Training Neural Networks: Optimization

Intro to Deep Learning, Spring 2026

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*

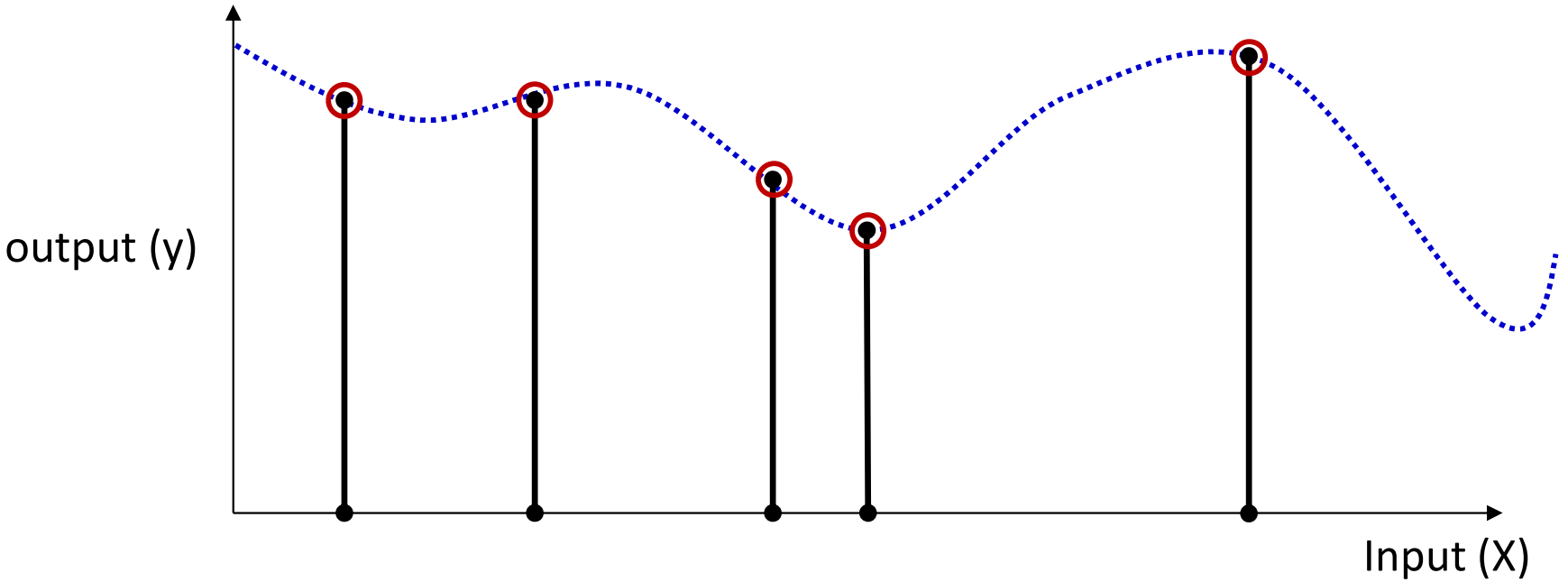
Moving on…

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

Moving on: Topics for the day

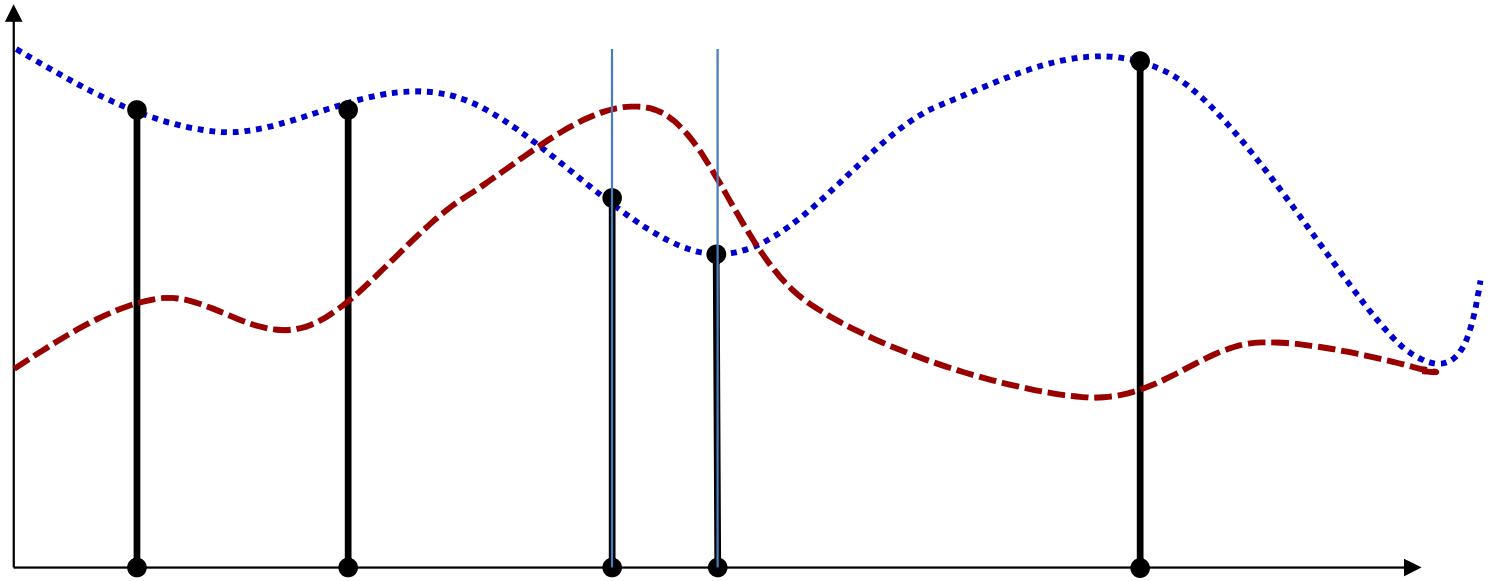
- Incremental updates
- Revisiting “trend” algorithms
- Generalization
- Tricks of the trade
  - Divergences..
  - Activations
  - Normalizations

The training formulation



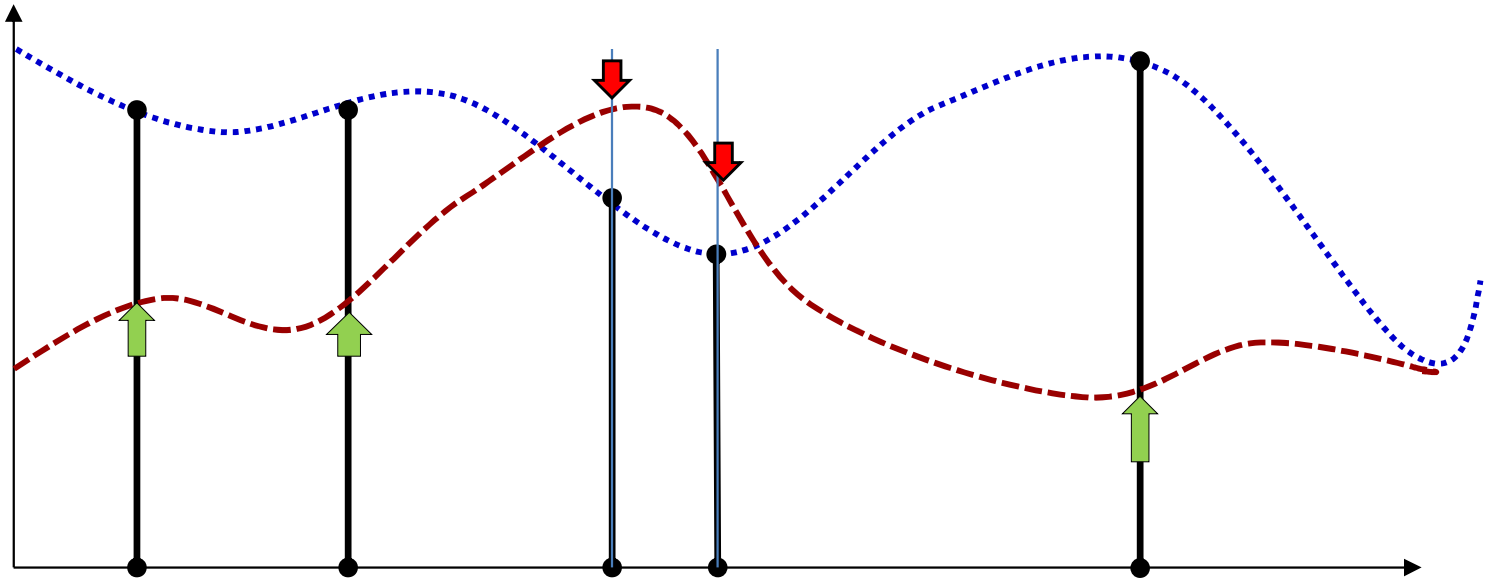
- Given input output pairs at a number of locations, estimate the entire function

Gradient descent



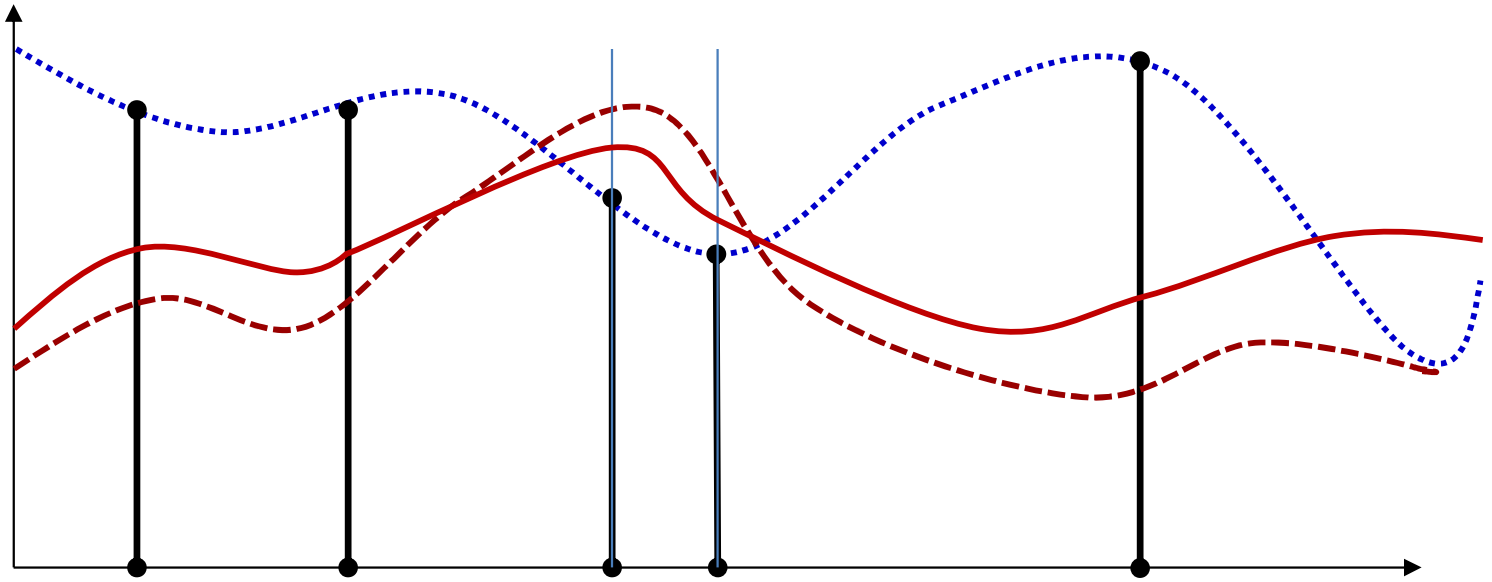
- Start with an initial function

Gradient descent



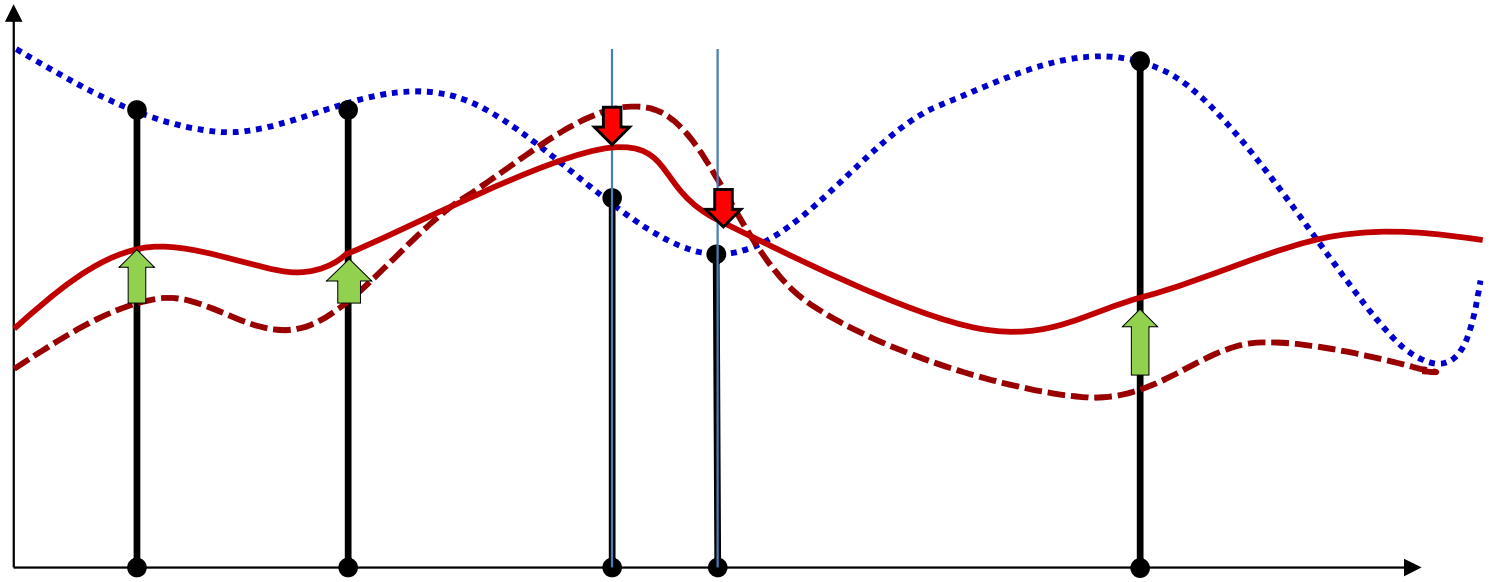
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



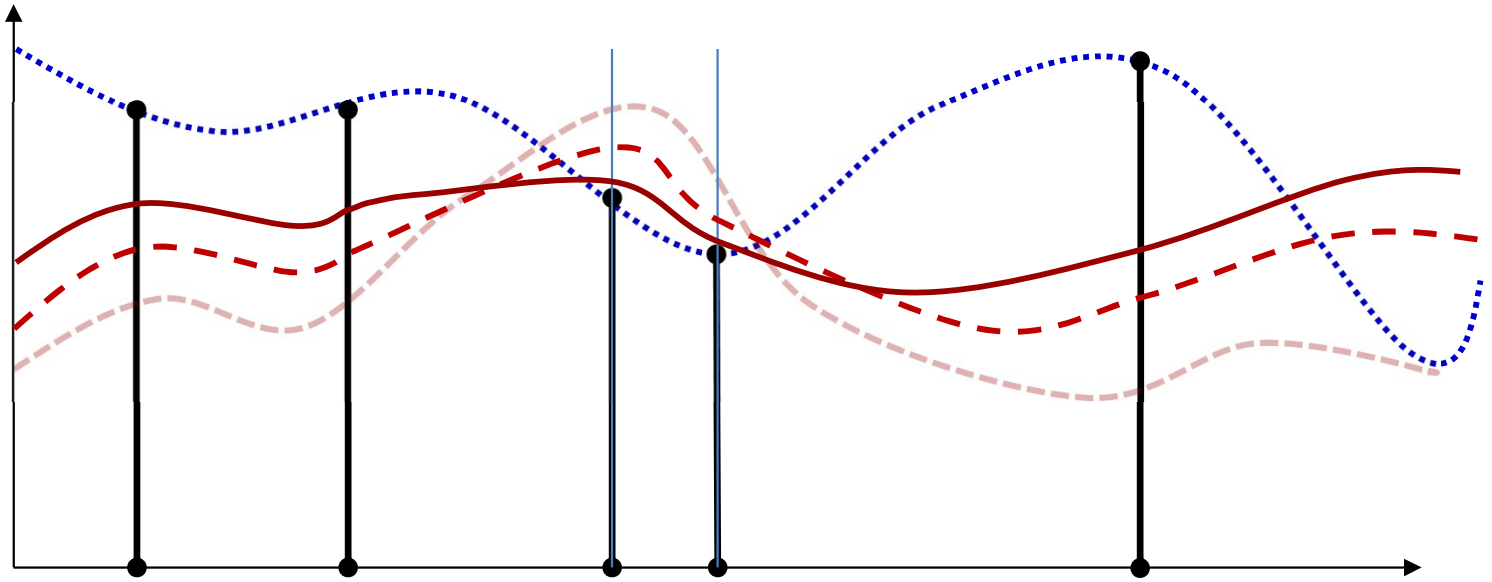
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



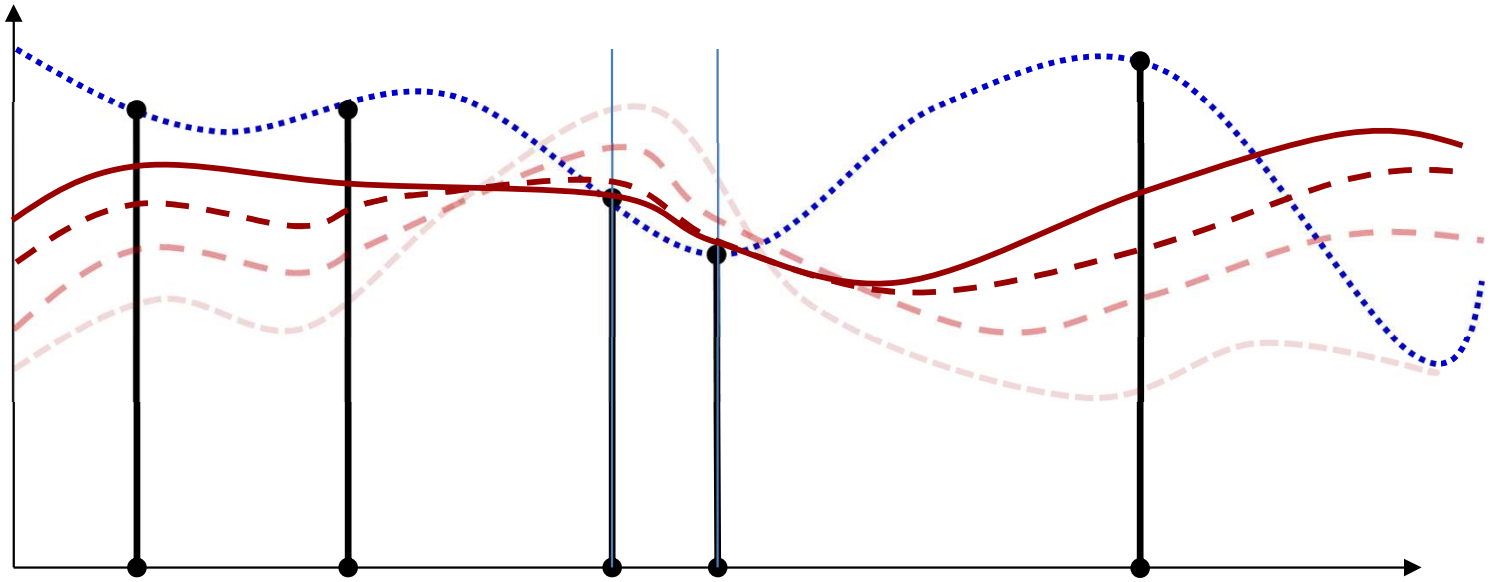
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



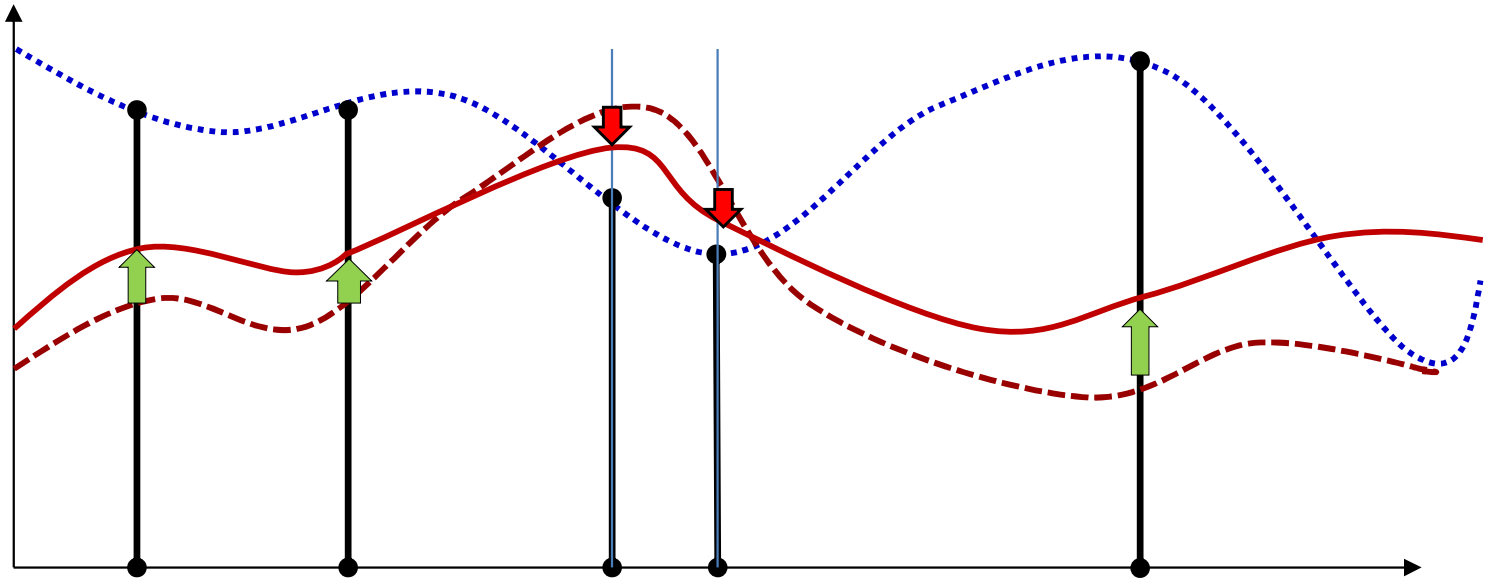
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

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

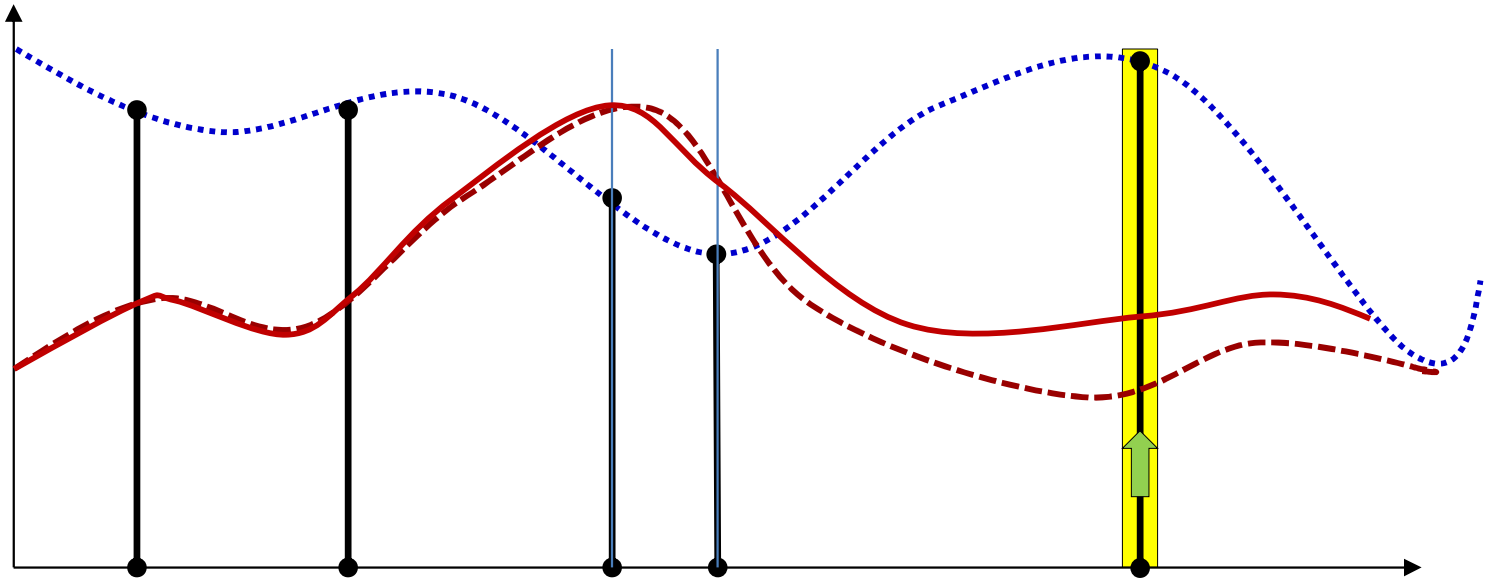
Poll 1

- Select all that are correct
  - The actual loss function we try to minimize requires batch updates
  - Batch updates minimize the total loss over the entire training data
  - Batch updates optimize the actual loss function
  - Batch updates require processing the entire training data before we perform a single update

Poll 1

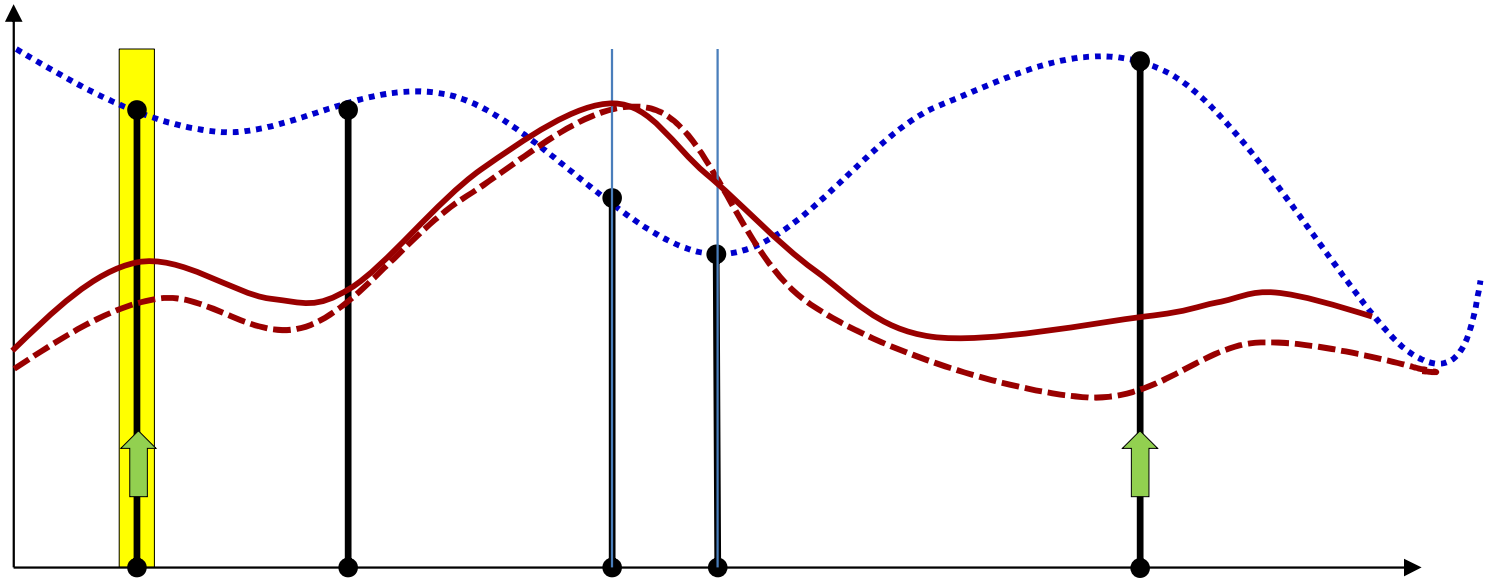
- Select all that are correct
  - **The actual loss function we try to minimize requires batch updates**
  - **Batch updates minimize the total loss over the entire training data**
  - **Batch updates optimize the actual loss function**
  - **Batch updates require processing the entire training data before we perform a single update**

Alternative: Incremental update



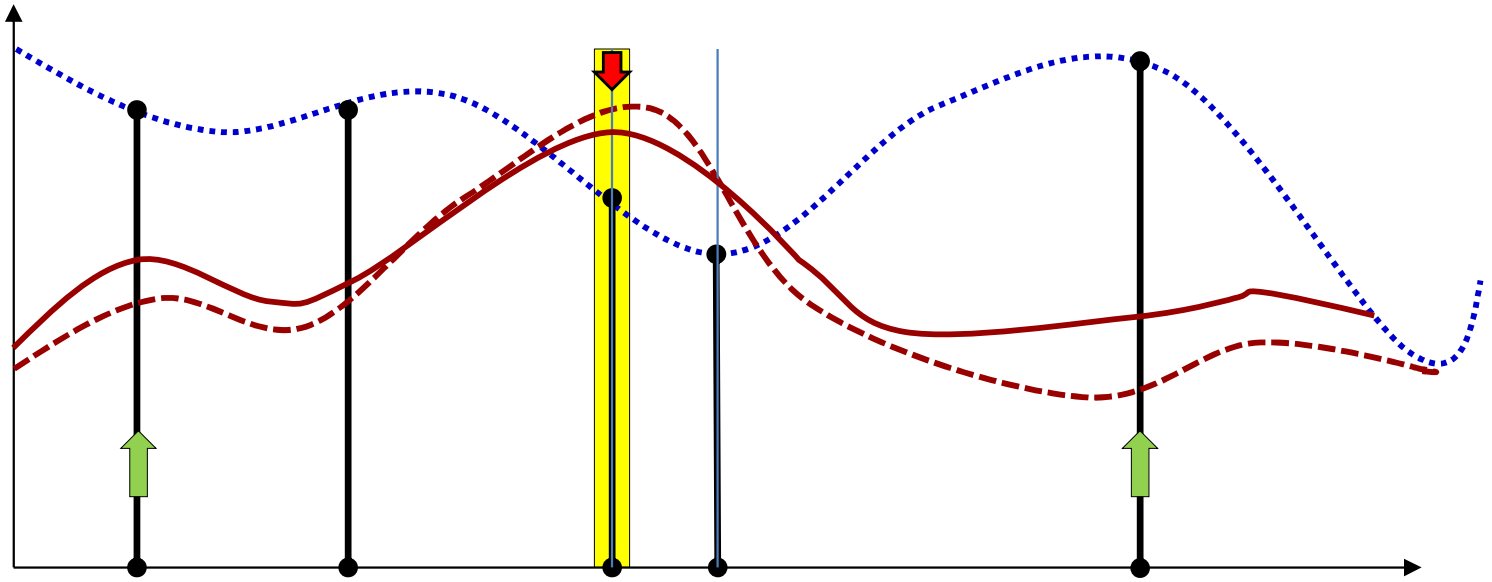
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

Alternative: Incremental update



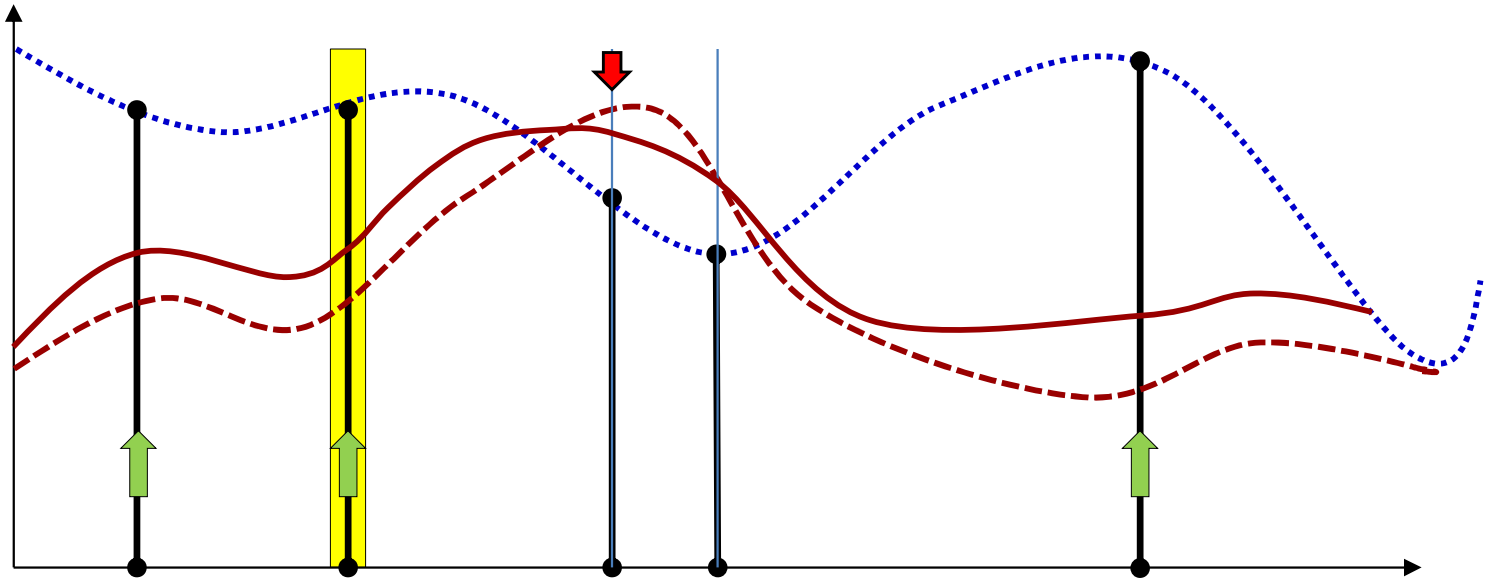
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

Alternative: Incremental update



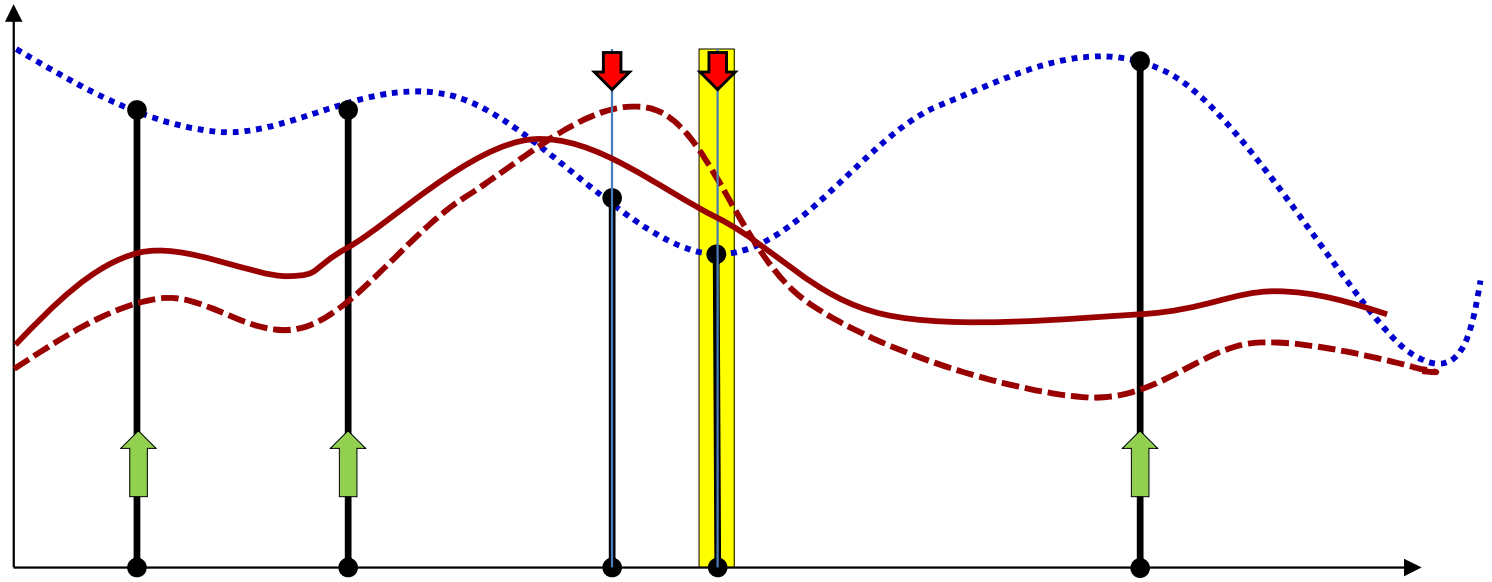
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

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

Incremental Updates

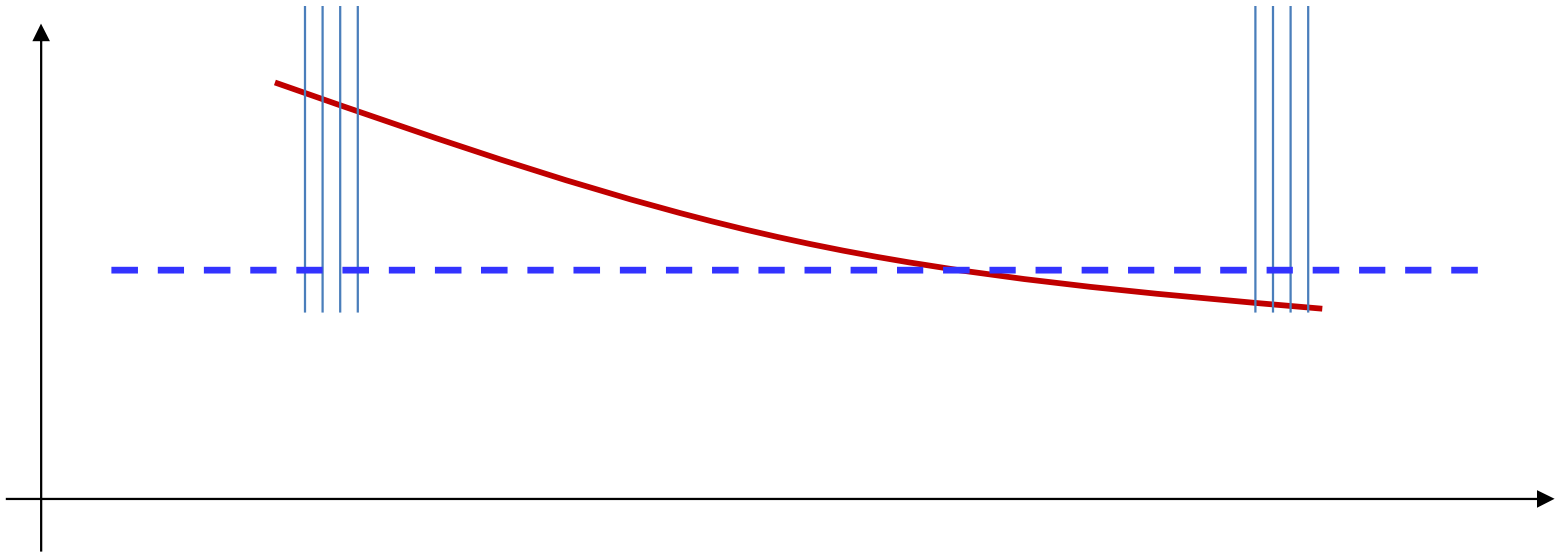
- The iterations can make multiple passes over the data
- A single pass through the entire training data is called an “epoch”
  - An epoch over a training set with T samples results in T updates of parameters

Incremental Update

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

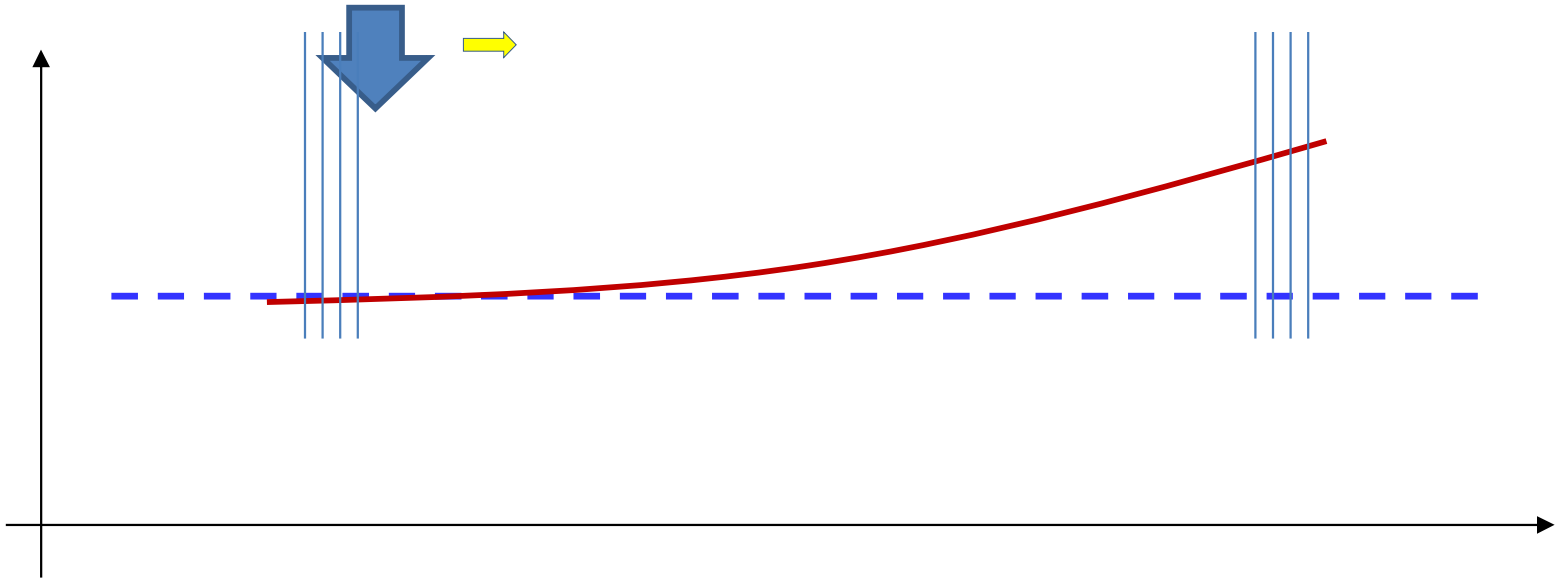
        $W_k = W_k - \eta \nabla_{W_k} \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})^T$ ← One update
- Until $Loss$ has converged

Caveats: order of presentation



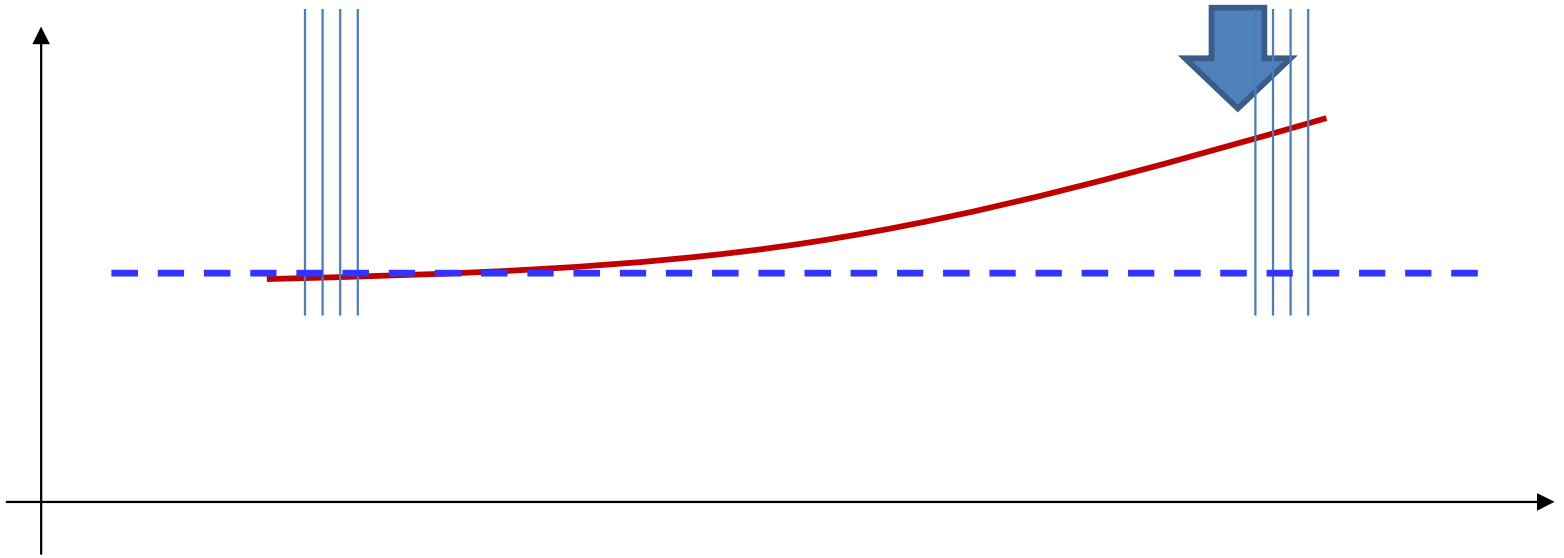
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



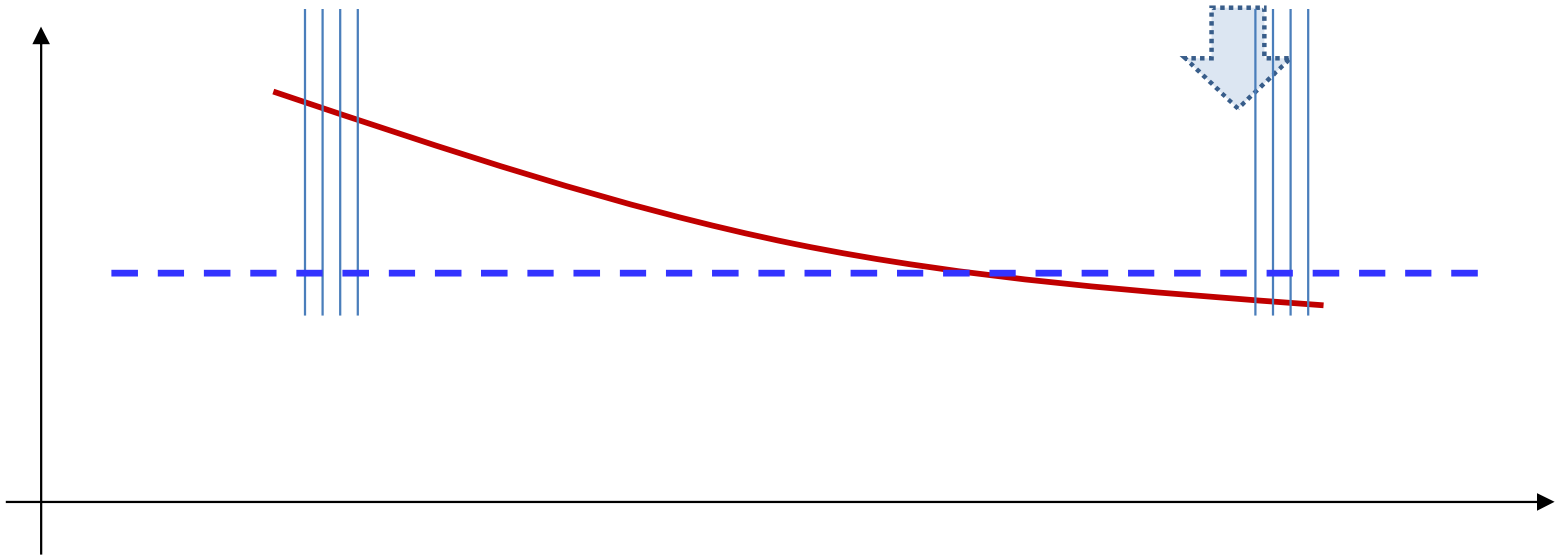
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



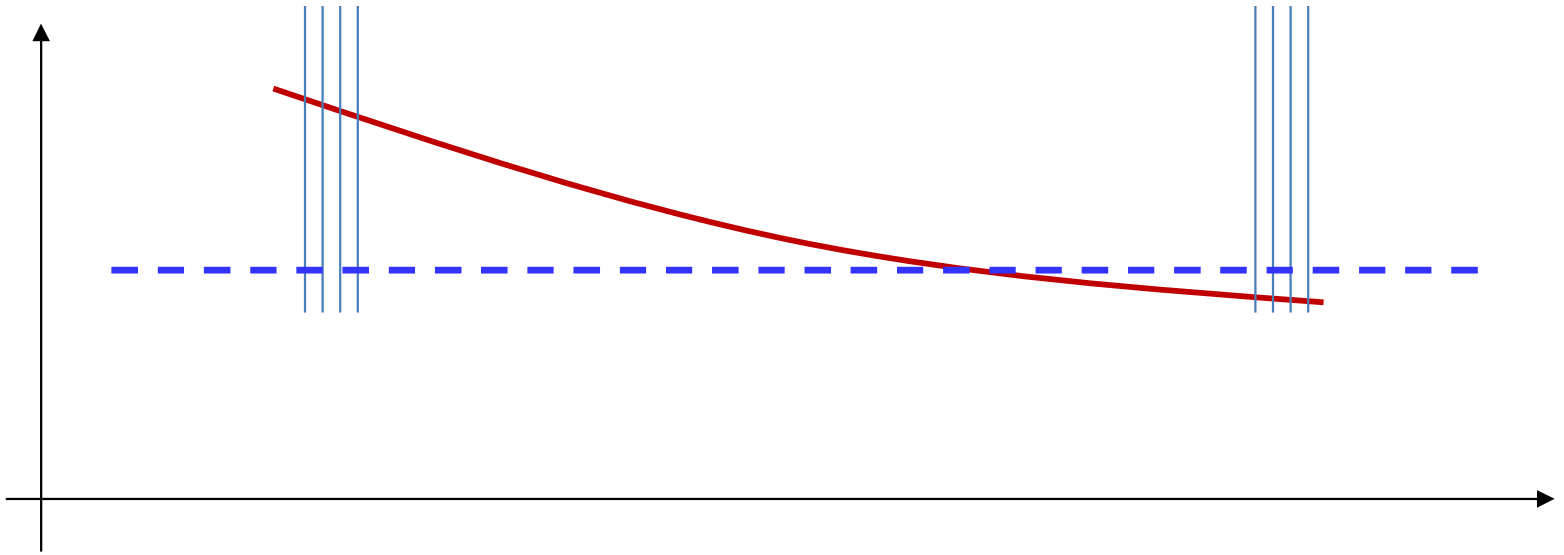
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



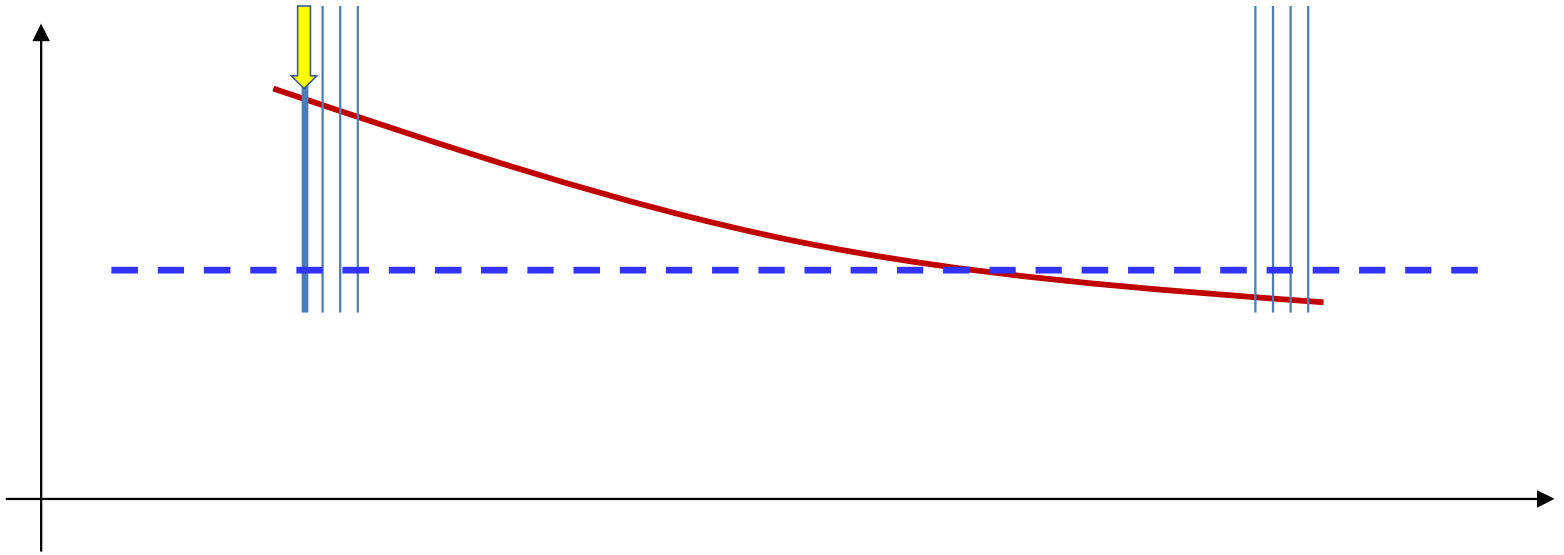
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



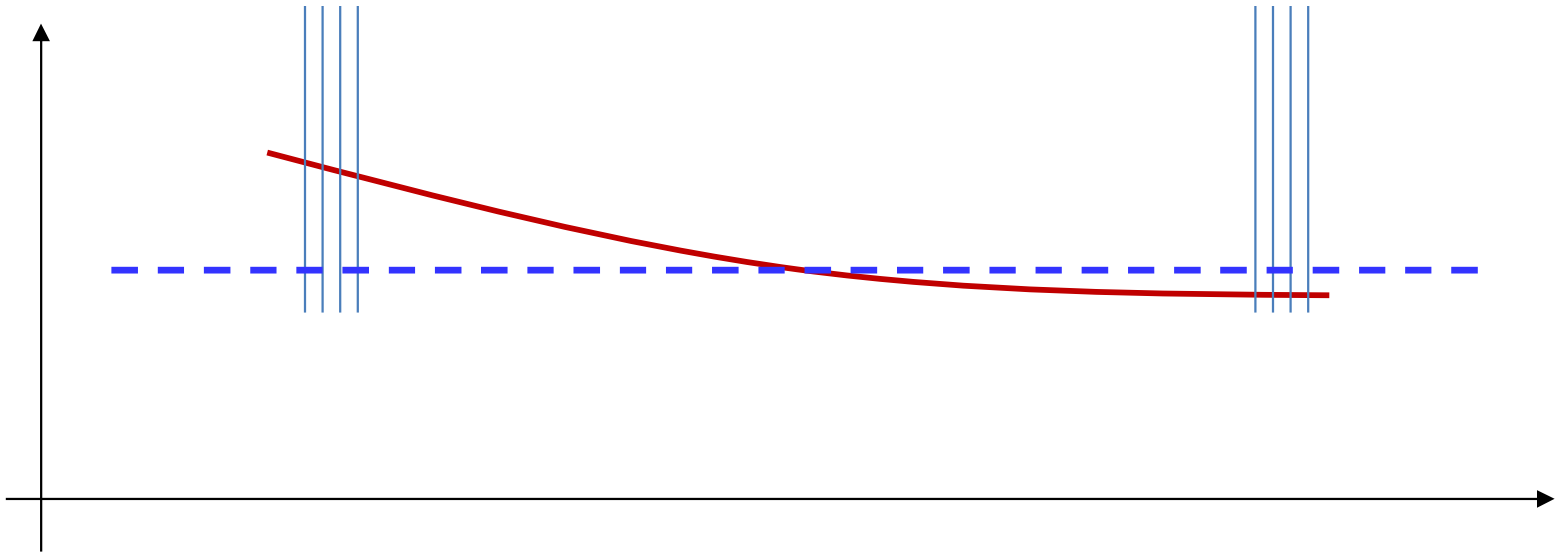
- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly* to get more convergent behavior

Caveats: order of presentation



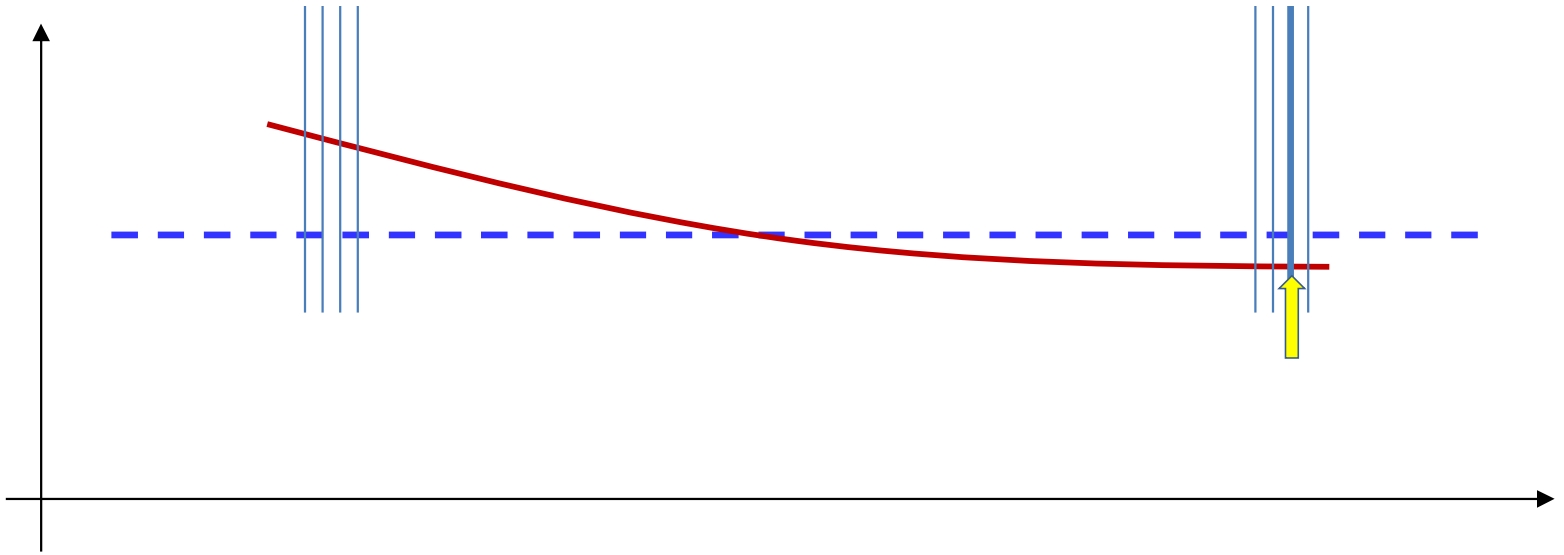
- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly* to get more convergent behavior

Caveats: order of presentation



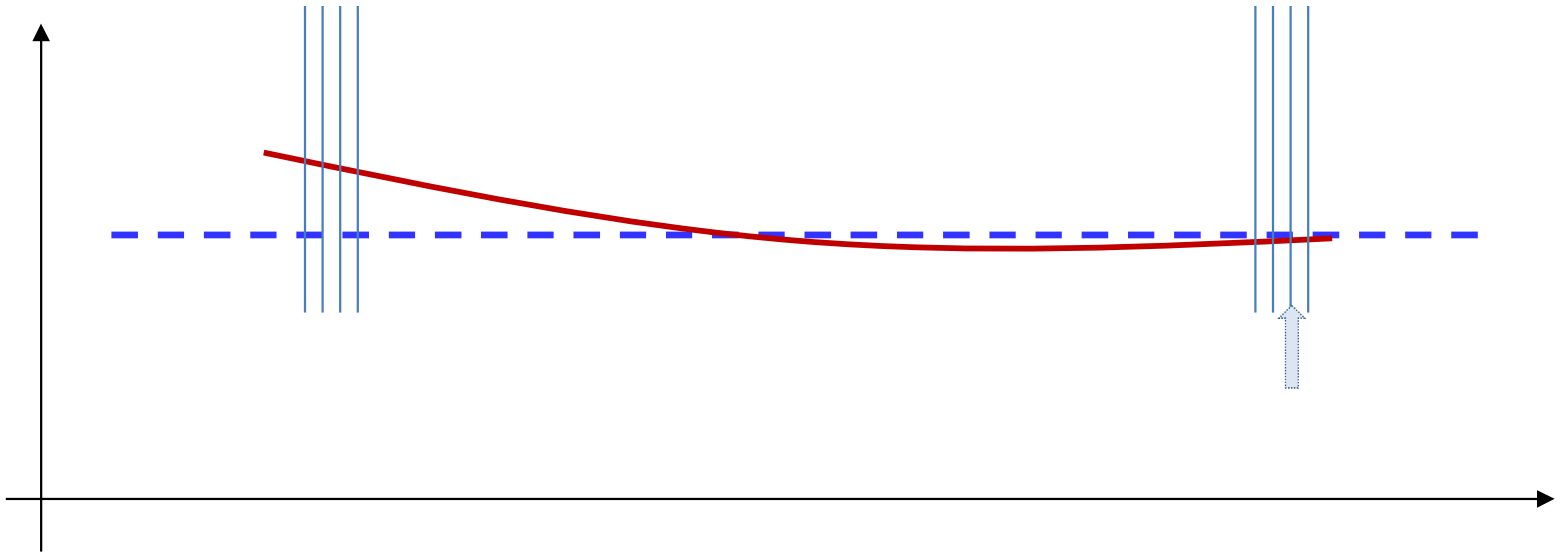
- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly* to get more convergent behavior

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

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

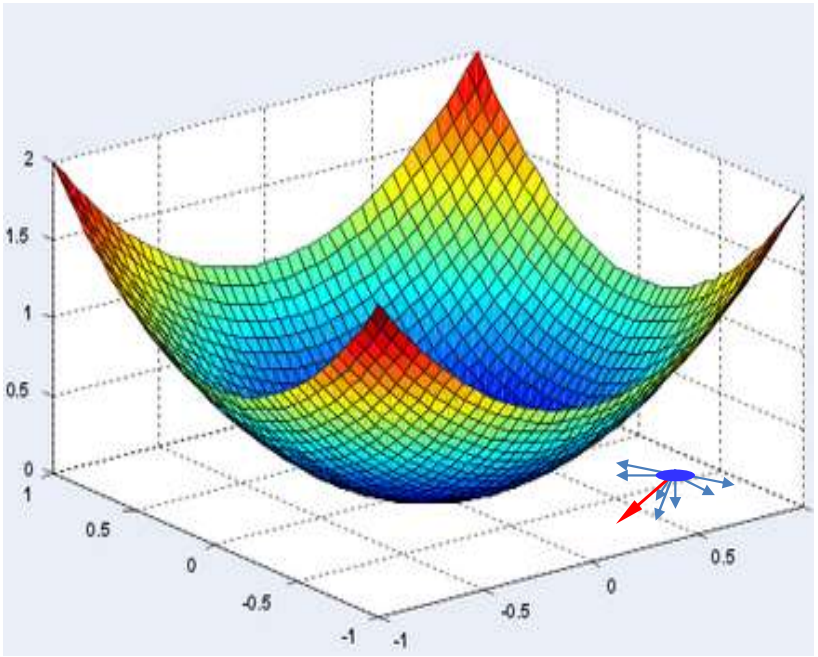
Explanations and restrictions

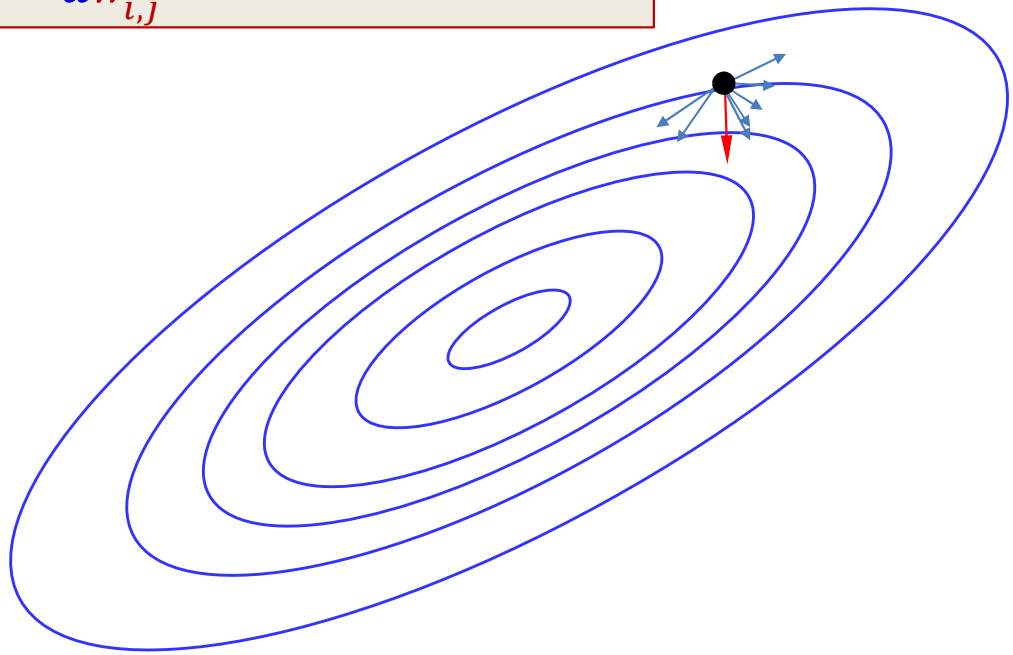
- So why does this process of incremental updates work?
- Under what conditions?

- For “why”: first consider a simplistic explanation that’s often given
  - Look at an extreme example

The expected behavior of the gradient

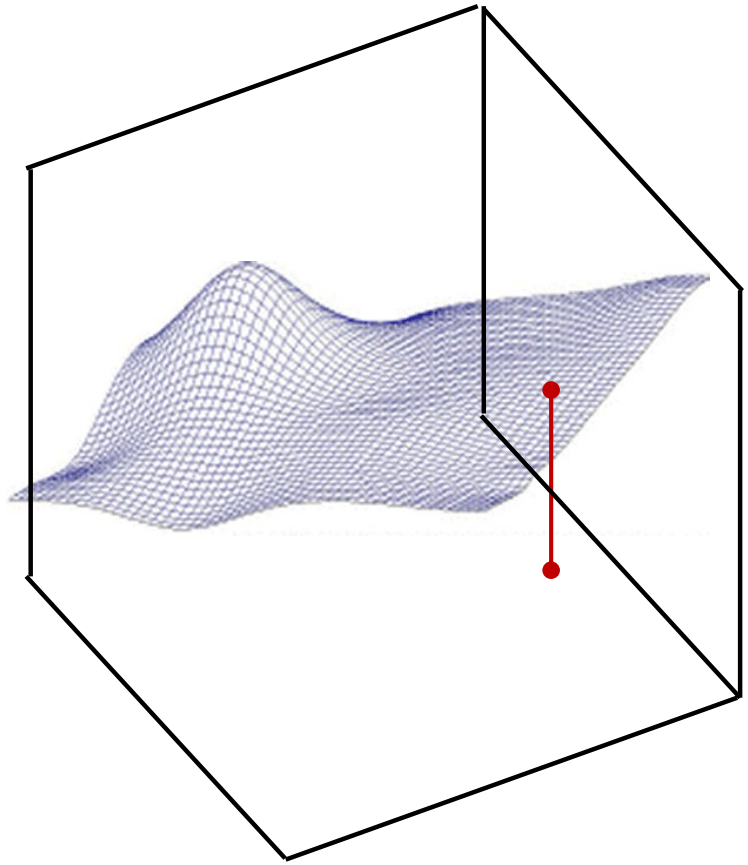
$$\frac{dE(\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \ldots, \boldsymbol{W}^{(K)})}{\boldsymbol{d}w_{i,j}^{(k)}} = \frac{1}{T}\sum_{i} \frac{\boldsymbol{dDiv}(\boldsymbol{Y}(\boldsymbol{X_i}), \boldsymbol{d_i}; \boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \ldots, \boldsymbol{W}^{(K)})}{\boldsymbol{d}w_{i,j}^{(k)}}$$





- 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

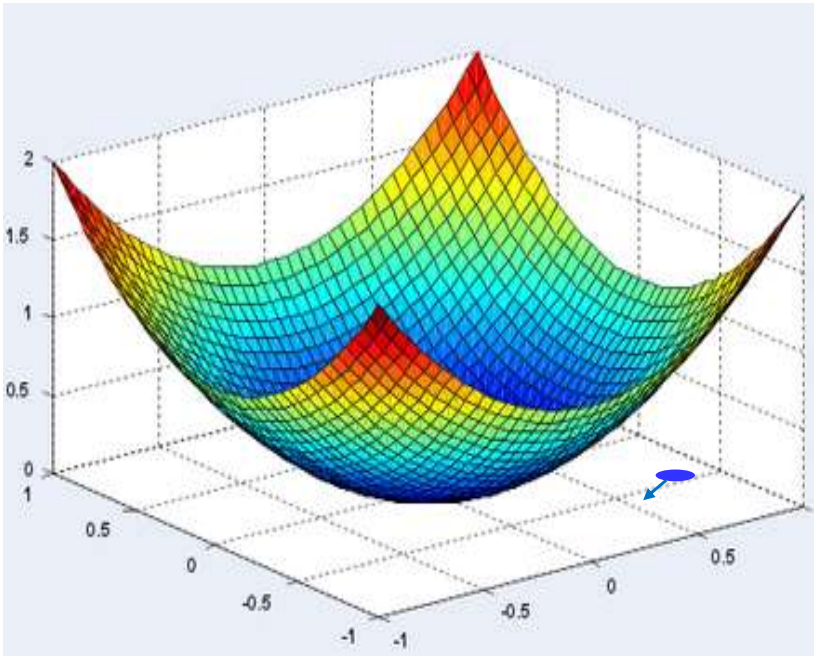


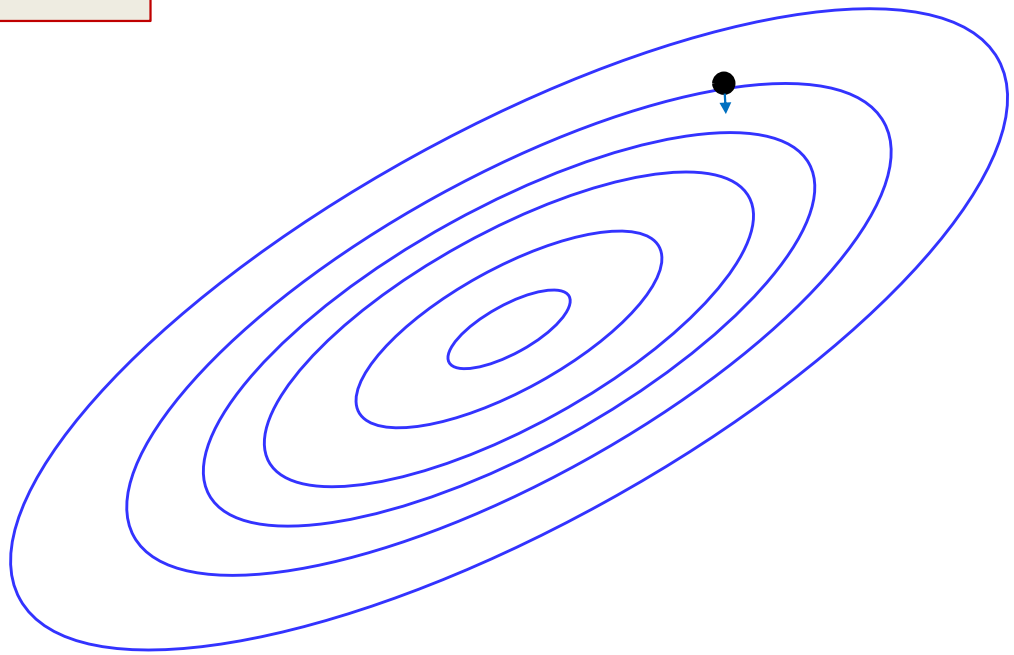
$$X_1 = X_2 = \cdots = X_T$$

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

The expected behavior of the gradient

$$\frac{dE}{dw_{i,j}^{(k)}} = \frac{1}{T}\sum_{i}\frac{dDiv(Y(X_i), d_i)}{dw_{i,j}^{(k)}} = \frac{dDiv(Y(X_i), d_i)}{dw_{i,j}^{(k)}}$$

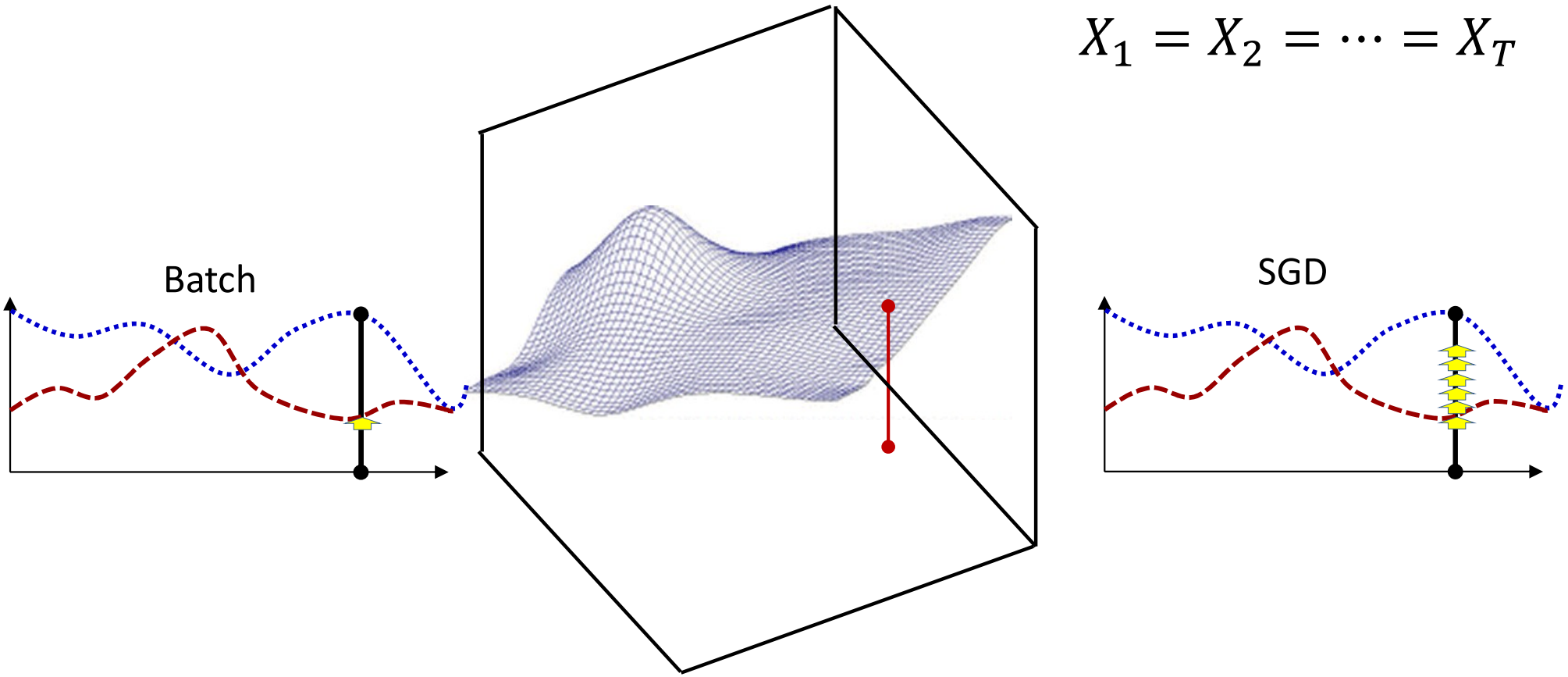




- The individual training instance contribute identical directions to the overall gradient
  - The final gradient points is simply the gradient for an individual instance

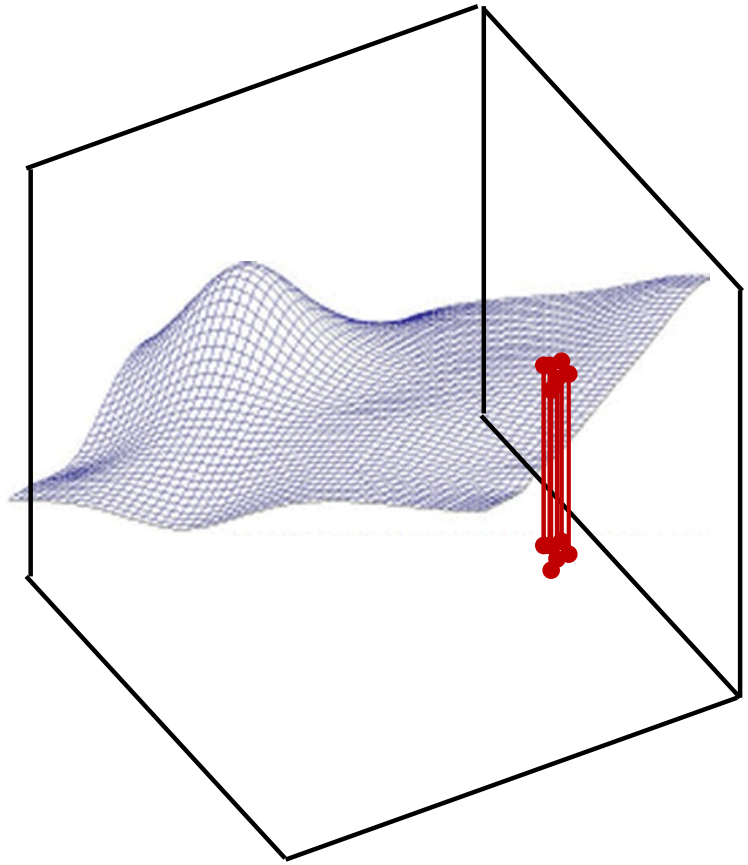
Batch vs SGD

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



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

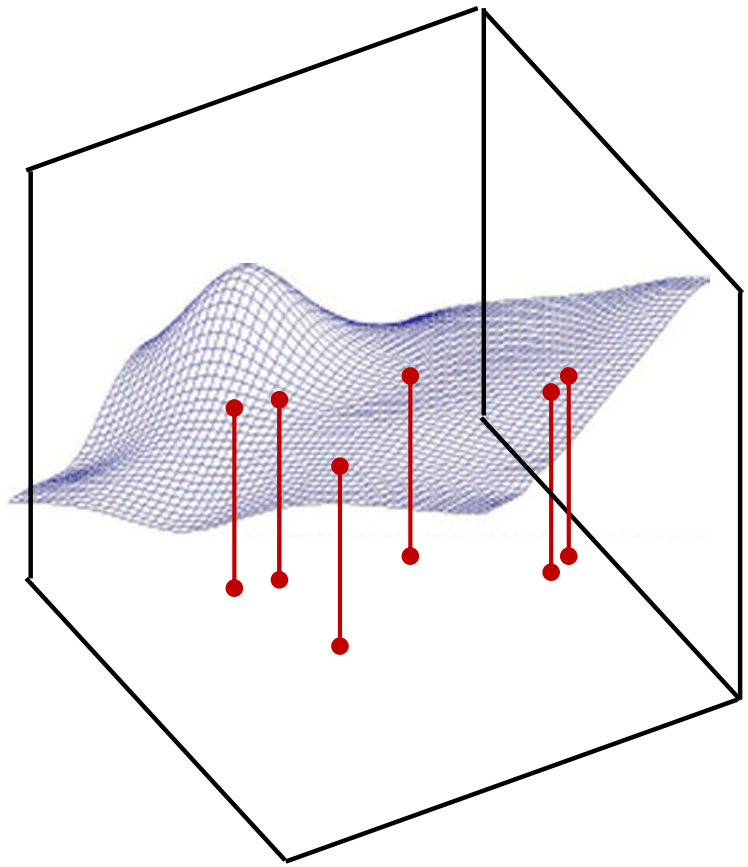
Clumpy data..



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

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

Clumpy data..

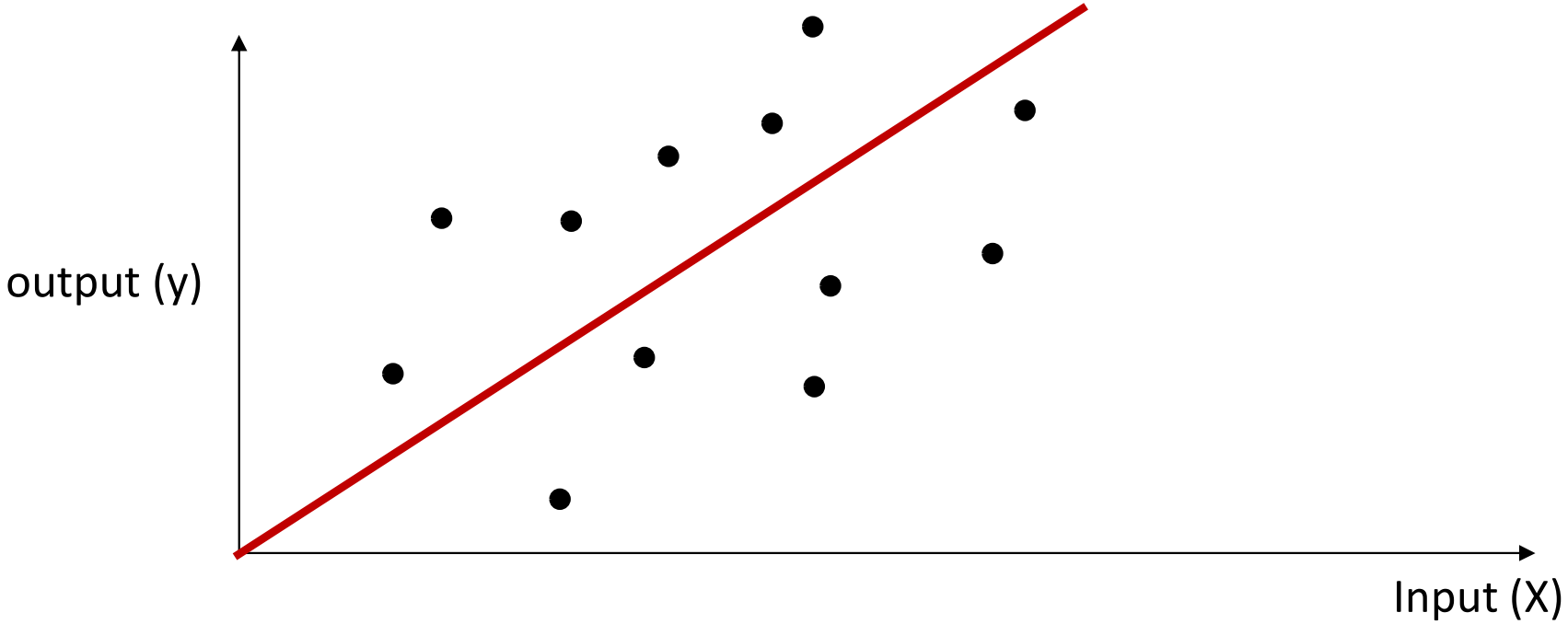


- As data get increasingly diverse, the benefits of incremental updates decrease, but do not entirely vanish

When does it work

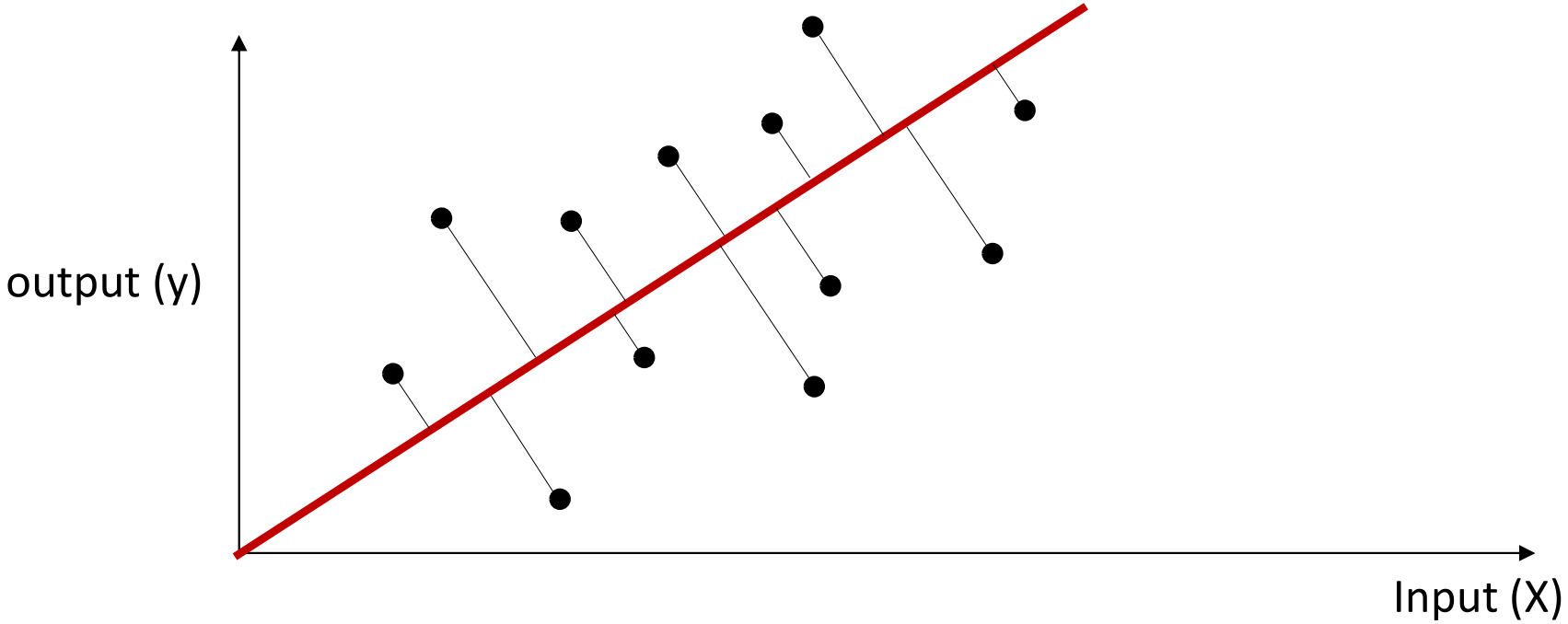
- What are the considerations?
- And how well does it work?

Incremental learning runs the risk of always chasing the latest input



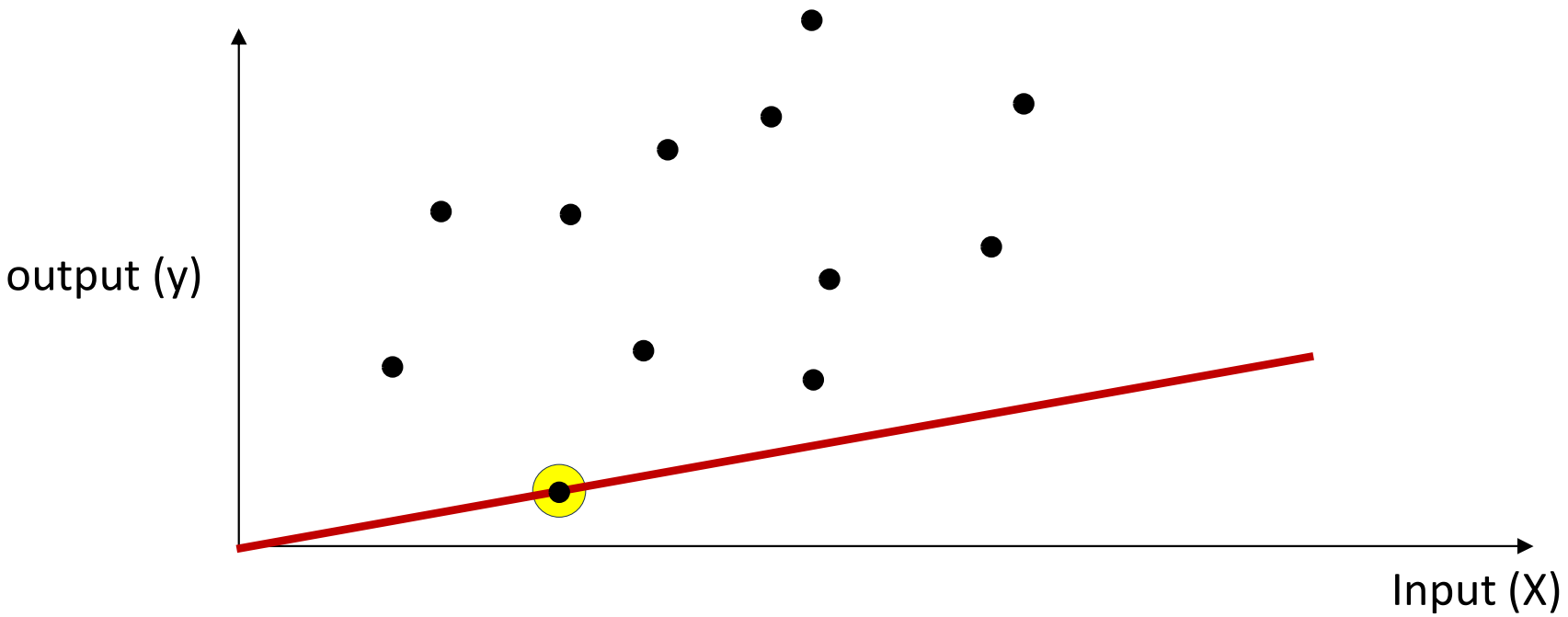
- **Modelling problem:** Find a linear regression line (through origin) to model the data

Incremental learning runs the risk of always chasing the latest input



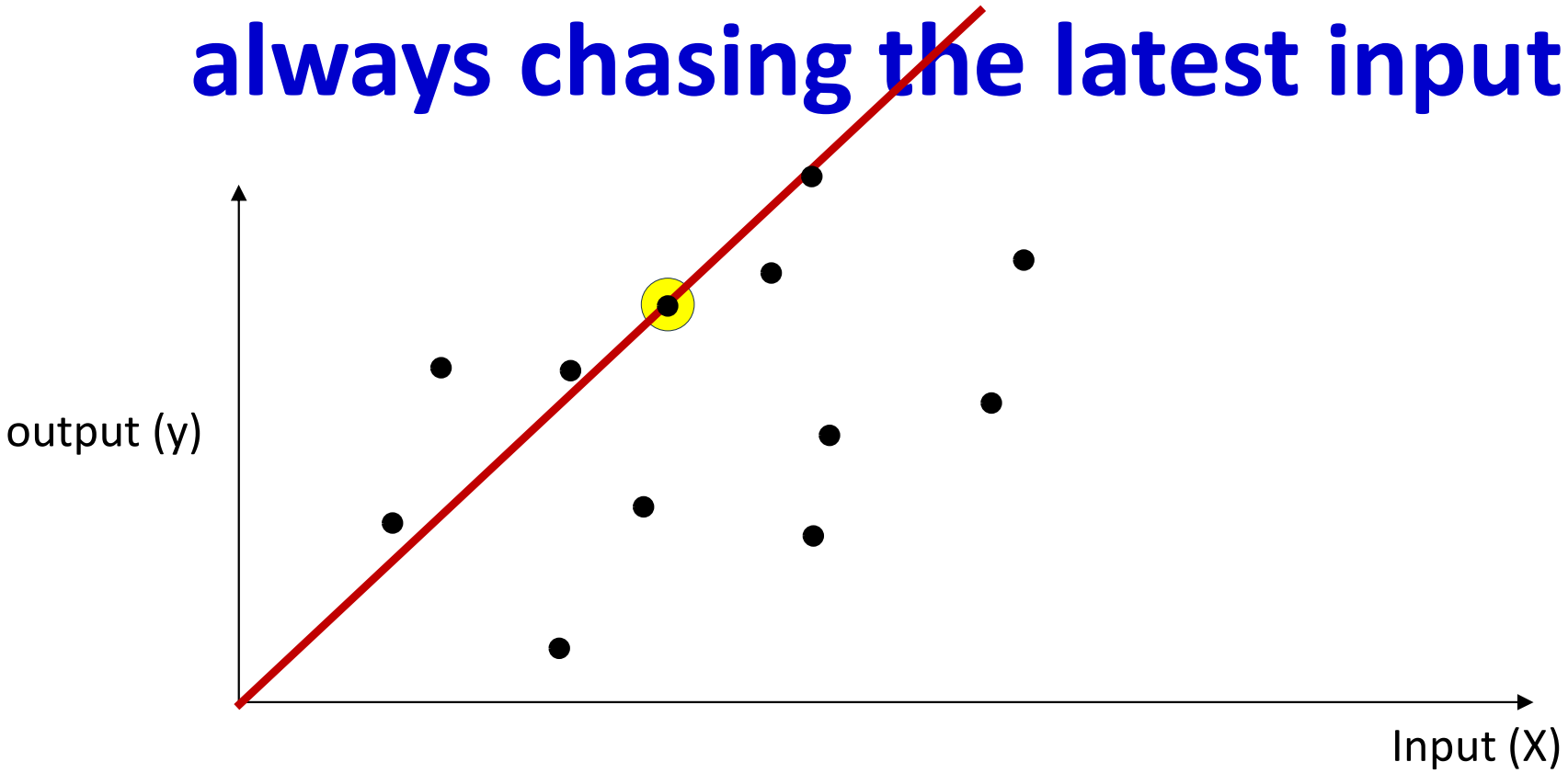
- **Modelling problem:** Find a linear regression line (through origin) to model the data
  - **Batch processing:** Find the line through origin that has the lowest overall squared projection error w.r.t. data

Incremental learning runs the risk of always chasing the latest input



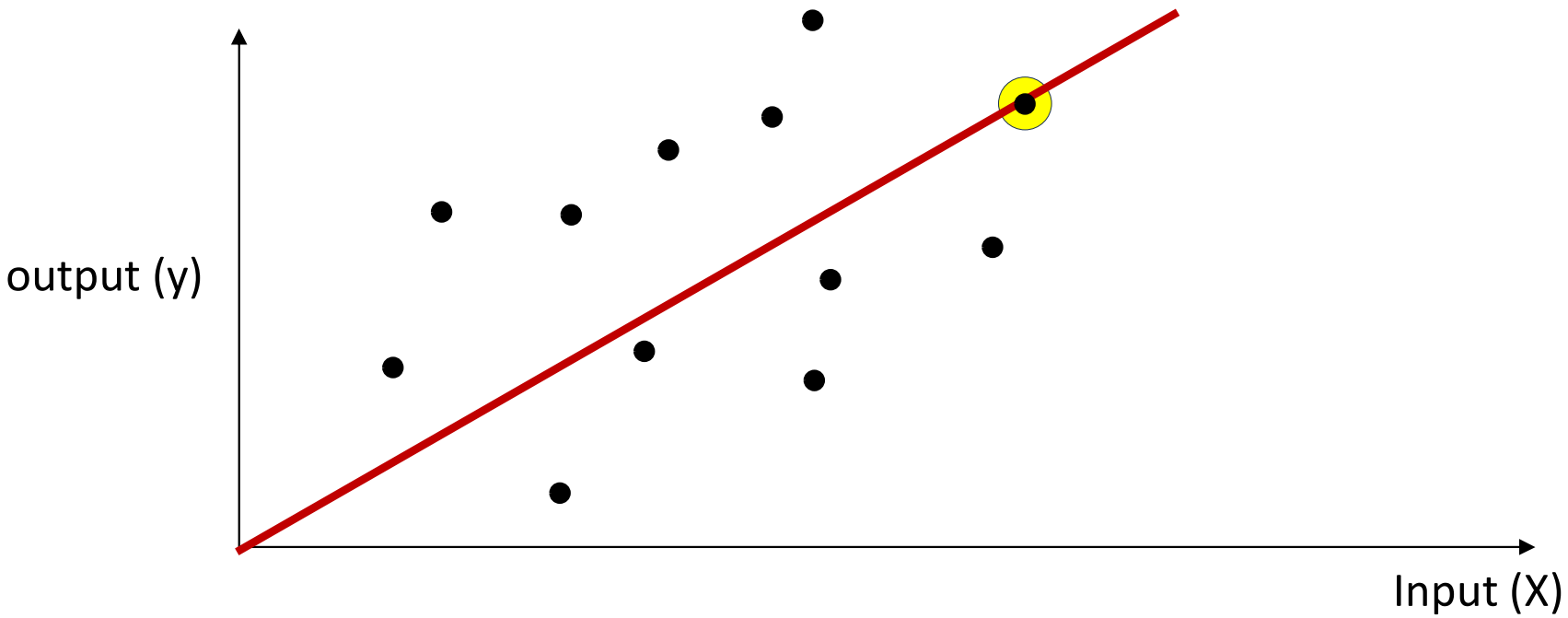
- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - It will never converge

Incremental learning runs the risk of always chasing the latest input



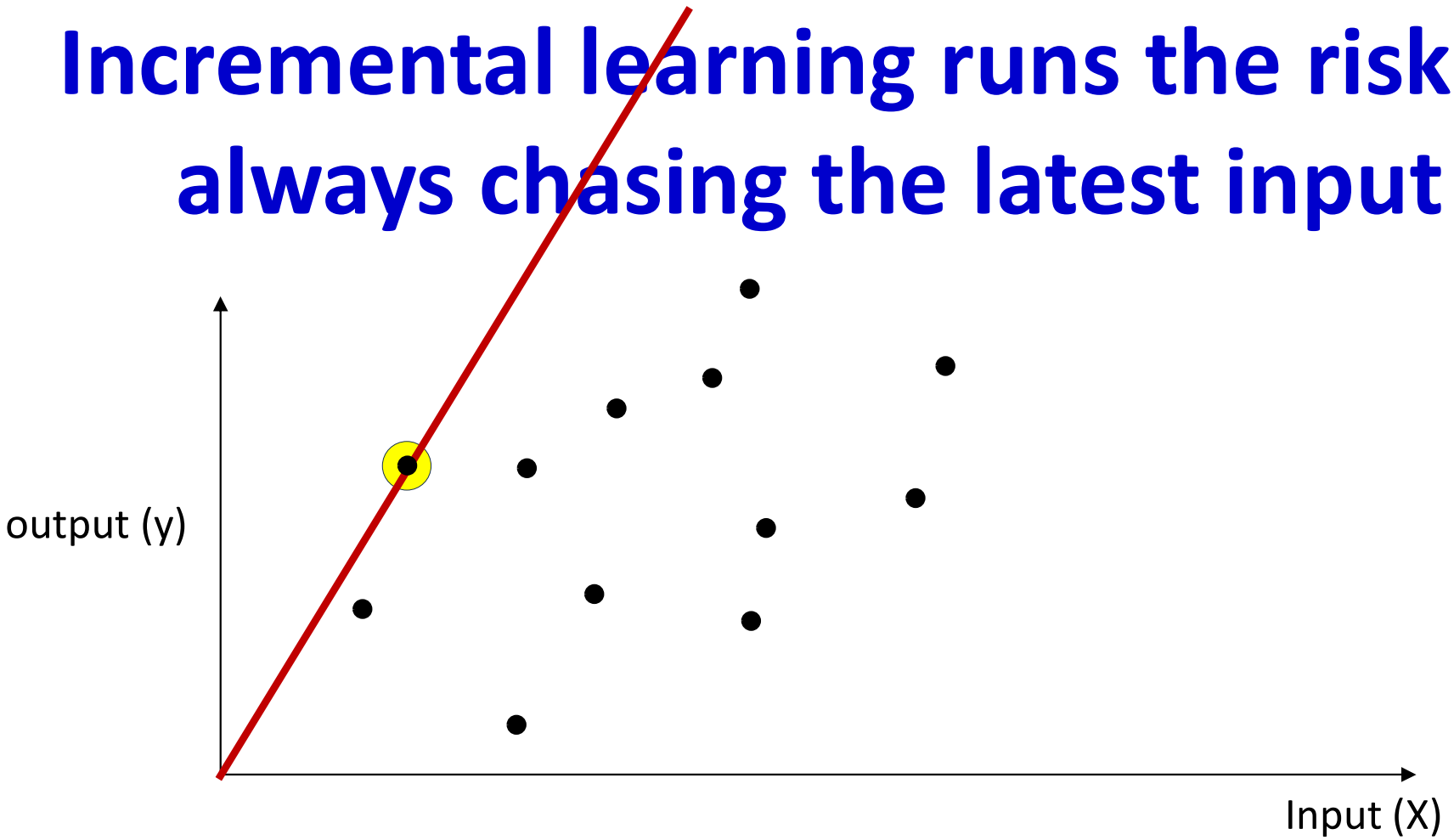
- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - It will never converge

Incremental learning runs the risk of always chasing the latest input



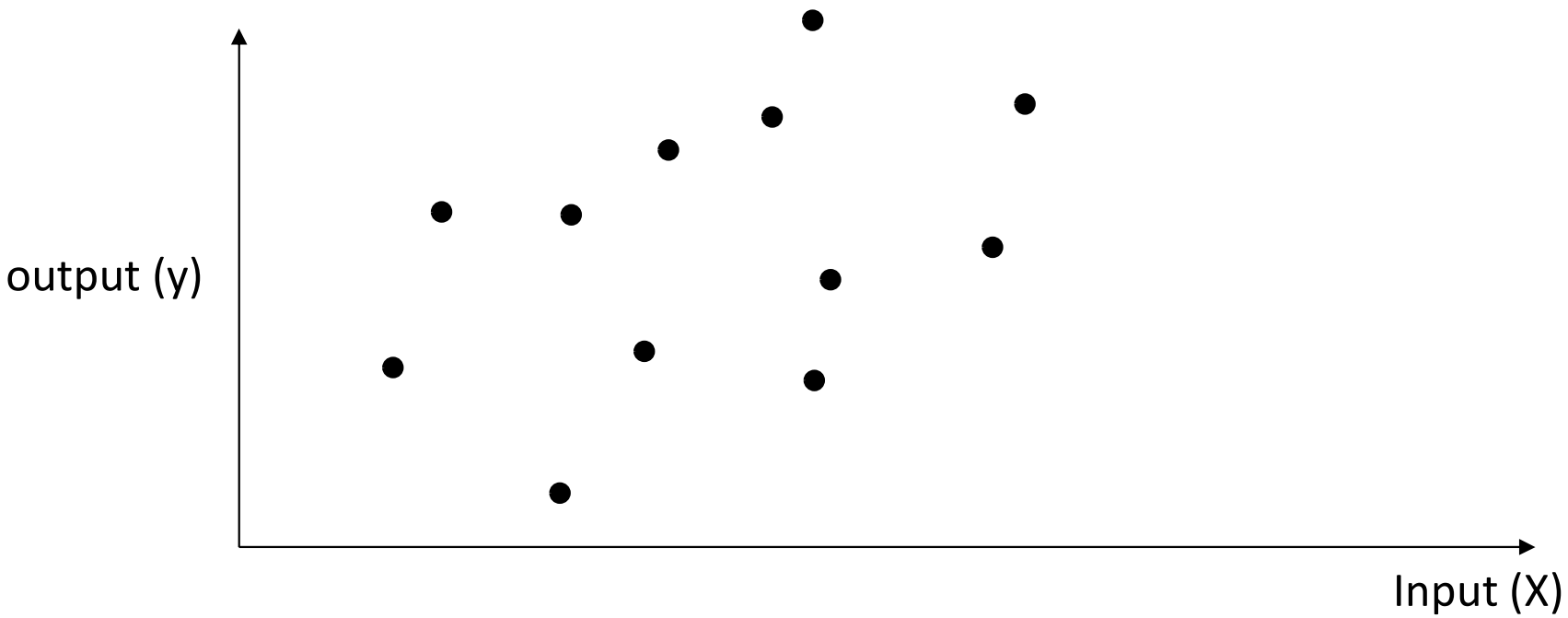
- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - It will never converge

Incremental learning runs the risk of always chasing the latest input



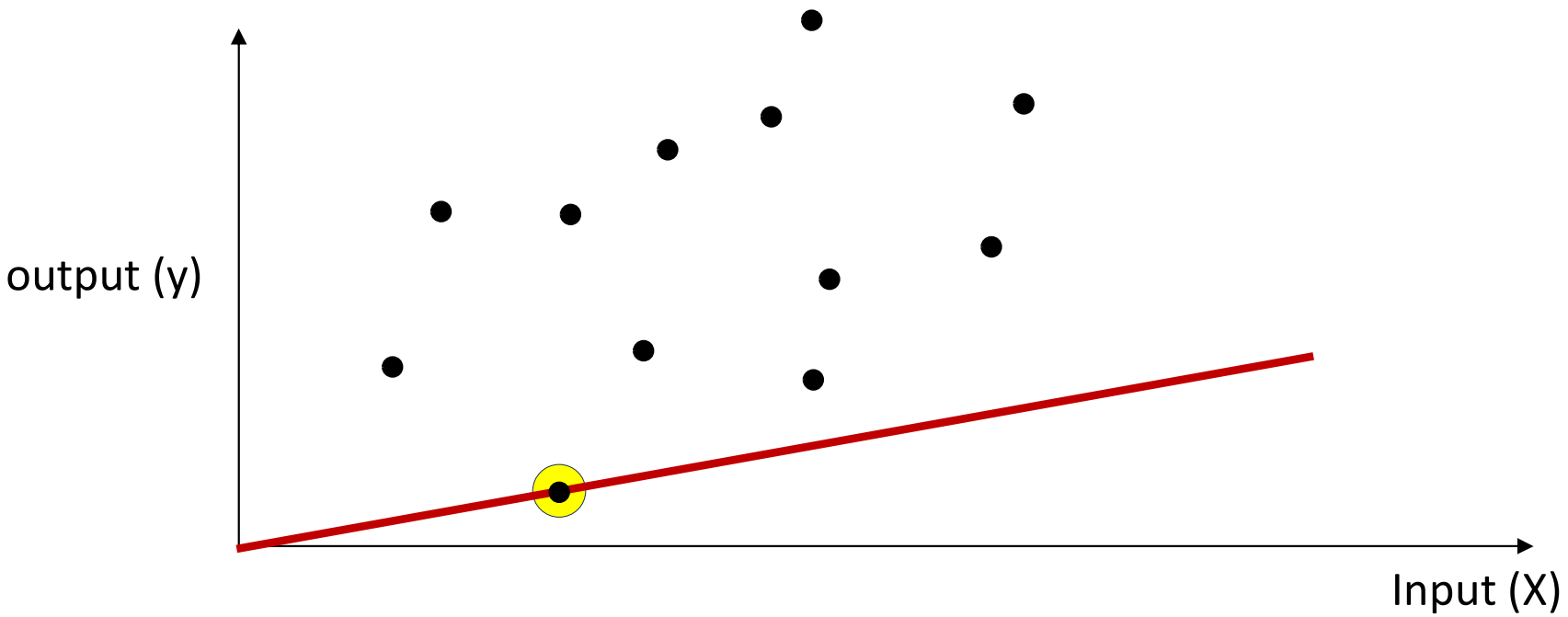
- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - It will never converge
  - **Solution?**

Incremental learning runs the risk of always chasing the latest input



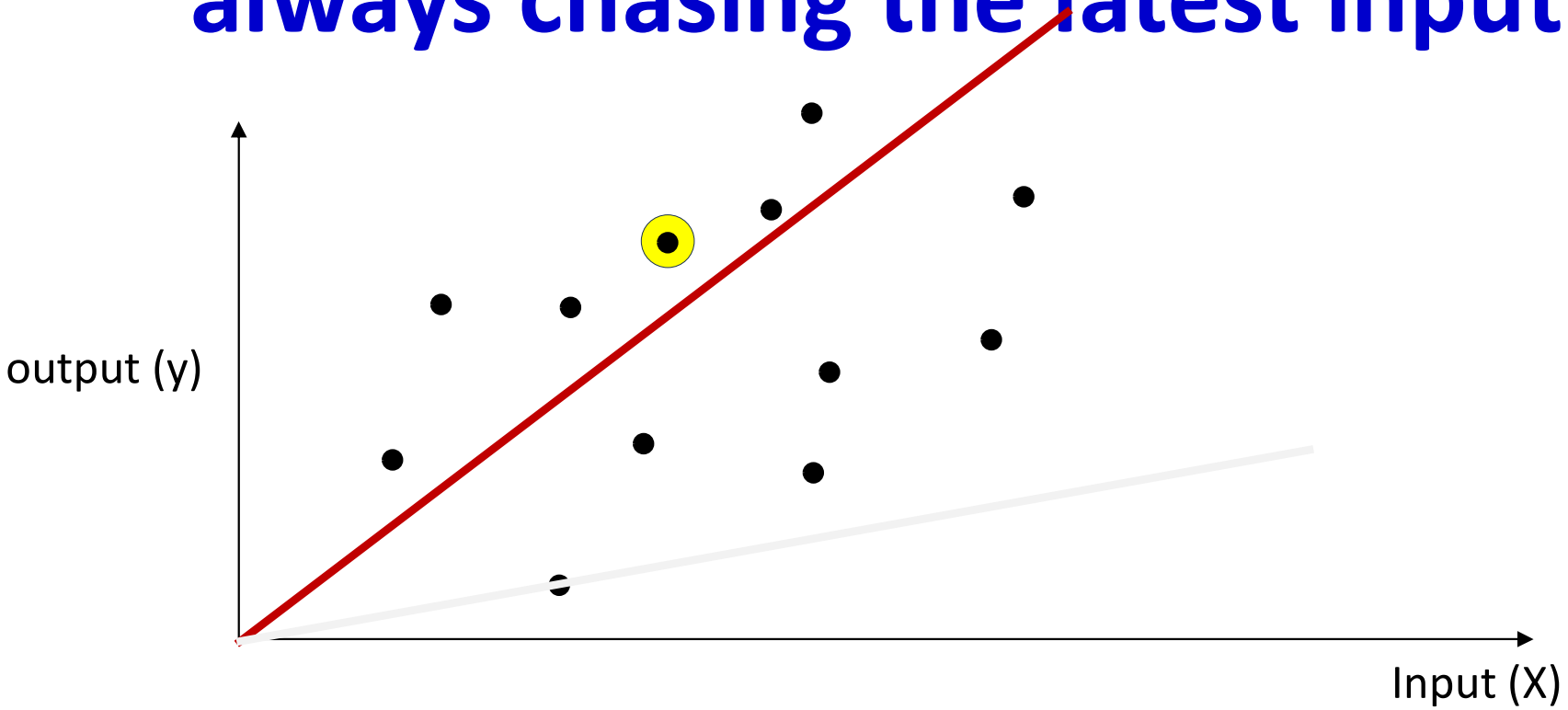
- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - Shrink the learning rate with iterations
  - With increasing iterations, it will swing less and less towards the new point

Incremental learning runs the risk of always chasing the latest input



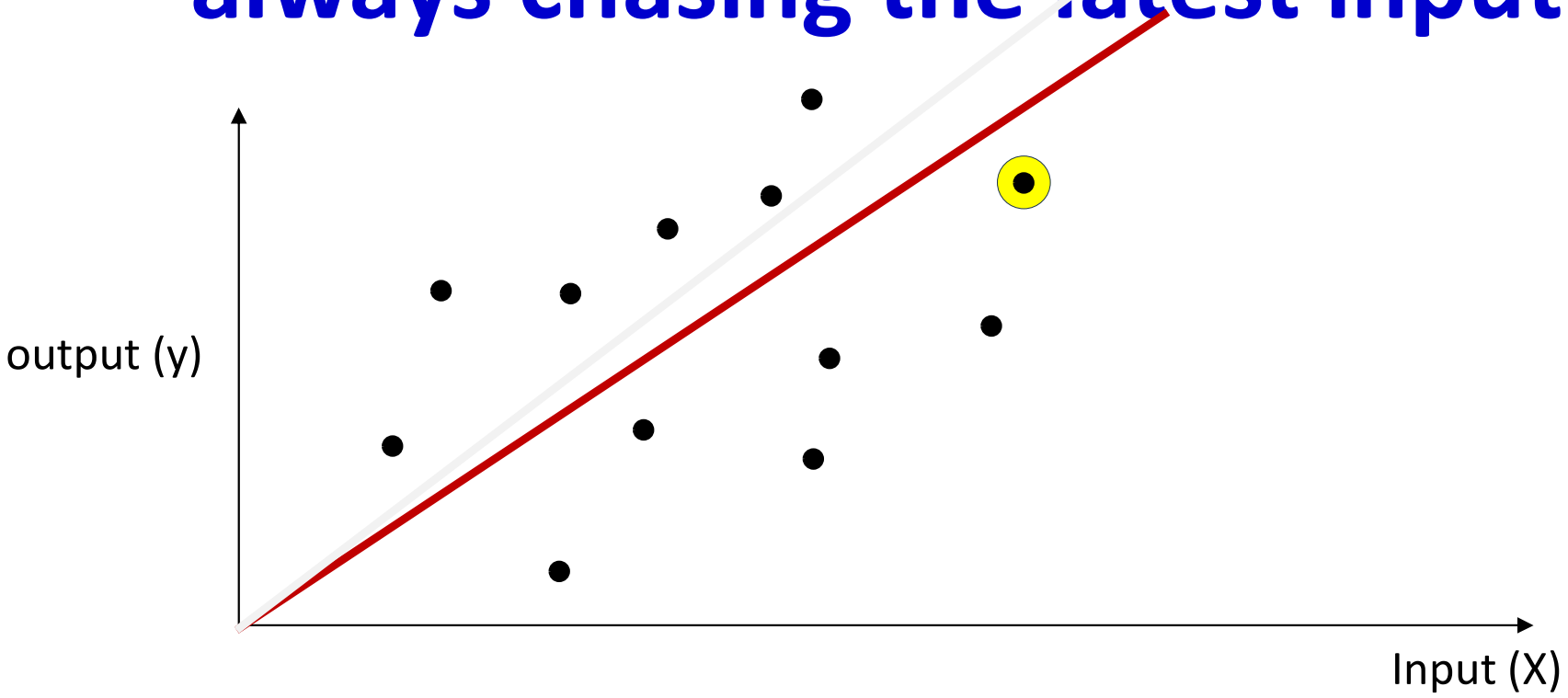
- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - Shrink the learning rate with iterations
  - With increasing iterations, it will swing less and less towards the new point

Incremental learning runs the risk of always chasing the latest input



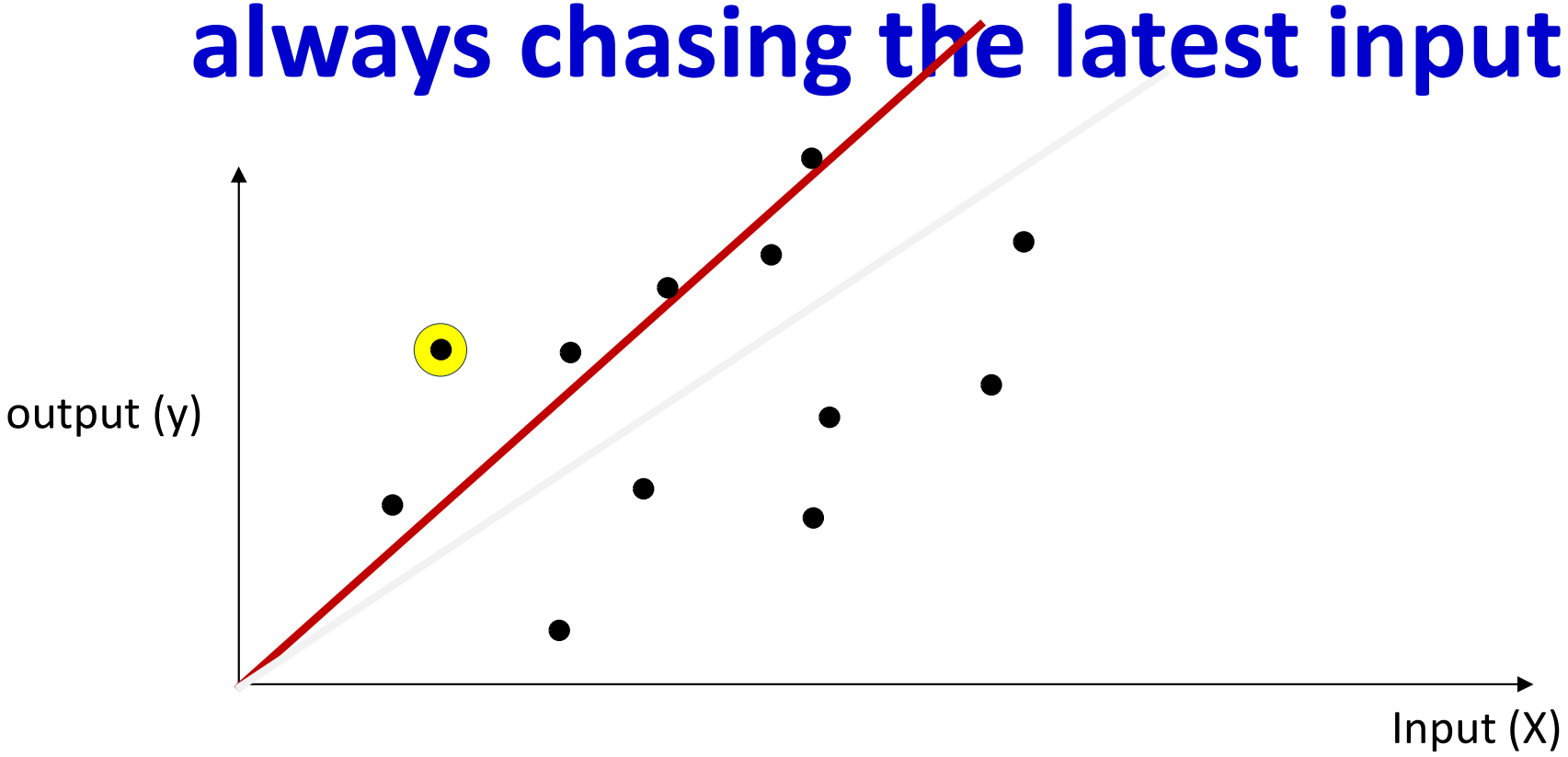
- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - Shrink the learning rate with iterations
  - With increasing iterations, it will swing less and less towards the new point

Incremental learning runs the risk of always chasing the latest input



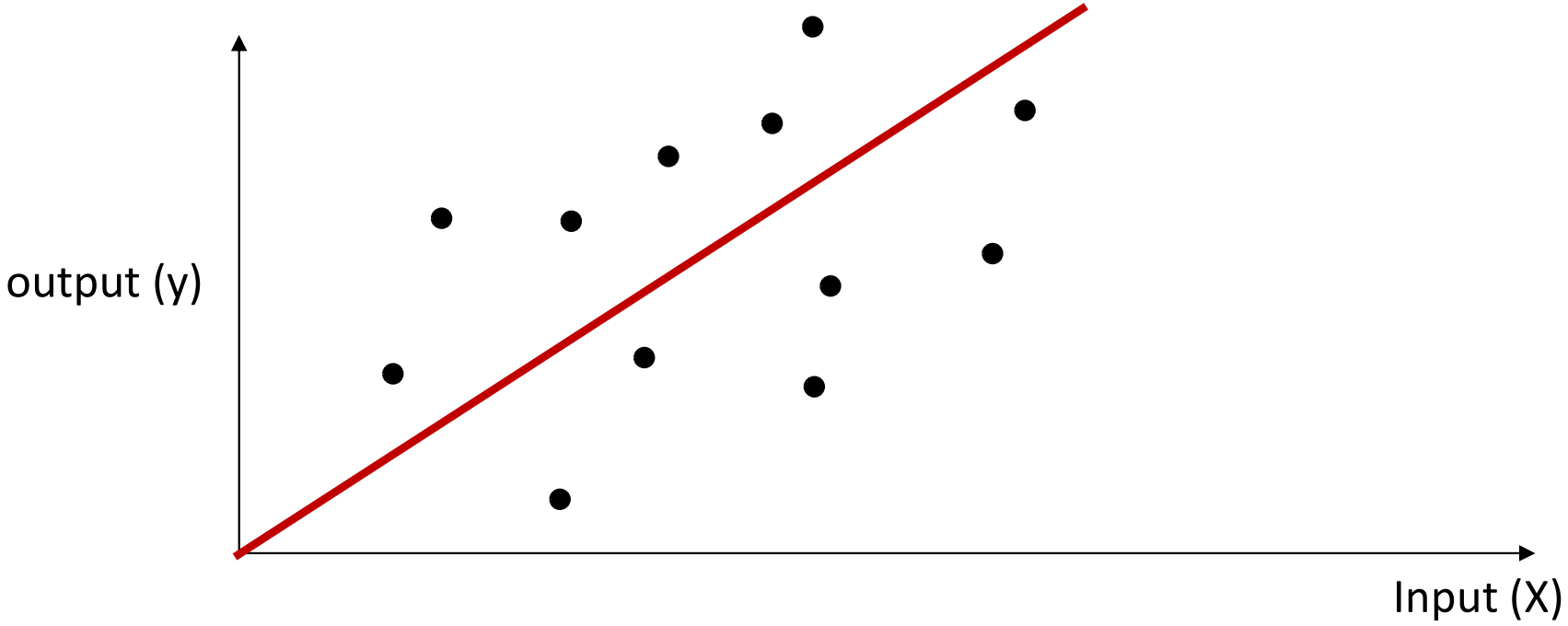
- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - Shrink the learning rate with iterations
  - With increasing iterations, it will swing less and less towards the new point

Incremental learning runs the risk of always chasing the latest input



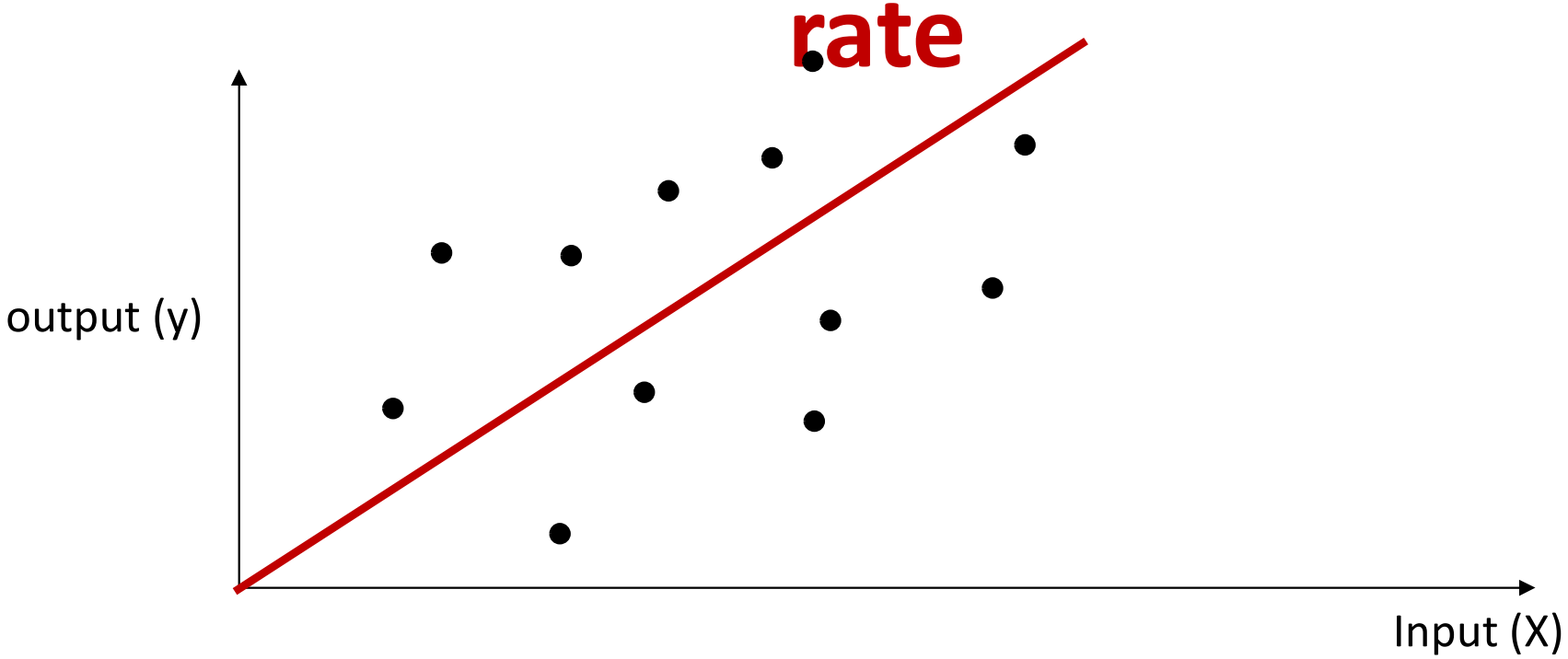
- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - Shrink the learning rate with iterations
  - With increasing iterations, it will swing less and less towards the new point

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
  - 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 learning caveat: learning rate



- **Incremental learning:** Update the model to always minimize the error on the latest instance
  - **Caveat:** We must *shrink* the learning rate with iterations for convergence
    - Correction for individual instances with the eventual miniscule learning rates will not modify the function

Incremental Update: Stochastic Gradient Descent

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

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

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

Randomize input order

Learning rate reduces with j

- 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 (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
  - $\left|f\left(W^{(k)}\right) - f(W^*)\right| < \epsilon$
  - 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,

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

  - 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,

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

  - 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 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 $\mathbb{E}\left[\|\hat{g}_t\|^2\right] \leq G^2$ 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

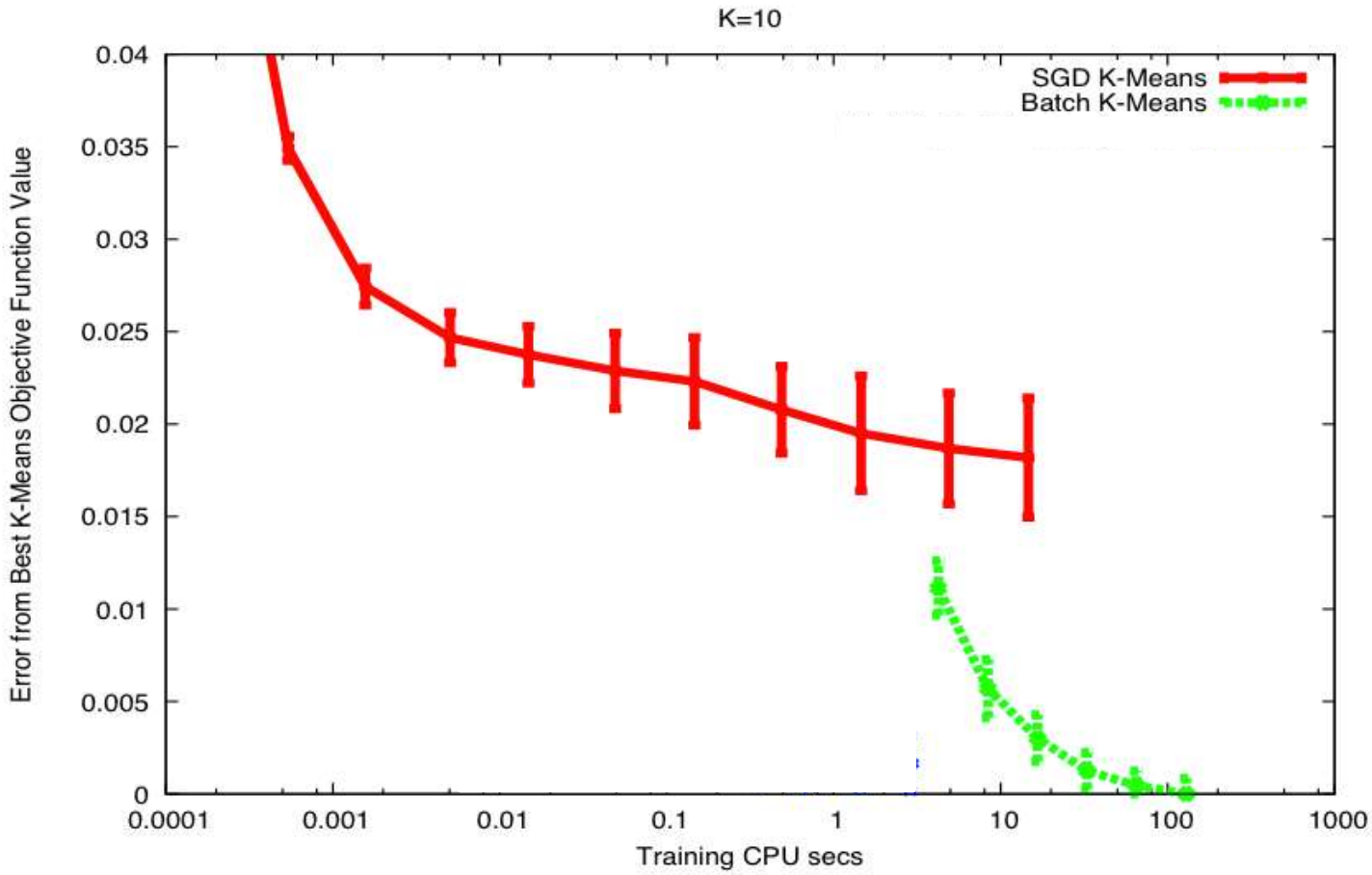
Polynomial Decay Averaging:

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

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..

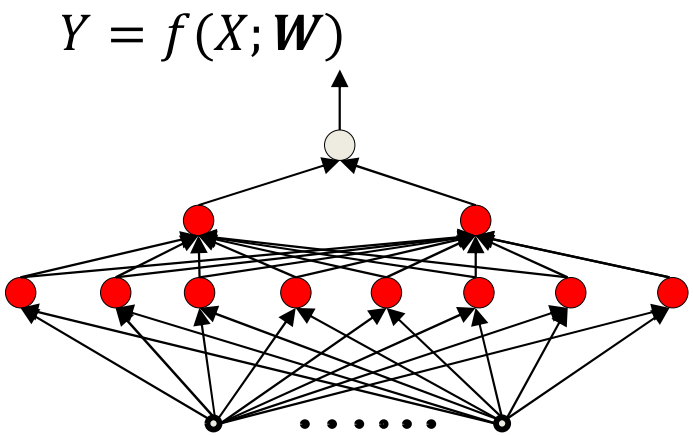
Poll 2

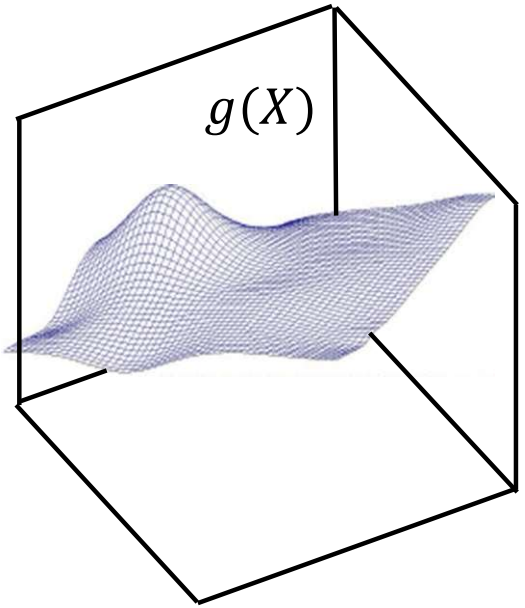
- Select all that are true
  - SGD is an online version of batch updates
  - SGD can have oscillatory behavior if we do not randomize the order of inputs
  - SGD can converge faster than batch updates, but arrive at poorer optima
  - SGD convergence to the global minimum can only be guaranteed if step sizes shrink across iterations, but sum to infinity in the limit

Poll 2

- Select all that are true
  - **SGD is an online version of batch updates**
  - **SGD can have oscillatory behavior if we do not randomize the order of inputs**
  - **SGD can converge faster than batch updates, but arrive at poorer optima**
  - **SGD convergence to the global minimum can only be guaranteed if step sizes shrink across iterations, but sum to infinity in the limit**

Recall: Modelling a function



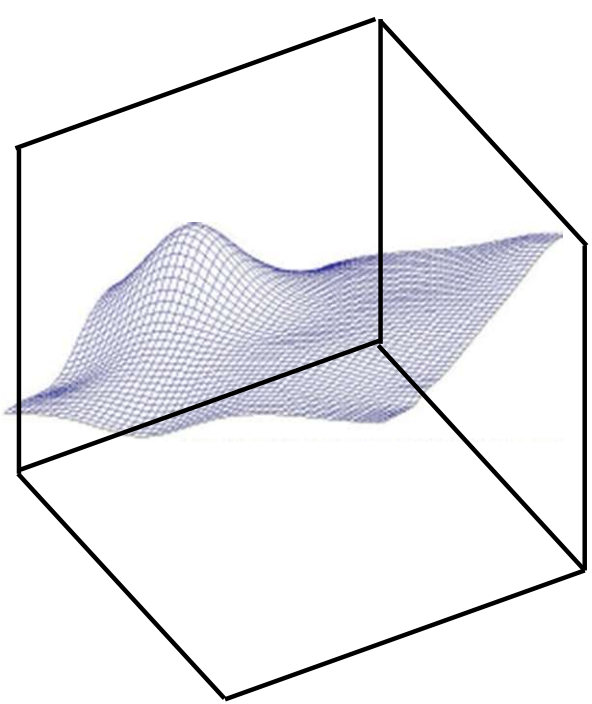


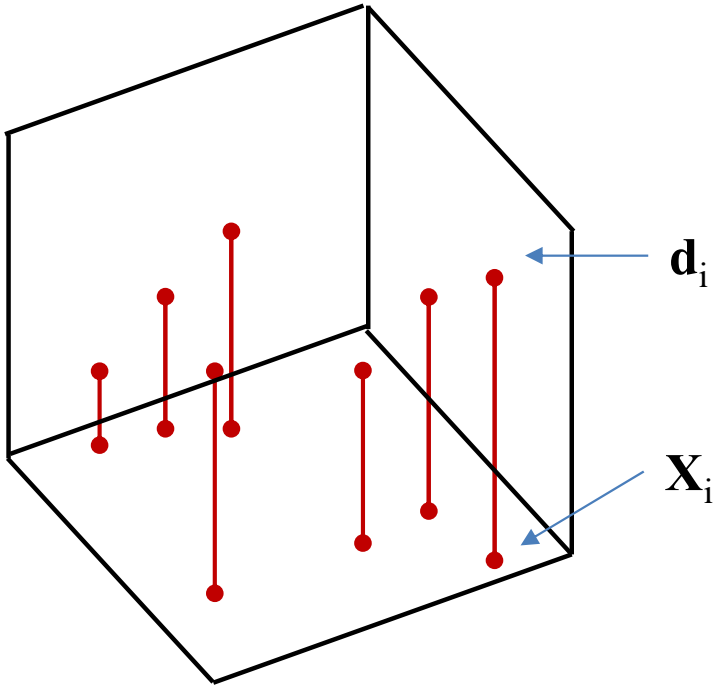
- To learn a network $f(X; \boldsymbol{W})$ to model a function $g(X)$ we minimize the *expected divergence*

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

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

Recall: The *Empirical* risk





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

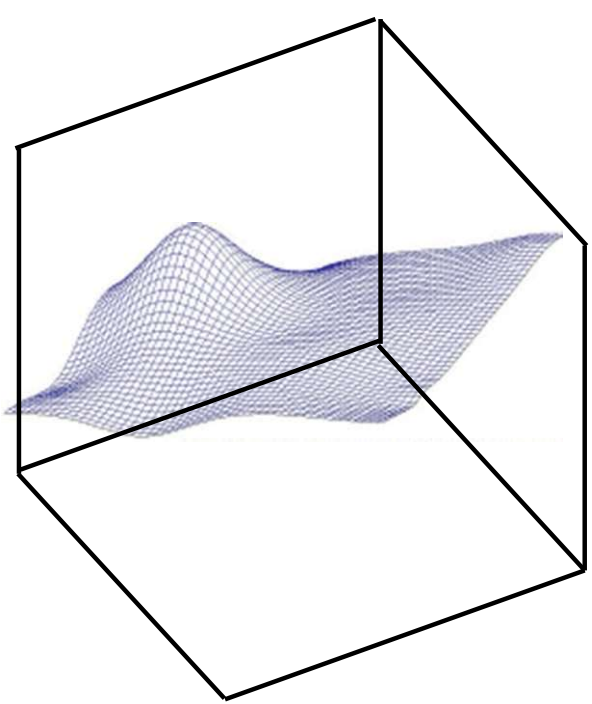
$$Loss(W) = \frac{1}{N}\sum_{i=1}^{N} div(f(X_i; W), d_i)$$

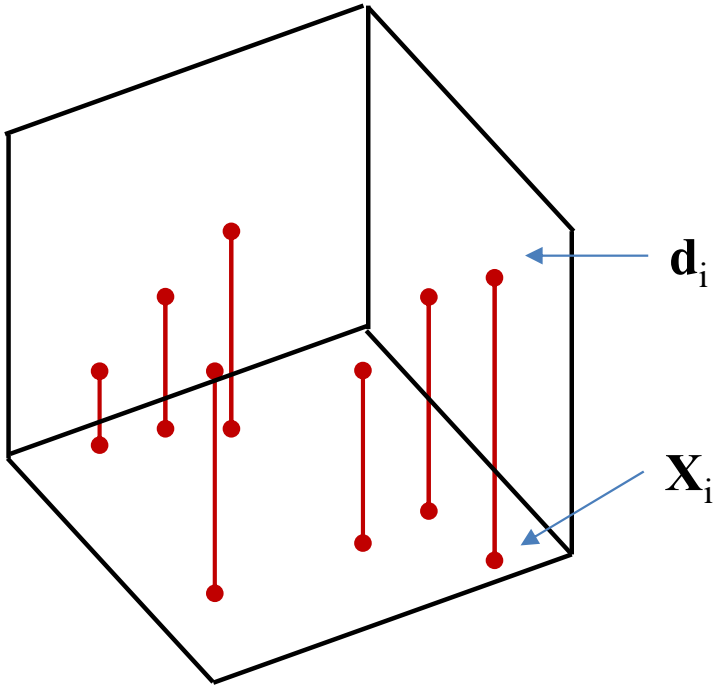
$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}}\, Loss(W)$$

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

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

Recall: The *Empirical* risk





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

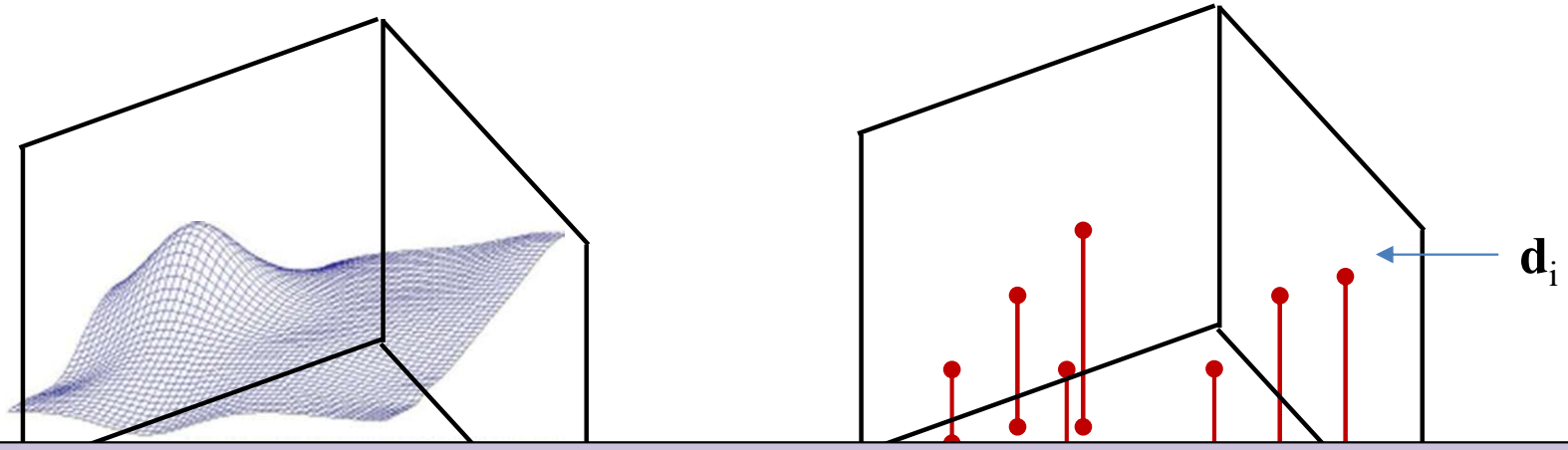
$$Loss(W) = \frac{1}{N}\sum_{i=1}^{N} div(f(X_i; W), d_i)$$

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

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

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

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 divergence

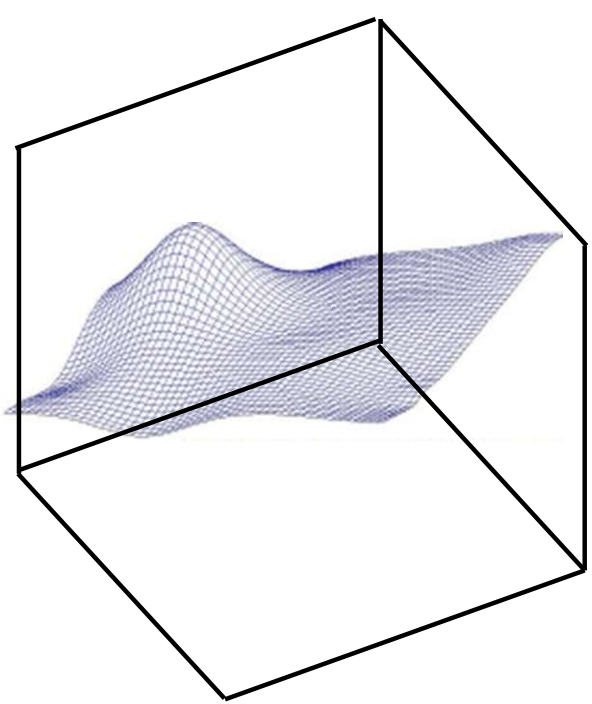
$$Loss(W) = \frac{1}{N}\sum_{i=1}^{N} div(f(X_i; W), d_i)$$

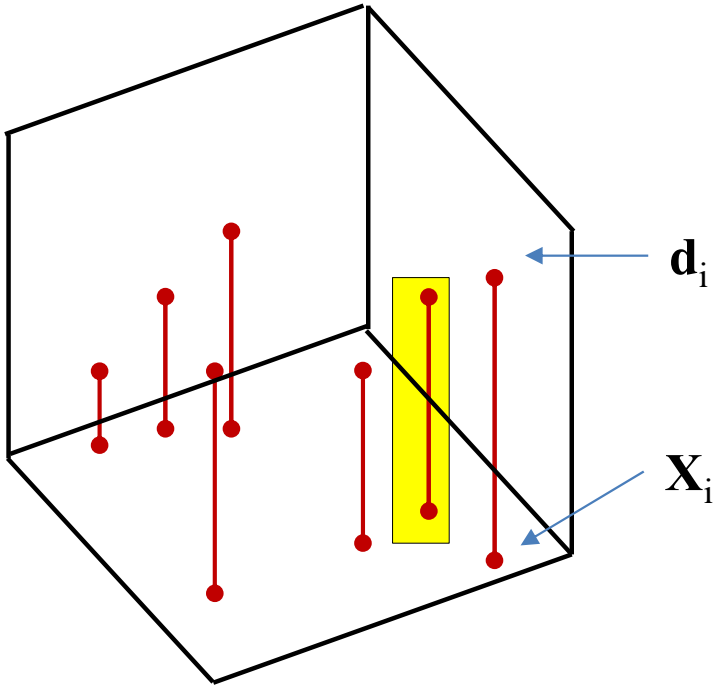
The empirical risk is an *unbiased* estimate of the expected divergence

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

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

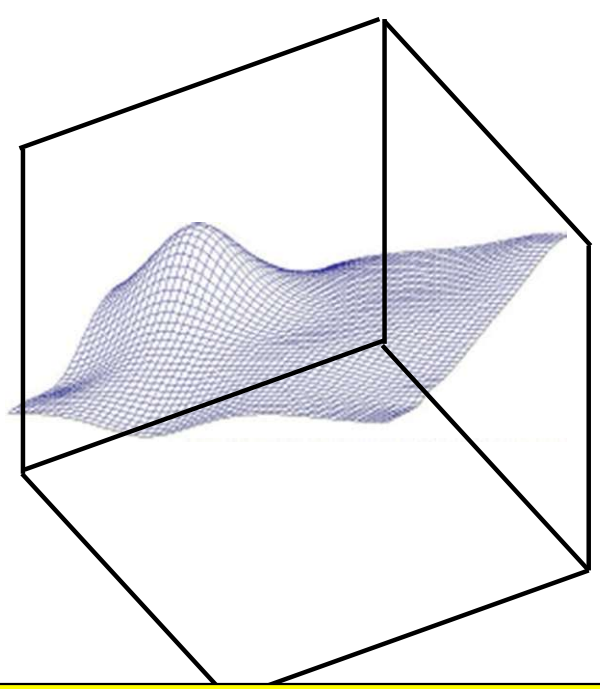
SGD

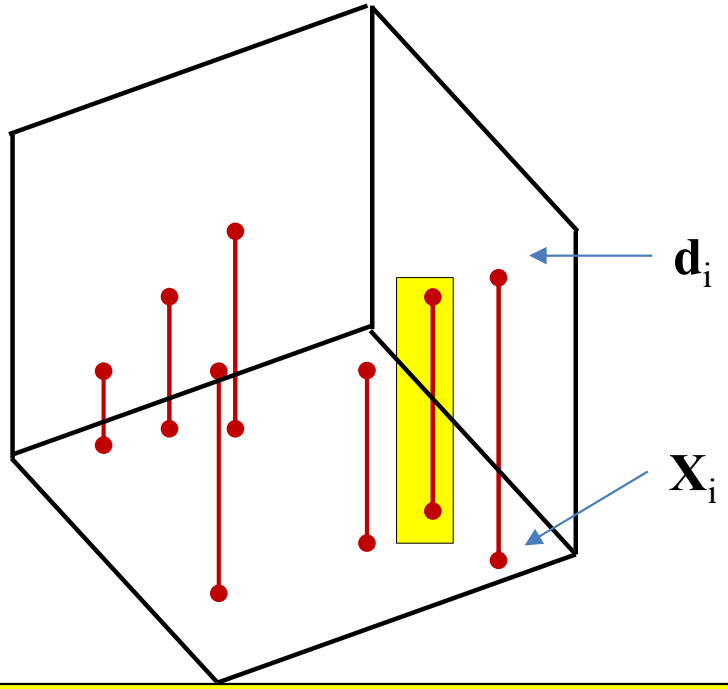




- At each iteration, **SGD** focuses on the divergence of a ***single*** sample $div(f(X_i; W), d_i)$
- The *expected value* of the *sample error* is ***still*** the *expected divergence* $E\left[div\left(f(X; W), g(X)\right)\right]$

SGD

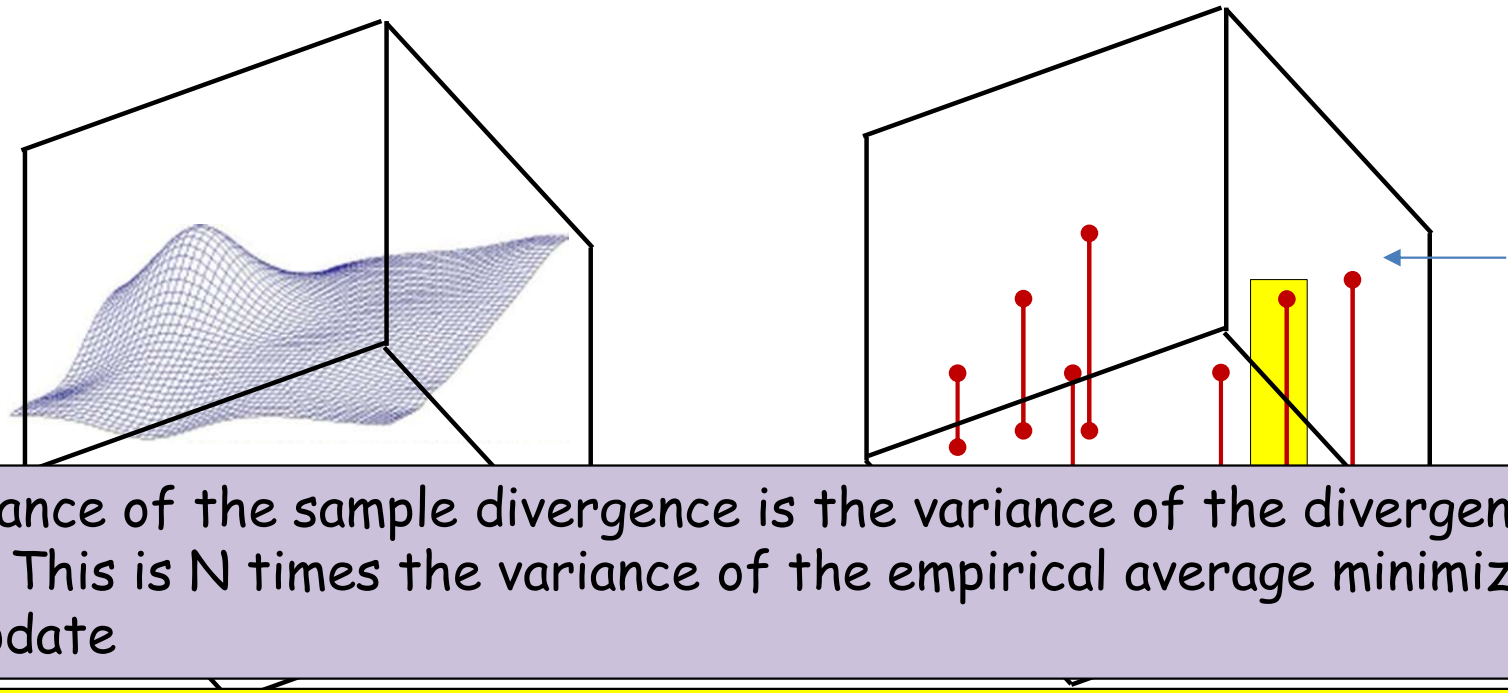




The sample divergence is also an *unbiased* estimate of the expected error

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

SGD

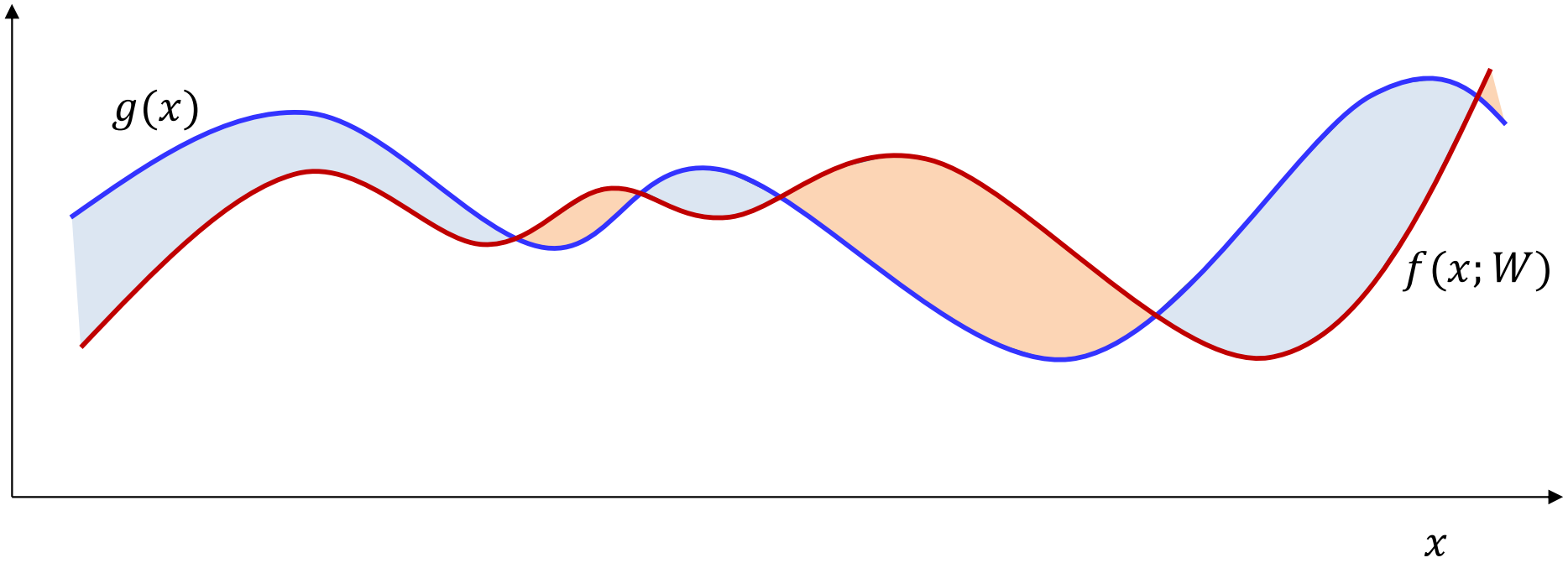


The variance of the sample divergence is the variance of the divergence itself: var(div). This is N times the variance of the empirical average minimized by batch update

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

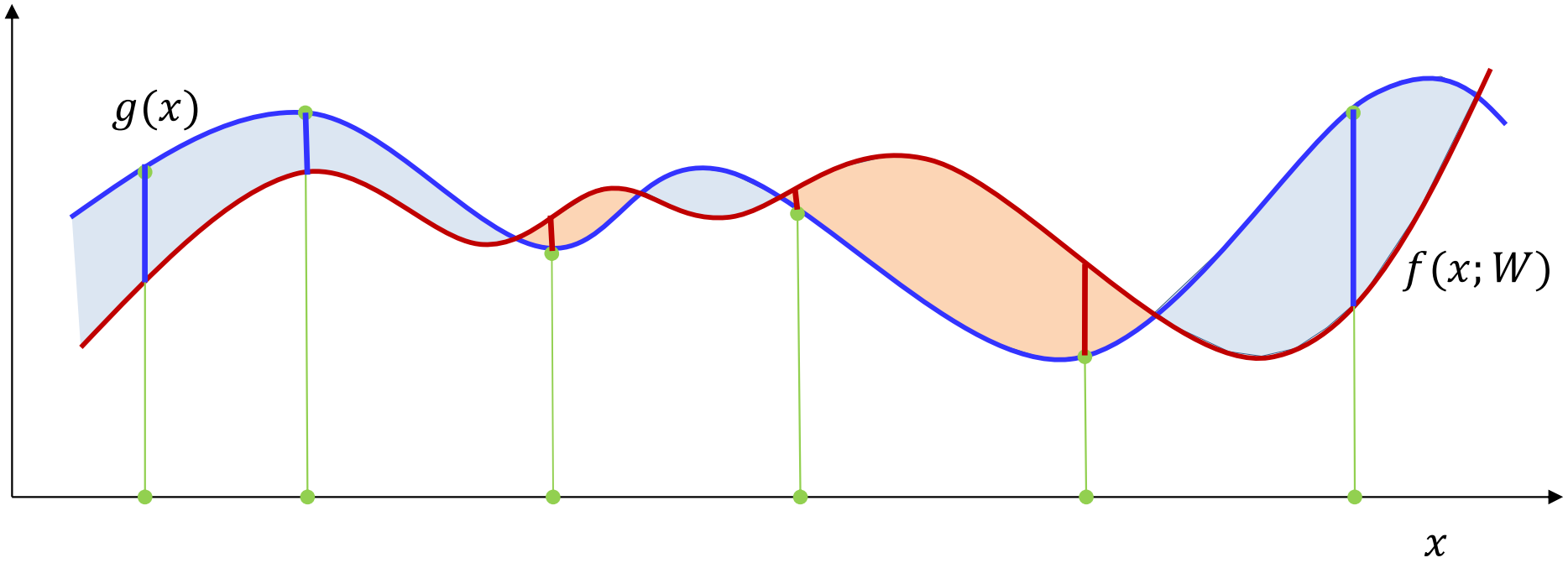
- At each iteration, **SGD** focuses on the divergence of a ***single*** sample $div(f(X_i; W), d_i)$
- The *expected value* of the *sample error* is ***still*** the *expected divergence* $E\left[div\left(f(X; W), g(X)\right)\right]$

Explaining the variance



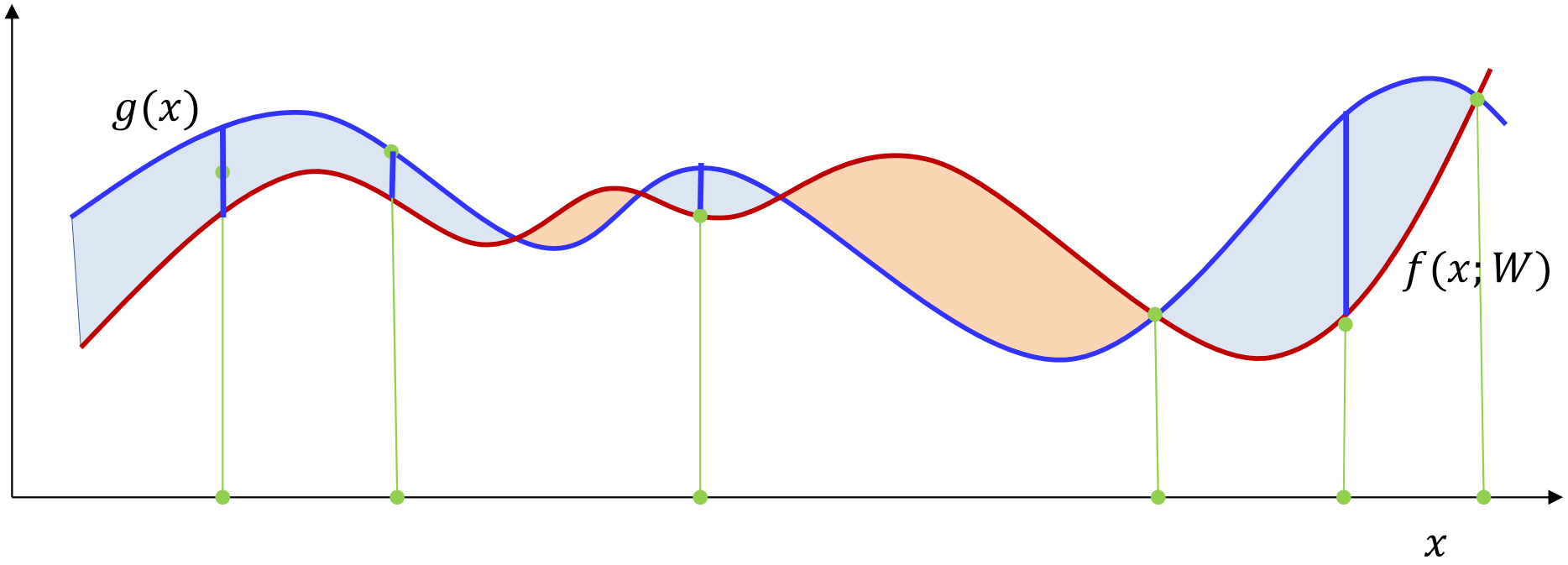
- The blue curve is the function being approximated
- The red curve is the approximation by the model at a given W
- The heights of the shaded regions represent the point-by-point error
  - The divergence is a function of the error
  - We want to find the W that minimizes the average divergence

Explaining the variance



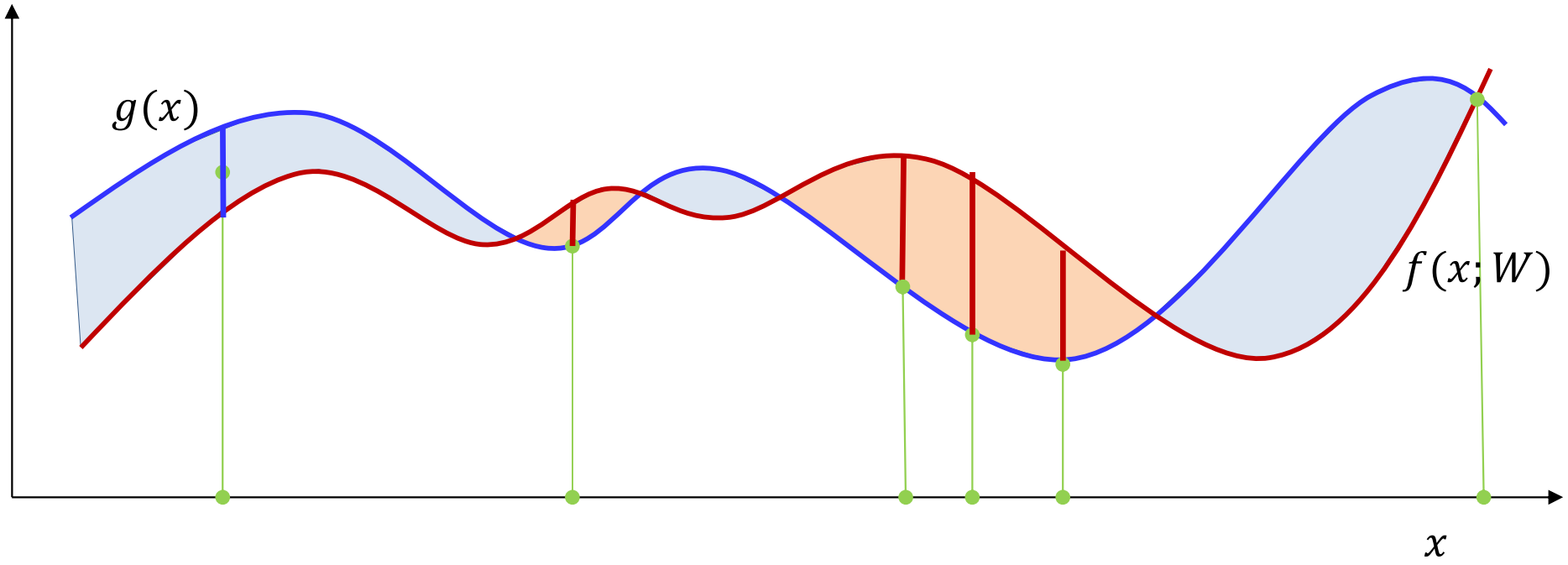
- Sample estimate approximates the shaded area with the average length of the error lines

Explaining the variance



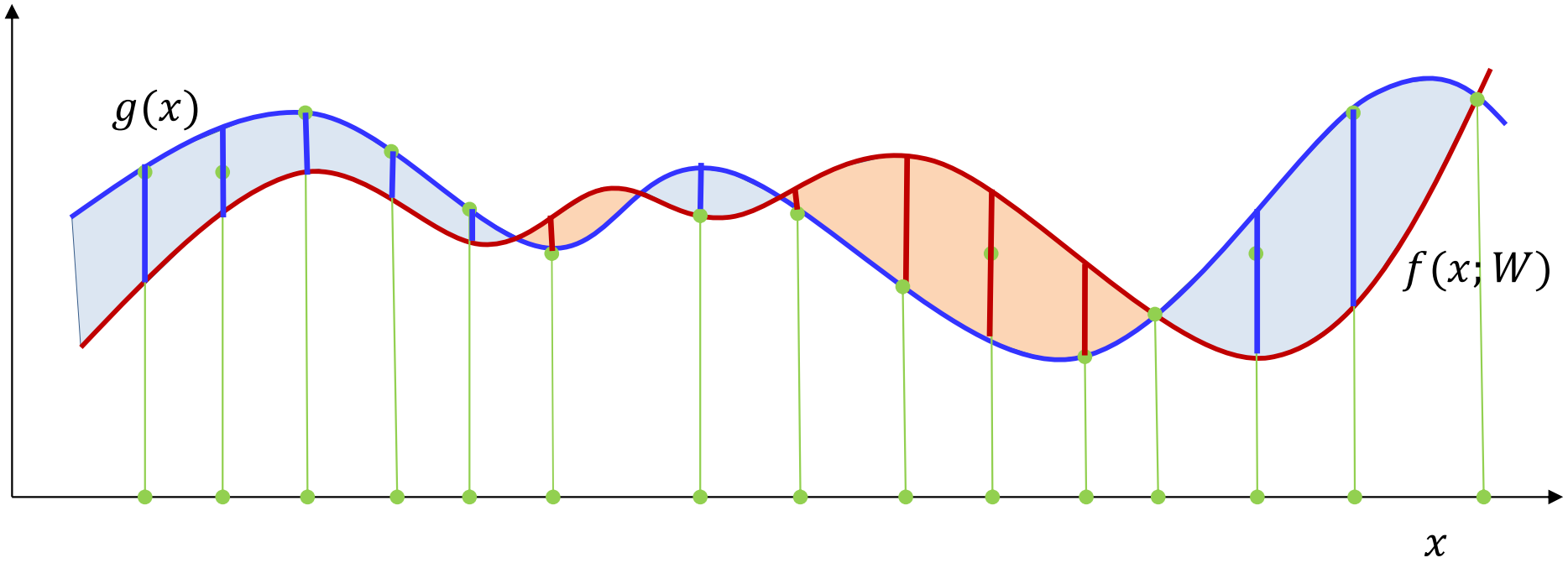
- Sample estimate approximates the shaded area with the average length of the error lines
- This average length will change with position of the samples

Explaining the variance



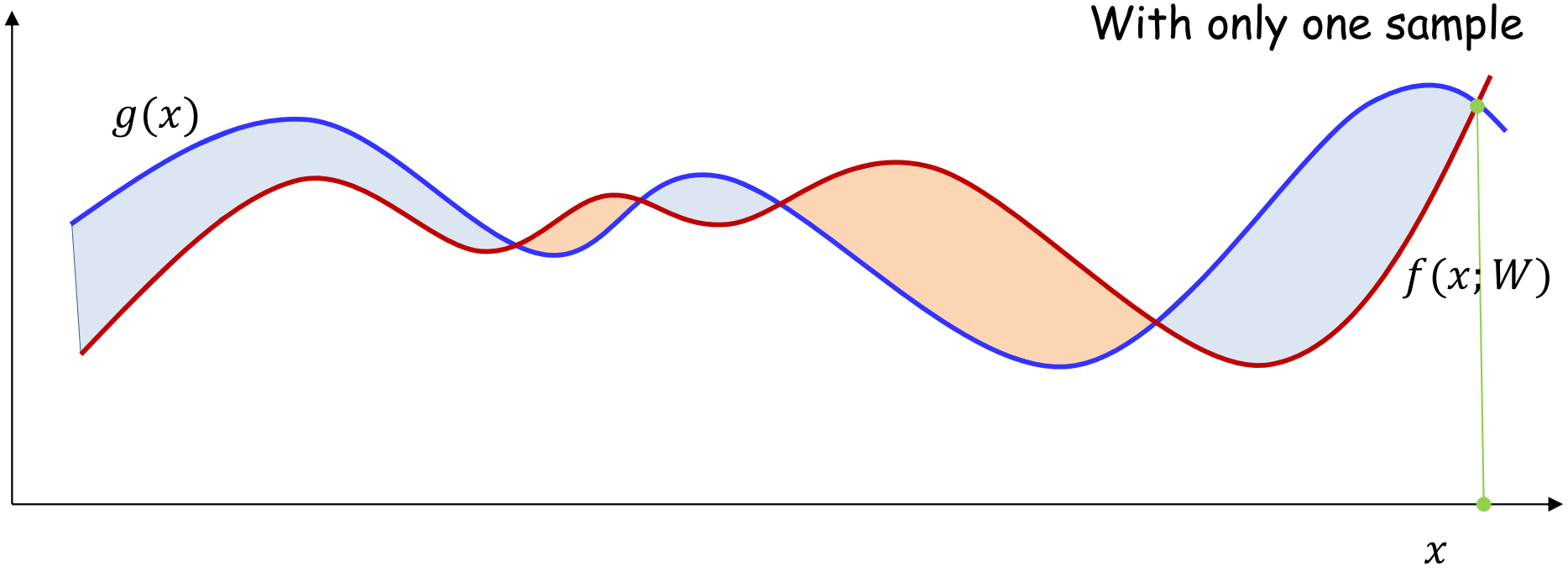
- Sample estimate approximates the shaded area with the average length of the error lines
- This average length will change with position of the samples

Explaining the variance



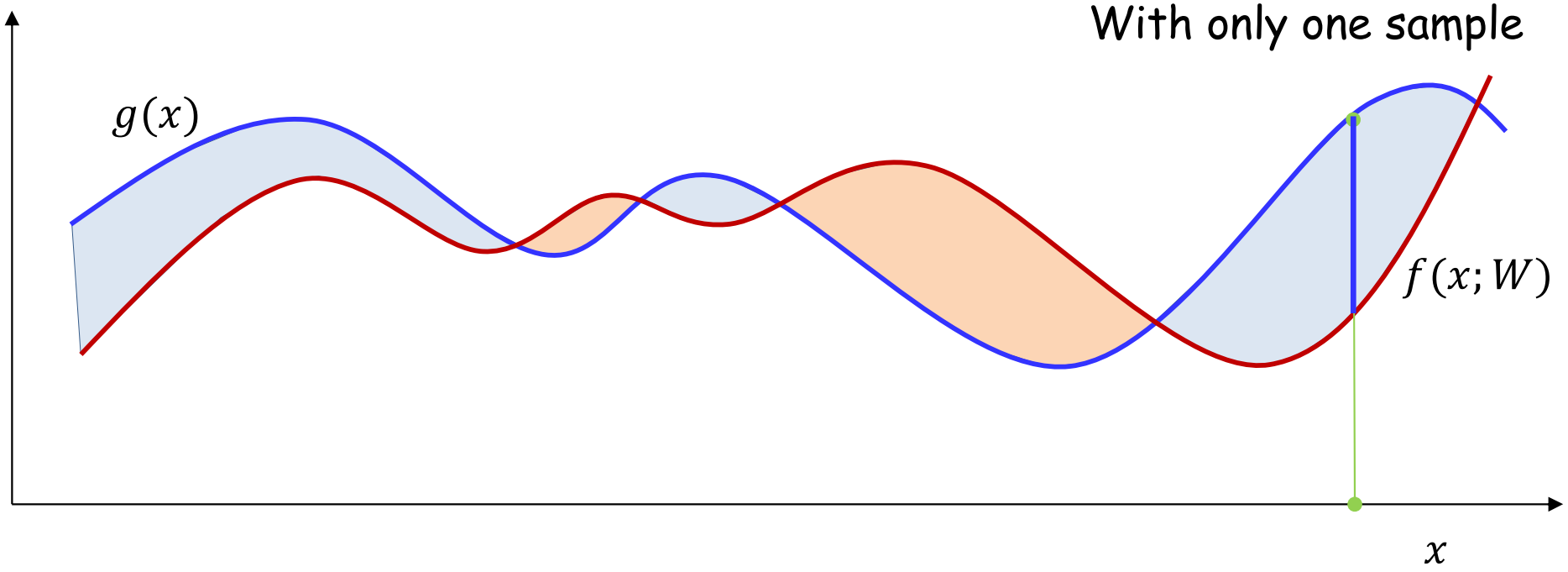
- Having more samples makes the estimate more robust to changes in the position of samples
  - The variance of the estimate is smaller

Explaining the variance



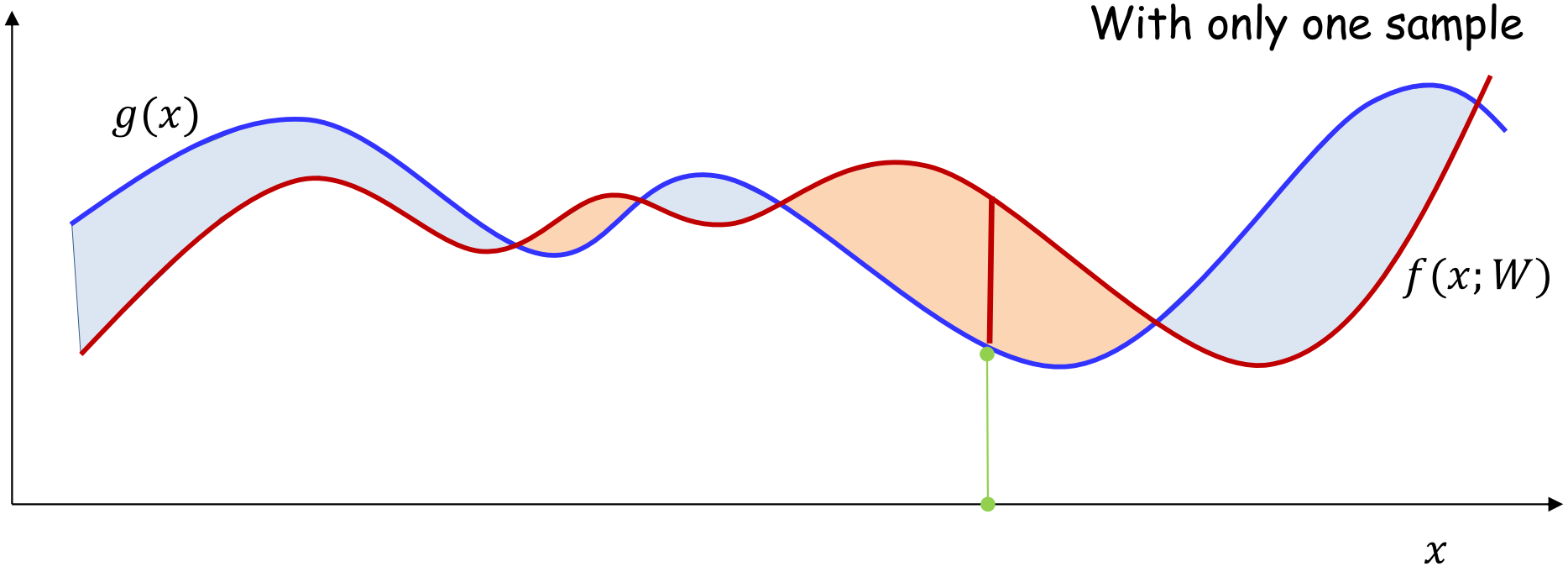
- Having very few samples makes the estimate swing wildly with the sample position
  - Since our estimator learns the W to minimize this estimate, the learned W too can swing wildly

Explaining the variance



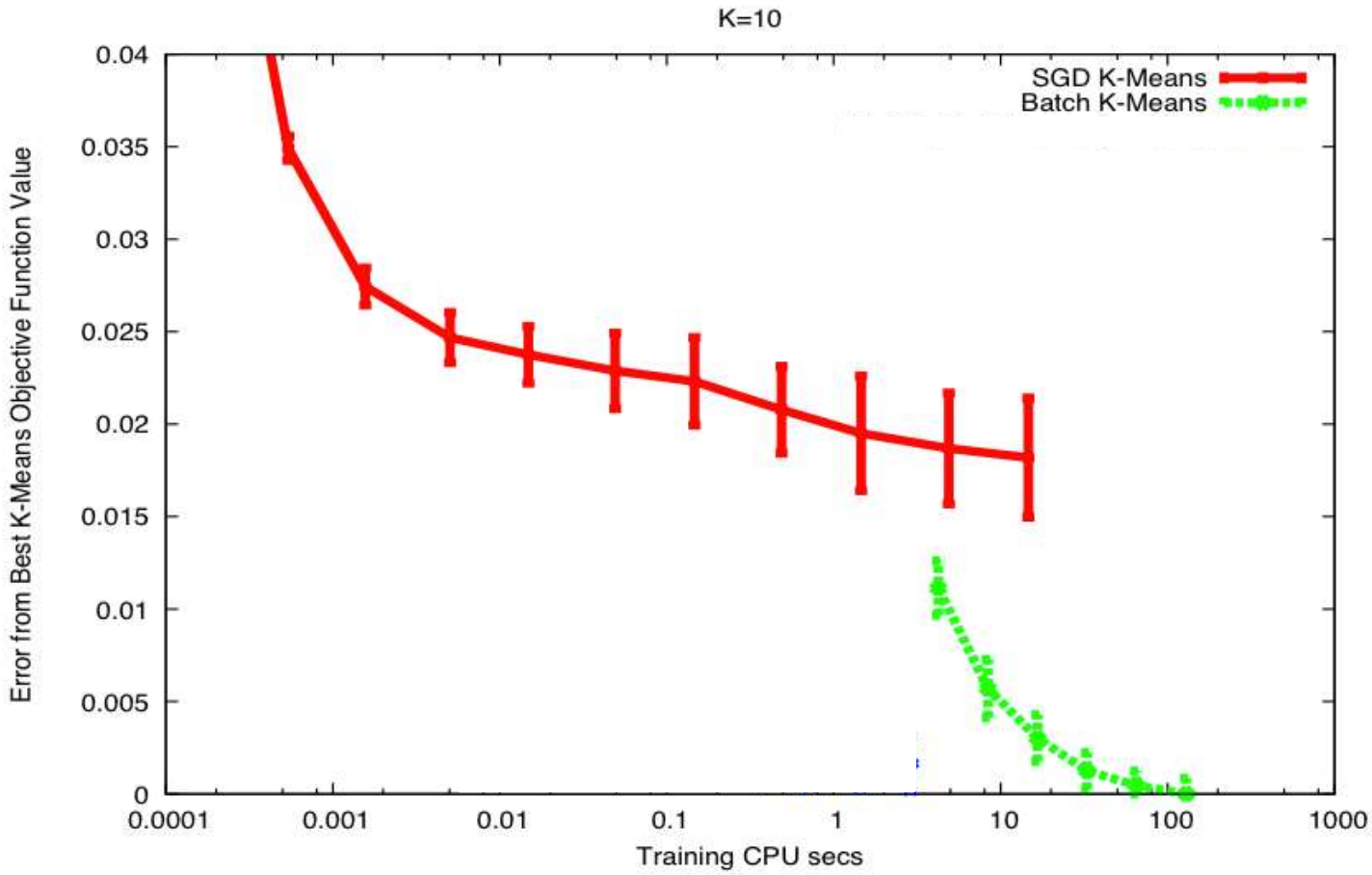
- Having very few samples makes the estimate swing wildly with the sample position
  - Since our estimator learns the W to minimize this estimate, the learned W too can swing wildly

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



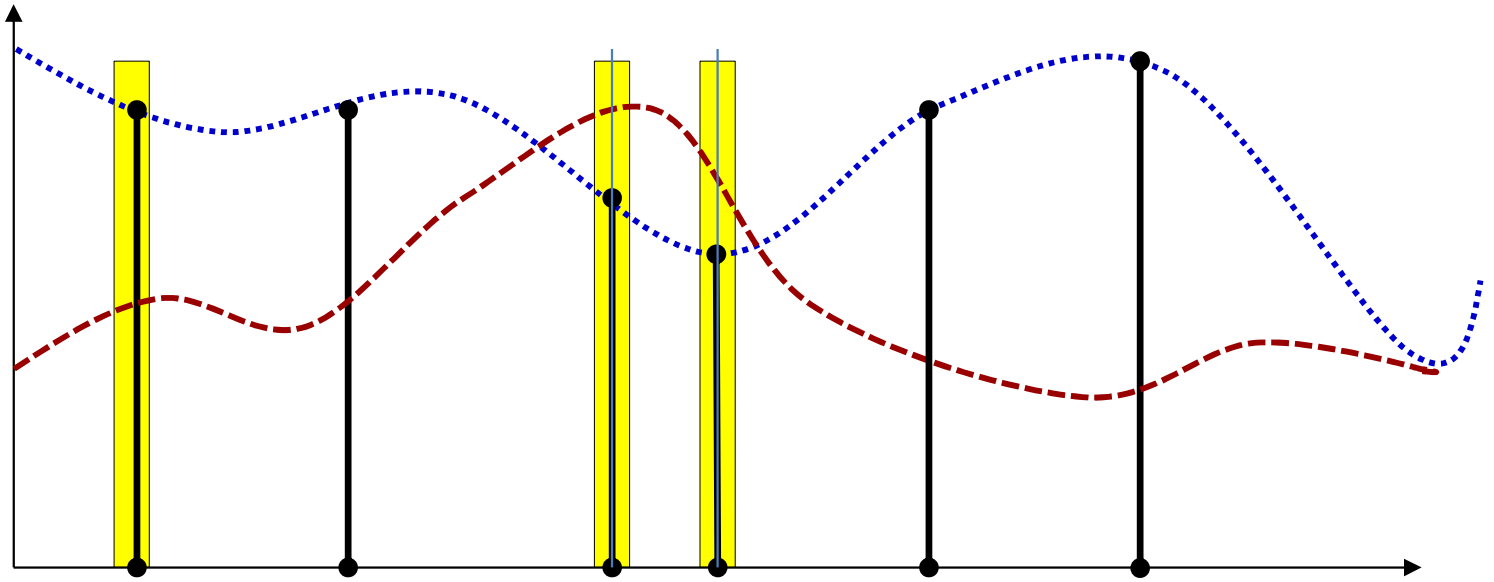
- A simpler problem: K-means
- Note: SGD converges faster
- But also has large variation between runs

SGD vs batch

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

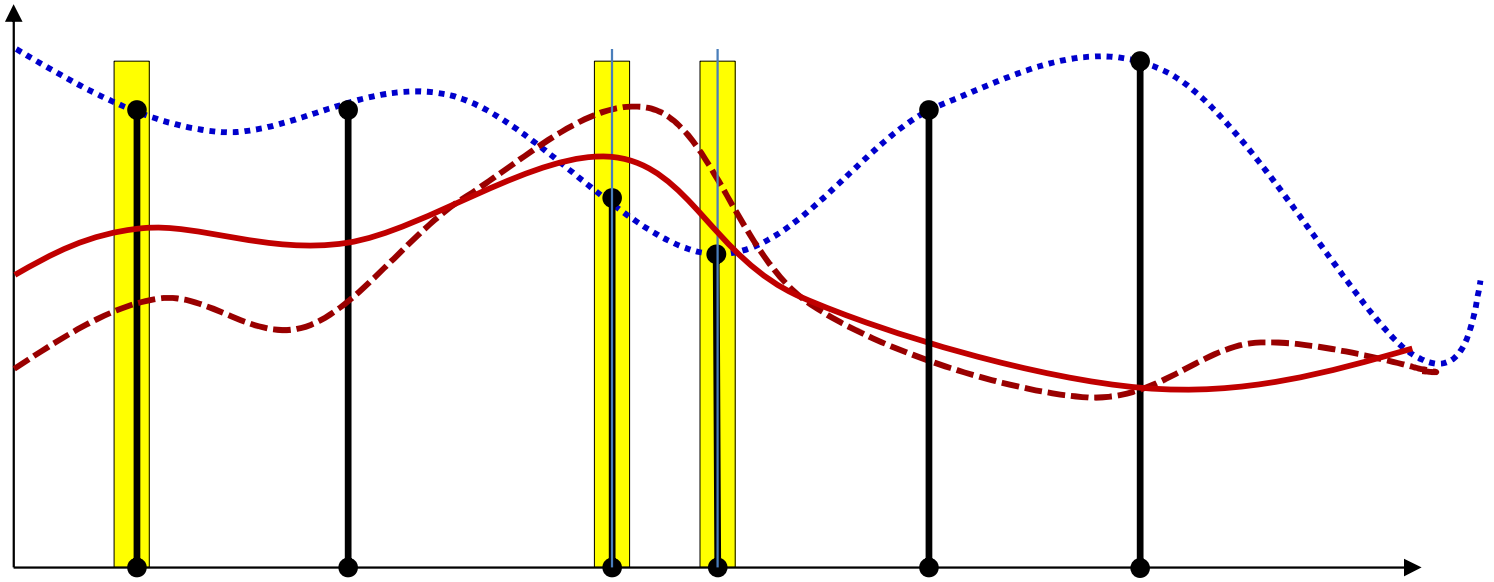
- But also provides significantly quicker updates than batch
- Is there a good medium?

Alternative: Mini-batch update



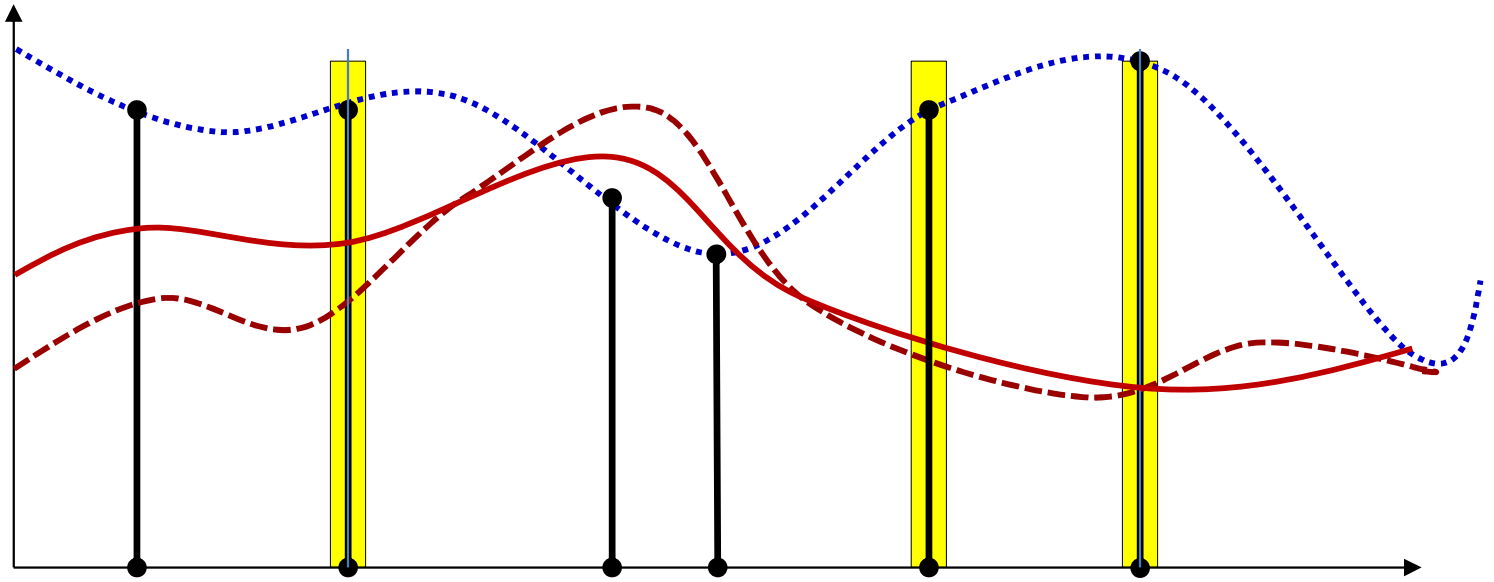
- Alternative: adjust the function at a small, randomly chosen subset of points
  - Keep adjustments small
  - If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data

Alternative: Mini-batch update



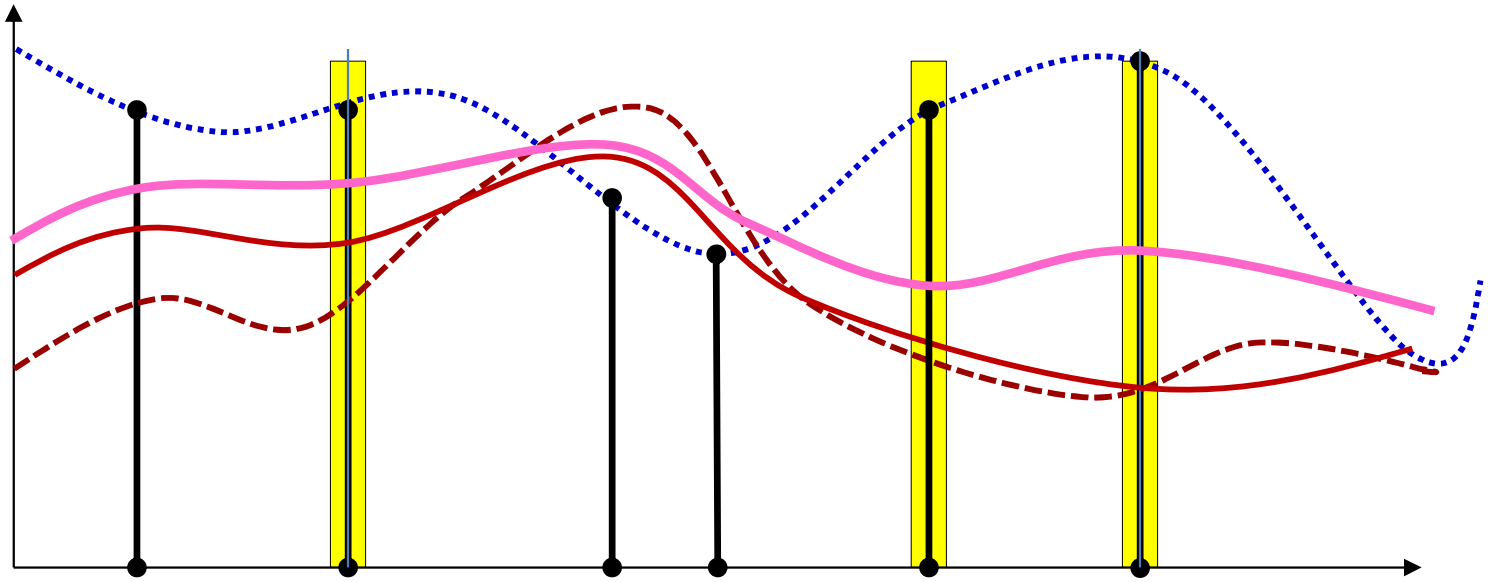
- Alternative: adjust the function at a small, randomly chosen subset of points
  - Keep adjustments small
  - If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data

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 $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$
- 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 $t = 1 : b : T$
    - $j = j + 1$
    - 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:

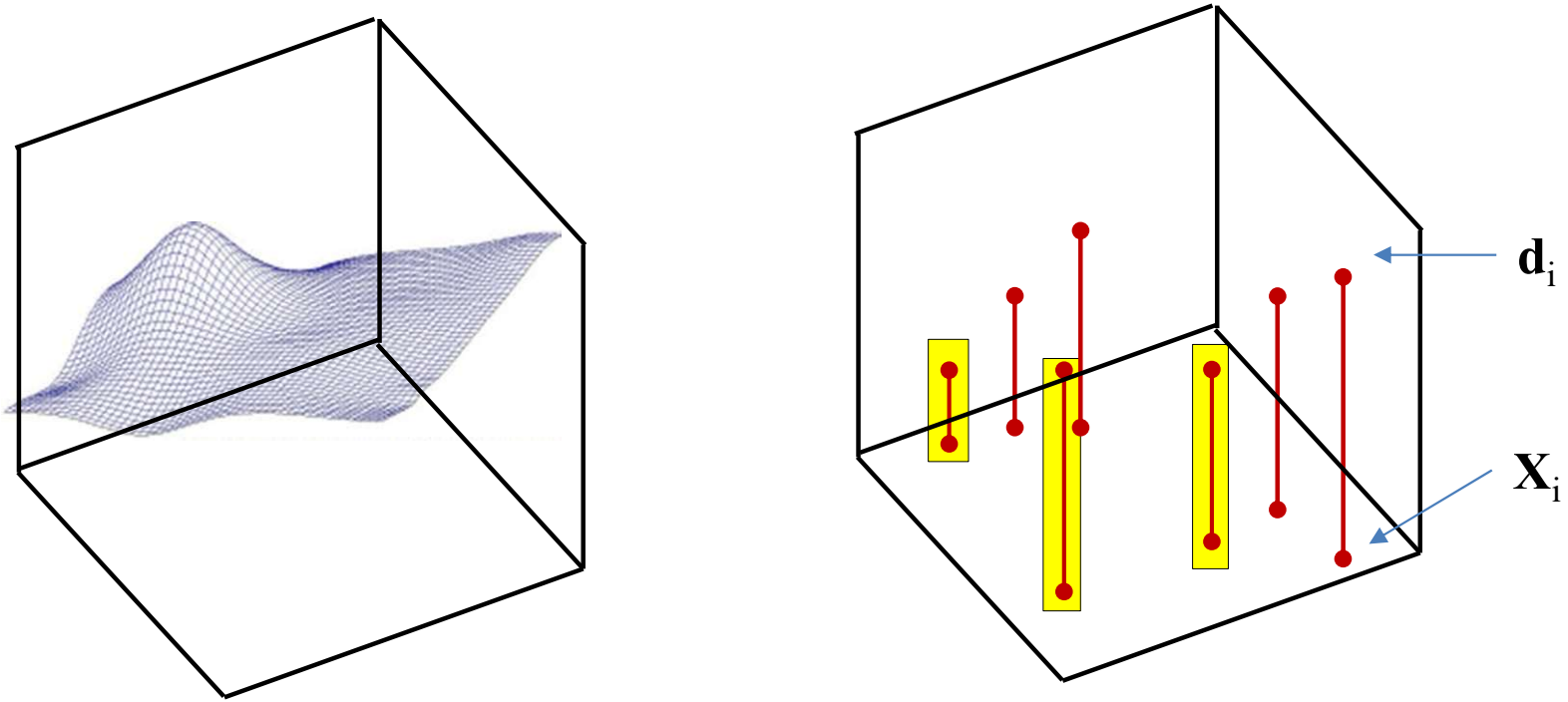
$$W_k = W_k - \eta_j \Delta W_k$$

Mini-batch size

Shrinking step size

- Until Err has converged

Mini Batches



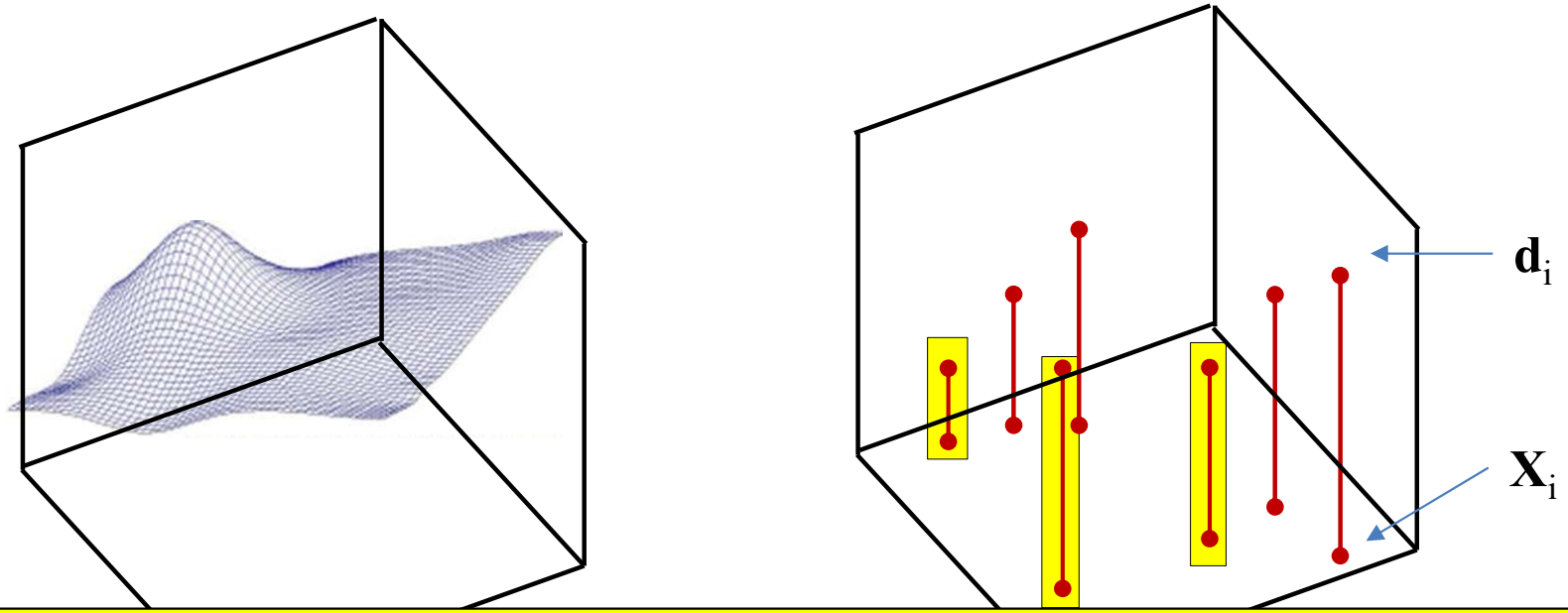
- Mini-batch updates compute and minimize a *batch loss*

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

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

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

Mini Batches



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

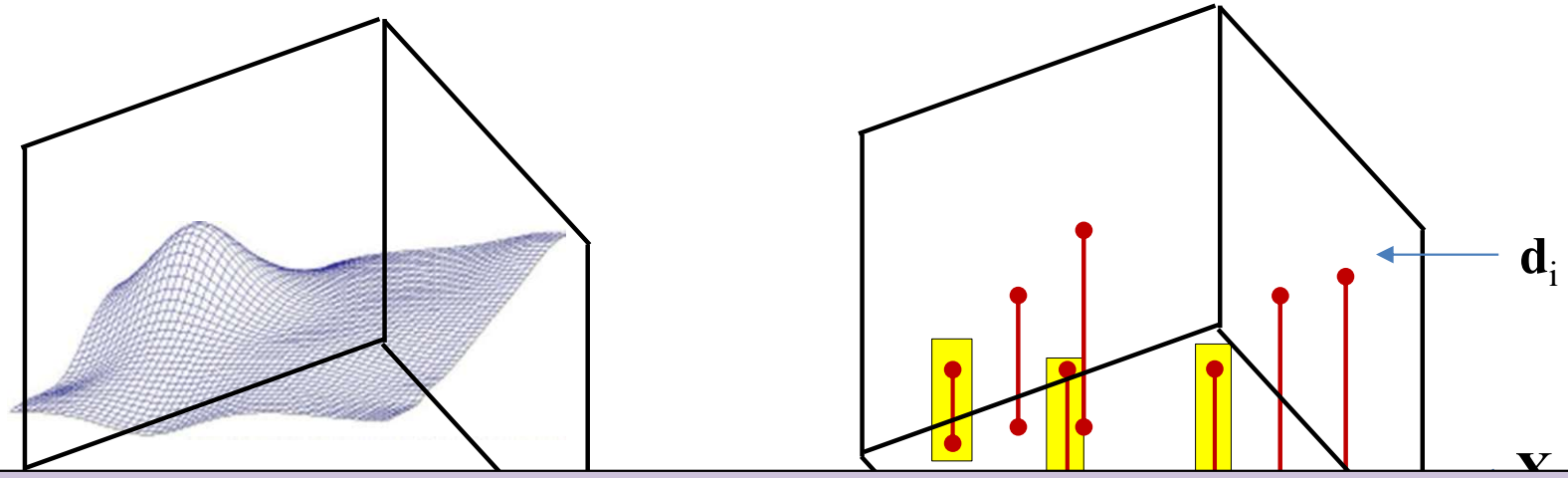
- Mini-batch updates compute and minimize a *batch loss*

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

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

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

Mini Batches



The variance of the minibatch loss: var(BatchLoss) = 1/b var(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*

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

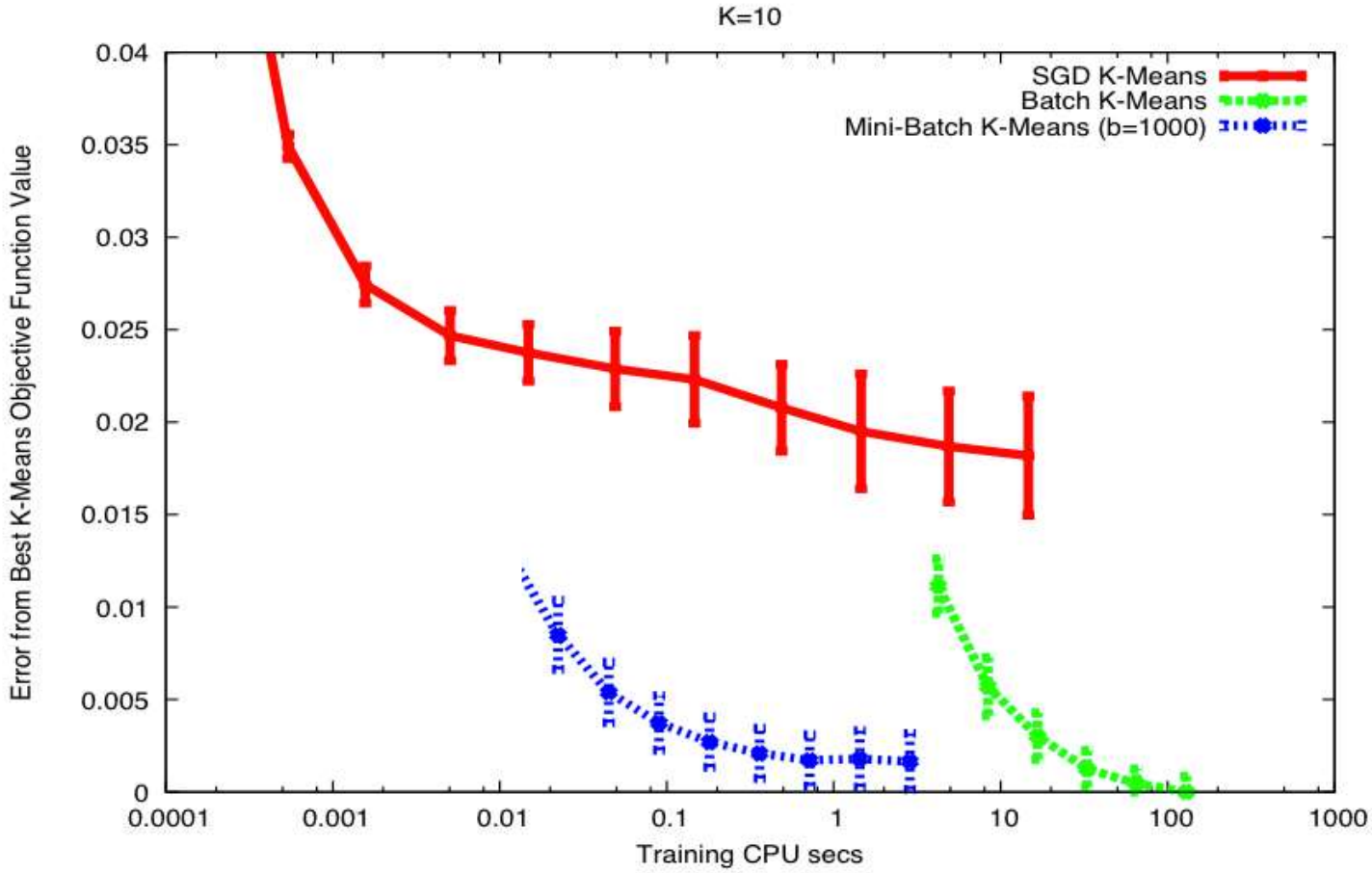
- The *expected value* of the *batch loss* is also the *expected divergence*

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

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

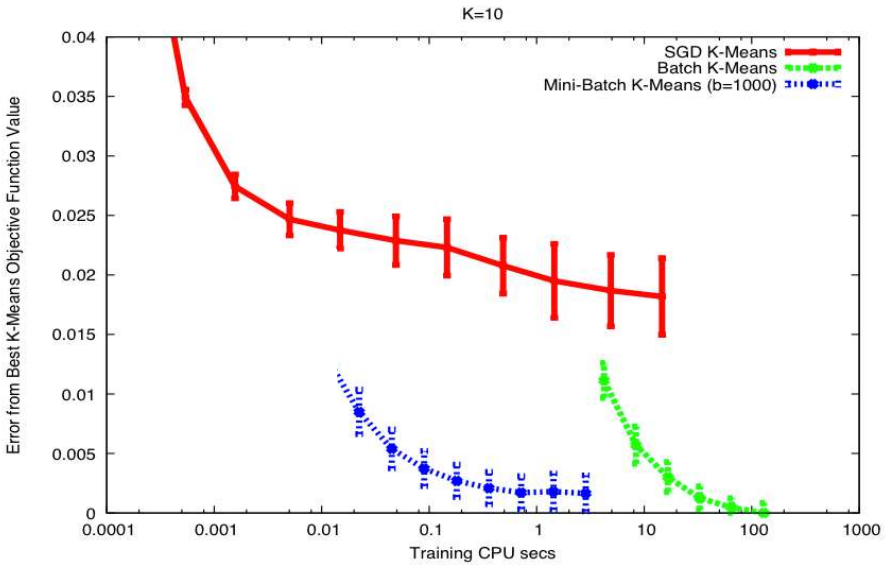
SGD example



- 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 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 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

Poll 3

- Select all that are true
  - Minibatch descent is an online version of batch updates
  - Minibatch descent is faster than SGD when the batch size is 1
  - The variance of minibatch updates decreases with batch size
  - Minibatch gradient approaches batch updates in variance, but SGD in efficiency when we use vector processing and large batches

Poll 3

- Select all that are true
  - **Minibatch descent is an online version of batch updates**
  - Minibatch descent is faster than SGD when the batch size is 1
  - **The variance of minibatch updates decreases with batch size**
  - **Minibatch gradient approaches batch updates in variance, but SGD in efficiency when we use vector processing and large batches**

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

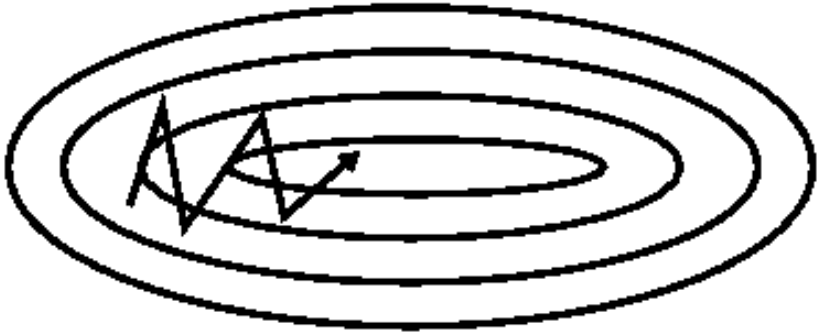
Moving on: Topics for the day

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

Recall: Momentum Update

Plain gradient update

With momentum



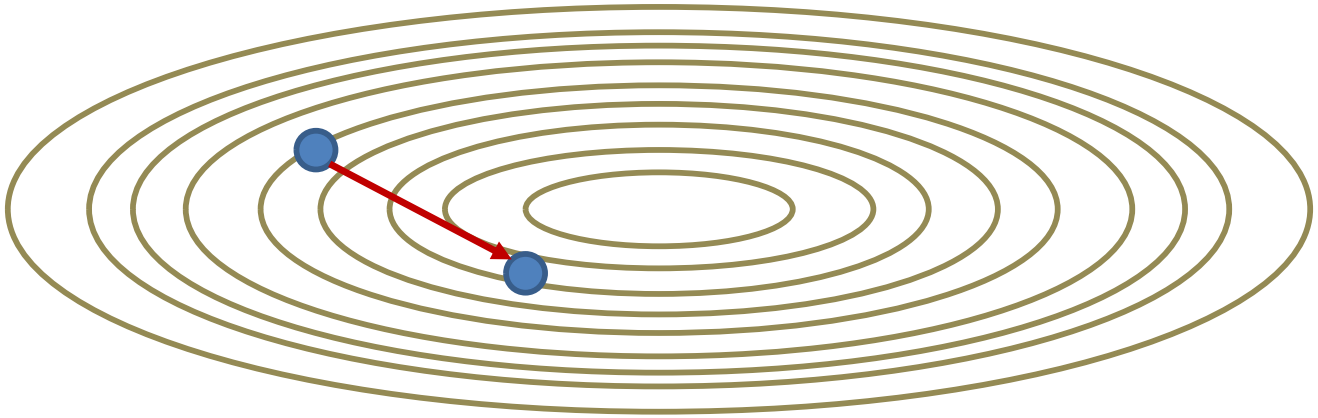
- The momentum method maintains a running average of all gradients until the *current* step

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

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

  - Typical β value is 0.9
- The running average steps
  - Get longer in directions where gradient retains the same sign
  - Become shorter in directions where the sign keeps flipping

Recall: Momentum Update

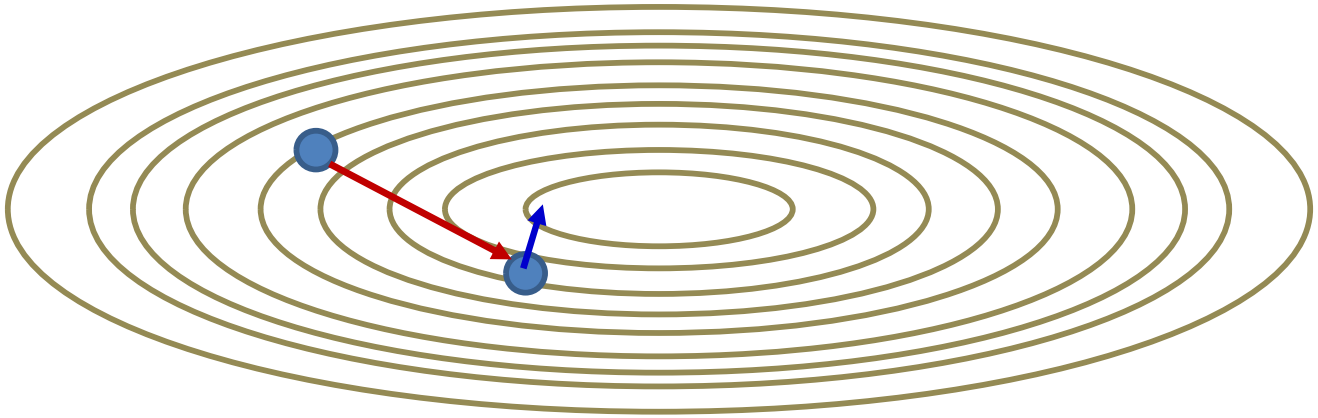


- The momentum method

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

- At any iteration, to compute the current step:

Recall: Momentum Update

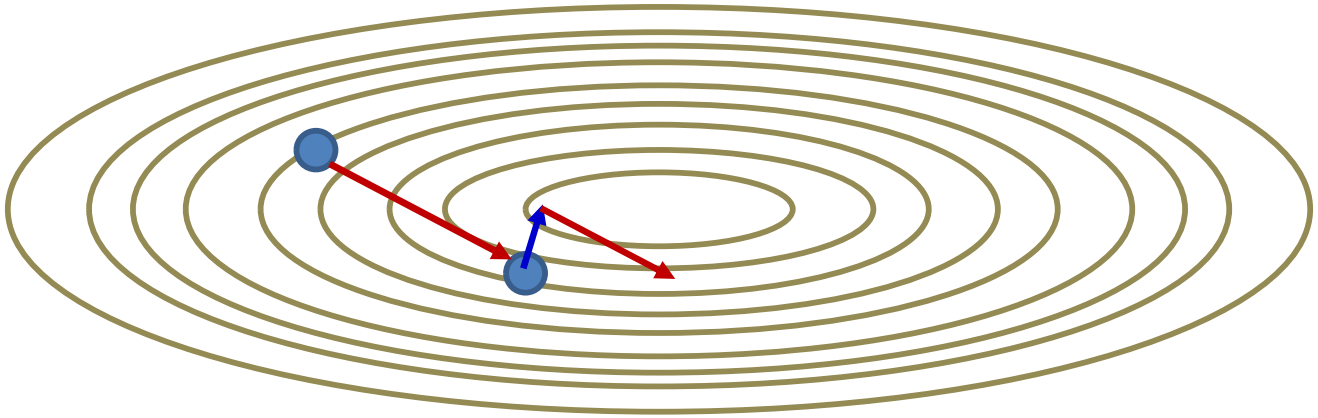


- The momentum method

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

- At any iteration, to compute the current step:
  - First compute the gradient step at the current location

Recall: Momentum Update

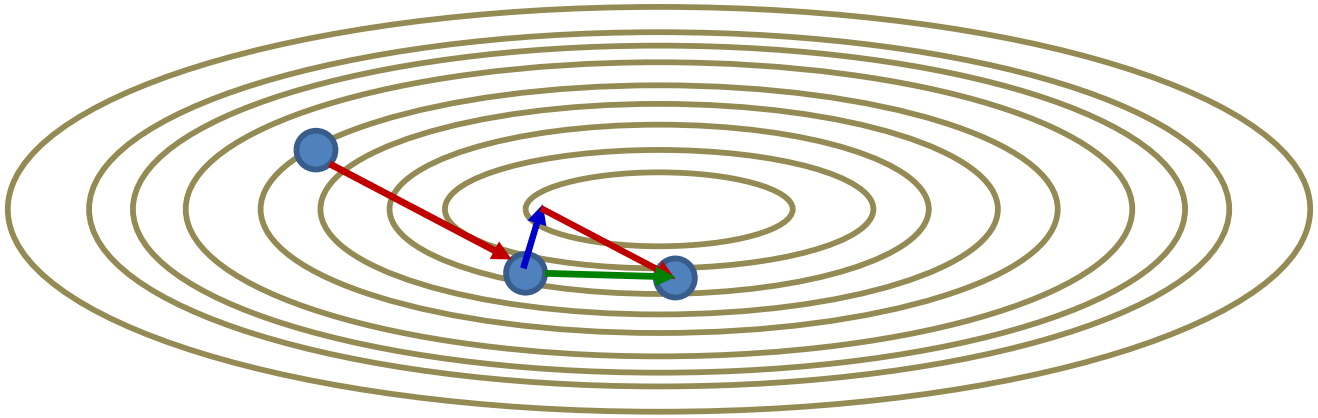


- The momentum method

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

- 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

Recall: Momentum Update

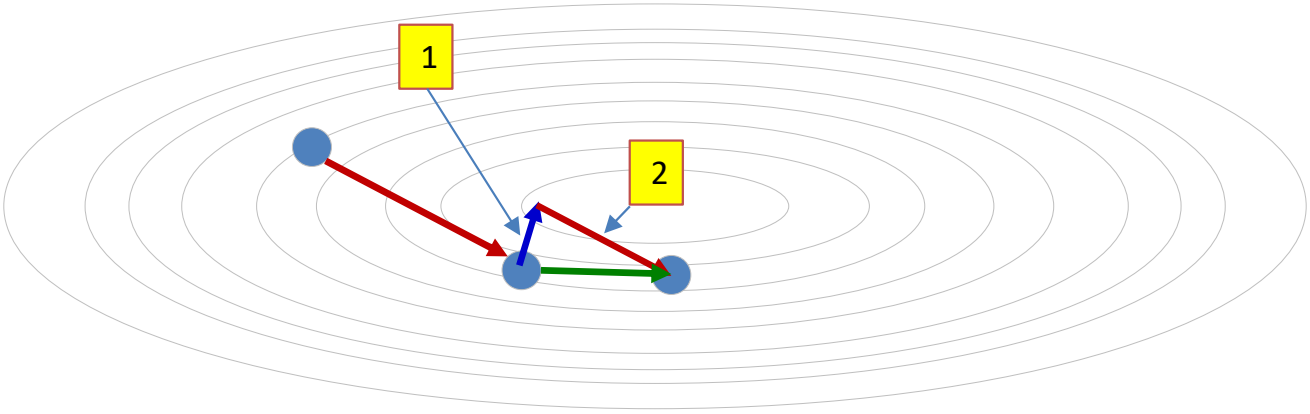


- The momentum method

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

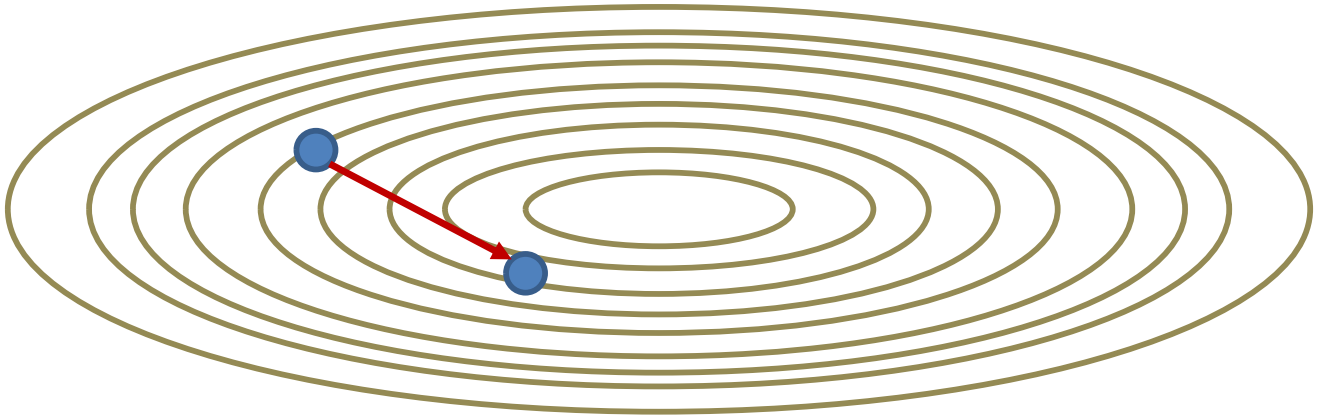
- At any iteration, to compute the current step:
  - First compute the gradient step at the current location
  - Then add in the scaled *previous* step
    - Which is actually a running average
  - To get the final step

Momentum update



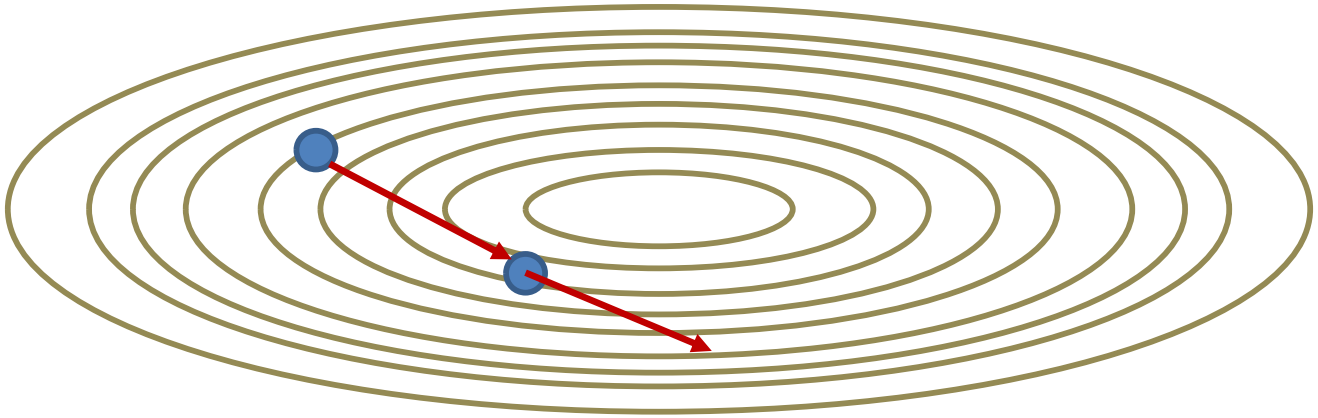
- Momentum update steps are actually computed in two stages
  - First: We take a step against the gradient at the current location
  - Second: Then we add a scaled version of the previous step
- The procedure can be made more optimal by reversing the order of operations..

Nestorov's Accelerated Gradient



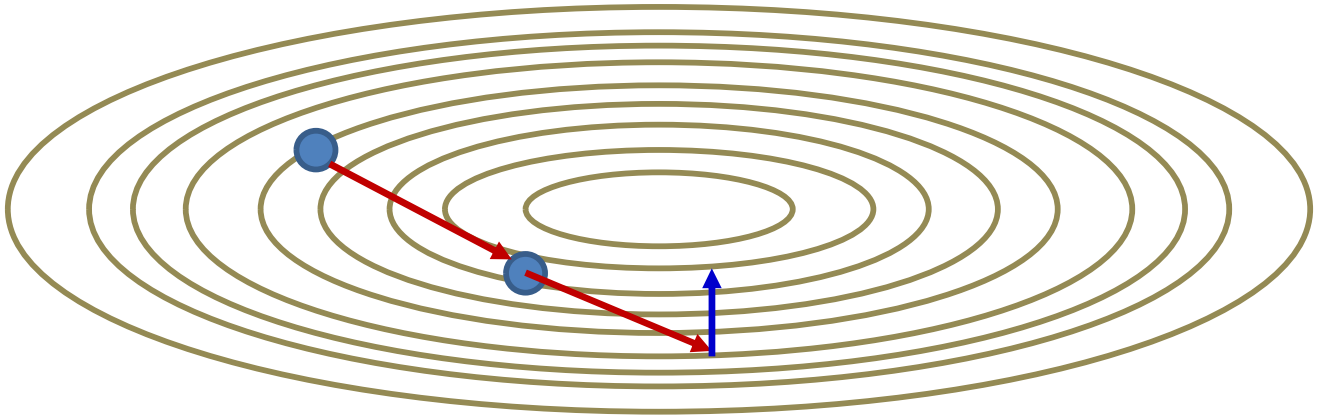
- Change the order of operations
- At any iteration, to compute the current step:

Nestorov's Accelerated Gradient



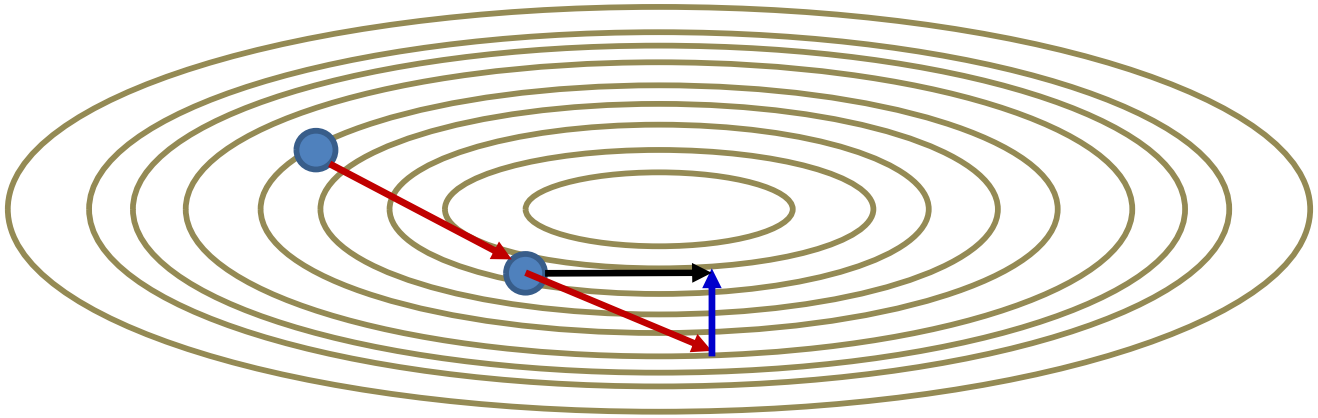
- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step

Nestorov's Accelerated Gradient



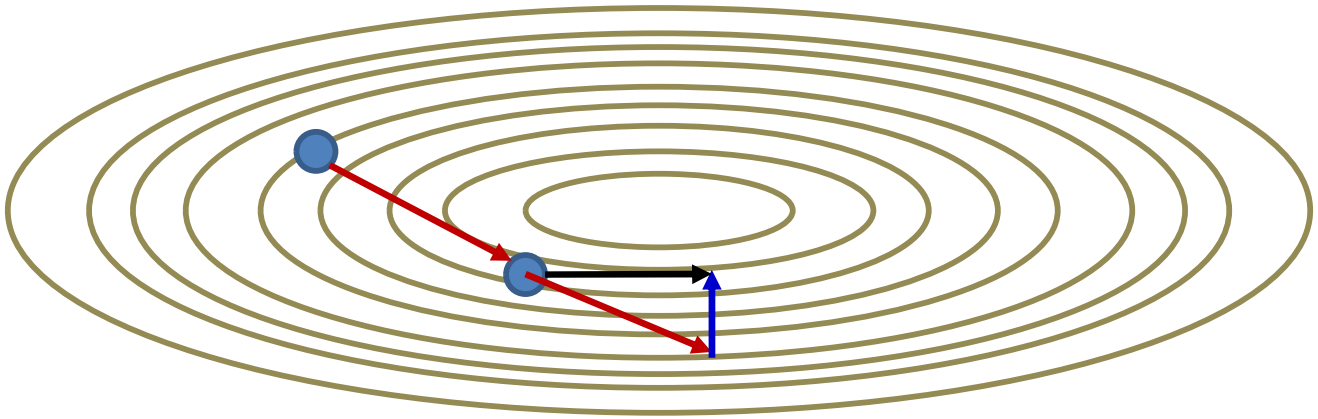
- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step
  - Then compute the gradient step at the resultant position

Nestorov's Accelerated Gradient



- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step
  - Then compute the gradient step at the resultant position
  - Add the two to obtain the final step

Nestorov's Accelerated Gradient

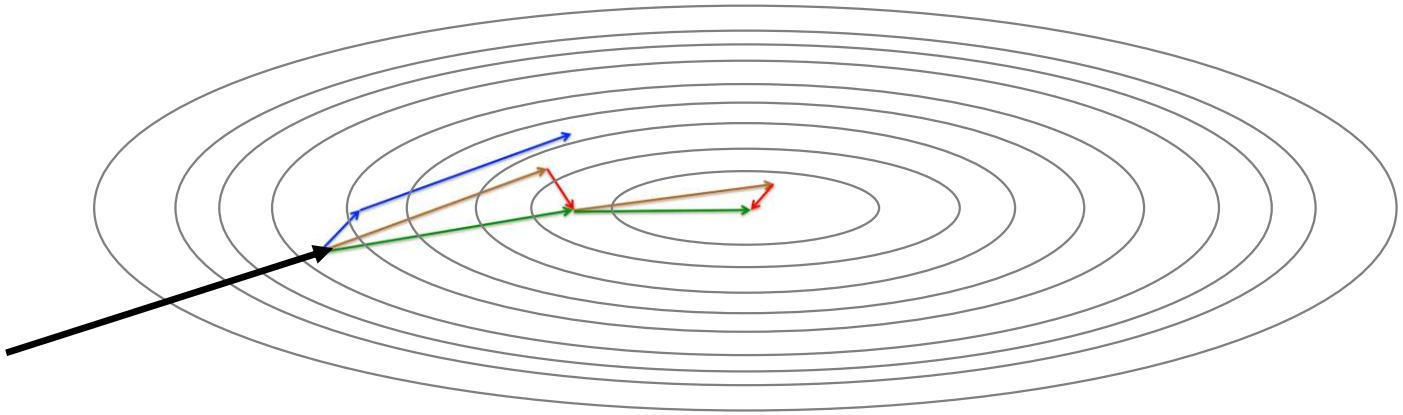


- Nestorov's method

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

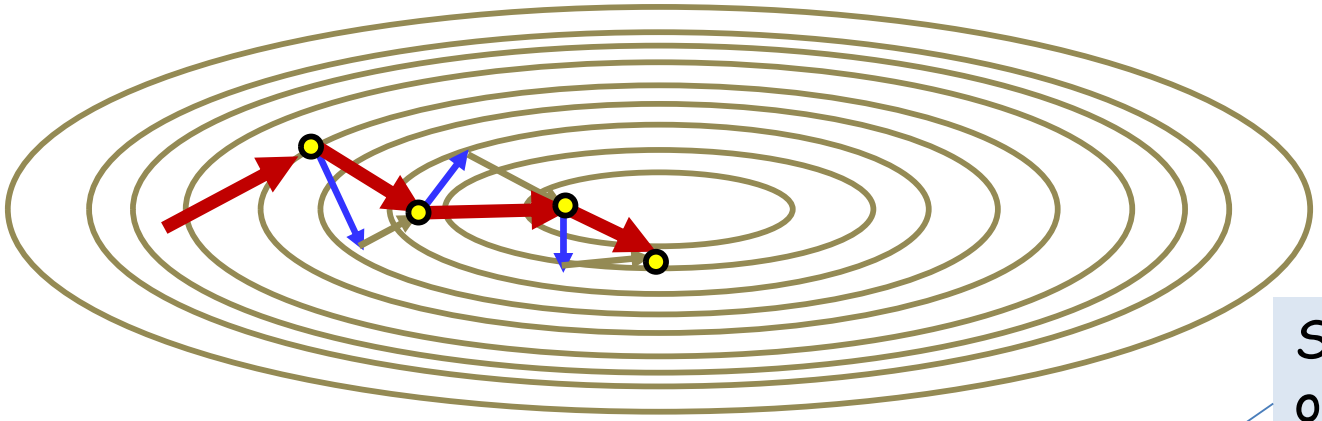
$$W^{(k)} = W^{(k-1)} + \Delta W^{(k)}$$

Nestorov's Accelerated Gradient



- Comparison with momentum (example from Hinton)
- Converges much faster

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

Momentum: Mini-batch update

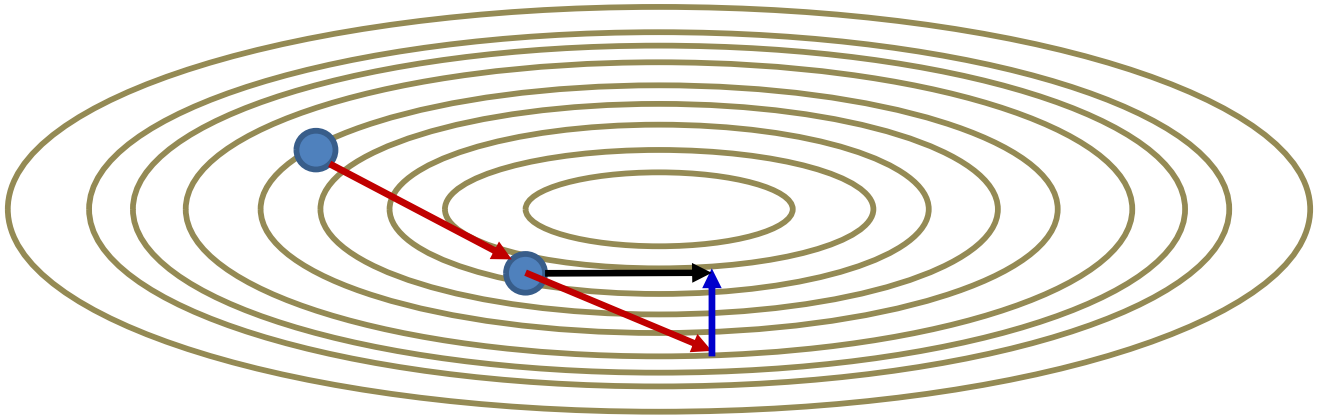
- Given (X_1, d_1), (X_2, d_2),…, (X_T, d_T)
- Initialize all weights $W_1, W_2, \ldots, W_K$; $j = 0$, $\Delta W_k = 0$
- Do:
  - Randomly permute (X_1, d_1), (X_2, d_2),…, (X_T, d_T)
  - For $t = 1 : b : T$
    - $j = j + 1$
    - For every layer k:
      - $\nabla_{W_k} Loss = 0$
    - For t' = t : t+b-1
      - For every layer k:
        - Compute $\nabla_{W_k} Div(Y_t, d_t)$
        - $\nabla_{W_k} Loss \mathrel{+}= \frac{1}{b} \nabla_{W_k} \boldsymbol{Div}(Y_t, d_t)$
    - Update
      - For every layer k:

$$\Delta W_k = \beta \Delta W_k - \eta_j (\nabla_{W_k} Loss)^T$$

$$W_k = W_k + \Delta W_k$$

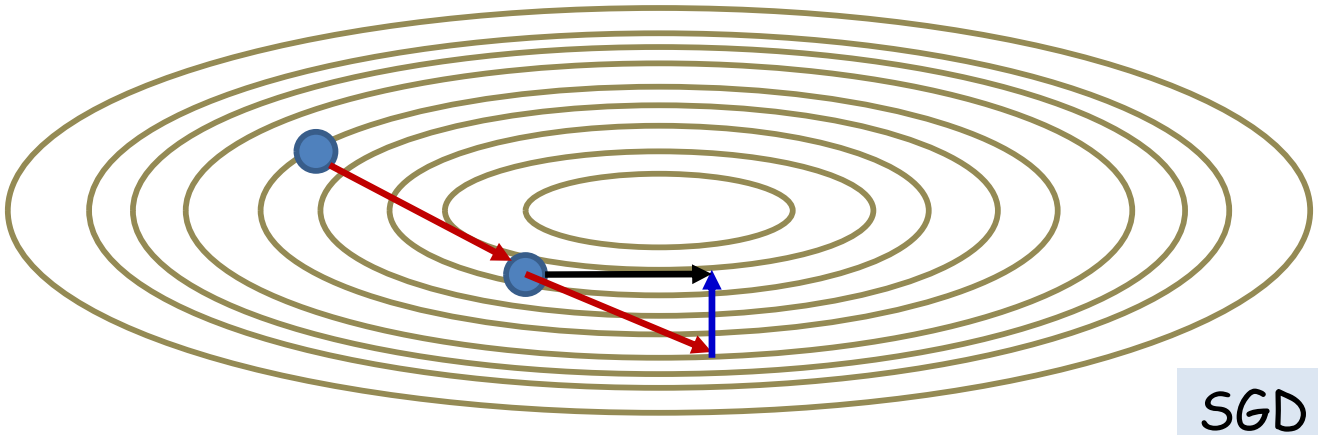
- Until $Loss$ has converged

Nestorov'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



SGD instance or minibatch loss

- Nestorov's method

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

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

Nestorov: Mini-batch update

- Given (X_1, d_1), (X_2, d_2),…, (X_T, d_T)
- Initialize all weights $W_1, W_2, \ldots, W_K$; $j = 0$, $\Delta W_k = 0$
- 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 $\nabla_{W_k} Div(Y_t, d_t)$
        - $\nabla_{W_k} Loss \mathrel{+}= \frac{1}{b} \nabla_{W_k} \boldsymbol{Div}(Y_t, d_t)$
    - 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

The other term in the update

- 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

The other term in the update

- 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
- Momentum methods fix this term to reduce unstable oscillation

The other term in the update

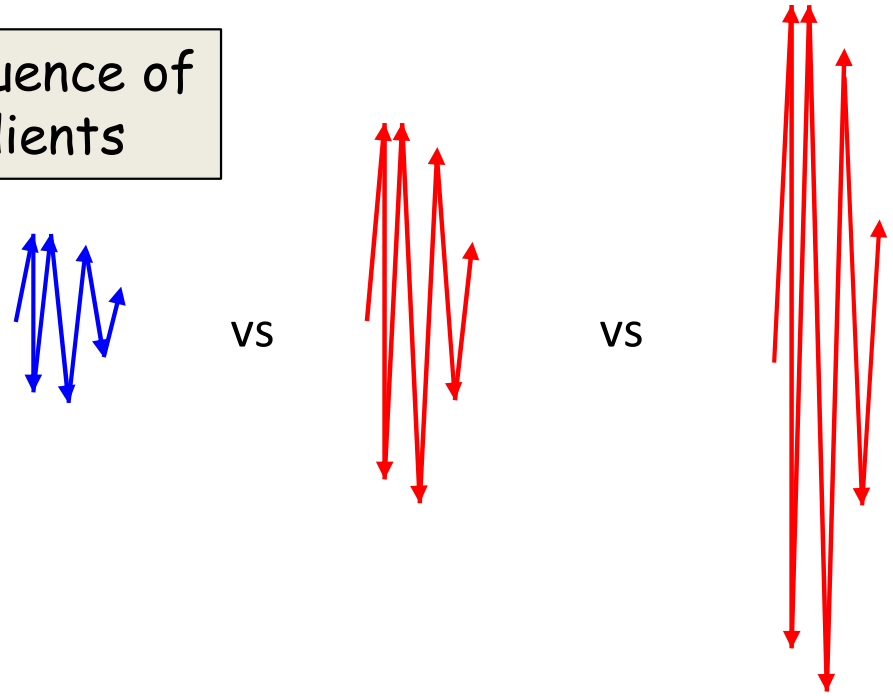
- Standard gradient descent rule

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

- Gradient descent invokes two terms for updates
  - The derivative
  - and the learning rate
- Momentum methods fix this term to reduce unstable oscillation
- What about this term?

Adjusting the learning rate

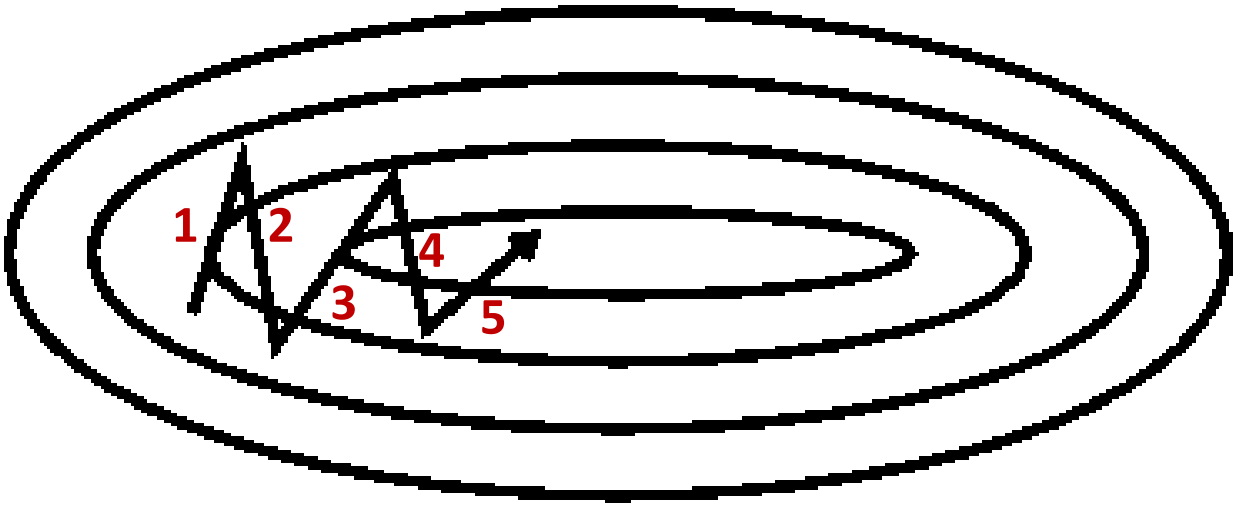
Sequence of gradients



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 RMS value

RMS Prop

- Notation:
  - Formulae 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 $E\left[\partial_w^2 D\right]$
- 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

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

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

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 learning rate of the parameter by the *inverse* of the *root mean squared* derivative

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

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

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, $E\left[\partial_w^2 D\right]_k = 0$
  - 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 $\boldsymbol{Y(X_{t+b})}$
        - Compute gradient $\frac{\boldsymbol{dDiv(Y(X_{t+b}),d_{t+b})}}{\boldsymbol{dw}}$
        - Compute $(\partial_w D)_k \mathrel{+}= \frac{1}{B}\frac{\boldsymbol{dDiv(Y(X_{t+b}),d_{t+b})}}{\boldsymbol{dw}}$
    - update: for all $w \in \left\{w_{\{ij\}}^k \forall i,j,k\right\}$

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

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

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

Typical values:
$\gamma = 0.9$
$\eta = 0.001$

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

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

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

- How about combining both?

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 derivative for each parameter
  - Maintain a running estimate of the mean squared value of derivatives for each parameter
  - Learning rate is proportional to the *inverse* of the *root mean squared* derivative

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

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

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

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

ADAM: RMSprop with momentum

- RMS prop only adapts the learning rate
- Momentum only smooths the gradient
- ADAM combines the two

- **Procedure:**
  - Maintain a running estimate of the mean derivative for each parameter
  - Maintain a running estimate of the mean squared valu parameter
  - Learning rate is proportional to the *inverse* of the *r* derivative

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

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

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

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

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

ADAM: RMSprop with momentum

Typically μ_0 is 0 and δ is close to 1. So $(1-\delta) \approx 0$.
Without the denominator term μ_k will stay close to 0 for $k = 0,1,2, \ldots$ for a long time, resulting in minimal parameter updates

The denominator term ensures that $\mu_1 = (\partial_w D)_1$ and updates actually happen

For large k, the denominator just becomes 1

- Maintain a running estimate of the mean squared valu… parameter
- Learning rate is proportional to the *inverse* of the *r*… derivative

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

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

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

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

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

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

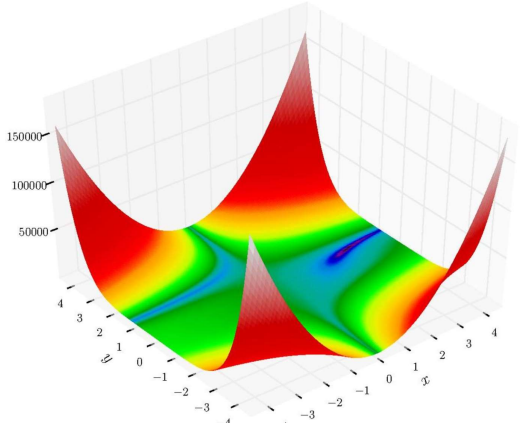
Poll 4

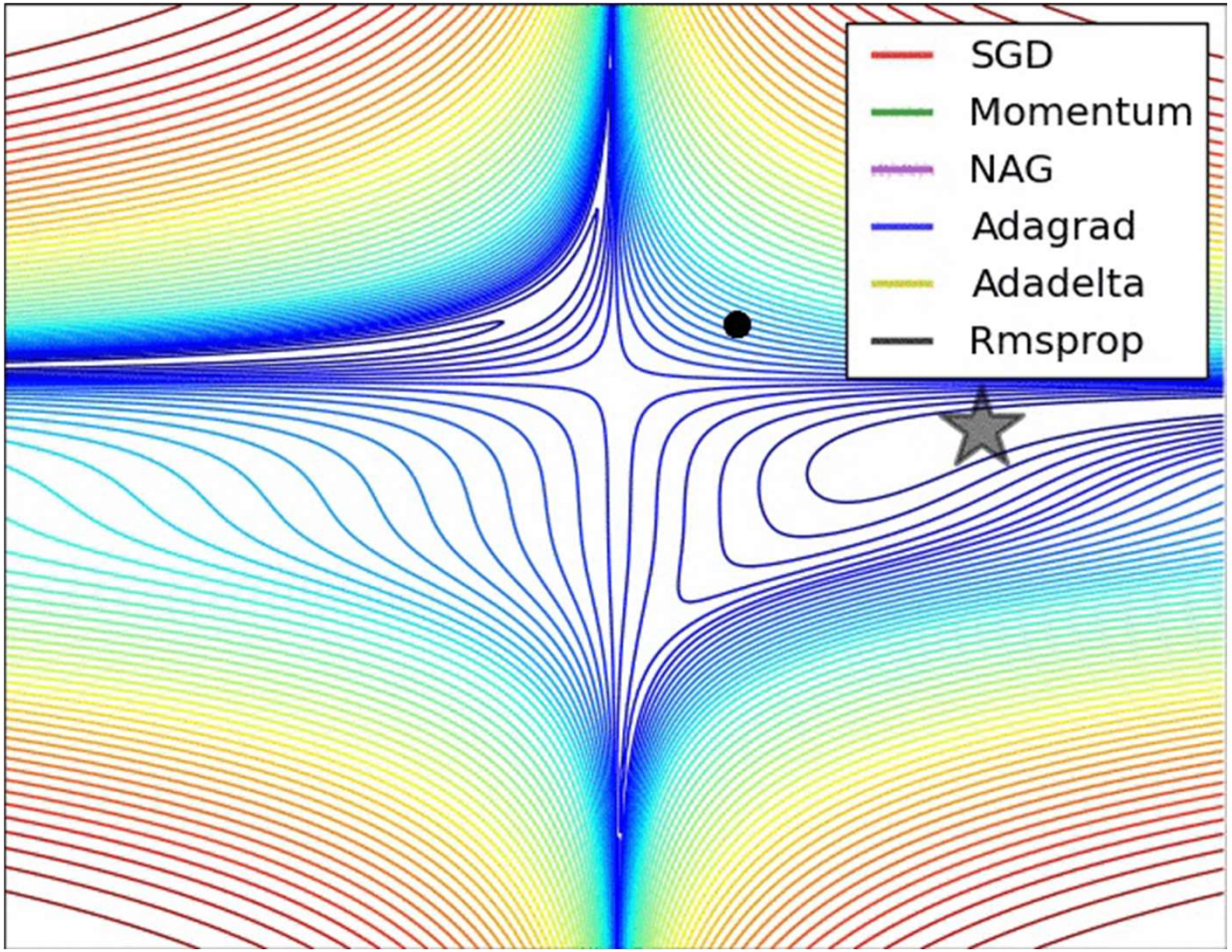
- Which of the following are true
  - Vanilla SGD considers the long-term trends of gradients in update steps
  - Momentum methods consider the long-term average of derivatives to make updates
  - RMSprop only considers the second-order moment of derivatives, but not their average trend, to make updates
  - ADAM considers both, the average trend and the second moment of derivatives to make updates
  - Trend-based optimizers like momentum, RMSprop and ADAM are important to smooth out the variance of SGD or mini-batch updates

Poll 4

- Which of the following are true
  - **Vanilla SGD considers the long-term trends of gradients in update steps**
  - **Momentum methods consider the long-term average of derivatives to make updates**
  - **RMSprop only considers the second-order moment of derivatives, but not their average trend, to make updates**
  - **ADAM considers both, the average trend and the second moment of derivatives to make updates**
  - **Trend-based optimizers like momentum, RMSprop and ADAM are important to smooth out the variance of SGD or mini-batch updates**

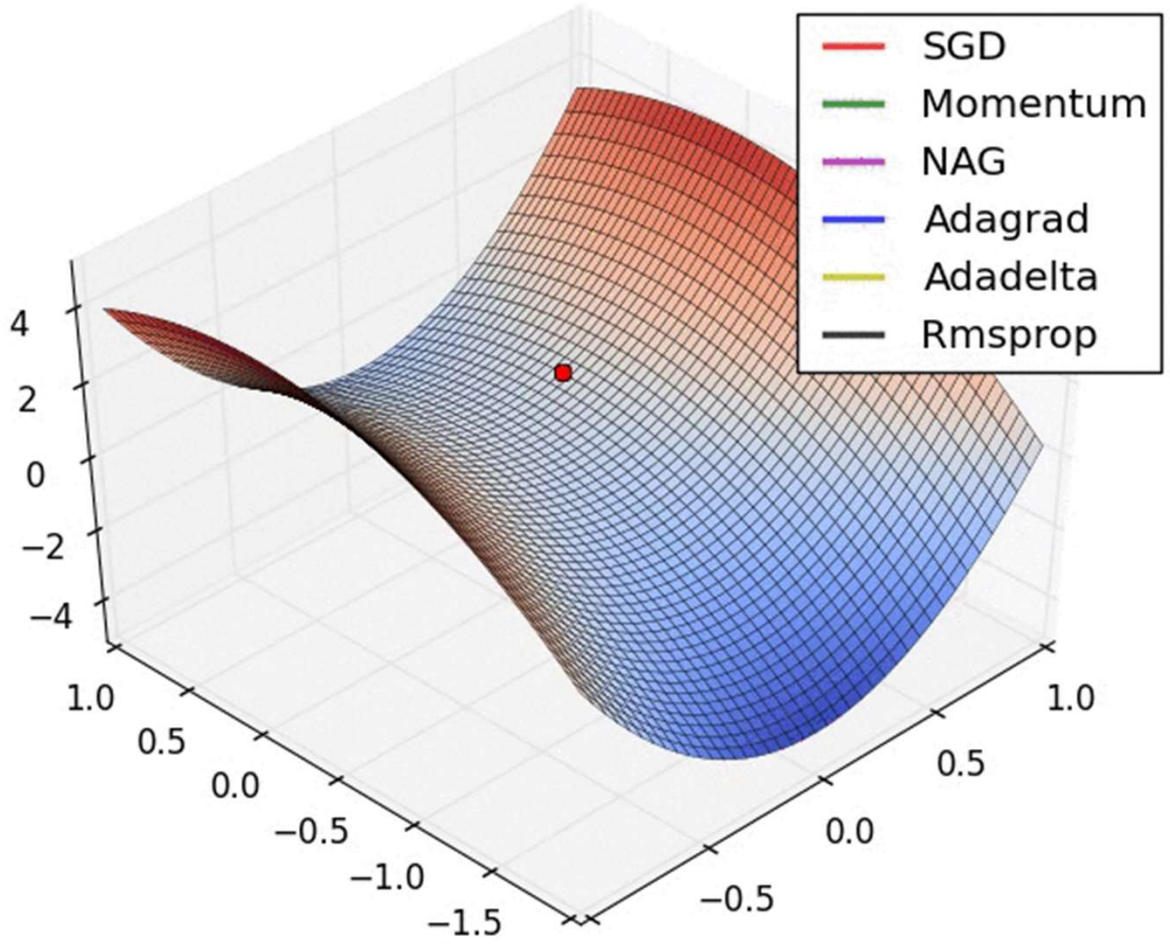
Visualizing the optimizers: Beale's Function





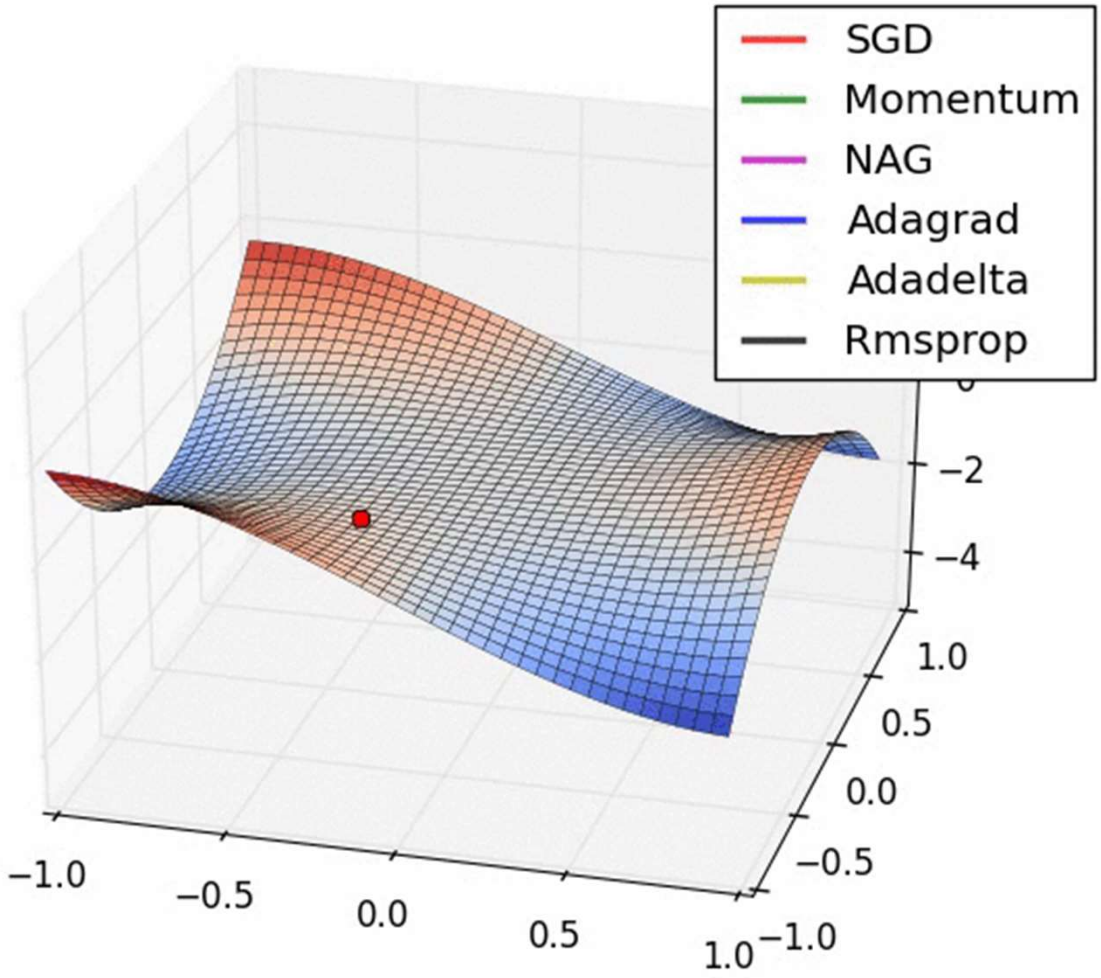
- http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

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