Train Your Model: Tricks

CSE 849 Deep Learning

Spring 2025

Zijun Cui

Partial slides are adapted from CMU 11-785 Introduction to Deep Learning by Bhiksha Raj

Class Schedule

- We are still on schedule to cover all the topics
- We will follow original class structure

Week of	Tuesday	Thursday
1/13	1 Introduction	2 Why Deep
1/20	3 Learning a Neural Network	4 Forward and Backward Propagation **Project 0 (HW1) Out**
1/27	5 Automatic Differentiation	6 Create Your Model with PyTorch
2/3	7 Train Your Model: Optimization 1 (Convergence and Learning Rate)	8 Train Your Model: Optimization 2 (Stochastic GD and Mini Batch) **Project 1 Out; Project 0 (HW1) Due**
2/10	**Classes Cancelled**	**Classes Not Held**
2/17	9 Tame Your Model: Tricks **HW 2 Out.**	10 Convolutional Neural Network
2/24	11 Convolutional Neural Networks - 2	12 Recurrent Neural Network **Project 2 Out; Project 1 Due**
3/3	**No Class (Spring Break)**	**No Class (Spring Break)**
3/10 *** Last day to drop classes**	13 Recurrent Neural Network – 2 **HW 2 Due.**	14 Seq-to-Seq
3/17	15 Attention	16 Transformer **Project 3 Out; Project 2 Due**
3/24	17 Transformer and LLMs 1	18 Graph Neural Network
3/31	19 Graph Neural Network - 2	20 Probabilistic Deep Learning: Introduction
4/7	21 Generative Modeling	22 Generative Modeling 2 **Project 4 Out; Project 3 Due**
4/14	23 Diffusion Models	24 Diffusion Models 2
4/21	25 Bayesian Deep Learning	26 Deep Probabilistic Graphical Models

Correction

Select all that are wrong

A. SGD is an online version of batch updates

B. SGD can have oscillatory behavior if we do not randomize the order of the inputs

C. SGD can converge faster than batch updates, but arrive at poorer optima

D. SGD convergence to the global optimum can only be guaranteed if step sizes shrink across iterations, but sum to infinity in the limit

E. None of above

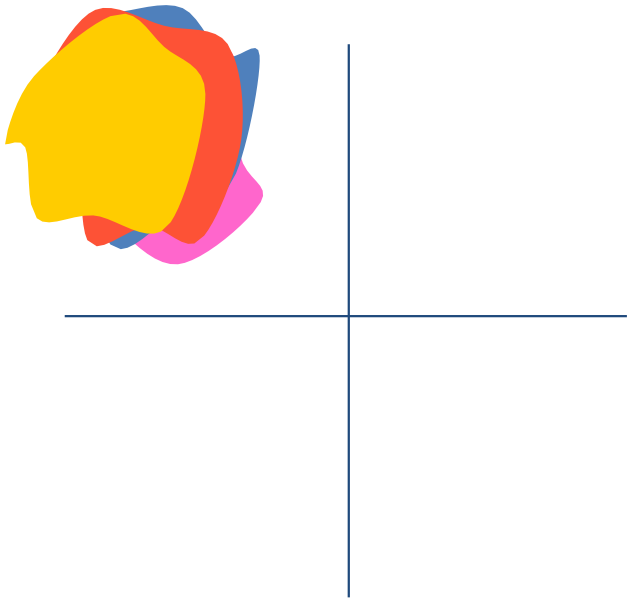
Correction

- Project 1
- Posted on Piazza
- In layers.py, it is said to calculate the gradient, update the weights, and then calculate grad_for_next.
- This is wrong.
- You should calculate grad_for_next **BEFORE** you update your weights.

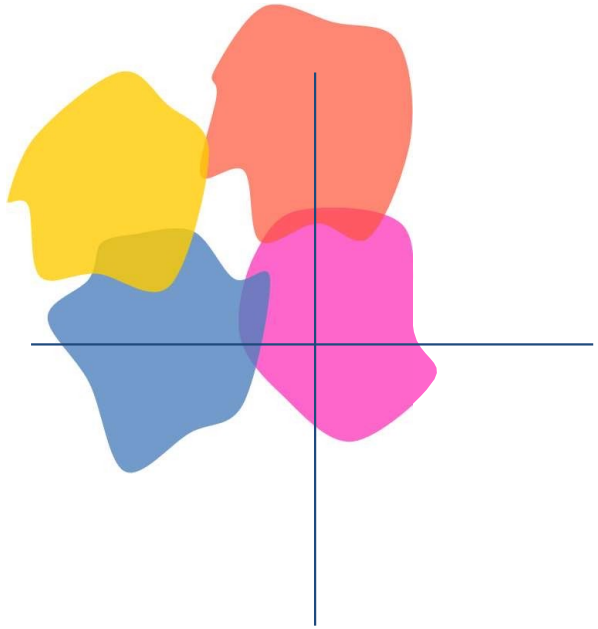
Topics for the day

- Tricks of the trade
  - Normalizations
  - Dropout
  - Other tricks
    - Gradient clipping
    - Data augmentation
    - Other hacks.

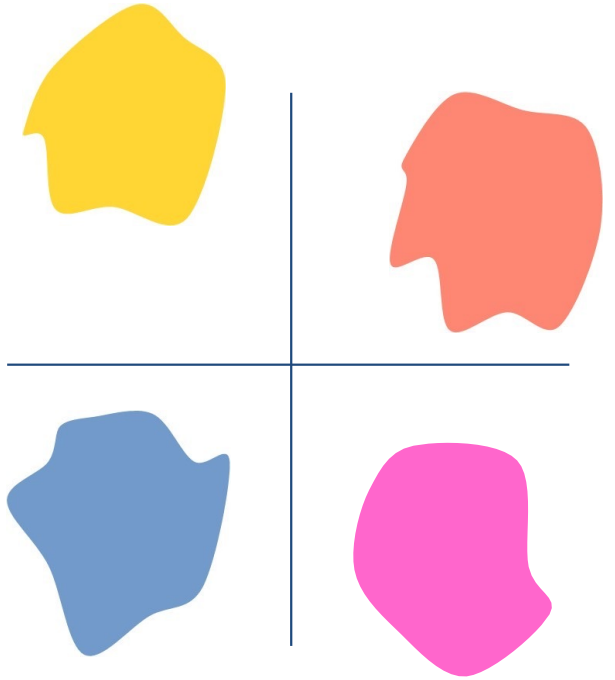
The problem of covariate shifts



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

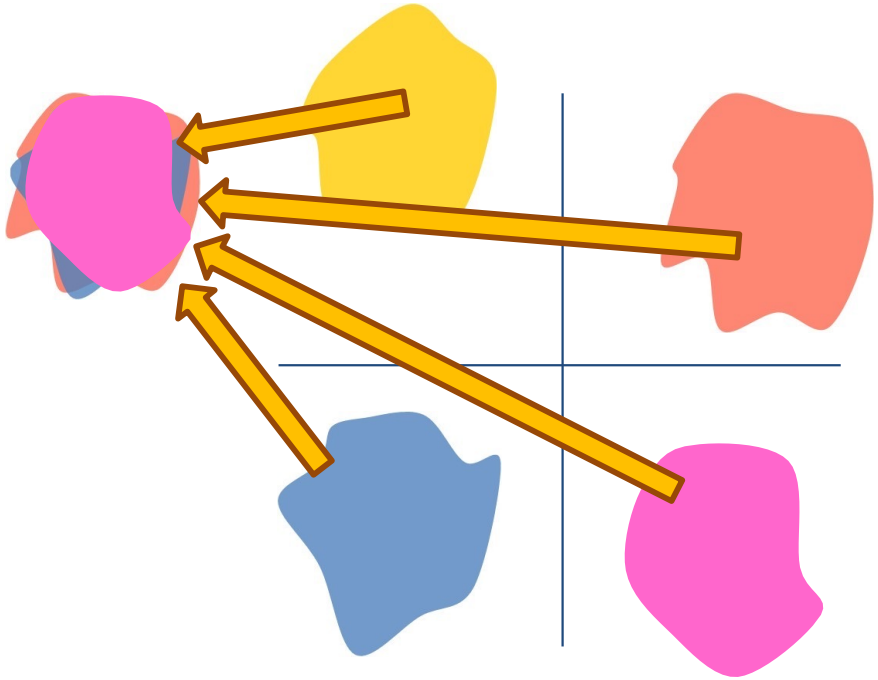


- In practice, each minibatch may have a different distribution
  - A “covariate shift”
  - Which may occur in *each* layer of the network



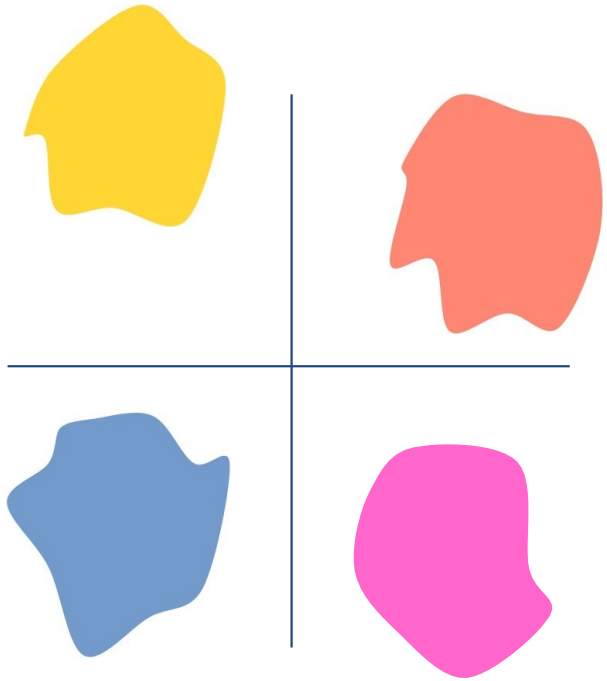
- The shifts can be large!
  - Can affect training badly

Solution: Move all minibatches to a "standard" location

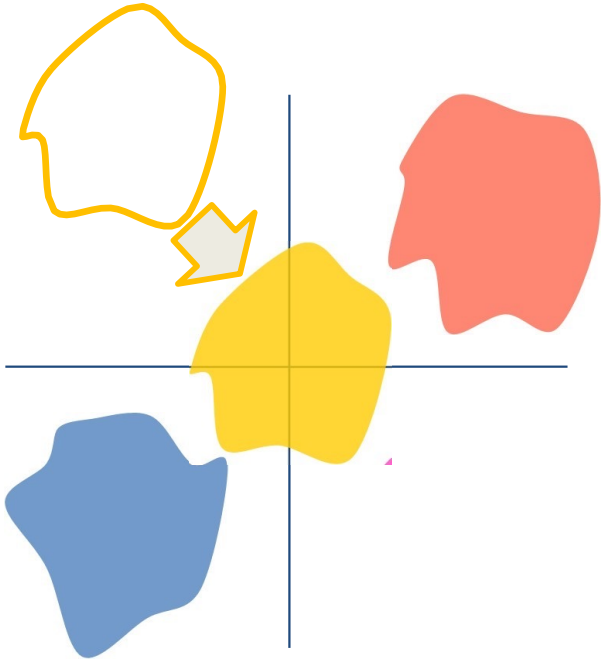


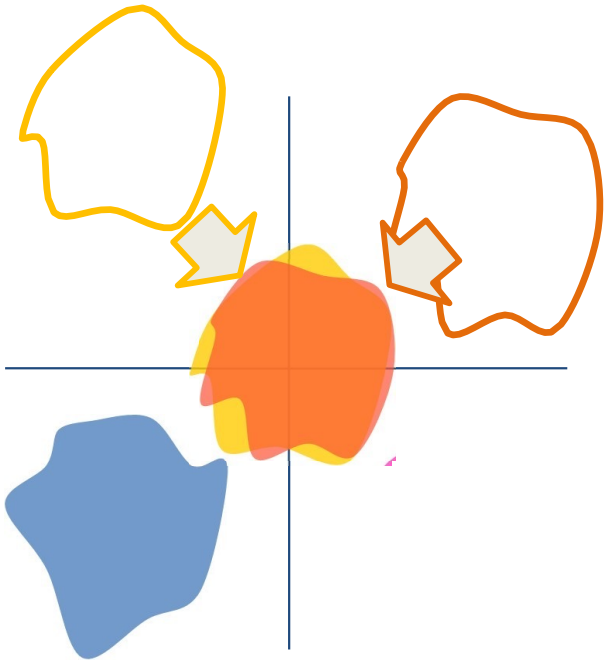
- "Move" all batches to a "standard" location of the space
  - But where?
  - To determine, we will follow a two-step process

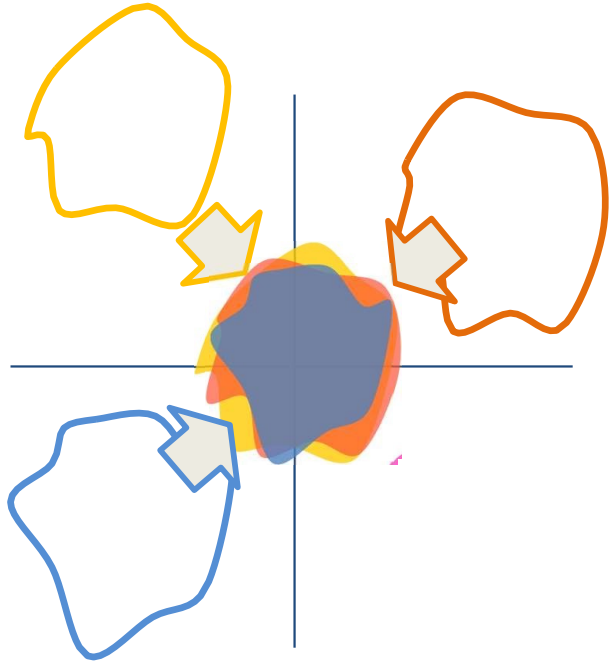
Move all minibatches to a “standard” location

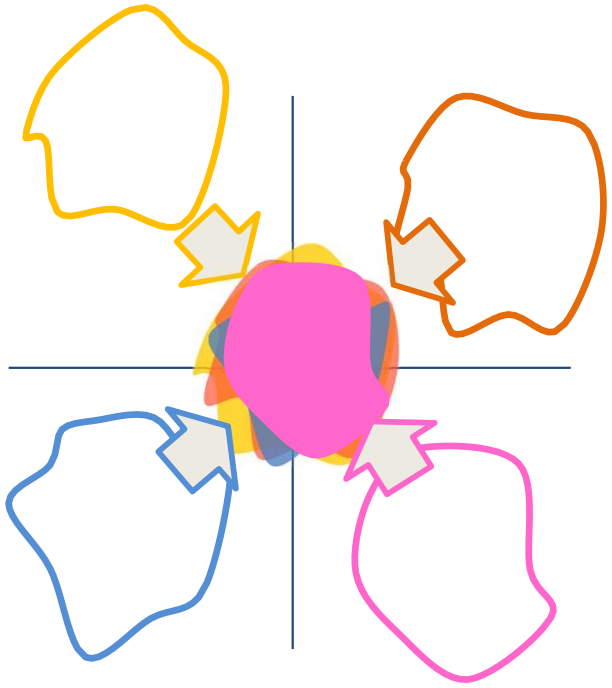


- “Move” all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches

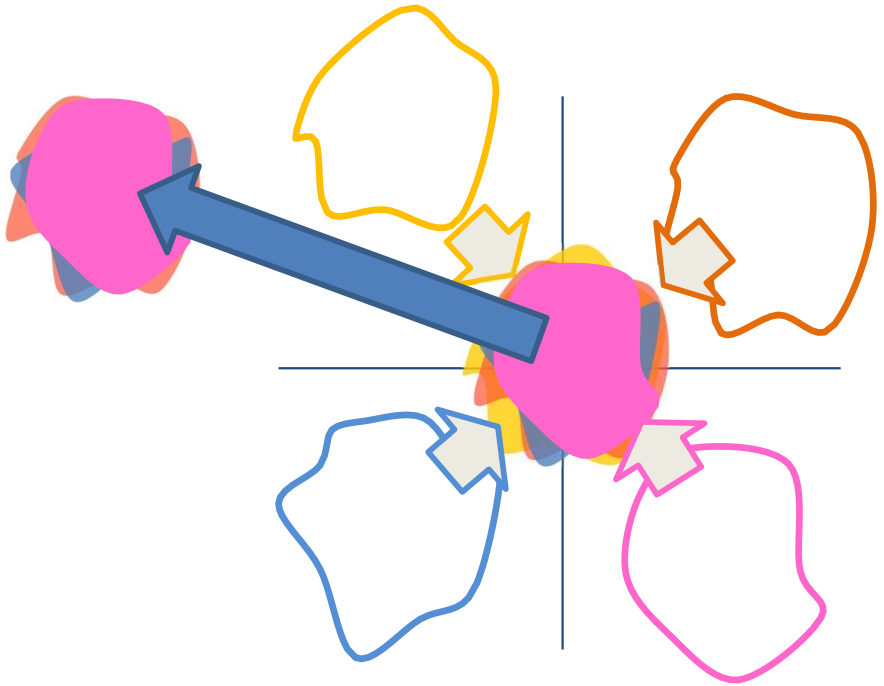






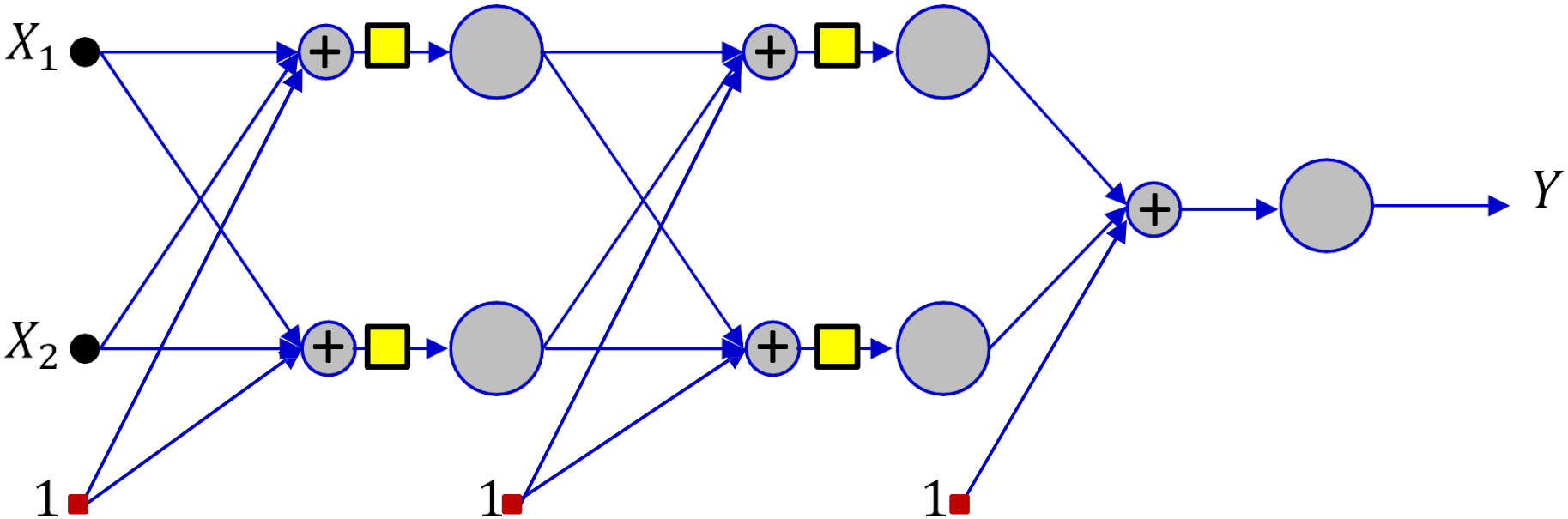


(Mini)Batch Normalization



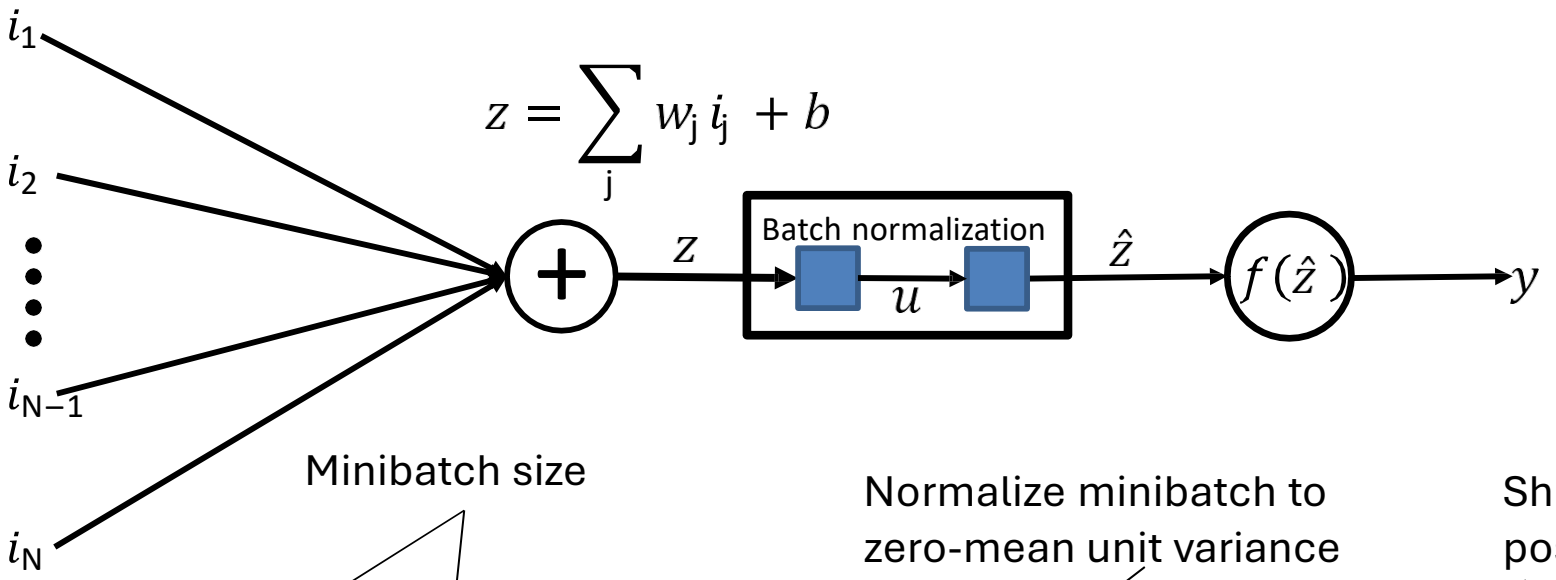
- “Move” all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches
- Then move the entire collection to the appropriate location

Batch normalization



- Batch normalization is a shift-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: Training



Minibatch size

Minibatch standard deviation

Normalize minibatch to zero-mean unit variance

Shift to right position

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

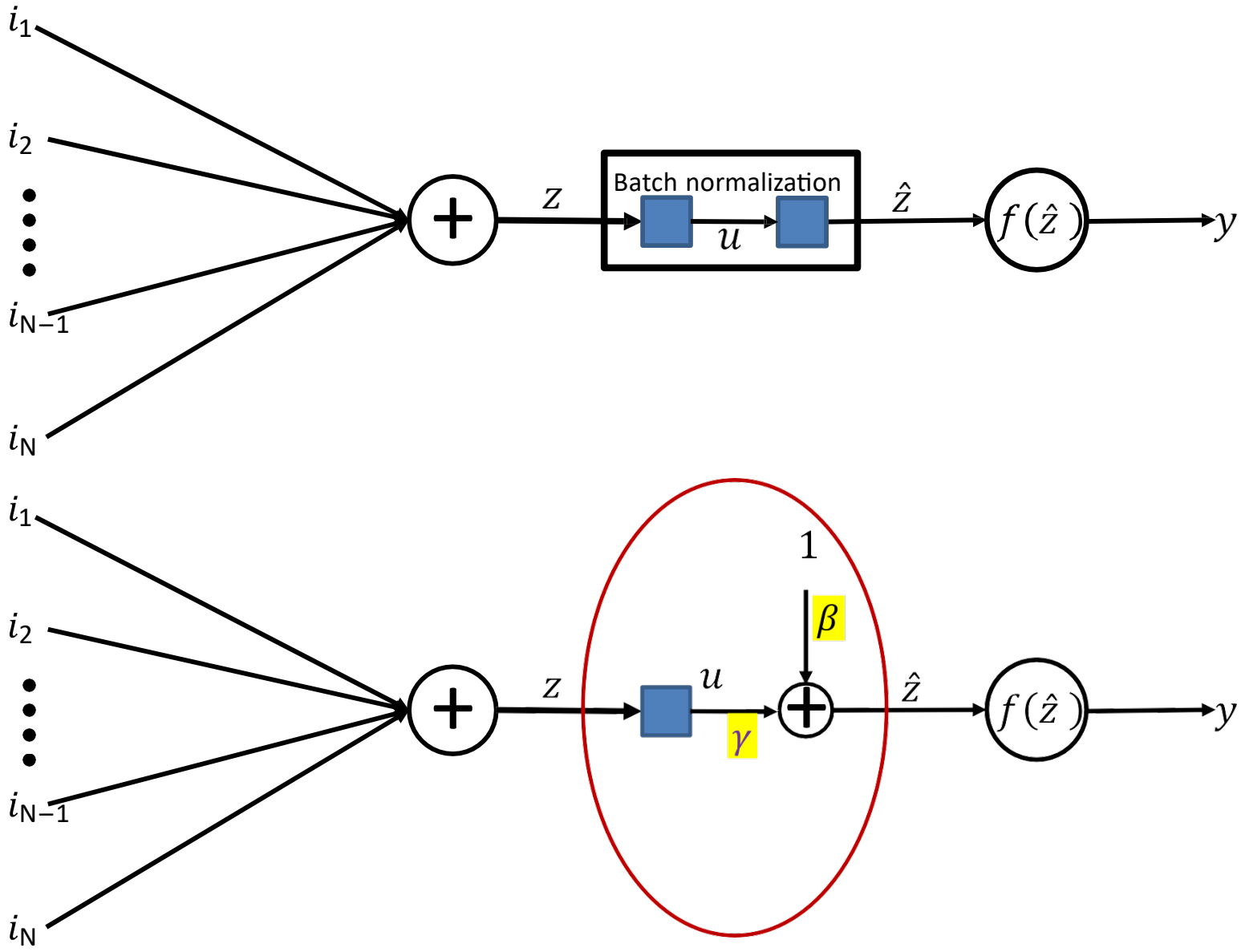
$\sigma_B^2 - \frac{1}{B}\sum_{i=1}^{B}(z_i - \mu_B)^2$

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

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

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

A better picture for batch norm



A note on derivatives: Usual

- **In conventional training:**

- The minibatch loss is the average of the divergence between the actual and desired outputs of the network for all inputs in the minibatch

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

- The derivative of the minibatch loss w.r.t. network parameters is the average of the derivatives of the divergences for the *individual* training instances w.r.t. parameters

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

- The output of the network in response to an input, and the derivative of the divergence for any input are **independent** of other inputs in the minibatch

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

A note on derivatives: BatchNorm

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

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

- The Divergence for each Y_t depends on *all* the X_t within the minibatch
  - **Training instances within the minibatch are no longer independent**

The actual divergence with BN

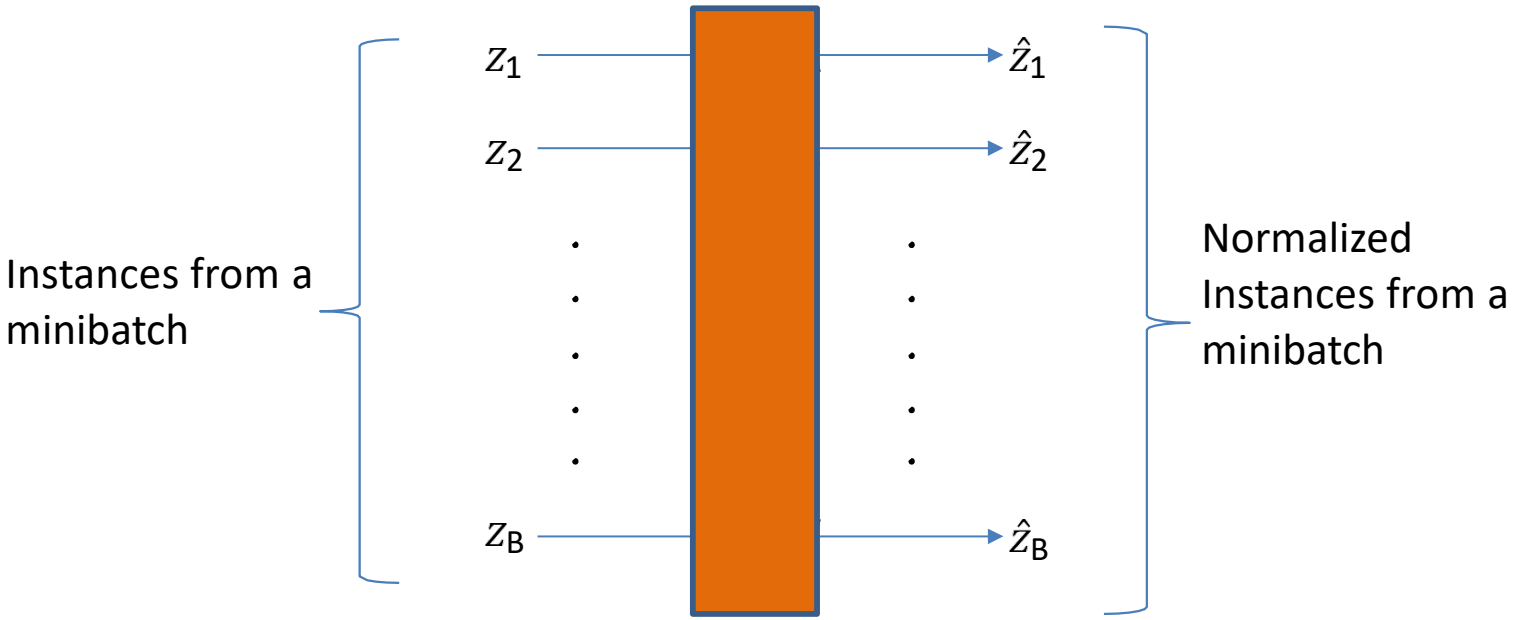
- The actual divergence for any minibatch with terms explicity written

$$Loss(minibatch)$$

$$= \frac{1}{B}\sum_{t} Div\left(Y_t\left(X_t, \mu_B(X_t, X_{t'\neq t}), \sigma_B^2\left(X_t, X_{t'\neq t}, \mu_B(X_t, X_{t'\neq t})\right)\right), d_t(X_t)\right)$$

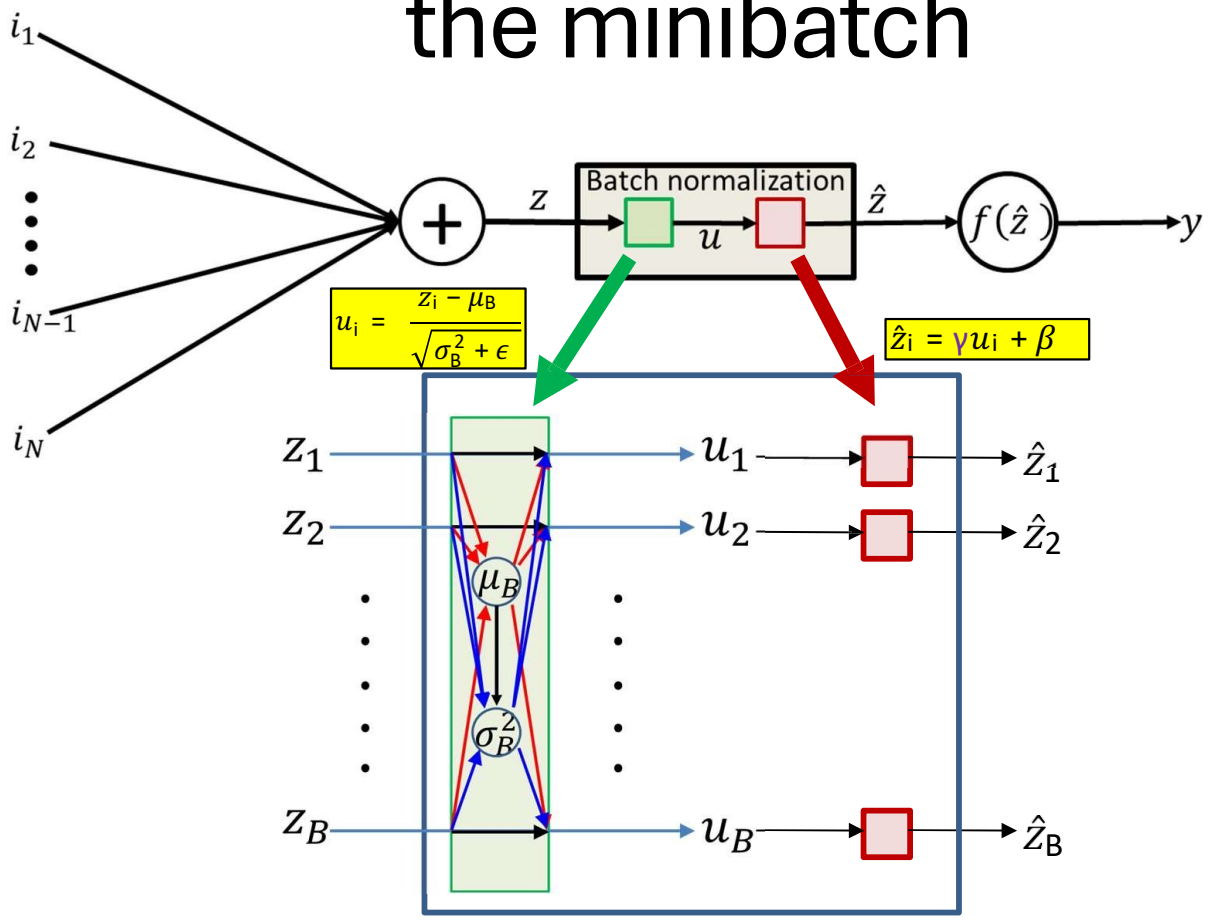
- We need the derivative for this function
- To derive the derivative lets consider the dependencies at a *single* neuron

Batchnorm is a vector function over the minibatch



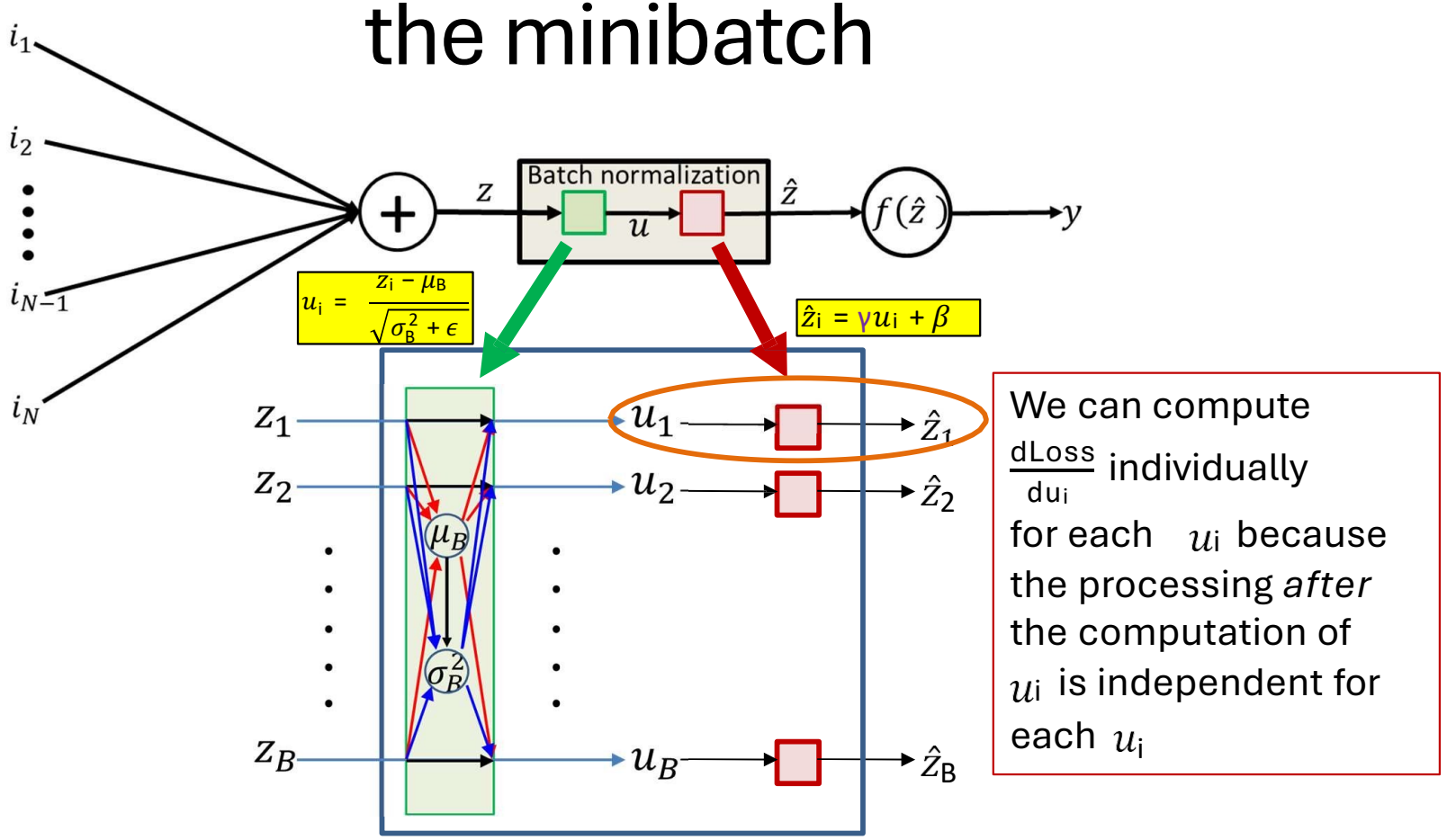
- Batch normalization is really a *vector* function applied over all the inputs from a minibatch
  - Every z_i affects every z_j
- To compute the derivative of the minibatch loss w.r.t any z_i, we must consider all $\hat{z}_j$ s in the batch

Batchnorm is a vector function over the minibatch



- The computation of mini-batch normalized u's is a vector function
  - Invoking mean and variance statistics across the minibatch
- The subsequent shift and scaling is individually applied to each u to compute the corresponding $\hat{z}$

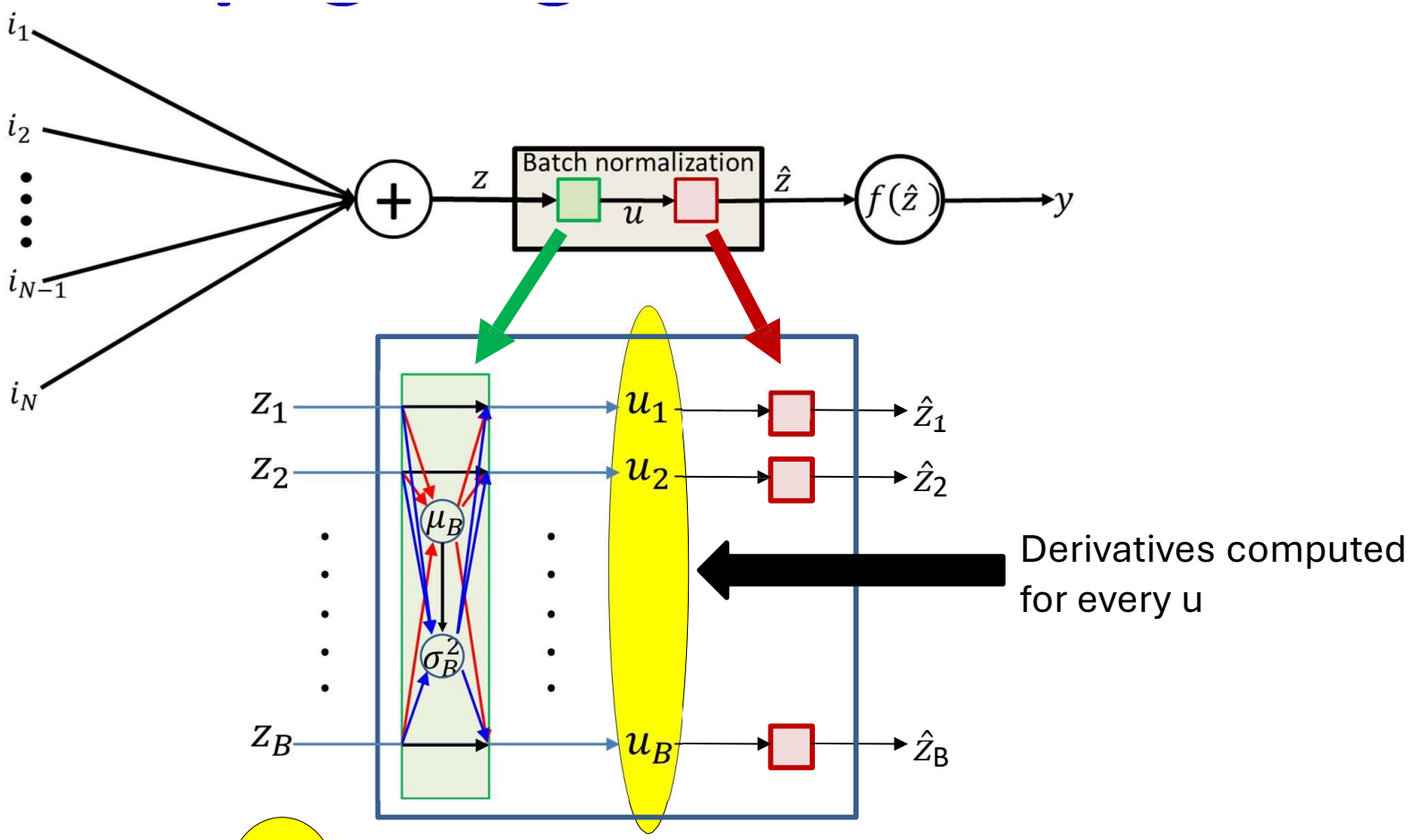
Batchnorm is a vector function over the minibatch



We can compute $\frac{dLoss}{du_i}$ individually for each u_i because the processing *after* the computation of u_i is independent for each u_i

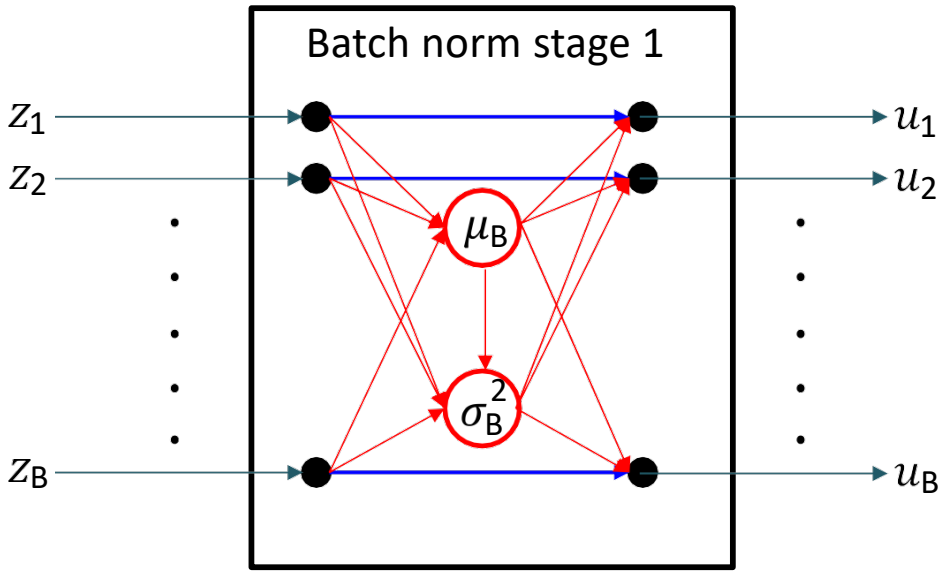
- The computation of mini-batch normalized u's is a vector function
  - Invoking mean and variance statistics across the minibatch
- The subsequent shift and scaling is individually applied to each u to compute the corresponding $\hat{z}$

Propogating the derivative



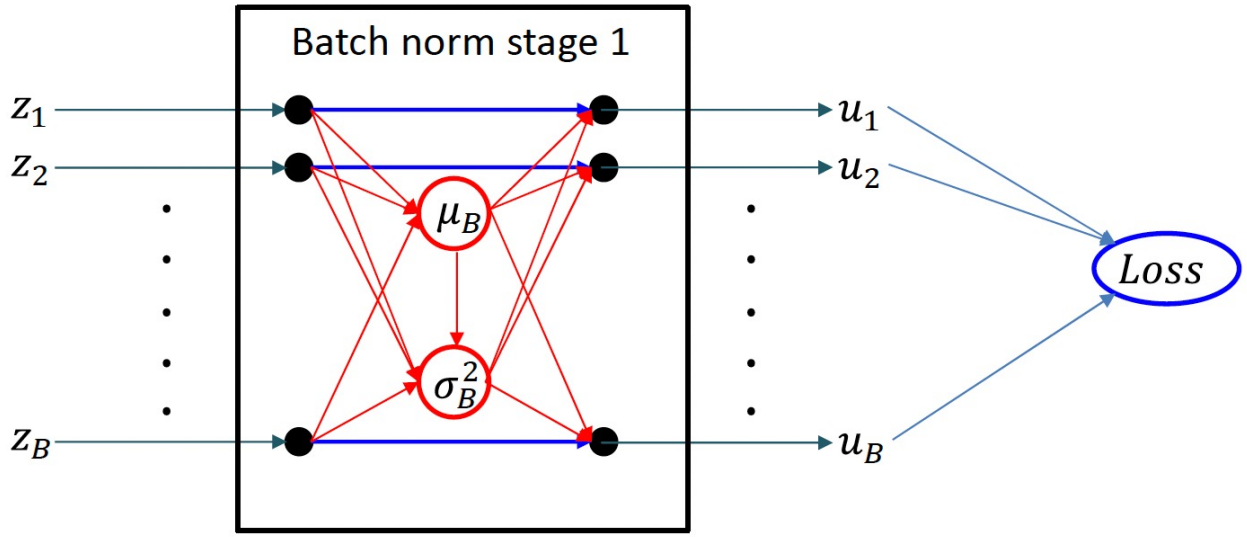
- Suppose we have $\frac{dLoss}{du_i}$ for every u_i
- We must propagate the derivative through the first stage of BN
  - Which is a vector operation over the minibatch

The first stage of batchnorm



- The complete dependency figure for the first “normalization” stage of Batchnorm
- Note : inputs and outputs are different *instances* in a minibatch
  - The diagram represents BN occurring at a *single neuron*

The first stage of Batchnorm



- The complete derivative of the mini-batch loss w.r.t. z_i

$$\frac{dLoss}{dz_i} = \sum_j \frac{dLoss}{du_j} \boxed{\frac{du_j}{dz_i}}$$

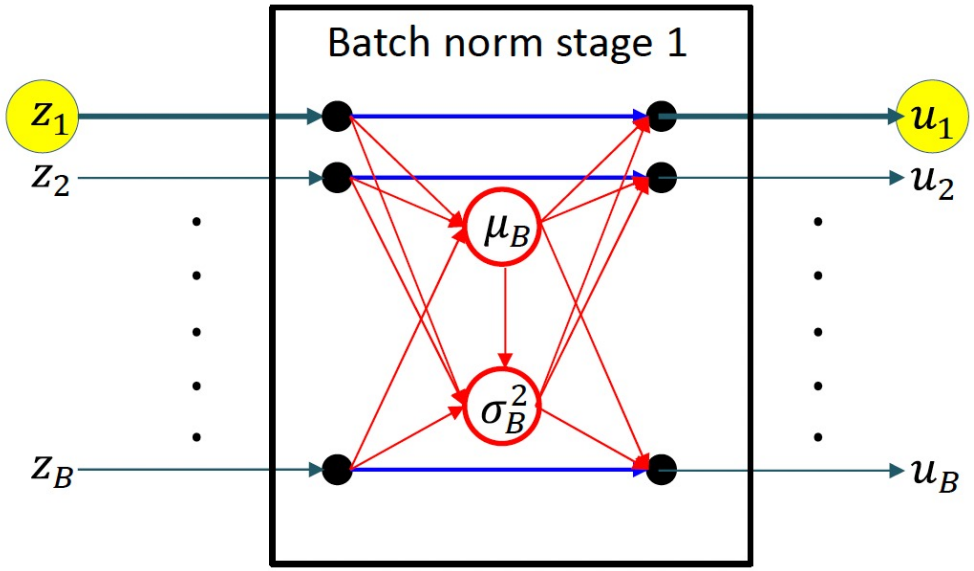
Must compute for every i,j pair

The first stage of Batchnorm

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

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

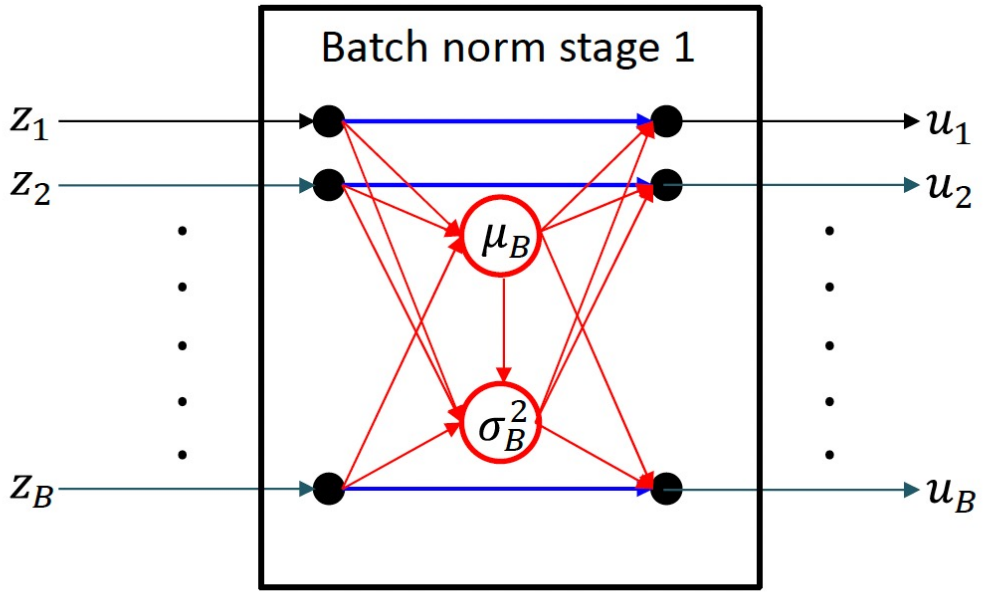
$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$



- The derivative for the "through" line ($i = j$)

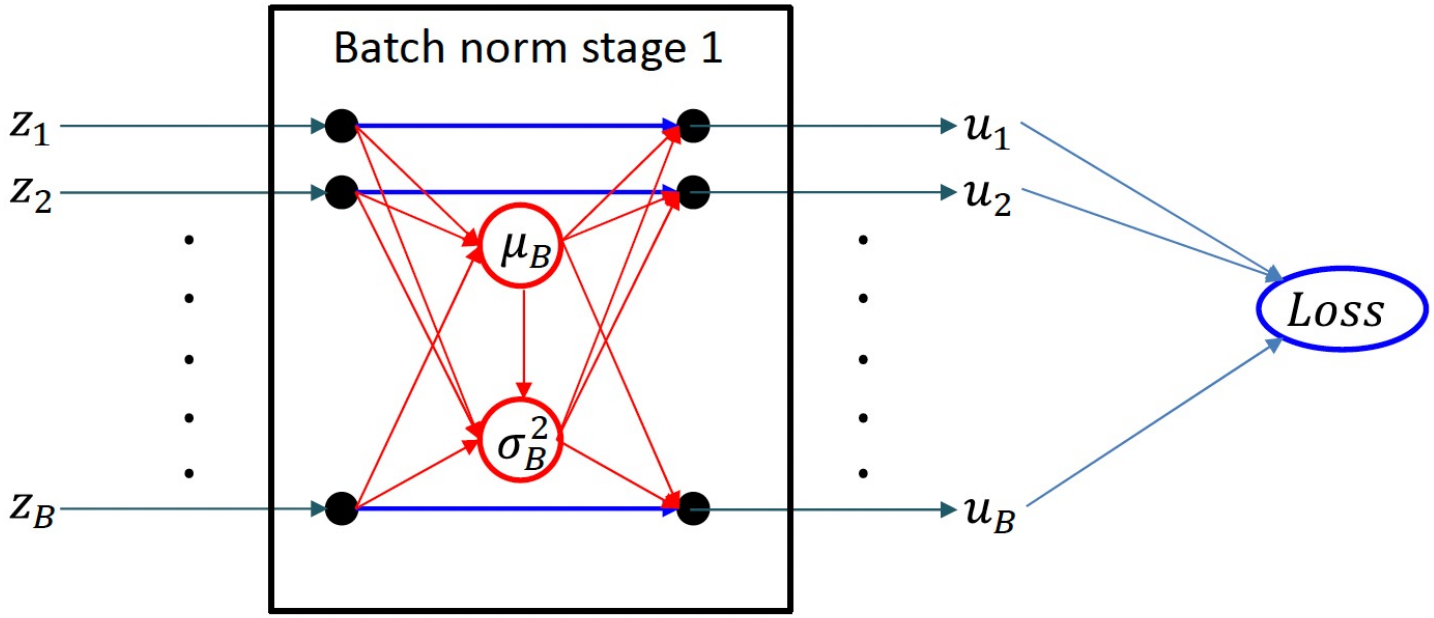
$$\frac{du_i}{dz_i} = \frac{\partial u_i}{\partial z_i} + \frac{\partial u_i}{\partial \mu_B}\frac{\partial \mu_B}{\partial z_i} + \frac{\partial u_i}{\partial \sigma_B^2}\frac{d\sigma_B^2}{dz_i}$$

The first stage of Batchnorm



$$\frac{du_j}{dz_i} = \begin{cases} \dfrac{1}{\sqrt{\sigma_B^2+\epsilon}} + \dfrac{-1}{B\sqrt{\sigma_B^2+\epsilon}} + \dfrac{-(z_i-\mu_B)^2}{B(\sigma_B^2+\epsilon)^{3/2}} & \text{if } j = i \\ \dfrac{-1}{B\sqrt{\sigma_B^2+\epsilon}} + \dfrac{-(z_i-\mu_B)(z_j-\mu_B)}{B(\sigma_B^2+\epsilon)^{3/2}} & \text{if } j \neq i \end{cases}$$

The first stage of Batchnorm



- The complete derivative of the mini-batch loss w.r.t. z_i

$$\frac{dLoss}{dz_i} = \sum_j \frac{dLoss}{du_j}\frac{du_j}{dz_i}$$

The first stage of Batchnorm

$$\frac{dLoss}{dz_i} = \sum_j \frac{dLoss}{du_j}\frac{du_j}{dz_i}$$

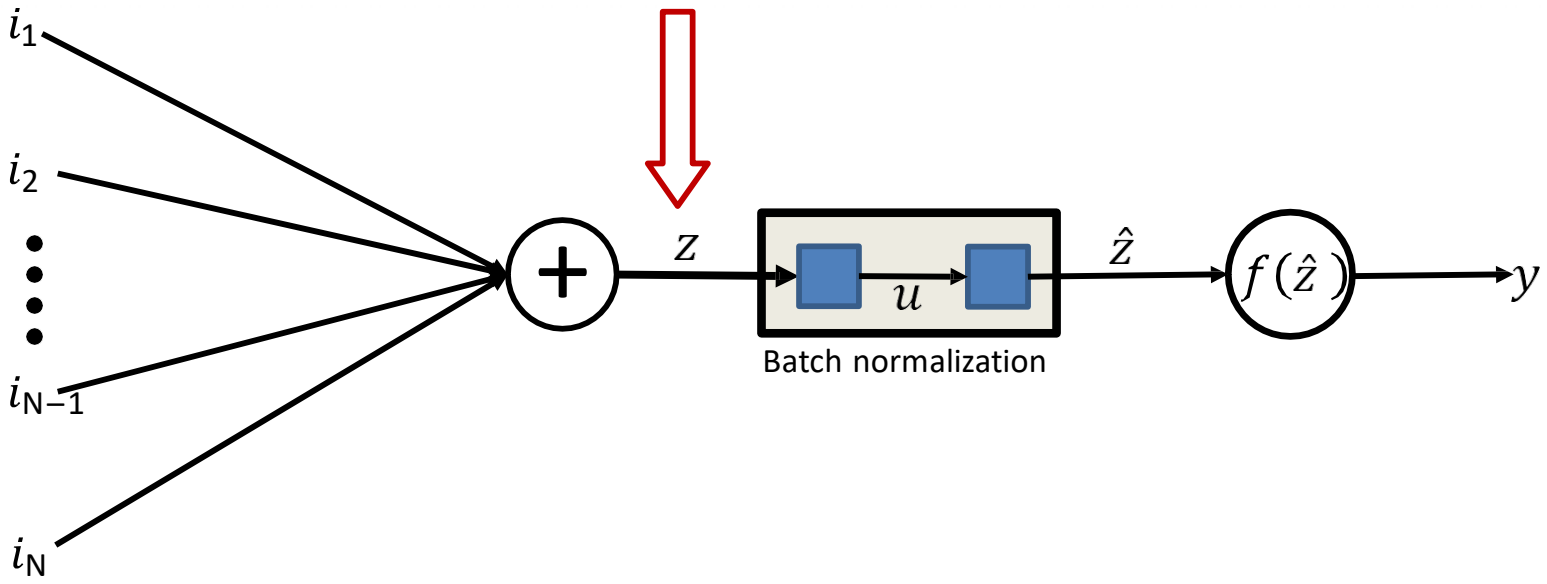
$$\frac{du_j}{dz_i} = \begin{cases} \frac{1}{\sqrt{\sigma_B^2+\epsilon}} + \frac{-1}{B\sqrt{\sigma_B^2+\epsilon}} + \frac{-(z_i-\mu_B)^2}{B(\sigma_B^2+\epsilon)^{3/2}} & \text{if } j = i \\ \frac{-1}{B\sqrt{\sigma_B^2+\epsilon}} + \frac{-(z_i-\mu_B)(z_j-\mu_B)}{B(\sigma_B^2+\epsilon)^{3/2}} & \text{if } j \neq i \end{cases}$$

- The complete derivative of the mini-batch loss w.r.t. z_i

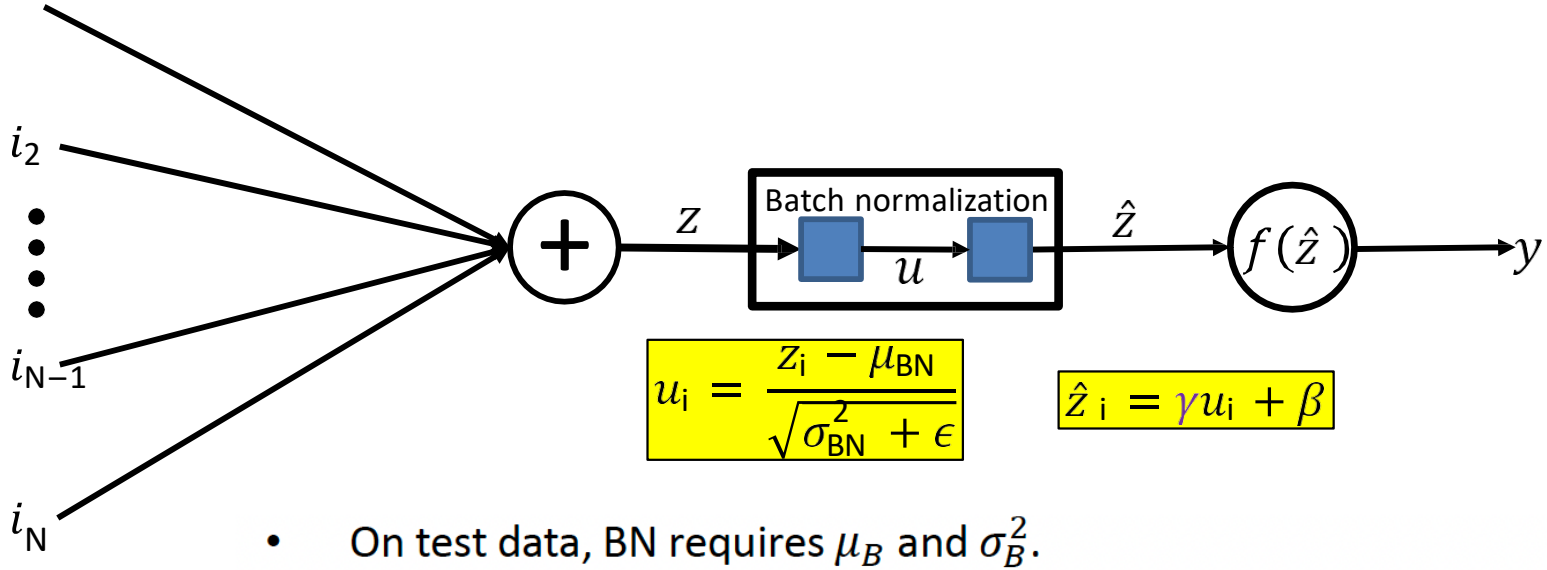
$$\frac{dLoss}{dz_i} = \frac{1}{\sqrt{\sigma_B^2+\epsilon}}\frac{dLoss}{du_i} - \frac{1}{B\sqrt{\sigma_B^2+\epsilon}}\sum_j \frac{dLoss}{du_j} - \frac{(z_i-\mu_B)}{B(\sigma_B^2+\epsilon)^{3/2}}\sum_j \frac{dLoss}{du_j}(z_j-\mu_B)$$

Batch normalization: Backpropagation

$$\frac{dLoss}{dz_i} = \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} \frac{dLoss}{du_i} - \frac{1}{B\sqrt{\sigma_B^2 + \epsilon}} \sum_j \frac{dLoss}{du_j} - \frac{(z_i - \mu_B)}{B(\sigma_B^2 + \epsilon)^{3/2}} \sum_j \frac{dLoss}{du_j} (z_j - \mu_B)$$



Batch normalization: Inference



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

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

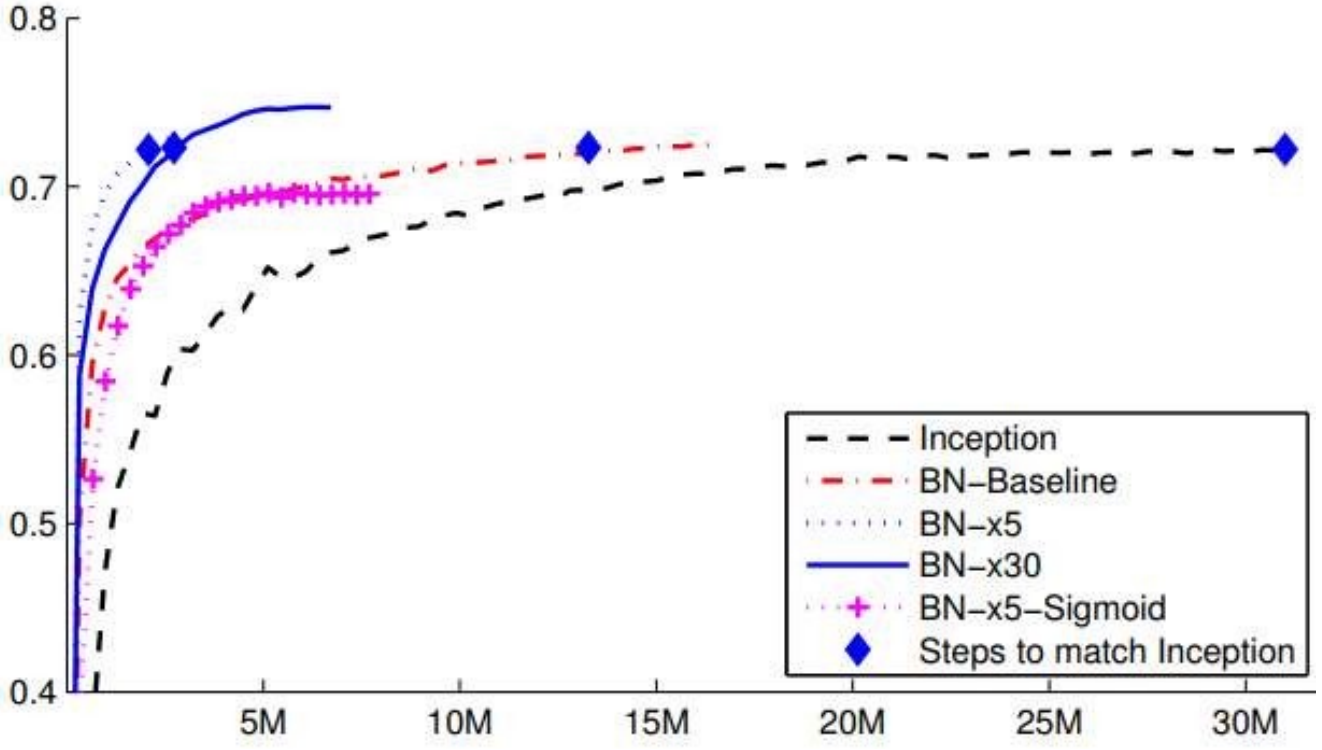
$$\sigma_{BN}^2 = \frac{B}{(B-1)Nbatches} \sum_{batc} \sigma_B^2(batch)$$

- 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

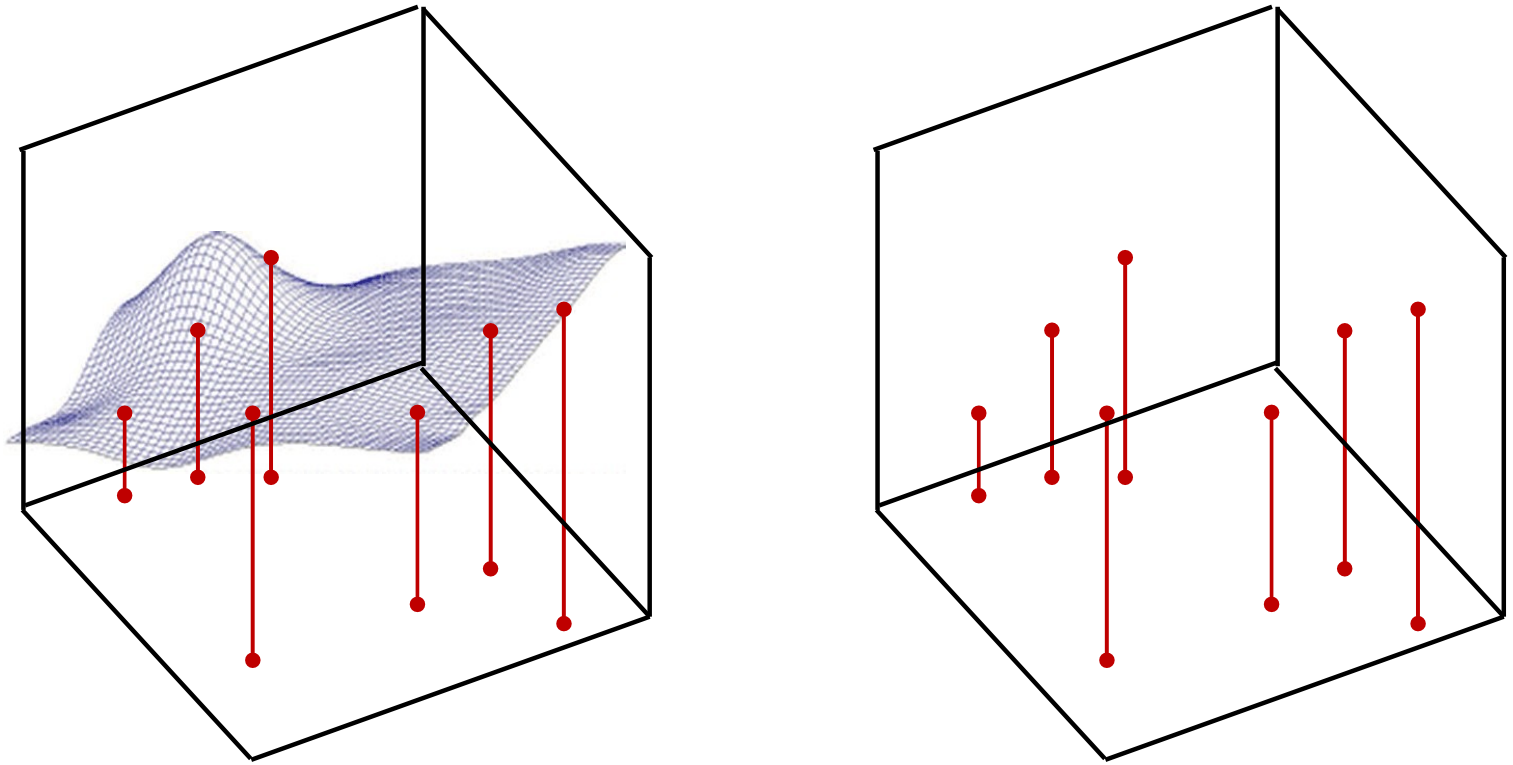
- Batch normalization may only be applied to **some layers**
  - Or even only selected neurons in the layer
- Improves both convergence rate and neural network performance
  - To get maximum benefit from BN, learning rates must be increased and learning rate decay can be faster
    - Since the data generally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

Batch Normalization: Typical result



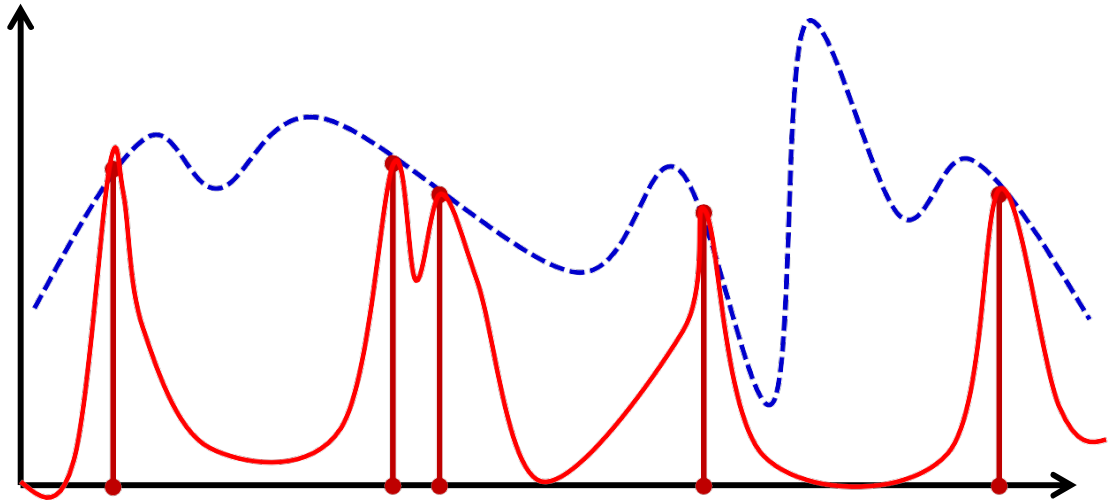
- Performance on Imagenet, from Ioffe and Szegedy, JMLR 2015

Learning the network



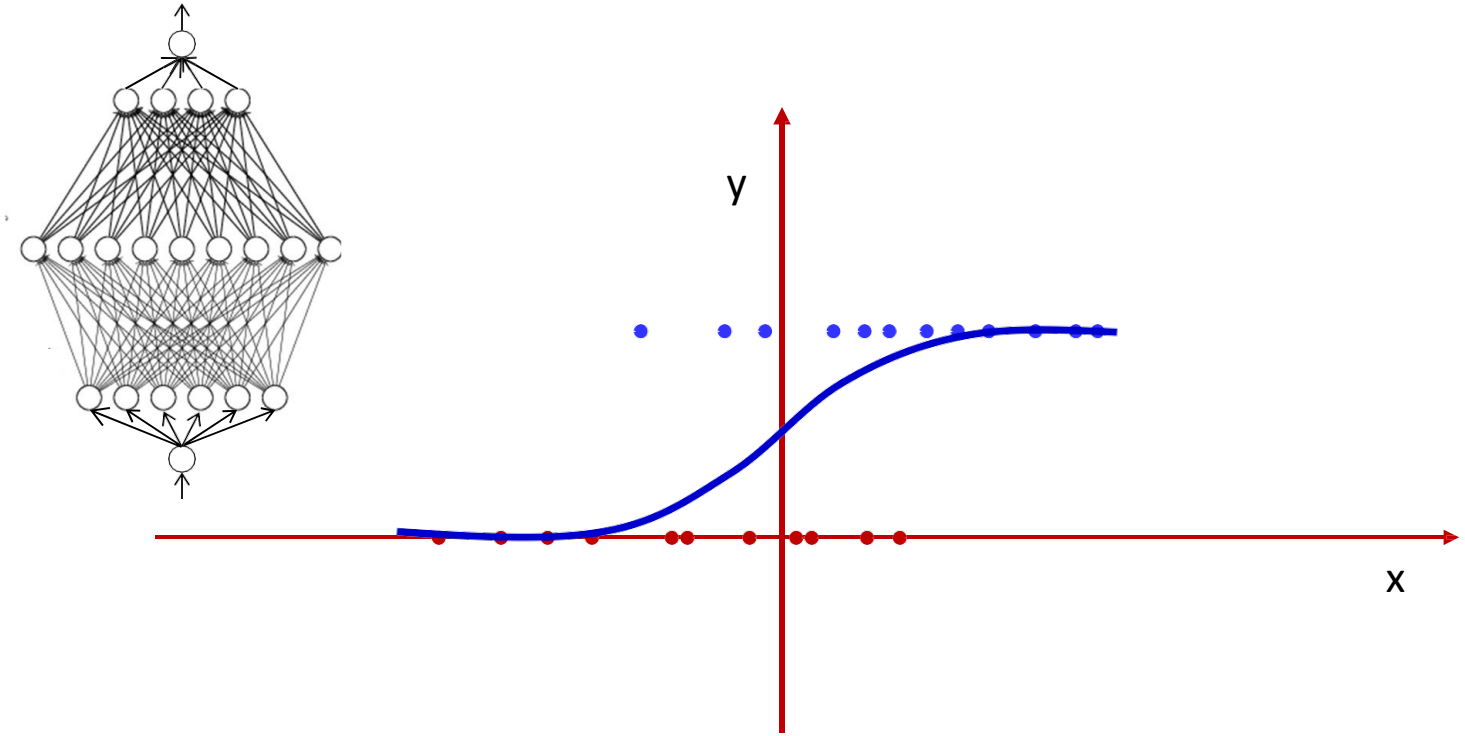
- We attempt to learn an entire function from just a few *snapshots* of it

Overfitting



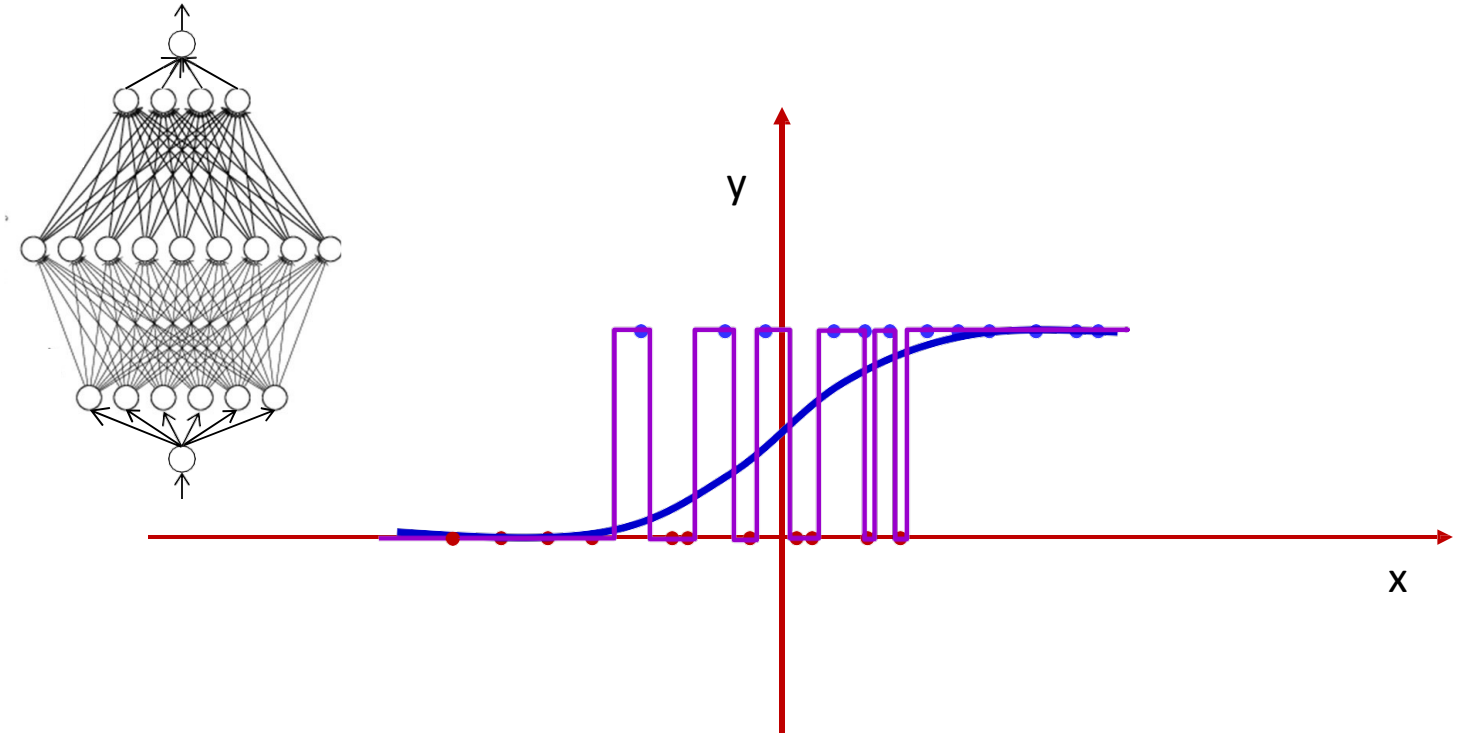
- Problem: Network may just learn the values at the inputs
  - Learn the red curve instead of the dotted blue one
    - Given only the red vertical bars as inputs
- Need additional "**smoothing**" constraints that will "fill in" the missing regions acceptably
  - Generalization

Smoothness through weight manipulation



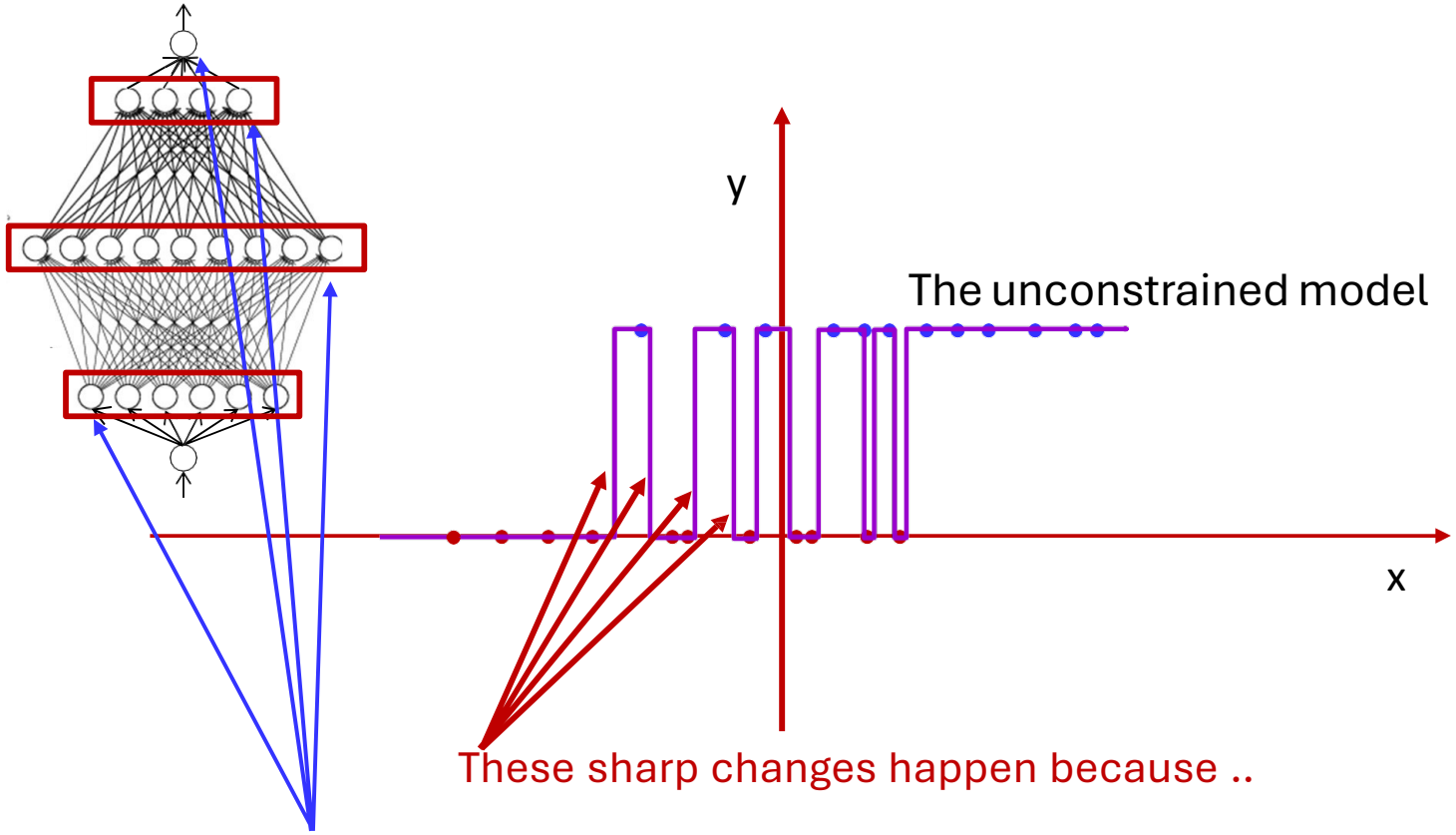
- Illustrative example: Simple binary classifier
  - The “desired” output is generally smooth
    - Capture statistical or average trends

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 – try to fit every point

Why overfitting

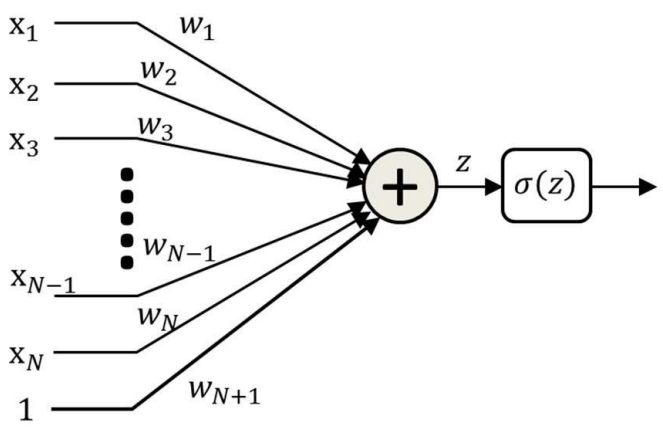


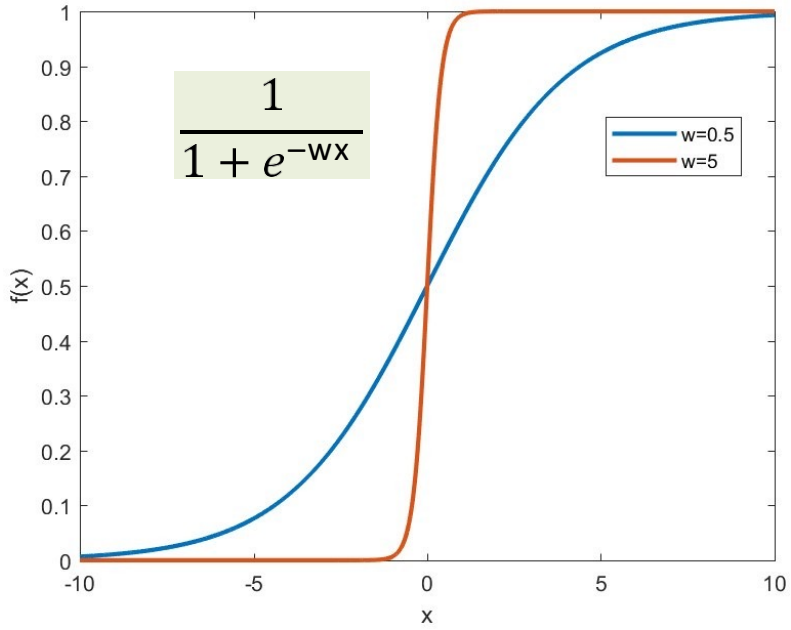
These sharp changes happen because ..

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

They hence will adjust their parameters to fit every individual data point perfectly

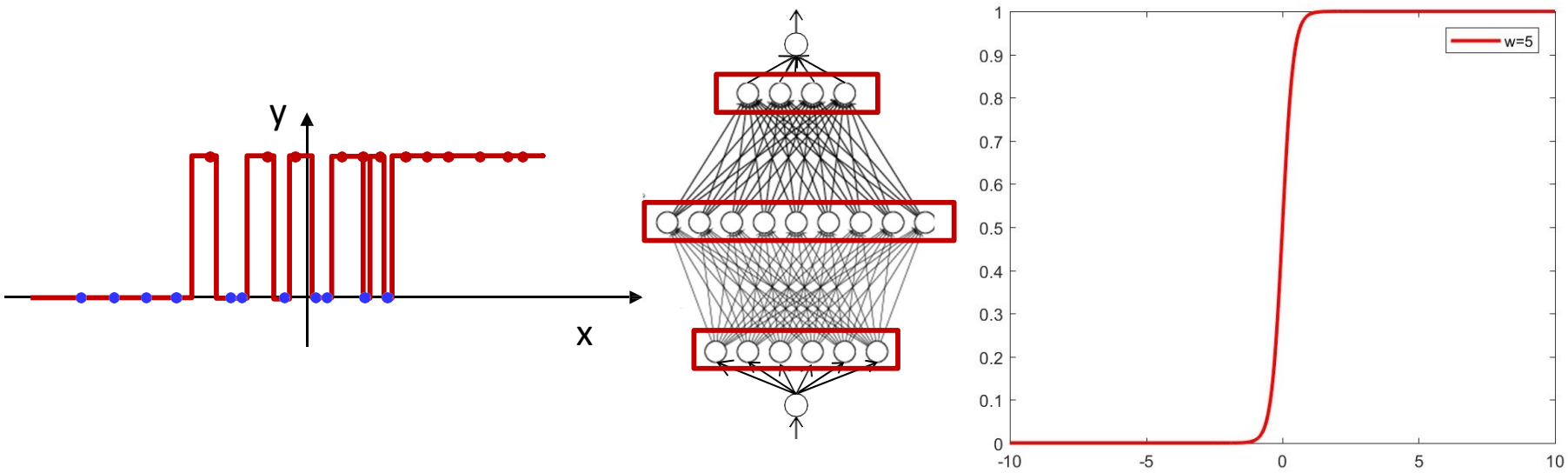
The individual perceptron





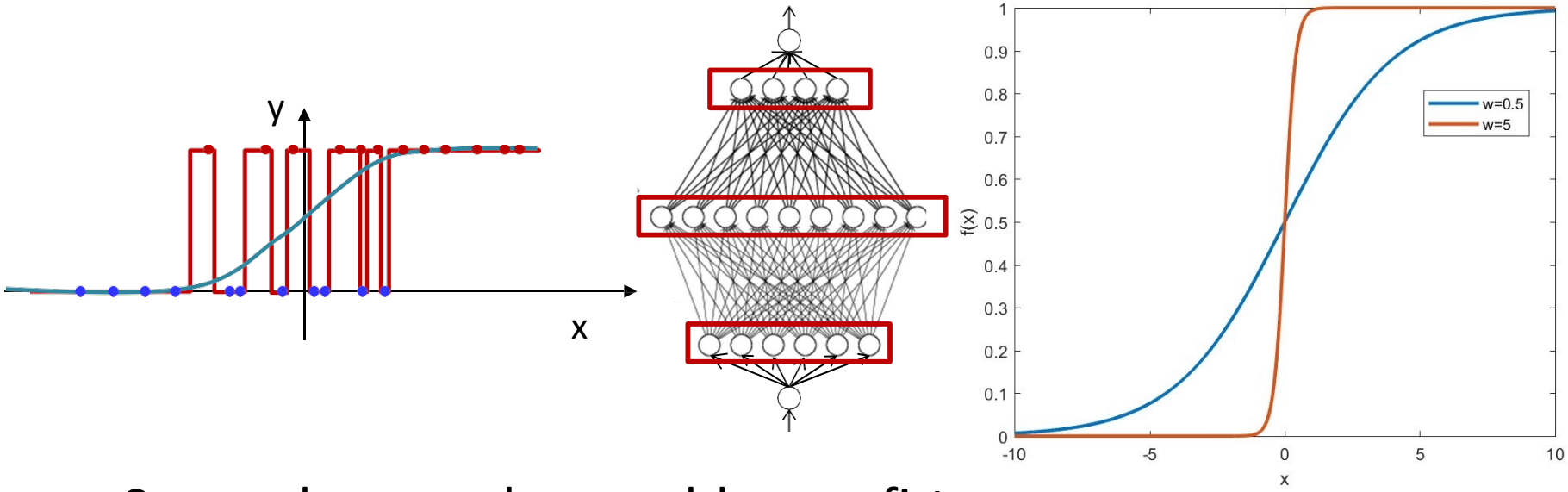
- Using a sigmoid activation
  - As $|w|$ increases, the response becomes steeper

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

Smoothness through weight constraints

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

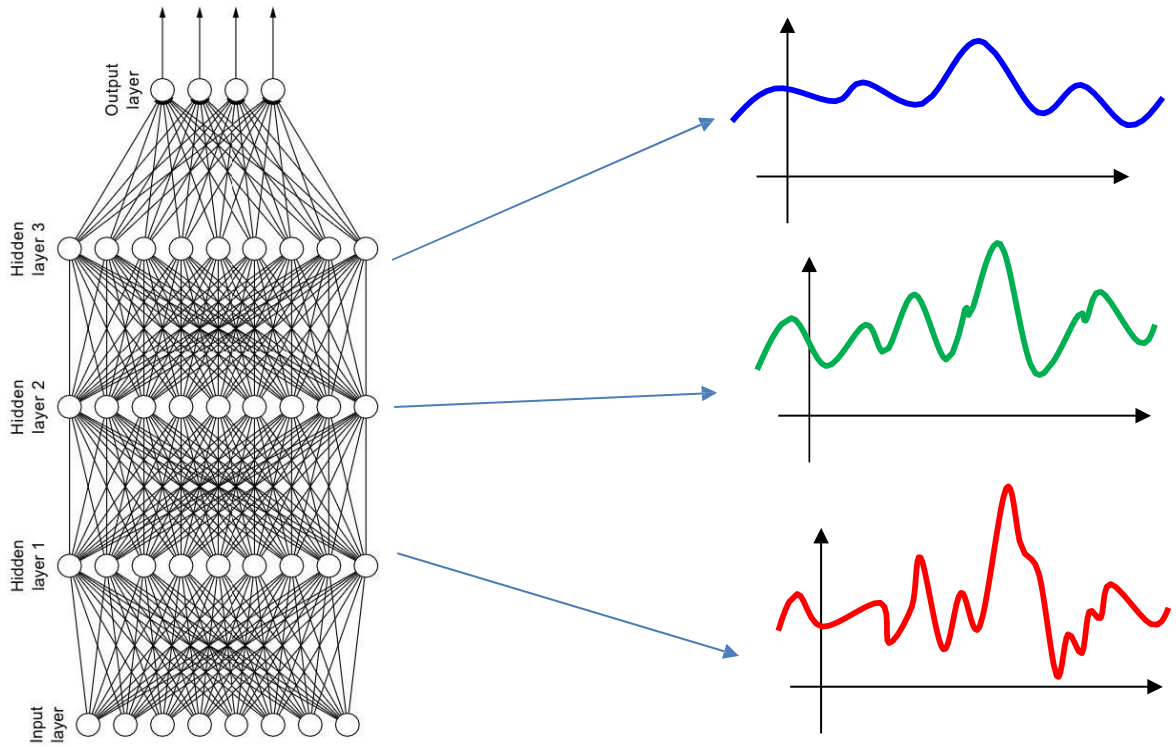
$$L(W_1, W_2, \ldots, W_K) = \frac{1}{T}\sum_t Div(Y_t, d_t; W_1, W_2, \ldots, W_K) + \frac{1}{2}\lambda \sum_k \|W_k\|_F^2$$

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

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

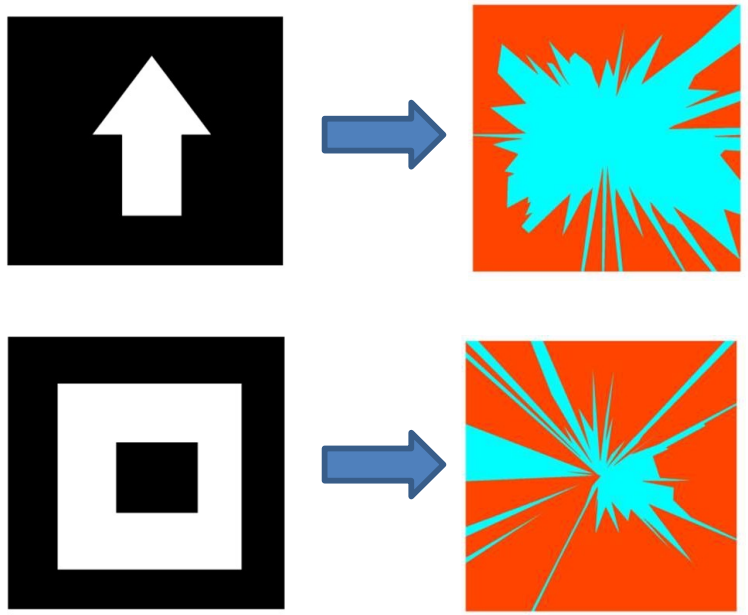
  - Increasing λ assigns greater importance to shrinking the weights
    - Make greater error on training data, to obtain a more acceptable network

Smoothness through network structure



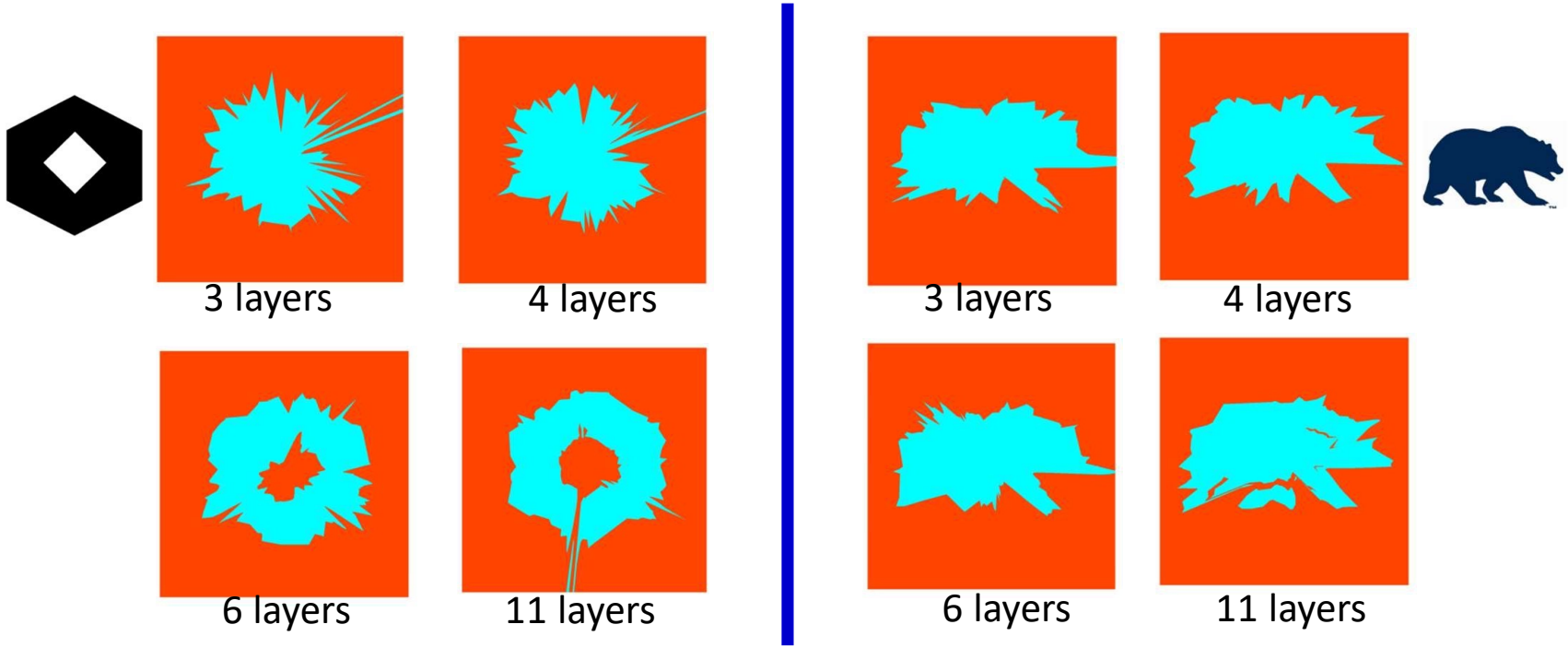
- Smoothness constraints can also be imposed through the network *structure*
- For a given number of parameters, deeper networks impose more smoothness than shallow ones
  - Each layer works on the already smooth surface output by the previous layer

Minimal correct architectures are hard to train



- Typical results (varies with initialization)
- Requires more training samples to converge– orders of magnitude more than you usually get

But depth and training data help



- Deeper networks seem to learn better, for the same number of total neurons

- Training with more data is also better

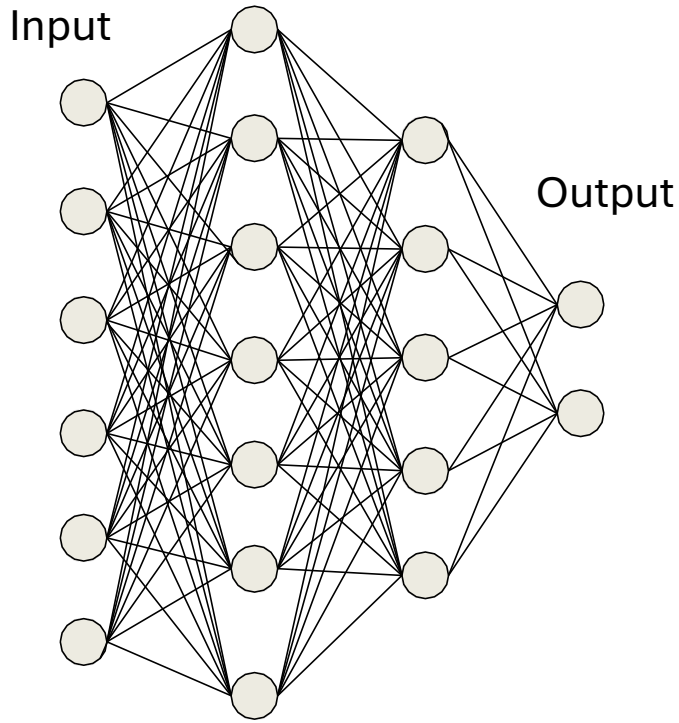
10000 training instances

Regularization..

- Other techniques have been proposed to improve the smoothness of the learned function
  - L_1 regularization of network activations
  - Regularizing with added noise..

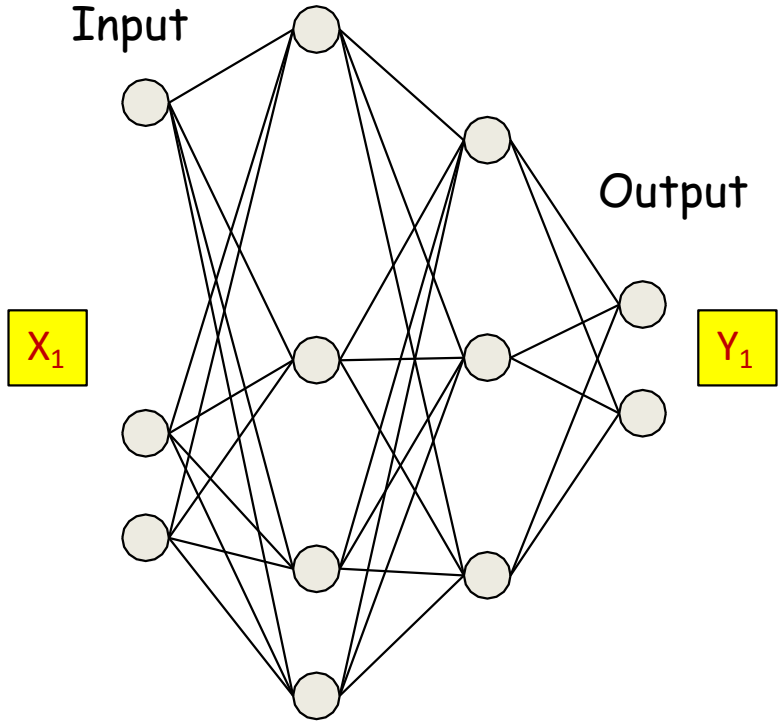
- Possibly the most influential method has been “dropout”

Dropout



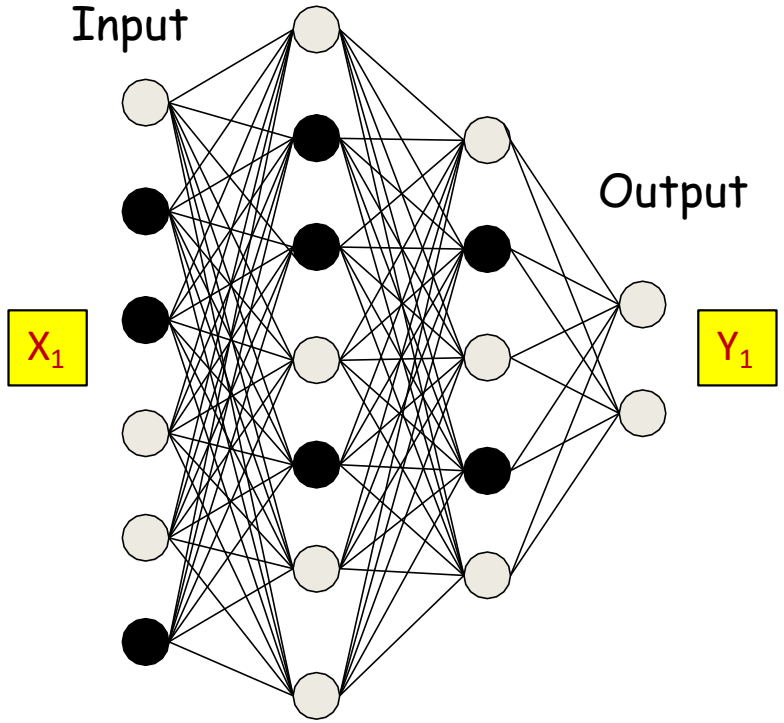
- **During training:** For each input, at each iteration, “turn off” each neuron with a probability 1-a

Dropout



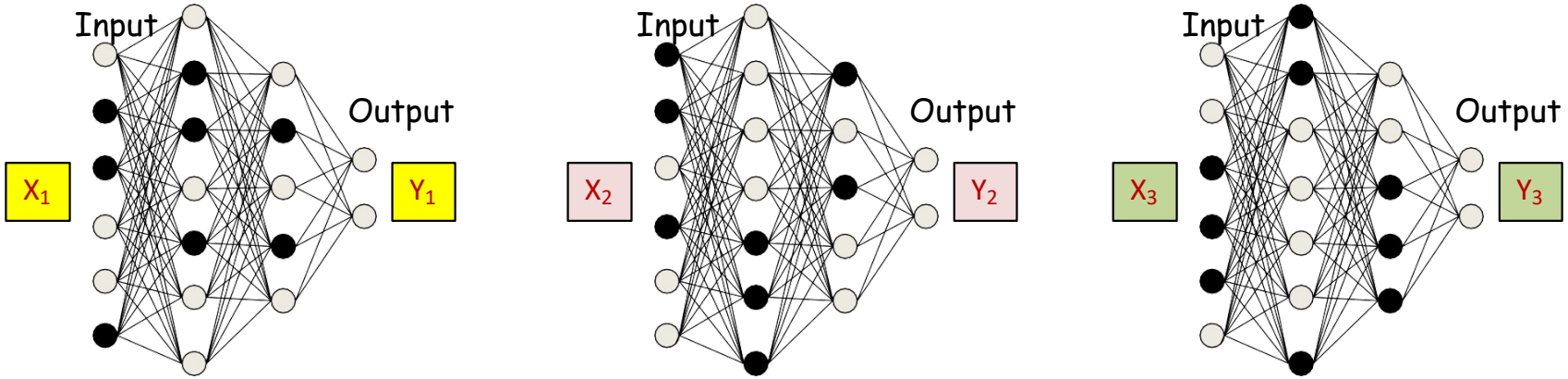
- **During training:** For each input, at each iteration, "turn off" each neuron with a probability 1-a
  - Also turn off inputs similarly

Dropout



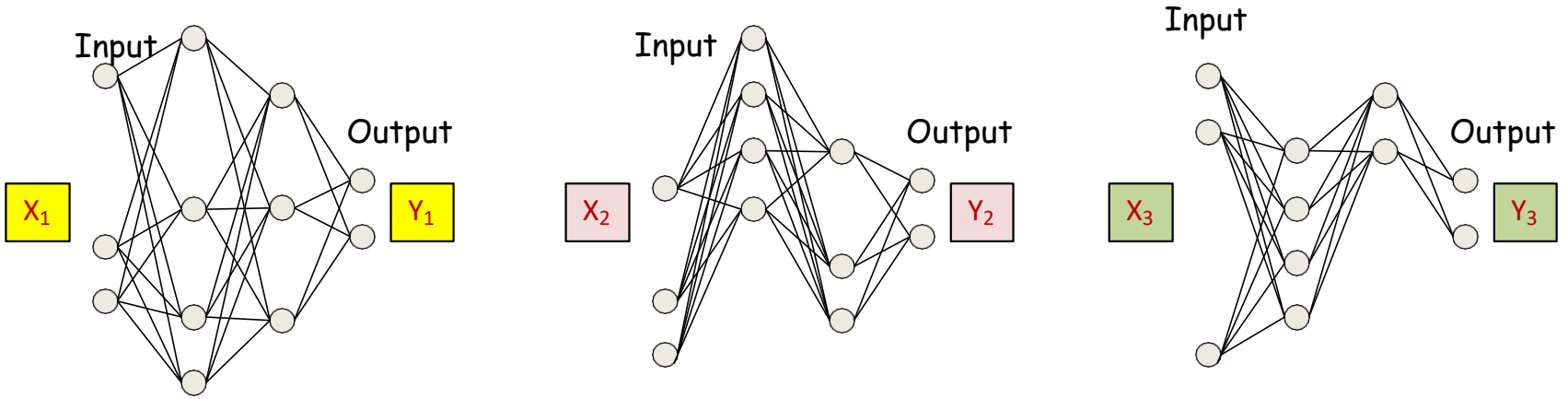
- **During training:** For each input, at each iteration, "turn off" each neuron (including inputs) with a probability 1-a
  - In practice, set them to 0 according to the failure of a Bernoulli random number generator with success probability a

Dropout



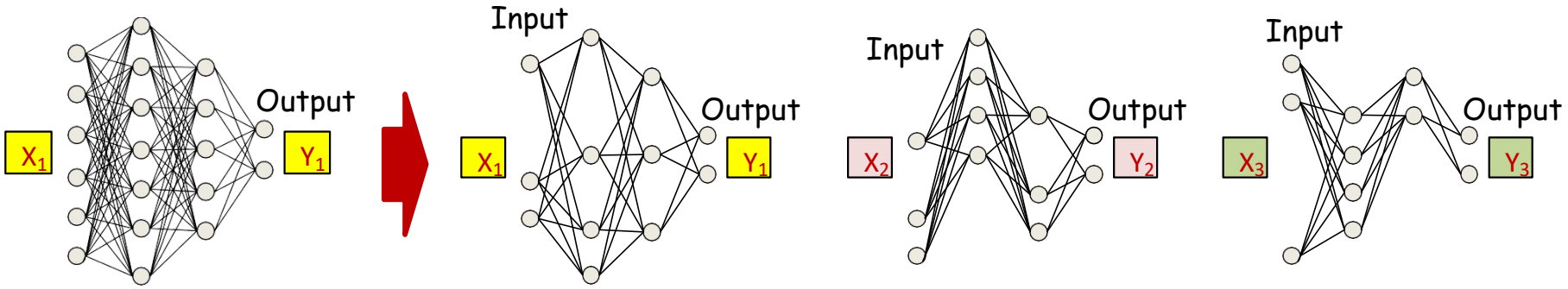
The pattern of dropped nodes changes for *each input i.e.* in every pass through the net

Dropout



- **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

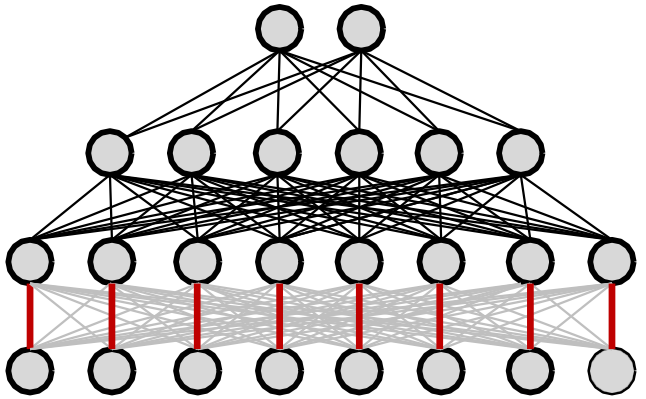
Statistical Interpretation



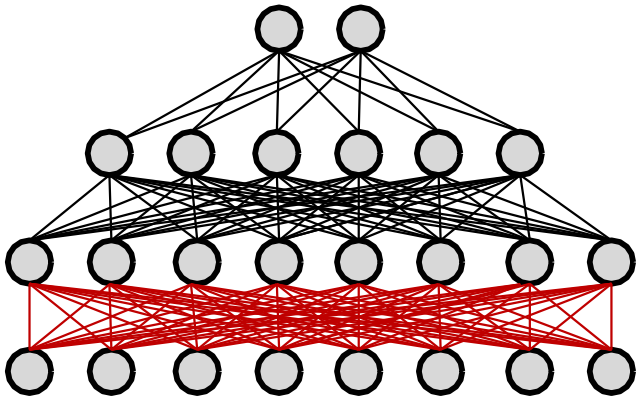
- For a network with a total of N neurons, there are 2^N possible sub-networks
  - Obtained by choosing different subsets of nodes
  - Dropout samples over all 2^N possible networks
  - Effectively learns a network that averages over all possible networks

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 for Training

- 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 $k = 1 \ldots N$

  # Mask takes value 1 with prob. α, 0 with prob $1 - \alpha$
  - $mask(k-1, j) = Bernoulli(\alpha)\ ,\ j = 1 \ldots D_{k-1}$
  - $y_j^{(k-1)} = y_j^{(k-1)}.mask(k-1, j),\ j = 1 \ldots D_{k-1}$
  - For $j = 1 \ldots D_k$
    - $z_j^{(k)} = \sum_{i=0}^{N_k} w_{i,j}^{(k)} y_i^{(k-1)} + b_j^{(k)}$
    - $y_j^{(k)} = f_k\left(z_j^{(k)}\right)$
- 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 $i = 1 \ldots D_k$
    - $\frac{\partial Div}{\partial y_i^{(k)}} = mask(k,i)\sum_j w_{ij}^{(k+1)}\frac{\partial Div}{\partial z_j^{(k+1)}}$
    - $\frac{\partial Div}{\partial z_i^{(k)}} = f_k'\left(z_i^{(k)}\right)\frac{\partial Div}{\partial y_i^{(k)}}$
    - $\frac{\partial Div}{\partial w_{ij}^{(k+1)}} = y_i^{(k)}\frac{\partial Div}{\partial z_j^{(k+1)}}$ for $j = 1 \ldots D_{k+1}$

Testing with Dropout

- Dropout effectively trains 2^N networks
- On test data the "Bagged" output, in principle, is the ensemble average over all 2^N networks and is thus the statistical expectation of the output over all networks

$$Y = E\left[network\left(y_j^{(k)}, j = 1 \ldots D_k, k = 1 \ldots K\right)\right]$$

  - Explicitly showing the network as a function of the outputs of individual neurons in the net

- We cannot explicitly compute this expectation

- Instead, we will use the following approximation

$$E\left[network\left(y_j^{(k)}, \forall k, j\right)\right] = network\left(E[y_j^{(k)}]\forall k, j\right)$$

  - Where $E[y_j^{(k)}]$ is the expected output of the jth neuron in the kth layer over all networks in the ensemble
  - I.e. approximate the expectation of a function as the function of expectations

- We require $E[y_j^{(k)}]$ to compute this

What each neuron computes

- Each neuron actually has the following activation:

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

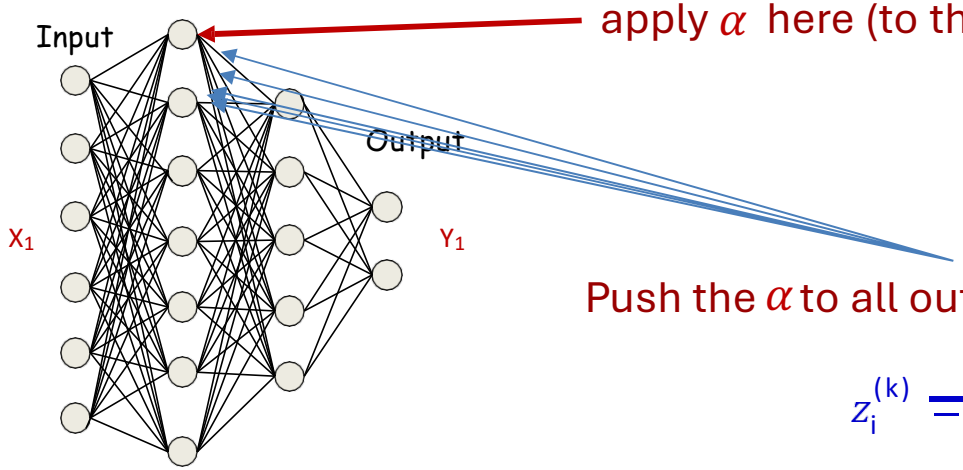
  - Where D is a Bernoulli variable that takes a value 1 with probability α

- D may be switched on or off for individual sub networks, but over the ensemble, the *expected output* of the neuron is

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

- During *test* time, we will use the *expected* output of the neuron
  - 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

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

$$z_i^{(k)} = \sum_j w_{ji}^{(k)}\, y_j^{(k-1)} + b_i^{(k)}$$

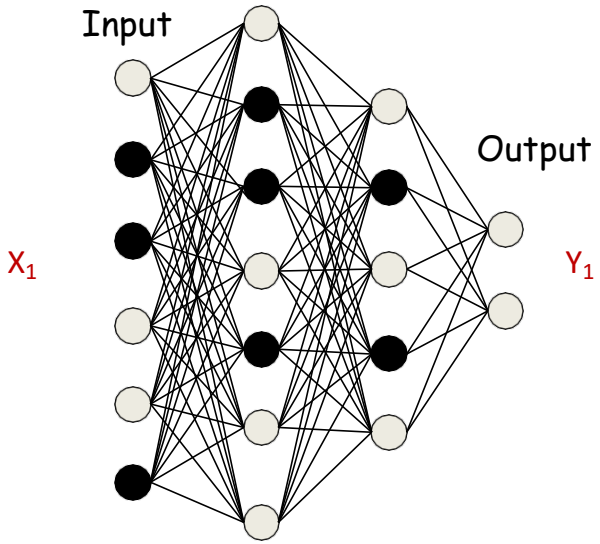
$$= \sum_j w_{ji}^{(k)}\, \alpha\, \sigma\left(z_j^{(k-1)}\right) + b_i^{(k)}$$

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

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

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

Dropout : alternate implementation

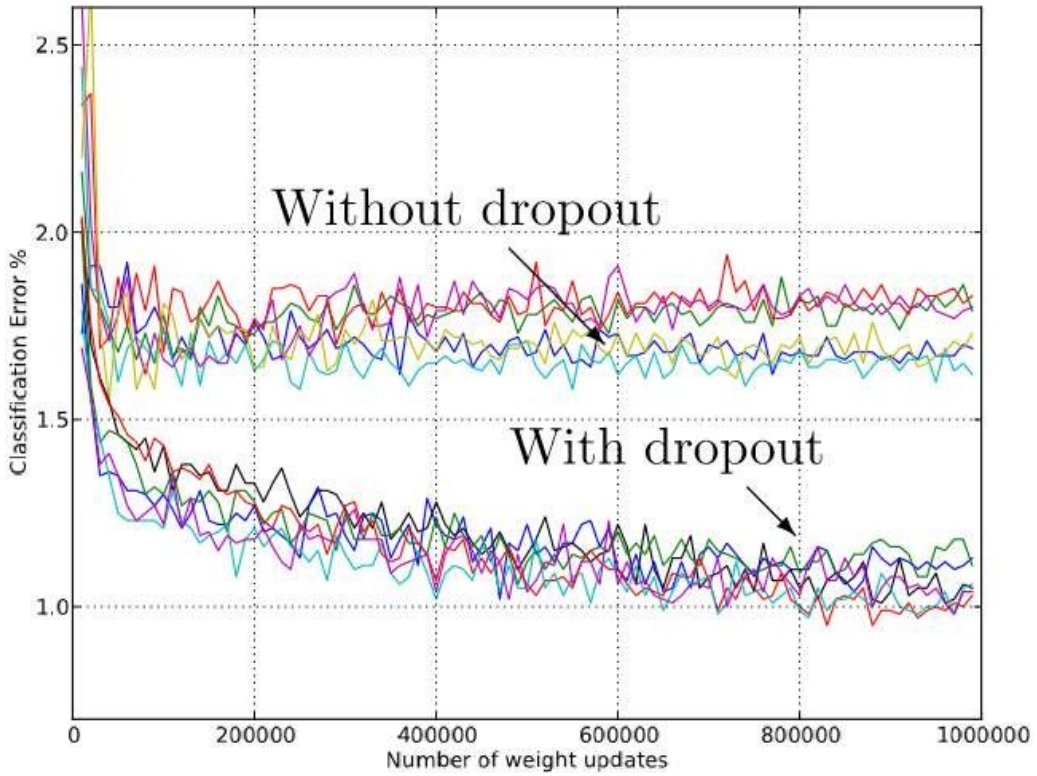


- Alternately, during *training,* replace the activation of all neurons in the network by $\alpha^{-1}\sigma(.)$
  - This does not affect the dropout procedure itself
  - We will use $\sigma(.)$ as the activation during testing, and not modify the weights

Inference with dropout (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
  - $y_j^{(0)} = x_j,\ j = 1 \ldots D; \qquad y_0^{(k=1\ldots N)} = x_0 = 1$
- For layer $k = 1 \ldots N$
  - For $j = 1 \ldots D_k$
    - $z_j^{(k)} = \sum_{i=0}^{N_k} w_{i,j}^{(k)}\, y_i^{(k-1)} + b_j^{(k)}$
    - $y_j^{(k)} = \alpha f_k\left(z_j^{(k)}\right)$
- 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

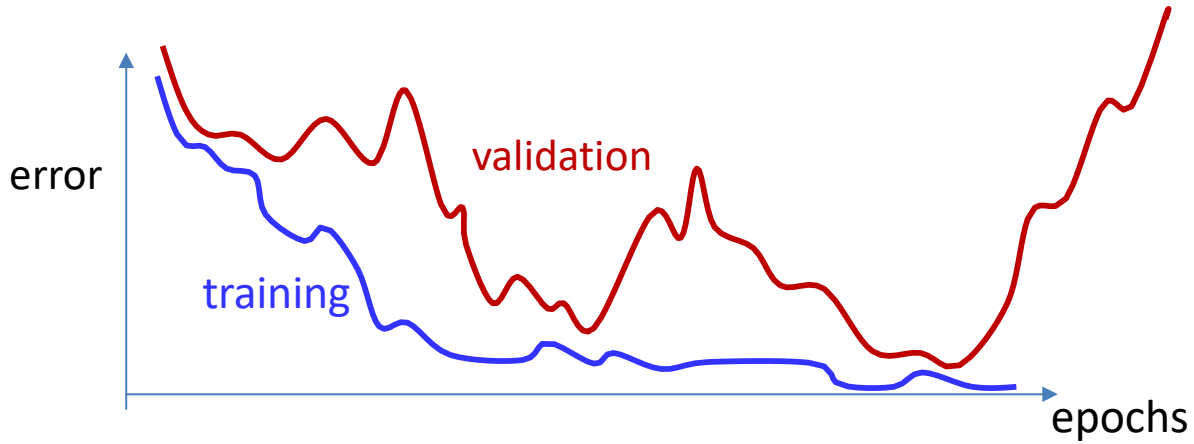
$$|w| \rightarrow \alpha|w| + (1-\alpha)c$$

  - Fix remaining weights to a negative constant

$$w \rightarrow -c$$

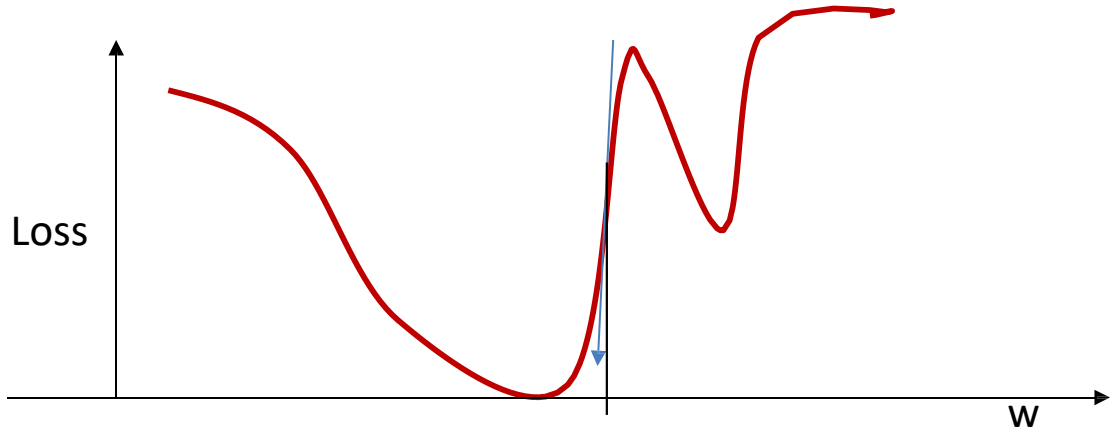
- Whiteout
  - Add or multiply weight-dependent Gaussian noise to the signal on each connection

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

- 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:
  - Normalize 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
    - Xavier, Kaiming, SVD, etc.
    - Key point: neurons with identical connections that are identically initialized will never diverge

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
- Choose the appropriate divergence function
  - Choose regularization
- Choose heuristics (batch norm, dropout, etc.)
- Choose optimization algorithm
  - E.g. ADAM
- Train
  - Evaluate periodically on validation data, for early stopping if required
- Perform a grid search for hyper parameters (learning rate, regularization parameter, …) on held-out data

In closing

- Have outlined the process of training neural networks
  - Some history
  - Forward and backward Propagation
  - Gradient-descent based techniques
  - Convergence
  - Regularization
  - Heuristics

- Practice makes perfect..