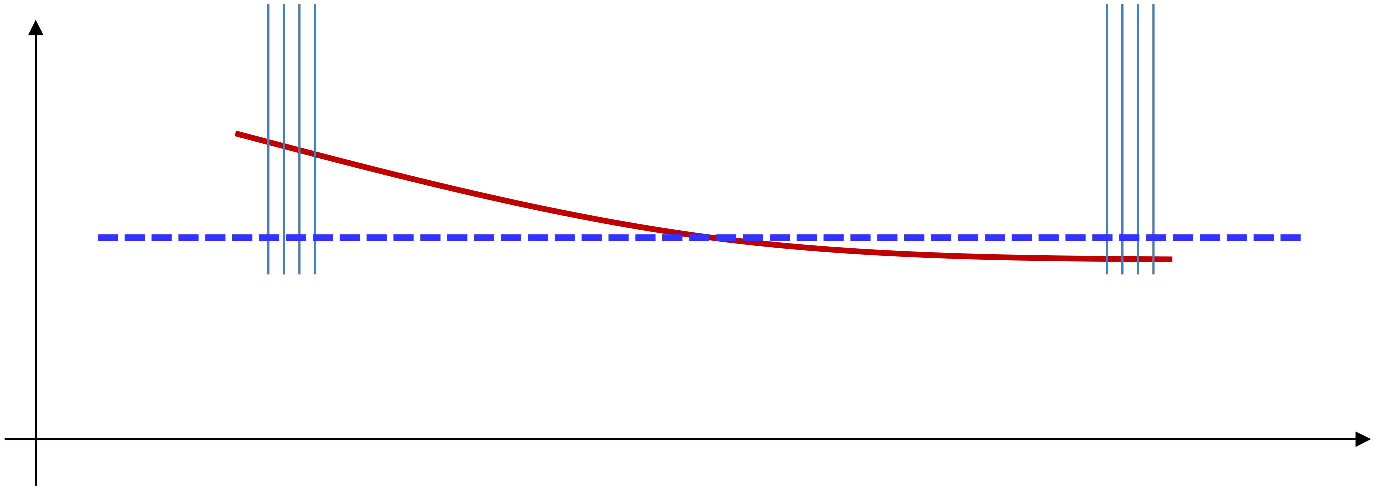
- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly* to get more convergent behavior

Caveats: order of presentation



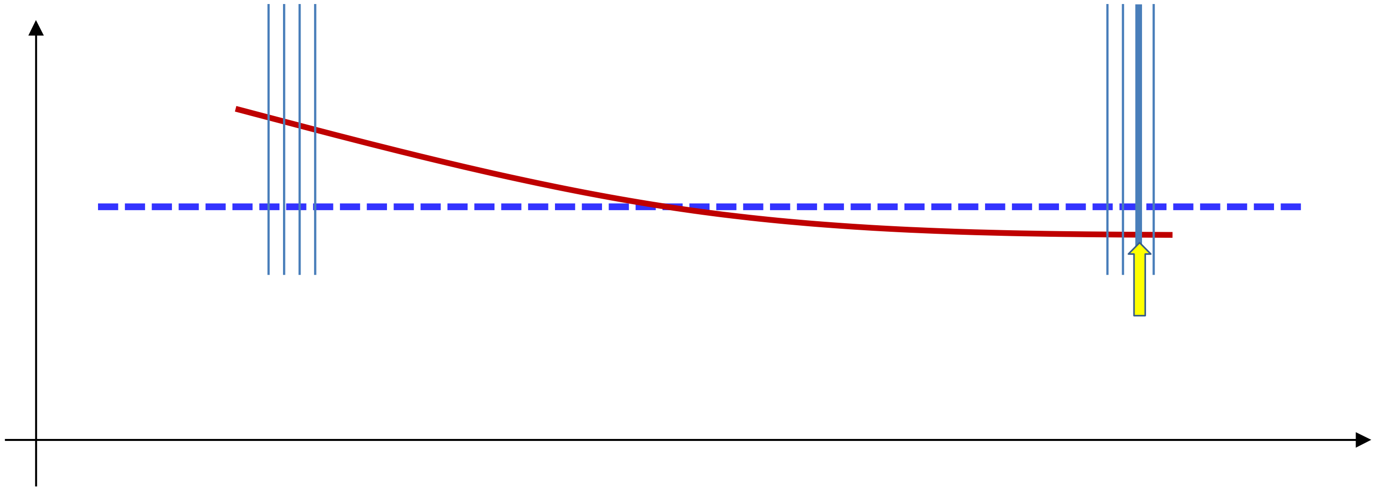
- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly* to get more convergent behavior

Caveats: order of presentation



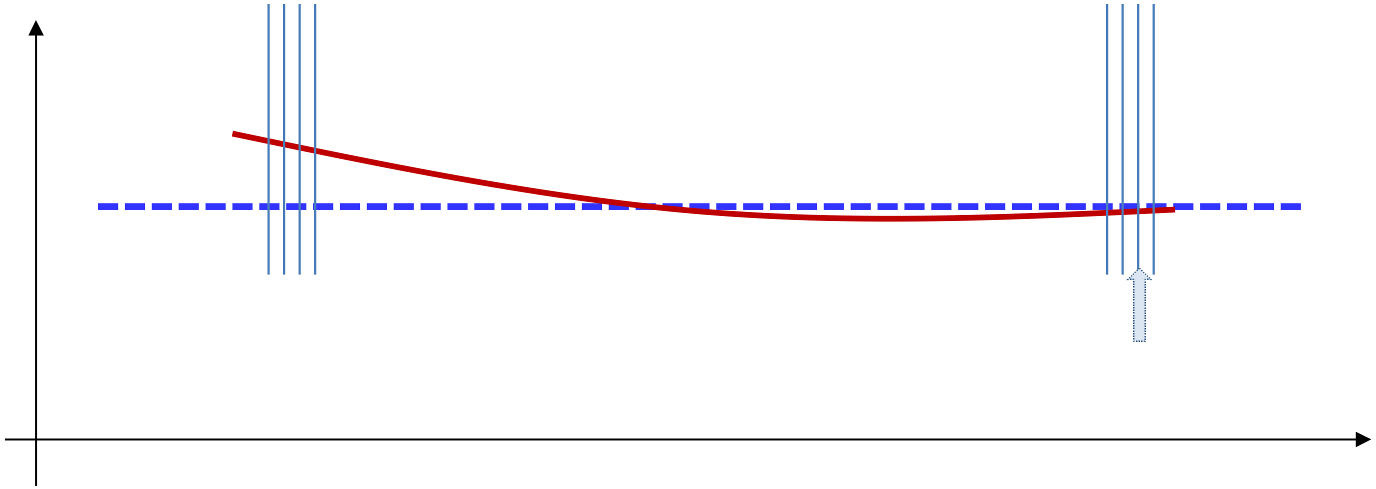
- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly* to get more convergent behavior

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 $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$
- 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

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

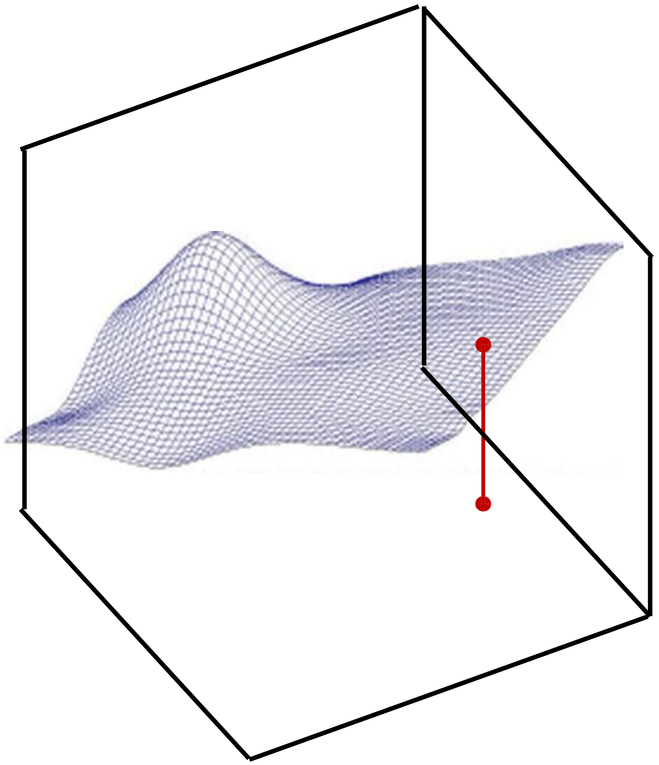
- Until $Loss$ has converged

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

Extreme example

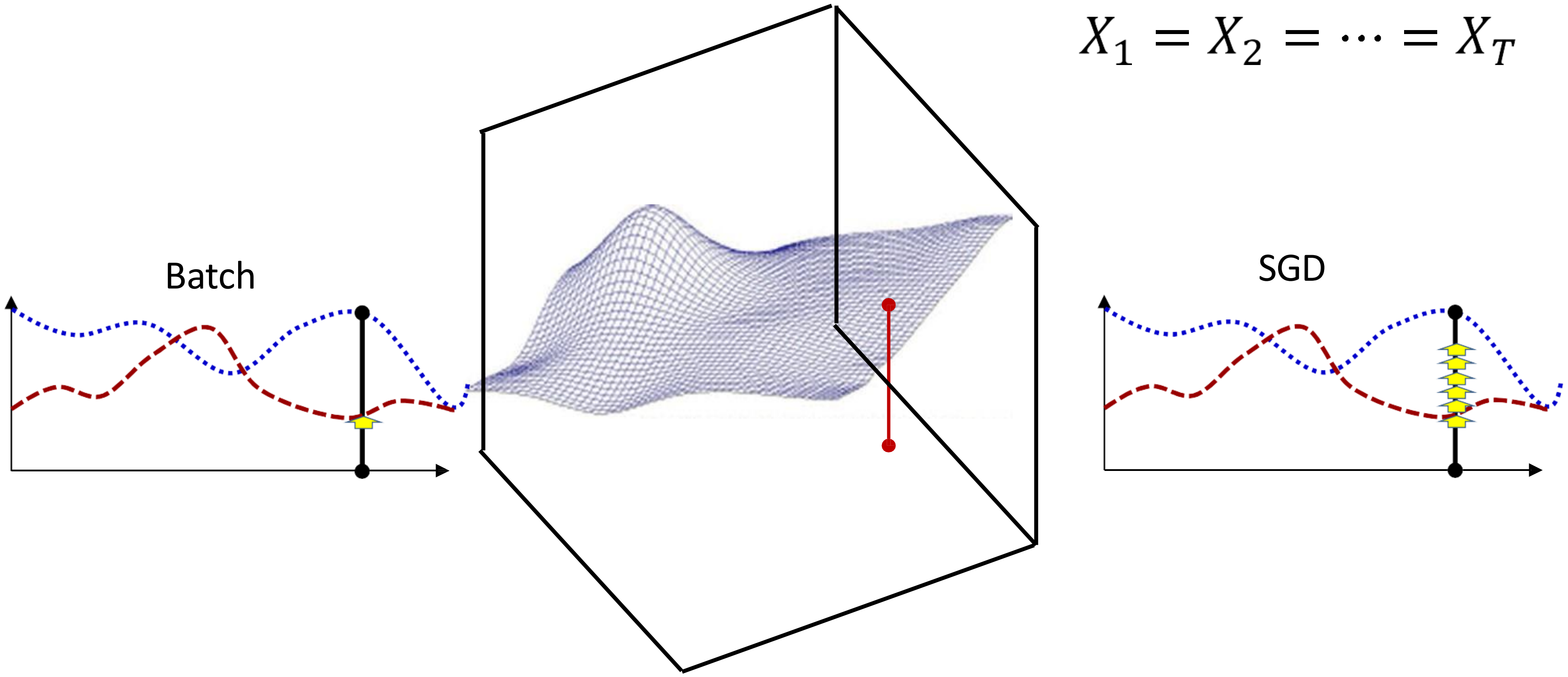


$$X_1 = X_2 = \cdots = X_T$$

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

Batch vs SGD

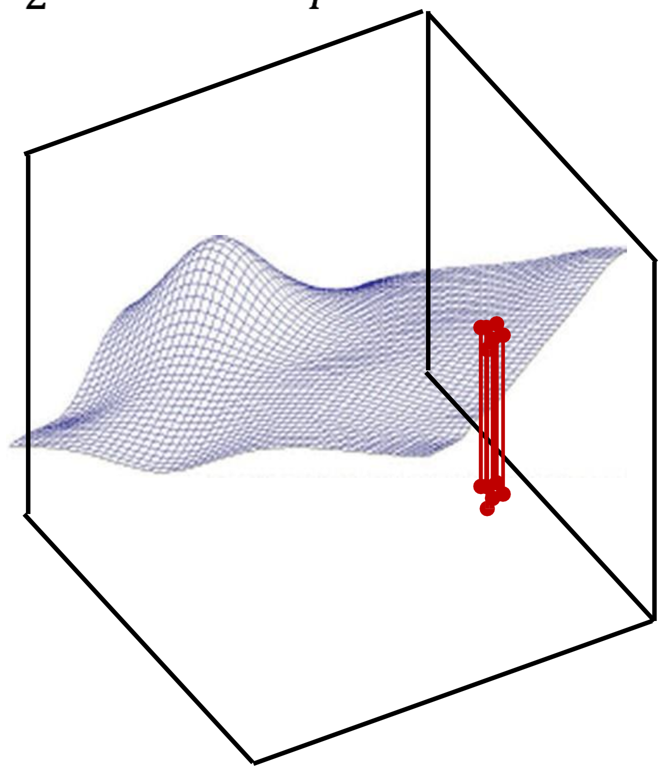
$$X_1 = X_2 = \cdots = X_T$$

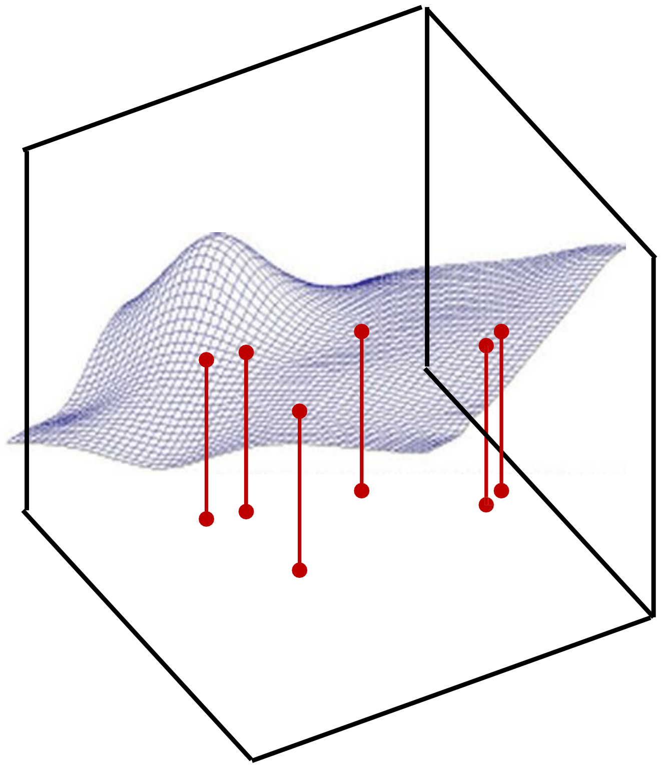


- Batch gradient descent operates over *T* training instances to get a *single* update
- SGD gets *T* updates for the same computation

Clumpy data..

$X_1 \approx X_2 \approx \cdots \approx X_T$





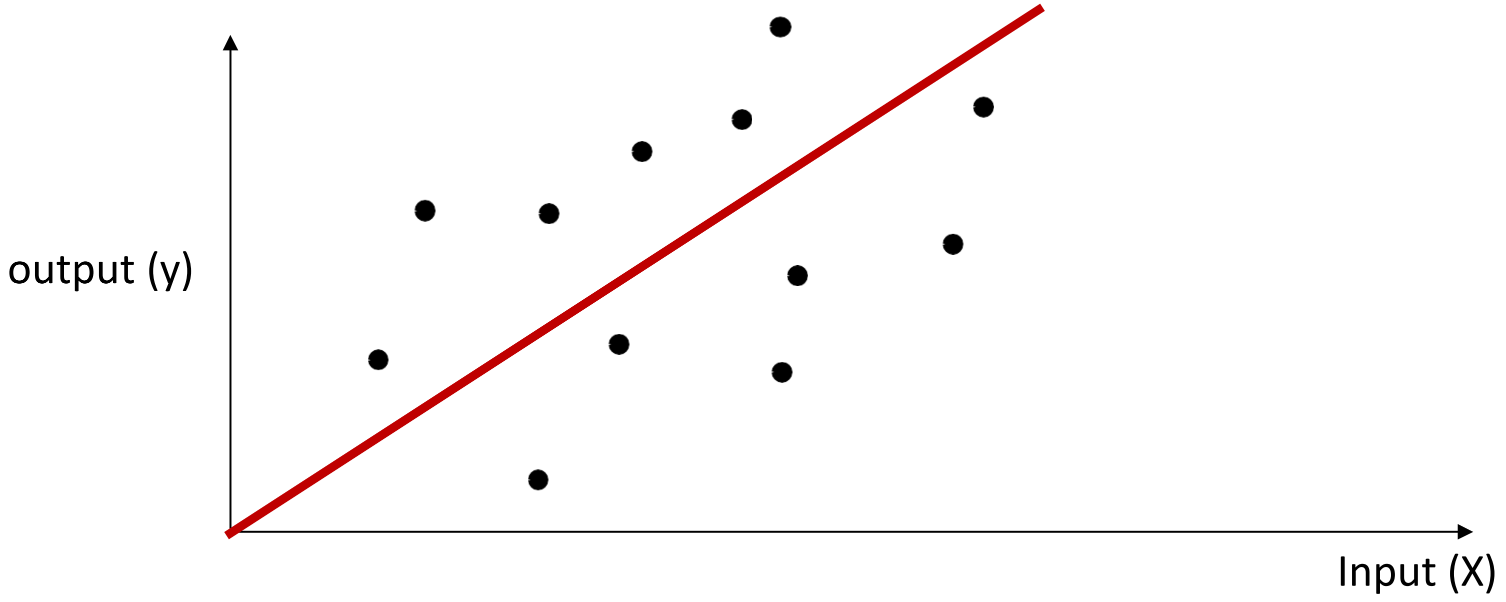
- Also holds if all the data are not identical, but are tightly clumped together
- As data get increasingly diverse, the benefits of incremental updates from this perspective decrease, but do not entirely vanish

When does it work

- What are the considerations?

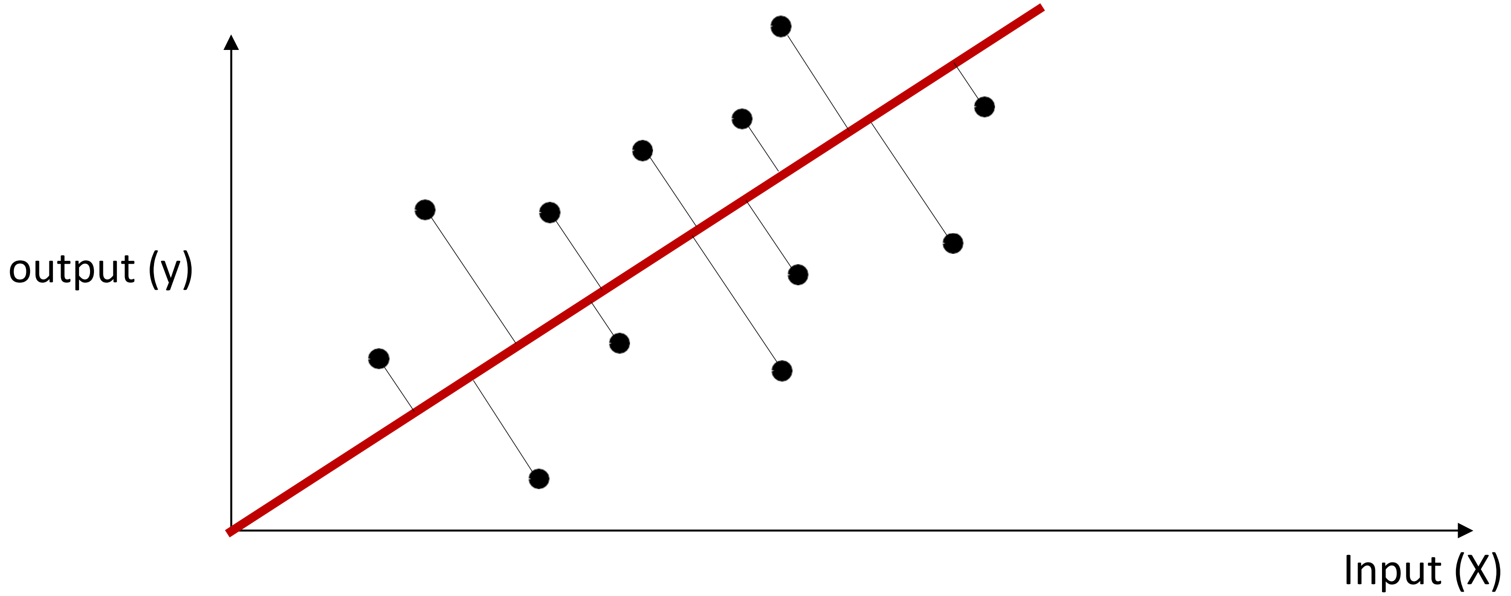
- And how well does it work?

Incremental learning runs the risk of always chasing the latest input



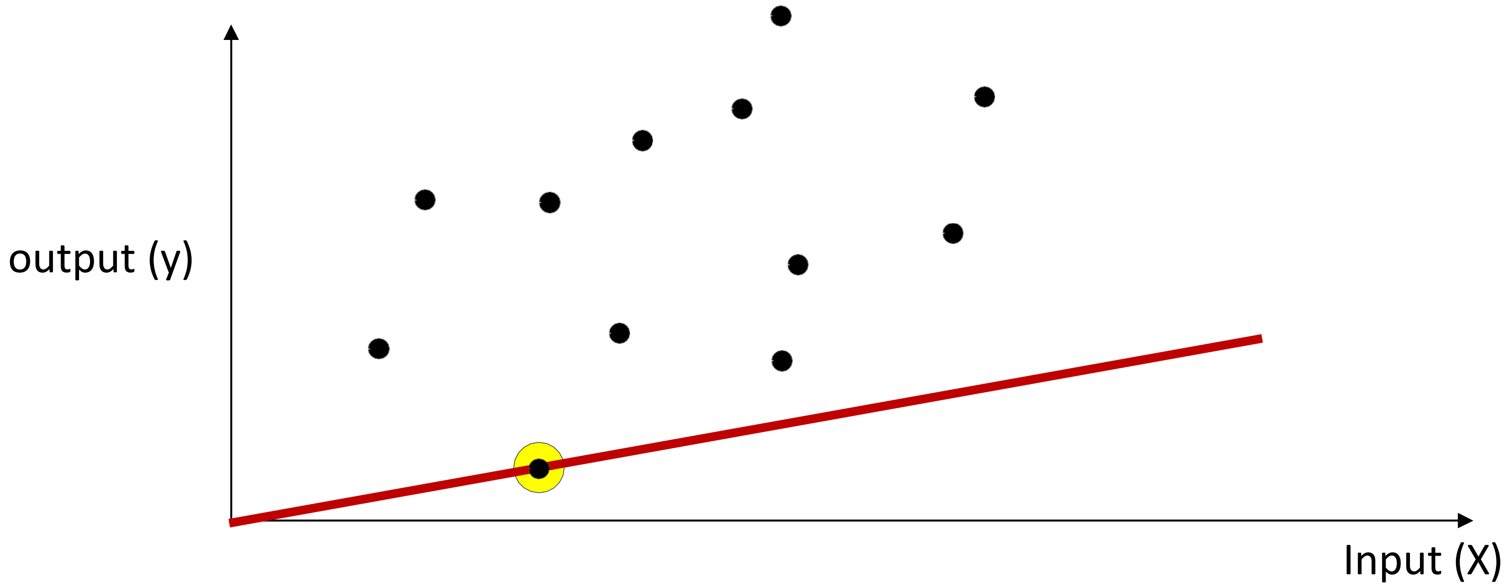
- **Modelling problem:** Find a linear regression line (through origin) to model the data

Incremental learning runs the risk of always chasing the latest input

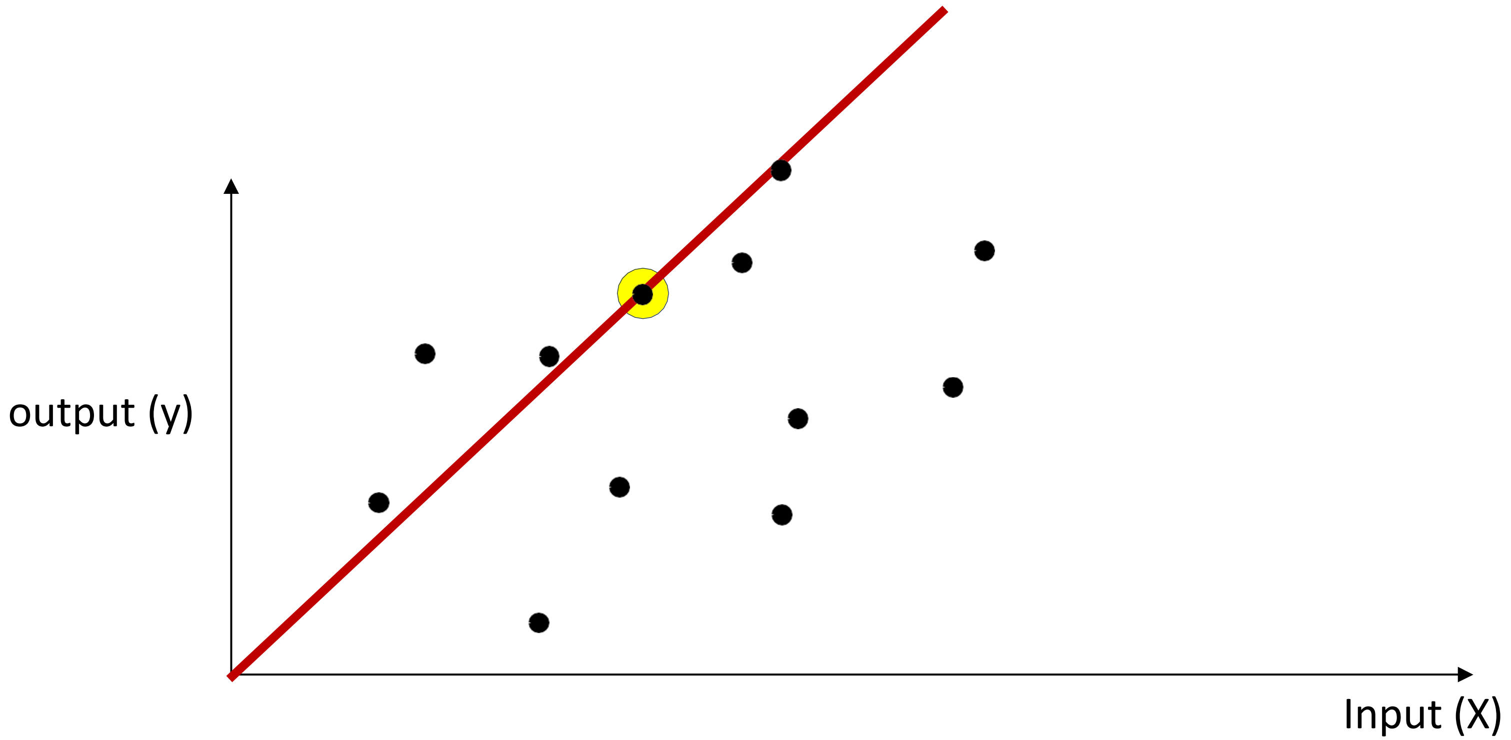


- **Modelling problem:** Find a linear regression line (through origin) to model the data
  - **Batch processing:** Find the line through origin that has the lowest overall squared projection error w.r.t. data

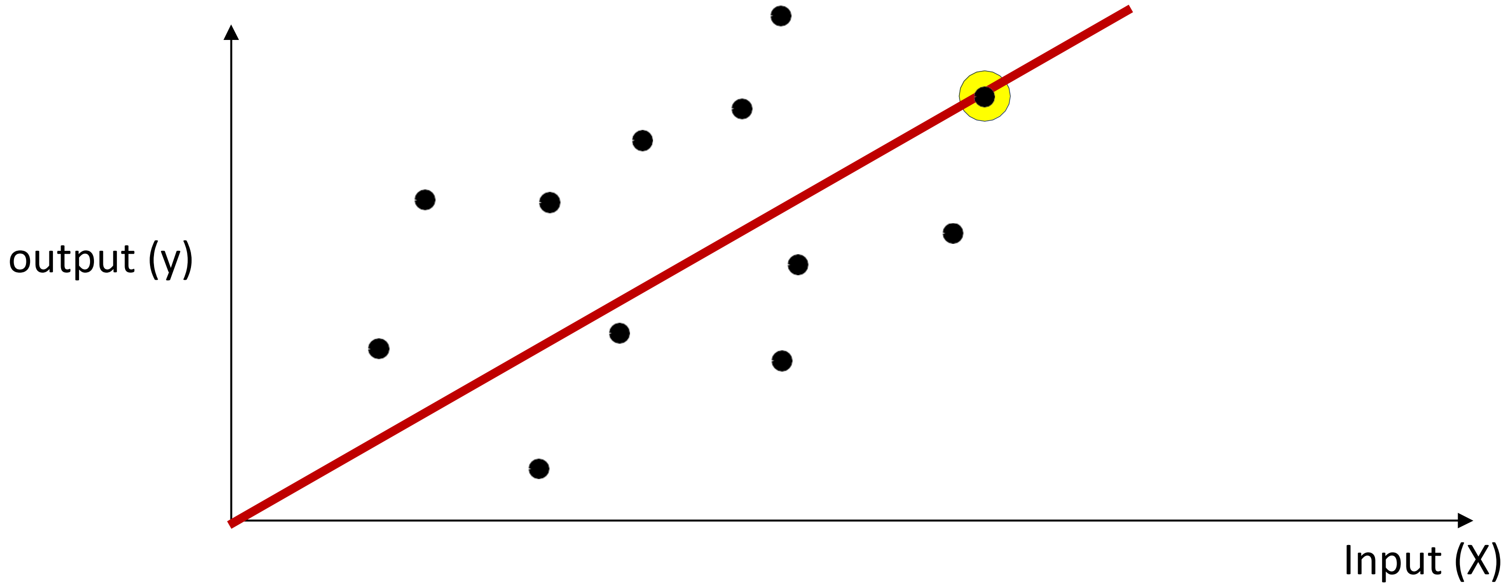
Incremental learning runs the risk of always chasing the latest input



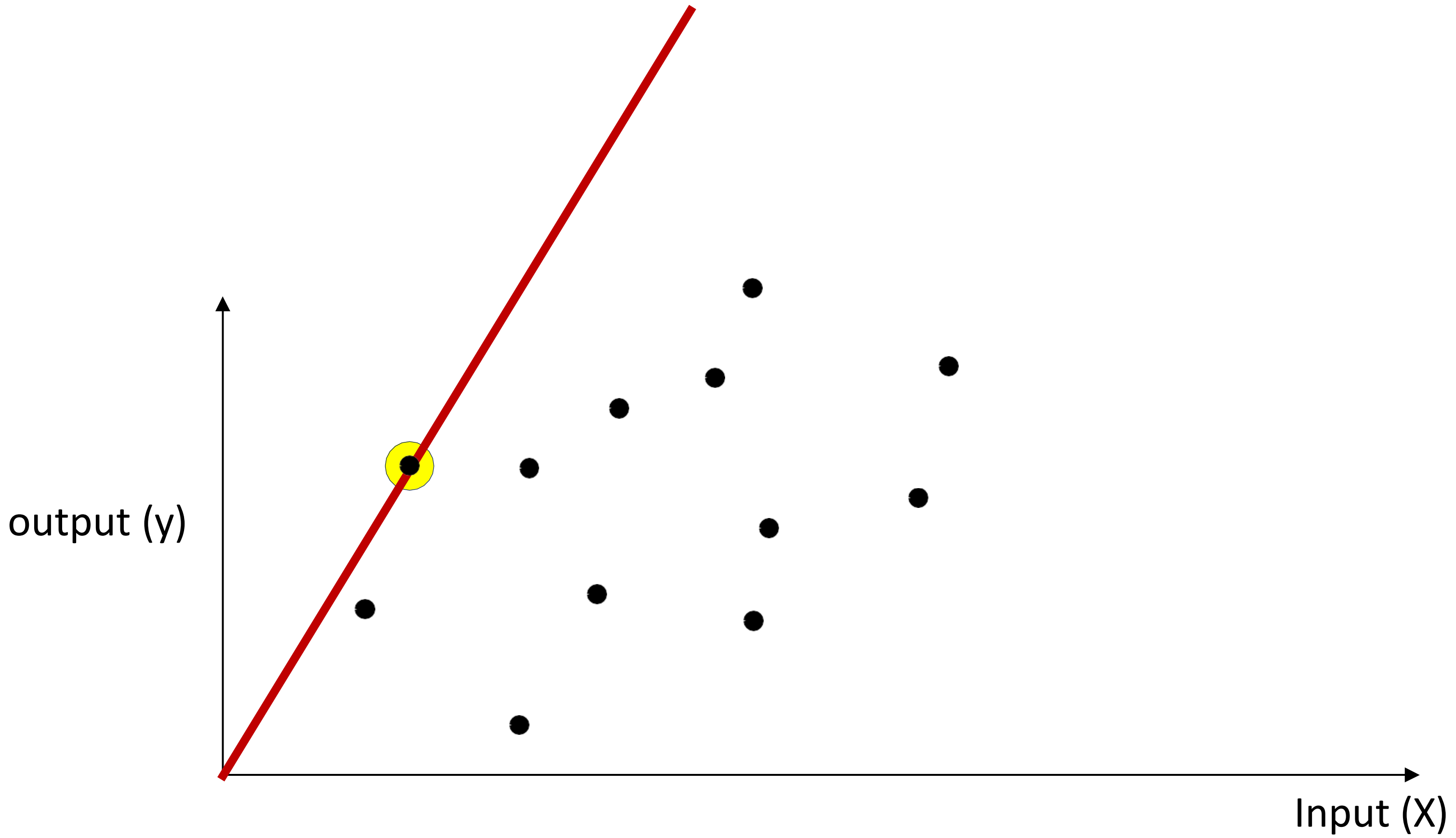
- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - It will never converge



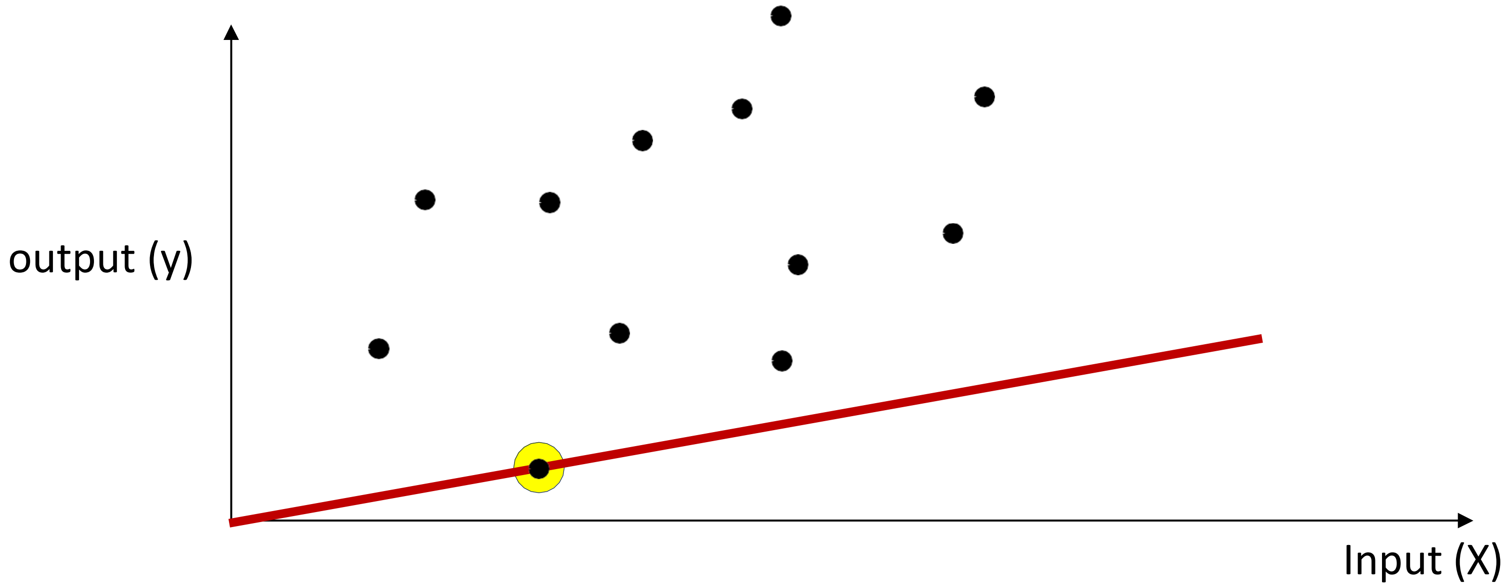
- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - It will never converge



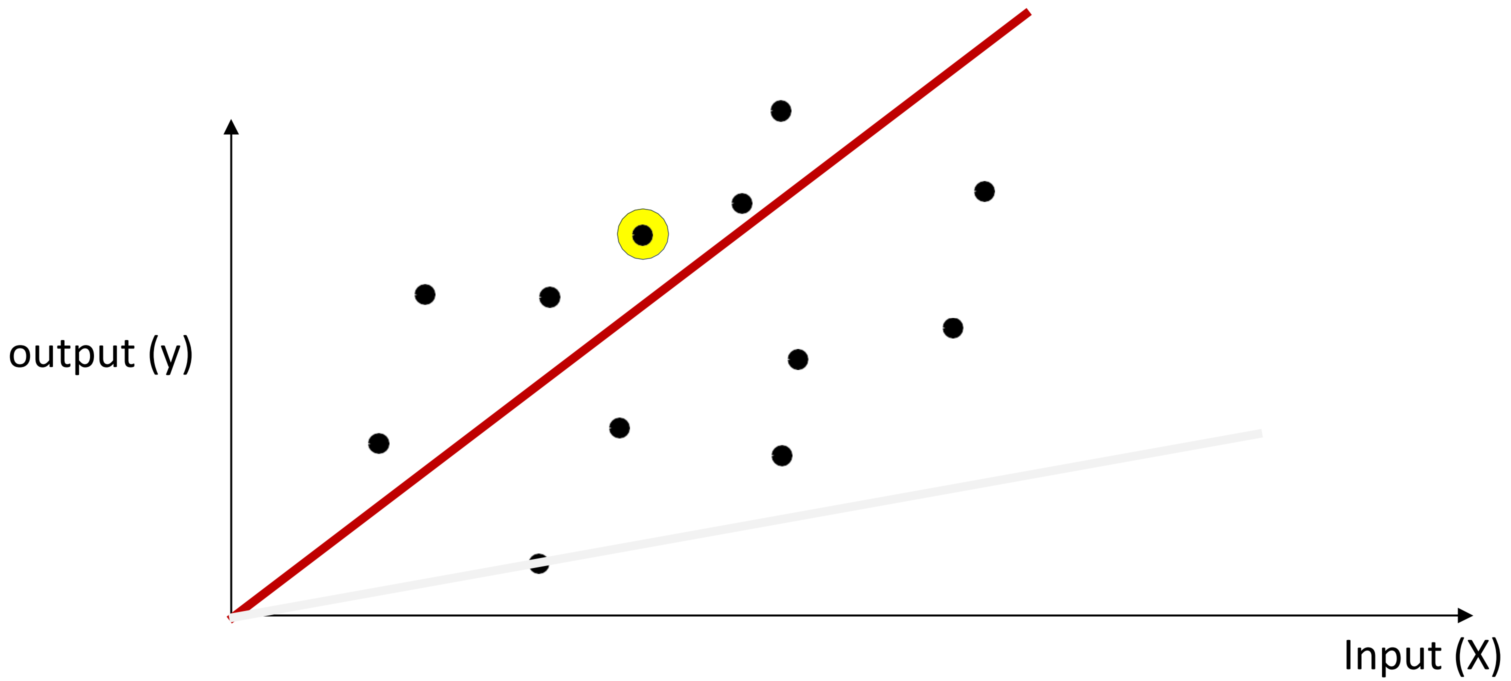
- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - It will never converge



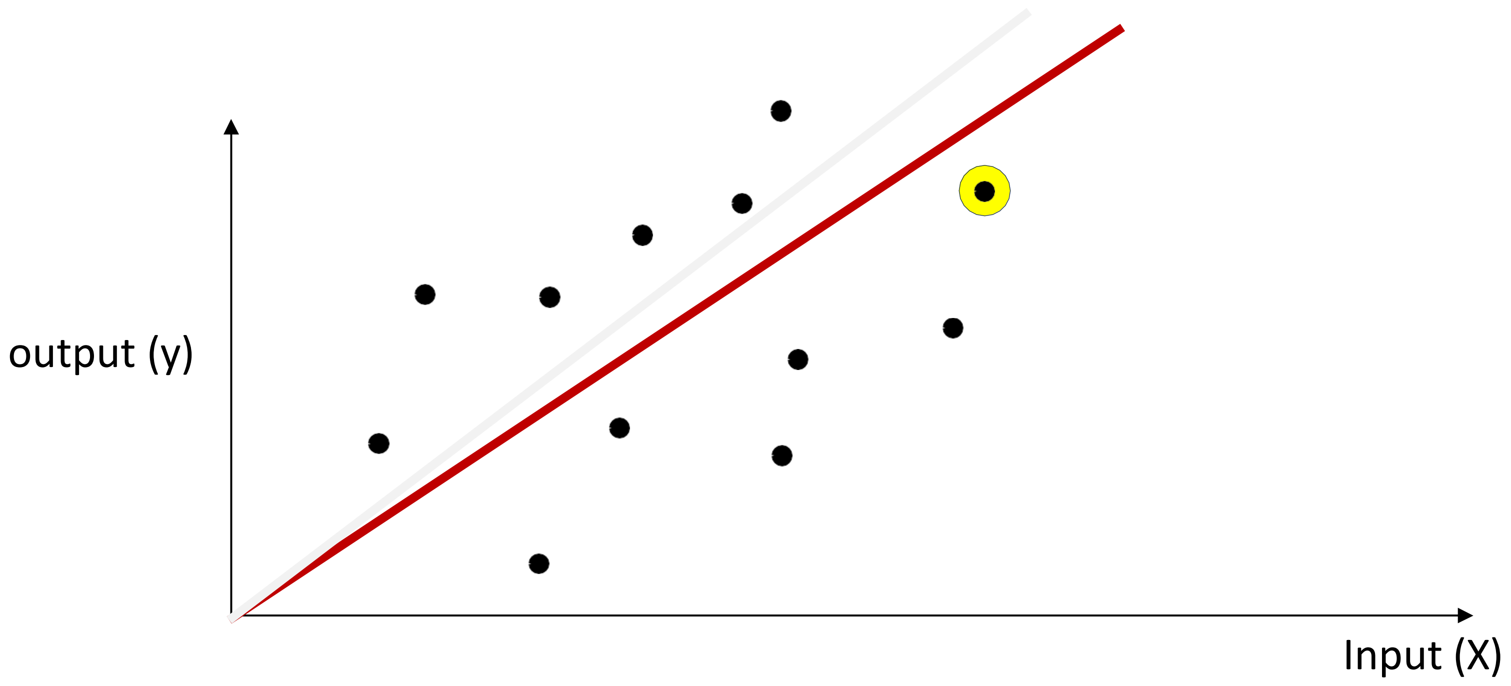
- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - It will never converge
  - **Solution?**
  - Shrink the learning rate with iterations
  - With increasing iterations, it will swing less and less towards the new point



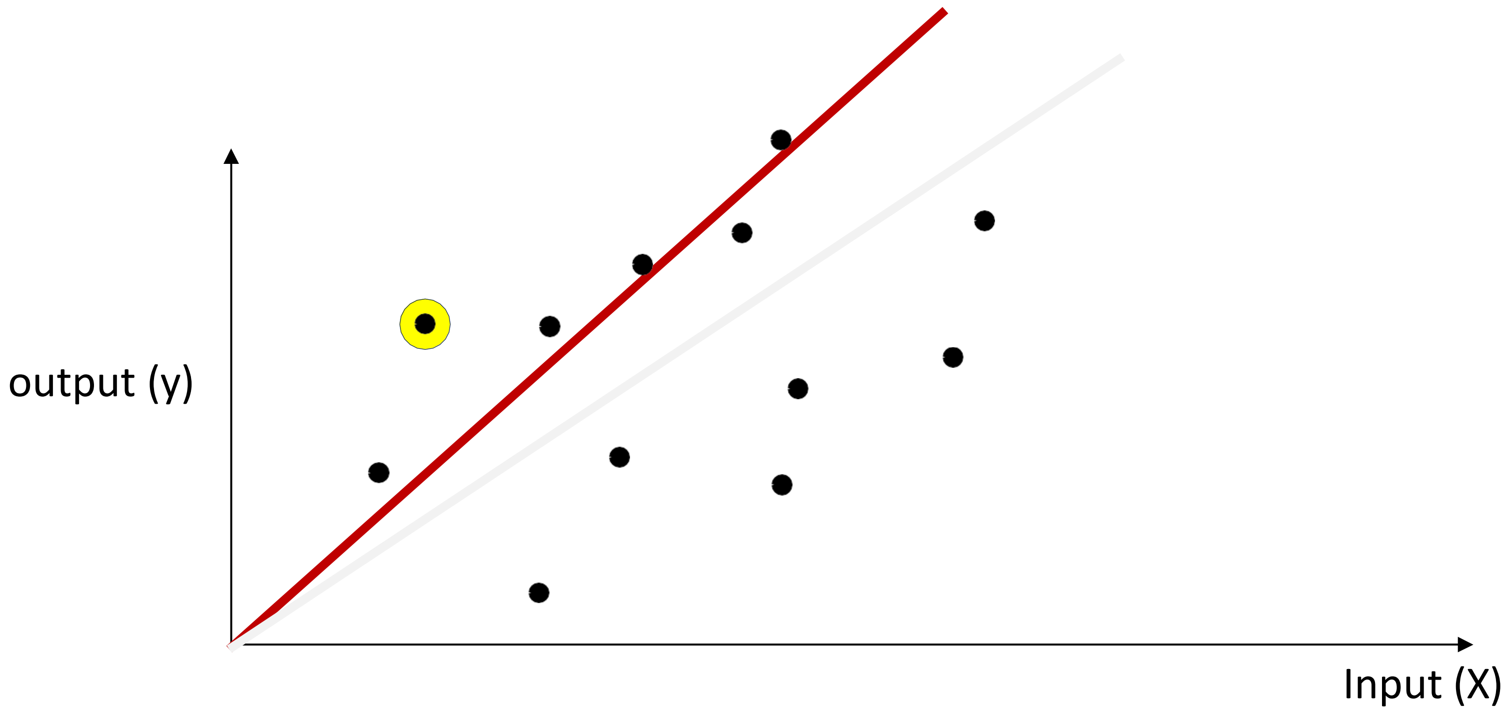
- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - Shrink the learning rate with iterations
  - With increasing iterations, it will swing less and less towards the new point



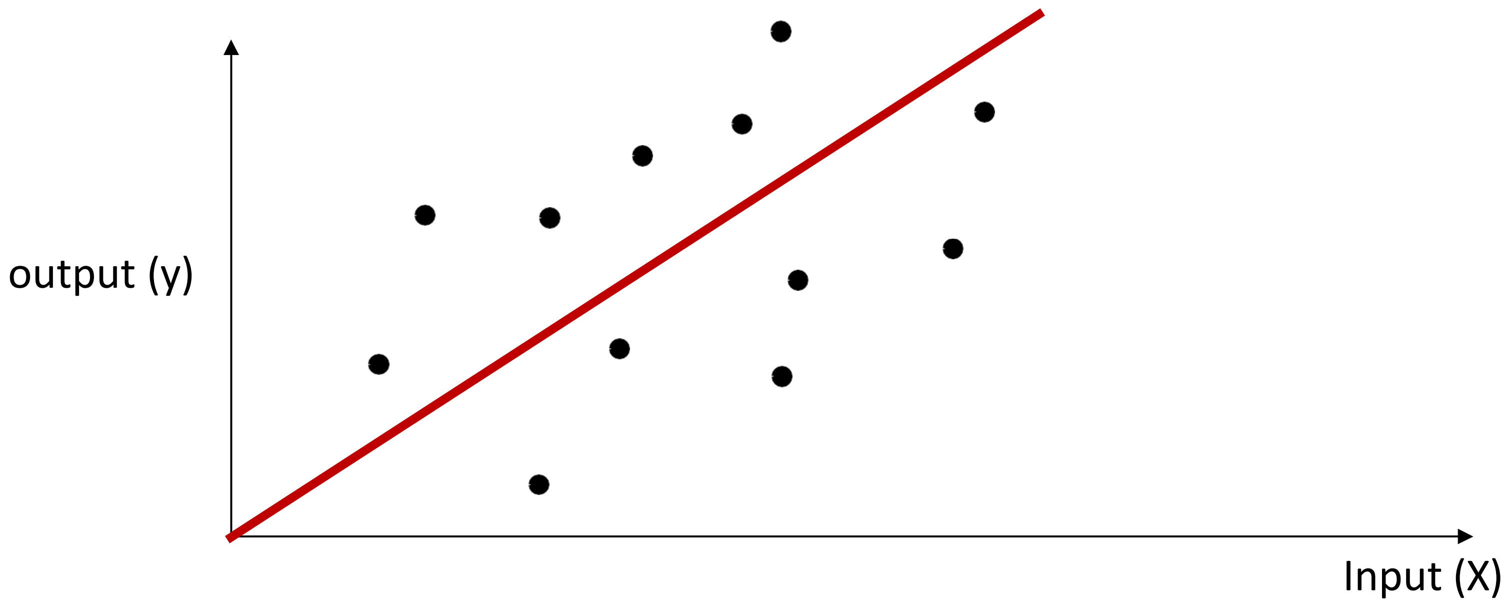
- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - Shrink the learning rate with iterations
  - With increasing iterations, it will swing less and less towards the new point



- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - Shrink the learning rate with iterations
  - With increasing iterations, it will swing less and less towards the new point



- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - Shrink the learning rate with iterations
  - With increasing iterations, it will swing less and less towards the new point



– **Eventually arriving at the correct solution and not moving much from it further because the step sizes are now too small…**

Incremental Update: Stochastic Gradient Descent

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

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

- Until $Loss$ has converged

Randomize input order

Learning rate reduces with j

We *shrink* the learning rate with iterations for convergence

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)
    - The sum of the step sizes over all iterations must diverge. Eventually the entire parameter space can be searched

$$\sum_k \eta_k = \infty$$

    - At the same time, the sum of squared step sizes must converge. Updates become sufficiently small to allow convergence, i.e., the steps shrink

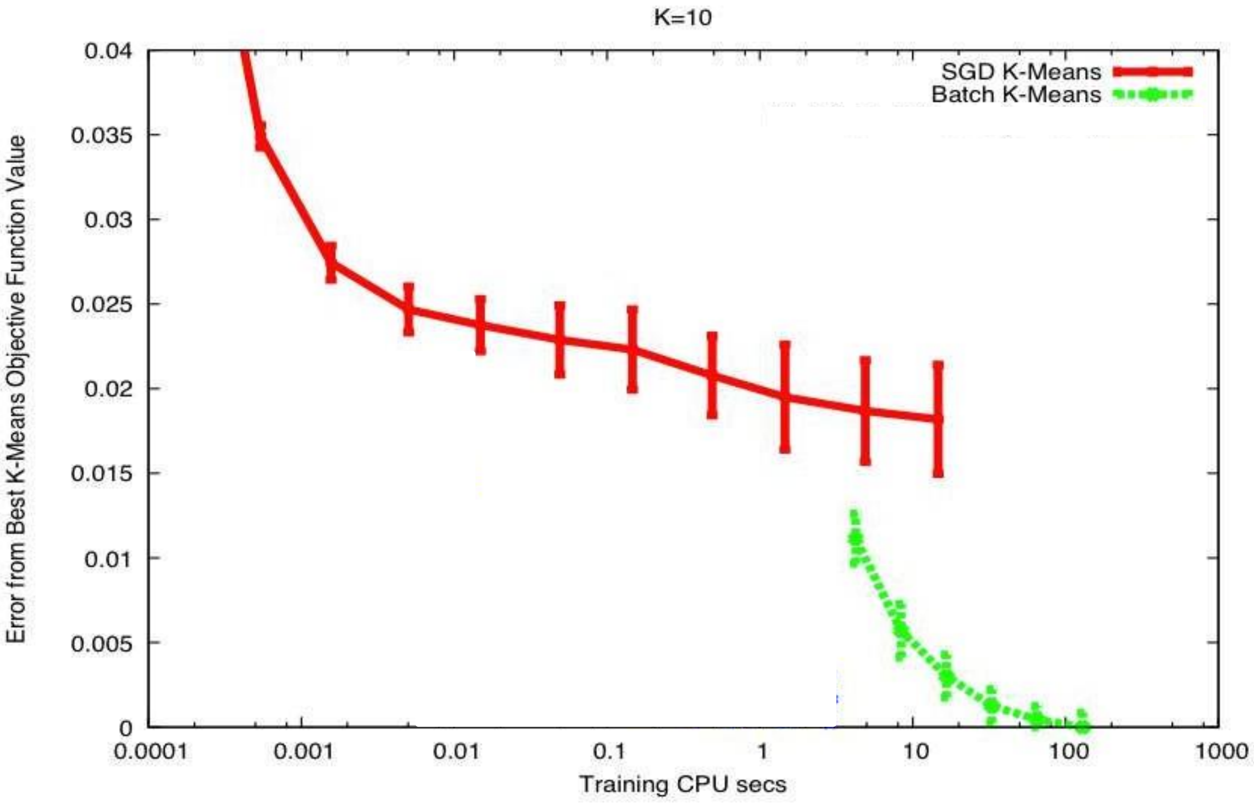
$$\sum_k \eta_k^2 < \infty$$

  - 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

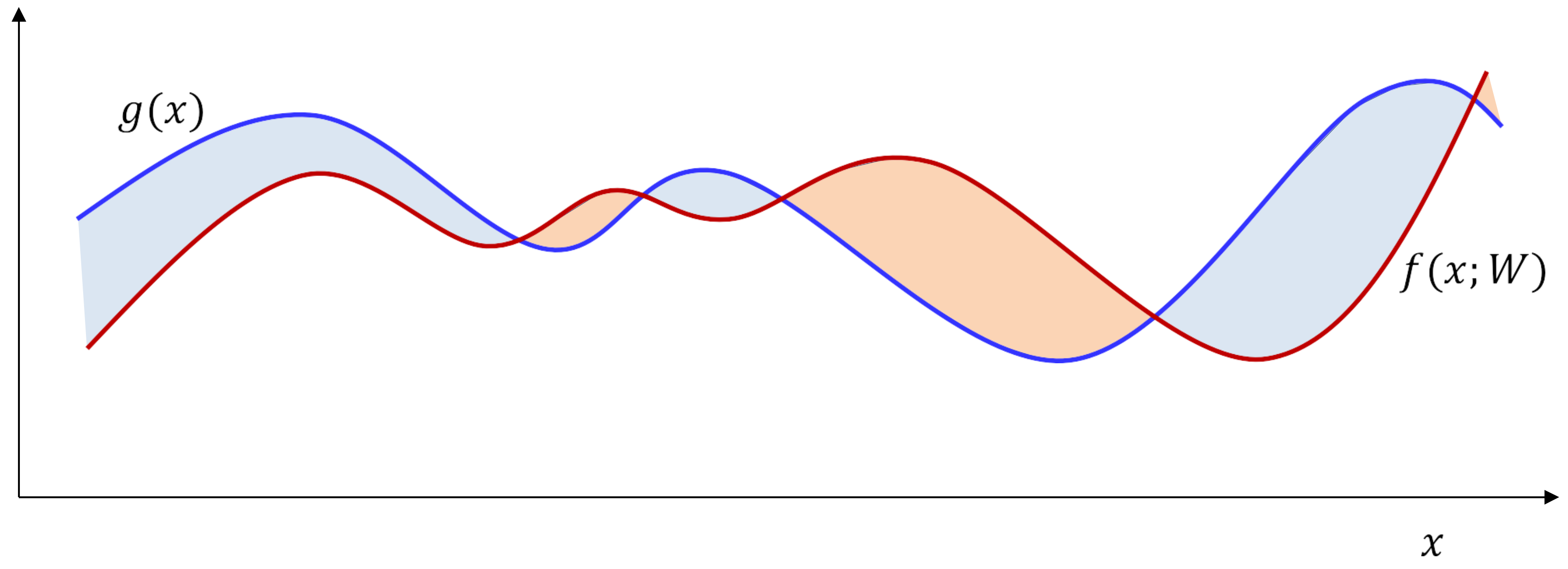
SGD example



- A simpler problem: K-means
- Note: SGD converges faster
  - But to a poorer minimum
- Also note the rather large variation between runs

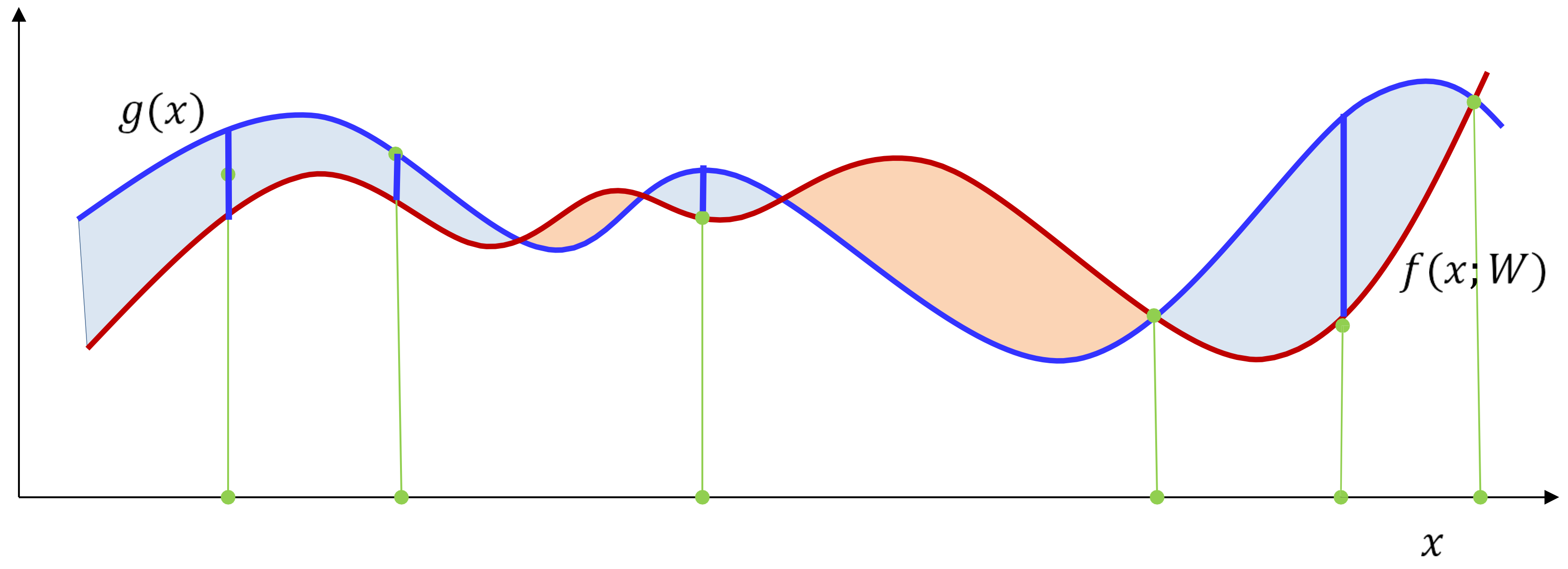
- **Let's try to understand these results..**

Explaining the variance

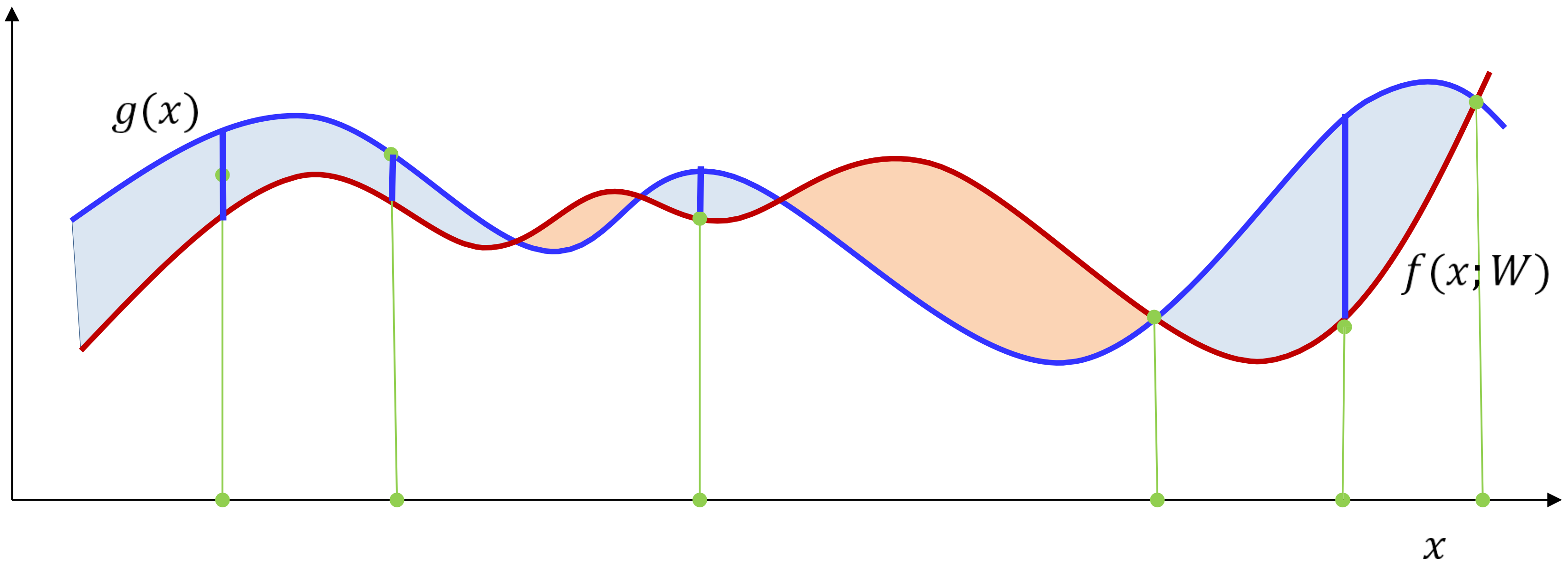


- The blue curve is the function being approximated
- The red curve is the approximation by the model at a given W
- The heights of the shaded regions represent the point-by-point error
  - The divergence is a function of the error
  - We want to find the W that minimizes the average divergence

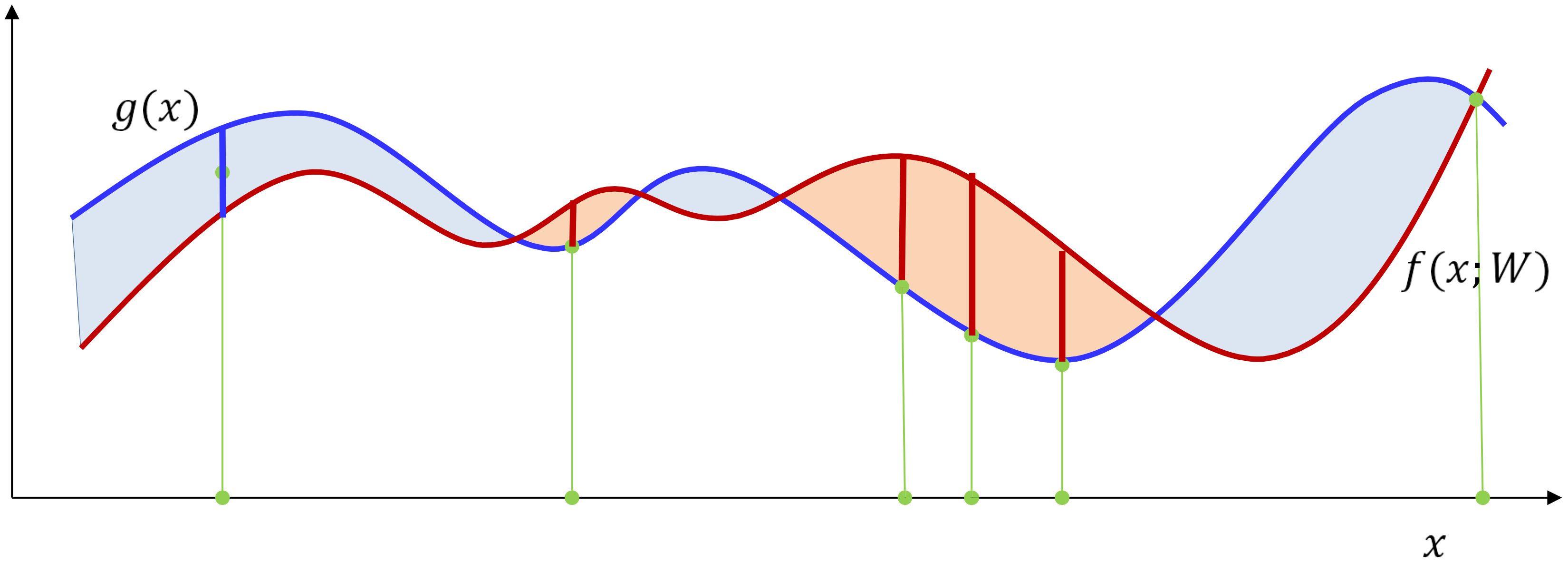
Explaining the variance



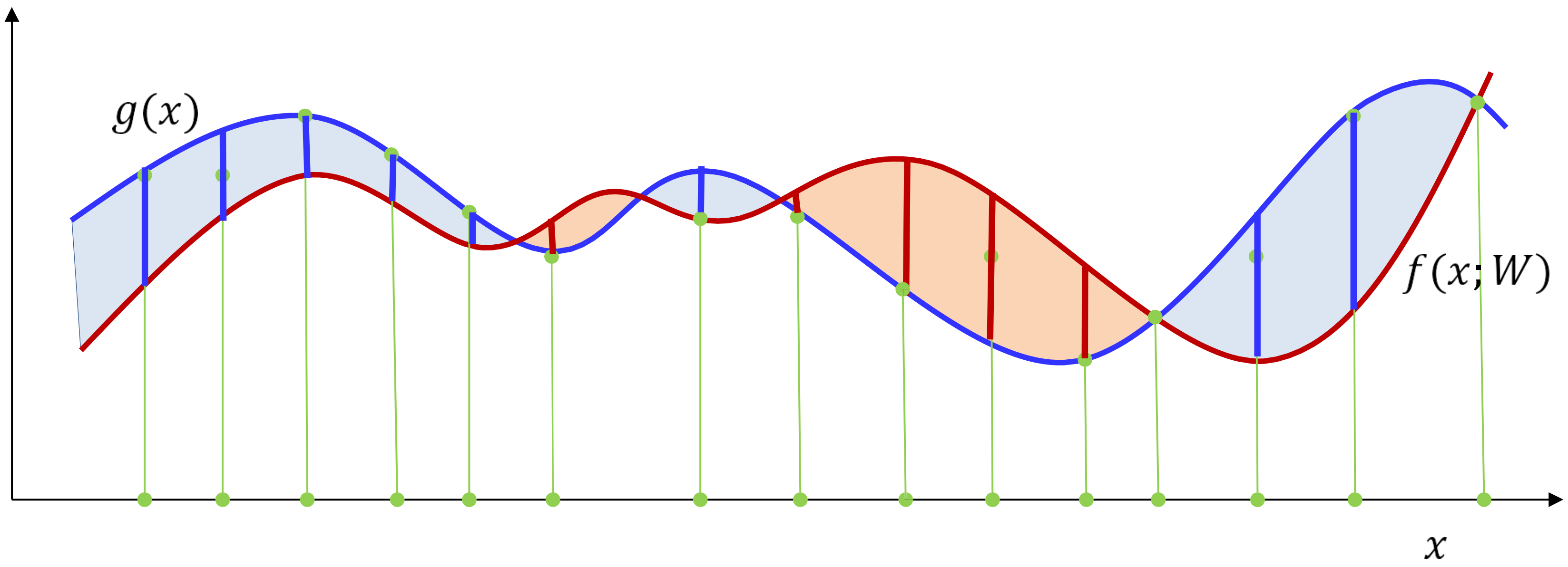
- Sample estimate approximates the shaded area with the average length of the error lines



This average length will change with position of the samples

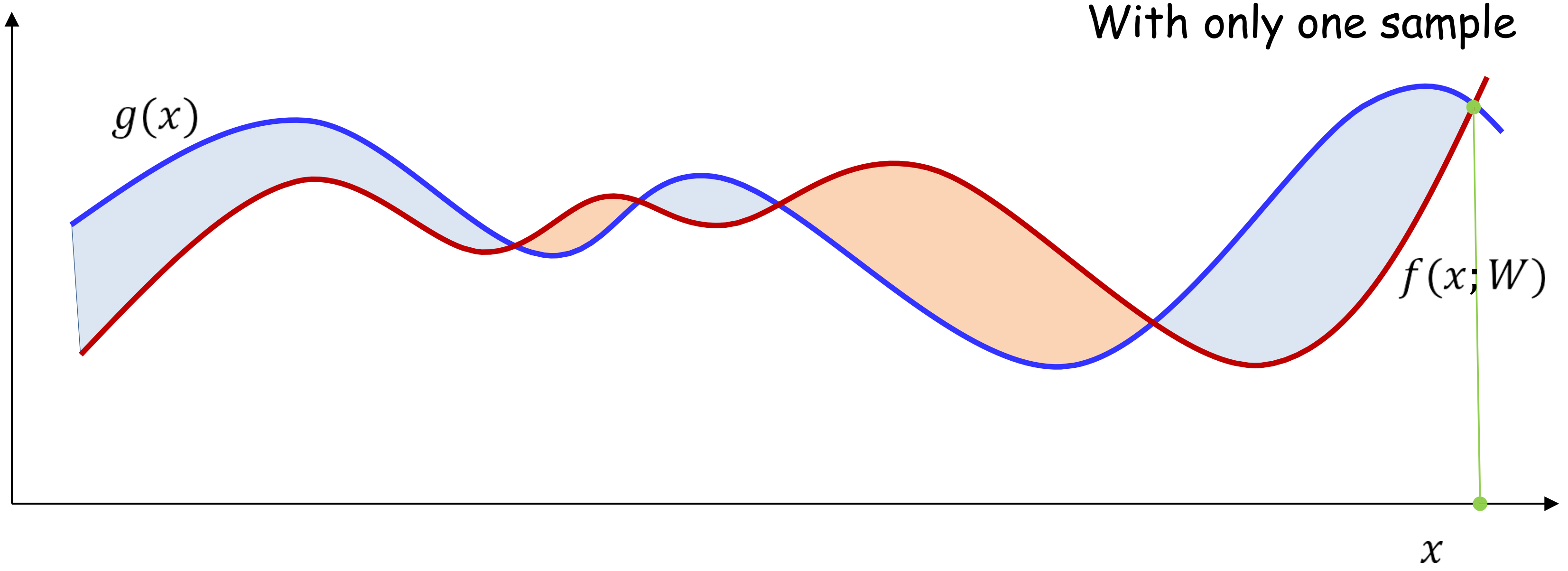


Explaining the variance



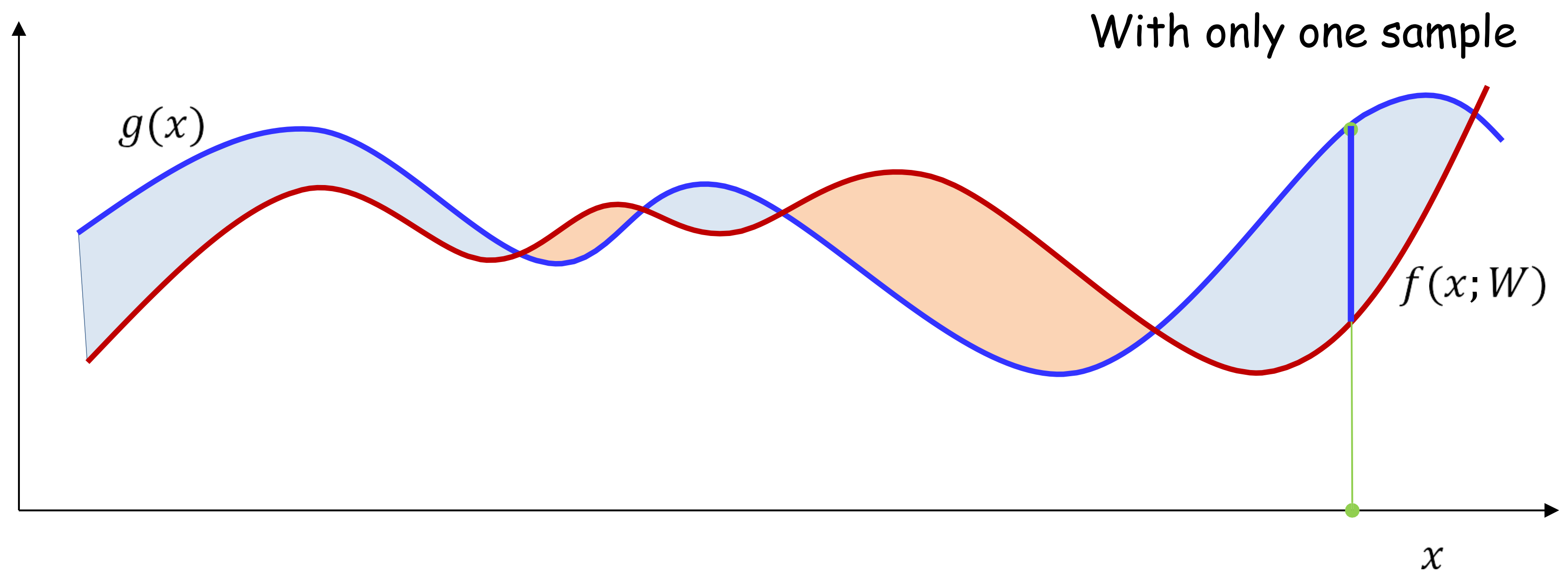
- Having more samples makes the estimate more robust to changes in the position of samples
  - The variance of the estimate is smaller

Explaining the variance



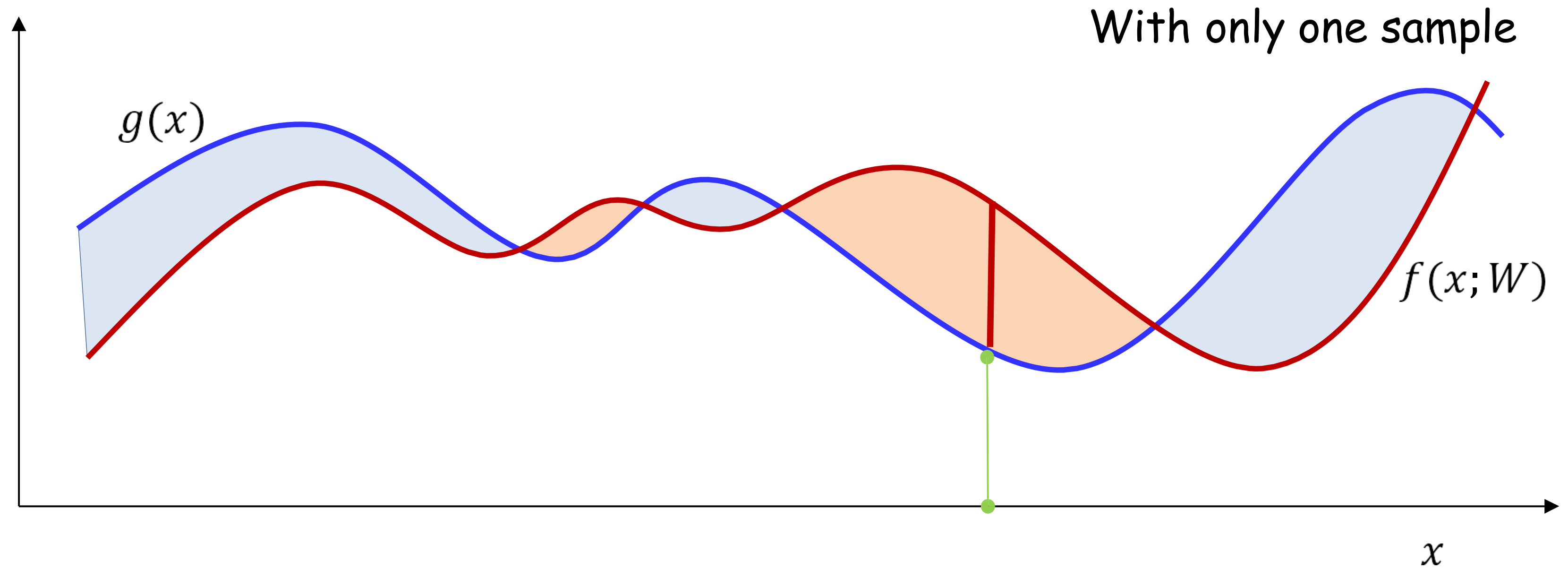
- Having very few samples makes the estimate swing wildly with the sample position
  - Since our estimator learns the W to minimize this estimate, the learned W too can swing wildly

Explaining the variance



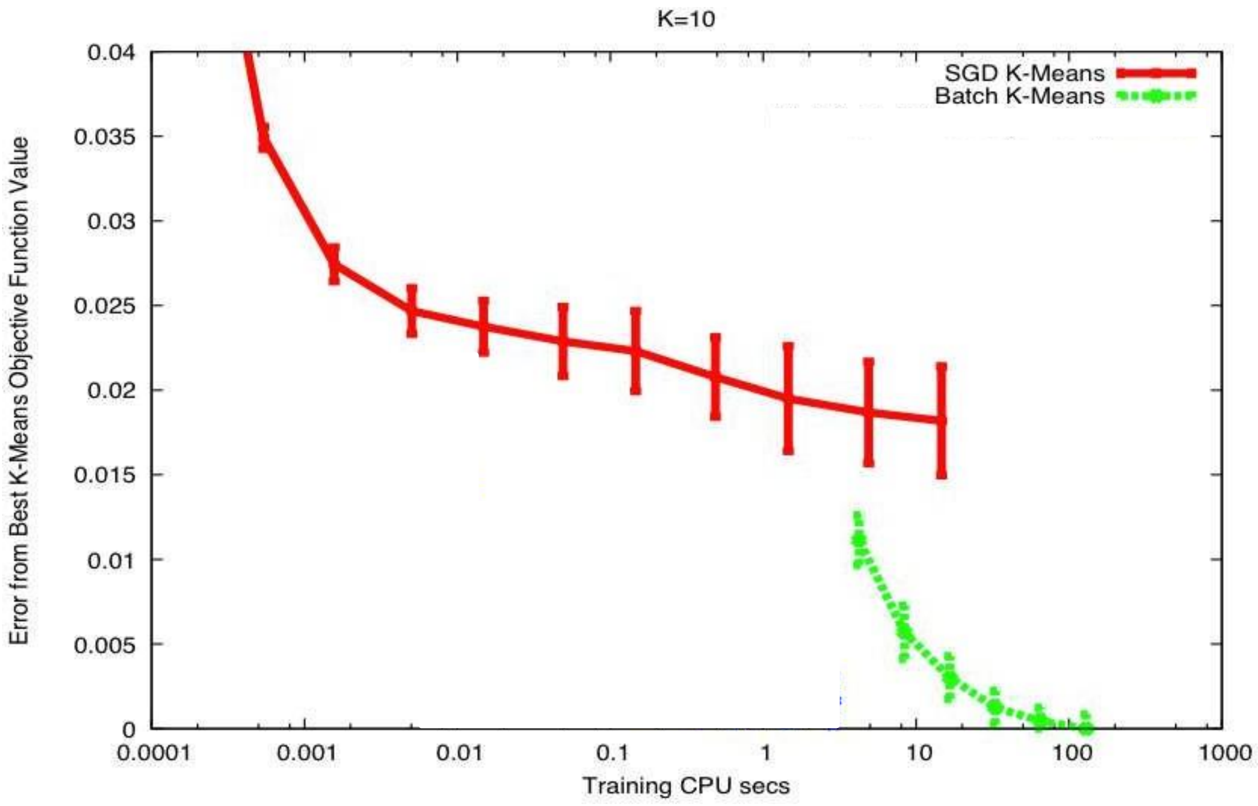
- Having very few samples makes the estimate swing wildly with the sample position
  - Since our estimator learns the W to minimize this estimate, the learned W too can swing wildly

Explaining the variance



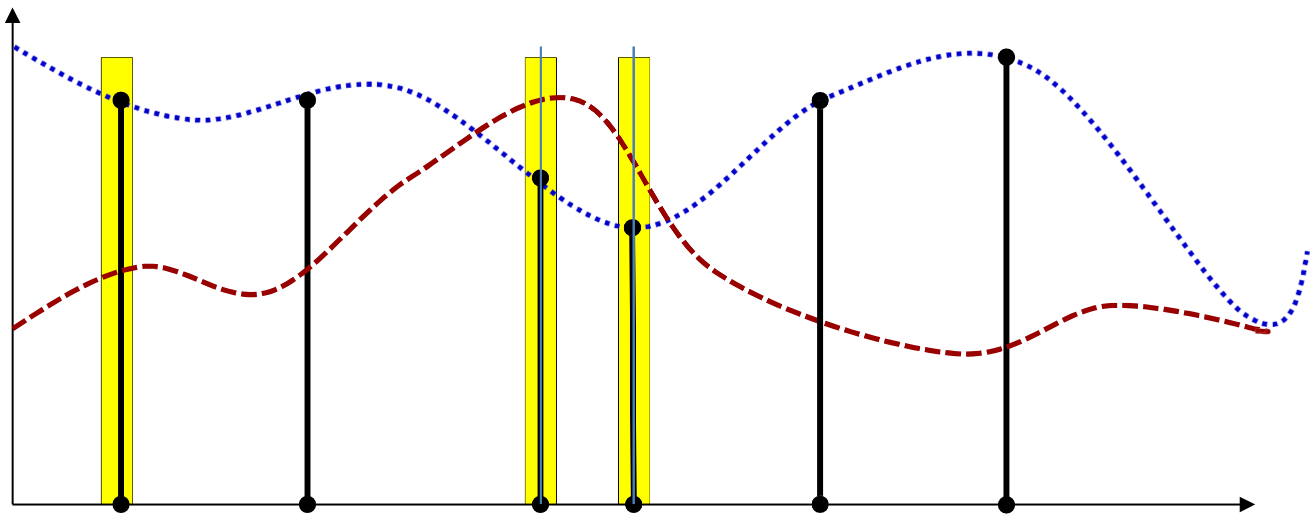
- Having very few samples makes the estimate swing wildly with the sample position
  - Since our estimator learns the W to minimize this estimate, the learned W too can swing wildly

SGD example

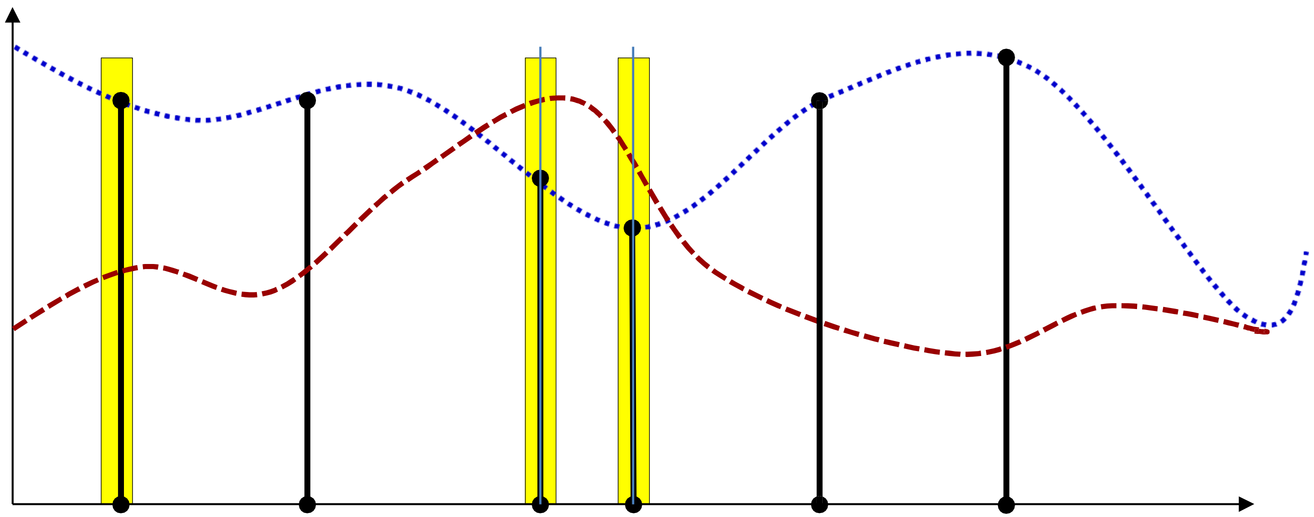


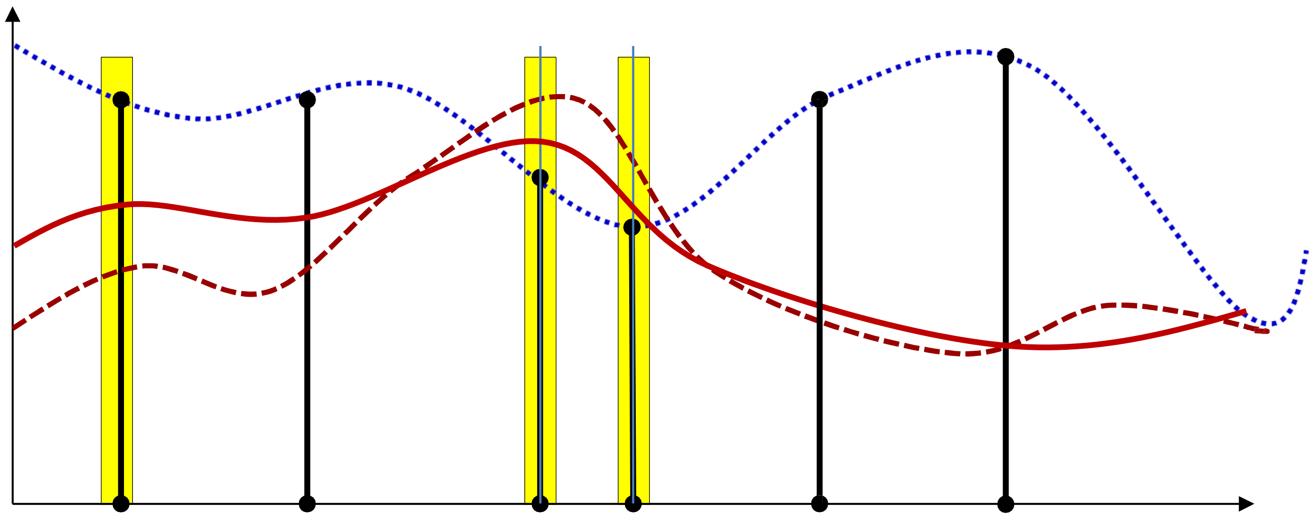
- SGD uses the gradient from only one sample at a time, and is consequently high variance
- But also provides significantly quicker updates than batch
- Is there a good medium?

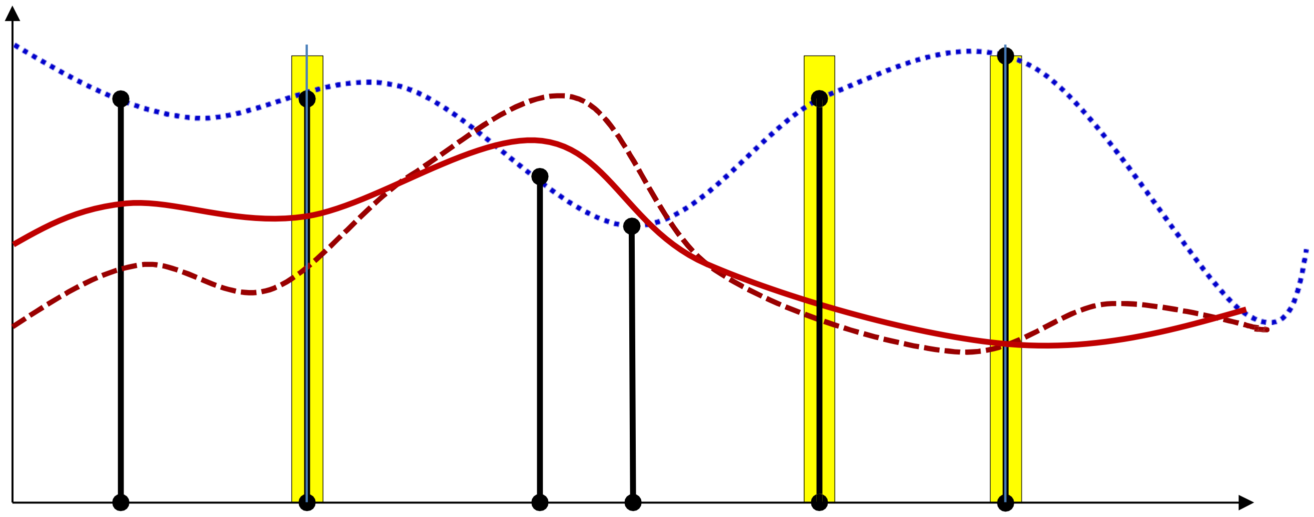
Mini-batch update

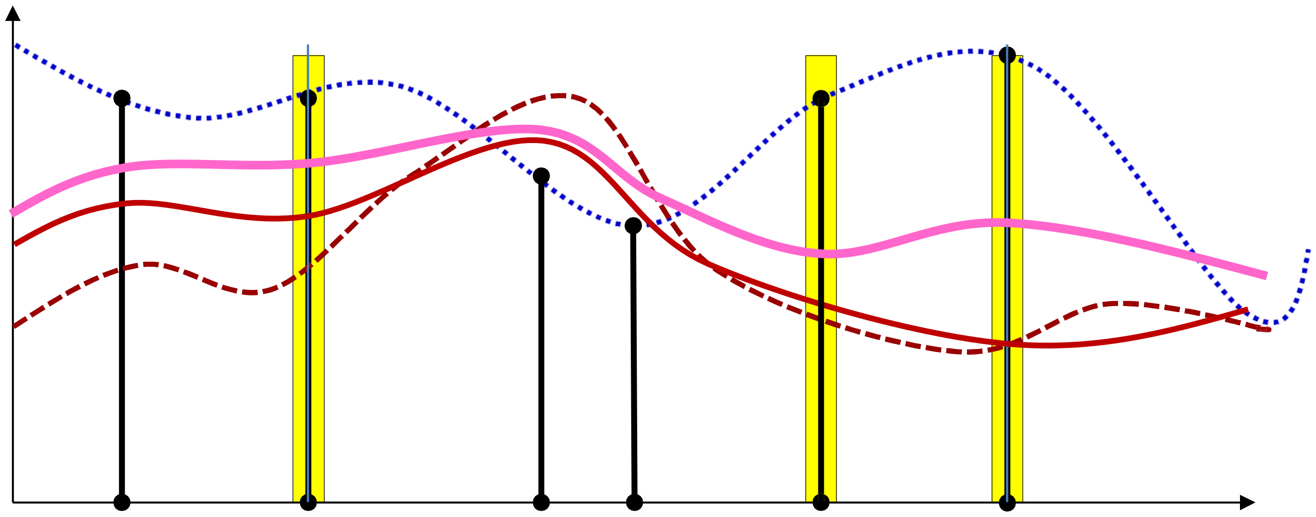


- Alternative: adjust the function at a small, randomly chosen subset of points
  - Keep adjustments small
  - If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data









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:

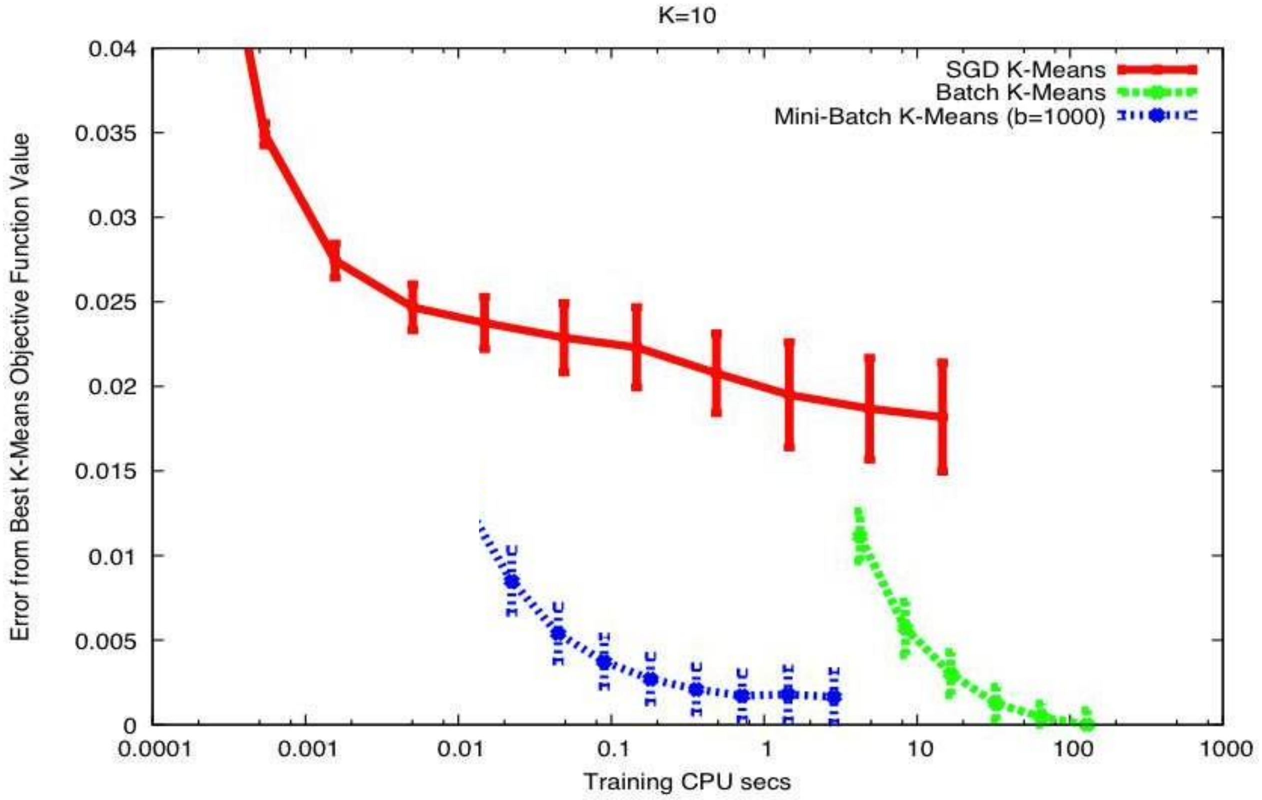
$$W_k = W_k - \eta_j \Delta W_k$$

- Until Err has converged

Mini-batch size

Shrinking step size

SGD example



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

-- And with smaller variance

Minibatch Convergence

- For convex functions, convergence rate for SGD is $\mathcal{O}\left(\frac{1}{\sqrt{k}}\right)$.

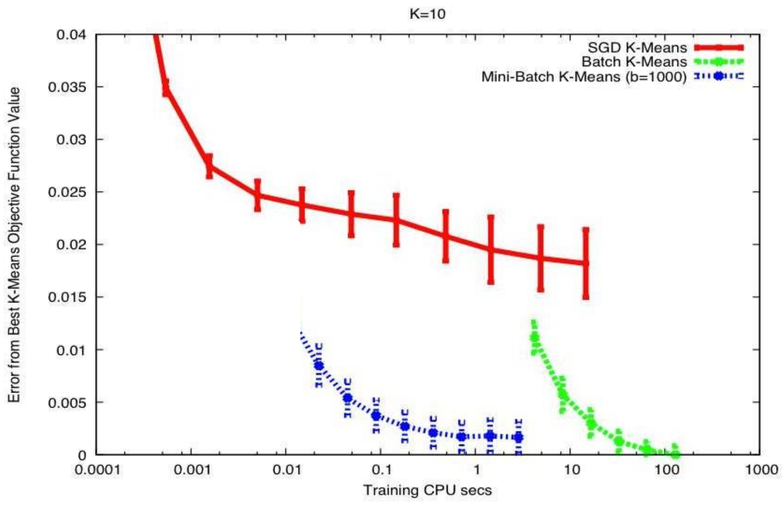
k refers to the number of iterations

- 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

- 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

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

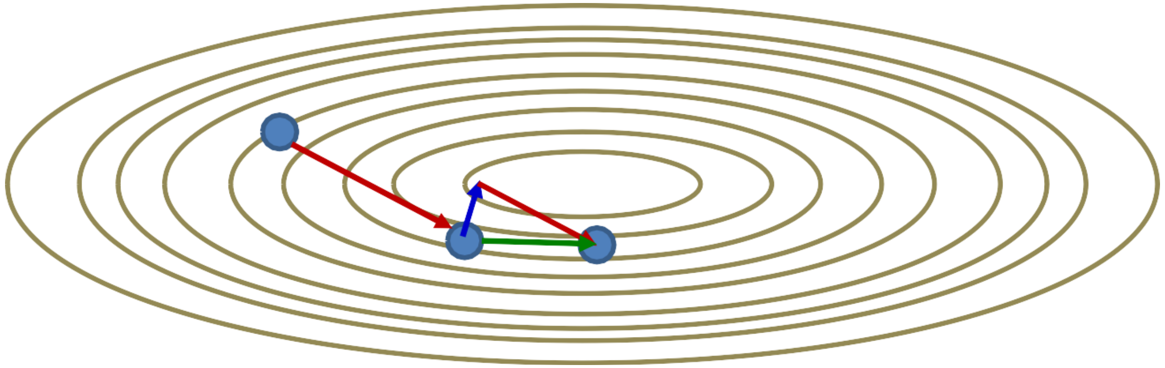
- More typically, we estimate is as
  - Divergence or classification error on a held-out set
  - Average sample/batch loss over the past N samples/batches

Training with minibatches

- In practice, training is usually performed using mini-batches
  - The mini-batch size is generally set to the largest that your hardware will support (in memory) without compromising overall compute time
    - Larger minibatches = less variance
    - Larger minibatches = few updates per epoch

- Convergence depends on step size
  - 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 (or step size) is itself determined as part of the estimation

Recall: Momentum Update

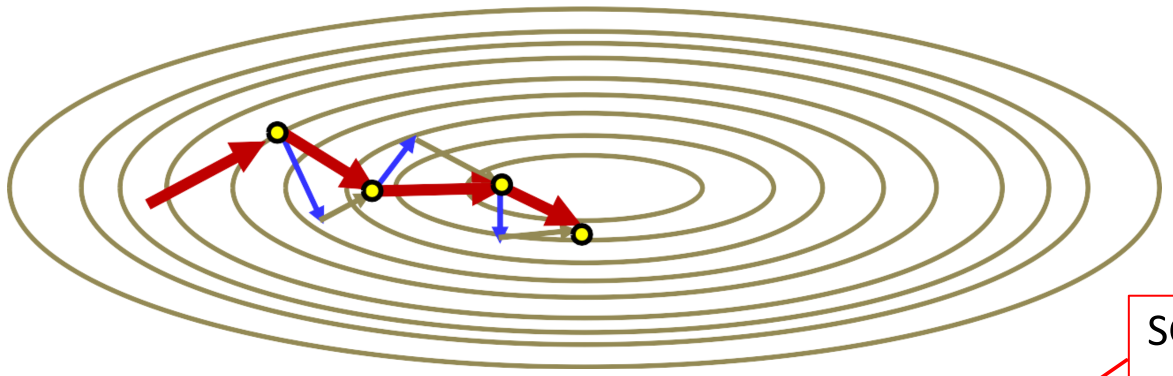


- The momentum method

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

- At any iteration, to compute the current step:
  – First compute the gradient step at the current location
  – Then add in the scaled *previous* step, which is actually a running average
  – To get the final step

Momentum and incremental updates



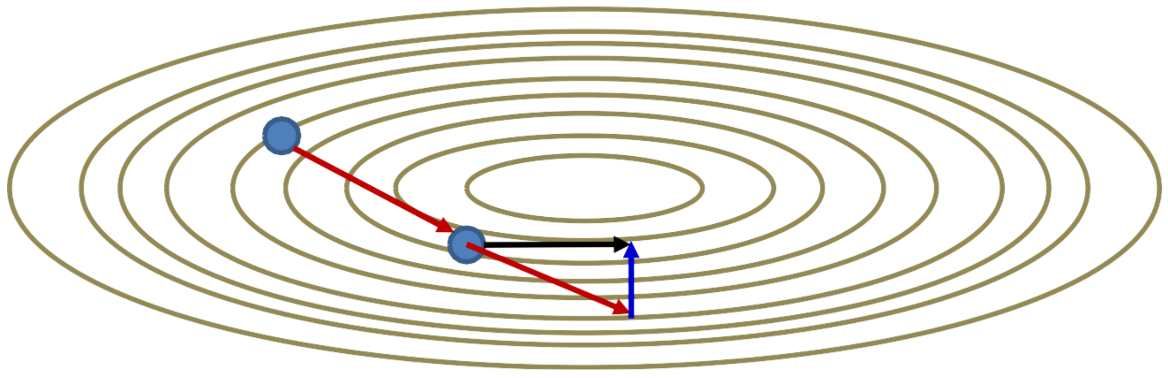
SGD instance or minibatch loss

The momentum method

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

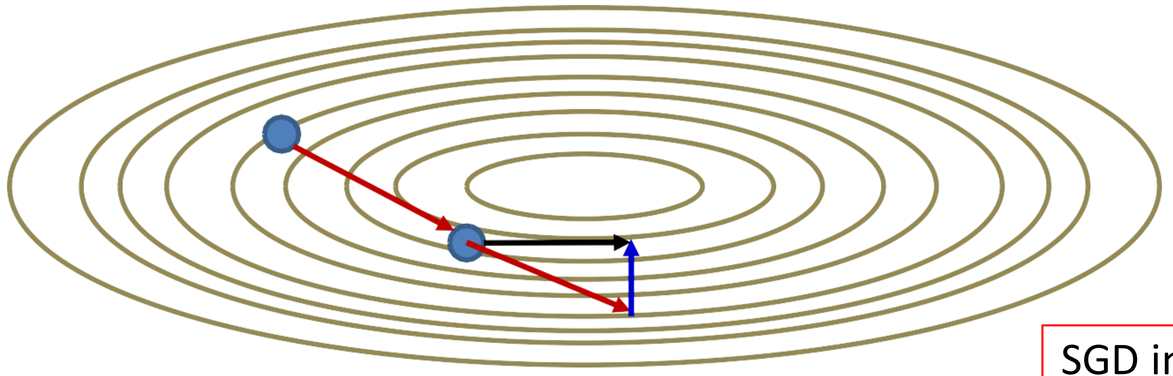
- Incremental SGD and mini-batch gradients tend to have high variance
- Momentum smooths out the variations
  - Smoother and faster convergence

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

Nestorov's Accelerated Gradient



SGD instance or minibatch loss

- Nestorov's method

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

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

- This also applies directly to incremental update methods
  - The accelerated gradient smooths out the variance in the gradients

Mini-batch update

Momentum

- Given (X_1, d_1), (X_2, d_2),…, (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),…, (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 $\nabla_{W_k} Div(Y_t, d_t)$
        - » $\nabla_{W_k} Loss += \frac{1}{b} \nabla_{W_k} \boldsymbol{Div}(Y_t, d_t)$
    - 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:

- Given (X_1, d_1), (X_2, d_2),…, (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),…, (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 $\nabla_{W_k} Div(Y_t, d_t)$
        - » $\nabla_{W_k} Loss += \frac{1}{b} \nabla_{W_k} \boldsymbol{Div}(Y_t, d_t)$
    - 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

Training with minibatches

- Standard gradient descent rule

$$W \leftarrow W - \eta \nabla_W L(W)$$

- Gradient descent invokes two terms for updates
  - The derivative
  - and the learning rate

Training with minibatches

- Standard gradient descent rule

$$W \leftarrow W - \eta \nabla_W L(W)$$

Momentum methods
fix this term to reduce
unstable oscillation

- Gradient descent invokes two terms for updates
  - The derivative
  - and the learning rate

Adjusting the learning rate

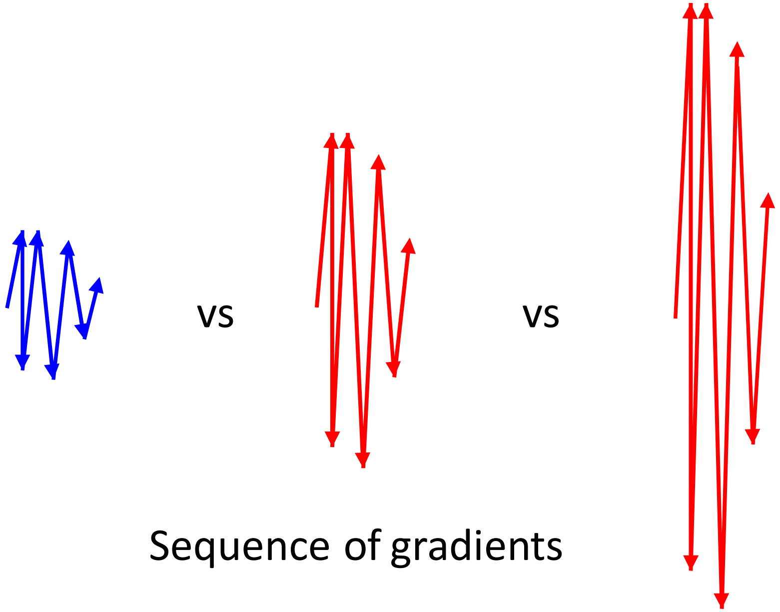
- Standard gradient descent rule

$$W \leftarrow W - \eta \nabla_W L(W)$$

What about this term?

- Gradient descent invokes two terms for updates
  - The derivative
  - and the learning rate

Adjusting the learning rate



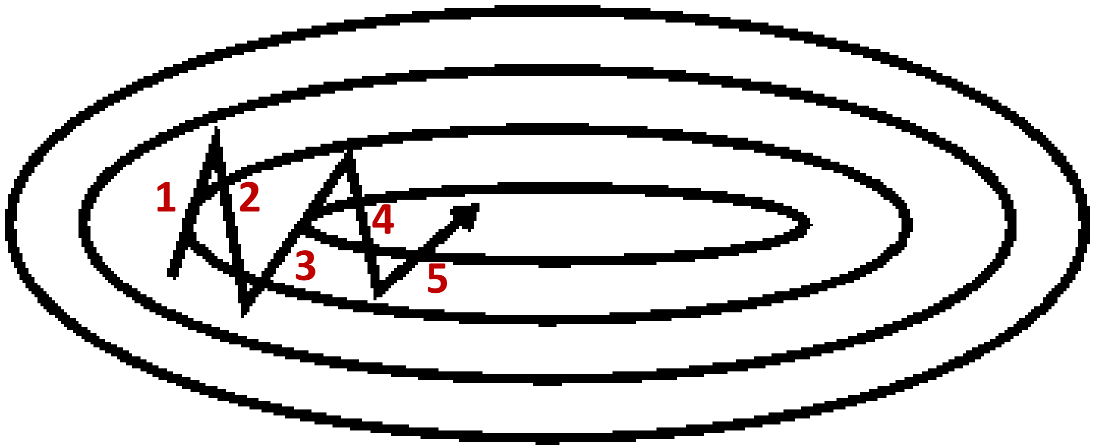
Sequence of gradients

With separate learning rates in each direction, which should have the lowest learning rate in the vertical direction?

- Have separate learning rates for each component
- Directions in which the derivatives swing more should likely have lower learning rates
  - Is likely indicative of more wildly swinging behavior

- Directions of greater swing are indicated by **total movement**
  - Direction of greater movement should have lower learning rate

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
- Solution: Lower learning rate in the vertical direction than in the horizontal direction
  - Based on total motion
  - **As quantified by *root mean squared* (RMS) value**

RMS Prop

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

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 $\boldsymbol{Y}(\boldsymbol{X}_{t+b})$
        - » Compute gradient $\frac{d Div(\boldsymbol{Y}(\boldsymbol{X}_{t+b}), \boldsymbol{d}_{t+b})}{d\boldsymbol{w}}$
        - » Compute $(\partial_w D)_k += \frac{1}{B} \frac{d Div(\boldsymbol{Y}(\boldsymbol{X}_{t+b}), \boldsymbol{d}_{t+b})}{d\boldsymbol{w}}$
    - update: for all $w \in \{w_{\{ij\}}^k \forall i,j,k\}$

$$\boxed{\begin{aligned} E[\partial_w^2 D]_k &= \gamma E[\partial_w^2 D]_{k-1} + (1-\gamma)(\partial_w^2 D)_k \\ w_{k+1} &= w_k - \frac{\eta}{\sqrt{E[\partial_w^2 D]_k + \epsilon}} \partial_w D \end{aligned}}$$

    - $k = k + 1$
- Until loss has converged

Typical values:

$\gamma = 0.9$

$\eta = 0.001$

All the terms in gradient descent

- Standard gradient descent rule

$$W \leftarrow W - \eta \nabla_W L(W)$$

- RMSprop only adapts the learning rate
  - by total movement
- Momentum only smooths the gradient

All the terms in gradient descent

- Standard gradient descent rule

$$W \leftarrow W - \eta \nabla_W L(W)$$

How about combining both?

ADAM

ADAM: RMSprop with momentum

- RMS prop only adapts the learning rate
- Momentum only smooths the gradient

ADAM combines the two

- **Procedure:**

Maintain a running estimate of the mean squared value of derivatives for each parameter

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

$$\Downarrow$$

$$\hat{v}_k = \frac{v_k}{1-\gamma^k}$$

Maintain a running estimate of the mean derivative for each parameter

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

$$\Downarrow$$

$$\hat{m}_k = \frac{m_k}{1-\delta^k}$$

Learning rate is proportional to the *inverse* of the *root mean squared* derivative

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

ADAM: RMSprop with momentum

Maintain a running estimate of the mean squared value of derivatives for each parameter

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

Maintain a running estimate of the mean derivative for each parameter

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

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

These two terms ensure that running average terms are not dominating at early stages

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

Typically, δ and γ will be some positive value close to 1 (smaller than 1)

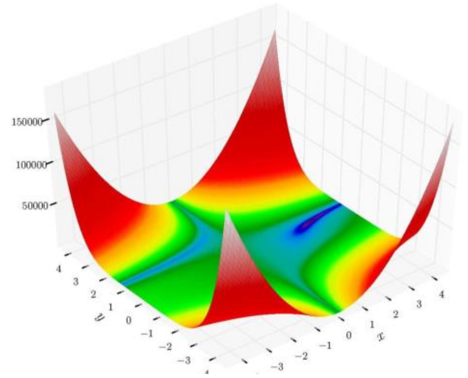
As number of iteration k increases, $1-\delta^k$ and $1-\gamma^k$ will close to 1

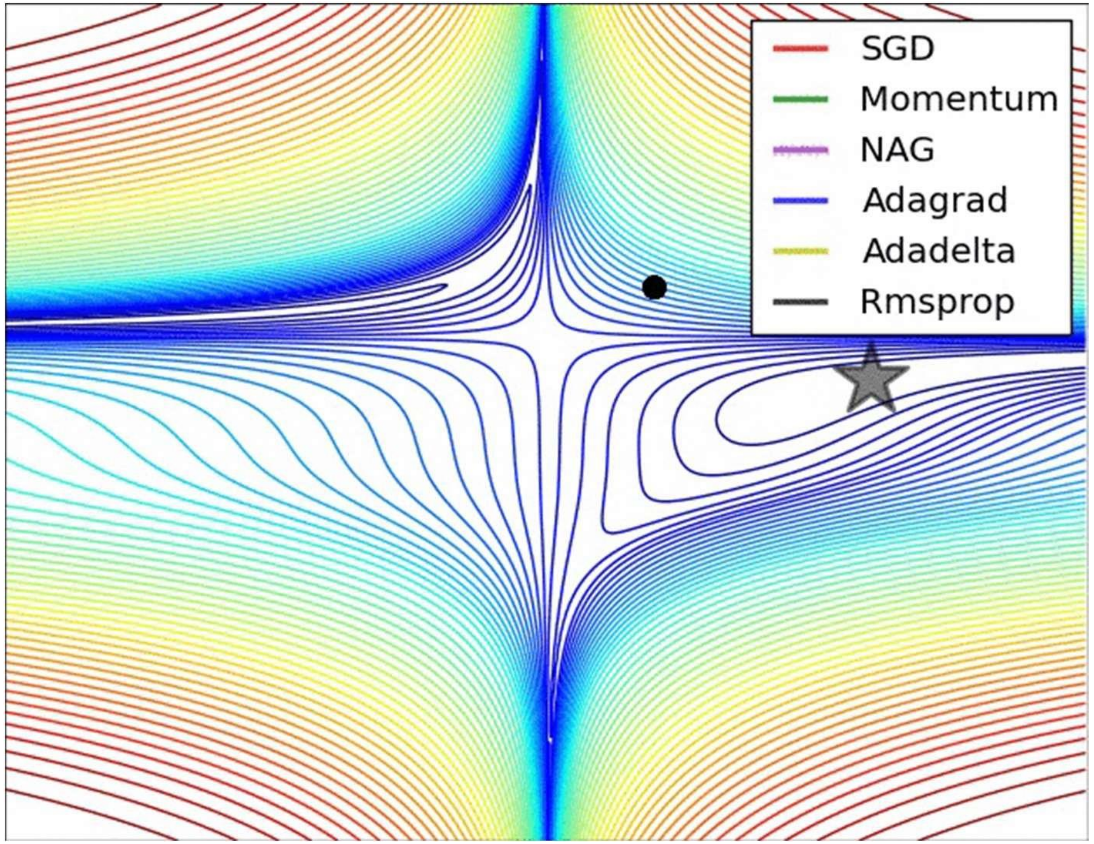
m_0 and v_0 are set to be zeros

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: $\eta = 0.001, \delta = 0.9, \gamma = 0.999$

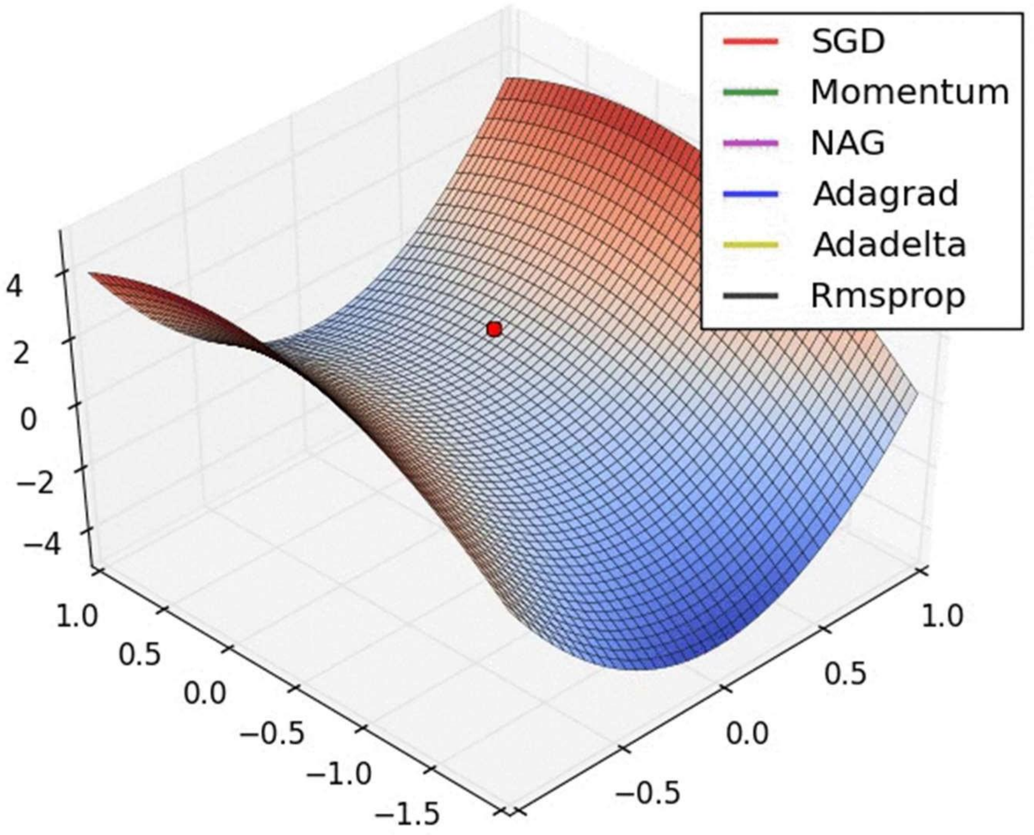
Visualizing the optimizers: Beale's Function





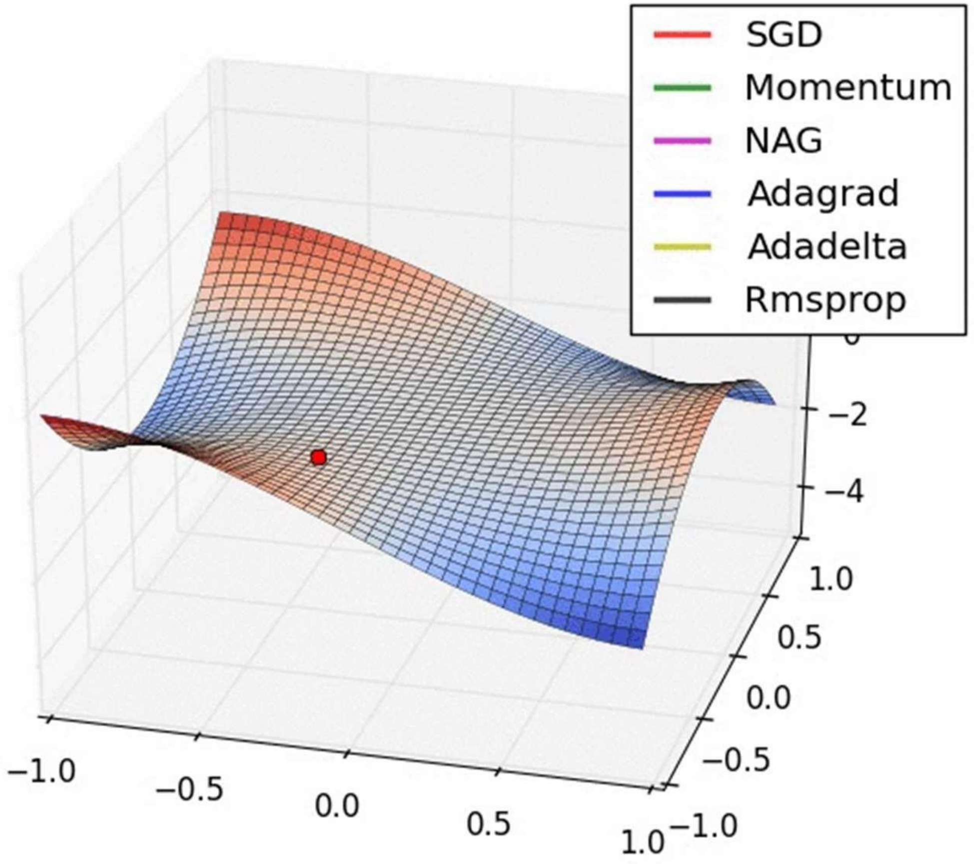
- http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

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