Training Neural Networks: Optimization: Part 2

Intro to Deep Learning, Spring 2019

Story so far

- SGD: Presenting training instances one-at-a-time can be more effective than full-batch training
  - Provided they are provided in random order
- For SGD to converge, the learning rate must shrink sufficiently rapidly with iterations
  - Otherwise the learning will continuously "chase" the latest sample
- SGD estimates have higher variance than batch estimates
- Minibatch updates operate on *batches* of instances at a time
  - Estimates have lower variance than SGD
  - Convergence rate is theoretically worse than SGD
  - But we compensate by being able to perform batch processing

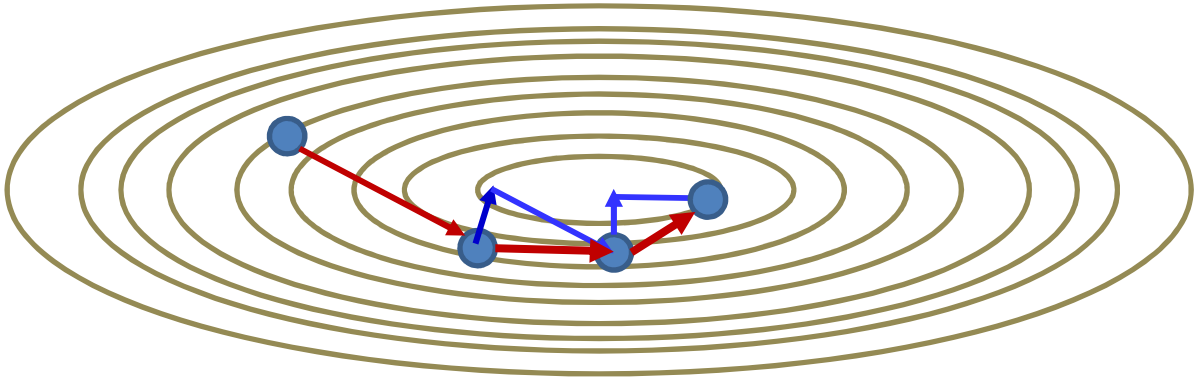
Training and minibatches

- Convergence depends on learning rate
  - Simple technique: fix learning rate until the error plateaus, then reduce learning rate by a fixed factor (e.g. 10)
  - ***Advanced methods***: Adaptive updates, where the learning rate is itself determined as part of the estimation

Moving on: Topics for the day

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

Recall: Momentum

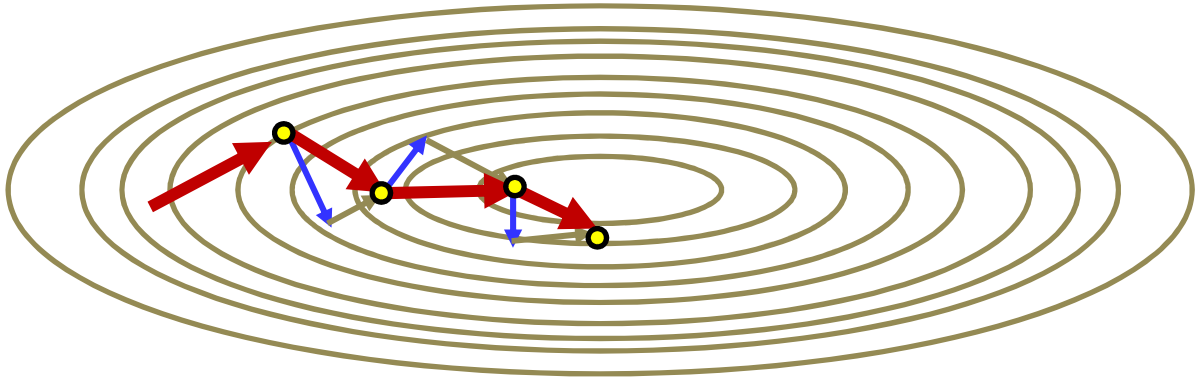


- The momentum method

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

- Updates using a running average of the gradient

Momentum and incremental updates



- The momentum method

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

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

Incremental Update: Mini-batch update

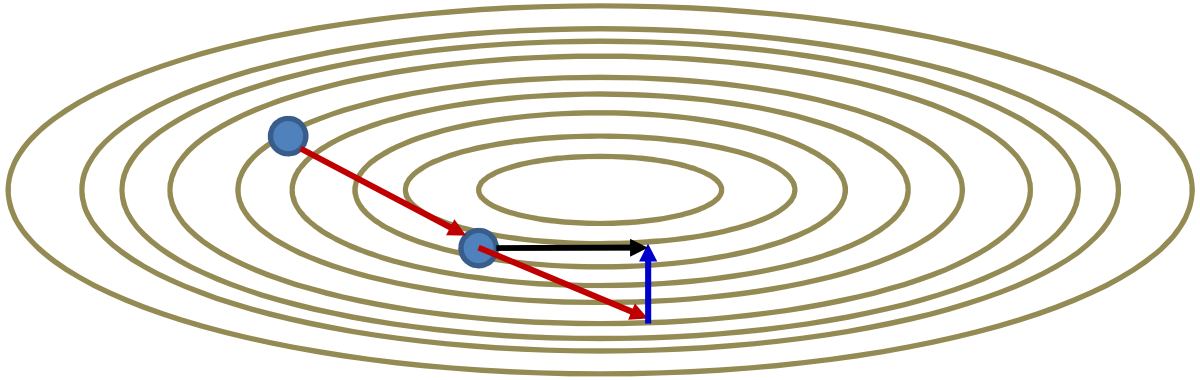
- Given $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$
- Initialize all weights $W_1, W_2, \ldots, W_K$; $j = 0$, $\Delta W_k = 0$
- Do:
  - Randomly permute $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$
  - For $t = 1 : b : T$
    - $j = j + 1$
    - For every layer k:
      - $\nabla_{W_k} Err = 0$
 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

$$\Delta W_k = \beta \Delta W_k - \eta_j \nabla_{W_k} Err$$

$$W_k = W_k + \Delta W_k$$

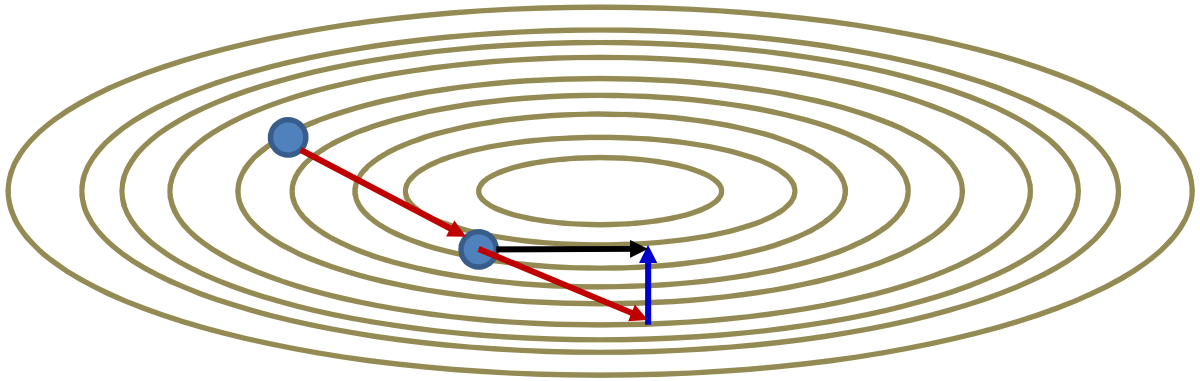
- Until Err has converged

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

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

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

Incremental Update: Mini-batch update

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

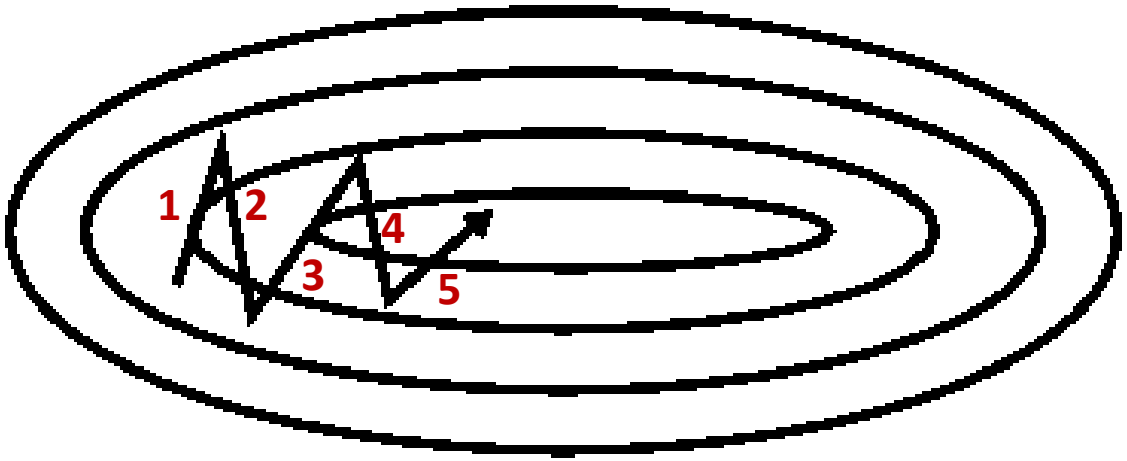
$$W_k = W_k - \eta_j \nabla_{W_k} Err$$
$$\Delta W_k = \beta \Delta W_k - \eta_j \nabla_{W_k} Err$$

- 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
  - Adagrad
  - AdaDelta
  - ADAM: very popular in practice
  - …
- 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 $\partial_w^2 D = (\partial_w D)^2$
    - Short-hand notation represents the squared derivative, not the second derivative
  - The ***mean squared*** derivative is a running estimate of the average squared derivative. We will show this as $E[\partial_w^2 D]$
- 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

- 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

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

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

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

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

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

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

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

    - $k = k+1$
- Until $E(\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \ldots, \boldsymbol{W}^{(K)})$ has converged

ADAM: RMSprop with momentum

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

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

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

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

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

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

ADAM: RMSprop with momentum

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

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

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

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

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

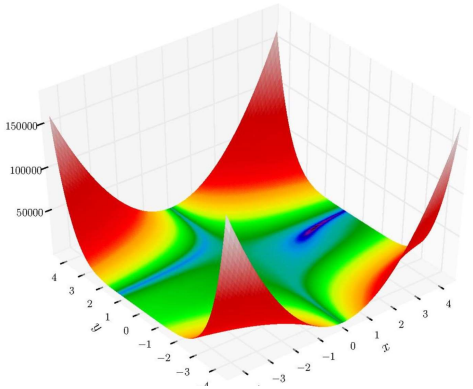
$$\hat{m}_k = \frac{m_k}{1-\delta^k}, \qquad \hat{v}_k = \frac{v_k}{1-\gamma^k}$$

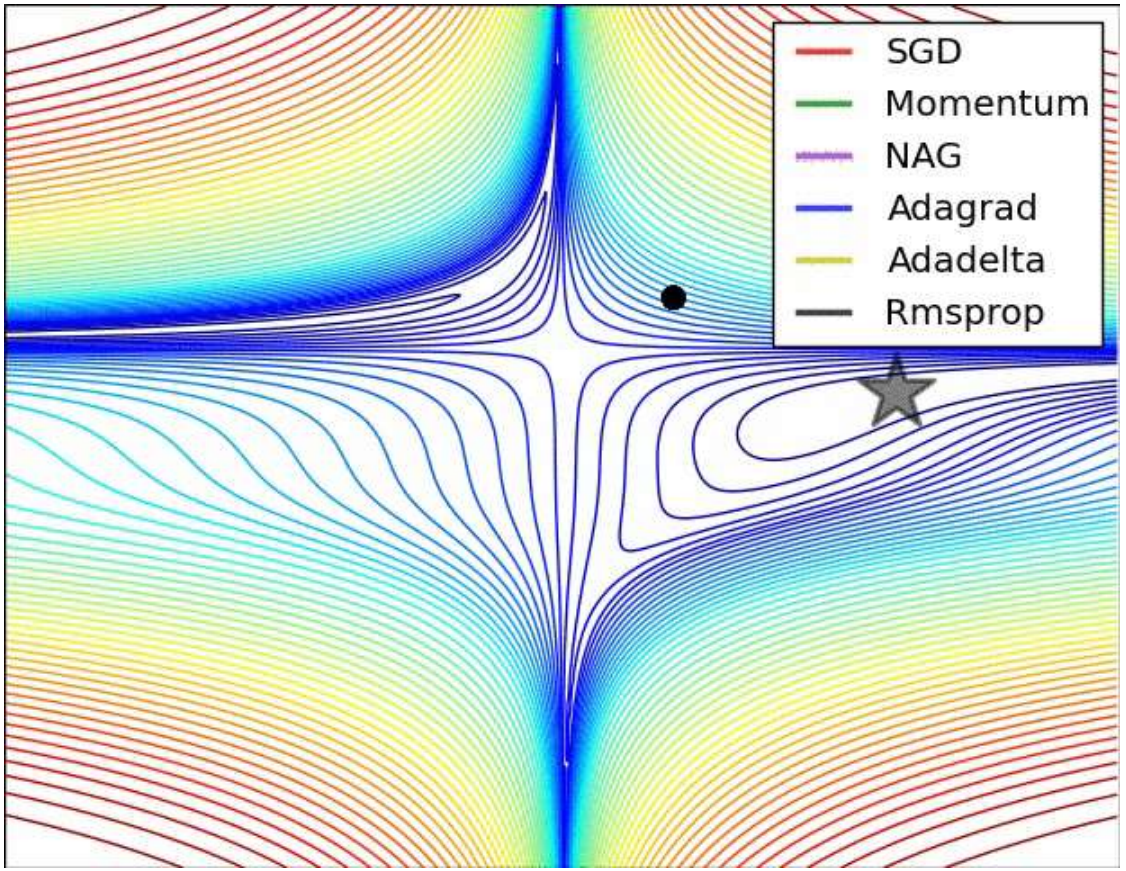
$$w_{k+1} = w_k - \frac{\eta}{\sqrt{\hat{v}_k + \epsilon}}\hat{m}_k$$

Other variants of the same theme

- Many:
  - Adagrad
  - AdaDelta
  - ADAM
  - AdaMax
  - …
- Generally no explicit learning rate to optimize
  - But come with other hyper parameters to be optimized
  - Typical params:
    - RMSProp: $\eta = 0.001, \gamma = 0.9$
    - ADAM: $\eta = 0.001, \delta = 0.9, \gamma = 0.999$

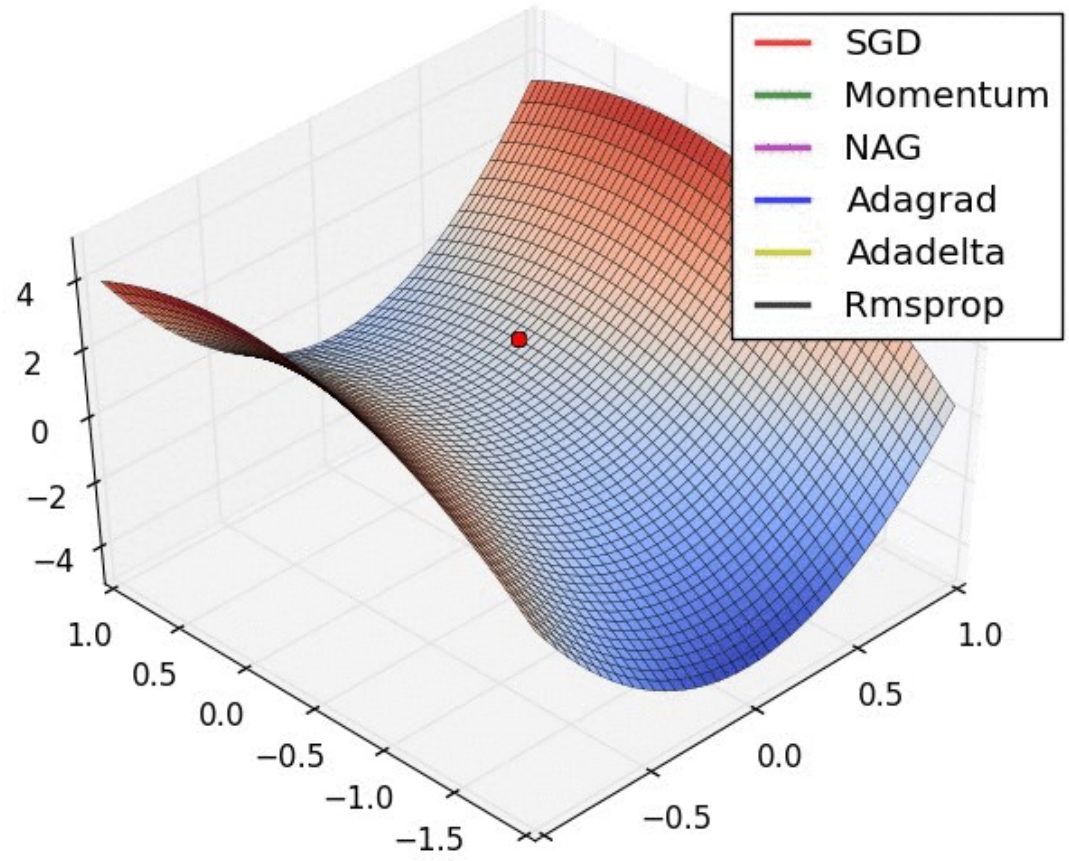
Visualizing the optimizers: Beale's Function





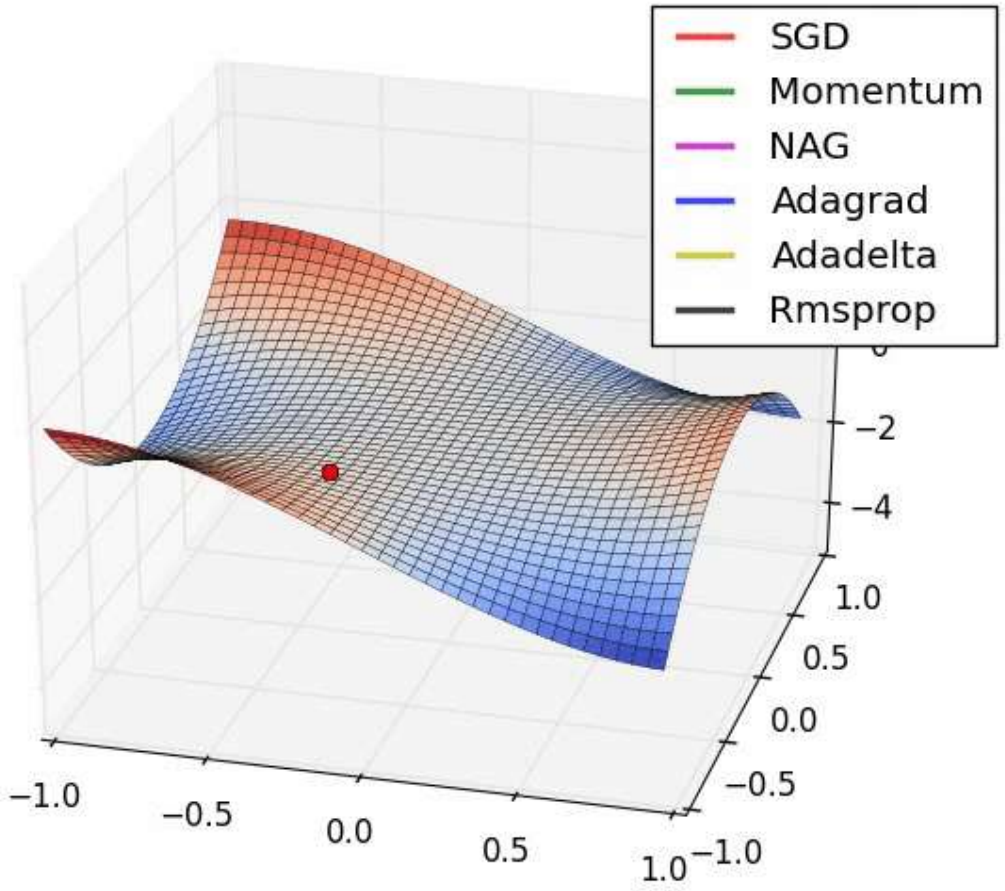
- http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Visualizing the optimizers: Long Valley



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

Visualizing the optimizers: Saddle Point



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

Story so far

- Gradient descent can be sped up by incremental updates
  - Convergence is guaranteed under most conditions
    - Learning rate must shrink with time for convergence
  - Stochastic gradient descent: update after each observation. Can be much faster than batch learning
  - Mini-batch updates: update after batches. Can be more efficient than SGD

- Convergence can be improved using smoothed updates
  - RMSprop and more advanced techniques

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..

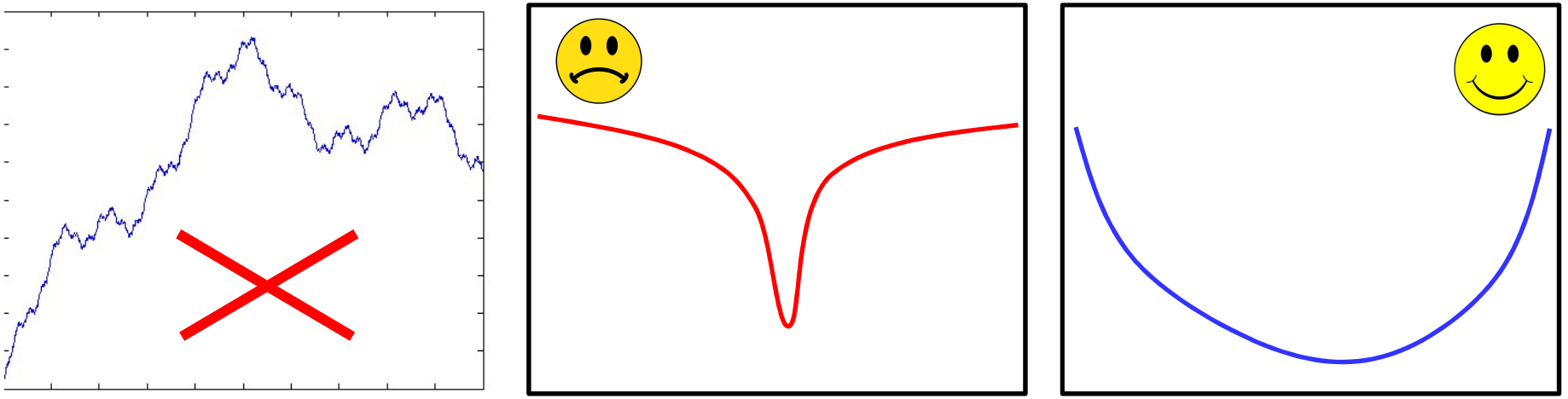
Training Neural Nets by Gradient Descent: The Divergence

Total training error:

$$Err = \frac{1}{T}\sum_{t} Div(\boldsymbol{Y}_t, \boldsymbol{d}_t; \mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K)$$

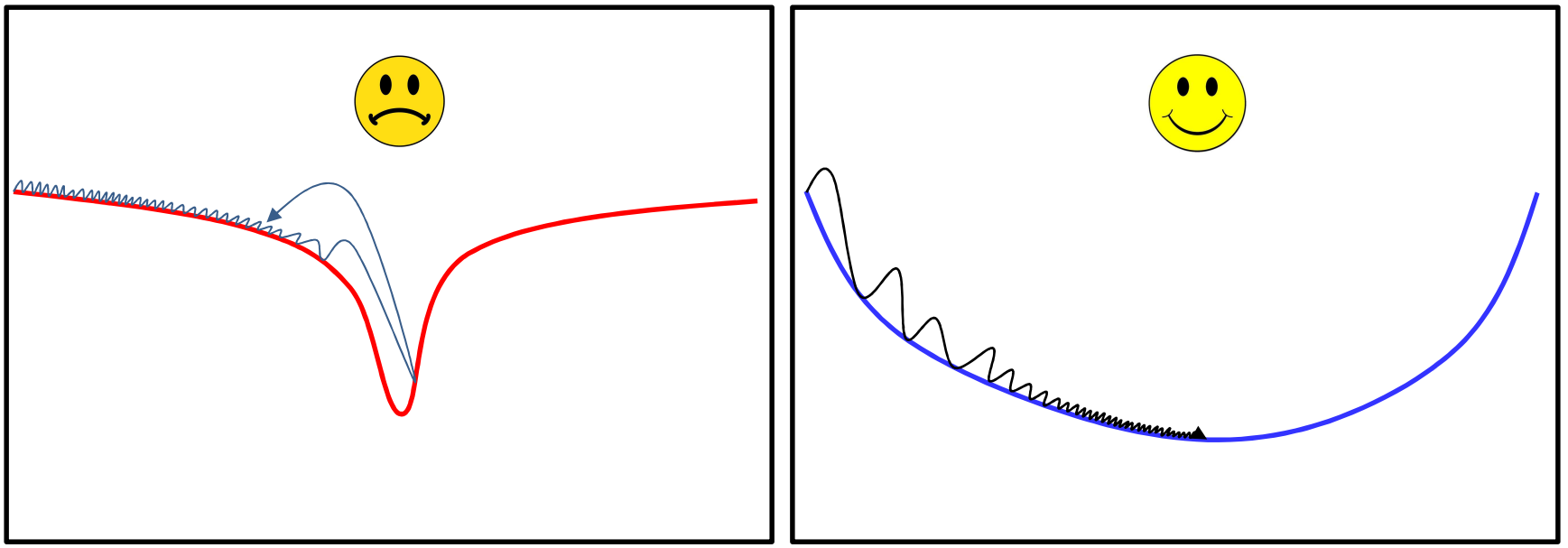
- The convergence of the gradient descent depends on the divergence
  - Ideally, must have a shape that results in a significant gradient in the right direction outside the optimum
    - To "guide" the algorithm to the right solution

Desiderata for a good divergence



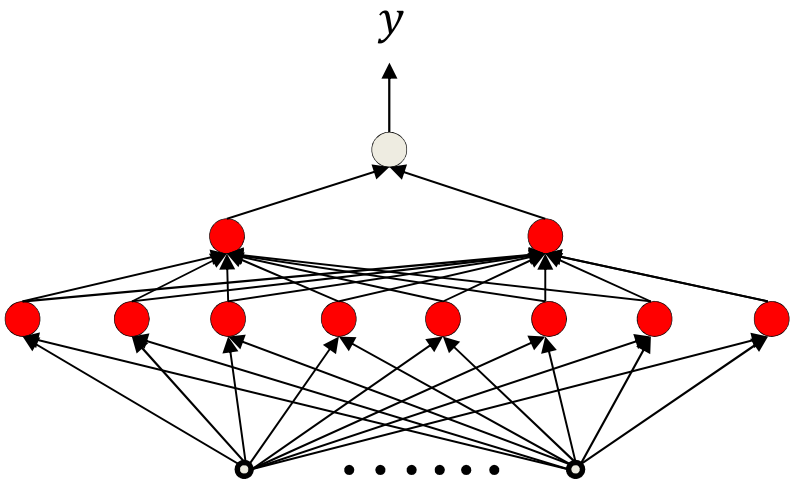
- Must be smooth and not have many poor local optima
- Low slopes far from the optimum == bad
  - Initial estimates far from the optimum will take forever to converge
- High slopes near the optimum == bad
  - Steep gradients

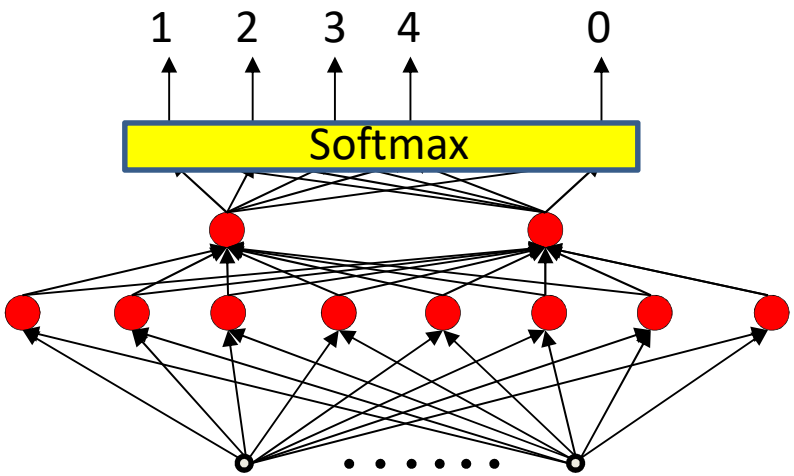
Desiderata for a good divergence



- 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





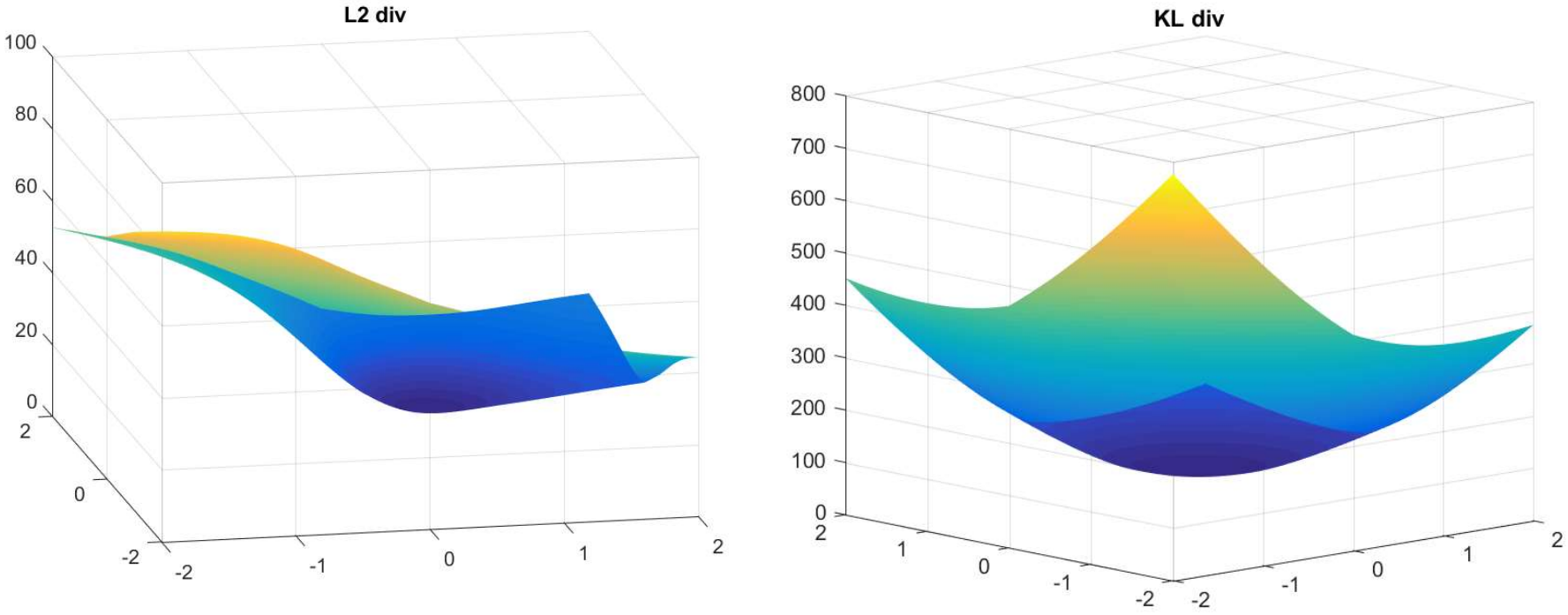
Desired output:	d	Desired output: $[0,0,\dots,1,\dots,0]$
L2	$Div = \frac{1}{2}(y-d)^2$	$Div = \frac{1}{2}\sum_i (y_i - d_i)^2$
KL	$Div = -d\log(y) - (1-d)\log(1-y)$	$Div = \sum_i d_i \log(d_i) - \sum_i d_i \log(y_i)$

- 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

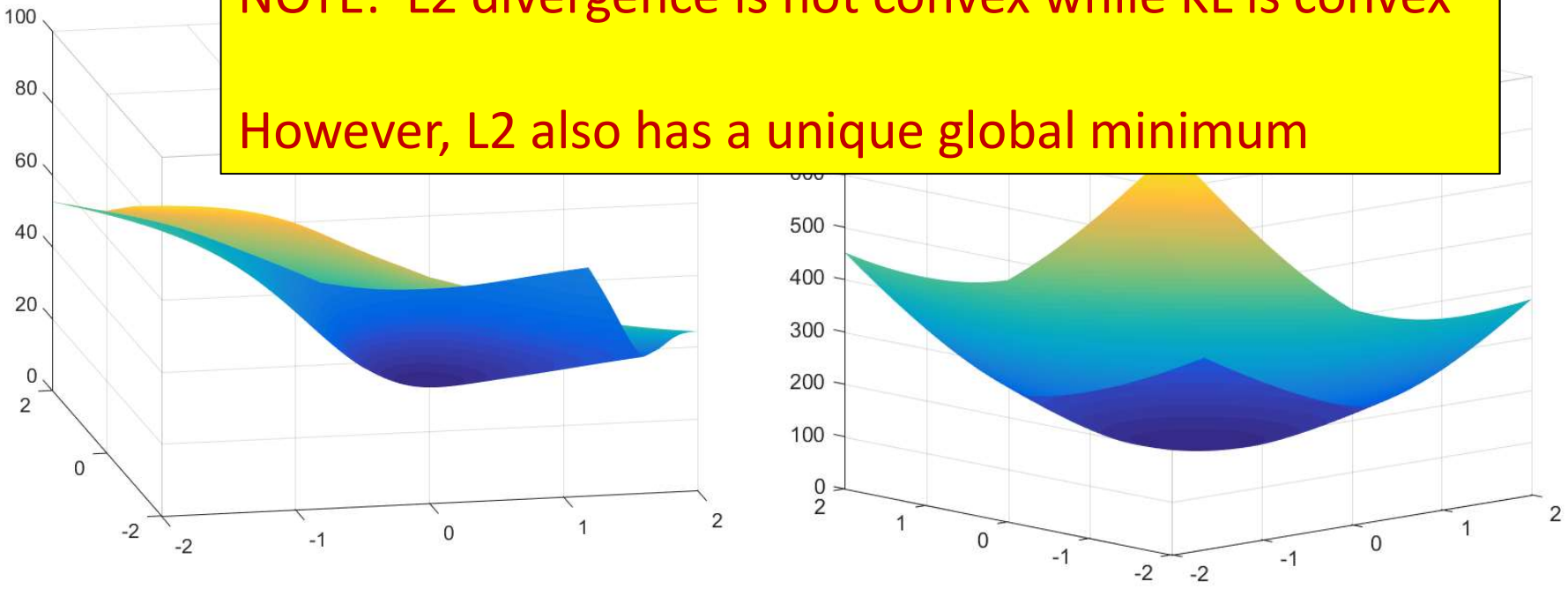


- Plot of L2 and KL divergences for a *single* perceptron, as function of weights
  - Setup: 2-dimensional input
  - 100 training examples randomly generated

L2 or KL

NOTE: L2 divergence is not convex while KL is convex

However, L2 also has a unique global minimum



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

A note on derivatives

- Note: For L2 divergence the derivative w.r.t. the pre-activation $\boldsymbol{z}$ of the output layer is:

$$\nabla_z \frac{1}{2}\|\boldsymbol{y}-\boldsymbol{d}\|^2 = (\boldsymbol{y}-\boldsymbol{d})J_{\boldsymbol{y}}(\boldsymbol{z})$$

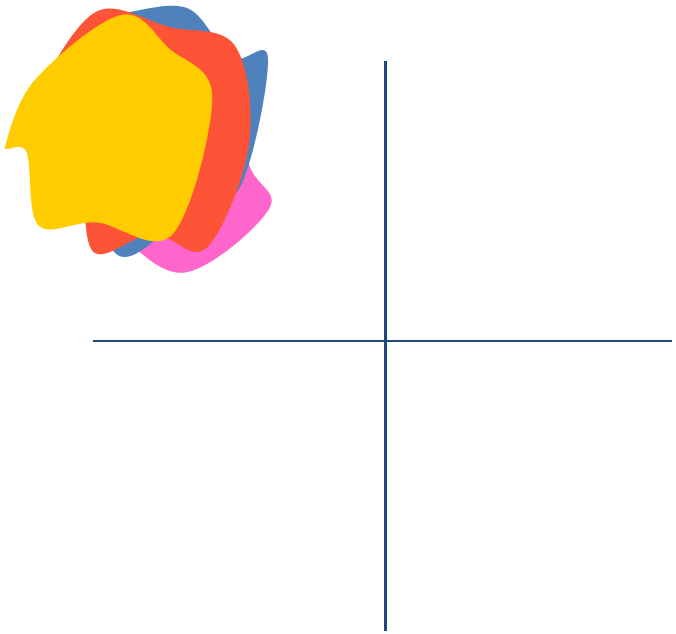
- We literally "propagate" the error $(\boldsymbol{y}-\boldsymbol{d})$ backward
  - Which is why the method is sometimes called "error backpropagation"

Story so far

- Gradient descent can be sped up by incremental updates
- Convergence can be improved using smoothed updates

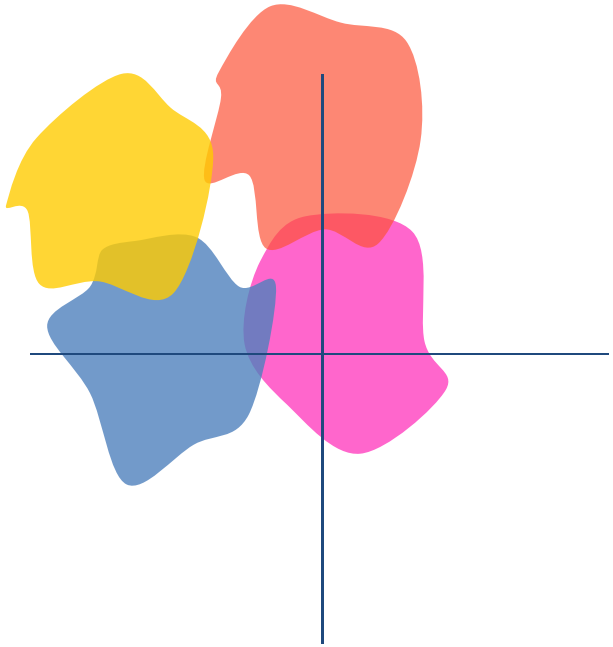
- The choice of divergence affects both the learned network and results

The problem of covariate shifts



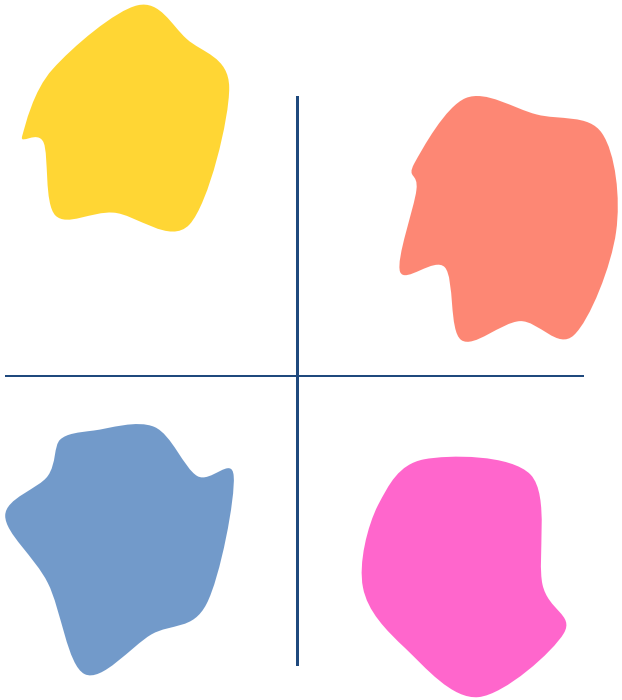
- Training assumes the training data are all similarly distributed
  - Minibatches have similar distribution

The problem of covariate shifts



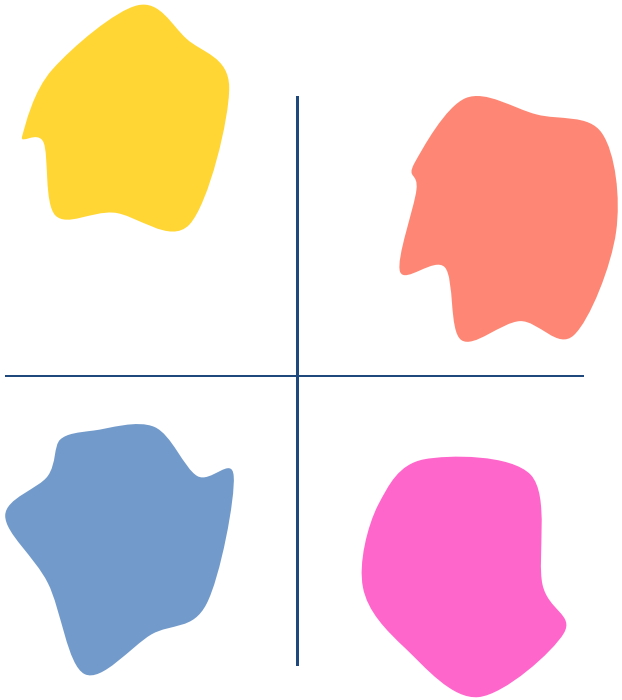
- Training assumes the training data are all similarly distributed
  - Minibatches have similar distribution
- In practice, each minibatch may have a different distribution
  - A “covariate shift”
  - Which may occur in *each* layer of the network

The problem of covariate shifts



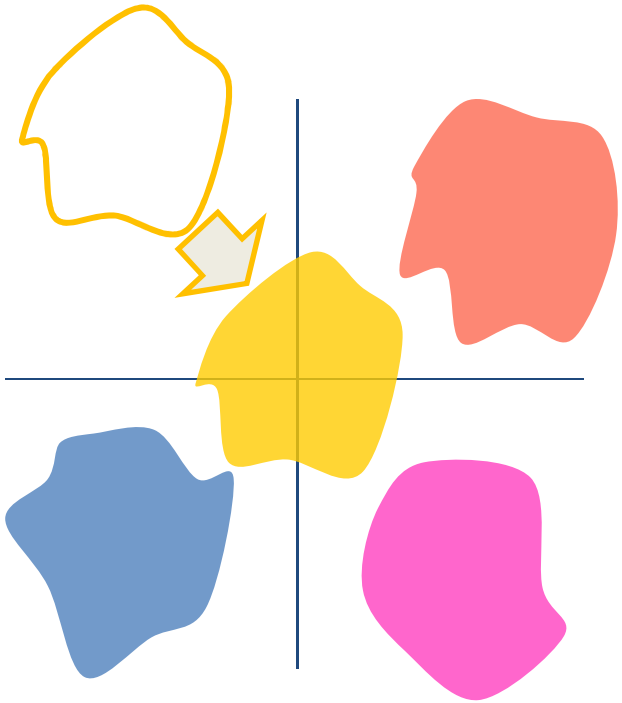
- Training assumes the training data are all similarly distributed
  - Minibatches have similar distribution
- In practice, each minibatch may have a different distribution
  - A “covariate shift”
- Covariate shifts can be large!
  - All covariate shifts can affect training badly

Solution: Move all subgroups to a "standard" location



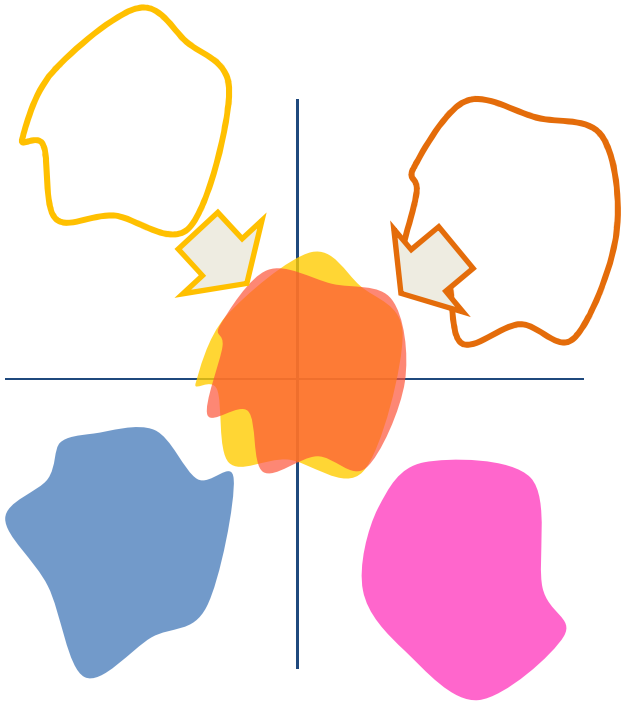
- "Move" all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches

Solution: Move all subgroups to a "standard" location



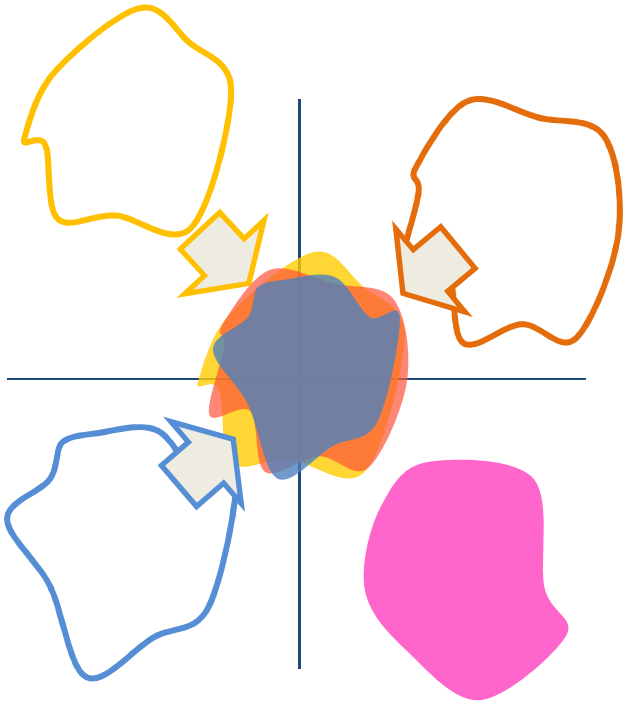
- "Move" all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches

Solution: Move all subgroups to a "standard" location



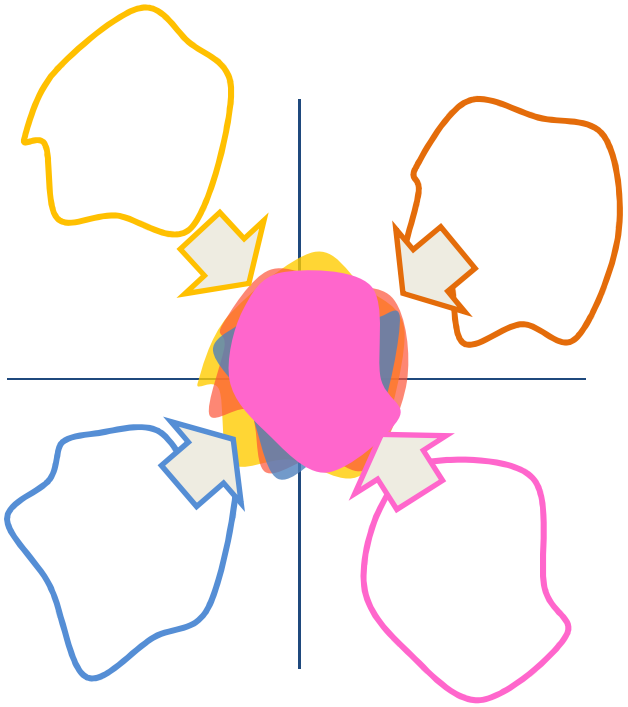
- "Move" all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches

Solution: Move all subgroups to a "standard" location



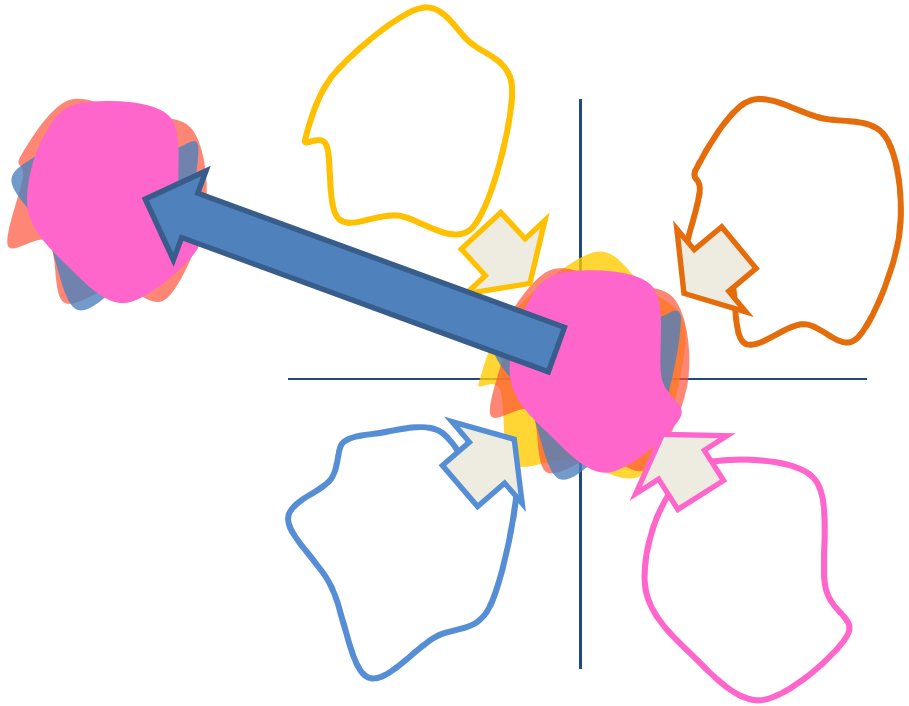
- "Move" all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches

Solution: Move all subgroups to a "standard" location



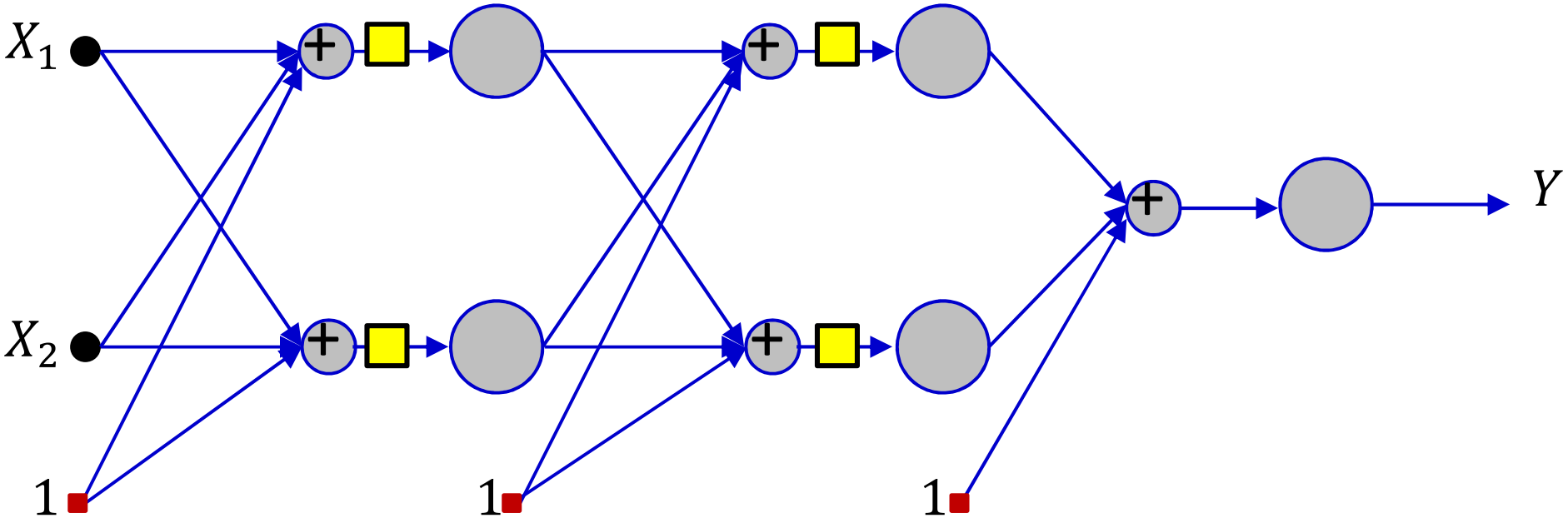
- "Move" all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches

Solution: Move all subgroups to a "standard" location



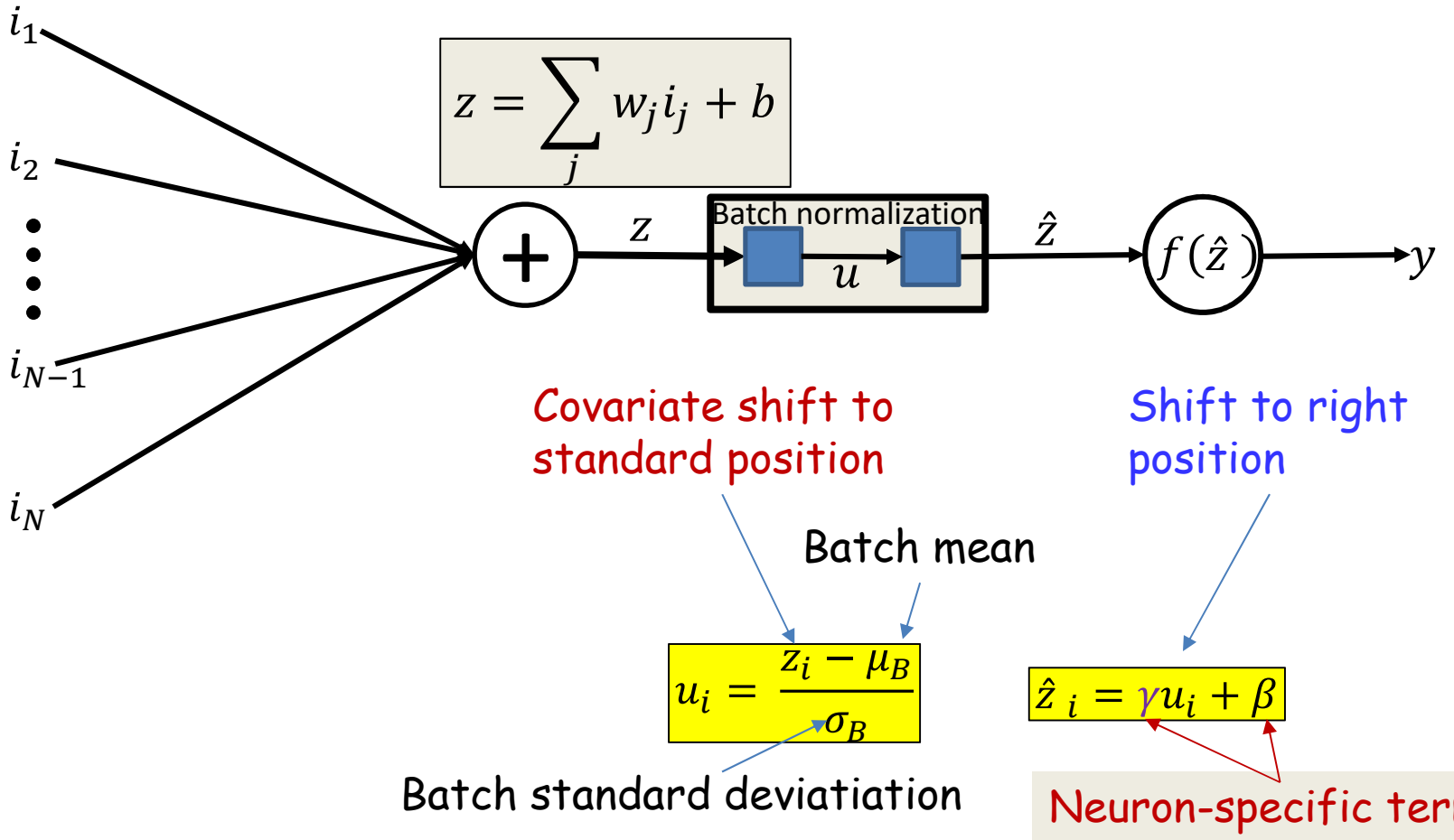
- "Move" all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches
  - Then move the entire collection to the appropriate location

Batch normalization



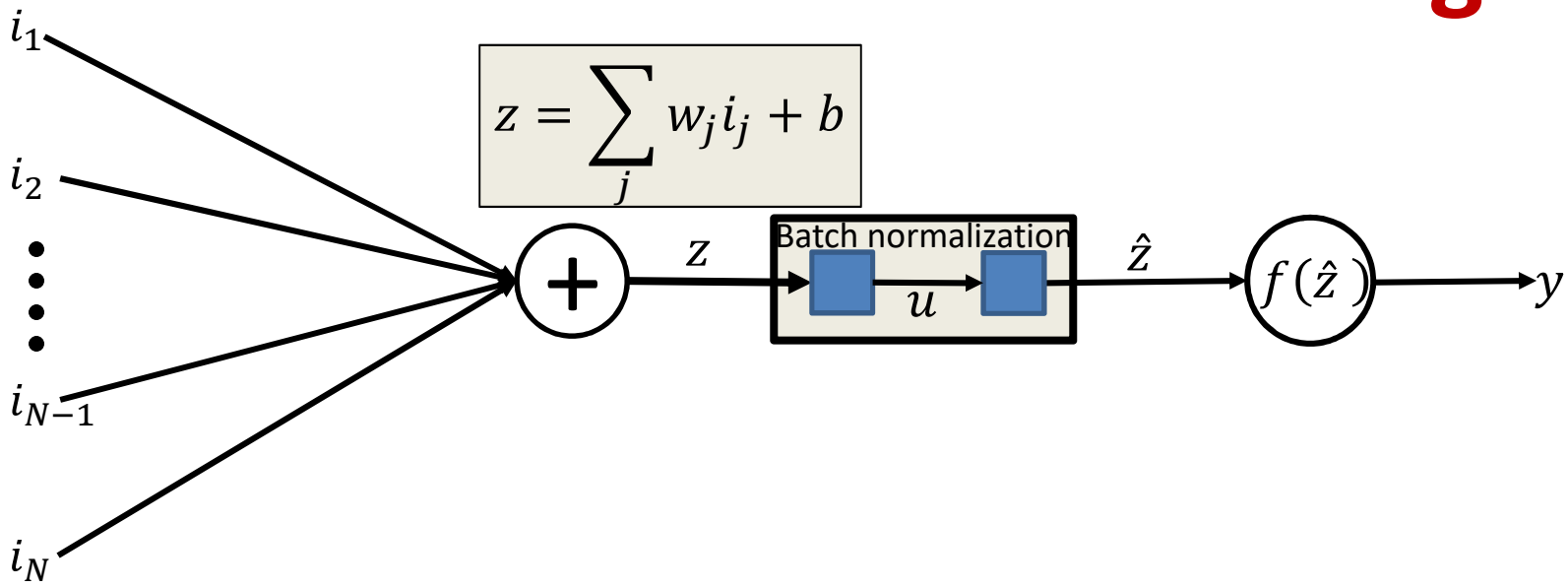
- Batch normalization is a covariate adjustment unit that happens after the weighted addition of inputs but before the application of activation
  - Is done independently for each unit, to simplify computation
- **Training: The adjustment occurs over individual minibatches**

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



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

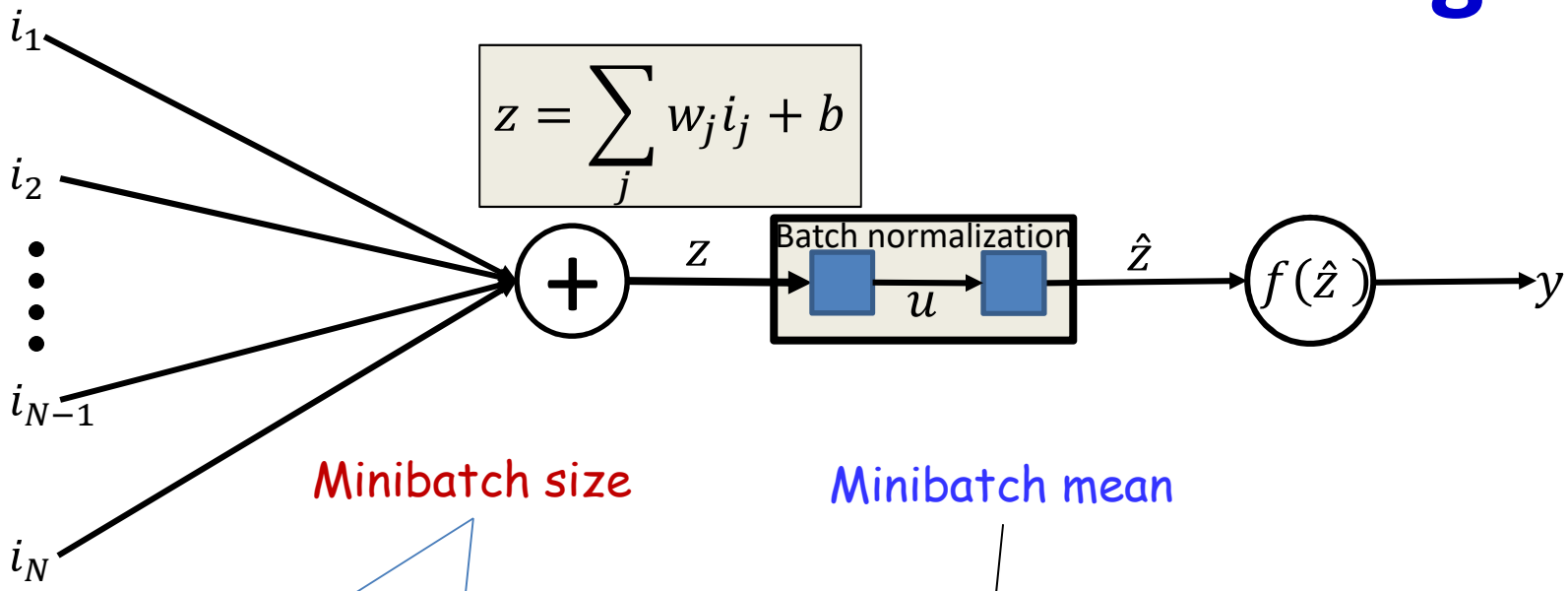
$$\sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B} (z_i - \mu_B)^2$$

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

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

- 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



Minibatch size

Minibatch mean

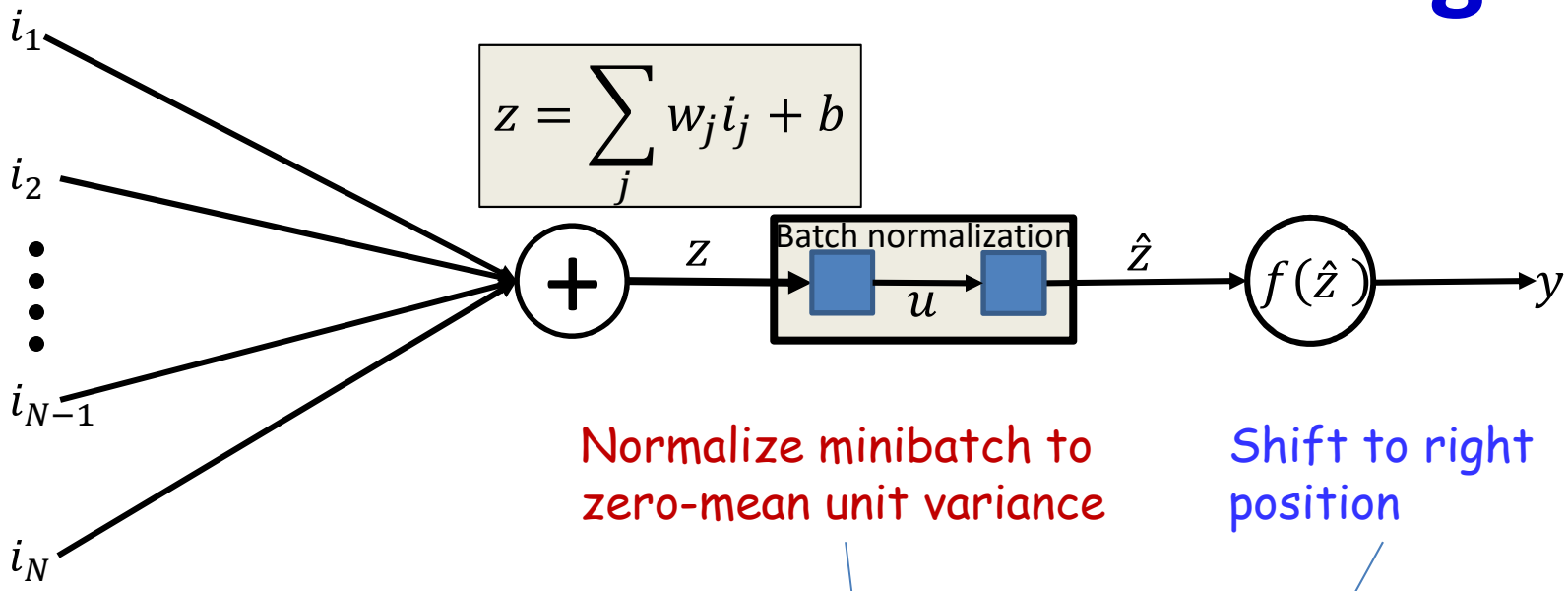
Minibatch standard deviation

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

$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \qquad \hat{z}_i = \gamma u_i + \beta$$

- 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



Normalize minibatch to zero-mean unit variance

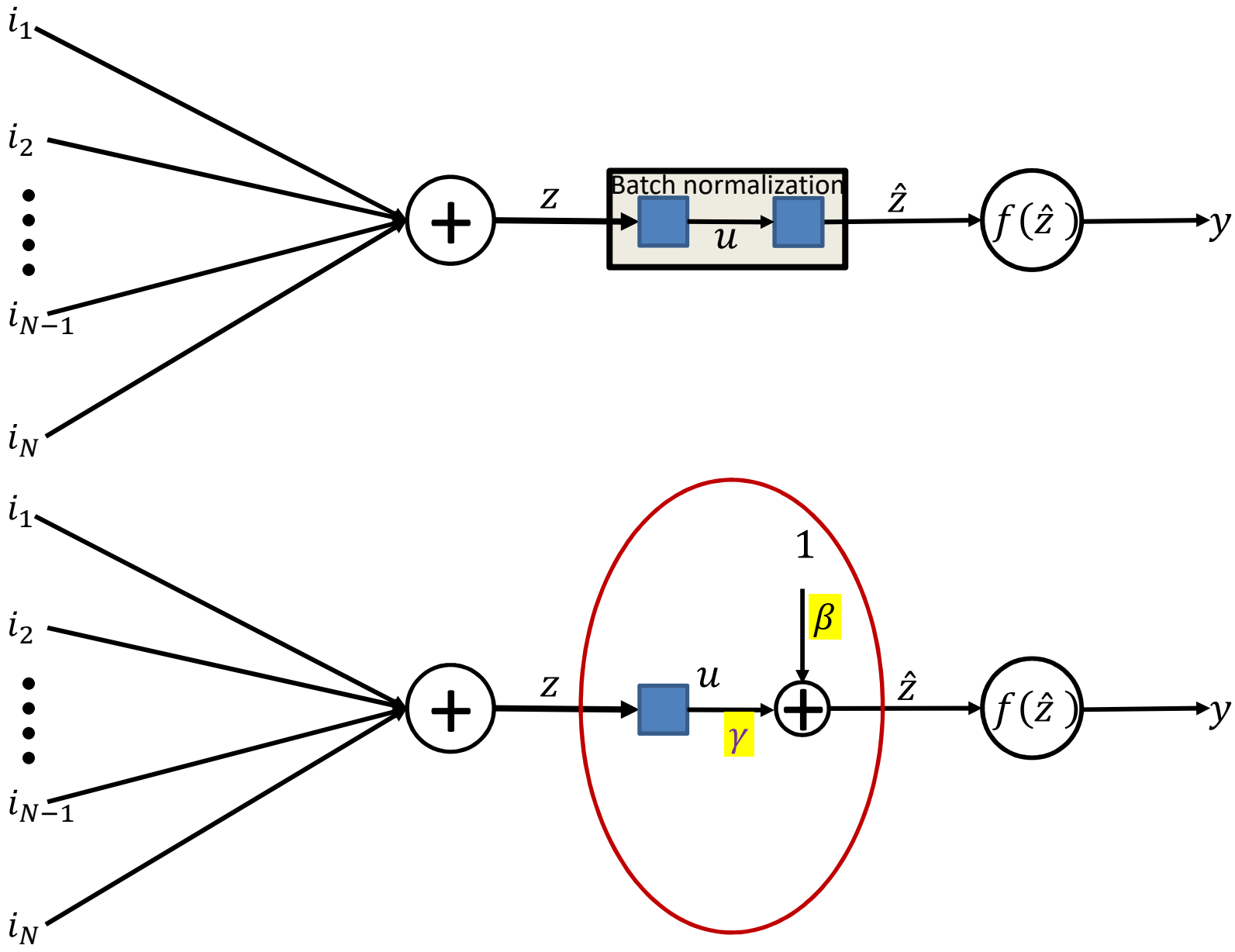
Shift to right position

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

$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \qquad \hat{z}_i = \gamma u_i + \beta$$

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

A better picture for batch norm



A note on derivatives

- In conventional learning, we attempt to compute the derivative of the divergence for *individual* training instances w.r.t. parameters

- This is based on the following relations

$$Div(minibatch) = \frac{1}{B}\sum_{t} Div(Y_t(X_t), d_t(X_t))$$

$$\frac{dDiv(minibatch)}{dw_{i,j}^{(k)}} = \frac{1}{T}\sum_{t} \frac{dDiv(Y_t(X_t), d_t(X_t))}{dw_{i,j}^{(k)}}$$

- If we use Batch Norm, the above relation gets a little complicated

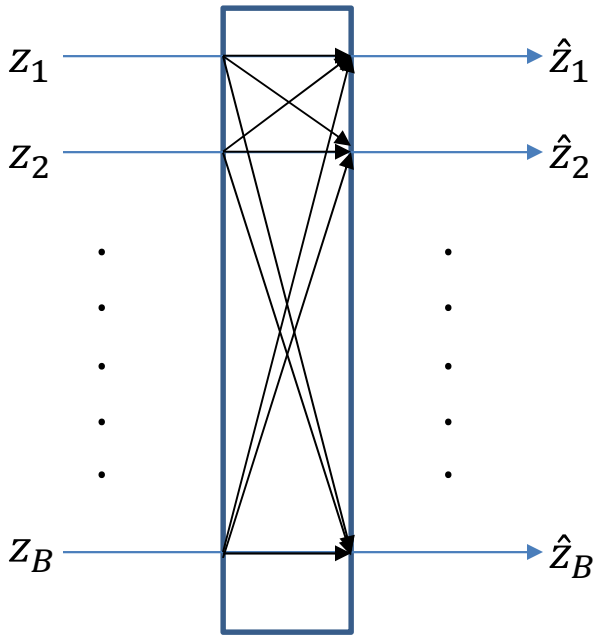
A note on derivatives

- The outputs are now functions of μ_B and σ_B^2 which are functions of the entire minibatch

$$Div(MB) = \frac{1}{B}\sum_t Div(Y_t(X_t, \mu_B, \sigma_B^2), d_t(X_t))$$

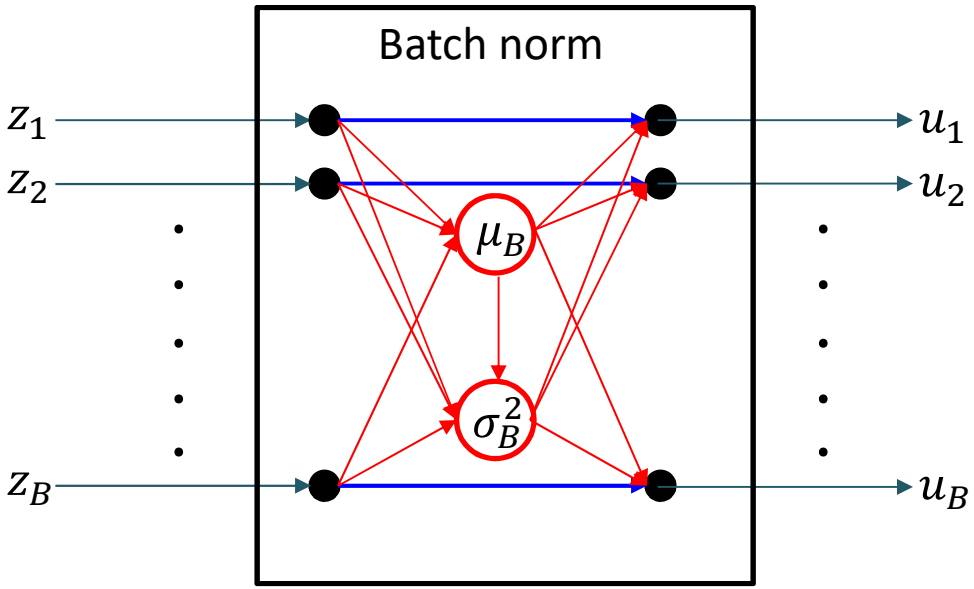
- The Divergence for each Y_t depends on *all* the X_t within the minibatch
- Specifically, within each layer, we get the relationship in the following slide

Batchnorm is a vector function over the minibatch



- Batch normalization is really a *vector* function applied over all the inputs from a minibatch
  - Every z_i affects every $\hat{z}_j$
  - Shown on the next slide
- To compute the derivative of the divergence w.r.t any z_i, we must consider all $\hat{z}_j s$ in the batch

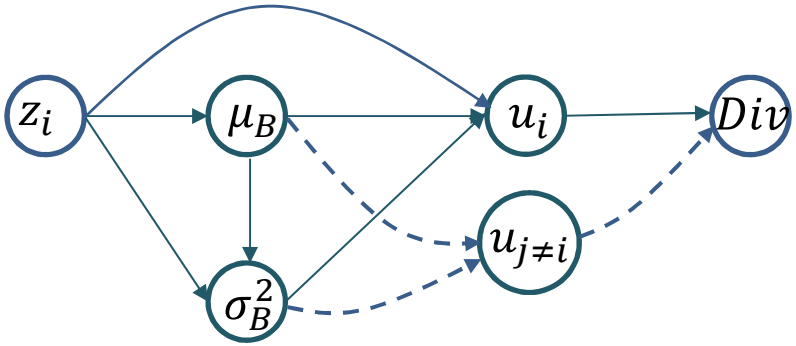
Batchnorm



- The complete dependency figure for Batchnorm
- Note : inputs and outputs are different *instances* in a minibatch
  - The diagram represents BN occurring at a *single neuron*
- You can use vector function differentiation rules to compute the derivatives
  - But the equations in the following slides summarize them for you
  - The actual derivation uses the simplified diagram shown in the next slide, but you could do it directly off the figure above and arrive at the same answers

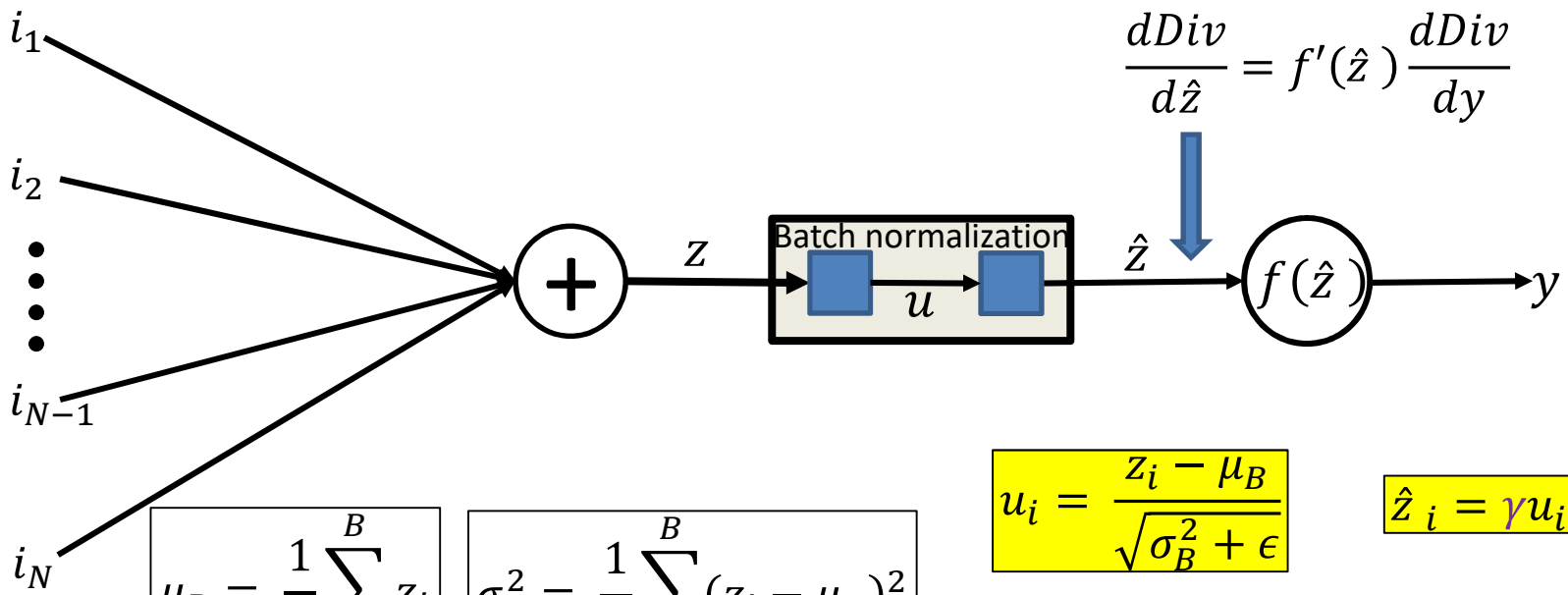
Batchnorm

Influence diagram



- Simplified diagram for a *single* input in a minibatch

Batch normalization: Backpropagation

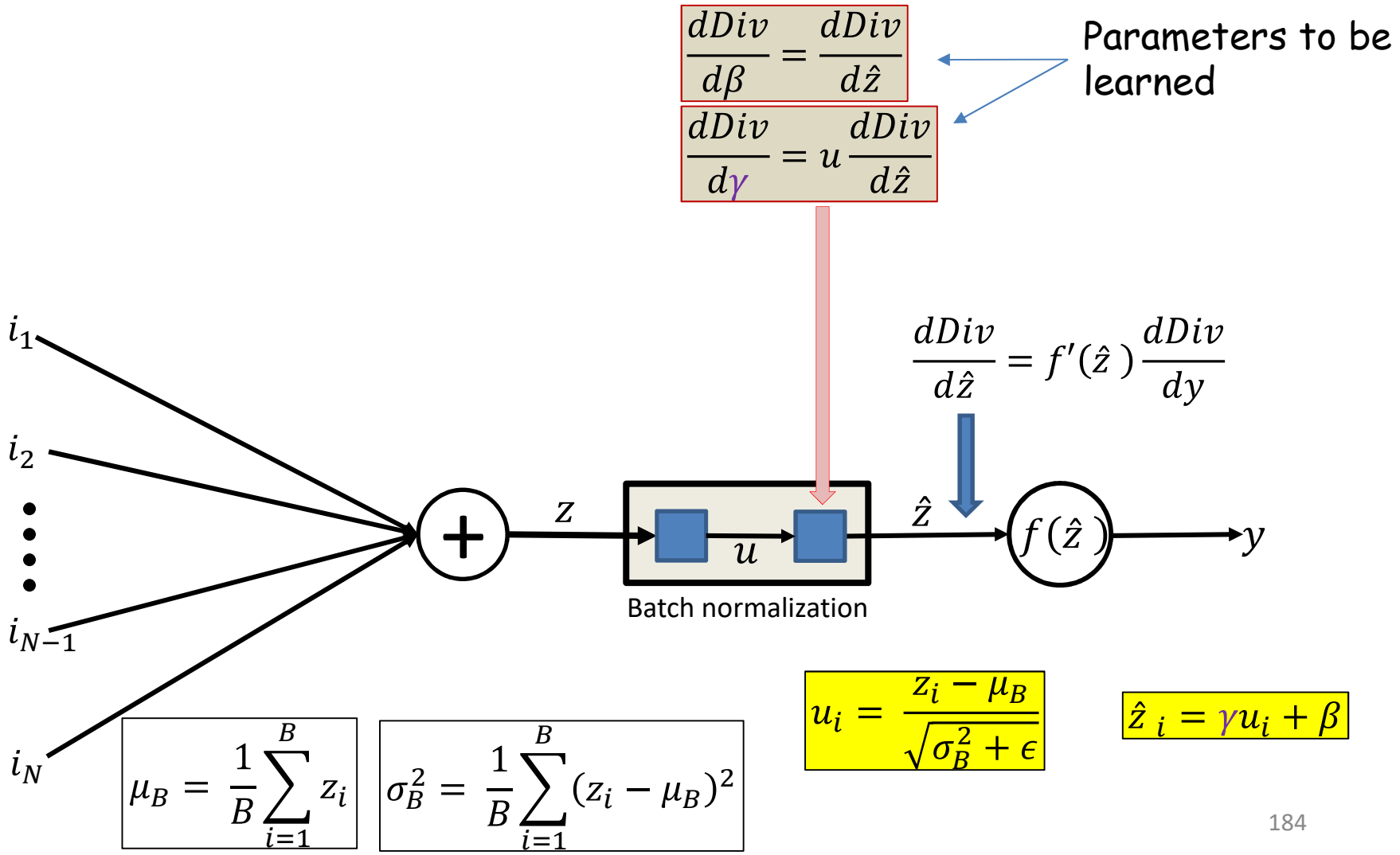


$$\mu_B = \frac{1}{B}\sum_{i=1}^{B} z_i \qquad \sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B}(z_i - \mu_B)^2$$

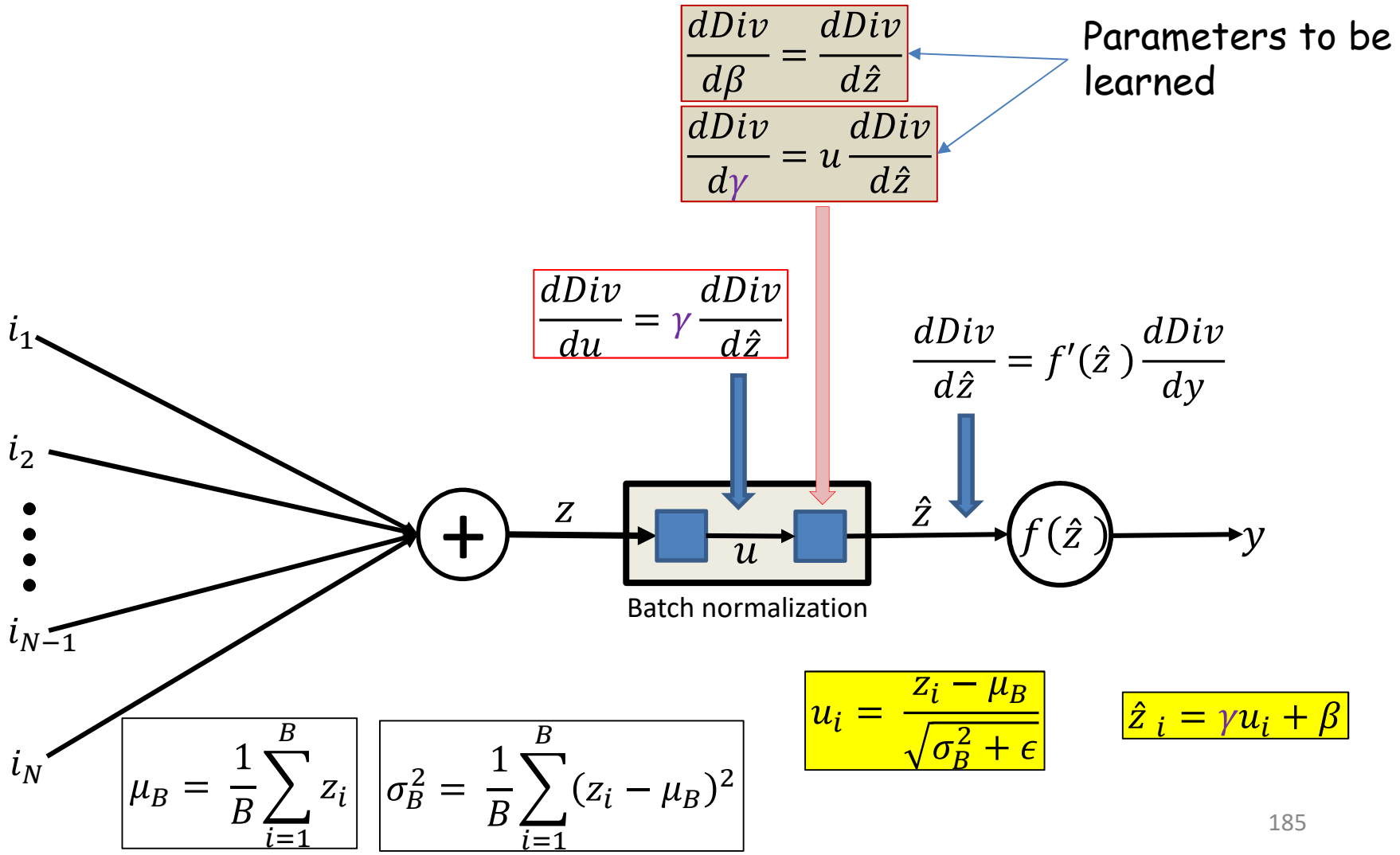
$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

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

Batch normalization: Backpropagation

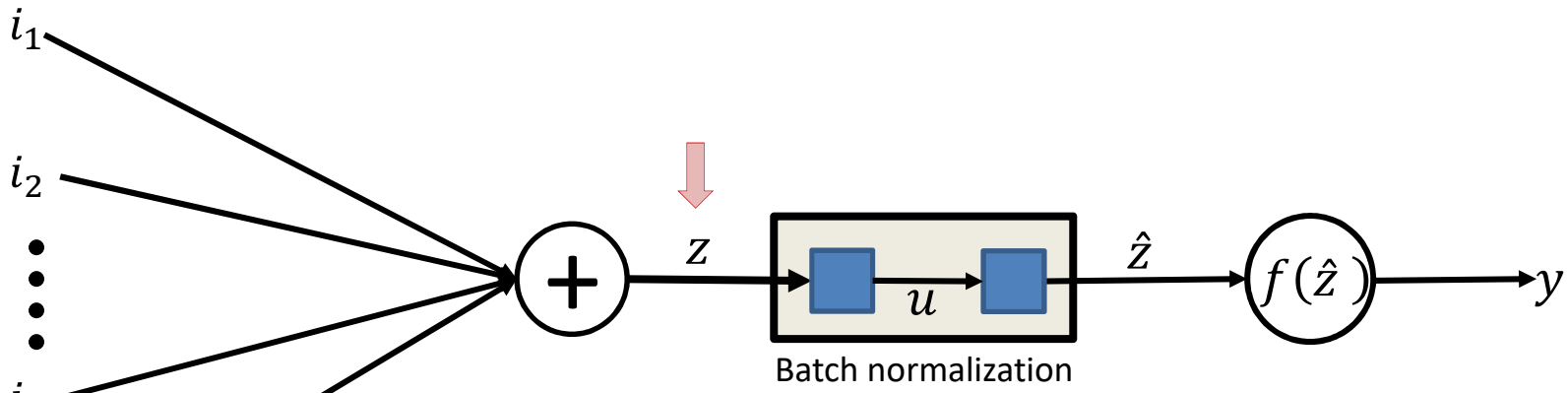


Batch normalization: Backpropagation



Batch normalization: Backpropagation

- Final step of backprop: compute $\frac{\partial Div}{\partial z_i}$



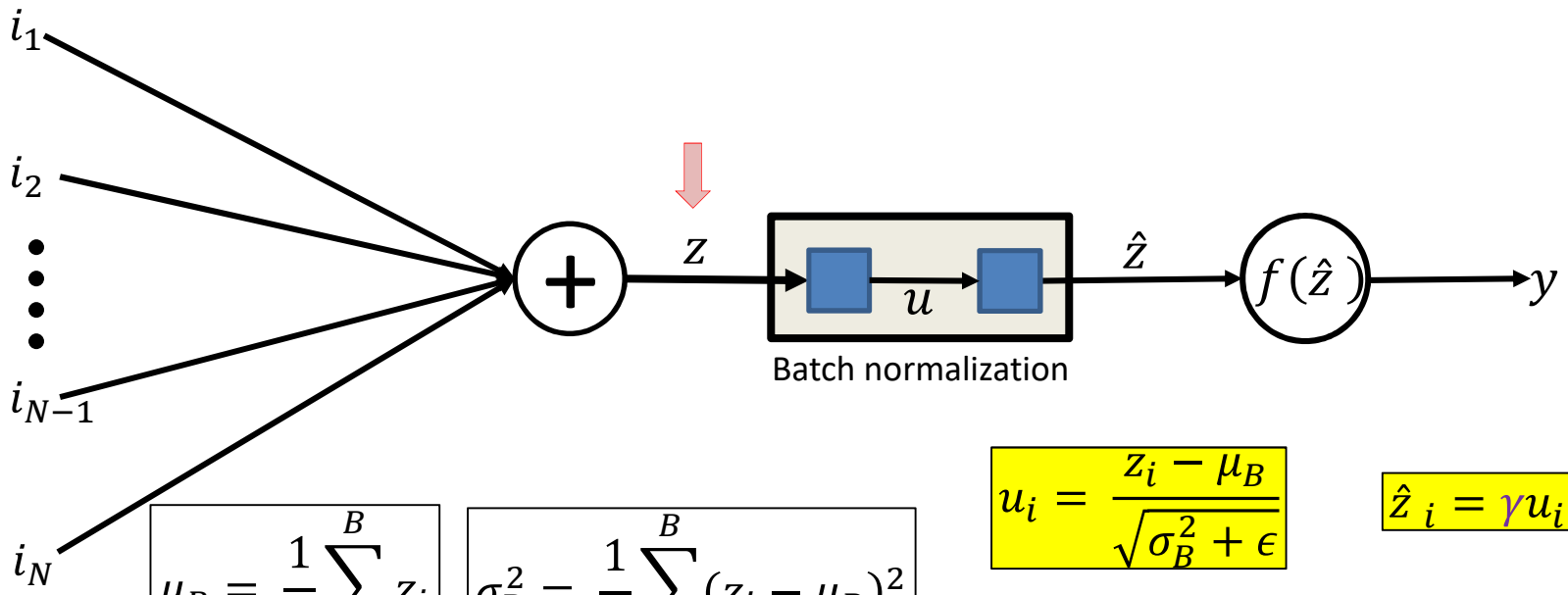
$$\mu_B = \frac{1}{B}\sum_{i=1}^{B} z_i \qquad \sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B}(z_i - \mu_B)^2$$

$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \qquad \hat{z}_i = \gamma u_i + \beta$$

Batch normalization: Backpropagation

$$Div = function(u_i, \mu_B, \sigma_B^2)$$

$$\frac{\partial Div}{\partial z_i} = \frac{\partial Div}{\partial u_i} \cdot \frac{\partial u_i}{\partial z_i} + \frac{\partial Div}{\partial \sigma_B^2} \cdot \frac{\partial \sigma_B^2}{\partial z_i} + \frac{\partial Div}{\partial \mu_B} \cdot \frac{\partial \mu_B}{\partial z_i}$$



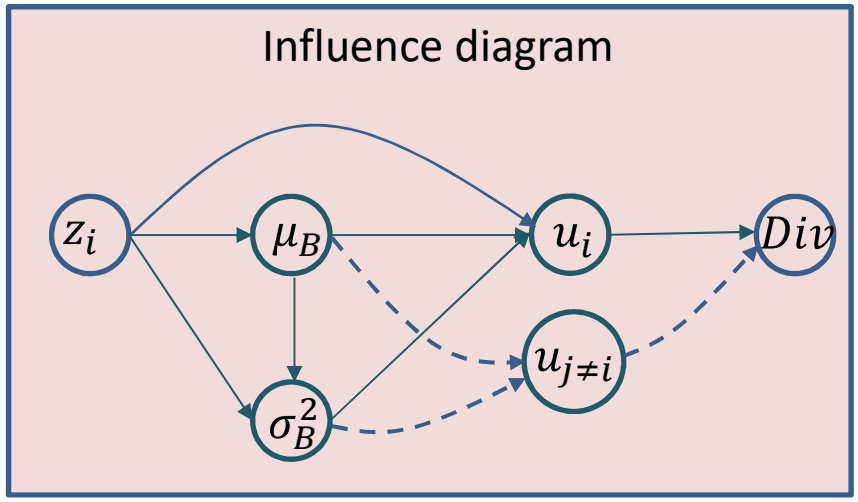
$$\mu_B = \frac{1}{B}\sum_{i=1}^{B} z_i$$

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

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

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

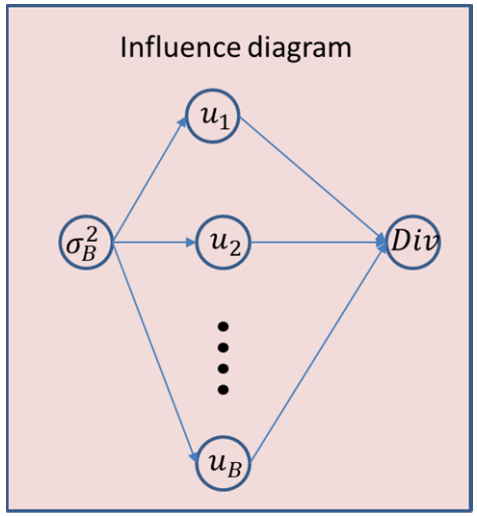
Batch normalization: Backpropagation

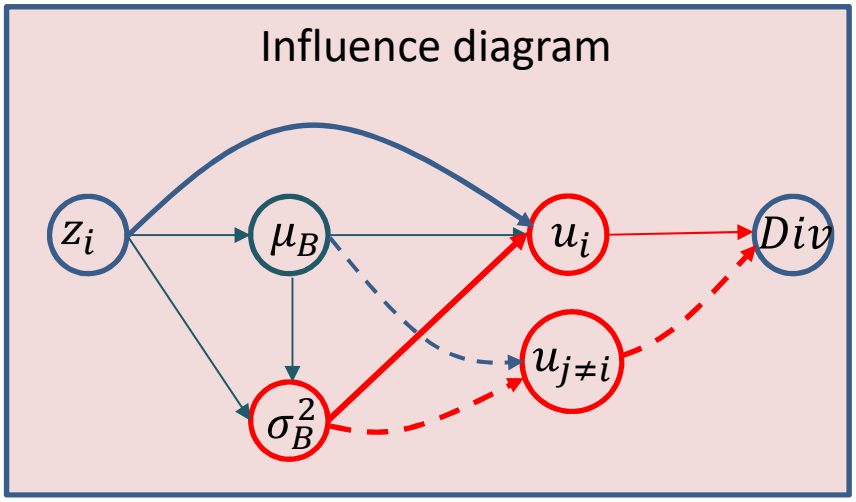


Dotted lines show dependence through other u_js because Divergence is computed over a minibatch

$$\frac{\partial Div}{\partial z_i} = \frac{\partial Div}{\partial u_i} \cdot \frac{\partial u_i}{\partial z_i} + \frac{\partial Div}{\partial \sigma_B^2} \cdot \frac{\partial \sigma_B^2}{\partial z_i} + \frac{\partial Div}{\partial \mu_B} \cdot \frac{\partial \mu_B}{\partial z_i}$$

Batch normalization: Backpropagation





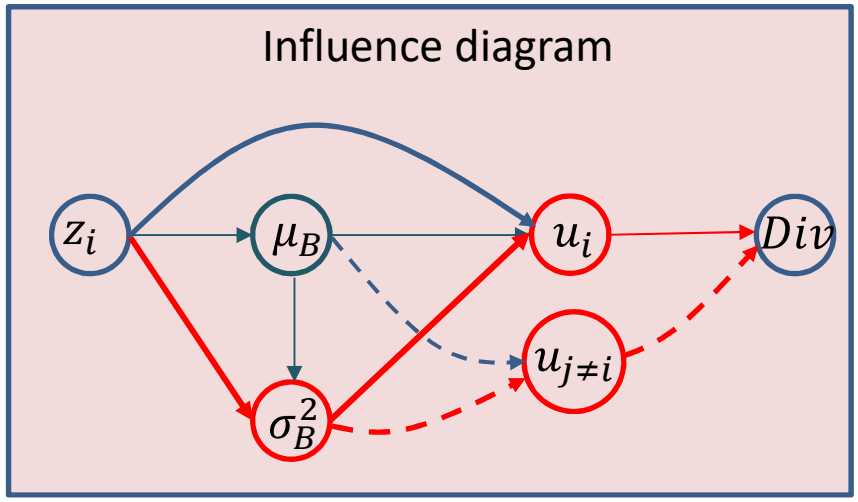
Dotted lines show dependence through other u_js because Divergence is computed over a minibatch

$$\frac{\partial Div}{\partial z_i} = \frac{\partial Div}{\partial u_i}\cdot\frac{\partial u_i}{\partial z_i} + \frac{\partial Div}{\partial \sigma_B^2}\cdot\frac{\partial \sigma_B^2}{\partial z_i} + \frac{\partial Div}{\partial \mu_B}\cdot\frac{\partial \mu_B}{\partial z_i}$$

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

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

Batch normalization: Backpropagation



Dotted lines show dependence through other u_js because Divergence is computed over a minibatch

$$\frac{\partial Div}{\partial z_i} = \frac{\partial Div}{\partial u_i}\cdot\frac{\partial u_i}{\partial z_i} + \frac{\partial Div}{\partial \sigma_B^2}\cdot\frac{\partial \sigma_B^2}{\partial z_i} + \frac{\partial Div}{\partial \mu_B}\cdot\frac{\partial \mu_B}{\partial z_i}$$

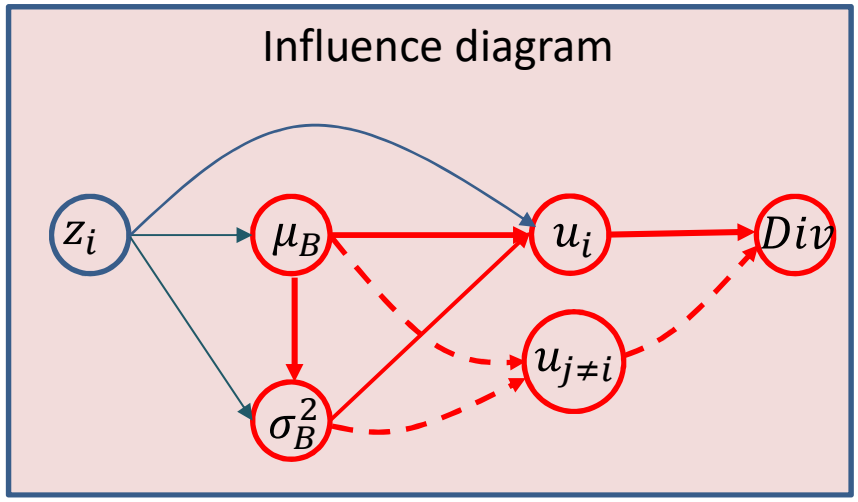
$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

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

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

$$\frac{\partial \sigma_B^2}{\partial z_i} = \frac{2(z_i - \mu_B)}{B}$$

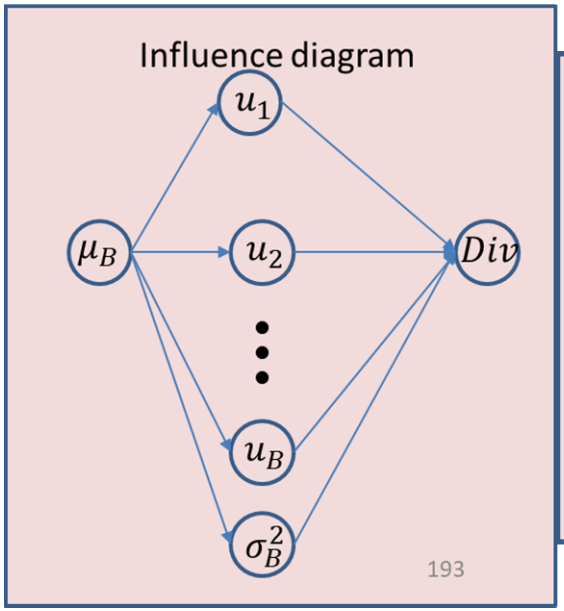
Batch normalization: Backpropagation

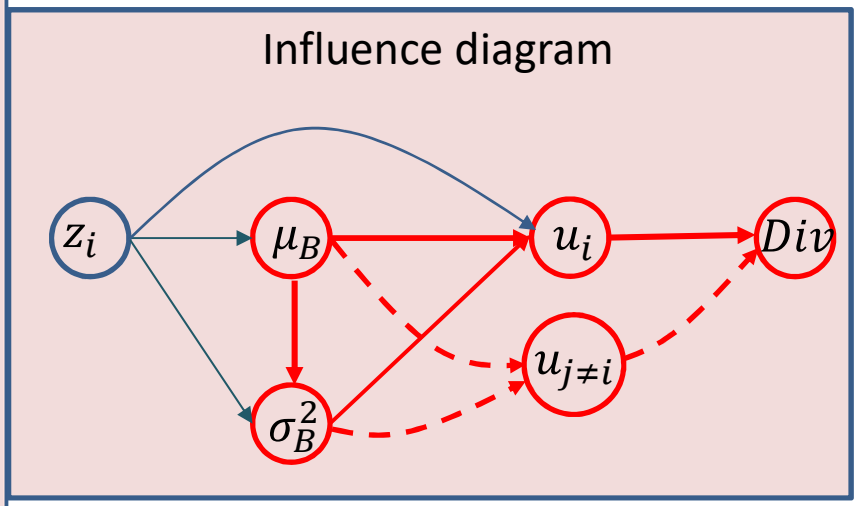


Dotted lines show dependence through other u_js because Divergence is computed over a minibatch

$$\frac{\partial Div}{\partial z_i} = \frac{\partial Div}{\partial u_i} \cdot \frac{\partial u_i}{\partial z_i} + \frac{\partial Div}{\partial \sigma_B^2} \cdot \frac{\partial \sigma_B^2}{\partial z_i} + \frac{\partial Div}{\partial \mu_B} \cdot \frac{\partial \mu_B}{\partial z_i}$$

Batch normalization: Backpropagation





Dotted lines show dependence through other u_js because Divergence is computed over a minibatch

$$\frac{\partial Div}{\partial z_i} = \frac{\partial Div}{\partial u_i}\cdot\frac{\partial u_i}{\partial z_i} + \frac{\partial Div}{\partial \sigma_B^2}\cdot\frac{\partial \sigma_B^2}{\partial z_i} + \frac{\partial Div}{\partial \mu_B}\cdot\frac{\partial \mu_B}{\partial z_i}$$

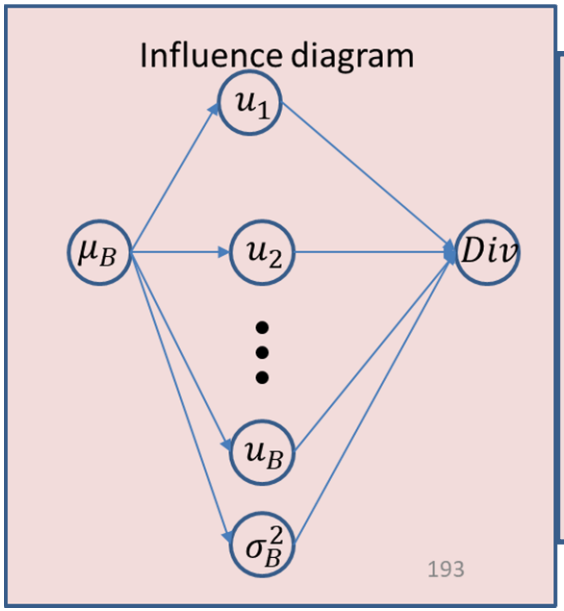
Second term goes to 0

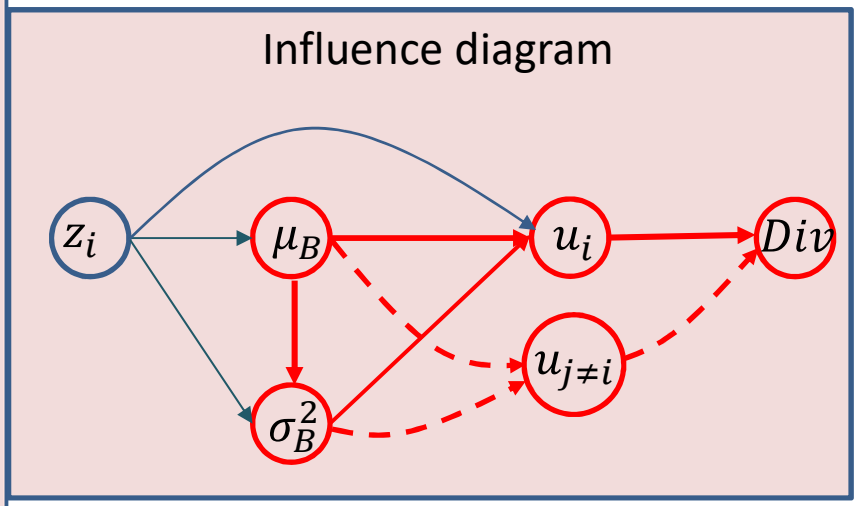
$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

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

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

Batch normalization: Backpropagation





Dotted lines show dependence through other u_js because Divergence is computed over a minibatch

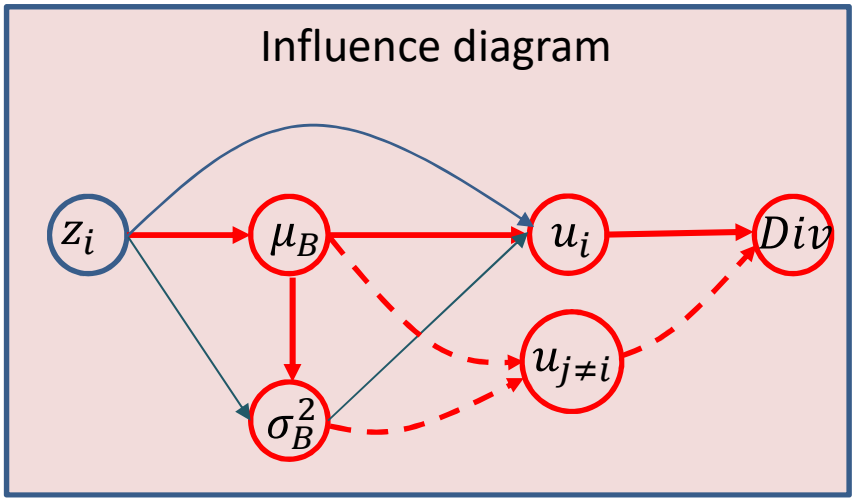
$$\frac{\partial Div}{\partial z_i} = \frac{\partial Div}{\partial u_i}\cdot\frac{\partial u_i}{\partial z_i} + \frac{\partial Div}{\partial \sigma_B^2}\cdot\frac{\partial \sigma_B^2}{\partial z_i} + \frac{\partial Div}{\partial \mu_B}\cdot\frac{\partial \mu_B}{\partial z_i}$$

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

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

$$\frac{\partial Div}{\partial \mu_B} = \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}}\sum_{i=1}^{B}\frac{\partial Div}{\partial u_i}$$

Batch normalization: Backpropagation



Dotted lines show dependence through other u_js because Divergence is computed over a minibatch

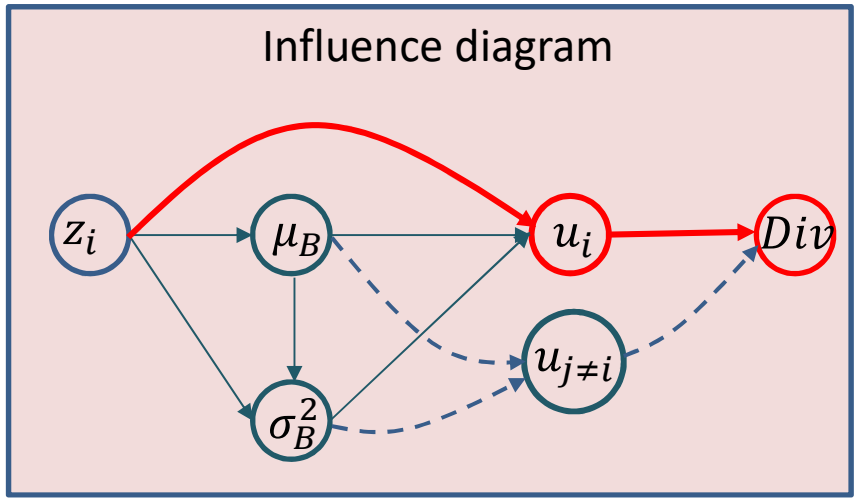
$$\frac{\partial Div}{\partial z_i} = \frac{\partial Div}{\partial u_i}\cdot\frac{\partial u_i}{\partial z_i} + \frac{\partial Div}{\partial \sigma_B^2}\cdot\frac{\partial \sigma_B^2}{\partial z_i} + \frac{\partial Div}{\partial \mu_B}\cdot\frac{\partial \mu_B}{\partial z_i}$$

$$\frac{\partial \mu_B}{\partial z_i} = \frac{1}{B}$$

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

$$\frac{\partial Div}{\partial \mu_B} = \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}}\sum_{i=1}^{B}\frac{\partial Div}{\partial u_i}$$

Batch normalization: Backpropagation



Dotted lines show dependence through other u_js because Divergence is computed over a minibatch

$$\frac{\partial Div}{\partial z_i} = \frac{\partial Div}{\partial u_i}\cdot\frac{\partial u_i}{\partial z_i} + \frac{\partial Div}{\partial \sigma_B^2}\cdot\frac{\partial \sigma_B^2}{\partial z_i} + \frac{\partial Div}{\partial \mu_B}\cdot\frac{\partial \mu_B}{\partial z_i}$$

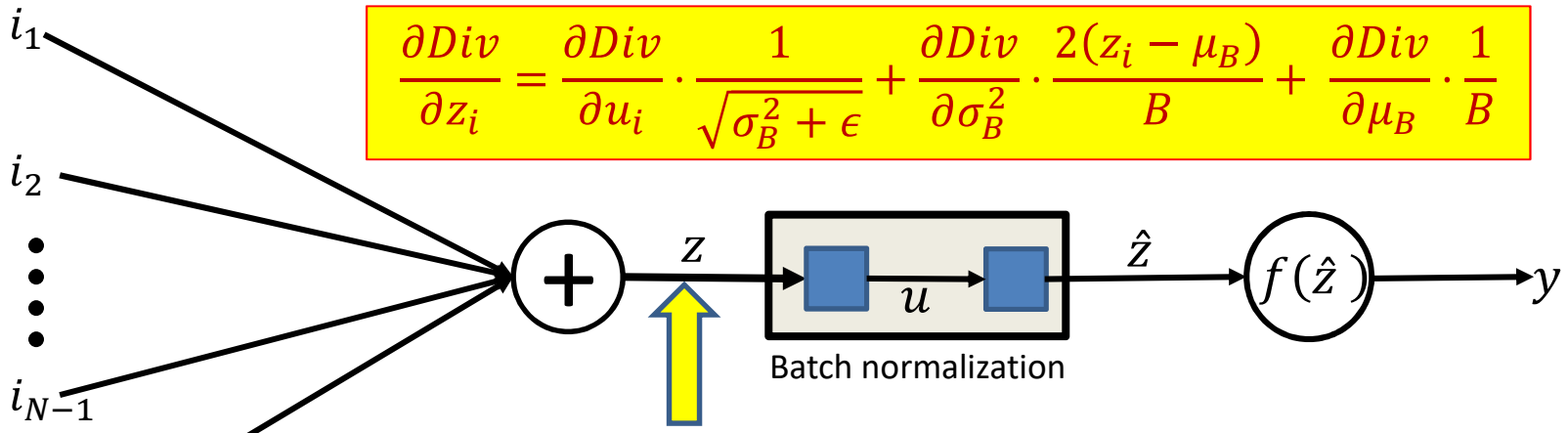
$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\frac{\partial Div}{\partial u_i}\cdot\frac{1}{\sqrt{\sigma_B^2 + \epsilon}}$$

Batch normalization: Backpropagation

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

$$\frac{\partial Div}{\partial \mu_B} = \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \sum_{i=1}^{B} \frac{\partial Div}{\partial u_i}$$



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

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

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

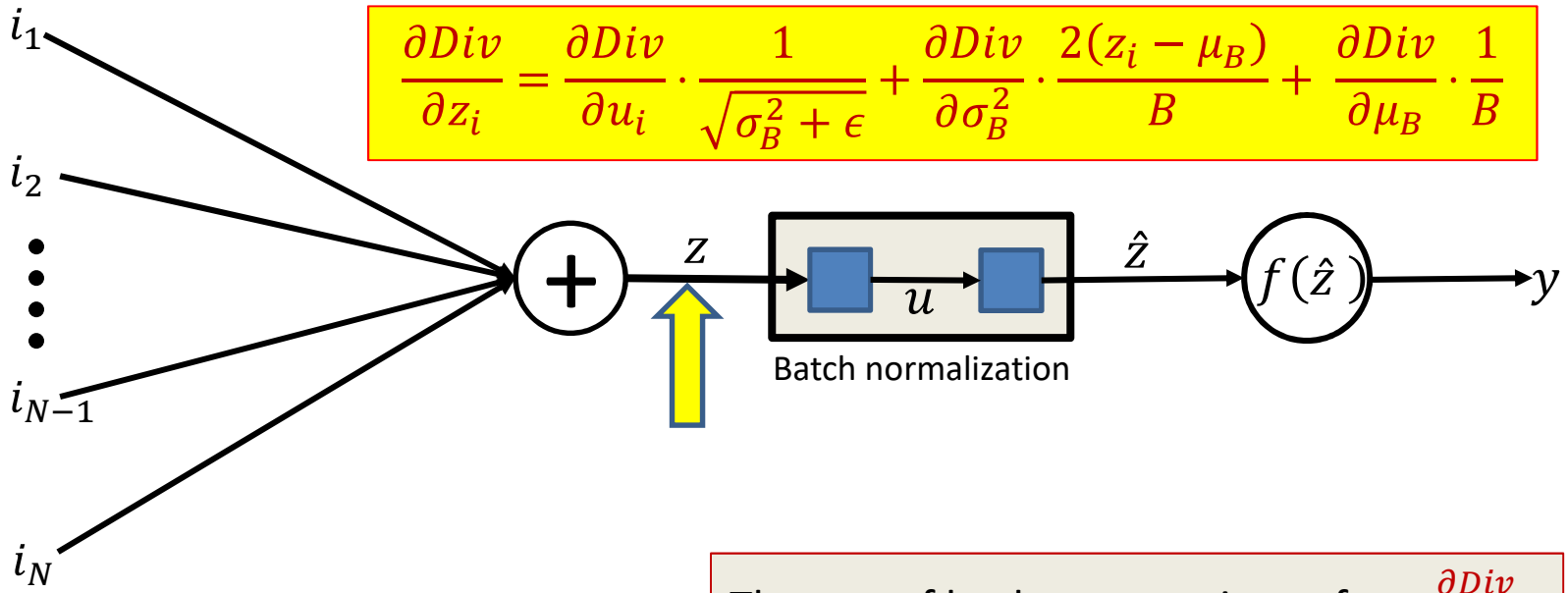
$$\hat{z}_i = \gamma u_i + \beta$$

Batch normalization: Backpropagation

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

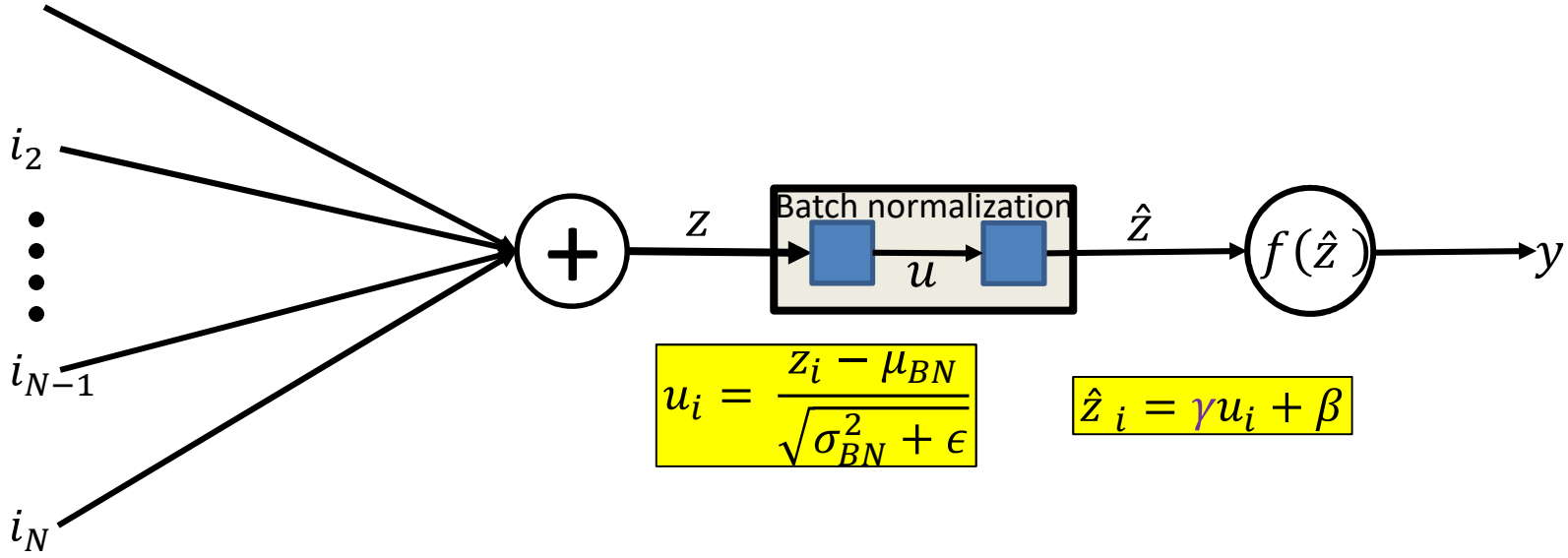
$$\frac{\partial Div}{\partial \mu_B} = \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \sum_{i=1}^{B} \frac{\partial Div}{\partial u_i}$$

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



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

Batch normalization: Inference



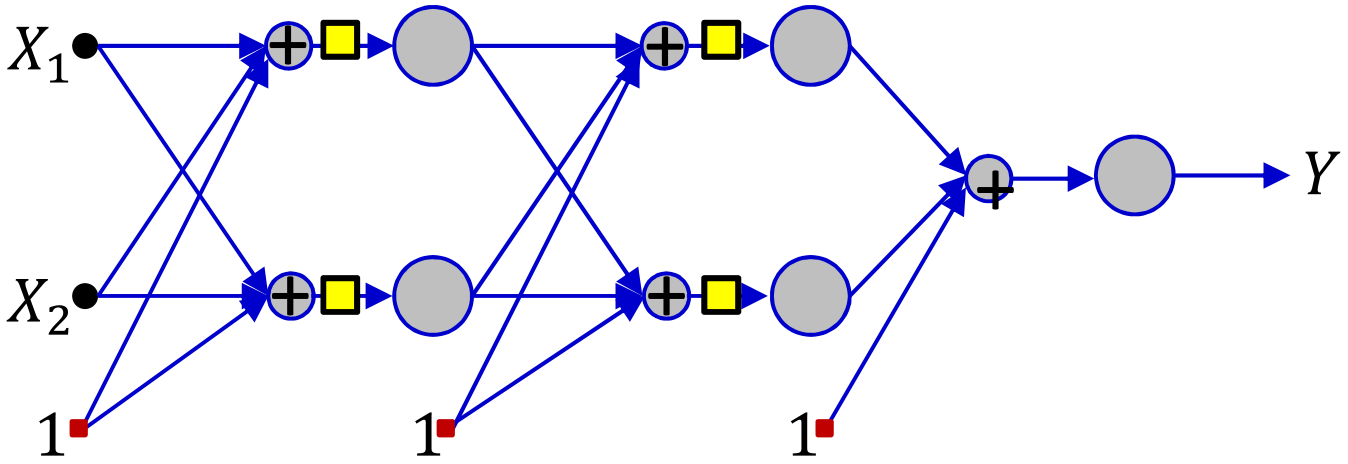
- On test data, BN requires μ_B and σ_B^2.
- We will use the *average over all training minibatches*

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

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

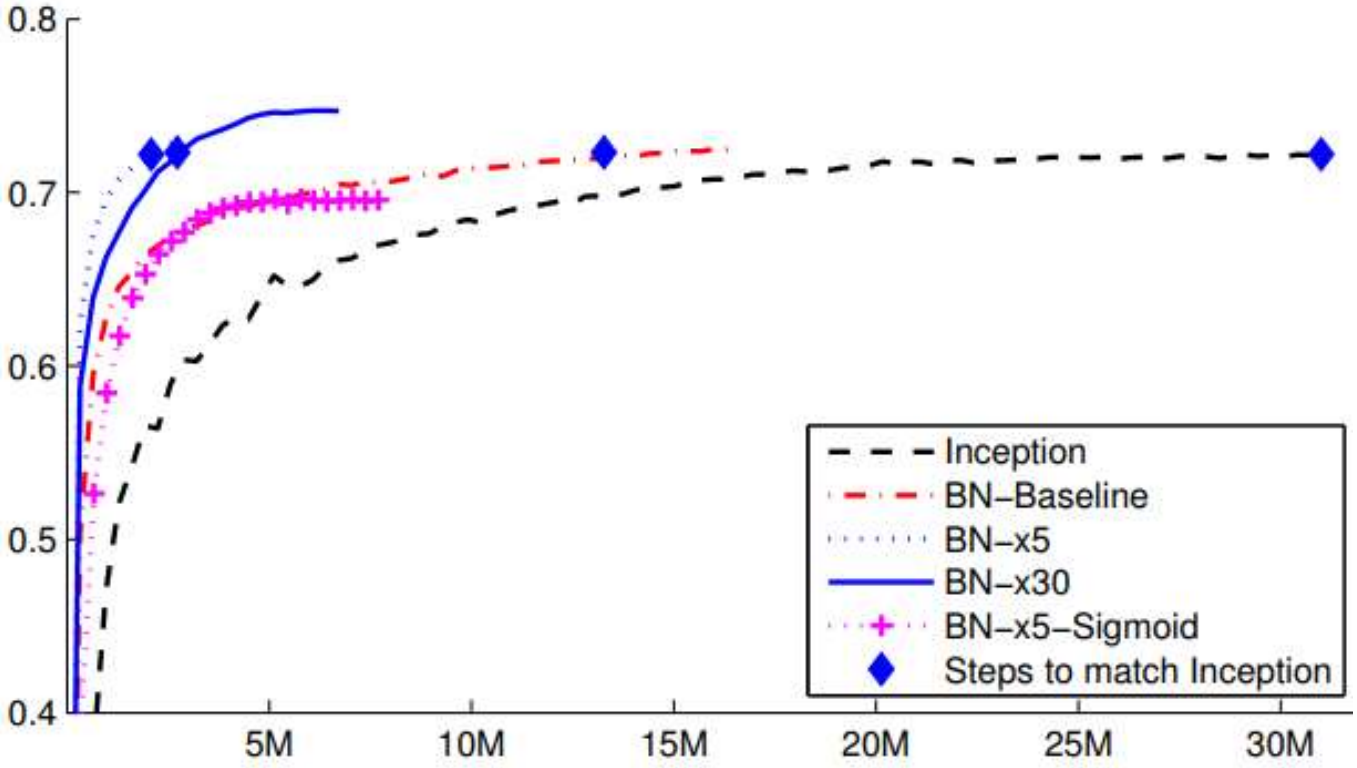
- Note: these are *neuron-specific*
  - $\mu_B(batch)$ and $\sigma_B^2(batch)$ here are obtained from the *final converged network*
  - The $B/(B-1)$ term gives us an unbiased estimator for the variance

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

Story so far

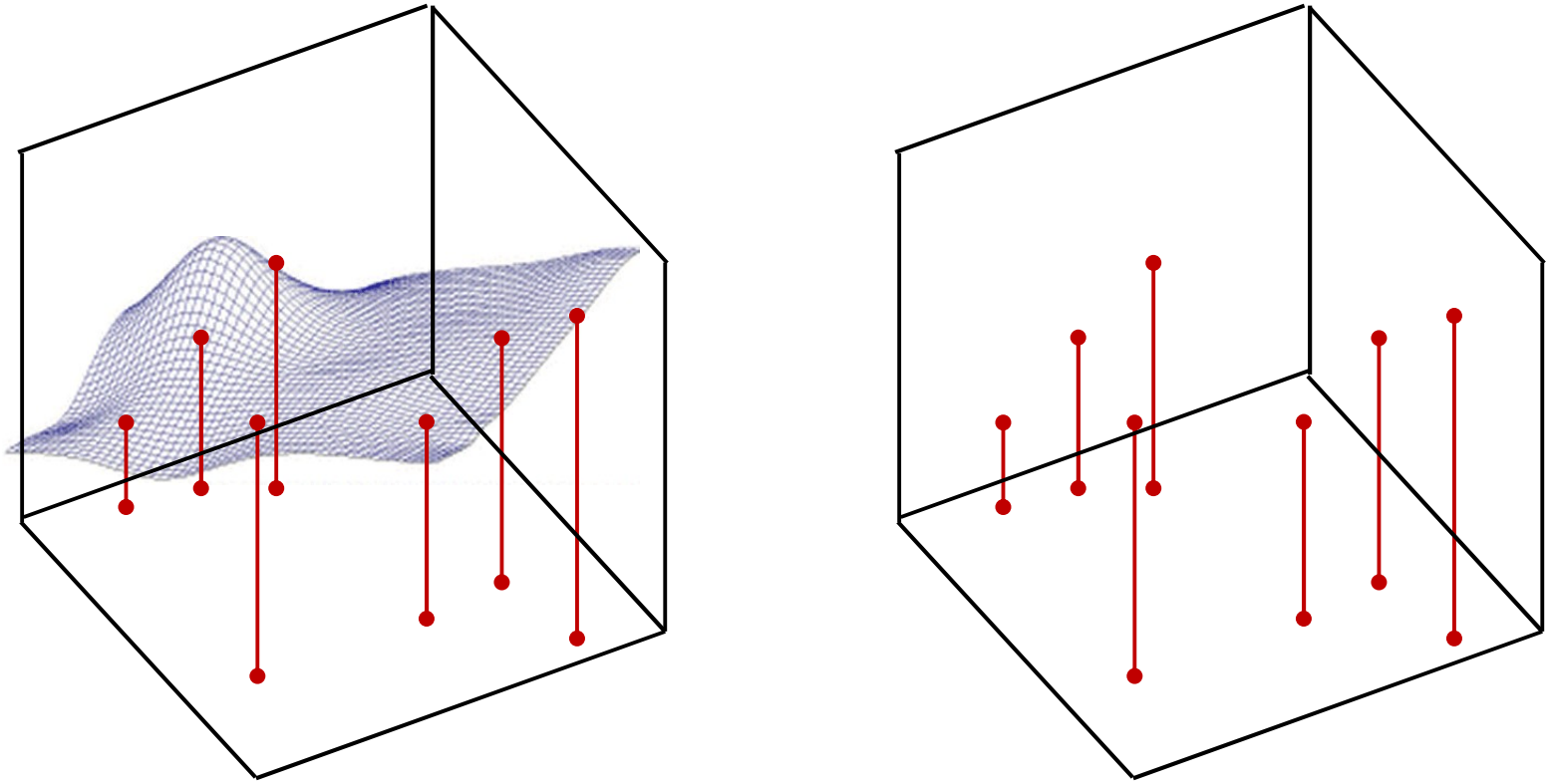
- Gradient descent can be sped up by incremental updates
- Convergence can be improved using smoothed updates

- The choice of divergence affects both the learned network and results
- Covariate shift between training and test may cause problems and may be handled by batch normalization

The problem of data underspecification

- The figures shown to illustrate the learning problem so far were *fake news*..

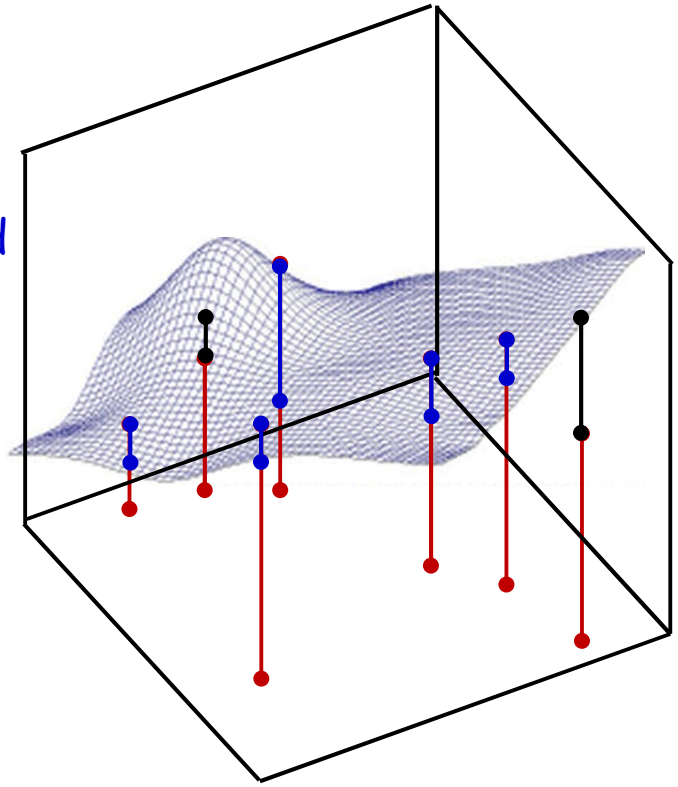
Learning the network



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

General approach to training

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

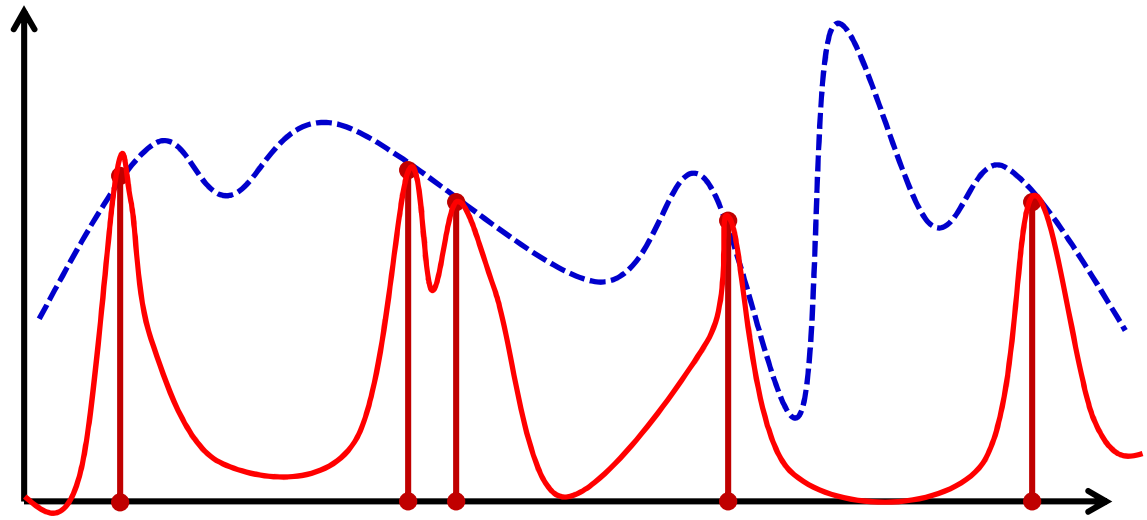


Black lines: error when function is *above* desired output

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

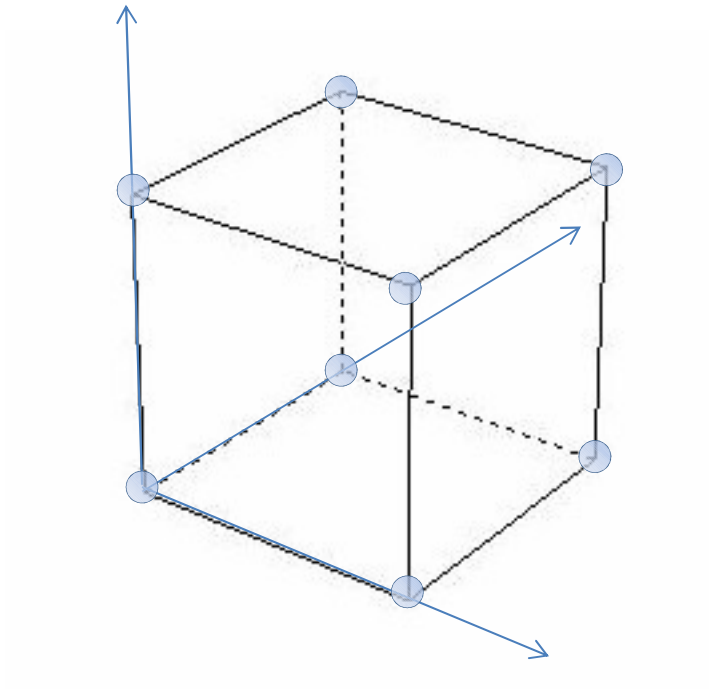
- Define an *error* between the ***actual*** network output for any parameter value and the *desired* output
  - Error typically defined as the *sum* of the squared error over individual training instances

Overfitting



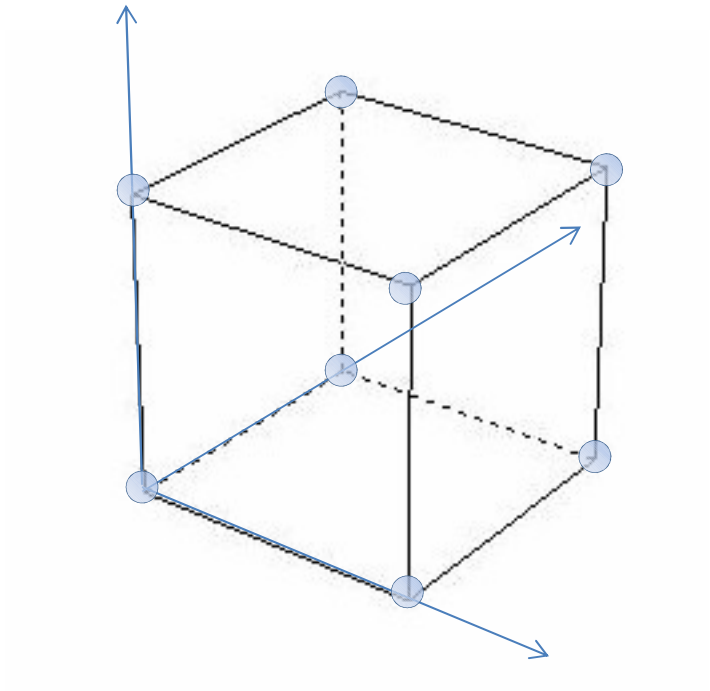
- Problem: Network may just learn the values at the inputs
  - Learn the red curve instead of the dotted blue one
    - Given only the red vertical bars as inputs

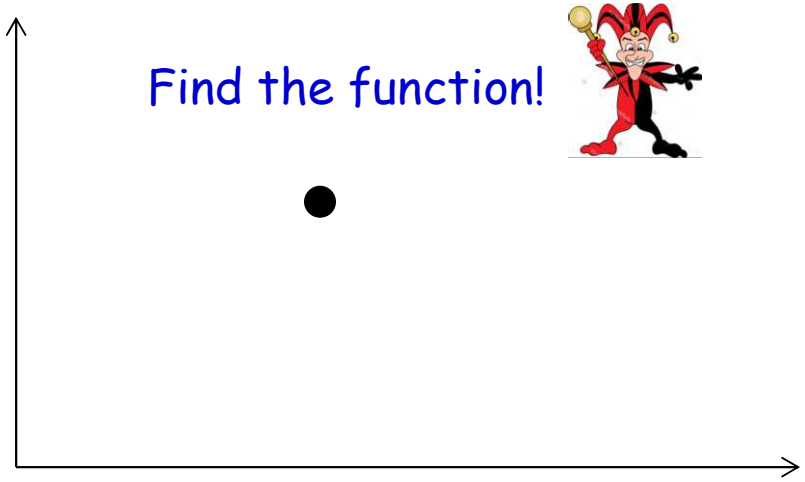
Data under-specification



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

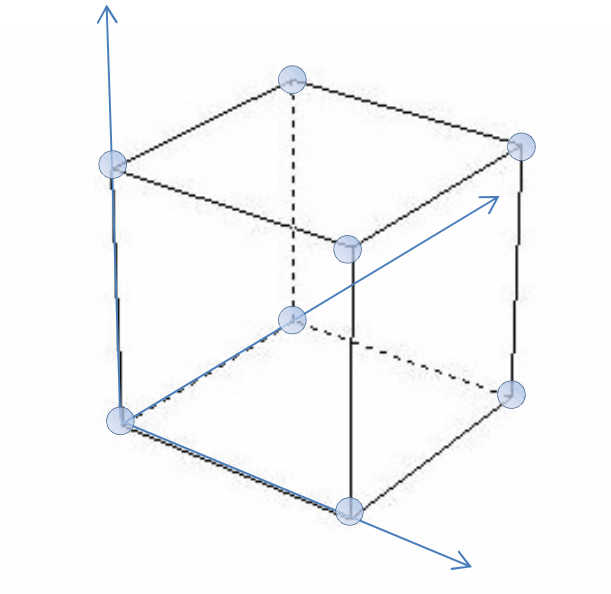
Data under-specification in learning

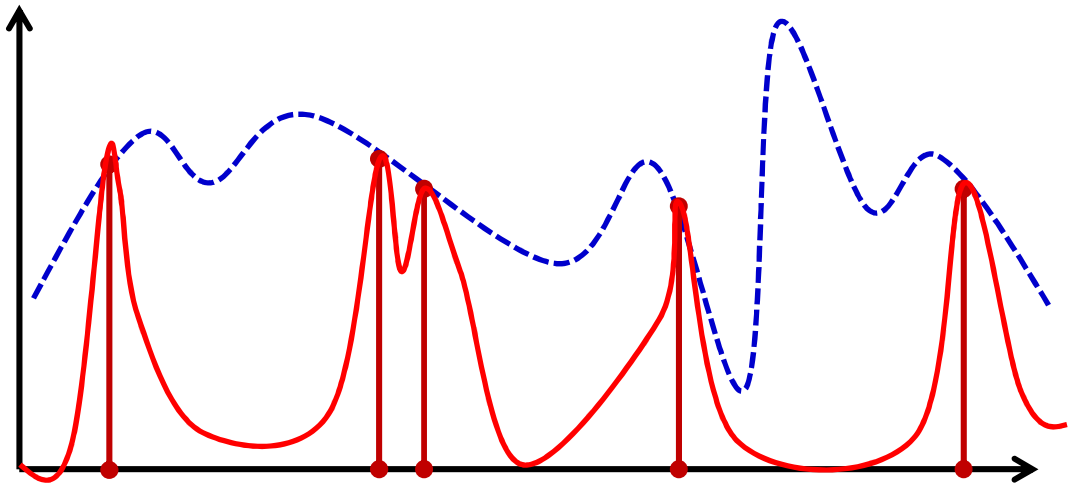




- Consider a binary 100-dimensional input
- There are $2^{100}=10^{30}$ possible inputs
- Complete specification of the function will require specification of 10^{30} output values
- A training set with only 10^{15} training instances will be off by a factor of 10^{15}

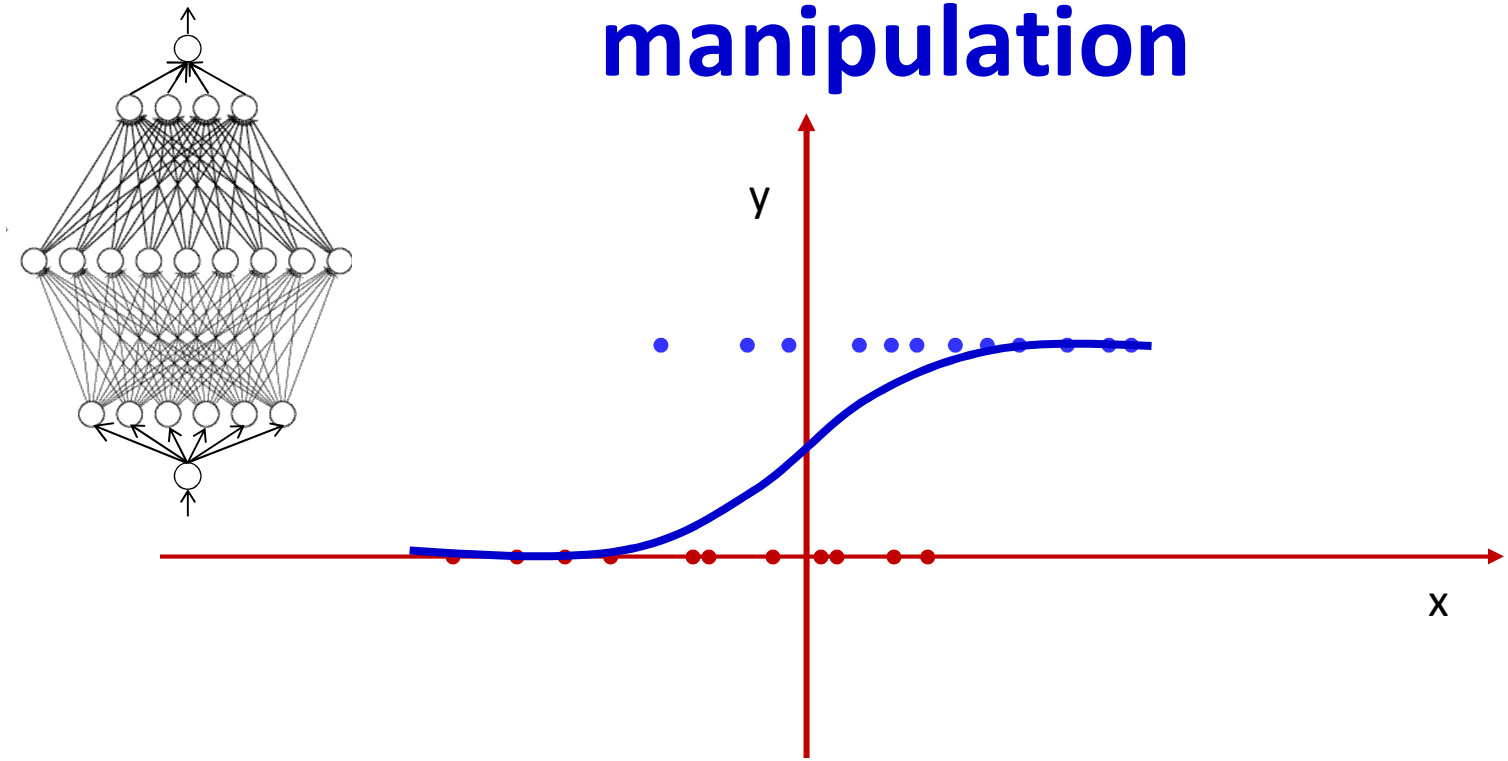
Need “smoothing” constraints





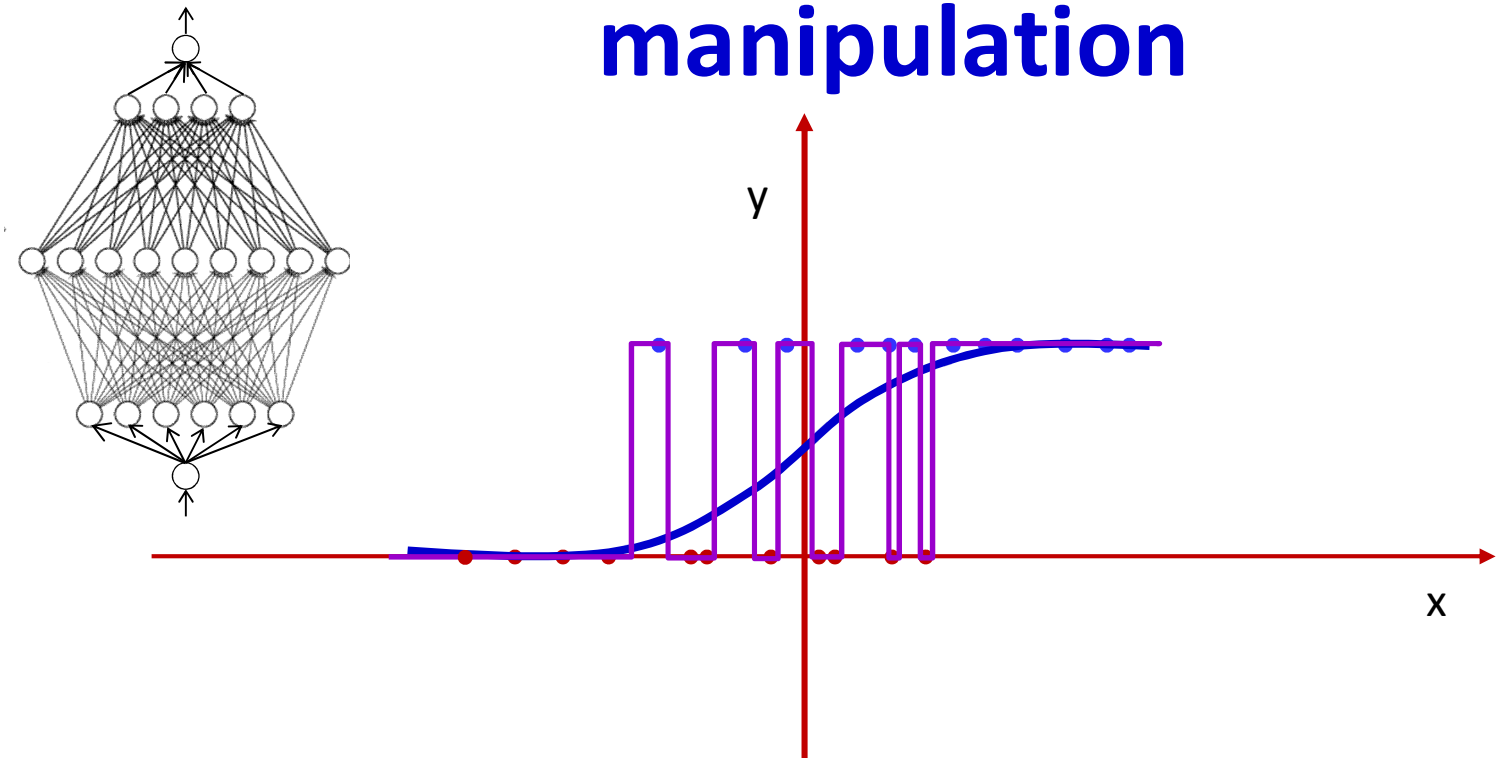
- Need additional constraints that will “fill in” the missing regions acceptably
  - Generalization

Smoothness through weight manipulation



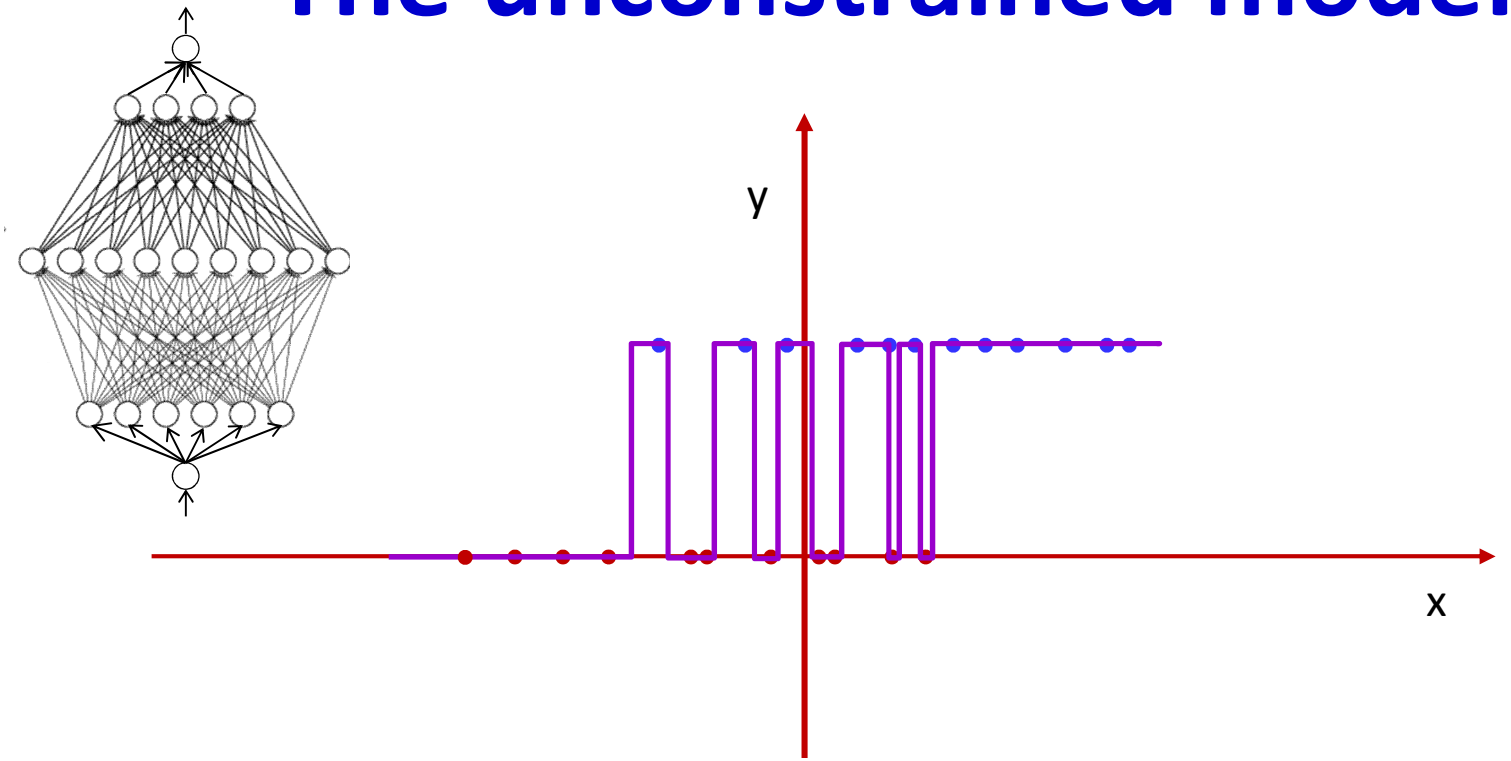
- Illustrative example: Simple binary classifier
  - The “desired” output is generally smooth

Smoothness through weight manipulation



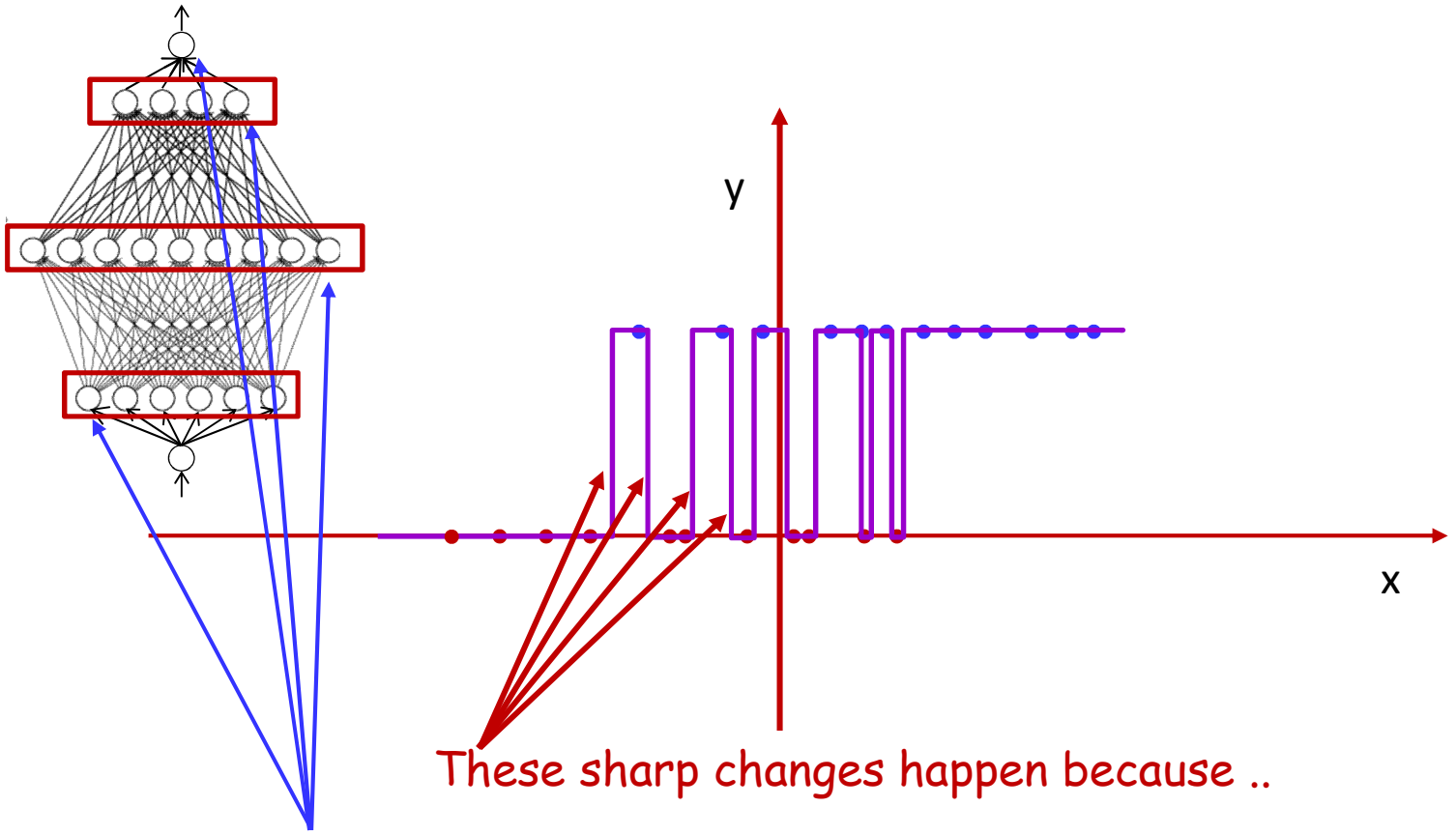
- Illustrative example: Simple binary classifier
  - The "desired" output is generally smooth
    - Capture statistical or average trends
  - An unconstrained model will model individual instances instead

The unconstrained model



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

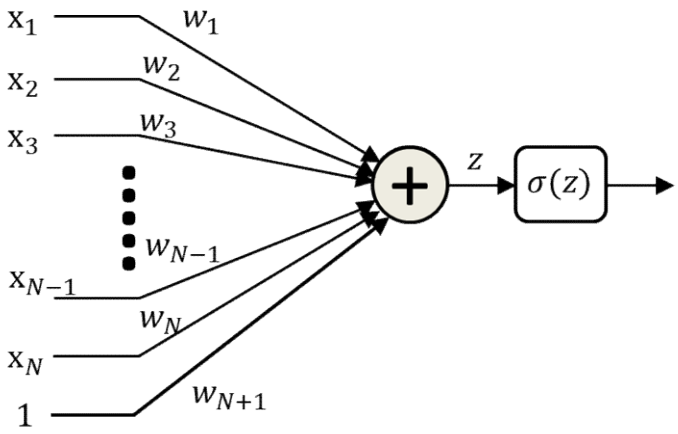
Why overfitting

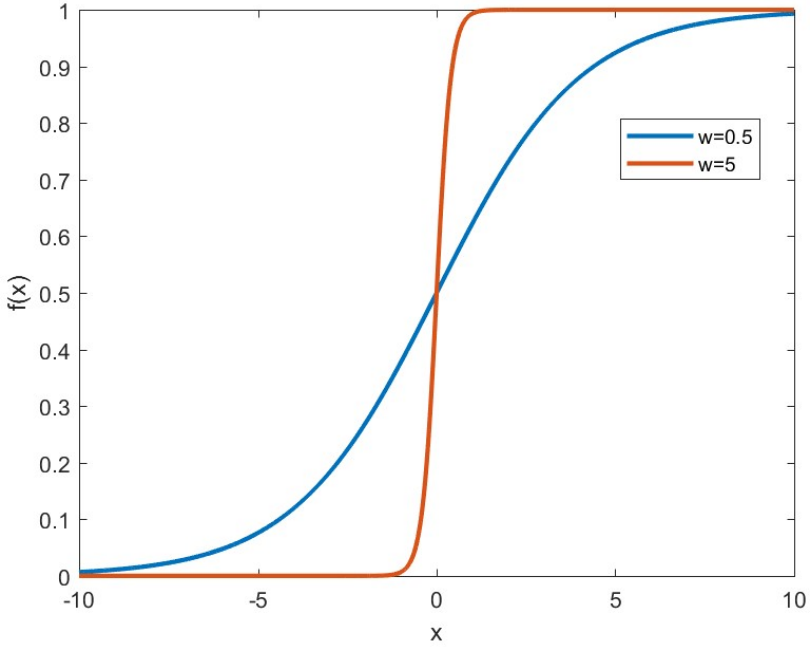


These sharp changes happen because ..

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

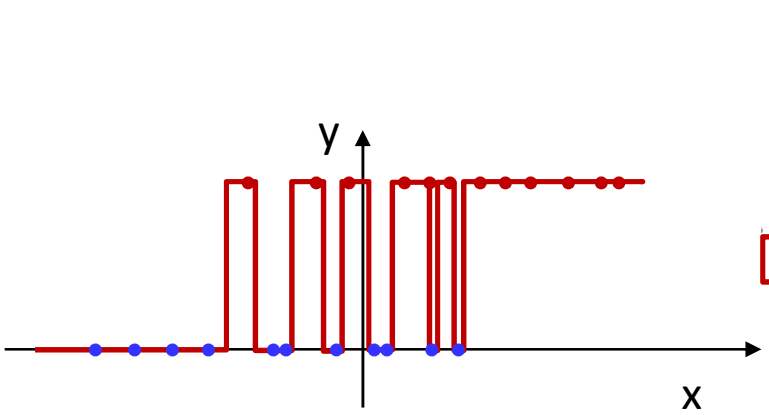
The individual perceptron

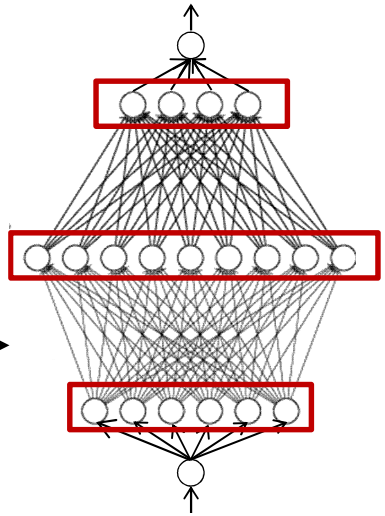


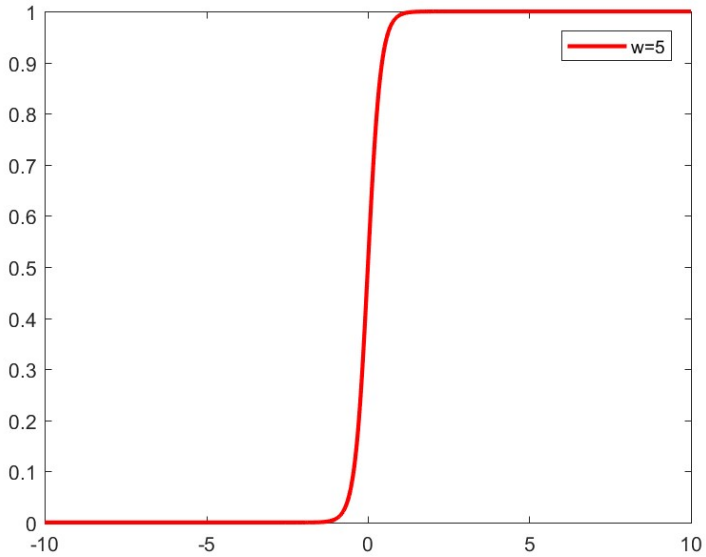


- Using a sigmoid activation
  - As $|w|$ increases, the response becomes steeper

Smoothness through weight manipulation

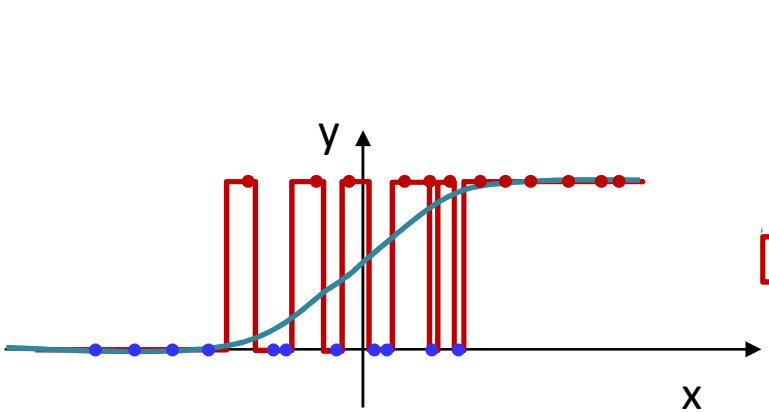


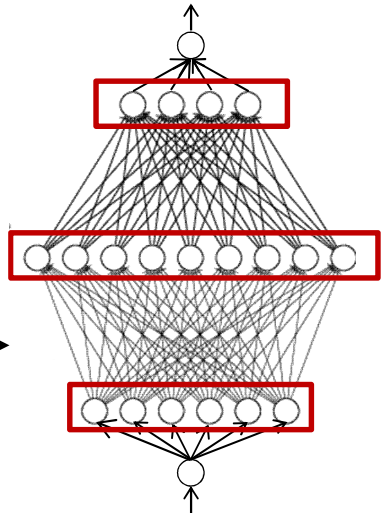


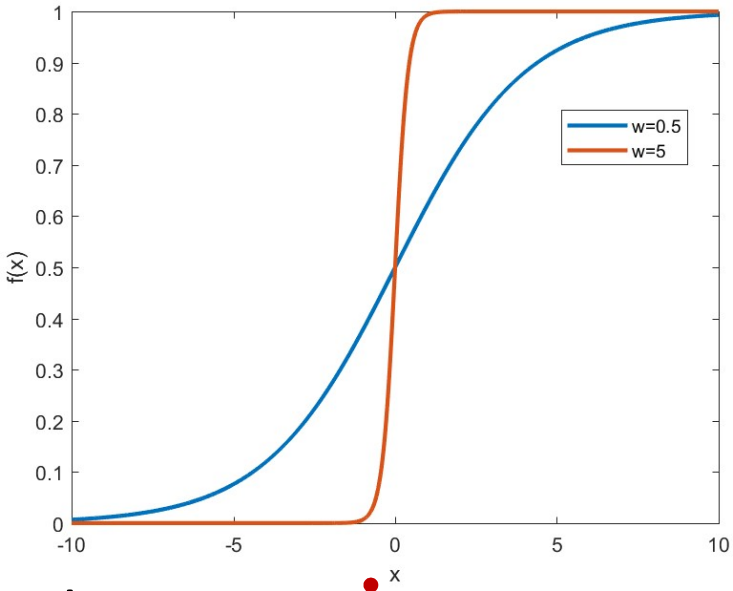


- Steep changes that enable overfitted responses are facilitated by perceptrons with large w

Smoothness through weight manipulation

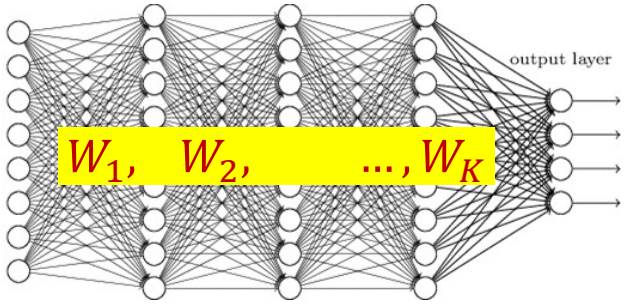






- Steep changes that enable overfitted responses are facilitated by perceptrons with large w
- Constraining the weights w to be low will force slower perceptrons and smoother output response

Objective function for neural networks



Y_t

Desired output of network: d_t

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

Batch training error:

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

- Conventional training: minimize the total error:

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

Smoothness through weight constraints

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

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

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

- λ is the regularization parameter whose value depends on how important it is for us to want to minimize the weights

- Increasing λ assigns greater importance to shrinking the weights
  - Make greater error on training data, to obtain a more acceptable network

Regularizing the weights

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

- Batch mode:

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

- SGD:

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

- Minibatch:

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

- Update rule:

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

Incremental Update: Mini-batch update

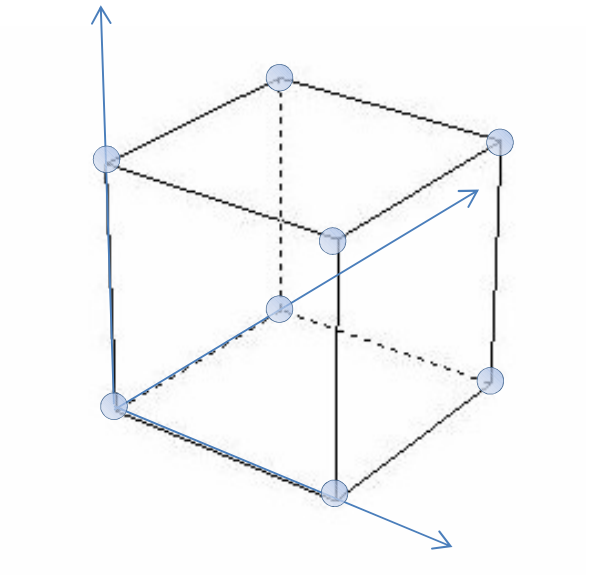
- Given (X_1, d_1), (X_2, d_2),…, (X_T, d_T)
- Initialize all weights $W_1, W_2, \ldots, W_K$; $j = 0$
- Do:
  - Randomly permute (X_1, d_1), (X_2, d_2),…, (X_T, d_T)
  - For $t = 1 : b : T$
    - $j = j + 1$
    - For every layer k:
      - $\Delta W_k = 0$
    - For t' = t : t+b-1
      - For every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

$$W_k = W_k - \eta_j(\Delta W_k + \lambda W_k)$$

- Until Err has converged

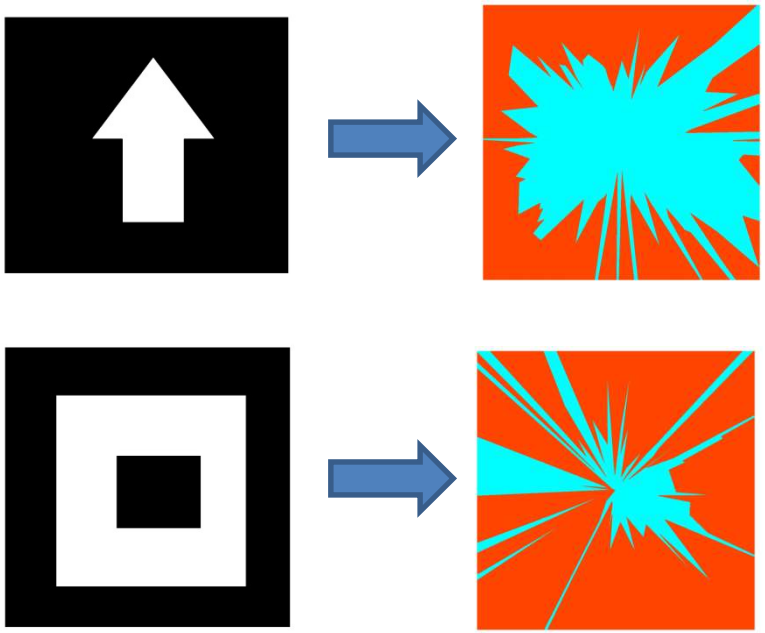
Smoothness through network structure

- 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



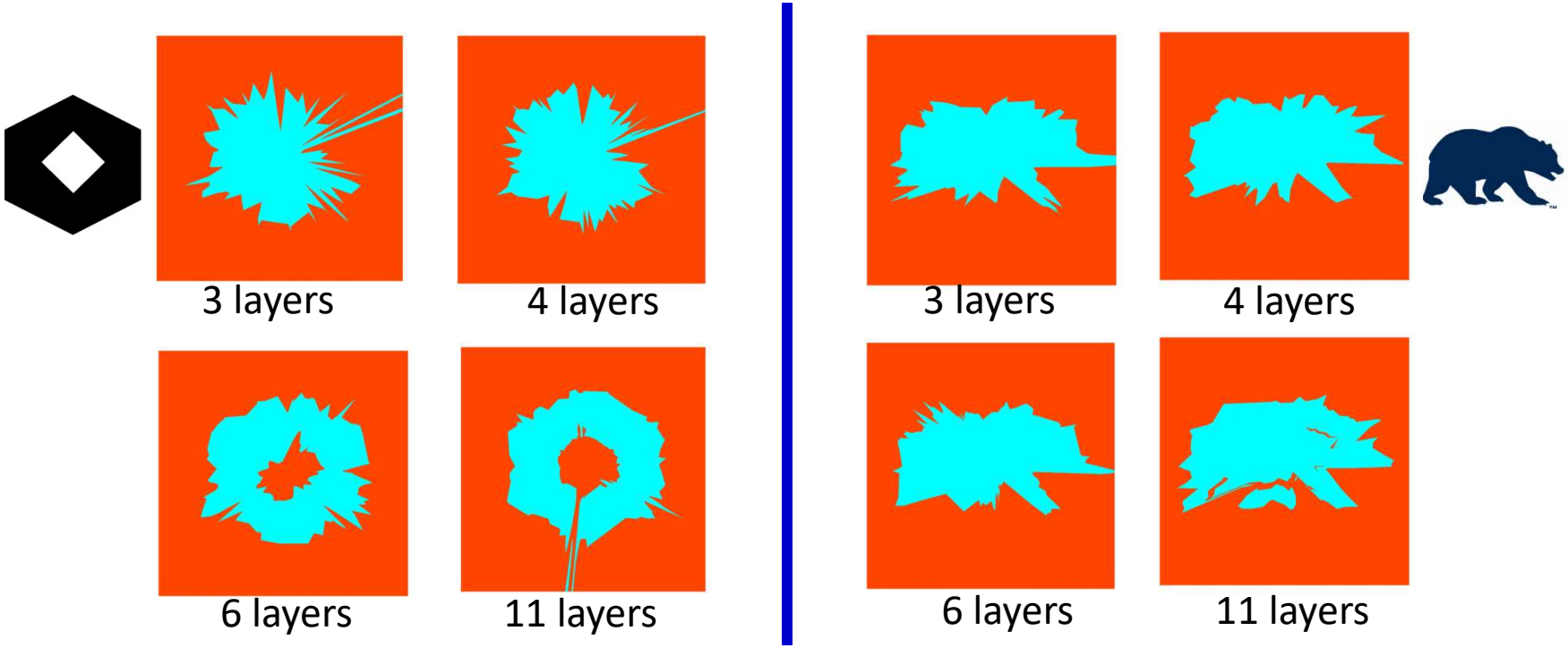
- *For a given number of parameters deeper networks impose more smoothness than shallow ones*
  - Each layer works on the already smooth surface output by the previous layer

Even when we get it all right



- Typical results (varies with initialization)
- 1000 training points – 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..

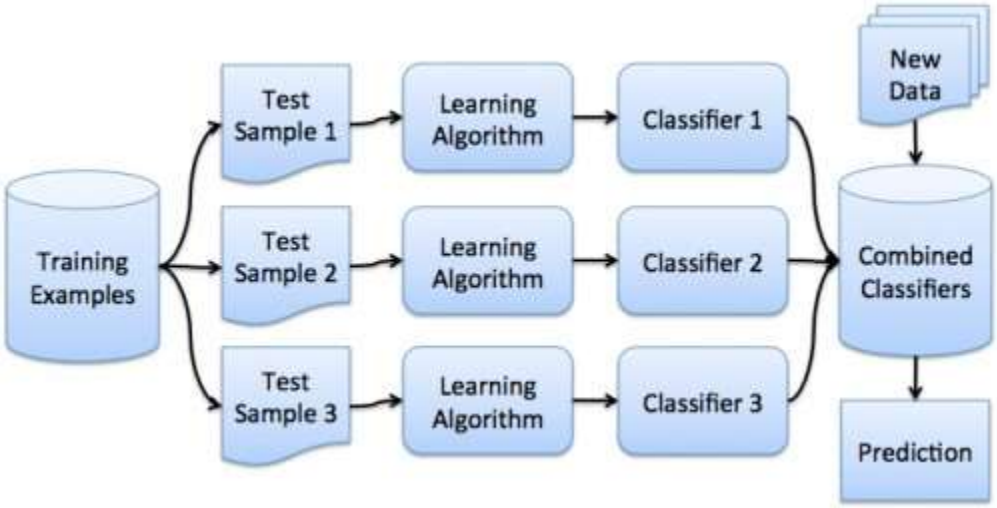
- Possibly the most influential method has been "dropout"

Story so far

- Gradient descent can be sped up by incremental updates
- Convergence can be improved using smoothed updates

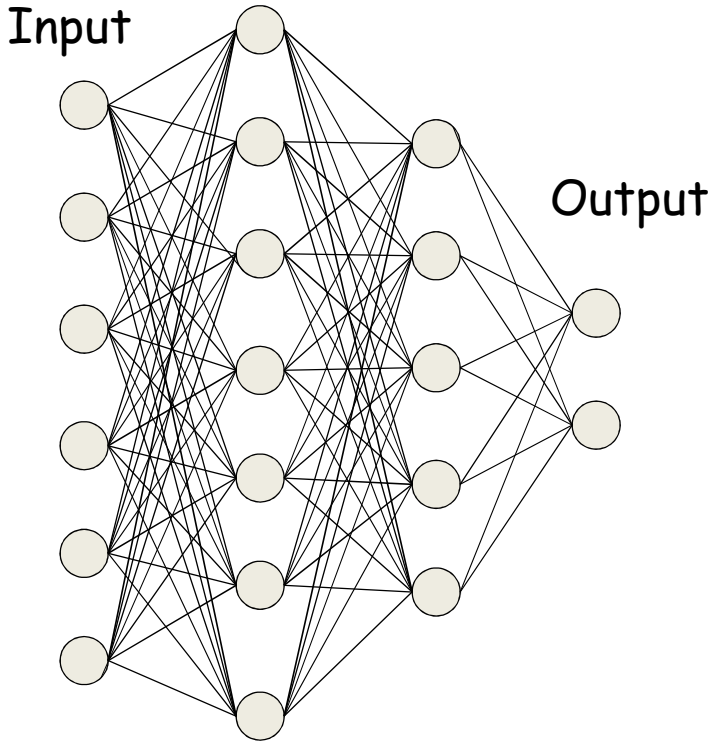
- The choice of divergence affects both the learned network and results
- Covariate shift between training and test may cause problems and may be handled by batch normalization
- Data underspecification can result in overfitted models and must be handled by regularization and more constrained (generally deeper) network architectures

A brief detour.. Bagging



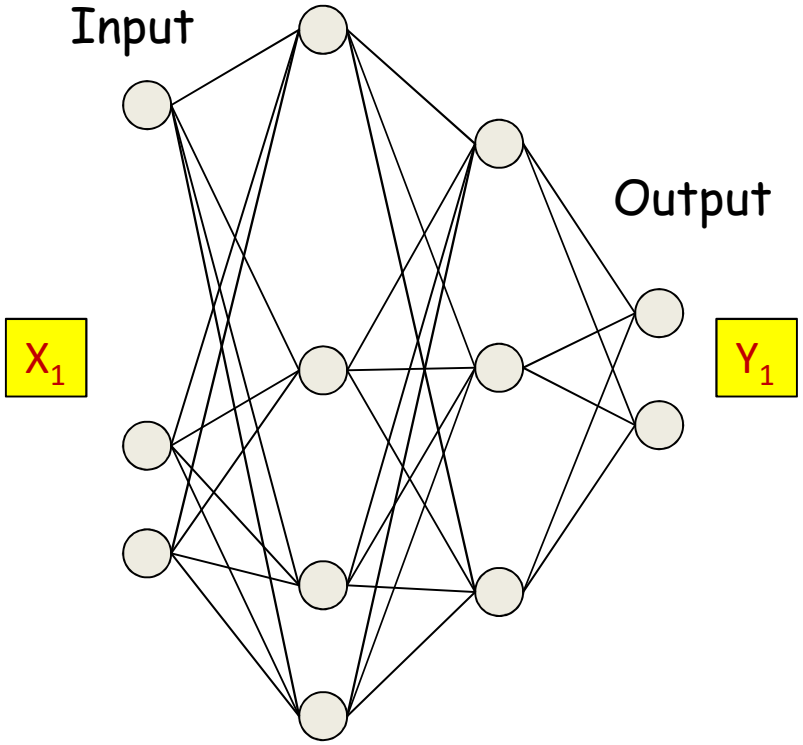
- Popular method proposed by Leo Breiman:
  - Sample training data and train several different classifiers
  - Classify test instance with entire ensemble of classifiers
  - Vote across classifiers for final decision
  - Empirically shown to improve significantly over training a single classifier from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

Dropout



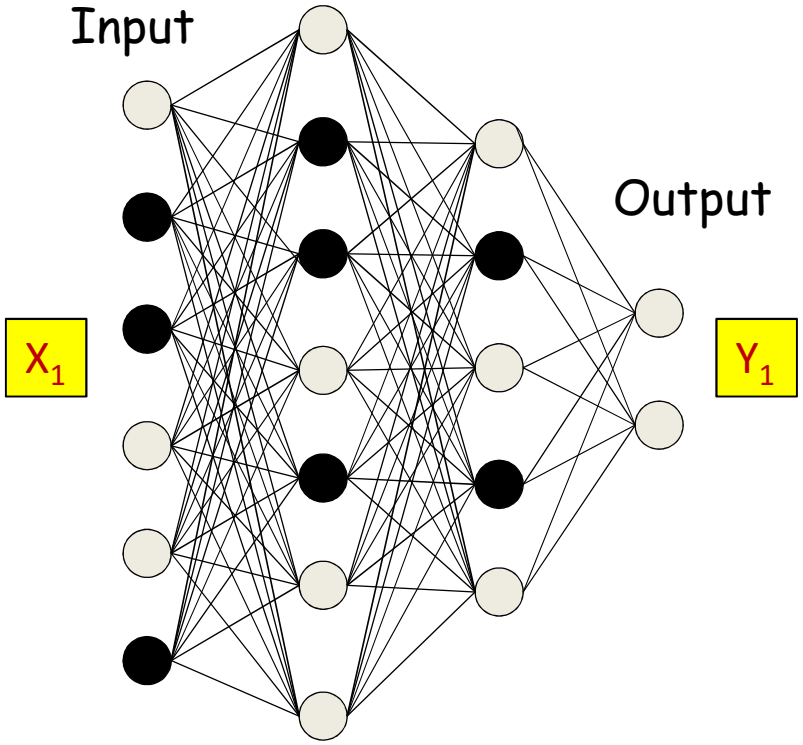
- **During training:** For each input, at each iteration, "turn off" each neuron with a probability 1-α

Dropout



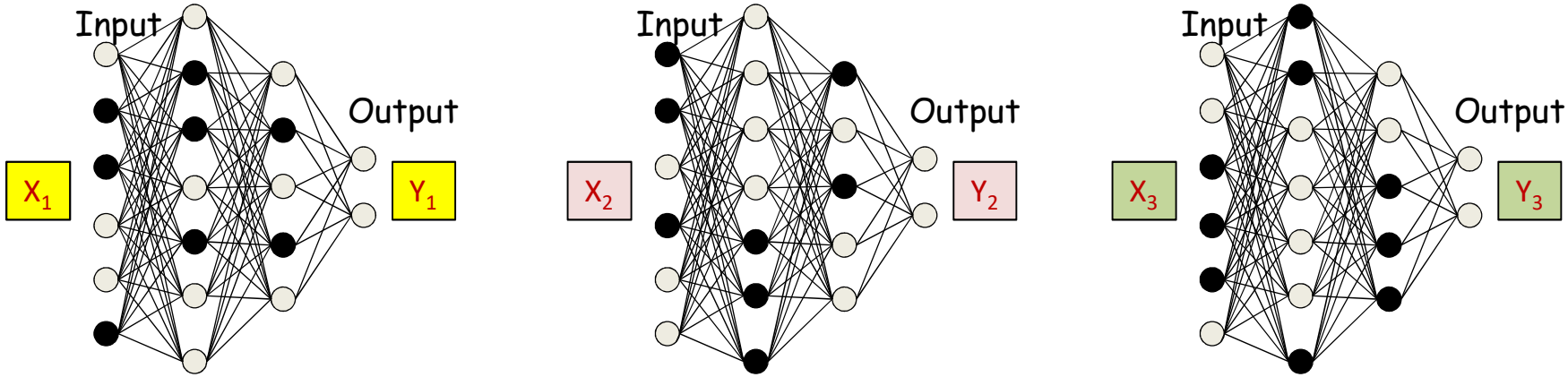
- **During training:** For each input, at each iteration, "turn off" each neuron with a probability 1-α
  - Also turn off inputs similarly

Dropout



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

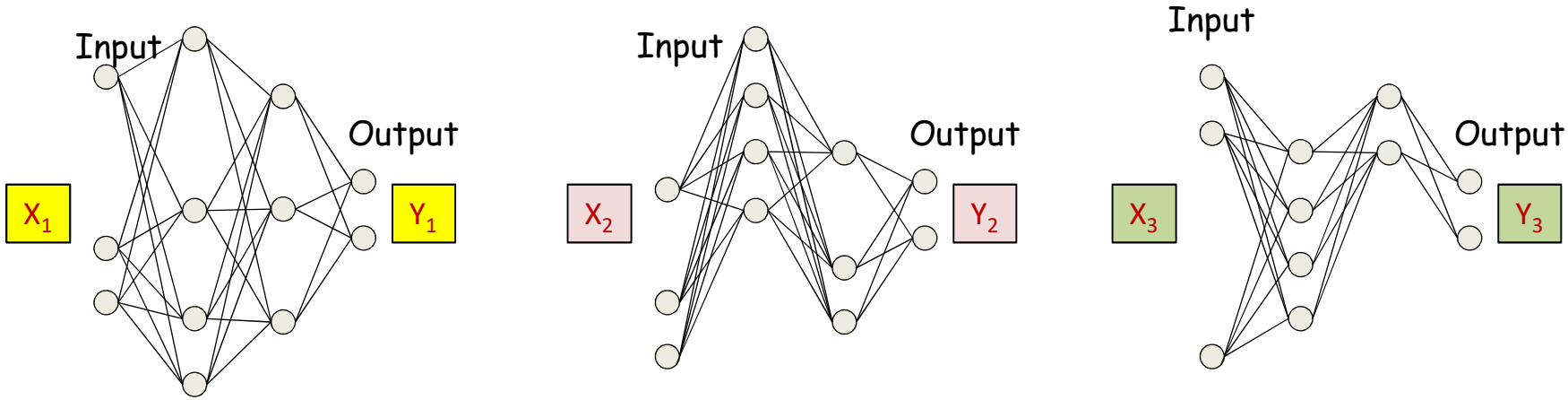
Dropout



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-α
  - In practice, set them to 0 according to the success of a Bernoulli random number generator with success probability 1-α

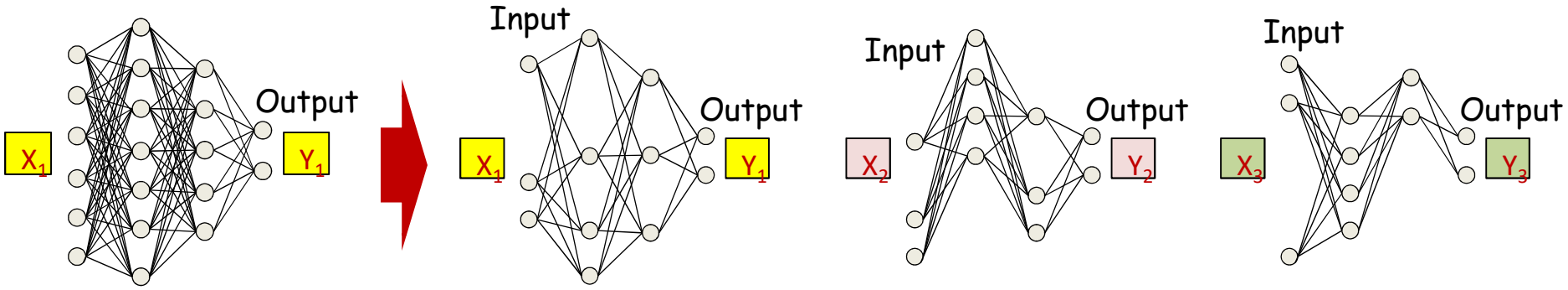
Dropout



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

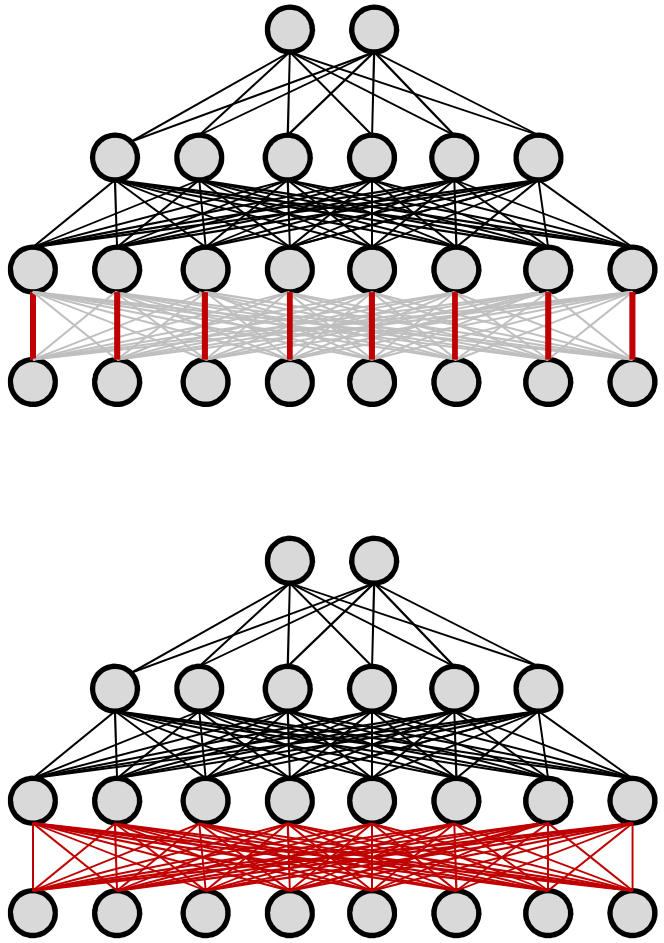
Statistical Interpretation



- 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

Dropout as a mechanism to increase pattern density

- Dropout forces the neurons to learn “rich” and redundant patterns
- E.g. without dropout, a non-compressive layer may just “clone” its input to its output
  - Transferring the task of learning to the rest of the network upstream
- Dropout forces the neurons to learn denser patterns
  - With redundancy



The forward pass

- Input: D dimensional vector $\mathbf{x} = [x_j,\ j = 1 \ldots D]$
- Set:
  - $D_0 = D,$ is the width of the 0th (input) layer
  - $y_j^{(0)} = x_j,\ j = 1 \ldots D; \qquad y_0^{(k=1\ldots N)} = x_0 = 1$
- For layer $k = 1 \ldots N$
  - For $j = 1 \ldots D_k$
    - $z_j^{(k)} = \sum_{i=0}^{N_k} w_{i,j}^{(k)} y_i^{(k-1)} + b_j^{(k)}$
    - $y_j^{(k)} = f_k\left(z_j^{(k)}\right)$
    - If $(k = dropout\ layer)$:
      - $mask(k,j) = Bernoulli(\alpha)$
      - If $mask(k,j) == 0$
        - » $y_j^{(k)} = 0$
- Output:
  - $Y = y_j^{(N)}, j = 1..D_N$

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 $i = 1 \ldots D_k$
    - If (not dropout layer OR $mask(k,i)$)
      - $\frac{\partial Div}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial Div}{\partial z_j^{(k+1)}} mask(k+1,j)$
      - $\frac{\partial Div}{\partial z_i^{(k)}} = f_k'\left(z_i^{(k)}\right)\frac{\partial Div}{\partial y_i^{(k)}}$
      - $\frac{\partial Div}{\partial w_{ij}^{(k+1)}} = y_i^{(k)} \frac{\partial Div}{\partial z_j^{(k+1)}} mask(k+1,j)$ for $j = 1 \ldots D_{k+1}$
    - Else
      - $\frac{\partial Di}{\partial z_i^{(k)}} = 0$

What each neuron computes

- Each neuron actually has the following activation:

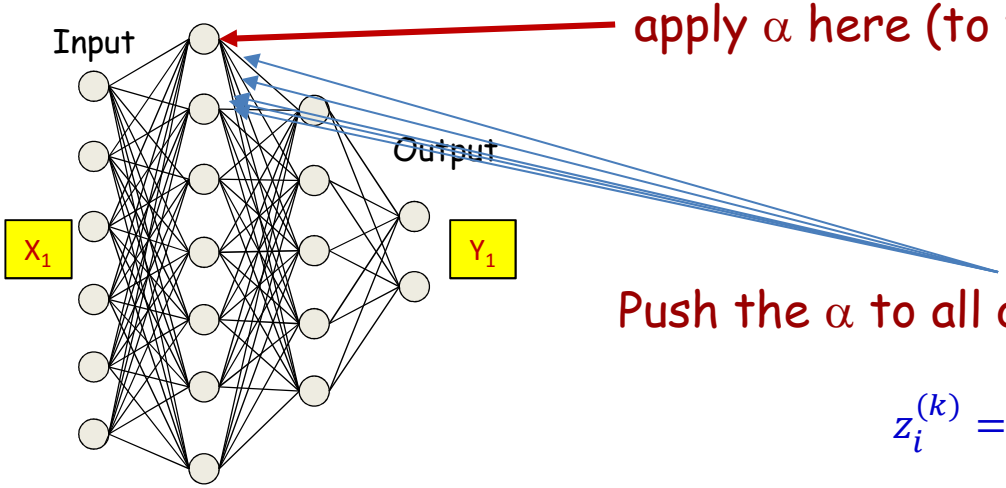
$$y_i^{(k)} = D\sigma\left(\sum_j w_{ji}^{(k)} y_j^{(k-1)} + b_i^{(k)}\right)$$

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

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

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

Dropout during test: implementation



apply α here (to the output of the neuron) OR..

Push the α to all outgoing weights

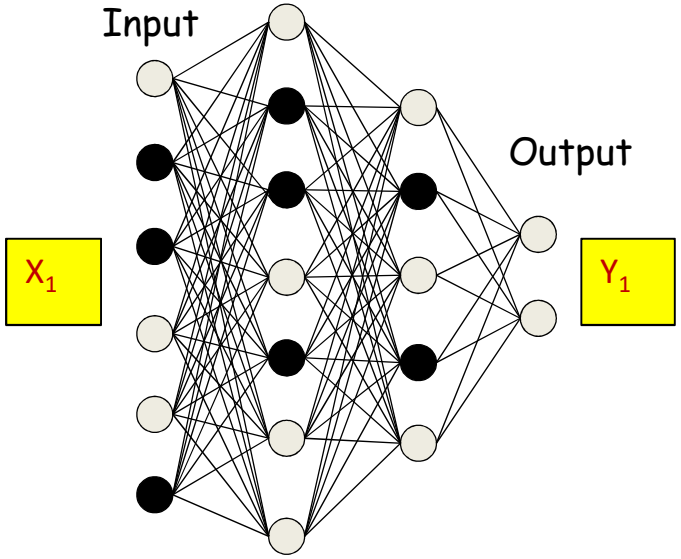
$$y_i^{(k)} = \alpha\sigma\left(z_i^{(k)}\right)$$

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

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

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

Dropout : alternate implementation

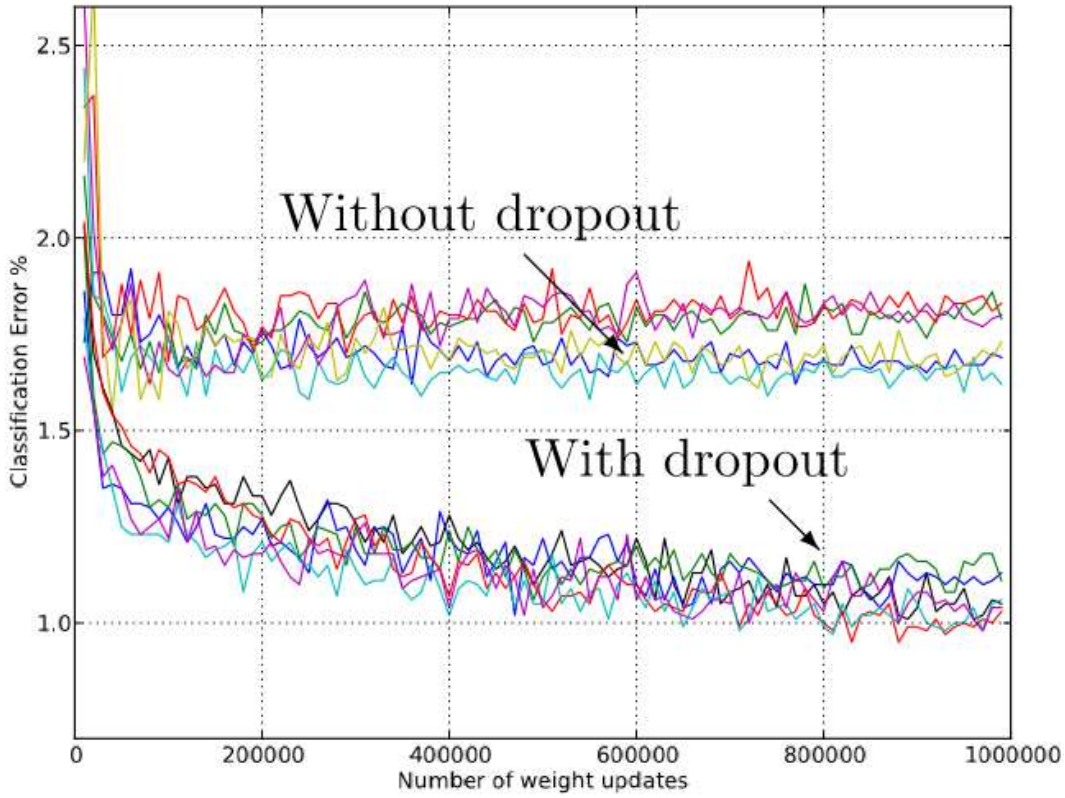


- Alternately, during *training,* replace the activation of all neurons in the network by $\alpha^{-1}\sigma(.)$
  - This does not affect the dropout procedure itself
  - We will use $\sigma(.)$ as the activation during testing, and not modify the weights

The forward pass (testing)

- Input: D dimensional vector $\mathbf{x} = [x_j,\ j = 1 \ldots D]$
- Set:
  - $D_0 = D$, is the width of the 0th (input) layer
  - $y_j^{(0)} = x_j,\ j = 1 \ldots D; \quad y_0^{(k=1\ldots N)} = x_0 = 1$
- For layer $k = 1 \ldots N$
  - For $j = 1 \ldots D_k$
    - $z_j^{(k)} = \sum_{i=0}^{N_k} w_{i,j}^{(k)} y_i^{(k-1)} + b_j^{(k)}$
    - $y_j^{(k)} = f_k\left(z_j^{(k)}\right)$
    - If $(k = dropout\ layer)$:
      - » $y_j^{(k)} = y_j^{(k)}/\alpha$
    - Else
      - » $y_j^{(k)} = 0$
- Output:
  - $Y = y_j^{(N)}, j = 1..D_N$

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

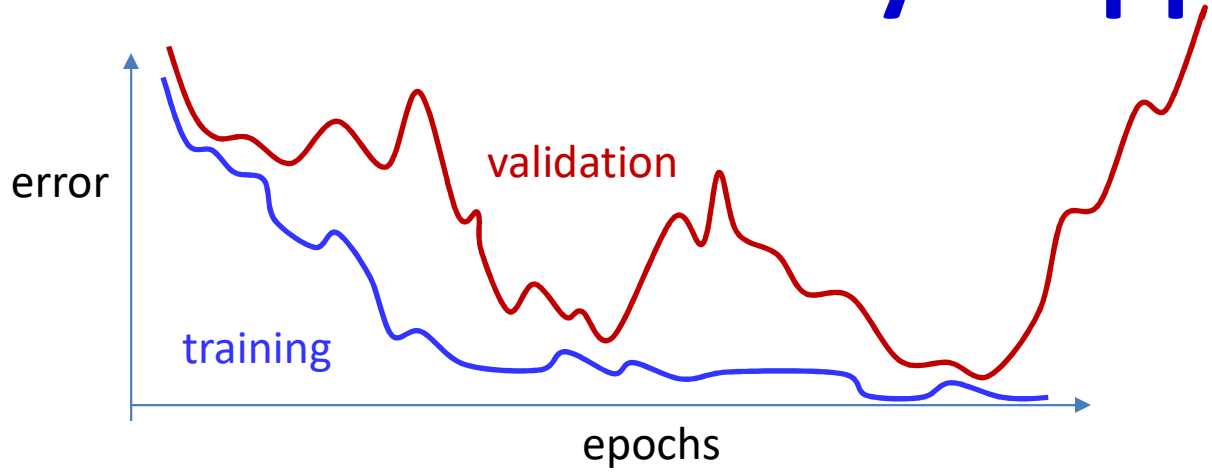
- Zoneout: For RNNs
  - Randomly chosen units remain unchanged across a time transition
- Dropconnect
  - Drop individual connections, instead of nodes
- Shakeout
  - Scale *up* the weights of randomly selected weights
    - $|w| \rightarrow \alpha|w| + (1-\alpha)c$
  - Fix remaining weights to a negative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

Story so far

- Gradient descent can be sped up by incremental updates
- Convergence can be improved using smoothed updates

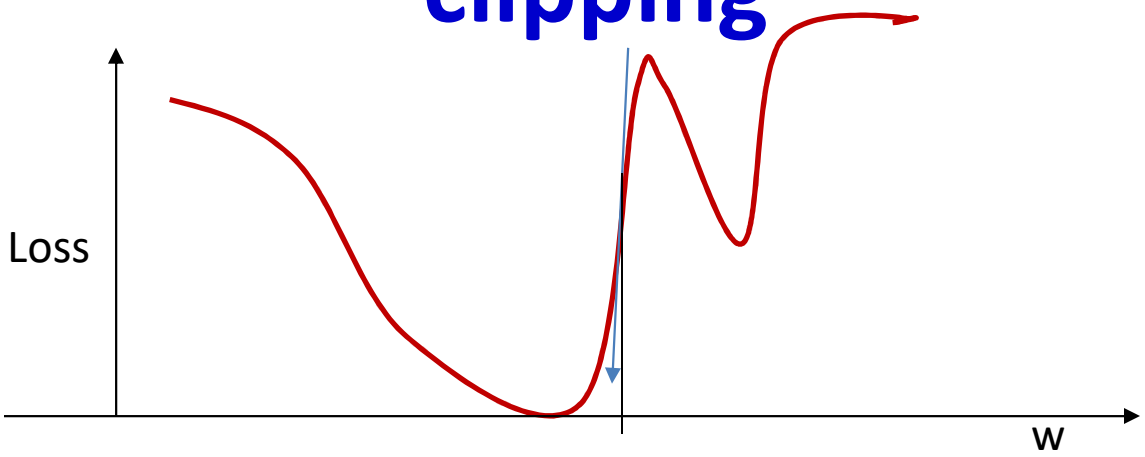
- The choice of divergence affects both the learned network and results
- Covariate shift between training and test may cause problems and may be handled by batch normalization
- Data underspecification can result in overfitted models and must be handled by regularization and more constrained (generally deeper) network architectures
- "Dropout" is a stochastic data/model erasure method that sometimes forces the network to learn more robust models

Other heuristics: Early stopping



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

Additional heuristics: Gradient clipping



- Often the derivative will be too high
  - When the divergence has a steep slope
  - This can result in instability
- **Gradient clipping**: set a ceiling on derivative value

$$if\ \partial_w D > \theta\ then\ \partial_w D = \theta$$

  - Typical θ value is 5

Additional heuristics: Data Augmentation


CocaColaZero1_1.png


CocaColaZero1_2.png


CocaColaZero1_3.png


CocaColaZero1_4.png


CocaColaZero1_5.png


CocaColaZero1_6.png


CocaColaZero1_7.png


CocaColaZero1_8.png

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

Other tricks

- Normalize the input:
  - Apply covariate shift to entire training data to make it 0 mean, unit variance
  - Equivalent of batch norm on input

- A variety of other tricks are applied
  - Initialization techniques
    - Typically initialized randomly
    - Key point: neurons with identical connections that are identically initialized will never diverge
  - Practice makes man perfect

Setting up a problem

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

In closing

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