

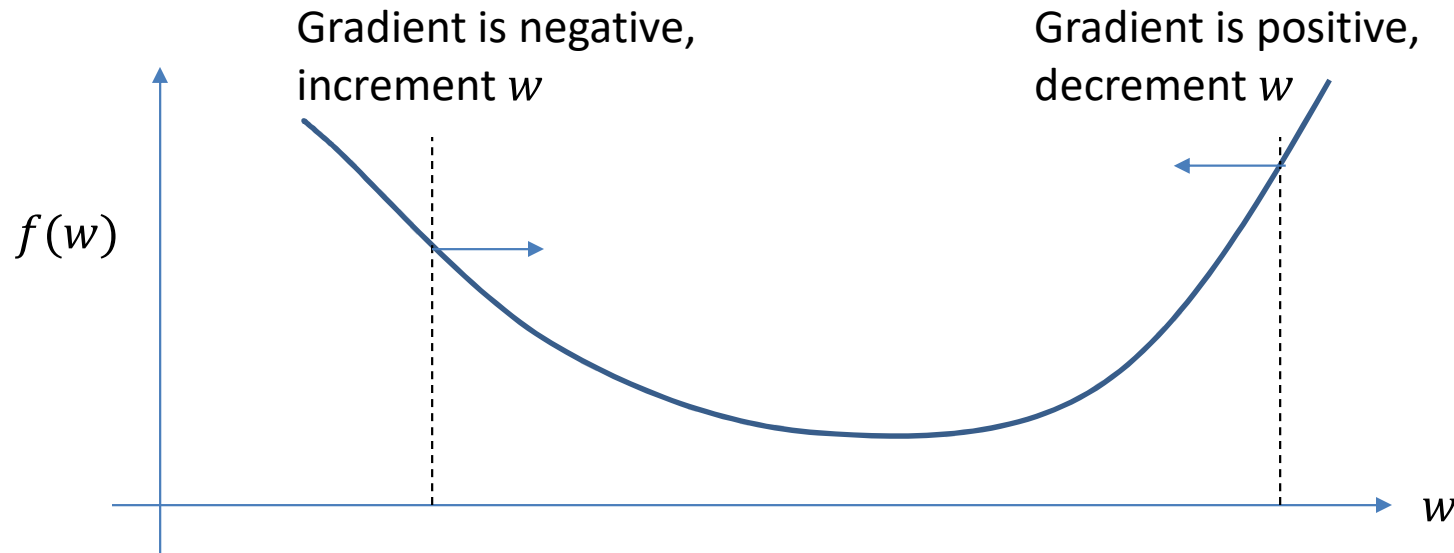
# Training Neural Networks: Optimization

**Intro to Deep Learning, Spring 2018**

# Quick Recap

- Gradient descent, Backprop

# Quick Recap: Gradient Descent



- Gradient *descent* to find the minimum of any “loss” function  $f(w)$ :

- Initialize  $w_0$
- Iterate until the gradient (nearly) vanishes

$$w_k = w_{k-1} - \eta \nabla_w f(w_{k-1})^T$$

# Quick Recap: Training a network

Diagram illustrating the components of the loss function  $L(W)$ :

- Total loss** points to  $L(W)$ .
- Sum over all training instances** points to  $\frac{1}{N_X}$ .
- Divergence between desired output and actual output of net for a given input  $X$**  points to  $div(f(X; W), D(X))$ .
- Output of net in response to input  $X$**  points to  $f(X; W)$ .
- Desired output in response to input  $X$**  points to  $D(X)$ .

$$L(W) = \frac{1}{N_X} \sum_X div(f(X; W), D(X))$$
$$\hat{W} = \arg \min_W L(W)$$

- Define a total “loss” over all training instances
  - Quantifies the difference between desired output and the actual output, as a function of weights
- Find the weights that minimize the loss

# Quick Recap: Training networks by gradient descent

$$L(W) = \frac{1}{N_X} \sum_X \text{div}(f(X; W), D(X))$$

$$\nabla_W L(W) = \frac{1}{N_X} \sum_X \nabla_W \text{div}(f(X; W), D(X))$$

Solved through  
gradient descent as

$$\hat{W} = \arg \min_W L(W)$$



$$W_k = W_{k-1} - \eta \nabla_W L(W)^T$$

- The gradient of the total loss is the average of the gradients of the loss for the individual instances
- The total gradient can be plugged into gradient descent update to learn the network

# Quick Recap: Training networks by gradient descent

$$L(W) = \sum_X$$

Computed using  
backpropagation

$$\nabla_W L(W) = \sum_X \nabla_W \text{div}(f(X; W), D(X))$$

Solved through  
gradient descent as

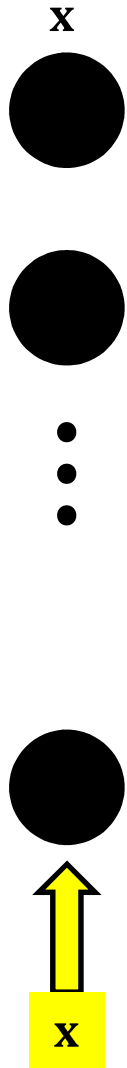
$$\hat{W} = \arg \min_W L(W)$$



$$W_k = W_{k-1} - \eta \nabla_W L(W)^T$$

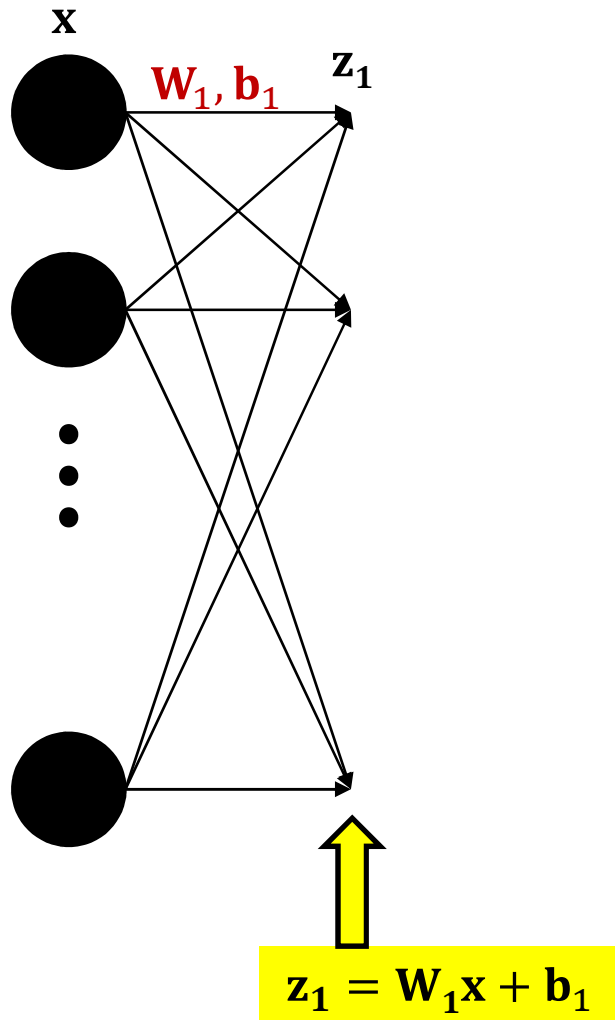
- The gradient of the total loss is the average of the gradients of the loss for the individual instances
- The total gradient can be plugged into gradient descent update to learn the network

# Quick recap of backprop: forward pass



- **Forward pass:** Compute output and all intermediate variables in the network, for the input  $X$

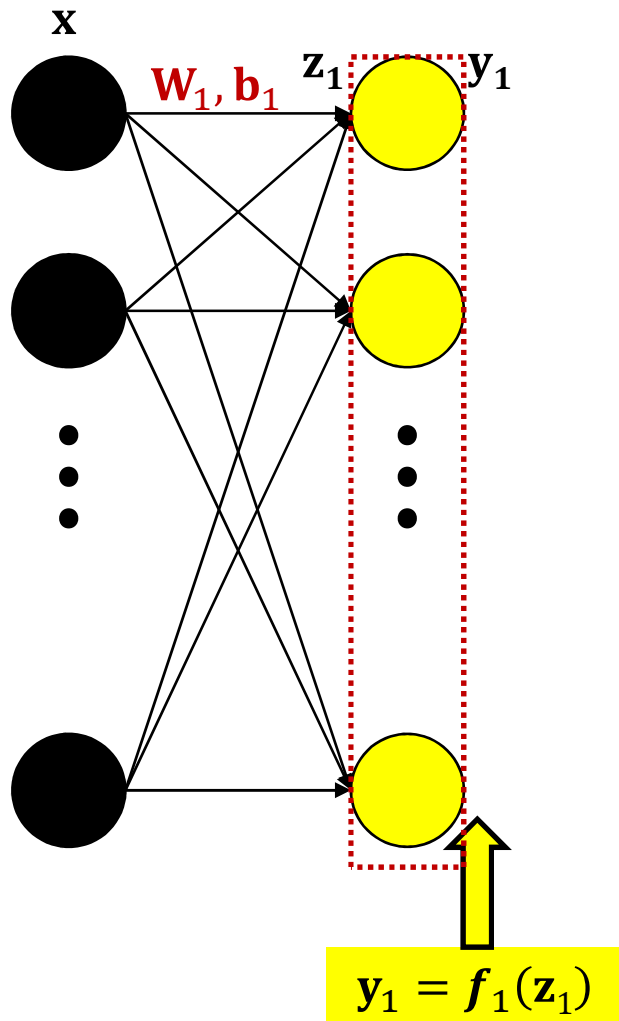
# Quick recap of backprop: forward pass



- **Forward pass:** Compute output and all intermediate variables in the network, for the input  $x$

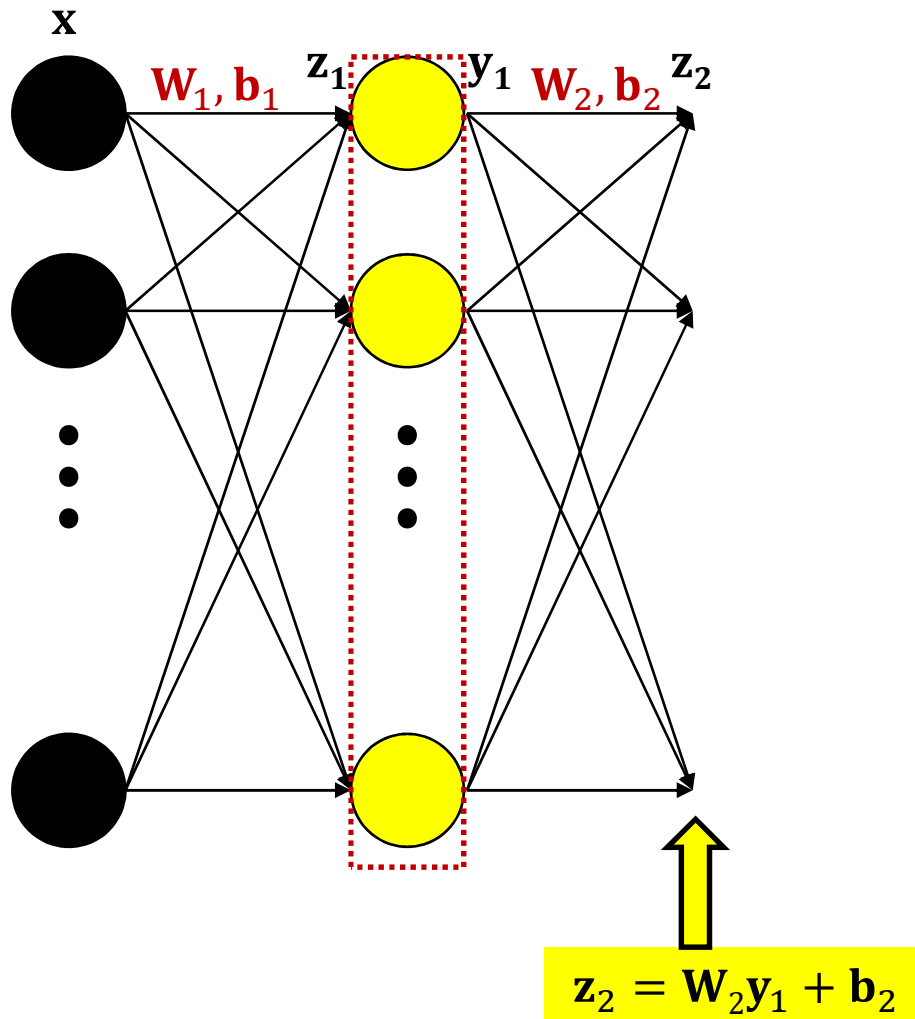


# Quick recap of backprop: forward pass



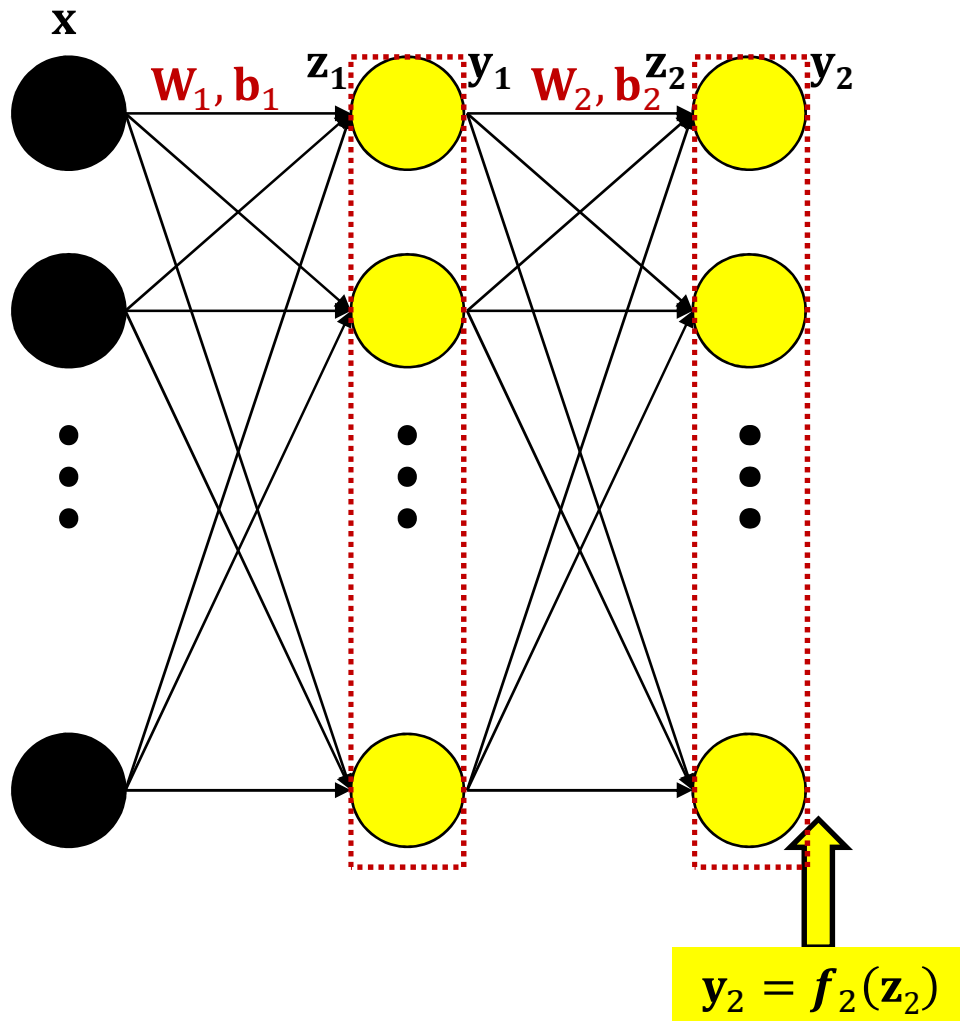
- **Forward pass:** Compute output and all intermediate variables in the network, for the input  $x$

# Quick recap of backprop: forward pass



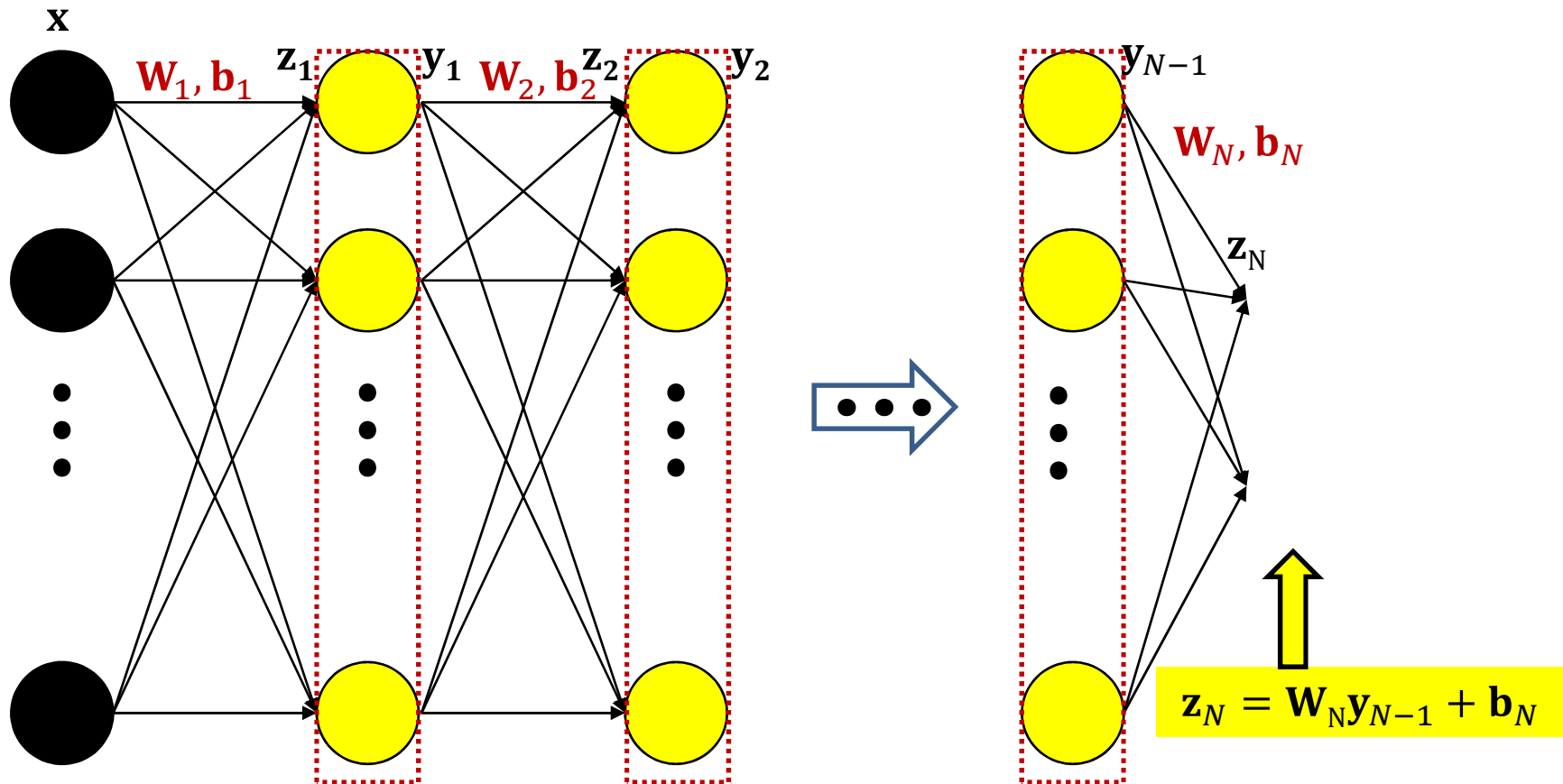
- **Forward pass:** Compute output and all intermediate variables in the network, for the input  $x$

# Quick recap of backprop: forward pass



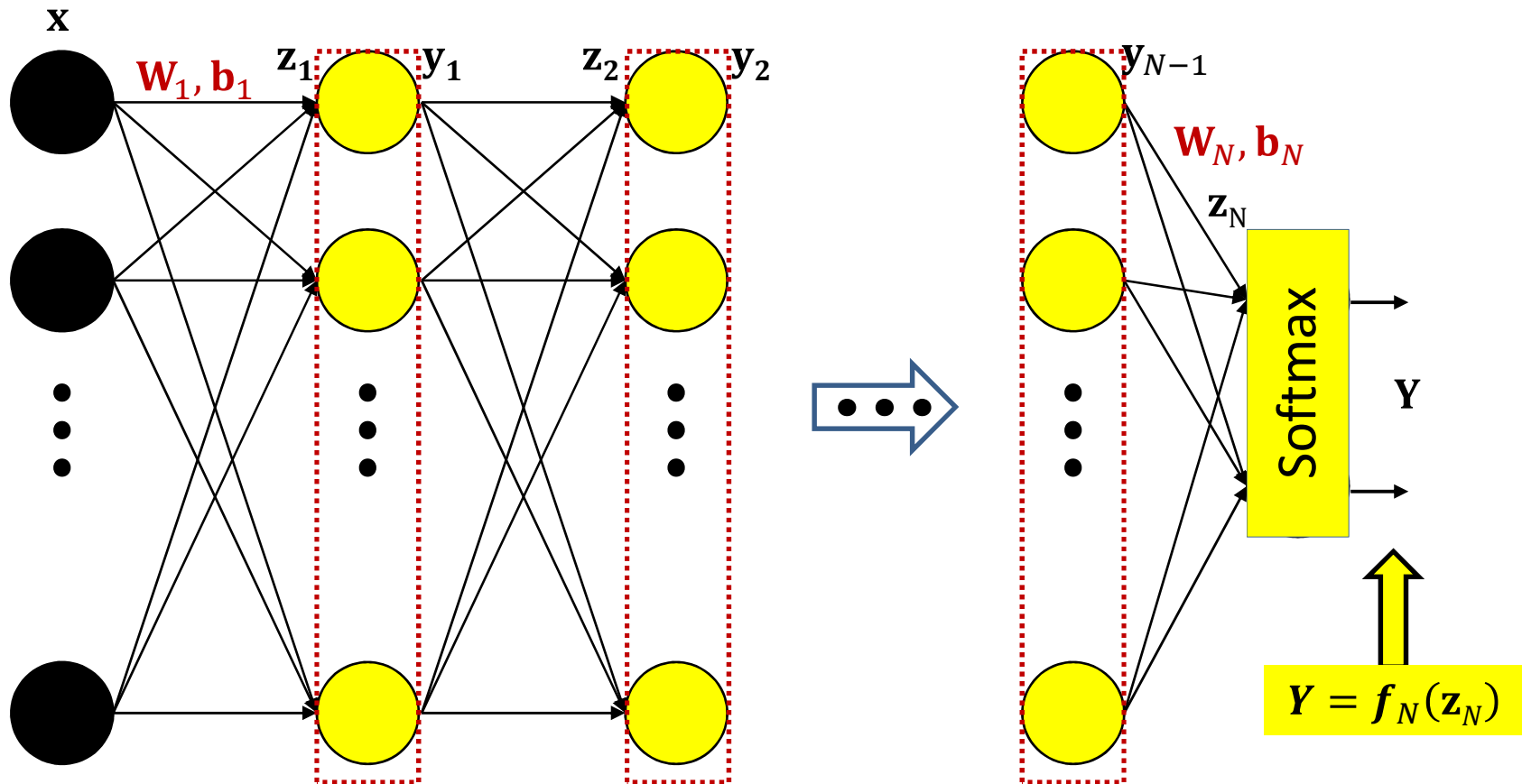
- **Forward pass:** Compute output and all intermediate variables in the network, for the input  $x$

# Quick recap of backprop: forward pass



- **Forward pass:** Compute output and all intermediate variables in the network, for the input  $\mathbf{x}$

# Quick recap of backprop: forward pass



- **Forward pass:** Compute output and all intermediate variables in the network, for the input  $\mathbf{x}$

# The Forward Pass

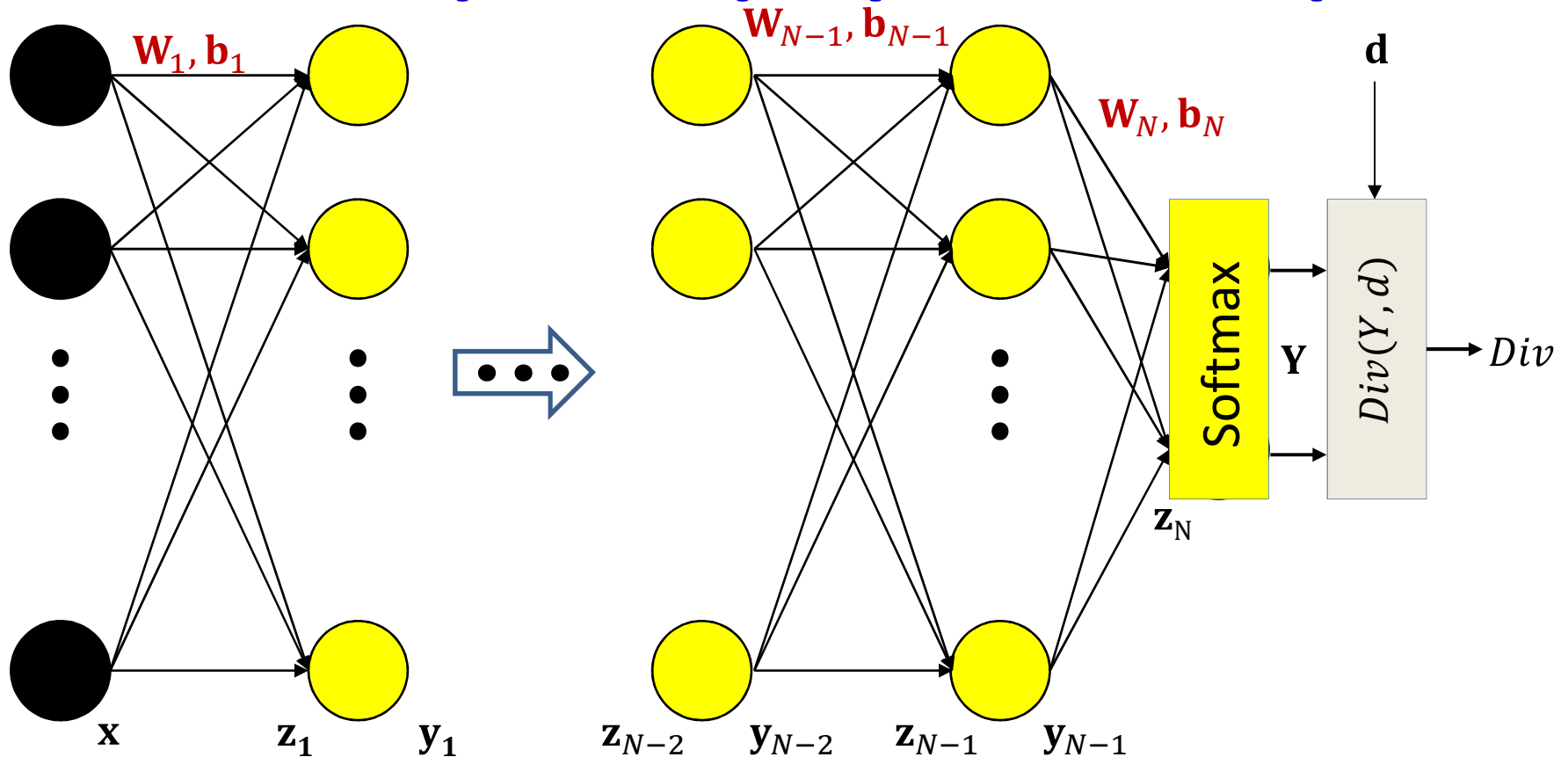
- Set  $\mathbf{y}_0 = \mathbf{x}$
- For layer  $k = 1$  to  $N$ :
  - Recursion:

$$\begin{aligned}\mathbf{z}_k &= \mathbf{W}_k \mathbf{y}_{k-1} + \mathbf{b}_k \\ \mathbf{y}_k &= \mathbf{f}_k(\mathbf{z}_k)\end{aligned}$$

- Output:

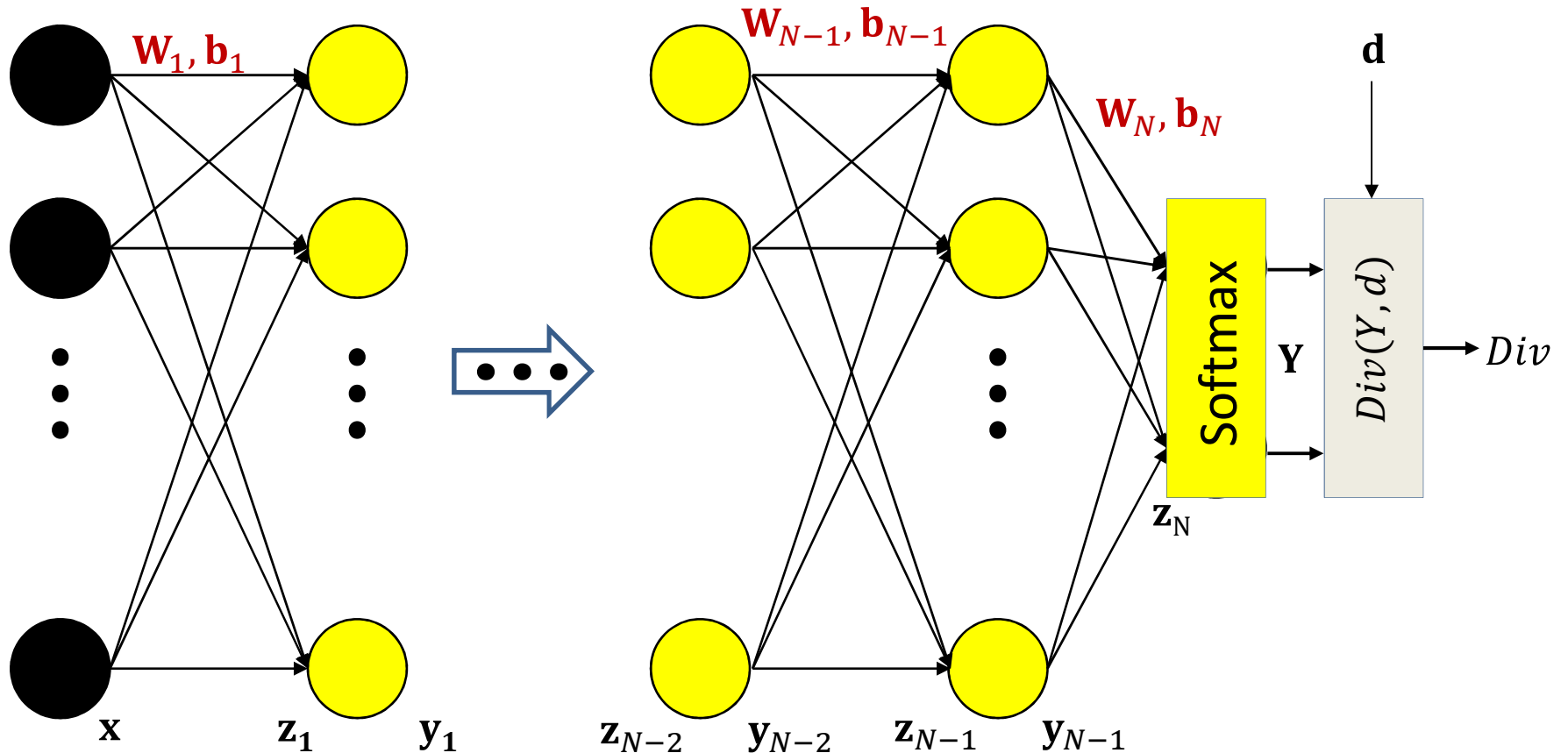
$$\mathbf{Y} = \mathbf{y}_N$$

# Quick Recap: Backprop. Forward pass



- **Forward pass:** Compute output and all intermediate variables in the network, for the input  $X$
- **Compute the divergence w.r.t. *desired* output**

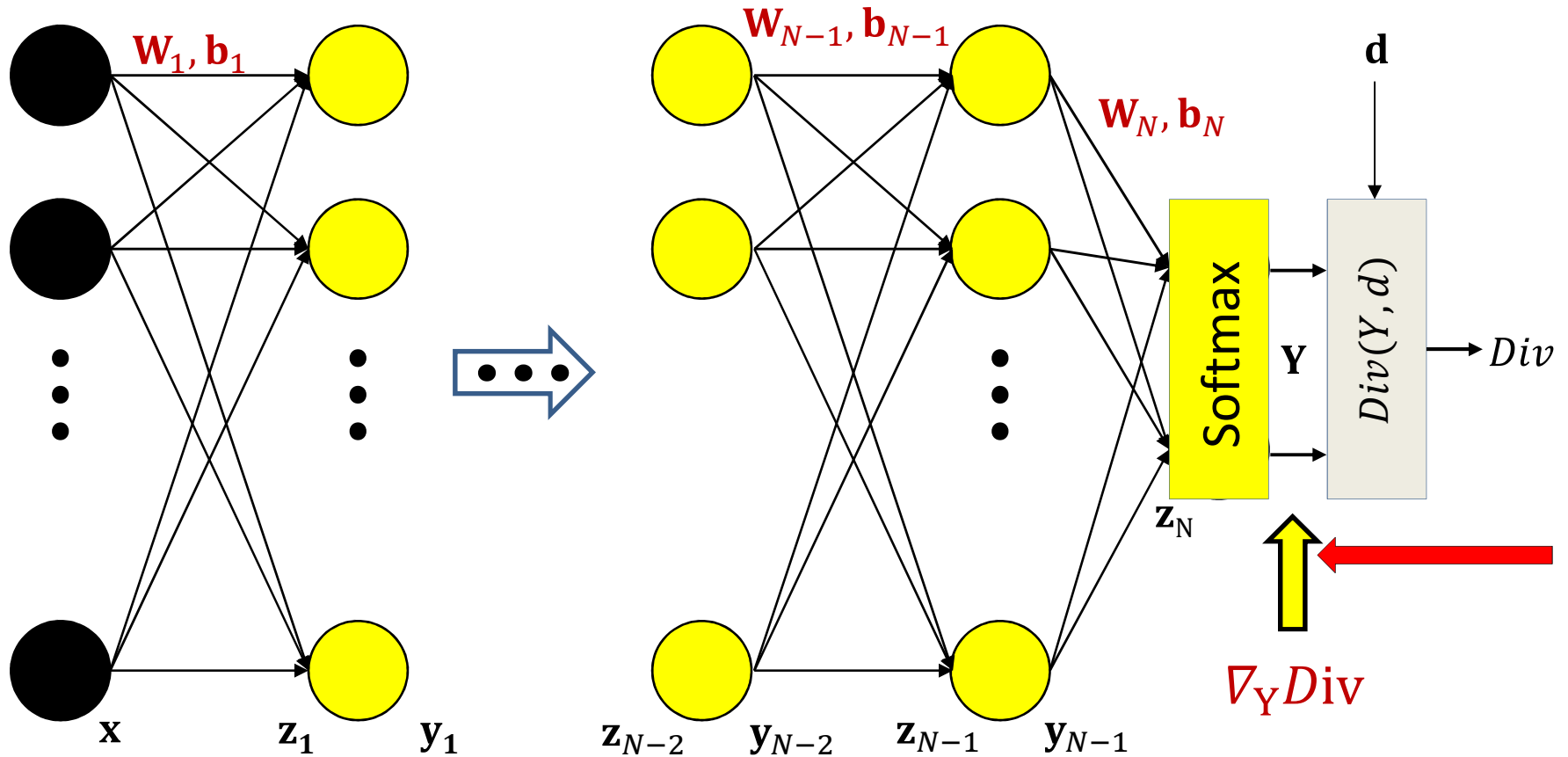
# Quick Recap: Backpropagation



- Now work your way *backward* through the net to compute the derivative w.r.t each intermediate variable and each weight/bias

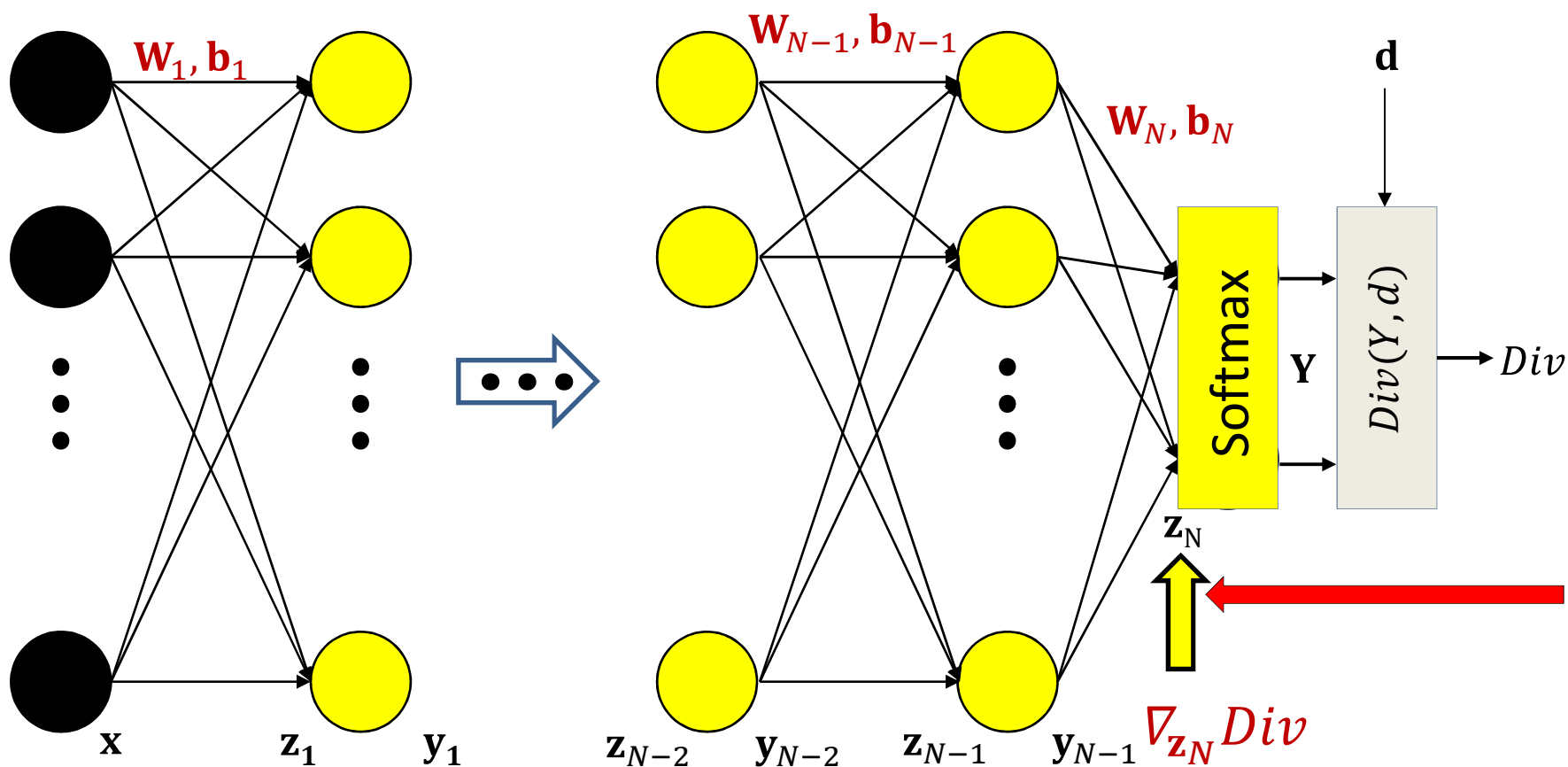


# Backprop



First compute the gradient of the divergence w.r.t.  $Y$ .  
The actual gradient depends on the divergence function.

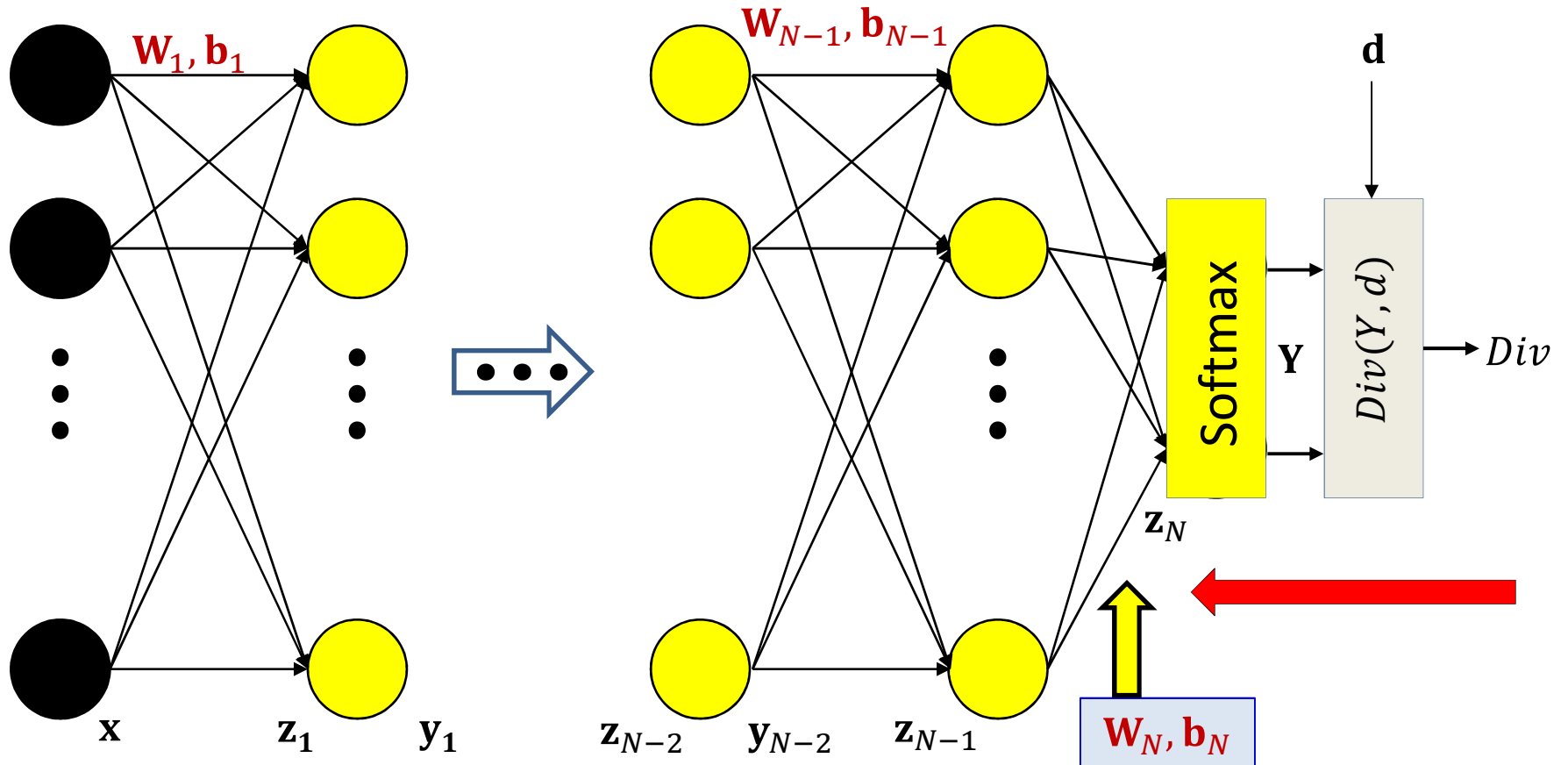
# Backprop



$$\nabla_{z_N} Div = \nabla_Y Div J_Y(z_N)$$

Chain rule (vector format; note order of multiplication)

# The backward pass

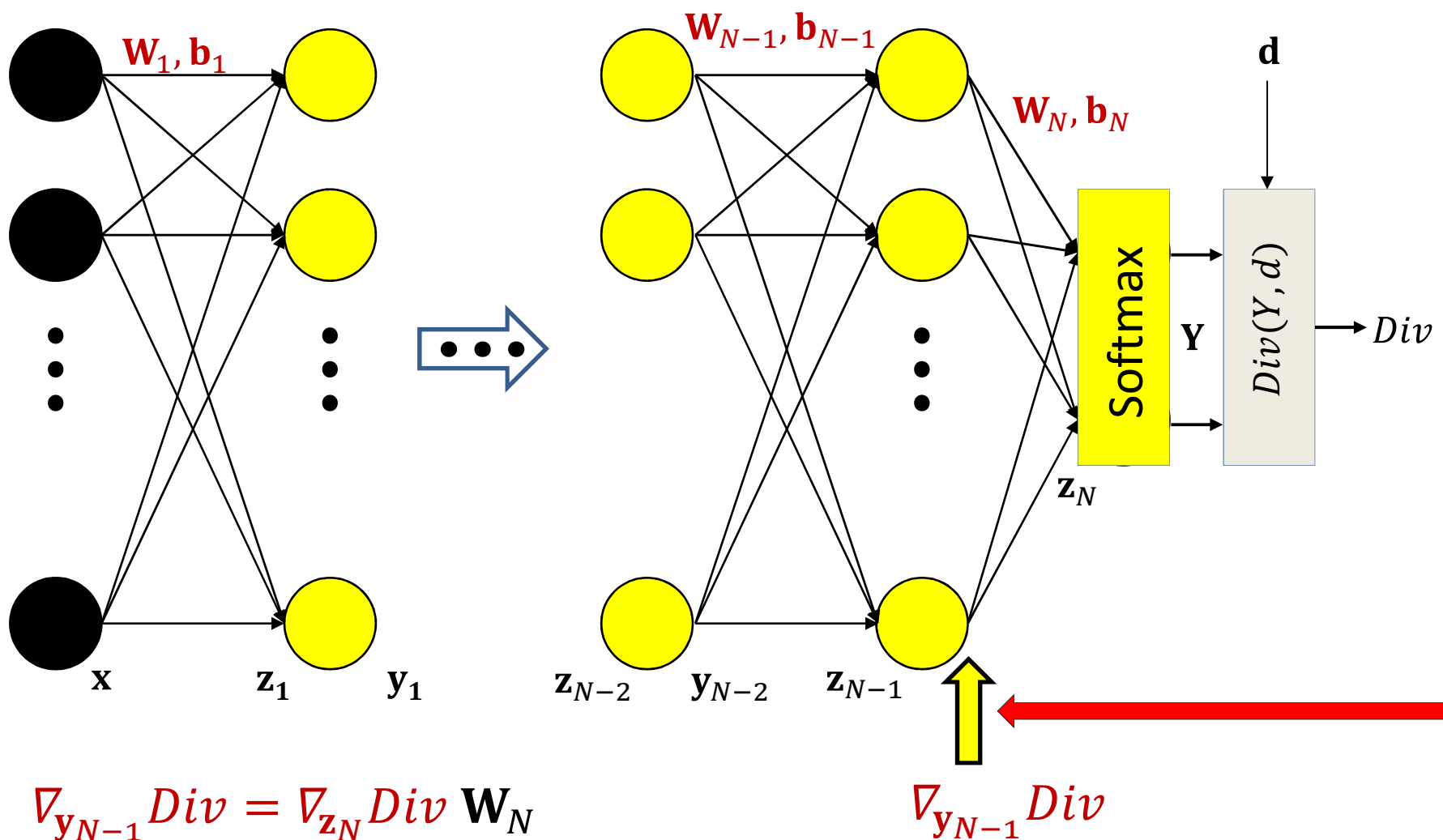


$$z_N = w_N y_{N-1} + b_N$$

$$\nabla_{w_N} Div = y_{N-1} \nabla_{z_N} Div$$

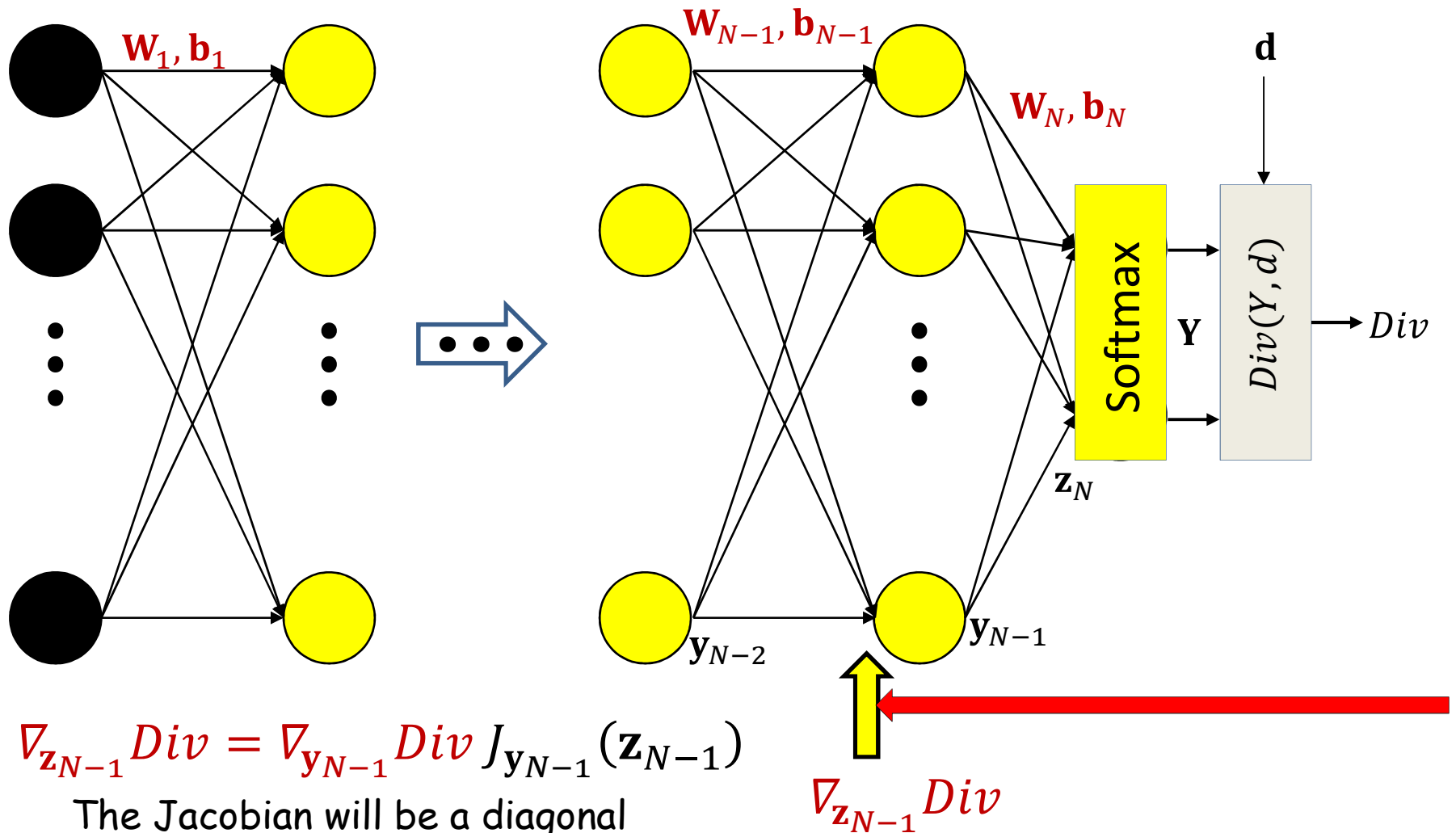
$$\nabla_{b_N} Div = \nabla_{z_N} Div$$

# Backprop



Chain rule (vector format; note order of multiplication)

# The backward pass

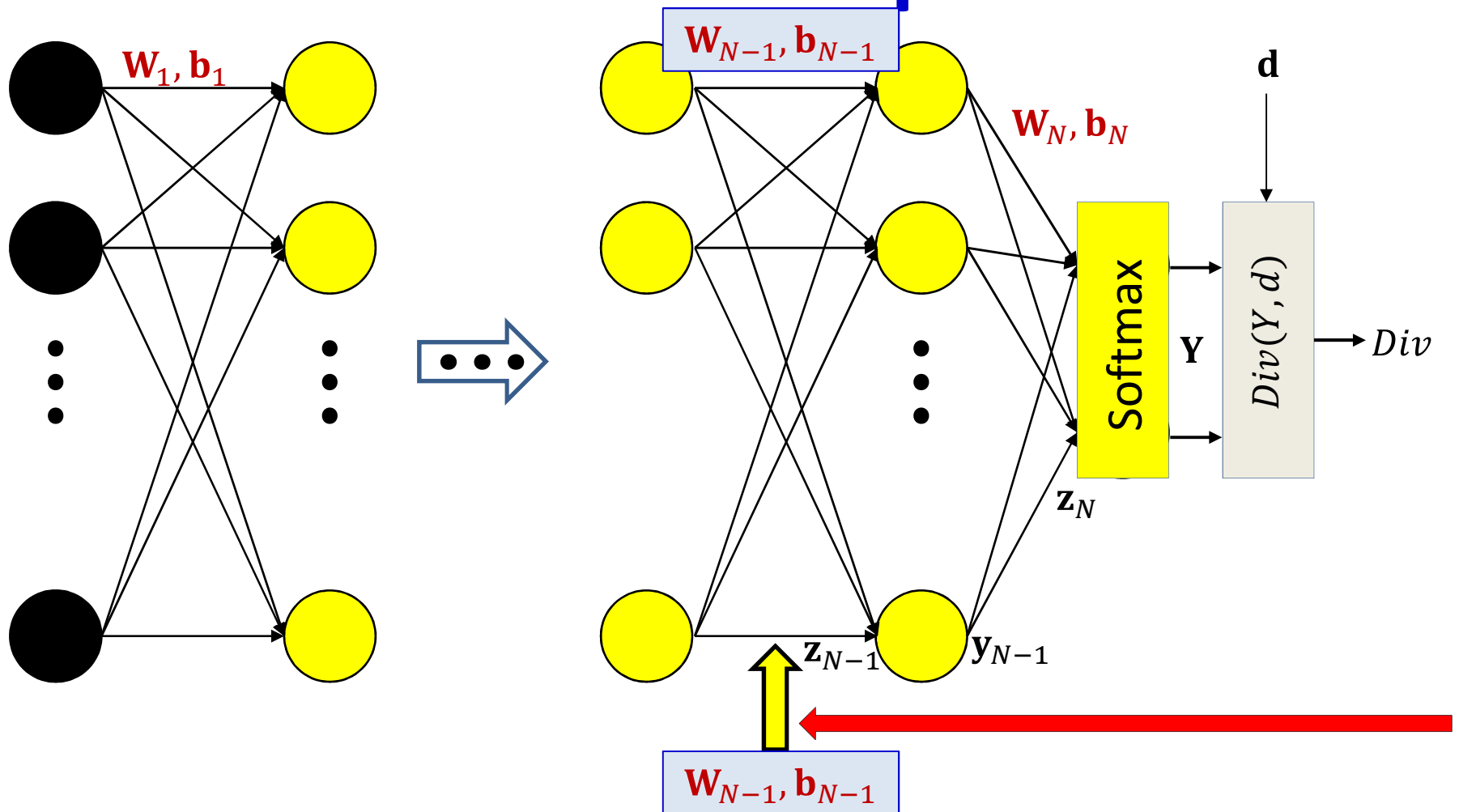


$$\nabla_{z_{N-1}} Div = \nabla_{y_{N-1}} Div J_{y_{N-1}}(z_{N-1})$$

The Jacobian will be a diagonal matrix for scalar activations

Chain rule (vector format; note order of multiplication)

# The backward pass

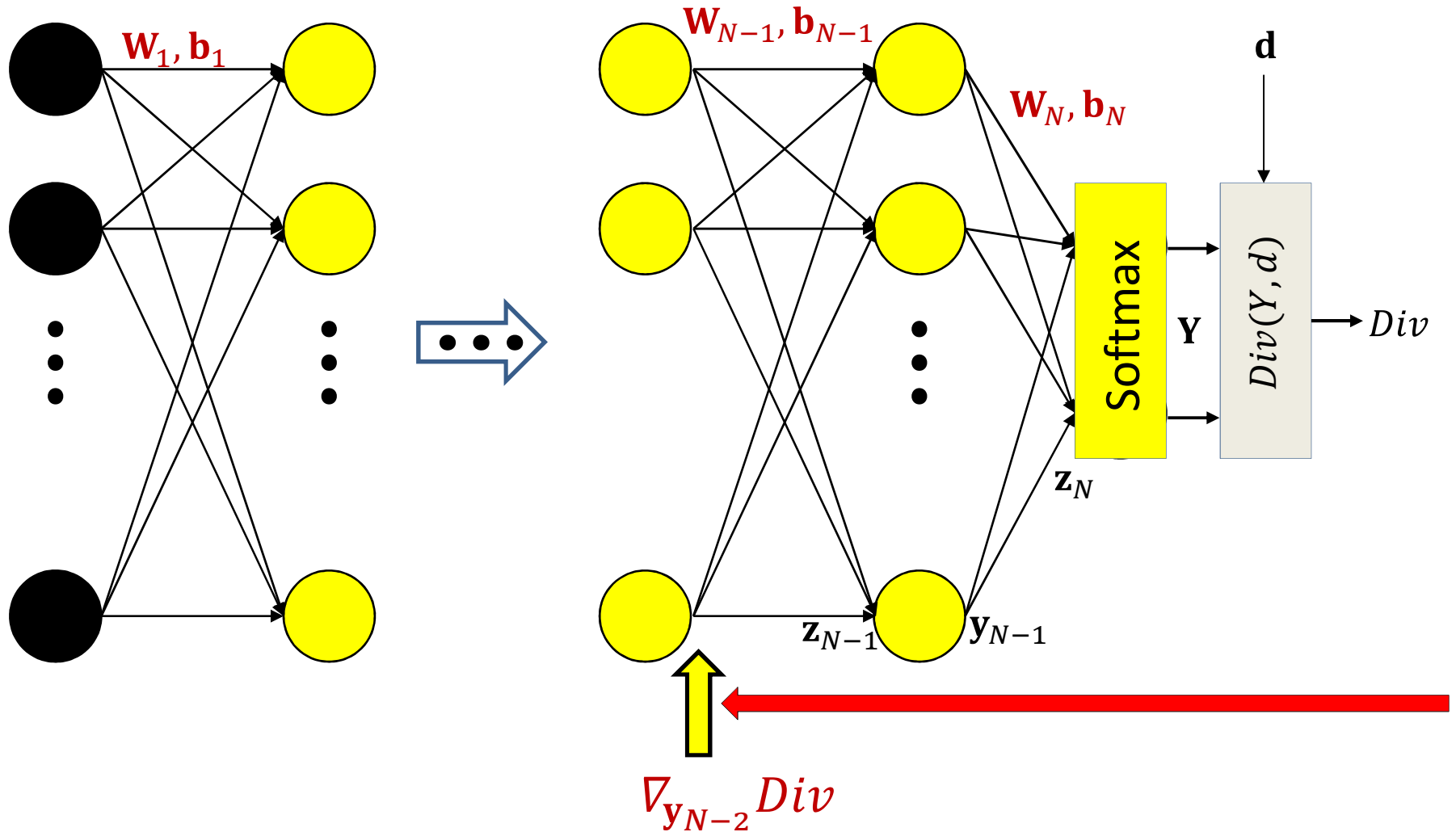


$$z_{N-1} = w_{N-1}y_{N-2} + b_{N-1}$$

$$\nabla_{w_{N-1}} Div = y_{N-2} \nabla_{z_{N-1}} Div$$

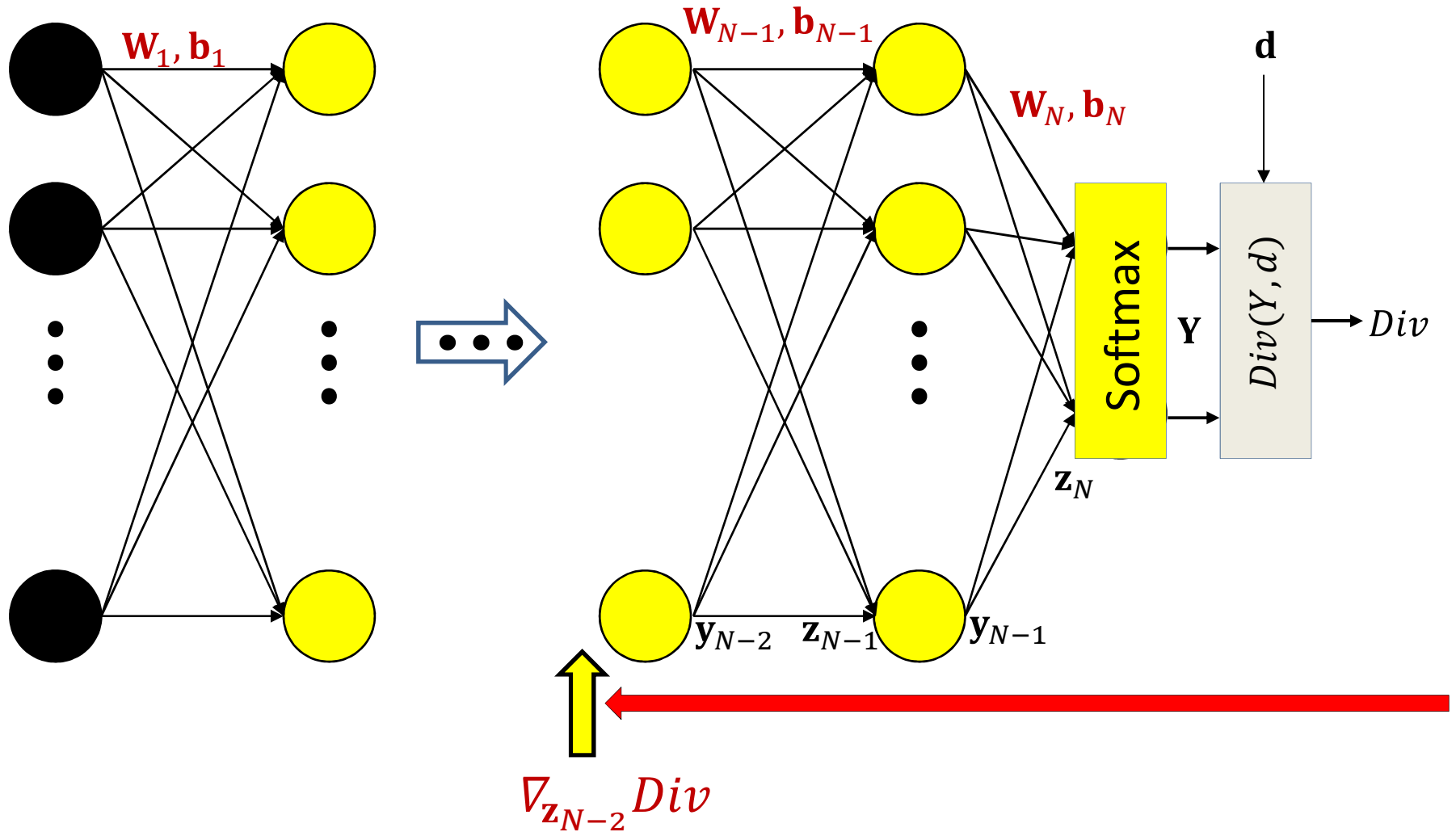
$$\nabla_{b_{N-1}} Div = \nabla_{z_{N-1}} Div$$

# The backward pass



$$\nabla_{\mathbf{y}_{N-2}} Div = \nabla_{\mathbf{z}_{N-1}} Div \mathbf{W}_{N-1}$$

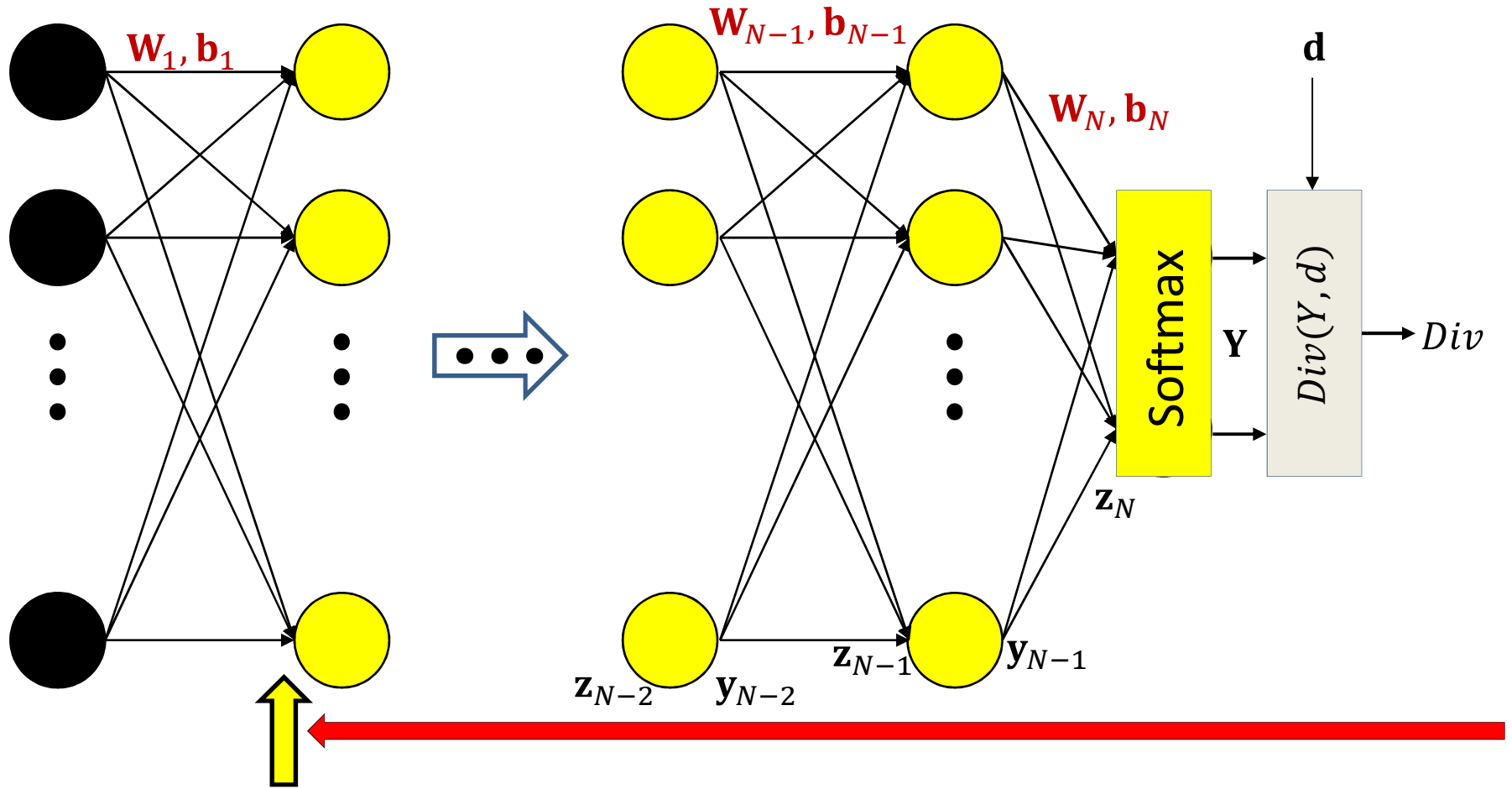
# The backward pass



$$\nabla_{z_{N-2}} Div = \nabla_{y_{N-2}} Div J_{y_{N-2}}(z_{N-2})$$

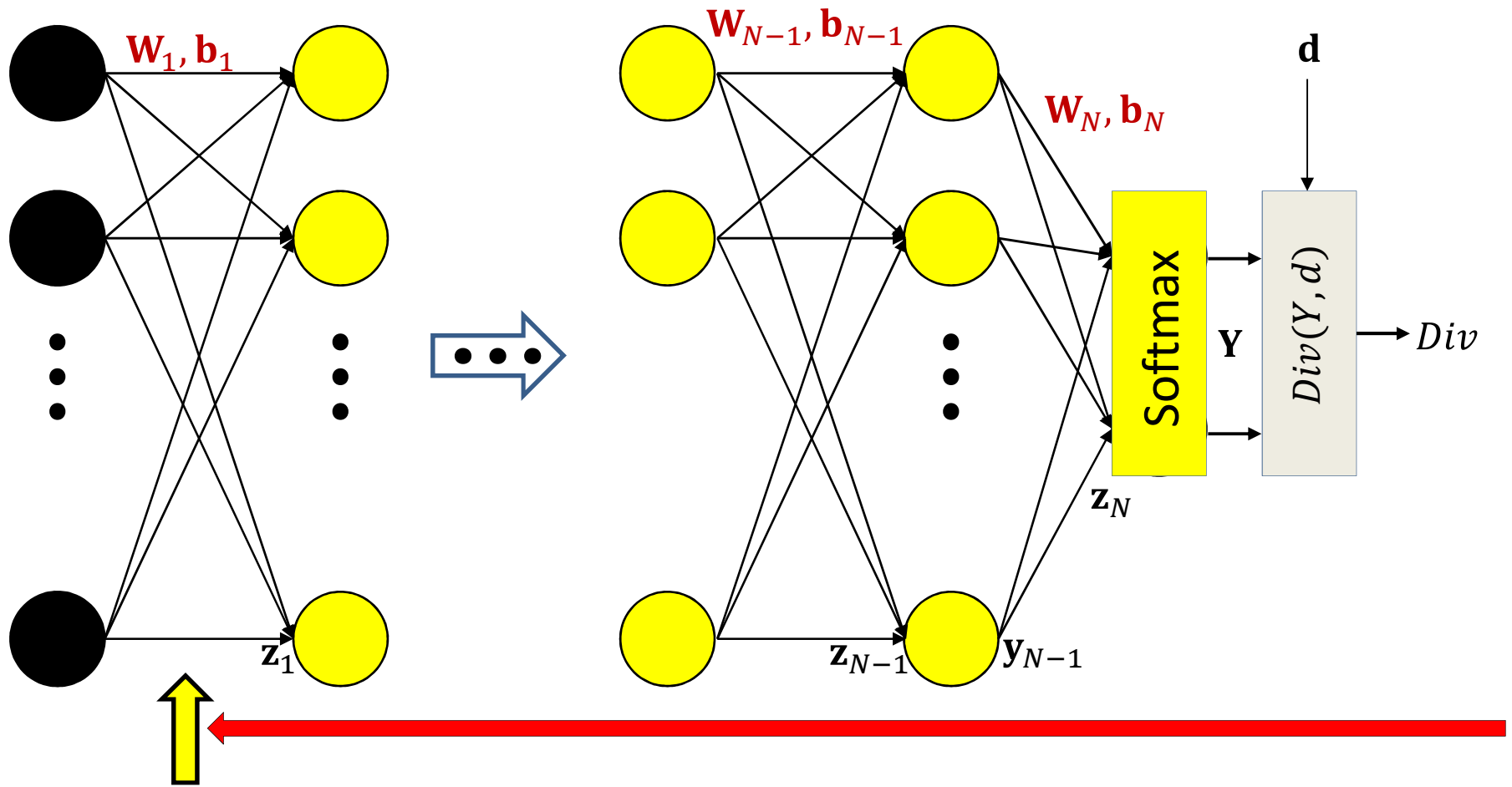


# The backward pass



$$\nabla_{z_1} Div = \nabla_{y_1} Div J_{y_1}(z_1)$$

# The backward pass



$$\nabla_{w_1} Div = x \nabla_{z_1} Div$$

$$\nabla_{b_1} Div = \nabla_{z_1} Div$$

In some problems we will also want to compute the derivative w.r.t. the input

# The Backward Pass

- Set  $\mathbf{y}_N = Y, \mathbf{y}_0 = \mathbf{x}$
- Initialize: Compute  $\nabla_{\mathbf{y}_N} Div = \nabla_Y Div$
- For layer  $k = N$  downto 1:

– Recursion:

$$\nabla_{\mathbf{z}_k} Div = \nabla_{\mathbf{y}_k} Div J_{\mathbf{y}_k}(\mathbf{z}_k)$$

$$\nabla_{\mathbf{y}_{k-1}} Div = \nabla_{\mathbf{z}_k} Div \mathbf{W}_k$$

– Gradient computation:

$$\nabla_{\mathbf{W}_k} Div = \mathbf{y}_{k-1} \nabla_{\mathbf{z}_k} Div$$

$$\nabla_{\mathbf{b}_k} Div = \nabla_{\mathbf{z}_k} Div$$

# Neural network training algorithm

- Initialize all weights and biases  $(\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2, \dots, \mathbf{W}_N, \mathbf{b}_N)$

- Do:

- $Err = 0$

- For all  $k$ , initialize  $\nabla_{\mathbf{W}_k} Err = 0, \nabla_{\mathbf{b}_k} Err = 0$

- For all  $t = 1:T$

- Forward pass : Compute

- Output  $\mathbf{Y}(X_t)$

- Divergence  $Div(\mathbf{Y}_t, \mathbf{d}_t)$

- $Err += Div(\mathbf{Y}_t, \mathbf{d}_t)$

- Backward pass: For all  $k$  compute:

- $\nabla_{\mathbf{W}_k} Div(\mathbf{Y}_t, \mathbf{d}_t); \nabla_{\mathbf{b}_k} Div(\mathbf{Y}_t, \mathbf{d}_t)$

- $\nabla_{\mathbf{W}_k} Err += \nabla_{\mathbf{W}_k} Div(\mathbf{Y}_t, \mathbf{d}_t); \nabla_{\mathbf{b}_k} Err += \nabla_{\mathbf{b}_k} Div(\mathbf{Y}_t, \mathbf{d}_t)$

- For all  $k$ , update:

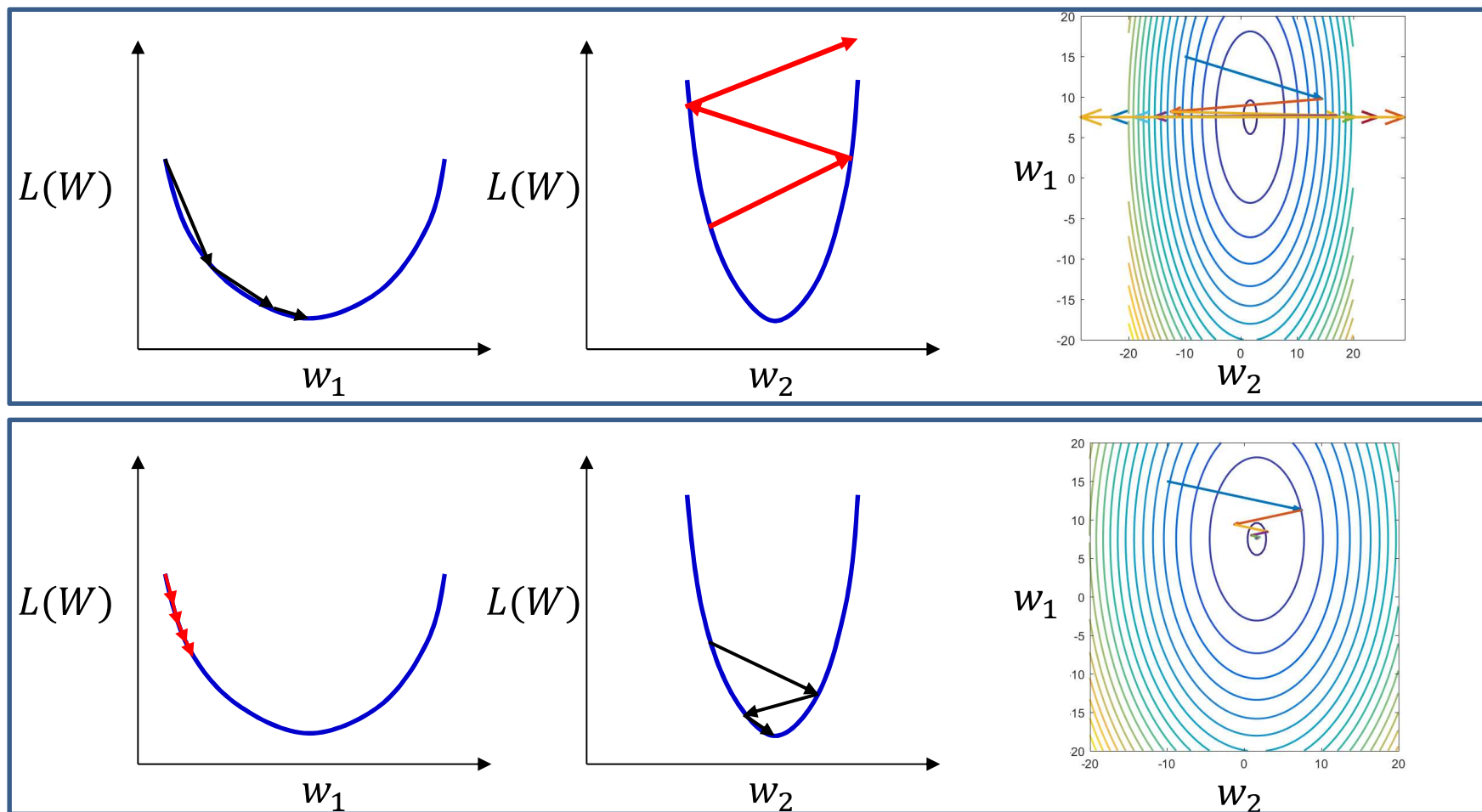
$$\mathbf{W}_k = \mathbf{W}_k - \frac{\eta}{T} (\nabla_{\mathbf{W}_k} Err)^T; \quad \mathbf{b}_k = \mathbf{b}_k - \frac{\eta}{T} (\nabla_{\mathbf{b}_k} Err)^T$$

- Until  $Err$  has converged

# Quick Recap

- Gradient descent, Backprop
- The issues with backprop and gradient descent
  - 1. Minimizes a *loss* which *relates* to classification accuracy, but is not actually classification accuracy
    - The divergence is a continuous valued proxy to classification error
    - Minimizing the loss is *expected* to, but not *guaranteed* to minimize classification error
  - 2. Simply minimizing the loss is hard enough..

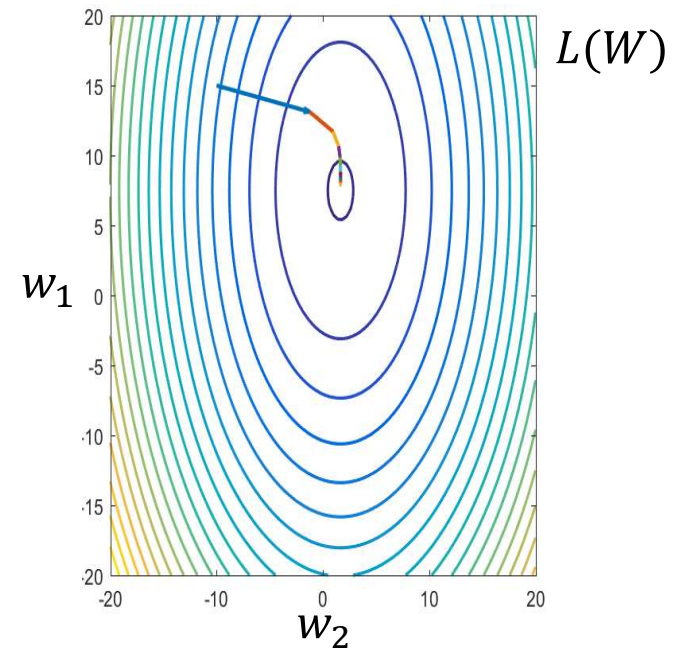
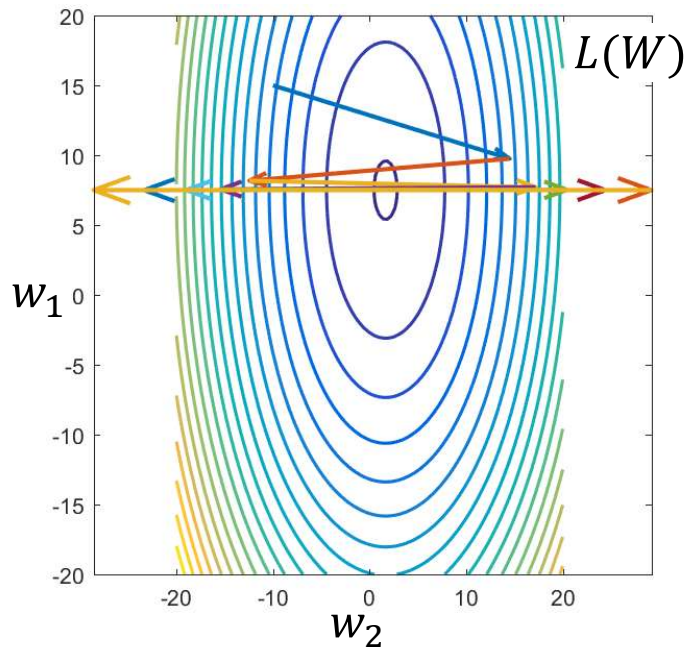
# Quick recap: Problem with gradient descent



$$W_k = W_{k-1} - \eta \nabla_w L(W)^T$$

- A step size that assures fast convergence for a given eccentricity can result in divergence at a higher eccentricity
- .. Or result in extremely slow convergence at lower eccentricity

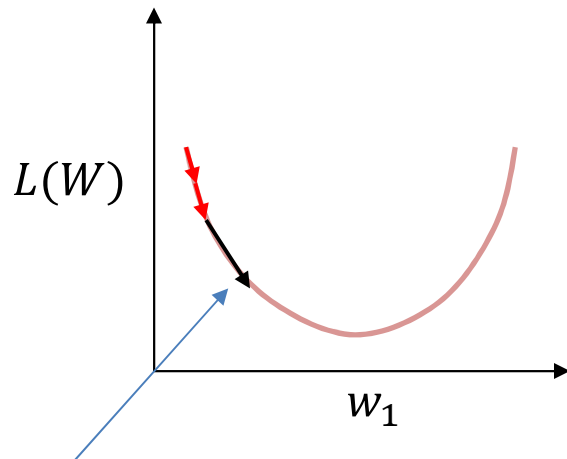
# Quick recap: Problem with gradient descent



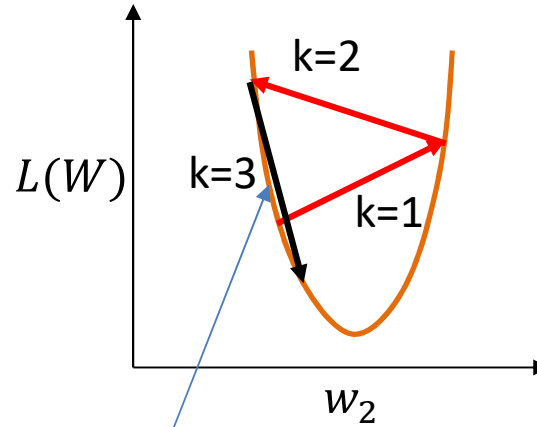
- The loss is a function of many weights (and biases)
  - Has different eccentricities w.r.t different weights
- A fixed step size for all weights in the network can result in the convergence of one weight, while causing a divergence of another

# Quick recap: Problem with gradient descent

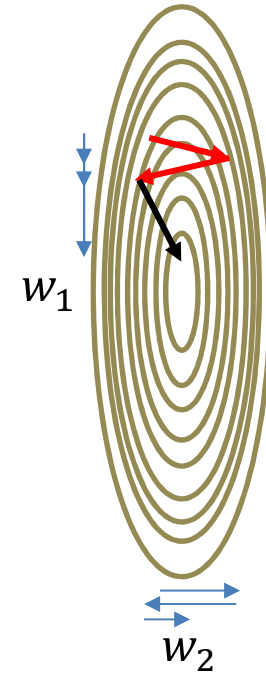
$$W_k = W_{k-1} - \eta \nabla_w L(W)^T$$



Increase stepsize because previous updates consistently moved weight right



Decrease stepsize because previous updates kept changing direction



Stepsize shrinks along  $w_2$  but increases along  $w_1$

- Ideally: Have component-specific step size
  - Too many independent parameters (maintain a step size for every weight/bias)
- Adaptive solution: Start with a common step size
  - *Shrink* step size in directions where the weight oscillates
  - *Expand* step size in directions where the weight moves consistently in one direction



# Quick Recap

- Gradient descent, Backprop
- The issues with backprop and gradient descent
- Rprop and Momentum

# Quick recap: Rprop

- RPROP: Independently update step size of every weight
  - But step size is independent of actual gradient
    - Gradient only used to determine *direction* of update
- Initialize stepsize  $\eta_w = 1$  for all weights
  - If  $\text{sign}\left(\frac{\partial L(W_{k-1})}{w_{k-1}}\right) \text{sign}\left(\frac{\partial L(W_k)}{w_k}\right) > 0$ :  $\eta_w = \alpha \eta_w$ , where  $\alpha > 1$
  - Else:  $\eta_w = \beta \eta_w$ , where  $\beta < 1$
  - $k$  represent iteration (rule applies individually to every weight  $w$  in the network)
- If the current gradient has the same sign as the last one, we are continuing to move in the same direction over consecutive steps
  - Test to check this: their product is positive
  - Increase the step size by a factor  $\alpha$
- Otherwise, we are reversing direction
  - Have overshoot the minimum and may be diverging
  - Decrease the step size by a factor  $\beta$

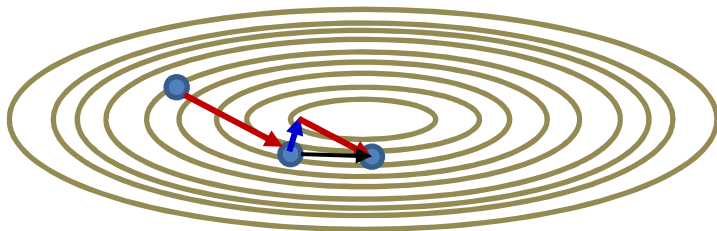
# Quick recap: Rprop

Note: Does not actually use the derivative values. Only the signs

- RPROP: Independently update step size of every weight
  - But step size is independent of actual gradient
    - Gradient only used to determine *direction* of update
- Initialize stepsize  $\eta_w = 1$  for all weights
  - If  $\text{sign}\left(\frac{\partial L(W_{k-1})}{w_{k-1}}\right) \text{sign}\left(\frac{\partial L(W_k)}{w_k}\right) > 0$ :  $\eta_w = \alpha \eta_w$ , where  $\alpha > 1$
  - Else:  $\eta_w = \beta \eta_w$ , where  $\beta < 1$
  - $k$  represent iteration (rule applies individually to every weight  $w$  in the network)
- If the current gradient has the same sign as the last one, we are continuing to move in the same direction over consecutive steps
  - Test to check this: their product is positive
  - Increase the step size by a factor  $\alpha$
- Otherwise, we are reversing direction
  - Have overshoot the minimum and may be diverging
  - Decrease the step size by a factor  $\beta$

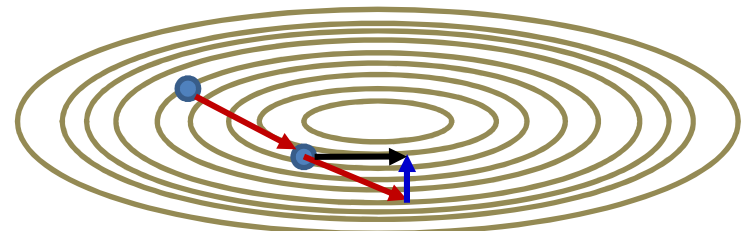
# Quick recap: Momentum methods

Momentum



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

Nestorov



$$\begin{aligned} W_{\text{extend}}^{(k)} &= W^{(k-1)} + \beta \Delta W^{(k-1)} \\ \Delta W^{(k)} &= \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Err}(W_{\text{extend}}^{(k)}) \\ W^{(k)} &= W^{(k-1)} + \Delta W^{(k)} \end{aligned}$$

Note: now a *vector* update rule(not component wise like rprop)

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

# 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: Topics for the day

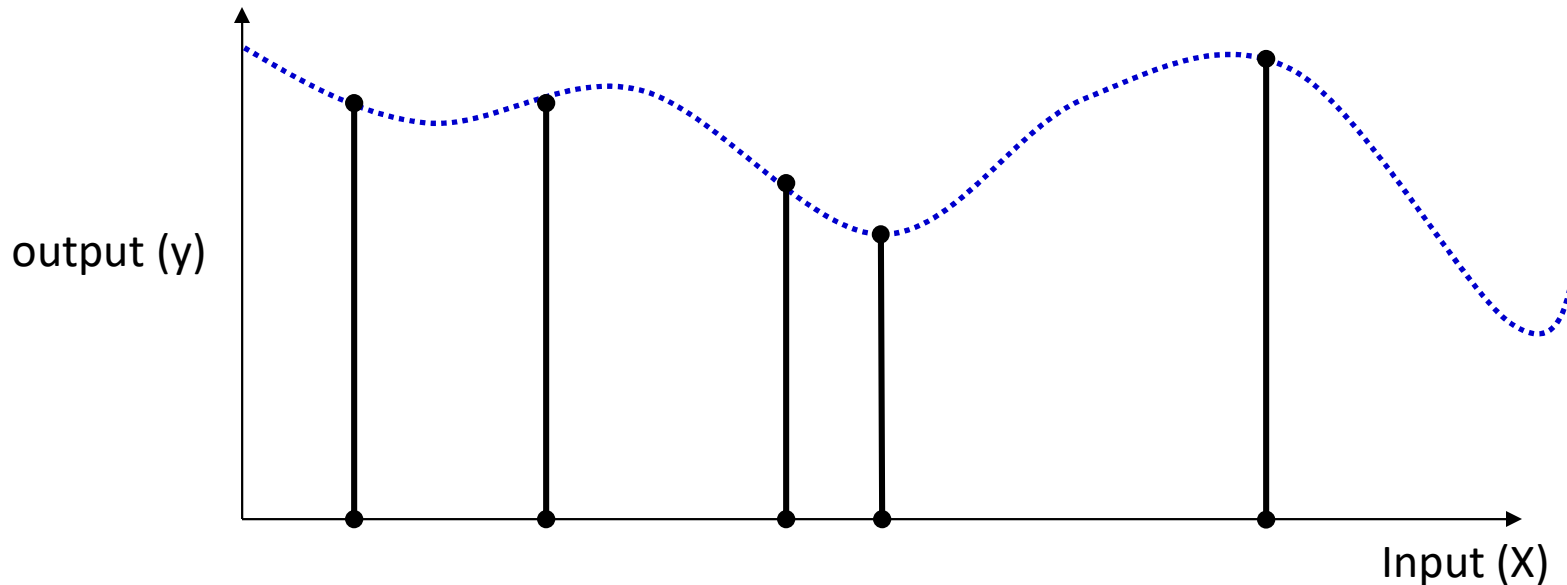
- Incremental updates
- Revisiting “trend” algorithms
- Generalization
- Tricks of the trade
  - Divergences..
  - Activations
  - Normalizations

# Moving on: Topics for the day

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

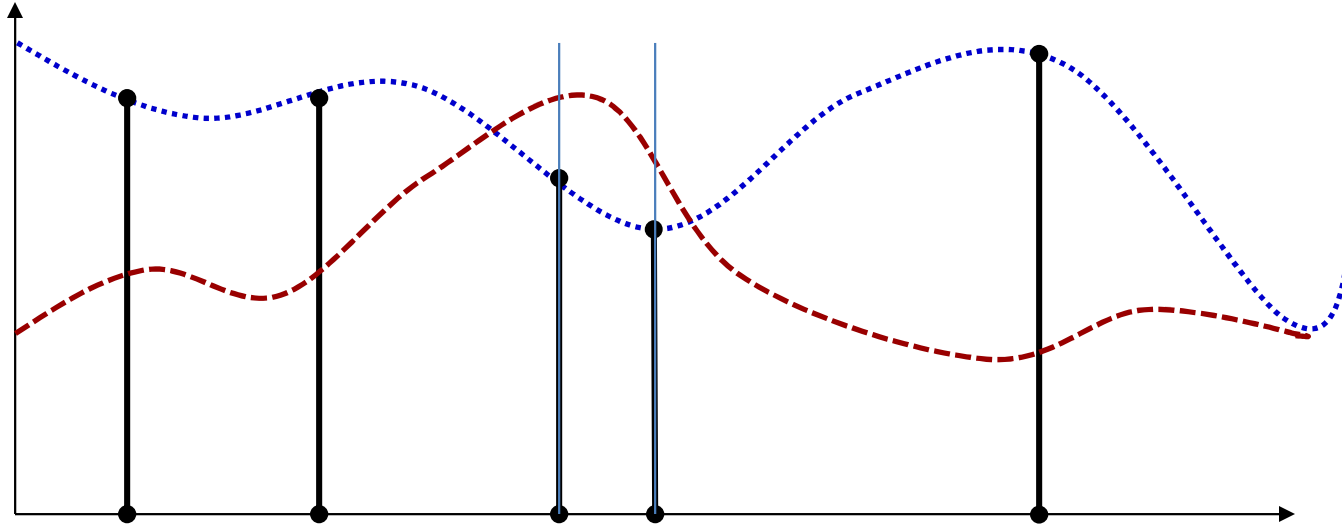


# The training formulation



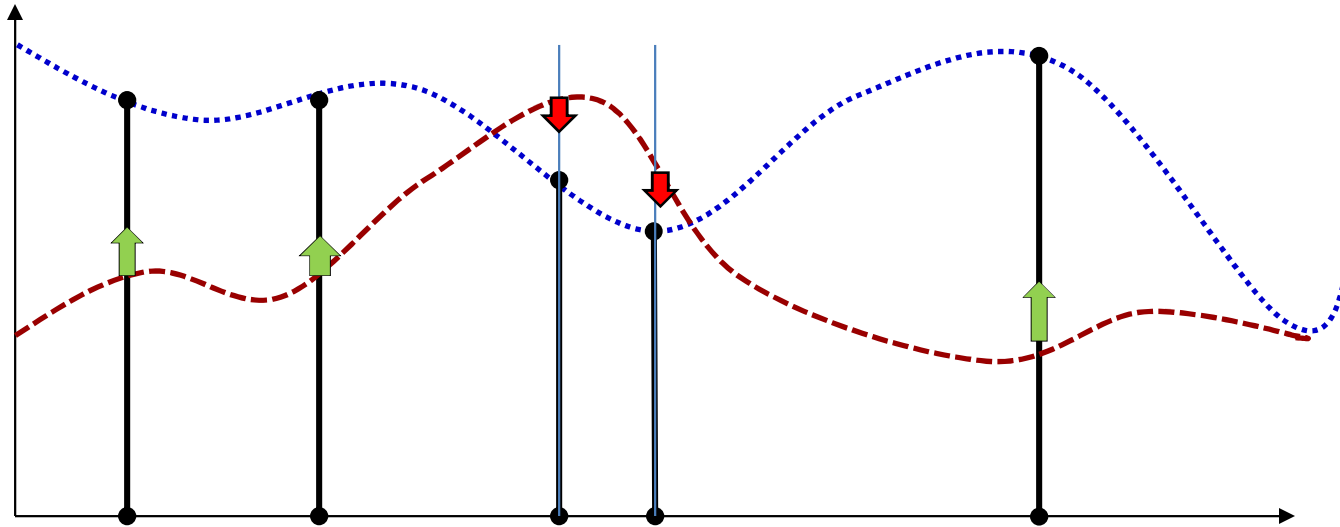
- Given input output pairs at a number of locations, estimate the entire function

# Gradient descent



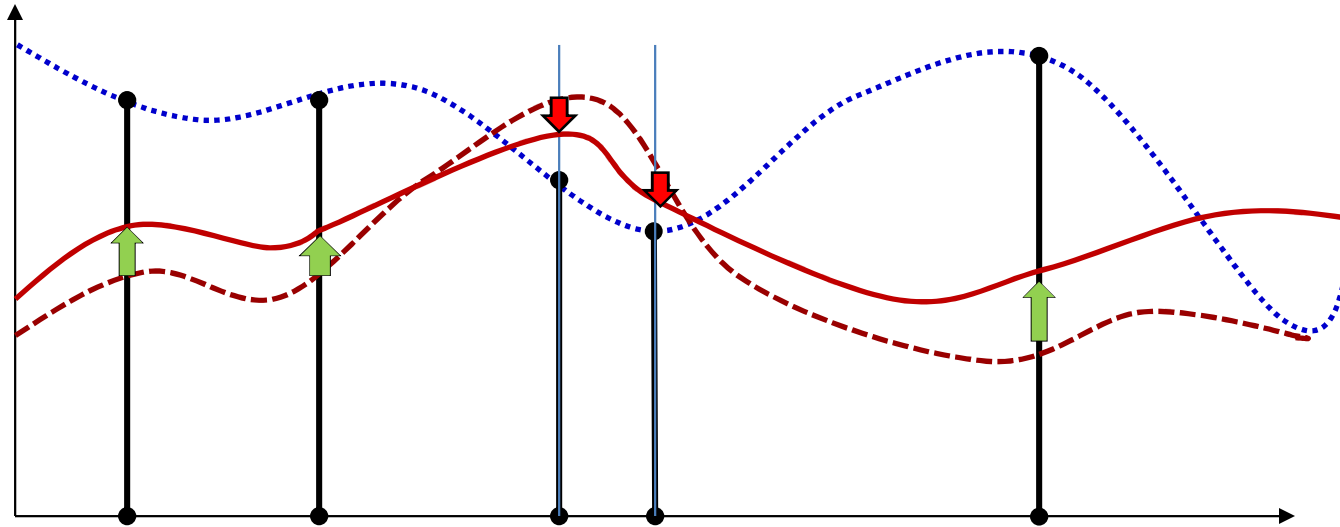
- Start with an initial function

# Gradient descent



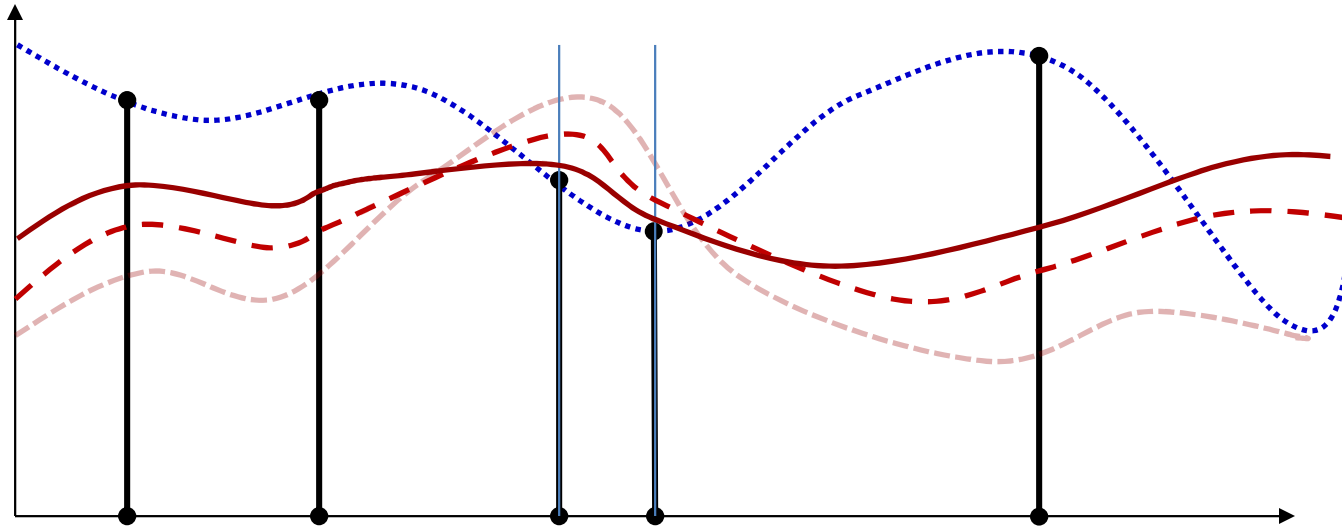
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

# Gradient descent



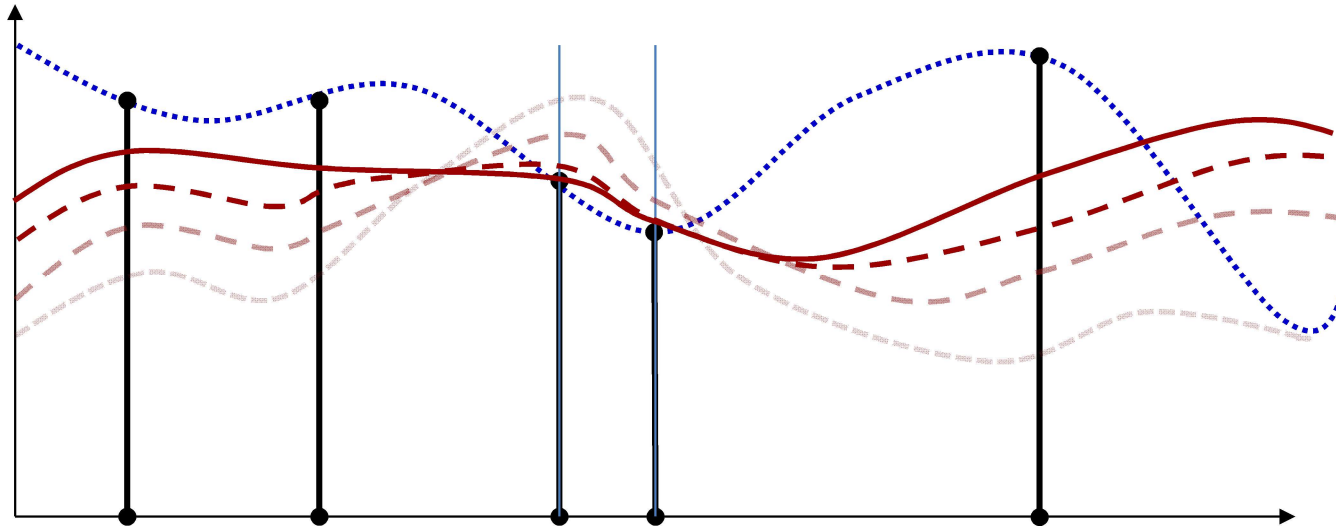
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

# Gradient descent



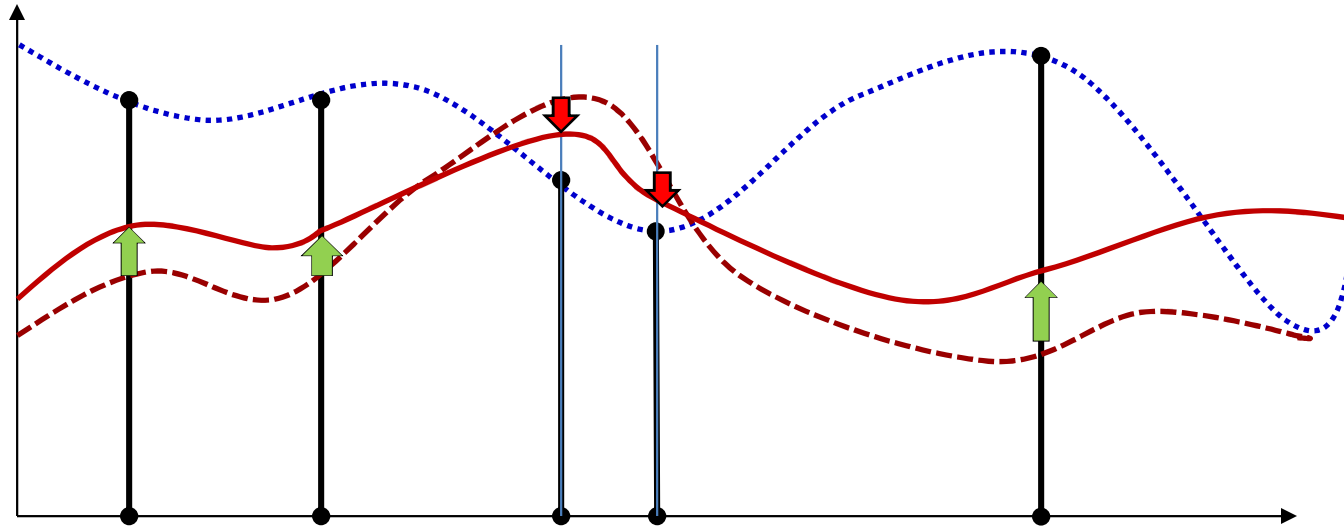
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

# Gradient descent



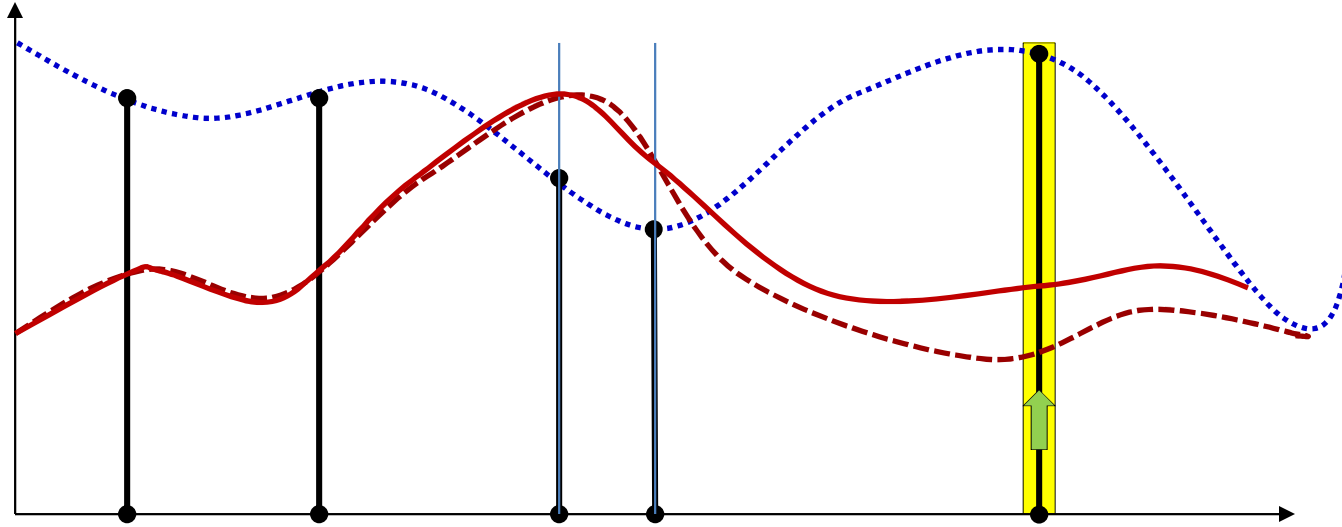
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

# 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

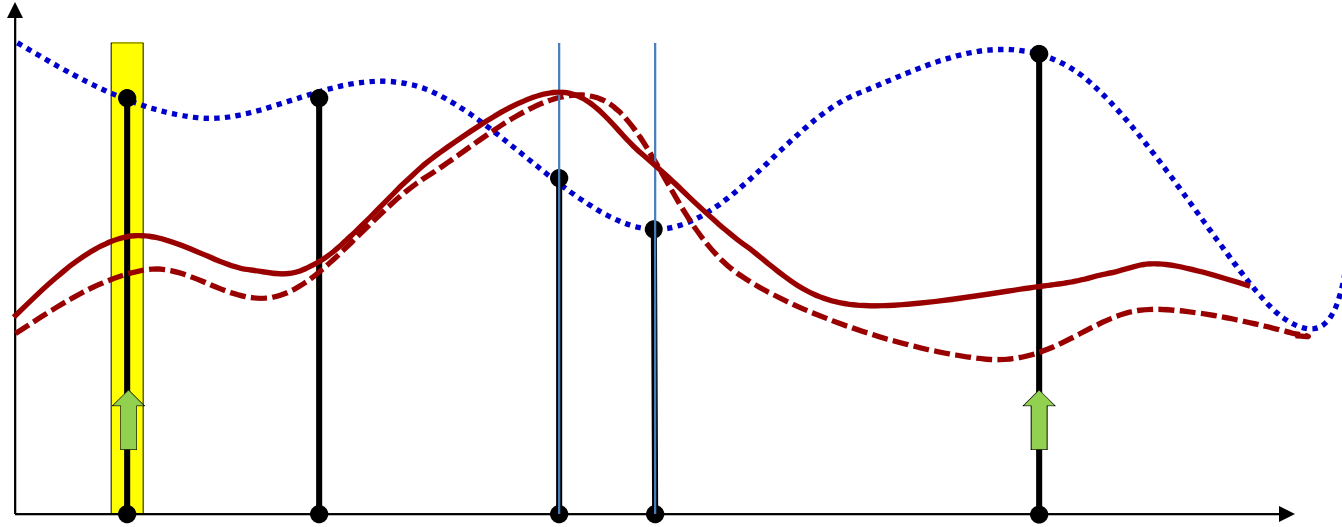
# Alternative: Incremental update



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

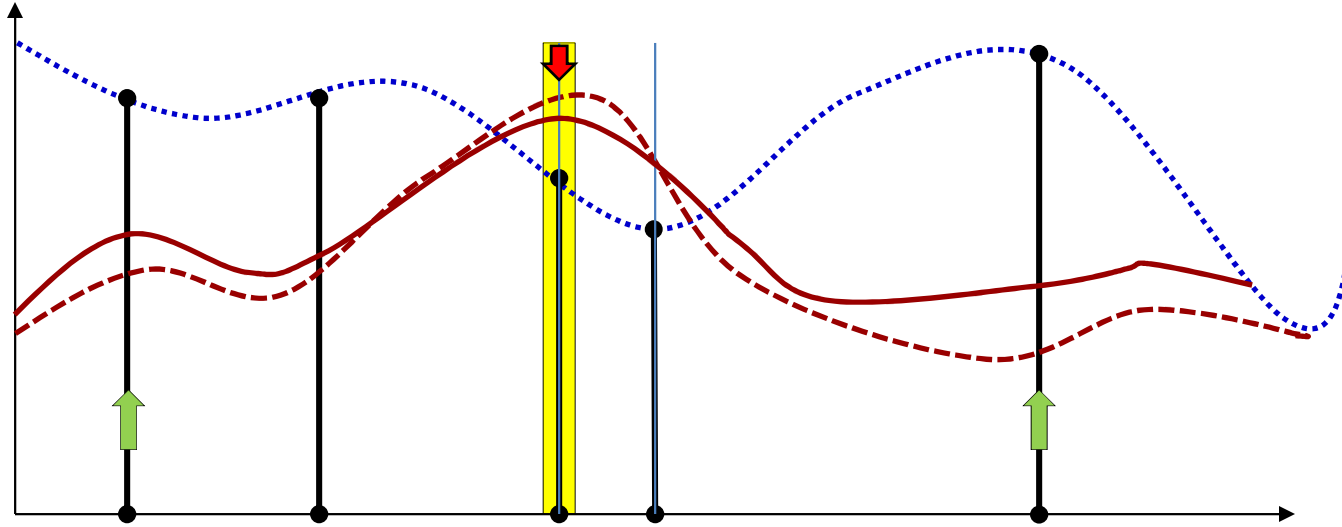


# Alternative: Incremental update



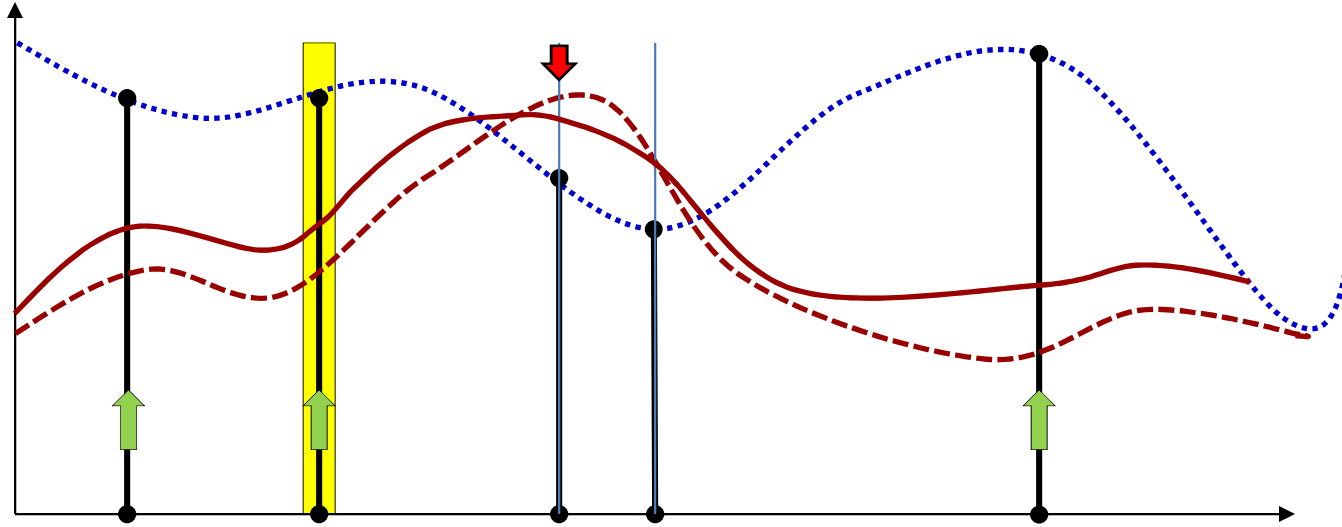
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

# Alternative: Incremental update



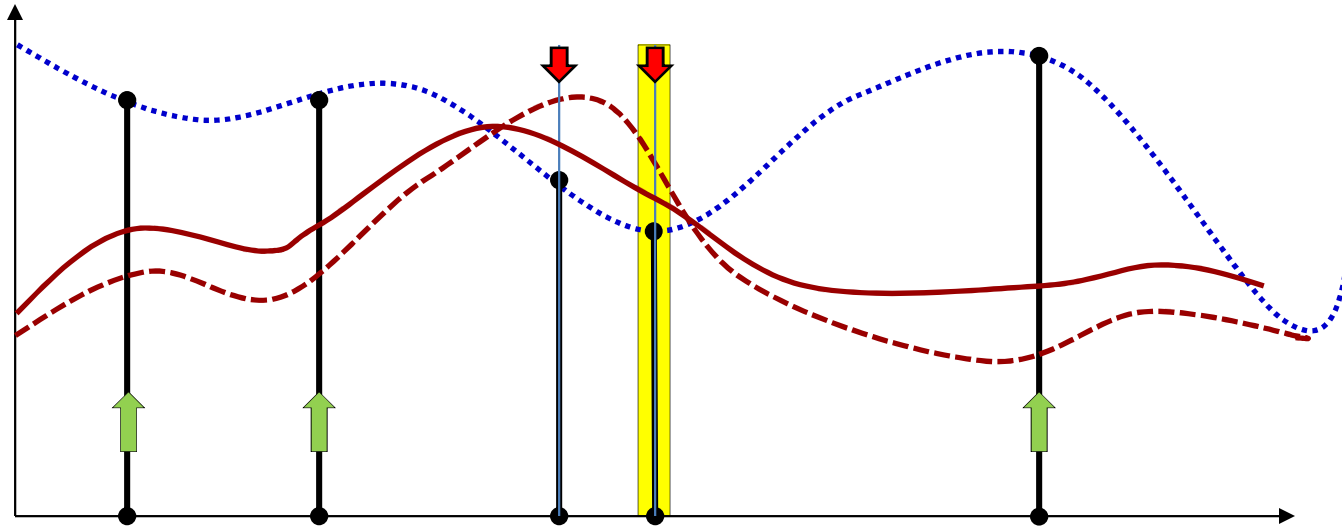
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

# Alternative: Incremental update



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

# Alternative: Incremental update

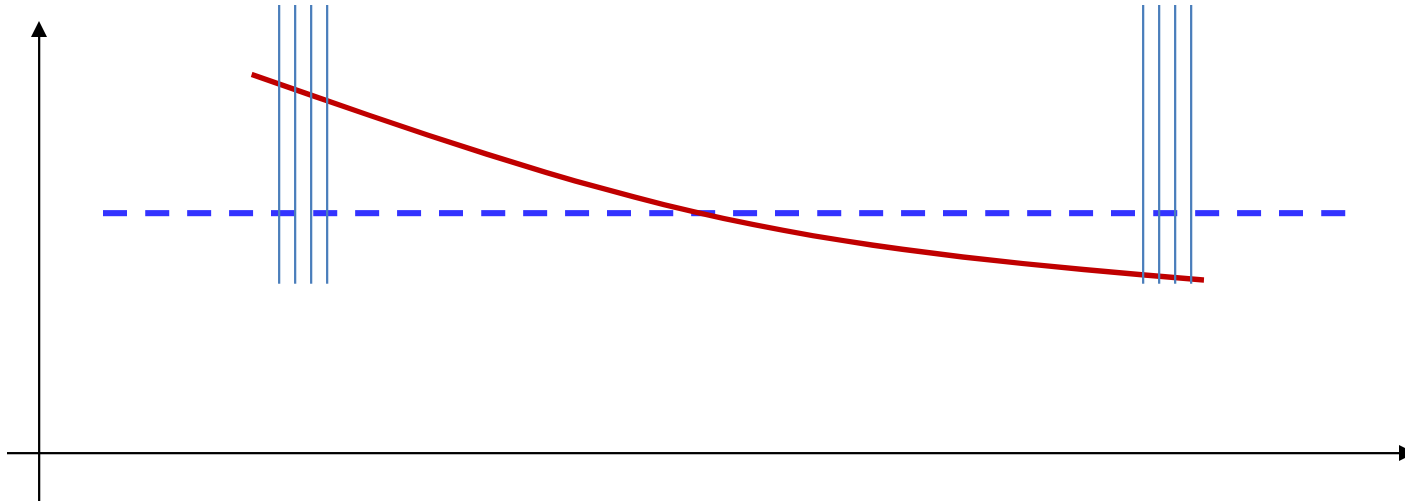


- Alternative: adjust the function at one training point at a time
  - Keep adjustments small
  - Eventually, when we have processed all the training points, we will have adjusted the entire function
    - With *greater* overall adjustment than we would if we made a single “Batch” update

# Incremental Update: Stochastic Gradient Descent

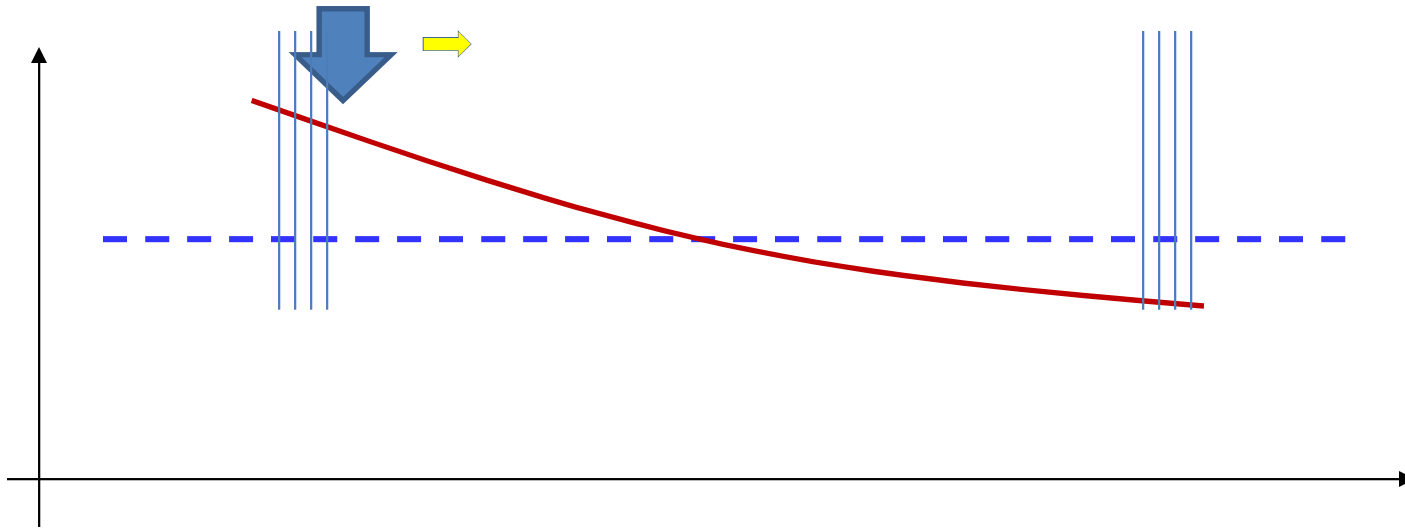
- Given  $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights  $W_1, W_2, \dots, W_K$
- Do:
  - For all  $t = 1:T$ 
    - For every layer  $k$ :
      - Compute  $\nabla_{W_k} \text{Div}(Y_t, d_t)$
      - Update
$$W_k = W_k - \eta \nabla_{W_k} \text{Div}(Y_t, d_t)$$
- Until  $Err$  has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

# Caveats: order of presentation



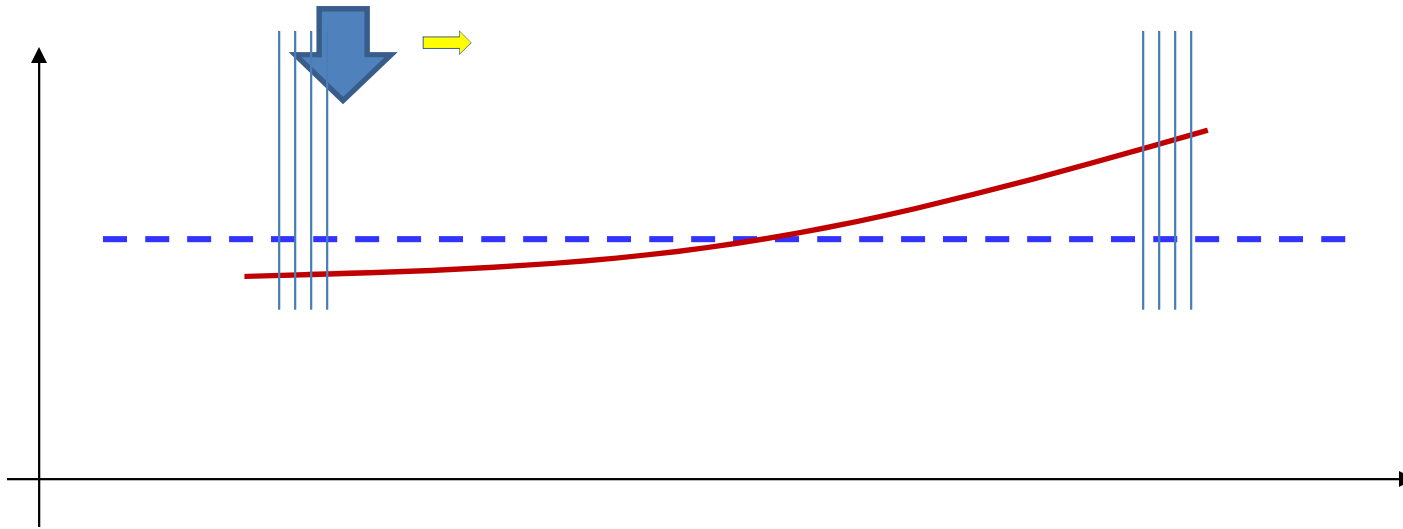
- If we loop through the samples in the same order, we may get *cyclic* behavior

# 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*

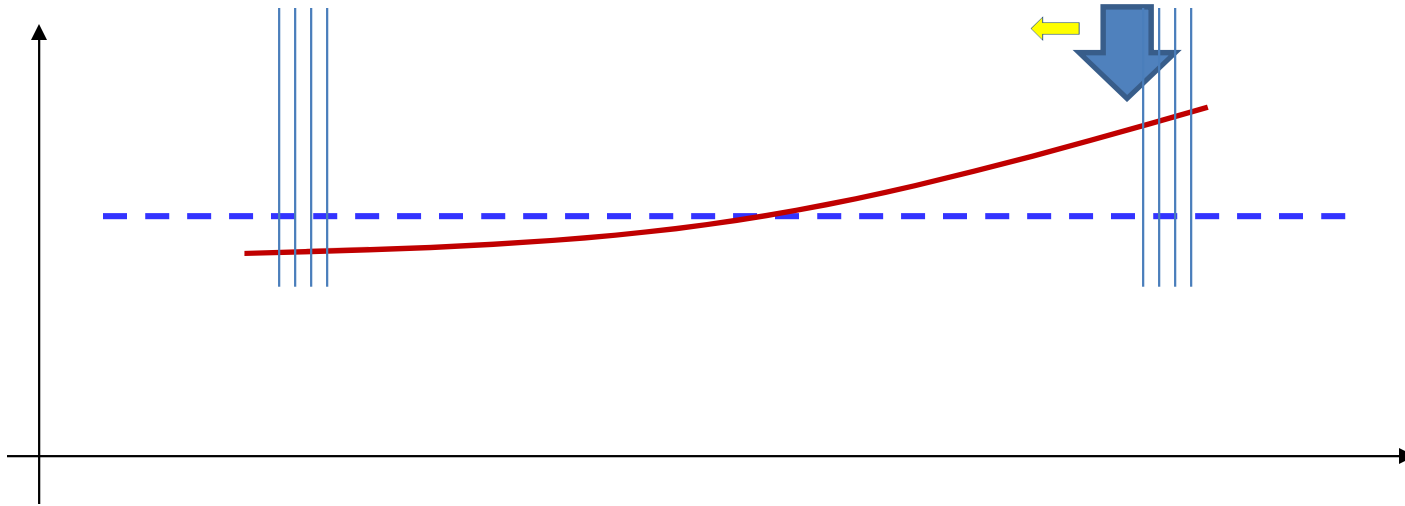
# Caveats: order of presentation



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

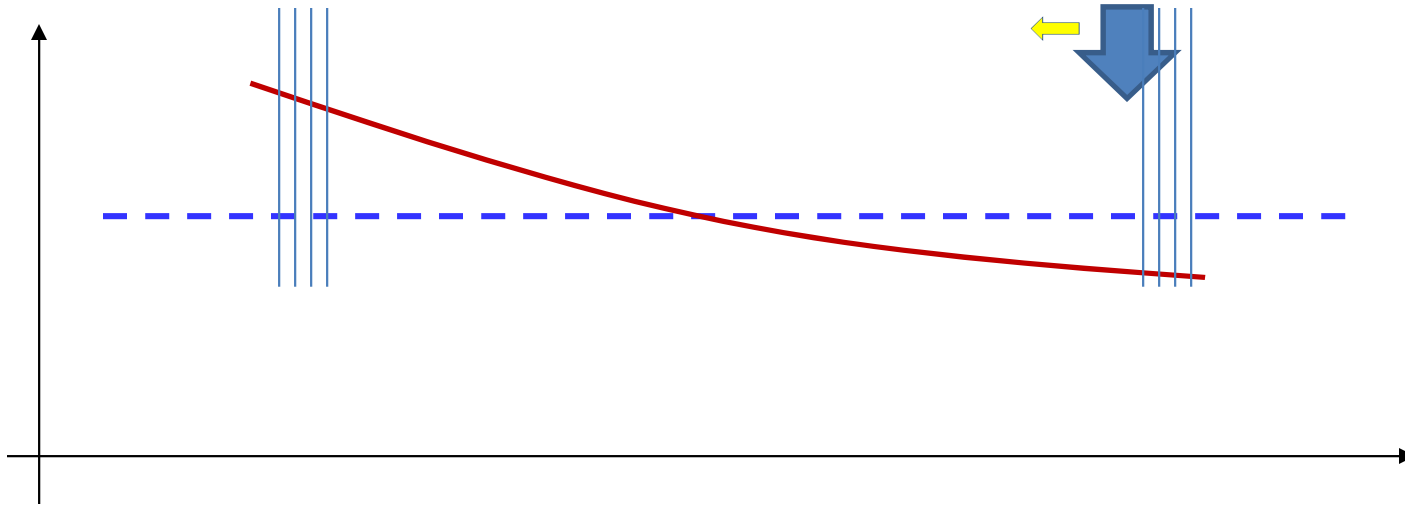


# Caveats: order of presentation



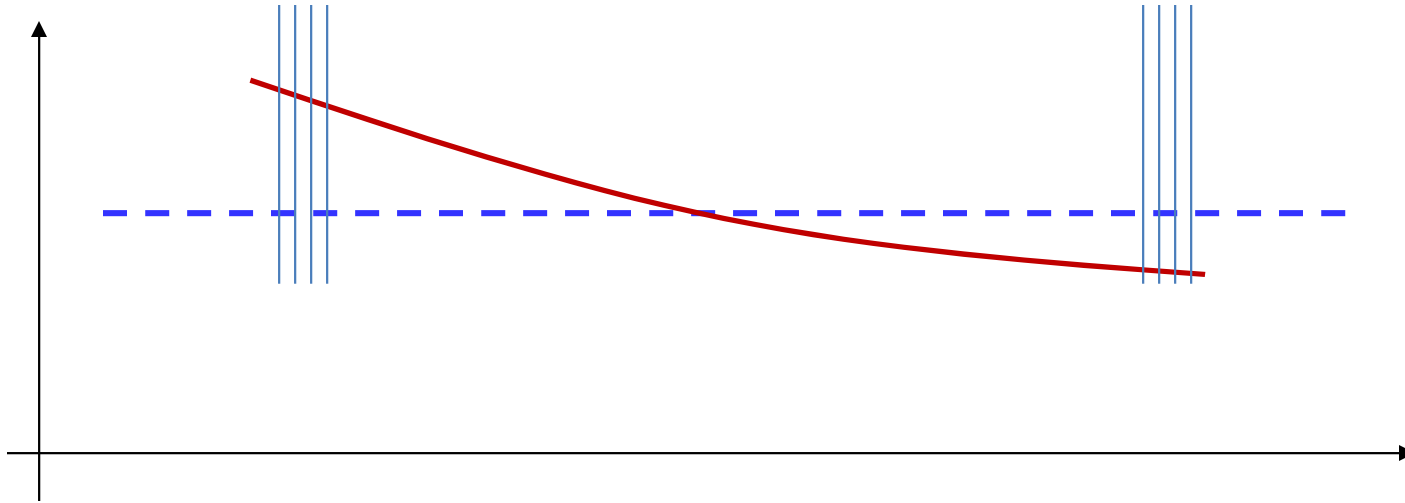
- If we loop through the samples in the same order, we may get *cyclic* behavior

# Caveats: order of presentation



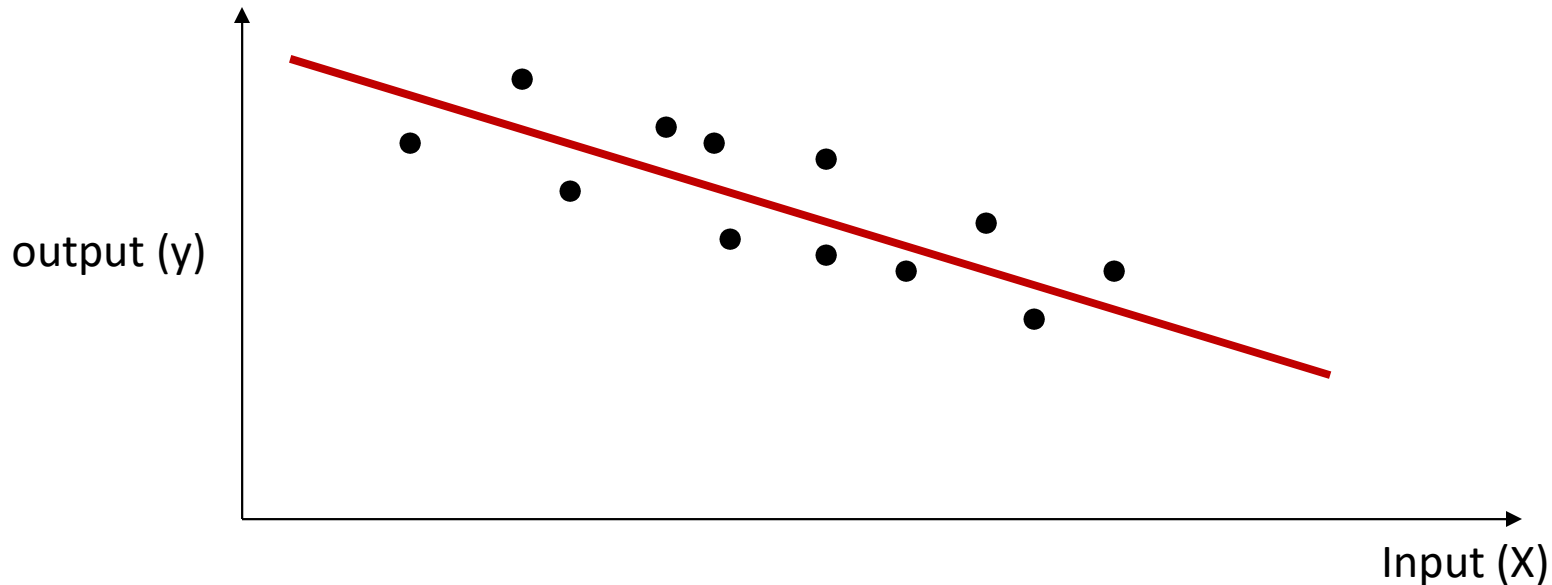
- If we loop through the samples in the same order, we may get *cyclic* behavior

# 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



# SGD convergence

- We will define convergence in terms of the number of iterations taken to get within  $\epsilon$  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  $\epsilon$  convergence as  $O\left(\frac{1}{\epsilon}\right)$

- For generically convex (but not strongly convex) function, various proofs report an  $\epsilon$  convergence of  $\frac{1}{\sqrt{k}}$  using a learning rate of  $\frac{1}{\sqrt{k}}$ .

# Batch gradient convergence

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

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

– Giving us the iterations to  $\epsilon$  convergence as  $O\left(\log\left(\frac{1}{\epsilon}\right)\right)$

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

# SGD Convergence: Loss value

If:

- $f$  is  $\lambda$ -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  $w_T$  at **any single iteration** to  $\mathcal{O}\left(\frac{\log(T)}{T}\right)$  for strongly convex loss or  $\mathcal{O}\left(\frac{\log(T)}{\sqrt{T}}\right)$  for convex loss
- Averaging schemes can improve the bound to  $\mathcal{O}\left(\frac{1}{T}\right)$  and  $\mathcal{O}\left(\frac{1}{\sqrt{T}}\right)$
- **Smoothness** of the loss is **not required**

# SGD Convergence and weight averaging

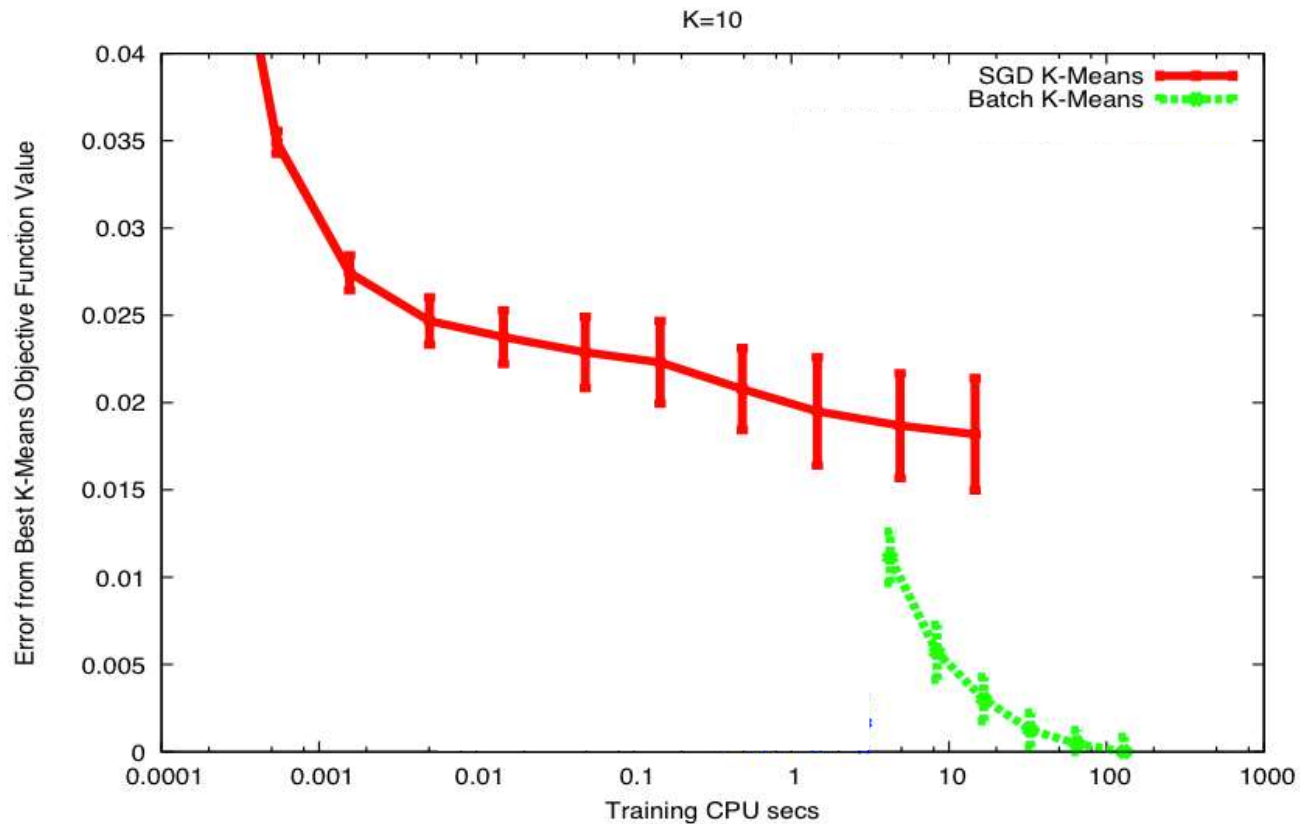
Polynomial Decay Averaging:

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

With  $\gamma$  some small positive constant, e.g.  $\gamma = 3$

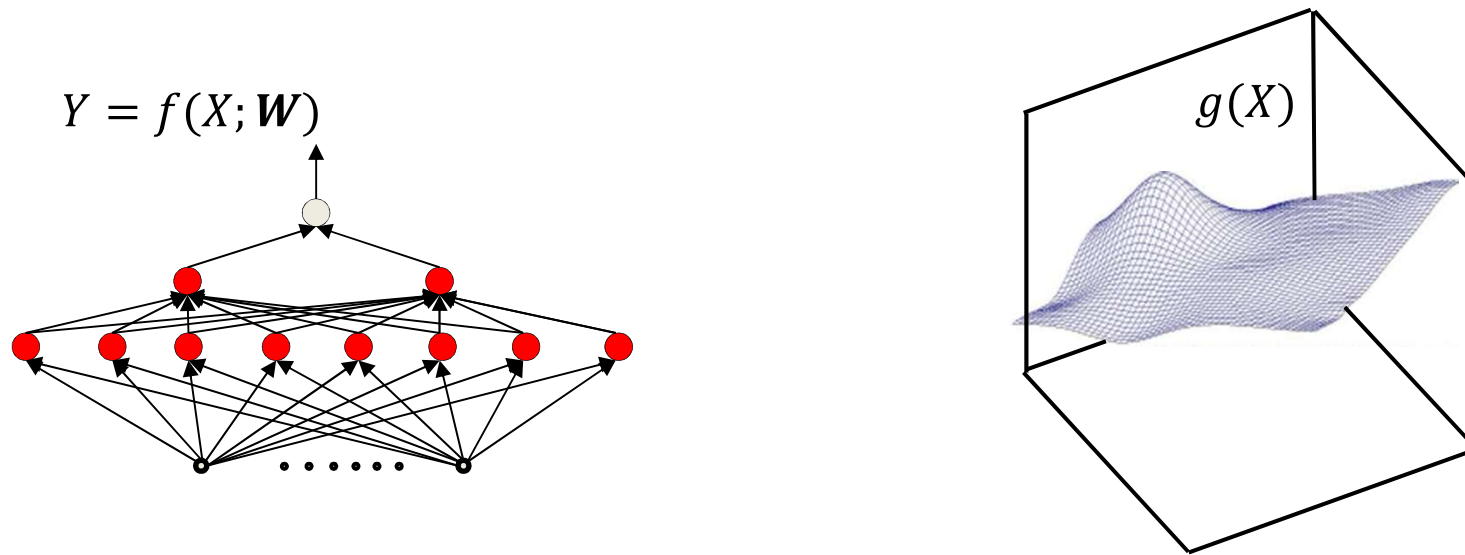
Achieves  $\mathcal{O}\left(\frac{1}{T}\right)$  (strongly convex) and  $\mathcal{O}\left(\frac{1}{\sqrt{T}}\right)$  (convex) convergence

# SGD example



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

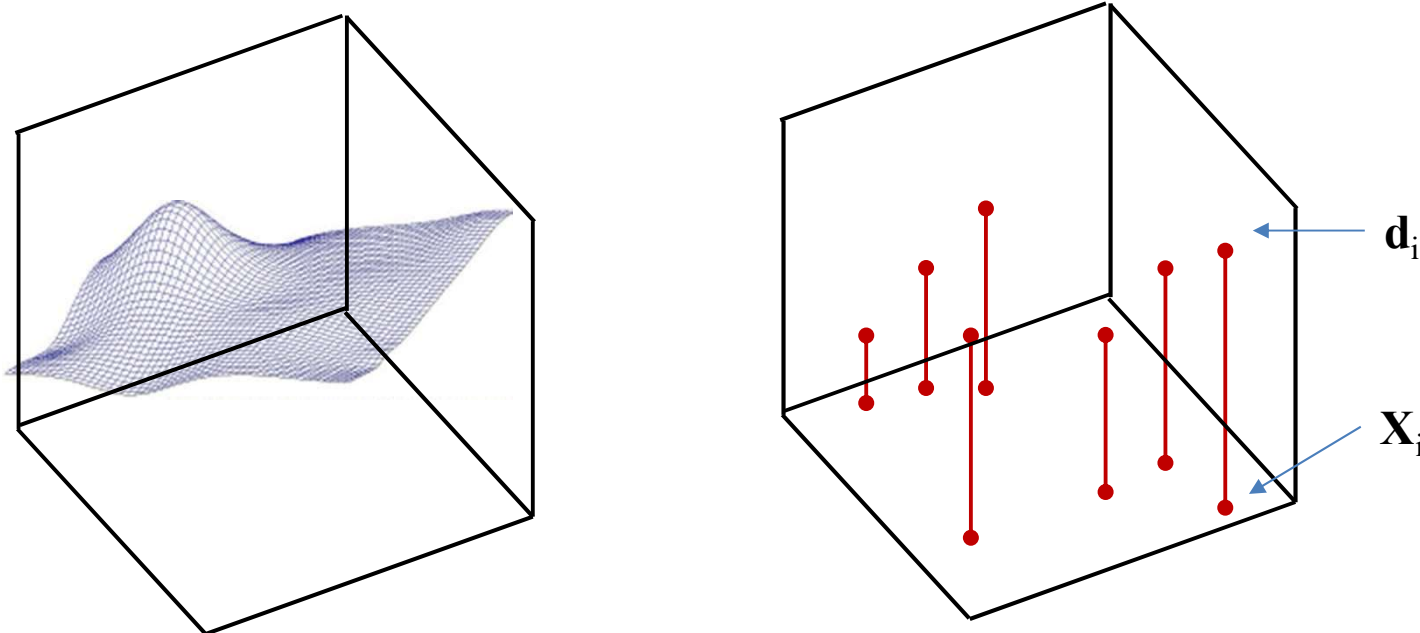
# Recall: Modelling a function



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

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

# Recall: The *Empirical* risk



- In practice, we minimize the *empirical error*

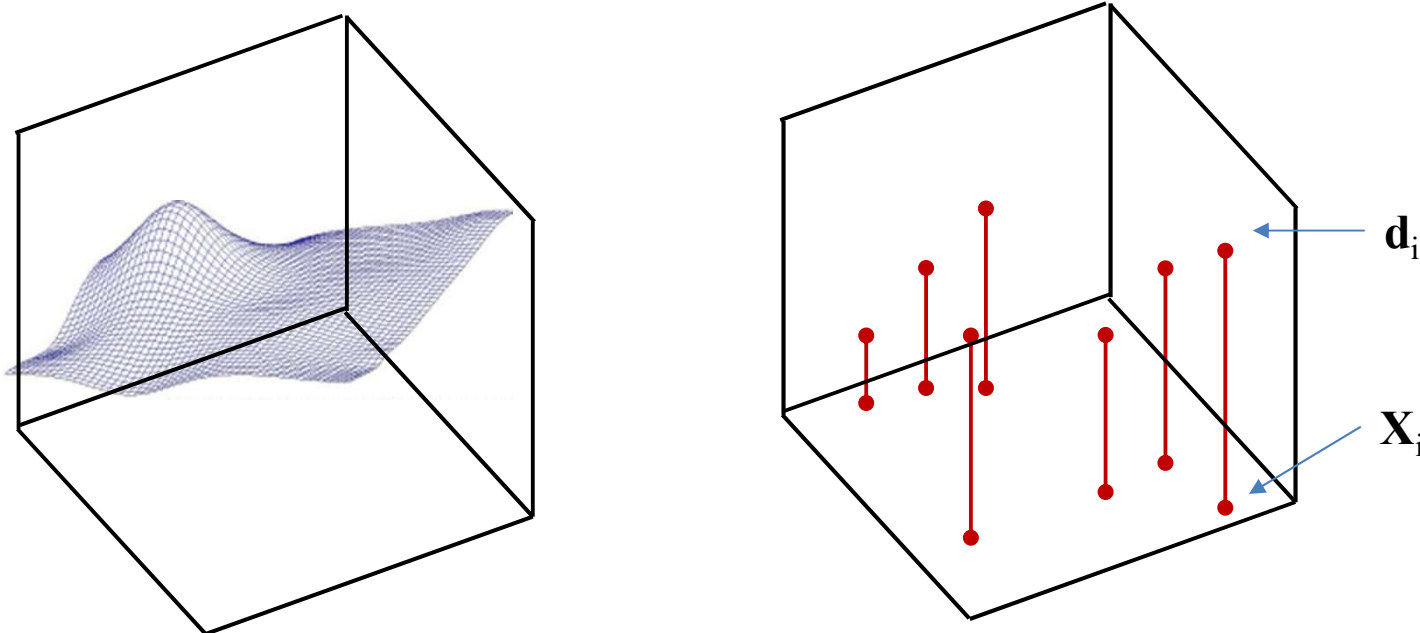
$$Err(f(X; W), g(X)) = \frac{1}{N} \sum_{i=1}^N div(f(X_i; W), d_i)$$
$$\widehat{W} = \underset{W}{\operatorname{argmin}} Err(f(X; W), g(X))$$

- 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*

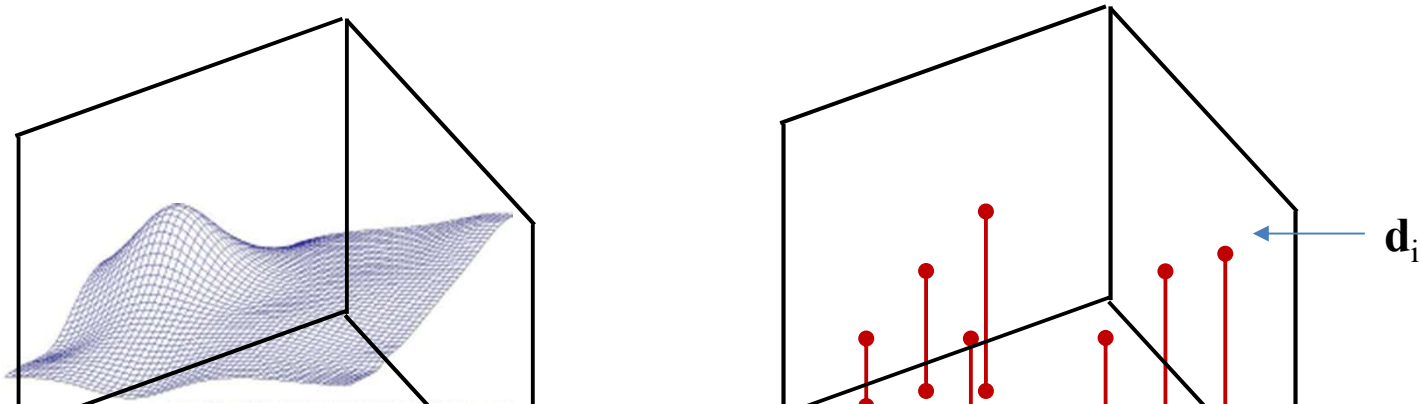
$$Err(f(X; W), g(X)) = \frac{1}{N} \sum_{i=1}^N div(f(X_i; W), d_i)$$

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

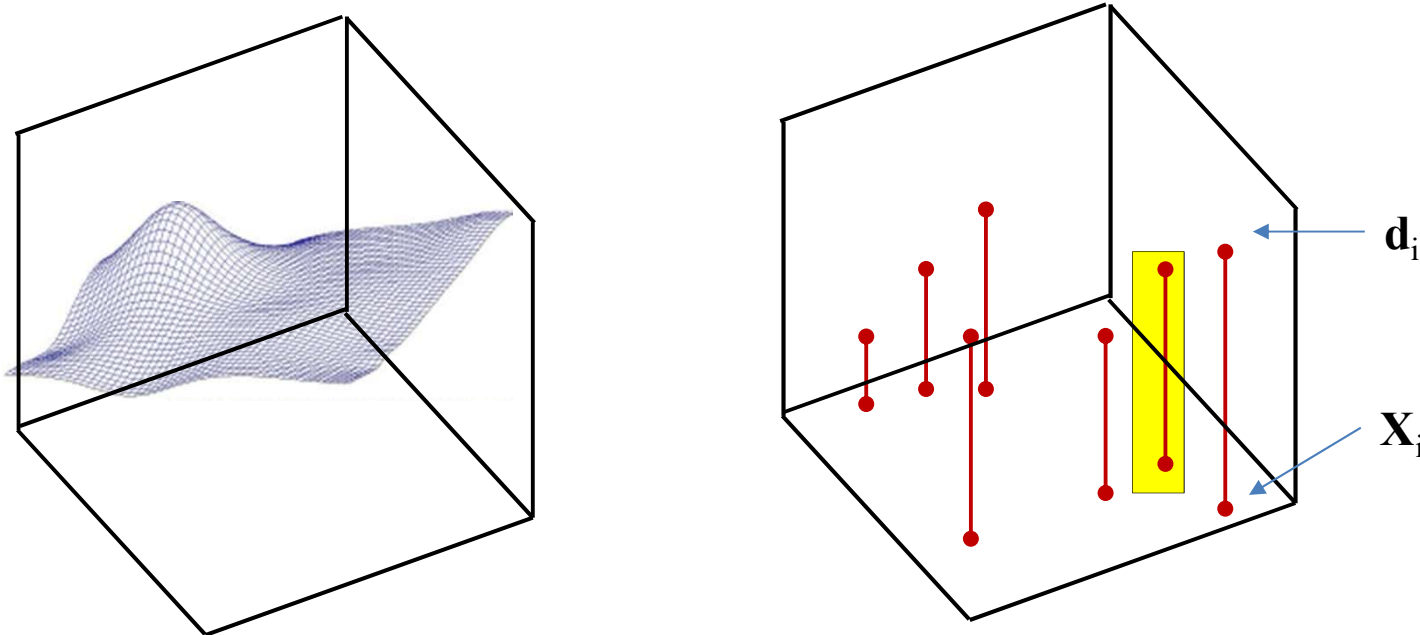
$$\text{Err}(f(X; W), g(X)) = \frac{1}{N} \sum_{i=1}^N \text{div}(f(X_i; W), d_i)$$

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

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

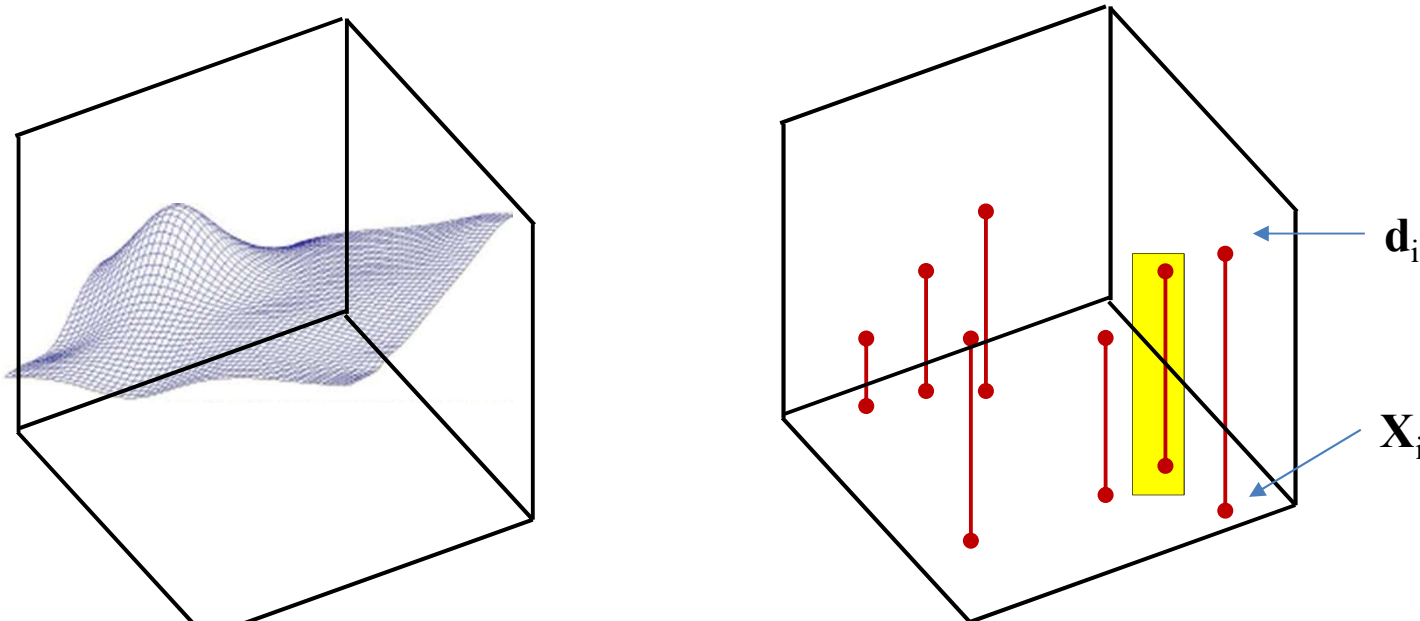
$$E[\text{Err}(f(X; W), g(X))] = E[\text{div}(f(X; W), g(X))]$$

# 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*  $E[div(f(X; W), g(X))]$

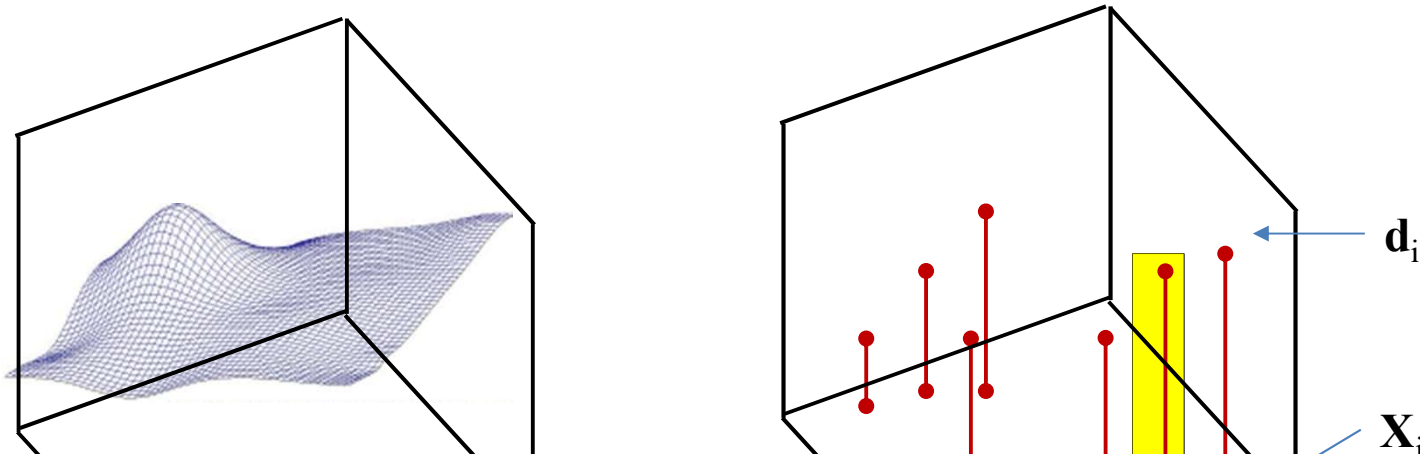
# SGD



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

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

# SGD

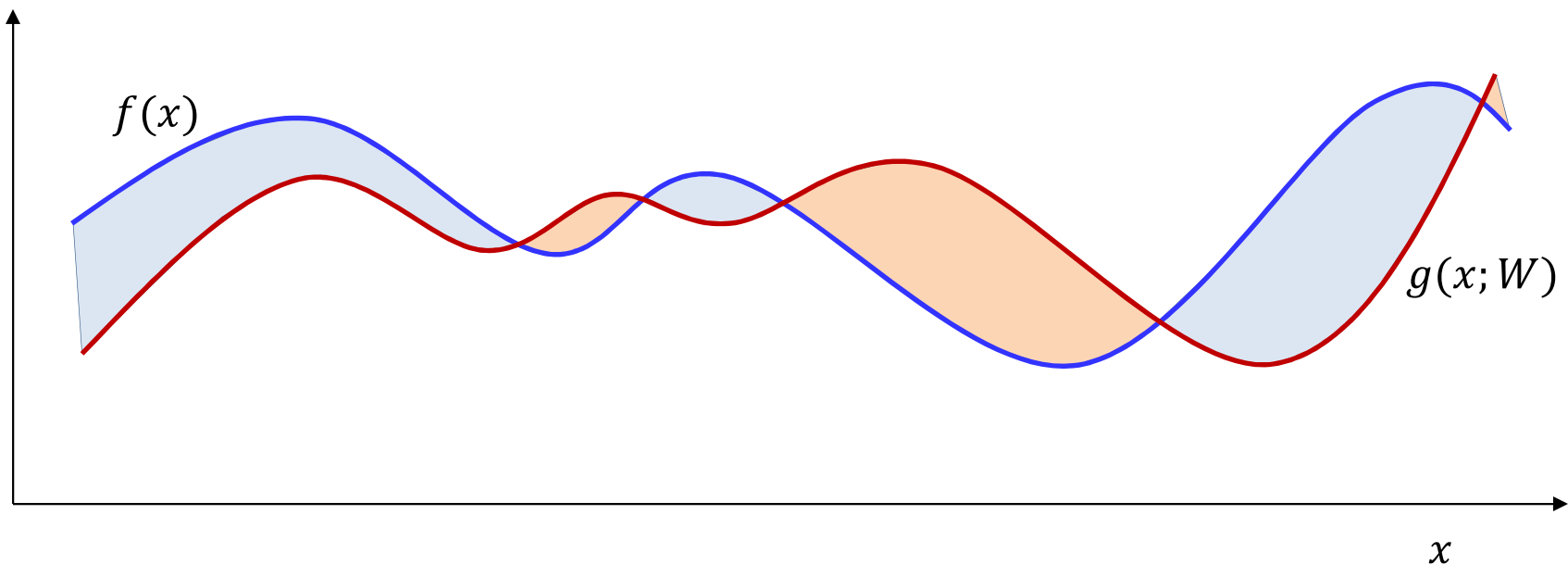


The variance of the sample error is the variance of the divergence itself:  $\text{var}(\text{div})$   
This is  $N$  times the variance of the empirical average minimized by batch update

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

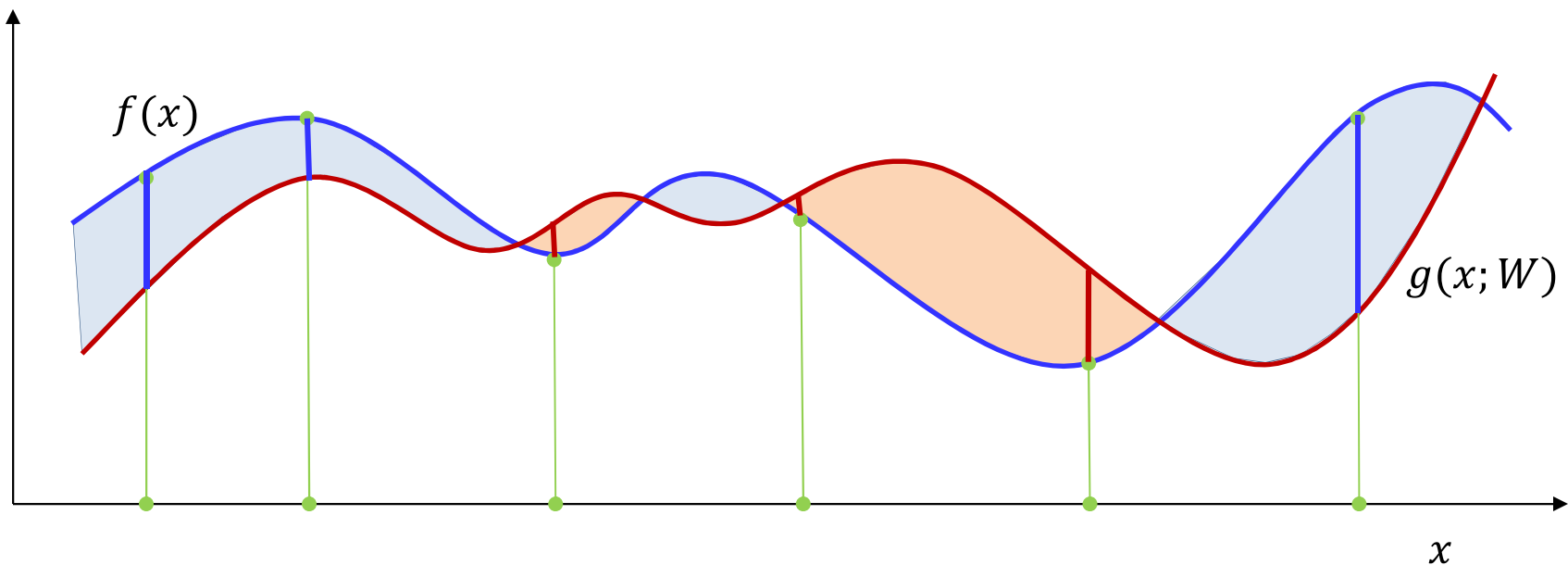
- At each iteration, **SGD** focuses on the divergence of a **single** sample  $\text{div}(f(X_i; W), d_i)$
- The *expected value* of the *sample error* is **still** the *expected divergence*  $E[\text{div}(f(X; W), g(X))]$

# Explaining the variance



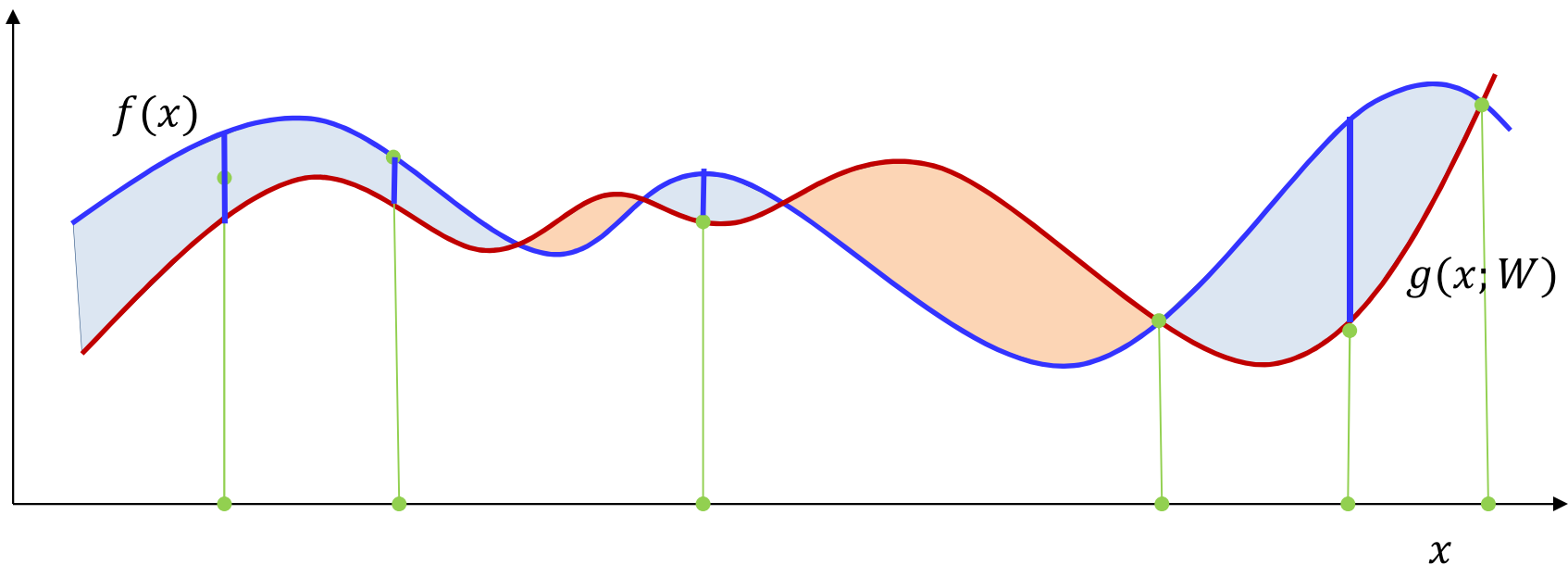
- The blue curve is the function being approximated
- The red curve is the approximation by the model at a given  $W$
- The heights of the shaded regions represent the point-by-point error
  - The divergence is a function of the error
  - We want to find the  $W$  that minimizes the average divergence

# Explaining the variance



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

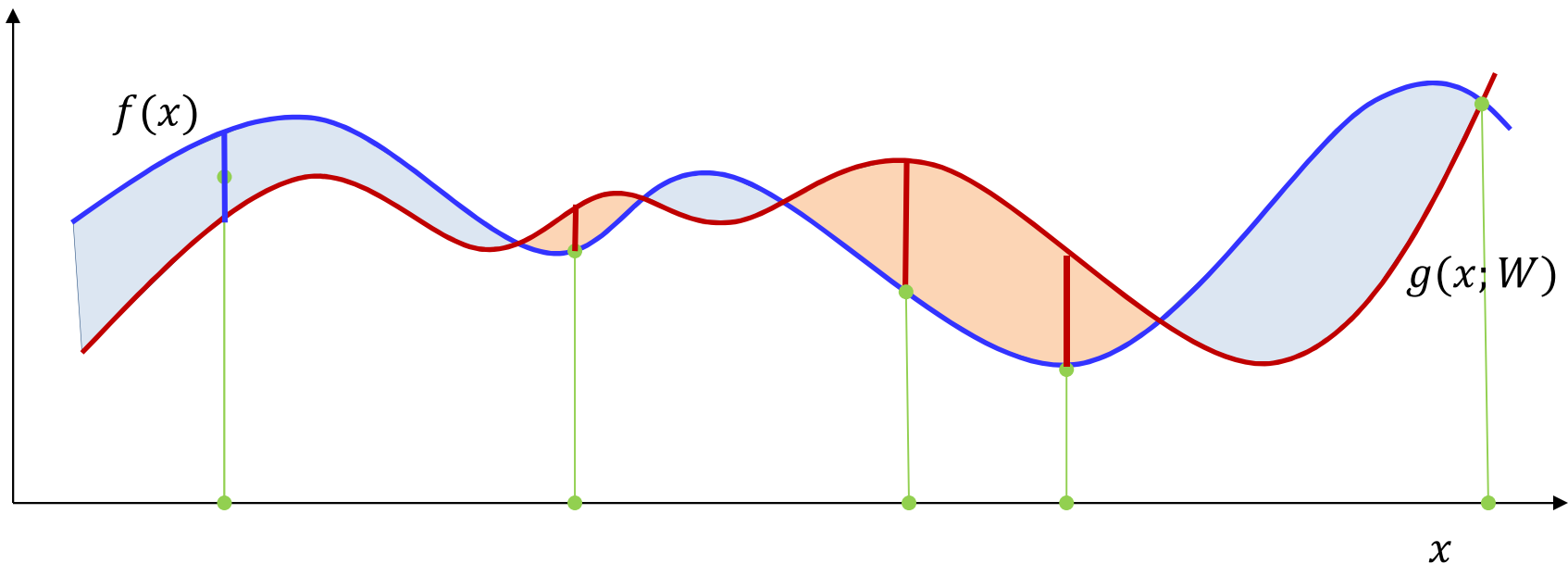
# Explaining the variance



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

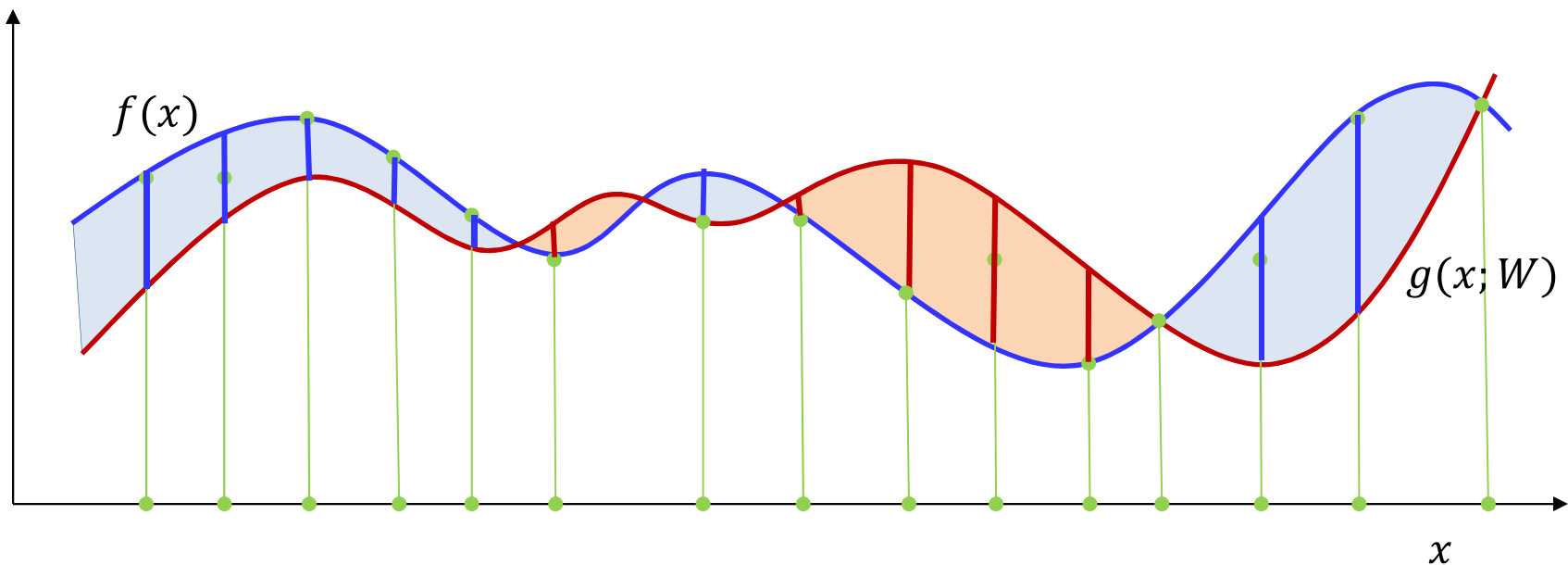


# Explaining the variance



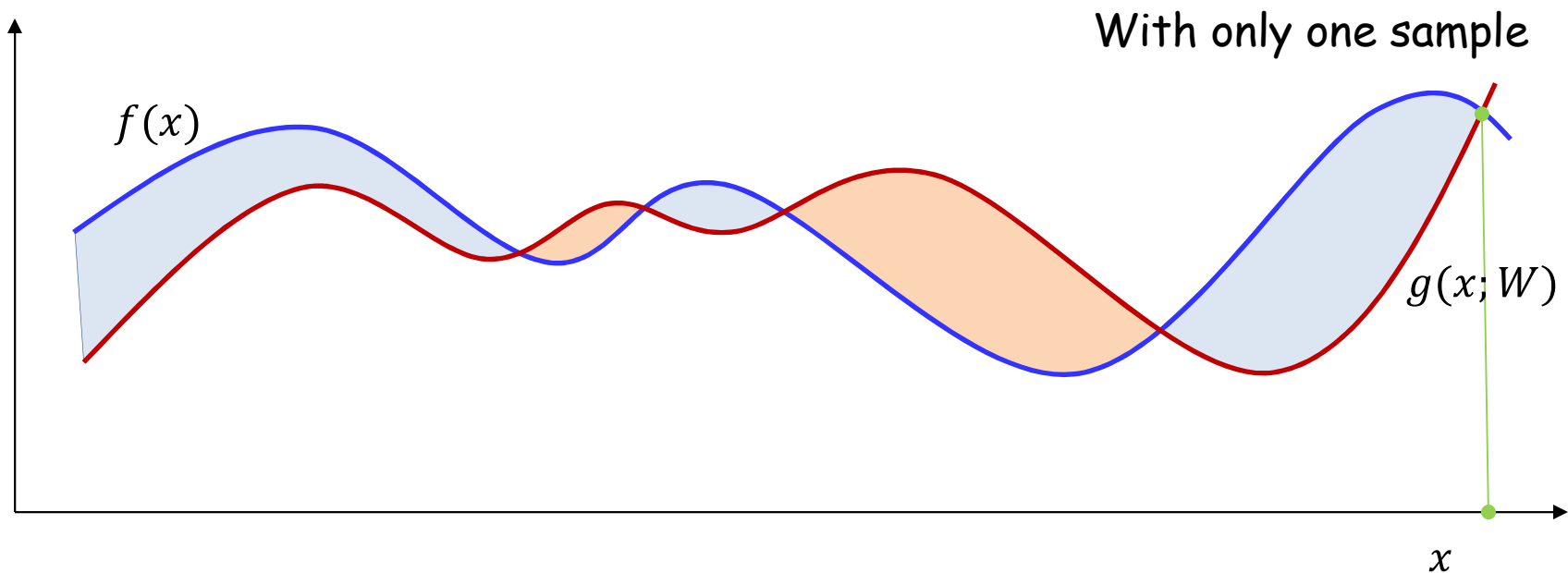
- Sample estimate approximates the shaded area with the average length of the lines
- This average length will change with position of the samples

# Explaining the variance



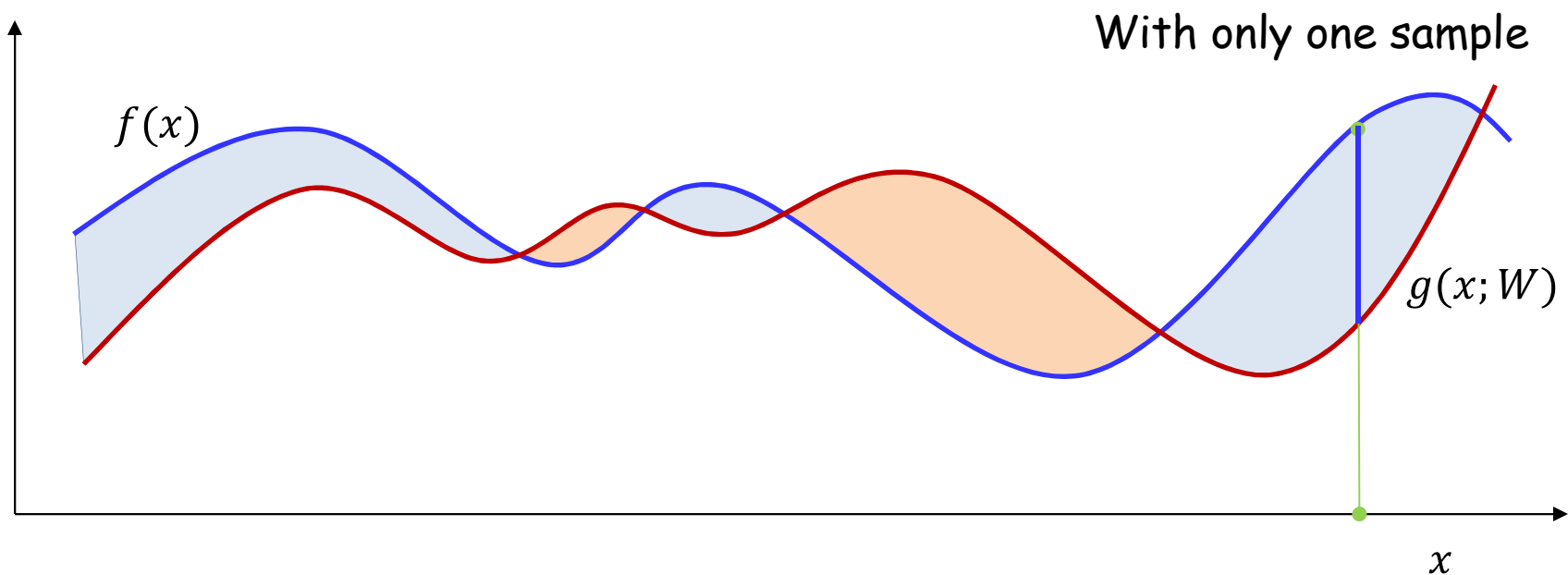
- Having more samples makes the estimate more robust to changes in the position of samples
  - The variance of the estimate is smaller

# Explaining the variance



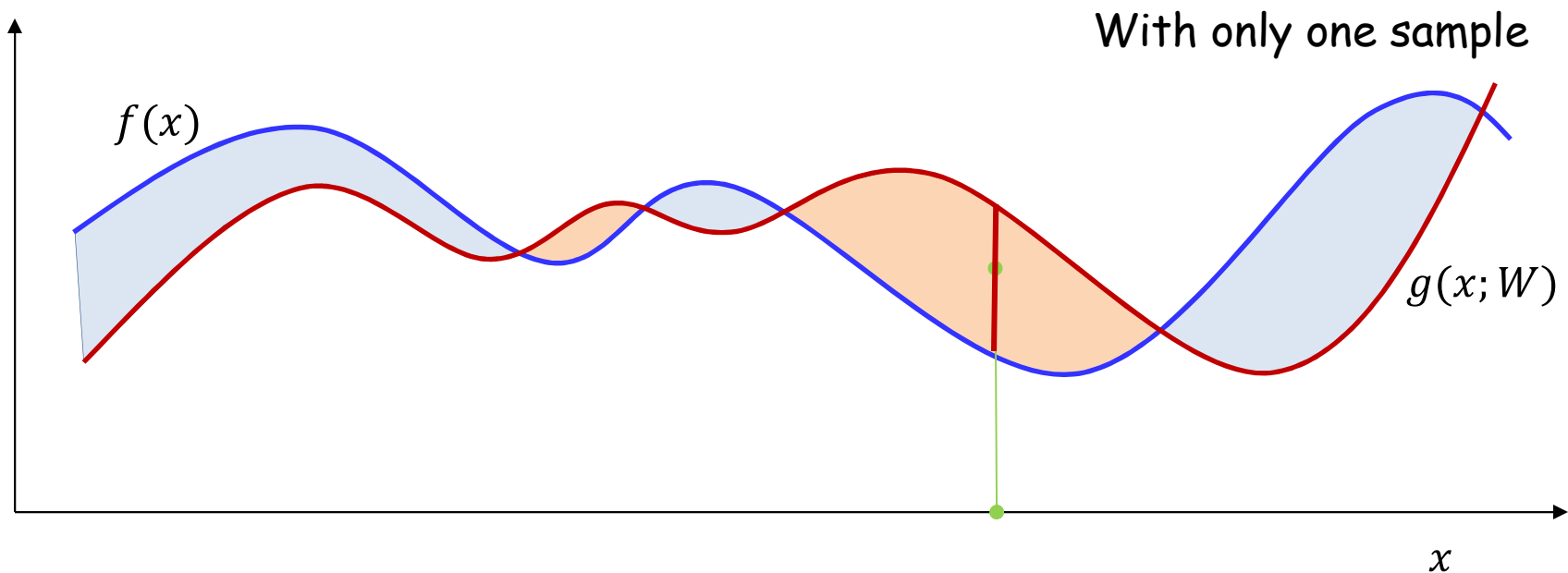
- Having very few samples makes the estimate swing wildly with the sample position
  - Since our estimator learns the  $W$  to minimize this estimate, the learned  $W$  too can swing wildly

# Explaining the variance



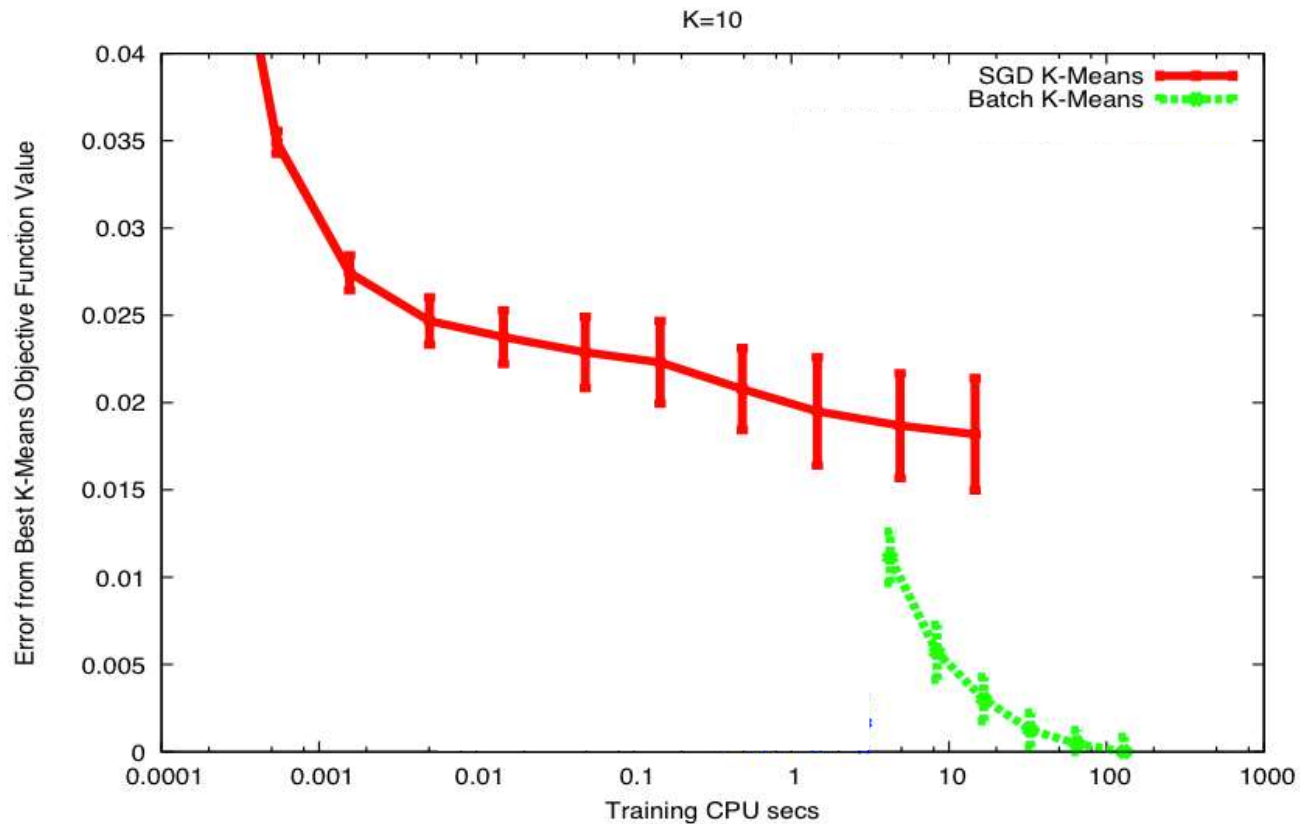
- Having very few samples makes the estimate swing wildly with the sample position
  - Since our estimator learns the  $W$  to minimize this estimate, the learned  $W$  too can swing wildly

# 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

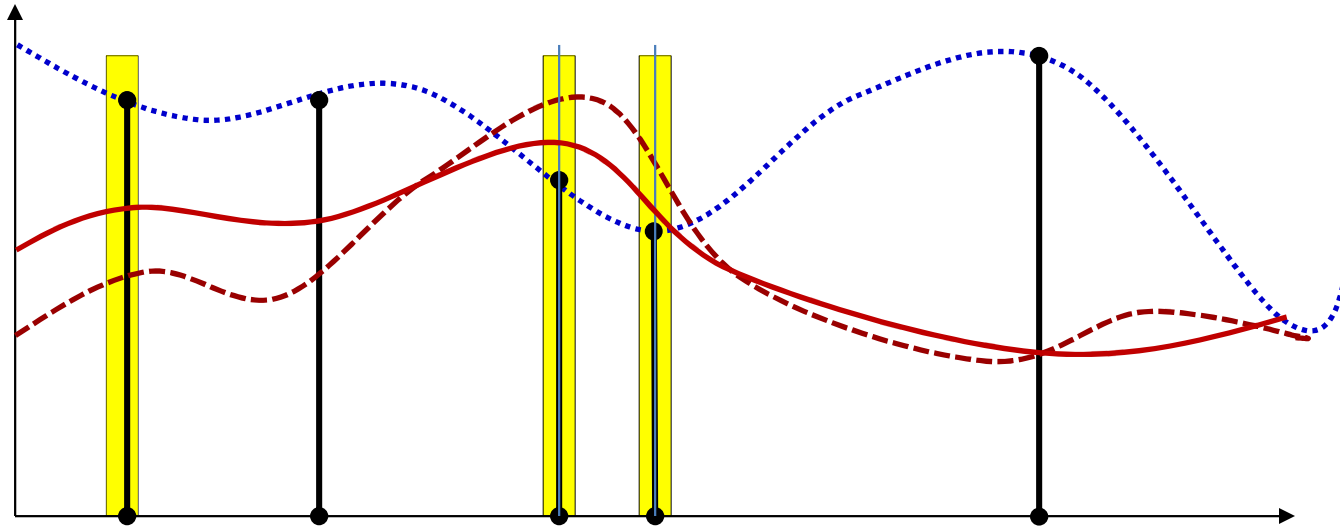


- A simpler problem: K-means
- Note: SGD converges slower
- Also has large variation between runs

# 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



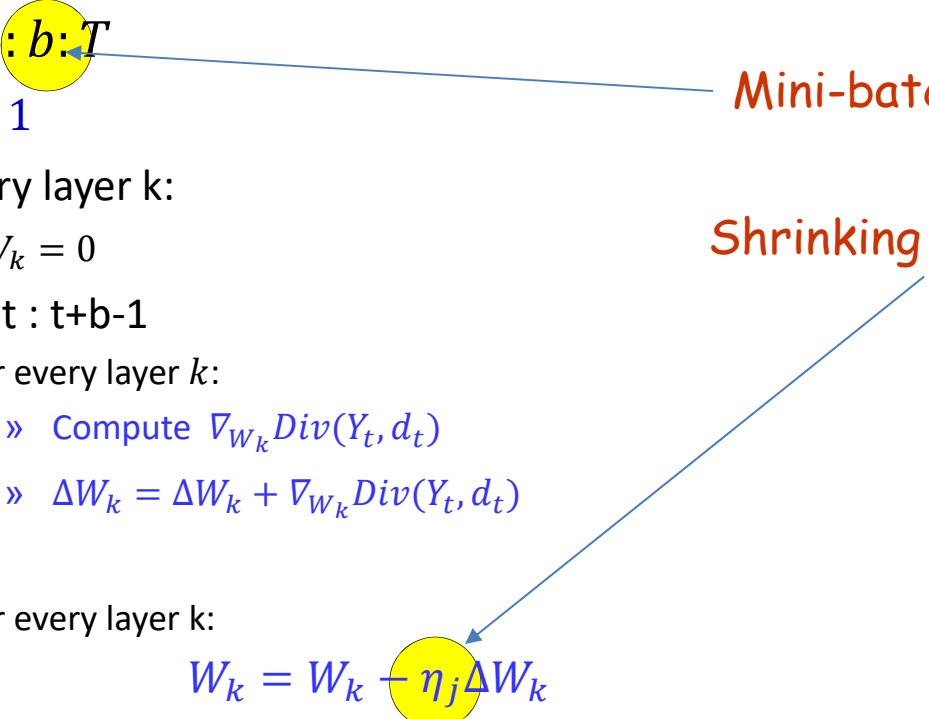
- Alternative: adjust the function at a small, randomly chosen subset of points
  - Keep adjustments small
  - If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data



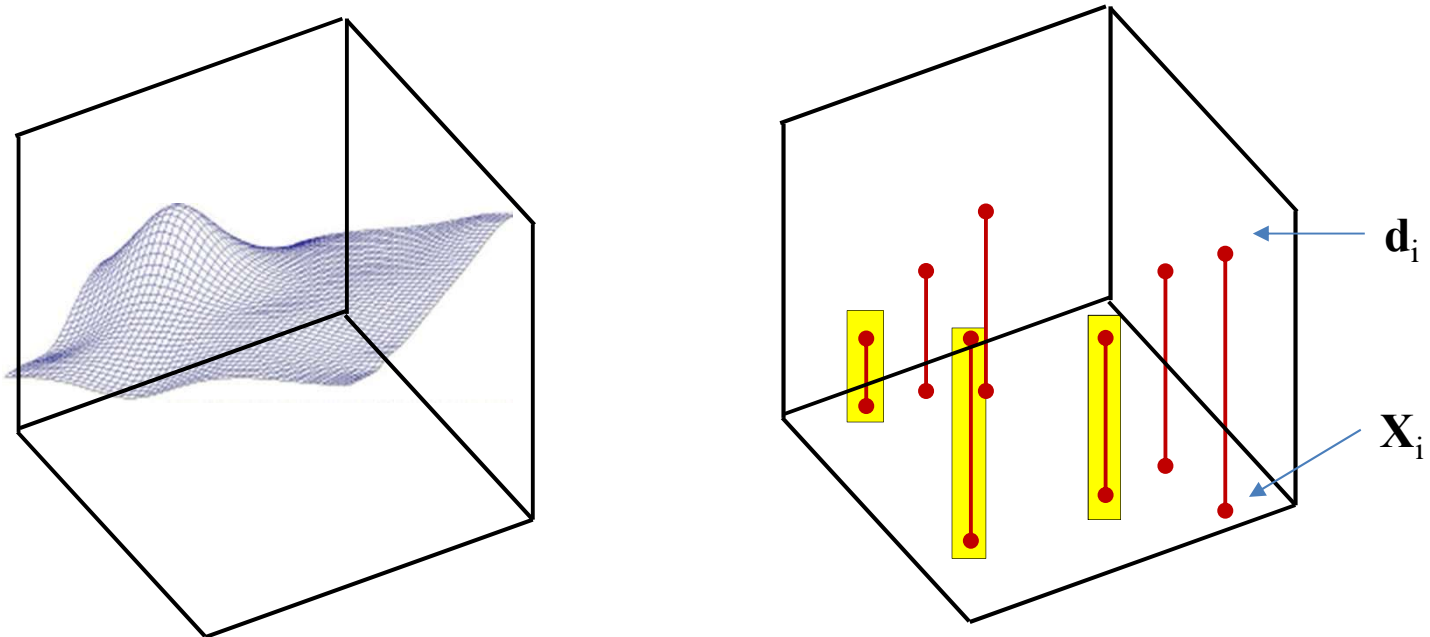
# Incremental Update: Mini-batch update

- 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



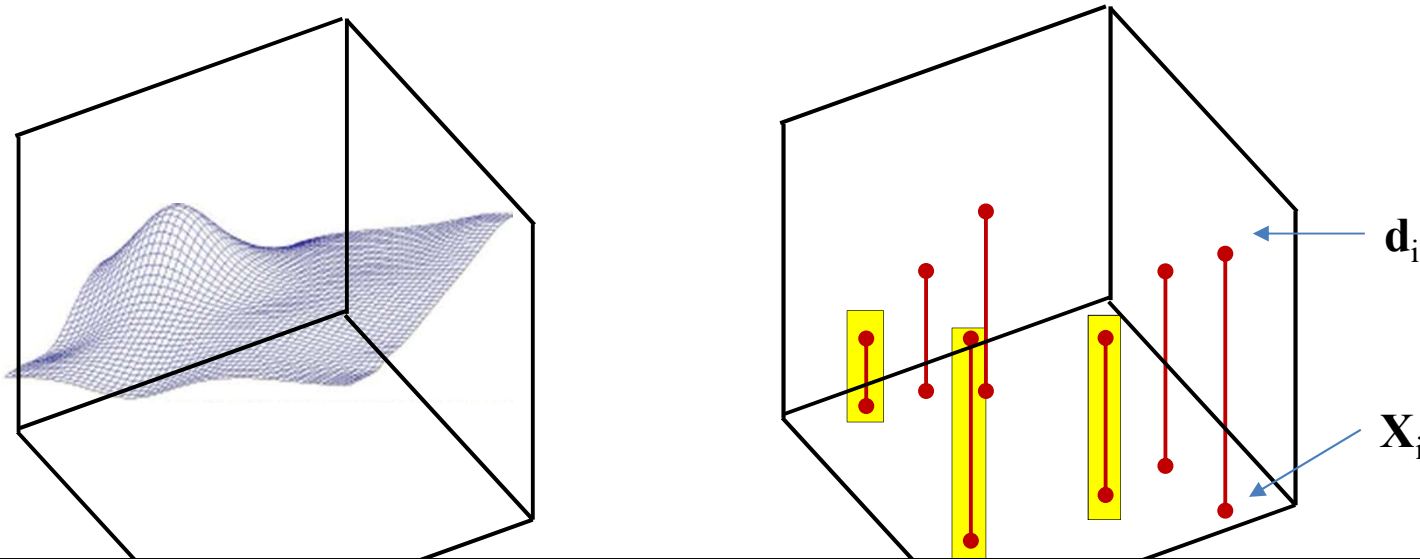
- Mini-batch updates compute and minimize a *batch error*

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

- 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



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

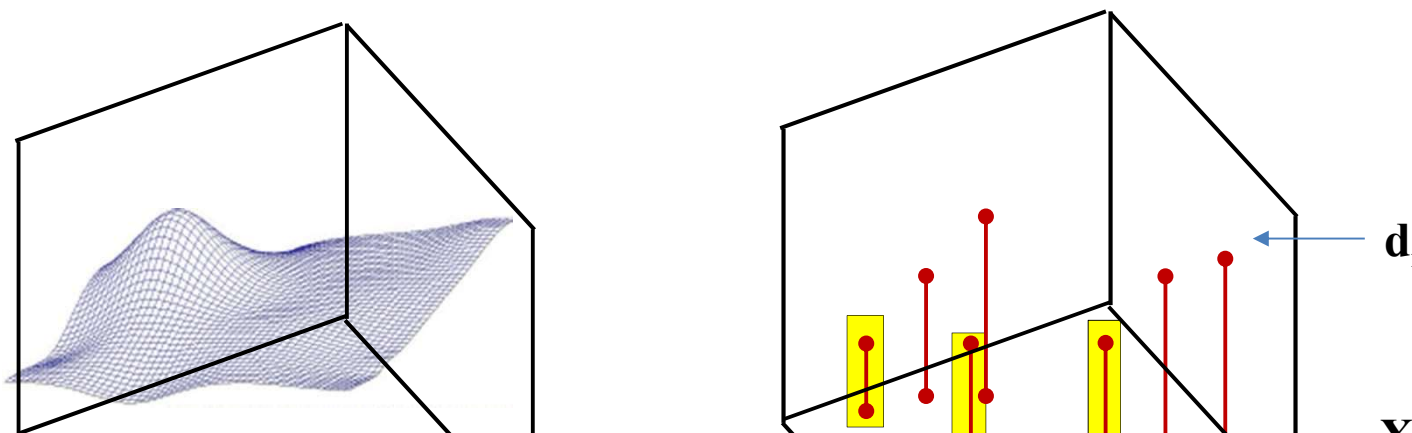
- Mini-batch updates computes an *empirical batch error*

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

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

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

# Mini Batches



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

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

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

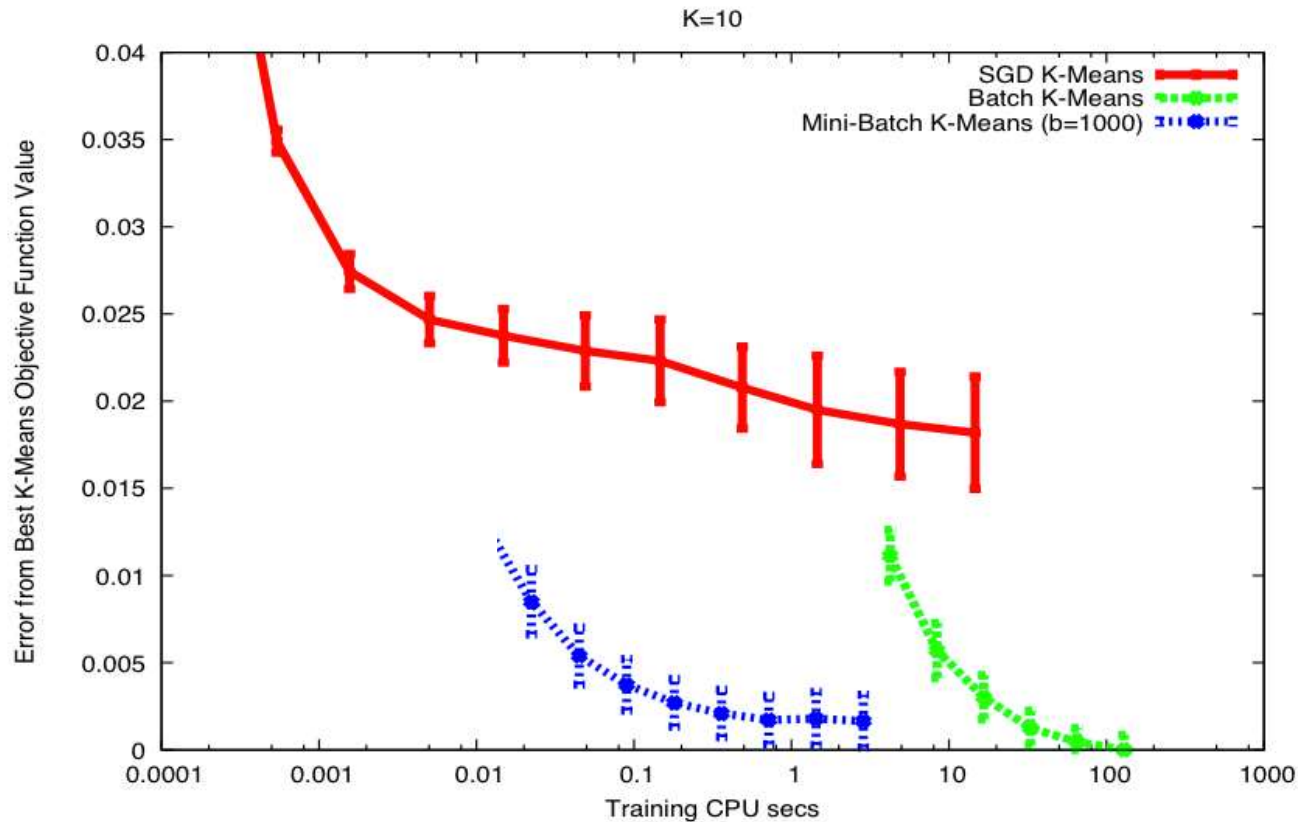
- The *expected value* of the *batch error* is also the *expected divergence*

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

# 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

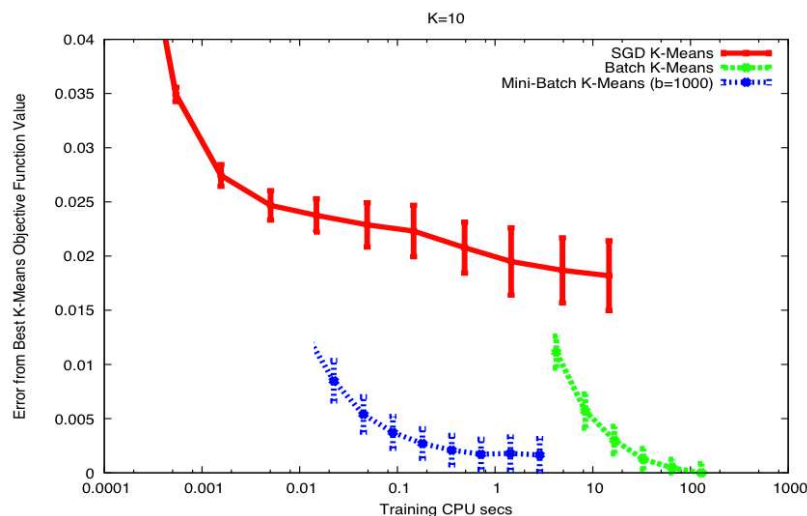
# SGD example



- 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 is 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

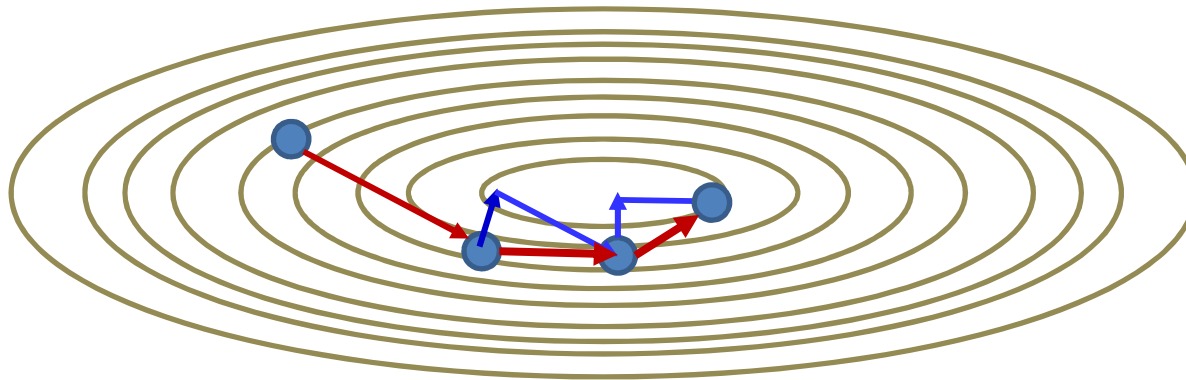
# Training and minibatches

- 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

# Moving on: Topics for the day

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

# Recall: Momentum

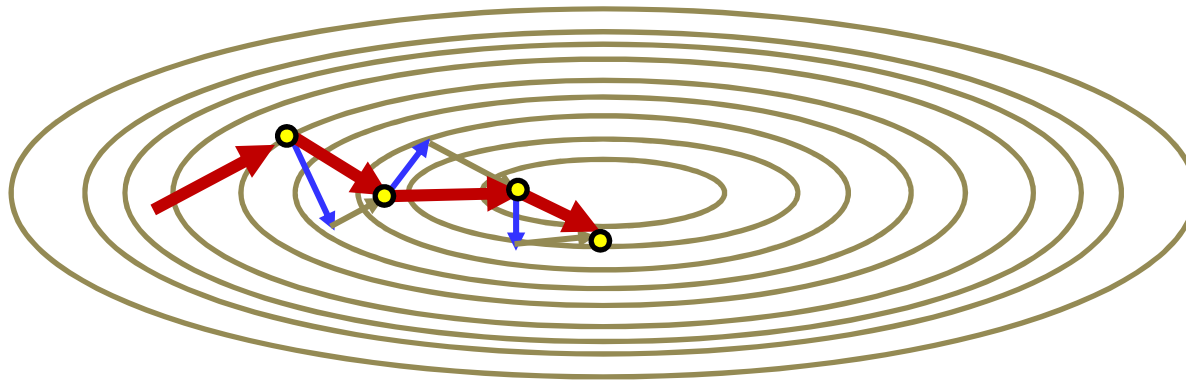


- The momentum method

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

- Updates using a running average of the gradient

# Momentum and incremental updates



- The momentum method

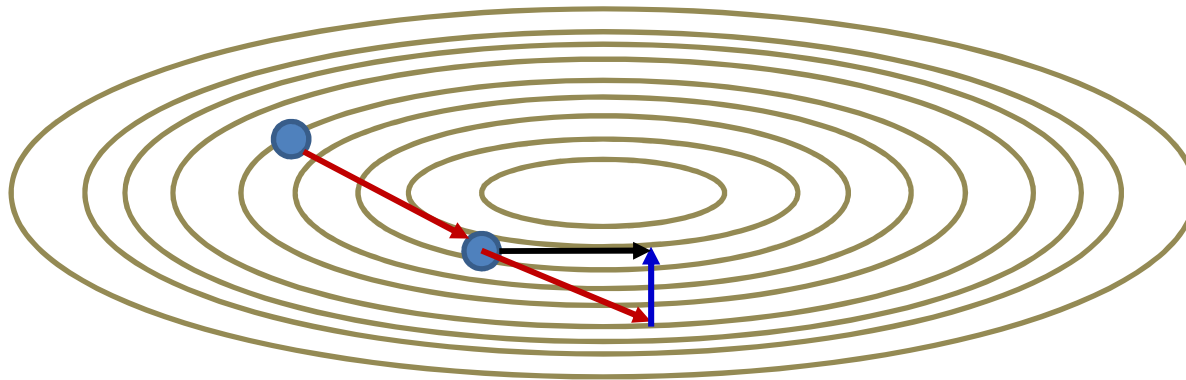
$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} + \eta \nabla_W \text{Err}(W^{(k-1)})$$

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

# Training with momentum

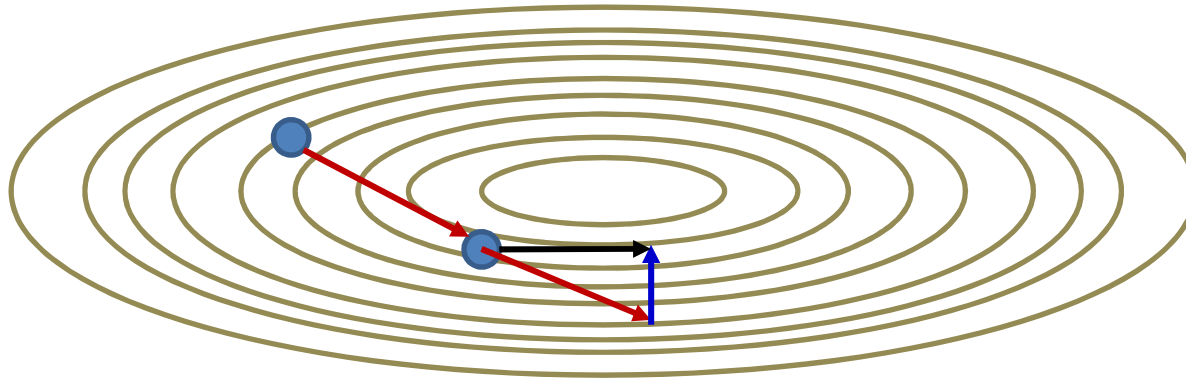
- Initialize all weights  $\mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_K$
- Do:
  - For all layers  $k$ , initialize  $\nabla_{W_k} Err = 0, \Delta W_k = 0$
  - For all  $t = 1:T$ 
    - For every layer  $k$ :
      - Compute gradient  $\nabla_{W_k} Div(Y_t, d_t)$
      - $\nabla_{W_k} Err += \nabla_{W_k}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

# Nestorov's Accelerated Gradient



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

# Nestorov's Accelerated Gradient



- Nestorov's method

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

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



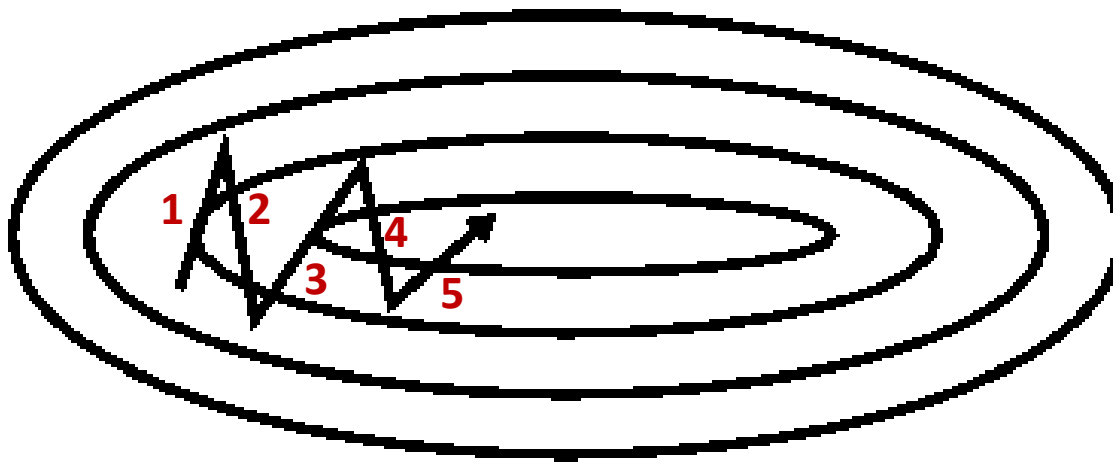
# Training with Nestorov's

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

# 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

# Smoothing the trajectory



Step	X component	Y component
1	1	+2.5
2	1	-3
3	3	+2.5
4	1	-2
5	2	1.5

- Simple gradient and acceleration methods still demonstrate oscillatory behavior in some directions
- 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
- Improvement: Dampen step size in directions with high motion
  - *Second order term*

# 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
  - *According to their variation (and not just their average)*
- 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:

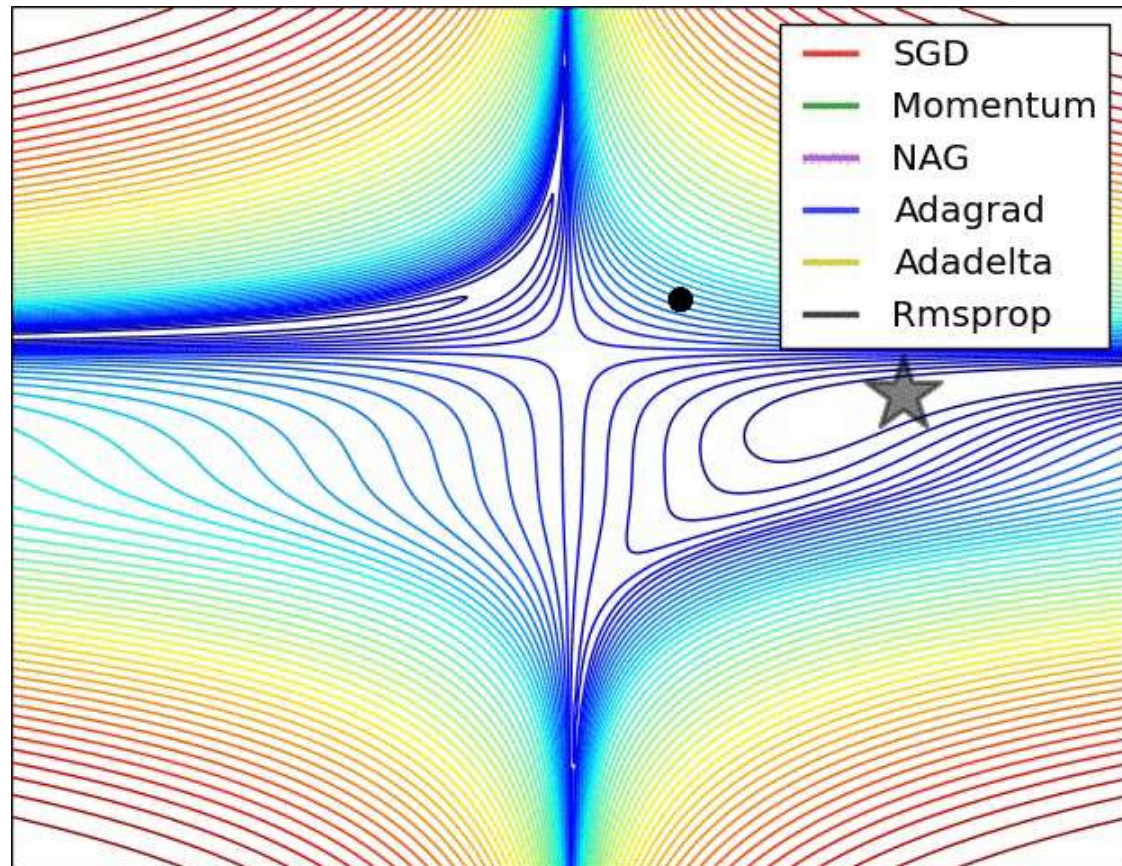
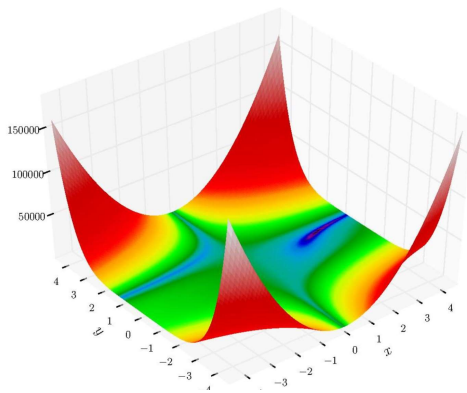
$$E[\partial_w^2 D]_k = \gamma E[\partial_w^2 D]_{k-1} + (1 - \gamma)(\partial_w^2 D)_k$$
$$w_{k+1} = w_k - \frac{\eta}{\sqrt{E[\partial_w^2 D]_k + \epsilon}} \partial_w D$$
  - $k = k + 1$
- Until  $E(W^{(1)}, W^{(2)}, \dots, W^{(K)})$  has converged

# Other variants of the same theme

- Many:
  - Adagrad
  - AdaDelta
  - ADAM
  - AdaMax
  - ...
- Note: no need to decide a learning rate schedule
  - Automatically updated
  - Its not even necessary to set the initial learning rate  $\eta$  in most cases
    - Setting it to 1.0 works just fine

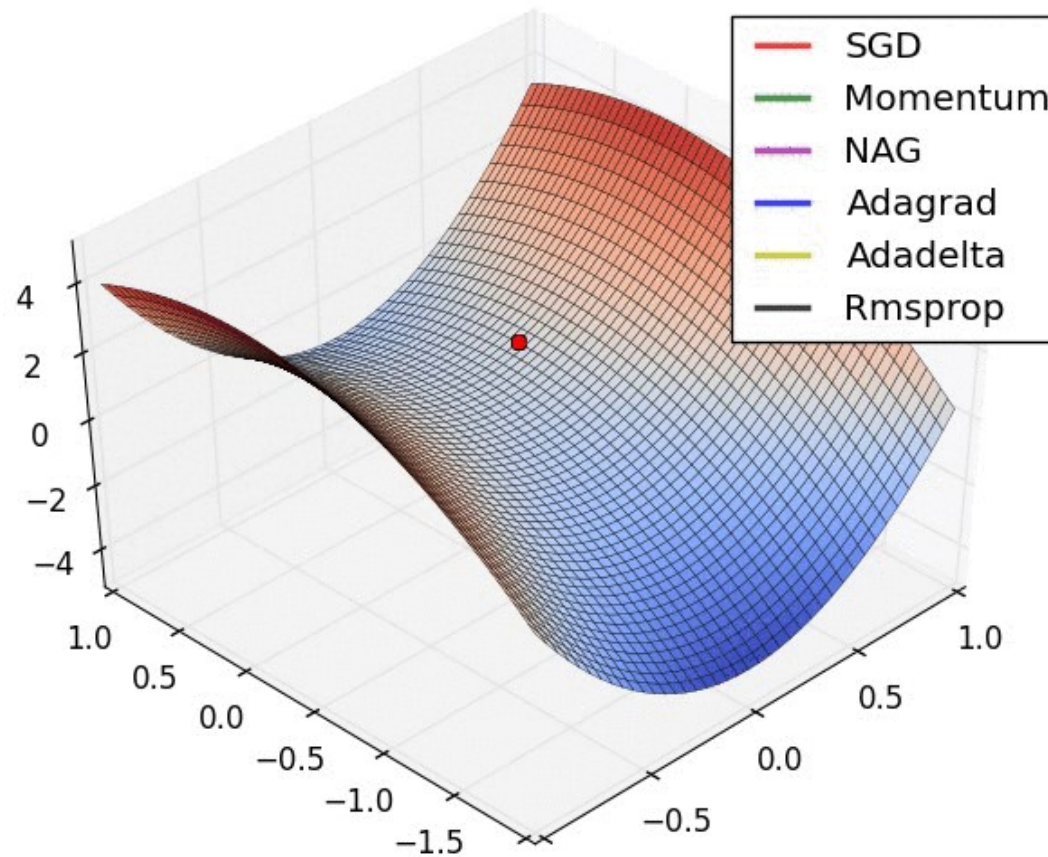


# Visualizing the optimizers: Beale's Function



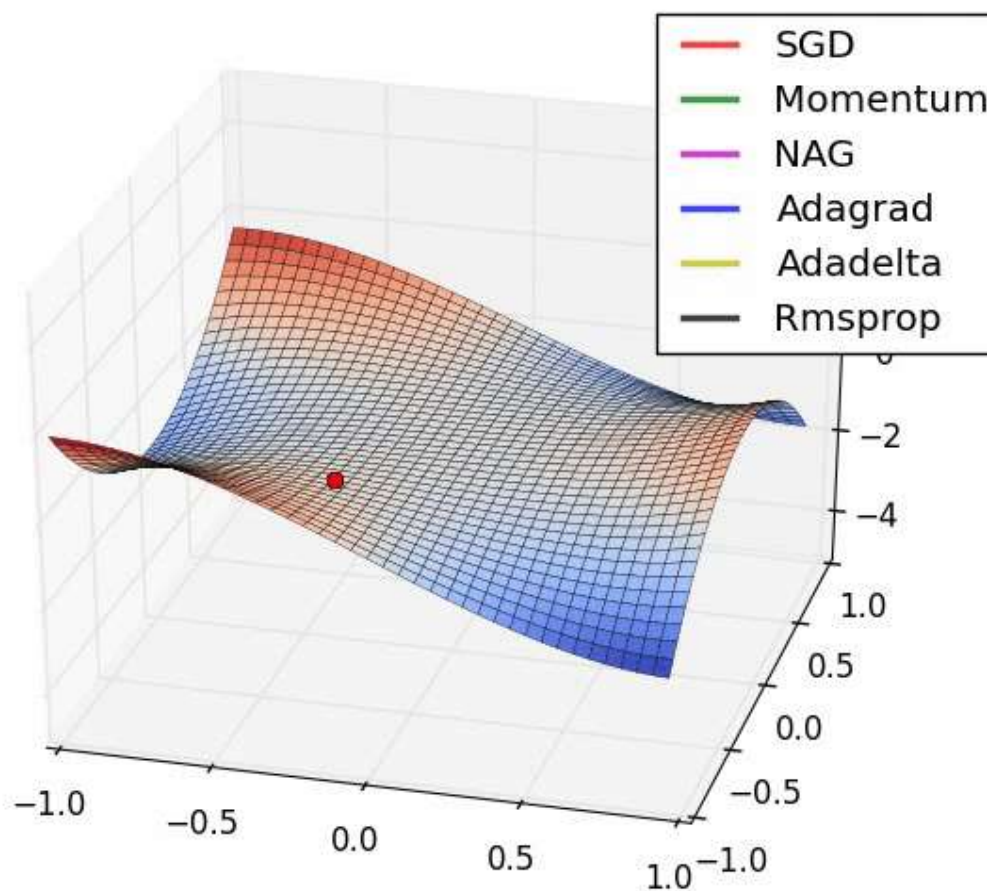
- <http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html>

# 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

# Moving on: 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

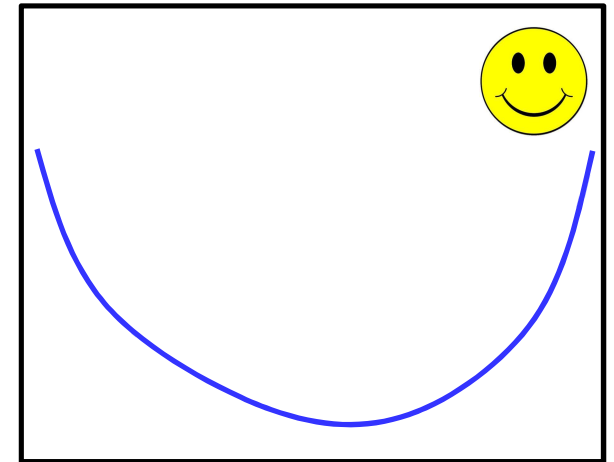
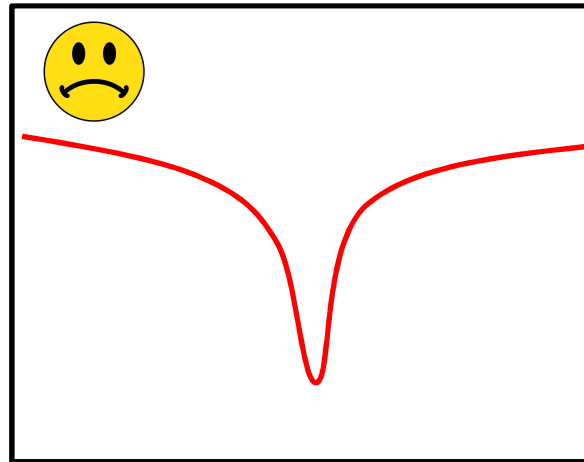
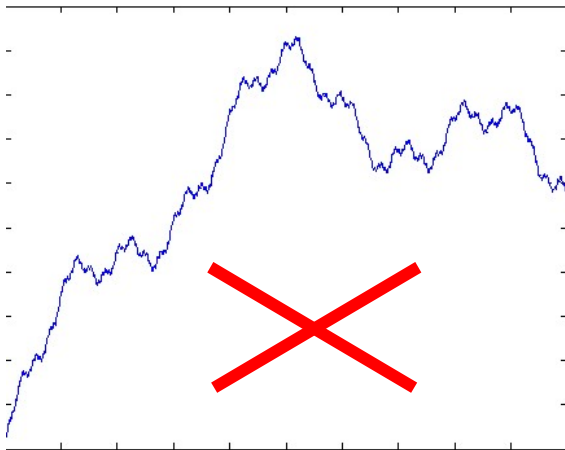
Total training error:

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

- 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



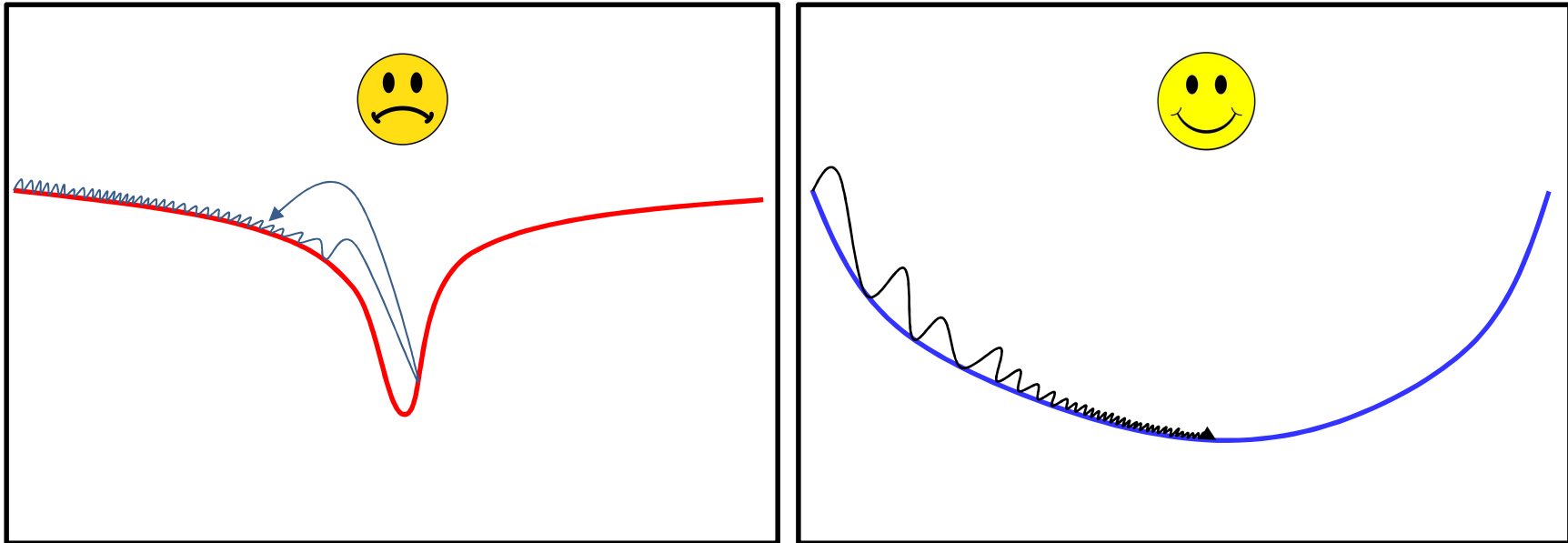
# Desiderata for a good divergence



- 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

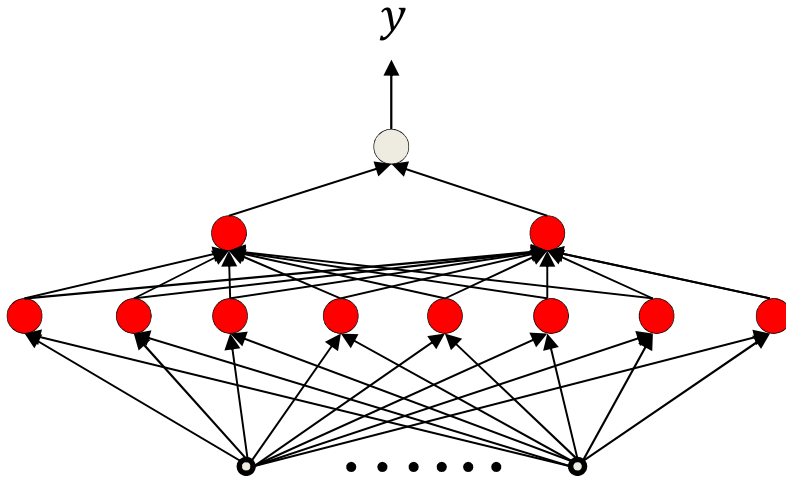


# Desiderata for a good divergence



- 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

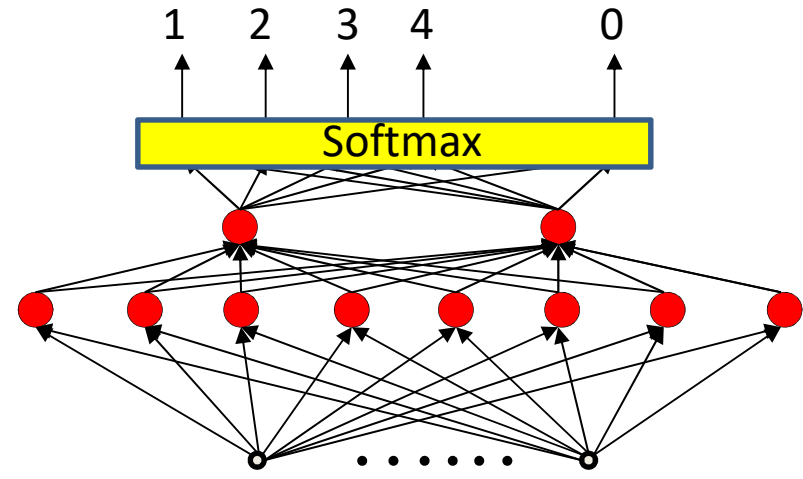
# Choices for divergence



Desired output:  $d$

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

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



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

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

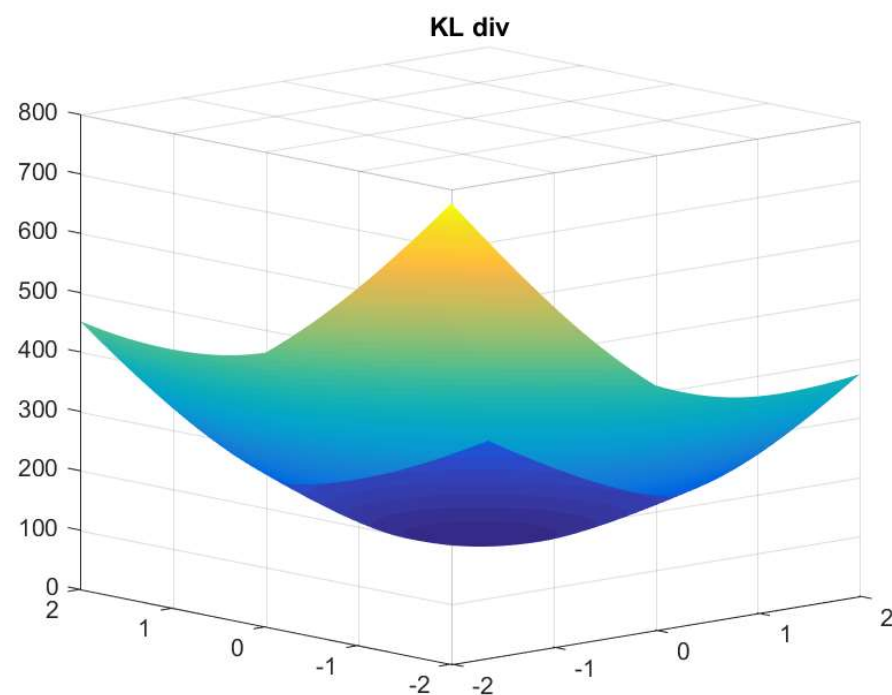
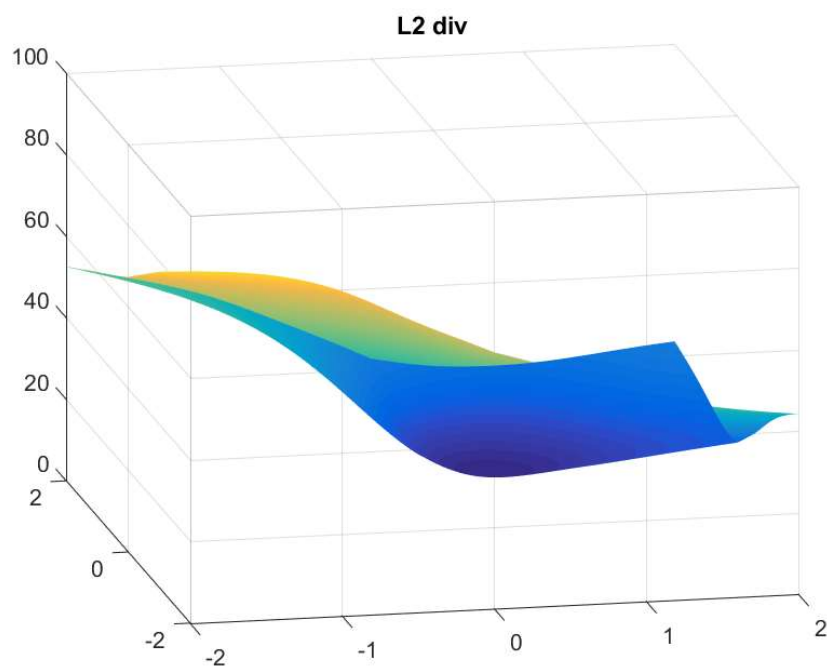
$$Div = \sum_i d_i \log(y_i)$$

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

## L2 or KL?

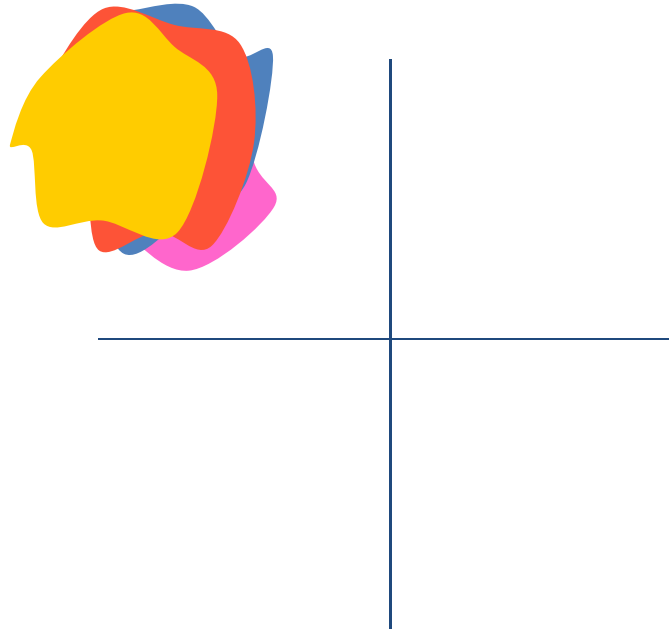
- The L2 divergence has long been favored in most applications
- It is particularly appropriate when attempting to perform *regression*
  - Numeric prediction
- The KL divergence is better when the intent is classification
  - The output is a probability vector

# L2 or KL



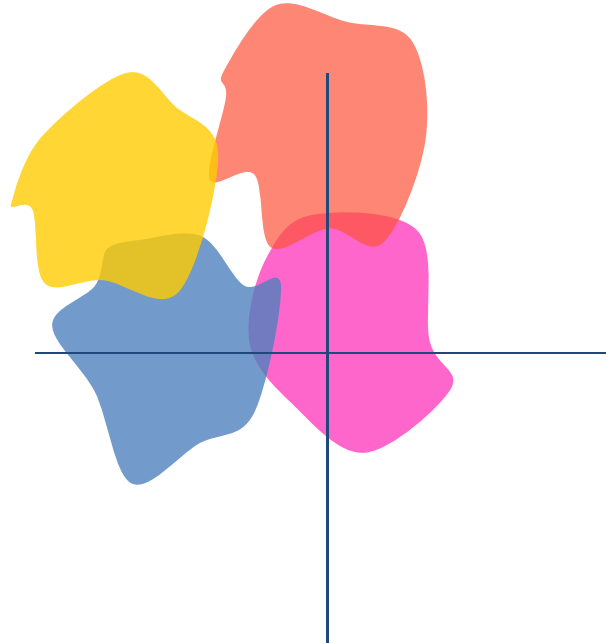
- Plot of L2 and KL divergences for a *single* perceptron, as function of weights
  - Setup: 2-dimensional input
  - 100 training examples randomly generated

# The problem of covariate shifts



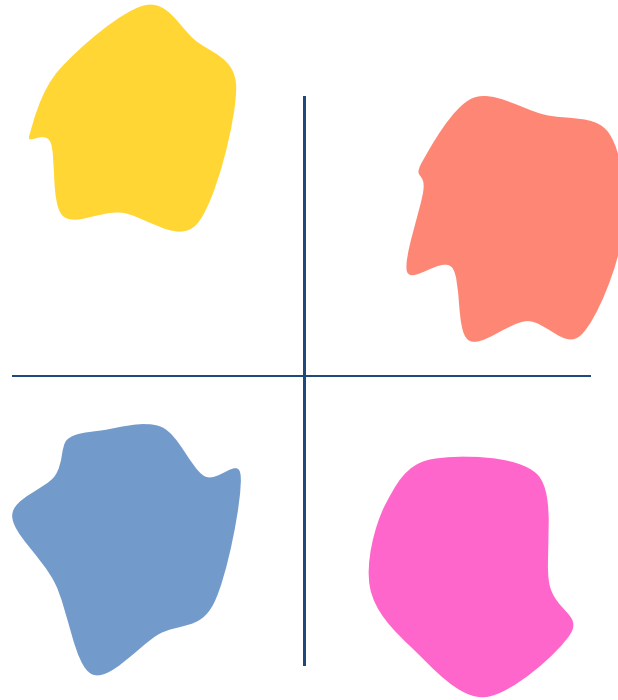
- Training assumes the training data are all similarly distributed
  - Minibatches have similar distribution

# The problem of covariate shifts



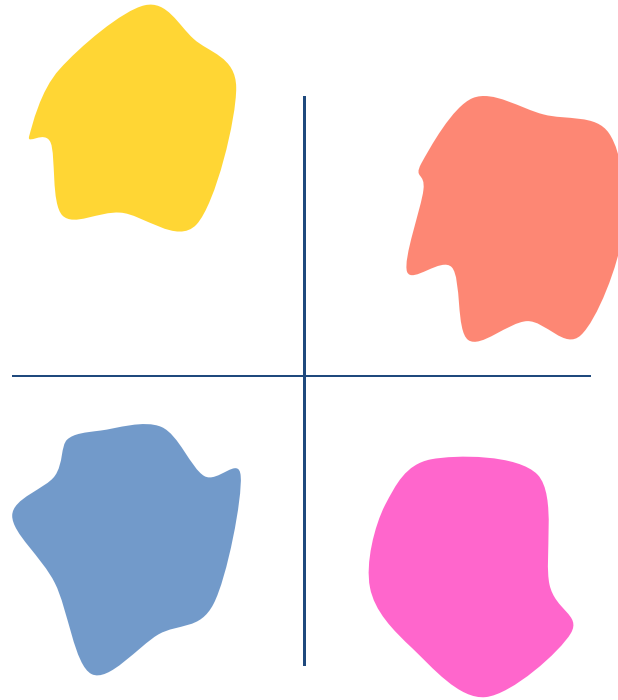
- Training assumes the training data are all similarly distributed
  - Minibatches have similar distribution
- In practice, each minibatch may have a different distribution
  - A “covariate shift”
  - Which may occur in *each* layer of the network

# 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

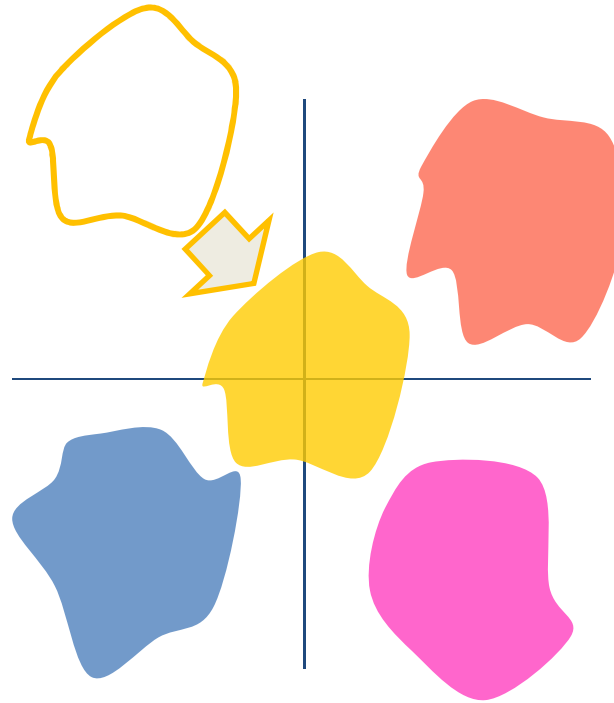
## **Solution:** Move all subgroups to a “standard” location



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

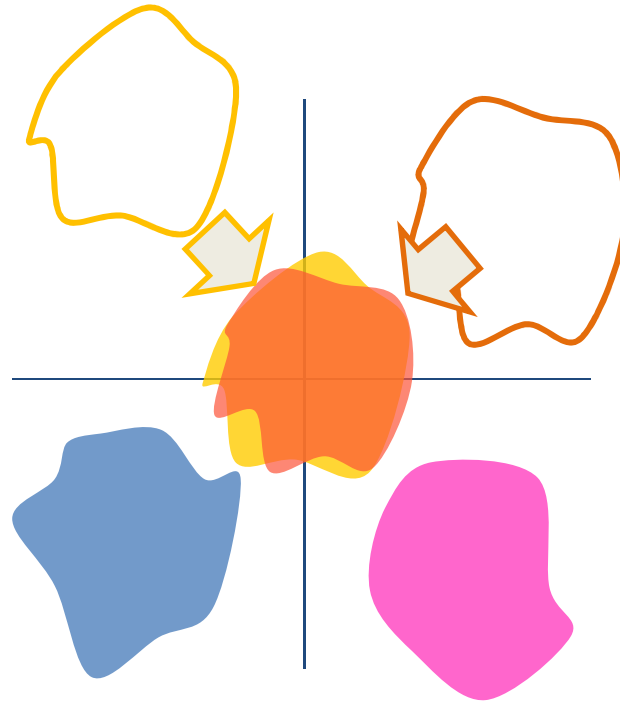


# Solution: Move all subgroups to a “standard” location



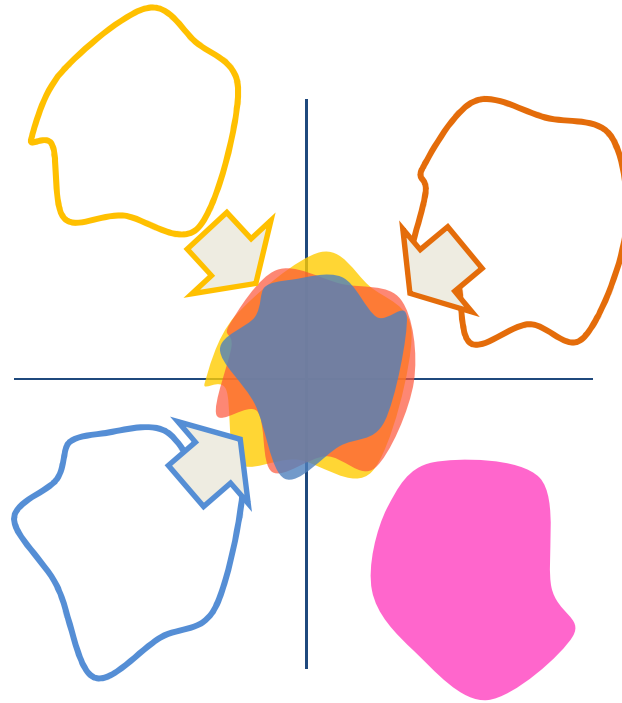
- “Move” all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches

# Solution: Move all subgroups to a “standard” location



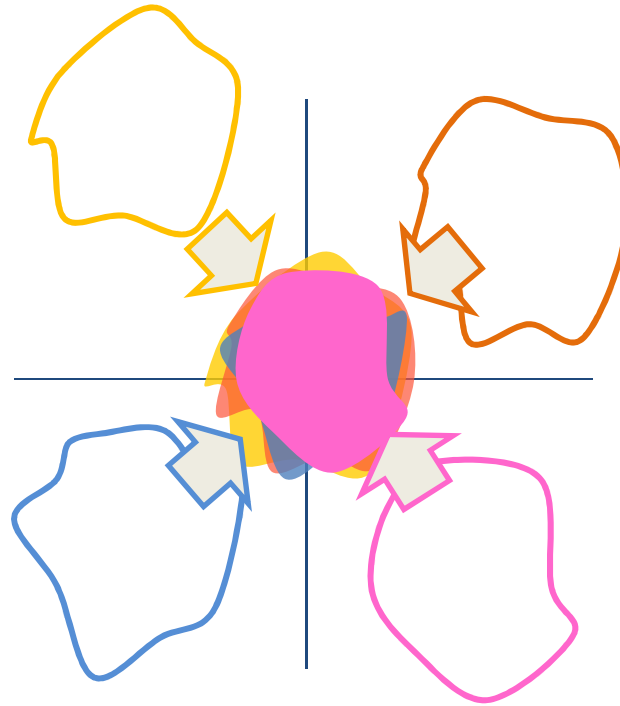
- “Move” all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches

# Solution: Move all subgroups to a “standard” location



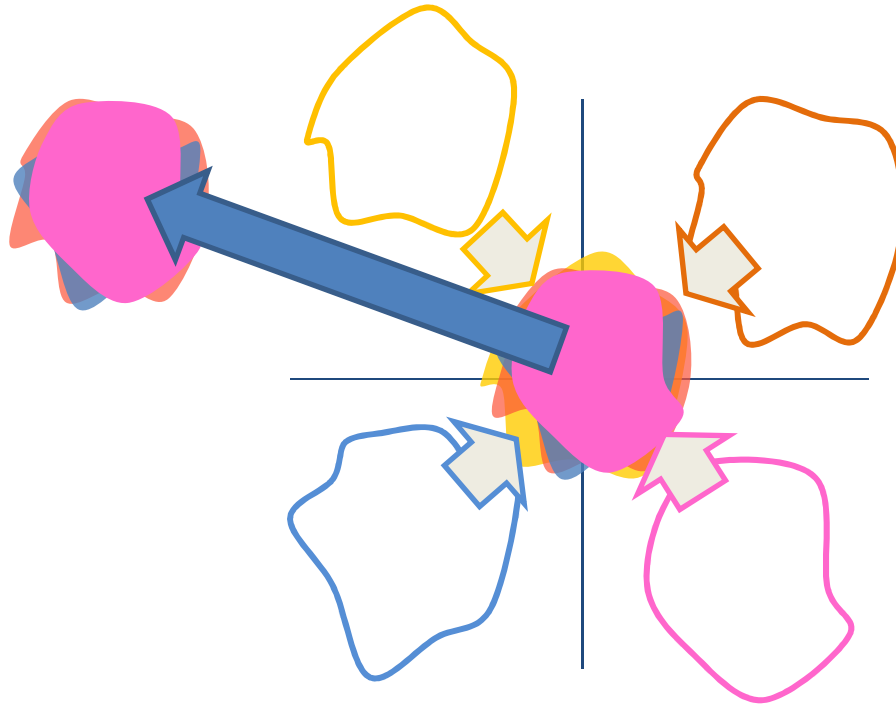
- “Move” all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches

# Solution: Move all subgroups to a “standard” location



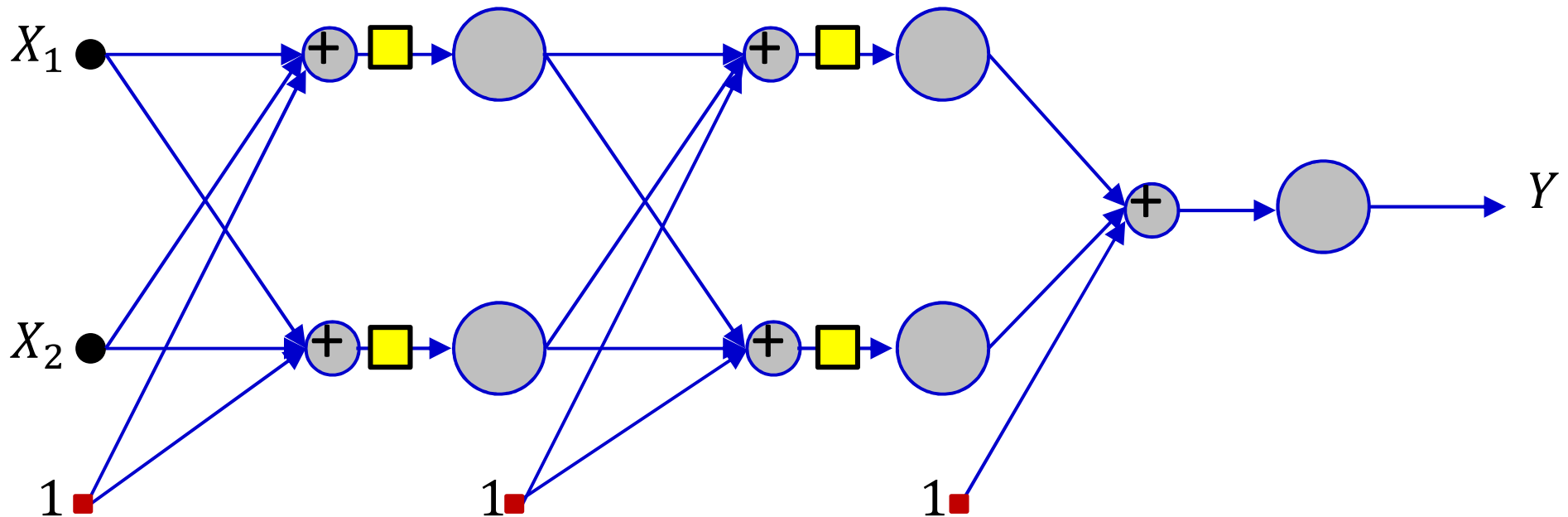
- “Move” all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches

# Solution: Move all subgroups to a “standard” location



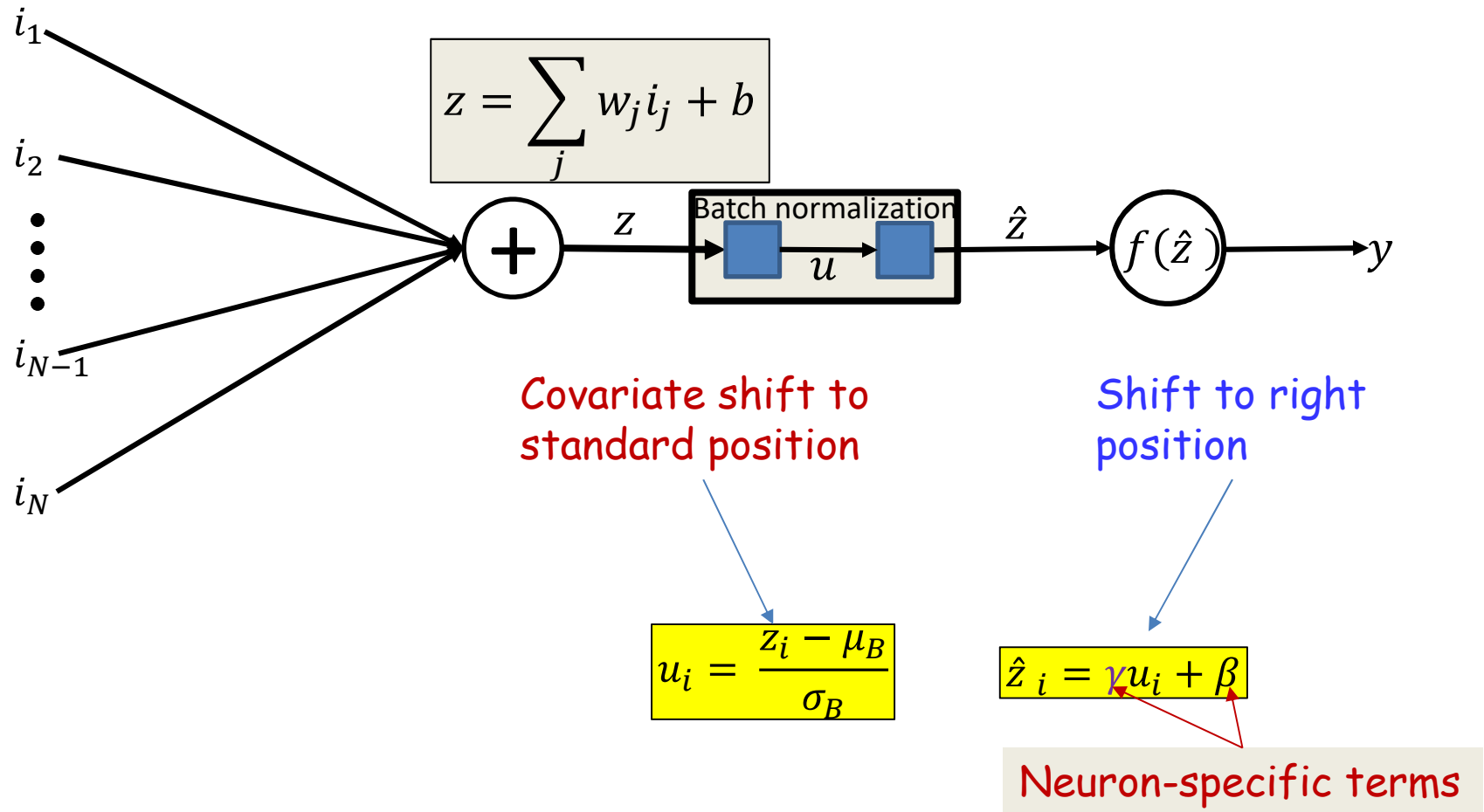
- “Move” all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches
  - Then move the entire collection to the appropriate location

# Batch normalization



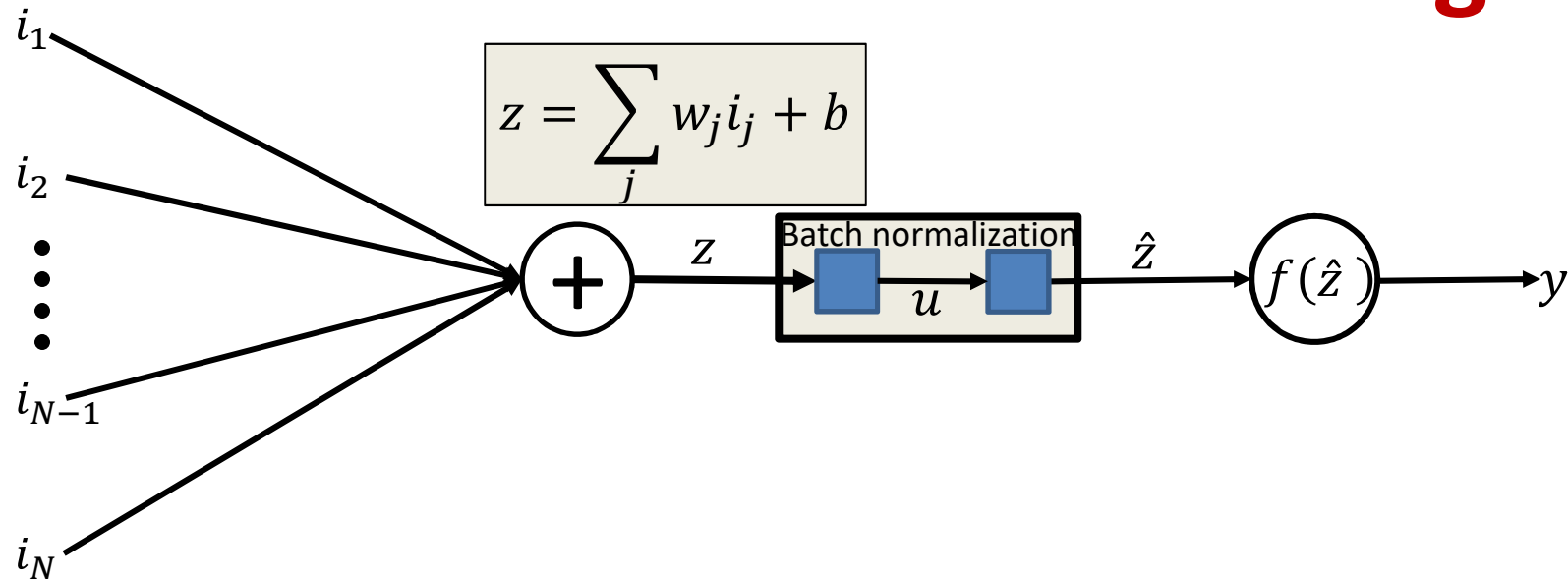
- Batch normalization is a covariate adjustment unit that happens after the weighted addition of inputs but before the application of activation
  - Is done independently for each unit, to simplify computation
- **Training:** The adjustment occurs over individual minibatches

# 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



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

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

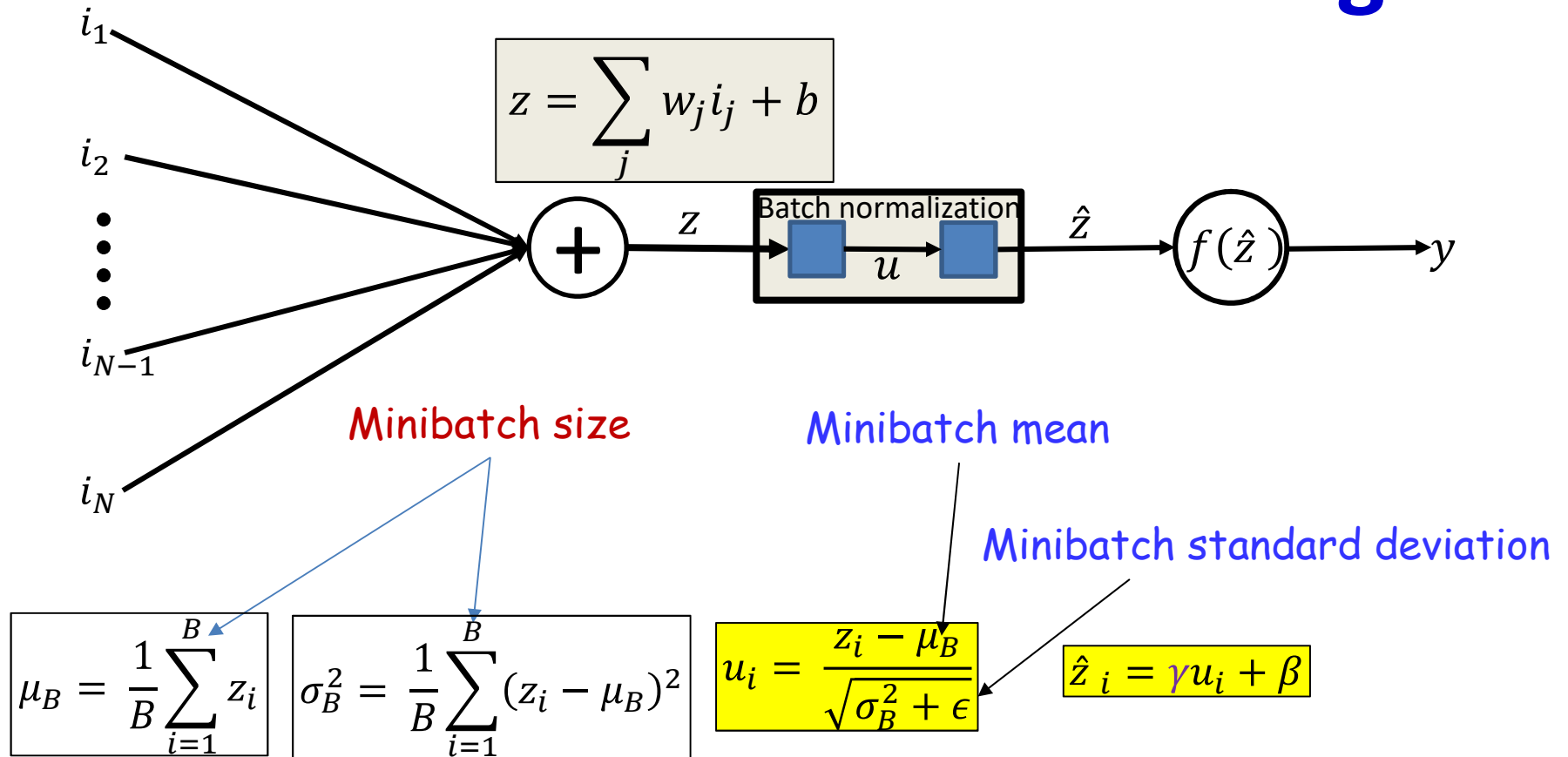
$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

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

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

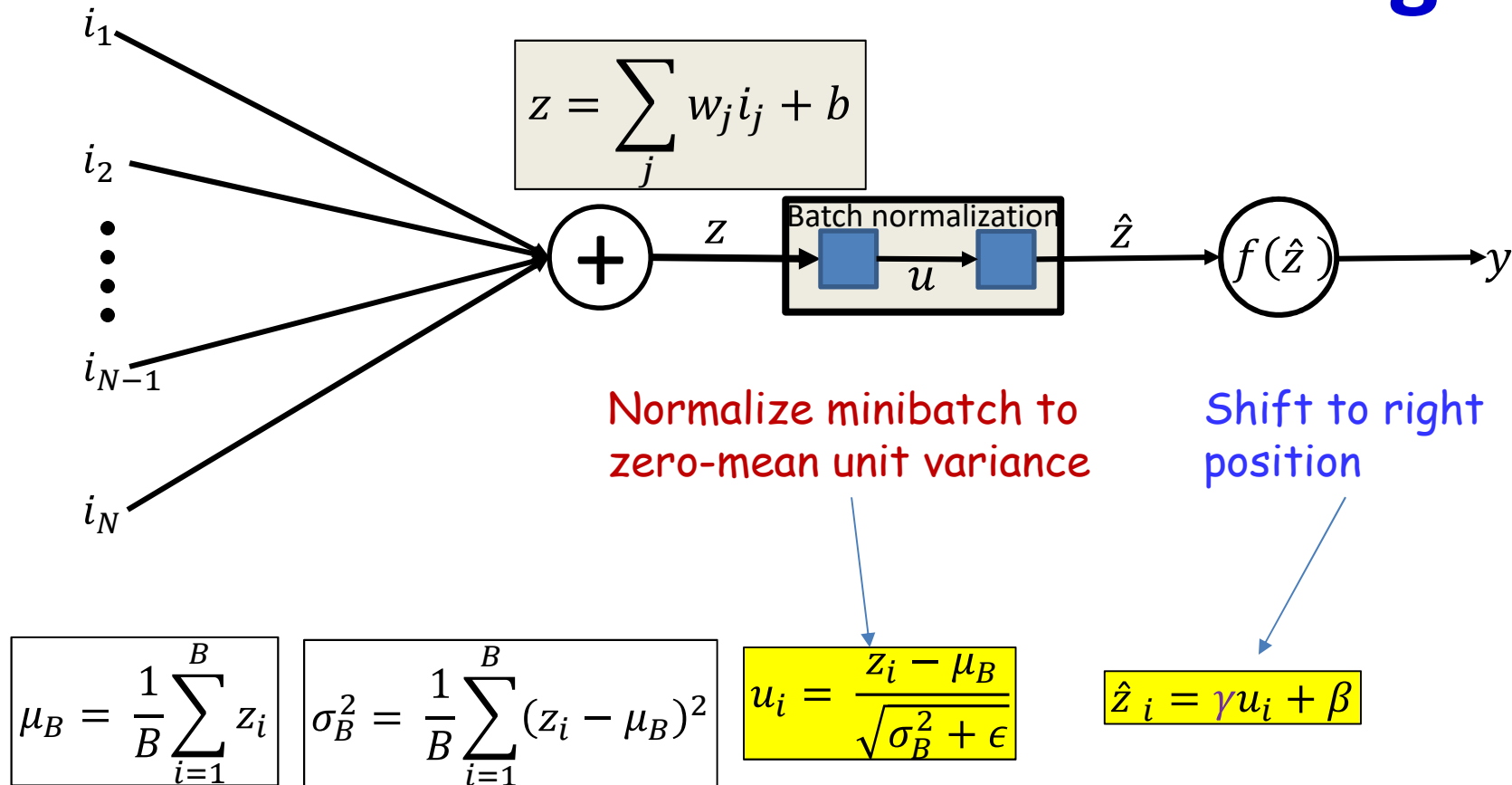


# Batch normalization: Training



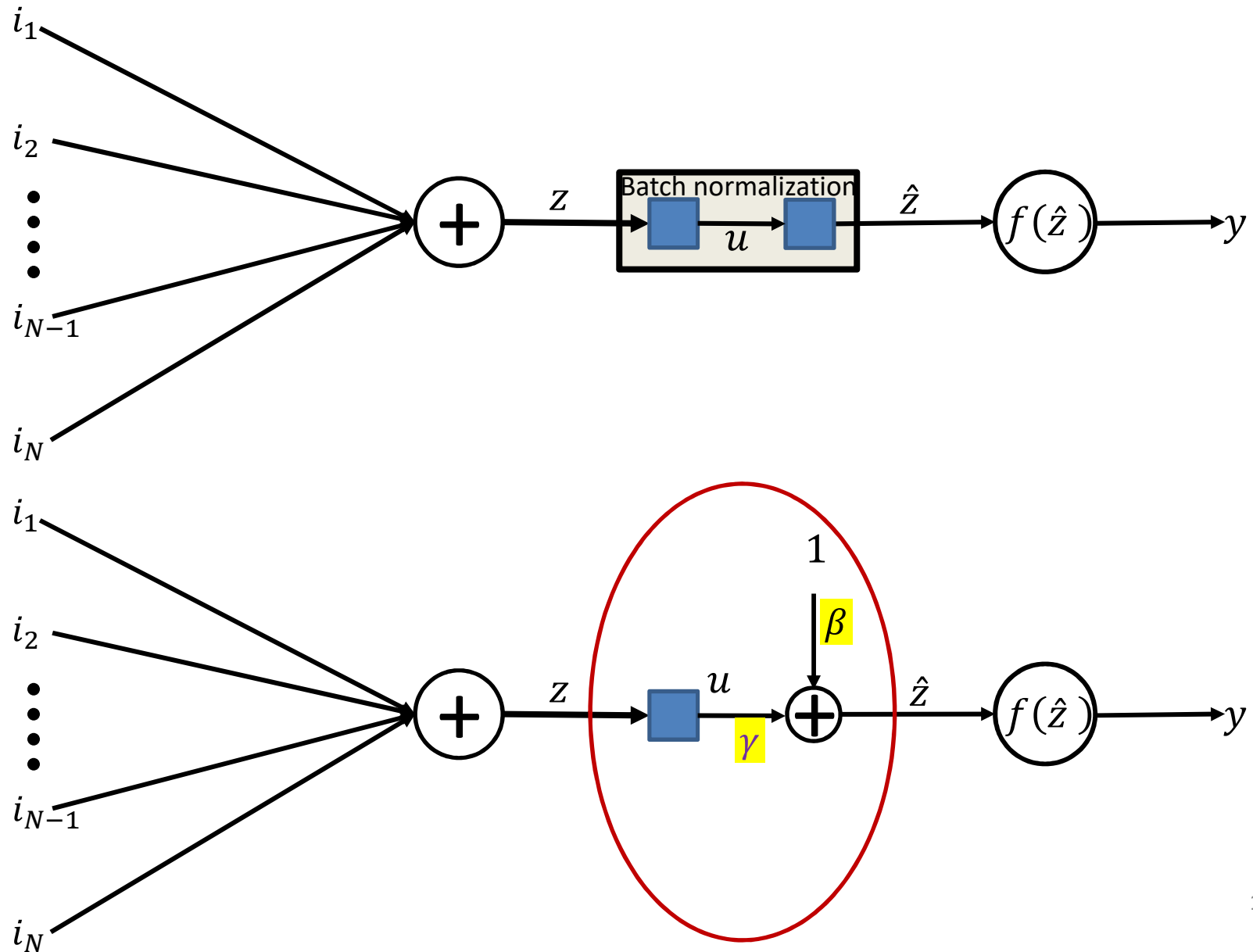
- BN aggregates the statistics over a minibatch and normalizes the batch by them
- Normalized instances are “shifted” to a *unit-specific* location

# Batch normalization: Training

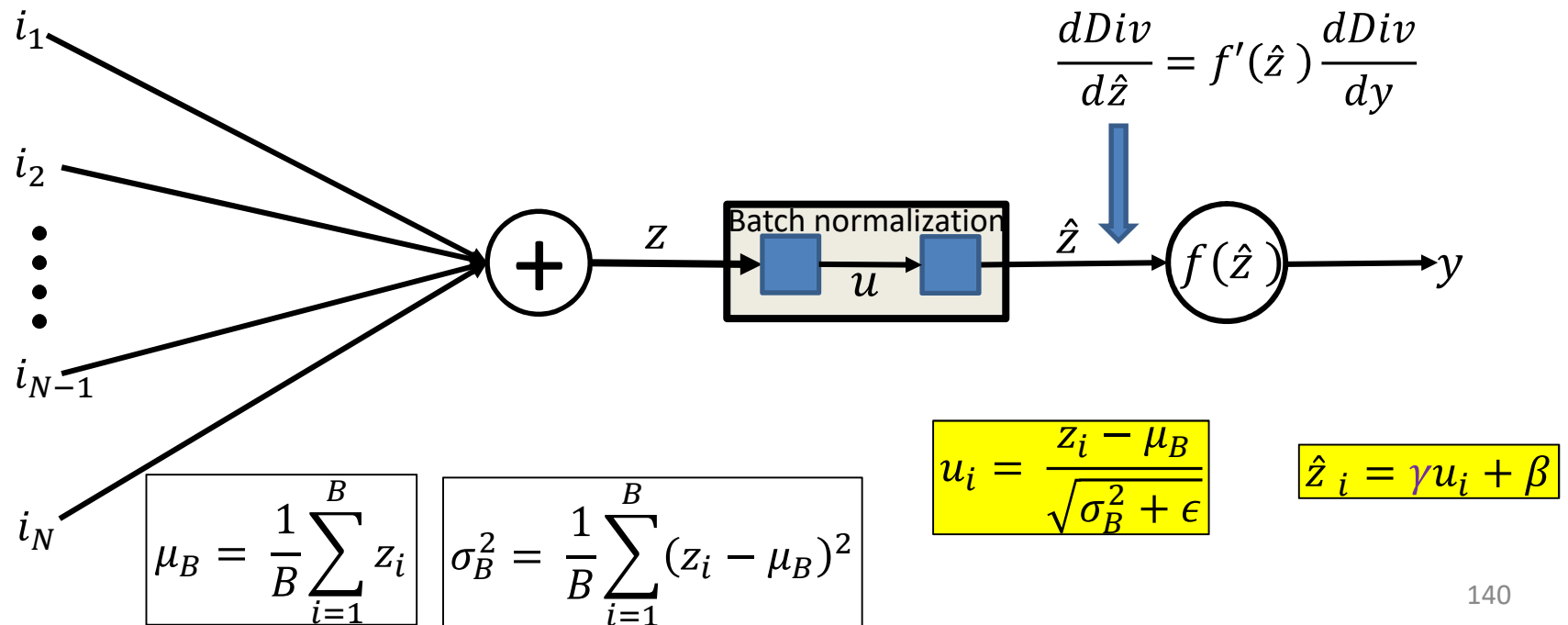


- BN aggregates the statistics over a minibatch and normalizes the batch by them
- Normalized instances are “shifted” to a *unit-specific* location

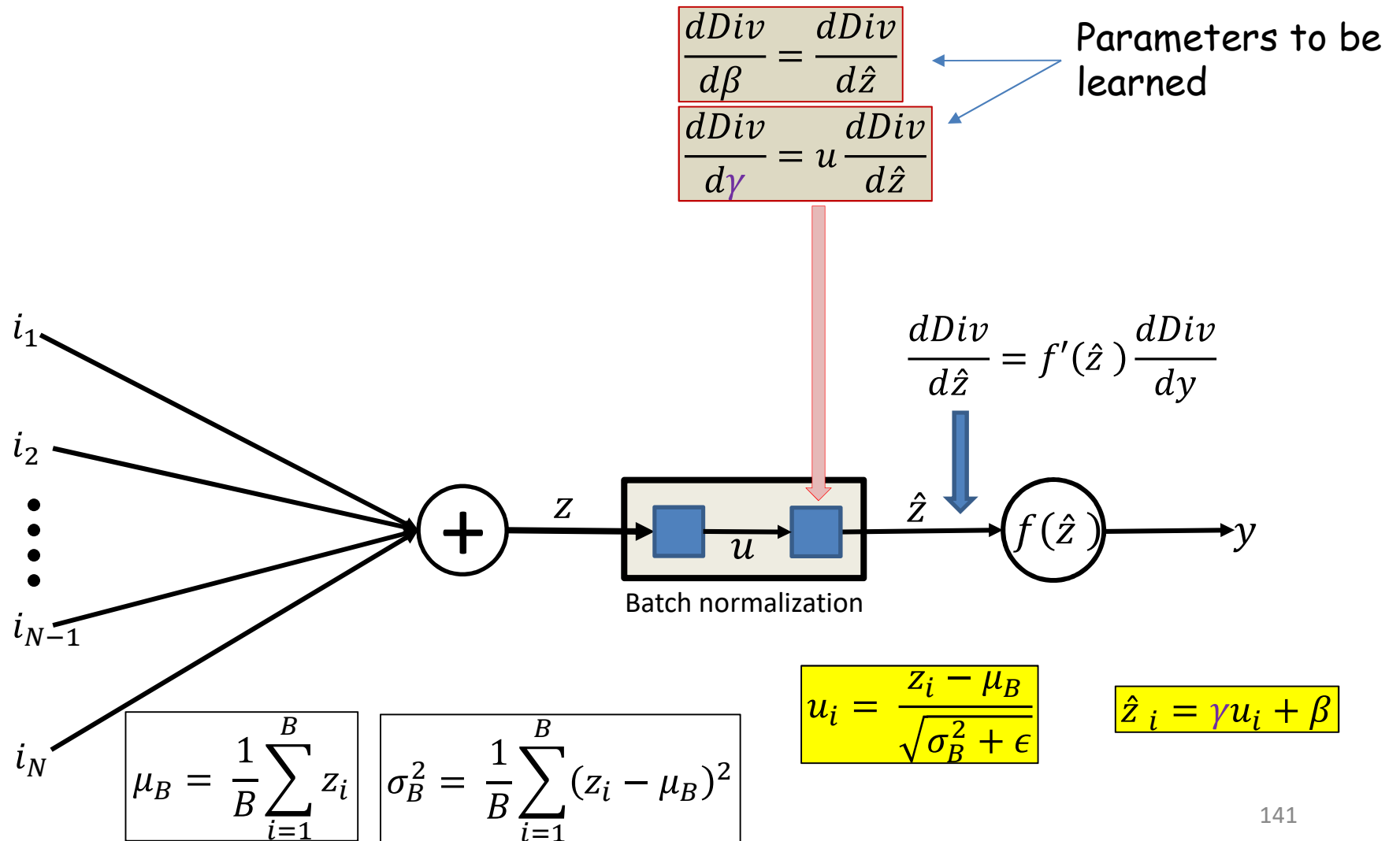
# A better picture for batch norm



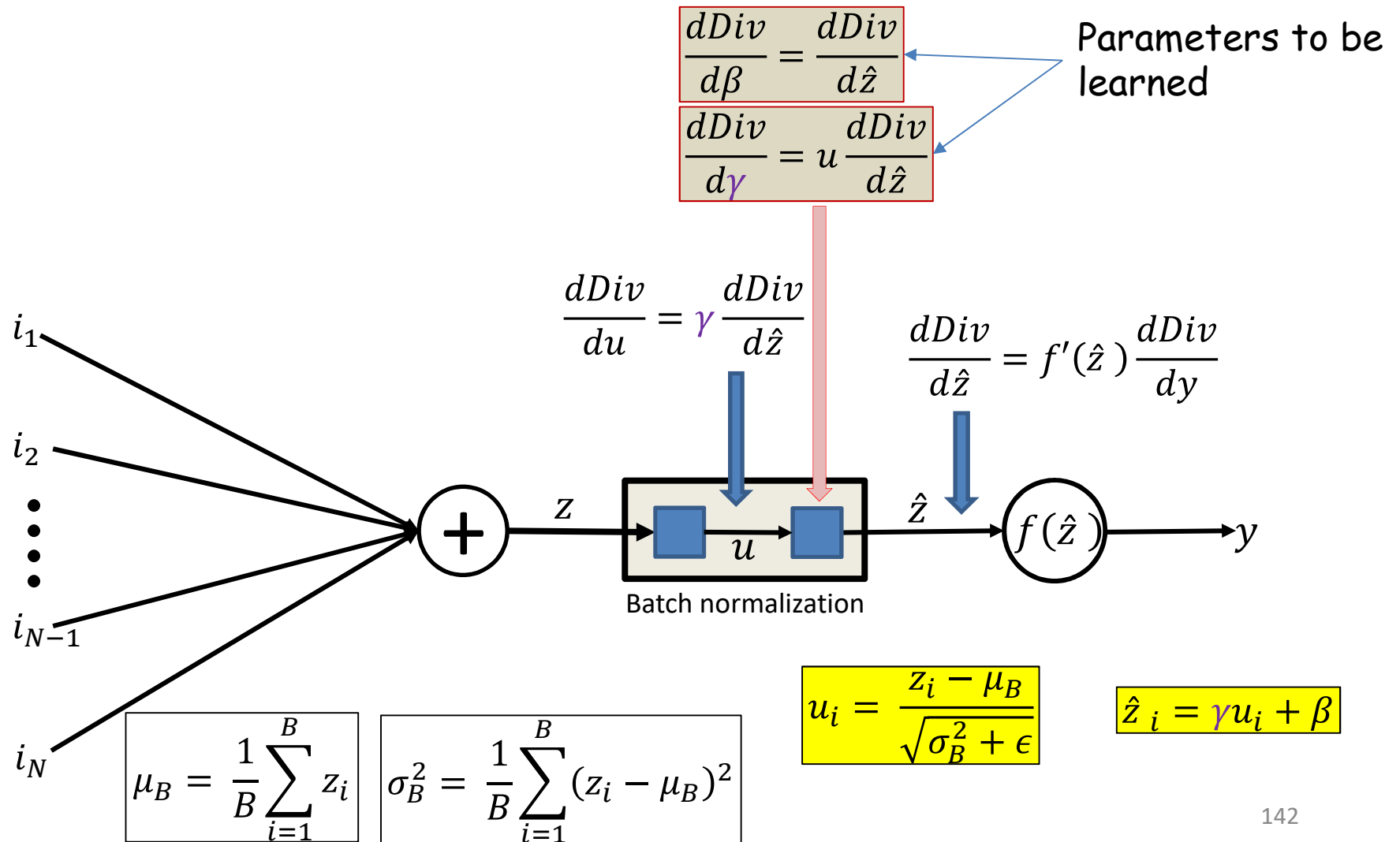
# Batch normalization: Backpropagation



# Batch normalization: Backpropagation

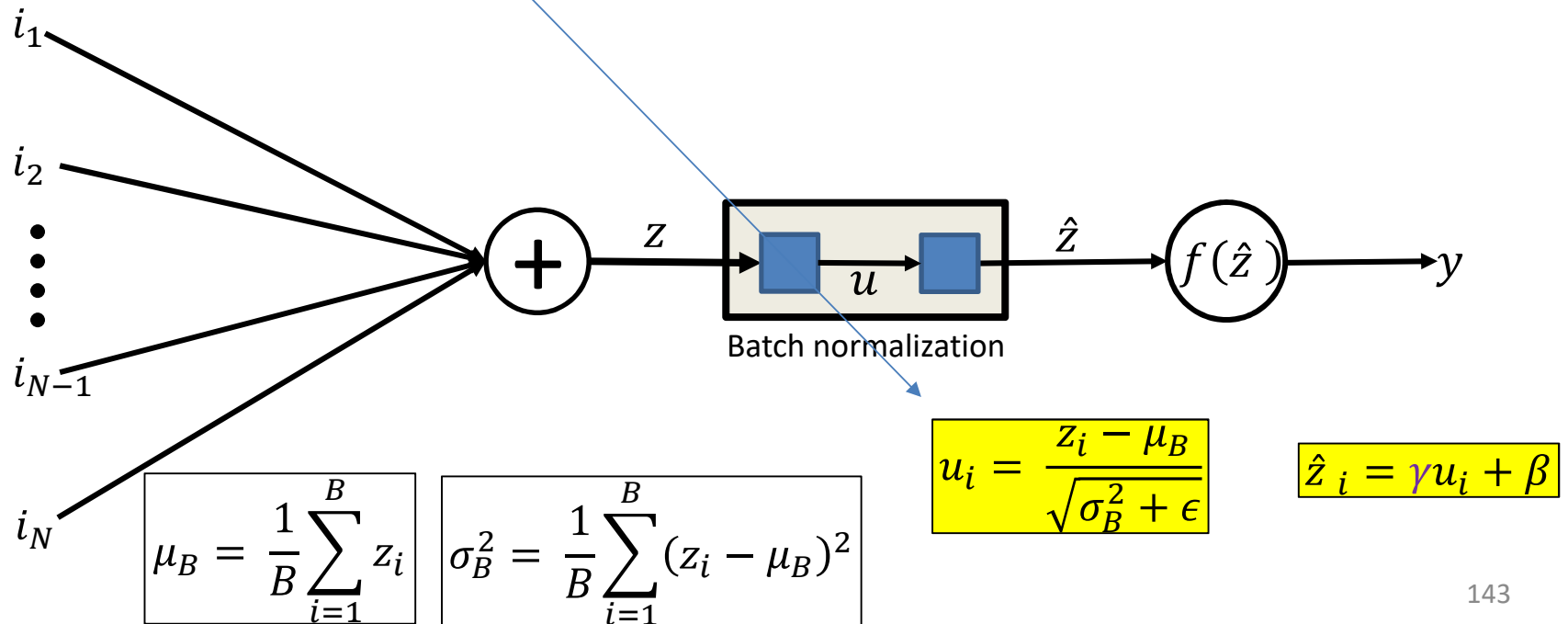


# Batch normalization: Backpropagation



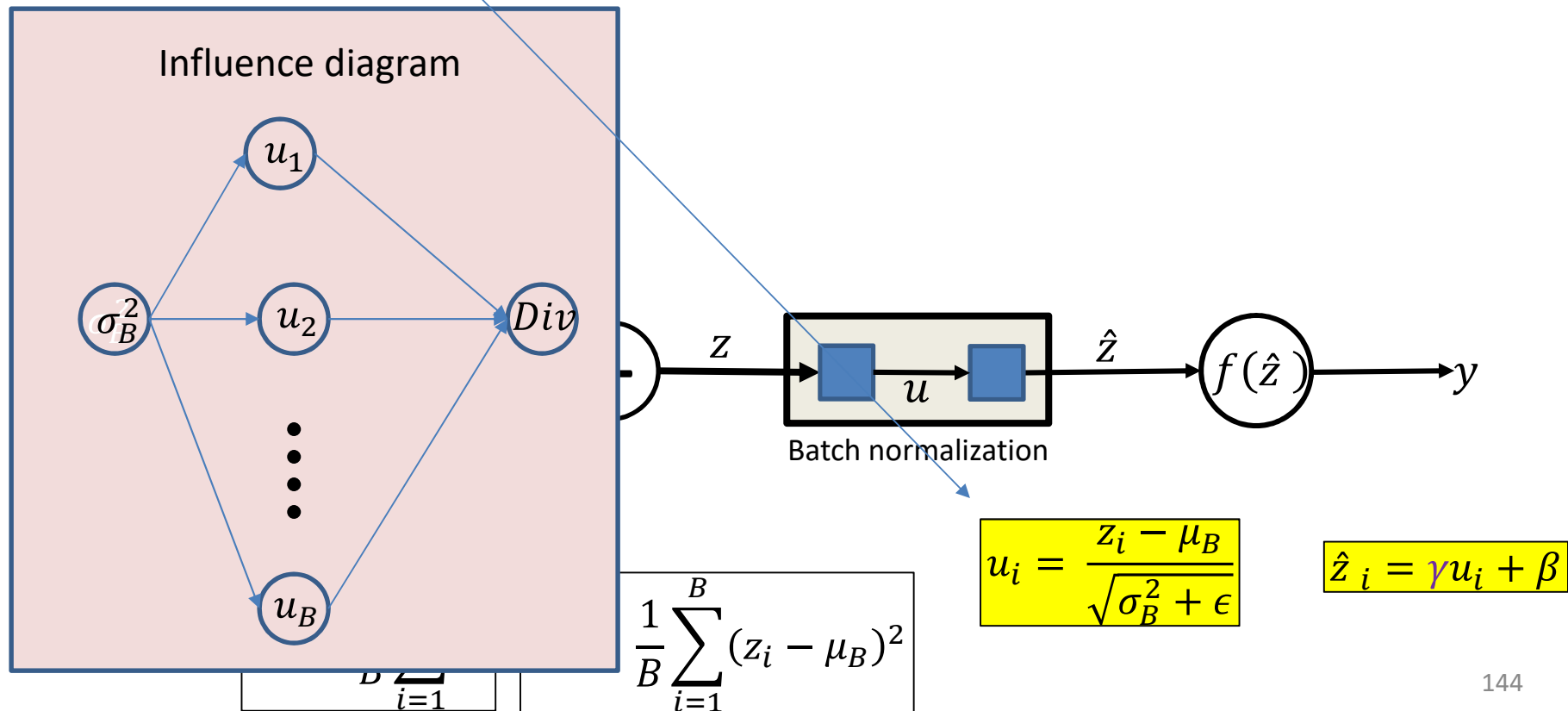
# Batch normalization: Backpropagation

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



# Batch normalization: Backpropagation

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

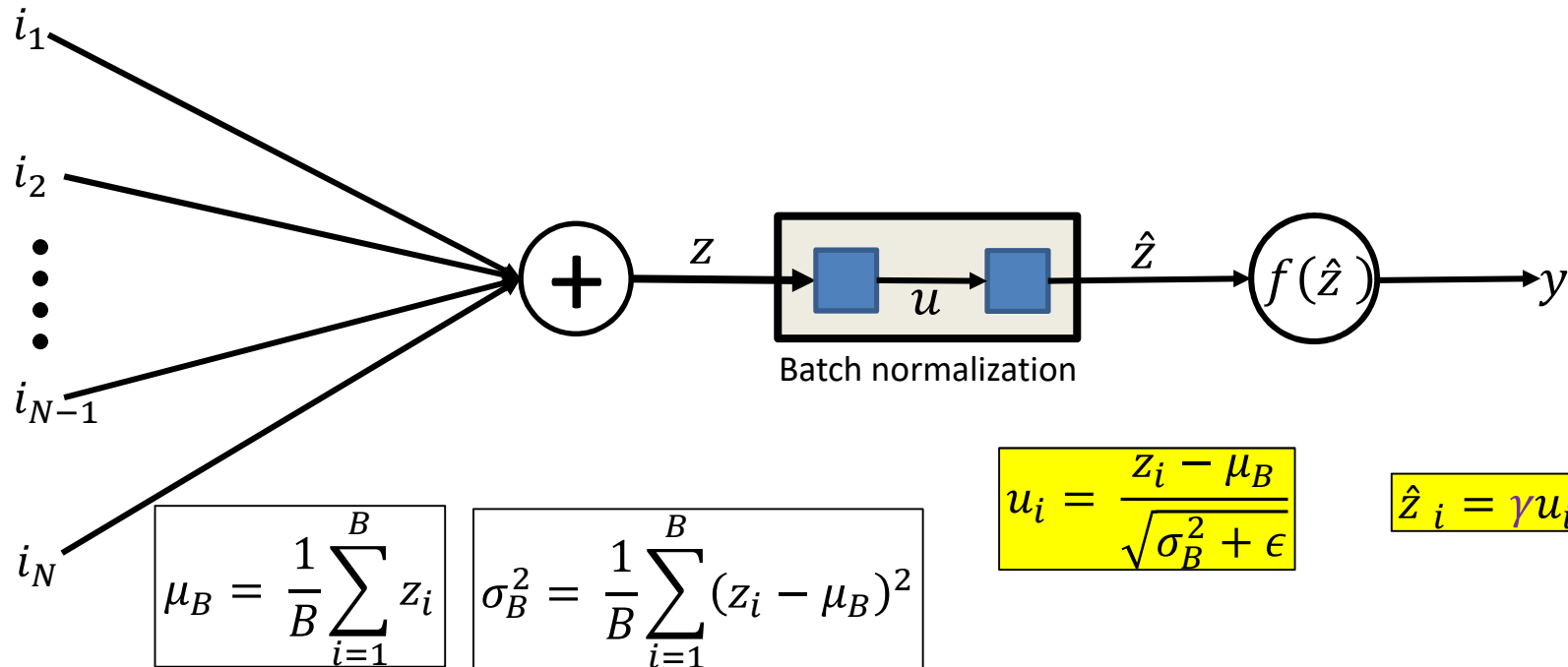




# Batch normalization: Backpropagation

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

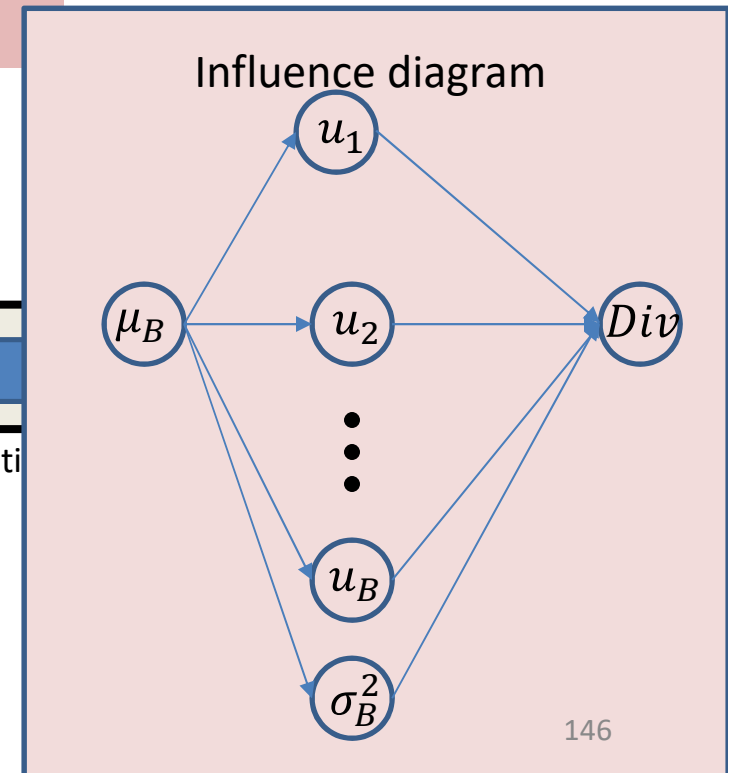
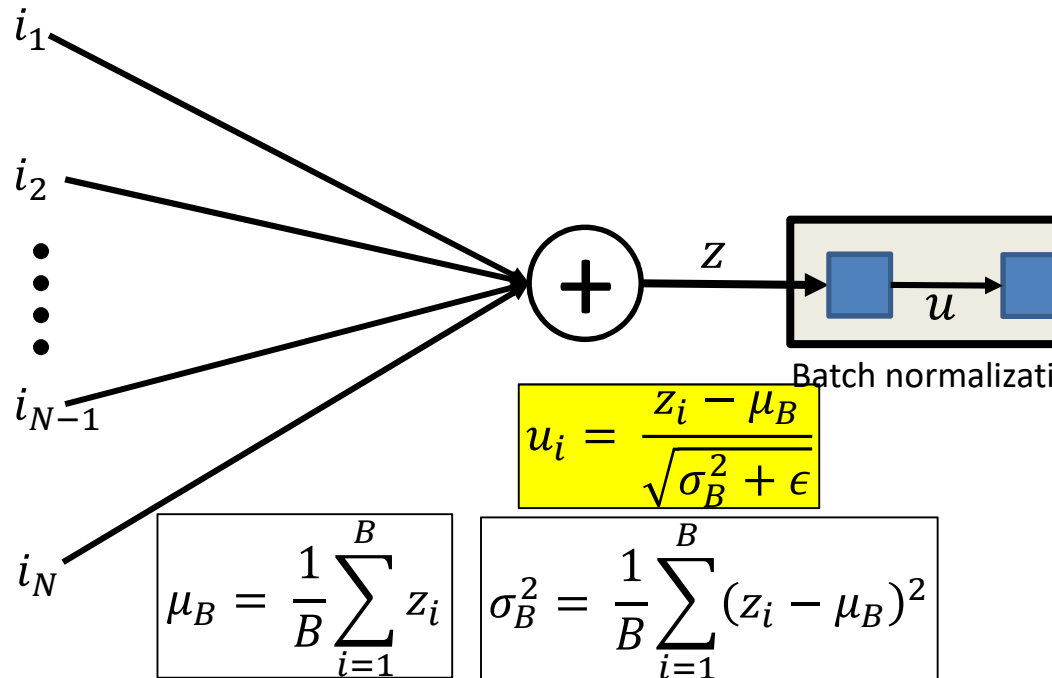
$$\frac{\partial \text{Div}}{\partial \mu_B} = \left( \sum_{i=1}^B \frac{\partial \text{Div}}{\partial u_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \frac{\partial \text{Div}}{\partial \sigma_B^2} \cdot \frac{\sum_{i=1}^B -2(z_i - \mu_B)}{B}$$



# Batch normalization: Backpropagation

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

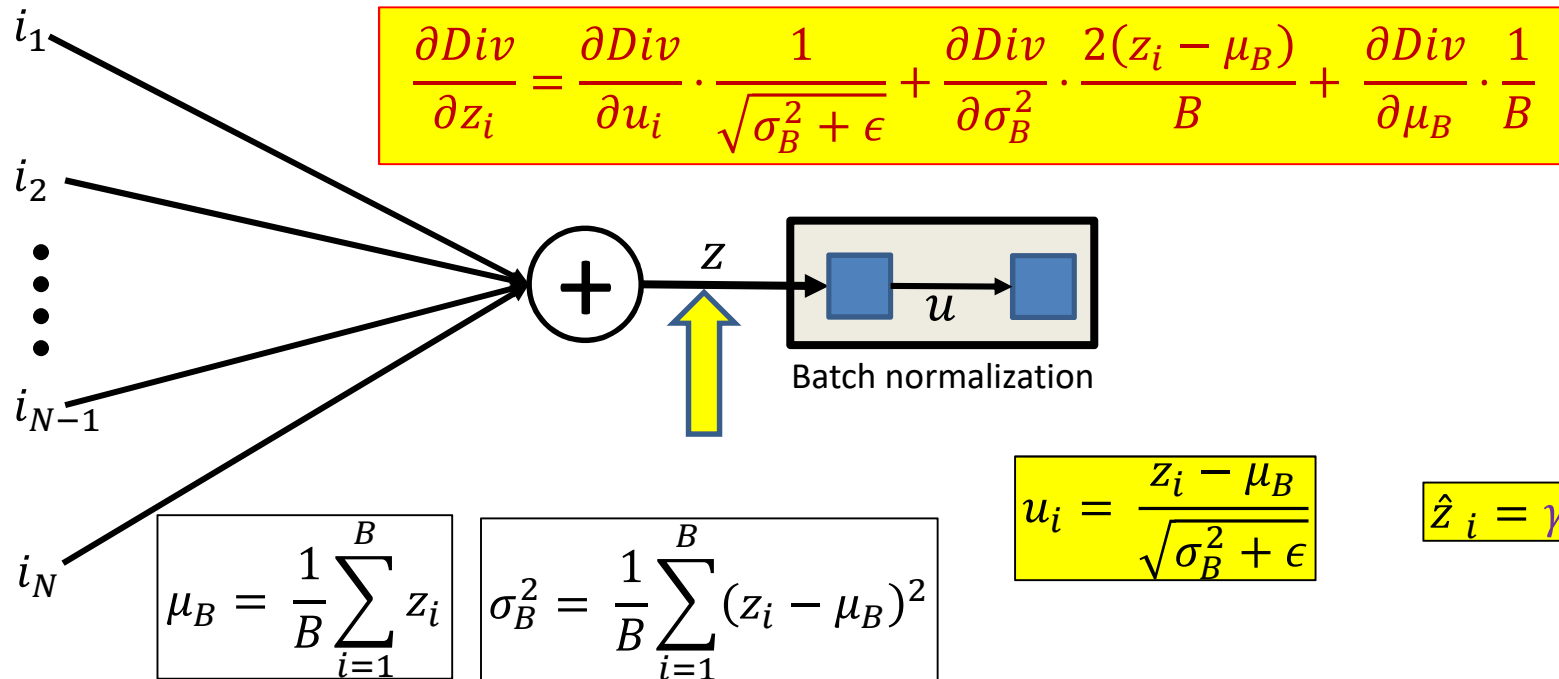
$$\frac{\partial \text{Div}}{\partial \mu_B} = \left( \sum_{i=1}^B \frac{\partial \text{Div}}{\partial u_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \frac{\partial \text{Div}}{\partial \sigma_B^2} \cdot \frac{\sum_{i=1}^B -2(z_i - \mu_B)}{B}$$



# Batch normalization: Backpropagation

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

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

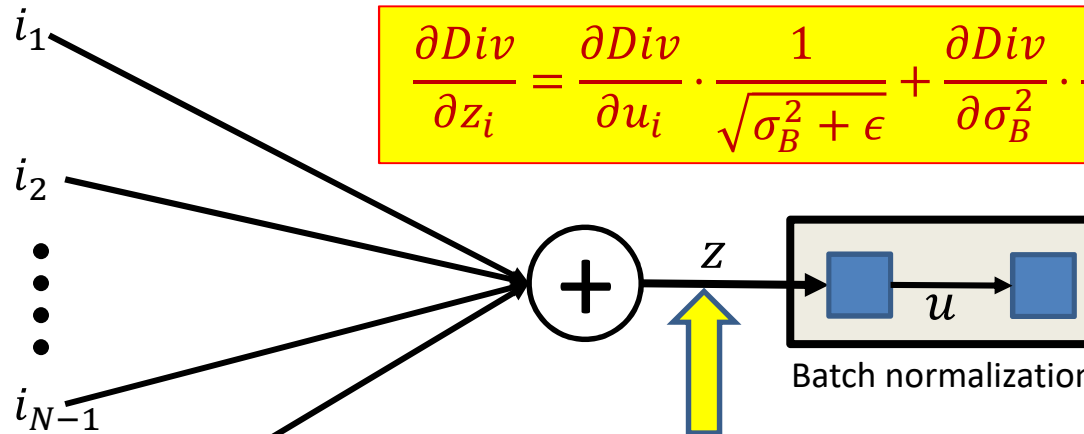
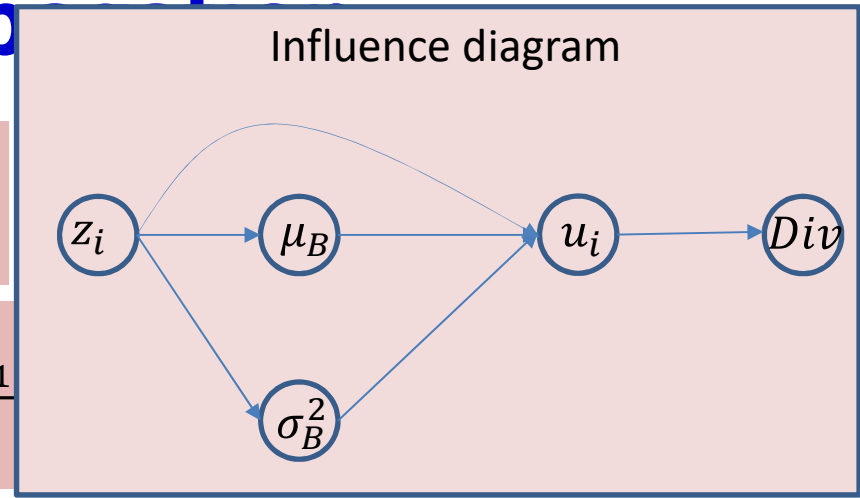


# Batch normalization:

## Backprop

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

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



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

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

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

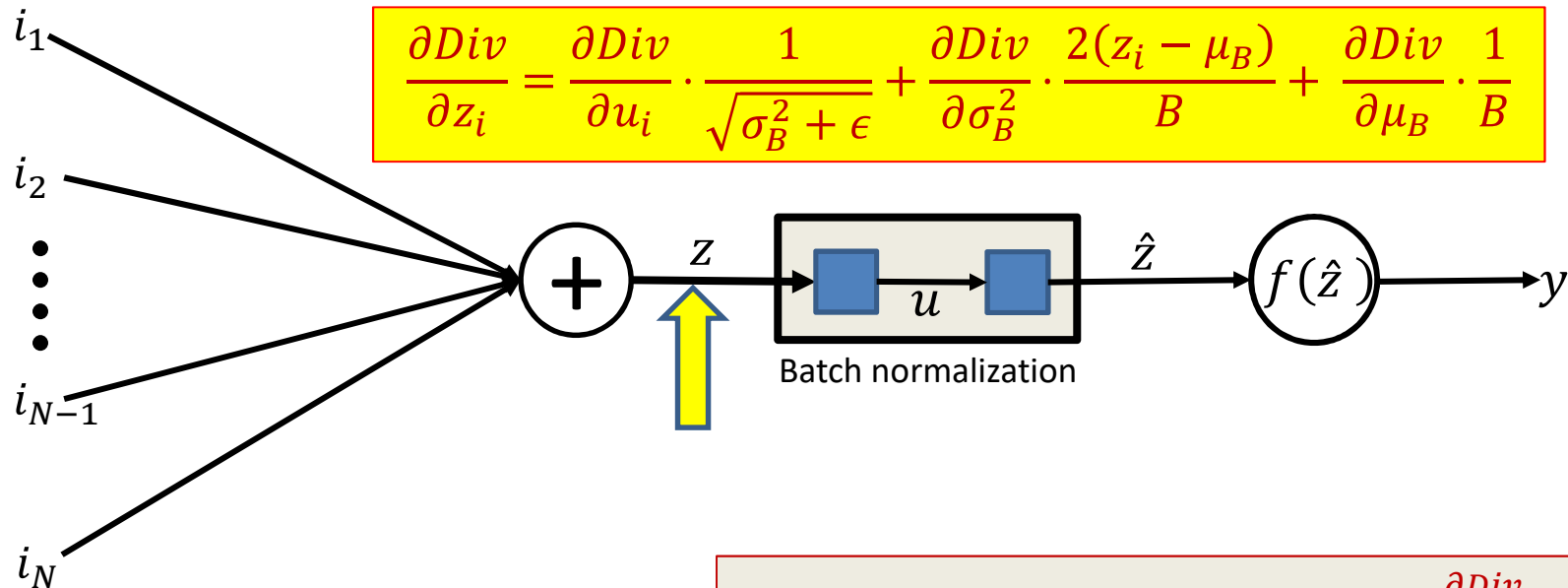
$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

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

# Batch normalization: Backpropagation

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

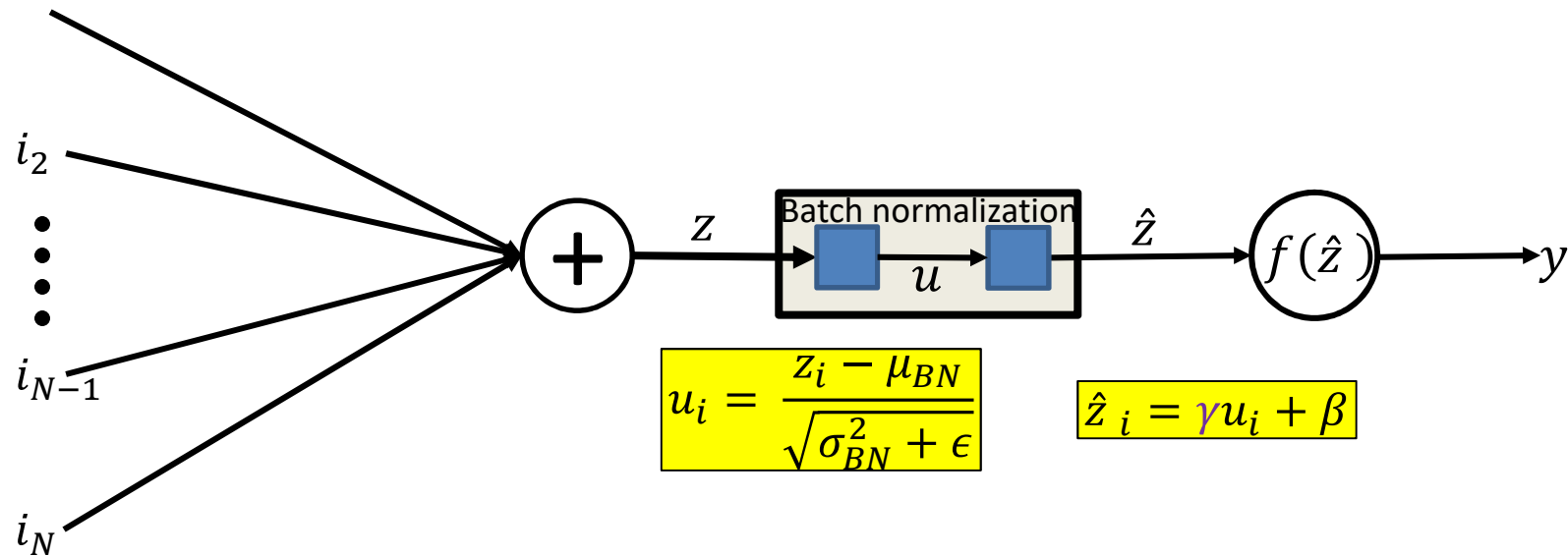
$$\frac{\partial Div}{\partial \mu_B} = \left( \sum_{i=1}^B \frac{\partial Div}{\partial u_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \frac{\partial Div}{\partial \sigma_B^2} \cdot \frac{\sum_{i=1}^B -2(z_i - \mu_B)}{B}$$



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

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

# Batch normalization: Inference



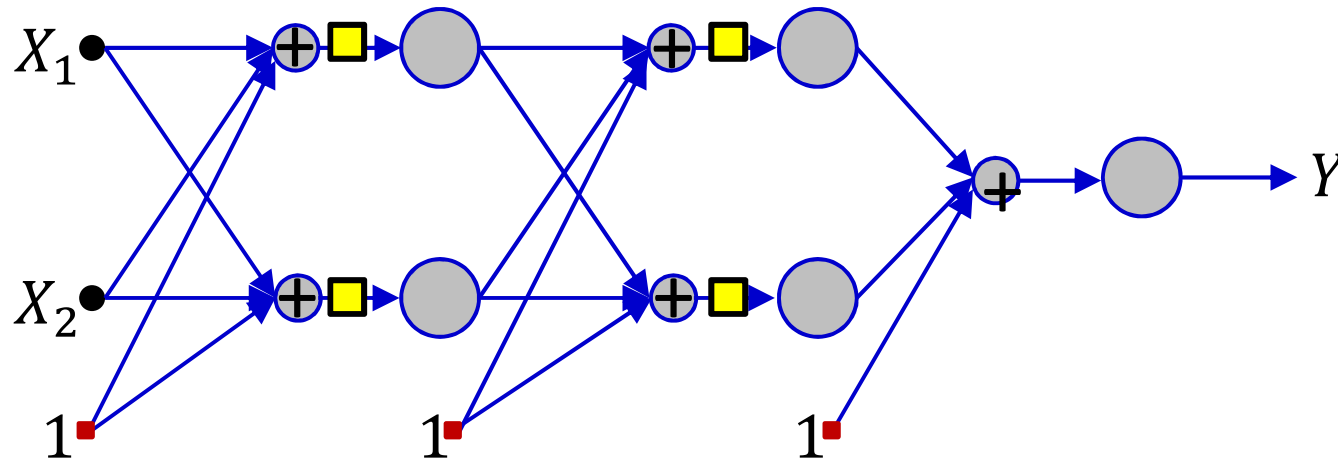
- On test data, BN requires  $\mu_B$  and  $\sigma_B^2$ .
- We will use the *average over all training minibatches*

$$\mu_{BN} = \frac{1}{Nbatches} \sum_{batch} \mu_B(batch)$$

$$\sigma_{BN}^2 = \frac{B}{(B-1)Nbatches} \sum_{batch} \sigma_B^2(batch)$$

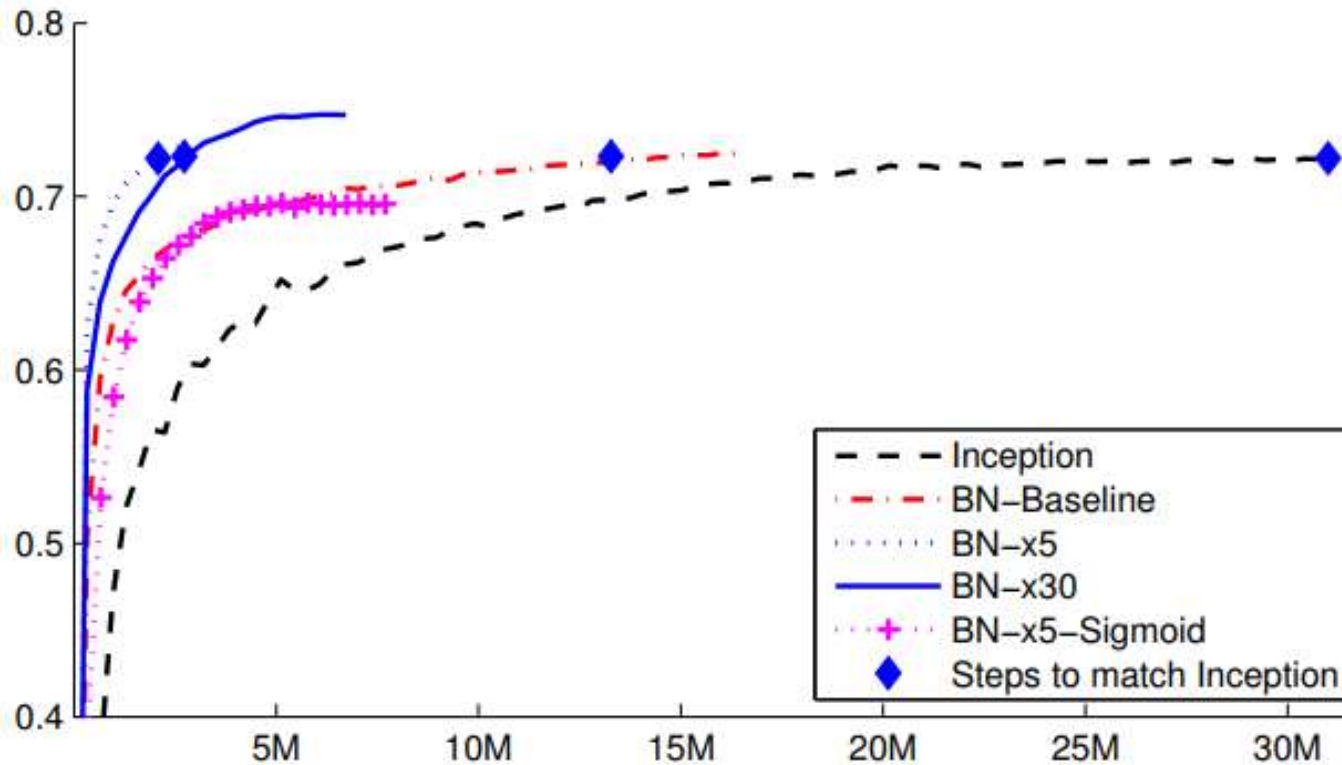
- Note: these are *neuron-specific*
  - $\mu_B(batch)$  and  $\sigma_B^2(batch)$  here are obtained from the *final converged network*
  - The  $B/(B-1)$  term gives us an unbiased estimator for the variance

# 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

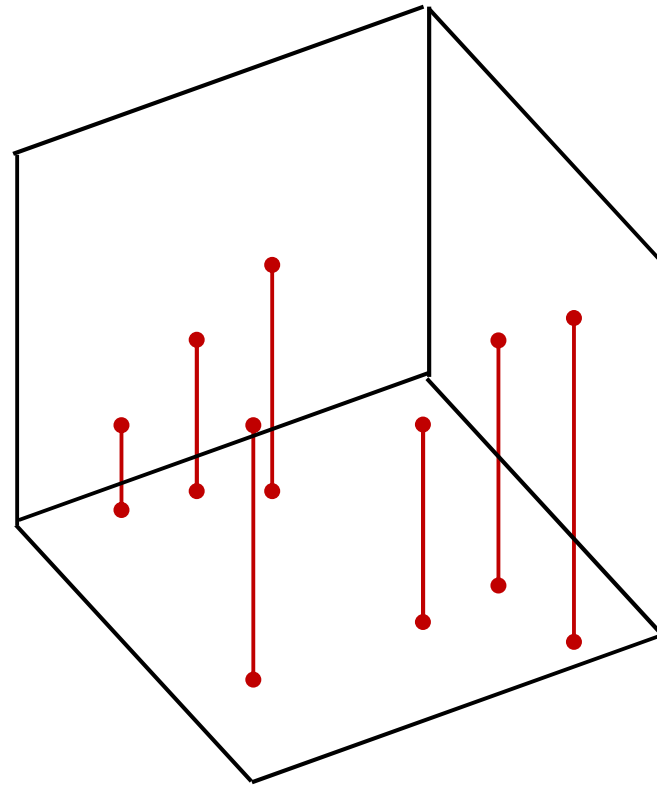
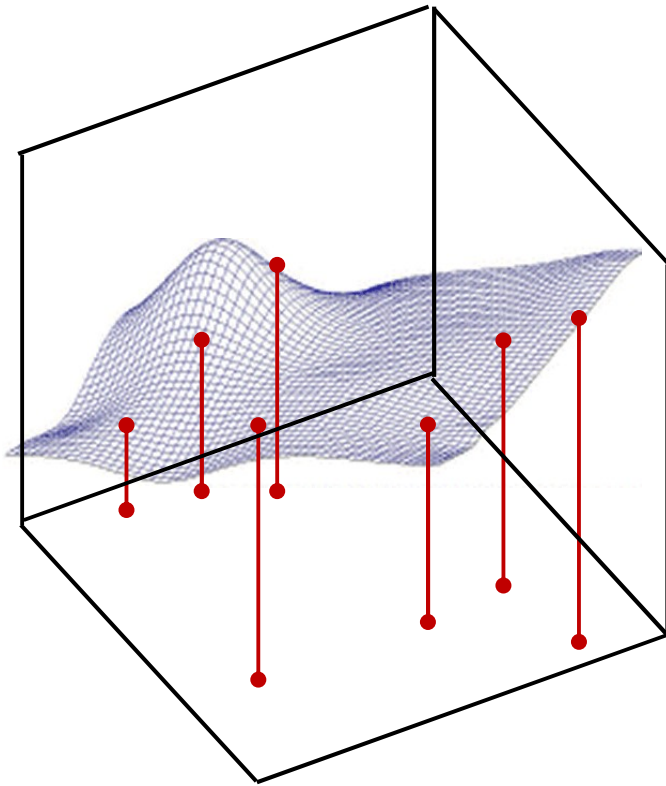


# The problem of data underspecification

- The figures shown so far were *fake news*..

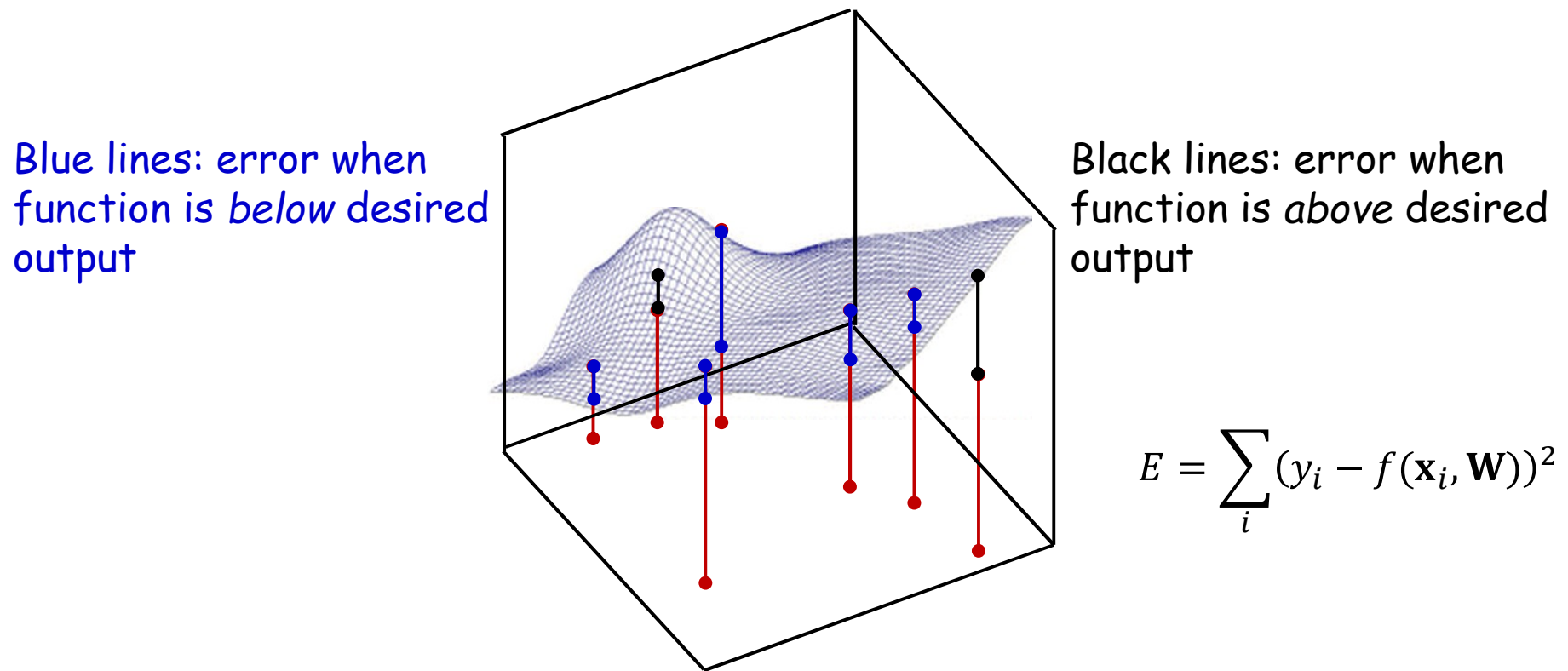


# Learning the network



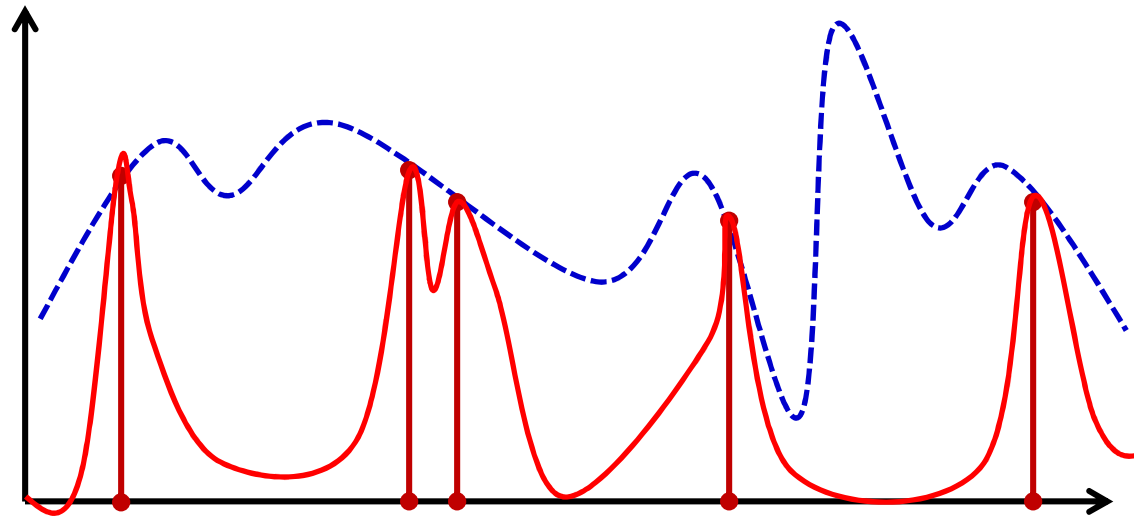
- We attempt to learn an entire function from just a few *snapshots* of it

# General approach to training



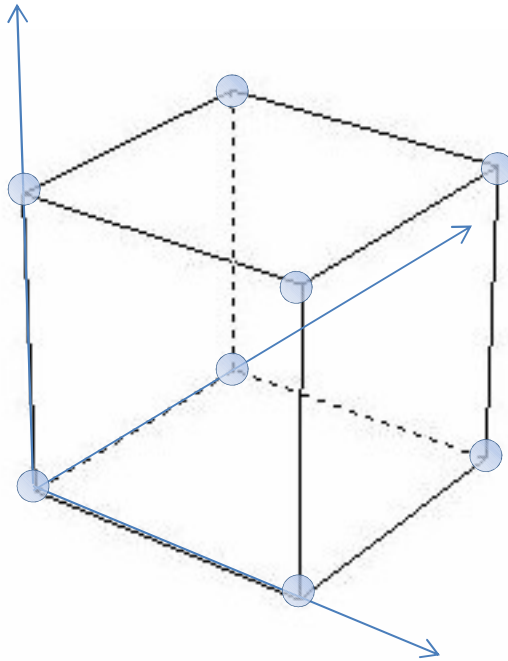
- Define an *error* between the **actual** network output for any parameter value and the *desired* output
  - Error typically defined as the *sum* of the squared error over individual training instances

# Overfitting



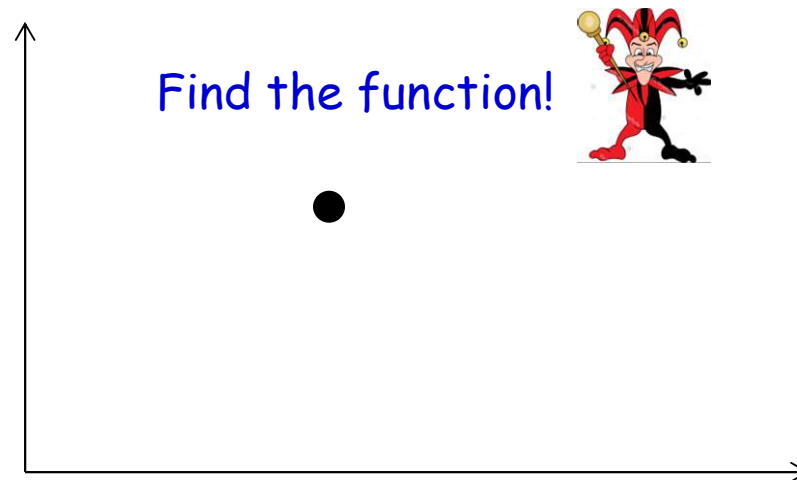
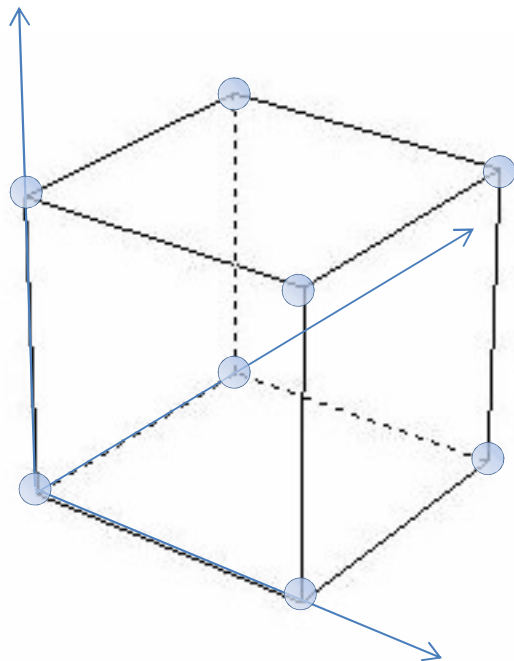
- Problem: Network may just learn the values at the inputs
  - Learn the red curve instead of the dotted blue one
    - Given only the red vertical bars as inputs

# Data under-specification



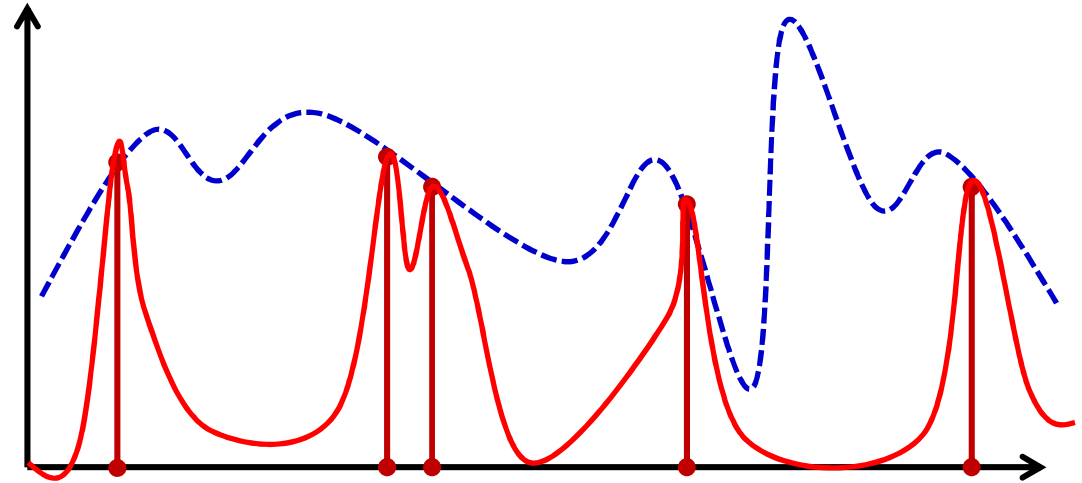
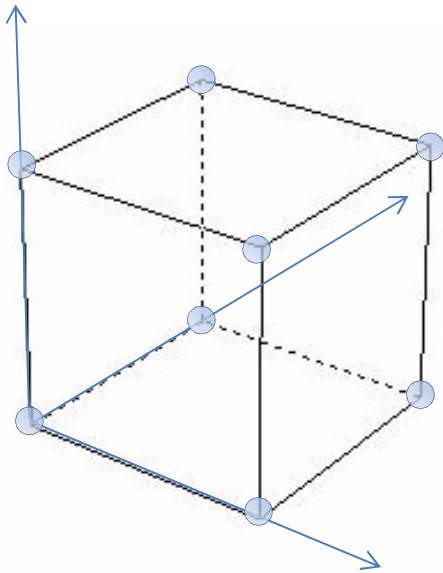
- Consider a binary 100-dimensional input
- There are  $2^{100} = 10^{30}$  possible inputs
- Complete specification of the function will require specification of  $10^{30}$  output values
- A training set with only  $10^{15}$  training instances will be off by a factor of  $10^{15}$

# Data under-specification in learning



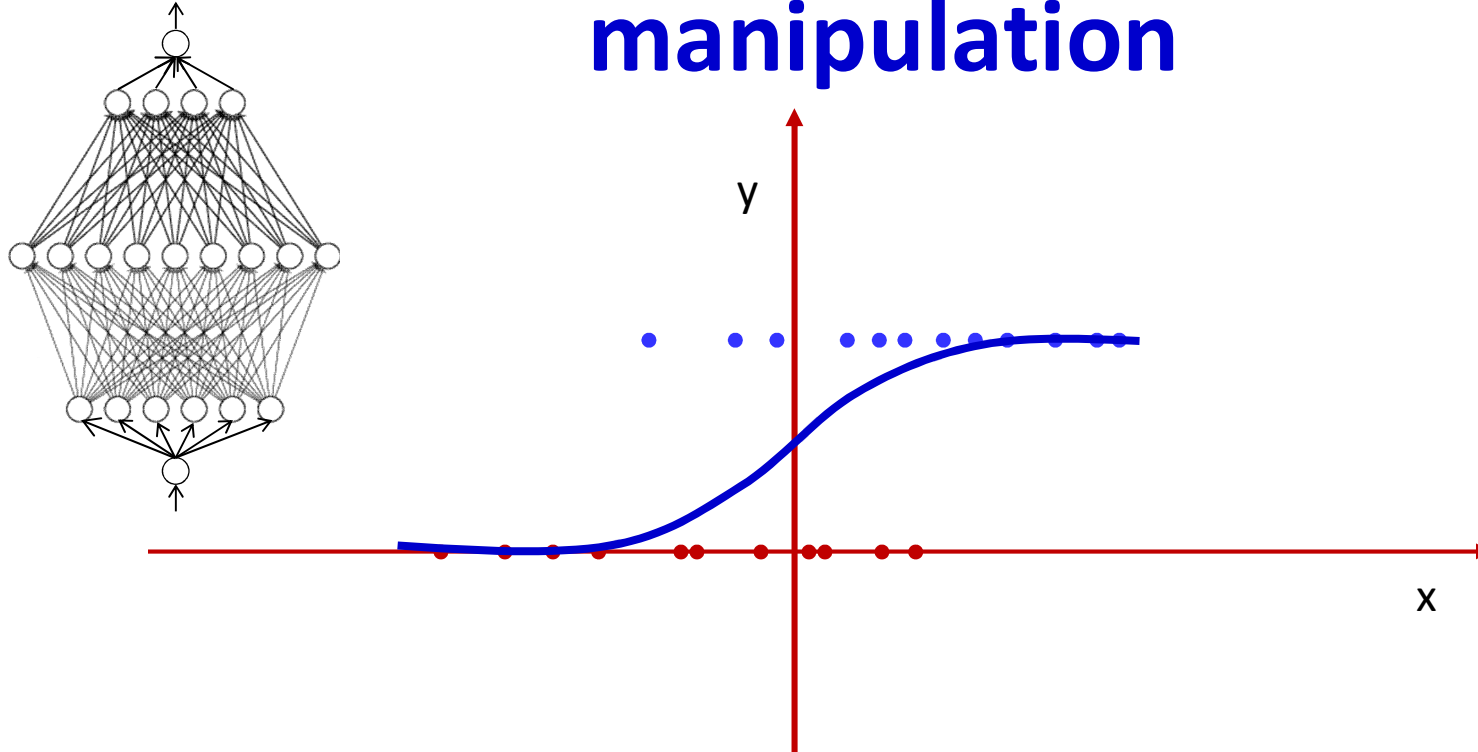
- Consider a binary 100-dimensional input
- There are  $2^{100}=10^{30}$  possible inputs
- Complete specification of the function will require specification of  $10^{30}$  output values
- A training set with only  $10^{15}$  training instances will be off by a factor of  $10^{15}$

# Need “smoothing” constraints



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

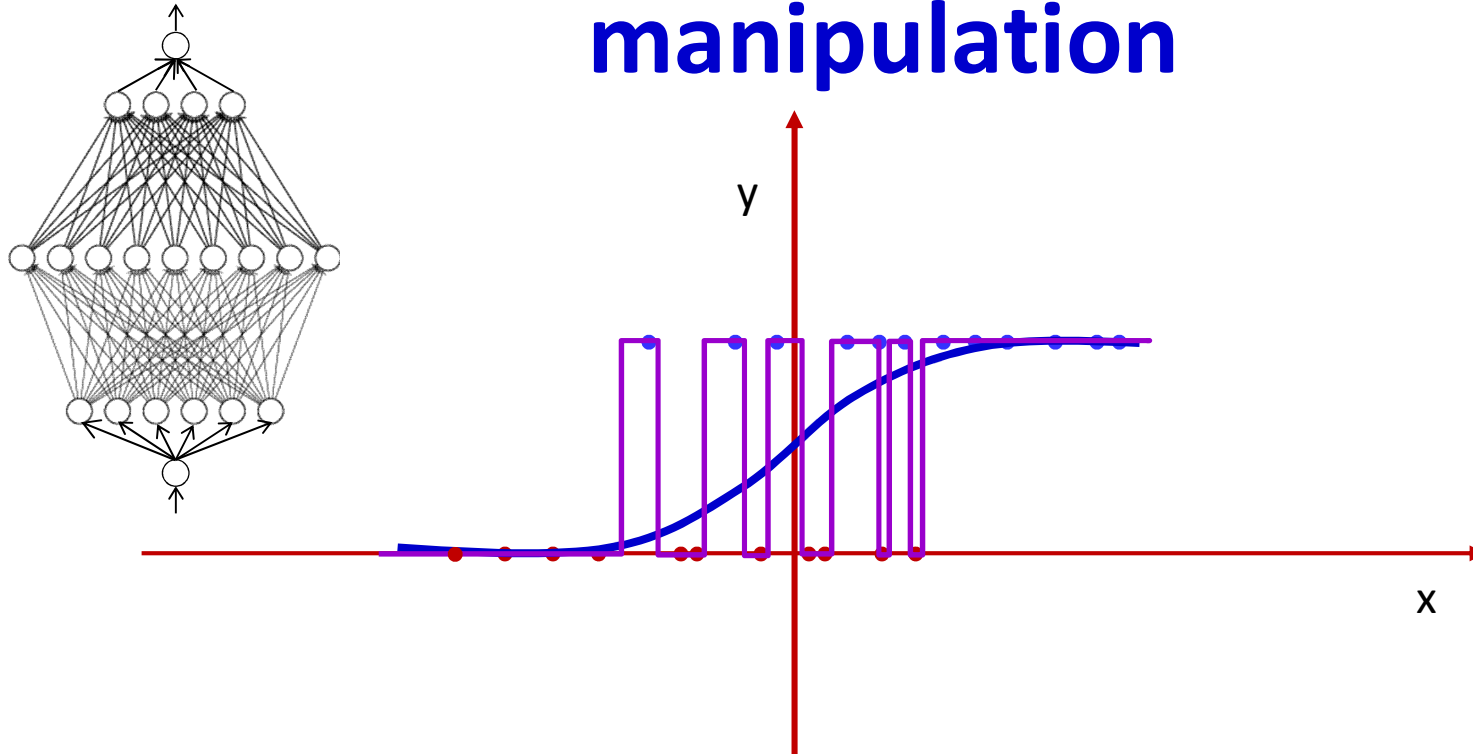
# Smoothness through weight manipulation



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

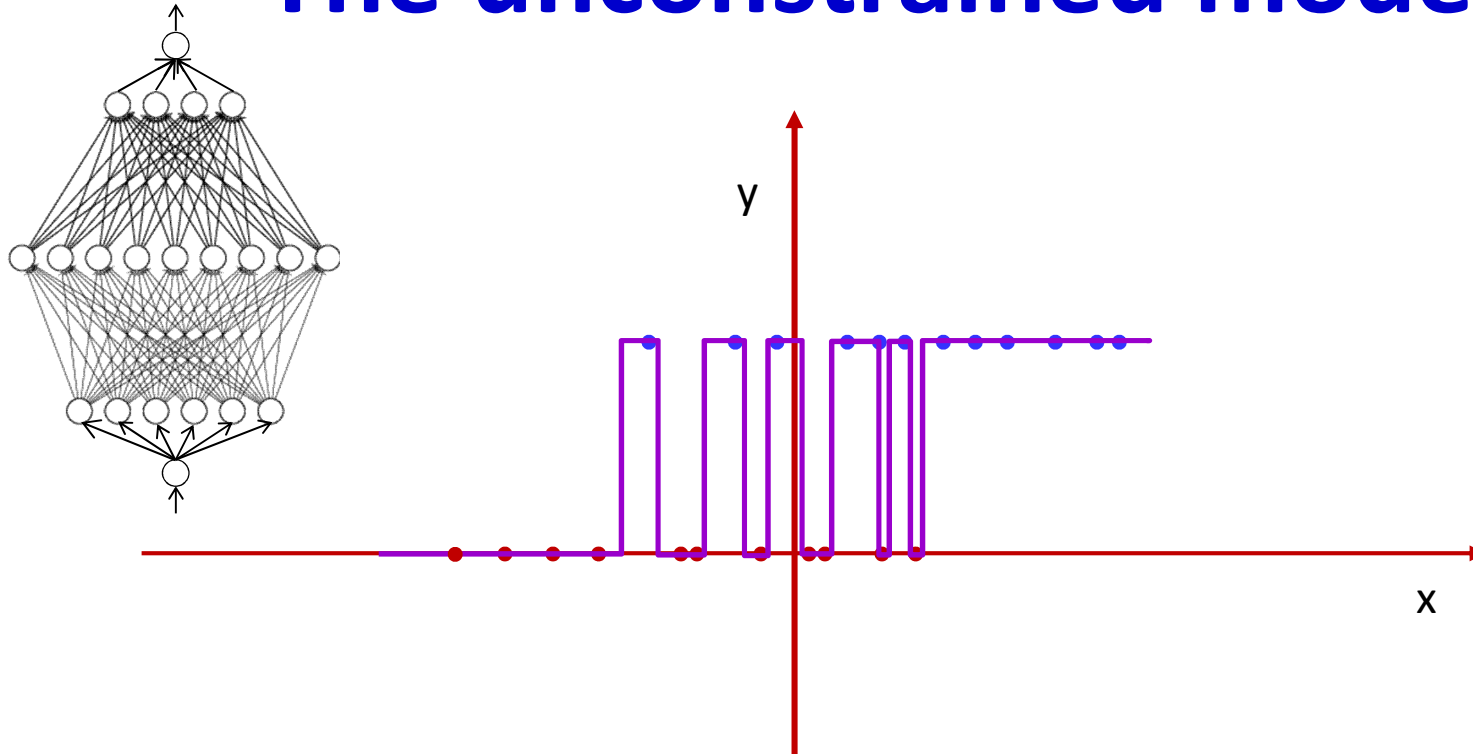


# Smoothness through weight manipulation



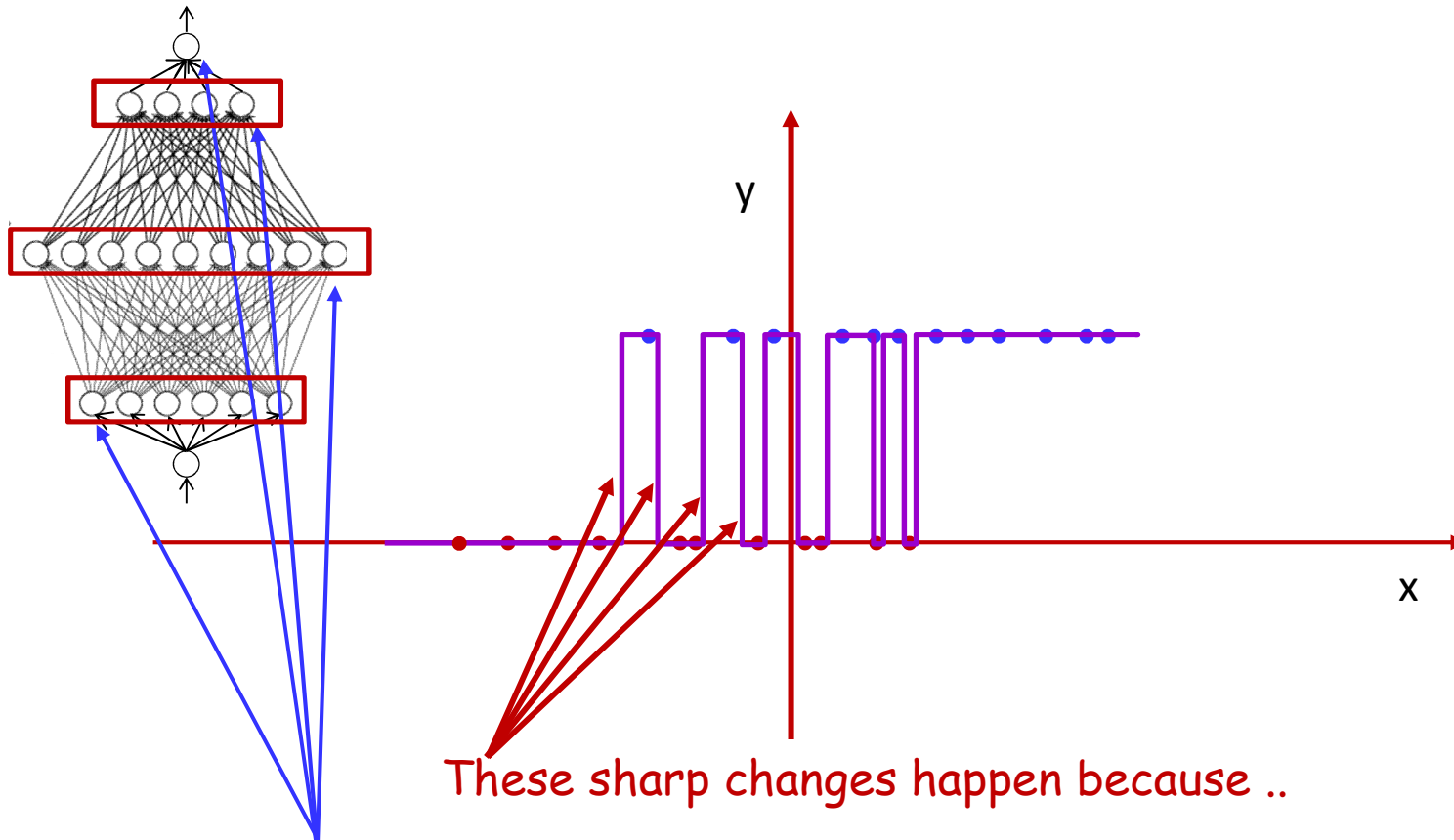
- Illustrative example: Simple binary classifier
  - The “desired” output is generally smooth
    - Capture statistical or average trends
  - An unconstrained model will model individual instances instead

# 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

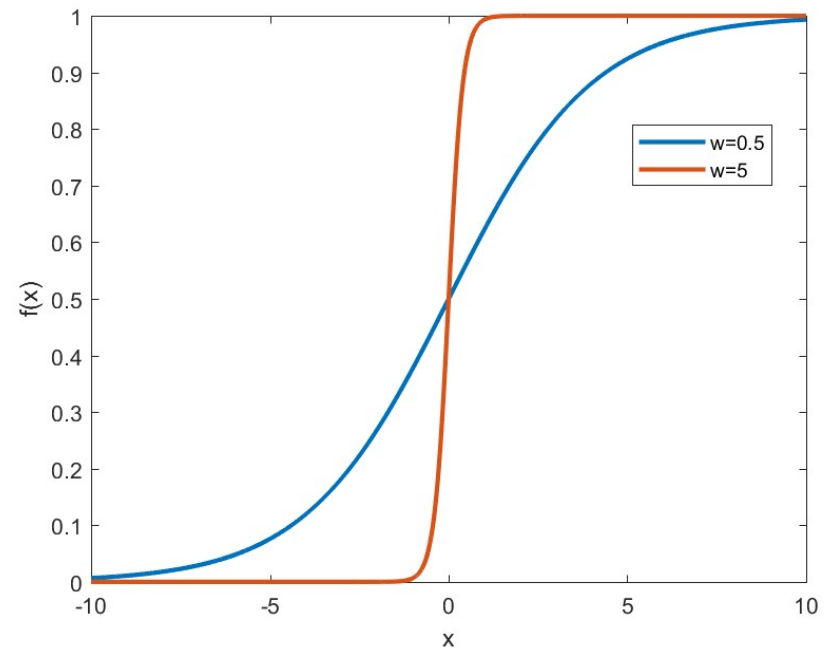
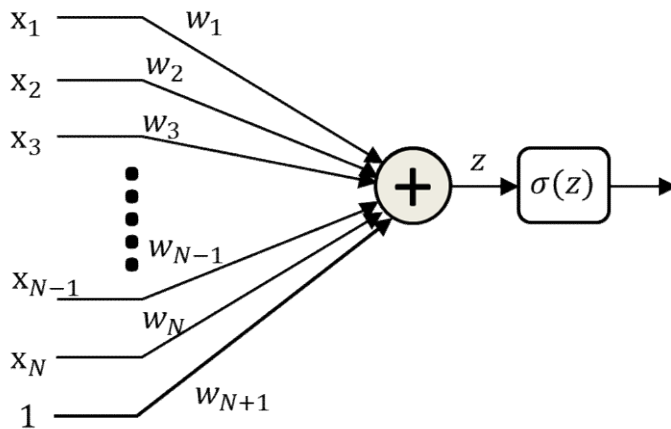
# Why overfitting



These sharp changes happen because ..

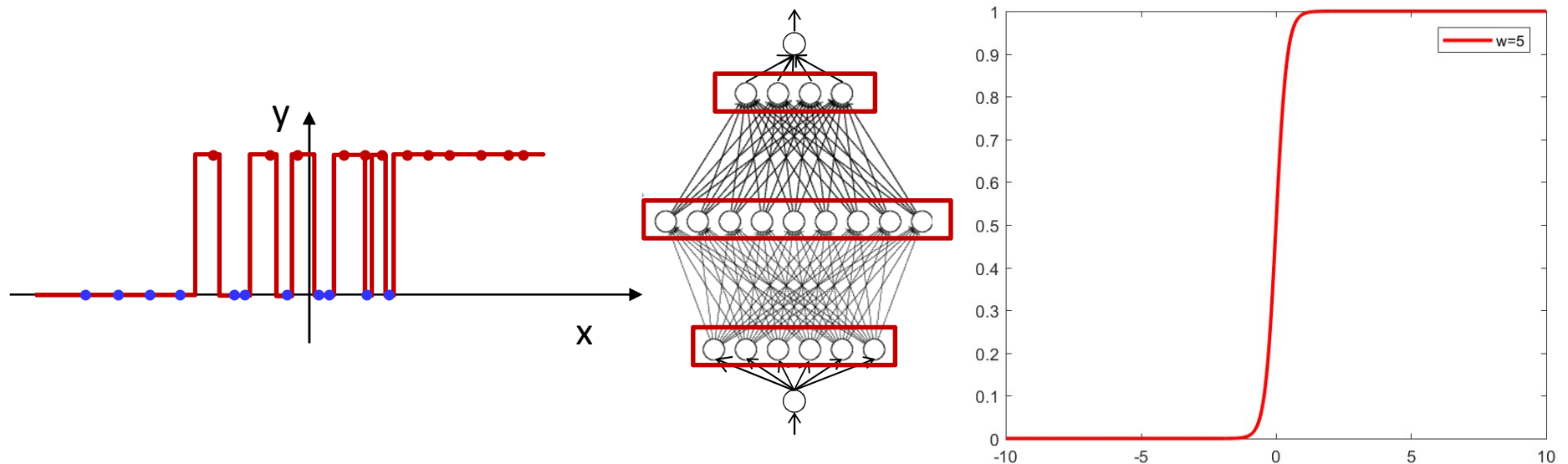
..the perceptrons in the network are individually capable of sharp changes in output

# The individual perceptron



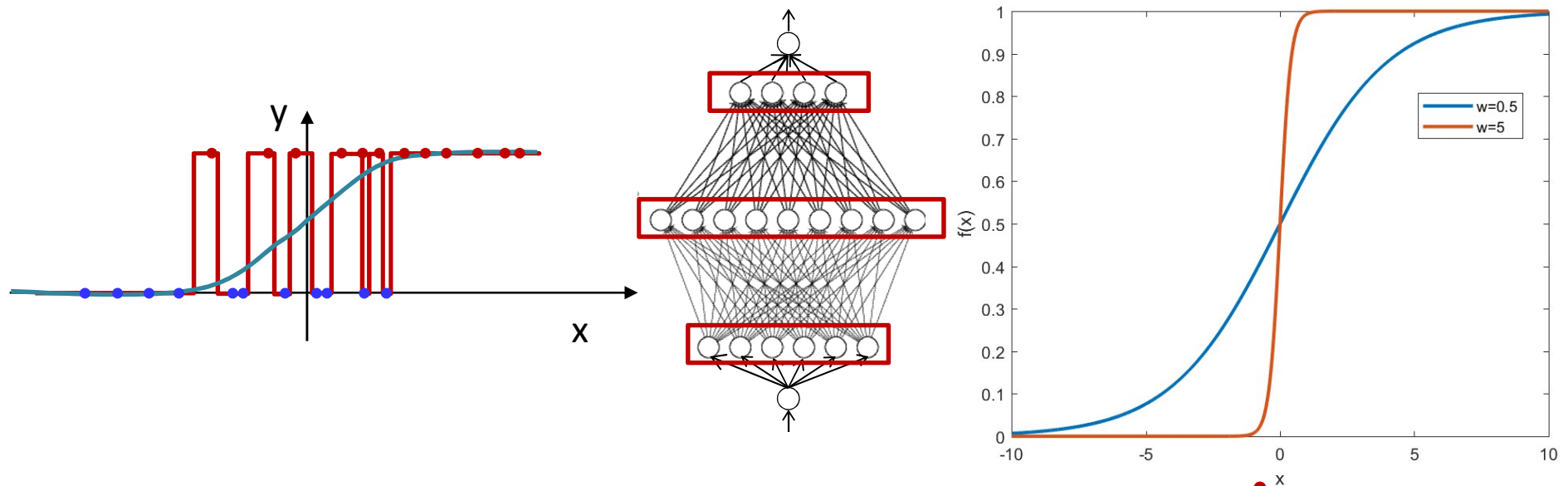
- Using a sigmoid activation
  - As  $|w|$  increases, the response becomes steeper

# Smoothness through weight manipulation



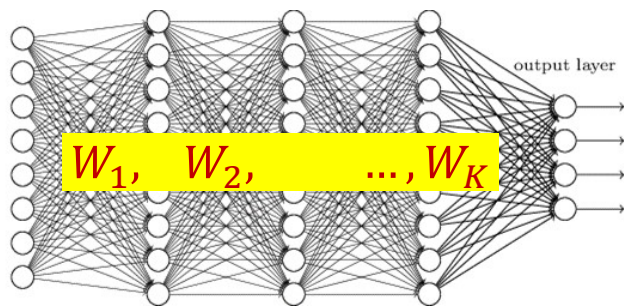
- Steep changes that enable overfitted responses are facilitated by perceptrons with large  $w$

# 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:

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

# Smoothness through weight constraints

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

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

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

- $\lambda$  is the regularization parameter whose value depends on how important it is for us to want to minimize the weights
- Increasing  $\lambda$  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:

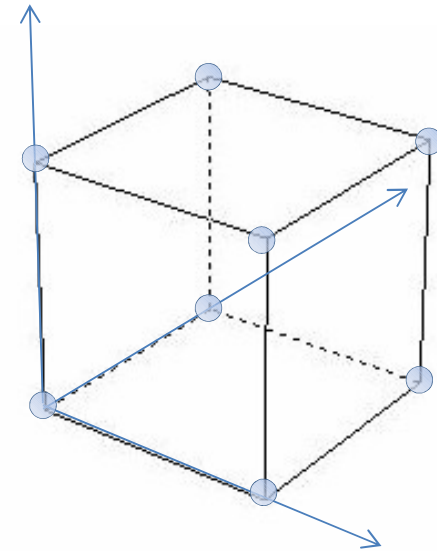
$$W_k \leftarrow W_k - \eta \Delta W_k$$

# Incremental Update: Mini-batch update

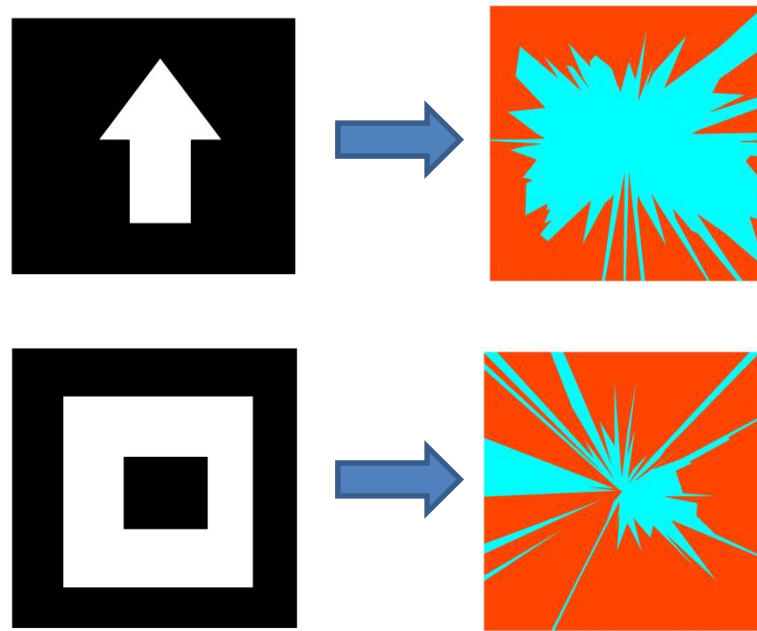
- Given  $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights  $W_1, W_2, \dots, W_K; j = 0$
- Do:
  - Randomly permute  $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
  - For  $t = 1:b:T$ 
    - $j = j + 1$
    - For every layer  $k$ :
      - $\Delta W_k = 0$
    - For  $t' = t : t+b-1$ 
      - 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

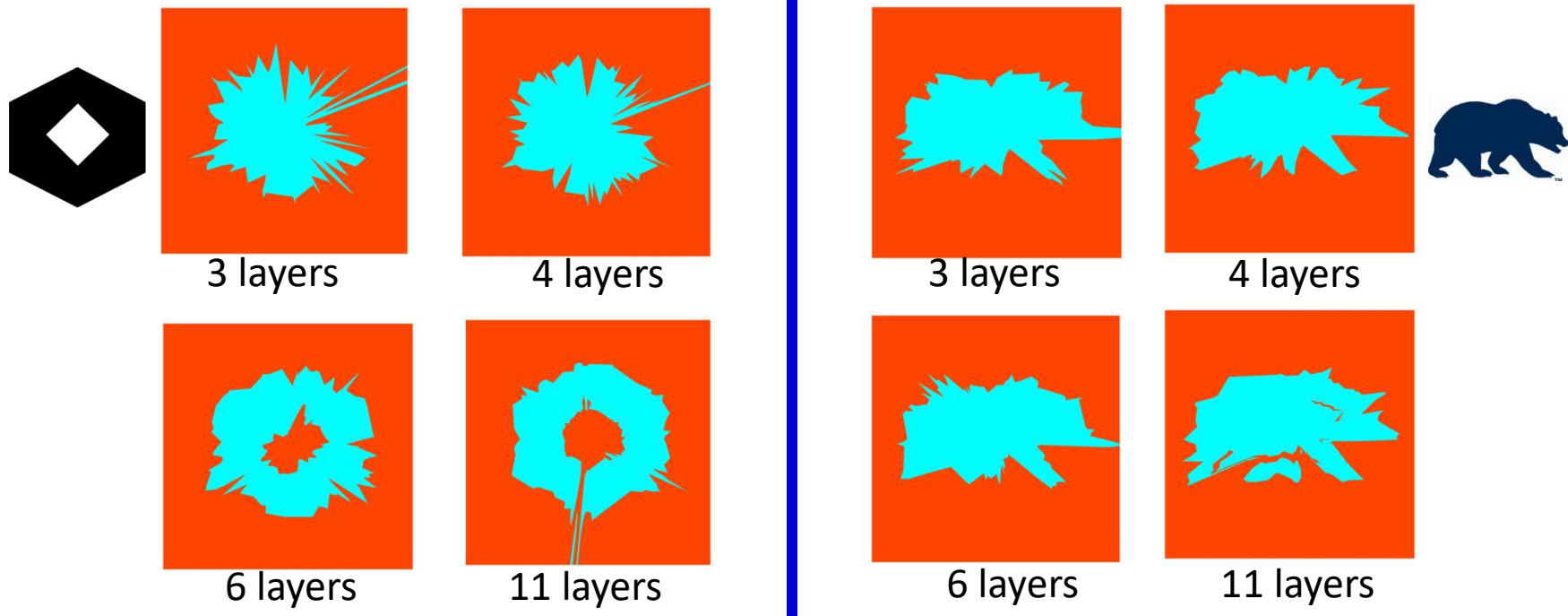


# Even when we get it all right



- 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!!**

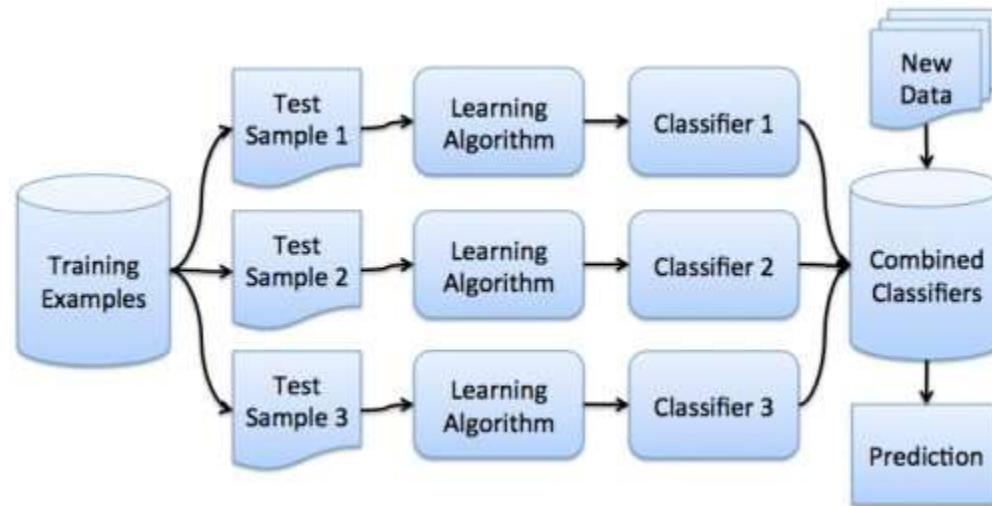
10000 training instances



# Regularization..

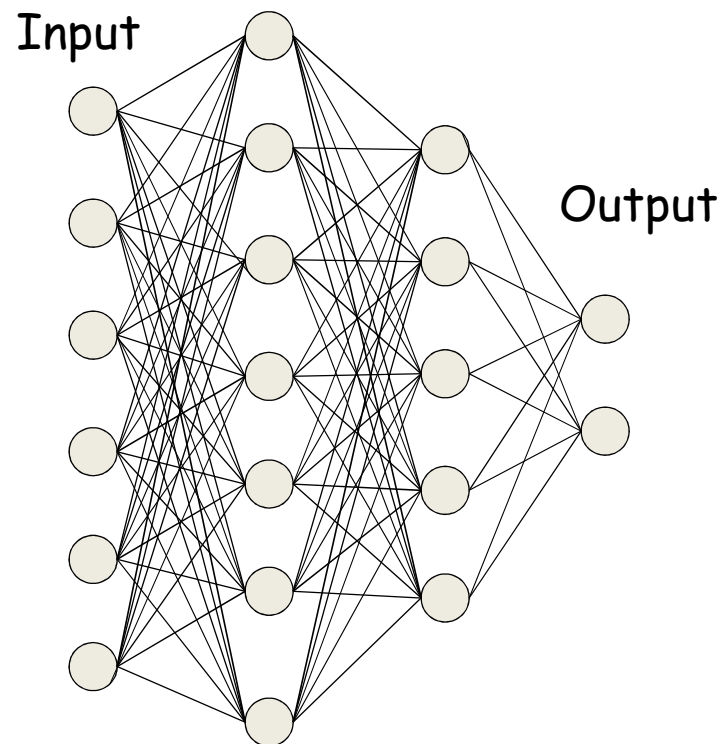
- Other techniques have been proposed to improve the smoothness of the learned function
  - $L_1$  regularization of network activations
  - Regularizing with added noise..
- Possibly the most influential method has been “dropout”

# A brief detour.. Bagging



- Popular method proposed by Leo Breiman:
  - Sample training data and train several different classifiers
  - Classify test instance with entire ensemble of classifiers
  - Vote across classifiers for final decision
  - Empirically shown to improve significantly over training a single classifier from combined data
- Returning to our problem....

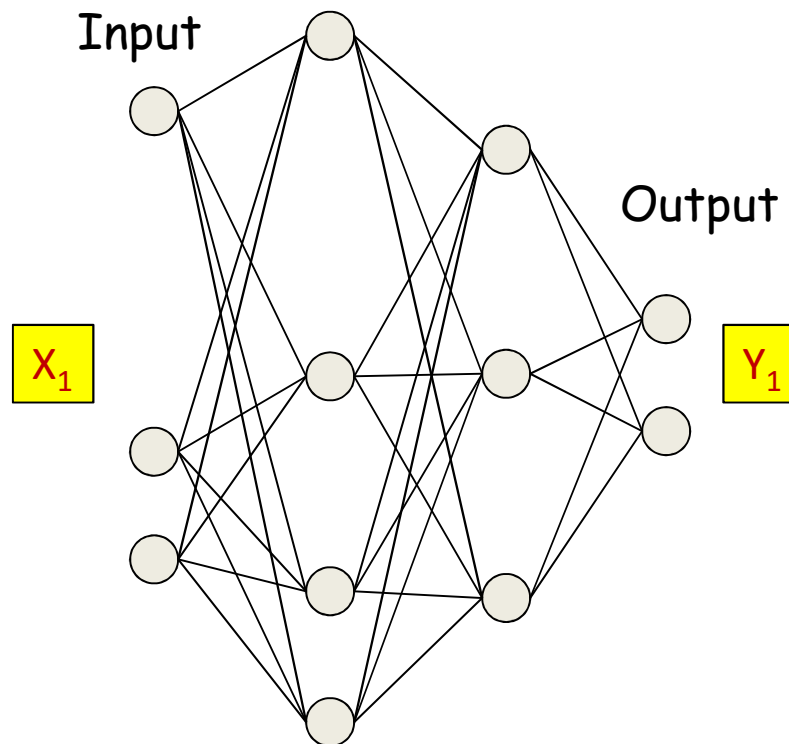
# Dropout



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

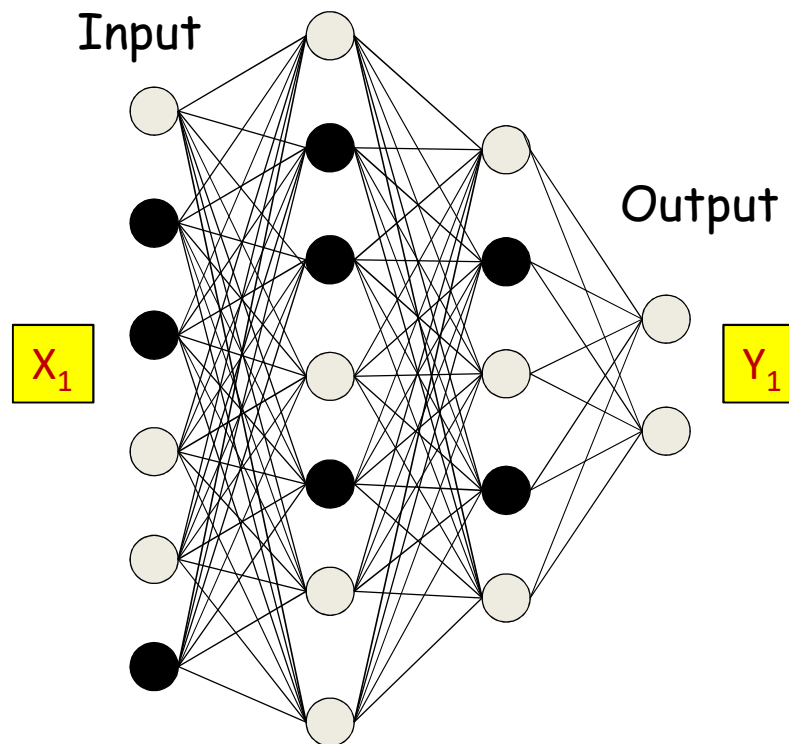


# Dropout



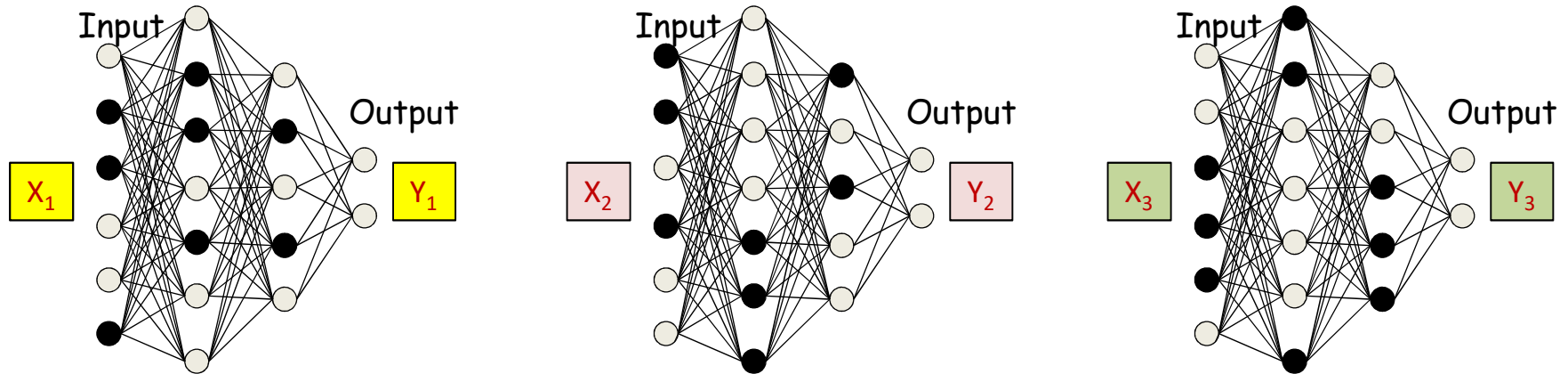
- **During training:** For each input, at each iteration, “turn off” each neuron with a probability  $1-\alpha$ 
  - Also turn off inputs similarly

# 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  $1-\alpha$

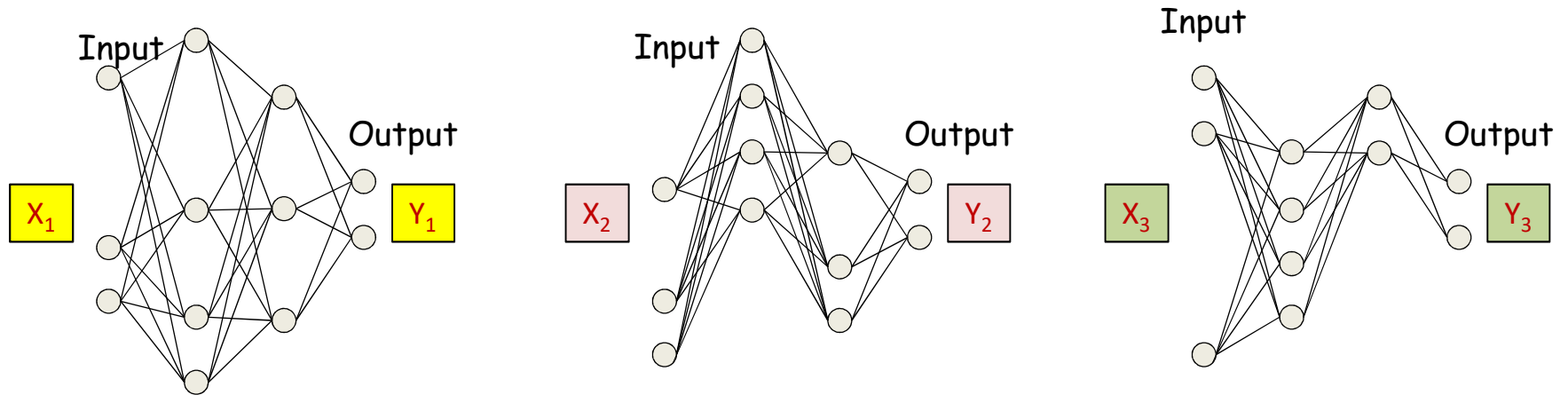
# Dropout



*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  $1-\alpha$

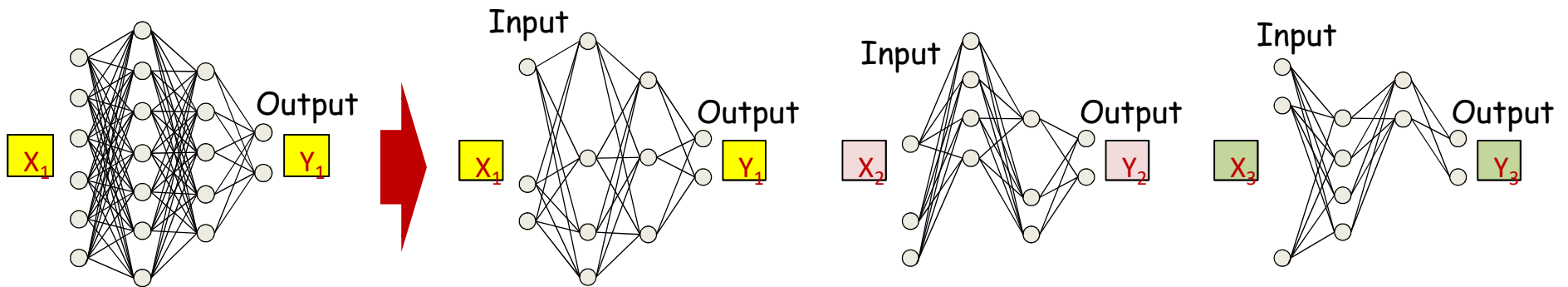
# Dropout



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
  - Effectively 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 0<sup>th</sup> (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  $\text{mask}(k, j) == 0$ 
      - »  $y_j^{(k)} = 0$

- 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:

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

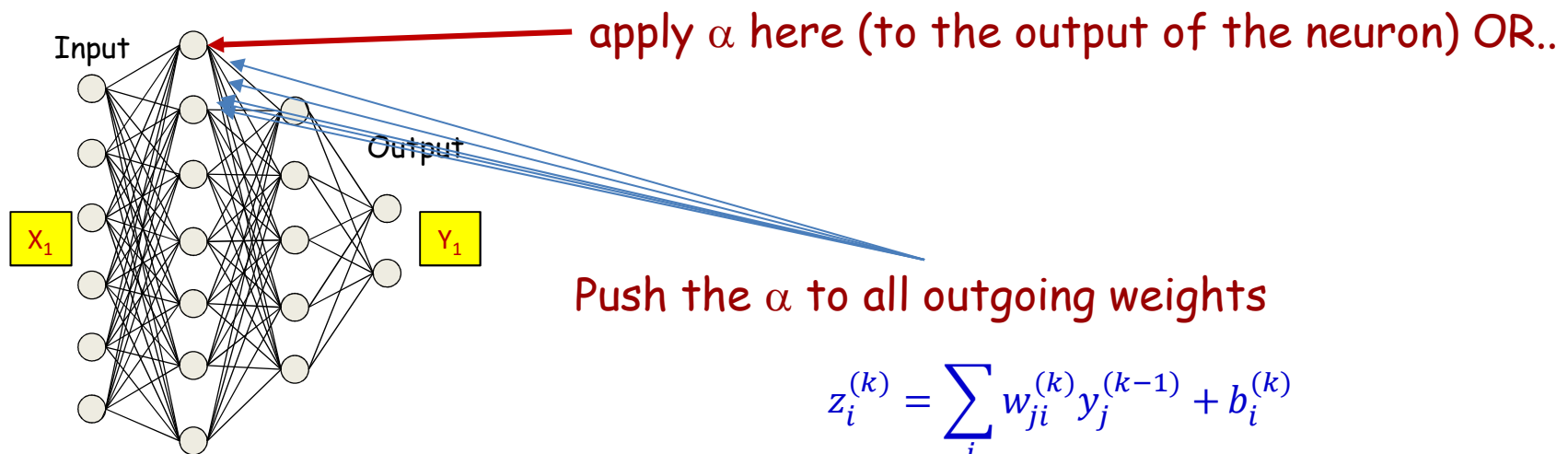
- Where  $D$  is a Bernoulli variable that takes a value 1 with probability  $\alpha$
- $D$  may be switched on or off for individual sub networks, but over the ensemble, the *expected output* of the neuron is

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

- 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  $\alpha$



# Dropout during test: implementation



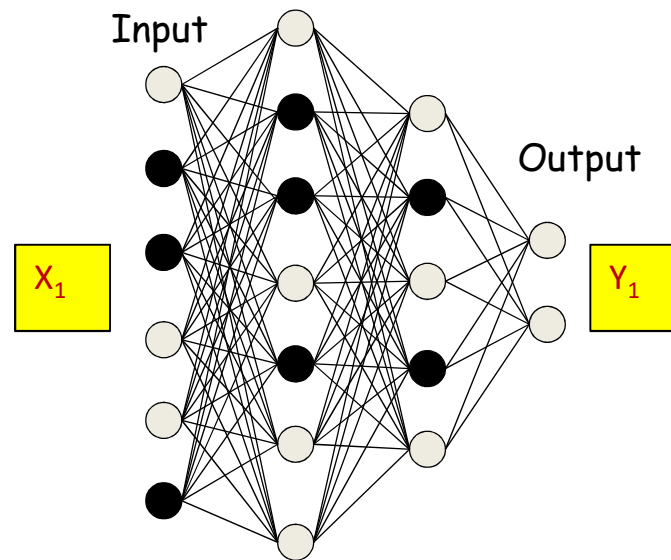
$$y_i^{(k)} = \alpha \sigma(z_i^{(k)})$$

$$\begin{aligned} z_i^{(k)} &= \sum_j w_{ji}^{(k)} y_j^{(k-1)} + b_i^{(k)} \\ &= \sum_j w_{ji}^{(k)} \alpha \sigma(z_j^{(k-1)}) + b_i^{(k)} \\ &= \sum_j (\alpha w_{ji}^{(k)}) \sigma(z_j^{(k-1)}) + b_i^{(k)} \end{aligned}$$

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

- Instead of multiplying every output by  $\alpha$ , multiply all weights by  $\alpha$

# 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

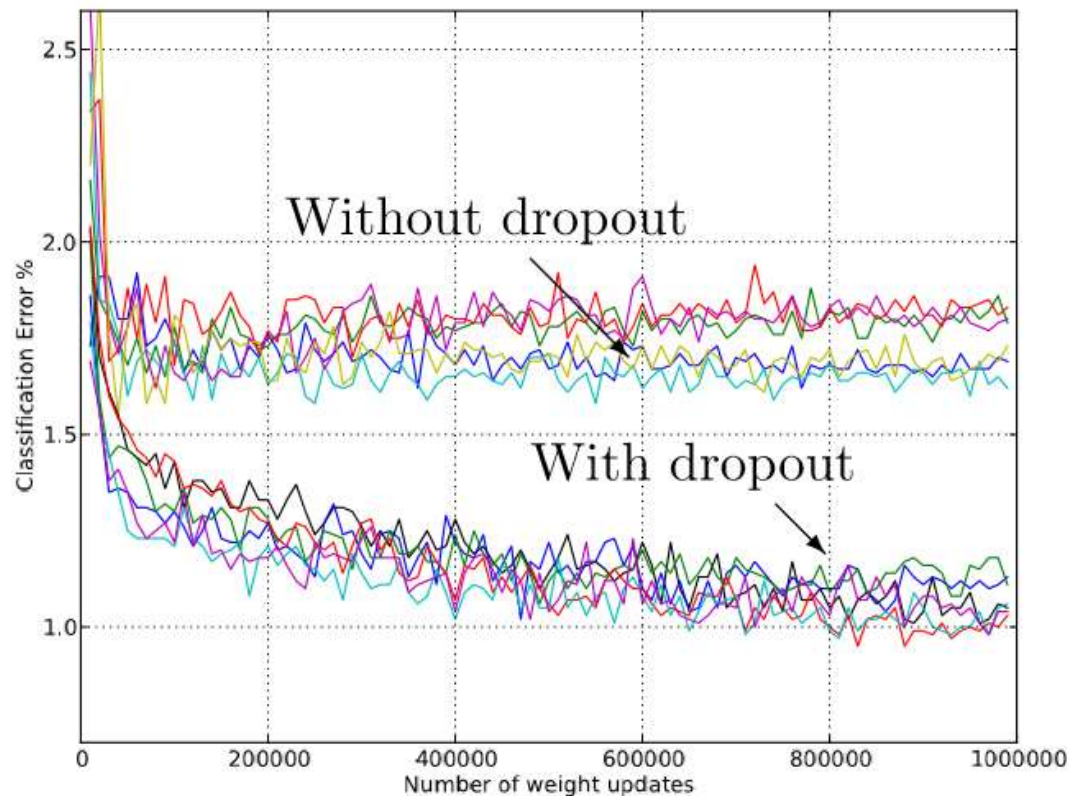
# The forward pass (training)

- Input:  $D$  dimensional vector  $\mathbf{x} = [x_j, j = 1 \dots D]$
- Set:
  - $D_0 = D$ , is the width of the 0<sup>th</sup> (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:
  - $Y = y_j^{(N)}, j = 1 \dots D_N$

# Dropout: Typical results

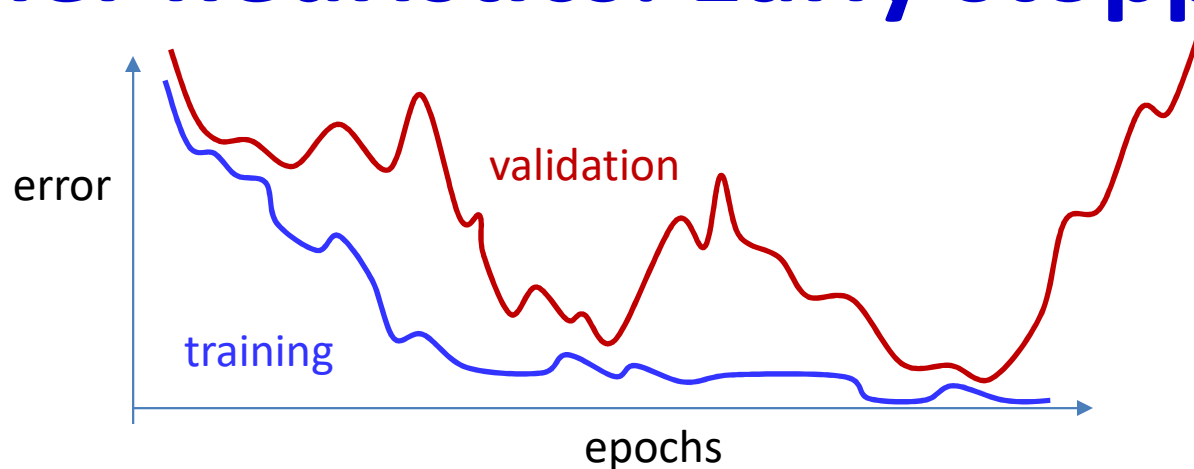


- From Srivastava et al., 2013. Test error for different architectures on MNIST with and without dropout
  - 2-4 hidden layers with 1024-2048 units

# Variations on dropout

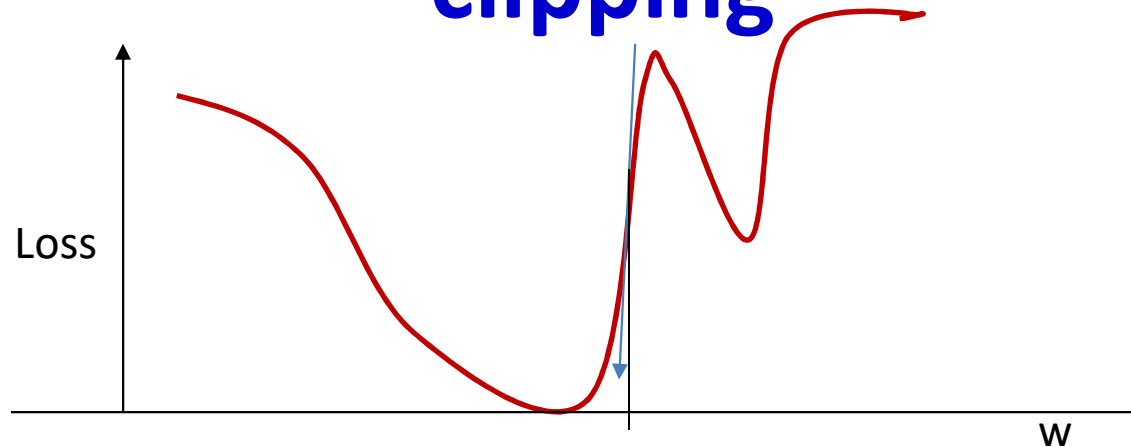
- Zoneout: For RNNs
  - Randomly chosen units remain unchanged across a time transition
- Dropconnect
  - Drop individual connections, instead of nodes
- Shakeout
  - Scale *up* the weights of randomly selected weights
    - $|w| \rightarrow \alpha|w| + (1 - \alpha)c$
  - Fix remaining weights to a negative constant
    - $w \rightarrow -c$
- Whiteout
  - Add or multiply weight-dependent Gaussian noise to the signal on each connection

# 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  $\theta$  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..