

Neural Networks: Optimization Part 2

Intro to Deep Learning, Fall 2017

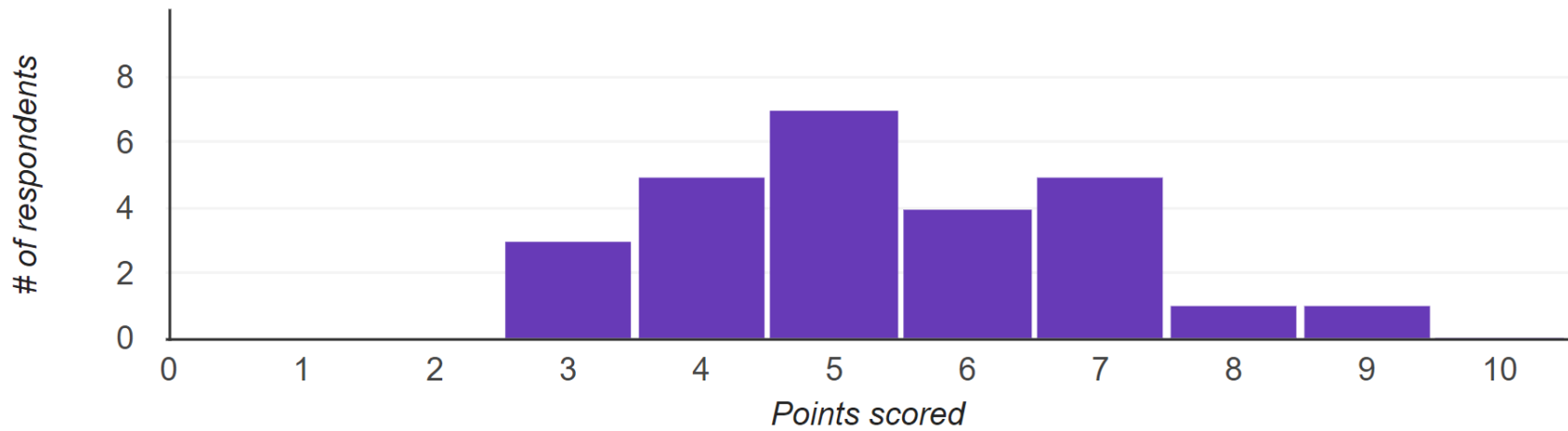
Quiz 3

Average
5.38 / 10 points

Median
5 / 10 points

Range
3 - 9 points

Total points distribution



Quiz 3

- Which of the following are necessary conditions for a value x to be a local minimum of a twice differentiable function f defined over the reals having gradient g and hessian H (select all that apply)? Comparison operators are applied elementwise in this question
 - $g(x) \geq 0$
 - eigenvalues of $H(x) \geq 0$

Quiz 3

- Select all of the properties that are true of the gradient of an arbitrary differentiable scalar function with a vector input
 - It is invariant to a scaling transformation of the input
 - It is orthogonal to the level curve of the function
 - It is the direction in which the function is increasing most quickly
 - The dot product of $\langle g(x), x \rangle$ gives the instantaneous change in function value
 - $f(y) - f(x) \geq \langle g(x), (x - y) \rangle$

Quiz 3

- T/F: In a fully connected multi-layer perceptron with a softmax as its output layer, *every* weight in the network influences *every* output in the network
- T/F: In subgradient descent, any (negated) subgradient may be used as the direction of descent

Quiz 3

- T/F: The solution that gradient descent finds is not sensitive to the initialization of the weights in the network

Quiz 3

- In class, we discussed how back propagation with a mean squared error loss function may not find a solution which separates the classes in the training data even when the classes are linearly separable. Which of the following statements are true (select all that apply)?
 - Back-propagation can get stuck in a local minimum that does not separate the data
 - The global minimum may not separate the data
 - The perceptron learning rule may also not find a solution which separates the classes
 - The back-propagation is higher variance than the perceptron algorithm
 - The back-propagation algorithm is more biased than the perceptron algorithm

Quiz 3

- T/F If the Hessian of a loss function with respect to the parameters in a network is diagonal, then QuickProp is equivalent to gradient descent with optimal step size.

Quiz 3

- Which of the following is True of the RProp algorithm?
 - It sets the step size in a given component to either a -1 or 1
 - It increases the step size for a given component when the gradient has not changed size in that component
 - When the sign of the gradient has changed in any component after a step, RProp undoes that step
 - It uses the sign of the gradient in each component to determine which parameters in the network will be adjusted during the update
 - It uses the sign of the gradient to approximate second order information about the loss surface

Quiz 3

- When gradient descent is used to minimize a non-convex function, why is a large step size (e.g. more than twice the optimal step size for a quadratic approximation) useful for escaping "bad" local minima (select all that apply)?
 - A large step size tends to make the algorithm converge to a global minimum
 - A large step size tends to make the algorithm diverge when the function value is changing very quickly
 - A large step size tends to make the algorithm converge only where a local minimum is close in function value to the global minimum
 - A large step size increases the variance of the parameter estimates

Quiz 3

- When gradient descent is used to minimize a non-convex function, why is a large step size (e.g. more than twice the optimal step size for a quadratic approximation) useful for escaping "bad" local minima (select all that apply)?
 - A large step size tends to make the algorithm converge to a global minimum
 - A large step size tends to make the algorithm diverge when the function value is changing very quickly
 - A large step size tends to make the algorithm converge only where a local minimum is close in function value to the global minimum
 - **A large step size increases the variance of the parameter estimates:**
We're accepting this answer because it's open to interpretation:
 - **Increases the variance across single *updates***
 - **Decreases the variance across *runs***
 - **Why? We are biasing towards a type of answer**

Quiz 3

- When the gradient of a twice differentiable function is normalized by the Hessian during gradient descent, this is equivalent to a reparameterization of the function which _____ (select all that apply)
 - Makes the optimal learning rate the same for every parameter
 - Makes the function parameter space less eccentric
 - Makes the function parameter space axis aligned
 - Can also be achieved by rescaling the parameters independently

Recap

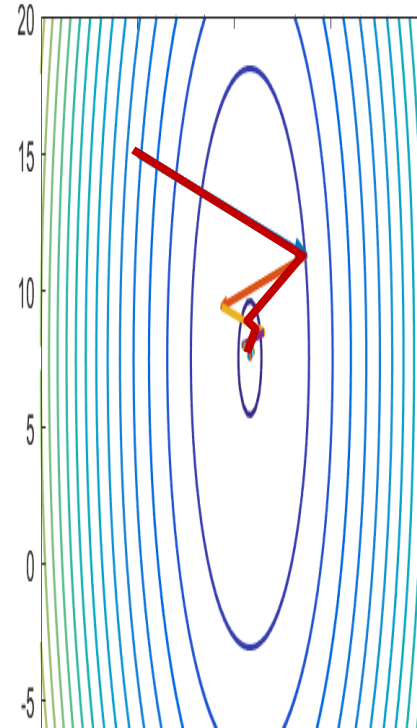
- Neural networks are universal approximators
- We must *train* them to approximate any function
- Networks are trained to minimize total “error” on a training set
 - We do so through empirical risk minimization
- We use variants of gradient descent to do so
 - Gradients are computed through backpropagation

Recap

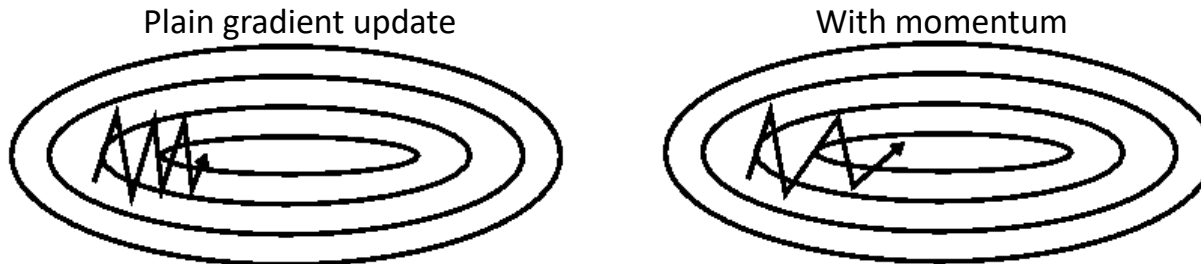
- Vanilla gradient descent may be too slow or unstable
- Better convergence can be obtained through
 - Second order methods that normalize the variation across dimensions
 - Adaptive or decaying learning rates can improve convergence
 - Methods like Rprop that decouple the dimensions can improve convergence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

The momentum methods

- Maintain a running average of all past steps
 - In directions in which the convergence is smooth, the average will have a large value
 - In directions in which the estimate swings, the positive and negative swings will cancel out in the average
- Update with the running average, rather than the current gradient



Momentum Update



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

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

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

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

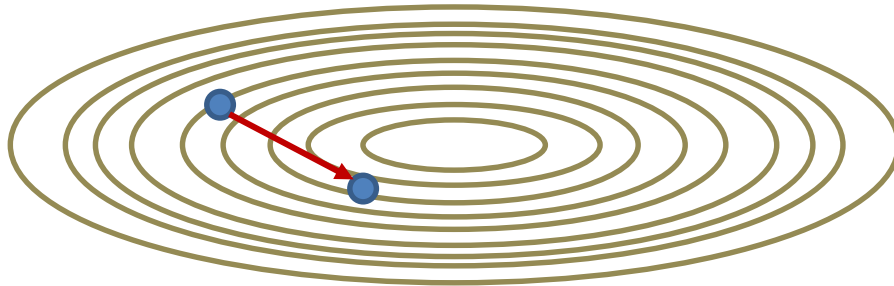
Training by gradient descent

- Initialize all weights $\mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_K$
- Do:
 - For all layers k , initialize $\nabla_{W_k} \text{Err} = 0$
 - For all $t = 1:T$
 - For every layer k :
 - Compute $\nabla_{W_k} \text{Div}(Y_t, d_t)$
 - Compute $\nabla_{W_k} \text{Err} += \frac{1}{T} \nabla_{W_k} \text{Div}(Y_t, d_t)$
 - For every layer k :
$$W_k = W_k - \eta \nabla_{W_k} \text{Err}$$
- Until Err has converged

Training with momentum

- Initialize all weights $\mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_K$
- Do:
 - For all layers k , initialize $\nabla_{W_k} \text{Err} = 0, \Delta W_k = 0$
 - For all $t = 1:T$
 - For every layer k :
 - Compute $\nabla_{W_k} \text{Div}(Y_t, d_t)$
 - Compute $\nabla_{W_k} \text{Err} += \frac{1}{T} \nabla_{W_k} \text{Div}(Y_t, d_t)$
 - For every layer k :
 - $$\Delta W_k = \beta \Delta W_k - \eta \nabla_{W_k} \text{Err}$$
$$W_k = W_k + \Delta W_k$$
- Until Err has converged

Momentum Update

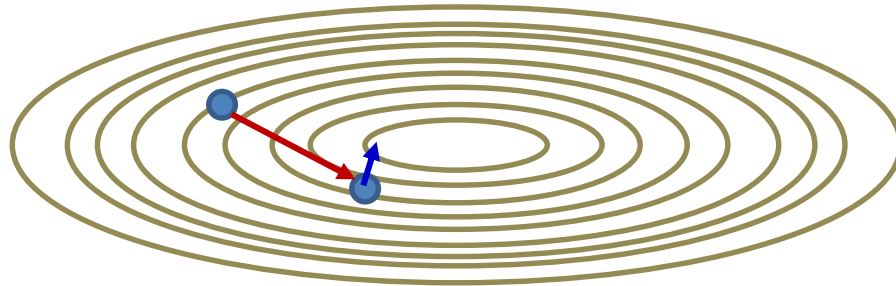


- The momentum method

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

- At any iteration, to compute the current step:

Momentum Update

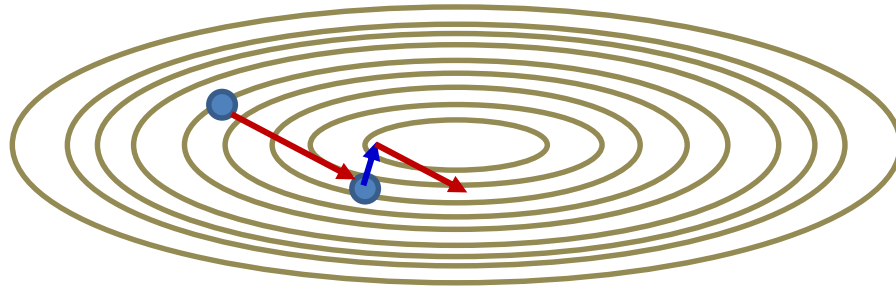


- The momentum method

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

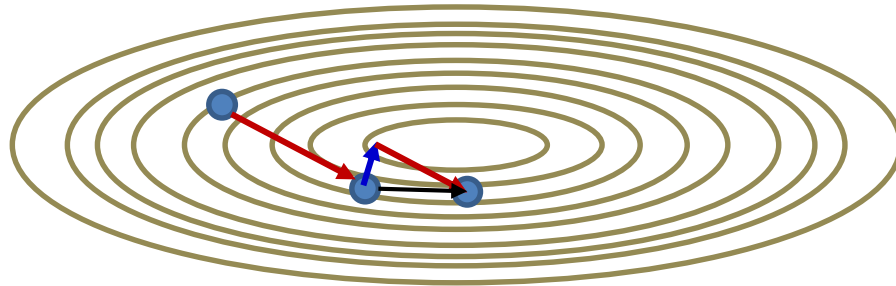
- At any iteration, to compute the current step:
 - First computes the gradient step at the current location

Momentum Update



- The momentum method
$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Err}(W^{(k-1)})$$
- At any iteration, to compute the current step:
 - First computes the gradient step at the current location
 - Then adds in the scaled *previous* step
 - Which is actually a running average

Momentum Update



- The momentum method

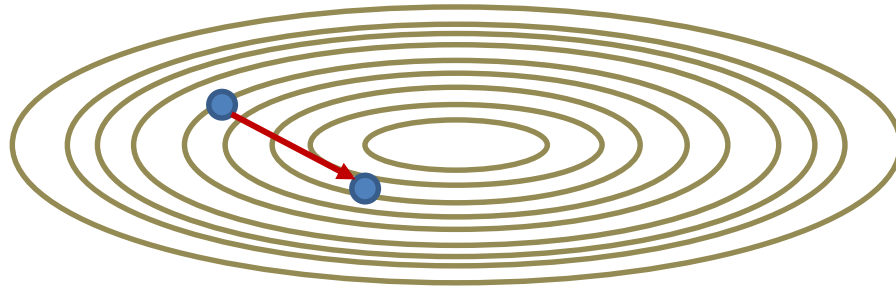
$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Err}(W^{(k-1)})$$

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

Momentum update

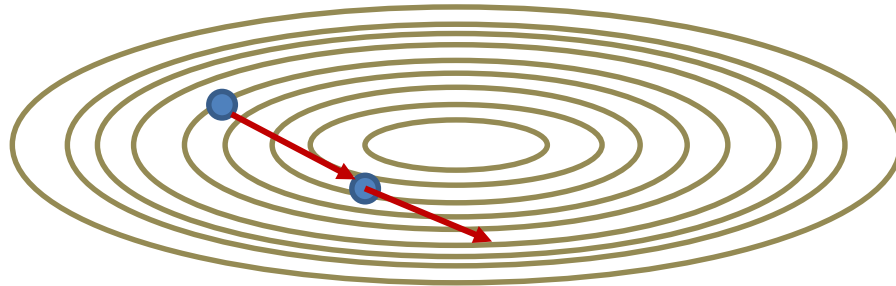
- Takes a step along the past running average *after* walking along the gradient
- The procedure can be made more optimal by reversing the order of operations..

Nestorov's Accelerated Gradient



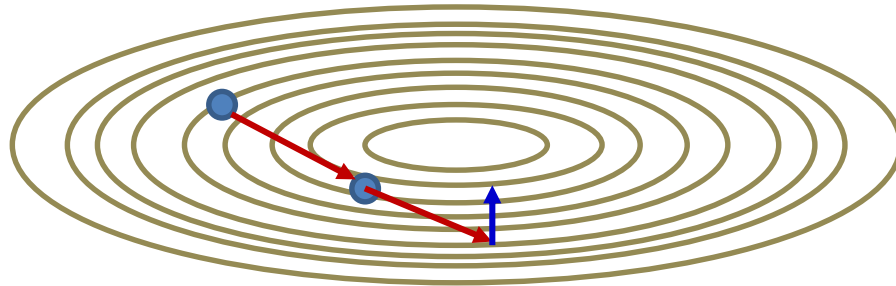
- Change the order of operations
- At any iteration, to compute the current step:

Nestorov's Accelerated Gradient



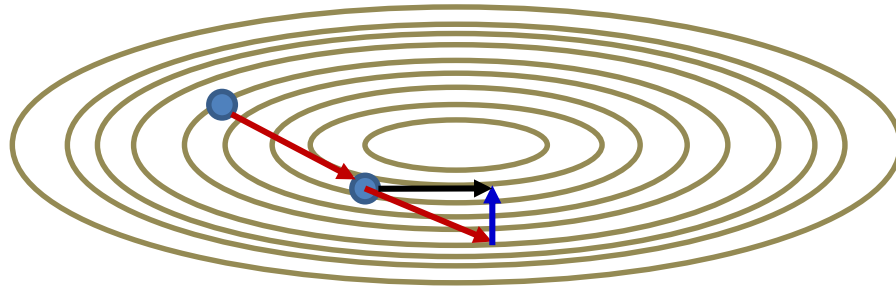
- Change the order of operations
- At any iteration, to compute the current step:
 - First extend the previous step

Nestorov's Accelerated Gradient



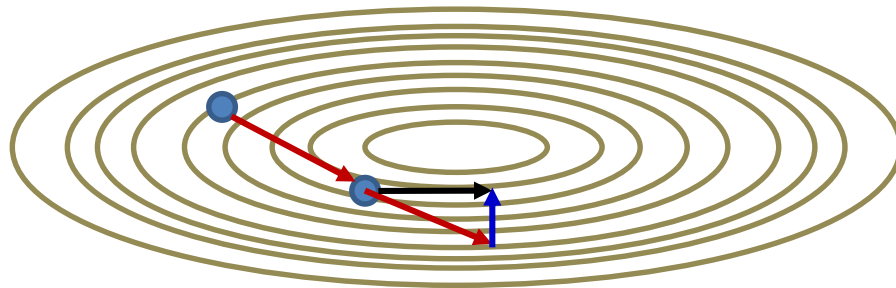
- Change the order of operations
- At any iteration, to compute the current step:
 - First extend the previous step
 - Then compute the gradient step at the resultant position

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

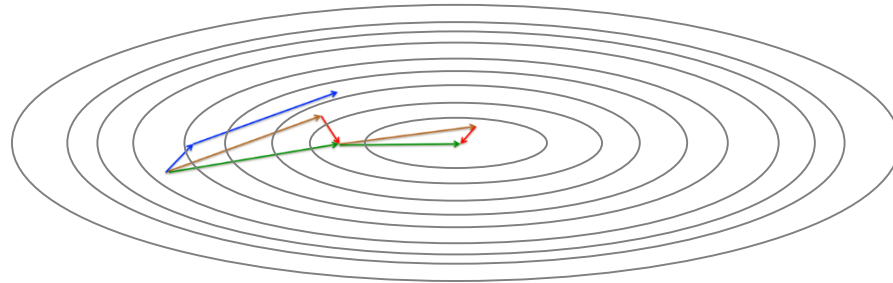


- Nestorov's method

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

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

Nestorov's Accelerated Gradient

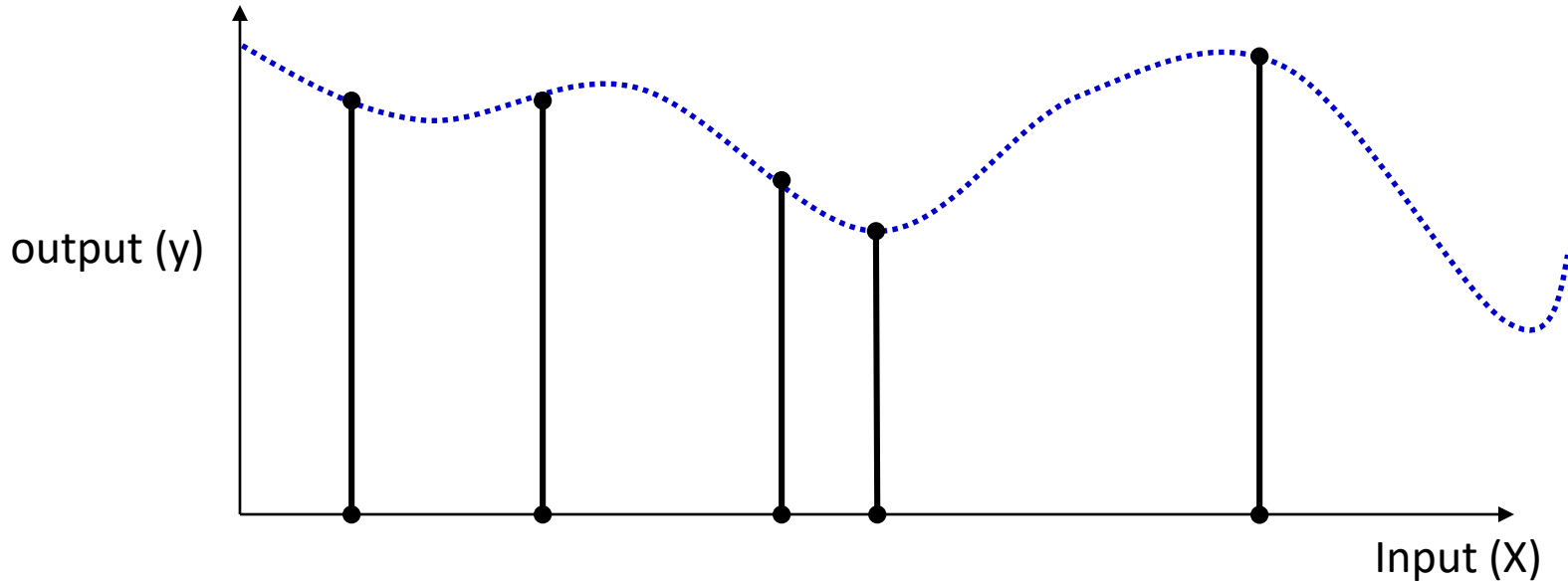


- Comparison with momentum (example from Hinton)
- Converges much faster

Moving on: Topics for the day

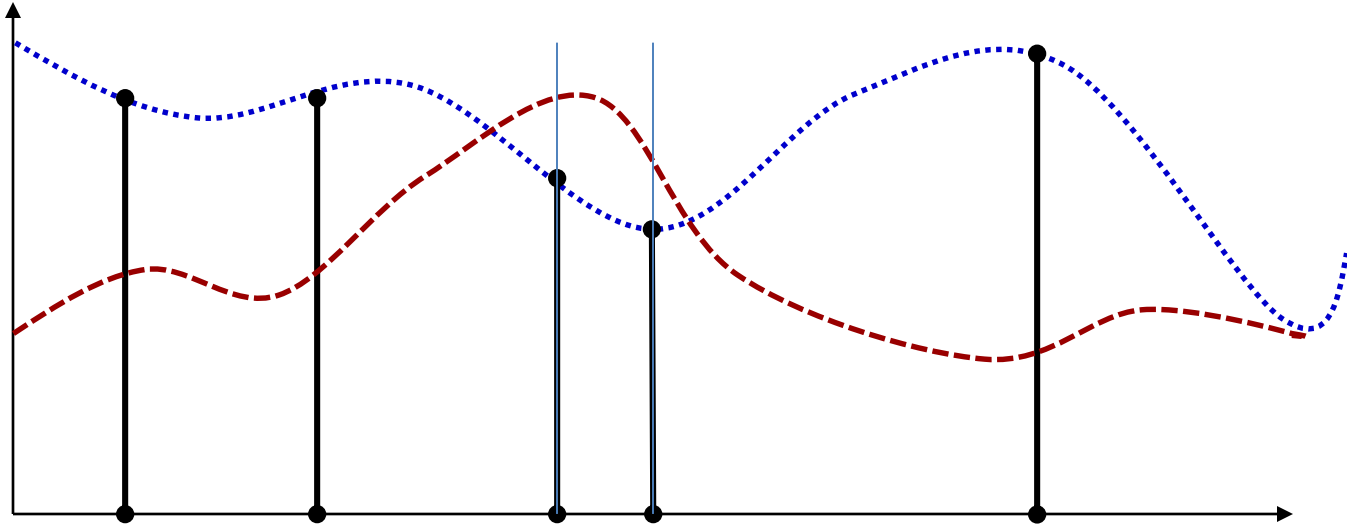
- Incremental updates
- Revisiting “trend” algorithms
- Generalization
- Tricks of the trade
 - Divergences..
 - Activations
 - Normalizations

The training formulation



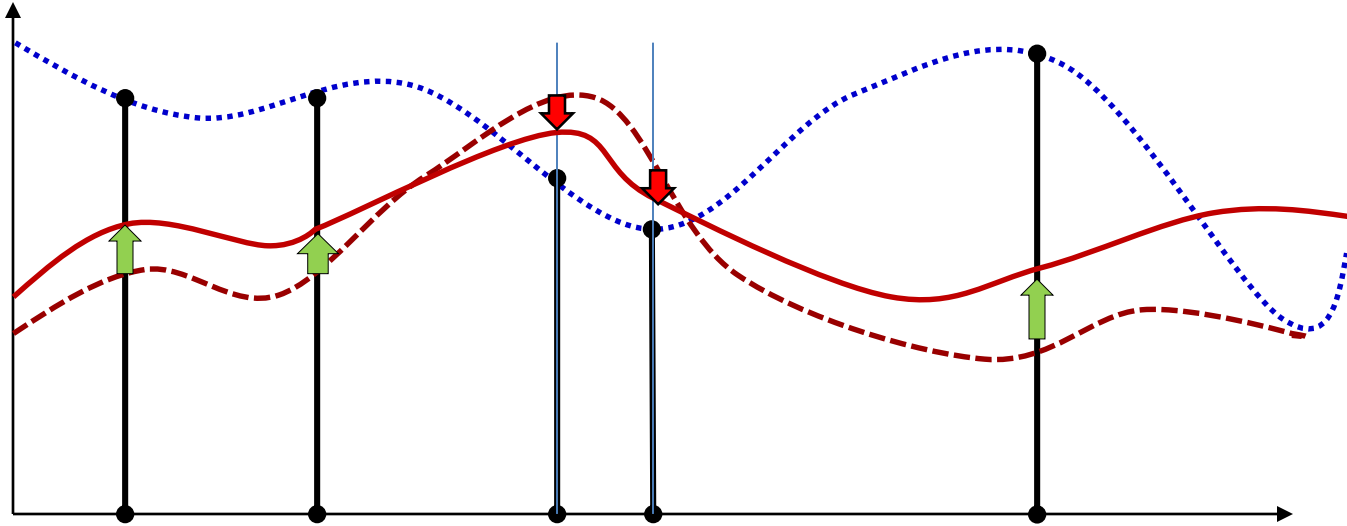
- Given input output pairs at a number of locations, estimate the entire function

Gradient descent



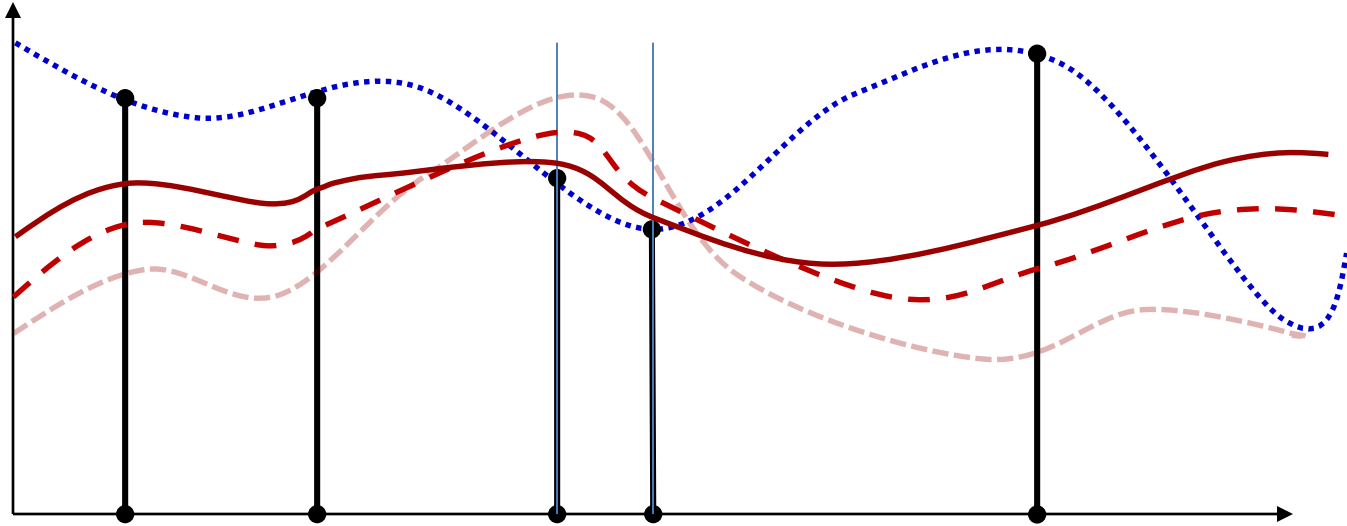
- Start with an initial function

Gradient descent



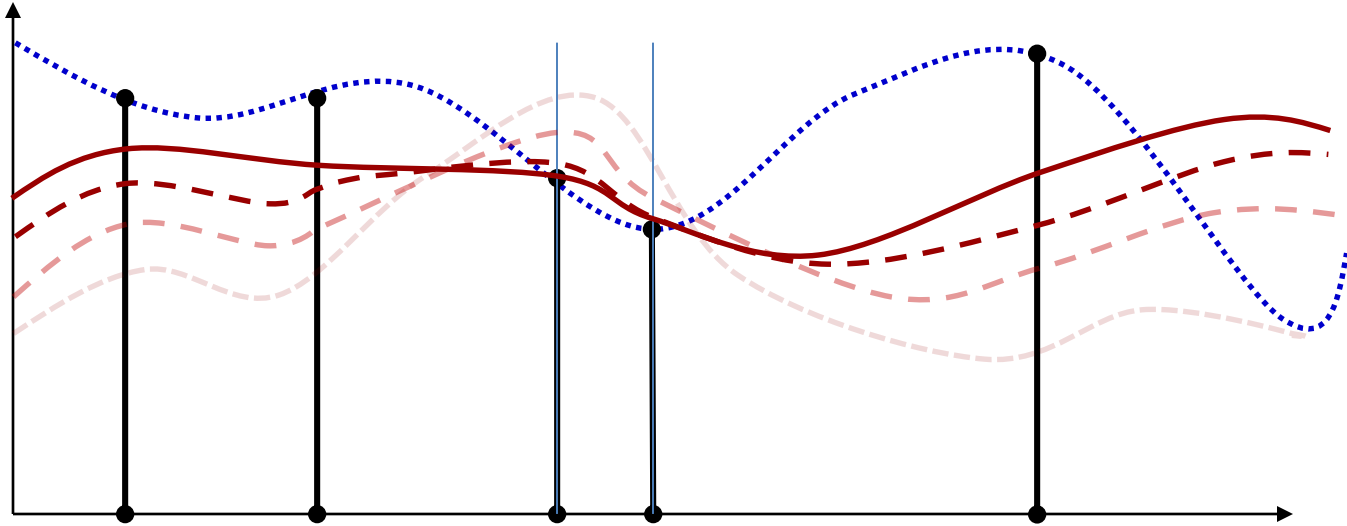
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
 - Gradient descent adjusts parameters to adjust the function value at *all* points
 - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



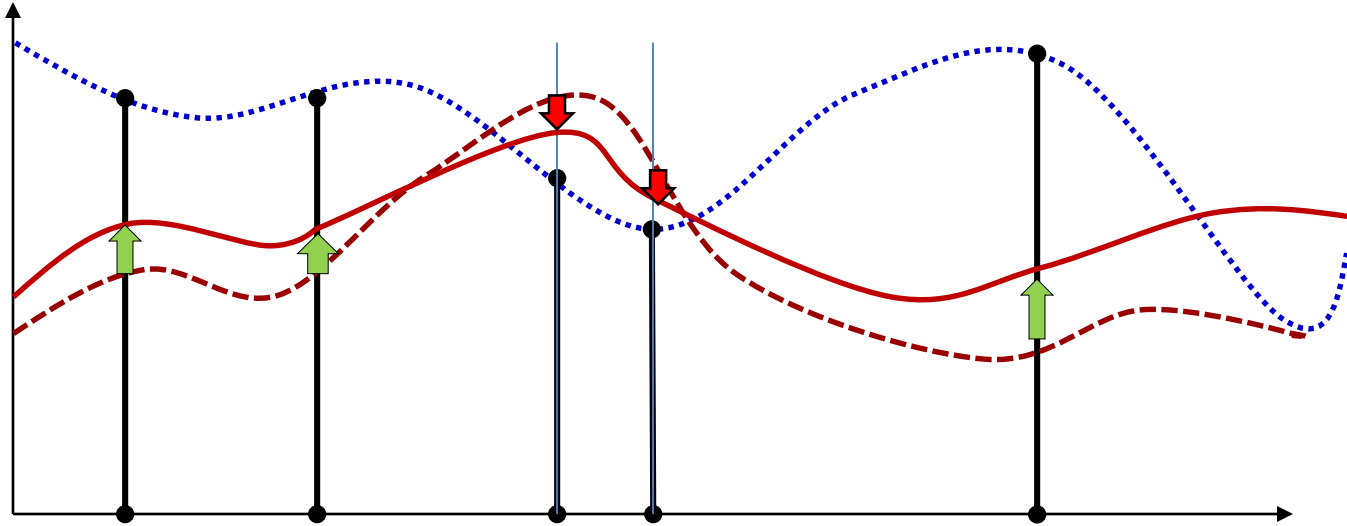
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
 - Gradient descent adjusts parameters to adjust the function value at *all* points
 - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



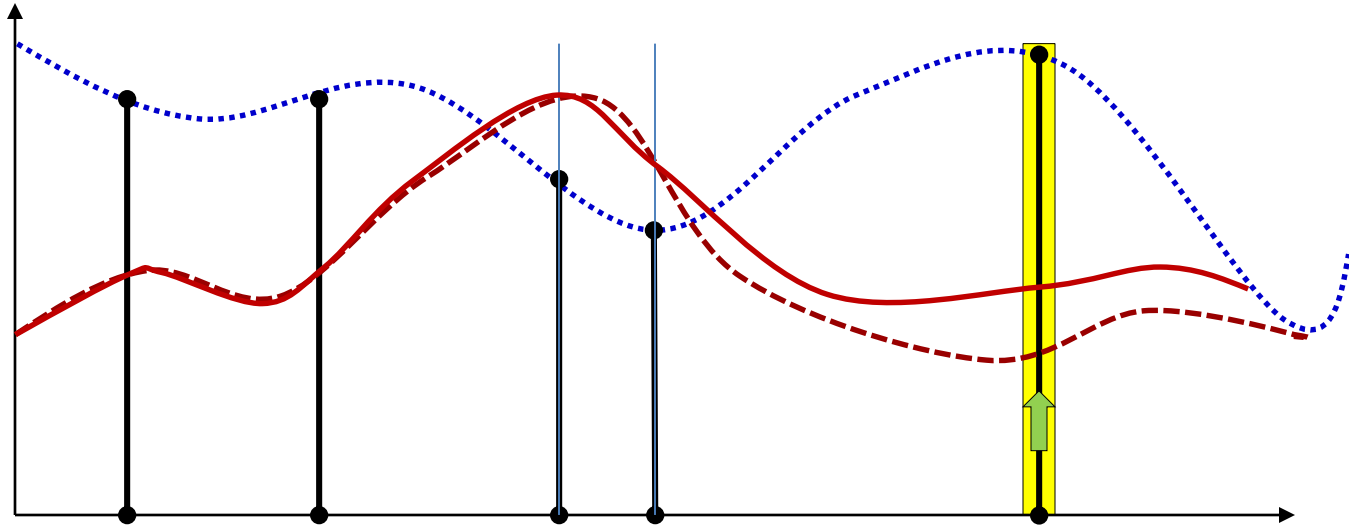
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
 - Gradient descent adjusts parameters to adjust the function value at *all* points
 - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Effect of number of samples



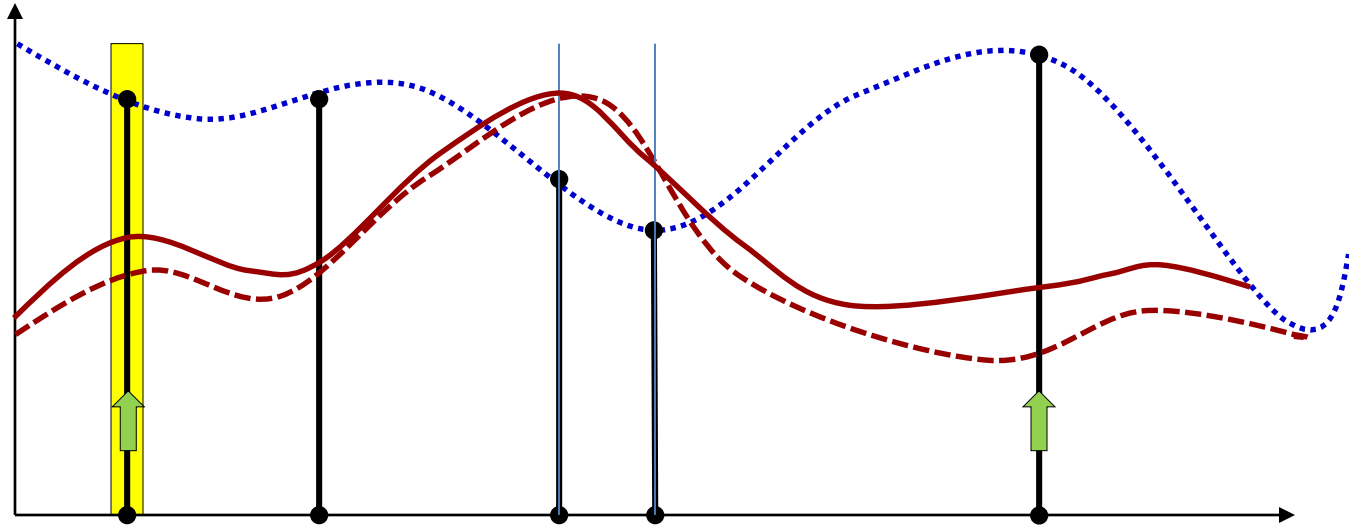
- Problem with conventional gradient descent: we try to simultaneously adjust the function at *all* training points
 - We must process *all* training points before making a single adjustment
 - “Batch” update

Alternative: Incremental update



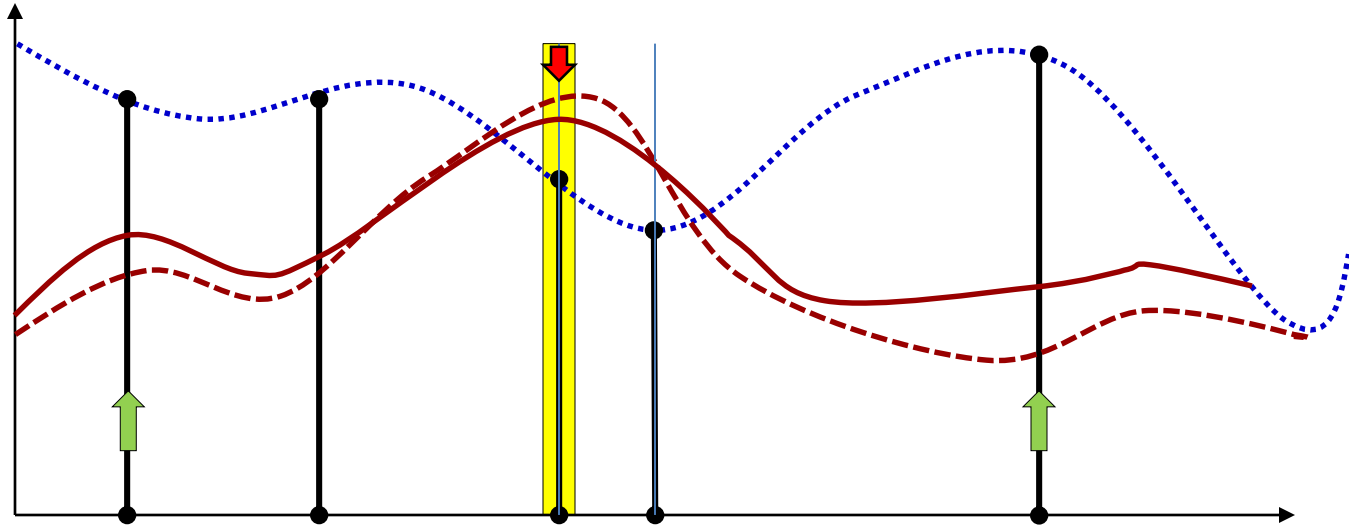
- Alternative: adjust the function at one training point at a time
 - Keep adjustments small

Alternative: Incremental update



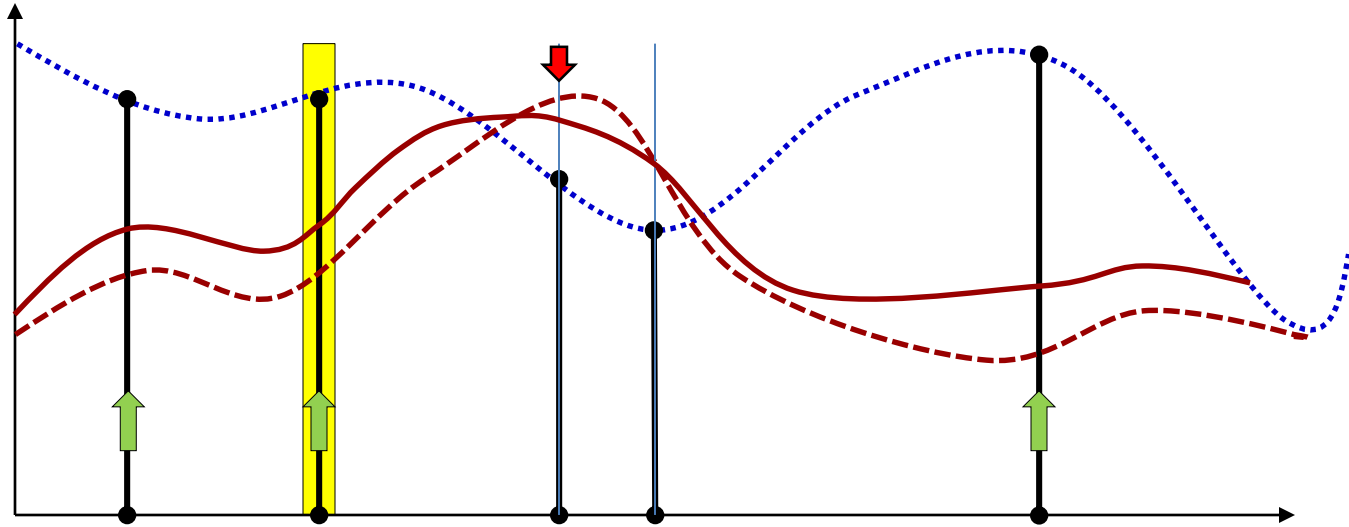
- Alternative: adjust the function at one training point at a time
 - Keep adjustments small

Alternative: Incremental update



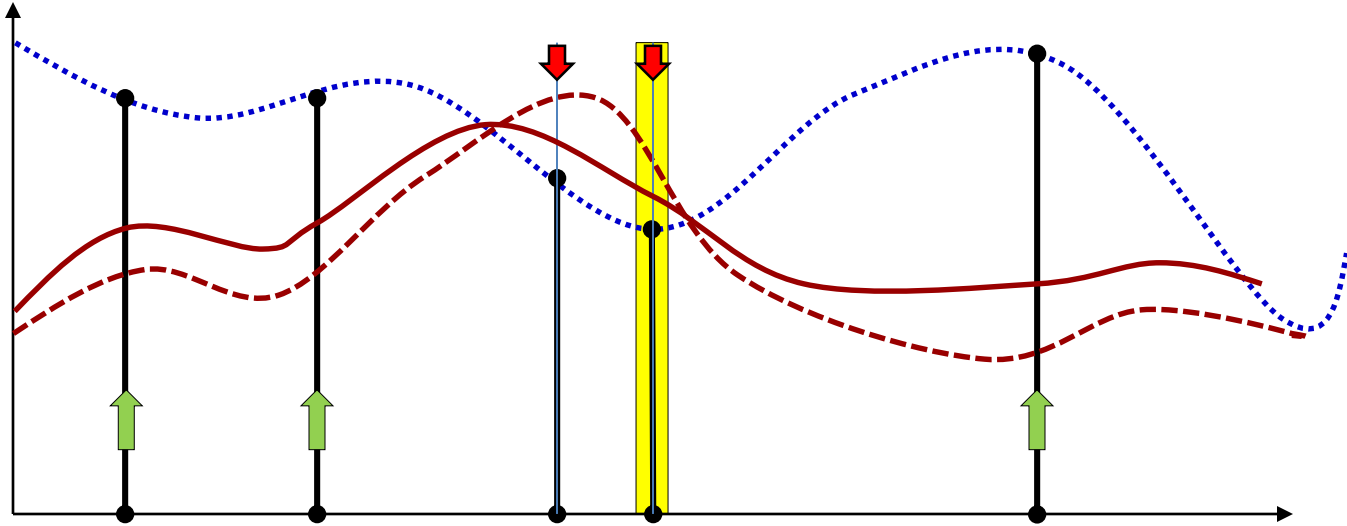
- Alternative: adjust the function at one training point at a time
 - Keep adjustments small

Alternative: Incremental update



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

Alternative: Incremental update

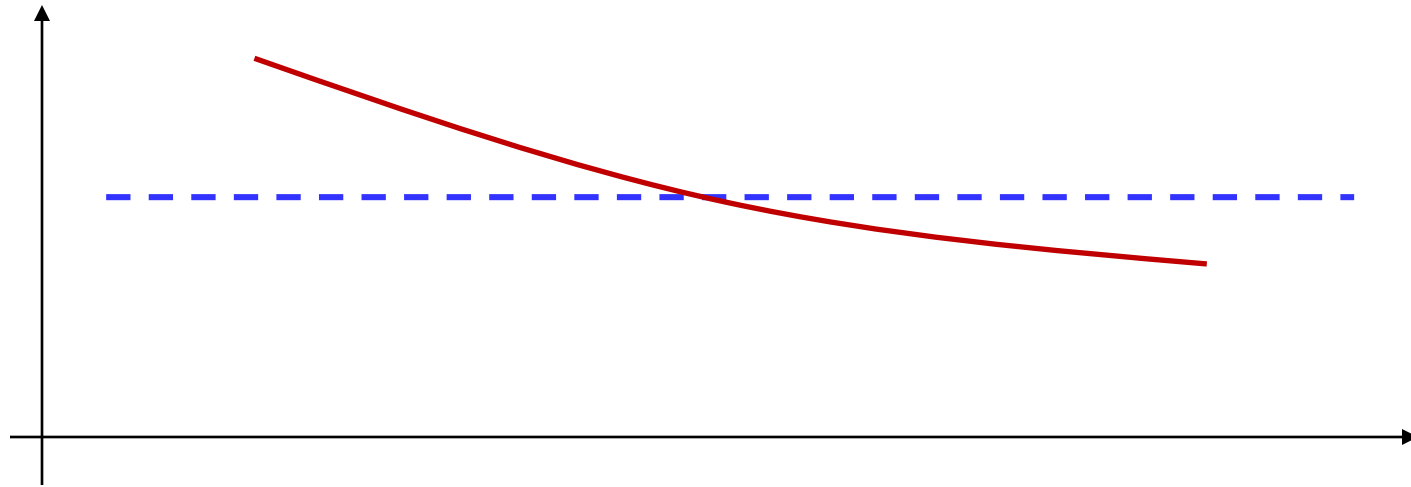


- Alternative: adjust the function at one training point at a time
 - Keep adjustments small
 - Eventually, when we have processed all the training points, we will have adjusted the entire function
 - With *greater* overall adjustment than we would if we made a single “Batch” update

Incremental Update: Stochastic Gradient Descent

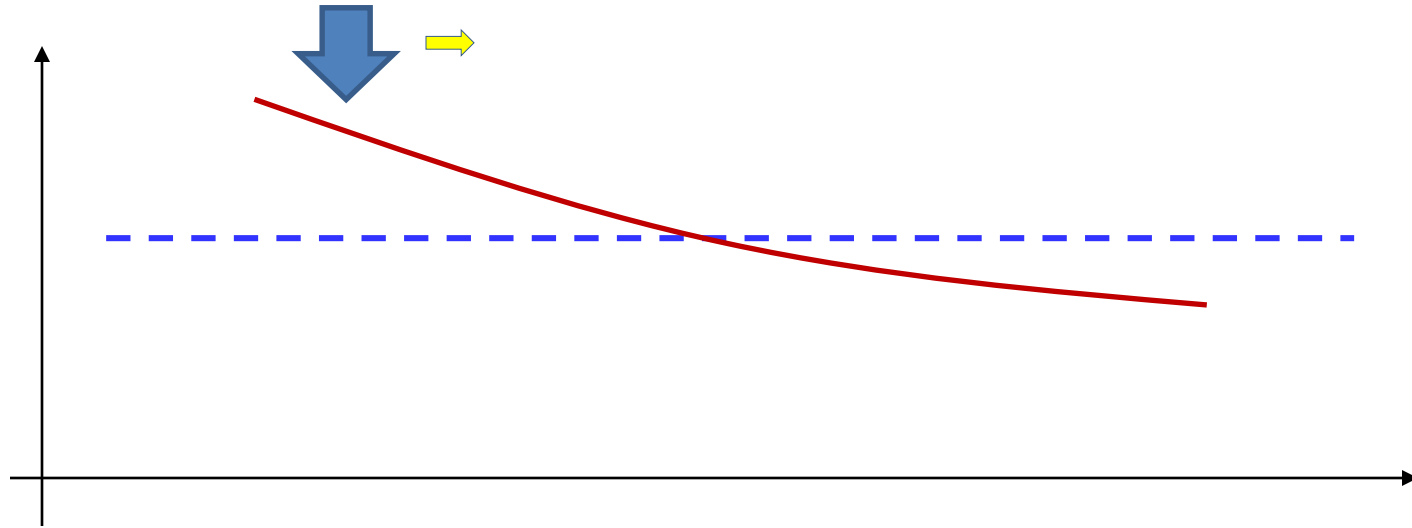
- Given $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights W_1, W_2, \dots, W_K
- Do:
 - For all $t = 1:T$
 - For every layer k :
 - Compute $\nabla_{W_k} \text{Div}(Y_t, d_t)$
 - Update
$$W_k = W_k - \eta \nabla_{W_k} \text{Div}(Y_t, d_t)$$
- Until Err has converged

Caveats: order of presentation



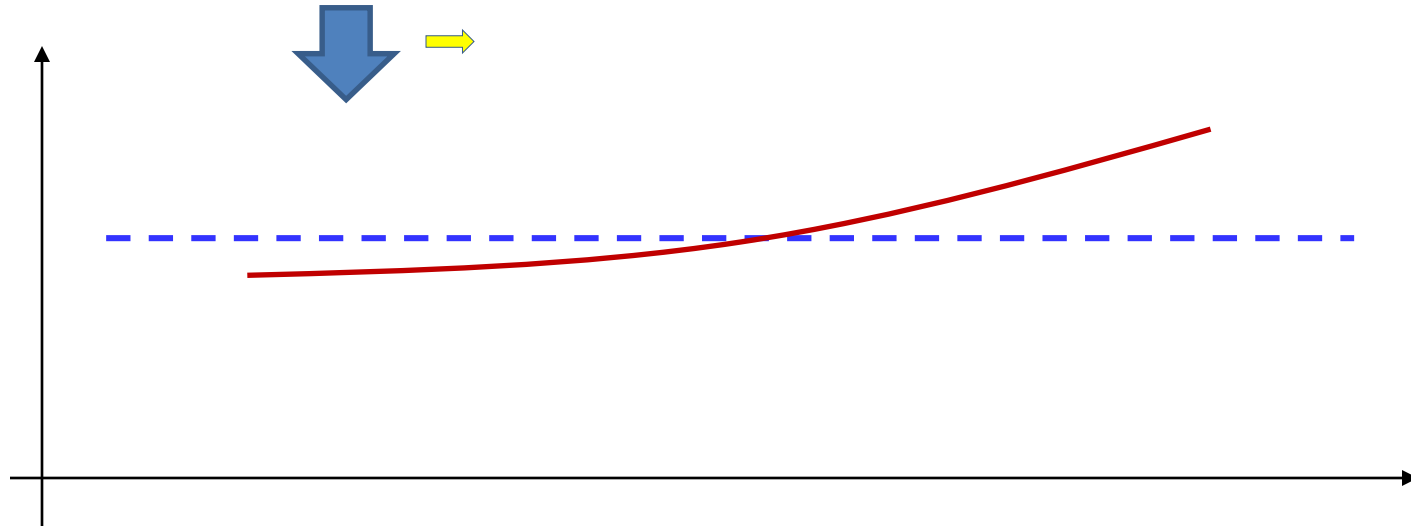
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



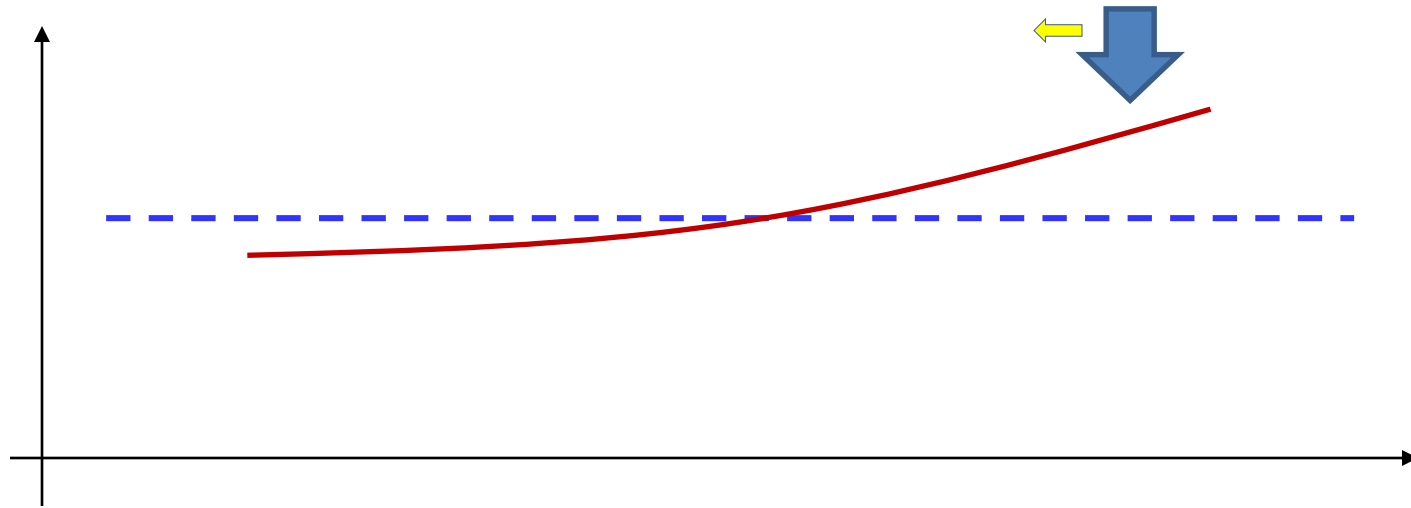
- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly*

Caveats: order of presentation



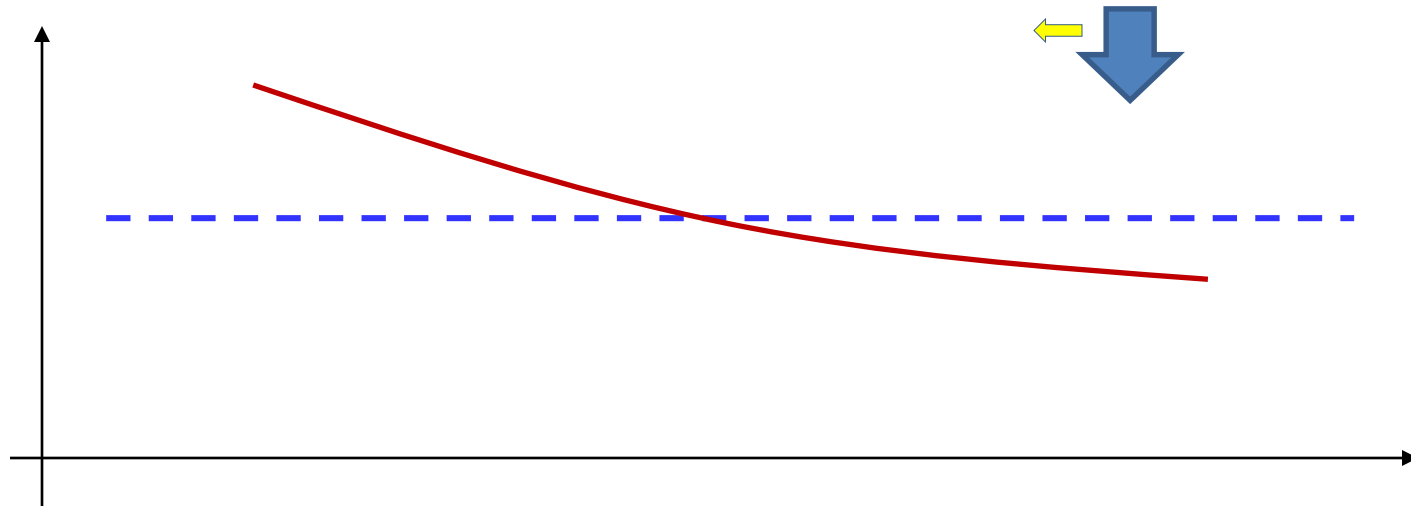
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



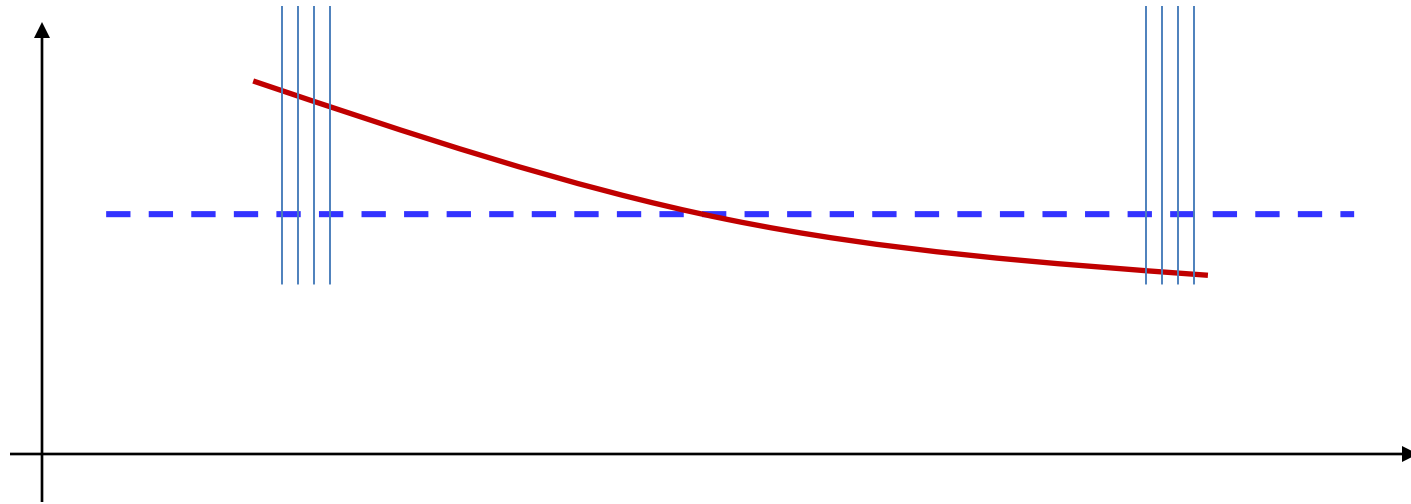
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



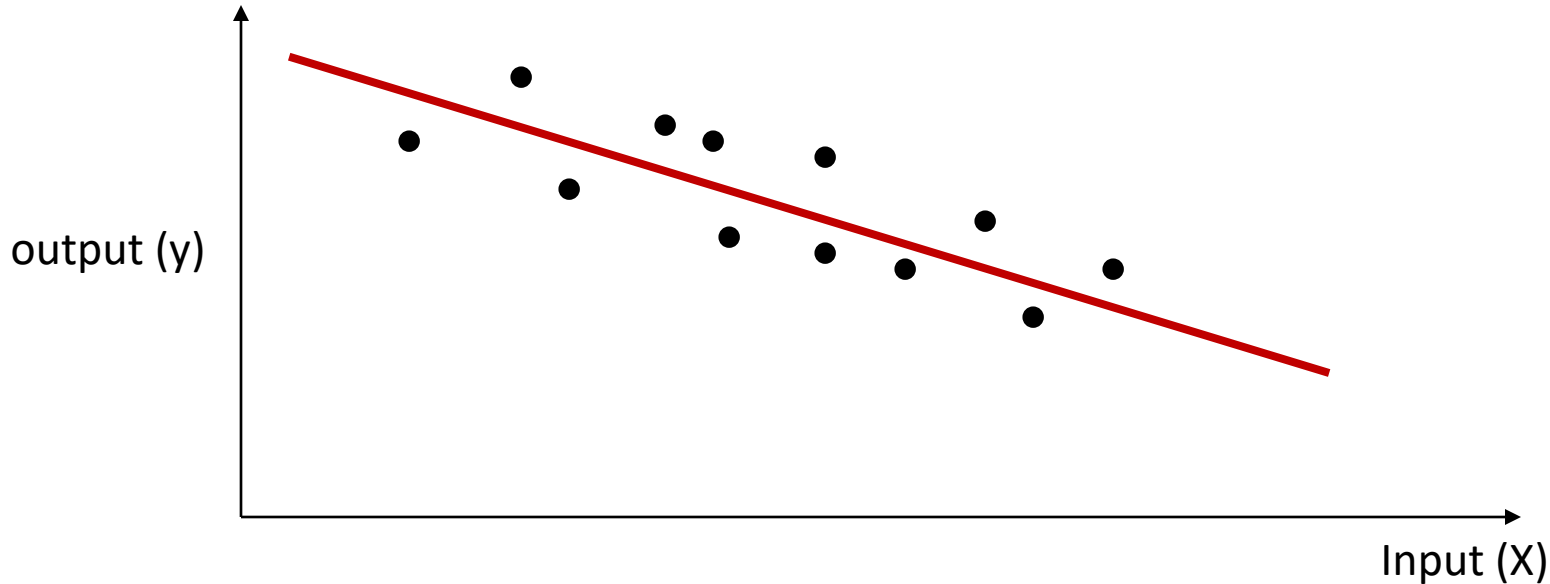
- If we loop through the samples in the same order, we may get *cyclic* behavior

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: learning rate



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

Incremental Update: Stochastic Gradient Descent

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

Incremental Update: Stochastic Gradient Descent

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

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

– For all $t = 1:T$

• $j = j + 1$

• For every layer k :

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

– Update

$$W_k = W_k - \eta_j \nabla_{W_k} \text{Div}(Y_t, d_t)$$

- Until Err has converged

Randomize input order

Learning rate reduces with j

Stochastic Gradient Descent

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

When does SGD work

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

$$\sum_k \eta_k = \infty$$

- Eventually the entire parameter space can be searched

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

- The steps shrink

- The fastest converging series that satisfies both above requirements is

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

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

Batch gradient convergence

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

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

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

SGD convergence

- We will define convergence in terms of the number of iterations taken to get within ϵ of the optimal solution
 - $|f(W^{(k)}) - f(W^*)| < \epsilon$
 - Note: $f(W)$ here is the error on the *entire* training data, although SGD itself updates after every training instance

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

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

- 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 $\frac{1}{\sqrt{k}}$.

SGD Convergence: Loss value

If:

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

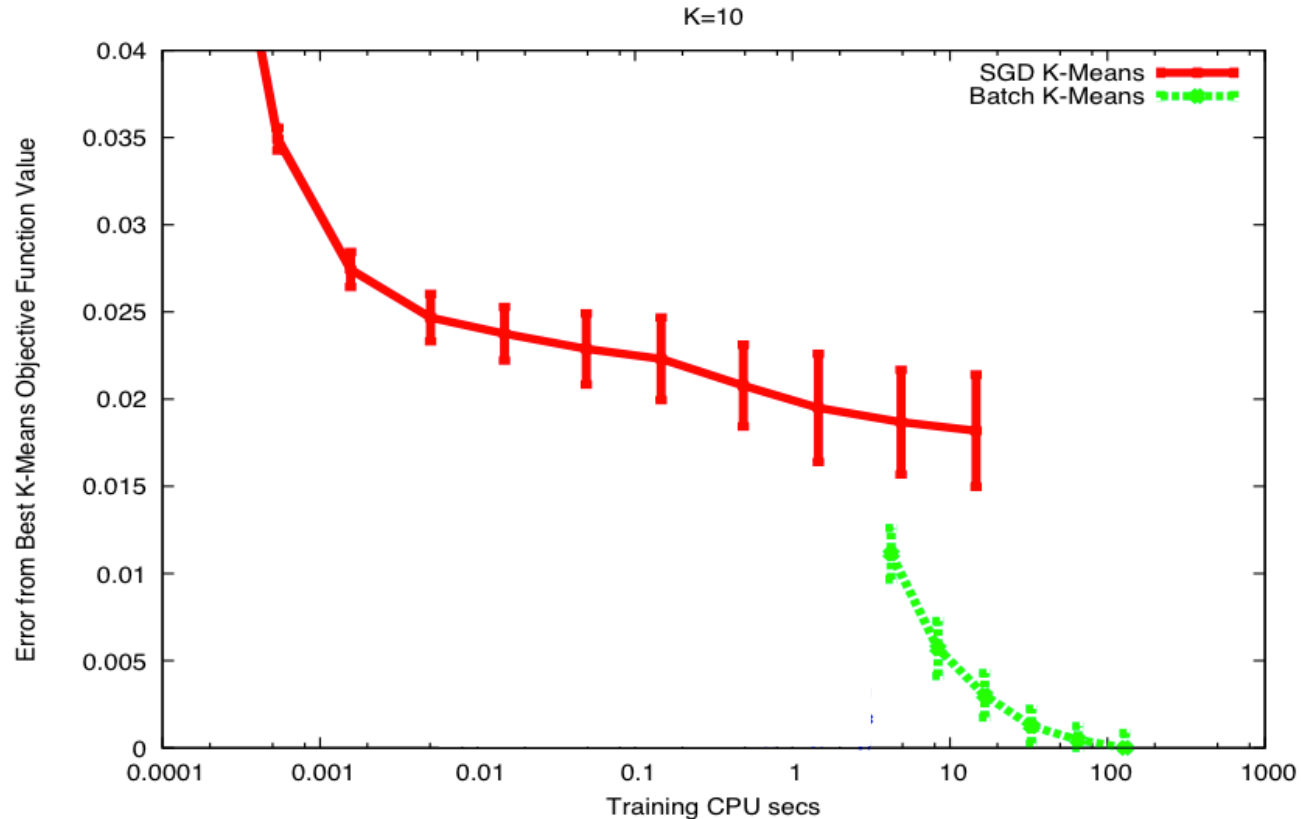
Then for any $T > 1$:

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

SGD Convergence

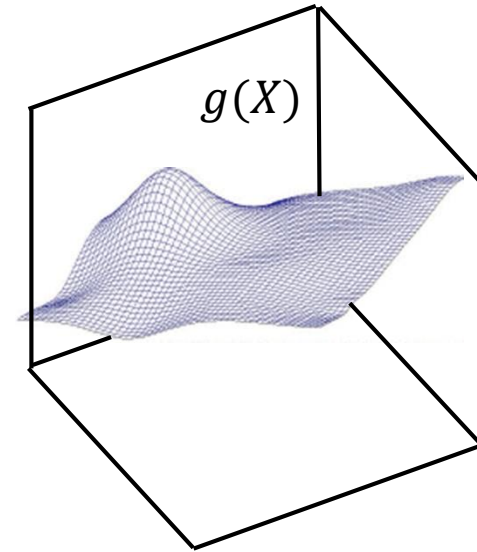
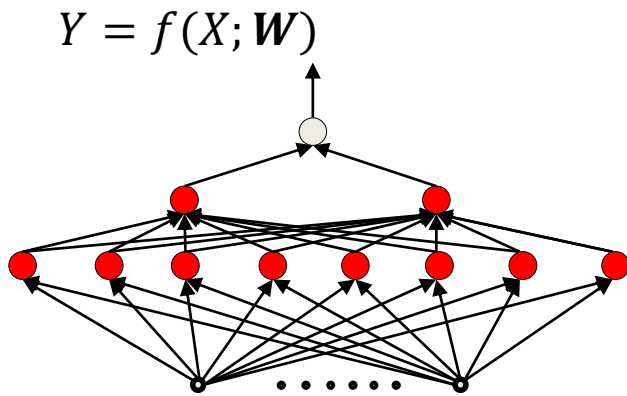
- We can bound the expected difference between the loss over our data using the optimal weights, w^* , and the weights at **any single iteration**, w_T , to $\mathcal{O}\left(\frac{\log(T)}{T}\right)$ for strongly convex loss or $\mathcal{O}\left(\frac{\log(T)}{\sqrt{T}}\right)$ for convex loss
- Averaging 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 example



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

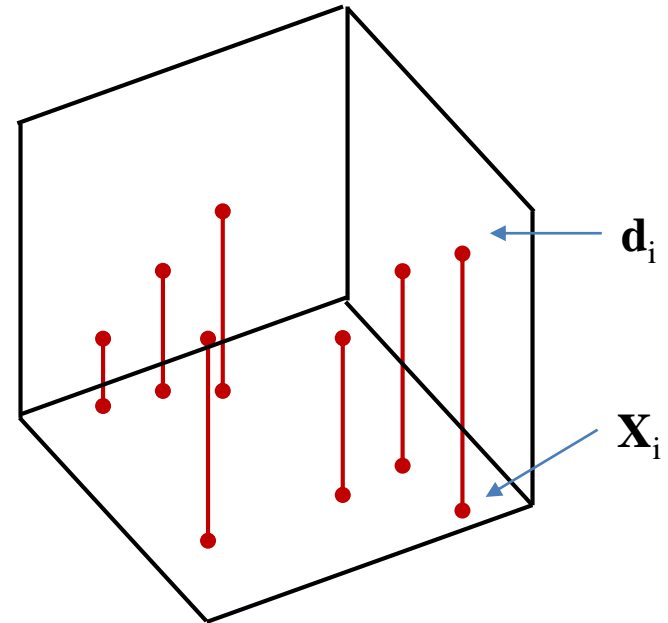
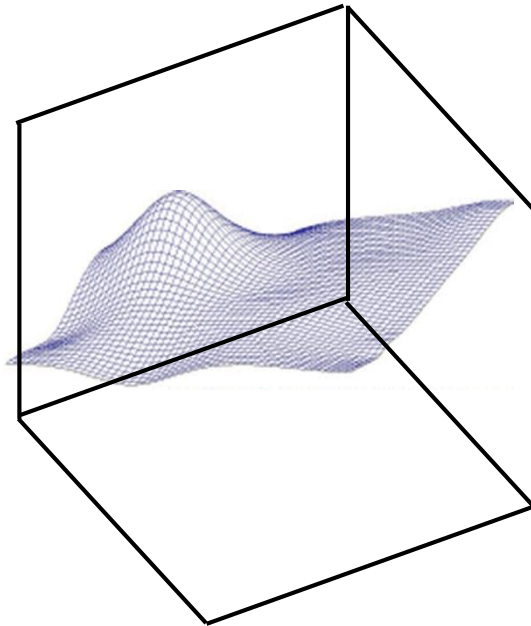
Recall: Modelling a function



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

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

Recall: The *Empirical* risk



- In practice, we minimize the *empirical error*

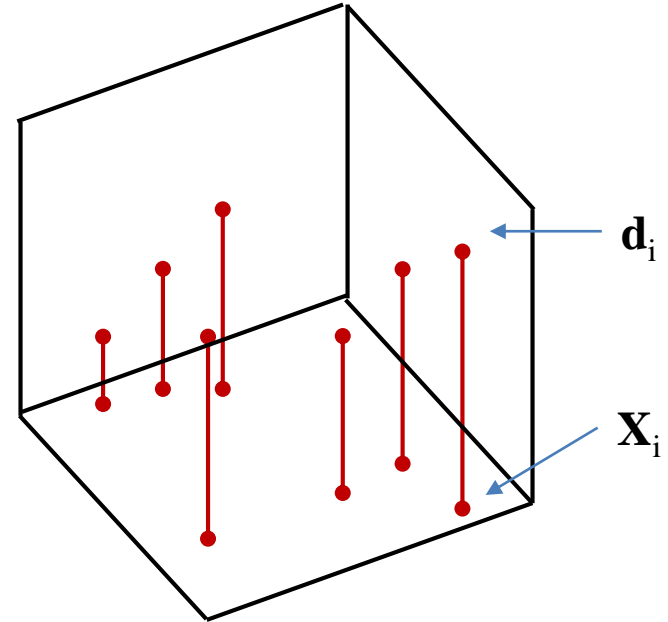
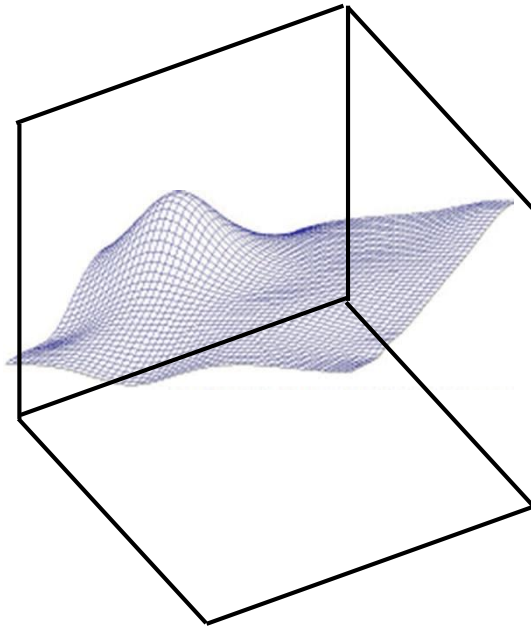
$$Err(f(X; W), g(X)) = \frac{1}{N} \sum_{i=1}^N div(f(X_i; W), d_i)$$

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

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

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

Recap: The *Empirical* risk



- In practice, we minimize the *empirical error*

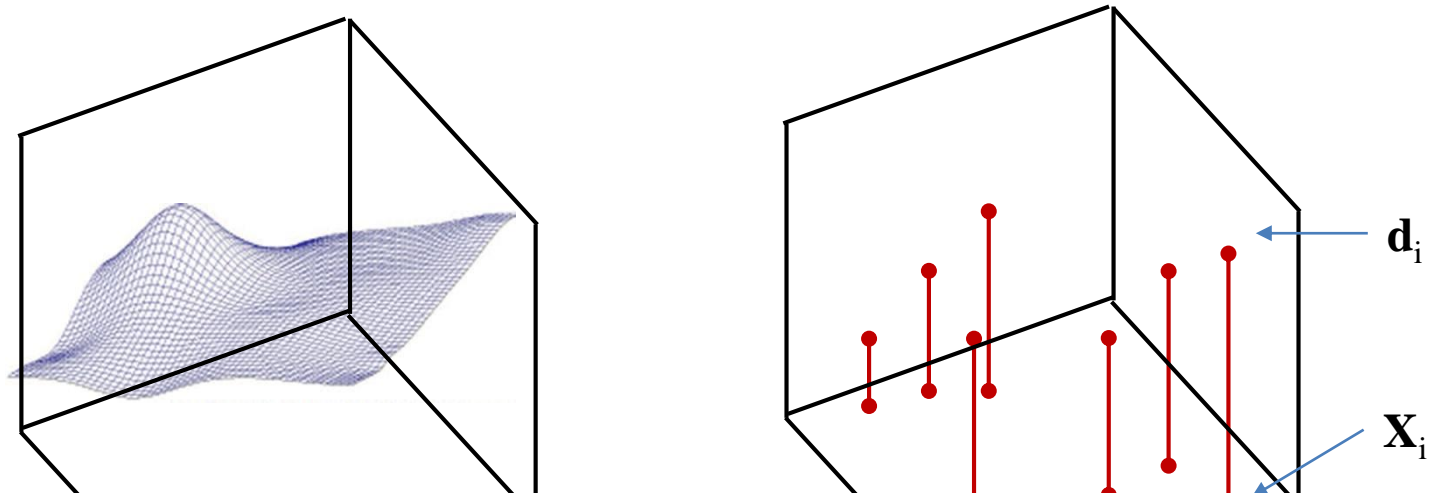
$$Err(f(X; W), g(X)) = \frac{1}{N} \sum_{i=1}^N div(f(X_i; W), d_i)$$

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

The expected value of the empirical error is actually the expected error

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

Recap: The *Empirical* risk



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

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

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

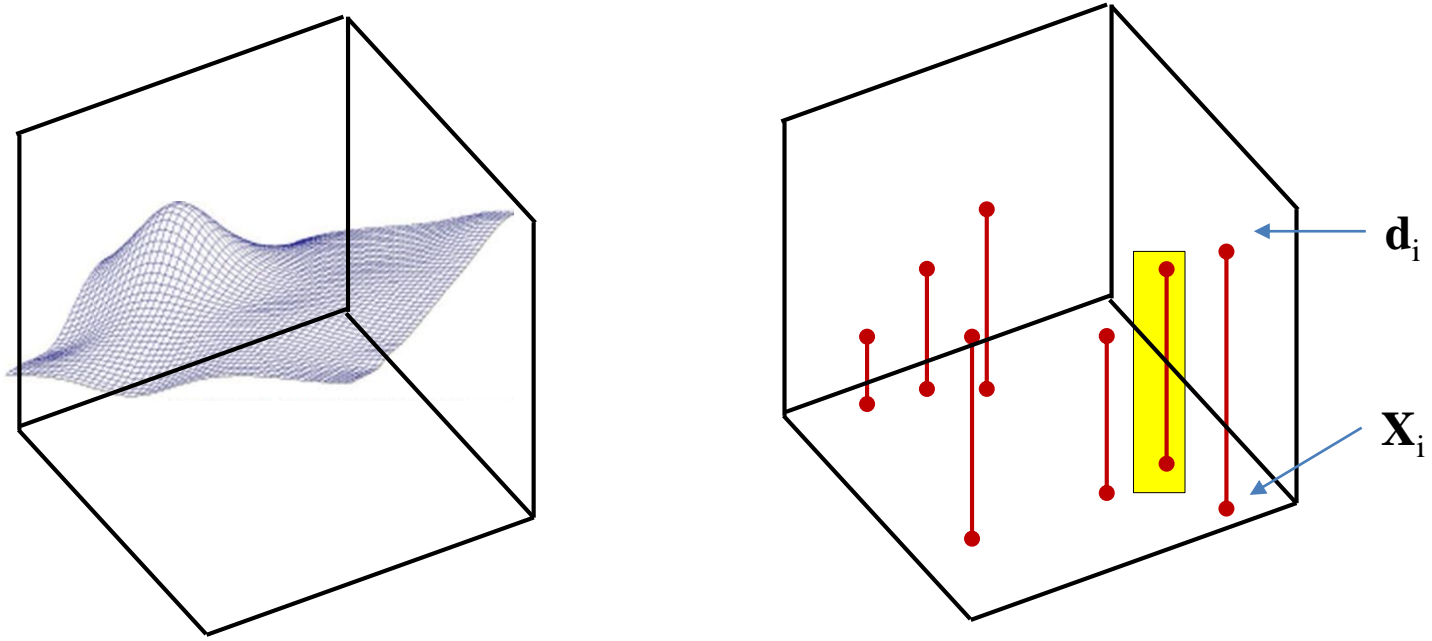
The empirical error is an *unbiased* estimate of the expected error

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

- The *expected* value of the *empirical* error is actually the *expected* error

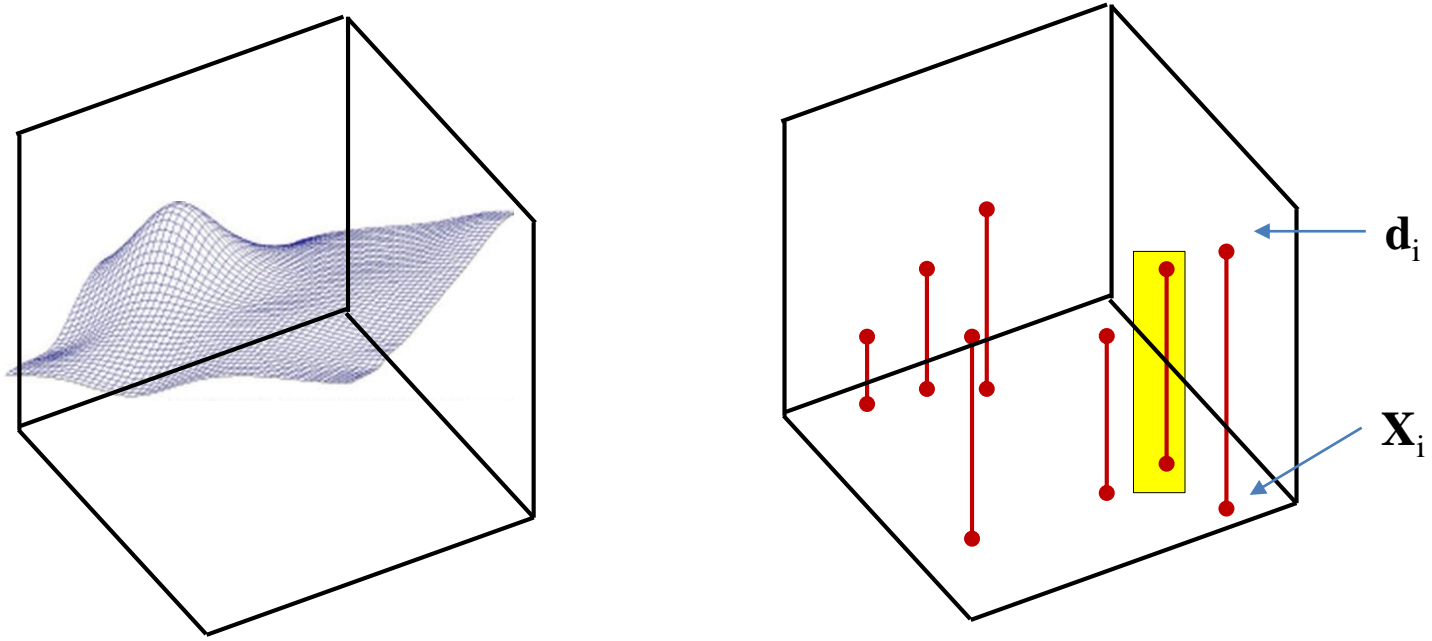
$$E[\text{Err}(f(X; W), g(X))] = E[\text{div}(f(X; W), g(X))]$$

SGD



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

SGD

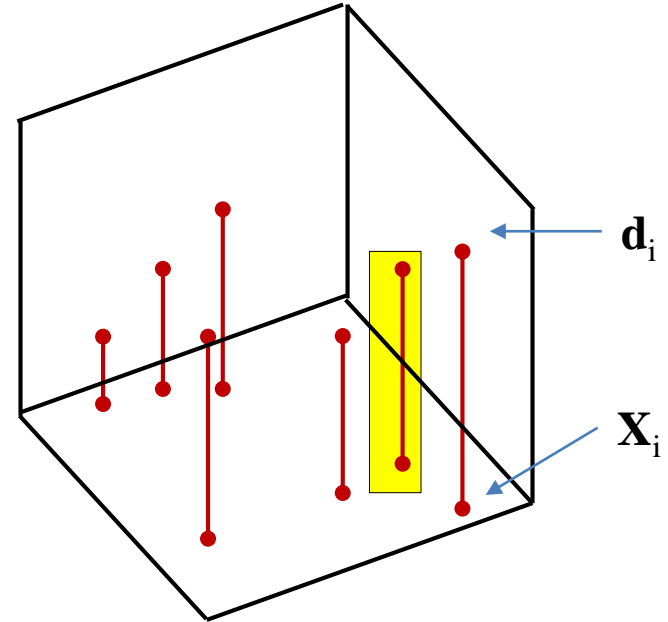
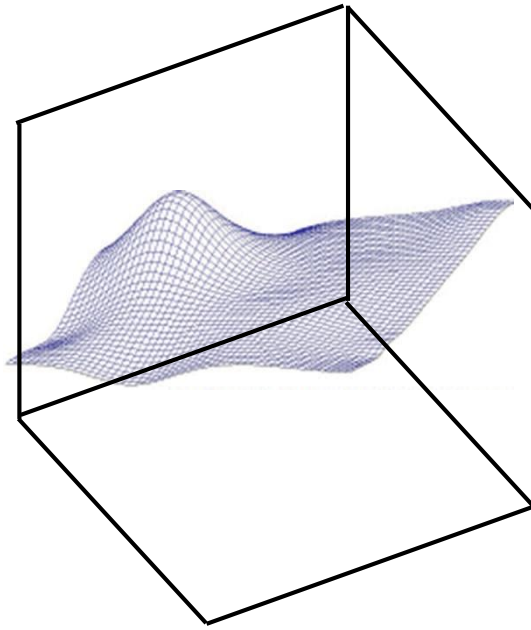


- At each iteration, **SGD** focuses on the divergence

$f(x; W)$ and $g(x)$. The sample error is also an *unbiased* estimate of the expected error

- The *expected value* of the *sample error* is **still** the *expected divergence* $E[\text{div}(f(X; W), g(X))]$

SGD

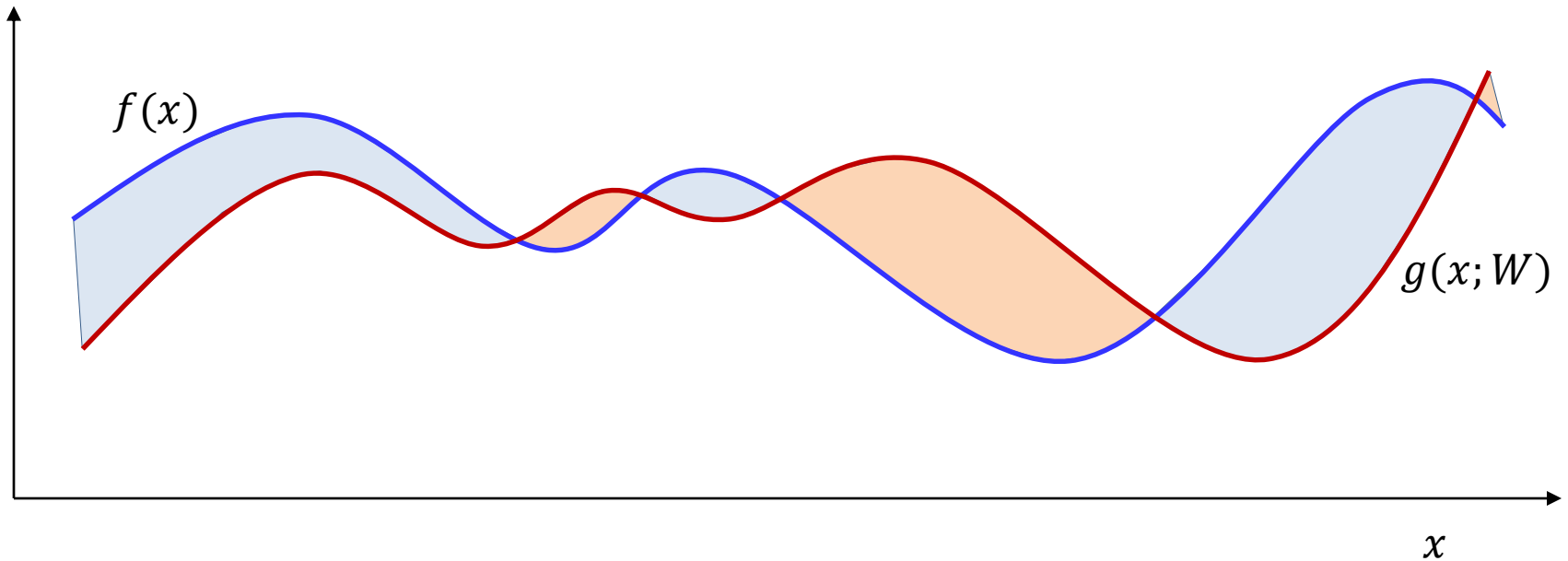


The variance of the sample error is the variance of the divergence itself: $\text{var}(\text{div})$
This is N times the variance of the empirical average minimized by batch update

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

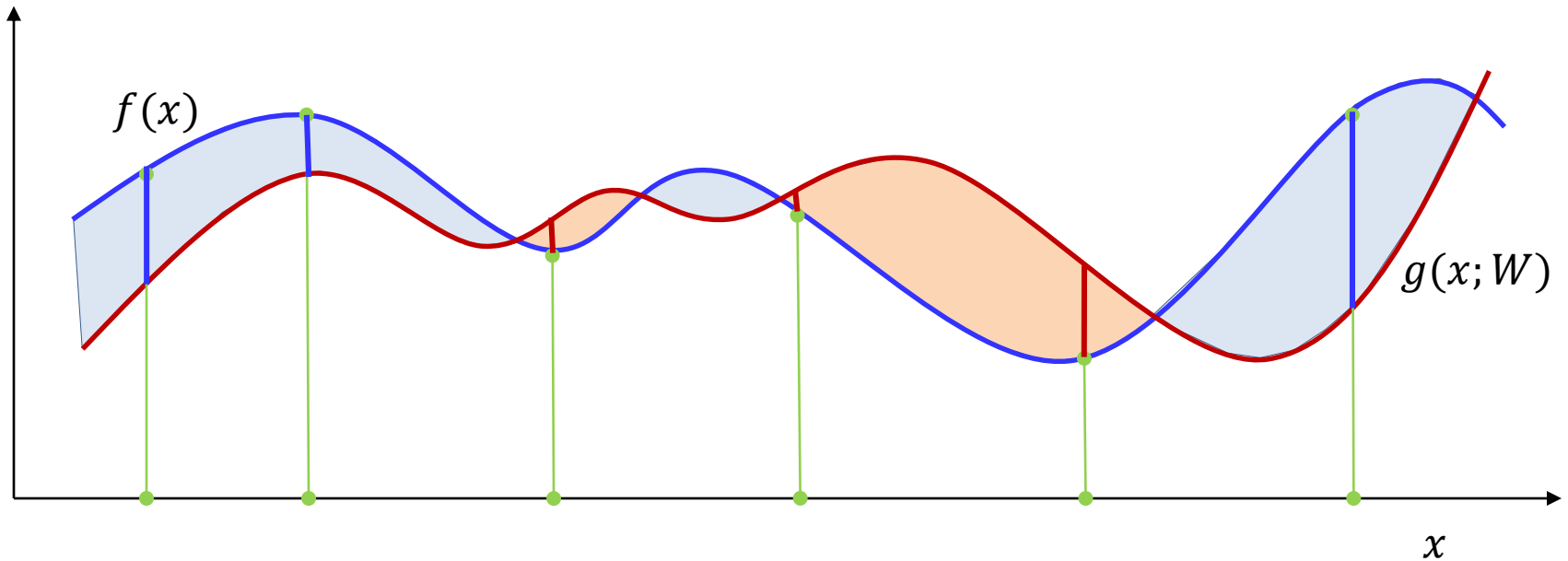
- The *expected value* of the *sample error* is **still** the *expected divergence* $E[\text{div}(f(X; W), g(X))]$

Explaining the variance



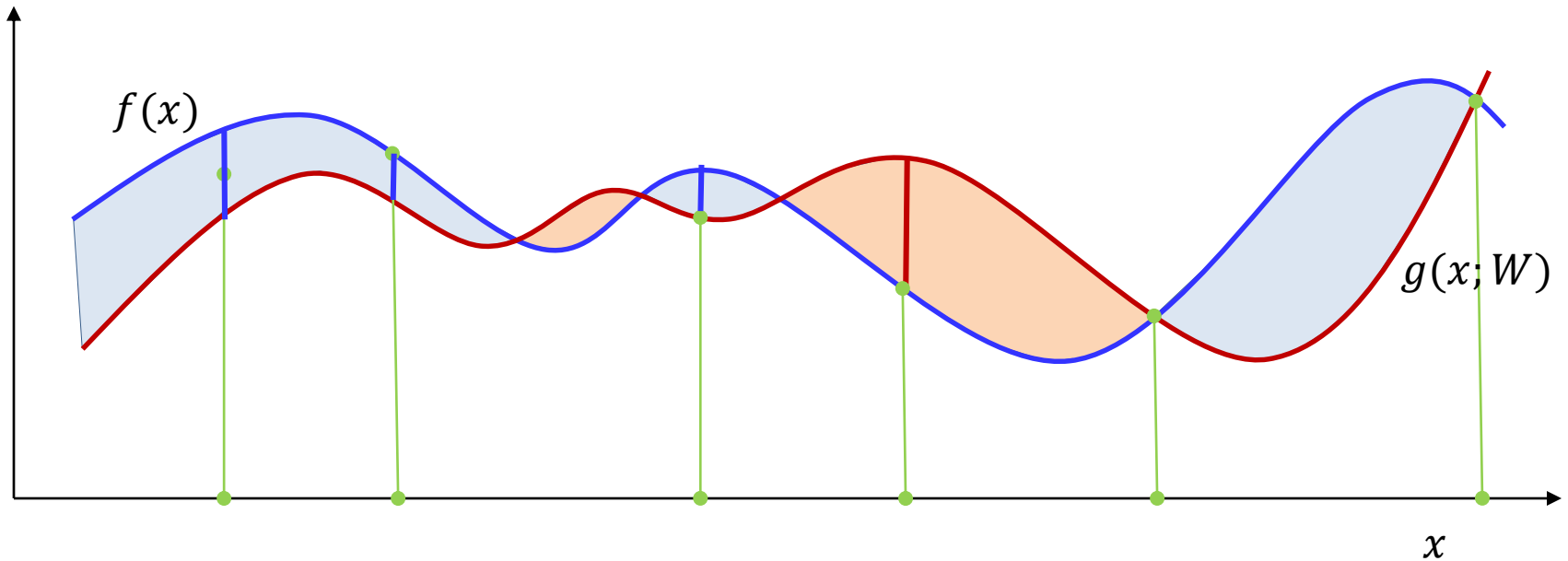
- The blue curve is the function being approximated
- The red curve is the approximation by the model at a given W
- The heights of the shaded regions represent the point-by-point error
 - The divergence is a function of the error
 - We want to find the W that minimizes the average divergence

Explaining the variance



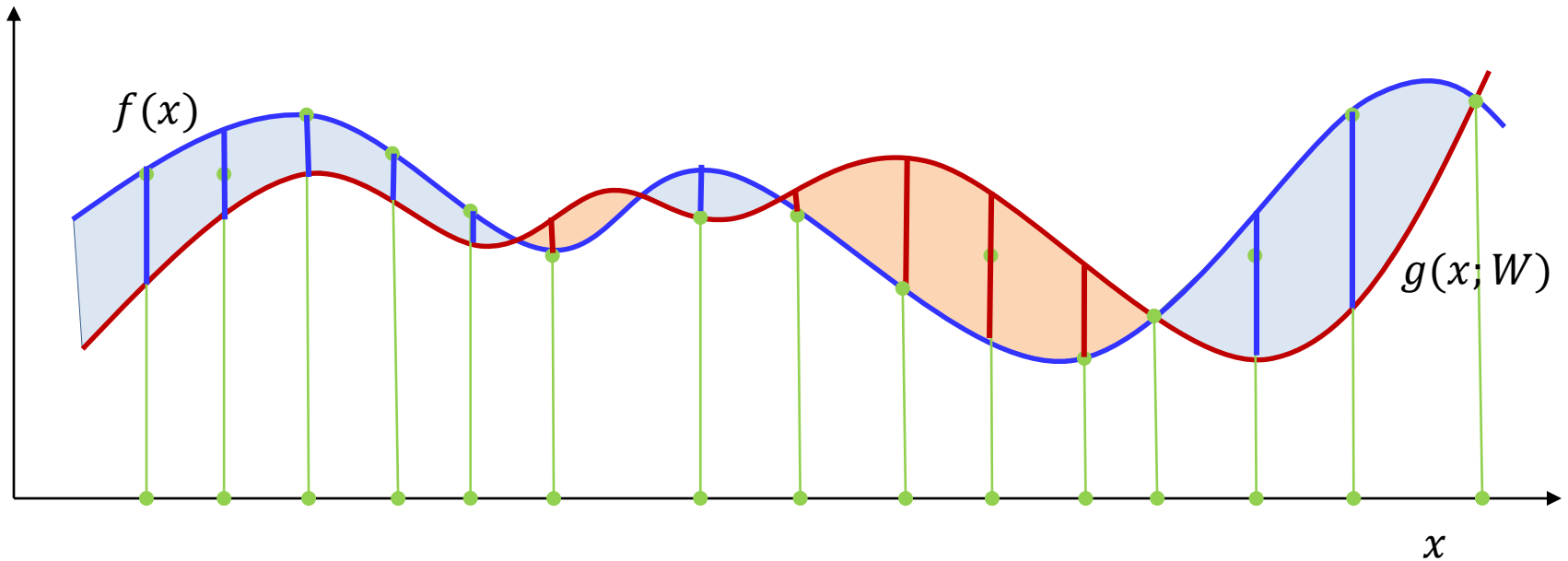
- Sample estimate approximates the shaded area with the average length of the lines

Explaining the variance



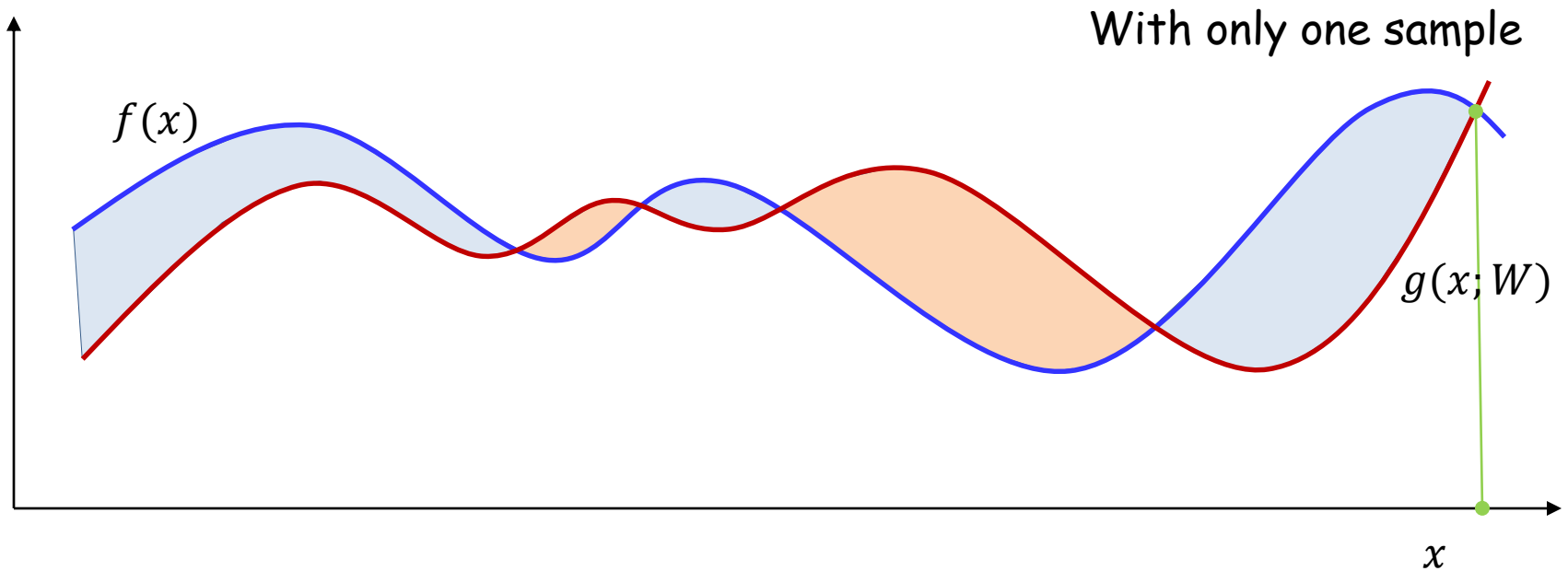
- Sample estimate approximates the shaded area with the average length of the lines
- This average length will change with position of the samples

Explaining the variance



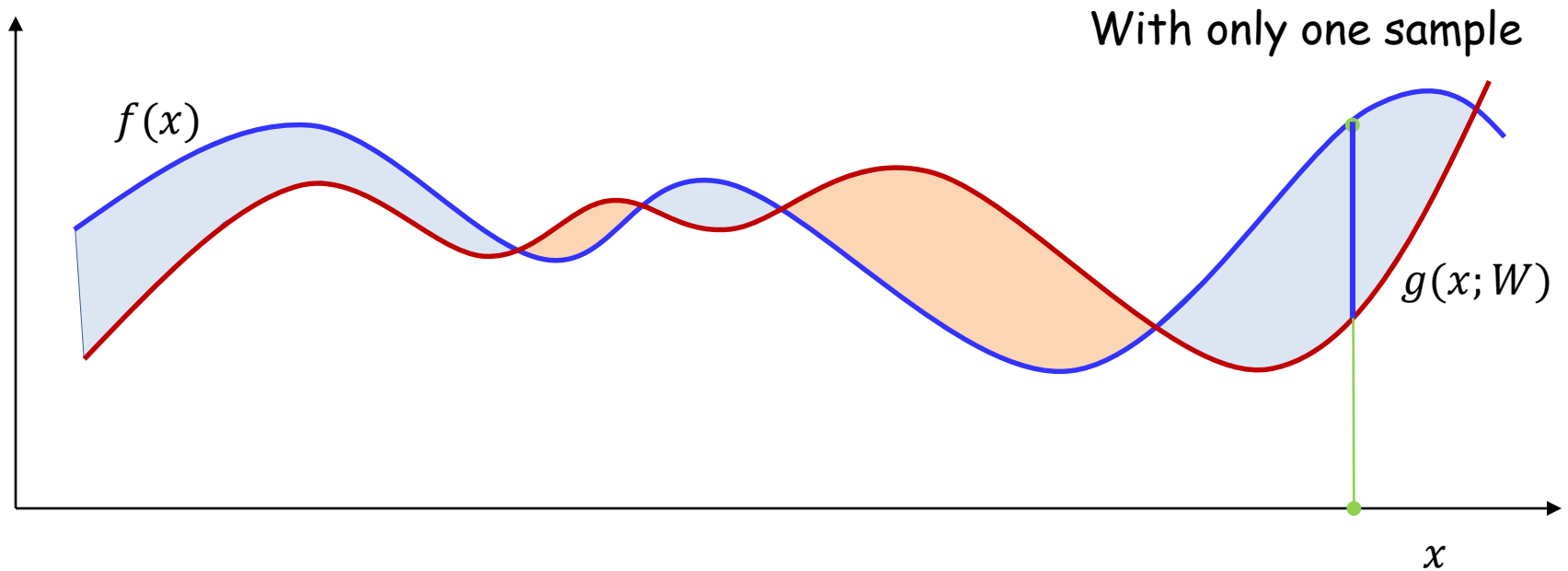
- Having more samples makes the estimate more robust to changes in the position of samples
 - The variance of the estimate is smaller

Explaining the variance



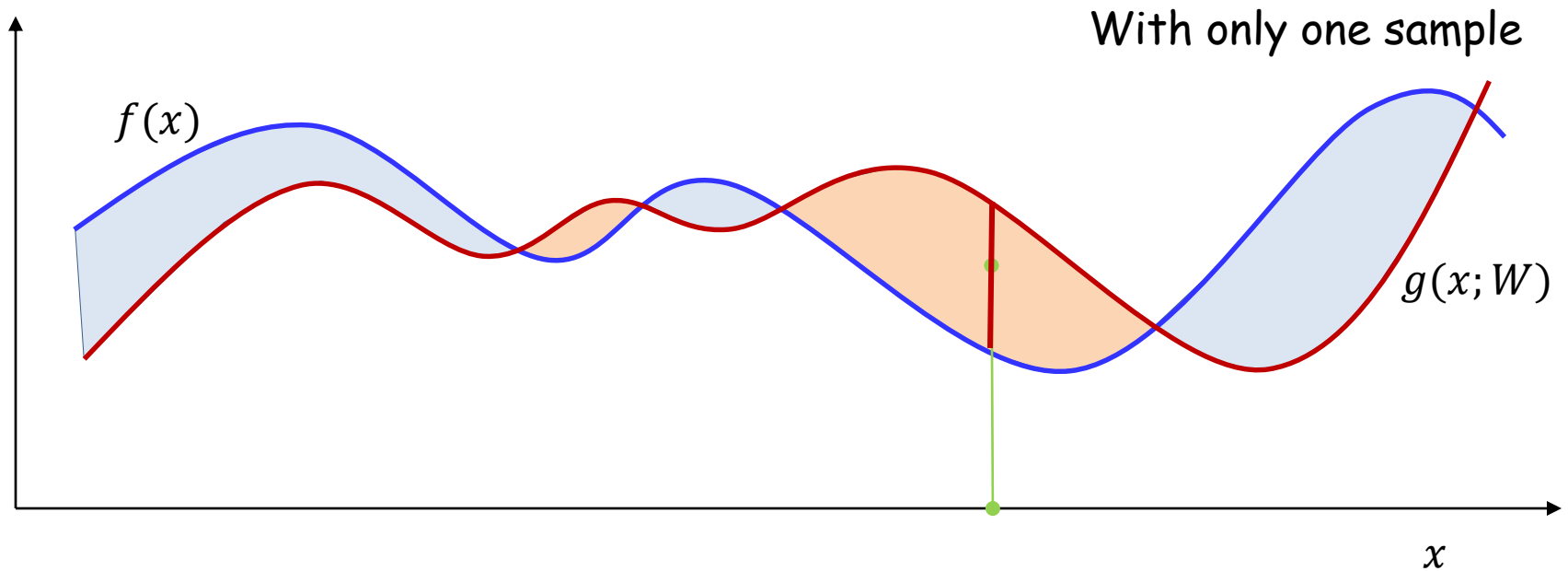
- Having very few samples makes the estimate swing wildly with the sample position
 - Since our estimator learns the W to minimize this estimate, the learned W too can swing wildly

Explaining the variance



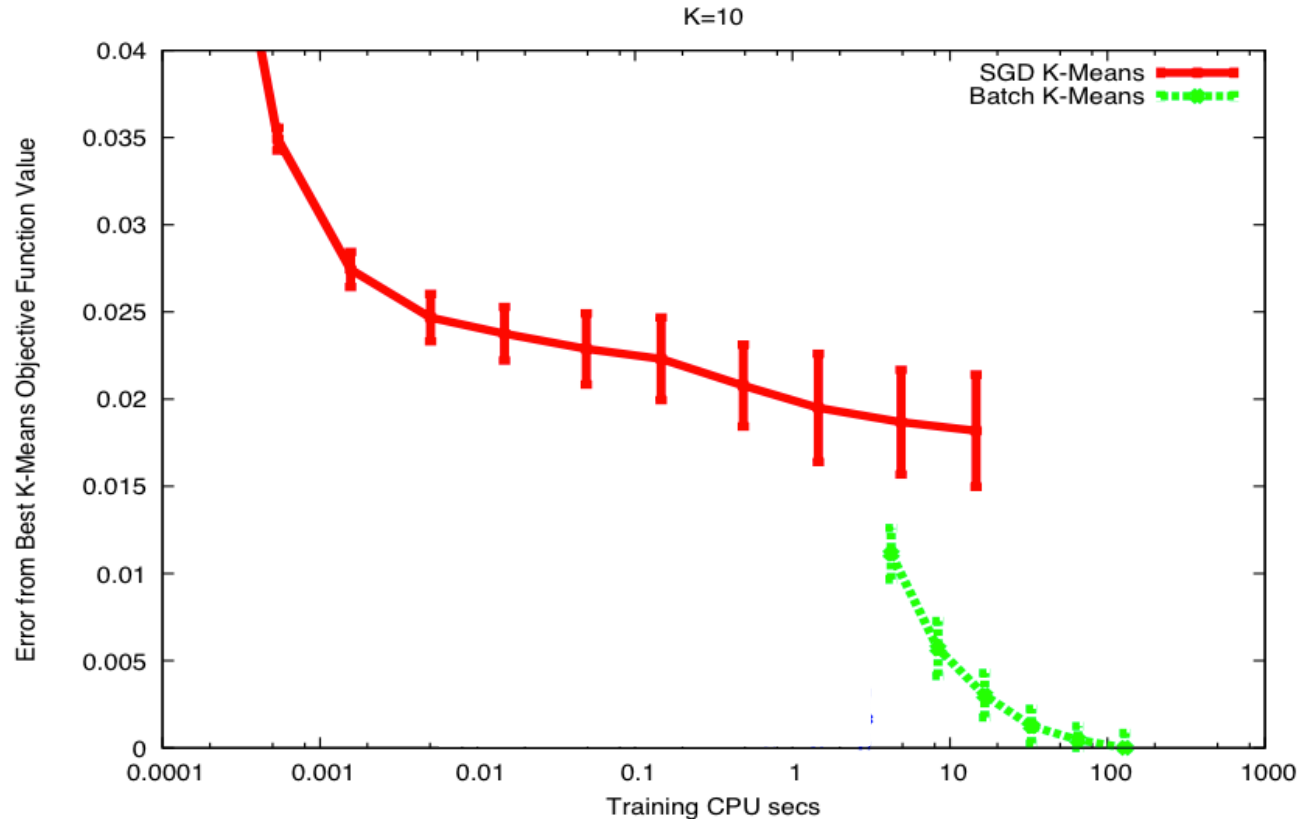
- Having very few samples makes the estimate swing wildly with the sample position
 - Since our estimator learns the W to minimize this estimate, the learned W too can swing wildly

Explaining the variance



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

SGD example

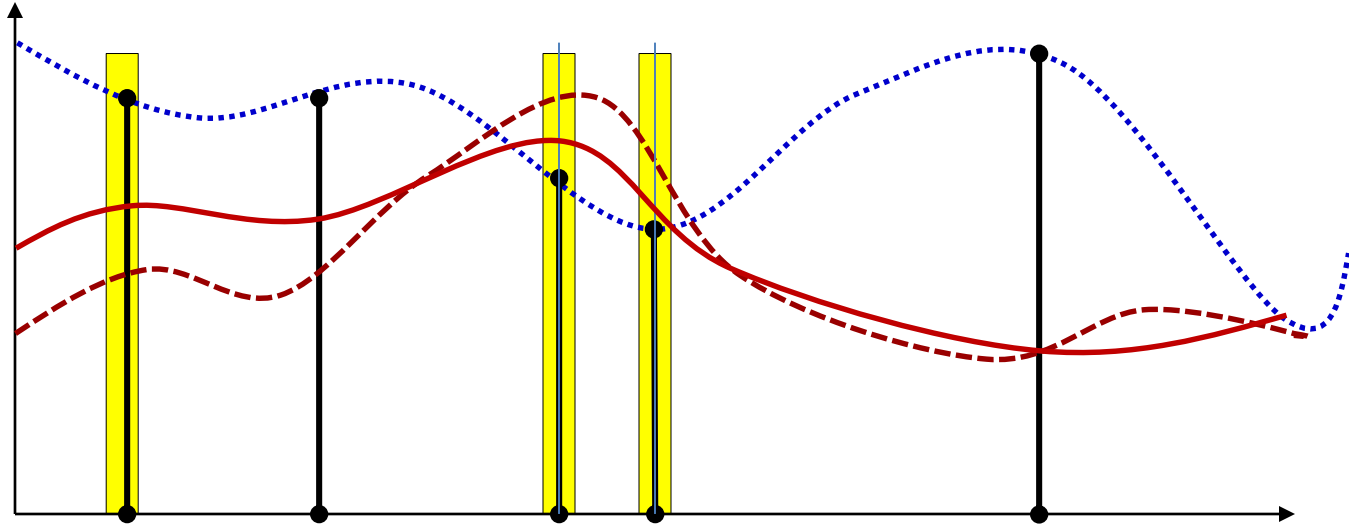


- A simpler problem: K-means
- Note: SGD converges slower
- Also has large variation between runs

SGD vs batch

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

Alternative: Mini-batch update

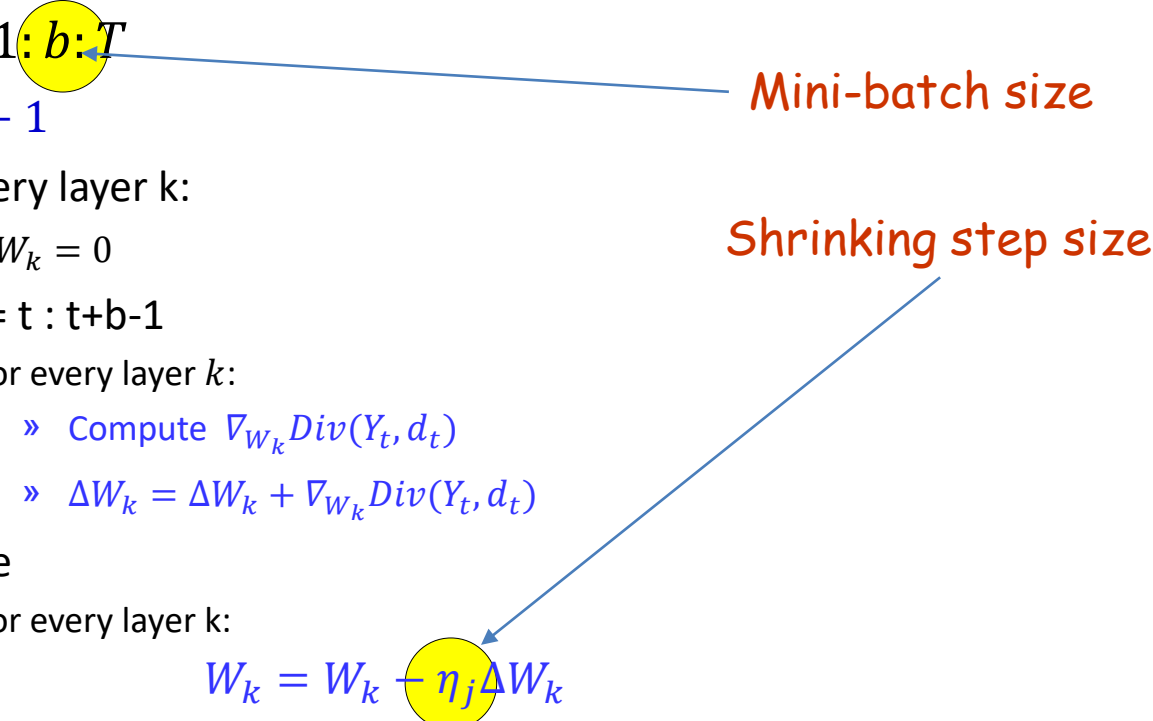


- Alternative: adjust the function at a small, randomly chosen subset of points
 - Keep adjustments small
 - If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data

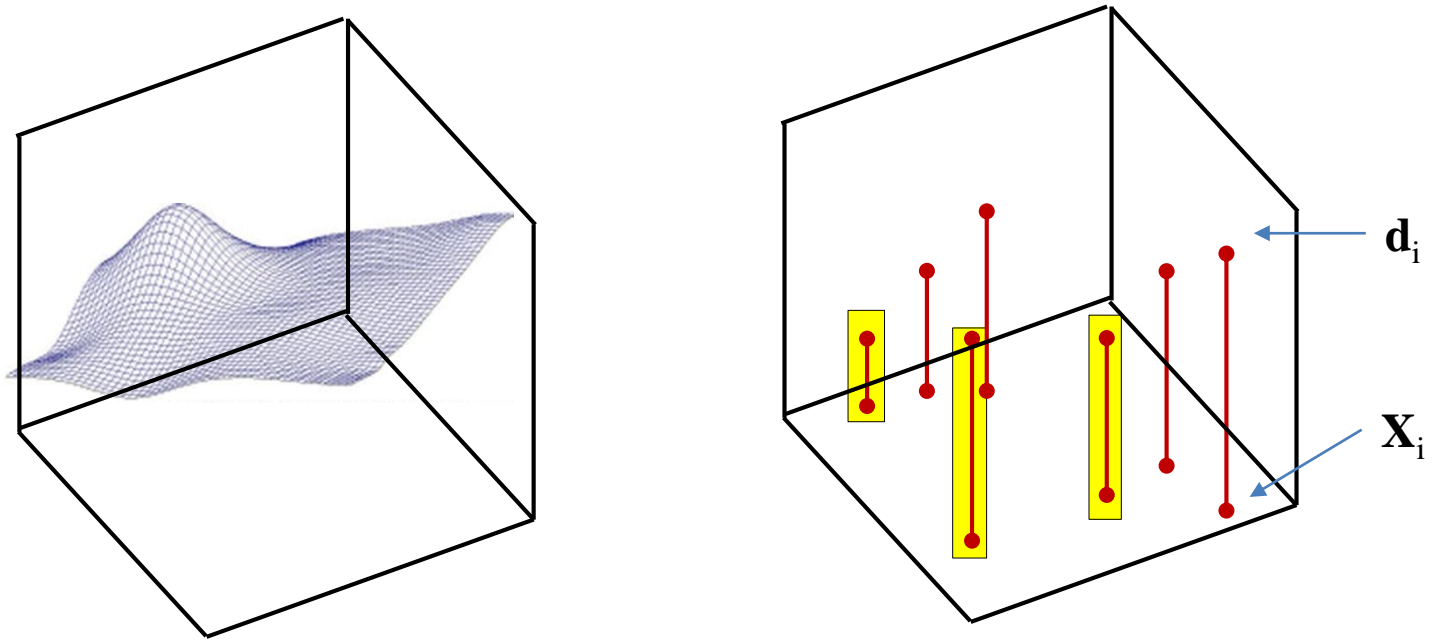
Incremental Update: Mini-batch update

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

Incremental Update: Mini-batch update

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

Mini Batches



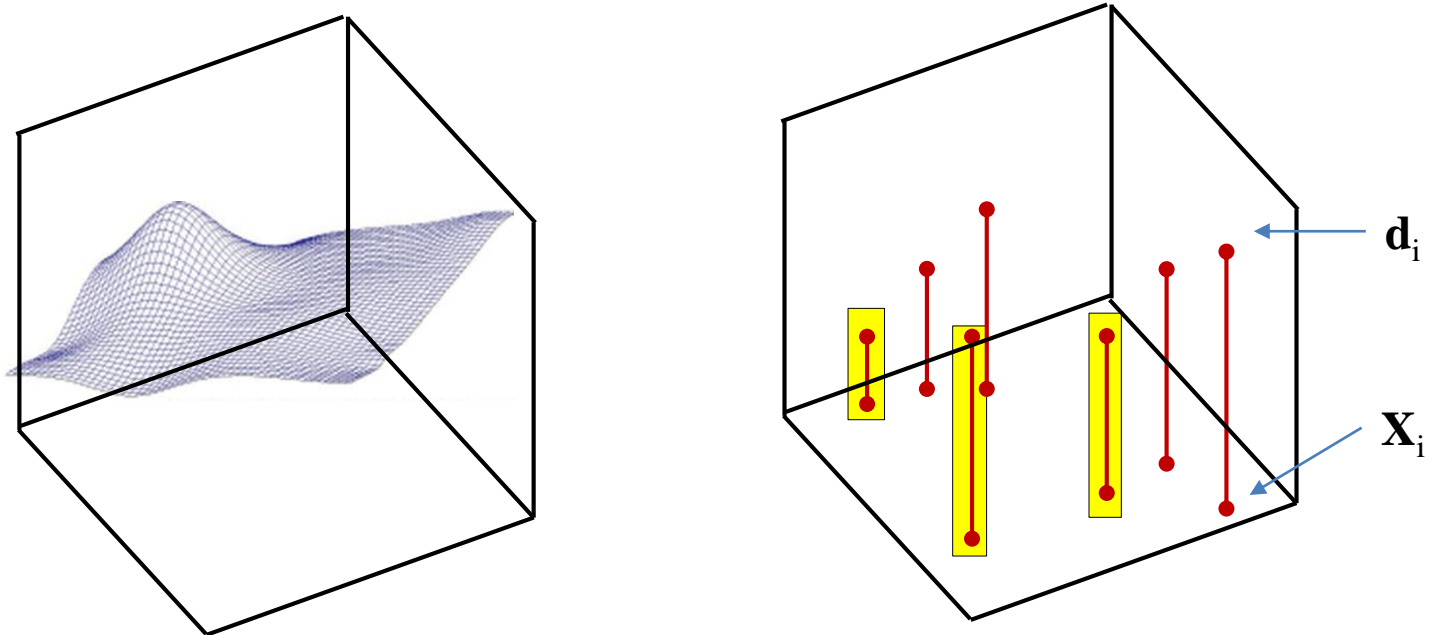
- Mini-batch updates compute and minimize a *batch error*

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

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

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

Mini Batches



- Mini-batch updates computes an *empirical batch error*

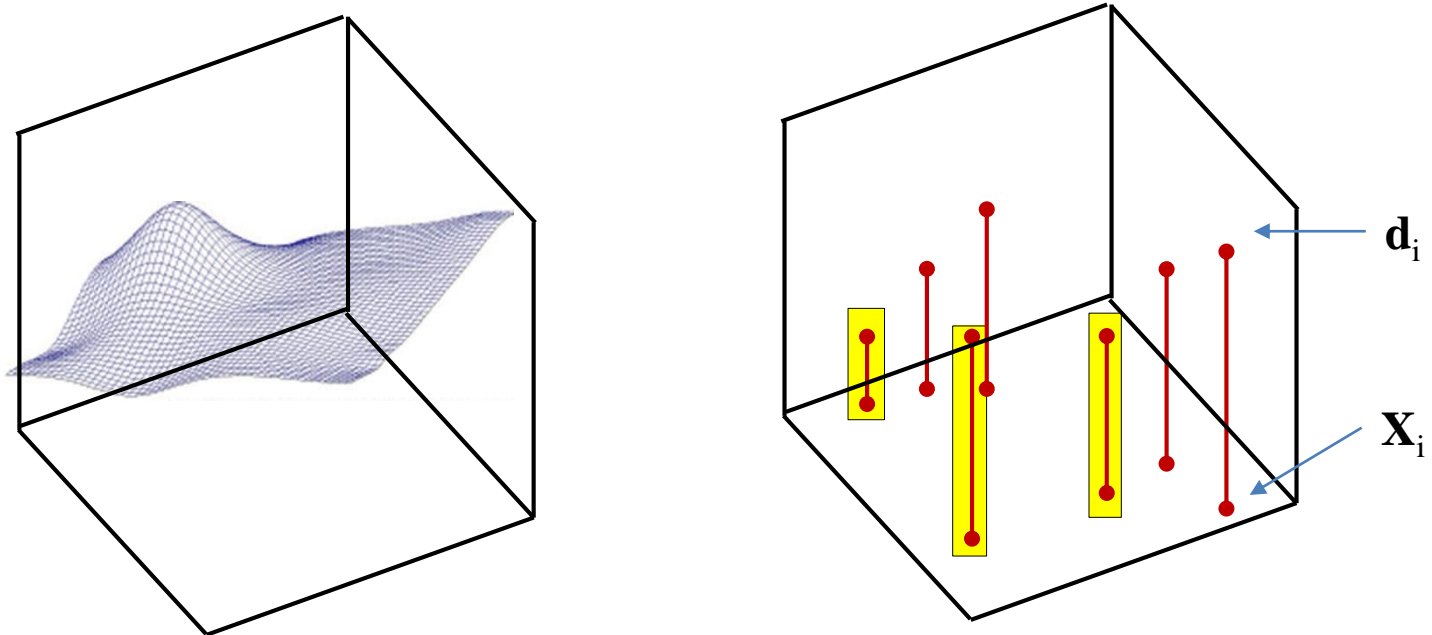
$$BatchErr(f(X;W), g(X)) = \frac{1}{b} \sum_{i=1}^b div(f(X_i;W), d_i)$$

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

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

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

Mini Batches



- Mini-batch updates computes an *empirical batch error*

The variance of the batch error: $\text{var}(\text{Err}) = 1/b \text{ var}(\text{div})$

This will be much smaller than the variance of the sample error in SGD

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

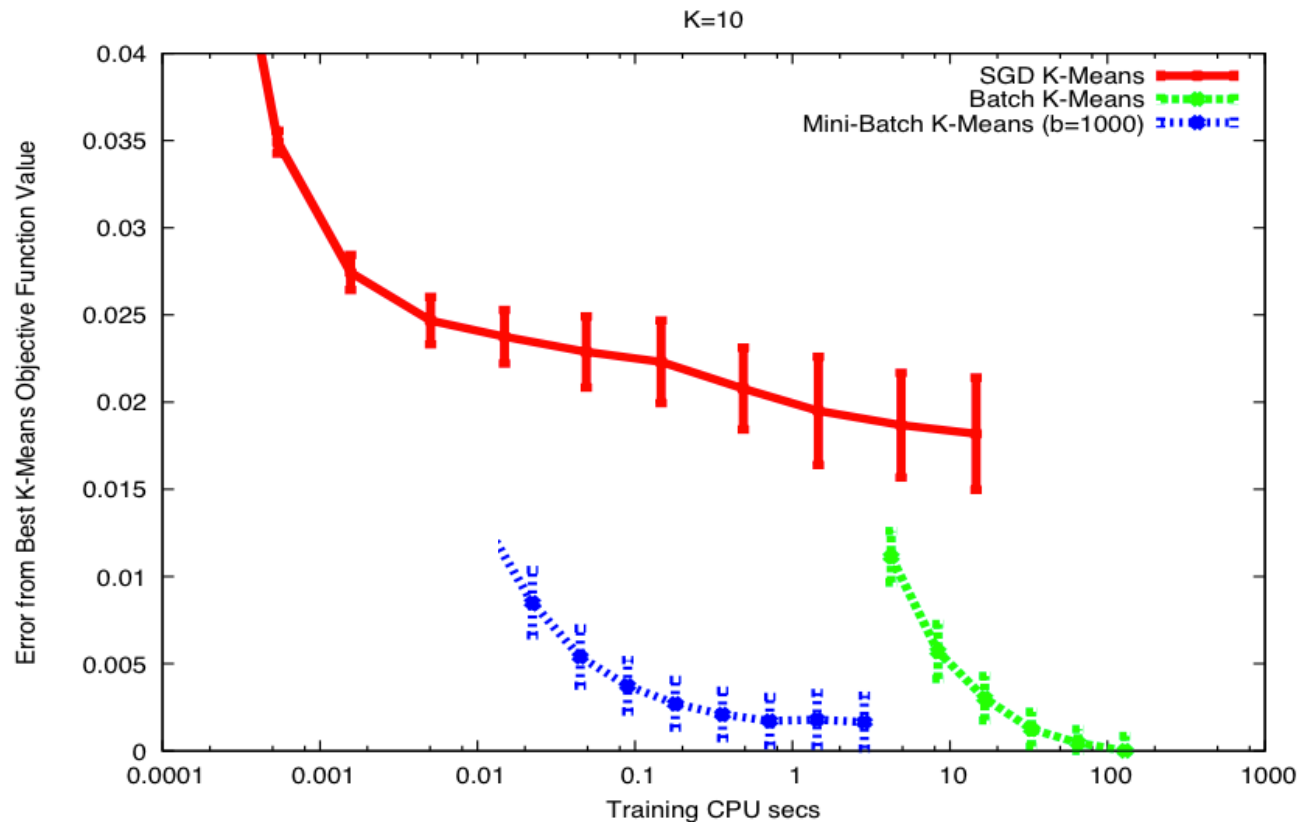
- The *expected value* of the *batch error* is also the *expected divergence*

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

Minibatch convergence

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

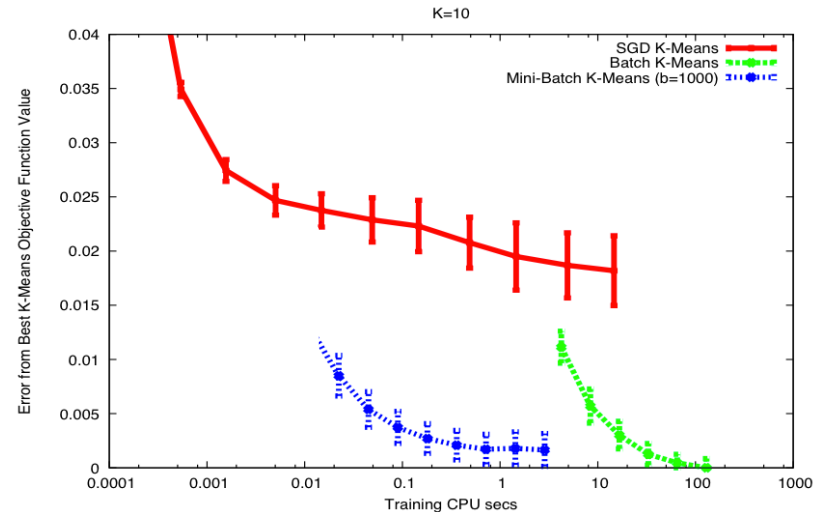
SGD example



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

Measuring Error

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



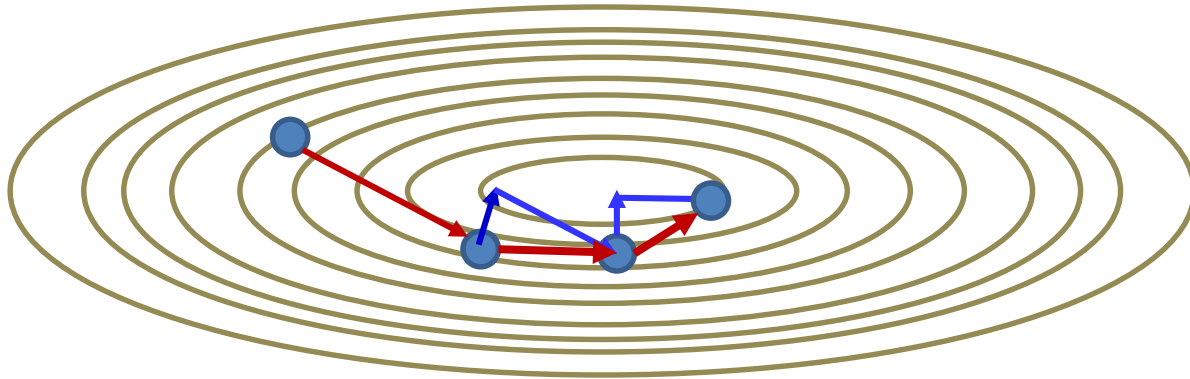
Training and minibatches

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

Training and minibatches

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

Recall: Momentum

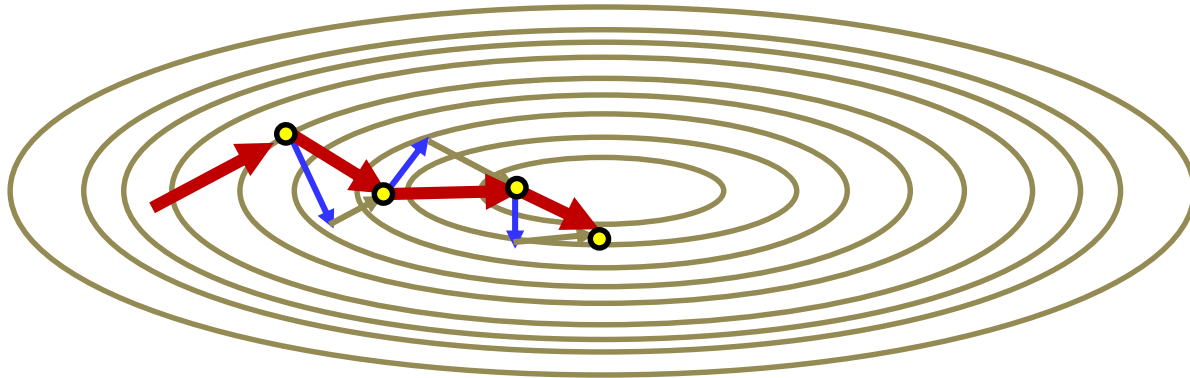


- The momentum method

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

- Updates using a running average of the gradient

Momentum and incremental updates

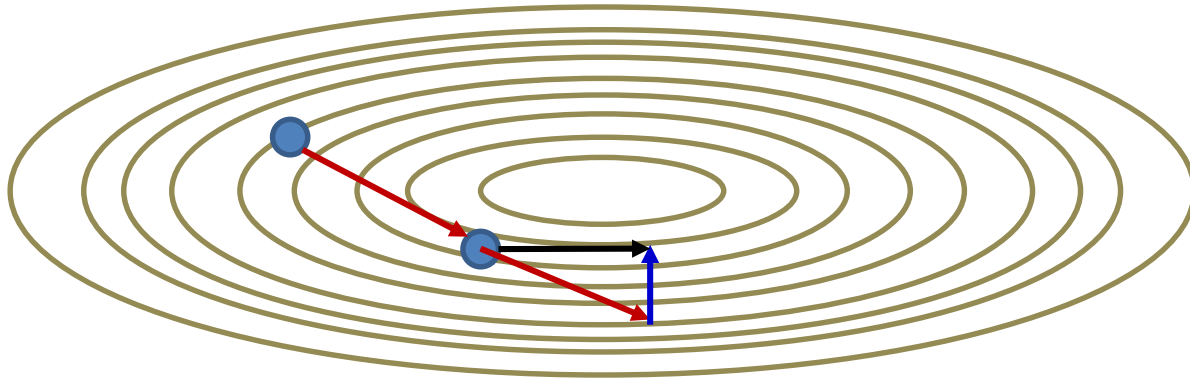


- The momentum method

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

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

Nestorov's Accelerated Gradient



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

More recent methods

- Several newer methods have been proposed that follow the general pattern of enhancing long-term trends to smooth out the variations of the mini-batch gradient
 - RMS Prop
 - **ADAM: very popular in practice**
 - Adagrad
 - AdaDelta
 - ...
- All roughly equivalent in performance

Variance-normalized step



- In recent past
 - Total movement in Y component of updates is high
 - Movement in X components is lower
- Current update, modify usual gradient-based update:
 - Scale *down* Y component
 - Scale *up* X component
- A variety of algorithms have been proposed on this premise
 - We will see a popular example

RMS Prop

- Notation:
 - Updates are *by parameter*
 - Sum derivative of divergence w.r.t any individual parameter w is shown as $\partial_w D$
 - The *squared* derivative is $\partial_w^2 D = (\partial_w D)^2$
 - The *mean squared* derivative is a running estimate of the average squared derivative. We will show this as $E[\partial_w^2 D]$
- Modified update rule: We want to
 - scale down updates with large mean squared derivatives
 - scale up updates with small mean squared derivatives

RMS Prop

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

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

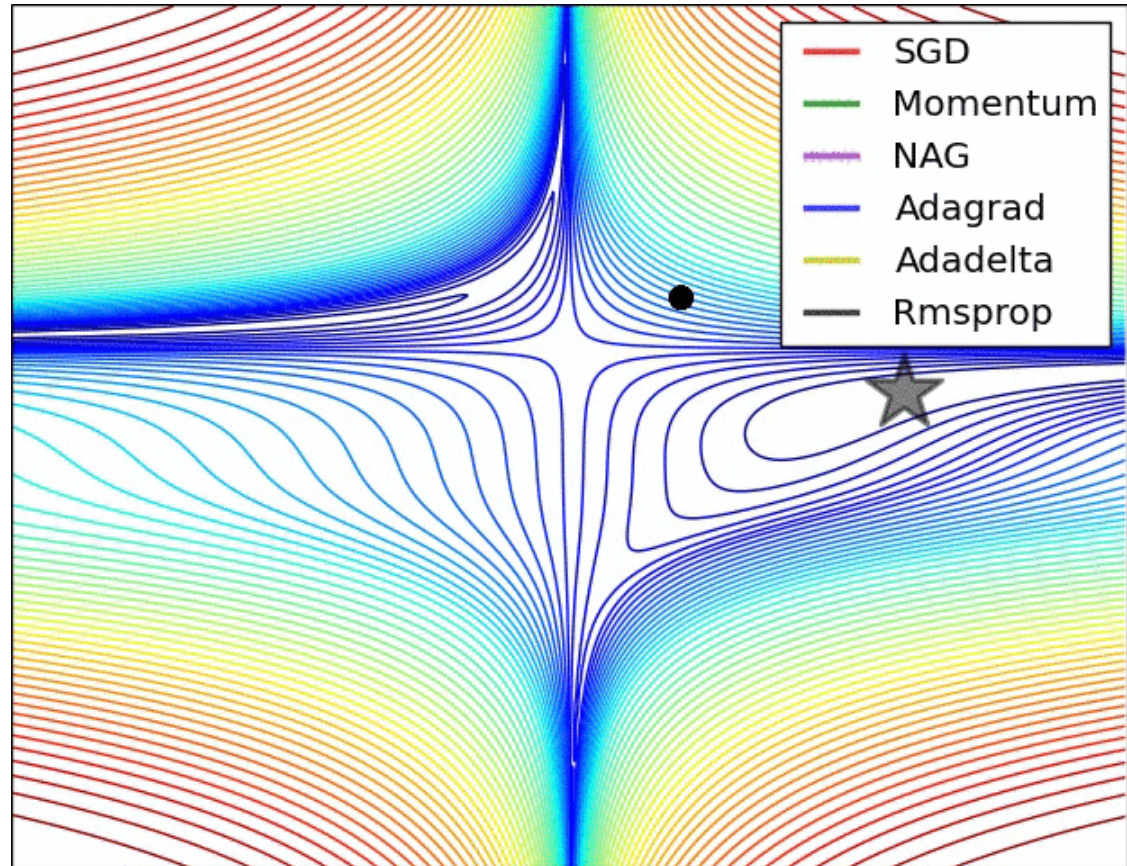
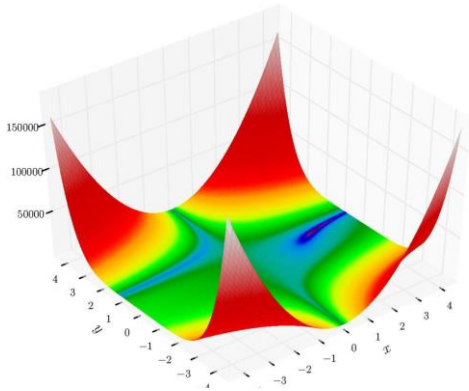
$$w_{k+1} = w_k - \frac{\eta}{\sqrt{E[\partial_w^2 D]_k + \epsilon}} \partial_w D$$

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

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

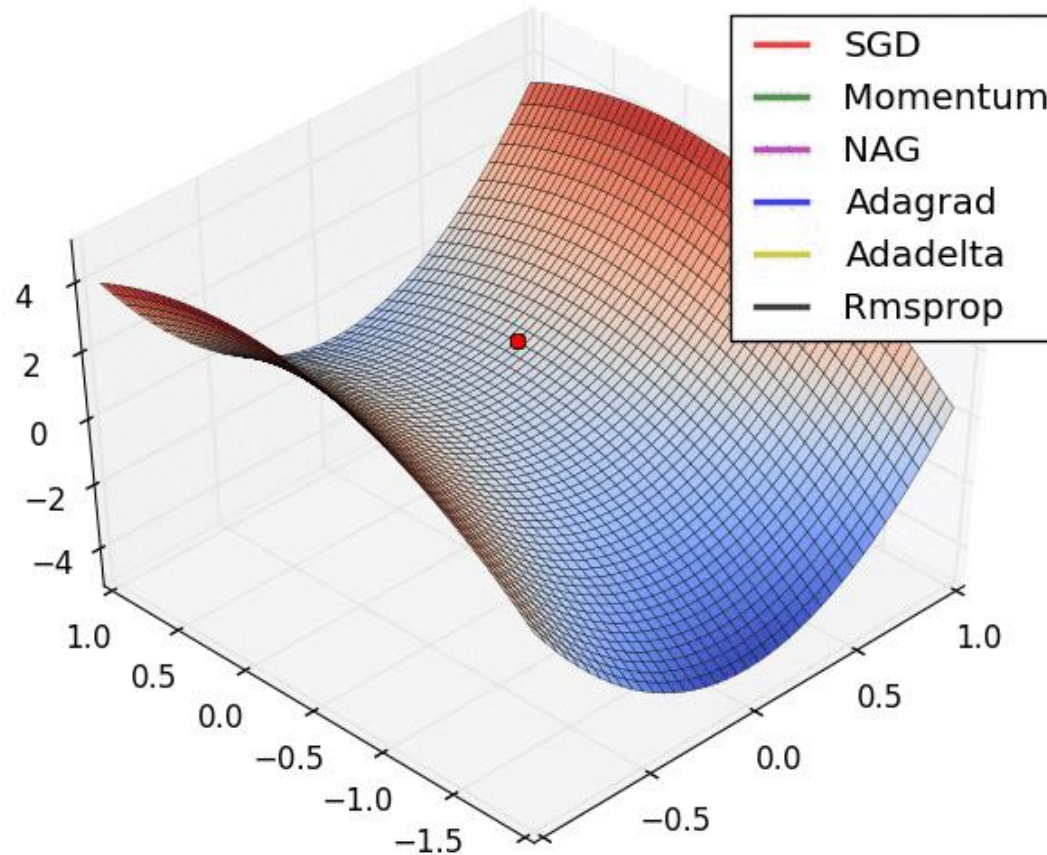
$$E[\partial_w^2 D]_k = \gamma E[\partial_w^2 D]_{k-1} + (1 - \gamma)(\partial_w^2 D)_k$$
$$w_{k+1} = w_k - \frac{\eta}{\sqrt{E[\partial_w^2 D]_k + \epsilon}} \partial_w D$$
 - $k = k + 1$
 - Until $E(W^{(1)}, W^{(2)}, \dots, W^{(K)})$ has converged

Visualizing the optimizers: Beale's Function



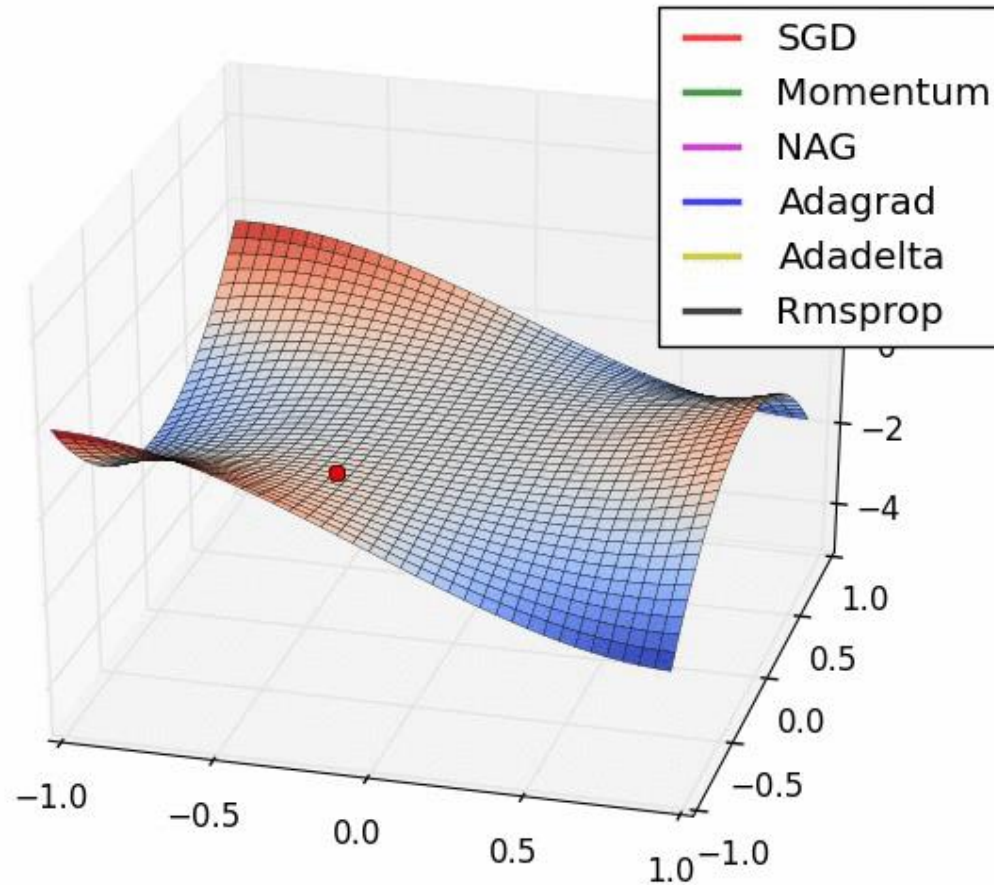
- <http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html>

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
 - 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

Topics for the day

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

Tricks of the trade..

- To make the network converge better
 - The Divergence
 - Dropout
 - Batch normalization
 - Other tricks
 - Gradient clipping
 - Data augmentation
 - Other hacks..

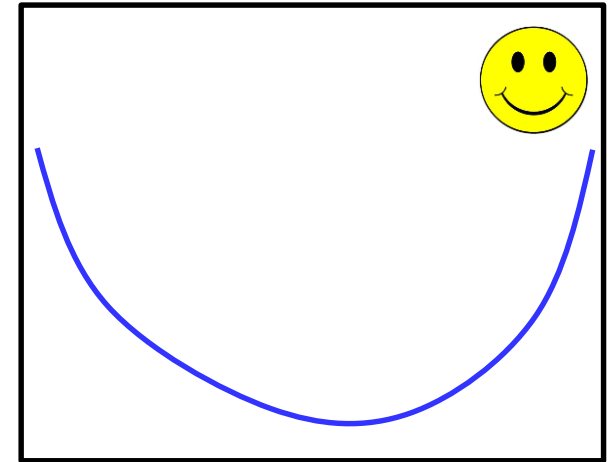
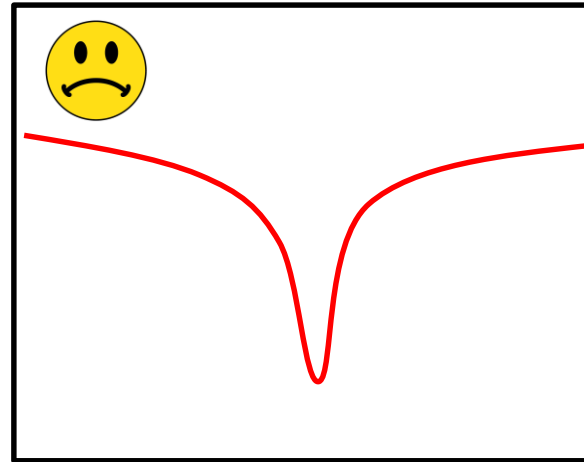
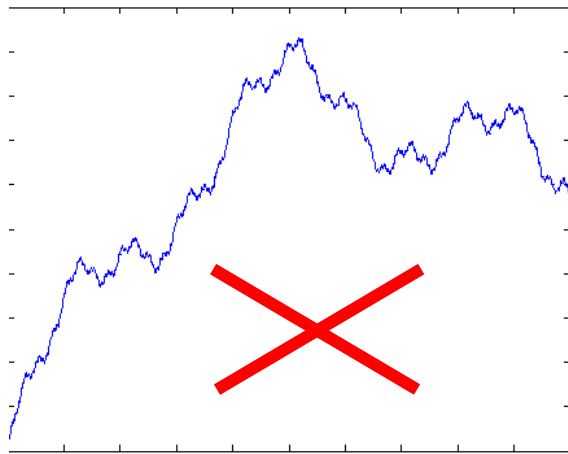
Training Neural Nets by Gradient Descent: The Divergence

Total training error:

$$Err = \frac{1}{T} \sum_t Div(\mathbf{Y}_t, \mathbf{d}_t; \mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_K)$$

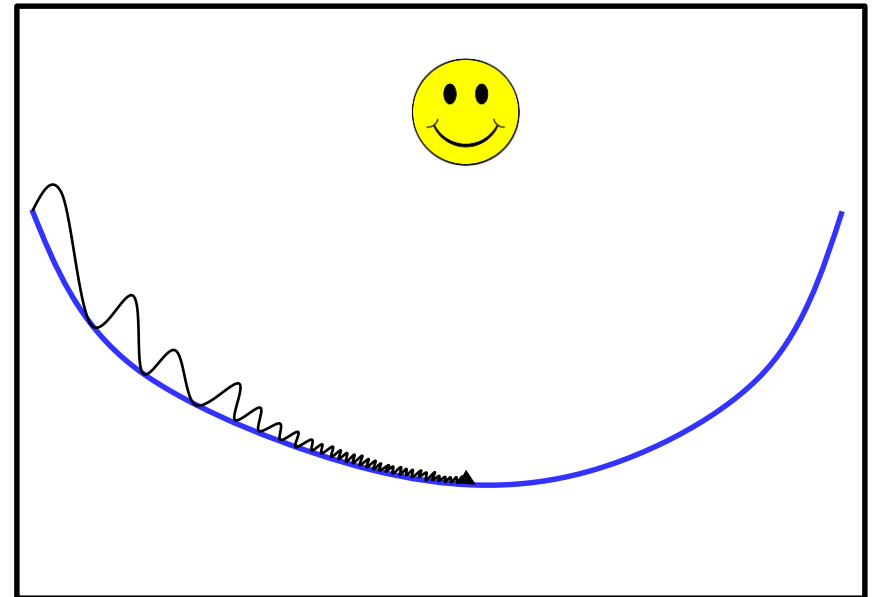
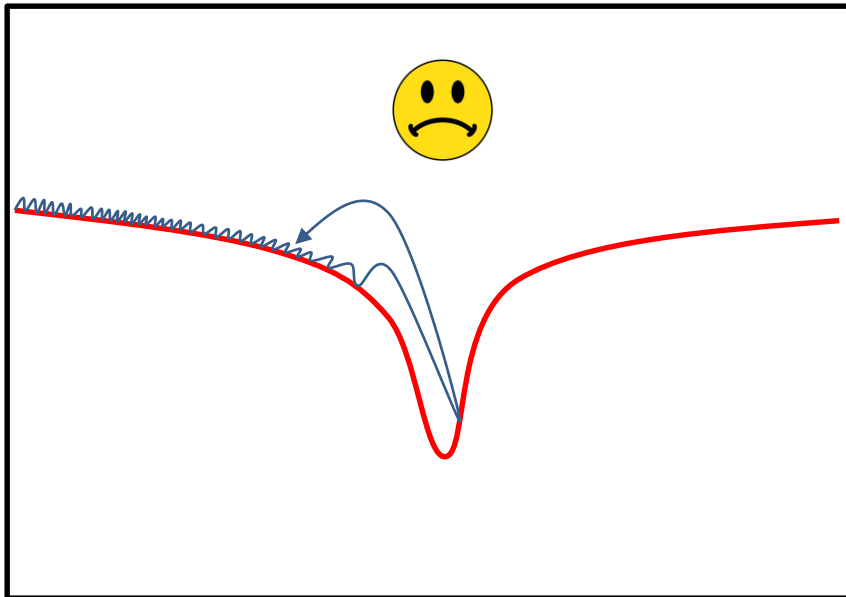
- The convergence of the gradient descent depends on the divergence
 - Ideally, must have a shape that results in a significant gradient in the right direction outside the optimum
 - To “guide” the algorithm to the right solution

Desiderata for a good divergence



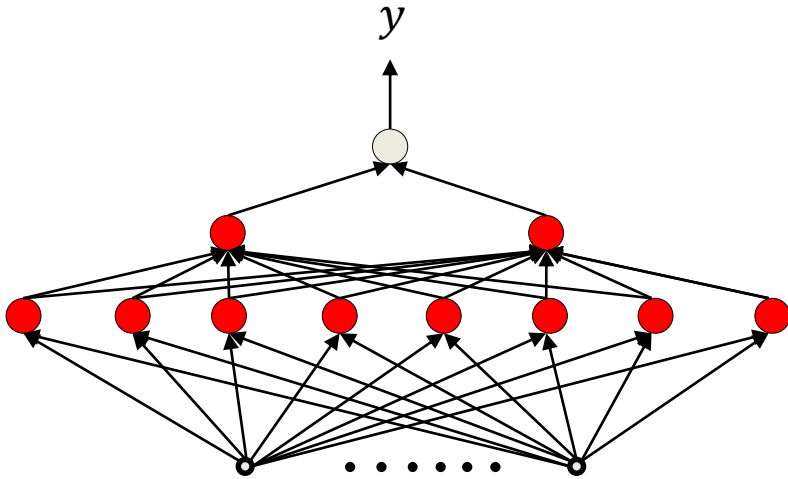
- Must be smooth and not have many poor local optima
- Low slopes far from the optimum == bad
 - Initial estimates far from the optimum will take forever to converge
- High slopes near the optimum == bad
 - Steep gradients

Desiderata for a good divergence



- Functions that are shallow far from the optimum will result in very small steps during optimization
 - Slow convergence of gradient descent
- Functions that are steep near the optimum will result in large steps and overshoot during optimization
 - Gradient descent will not converge easily
- The best type of divergence is steep far from the optimum, but shallow at the optimum
 - But not *too* shallow: ideally quadratic in nature

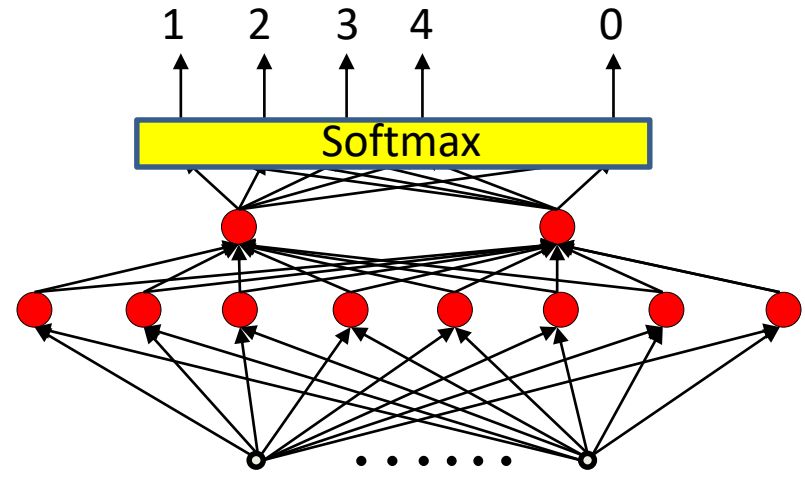
Choices for divergence



Desired output: d

L2 $Div = (y - d)^2$

KL $Div = d \log(y) + (1 - d) \log(1 - y)$



Desired output: $[0, 0, \dots, 1, \dots, 0]$

$$Div = \sum_i (y_i - d_i)^2$$

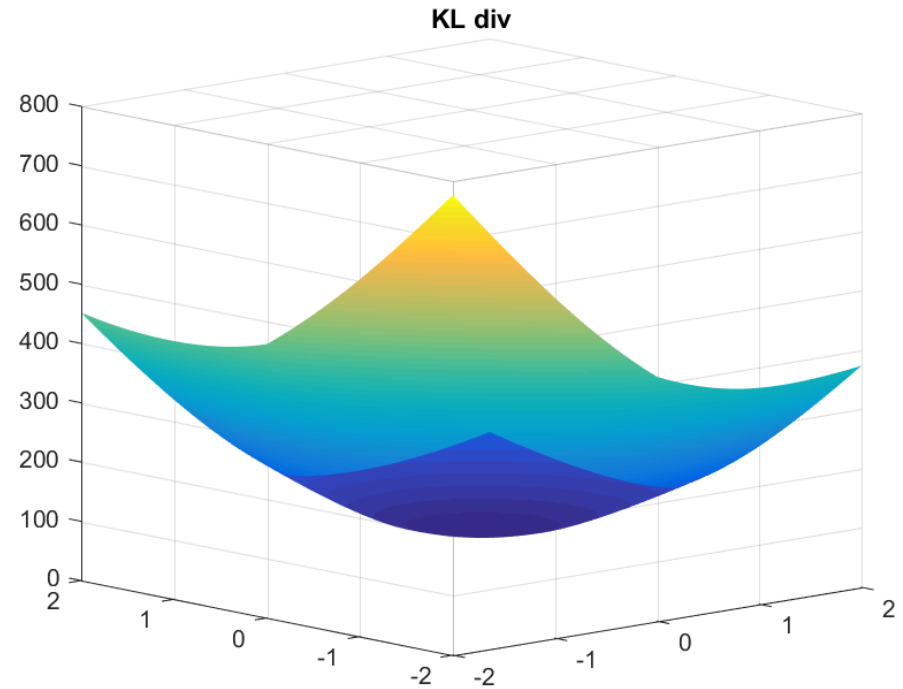
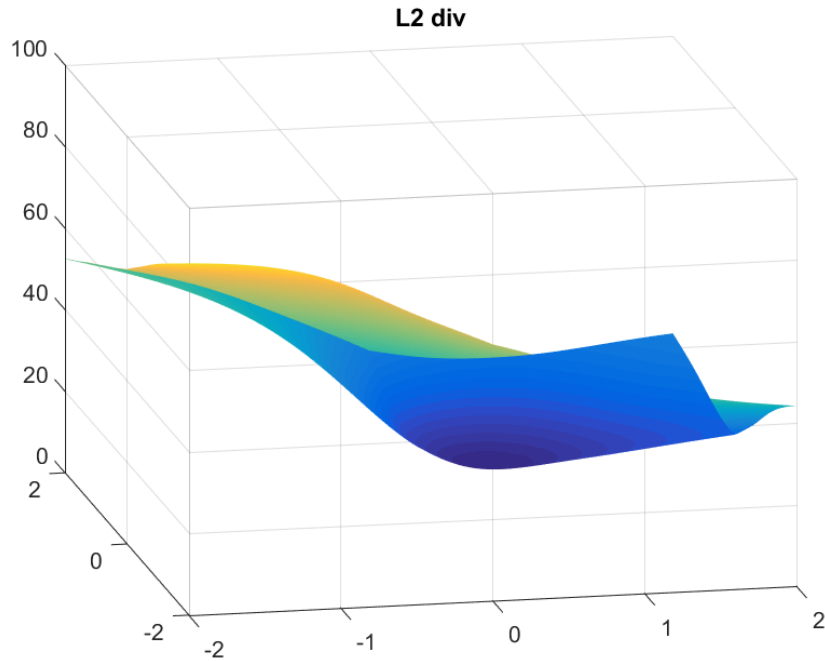
$$Div = \sum_i d_i \log(y_i)$$

- Most common choices: The L2 divergence and the KL divergence

L2 or KL?

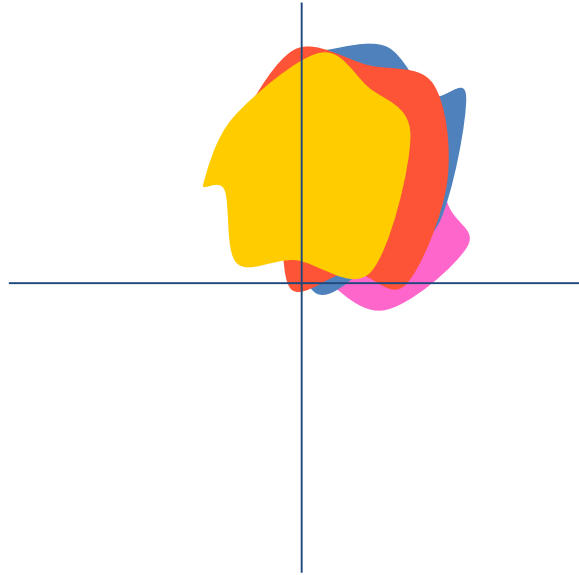
- The L2 divergence has long been favored in most applications
- It is particularly appropriate when attempting to perform *regression*
 - Numeric prediction
- The KL divergence is better when the intent is classification
 - The output is a probability vector

L2 or KL



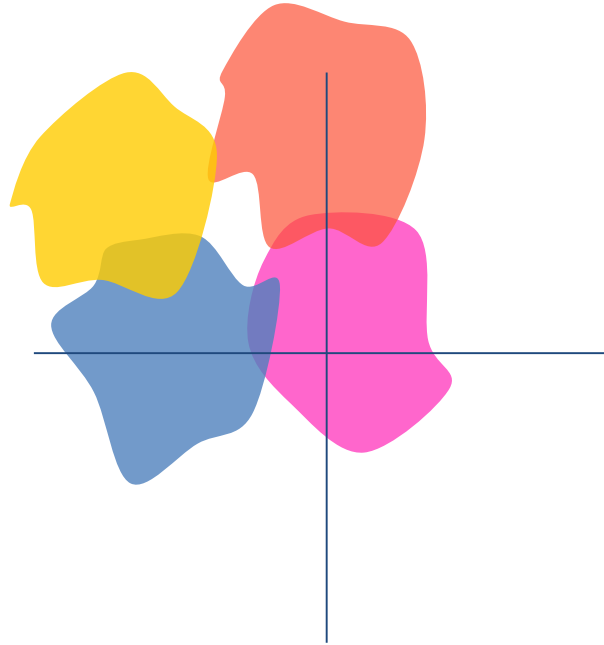
- Plot of L2 and KL divergences for a *single* perceptron, as function of weights
 - Setup: 2-dimensional input
 - 100 training examples randomly generated

The problem of covariate shifts



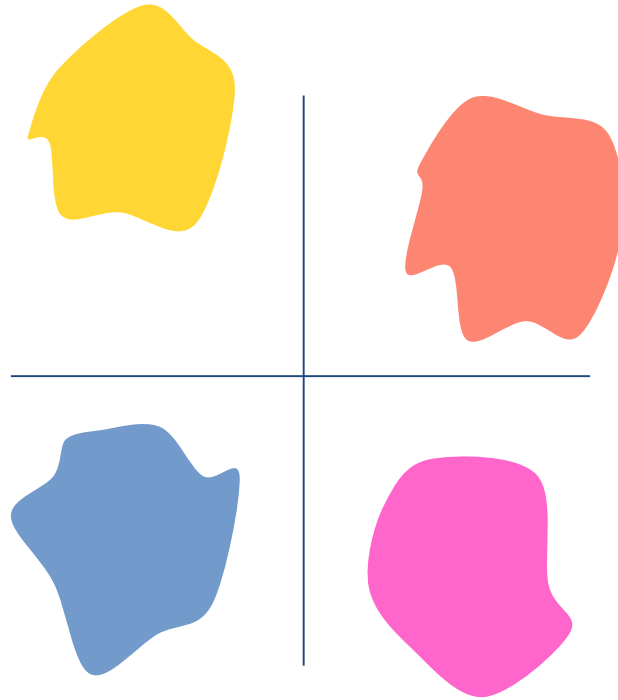
- Training assumes the training data are all similarly distributed
 - Minibatches have similar distribution

The problem of covariate shifts



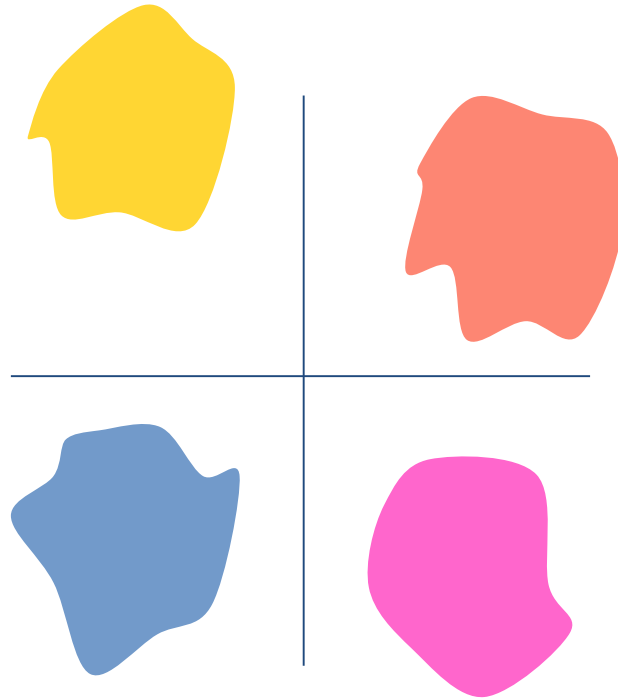
- Training assumes the training data are all similarly distributed
 - Minibatches have similar distribution
- In practice, each minibatch may have a different distribution
 - A “covariate shift”
 - Which may occur in *each* layer of the network

The problem of covariate shifts



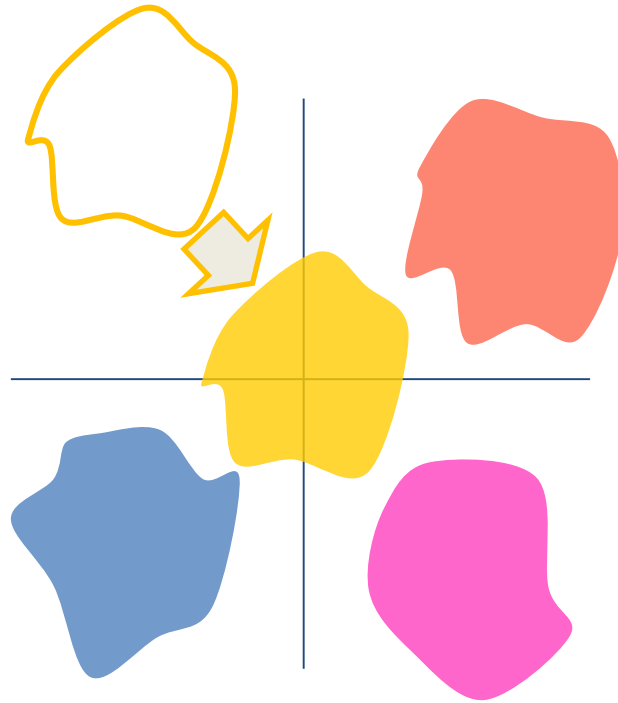
- Training assumes the training data are all similarly distributed
 - Minibatches have similar distribution
- In practice, each minibatch may have a different distribution
 - A “covariate shift”
- Covariate shifts can be large!
 - All covariate shifts can affect training badly

Solution: Move all subgroups to a “standard” location



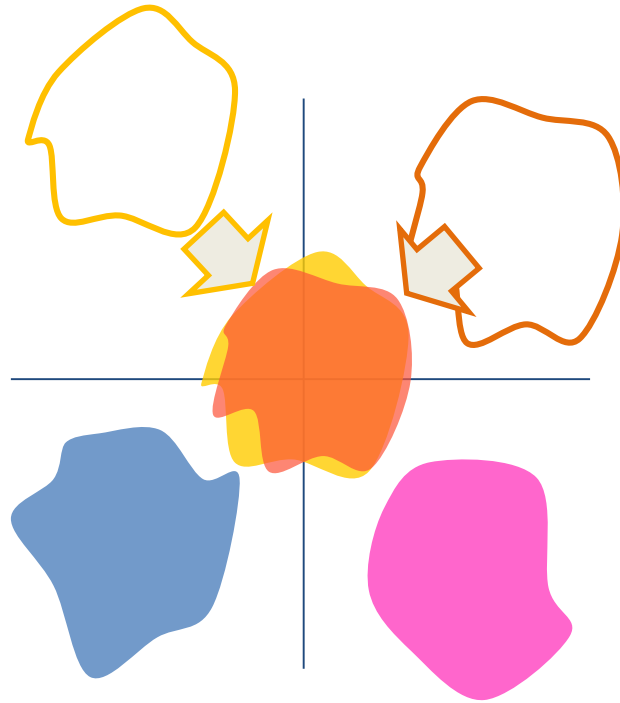
- “Move” all batches to have a mean of 0 and unit standard deviation
 - Eliminates covariate shift between batches

Solution: Move all subgroups to a “standard” location



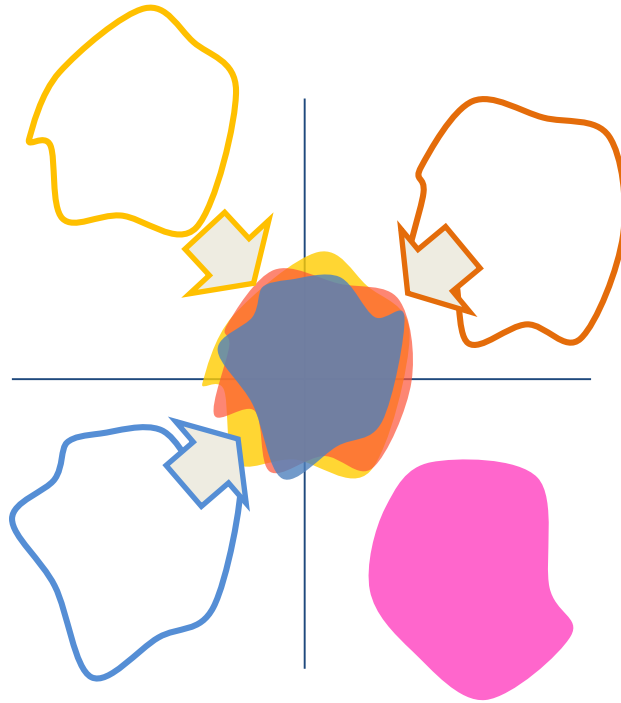
- “Move” all batches to have a mean of 0 and unit standard deviation
 - Eliminates covariate shift between batches

Solution: Move all subgroups to a “standard” location



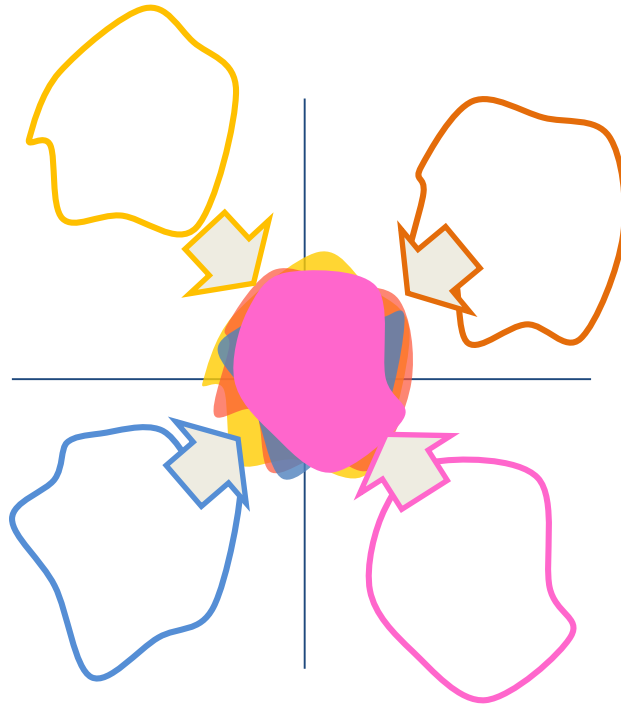
- “Move” all batches to have a mean of 0 and unit standard deviation
 - Eliminates covariate shift between batches

Solution: Move all subgroups to a “standard” location



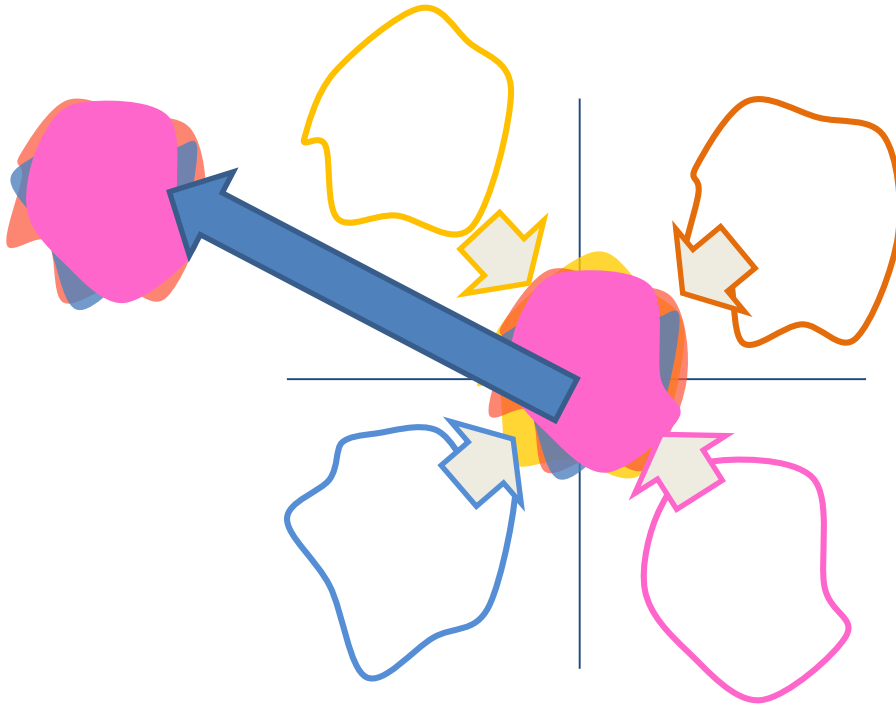
- “Move” all batches to have a mean of 0 and unit standard deviation
 - Eliminates covariate shift between batches

Solution: Move all subgroups to a “standard” location



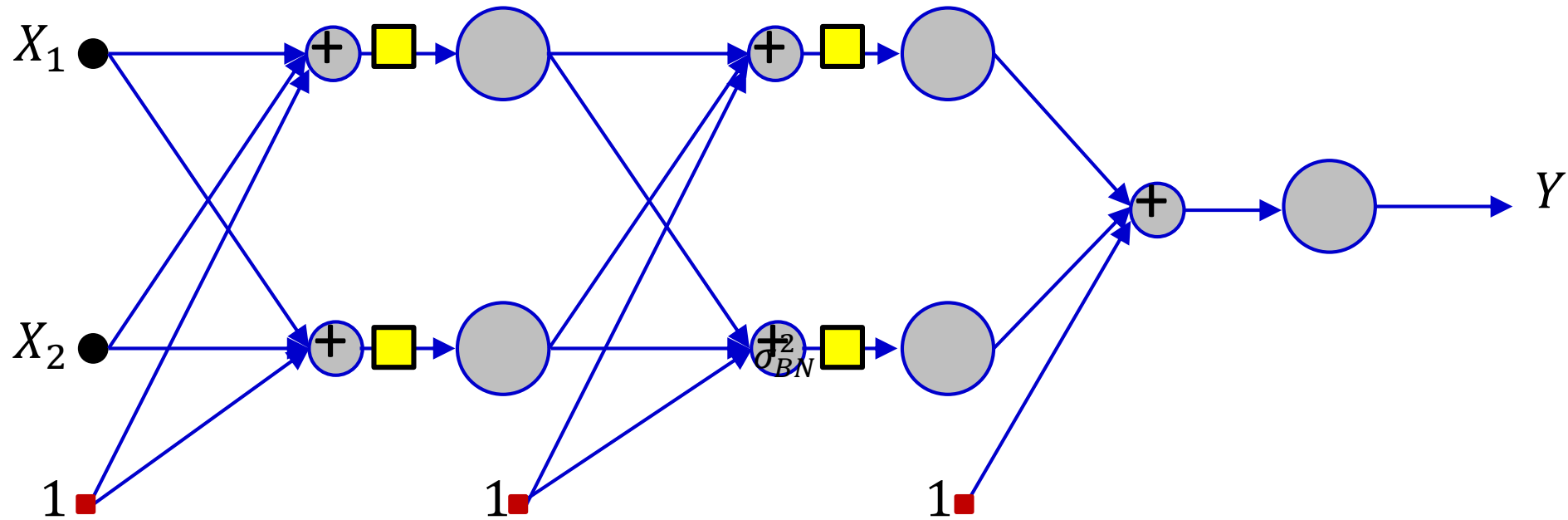
- “Move” all batches to have a mean of 0 and unit standard deviation
 - Eliminates covariate shift between batches

Solution: Move all subgroups to a “standard” location



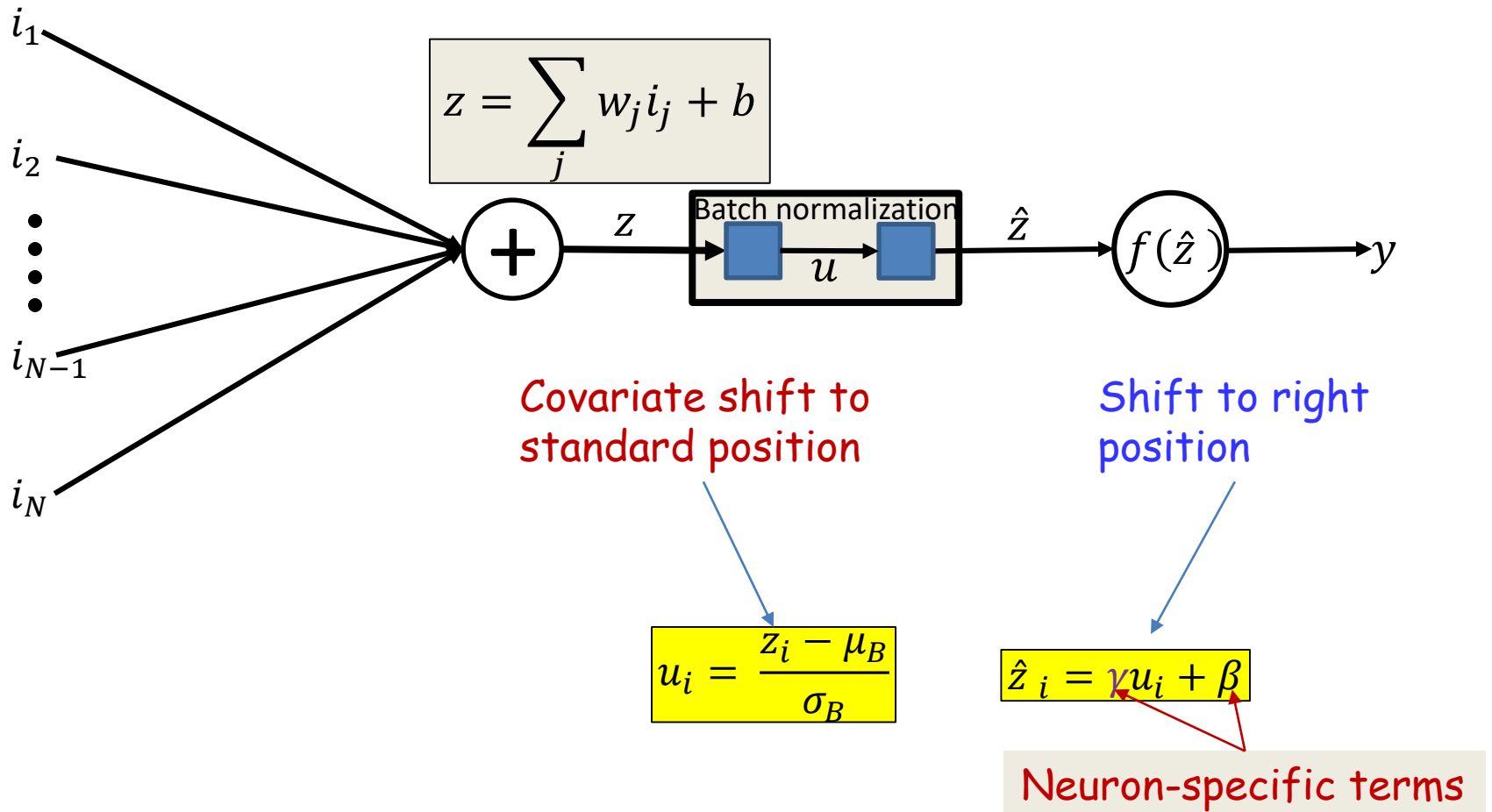
- “Move” all batches to have a mean of 0 and unit standard deviation
 - Eliminates covariate shift between batches
 - Then move the entire collection to the appropriate location

Batch normalization



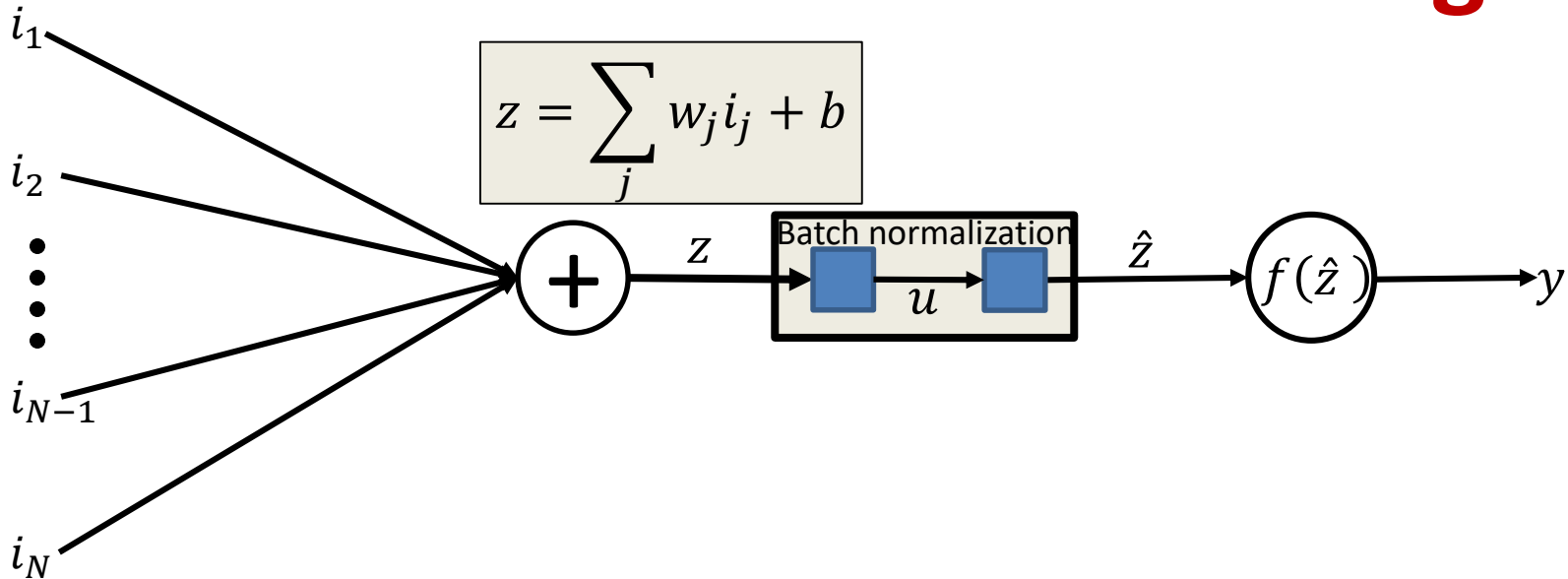
- Batch normalization is a covariate adjustment unit that happens after the weighted addition of inputs but before the application of activation
 - Is done independently for each unit, to simplify computation
- **Training:** The adjustment occurs over individual minibatches

Batch normalization



- BN aggregates the statistics over a minibatch and normalizes the batch by them
- Normalized instances are “shifted” to a *unit-specific* location

Batch normalization: Training



$$\mu_B = \frac{1}{B} \sum_{i=1}^B z_i$$

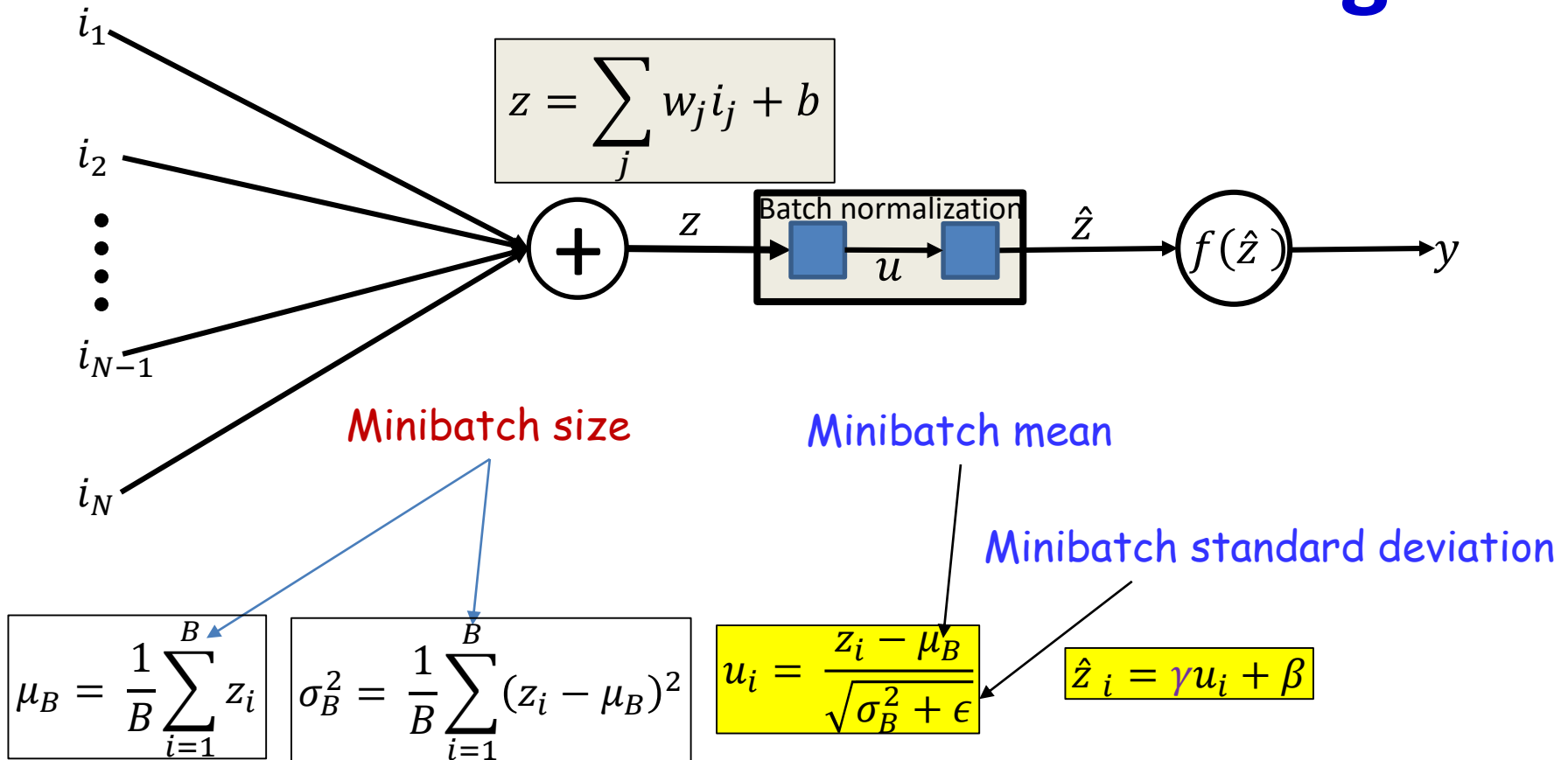
$$\sigma_B^2 = \frac{1}{B} \sum_{i=1}^B (z_i - \mu_B)^2$$

$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\hat{z}_i = \gamma u_i + \beta$$

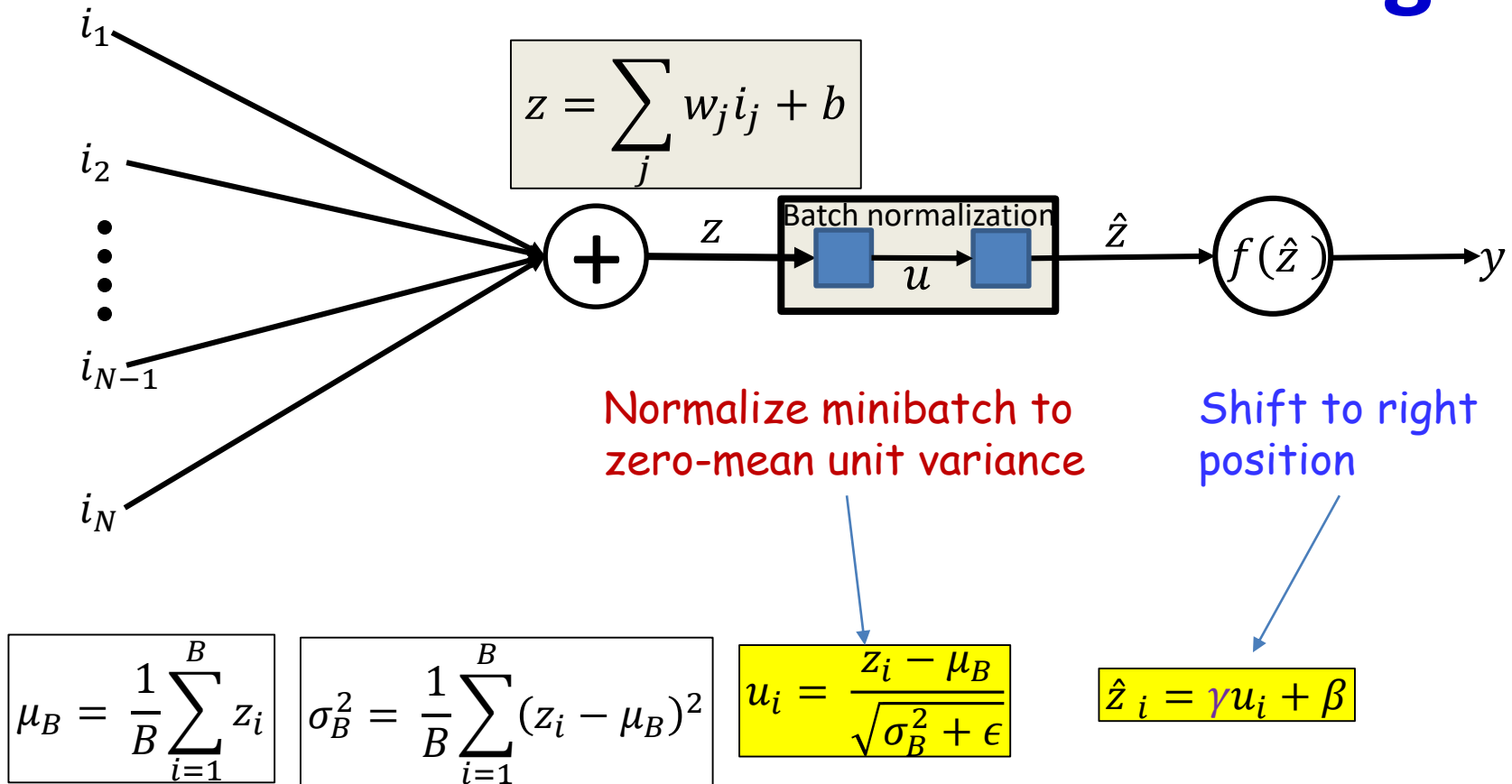
- BN aggregates the statistics over a minibatch and normalizes the batch by them
- Normalized instances are “shifted” to a *unit-specific* location

Batch normalization: Training



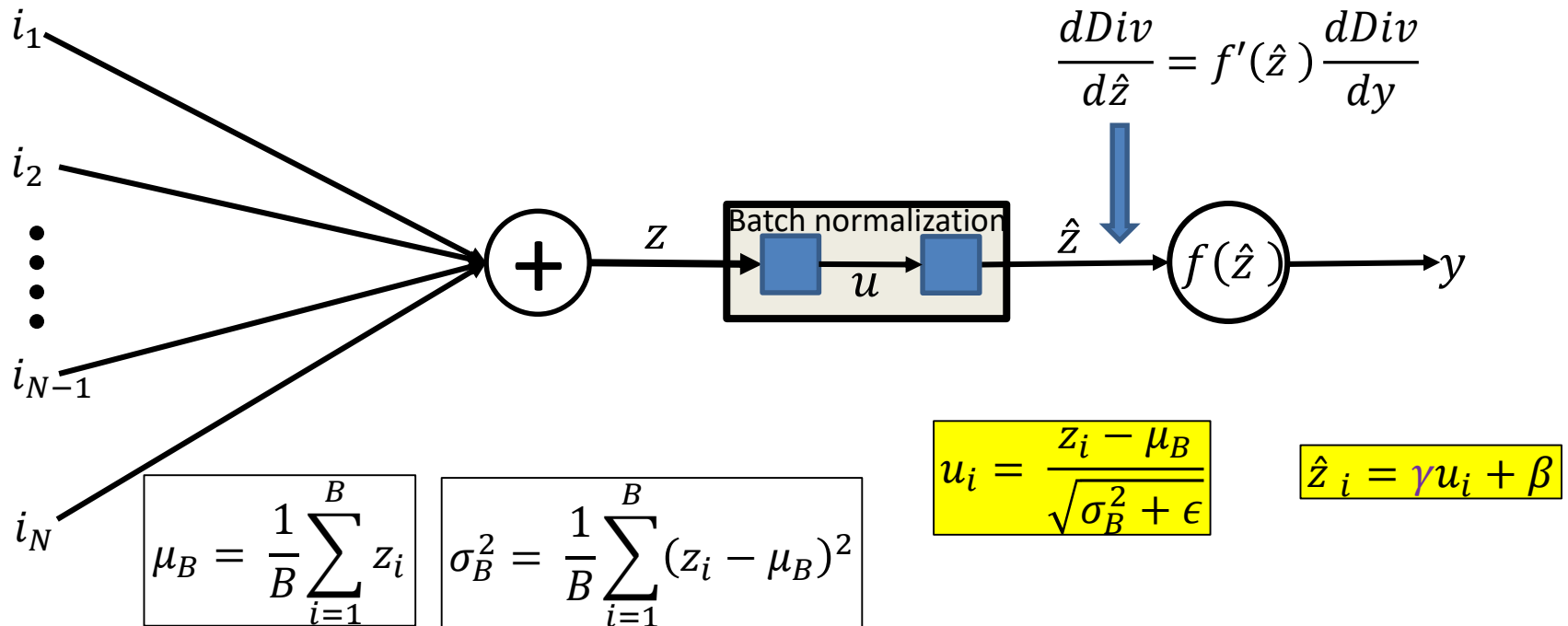
- BN aggregates the statistics over a minibatch and normalizes the batch by them
- Normalized instances are “shifted” to a *unit-specific* location

Batch normalization: Training

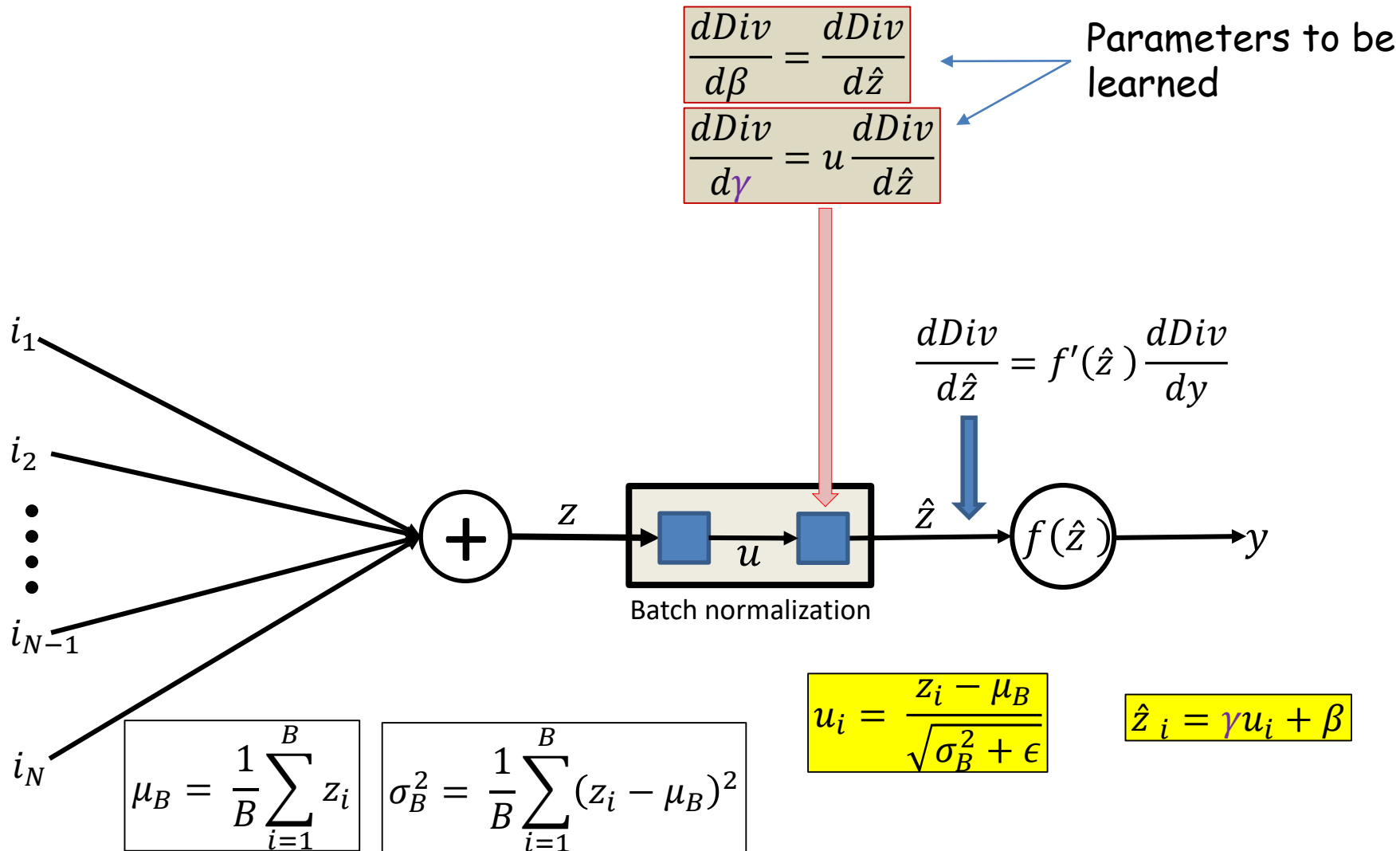


- BN aggregates the statistics over a minibatch and normalizes the batch by them
- Normalized instances are “shifted” to a *unit-specific* location

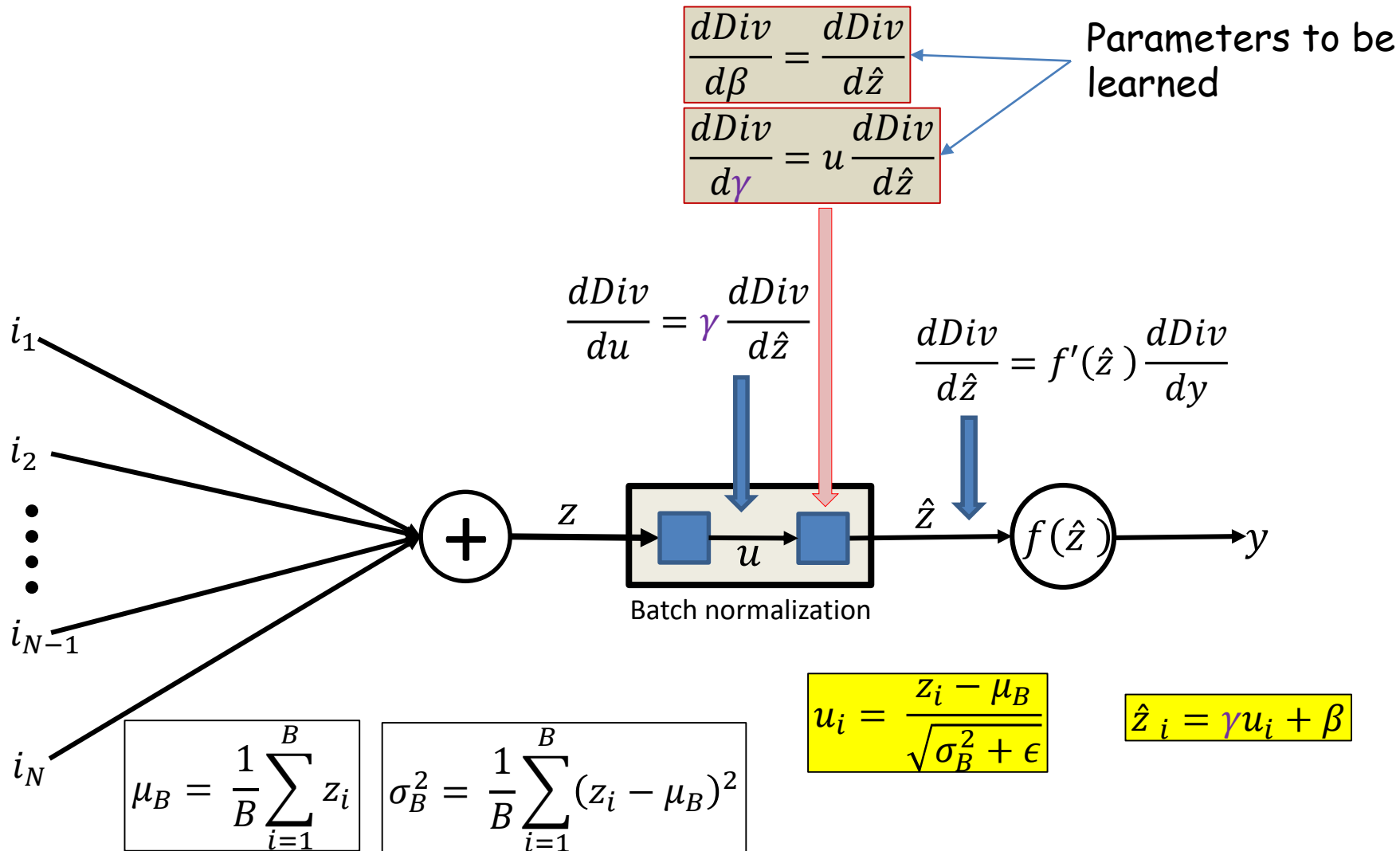
Batch normalization: Backpropagation



Batch normalization: Backpropagation

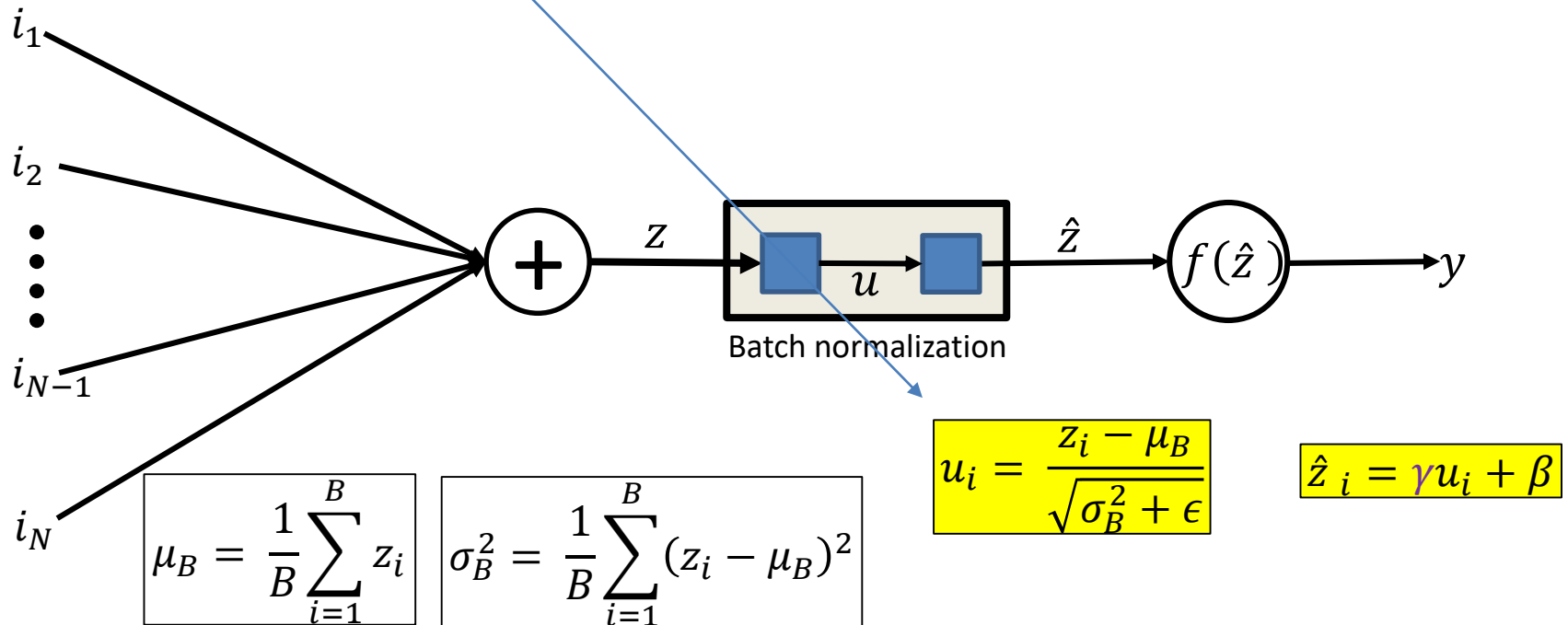


Batch normalization: Backpropagation



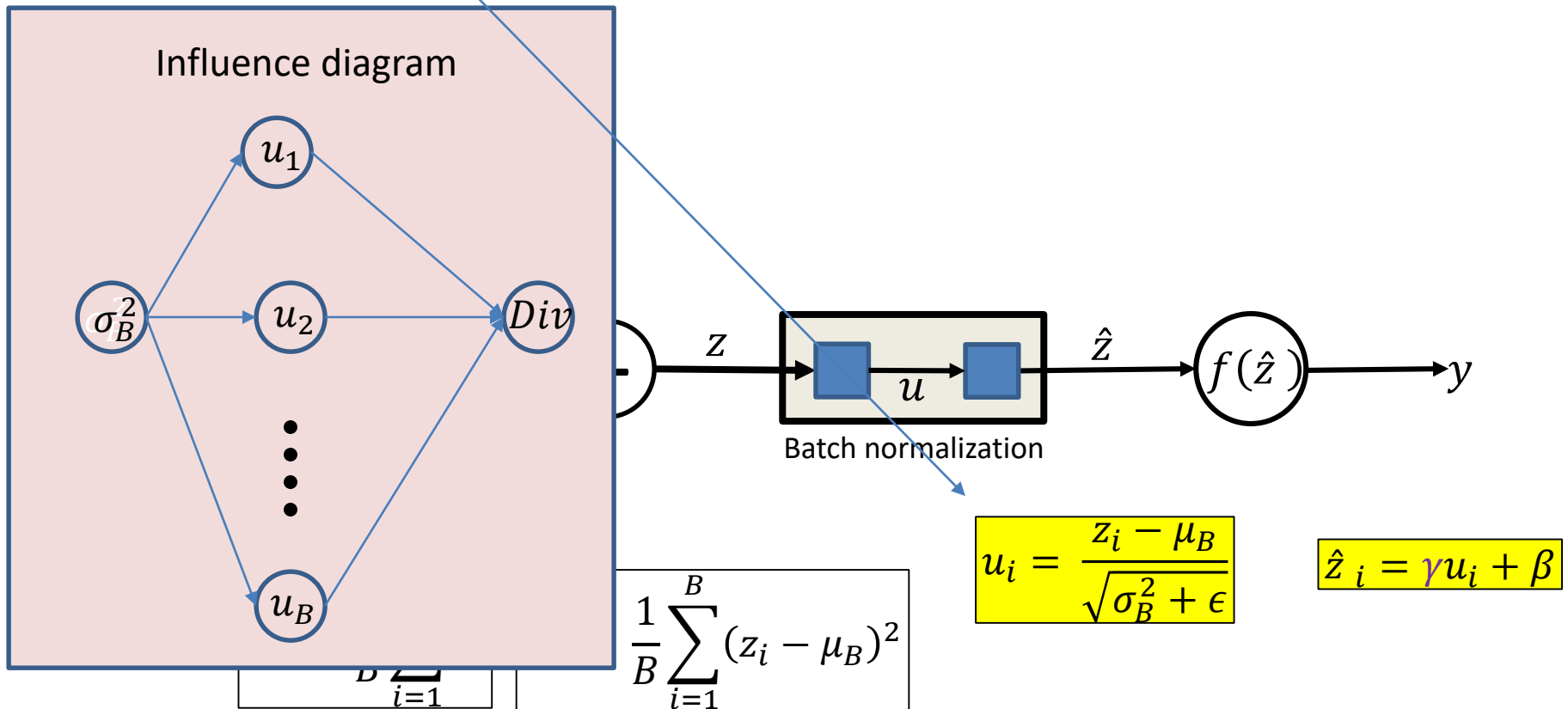
Batch normalization: Backpropagation

$$\frac{\partial \text{Div}}{\partial \sigma_B^2} = \sum_{i=1}^B \frac{\partial \text{Div}}{\partial u_i} (z_i - \mu_B) \cdot \frac{-1}{2} (\sigma_B^2 + \epsilon)^{-3/2}$$



Batch normalization: Backpropagation

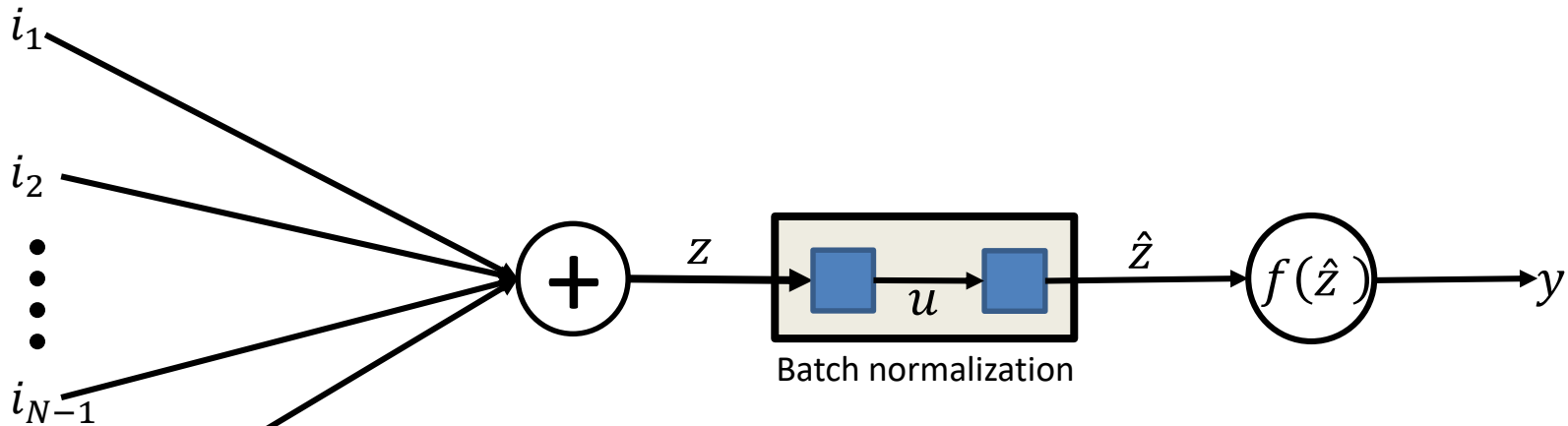
$$\frac{\partial \text{Div}}{\partial \sigma_B^2} = \sum_{i=1}^B \frac{\partial \text{Div}}{\partial u_i} (z_i - \mu_B) \cdot \frac{-1}{2} (\sigma_B^2 + \epsilon)^{-3/2}$$



Batch normalization: Backpropagation

$$\frac{\partial \text{Div}}{\partial \sigma_B^2} = \sum_{i=1}^B \frac{\partial \text{Div}}{\partial u_i} (z_i - \mu_B) \cdot \frac{-1}{2} (\sigma_B^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \text{Div}}{\partial \mu_B} = \left(\sum_{i=1}^B \frac{\partial \text{Div}}{\partial u_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \frac{\partial \text{Div}}{\partial \sigma_B^2} \cdot \frac{\sum_{i=1}^B -2(z_i - \mu_B)}{B}$$



$$\mu_B = \frac{1}{B} \sum_{i=1}^B z_i$$

$$\sigma_B^2 = \frac{1}{B} \sum_{i=1}^B (z_i - \mu_B)^2$$

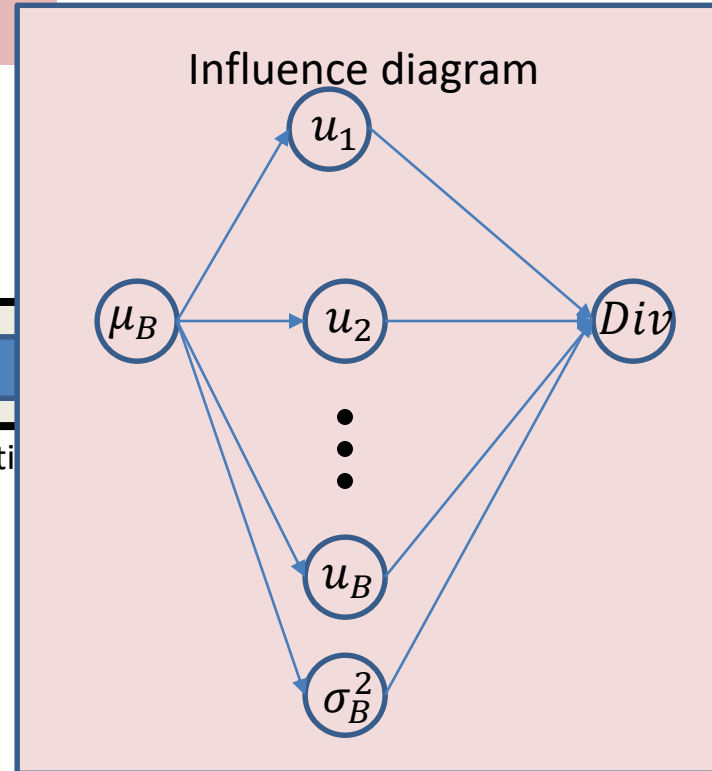
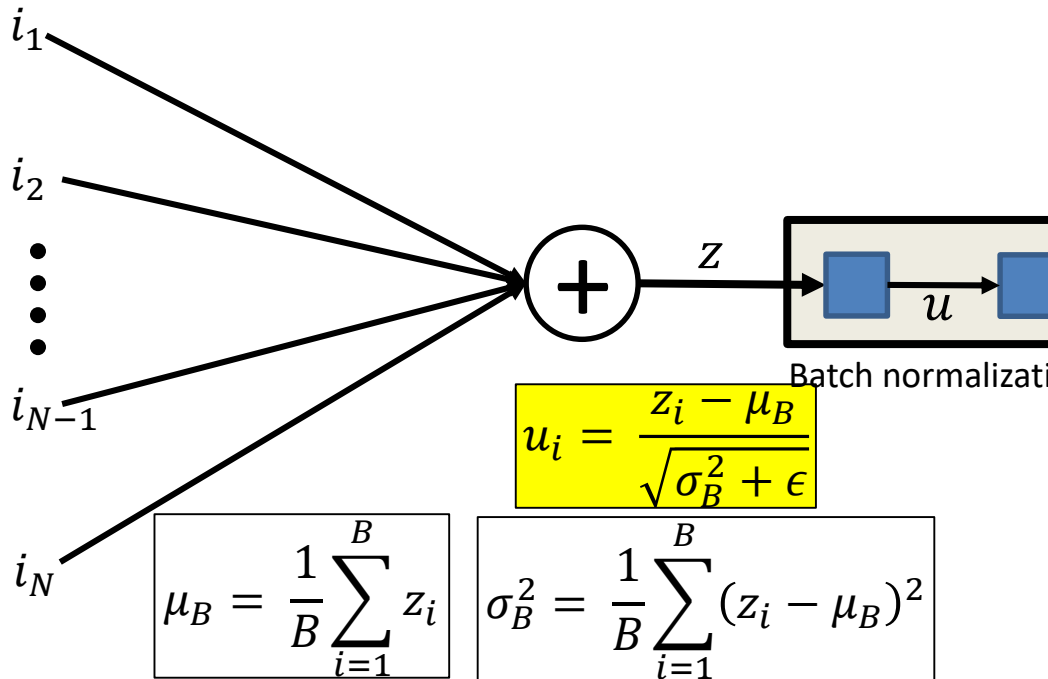
$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\hat{z}_i = \gamma u_i + \beta$$

Batch normalization: Backpropagation

$$\frac{\partial \text{Div}}{\partial \sigma_B^2} = \sum_{i=1}^B \frac{\partial \text{Div}}{\partial u_i} (z_i - \mu_B) \cdot \frac{-1}{2} (\sigma_B^2 + \epsilon)^{-3/2}$$

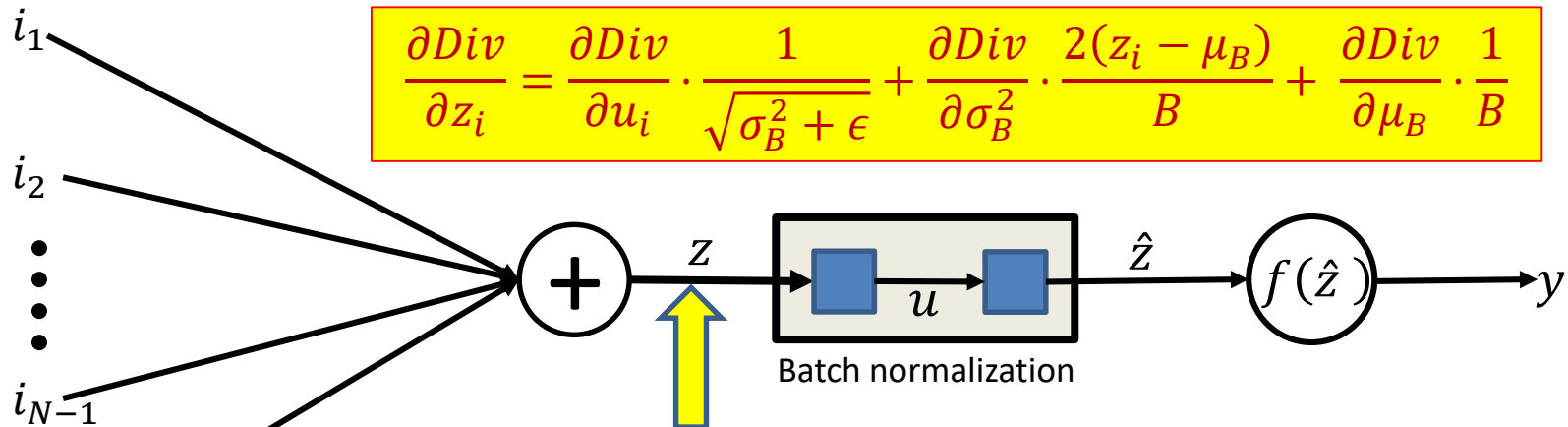
$$\frac{\partial \text{Div}}{\partial \mu_B} = \left(\sum_{i=1}^B \frac{\partial \text{Div}}{\partial u_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \frac{\partial \text{Div}}{\partial \sigma_B^2} \cdot \frac{\sum_{i=1}^B -2(z_i - \mu_B)}{B}$$



Batch normalization: Backpropagation

$$\frac{\partial \text{Div}}{\partial \sigma_B^2} = \sum_{i=1}^B \frac{\partial \text{Div}}{\partial u_i} (z_i - \mu_B) \cdot \frac{-1}{2} (\sigma_B^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \text{Div}}{\partial \mu_B} = \left(\sum_{i=1}^B \frac{\partial \text{Div}}{\partial u_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \frac{\partial \text{Div}}{\partial \sigma_B^2} \cdot \frac{\sum_{i=1}^B -2(z_i - \mu_B)}{B}$$



$$\frac{\partial \text{Div}}{\partial z_i} = \frac{\partial \text{Div}}{\partial u_i} \cdot \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial \text{Div}}{\partial \sigma_B^2} \cdot \frac{2(z_i - \mu_B)}{B} + \frac{\partial \text{Div}}{\partial \mu_B} \cdot \frac{1}{B}$$

$$\mu_B = \frac{1}{B} \sum_{i=1}^B z_i$$

$$\sigma_B^2 = \frac{1}{B} \sum_{i=1}^B (z_i - \mu_B)^2$$

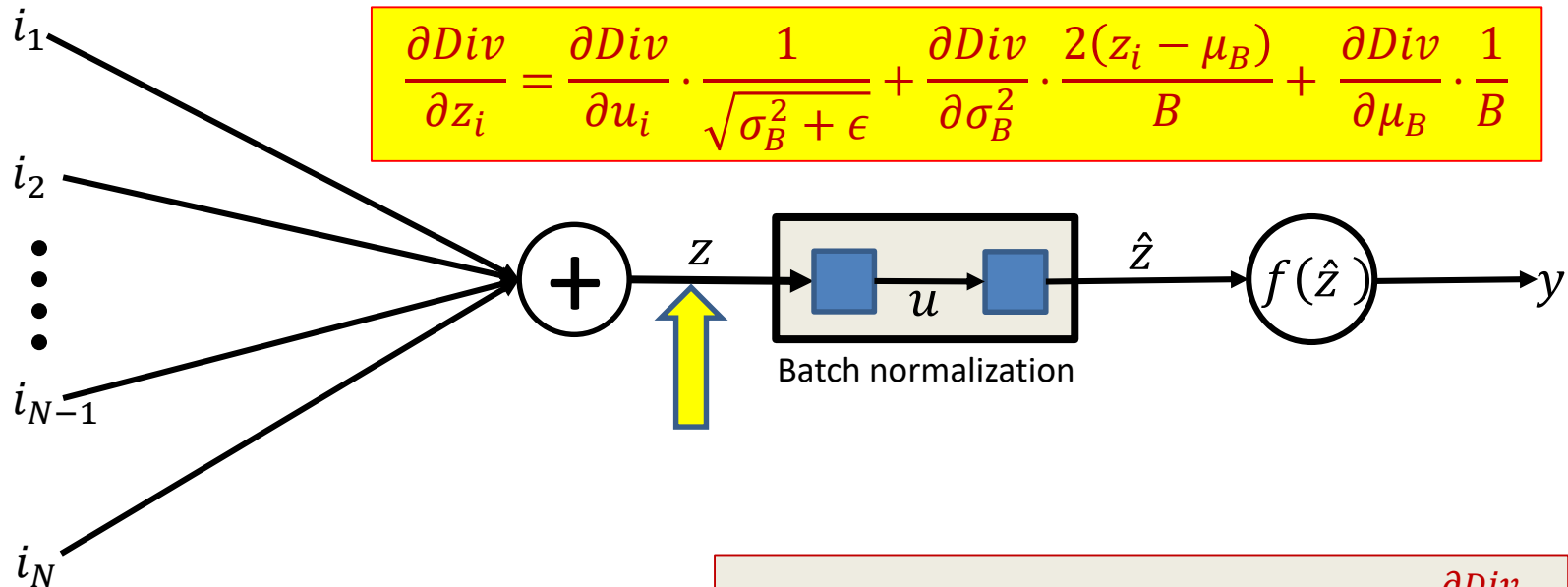
$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\hat{z}_i = \gamma u_i + \beta$$

Batch normalization: Backpropagation

$$\frac{\partial \text{Div}}{\partial \sigma_B^2} = \sum_{i=1}^B \frac{\partial \text{Div}}{\partial u_i} (z_i - \mu_B) \cdot \frac{-1}{2} (\sigma_B^2 + \epsilon)^{-3/2}$$

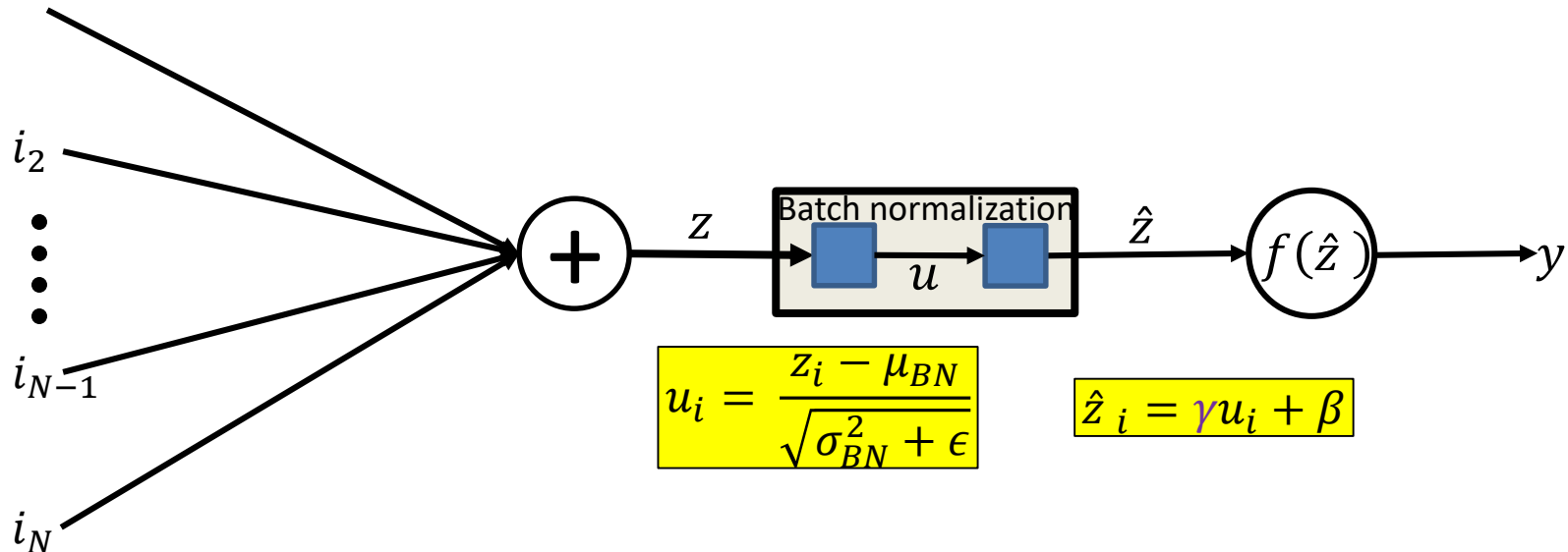
$$\frac{\partial \text{Div}}{\partial \mu_B} = \left(\sum_{i=1}^B \frac{\partial \text{Div}}{\partial u_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \frac{\partial \text{Div}}{\partial \sigma_B^2} \cdot \frac{\sum_{i=1}^B -2(z_i - \mu_B)}{B}$$



$$\frac{\partial \text{Div}}{\partial z_i} = \frac{\partial \text{Div}}{\partial u_i} \cdot \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial \text{Div}}{\partial \sigma_B^2} \cdot \frac{2(z_i - \mu_B)}{B} + \frac{\partial \text{Div}}{\partial \mu_B} \cdot \frac{1}{B}$$

The rest of backprop continues from $\frac{\partial \text{Div}}{\partial z_i}$

Batch normalization: Inference

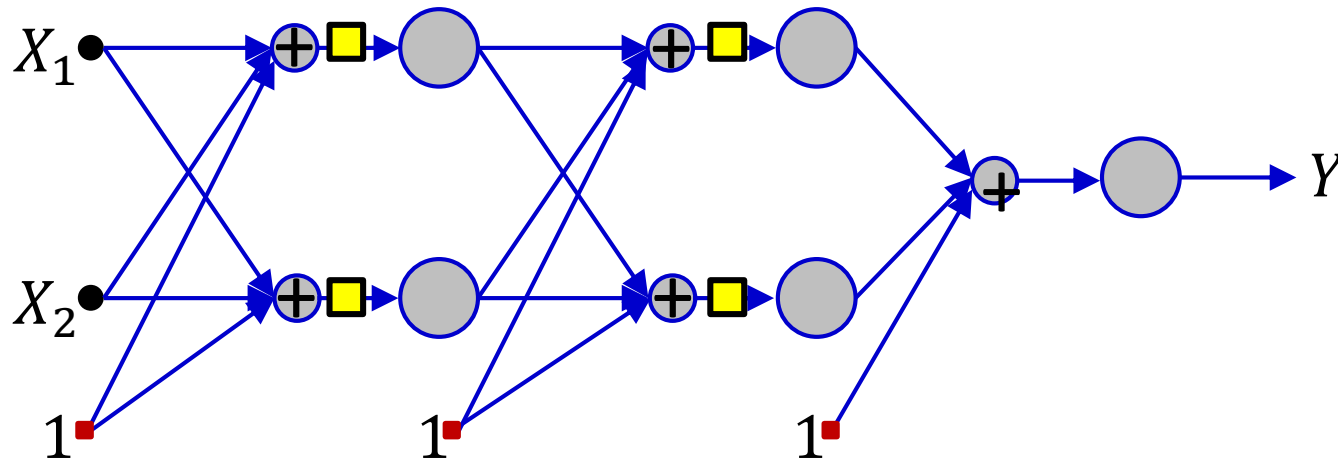


- On test data, BN requires μ_B and σ_B^2 .
- We will use the *average over all training minibatches*

$$\mu_{BN} = \frac{1}{N_{batches}} \sum_{batch} \mu_B(batch)$$
$$\sigma_{BN}^2 = \frac{B}{(B-1)N_{batches}} \sum_{batch} \sigma_B^2(batch)$$

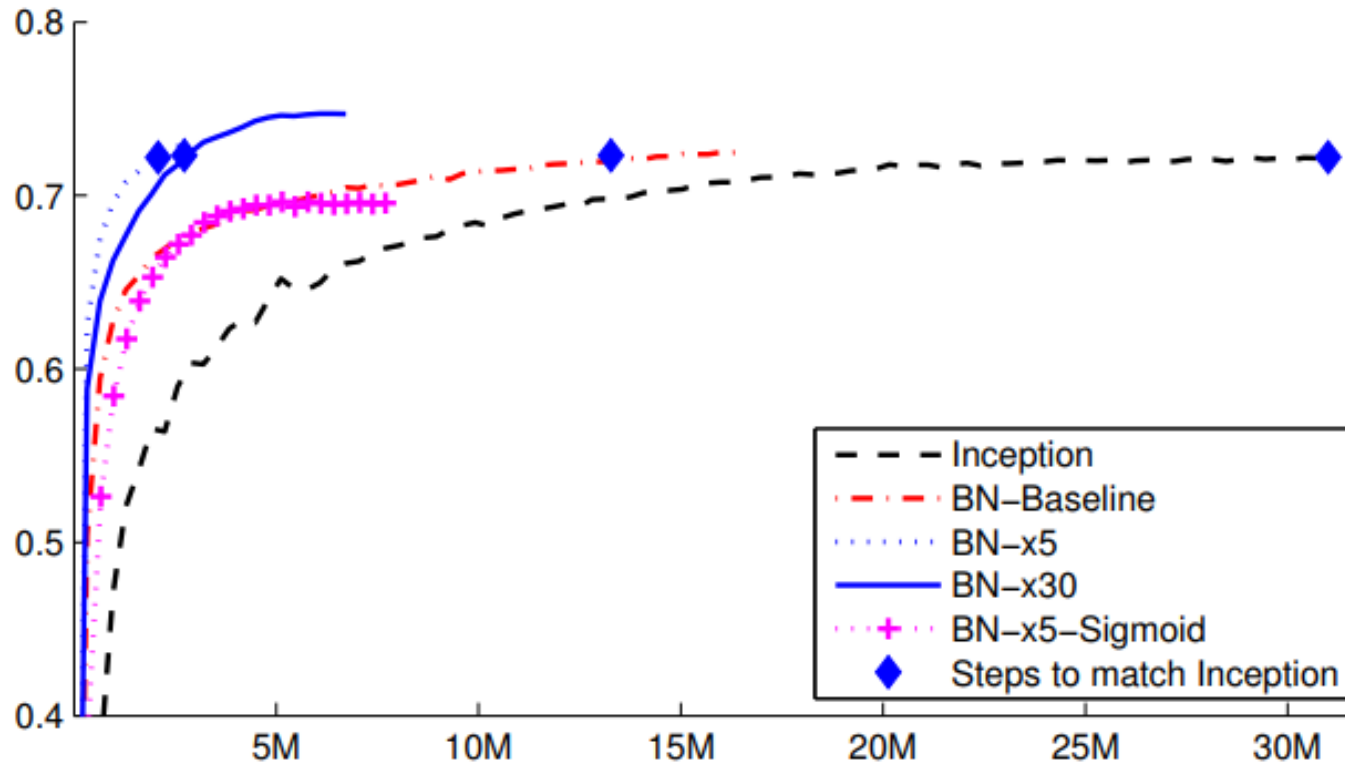
- Note: these are *neuron-specific*
 - $\mu_B(batch)$ and $\sigma_B^2(batch)$ here are obtained from the *final converged network*
 - The $B/(B-1)$ term gives us an unbiased estimator for the variance

Batch normalization



- Batch normalization may only be applied to *some* layers
 - Or even only selected neurons in the layer
- Improves both convergence rate and neural network performance
 - Anecdotal evidence that BN eliminates the need for dropout
 - To get maximum benefit from BN, learning rates must be increased and learning rate decay can be faster
 - Since the data generally remain in the high-gradient regions of the activations
 - Also needs better randomization of training data order

Batch Normalization: Typical result

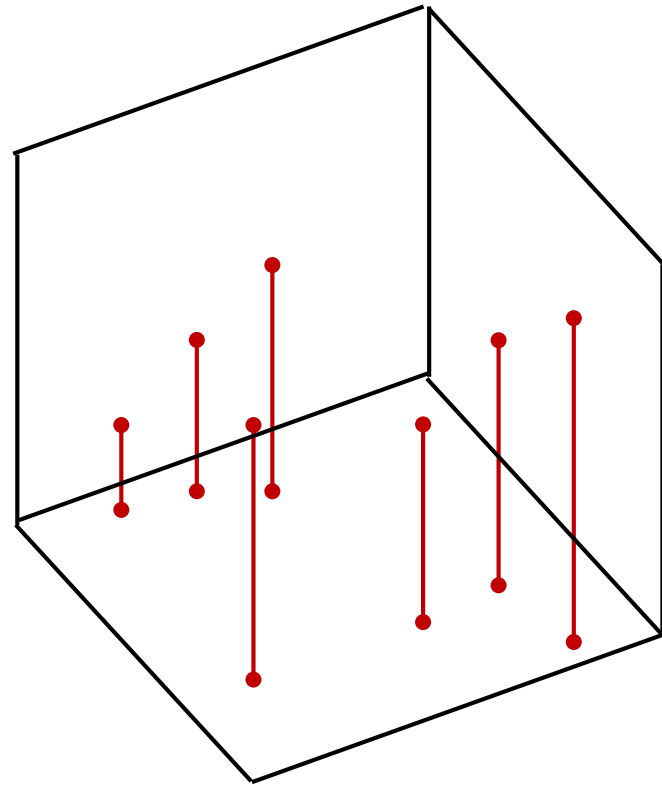
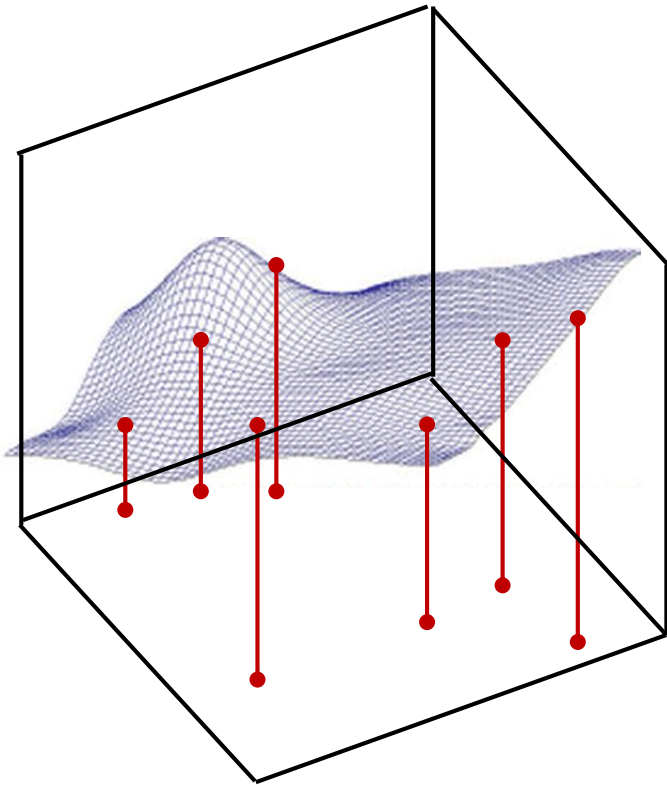


- Performance on Imagenet, from Ioffe and Szegedy, JMLR 2015

The problem of data underspecification

- The figures shown so far were fake news..

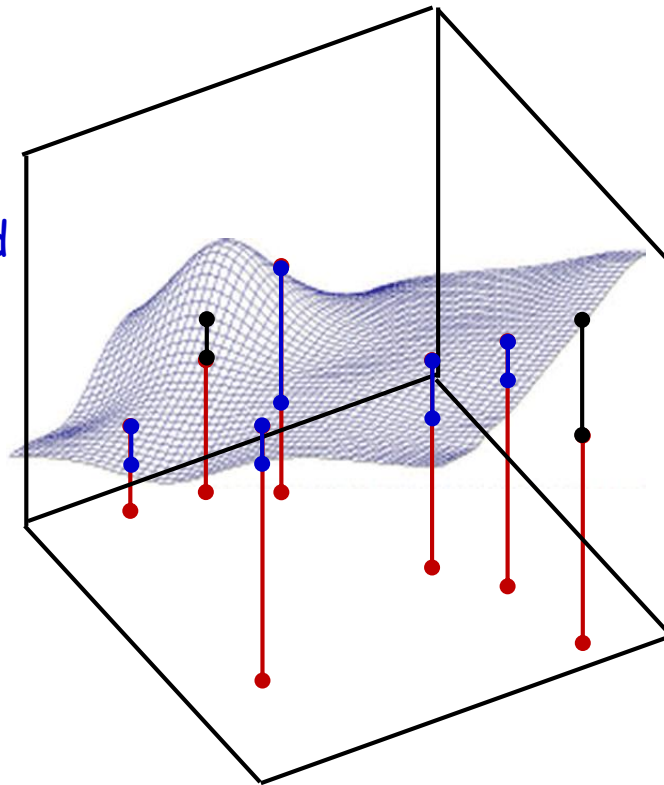
Learning the network



- We attempt to learn an entire function from just a few *snapshots* of it

General approach to training

Blue lines: error when function is *below* desired output

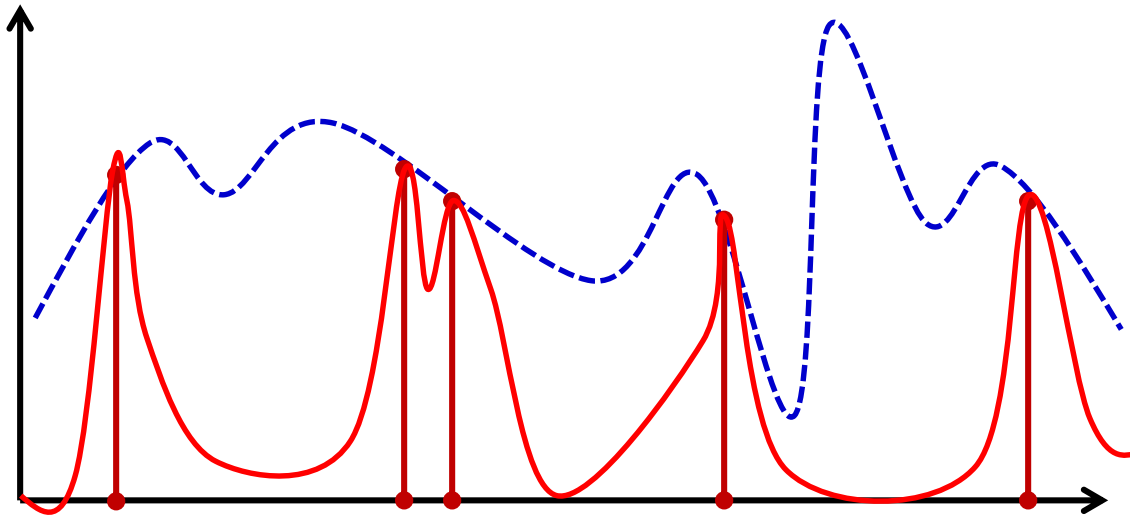


Black lines: error when function is *above* desired output

$$E = \sum_i (y_i - f(\mathbf{x}_i, \mathbf{W}))^2$$

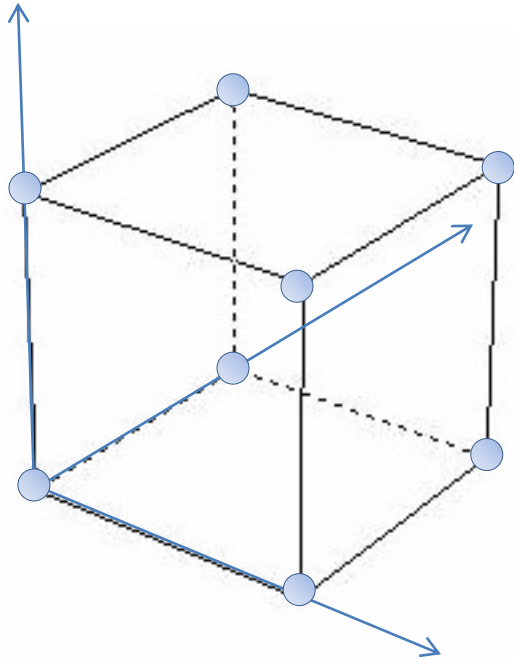
- Define an *error* between the **actual** network output for any parameter value and the *desired* output
 - Error typically defined as the *sum* of the squared error over individual training instances

Overfitting



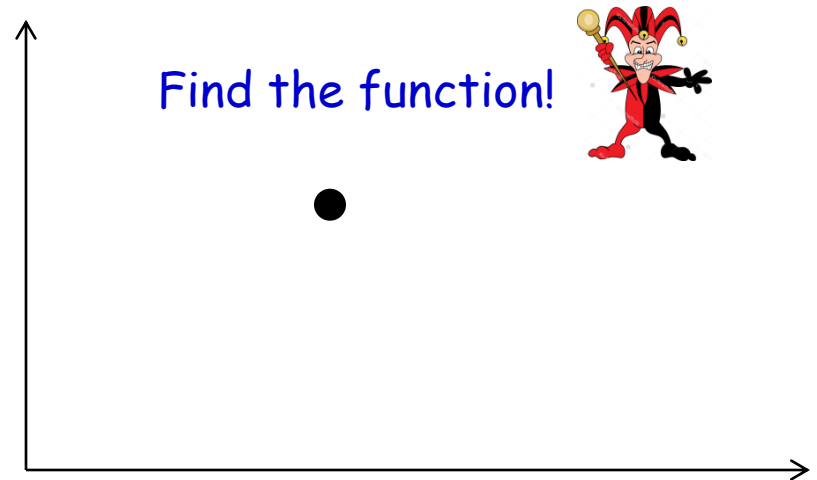
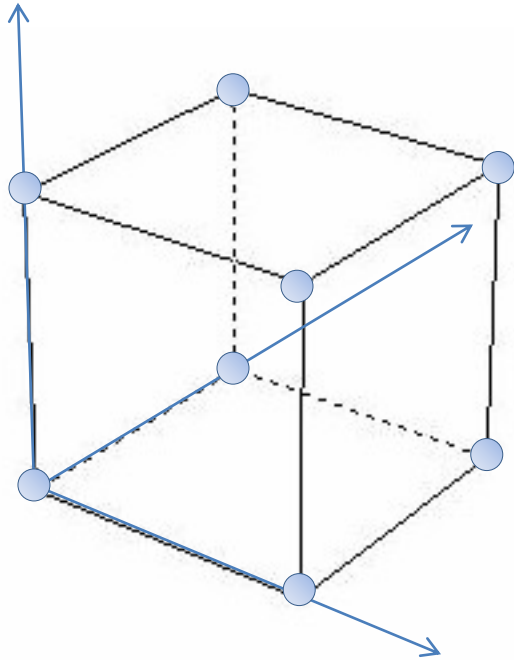
- Problem: Network may just learn the values at the inputs
 - Learn the red curve instead of the dotted blue one
 - Given only the red vertical bars as inputs

Data under-specification



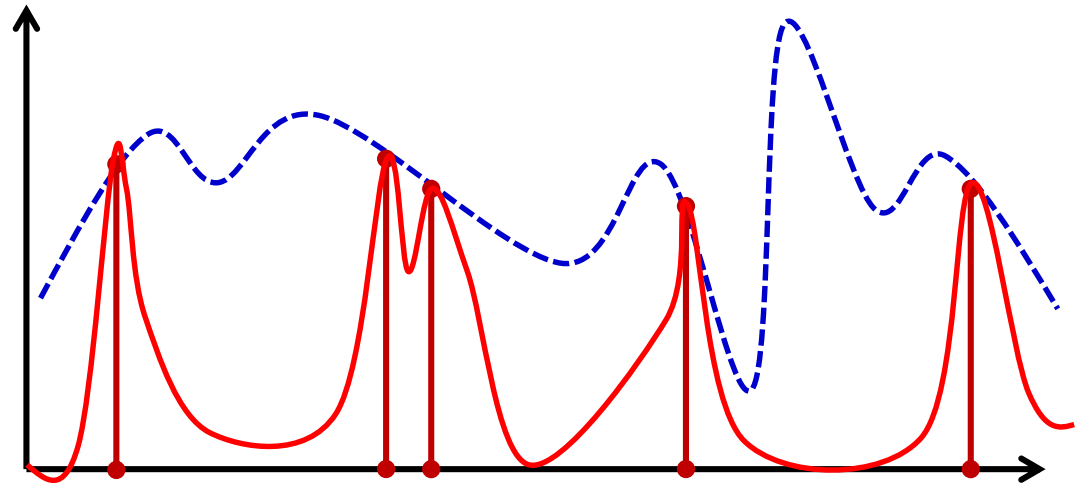
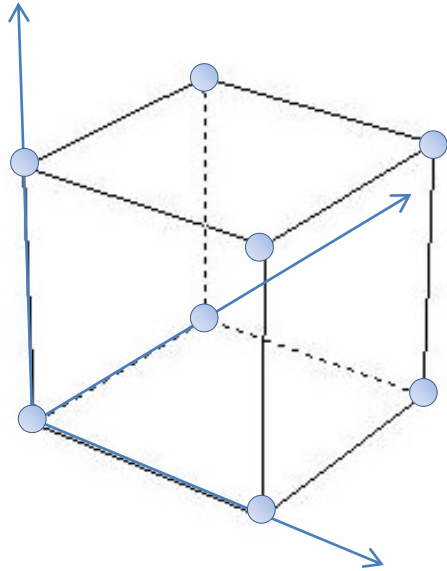
- Consider a binary 100-dimensional input
- There are $2^{100}=10^{30}$ possible inputs
- Complete specification of the function will require specification of 10^{30} output values
- A training set with only 10^{15} training instances will be off by a factor of 10^{15}

Data under-specification in learning



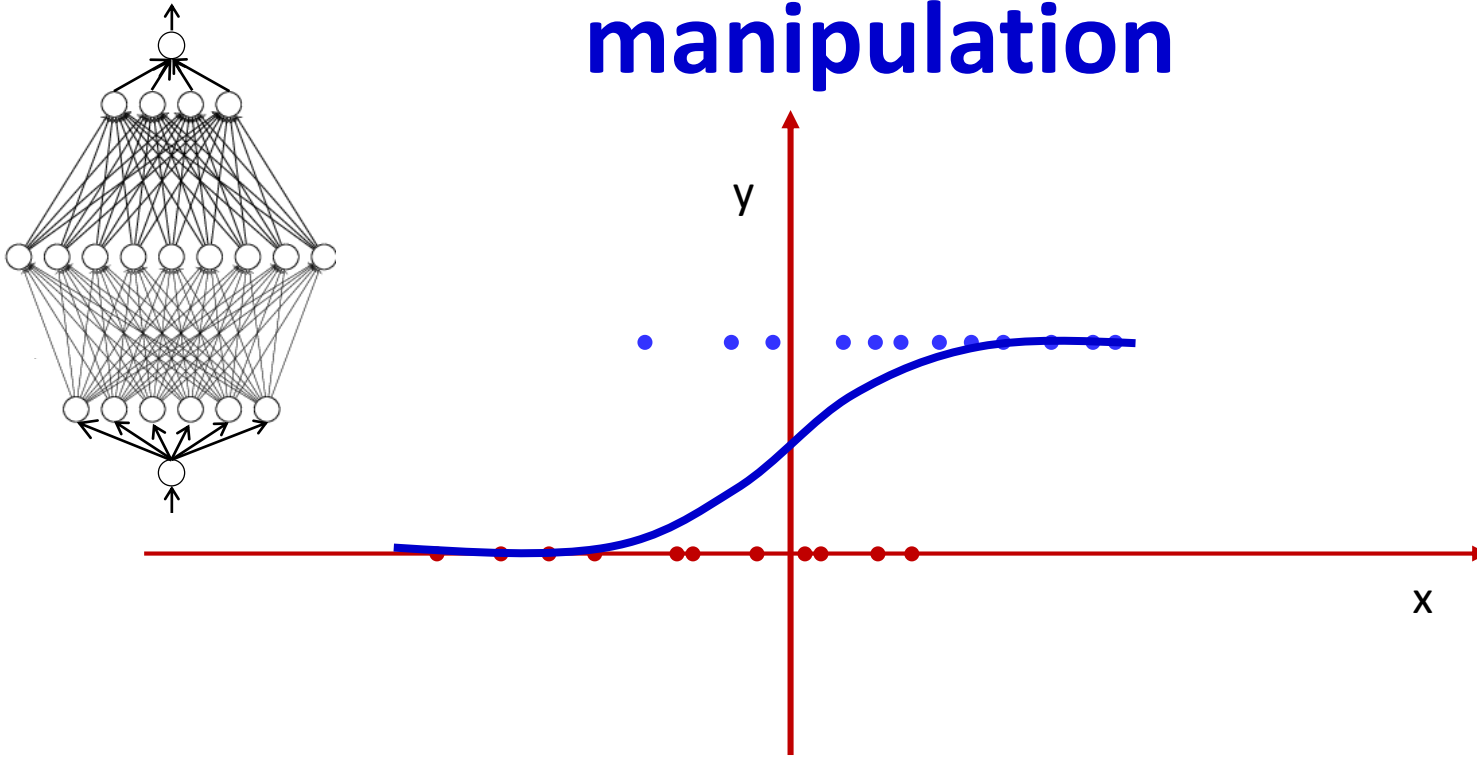
- Consider a binary 100-dimensional input
- There are $2^{100}=10^{30}$ possible inputs
- Complete specification of the function will require specification of 10^{30} output values
- A training set with only 10^{15} training instances will be off by a factor of 10^{15}

Need “smoothing” constraints



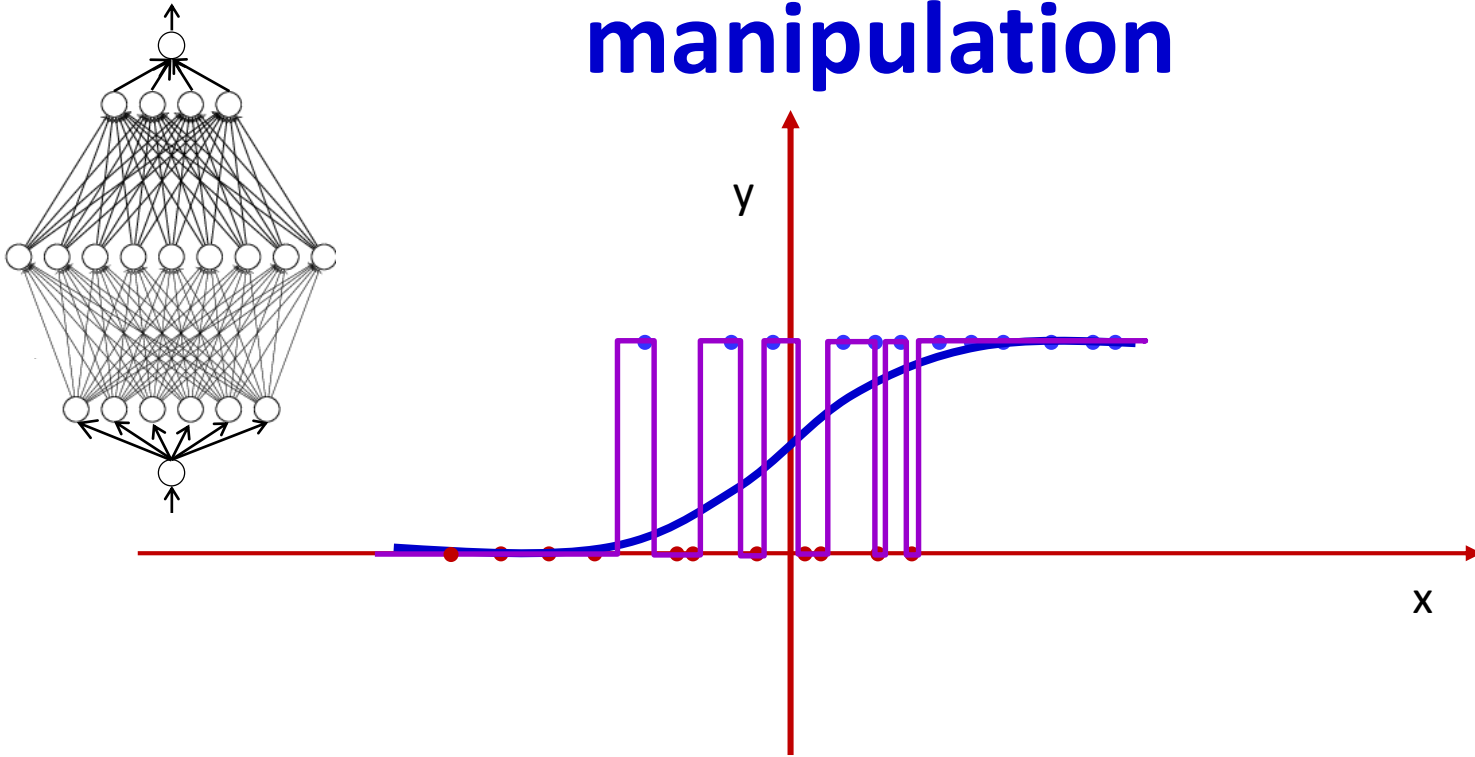
- Need additional constraints that will “fill in” the missing regions acceptably
 - Generalization

Smoothness through weight manipulation



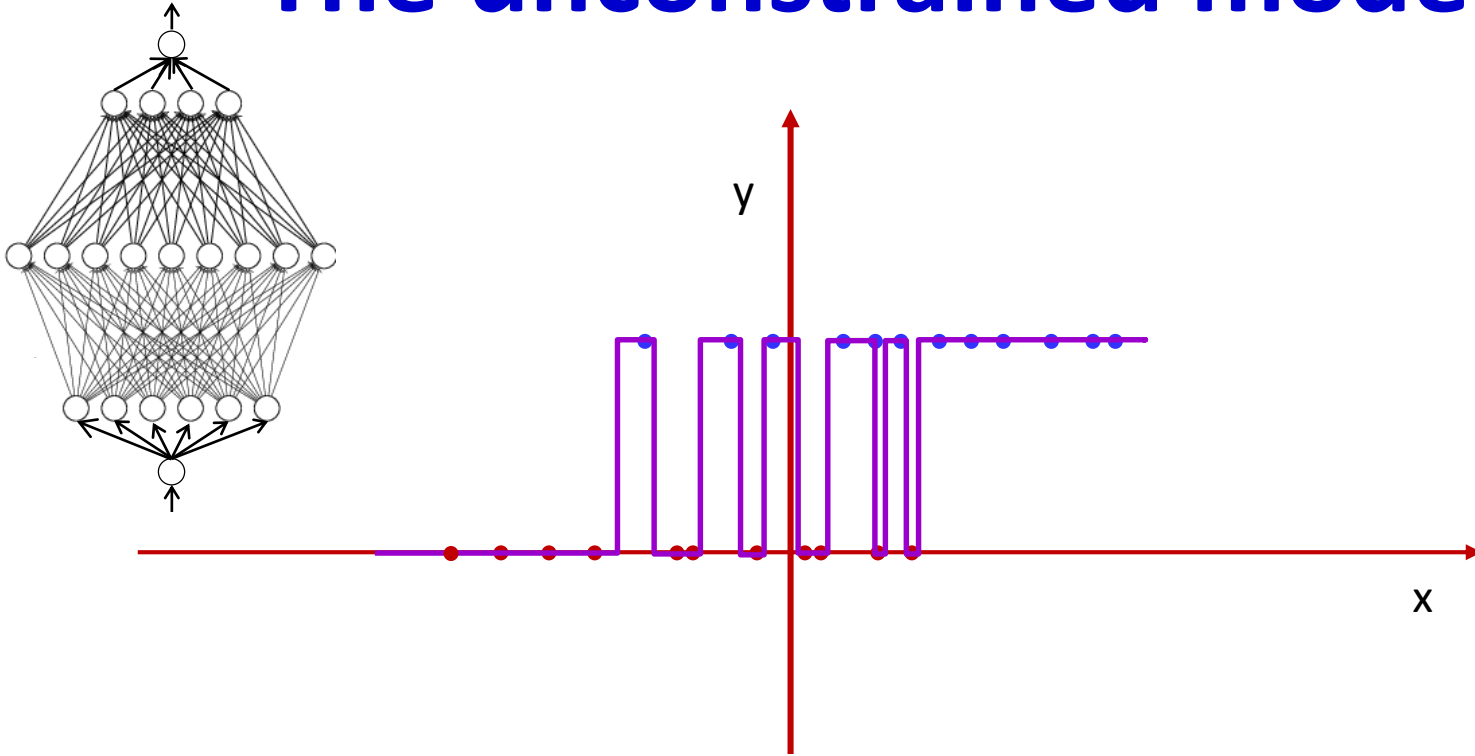
- Illustrative example: Simple binary classifier
 - The “desired” output is generally smooth

Smoothness through weight manipulation



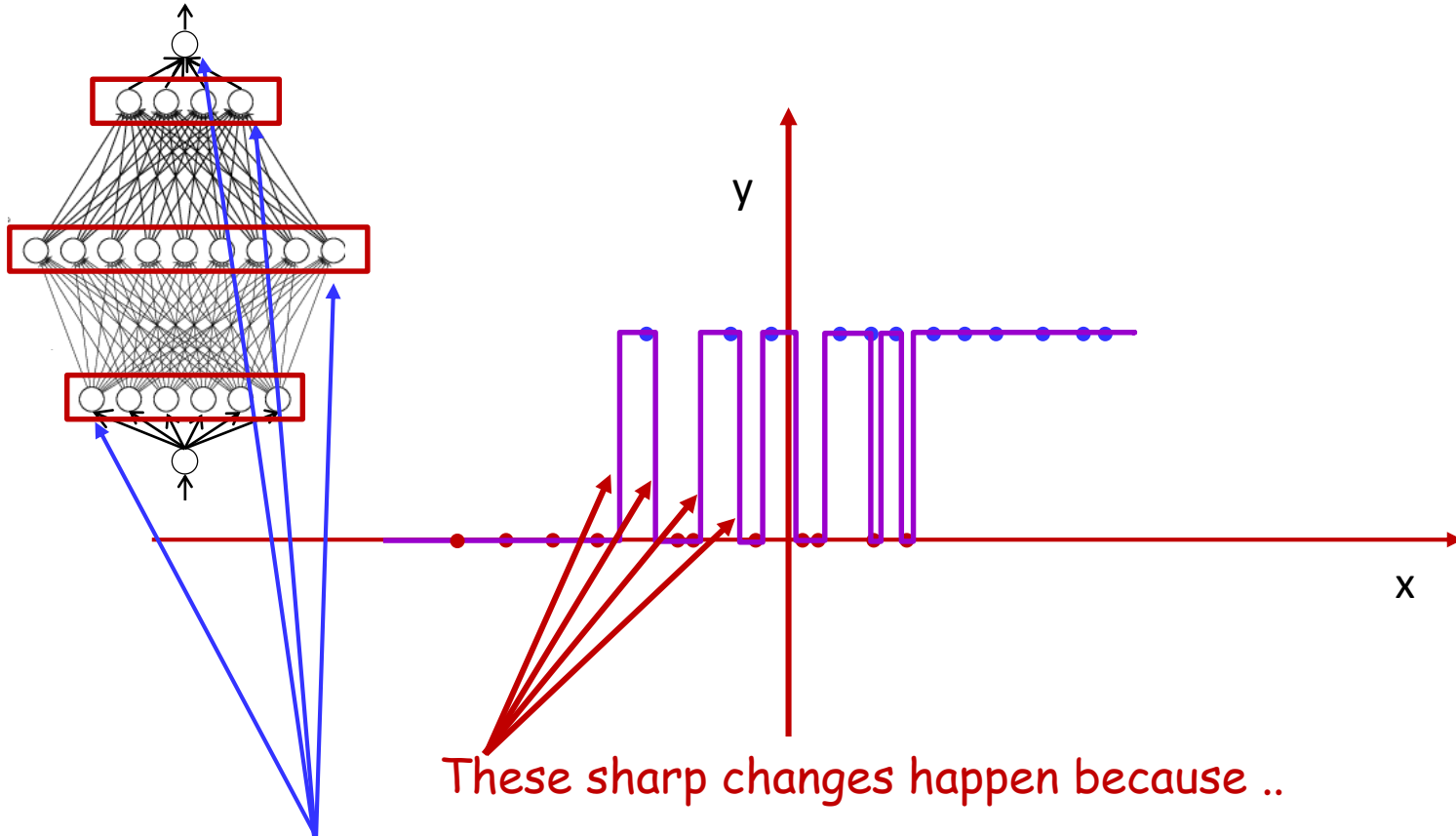
- Illustrative example: Simple binary classifier
 - The “desired” output is generally smooth
 - Capture statistical or average trends
 - An unconstrained model will model individual instances instead

The unconstrained model



- Illustrative example: Simple binary classifier
 - The “desired” output is generally smooth
 - Capture statistical or average trends
 - An unconstrained model will model individual instances instead

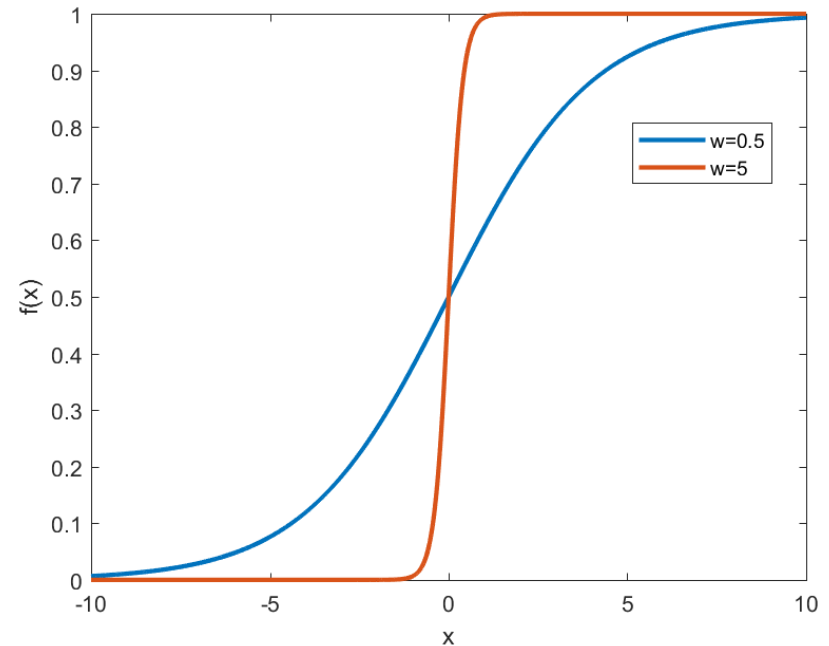
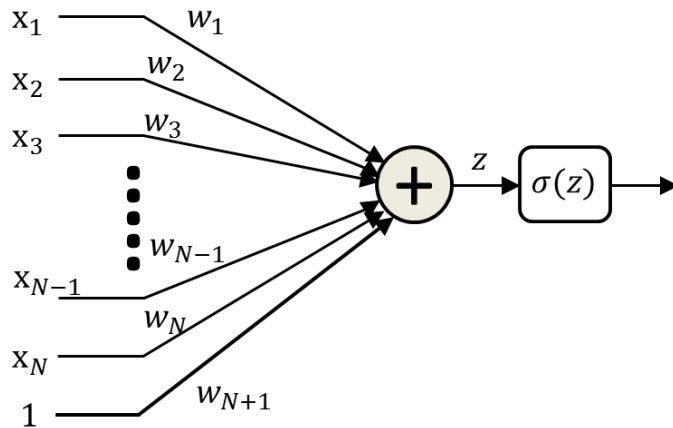
Why overfitting



These sharp changes happen because ..

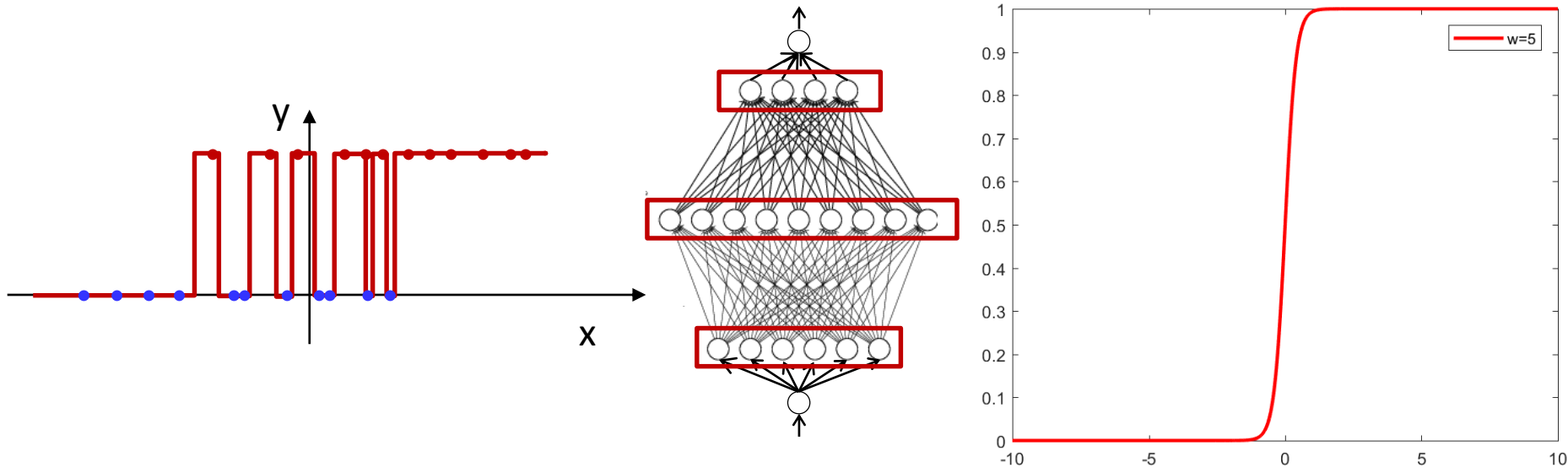
..the perceptrons in the network are individually capable of sharp changes in output

The individual perceptron



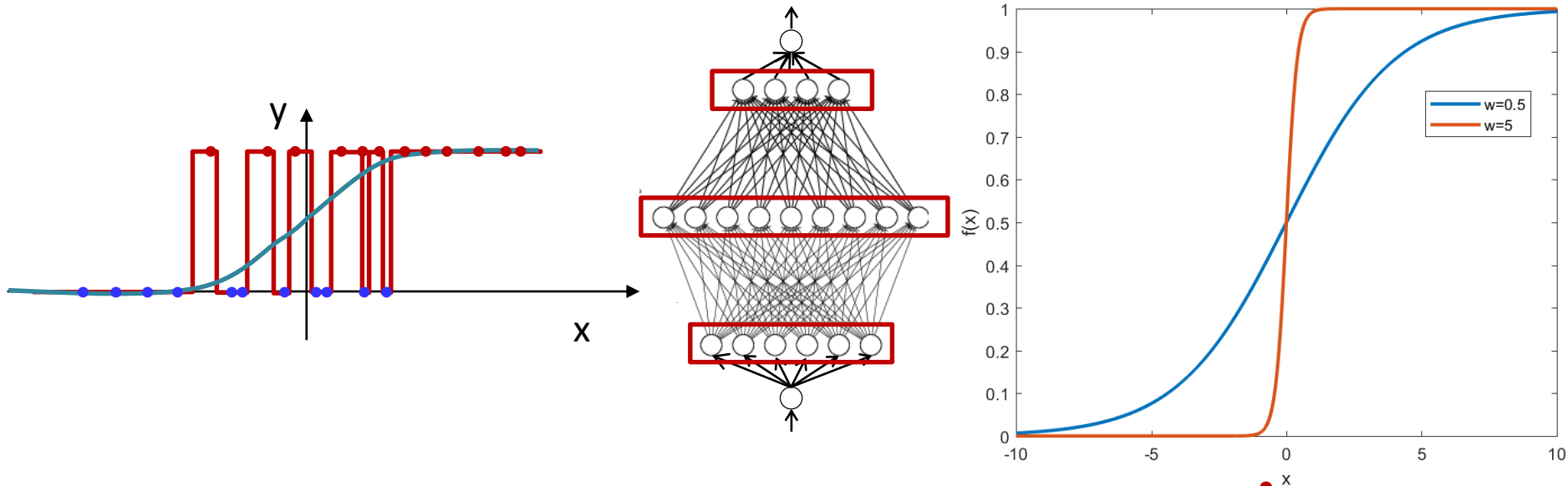
- Using a sigmoid activation
 - As $|w|$ increases, the response becomes steeper

Smoothness through weight manipulation



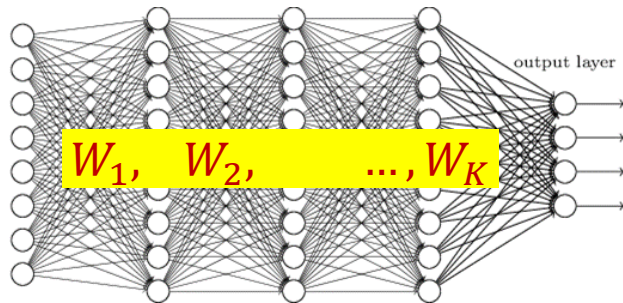
- Steep changes that enable overfitted responses are facilitated by perceptrons with large w

Smoothness through weight manipulation



- Steep changes that enable overfitted responses are facilitated by perceptrons with large w
- Constraining the weights w to be low will force slower perceptrons and smoother output response

Objective function for neural networks



Desired output of network: d_t

Error on i-th training input: $Div(Y_t, d_t; W_1, W_2, \dots, W_K)$

Batch training error:

$$Err(W_1, W_2, \dots, W_K) = \frac{1}{T} \sum_t Div(Y_t, d_t; W_1, W_2, \dots, W_K)$$

- Conventional training: minimize the total error:

$$\widehat{W}_1, \widehat{W}_2, \dots, \widehat{W}_K = \underset{W_1, W_2, \dots, W_K}{\operatorname{argmin}} Err(W_1, W_2, \dots, W_K)$$

Smoothness through weight constraints

- Regularized training: minimize the error while also minimizing the weights

$$L(W_1, W_2, \dots, W_K) = \text{Err}(W_1, W_2, \dots, W_K) + \frac{1}{2} \lambda \sum_k \|W_k\|_2^2$$

$$\hat{W}_1, \hat{W}_2, \dots, \hat{W}_K = \underset{W_1, W_2, \dots, W_K}{\text{argmin}} L(W_1, W_2, \dots, W_K)$$

- λ is the regularization parameter whose value depends on how important it is for us to want to minimize the weights
- Increasing λ assigns greater importance to shrinking the weights
 - Make greater error on training data, to obtain a more acceptable network

Regularizing the weights

$$L(W_1, W_2, \dots, W_K) = \frac{1}{T} \sum_t \text{Div}(Y_t, d_t) + \frac{1}{2} \lambda \sum_k \|W_k\|_2^2$$

- Batch mode:

$$\Delta W_k = \frac{1}{T} \sum_t \nabla_{W_k} \text{Div}(Y_t, d_t)^T + \lambda W_k$$

- SGD:

$$\Delta W_k = \nabla_{W_k} \text{Div}(Y_t, d_t)^T + \lambda W_k$$

- Minibatch:

$$\Delta W_k = \frac{1}{b} \sum_{\tau=t}^{t+b-1} \nabla_{W_k} \text{Div}(Y_\tau, d_\tau)^T + \lambda W_k$$

- Update rule:

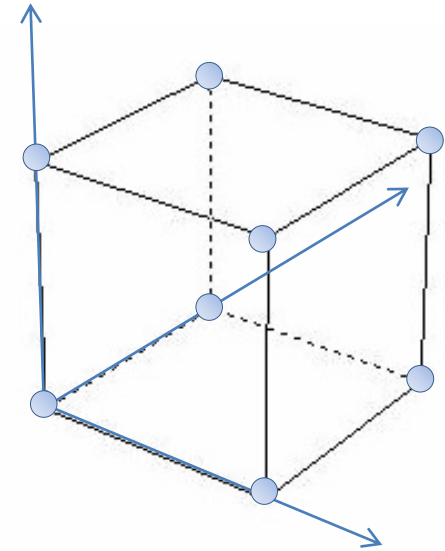
$$W_k \leftarrow W_k - \eta \Delta W_k$$

Incremental Update: Mini-batch update

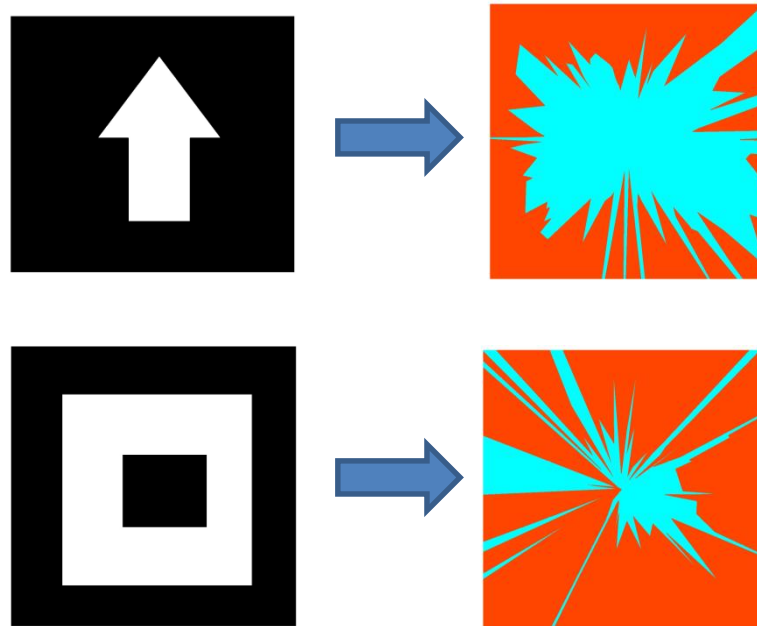
- Given $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights $W_1, W_2, \dots, W_K; j = 0$
- Do:
 - Randomly permute $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
 - For $t = 1:b:T$
 - $j = j + 1$
 - For every layer k :
 - $\Delta W_k = 0$
 - For $t' = t : t+b-1$
 - For every layer k :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

Smoothness through network structure

- MLPs naturally impose constraints
- MLPs are universal approximators
 - Arbitrarily increasing size can give you arbitrarily wiggly functions
 - The function will remain ill-defined on the majority of the space
- *For a given number of parameters deeper networks impose more smoothness than shallow ones*
 - Each layer works on the already smooth surface output by the previous layer

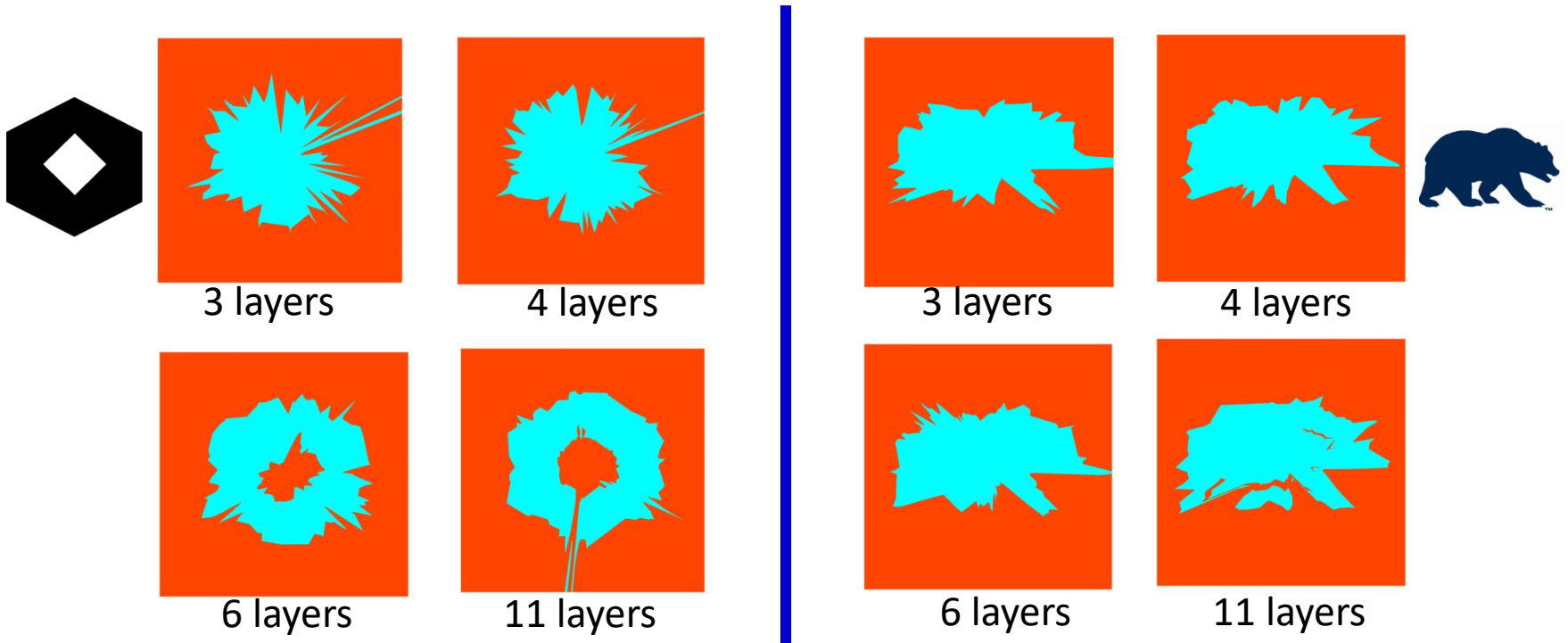


Even when we get it all right



- Typical results (varies with initialization)
- 1000 training points Many orders of magnitude more than you usually get
- All the training tricks known to mankind

But depth and training data help



- Deeper networks seem to learn better, for the same number of total neurons
 - *Implicit smoothness constraints*
 - *As opposed to explicit constraints from more conventional classification models*
- **Similar functions not learnable using more usual pattern-recognition models!!**

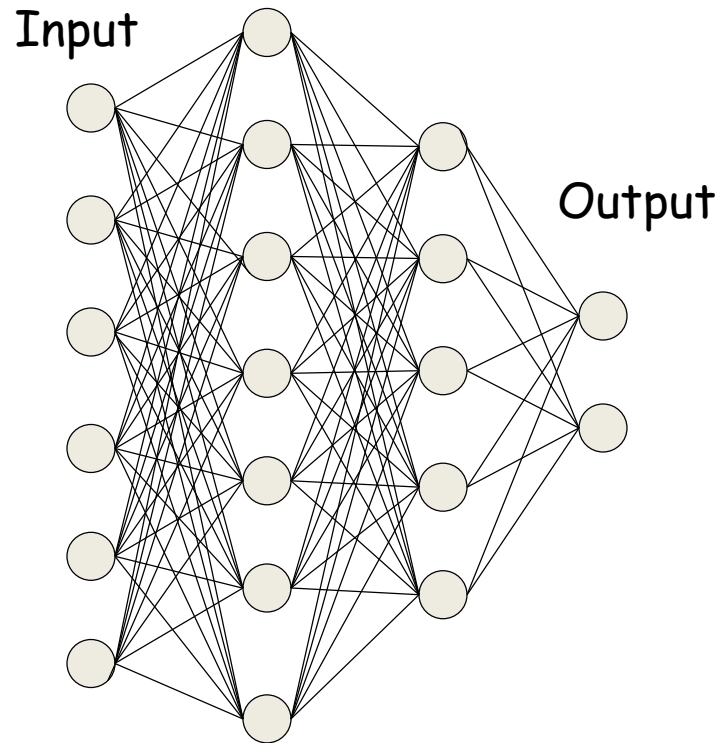
10000 training instances



Regularization..

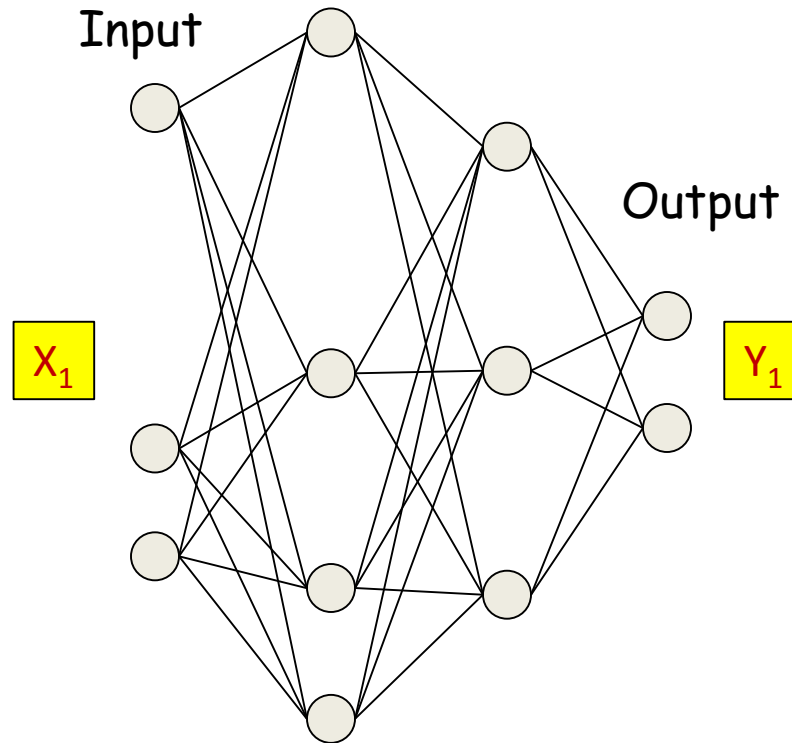
- Other techniques have been proposed to improve the smoothness of the learned function
 - L_1 regularization of network activations
 - Regularizing with added noise..
- Possibly the most influential method has been “dropout”

Dropout



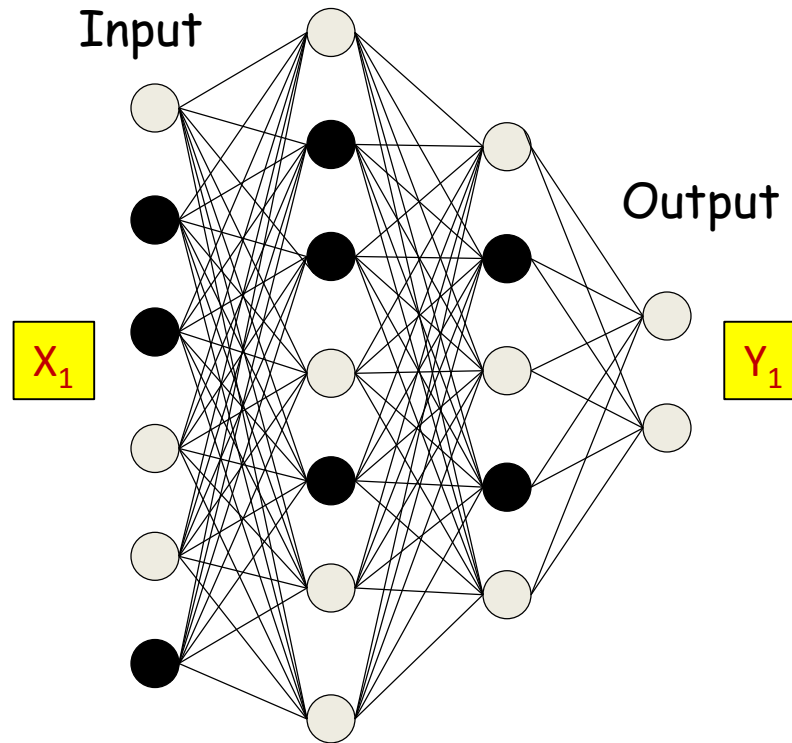
- **During training:** For each input, at each iteration, “turn off” each neuron with a probability $1-\alpha$

Dropout



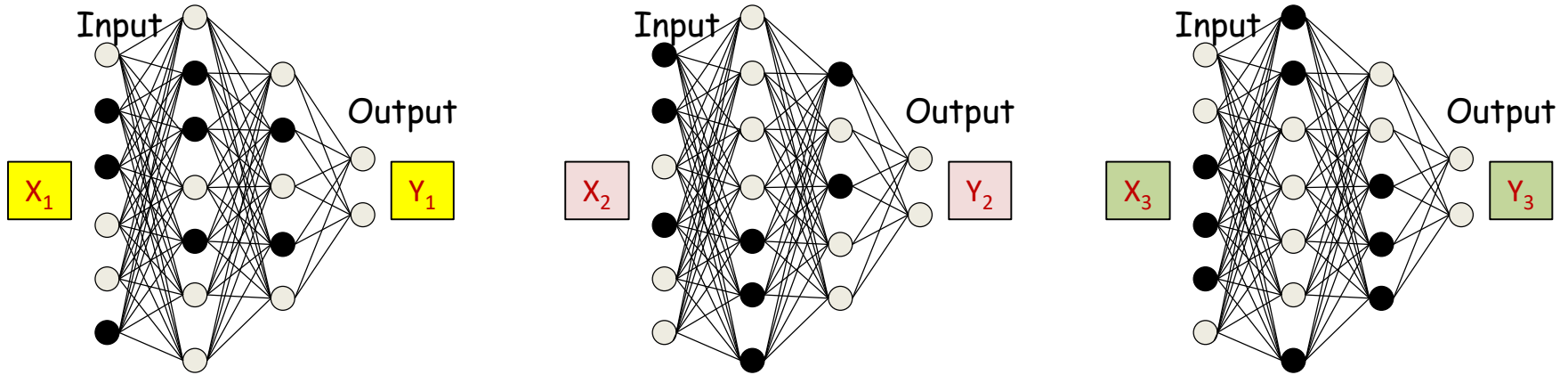
- **During training:** For each input, at each iteration, “turn off” each neuron with a probability $1-\alpha$
 - Also turn off inputs similarly

Dropout



- **During training:** For each input, at each iteration, “turn off” each neuron (including inputs) with a probability $1-\alpha$
 - In practice, set them to 0 according to the success of a Bernoulli random number generator with success probability $1-\alpha$

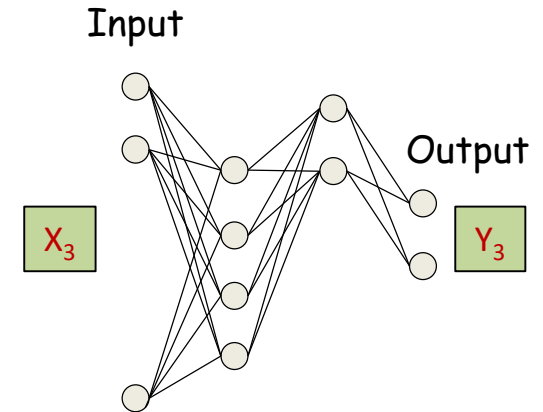
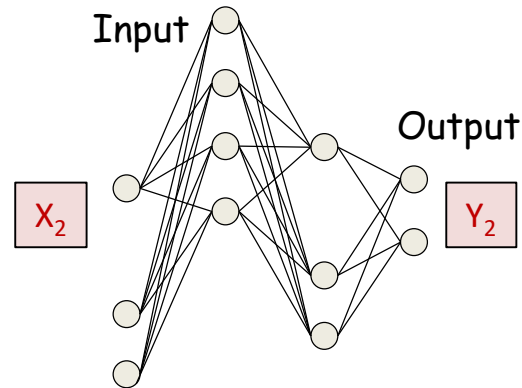
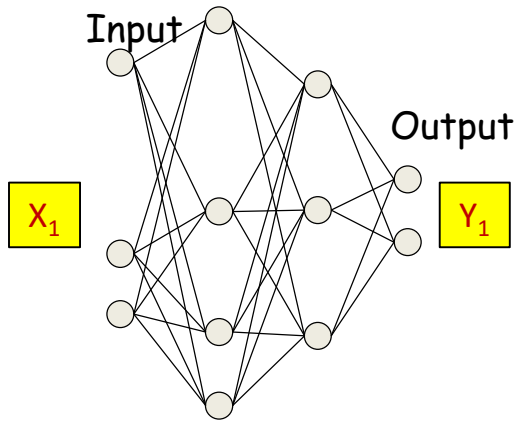
Dropout



The pattern of dropped nodes changes for each input i.e. in every pass through the net

- **During training:** For each input, at each iteration, “turn off” each neuron (including inputs) with a probability $1-\alpha$
 - In practice, set them to 0 according to the success of a Bernoulli random number generator with success probability $1-\alpha$

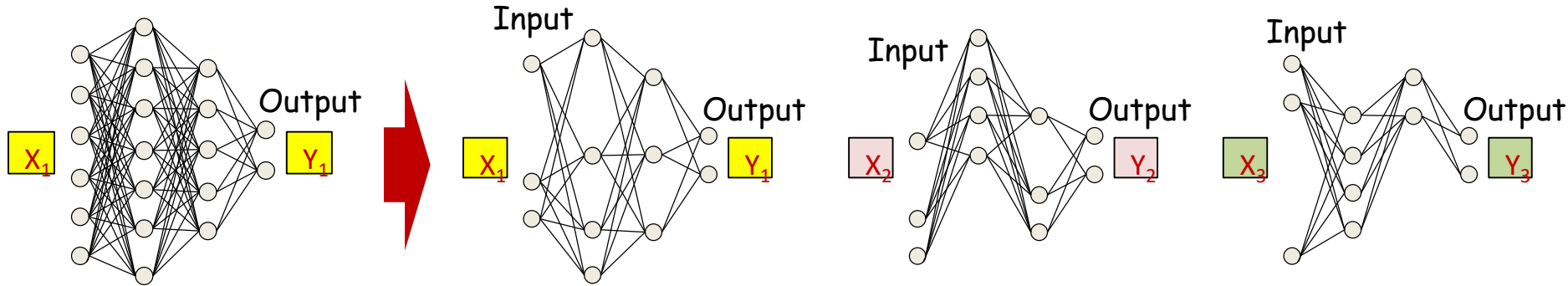
Dropout



The pattern of dropped nodes changes for each input
i.e. in every pass through the net

- **During training:** Backpropagation is effectively performed only over the remaining network
 - The effective network is different for different inputs
 - Gradients are obtained only for the weights and biases *from* “On” nodes *to* “On” nodes
 - For the remaining, the gradient is just 0

Statistical Interpretation



- For a network with a total of N neurons, there are 2^N possible sub-networks
 - Obtained by choosing different subsets of nodes
 - Dropout *samples* over all 2^N possible networks
 - Effective learns a network that *averages* over all possible networks
 - Bagging

The forward pass

- Input: D dimensional vector $\mathbf{x} = [x_j, j = 1 \dots D]$
- Set:
 - $D_0 = D$, is the width of the 0th (input) layer
 - $y_j^{(0)} = x_j, j = 1 \dots D; \quad y_0^{(k=1 \dots N)} = x_0 = 1$
- For layer $k = 1 \dots N$
 - For $j = 1 \dots D_k$

- $z_j^{(k)} = \sum_{i=0}^{N_k} w_{i,j}^{(k)} y_i^{(k-1)} + b_j^{(k)}$
- $y_j^{(k)} = f_k(z_j^{(k)})$
- If ($k = \text{dropout layer}$):
 - $\text{mask}(k,j) = \text{Bernoulli}(\alpha)$
 - If $\text{mask}(k,j)$
 - » $y_j^{(k)} = y_j^{(k)} / \alpha$
 - Else
 - » $y_j^{(k)} = 0$

- Output:
 - $Y = y_j^{(N)}, j = 1 \dots D_N$

Backward Pass

- Output layer (N) :

$$- \frac{\partial Div}{\partial Y_i} = \frac{\partial Div(Y, d)}{\partial y_i^{(N)}}$$

$$- \frac{\partial Div}{\partial z_i^{(k)}} = f'_k \left(z_i^{(k)} \right) \frac{\partial Div}{\partial y_i^{(k)}}$$

- For layer $k = N - 1$ *downto* 0

- For $i = 1 \dots D_k$

- If (not dropout layer OR $mask(k, i)$)

$$- \frac{\partial Div}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial Div}{\partial z_j^{(k+1)}}$$

$$- \frac{\partial Div}{\partial z_i^{(k)}} = f'_k \left(z_i^{(k)} \right) \frac{\partial Div}{\partial y_i^{(k)}}$$

$$- \frac{\partial Div}{\partial w_{ij}^{(k+1)}} = y_j^{(k)} \frac{\partial Div}{\partial z_i^{(k+1)}} \text{ for } j = 1 \dots D_{k+1}$$

- Else

$$- \frac{\partial Div}{\partial z_i^{(k)}} = 0$$

What each neuron computes

- Each neuron actually has the following activation:

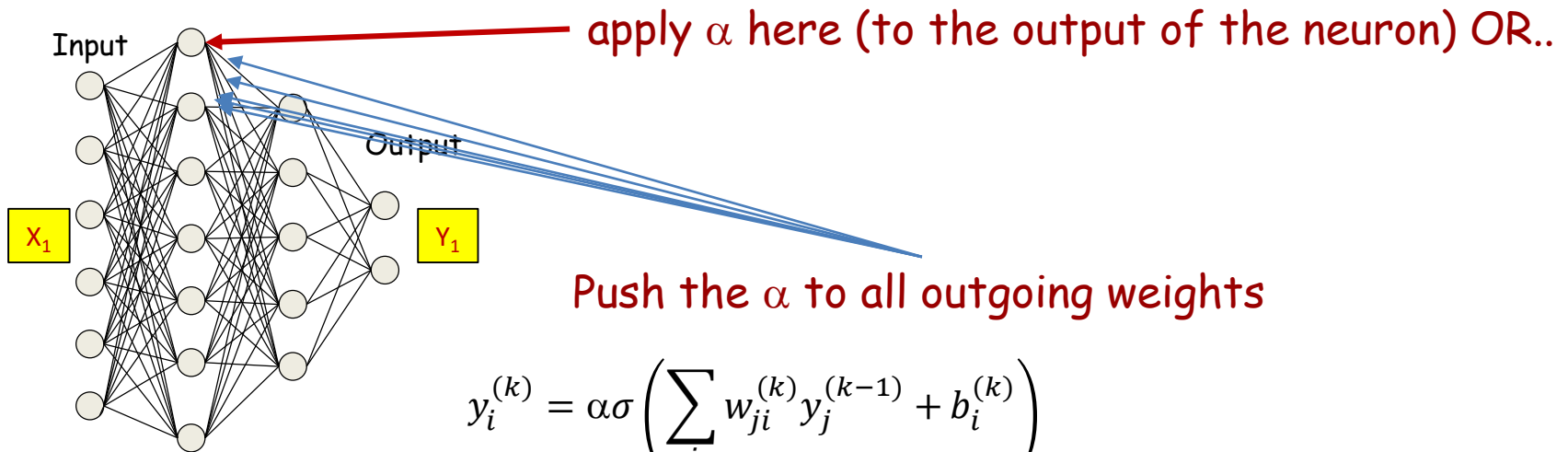
$$y_i^{(k)} = D \sigma \left(\sum_j w_{ji}^{(k)} y_j^{(k-1)} + b_i^{(k)} \right)$$

- Where D is a Bernoulli variable that takes a value 1 with probability α
- D may be switched on or off for individual sub networks, but over the ensemble, the *expected output* of the neuron is

$$y_i^{(k)} = \alpha \sigma \left(\sum_j w_{ji}^{(k)} y_j^{(k-1)} + b_i^{(k)} \right)$$

- During *test* time, we will use the *expected* output of the neuron
 - Which corresponds to the bagged average output
 - Consists of simply scaling the output of each neuron by α

Dropout during test: implementation

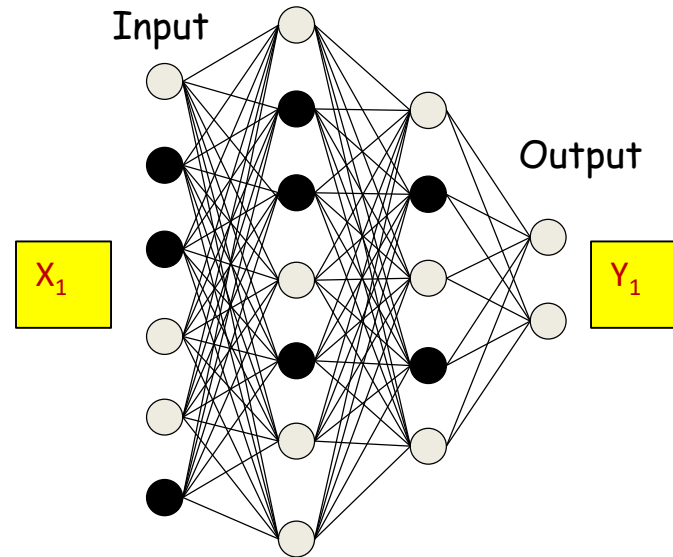


$$\begin{aligned}y_i^{(k)} &= \alpha \sigma \left(\sum_j w_{ji}^{(k)} y_j^{(k-1)} + b_i^{(k)} \right) \\&= \alpha \sigma \left(\sum_j w_{ji}^{(k)} \alpha \sigma \left(\sum_j w_{ji}^{(k-1)} y_j^{(k-2)} + b_i^{(k-1)} \right) + b_i^{(k)} \right) \\&= \alpha \sigma \left(\sum_j \left(\alpha w_{ji}^{(k)} \right) \sigma \left(\sum_j w_{ji}^{(k-1)} y_j^{(k-2)} + b_i^{(k-1)} \right) + b_i^{(k)} \right)\end{aligned}$$

$$W_{test} = \alpha W_{trained}$$

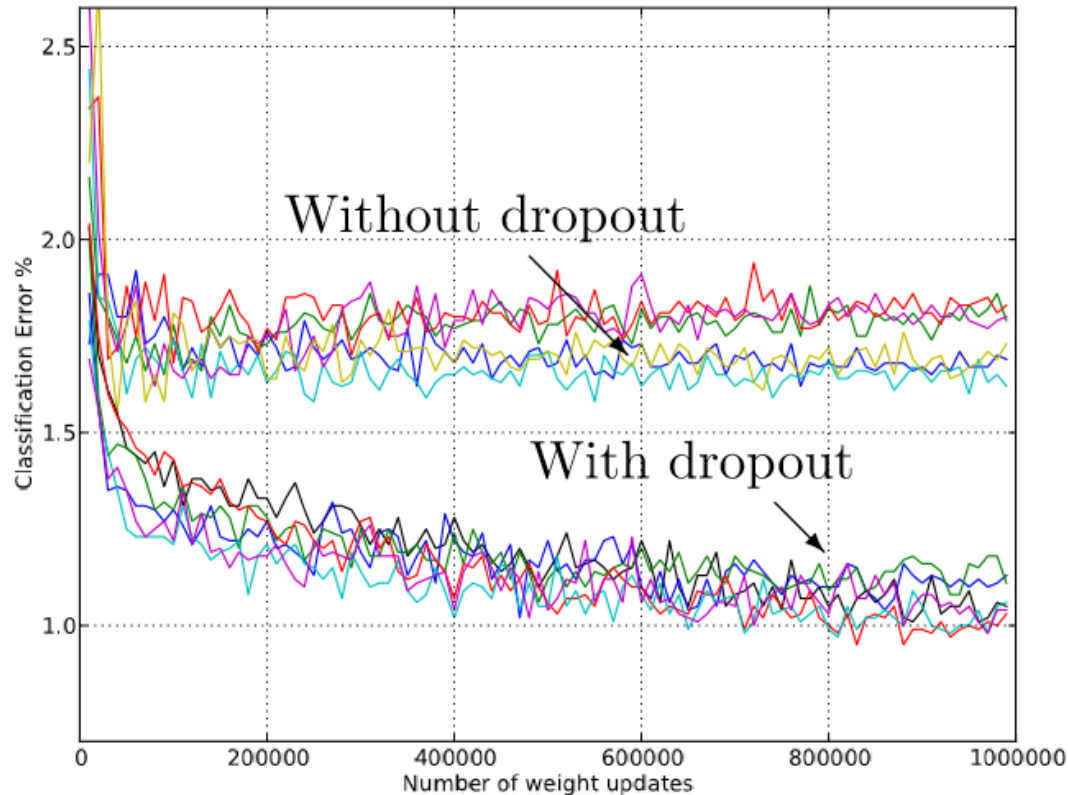
- Instead of multiplying every output by α , multiply all weights by α

Dropout : alternate implementation



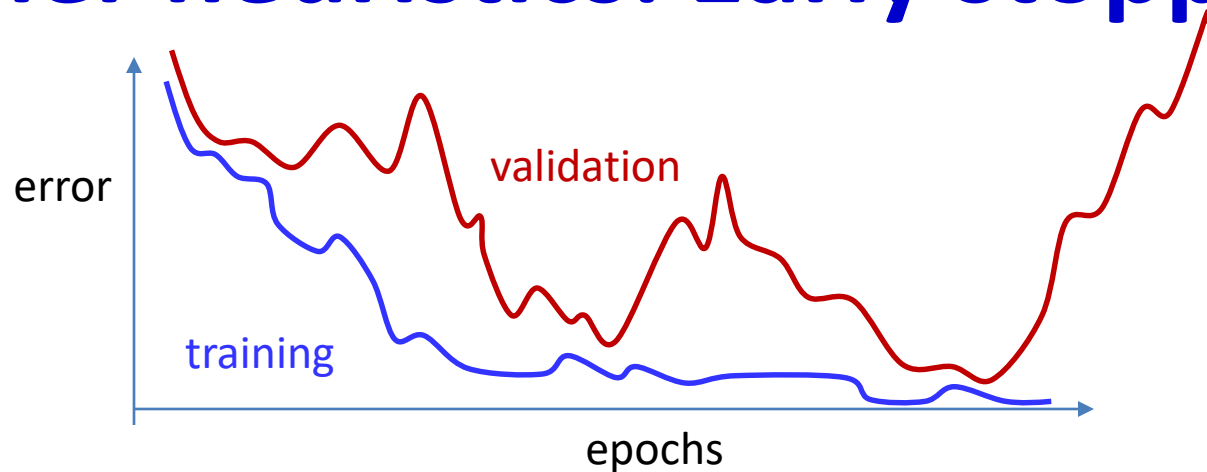
- Alternately, during *training*, replace the activation of all neurons in the network by $\alpha^{-1}\sigma(\cdot)$
 - This does not affect the dropout procedure itself
 - We will use $\sigma(\cdot)$ as the activation during testing, and not modify the weights

Dropout: Typical results



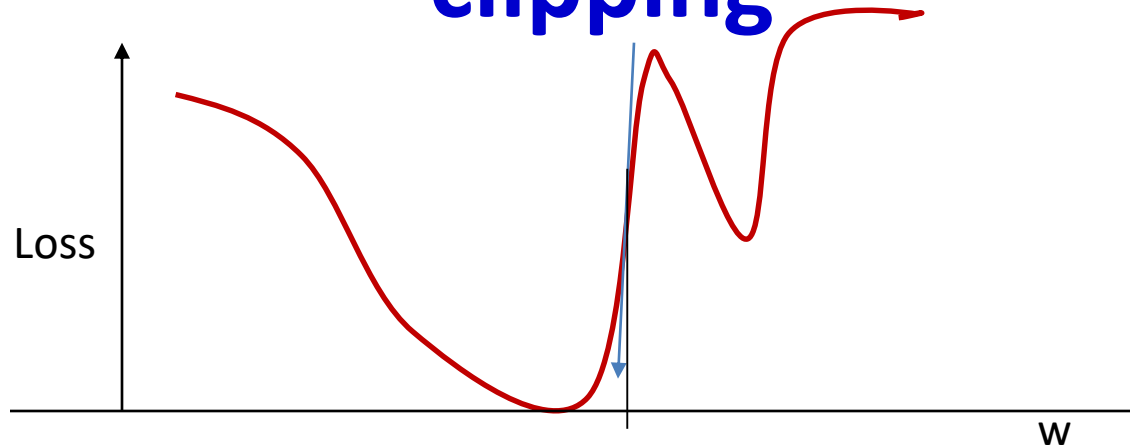
- From Srivastava et al., 2013. Test error for different architectures on MNIST with and without dropout
 - 2-4 hidden layers with 1024-2048 units

Other heuristics: Early stopping



- Continued training can result in severe over fitting to training data
 - Track performance on a held-out validation set
 - Apply one of several early-stopping criterion to terminate training when performance on validation set degrades significantly

Additional heuristics: Gradient clipping



- Often the derivative will be too high
 - When the divergence has a steep slope
 - This can result in instability
- **Gradient clipping**: set a ceiling on derivative value
 - if $\partial_w D > \theta$ then $\partial_w D = \theta$*
 - Typical θ value is 5

Additional heuristics: Data Augmentation



CocaColaZero1_1.png



CocaColaZero1_2.png



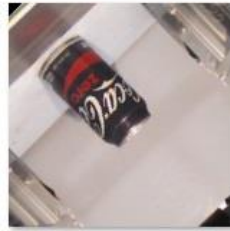
CocaColaZero1_3.png



CocaColaZero1_4.png



CocaColaZero1_5.png



CocaColaZero1_6.png



CocaColaZero1_7.png



CocaColaZero1_8.png

- Available training data will often be small
- “Extend” it by distorting examples in a variety of ways to generate synthetic labelled examples
 - E.g. rotation, stretching, adding noise, other distortion

Other tricks

- Normalize the input:
 - Apply covariate shift to entire training data to make it 0 mean, unit variance
 - Equivalent of batch norm on input
- A variety of other tricks are applied
 - Initialization techniques
 - Typically initialized randomly
 - Key point: neurons with identical connections that are identically initialized will never diverge
 - Practice makes man perfect

Setting up a problem

- Obtain training data
 - Use appropriate representation for inputs and outputs
- Choose network architecture
 - More neurons need more data
 - Deep is better, but harder to train
- Choose the appropriate divergence function
 - Choose regularization
- Choose heuristics (batch norm, dropout, etc.)
- Choose optimization algorithm
 - E.g. Adagrad
- Perform a grid search for hyper parameters (learning rate, regularization parameter, ...) on held-out data
- Train
 - Evaluate periodically on validation data, for early stopping if required

In closing

- Have outlined the process of training neural networks
 - Some history
 - A variety of algorithms
 - Gradient-descent based techniques
 - Regularization for generalization
 - Algorithms for convergence
 - Heuristics
- Practice makes perfect..