Training Neural Networks: Optimization

Intro to Deep Learning, Fall 2022

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

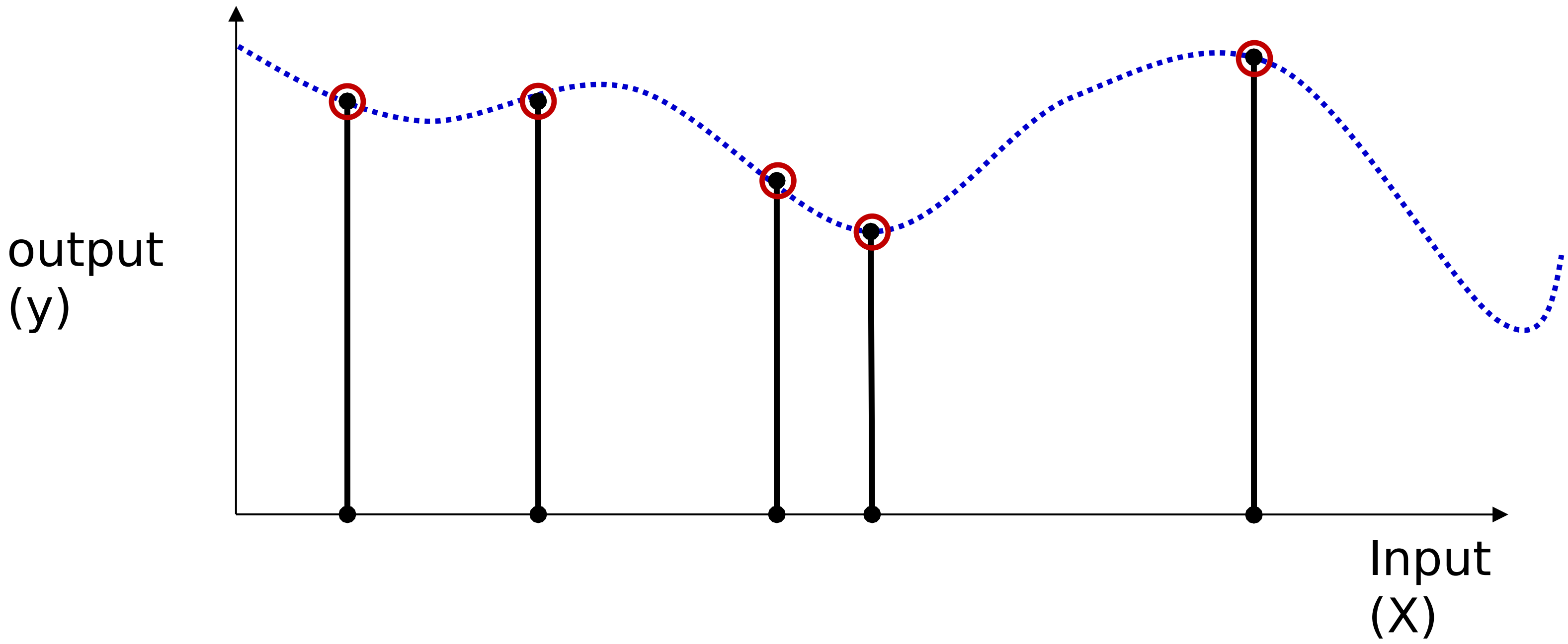
Moving on…

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

Moving on: Topics for the day

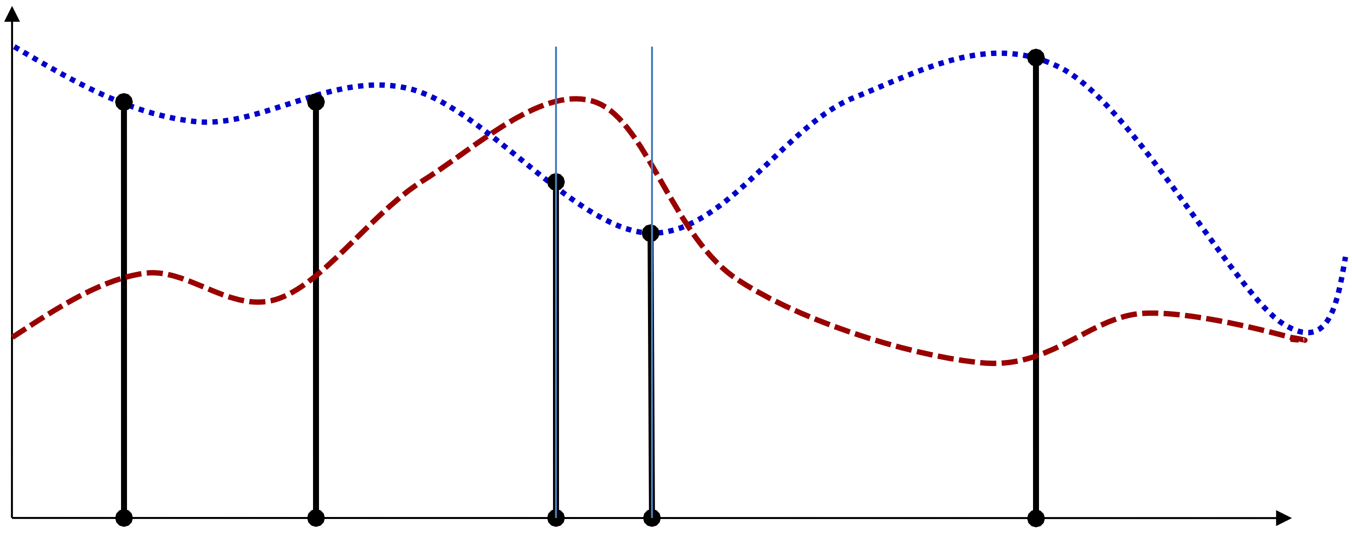
- Incremental updates
- Revisiting “trend” algorithms
- Generalization
- Tricks of the trade
  - Divergences..
  - Activations
  - Normalizations

The training formulation



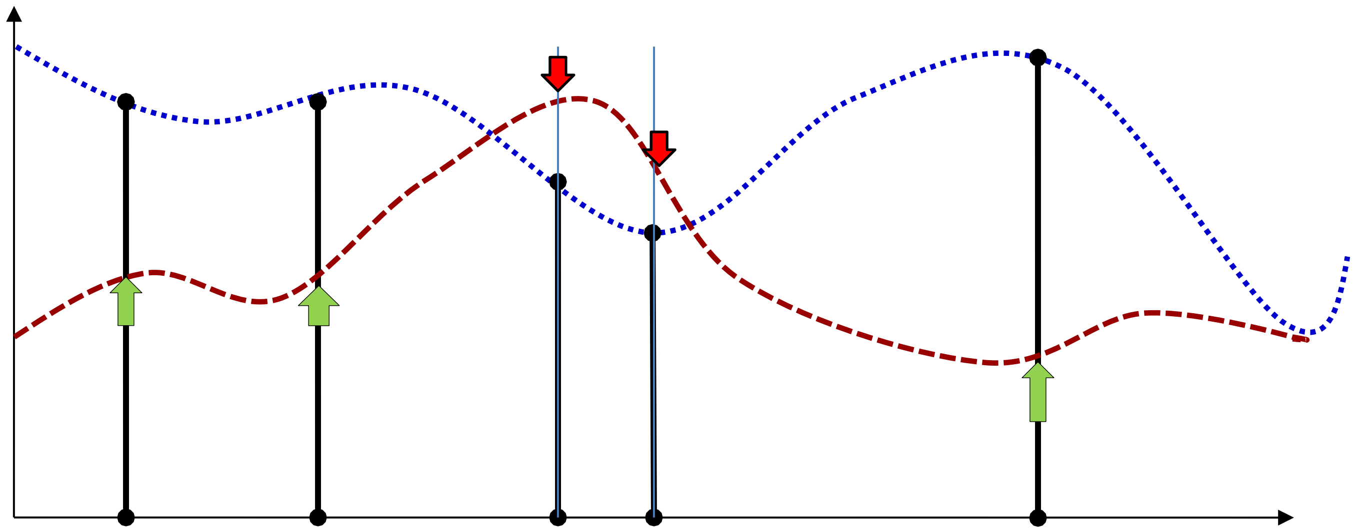
- Given input output pairs at a number of locations, estimate the entire function

Gradient descent



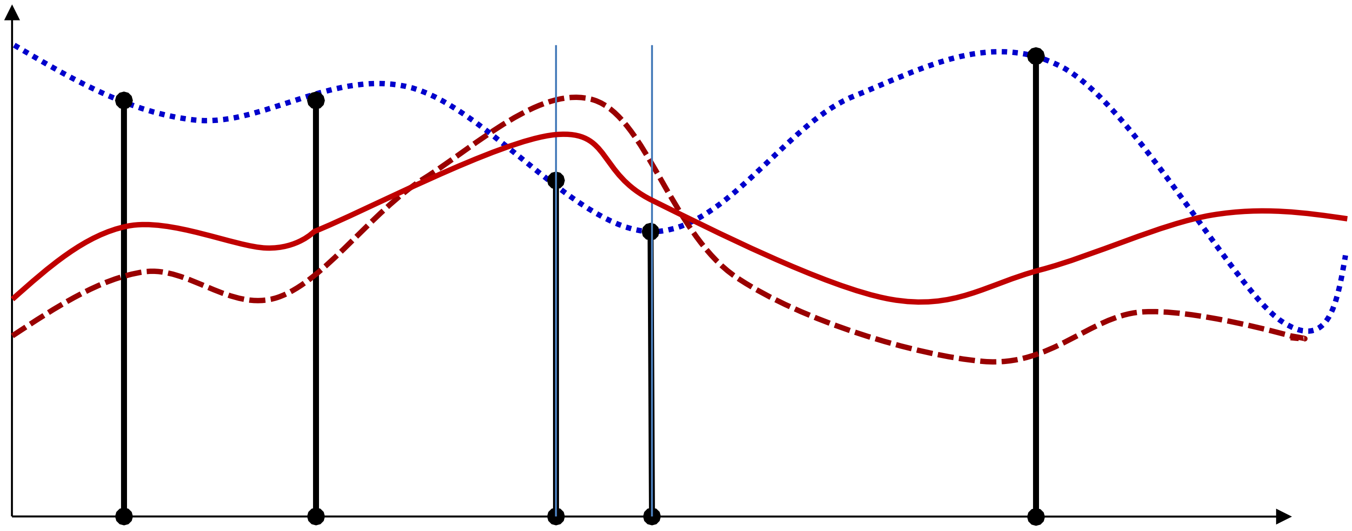
- Start with an initial function

Gradient descent



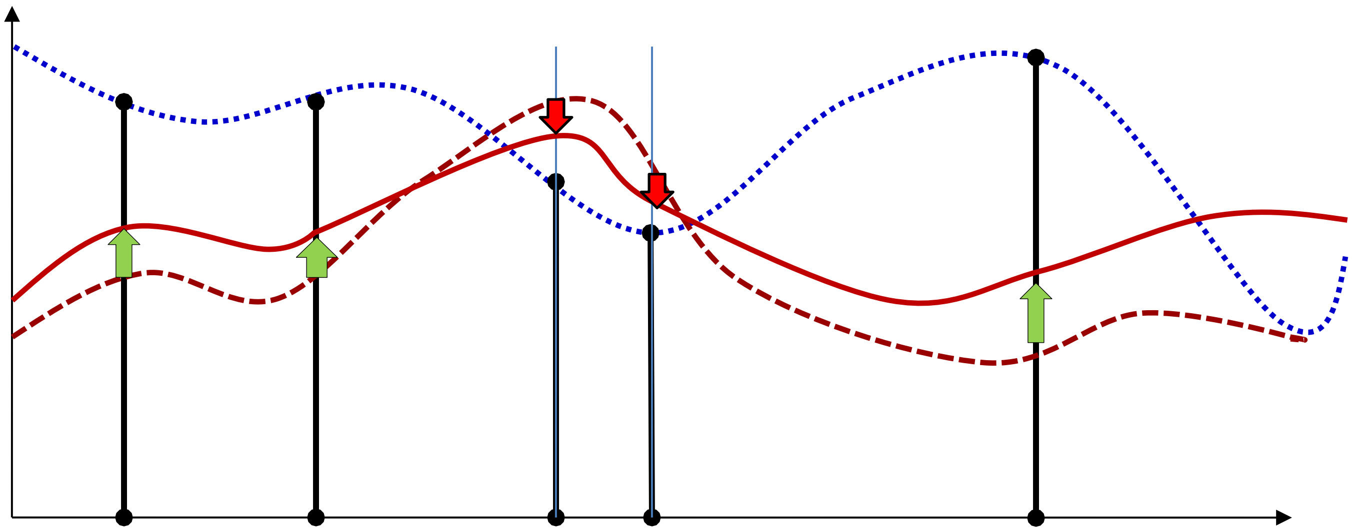
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



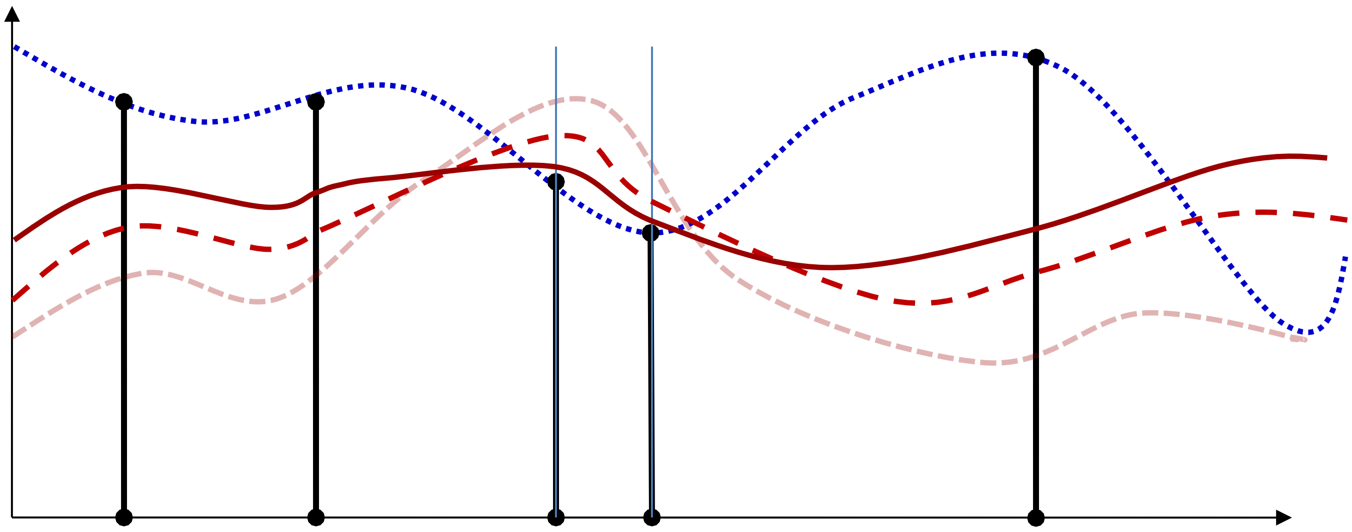
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



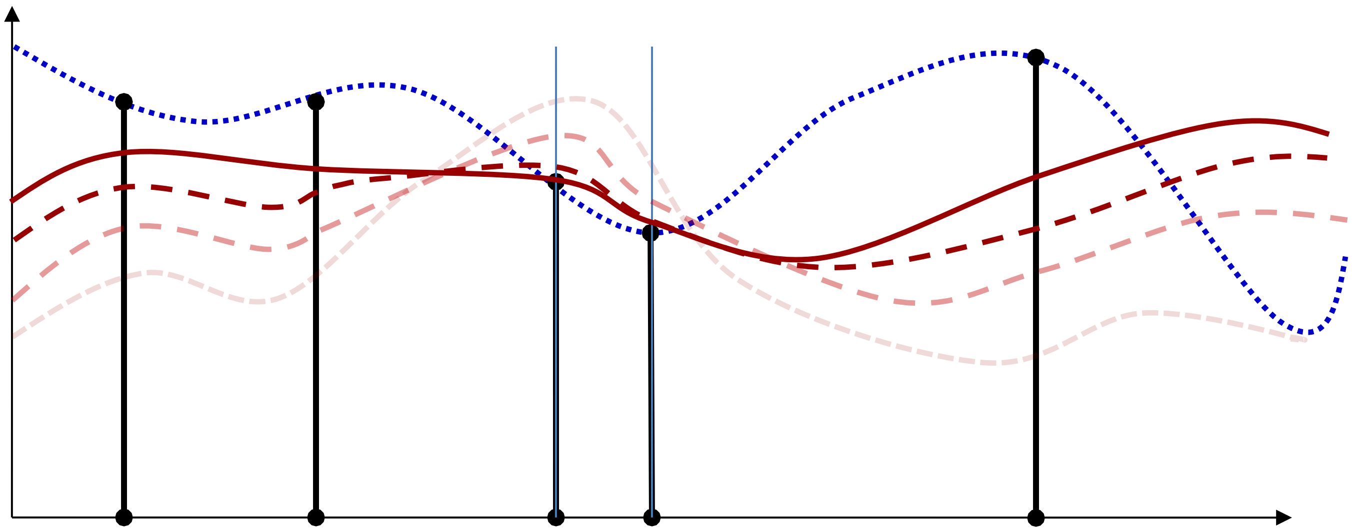
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



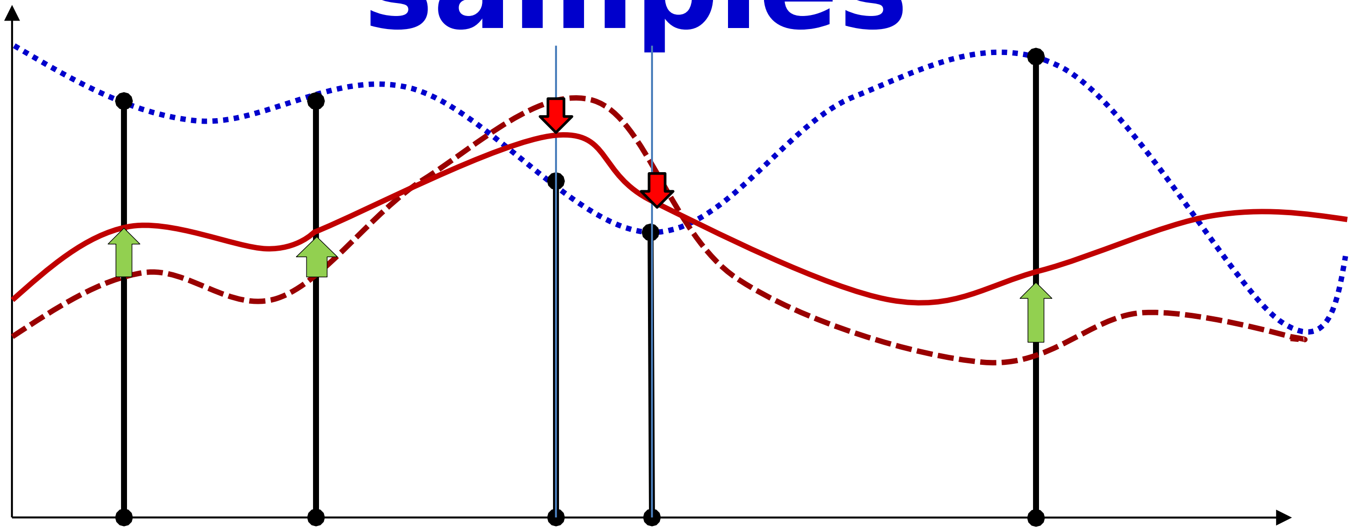
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

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

PIAZZA @575

Select all that are true

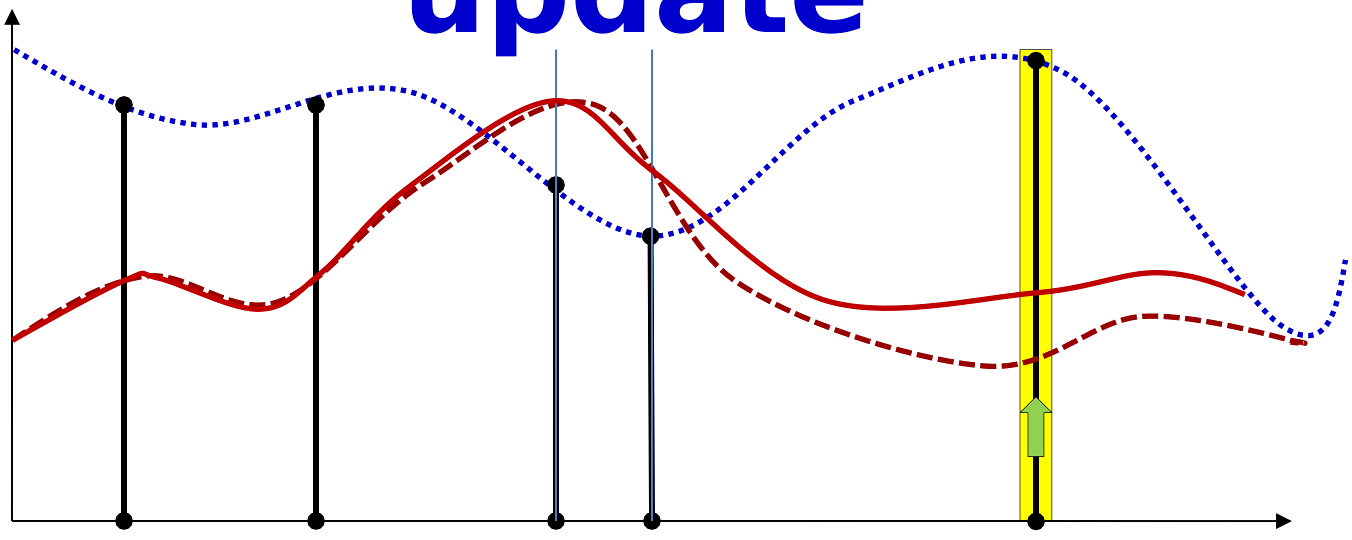
- The actual loss function we try to minimize requires batch updates
- Batch updates minimize the total loss over the entire training data
- Batch updates optimize the actual loss function
- Batch updates require processing the entire training data before we perform a single update

Poll 1

Select all that are true [**all correct**]

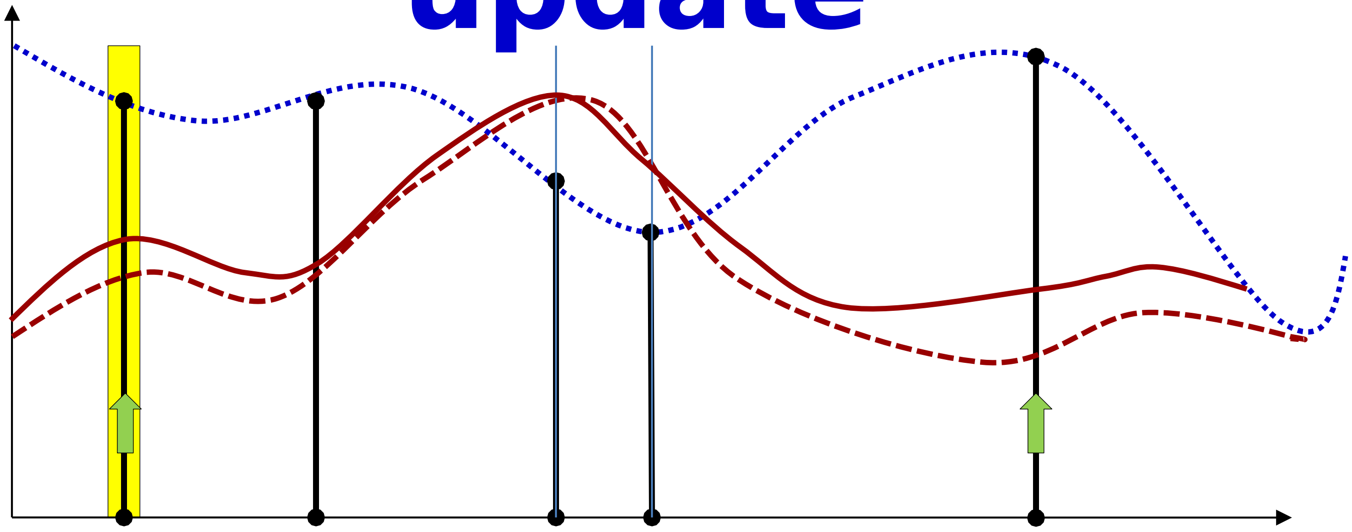
- The actual loss function we try to minimize requires batch updates
- Batch updates minimize the total loss over the entire training data
- Batch updates optimize the actual loss function
- Batch updates require processing the entire training data before we perform a single update

Alternative: Incremental update



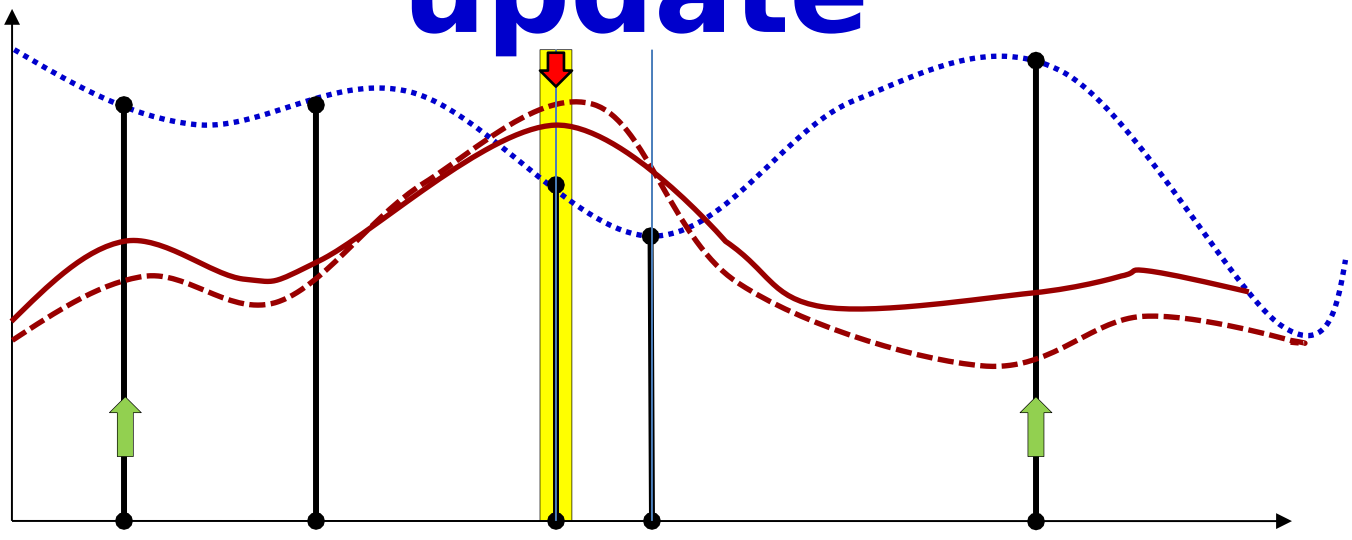
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

Alternative: Incremental update



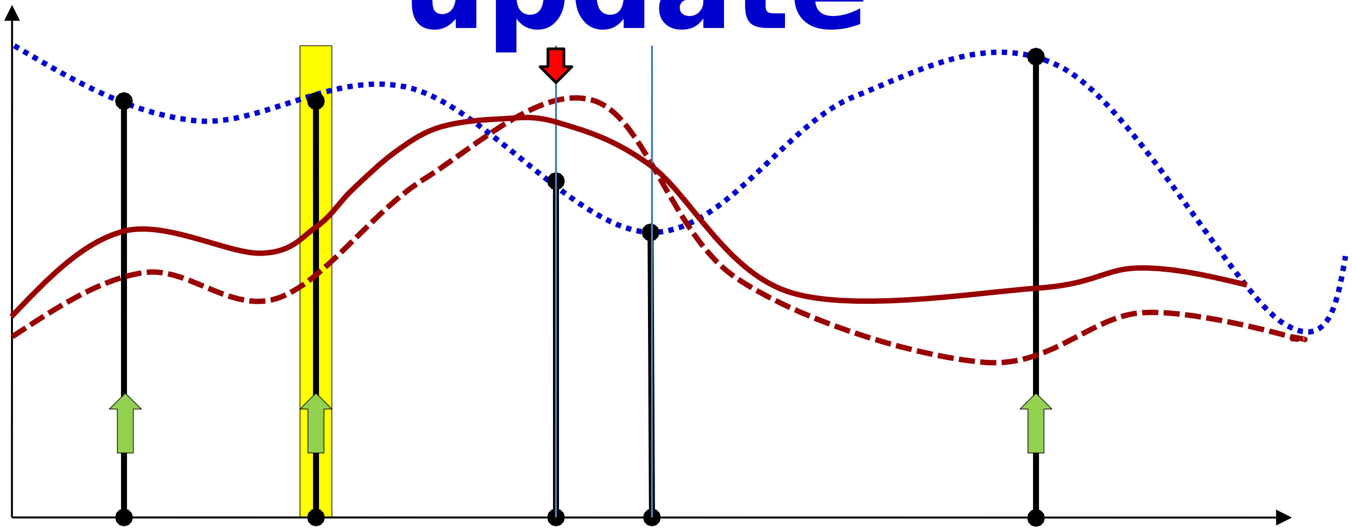
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

Alternative: Incremental update



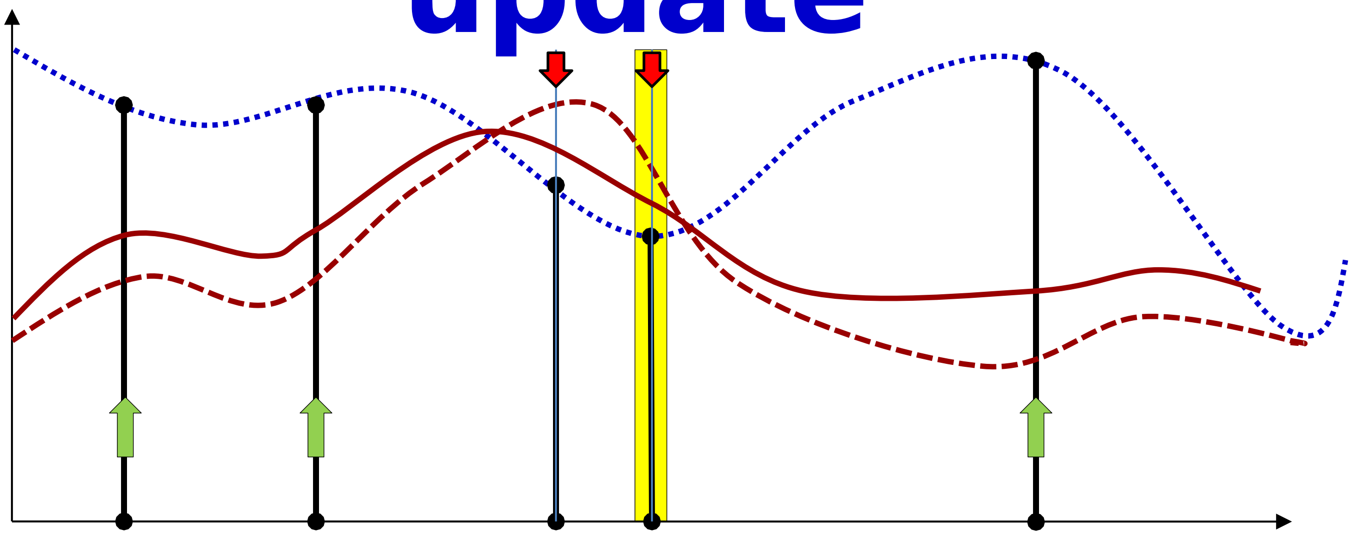
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

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 , ,...,
- Initialize all weights
- Do:
  - For all
    - For every layer :
      - Compute
      - Update
- Until 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 samples results in updates of parameters

Incremental Update

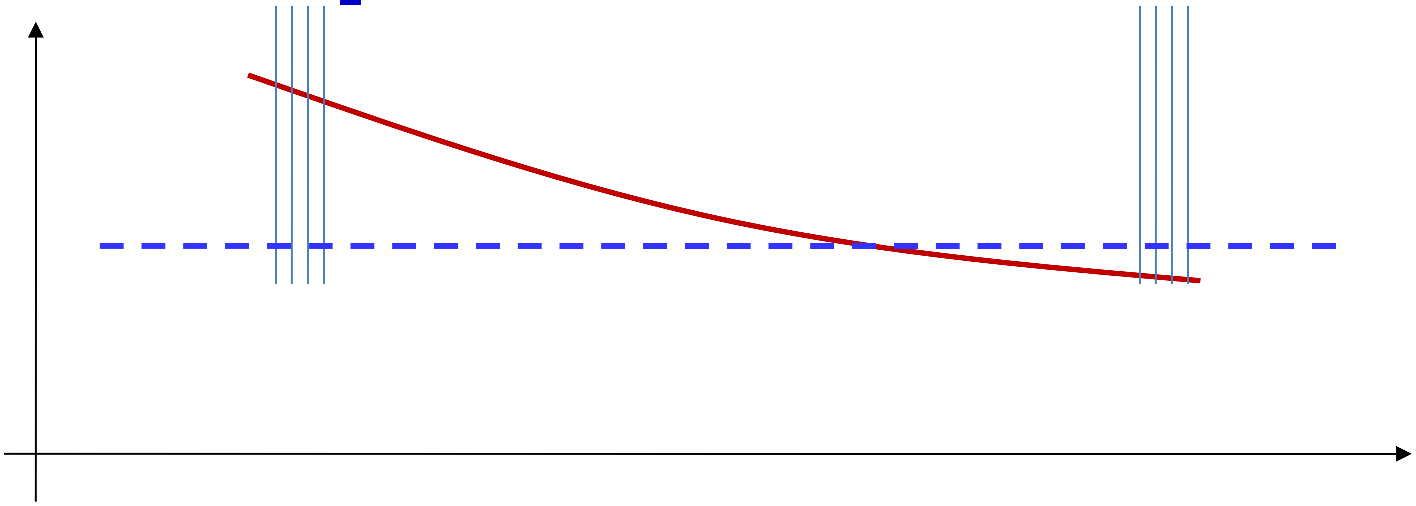
- Given , ,...,
- Initialize all weights
- Do:
  - For all
    - For every layer :
      - Compute
      - Update
- Until has converged

Over multiple epochs

One epoch

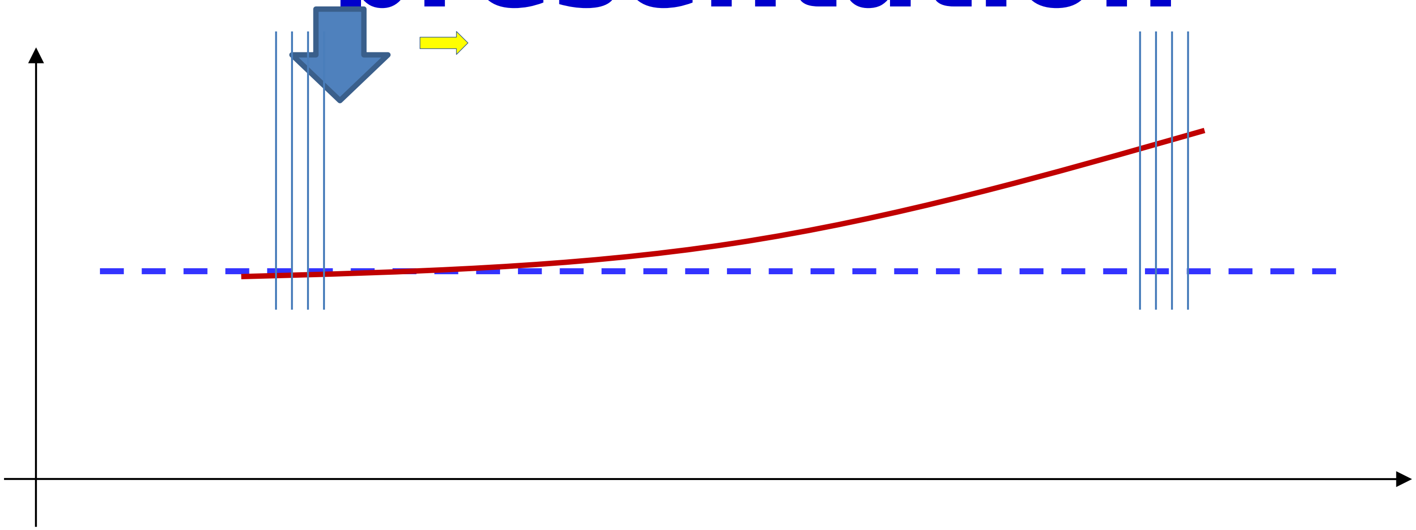
One update

Caveats: order of presentation



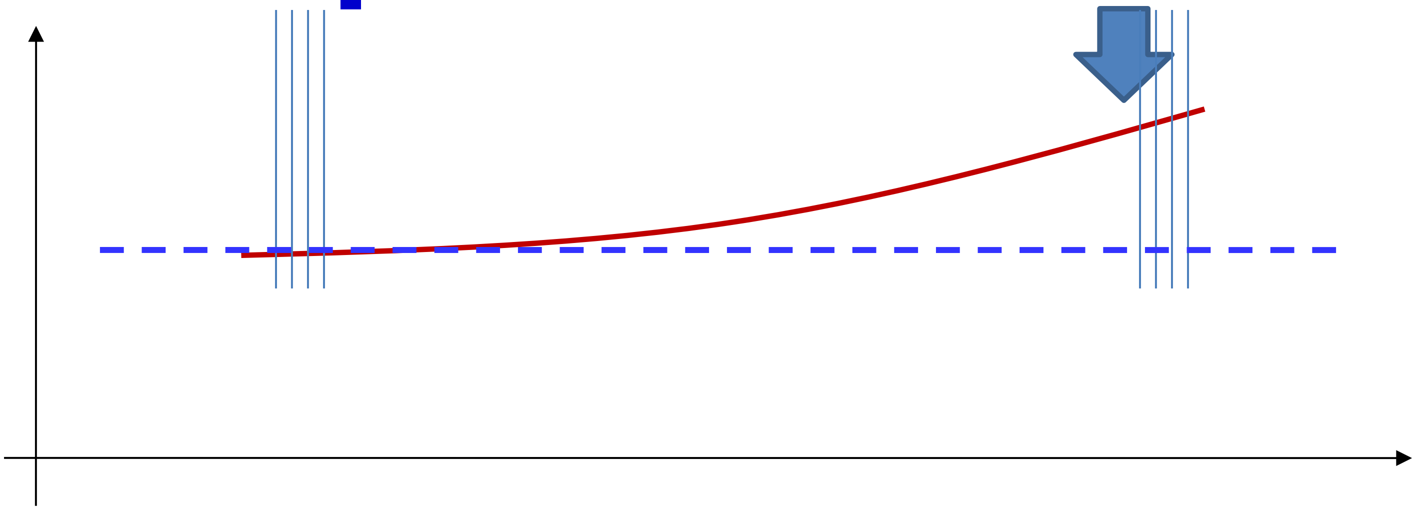
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



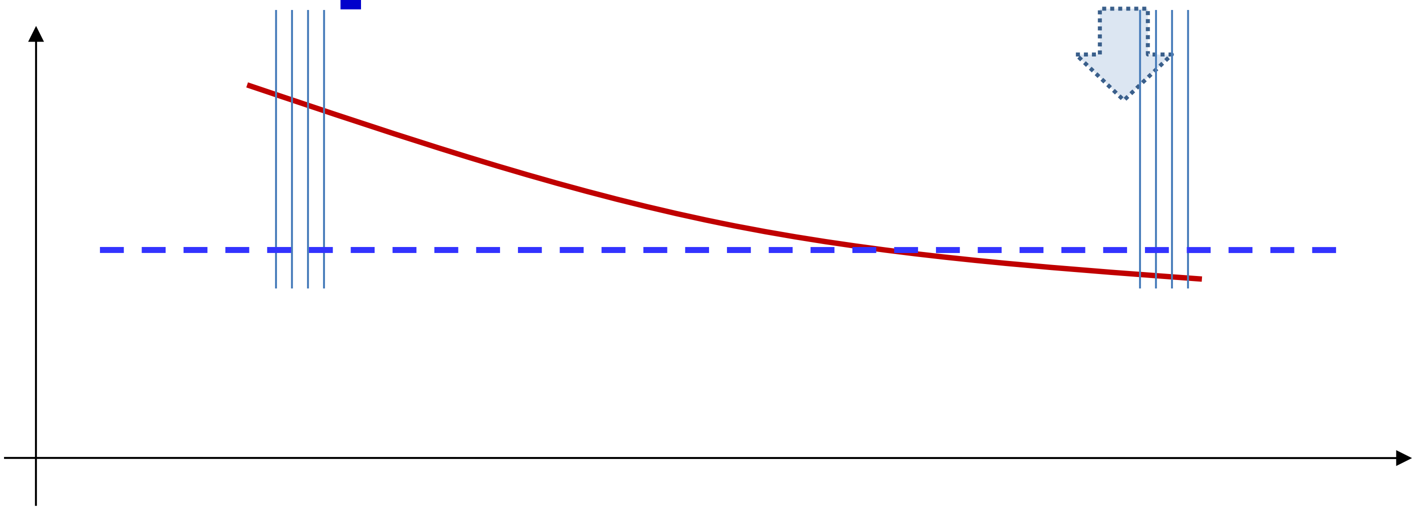
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



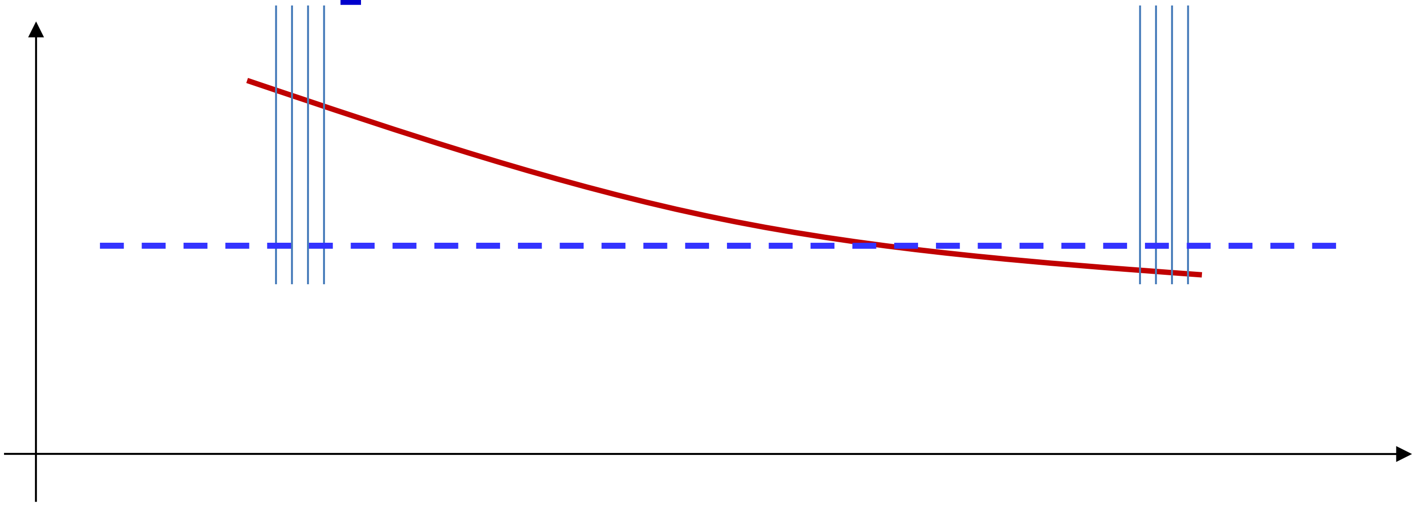
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



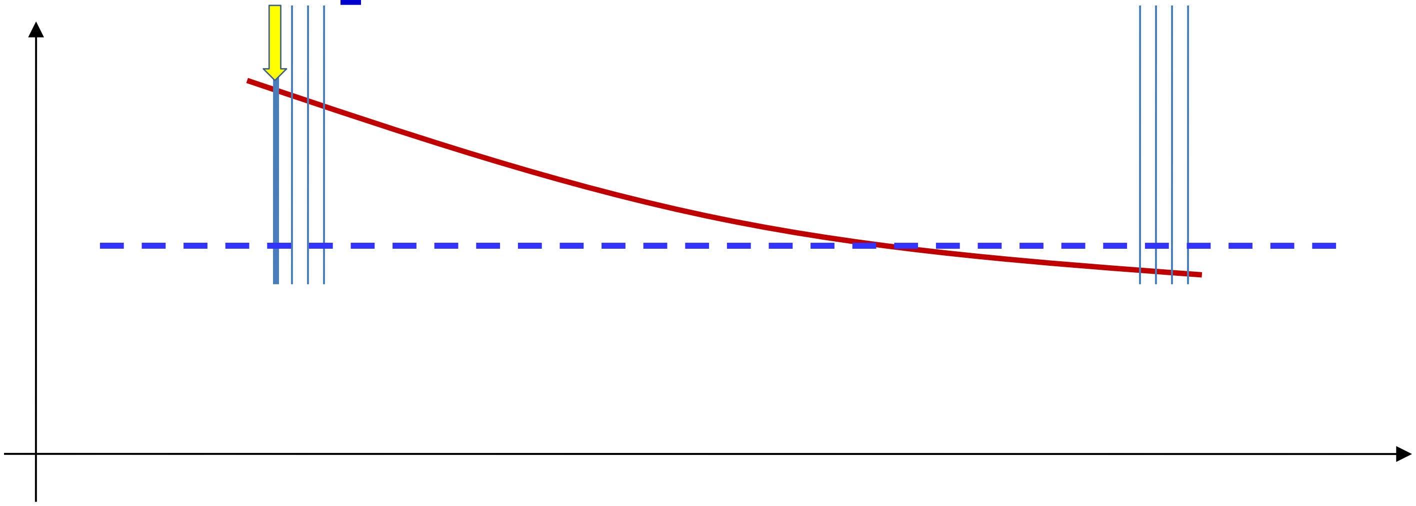
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



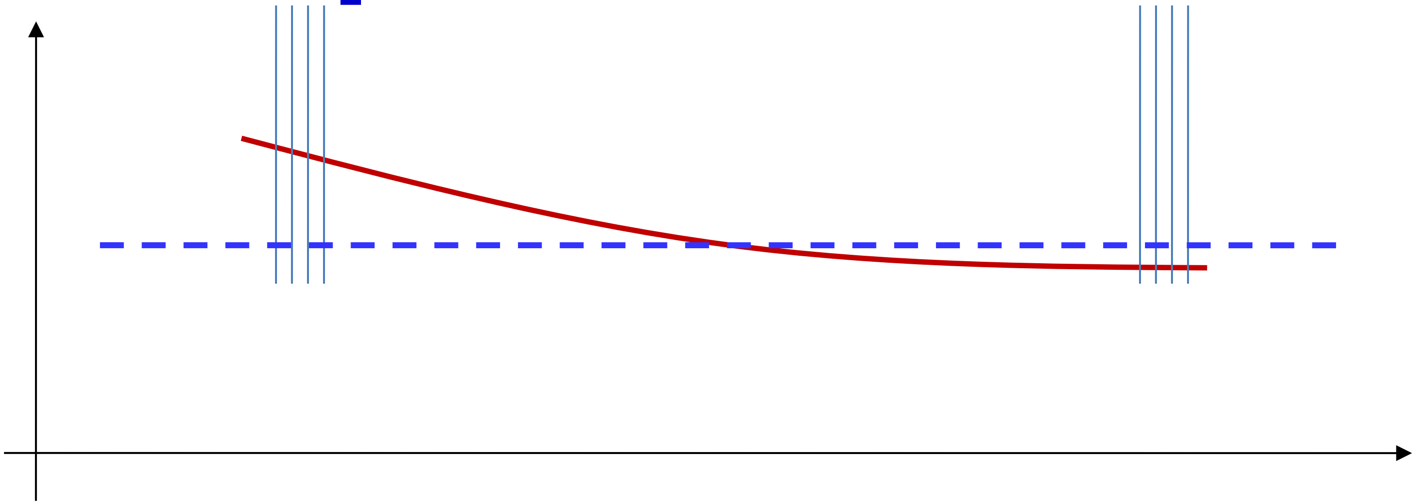
- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly* to get more convergent behavior

Caveats: order of presentation



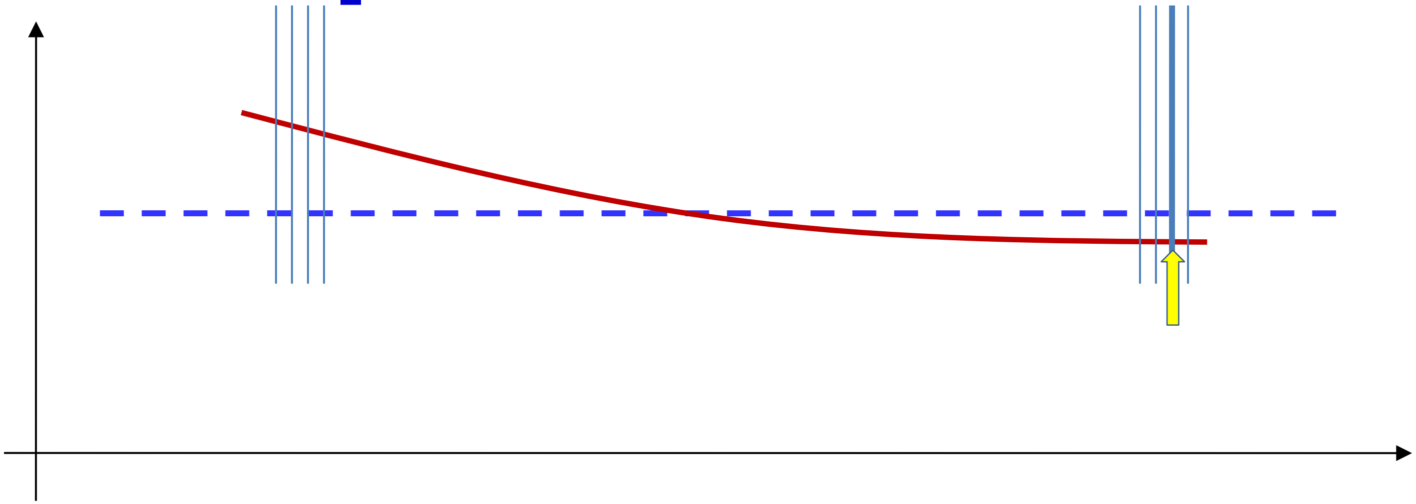
- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly* to get more convergent behavior

Caveats: order of presentation



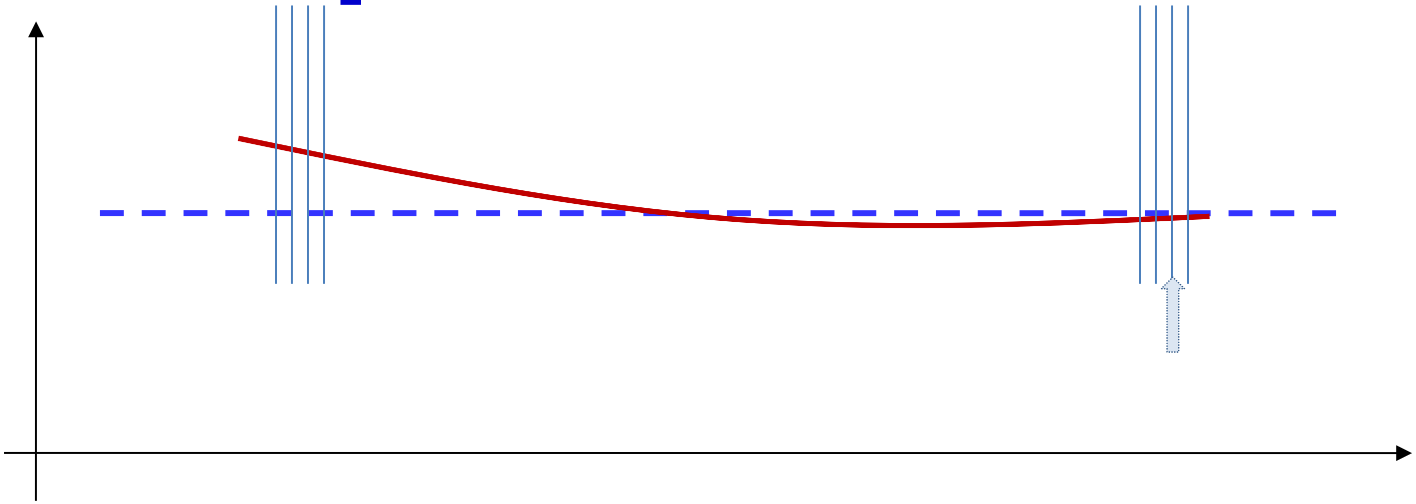
- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly* to get more convergent behavior

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 , ,…,
- Initialize all weights
- Do:
  - Randomly permute , ,…,
  - For all
    - For every layer :
      - Compute
      - Update
- Until has converged

Story so far

- In any gradient descent optimization problem, presenting training instances incrementally can be more effective than presenting them all at once
  - Provided training instances are provided in random order
  - "Stochastic Gradient Descent"

- This also holds for training neural networks

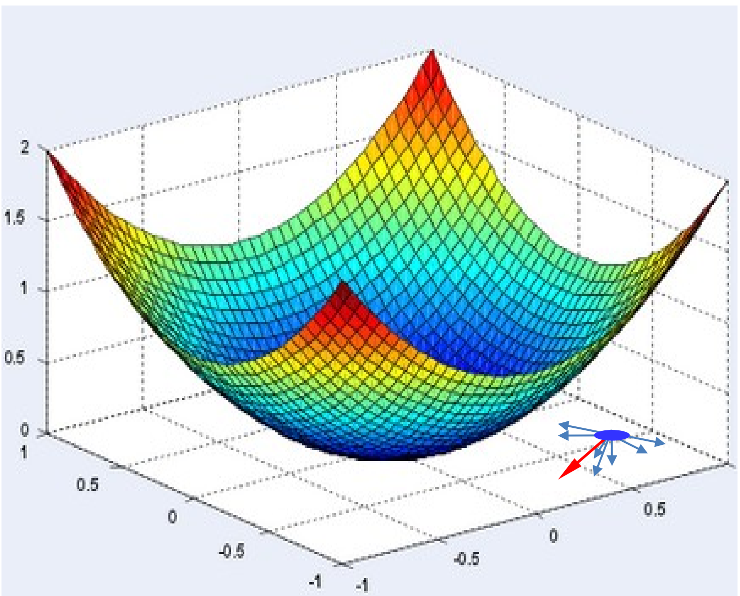
Explanations and restrictions

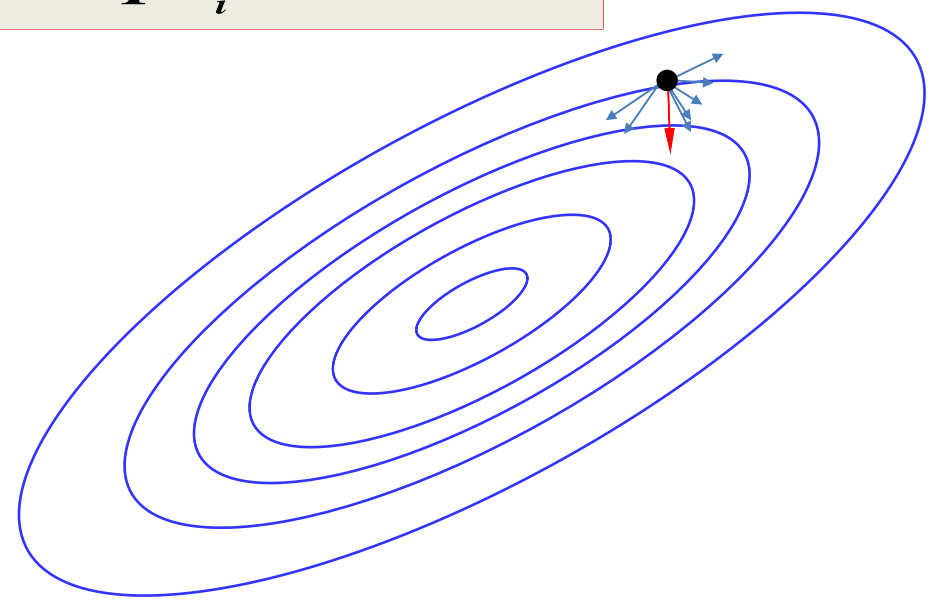
- So why does this process of incremental updates work?
- Under what conditions?

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

The expected behavior of the gradient

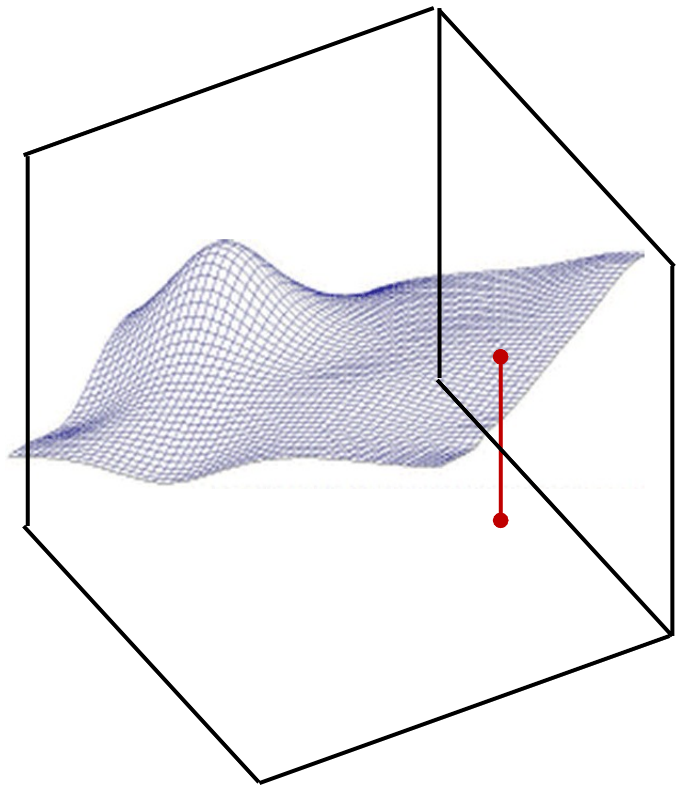
$$\frac{dE(W¿¿(1), W^{(2)}, \ldots, W^{(K)})}{dw_{i,j}^{(k)}} = \frac{1}{T}\sum_i dDiv ¿¿¿¿$$





- 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

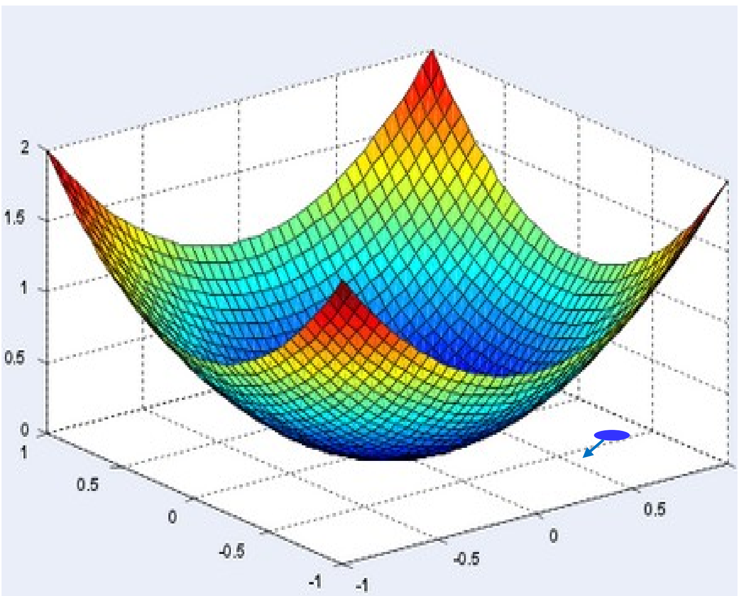


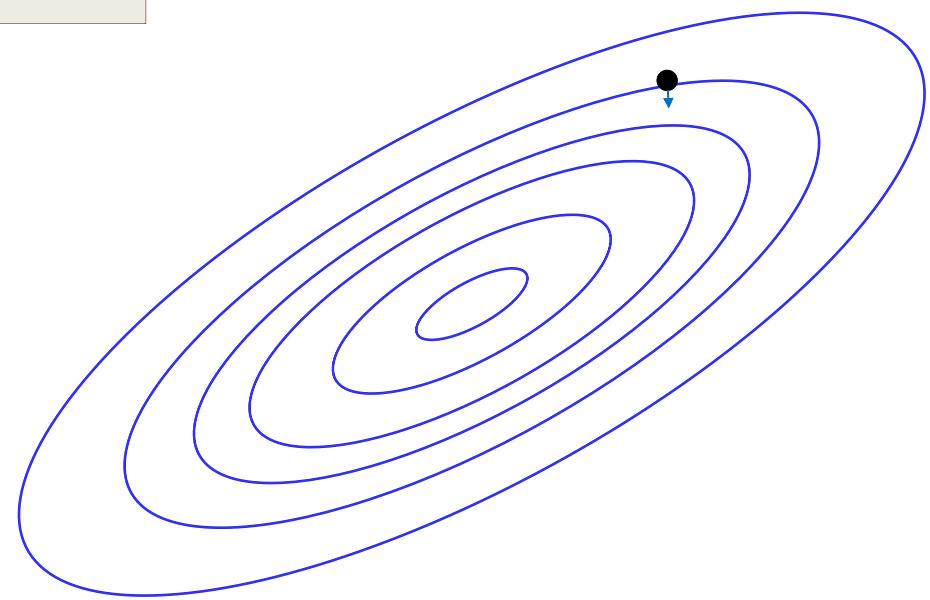
$$X_1 = X_2 = \ldots = X_T$$

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

The expected behavior of the gradient

$$\frac{dE}{dw_{i,j}^{(k)}} = \frac{1}{T}\sum_{i} dDiv \text{ ¿ ¿ ¿}$$

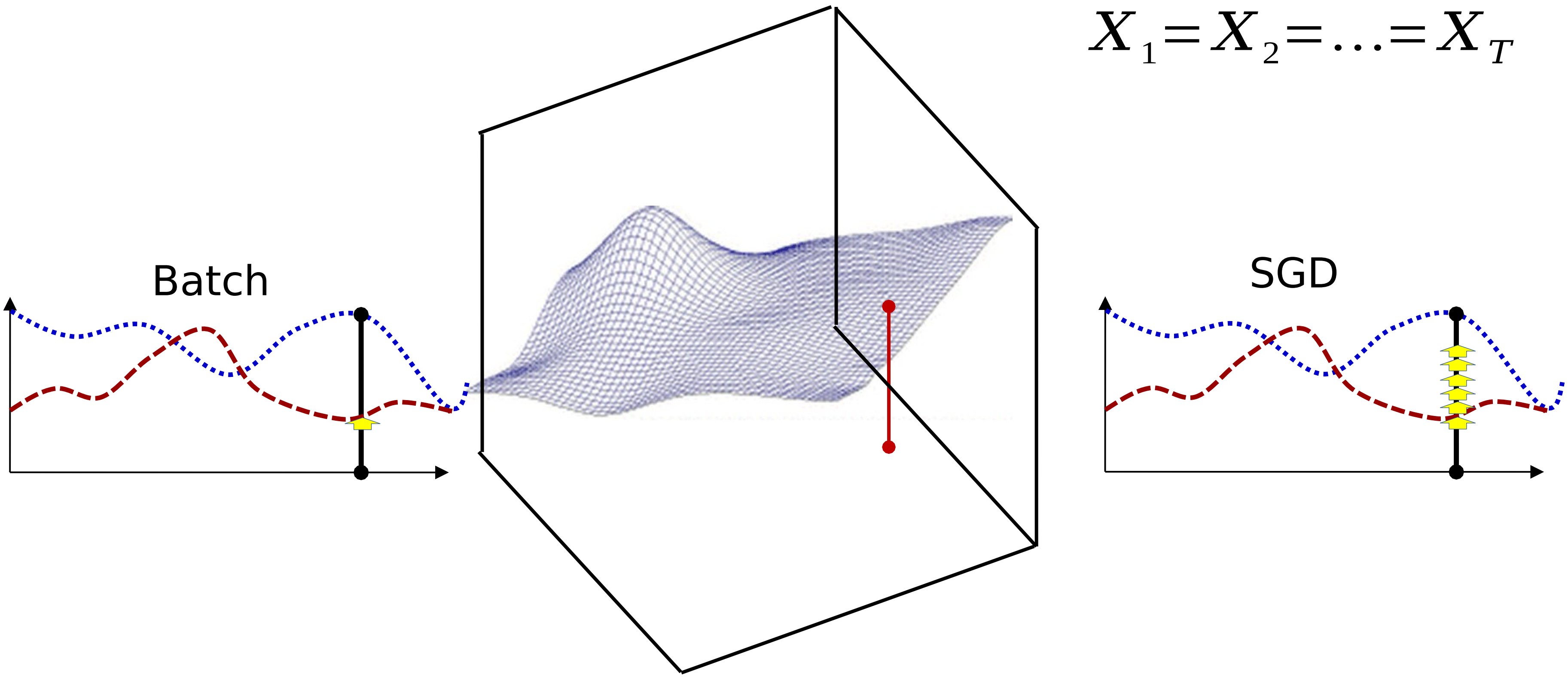




- The individual training instance contribute identical directions to the overall gradient
  - The final gradient points is simply the gradient for an individual instance

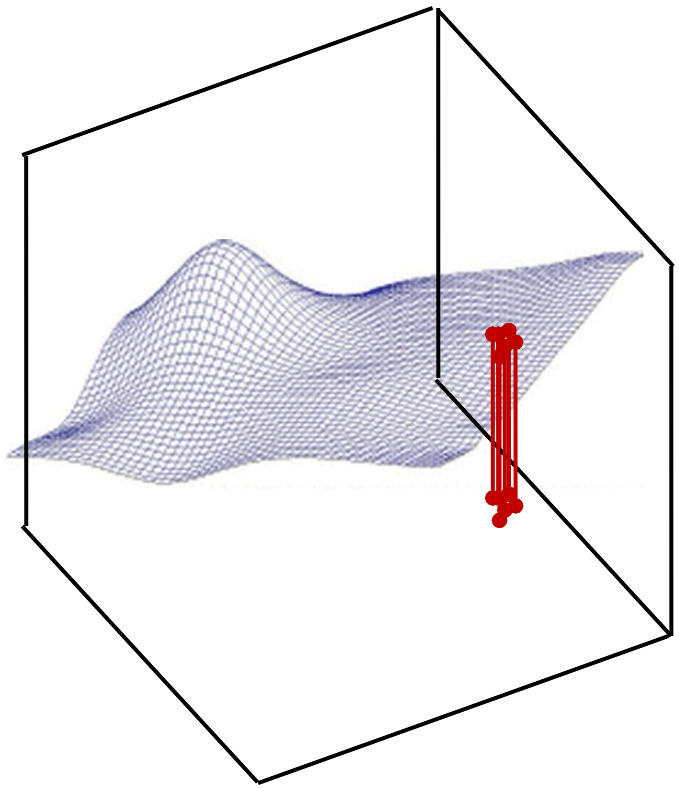
Batch vs SGD

$$X_1 = X_2 = \ldots = X_T$$



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

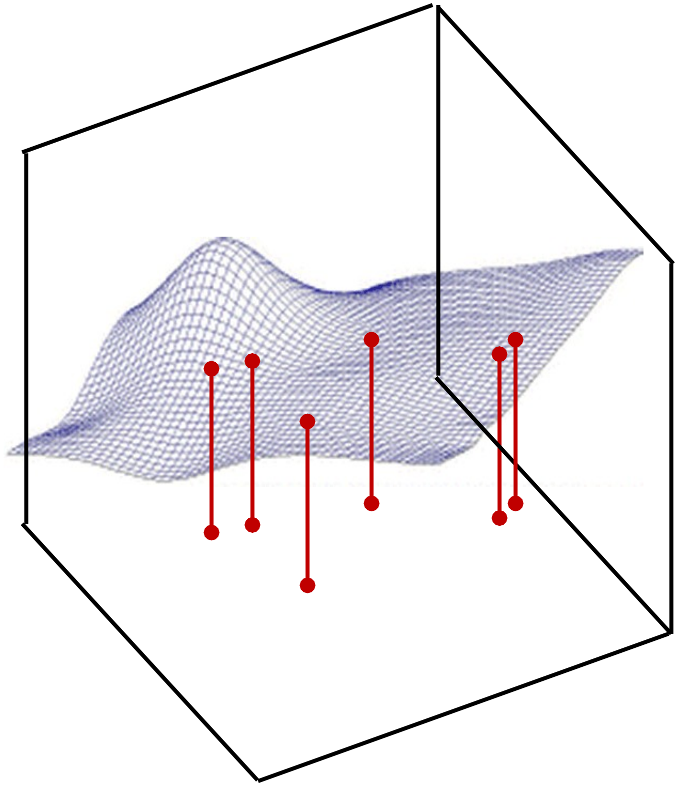
Clumpy data..



$$\boldsymbol{X}_1 \approx \boldsymbol{X}_2 \approx \ldots \approx \boldsymbol{X}_T$$

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

Clumpy data..

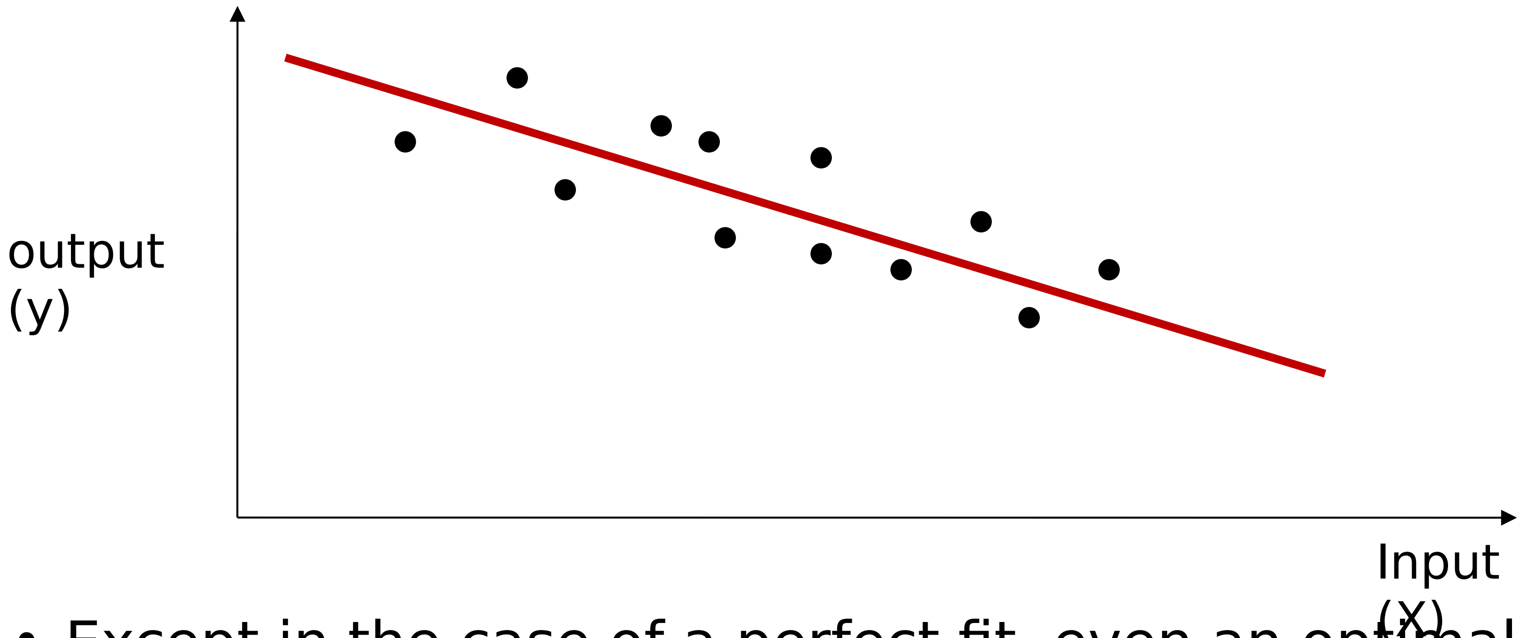


- As data get increasingly diverse, the benefits of incremental updates decrease, but do not entirely vanish

When does it work

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

Caveats: learning rate



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

Incremental Update: Stochastic Gradient Descent

- Given , ,…,
- Initialize all weights;
- Do:
  - Randomly permute , ,…,
  - For all
    - For every layer :
      - Compute
      - Update
- Until has converged

Incremental Update: Stochastic Gradient Descent

- Given , ,...,
- Initialize all weights;
- Do:
  - Randomly permute , ,...,
  - For all
    - For every layer :
      - Compute
      - Update
- Until has converged

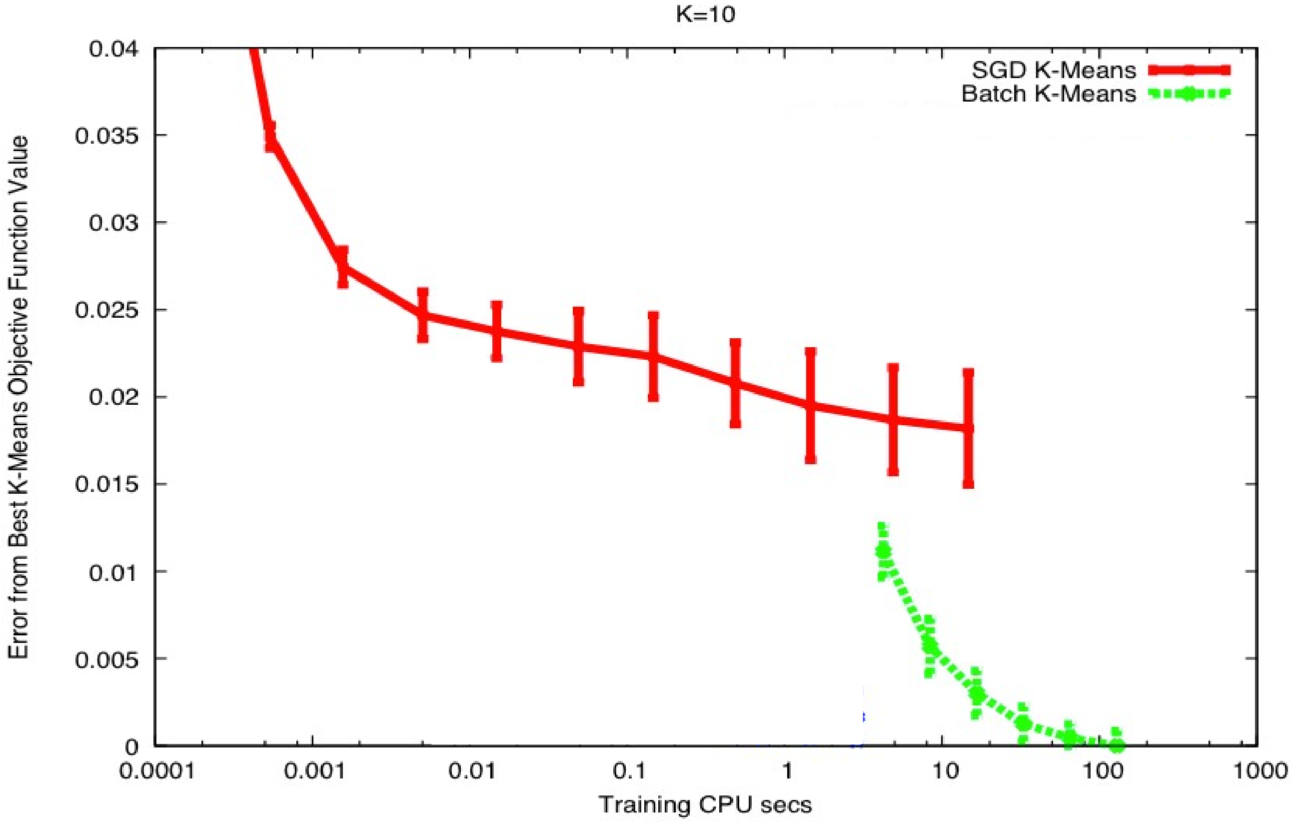
Randomize input order

Learning rate reduces with j

SGD convergence

- SGD converges "almost surely" to a global or local minimum for most functions
  - Sufficient condition: step sizes follow the following conditions (Robbins and Munro 1951)
    - Eventually the entire parameter space can be searched
    - The steps shrink
  - The fastest converging series that satisfies both above requirements is
    - 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..

Poll 2

PIAZZA @576

Select all that are true

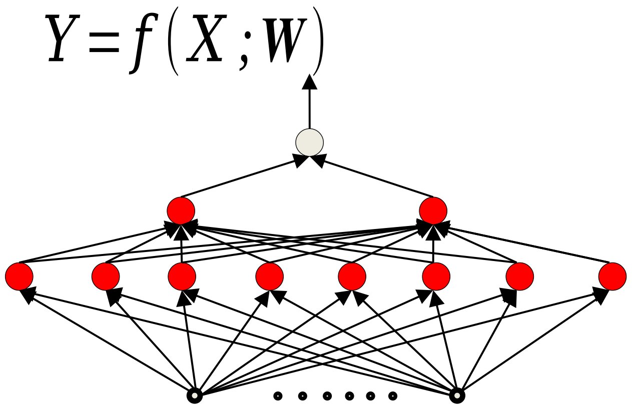
- SGD is an online version of batch updates
- SGD can have oscillatory behavior if we do not randomize the order of the inputs
- SGD can converge faster than batch updates, but arrive at poorer optima
- SGD convergence to the global optimum can only be guaranteed if step sizes shrink across iterations,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

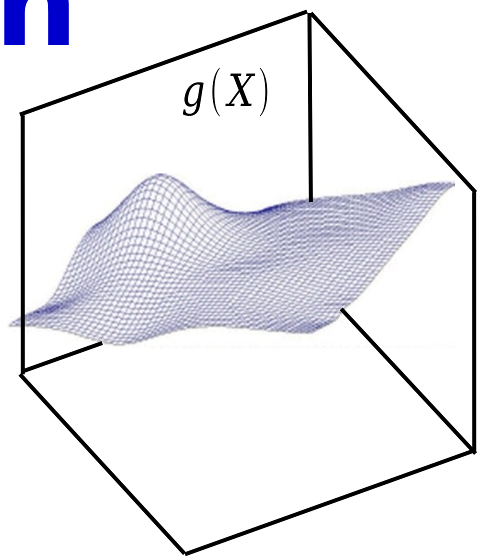
Poll 2

Select all that are true [**all correct**]

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

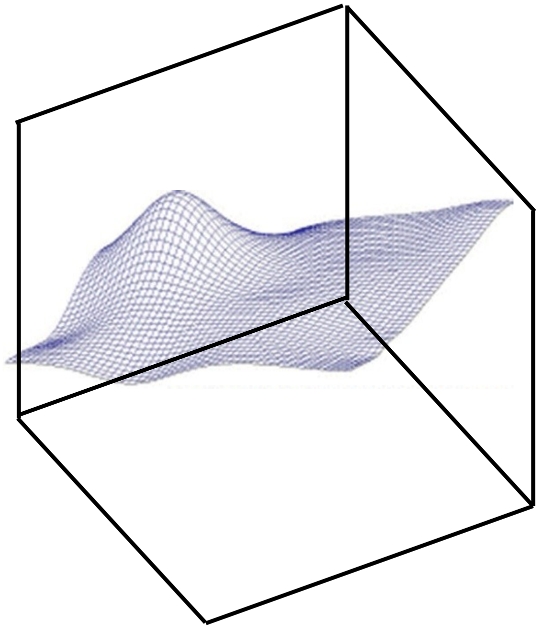
Recall: Modelling a function

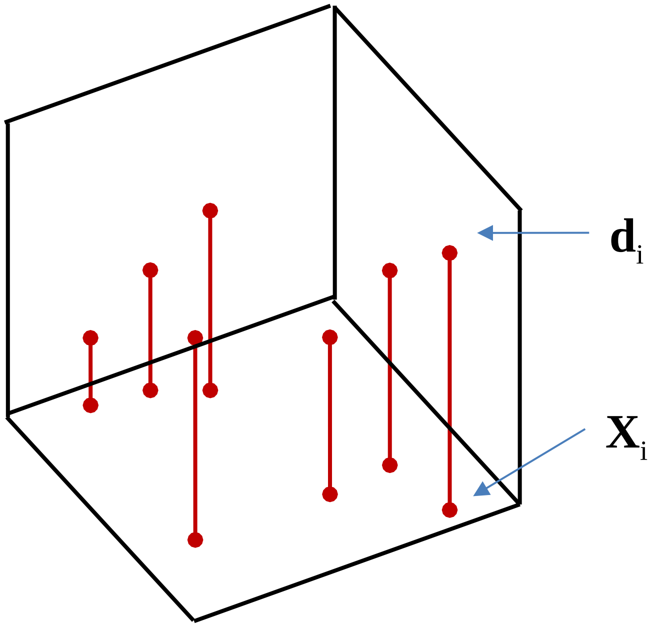




- To learn a network to model a function we minimize the *expected divergence*

Recall: The *Empirical* risk

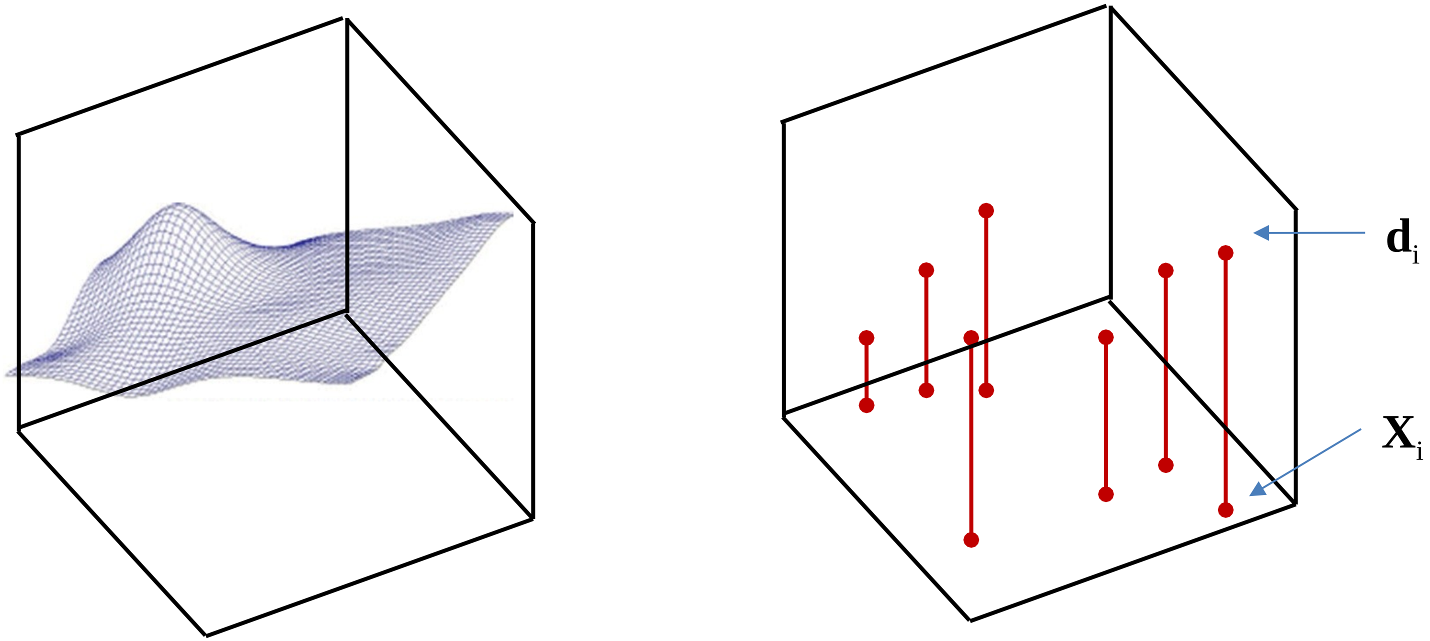




- In practice, we minimize the *empirical risk (or loss)*

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

Recall: The *Empirical* risk

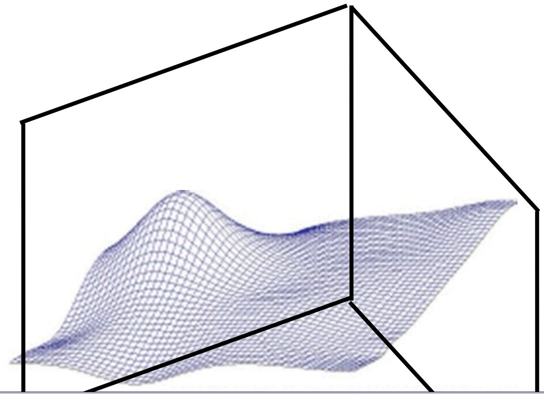


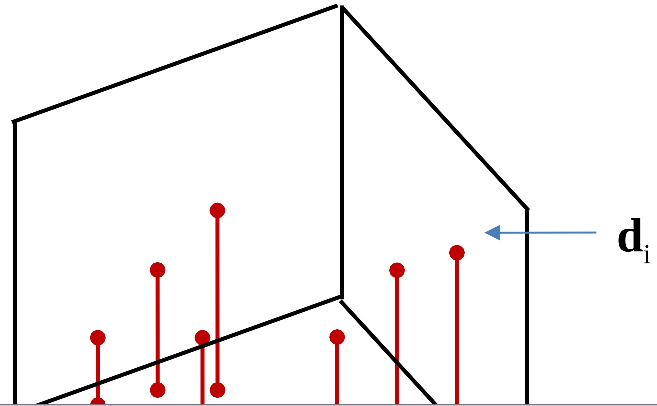
- In practice, we minimize the *empirical risk (or loss)*

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

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

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

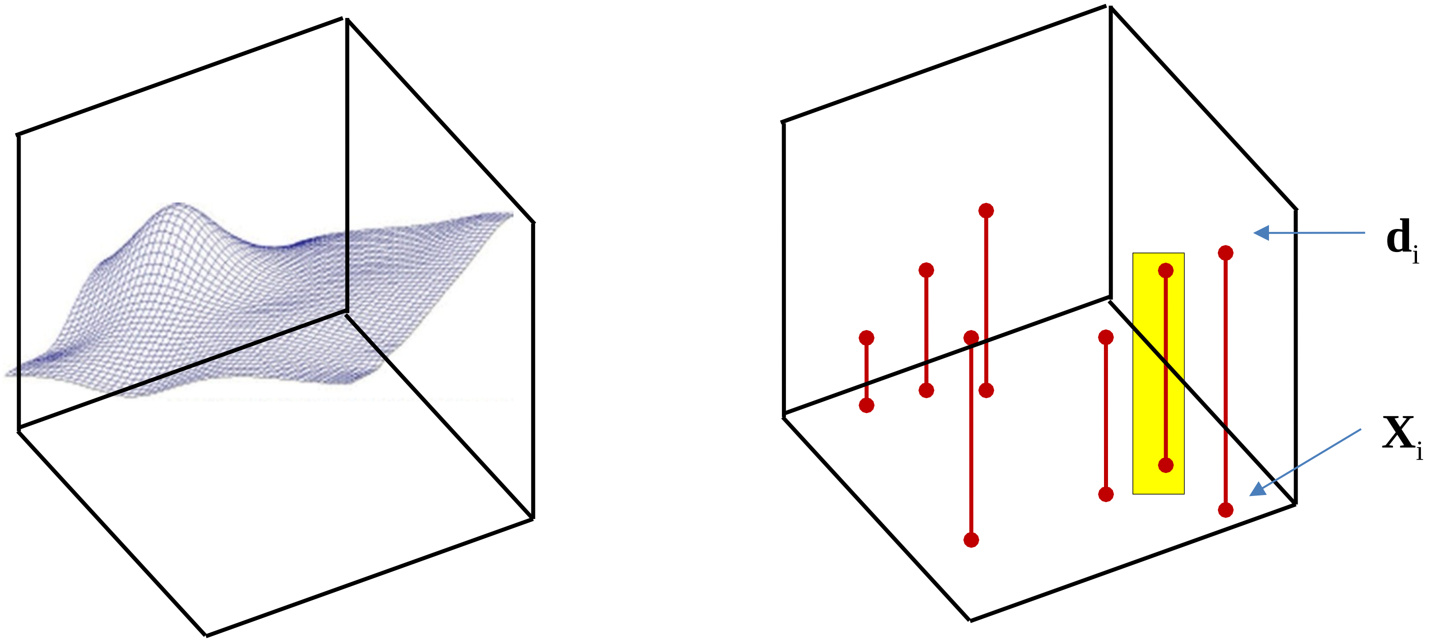
- In practice, we minimize the *empirical risk*

The empirical risk is an unbiased estimate of the expected divergence

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

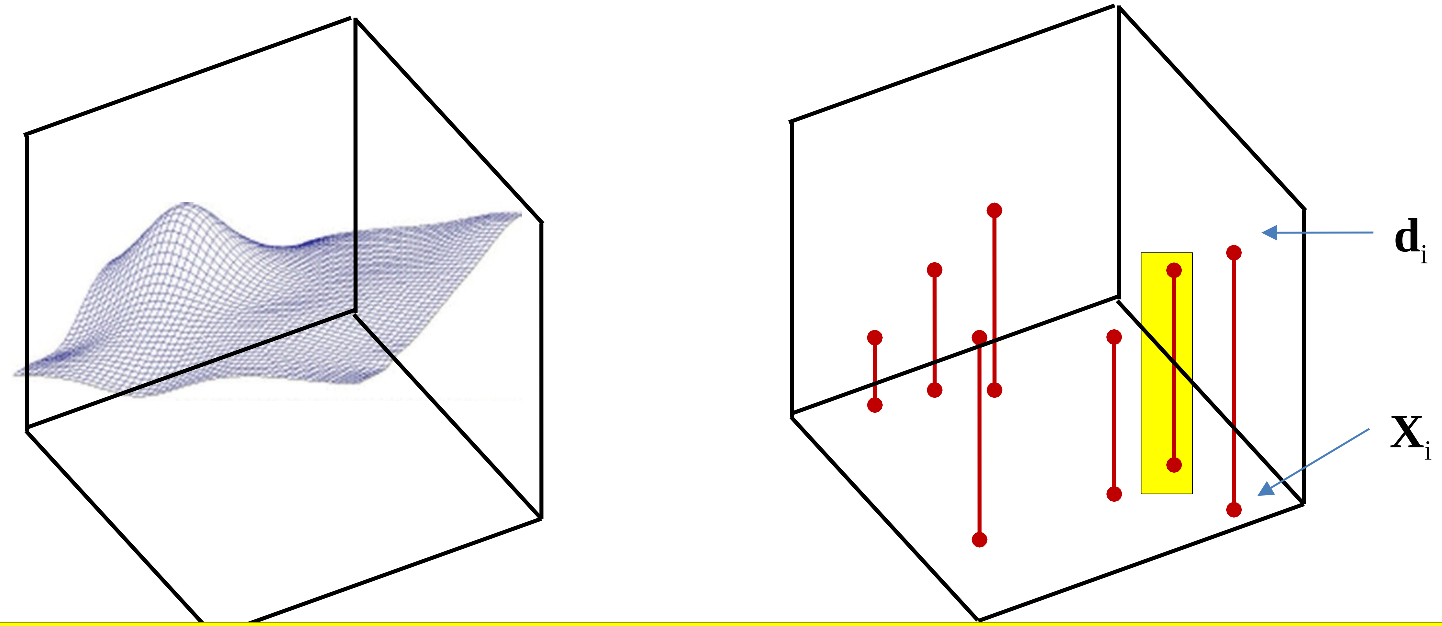
- The *expected value* of the *empirical risk* is actually the *expected divergence*

SGD



- At each iteration, **SGD** focuses on the divergence of a ***single*** sample
- The *expected value* of the *sample error* is ***still*** the *expected divergence*

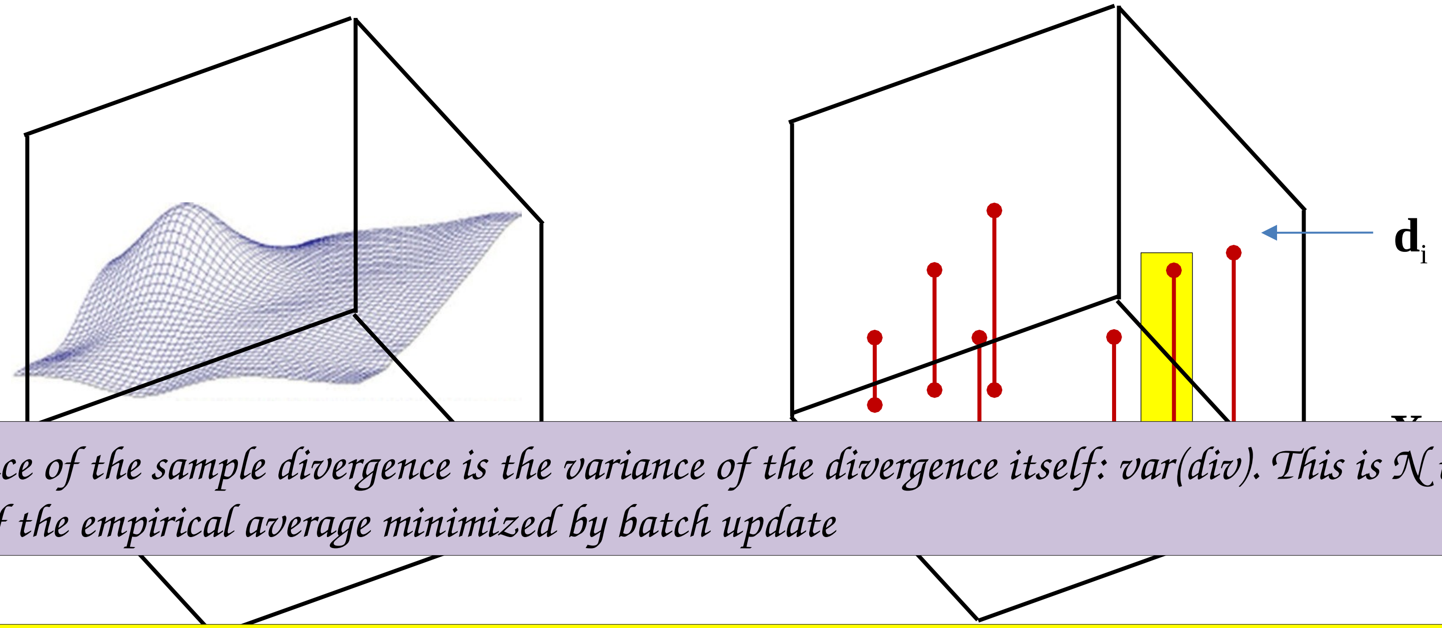
SGD



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

- At each iteration, **SGD** focuses on the divergence of a ***single*** sample
- The *expected value* of the *sample error* is ***still*** the *expected divergence*

SGD

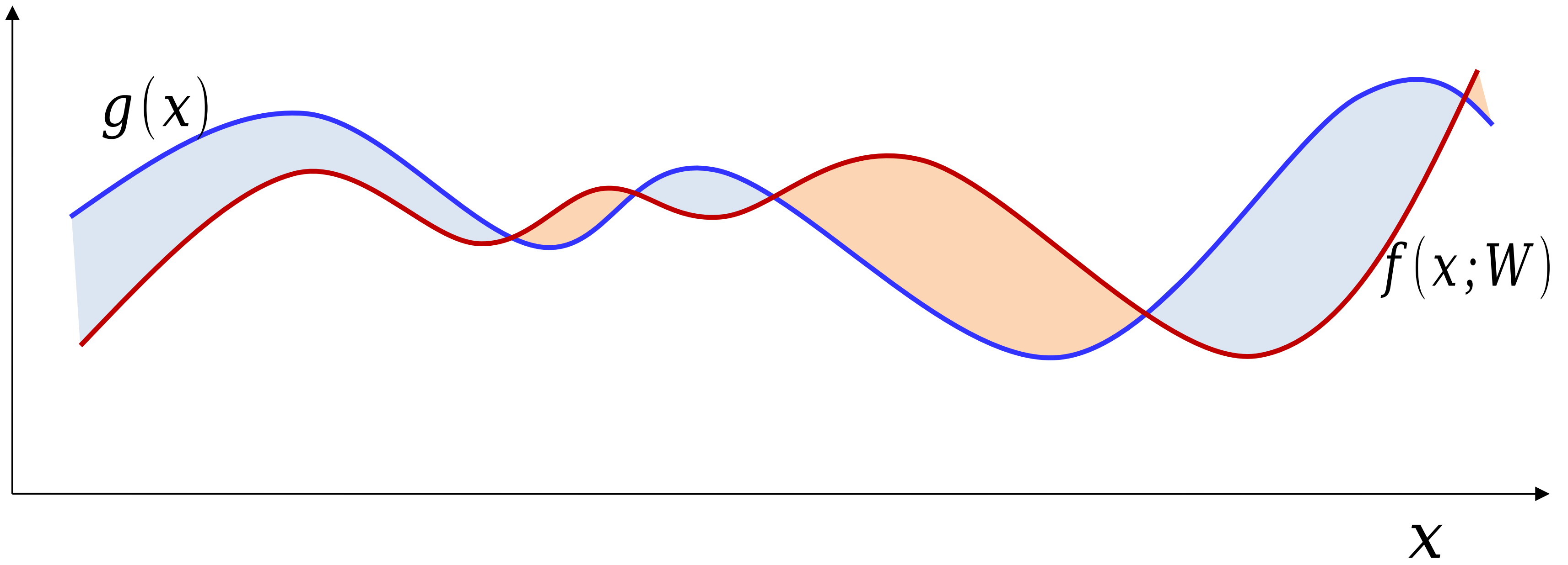


The variance of the sample divergence is the variance of the divergence itself: var(div). This is N times the variance of the empirical average minimized by batch update

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

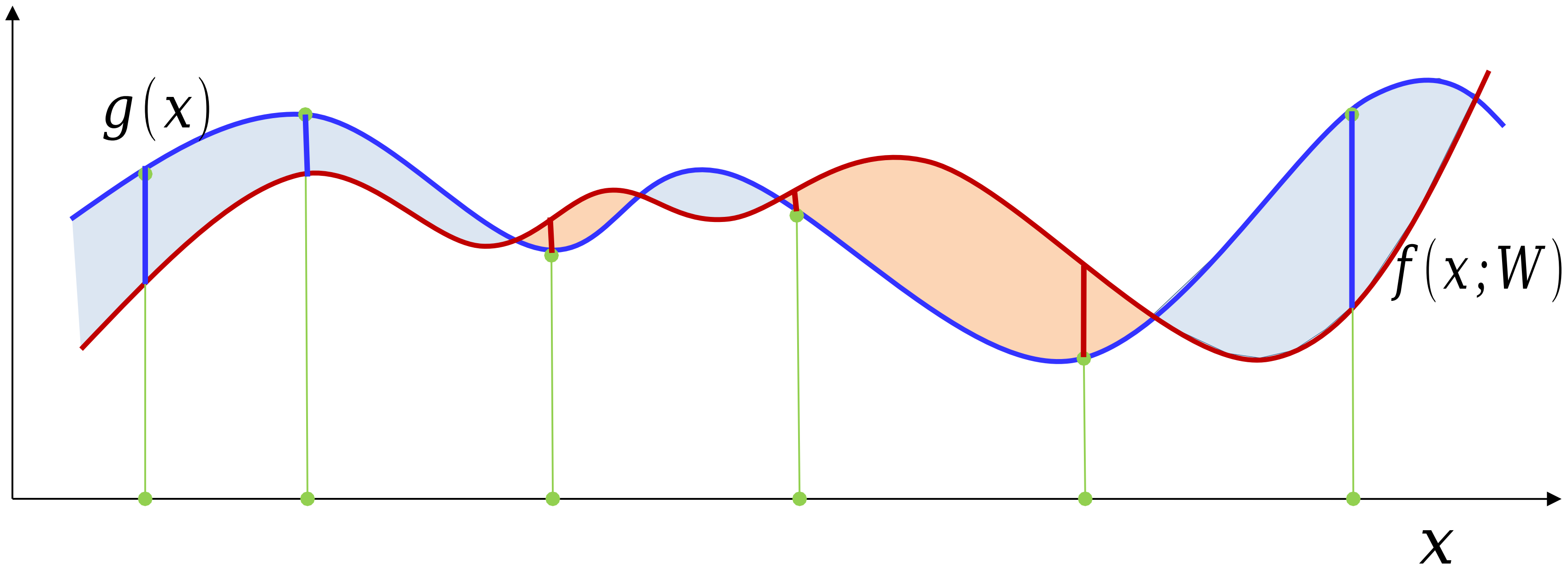
- At each iteration, **SGD** focuses on the divergence of a ***single*** sample
- The *expected value* of the *sample error* is ***still*** the *expected divergence*

Explaining the variance



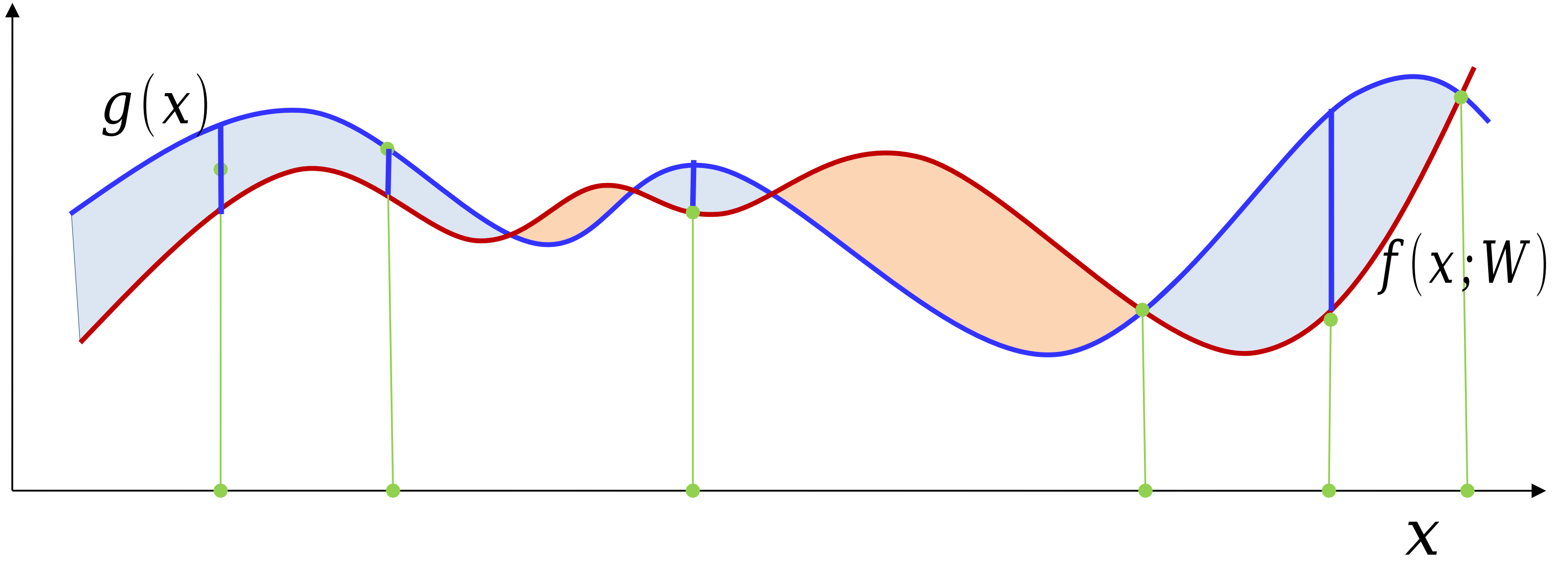
- The blue curve is the function being approximated
- The red curve is the approximation by the model at a given
- The heights of the shaded regions represent the point-by-point error
  - The divergence is a function of the error
  - We want to find the that minimizes the average divergence

Explaining the variance



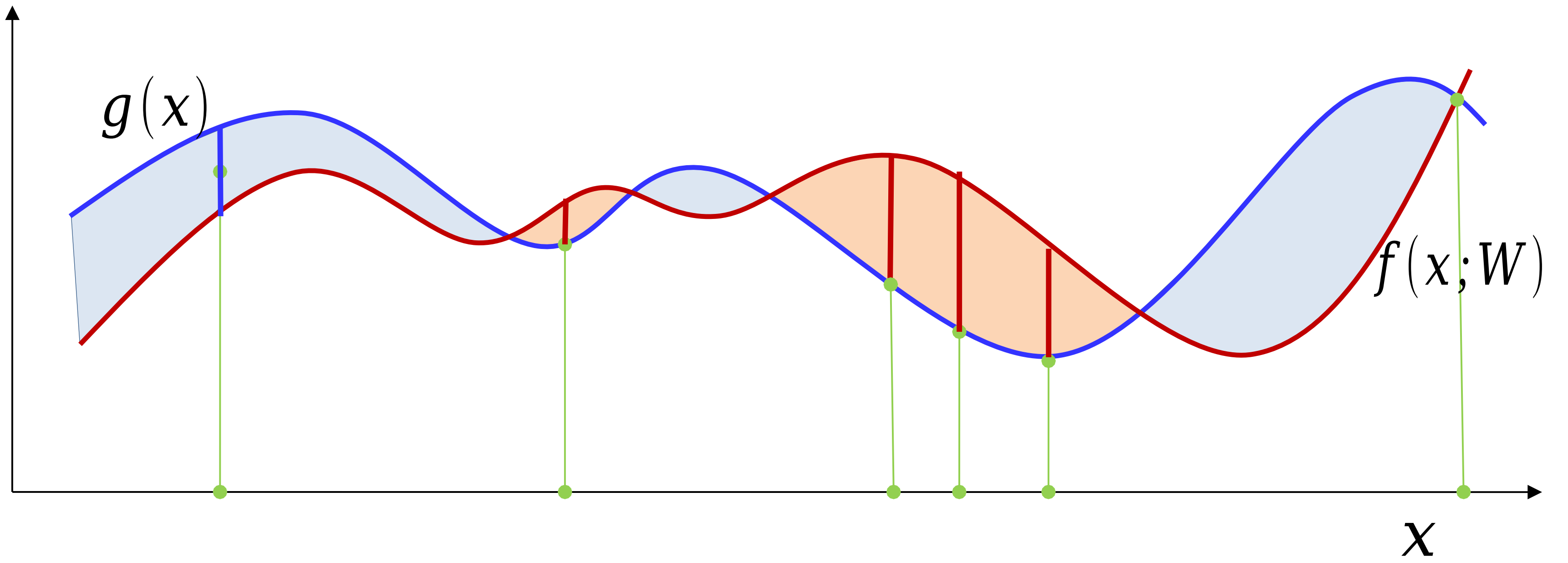
- Sample estimate approximates the shaded area with the average length of the lines

Explaining the variance



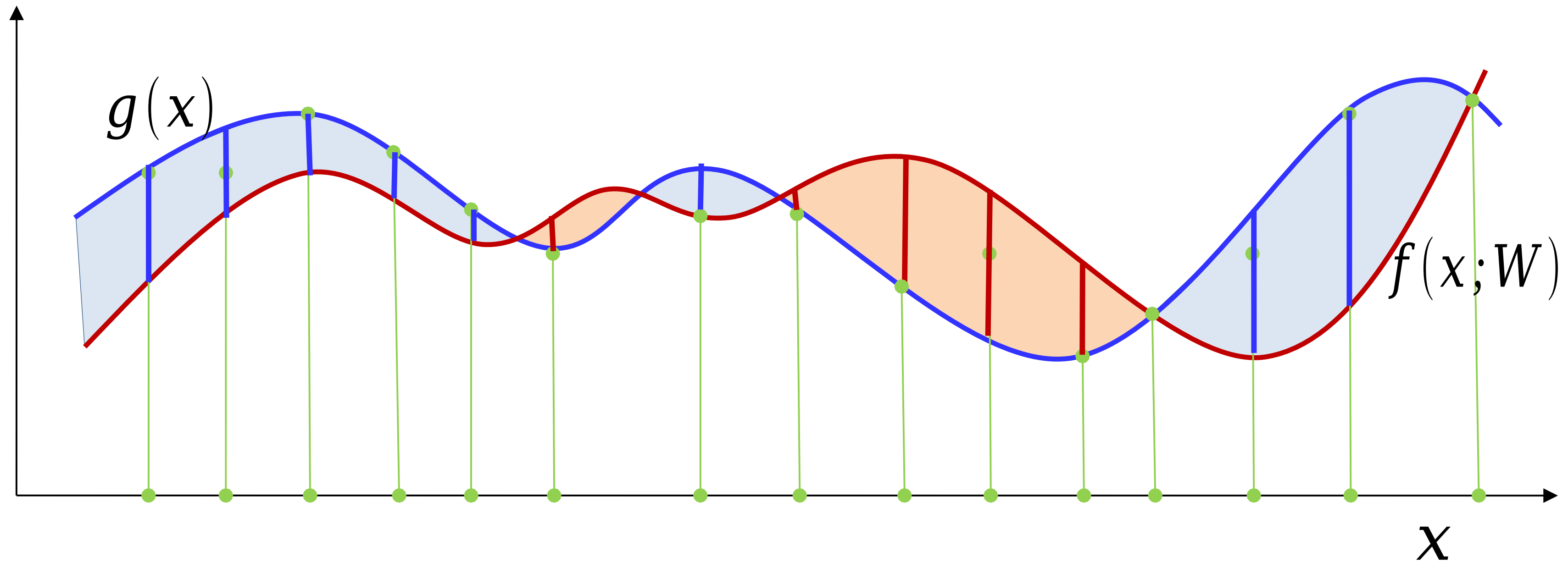
- Sample estimate approximates the shaded area with the average length of the lines
- This average length will change with position of the samples

Explaining the variance



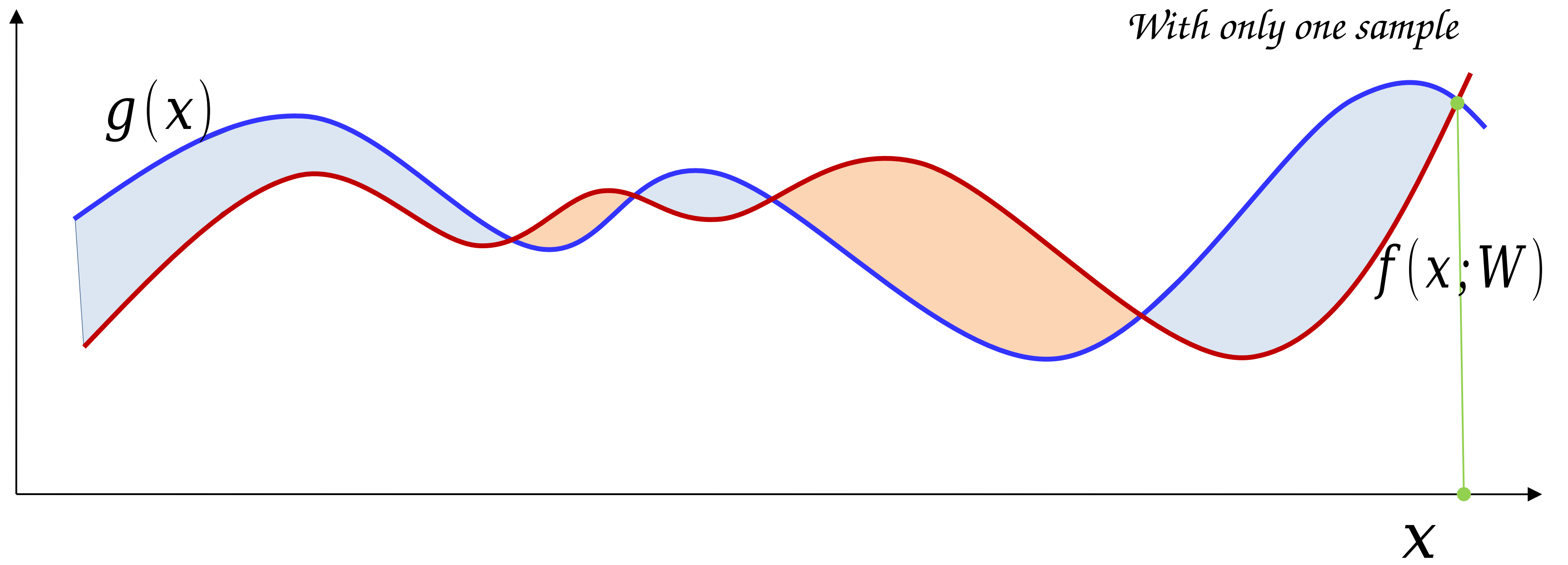
- Sample estimate approximates the shaded area with the average length of the lines
- This average length will change with position of the samples

Explaining the variance



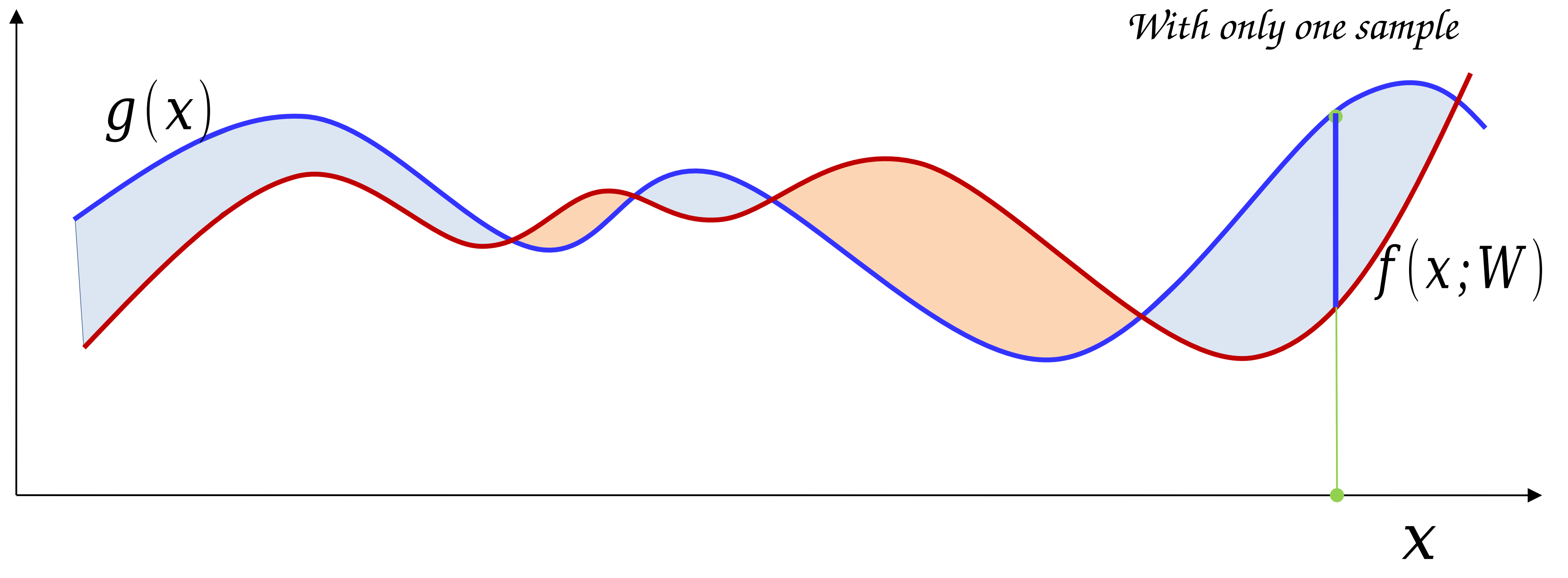
- Having more samples makes the estimate more robust to changes in the position of samples
  - The variance of the estimate is smaller

Explaining the variance



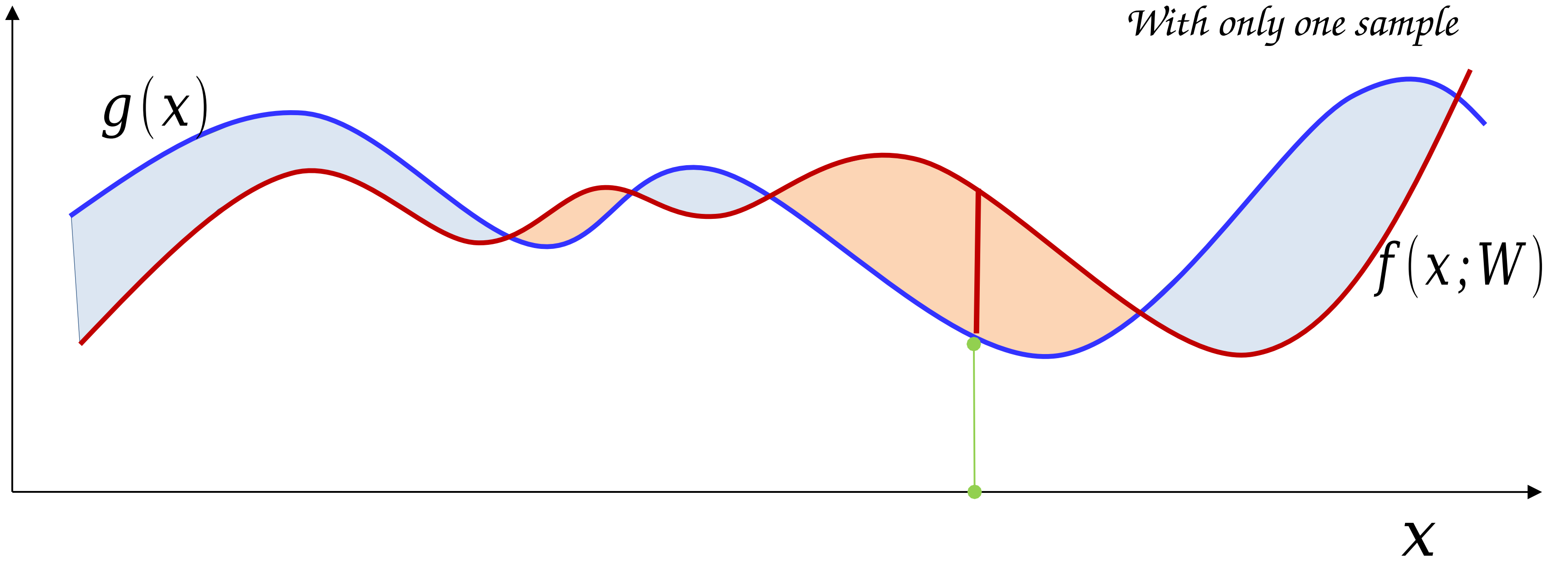
- Having very few samples makes the estimate swing wildly with the sample position
  - Since our estimator learns the to minimize this estimate, the learned too can swing wildly

Explaining the variance



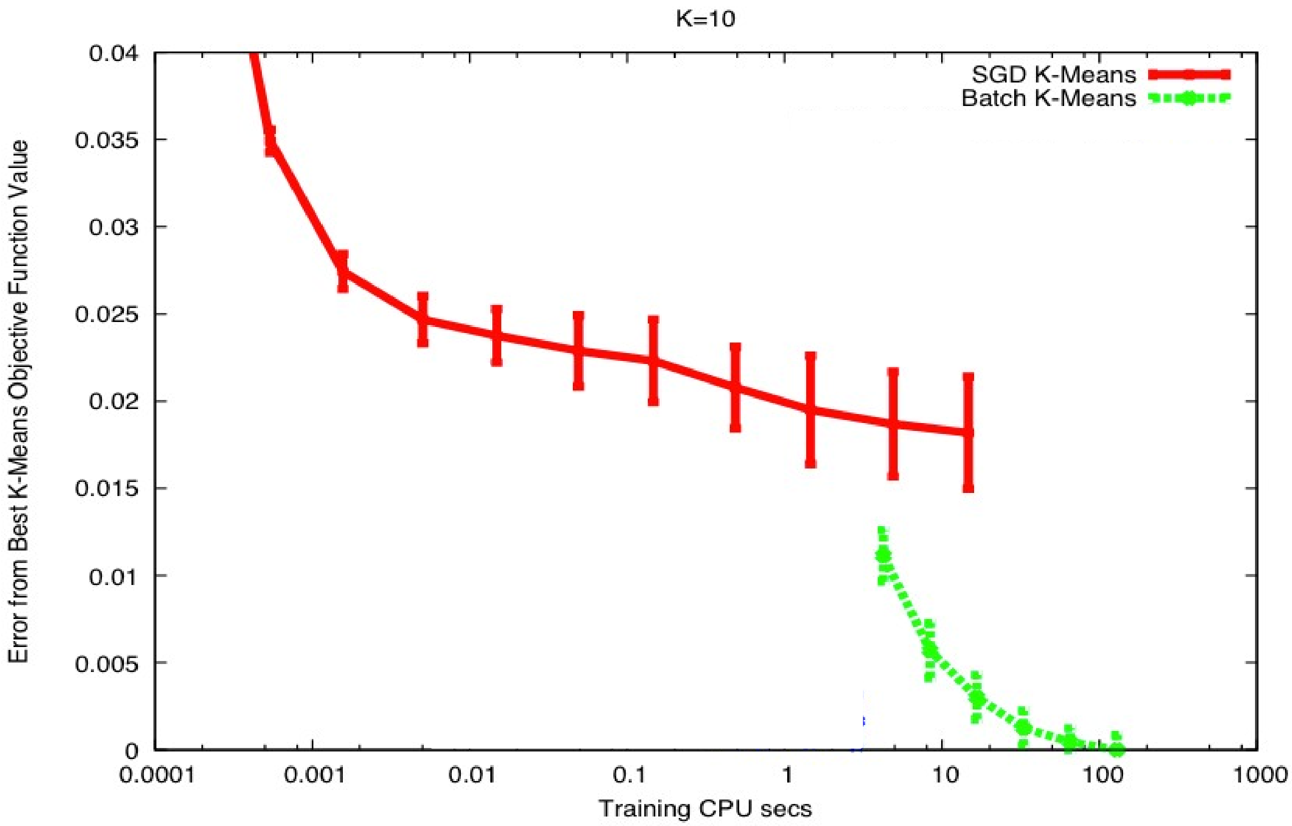
- Having very few samples makes the estimate swing wildly with the sample position
  - Since our estimator learns the to minimize this estimate, the learned too can swing wildly

Explaining the variance



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

SGD example



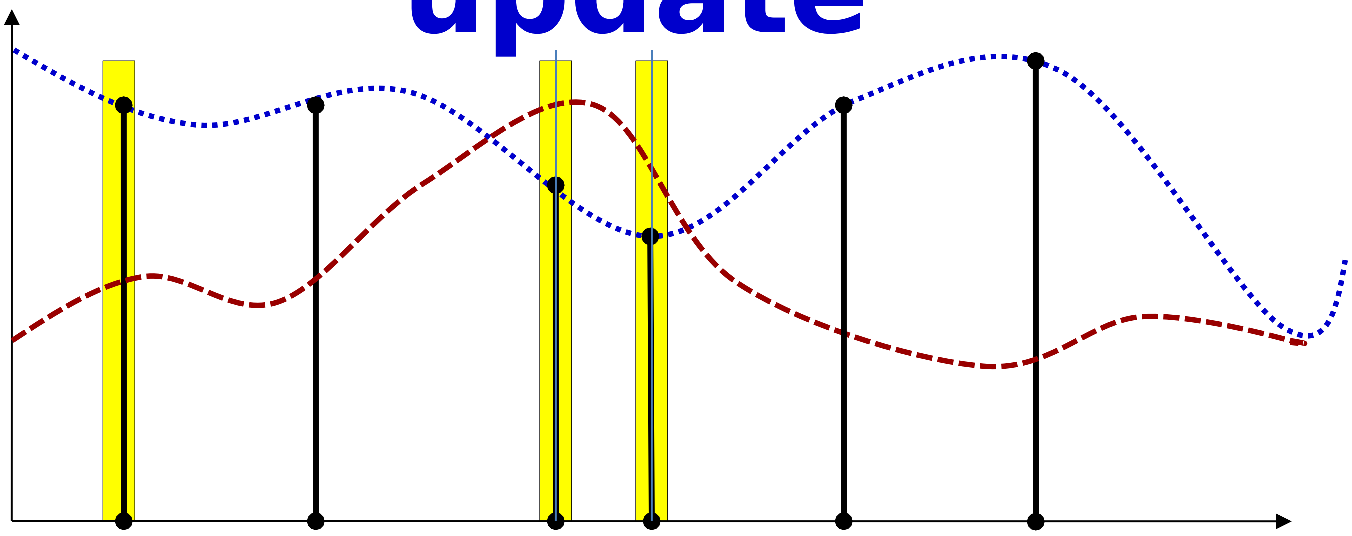
- A simpler problem: K-means
- Note: SGD converges faster
- But also has large variation between runs

SGD vs batch

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

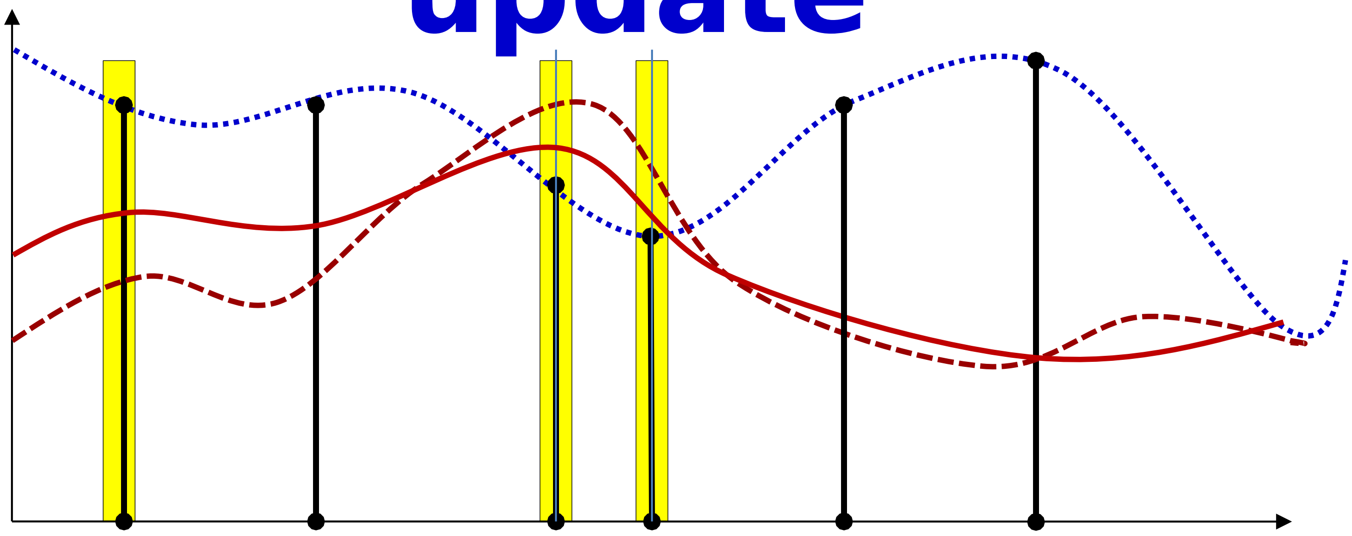
- But also provides significantly quicker updates than batch
- Is there a good medium?

Alternative: Mini-batch update



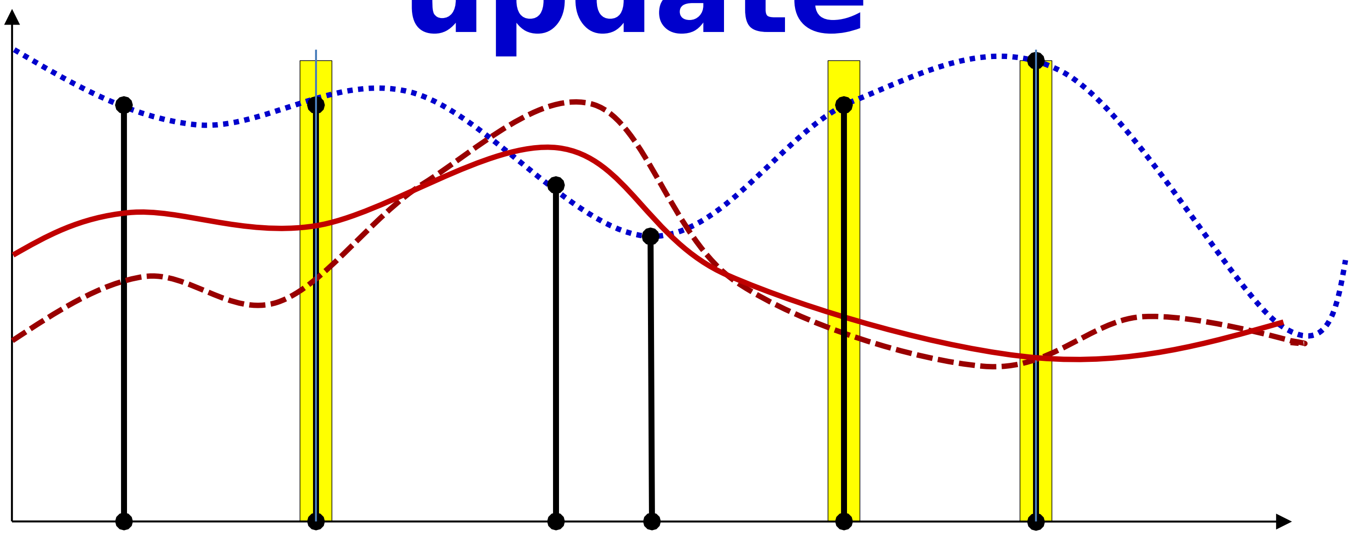
- Alternative: adjust the function at a small, randomly chosen subset of points
  - Keep adjustments small
  - If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data

Alternative: Mini-batch update



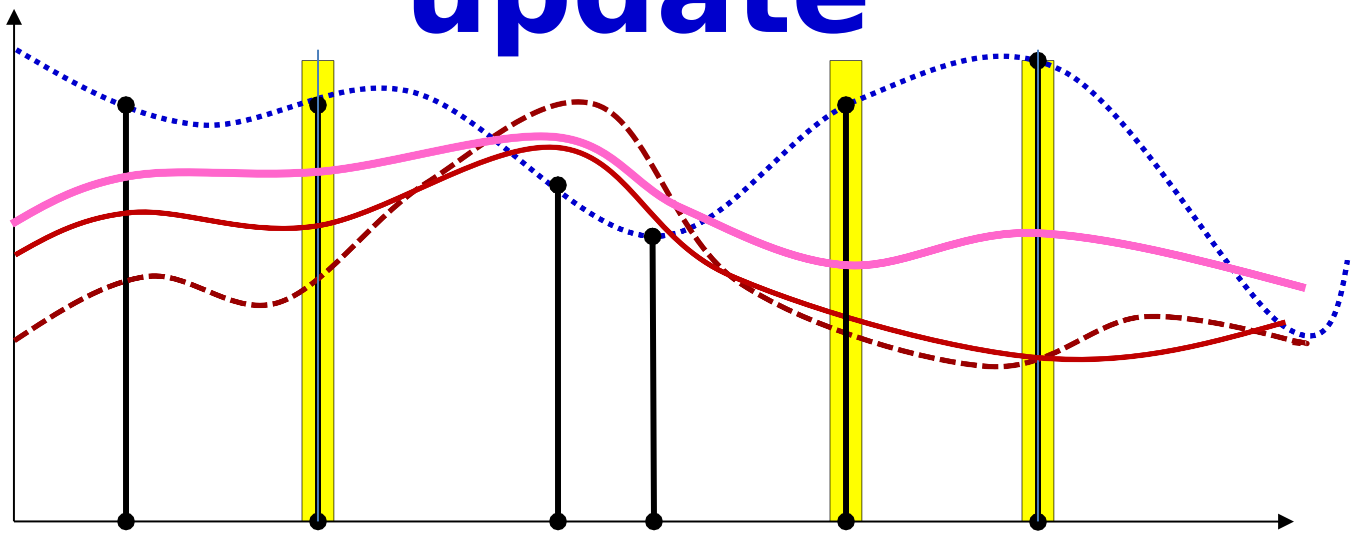
- Alternative: adjust the function at a small, randomly chosen subset of points
  - Keep adjustments small
  - If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data

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 , ,…,
- Initialize all weights*;*
- Do:
  - Randomly permute , ,…,
  - For
    - For every layer k:
    - For t’ = t : t+b-1
      - For every layer :
        - » Compute
    - Update
      - For every layer k:
- Until has converged

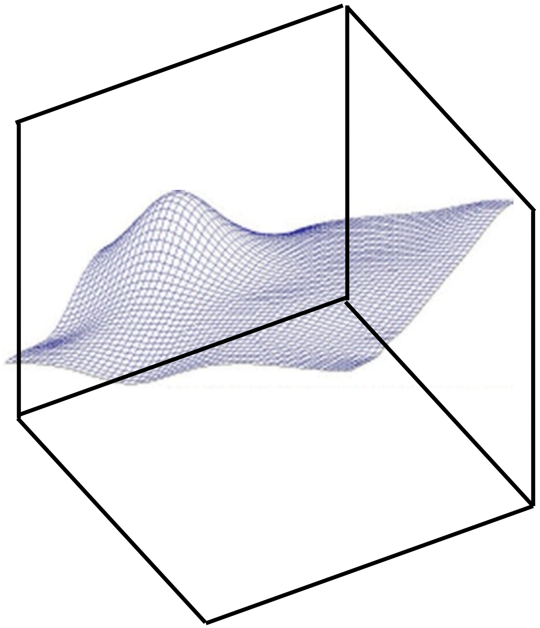
Incremental Update: Mini-batch update

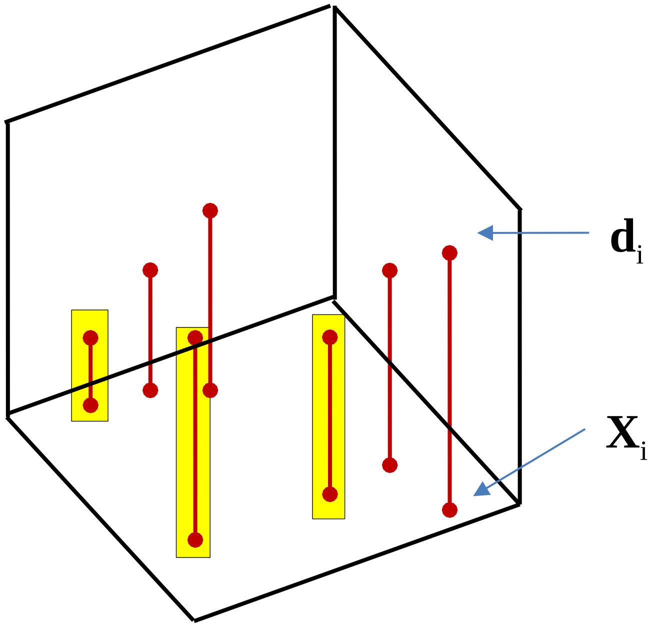
- Given , ,…,
- Initialize all weights*;*
- Do:
  - Randomly permute , ,…,
  - For
    - For every layer k:
    - For t' = t : t+b-1
      - For every layer :
        - » Compute
    - Update
      - For every layer k:
- Until has converged

Mini-batch size

Shrinking step size

Mini Batches

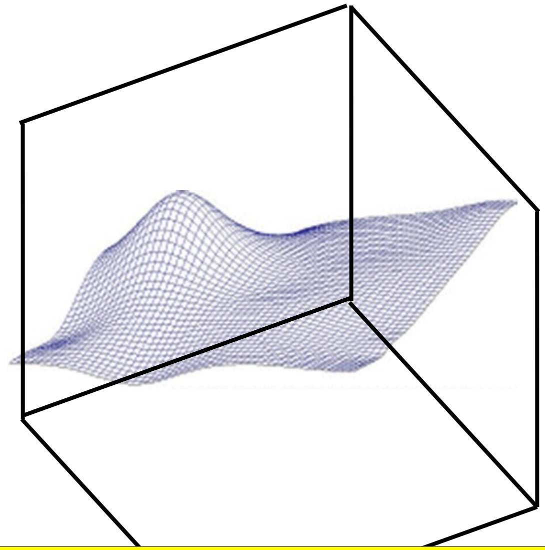


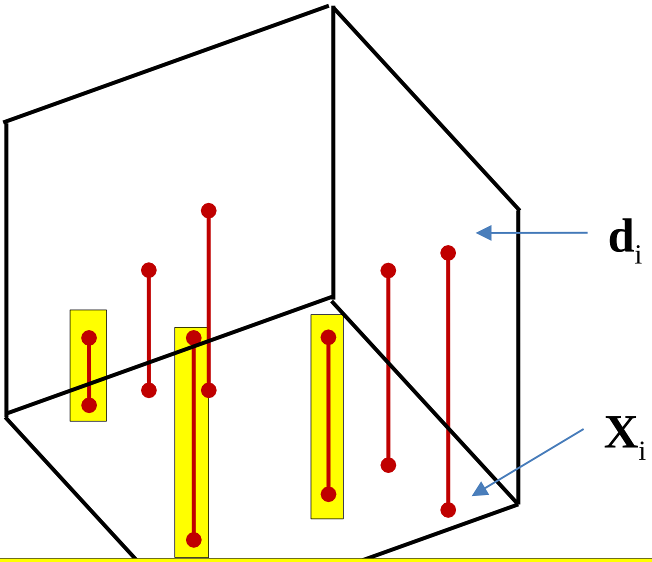


- Mini-batch updates compute and minimize a *batch loss*

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

Mini Batches



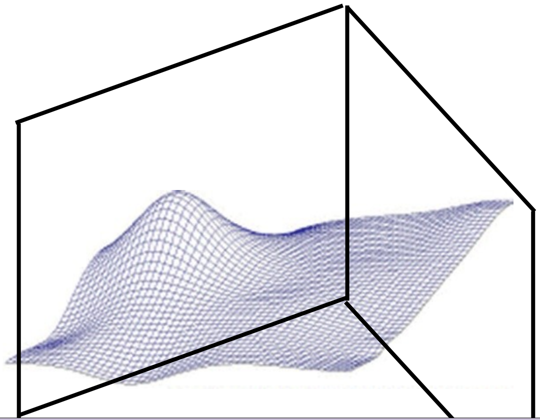


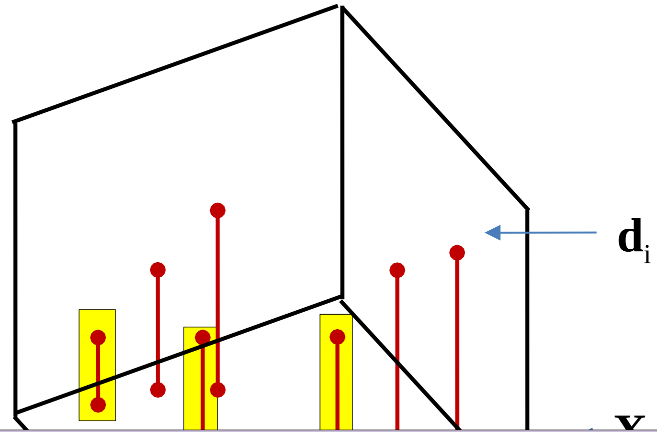
The minibatch loss is also an unbiased estimate of the expected loss

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

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

Mini Batches





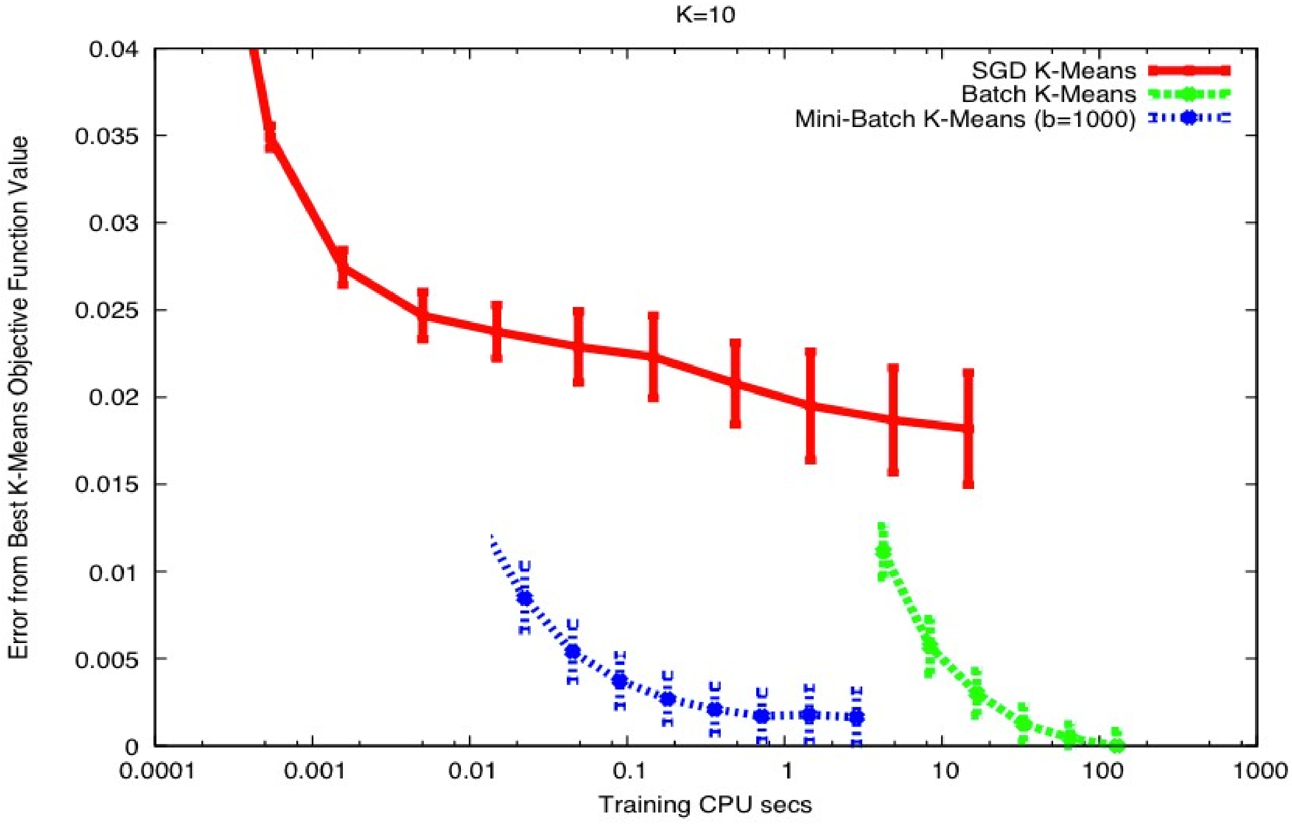
The variance of the minibatch loss: var(BatchLoss) = 1/b var(div)
This will be much smaller than the variance of the sample error in SGD

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

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

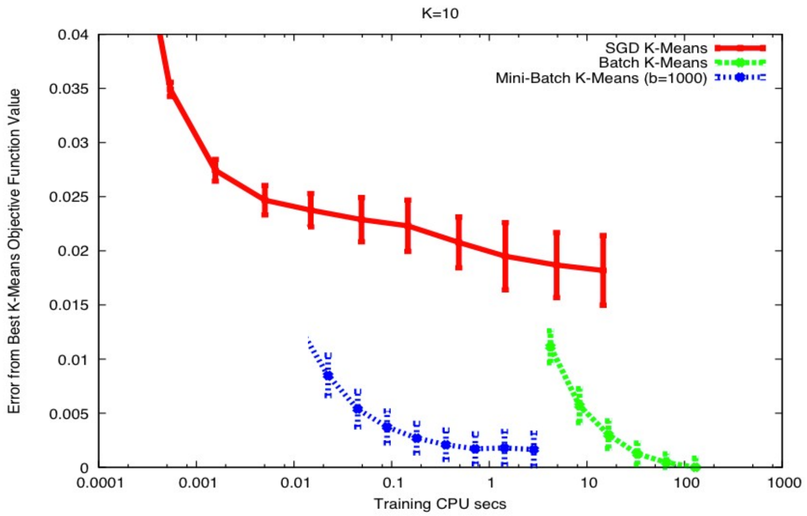
- The *expected value* of the *batch loss* is also the *expected divergence*

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 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

PIAZZA @577

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 **[false]**
- 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

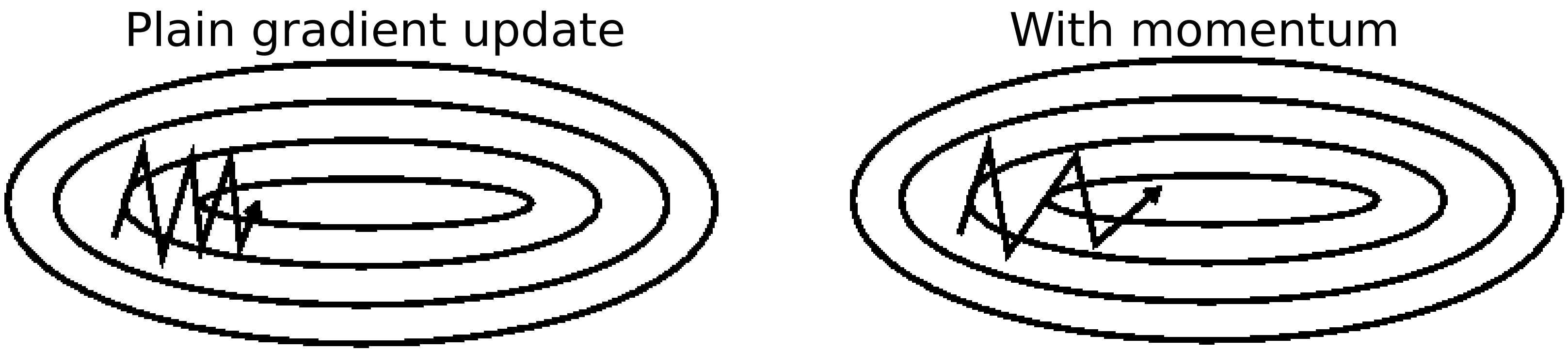
Story so far

- 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

Moving on: Topics for the day

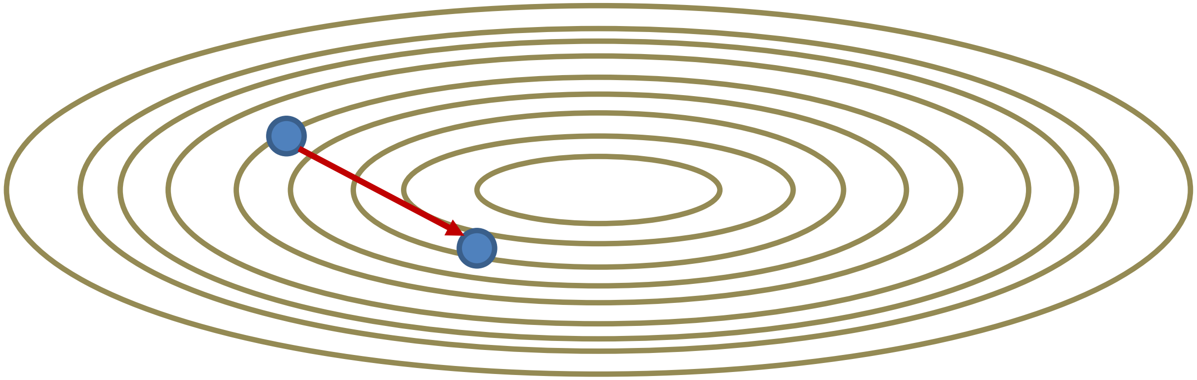
- Incremental updates
- Revisiting “trend” algorithms
- Generalization
- Tricks of the trade
  - Divergences..
  - Activations
  - Normalizations

Recall: Momentum Update



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

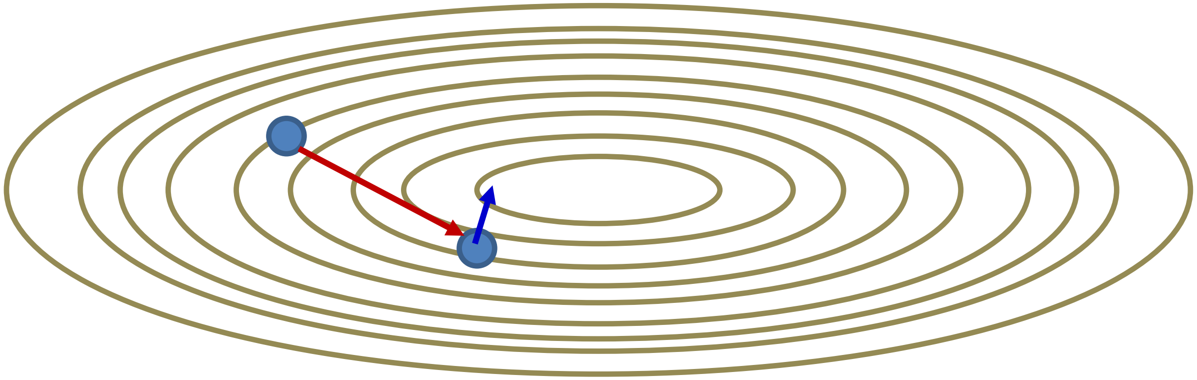
Recall: Momentum Update



- The momentum method

- At any iteration, to compute the current step:

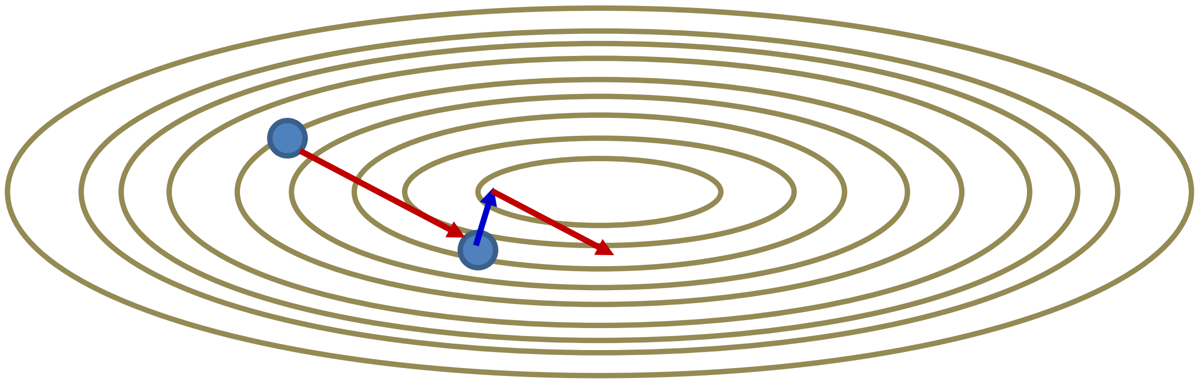
Recall: Momentum Update



- The momentum method

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

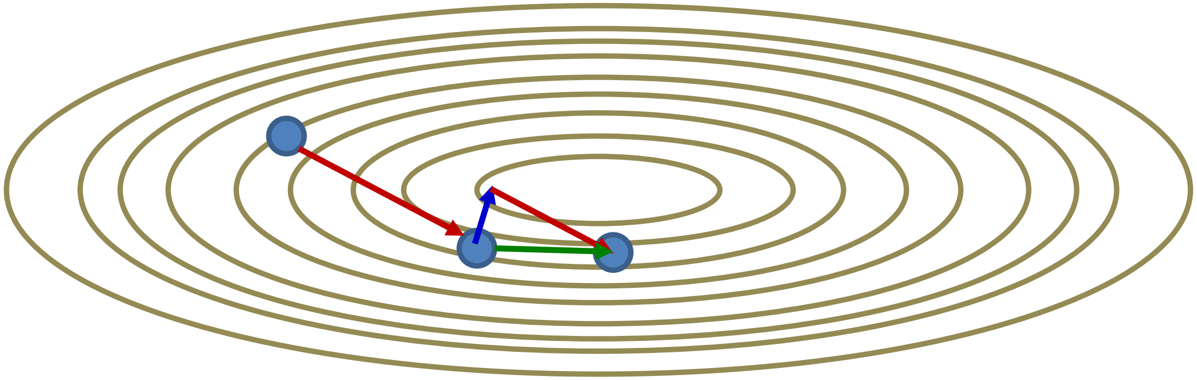
Recall: Momentum Update



- The momentum method

- 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

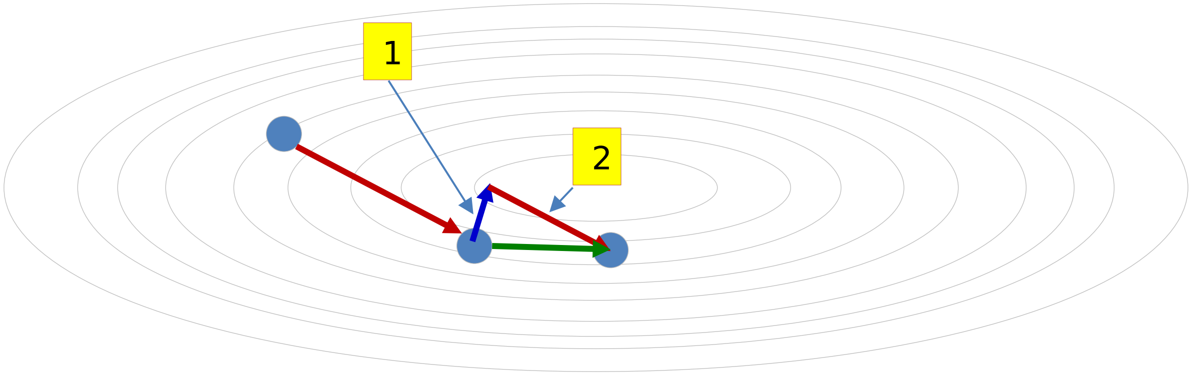
Recall: Momentum Update



- The momentum method

- 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

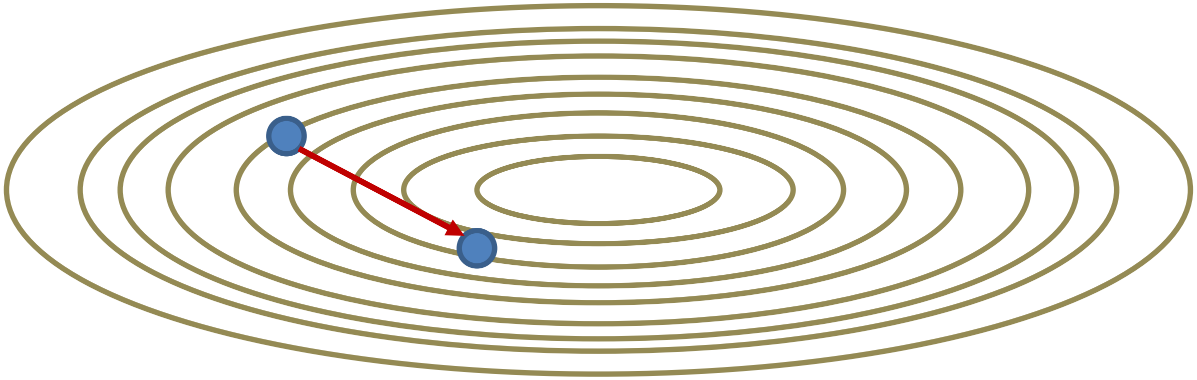
Momentum update



- 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

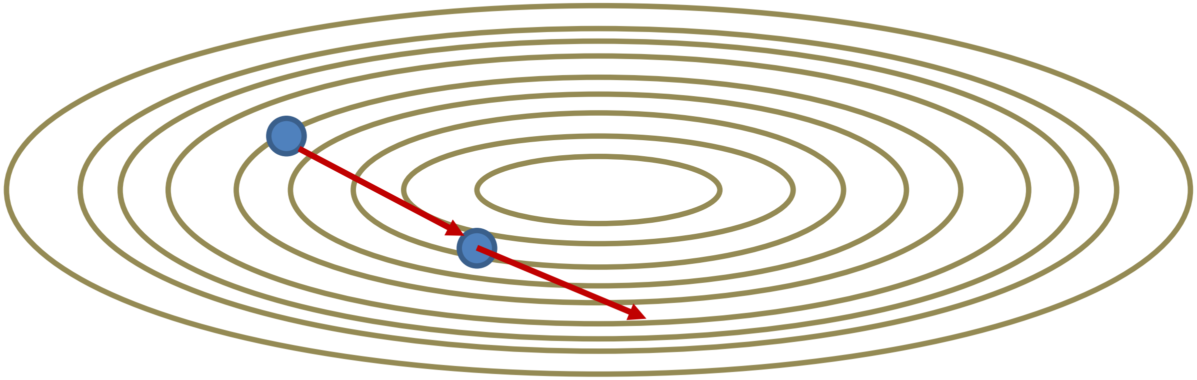
- The procedure can be made more optimal by reversing the order of operations..

Nestorov's Accelerated Gradient



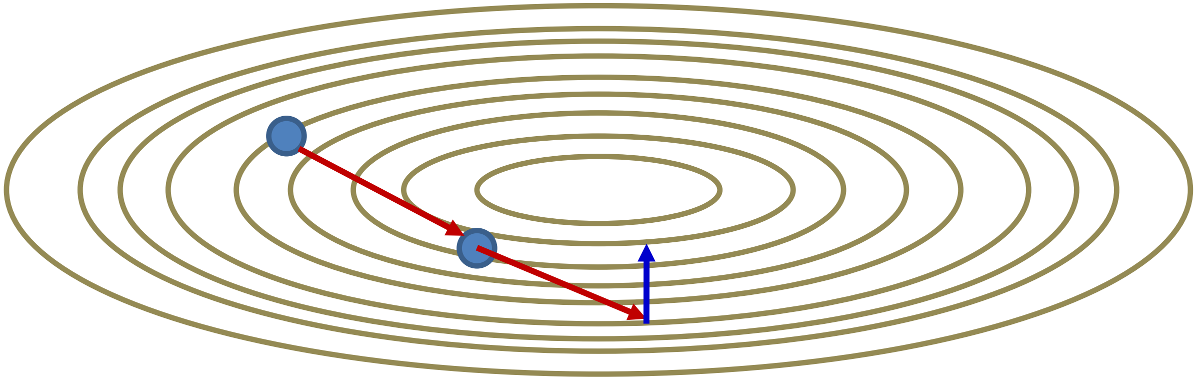
- Change the order of operations
- At any iteration, to compute the current step:

Nestorov’s Accelerated Gradient



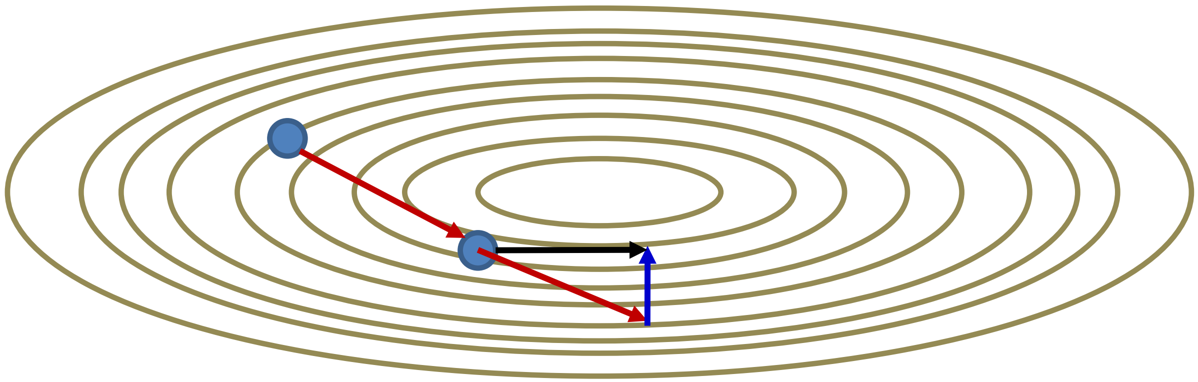
- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step

Nestorov's Accelerated Gradient



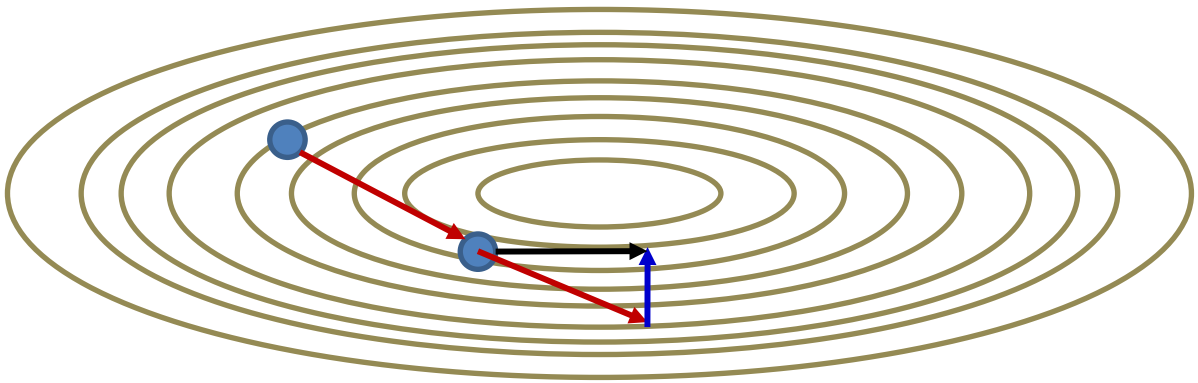
- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step
  - Then compute the gradient step at the resultant position

Nestorov's Accelerated Gradient



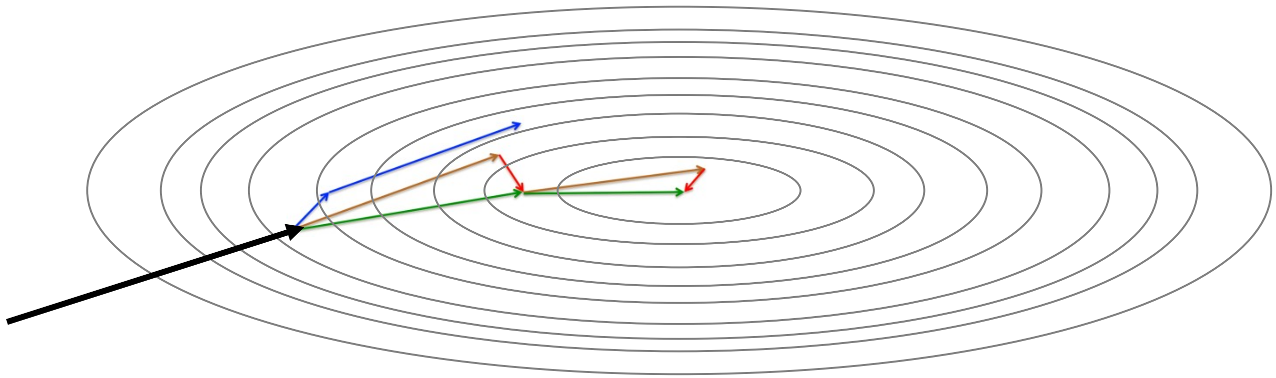
- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step
  - Then compute the gradient step at the resultant position
  - Add the two to obtain the final step

Nestorov's Accelerated Gradient



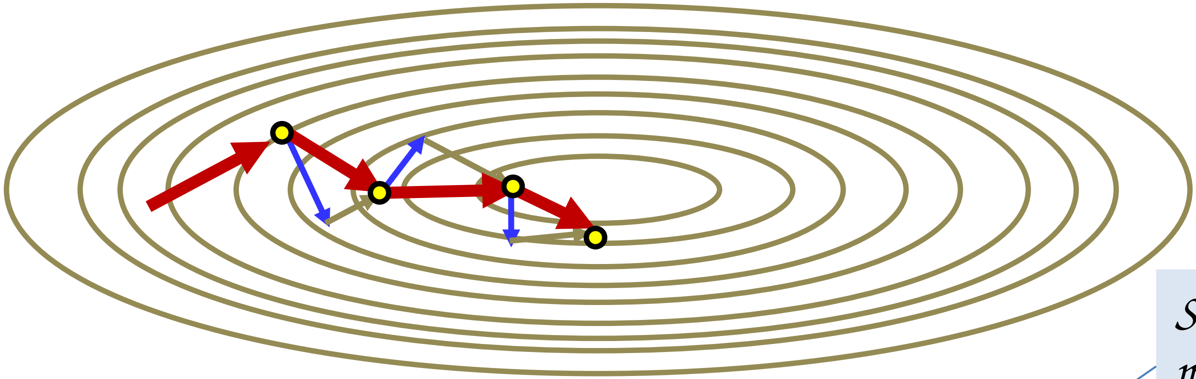
- Nestorov's method

Nestorov’s Accelerated Gradient



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

Momentum and incremental updates

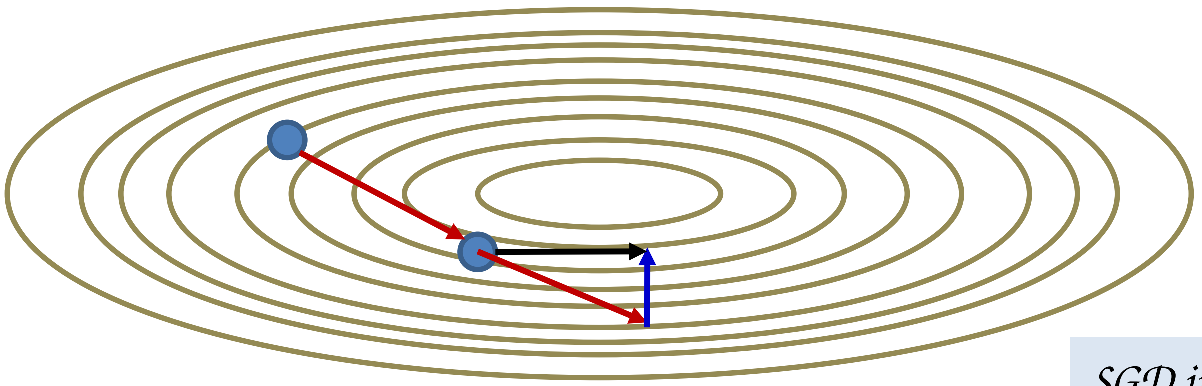


SGD instance or minibatch loss

- The momentum method

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

Nestorov's Accelerated Gradient



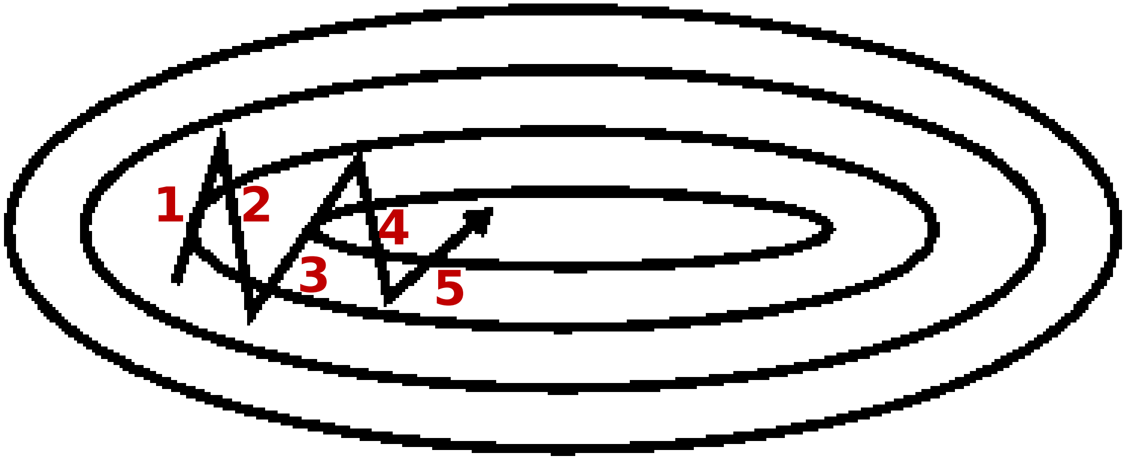
SGD instance or minibatch loss

- Nestorov's method

Still higher-order methods

- Momentum and Nestorov's method improve convergence by normalizing the *mean* of the derivatives
- More recent methods take this one step further by also considering their variance
  - RMS Prop
  - Adagrad
  - AdaDelta
  - ADAM: very popular in practice
  - …
- All roughly equivalent in performance

Smoothing the trajectory



Ste p	X componen t	Y component
1	1	+2.5
2	1	-3
3	2	+2.5
4	1	-2
5	1.5	1.5

- Observation: Steps in "oscillatory" directions show large total movement
  - In the example, total motion in the vertical direction is much greater than in the horizontal direction
  - Can happen even when momentum or Nesterov are used
- Improvement: Dampen step size in directions with high motion
  - ***Second order term***

Normalizing steps by second moment

- Modify usual gradient-based update:
  - Scale updates in every component in inverse proportion to the total movement of that component in recent past
    - *According to their variation (not just their average)*
- This will change the relative update sizes for the individual components
  - In the above example it would scale *down* Y component
  - And scale *up* X component (in comparison)
- We will see two popular methods that embody this principle…

RMS Prop

- Notation:
  - Updates are ***by parameter***
  - Derivative of loss w.r.t any individual parameter is shown as
    - Batch or minibatch loss, or individual divergence for batch/minibatch/SGD
  - The ***squared*** derivative is
    - 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
- Modified update rule: We want to
  - scale down updates with large mean squared derivatives
  - scale up updates with small mean squared derivatives

RMS Prop

- This is a variant on the *basic* mini-batch SGD algorithm

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

RMS Prop

- This is a variant on the *basic* mini-batch SGD algorithm

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

Note similarity to RPROP
The magnitude of the derivative is being normalized out

ADAM: RMSprop with momentum

- RMS prop only considers a second-moment normalized version of the current gradient
- ADAM utilizes a smoothed version of the *momentum-augmented* gradient
  - Considers both first and second moments

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

ADAM: RMSprop with momentum

- RMS prop only considers a second-moment normalized version of the current gradient
- ADAM utilizes a smoothed version of the *momentum-augmented* gradient

Ensures that the and terms do not dominate in early iterations

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

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: ,
    - ADAM: , ,

Poll 4

PIAZZA @578

Which of the following are true

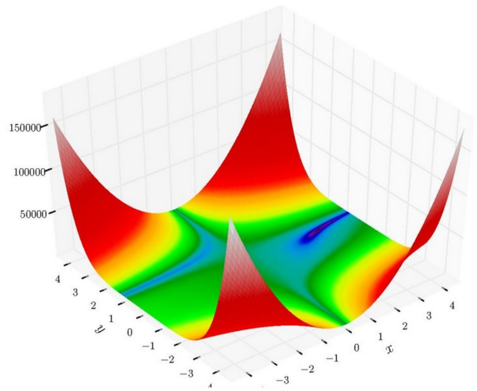
- Vanilla SGD considers the long-term trends of gradients in update steps
- Momentum methods consider the long-term average of derivatives to make updates
- RMSprop only considers the second order moment of derivatives, but not their average trend, to make updates
- ADAM considers both the average trend and 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 minibatch updates

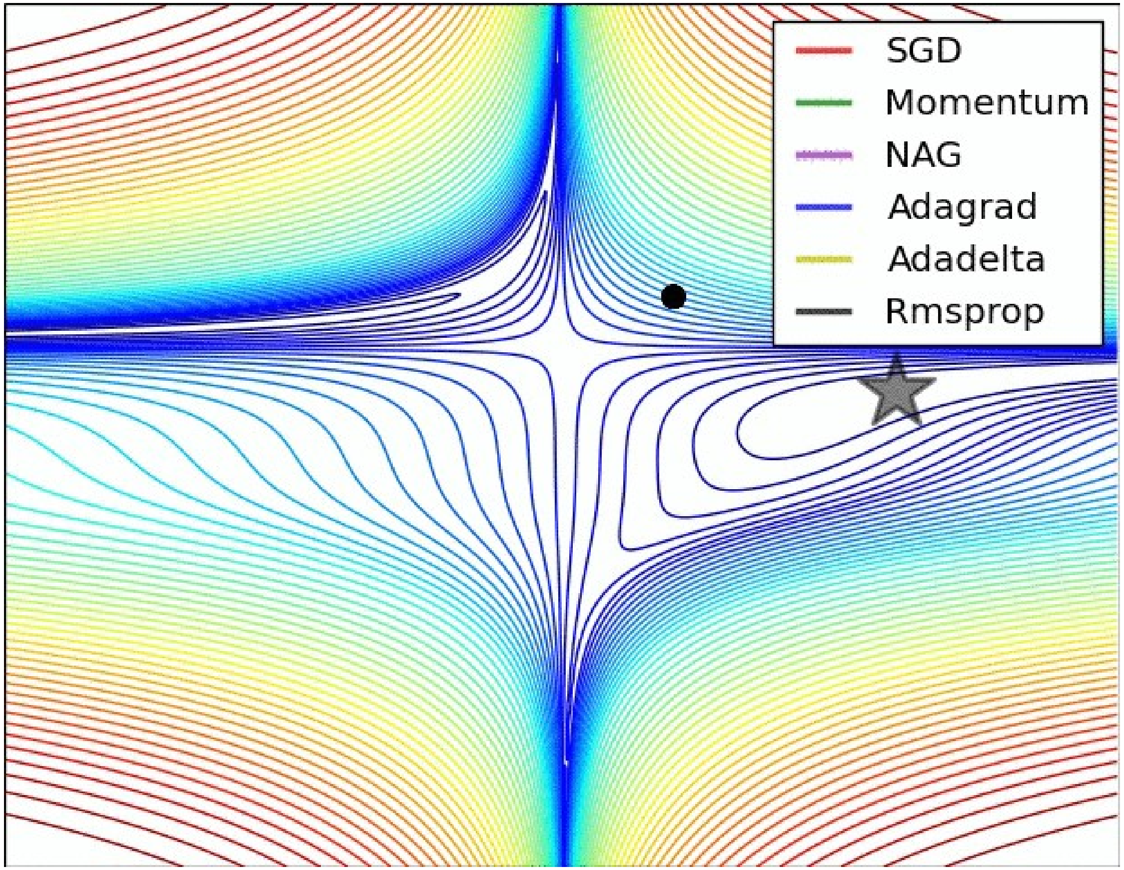
Poll 4

Which of the following are true

- Vanilla SGD considers the long-term trends of gradients in update steps **[false]**
- 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 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 minibatch updates

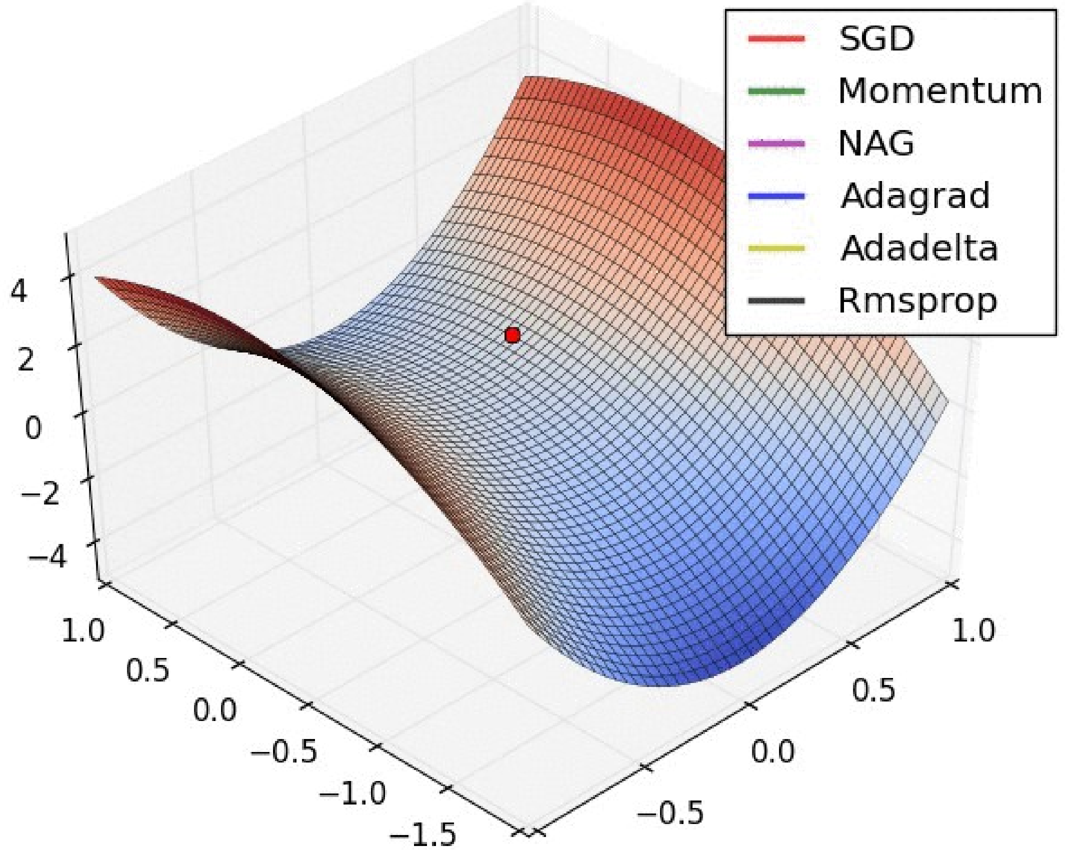
Visualizing the optimizers: Beale's Function





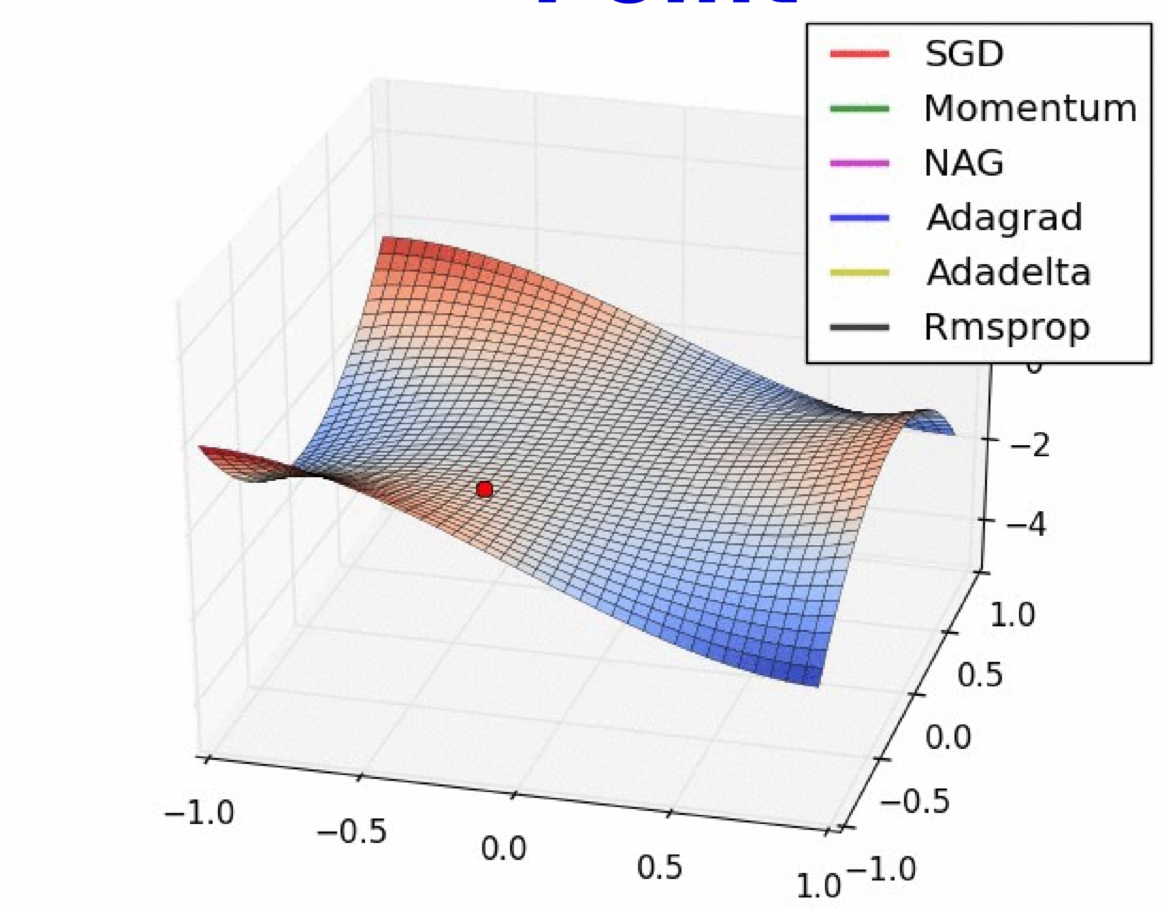
- http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

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