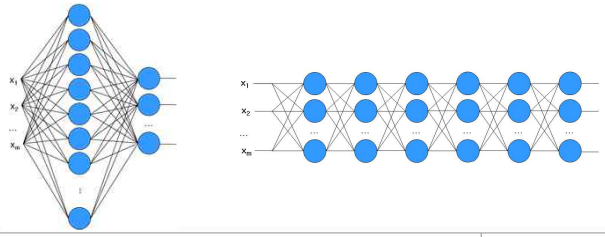
Limits of Neural Networks

Why Deep Learning

- **Biological Plausibility – e.g. Visual Cortex**
- **Hastad proof - Problems which can be represented with a polynomial number of nodes with *k* layers, may require an exponential number of nodes with *k*-1 layers (e.g. parity)**
- **Highly varying functions can be efficiently represented with deep architectures**
  - Less weights/parameters to update than a less efficient shallow representation
- **Sub-features created in deep architecture can potentially be shared between multiple tasks**

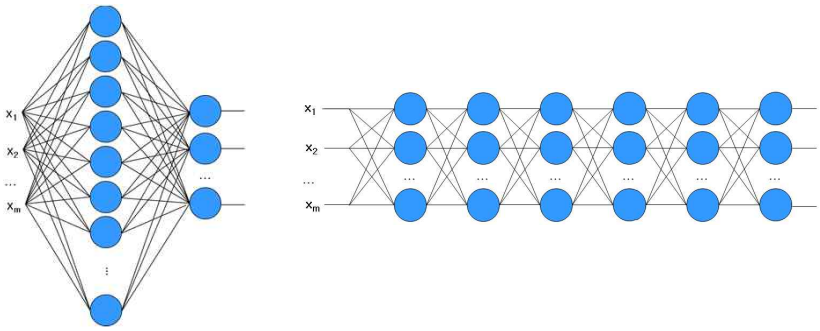
Limits of Neural Networks

- **Shallow Network vs Deep Network**
  - Theoretically, it can mimic any functions, but the number of nodes can exponentially increase in a shallow network
  - In order to avoid the exponentional explosion, we can adopt deep networks (many layers)
  - Deep networks have very strong power but cause many problems!!



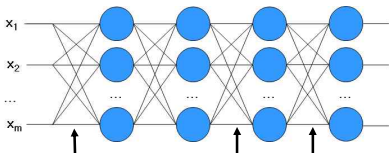
Limits of Neural Networks

- **Pros and Cons of Deep Networks**
  - Pros: Powers from deep structure
  - Cons: Hard to optimize, Overfitting, Internal covariate shift



Limits of Neural Networks

- **Hard to Optimize**
  - Vanishing Gradient: EBP may not work properly !!



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

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

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

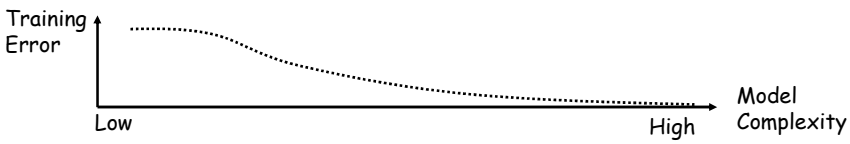
Very SLOW!!

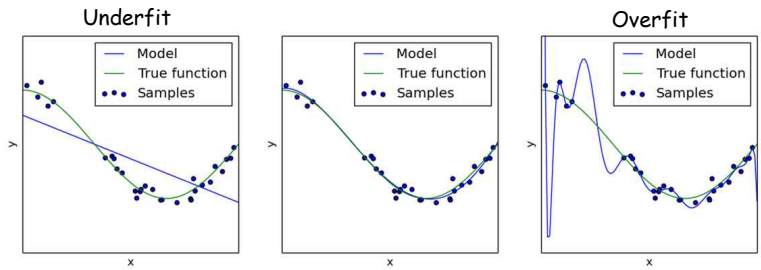
$$\frac{\partial E_n}{\partial w_{gh}} = \text{Some formular with 4 mulitiplications of } \frac{\partial \text{Sigmoid}}{\partial w}$$

which are smaller than 0.005

Limits of Neural Networks

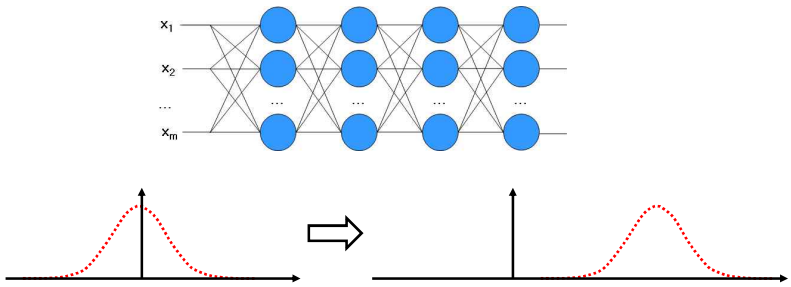
- **Overfitting**





Limits of Neural Networks

- **Internal Covariate Shift**
  - A small change in previous layers leads a large change in the later



Distribution of inputs of a node is easy to shift very fast

Limits of Neural Networks

- **Solutions**
  - High Model Complexity
    - Adopting deep but relatively simple structures
  - Hard to Optimize
    - Choosing good initial points
    - Using another activation functions
    - Mini-batch gradient descent
  - Overfitting
    - (A huge amount of data)
    - Various Regularization Method (including Dropout)
  - Internal Covariate Shift
    - Batch normalization

Limits of Neural Networks

- **Structures of Deep Networks**
  - Fully Connected Deep Networks
    - High structural complexity
    - Applicable to any data types
    - Deep Belief Network, Stacked Denoising Autoencoder, etc
  - Partially Connected Deep Networks
    - Reduction of complexity targeting to special datatype
    - Spatial or temporal sequential data
    - Convolutional Neural Networks (CNN)
      Recurrent Neural Networks (RNN)
      Recursive Neural Networks (RNN)

Various Technique to Optimize Neural Networks

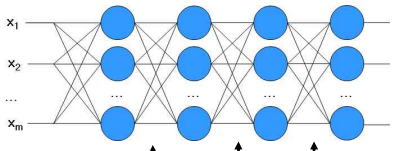
Content

- **Activation Functions**
- **Good Initialization**
- **Drop-Out**
- **Batch Normalization**

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.



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

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

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

Gradient Vanishing & Exploding

- **Why Gradient Exploding is bad?**
  - A big jump in the gradient descent method can cause a divergence which means a failure of learning

- **To avoid**
  - Use small learning rate

$$w^{t+1} = w^t - \eta \frac{\partial E}{\partial w}$$

Small learning rate can curtail a large gradient

  - Small learning rate causes slow learning!!
    - Anyway, it is one of good ways to avoid gradient exploding

Gradient Vanishing & Exploding

- **Then, how about Gradient Vanishing?**
  - Small learning rate aggravates gradient vanishing problems

$$w^{t+1} = w^t - \eta \frac{\partial E}{\partial w}$$

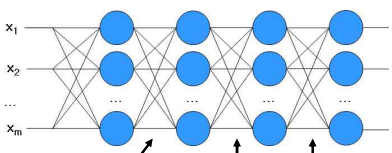
Small gradient

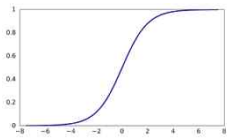
Small learning rate

  - We can find a solution in another way
    - Learning rate is small, so
    - Let's make the gradient not too small

Activation Function

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





$$\frac{\partial \text{Sigmoid}}{\partial w} \leq \frac{1}{4}$$

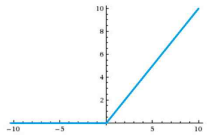
$$\frac{\partial E_n}{\partial w_{kj}} = -(t_{nk} - o_{nk}) o_{nk}(1 - o_{nk}) h_{nj}$$

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

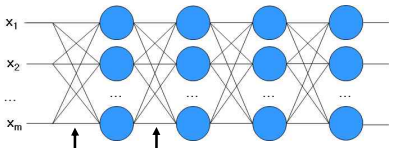
$$\frac{\partial E}{\partial w_{ip}} = \left(\sum_{j=1}^{J} \left(\sum_{k=1}^{K} -(t_n - o_{nk}) o_{nk}(1 - o_{nk}) w_{kj} \right) h_{nj}(1 - h_{nj}) w_{ji} \right) h_{ni}(1 - h_{ni}) h_{np}$$

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

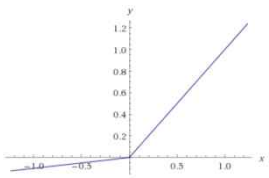
Activation Function

- **ReLU units can be fragile**
  - What if a node is never fired for all the data points (always negative) -> Never trained!!
  - For example,
    - If a large gradient flowing through a ReLU, it makes big updates of weights
    - Weights can locate at very bad points where the neuron will never activate on any data point again.
    - You may find that as much as 40% of your network can be "dead" (i.e. neurons that never activate across the entire training dataset) if the learning rate is set too high.
    - With a proper setting of the learning rate this is less frequently an issue.

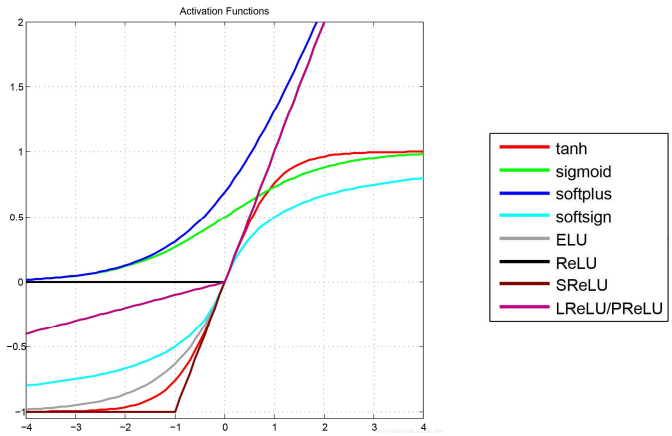
Activation Function

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

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



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

Good Initialization

Greedy Layer-wise Training

- **Fully-Connected Network**
  - Unsupervised pre-training
    - Greedy layer-wise training
    - Dimension reduction
    - Feature extraction
    - Providing Good Initial Weights to the next supervised learning step
  - Supervised fine tuning
    - Same as usually supervised learning
    - Whole weights are modified by gradient descent methods

Greedy Layer-wise Training

- **Curse of Dimensionality**
  Facial Expression Recognition
  - 256x256 image, 65,536 dimension vector
  - Is to good for facial recognition?
  - Maybe, a lower dimension vector is ENOUGH!!

Greedy Layer-wise Training

- **Curse of Dimensionality**
  Facial Expression Recognition
  - Define some facial keypoints
  - Use the distances between them as features
  - 60 dimension vector is enough

Greedy Layer-wise Training

- **Dimension Reduction**
  - Or, Feature Extraction

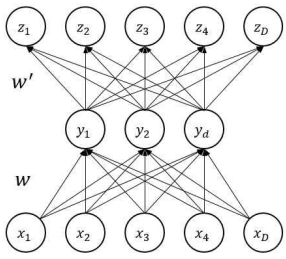
⇨

65,536 dimension

60 dimension

Greedy Layer-wise Training

- **Autoencoder**
  - A way of dimension reduction



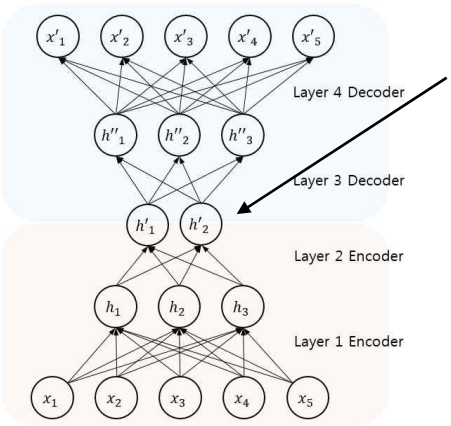
Decoder : $\mathrm{z} = g_{\theta'}(y) = \ s(W'y + b')$

Encoder : $y = f_\theta(x) = s(Wx + b)$

Values of hidden nodes can be interpreted as a low dimensional representation of the input

Greedy Layer-wise Training

- **Stacked Autoencoder (Deep Network)**



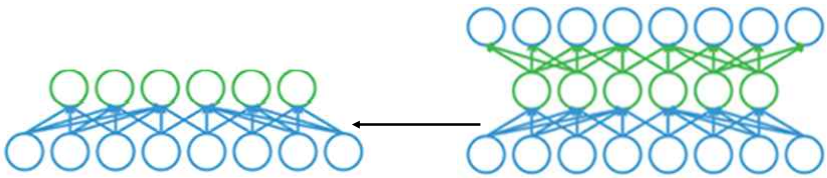
A very low dimensional representation of the input

How to train the network coping with VANISHING GRADIENTS?

Greedy Layer-wise Training

- **Greedy Layer-Wise Training**

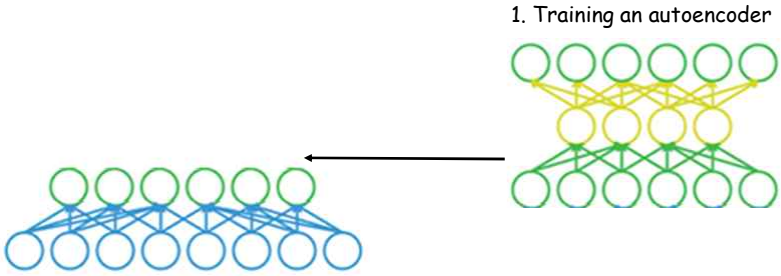
1. Training an autoencoder



2. Copy connection weights

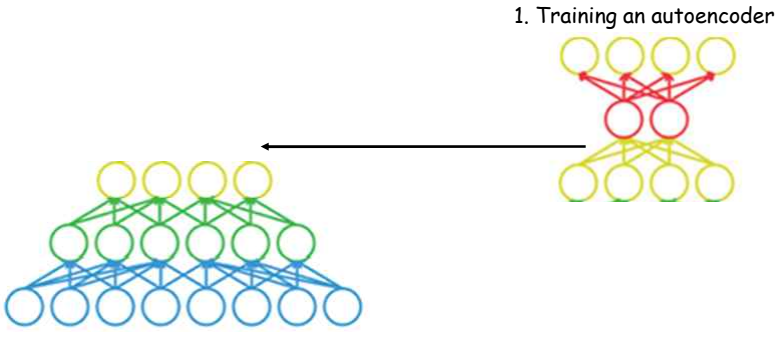
Greedy Layer-wise Training

- **Greedy Layer-Wise Training**



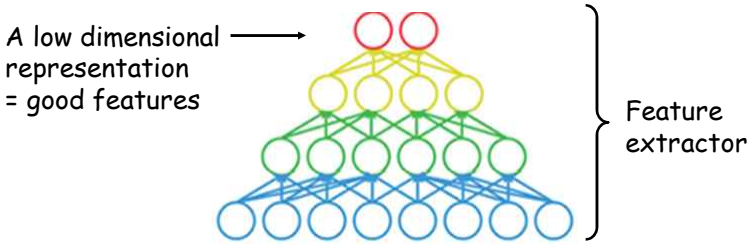
Greedy Layer-wise Training

- **Greedy Layer-Wise Training**



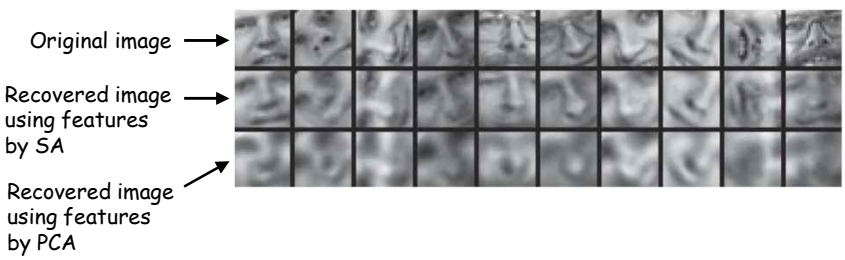
Greedy Layer-wise Training

- **Unsupervised Pre-training**
  - Feature extractor



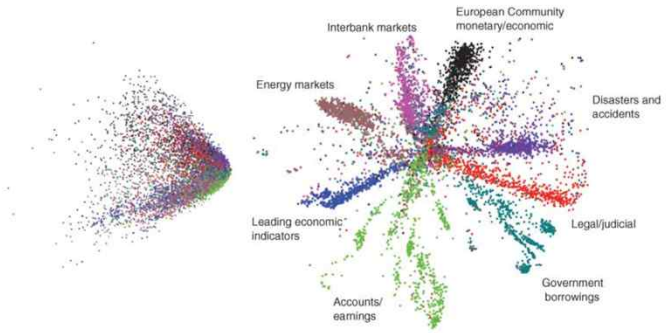
Greedy Layer-wise Training

- **Unsupervised Pre-training**
  - Feature extractor



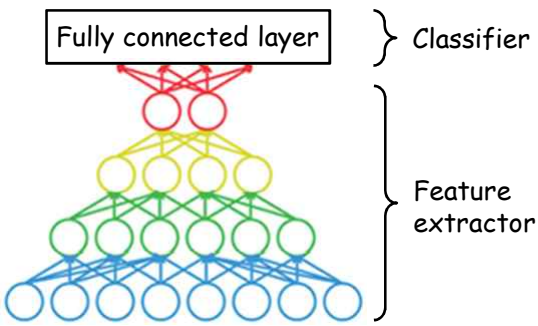
Greedy Layer-wise Training

- **Unsupervised Pre-training**
  - Documents



Greedy Layer-wise Training

- **Supervised Fine Tuning**
  - Stacking another fully connected layers
  - Training all the weights



Greedy Layer-wise Training

- **Types**
  - Deep Belief Networks (DBN)
  - Stacked Autoencoder (SA)
  - Stacked Denoising Autoencoder (SdA)
  - Sparse Autoencoder

Dropout

Regularization

- **What is Regularization**
  - Introducing additional information to prevent over-fitting
- **Approaches**
  - Early stopping, Max norm constraints, Weight decay
    Dropout, DropConnect, Stochastic pooling

Regularization

- **Early Stopping**
  - Stop learning before converging too much
  - Hard to determine the time to stop
  - Usually determined by validation dataset
- **Max norm constraints**
  - Enforcing weight vectors not to grow over given constant *c*
  - If it grows over *c*, cut its length to *c*.

Regularization

- **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.

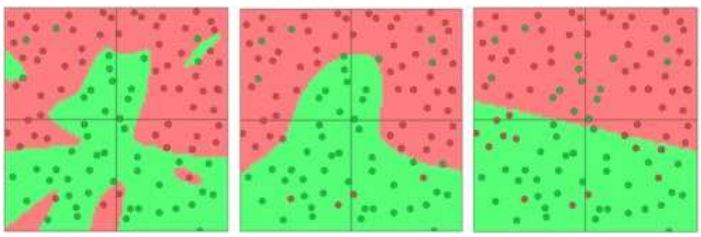
$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}|\mathbf{w}|$$

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

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

Regularization

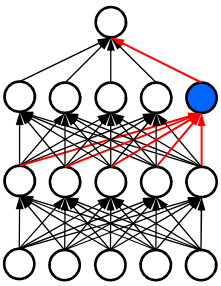
- **Weight Decay**
  - Example: Separating green and red



L2 regularization strengths of 0.01, 0.1, and 1

Dropout

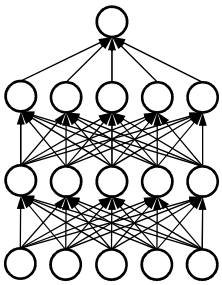
- **In a complex Neural Network**
  - All nodes do not take the same amount of responsibility
    - While training, some nodes are correlated
  - All nodes are not equally trained
    - Some nodes are trained much, but some are not

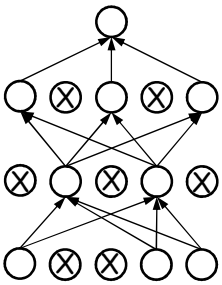


- If the output of the node is bad, the connection weight will decrease.
- If connection weight is close to 0, precedent connection weights are hardly trained.

Dropout

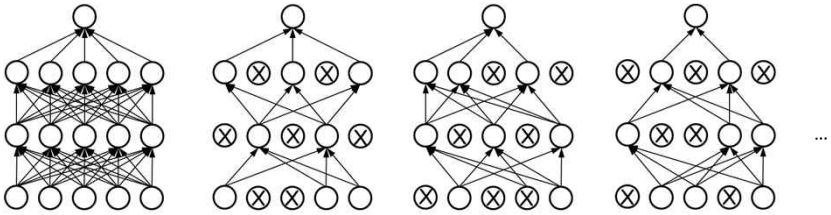
- **How can we reduce the structural complexity?**
  - Let's simply remove some nodes, and
  - Train the simplified neural network
  - Hmm??





Dropout

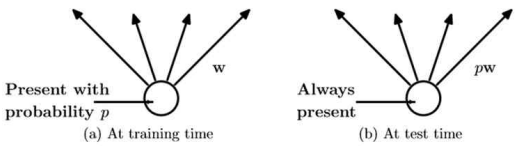
- **Do this at every epoch**
  - Randomly choose nodes with a probability of *p*
    - Usually p = 0.5
  - Train the simplified neural network
    - At every epoch, we train different neural network which share connection weight each other



Original network | Epoch 1 | Epoch 2 | Epoch 3 | ...

Regularization

- **Testing**
  - Use all the nodes without dropout
  - Instead, the connection weight is scaled by p
    - Why?

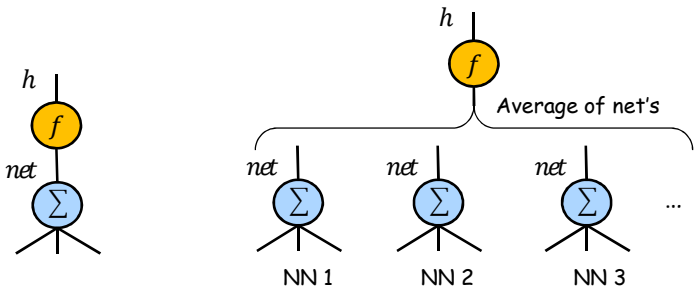


(a) At training time (b) At test time

Dropout

- **Testing**
  - Use all the nodes without dropout
  - Instead, the connection weight is scaled by p
    - Why?

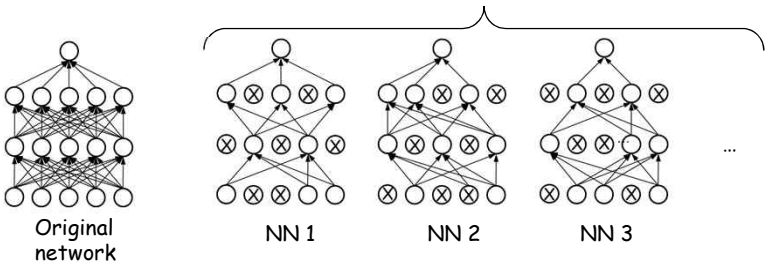
=> Some kind of averaging all the different neural networks



Dropout

- **Testing**
  - Use all the nodes without dropout
  - Instead, the connection weight is scaled by p
    - Why?

=> Some kind of averaging all the different neural networks



Dropout

- **Experimental Results**
  - SVHN – Street View House Numbers
    - Dropout is applied also to convolutional layers.
    - All hidden units are ReLUs.

Method	Error %
Binary Features (WDCH) (Netzer et al., 2011)	36.7
HOG (Netzer et al., 2011)	15.0
Stacked Sparse Autoencoders (Netzer et al., 2011)	10.3
KMeans (Netzer et al., 2011)	9.4
Multi-stage Conv Net with average pooling (Sermanet et al., 2012)	9.06
Multi-stage Conv Net + L2 pooling (Sermanet et al., 2012)	5.36
Multi-stage Conv Net + L4 pooling + padding (Sermanet et al., 2012)	4.90
Conv Net + max-pooling	3.95
Conv Net + max pooling + dropout in fully connected layers	3.02
Conv Net + stochastic pooling (Zeiler and Fergus, 2013)	2.80
Conv Net + max pooling + dropout in all layers	2.55
Conv Net + maxout (Goodfellow et al., 2013)	**2.47**
Human Performance	2.0

Dropout

- **Experimental Results**
  - CIFAR-10 and CIFAR-100:
  - Images drawn from 10 and 100 categories respectively.

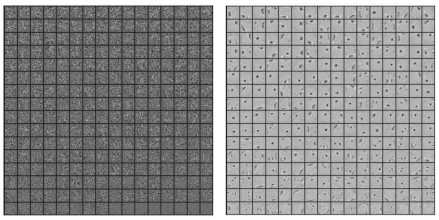
Method	CIFAR-10	CIFAR-100
Conv Net + max pooling (hand tuned)	15.60	43.48
Conv Net + stochastic pooling (Zeiler and Fergus, 2013)	15.13	42.51
Conv Net + max pooling (Snoek et al., 2012)	14.98	-
Conv Net + max pooling + dropout fully connected layers	14.32	41.26
Conv Net + max pooling + dropout in all layers	12.61	**37.20**
Conv Net + maxout (Goodfellow et al., 2013)	**11.68**	38.57

Dropout

- **The effect of dropout on learned features:**
  - Without dropout, units tend to compensate for mistakes of other units.
  - This leads to overfitting, since these co-adaptations do not generalize to unseen data.
  - Dropout prevents co-adaptations by making the presence of other hidden units unreliable.

MNIST, one hidden layer, 256 ReLUs

No Dropout

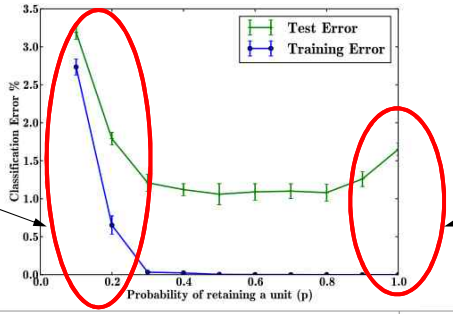


Dropout p=0.5

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).

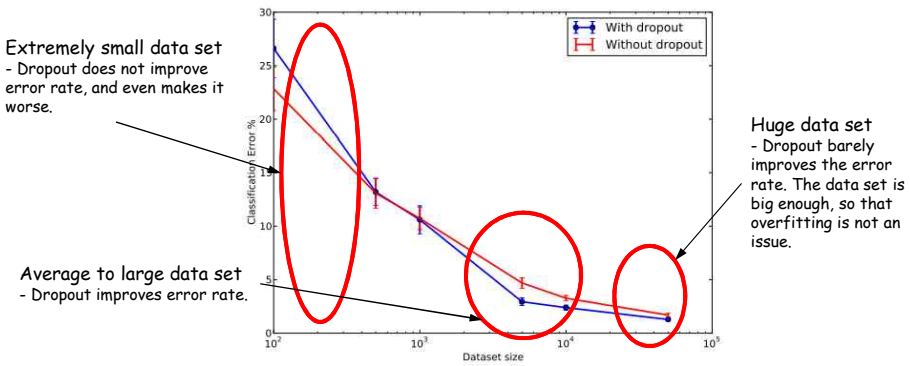
Dropout with high probability



No Dropout ->Overfitting

Dropout

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



Dropout

- **Limiting the growth of the weights in the network.**
  - A term is added to the original loss function, penalizing large weights:

$$\mathcal{L}_{new}(\boldsymbol{w}) = \mathcal{L}_{old}(\boldsymbol{w}) + \frac{1}{2}\lambda\|\boldsymbol{w}\|_2^2$$

  - A network architecture of 784-1024-1024-2048-10 is used on the MNIST data set, with different regularizations.

Method	Test Classification error %
L2	1.62
L2 + L1 applied towards the end of training	1.60
L2 + KL-sparsity	1.55
Max-norm	1.35
Dropout + L2	1.25
Dropout + Max-norm	**1.05**

Dropout

- **Dropout has more advantages over weight decay (Helmblod et al., 2016):**
  - Dropout is scale-free: dropout does not penalize the use of large weights when needed.
  - Dropout is invariant to parameter scaling: dropout is unaffected if weights in a certain layer are scaled up by a constant c, and the weights in another layer are scaled down by a constant c. This implies that dropout does not have an isolated local minima.

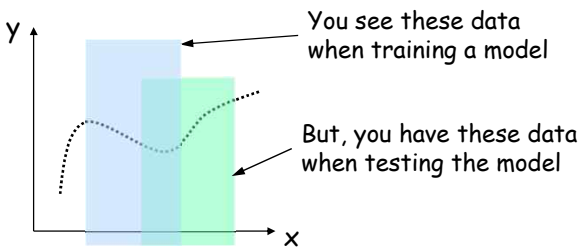
Dropout

- **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

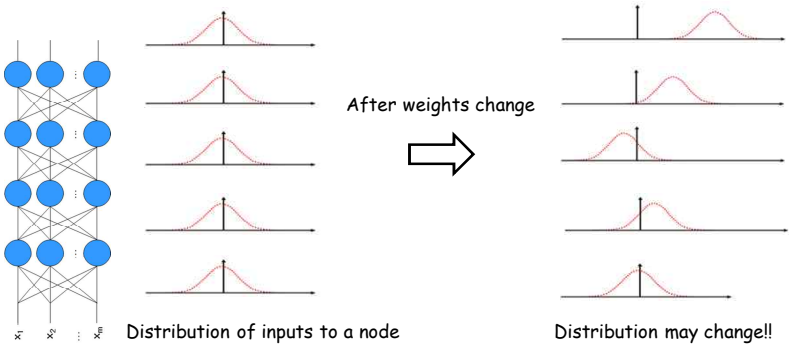
- **Covariate Shift**
  - A change in the distribution of a function's domain.



  - Can your model work properly?

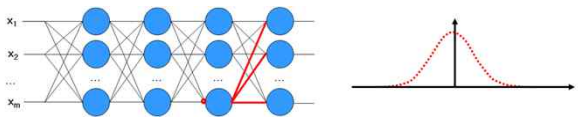
Batch Normalization

- **Internal Covariate Shift**
  - The distribution of the input to a layer changes if you change the connection weights
  - This change propagates to the upper layers

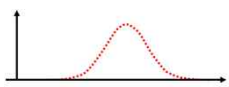


Batch Normalization

- **Internal Covariate Shift**
  - Input distribution of the red node



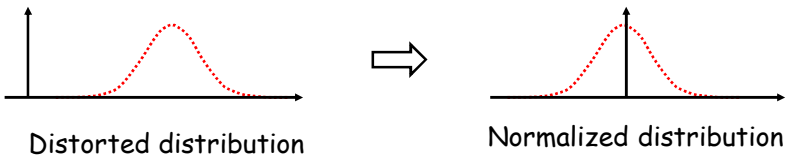
  - While learning, red connection weights will change based on the input distribution
  - After learning, the whole connection weights changes, which cause the change of the input distribution



  - The assumption of the learning is broken

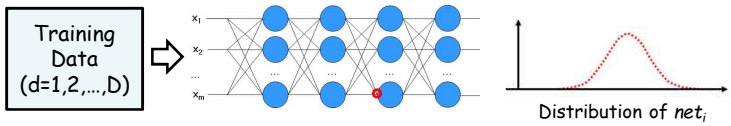
Batch Normalization

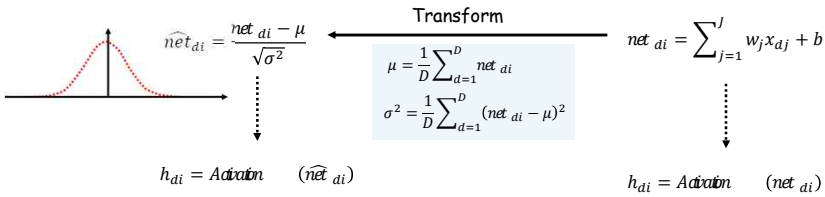
- **Internal Covariate Shift**
  - It disturbs the learning process,
  - Learning is getting slow down
- **What shall we do?**
  - Why don't we normalize the distribution of inputs



Batch Normalization

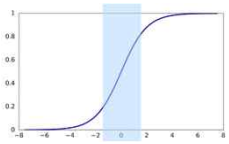
- **Input Normalization**
  - Let's assume that the input distribution of any nodes is Gaussian





Batch Normalization

- **Input Normalization**
  - Calculating the average and the variance of the whole training data is inefficient
    - Let's normalize inputs with respect to each batch
  - If we use Sigmoid or Tanh as the activation function, non-linearity disappear

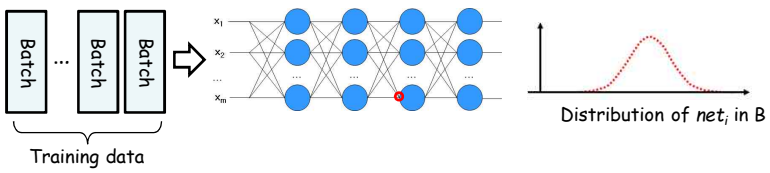


Behavior of Sigmoid around 0
is almost linear!!

    - Let's add linear transform after normalization

Batch Normalization

- **Input Normalization**
  - Let's assume that the input distribution of any nodes is Gaussian



$$\widehat{net}_{di} = \frac{net_{di} - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \quad \text{Transform} \quad net_{di} = \sum_{j=1}^{J} w_j x_{dj} + b$$

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

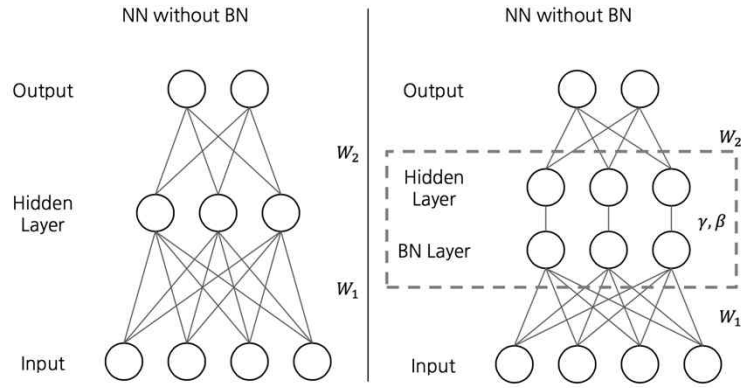
$$\sigma_B^2 = \frac{1}{D_B}\sum_{d=1}^{D_B} (net_{di} - \mu)^2$$

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

$$h_{di} = Activation\ (y_{di}) \qquad h_{di} = Activation\ (net_{di})$$

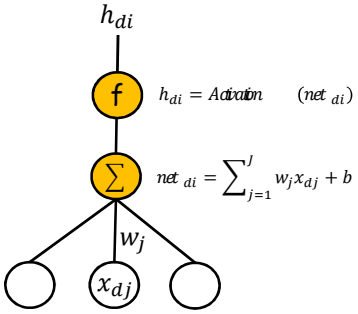
Batch Normalization

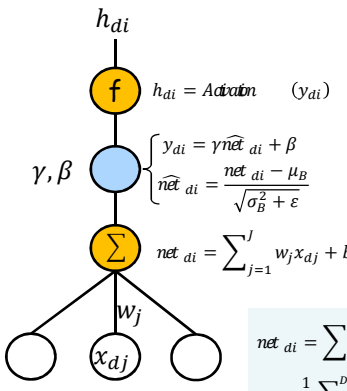
- **Structure of Networks**



Batch Normalization

- **For a Single Node**

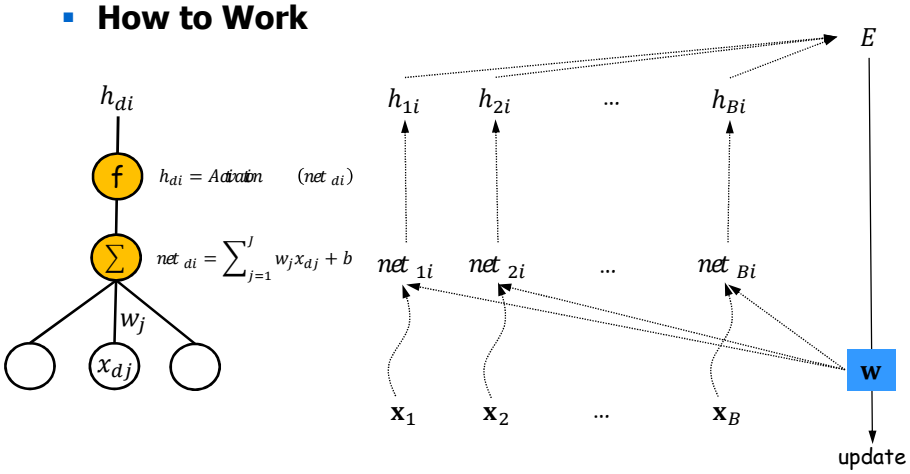




$$net_{di} = \sum_{j=1}^{J} w_j x_{dj} + b$$
$$\mu_B = \frac{1}{D_B} \sum_{d=1}^{D_B} net_{di}$$
$$\sigma_B^2 = \frac{1}{D_B} \sum_{d=1}^{D_B} (net_{di} - \mu)^2$$

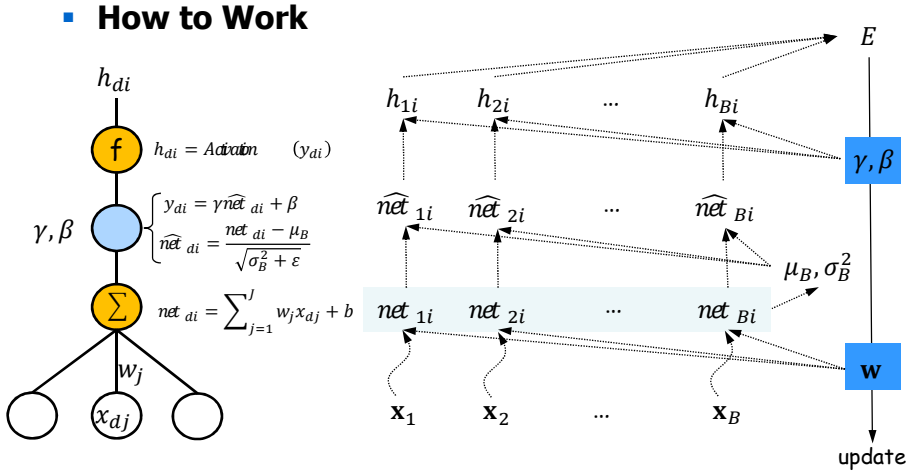
Batch Normalization

- **How to Work**



Batch Normalization

- **How to Work**



Batch Normalization

- **Input Transform**
  - Normalize each batch by both mean and variance.
  - For each node, normalization performed independently

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

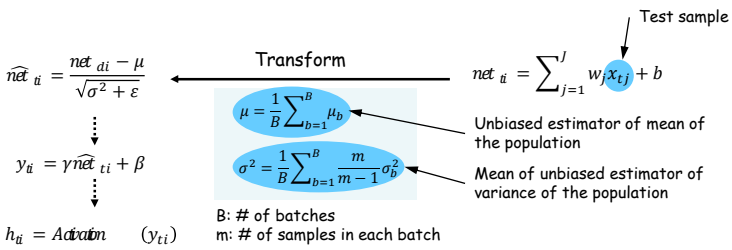
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

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

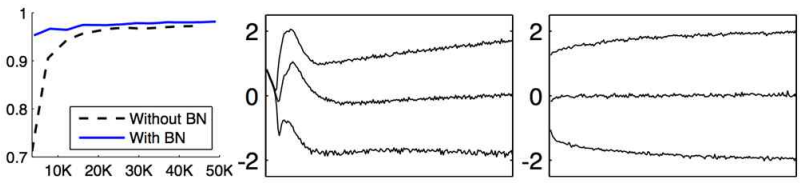


Batch Normalization

- **Advantage**
  - Reduces internal covariant shift.
  - Reduces the dependence of gradients on the scale of the parameters or their initial values.
  - Regularizes the model and reduces the need for dropout, photometric distortions, local response normalization and other regularization techniques.

Batch Normalization

- **Performance with BN**

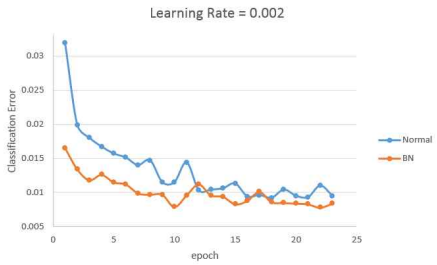


(a) *accuracy* (b) Without BN (c) With BN

input distributions

Batch Normalization

- **Advantage**
  - Learning faster: Learning rate can be increased compare to non-batch-normalized version.
  - Not-Stuck in the saturation mode: Even if ReLU is not used.
  - Higher accuracy: Flexible on data distribution in every hidden layer



Summary

- **Deep networks has strong power but cause many problems**
- **Solutions for Training**
  - High Model Complexity
    - Adopting deep but relatively simple structures
  - Hard to Optimize
    - Using another activation functions
    - Mini-batch gradient descent
  - Overfitting
    - (A huge amount of data)
    - Various Regularization Method (including Dropout)
  - Internal Covariate Shift
    - Batch normalization