Deep Learning

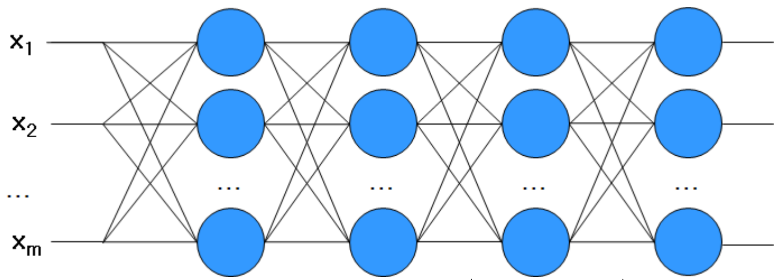
Content

- **Vanishing Gradient & Activation Functions**
- **Dropout**
- **Batch Normalization**

Gradient Vanishing & Activation Functions

Gradient Vanishing & Exploding

- **Gradient is easy to vanish or explode**
  - To many terms are multiplied.
  - If some are small numbers, gradient becomes very small.
  - If some are large numbers, gradient becomes very large.



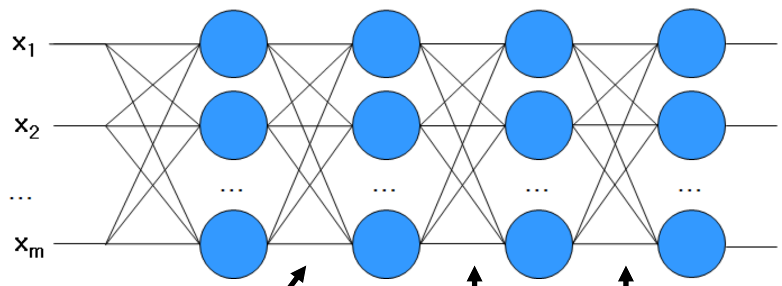
$$\frac{\partial E_n}{\partial w_{kj}} = -(t_{nk} - o_{nk})o_{nk}(1 - o_{nk})h_{nj}$$

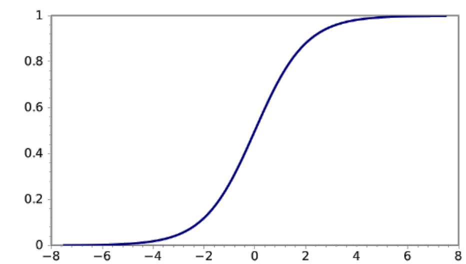
$$\frac{\partial E_n}{\partial w_{ji}} = -h_{nj}(1 - h_{nj})x_{ni}\sum_{k=1}^{m} w_{kj}(t_{nk} - o_{nk})o_{nk}(1 - o_{nk})$$

$$\frac{\partial E}{\partial w_{ip}} = \left(\sum_{j=1}^{J}\left(\sum_{k=1}^{K} -(t_n - o_{nk})o_{nk}(1 - o_{nk})w_{kj}\right)h_{nj}(1 - h_{nj})w_{ji}\right)h_{ni}(1 - h_{ni})h_{np}$$

Activation Function

- **Vanishing Gradient**
  - The major terms are the derivatives of the activation function





$$\frac{\partial \textbf{Sigmoid}}{\partial w} \le \frac{1}{4}$$

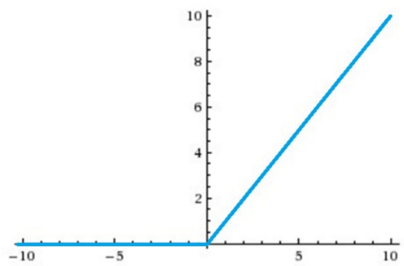
$$\frac{\partial E_n}{\partial w_{kj}} = -(t_{nk} - o_{nk})o_{nk}(1 - o_{nk})h_{nj}$$

$$\frac{\partial E_n}{\partial w_{ji}} = -h_{nj}(1 - h_{nj})x_{ni}\sum_{k=1}^{m} w_{kj}(t_{nk} - o_{nk})o_{nk}(1 - o_{nk})$$

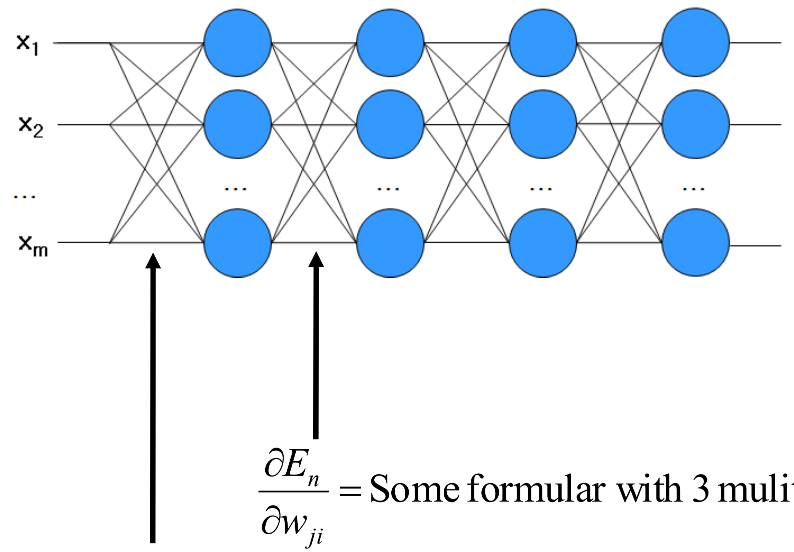
$$\frac{\partial E}{\partial w_{ip}} = \left(\sum_{j=1}^{J}\left(\sum_{k=1}^{K} -(t_n - o_{nk})o_{nk}(1 - o_{nk})w_{kj}\right)h_{nj}(1 - h_{nj})w_{ji}\right)h_{ni}(1 - h_{ni})h_{np}$$

Activation Function

- **Using another functions instead of sigmoid**
  - Rectified Linear Unit (ReLU)



$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$



$$\frac{\partial E_n}{\partial w_{ji}} = \text{Some formular with 3 mulitiplications of } \frac{\partial f}{\partial w}$$

$$\frac{\partial E_n}{\partial w_{hg}} = \text{Some formular with 4 mulitiplications of } \frac{\partial f}{\partial w}$$

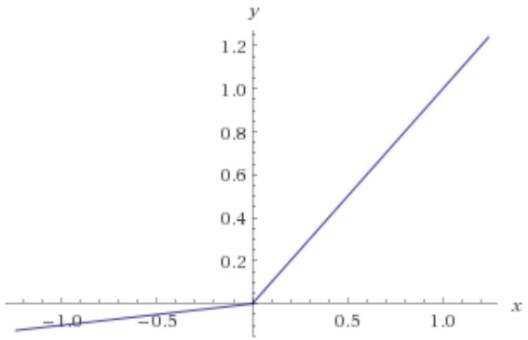
Activation Function

- **Advantage**
  - No vanishing gradient problems.
    - Deep networks can be trained without pre-training
  - Sparse activation
    - In a randomly initialized network, only about 50% of hidden units are activated
  - Fast computation:
    - 6 times faster than sigmoid function
- **Disadvantage**
  - Knockout Problem

Activation Function

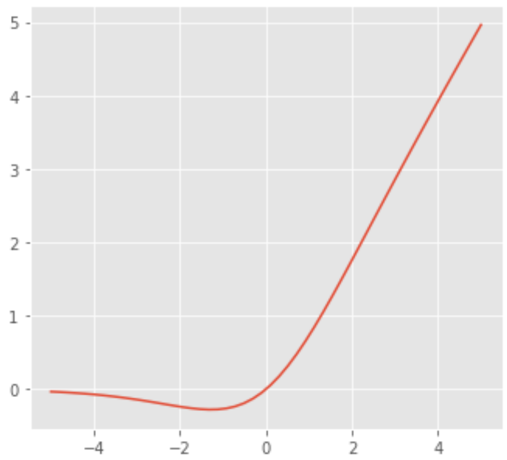
- **You may use another**
  - Leaky ReLU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$

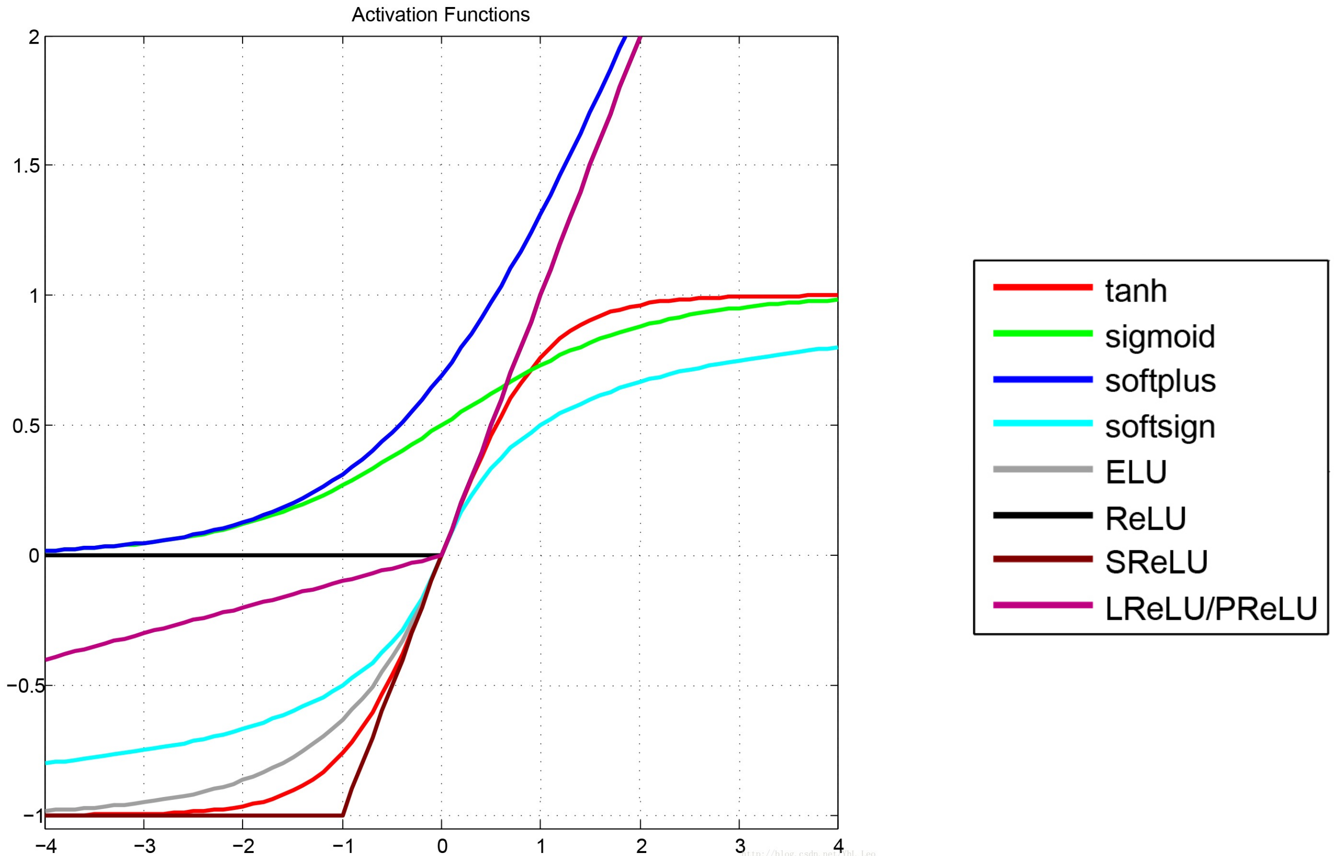


  - Swish (or SiLU-Sigmoid Linear Unit)

$$f(x) = \frac{x}{1 + e^{-x}}$$



Other Activation Functions



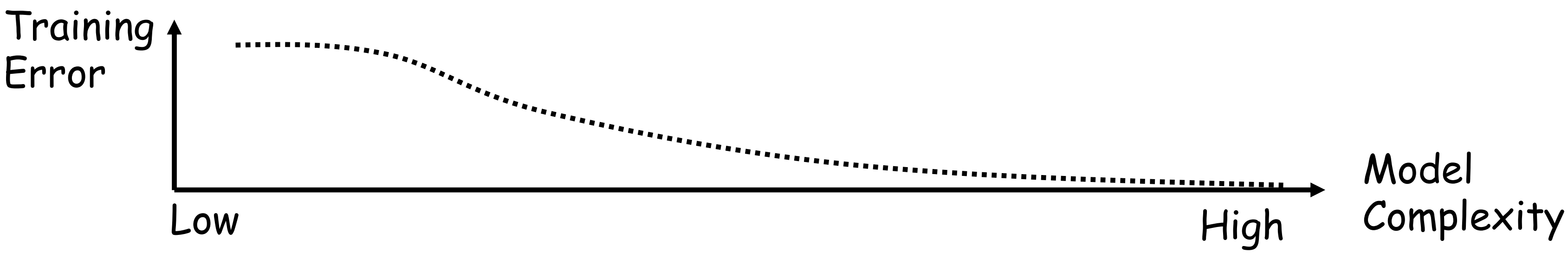
Activation Function

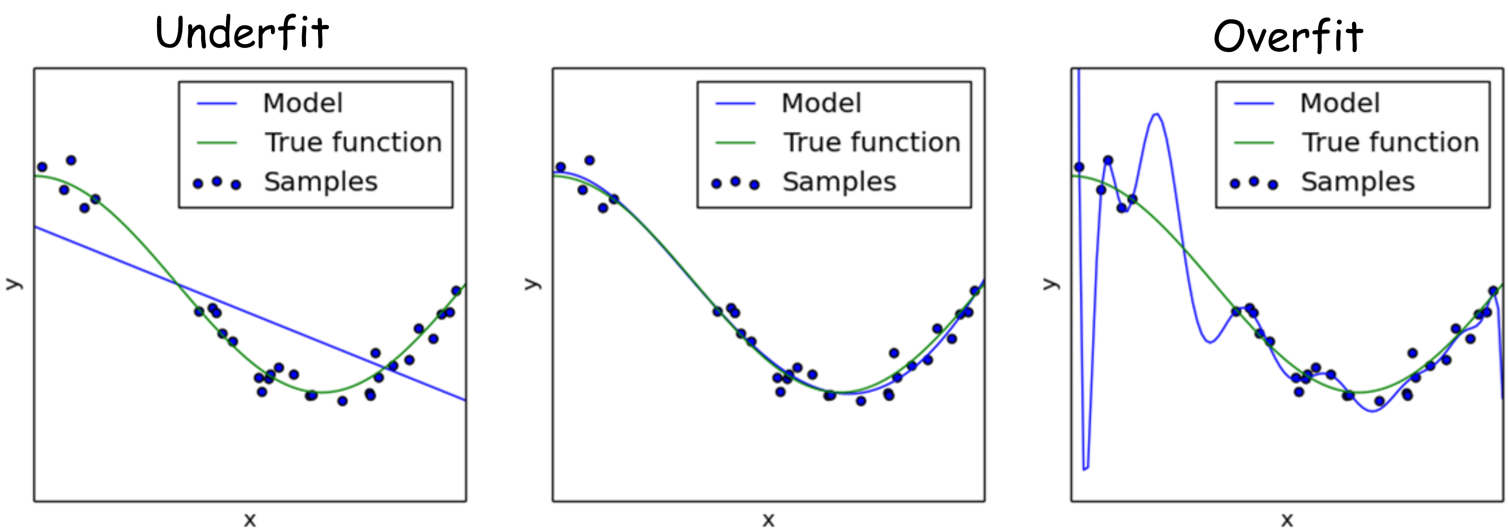
- **Summary**
  - Sigmoid functions and their combinations generally work better but are sometimes avoided due to the vanishing gradient problem
  - ReLU function is a general activation function and is used in most cases these days
  - If we encounter a case of dead neurons in our networks the leaky ReLU function is the best choice
  - ReLU function is usually used in the hidden layers
  - As a rule of thumb, you can begin with using ReLU function and then move over to other activation functions in case ReLU doesn’t provide with optimum results

Regularization

Overfitting

- **Overfitting**





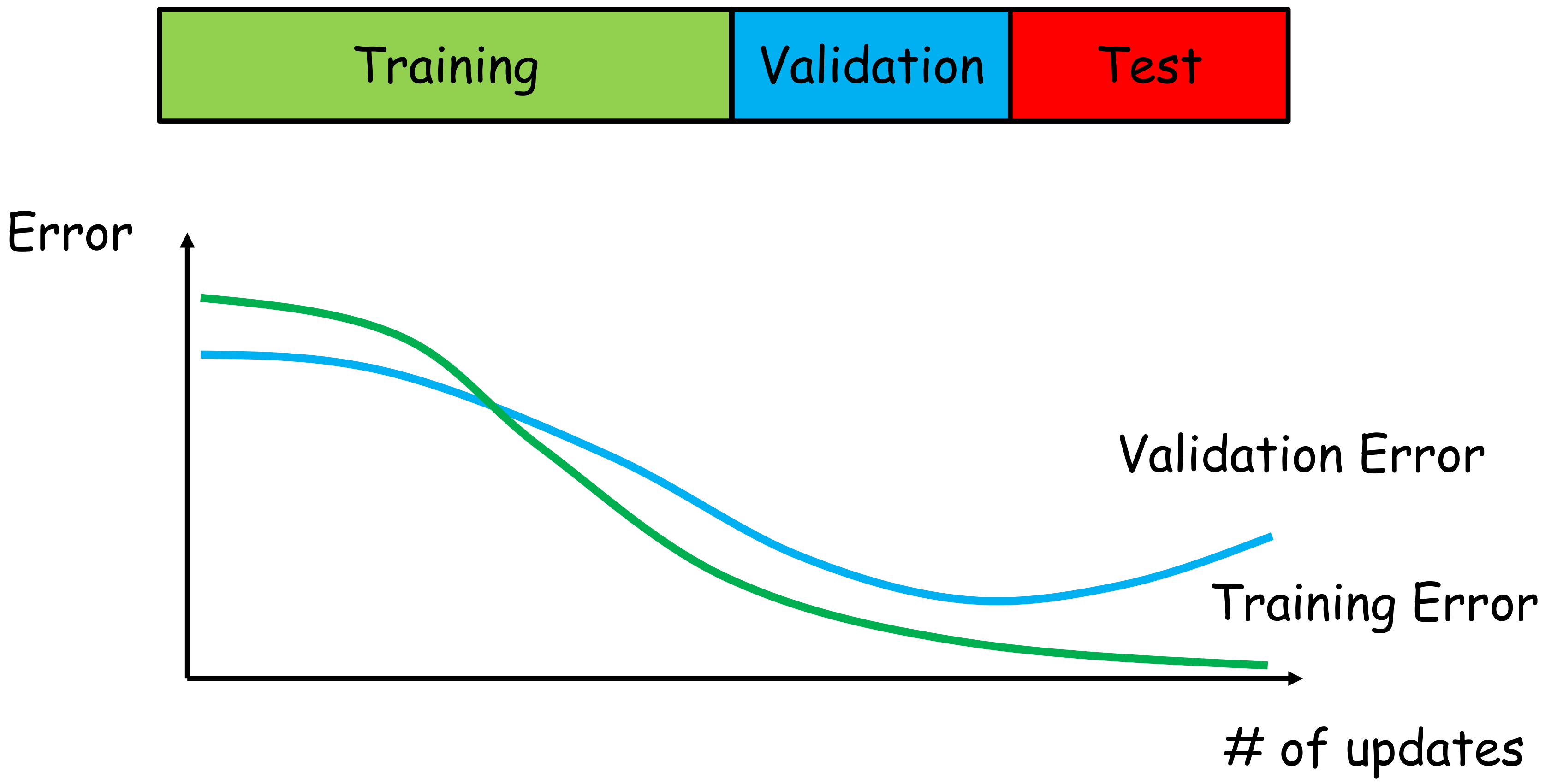
Regularization

- **What is Regularization**
  - Introducing additional information to prevent over-fitting

- **Approaches**
  - Proper Learning: Early stopping
  - Proper Structure: Weight decay, Dropout,
    DropConnect, Stochastic pooling

Early Stopping

- **Split data into 3 groups**



Weight Decay

- **L1 Regularization**
  - Leading most weights very close to zero
  - Choosing a small subset of most important inputs
  - Resistant to noise in the inputs.

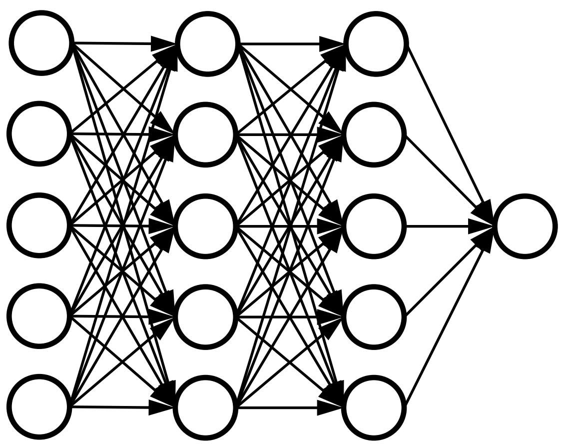
$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}|\mathbf{w}|$$

- **L2 Regularization**
  - Penalizing peaky weights
  - Encouraging to use all of its inputs a little rather than using only some of its inputs a lot.

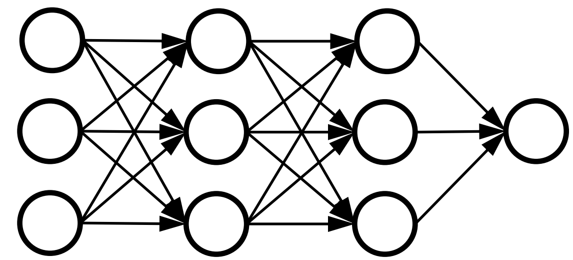
$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

Weight Decay

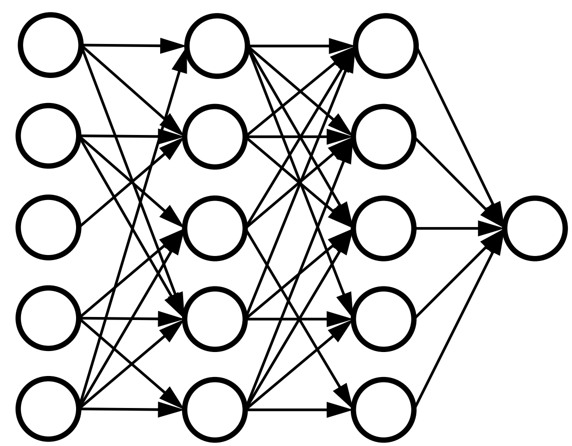
- **Complex Structure vs Simple Structure**



Node
Removal

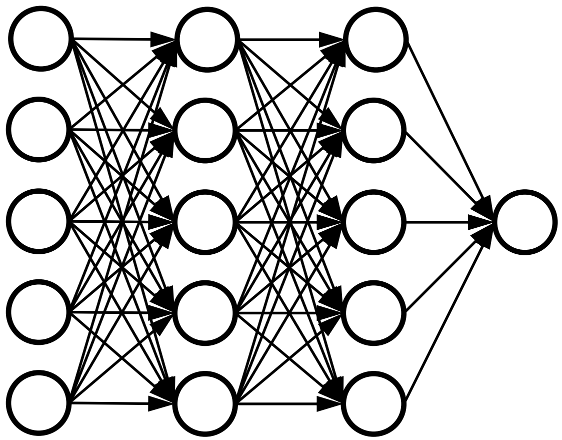


Link
Removal

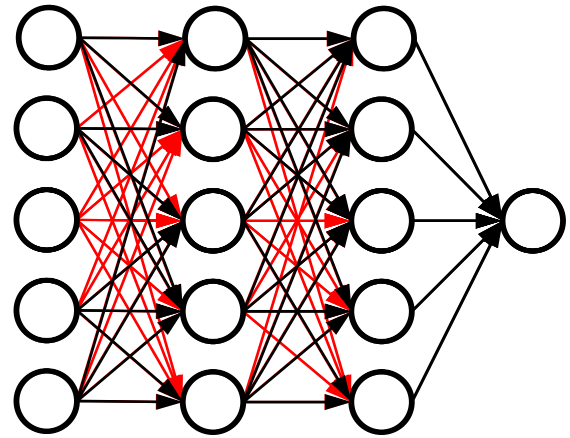


Weight Decay

- **Complex Structure vs Simple Structure**



Set
many links
to zero

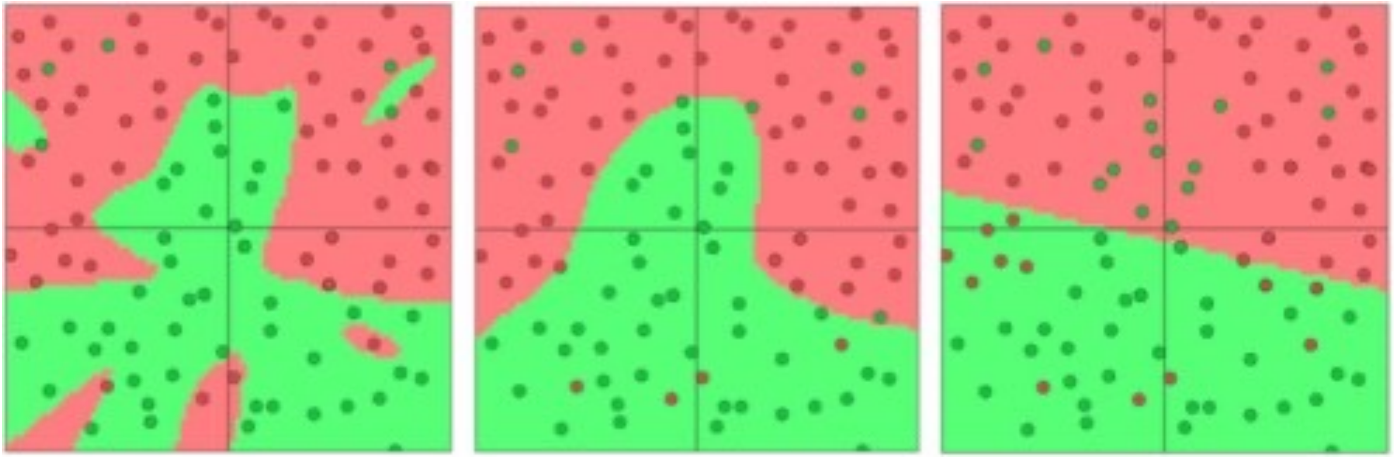


|w| is large <-> NN is Complex

|w| is small <-> NN is Simple

Weight Decay

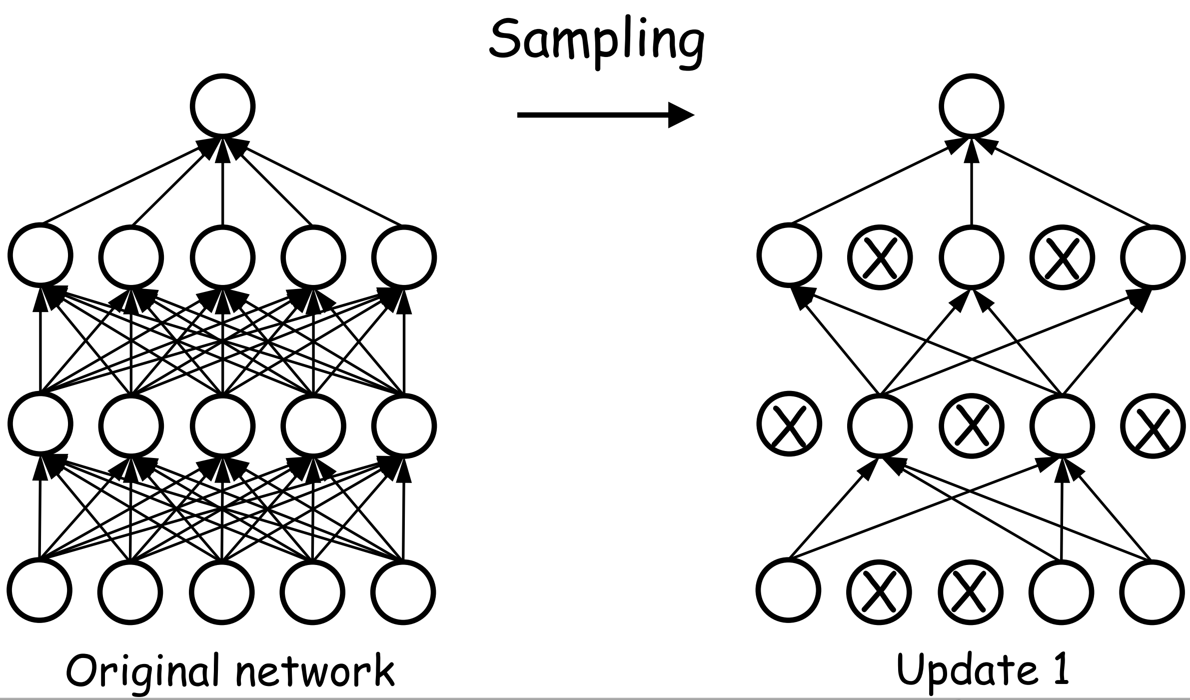
- **Example: Separating green and red**



L2 regularization strengths of 0.01, 0.1, and 1

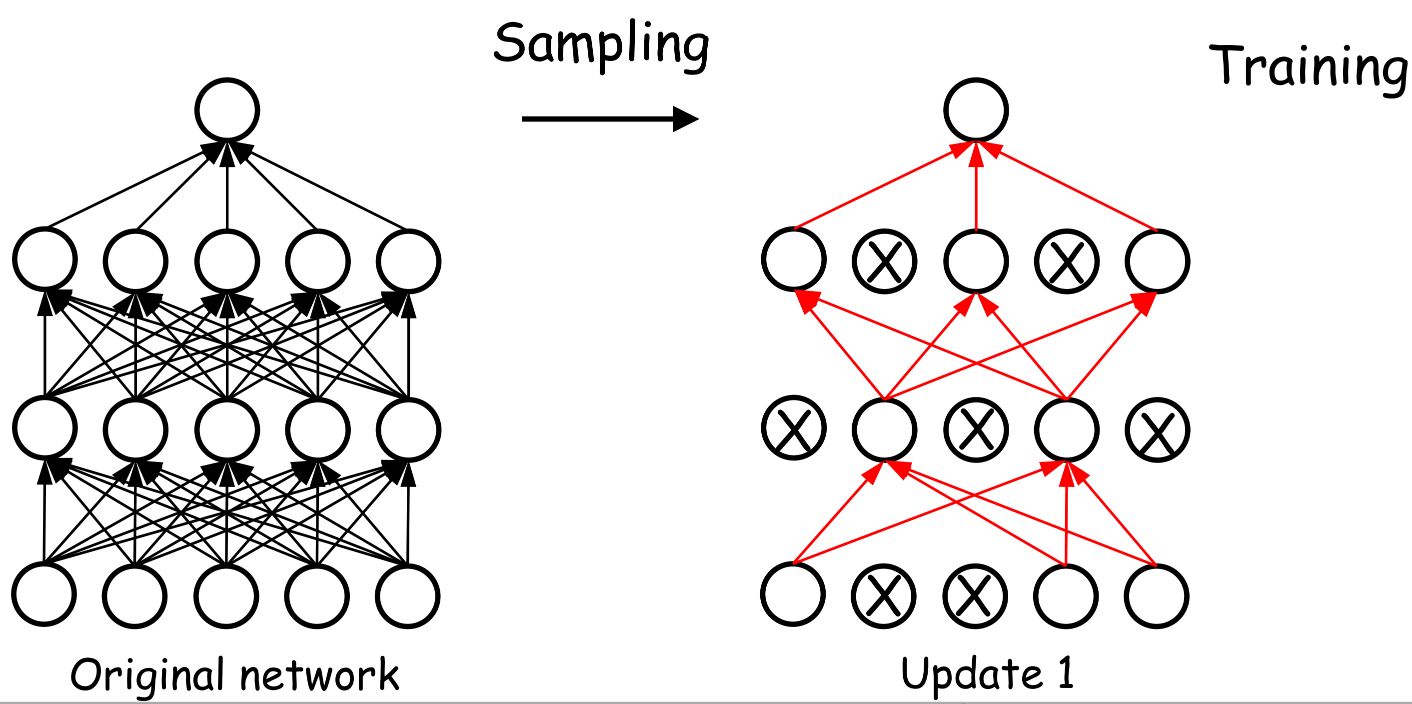
Dropout

- **How can we reduce the structural complexity without removing nodes?**
  - Hmm??



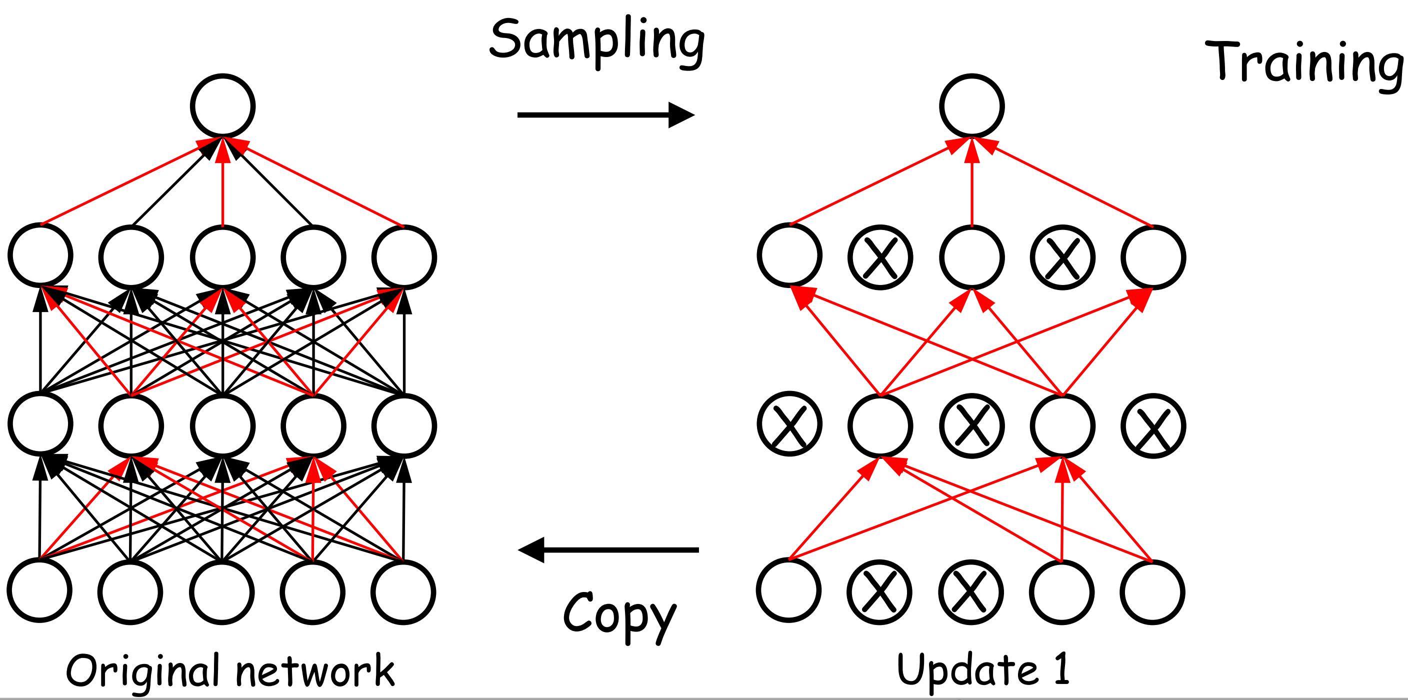
Dropout

- **How can we reduce the structural complexity without removing nodes?**
  - Hmm??



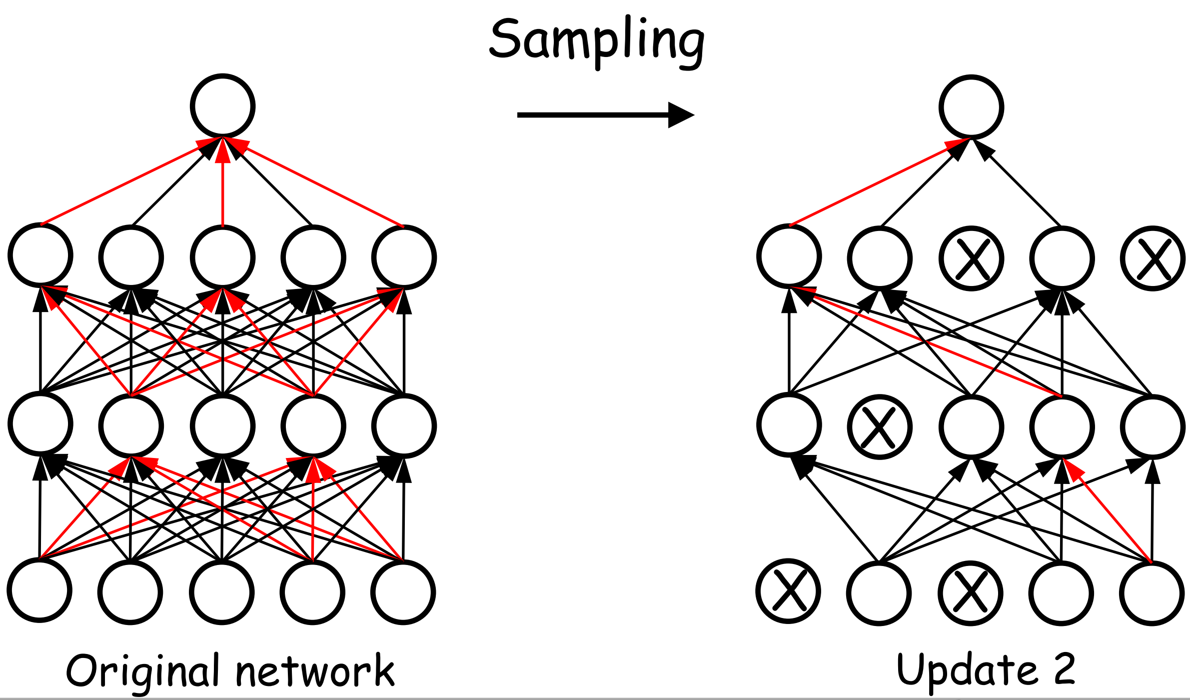
Dropout

- **How can we reduce the structural complexity without removing nodes?**
  - Hmm??



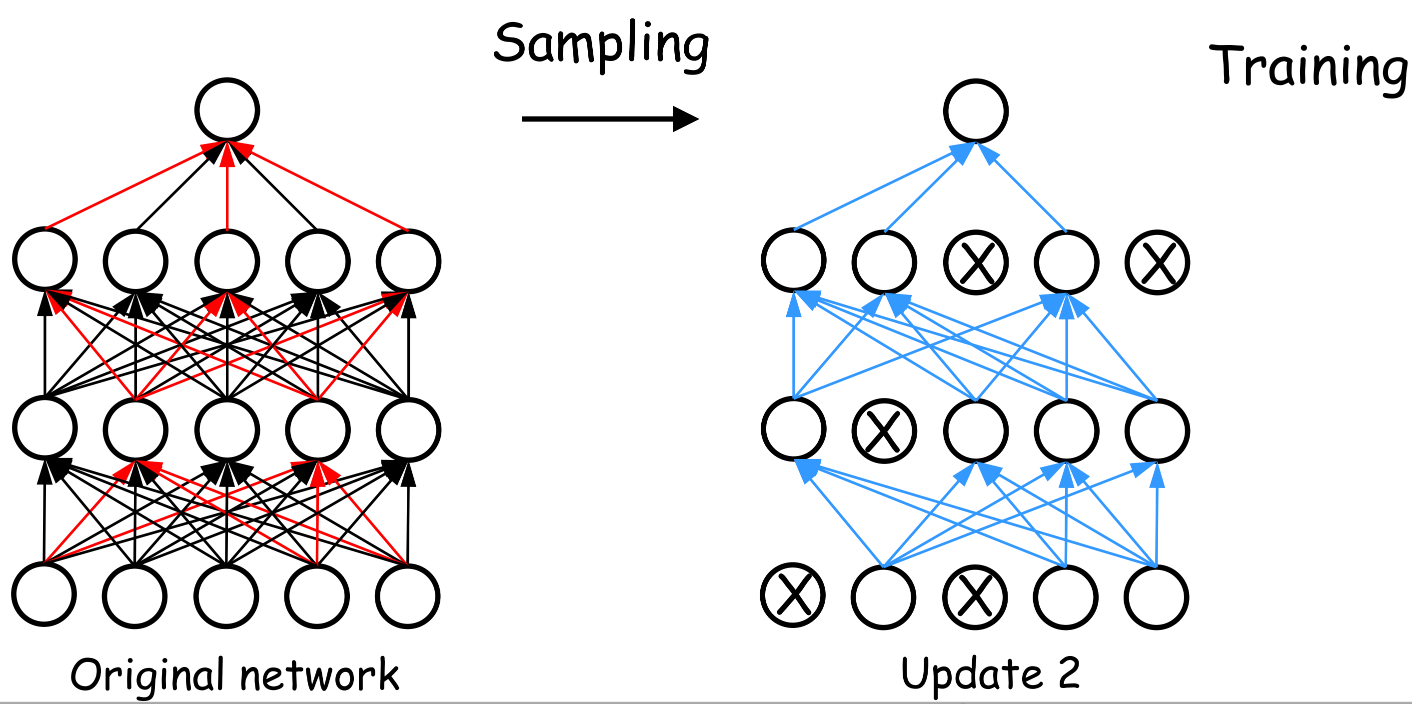
Dropout

- **How can we reduce the structural complexity without removing nodes?**
  - Hmm??



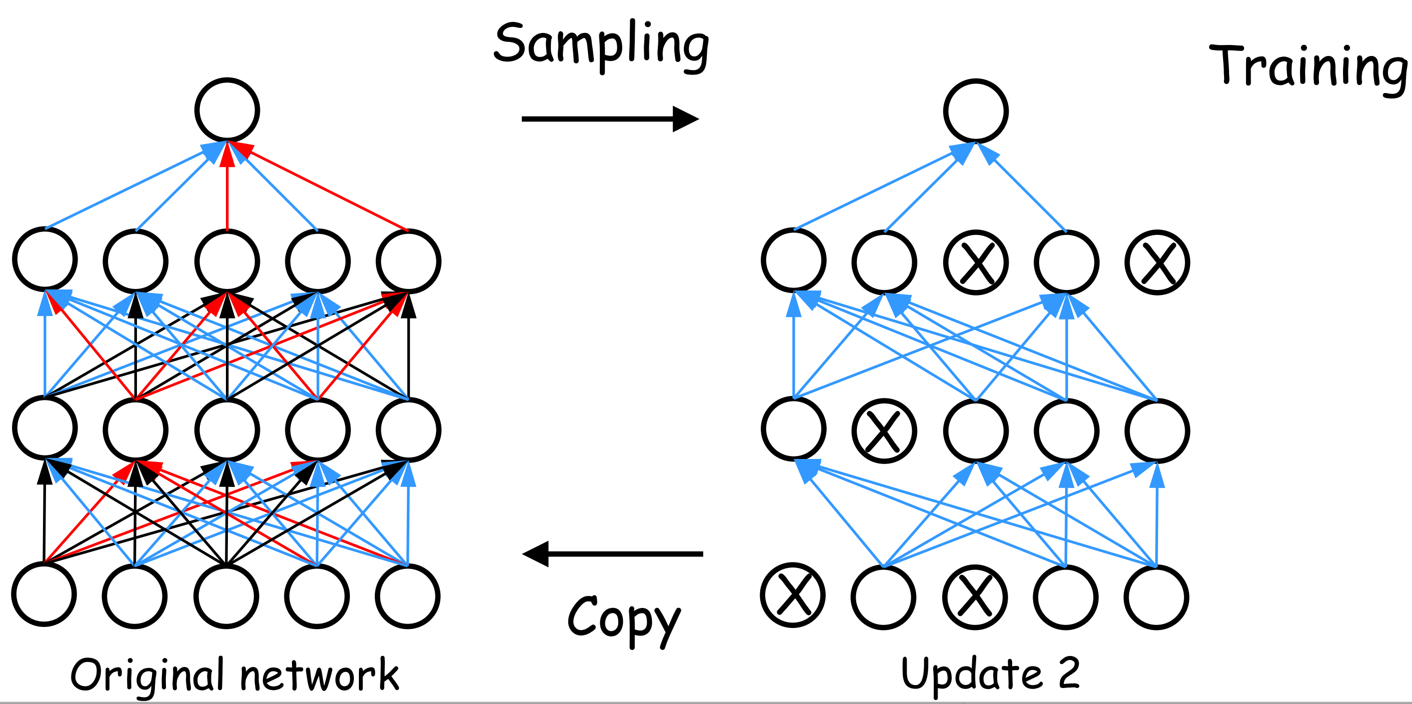
Dropout

- **How can we reduce the structural complexity without removing nodes?**
  - Hmm??



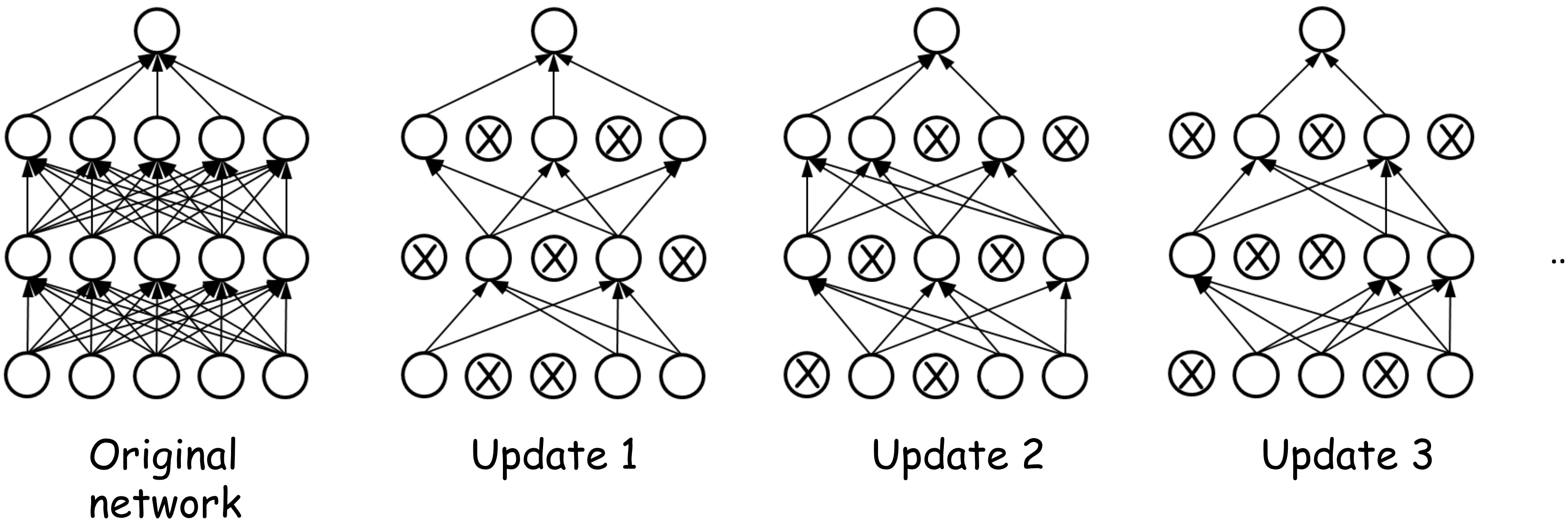
Dropout

- **How can we reduce the structural complexity without removing nodes?**
  - Hmm??



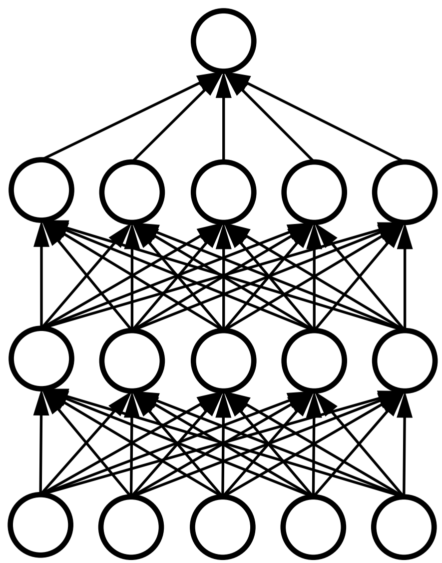
Dropout

- **Do this at every epoch**
  - Randomly choose nodes with a probability of *p*
    - Usually p = 0.5
  - Train the simplified neural network
    - At every epoch, we train different neural network which share connection weight each other



Dropout

- **Testing**
  - Use all the nodes without dropout

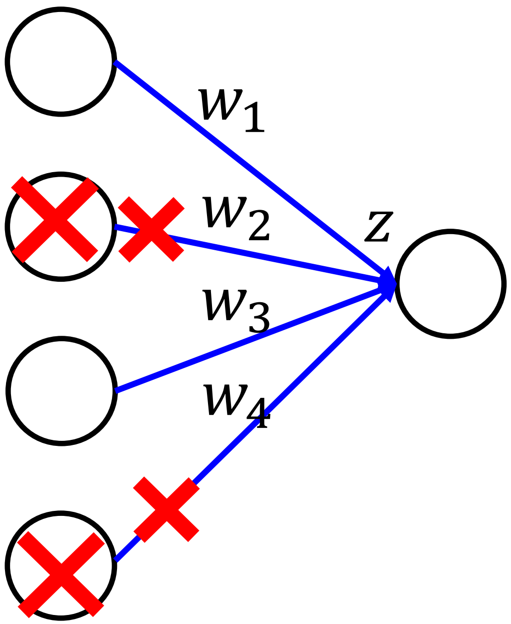


Dropout

- **Testing**

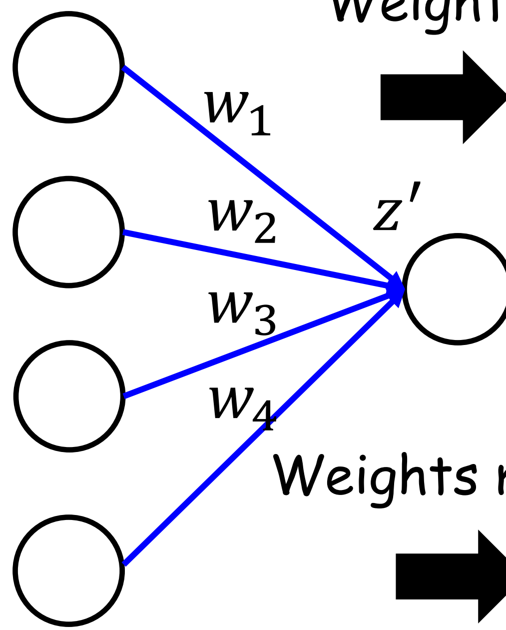
Training of Dropout

Assume dropout rate is 50%



Testing of Dropout

No dropout



Weights from training

$\Rightarrow z' \approx 2z$

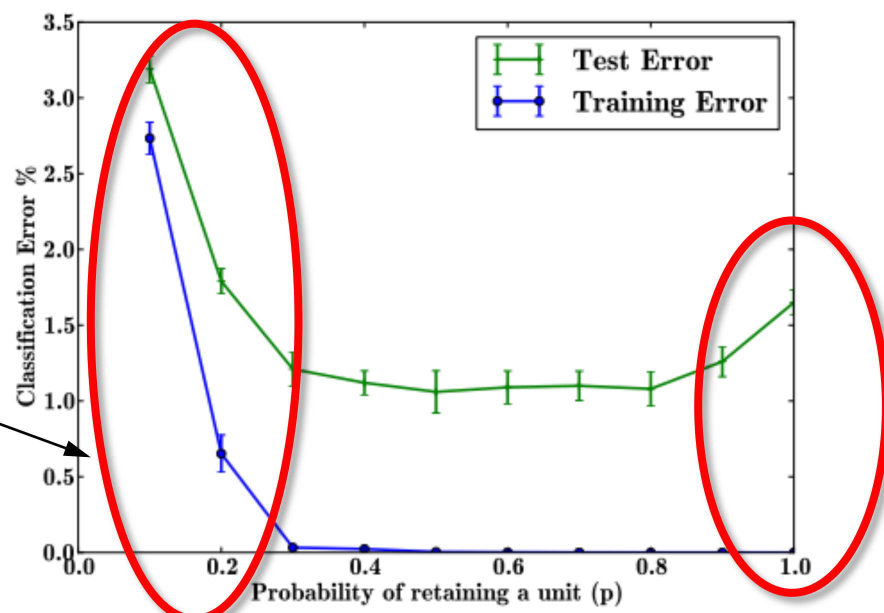
Weights multiply (1-p)%

$\Rightarrow z' \approx z$

Dropout

- **The effect of the dropout rate p:**
  - An architecture of 784-2048-2048-2048-10 is used on the MNIST dataset.
  - The dropout rate p is changed from small numbers (most units are dropped out) to 1.0 (no dropout).

Dropout with high probability

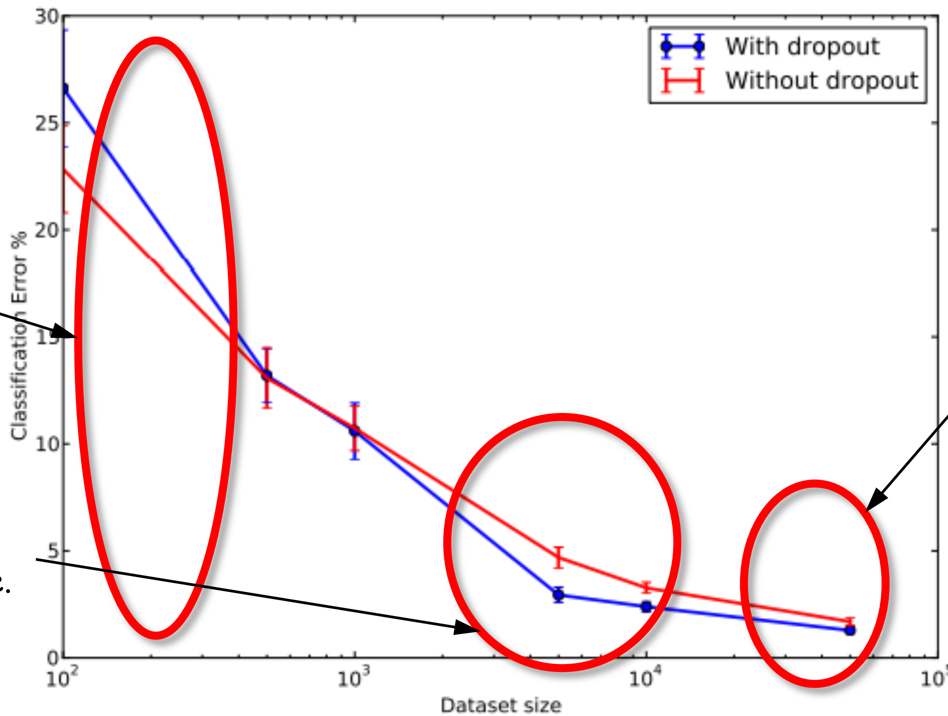


No Dropout ->Overfitting

Dropout

- **The effect of data set size:**
  - An architecture of 784-1024-1024-2048-10 is used on the MNIST dataset.

Extremely small data set
- Dropout does not improve error rate, and even makes it worse.



Average to large data set
- Dropout improves error rate.

Huge data set
- Dropout barely improves the error rate. The data set is big enough, so that overfitting is not an issue.

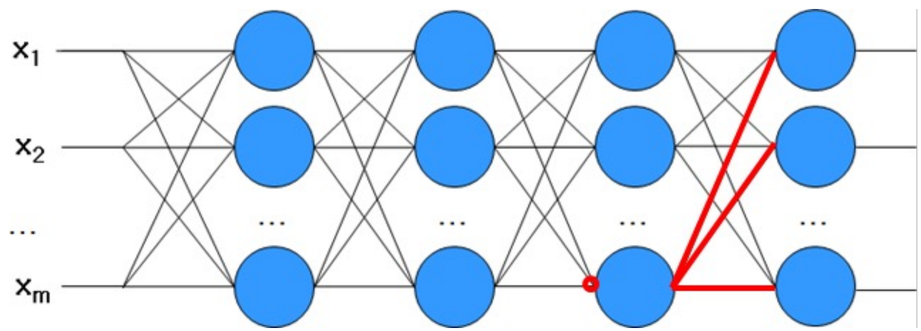
Dropout

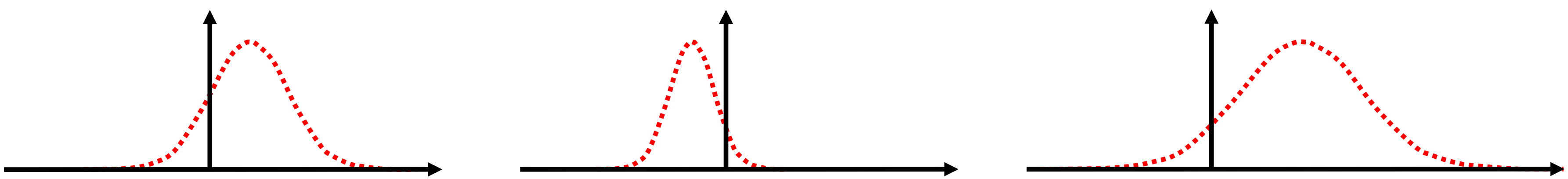
- **Summary**
  - Dropout is a very good and fast regularization method.
  - Dropout is a bit slow to train (2-3 times slower than without dropout).
  - If the amount of data is average-large – dropout excels. When data is big enough, dropout does not help much.
  - Dropout achieves better results than former used regularization methods (Weight Decay).

Batch Normalization

Batch Normalization

- **Distribution Shift**
  - Output distribution of the red node



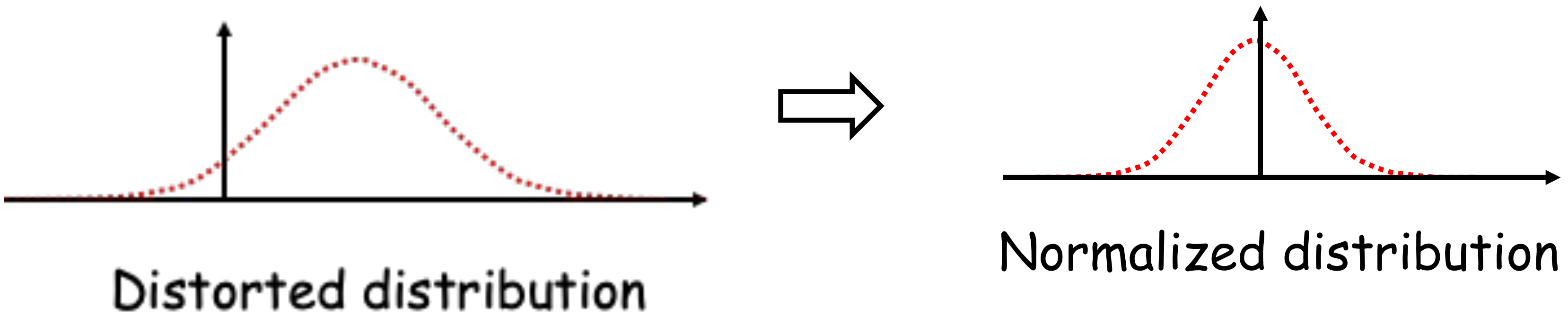


Update 1 Update 2 Update 3

Batch Normalization

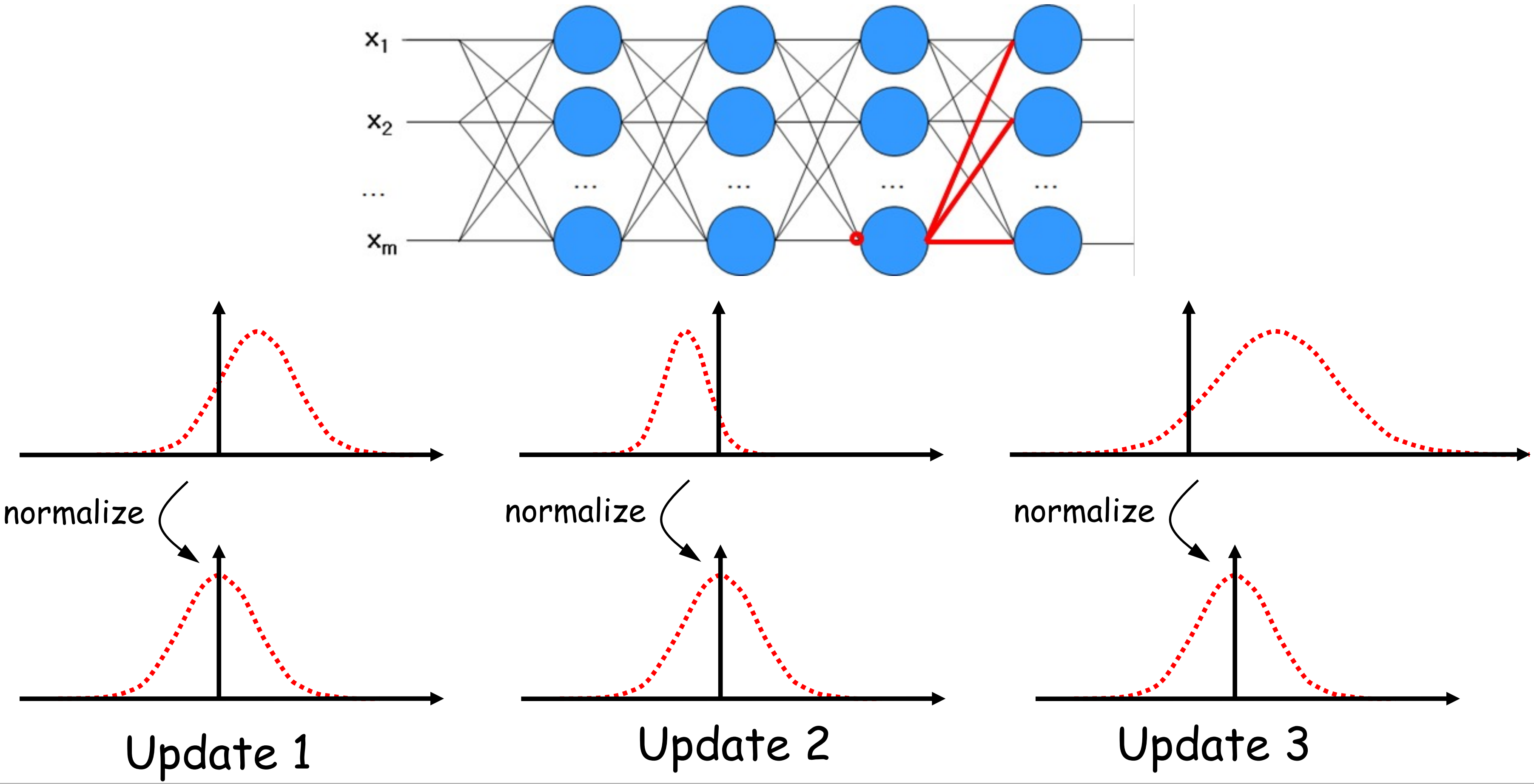
- **Distribution Shift**
  - It disturbs the learning process,
  - Learning is getting slow down

- **Why don't we normalize the distribution of inputs**



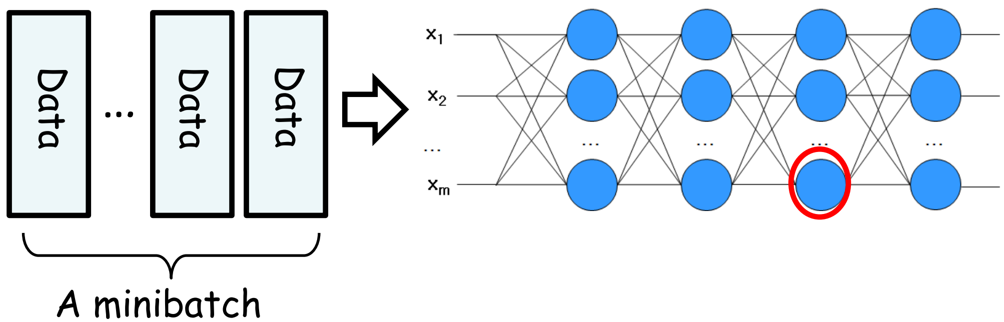
Batch Normalization

- **Normalization of outputs**

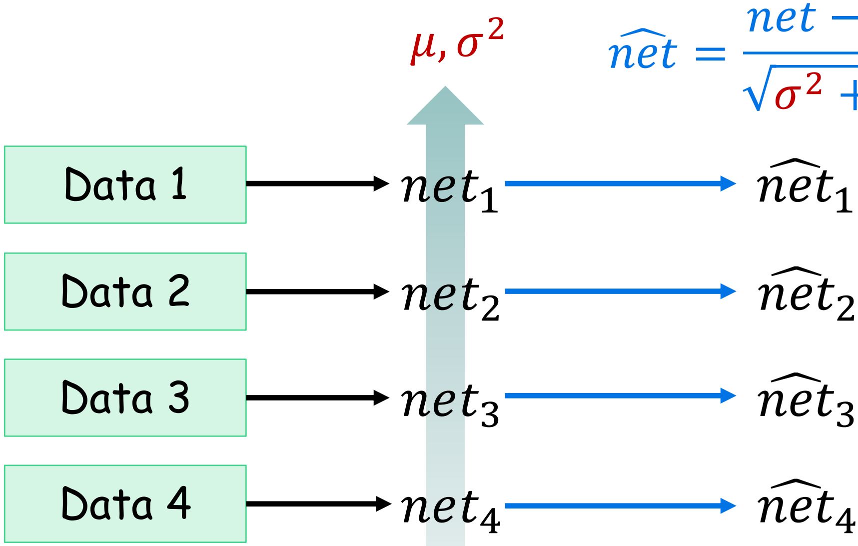


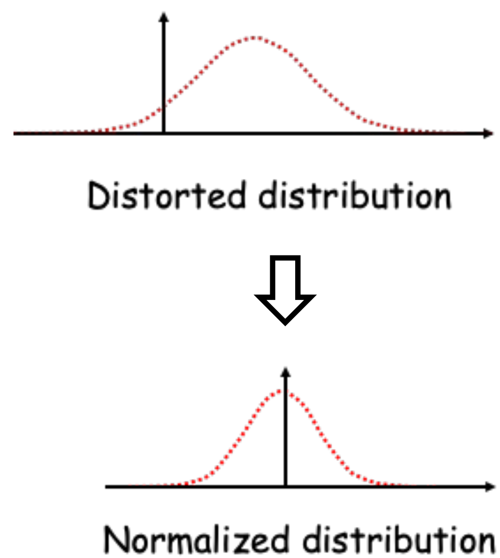
Batch Normalization

- **Input Normalization**



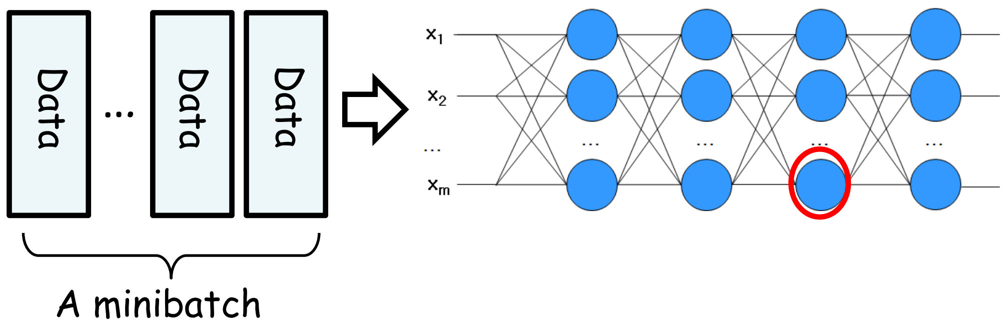
$$\widehat{net} = \frac{net - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$





Batch Normalization

▪ Input Normalization



μ, σ^2

$$\widehat{net} = \frac{net - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

Data 1 → net_1 → $\widehat{net}_1$ → $f(\widehat{net}_1)$

Data 2 → net_2 → $\widehat{net}_2$ → $f(\widehat{net}_2)$

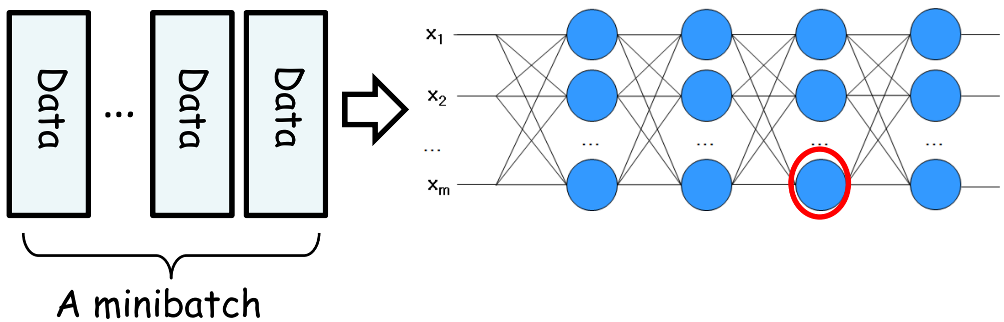
Data 3 → net_3 → $\widehat{net}_3$ → $f(\widehat{net}_3)$

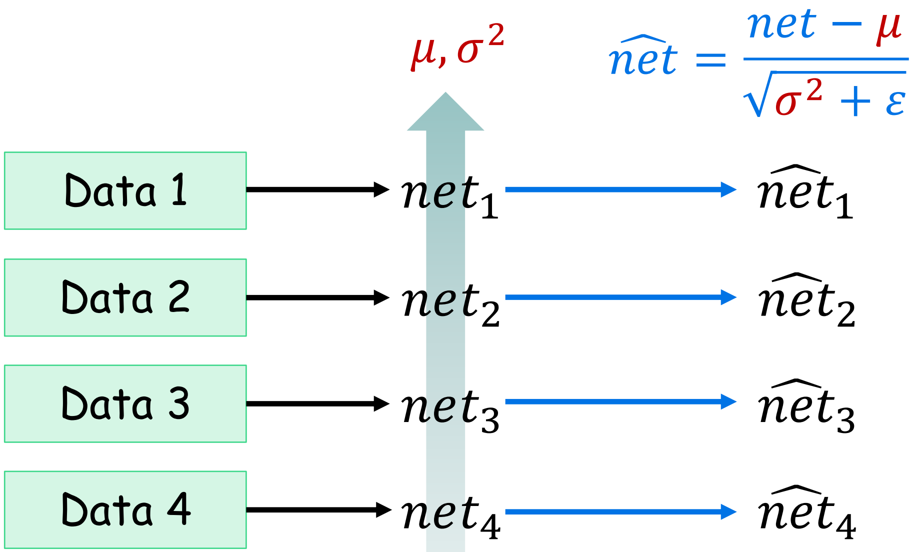
Data 4 → net_4 → $\widehat{net}_4$ → $f(\widehat{net}_4)$

?

Batch Normalization

Input Normalization





$$net = \mathbf{wx} + b$$

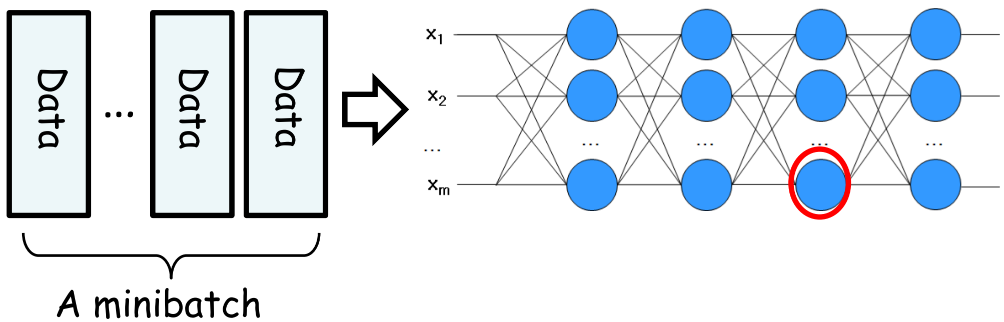
$$E(net) = E(\mathbf{wx}) + b$$

$$\widehat{net} = \frac{\mathbf{wx} + b - (E(\mathbf{wx}) + b)}{\sqrt{\sigma^2 + \varepsilon}}$$

$$\widehat{net} = \frac{\mathbf{wx} - E(\mathbf{wx})}{\sqrt{\sigma^2 + \varepsilon}}$$

Batch Normalization

- **Input Normalization**



μ, σ^2

$$\widehat{net} = \frac{net - \mu}{\sqrt{\sigma^2 + \varepsilon}} \qquad \widetilde{net} = \gamma \widehat{net} + \beta$$

Data 1 $\rightarrow net_1 \rightarrow \widehat{net}_1 \rightarrow \widetilde{net}_1 \rightarrow f(\widetilde{net}_1)$

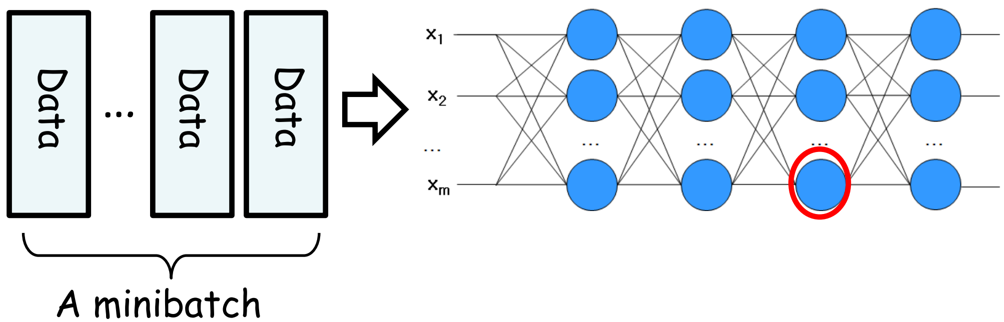
Data 2 $\rightarrow net_2 \rightarrow \widehat{net}_2 \rightarrow \widetilde{net}_2 \rightarrow f(\widetilde{net}_2)$

Data 3 $\rightarrow net_3 \rightarrow \widehat{net}_3 \rightarrow \widetilde{net}_3 \rightarrow f(\widetilde{net}_3)$

Data 4 $\rightarrow net_4 \rightarrow \widehat{net}_4 \rightarrow \widetilde{net}_4 \rightarrow f(\widetilde{net}_4)$

Batch Normalization

- **Input Normalization**



μ, σ^2

$$\widehat{net} = \frac{net - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

$$\widetilde{net} = \gamma \widehat{net} + \beta$$

Data 1 → net_1 → $\widehat{net}_1$ → $\widetilde{net}_1$ → $f(\widetilde{net}_1)$

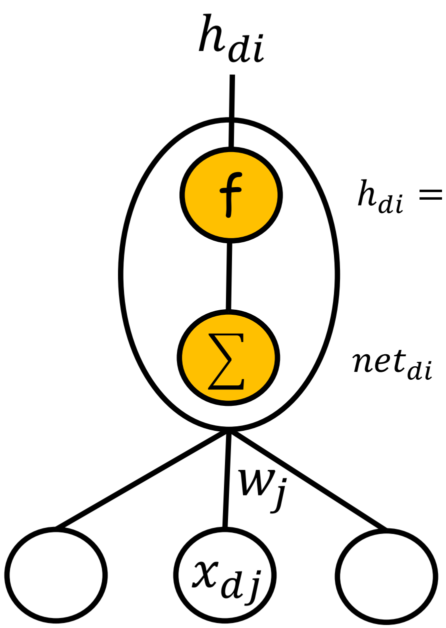
Data 2 → net_2 → $\widehat{net}_2$ → $\widetilde{net}_2$ → $f(\widetilde{net}_2)$

Data 3 → net_3 → $\widehat{net}_3$ → $\widetilde{net}_3$ → $f(\widetilde{net}_3)$

Data 4 → net_4 → $\widehat{net}_4$ → $\widetilde{net}_4$ → $f(\widetilde{net}_4)$

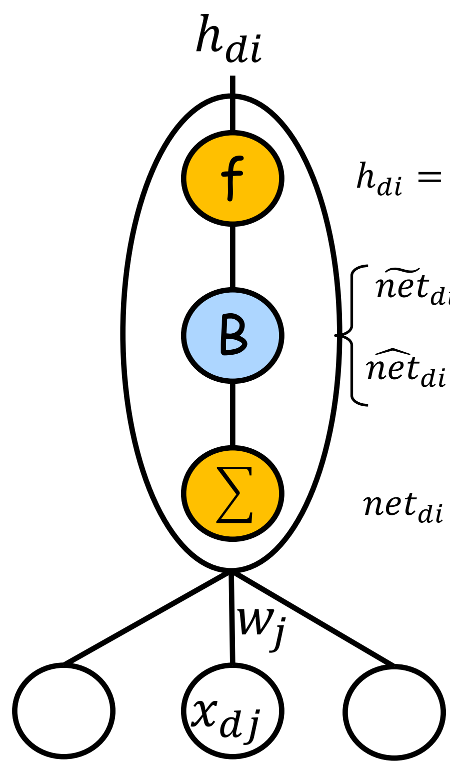
Batch Normalization

- **For a Single Node**



$$h_{di} = Activation(net_{di})$$

$$net_{di} = \sum_{j=1}^{J} w_j x_{dj} + b$$



$$h_{di} = Activation(\widetilde{net}_{di})$$

$$\begin{cases} \widetilde{net}_{di} = \gamma \widehat{net}_{di} + \beta \\ \widehat{net}_{di} = \dfrac{net_{di} - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \end{cases}$$

$$net_{di} = \sum_{j=1}^{J} w_j x_{dj} + b$$

$$\mu_B = \frac{1}{D_B} \sum_{d=1}^{D_B} net_{di}$$

$$\sigma_B^2 = \frac{1}{D_B} \sum_{d=1}^{D_B} (net_{di} - \mu)^2$$

Batch Normalization

- **Testing**
  - For Training, the mean and variance of each batch are used for normalization
  - For Testing, of which data the mean and variance will be used?
    - Estimated with those of batches in the training

$$\widehat{net}_{ti} = \frac{net_{di} - \mu}{\sqrt{\sigma^2 + \varepsilon}} \xleftarrow{\text{Transform}} net_{ti} = \sum_{j=1}^{J} w_j x_{tj} + b$$

Test sample

$$\mu = \frac{1}{B}\sum_{b=1}^{B} \mu_b$$

Unbiased estimator of mean of the population

$$\sigma^2 = \frac{1}{B}\sum_{b=1}^{B} \frac{m}{m-1}\sigma_b^2$$

Mean of unbiased estimator of variance of the population

B: # of batches
m: # of samples in each batch

$$\widetilde{net}_{di} = \gamma \widehat{net}_{di} + \beta$$

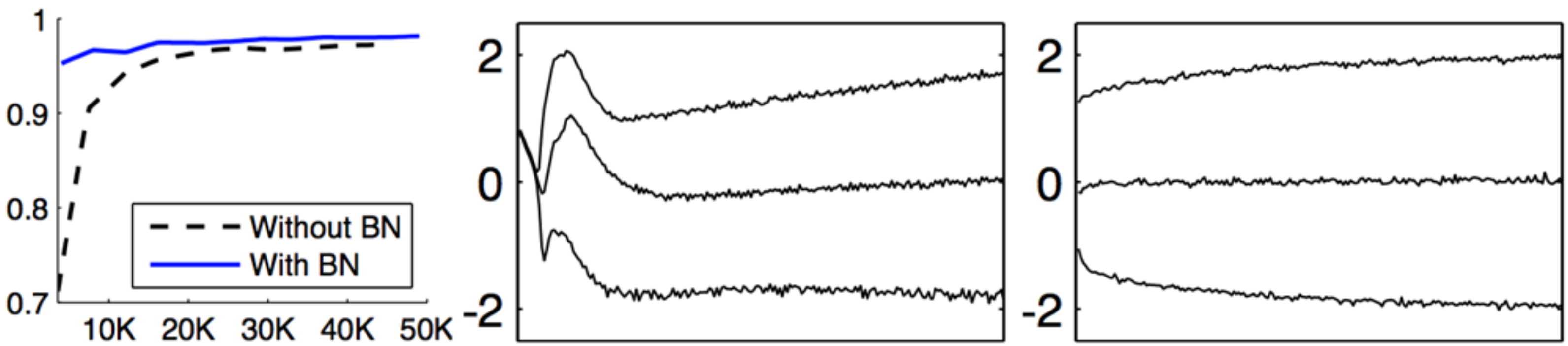
$$h_{di} = Activation(\widetilde{net}_{di})$$

Batch Normalization

- **Advantage**
  - Reduces internal covariant shift.
  - Reduces the dependence of gradients on the scale of the connection weights.
  - Regularizes the model and reduces the need for regularization techniques.
    - It adds some stochastic noise to the activations as a result of using noisy estimates computed on the mini-batches. This has a regularization effect in some applications,

Batch Normalization

- **Performance with BN**



(a) *accuracy* (b) Without BN (c) With BN

input distributions

Batch Normalization

- **Disadvantage**
  - Expensive: Memory and time
    - Must keep interim results of all instances in a batch
    - Especially in CNN, usually an image is large
  - Hard to apply when the batch size is small
    - If batches are small, the means and variances cannot approximate the global ones.
  - Hard to apply to recurrent networks
    - It doesn’t match to structure of recurrent networks
    - Hard to implement with recurrent networks

Layer Normalization

- **Recap: Batch Normalization**
  - Normalization of each node output

Batch normalization

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_{ij}$$
$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_{ij} - \mu_j)^2$$
$$\hat{x_{ij}} = \frac{x_{ij}-\mu_j}{\sqrt{\sigma_j^2+\epsilon}}$$

i , j : index of the batch and the node of hidden layers

Layer Normalization

- **Recap: Batch Normalization**
  - Normalization of each node output

Outputs of a layer for a mini-batch			Avg	Std	Normalized outputs		
1.0	3.0	6.0	3.3	2.5	-0.9	-0.1	1.1
2.0	2.0	2.0	2.0	0.0	0.0	0.0	0.0
0.0	1.0	5.0	2.0	2.6	-0.8	-0.4	1.1
4.0	6.0	1.0	3.7	2.5	0.1	0.9	-1.1
5.0	2.0	3.0	3.3	1.5	1.1	-0.9	-0.2
1.0	0.0	1.0	0.7	0.6	0.6	-1.2	0.6

Layer Normalization

- **Proposed as an alternative to Batch Normalization**
  - Works regardless of batch size (batch size = 1)
  - Performs well with RNNs

Layer normalization:

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_{ij}$$
$$\sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}(x_{ij} - \mu_i)^2$$
$$\hat{x_{ij}} = \frac{x_{ij} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

i , j : index of the batch and the node of hidden layers

Layer Normalization

Outputs of a layer for a mini-batch

	1.0	3.0	6.0
	2.0	2.0	2.0
	0.0	1.0	5.0
	4.0	6.0	1.0
	5.0	2.0	3.0
	1.0	0.0	1.0
Avg	2.2	2.3	3.0
Std	1.9	2.1	2.1

Normalized outputs

-0.6	0.3	1.4
-0.1	-0.2	-0.5
-1.1	-0.6	1.0
0.9	1.8	-1.0
1.5	-0.2	0.0
-0.6	-1.1	-1.0

Layer Normalization

- **Group Normalization shows consistent accuracy with smaller batches**
  - Tested on ImageNet (1000 Classes, 1.28M training, 50K validation), ResNet-50