Deep Learning
Part 3: Concepts & DBNNs

Nagiza F. Samatova, samatova@csc.ncsu.edu
Professor, Department of Computer Science
North Carolina State University

Senior Scientist, Computer Science & Mathematics Division
Oak Ridge National Laboratory

OAK RIDGE National Laboratory

NC STATE UNIVERSITY Department of Computer Science

Outline

- Core components of deep learning
  - Semi-supervised learning: partially labeled data
  - Rectified Linear Units as Interior Activation Functions
  - Dropout & Regularization
- Deep Belief Neural Networks (DBNN)
  - Restricted Boltzmann Machines
  - Training a DBNN
  - Layer-wise Sampling
  - Gibbs Sampling
  - Updating Weights & Biases
  - DBNN Backpropagation
  - DBNN Applications

A little bit of history

- Deep learning at the simplest level
  - A neural network with two or more layers
- Pitts (1943) introduced multilayer perceptron
  - Which is the ability to create deep neural networks
- Hinton (1984)
  - Enabled training such complex neural networks

Core Components of Deep Learning

A. Partially Labeled Data
B. Rectified Linear Units (ReLU) as Activation Function
C. Convolutional Neural Networks:
   - Sparse connectivity
   - Shared weights
D. Neuron Dropouts:
   - Regularization to prevent overfitting

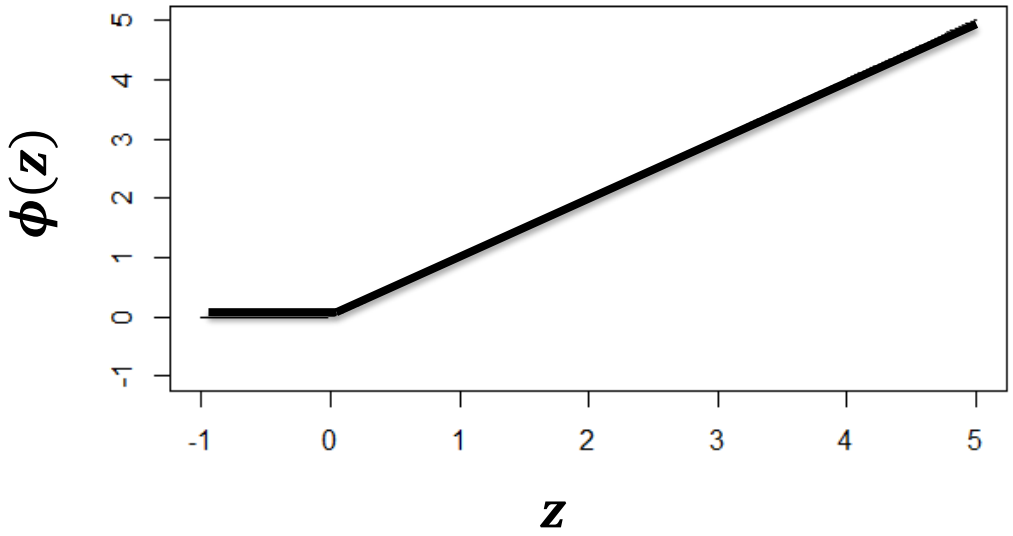
A. Partially Labeled Data

- **Deep Learning supports semi-supervised learning**
  - **Handle a combination of supervised & unsupervised data**
- **Deep Learning Architectures**
  - **Unsupervised Phase: Initialize weights by using the entire training data without outcomes**
    - This way individual layers could be trained without labels
    - This enable parallel training → scalability
  - **Supervised Phase: Tweak and optimize the weights**

Refresher: Rectified Linear Units (ReLU)

$$f(x_i, w_i) = \phi\left(\sum_i (w_i \times x_i)\right)$$

$$\phi(z) = max\,(0, z)$$



- Introduced by Teh & Hinton (2000)
- Highly recommended for getting superior training results:
  - Because it is a linear, non-saturating function
  - Unlike sigmoid/logistics or hyperbolic tangent AFs, it does not saturated to -1, 0, or 1
- NNs should utilize ReLUs on hidden layers and linear / softmax functions on the output layer

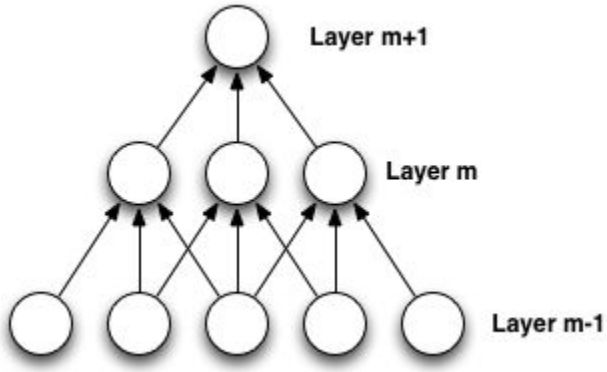
B. ReLU, Linear & Softmax AFs

- ReLU Activation Function (AF) for the Hidden Layers
  - Standard AF for the hidden layers of a deep NN
- AF for the Output Layer
  - Linear AF for regression
  - Softmax AF for classification

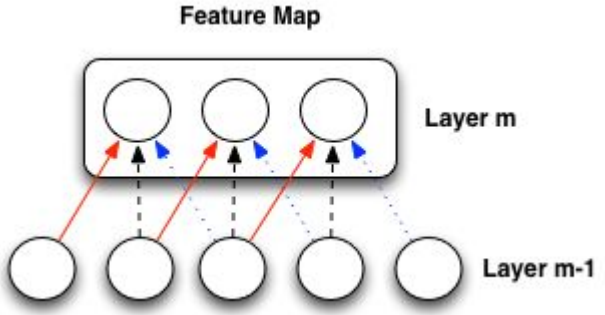
C. Convolutional Neural Networks (CNN)

- **Hinton (2014) introduced convolution**
  - **Allow for sparse connectivity:** do not create every possible weight (deep layers) as in feedforward NN
  - **Allow for sharing the weights:** to store complex structure while maintaining lower memory profile and getting computational efficiency

Sparse Connectivity



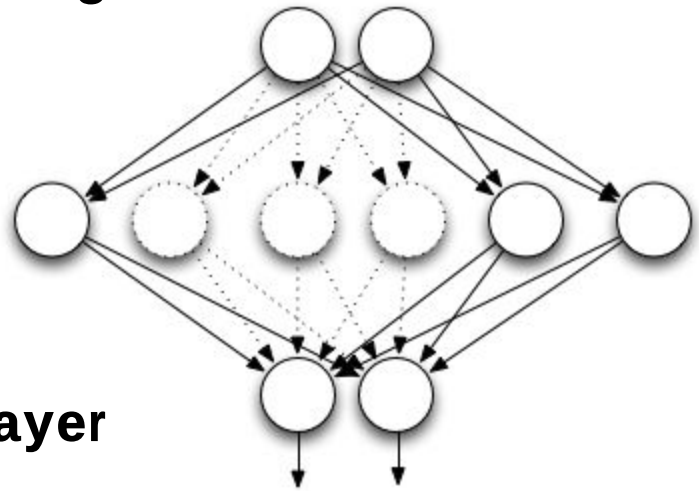
Shared Weights



edges of the same color share the weights

D. Dropouts: Neurons

- Hinton (2002) introduced dropout as a regularization algorithm to reduce overfitting
- Dropout
  - A regularization technique for neural networks
  - Dropout can prevent overfitting
- Application of neuron dropouts
  - Drop certain neurons in the dropout layer → connectors to/from the eliminated neurons are also removed
  - Layer-by-layer like in CNNs
  - Do NOT mix the dropout and convolutional layers within the same layer



So Why Deep Learning Works?

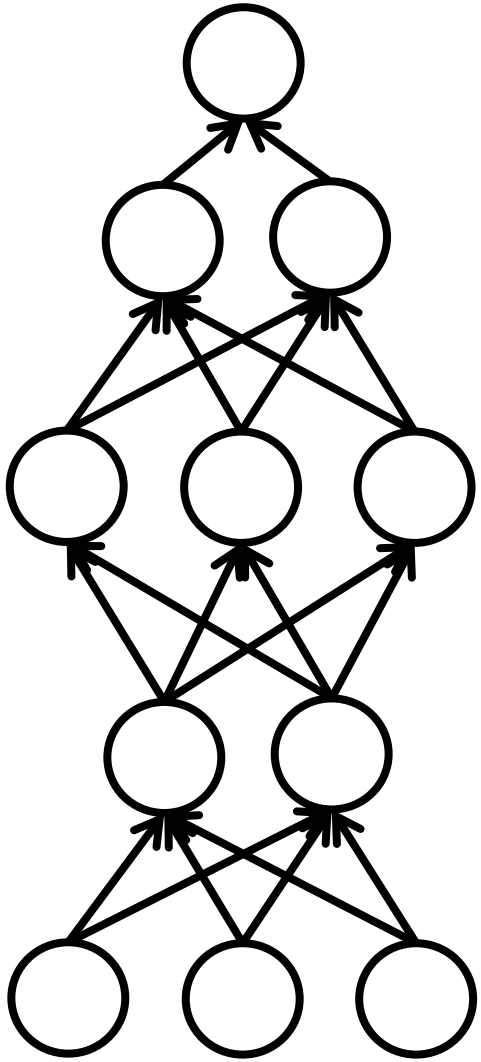
- It leverages the deep architecture to learn hierarchical structure within the data through a unsupervised and simple manor.
  - The hierarchical structure can be considered as well-defined features in the data, which usually requires manual crafting
  - The unsupervised approach in deep learning is an pipeline to learn the hierarchical structure automatically
- Multilayered Neural Network is just one way to achieve it

TUNING DEEP NEURAL NETWORK

Refresher– Why Deep Neural Network?

Leverage the deep architecture to learn hierarchical structure within the data through a unsupervised and simple manor.

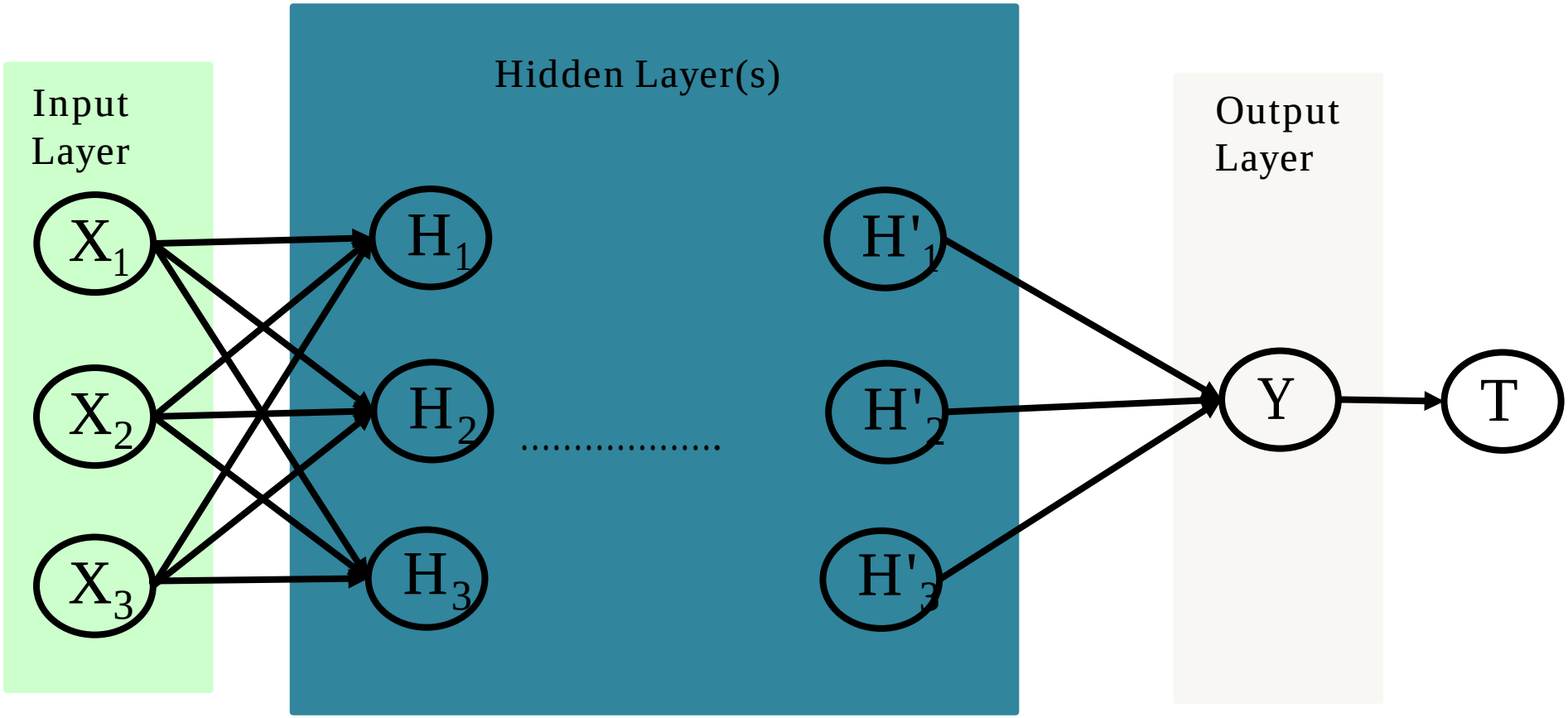
So we just add more layers, and speed up our machine and that's it?



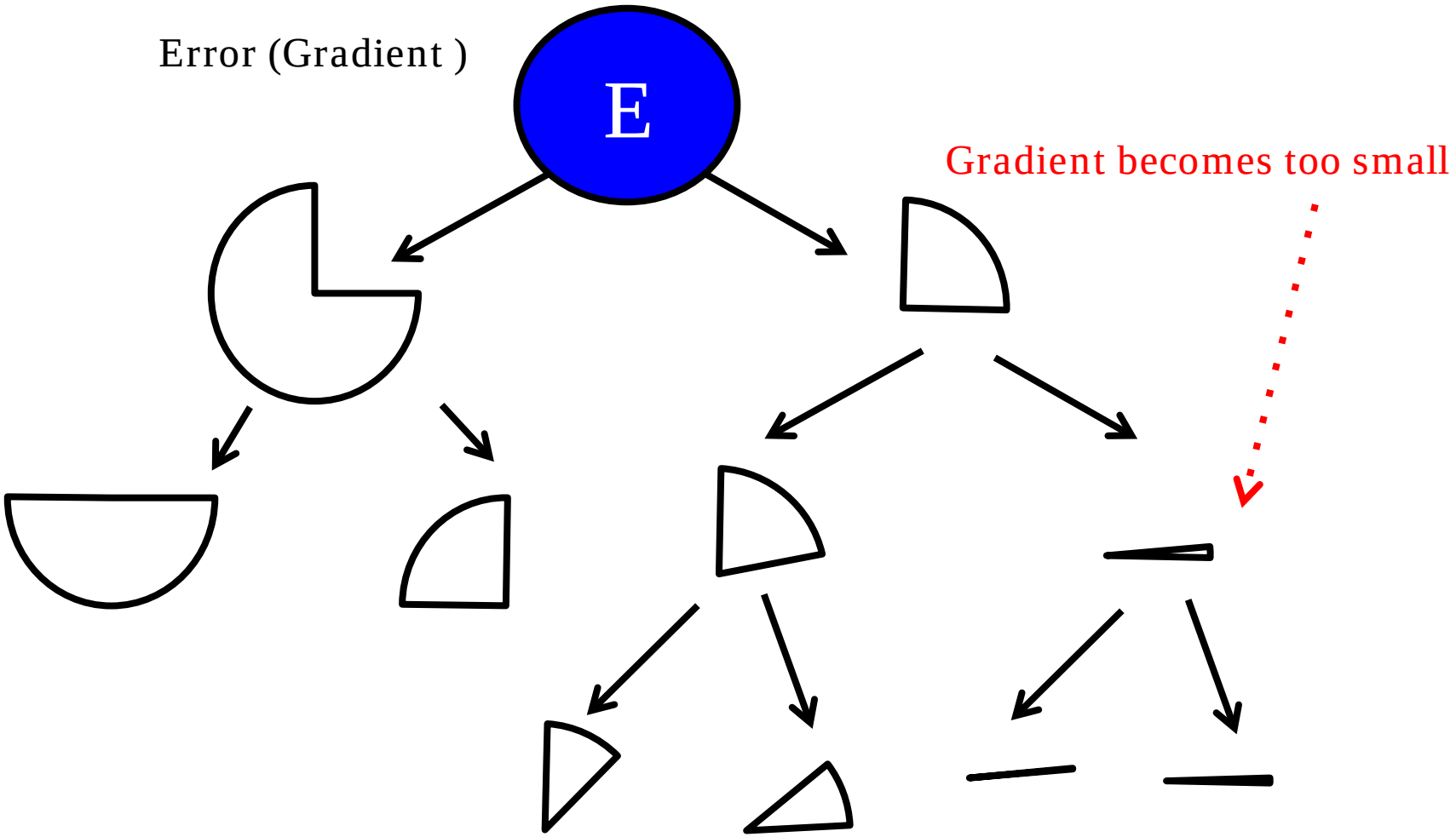
Back Propagation On Multiple Layers

Is it the same to train the weight matrices for deep hidden layers?

$$dE / d\,WRS_{ij} = ?$$



What Happens at Multi-Layers NN



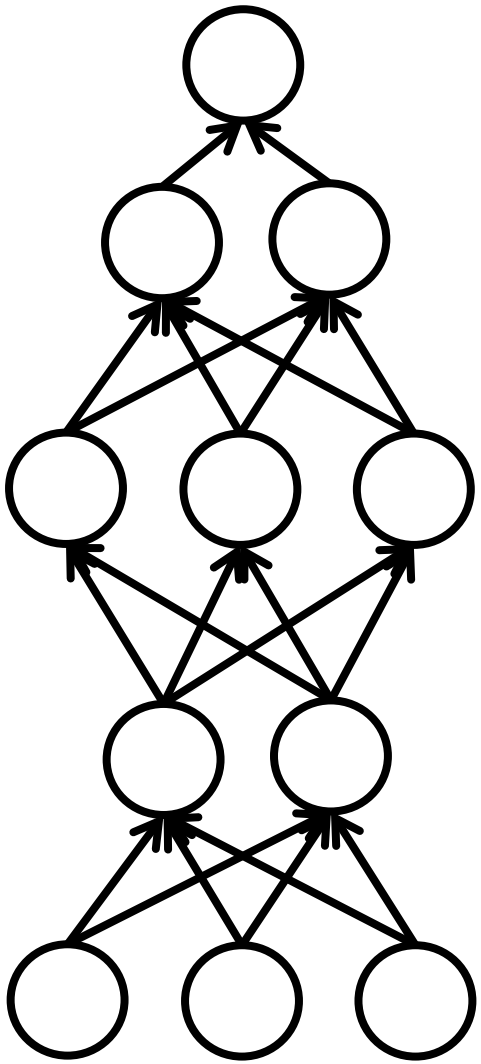
When It Goes Deep – Gradient Vanished

Vanishing Gradient:

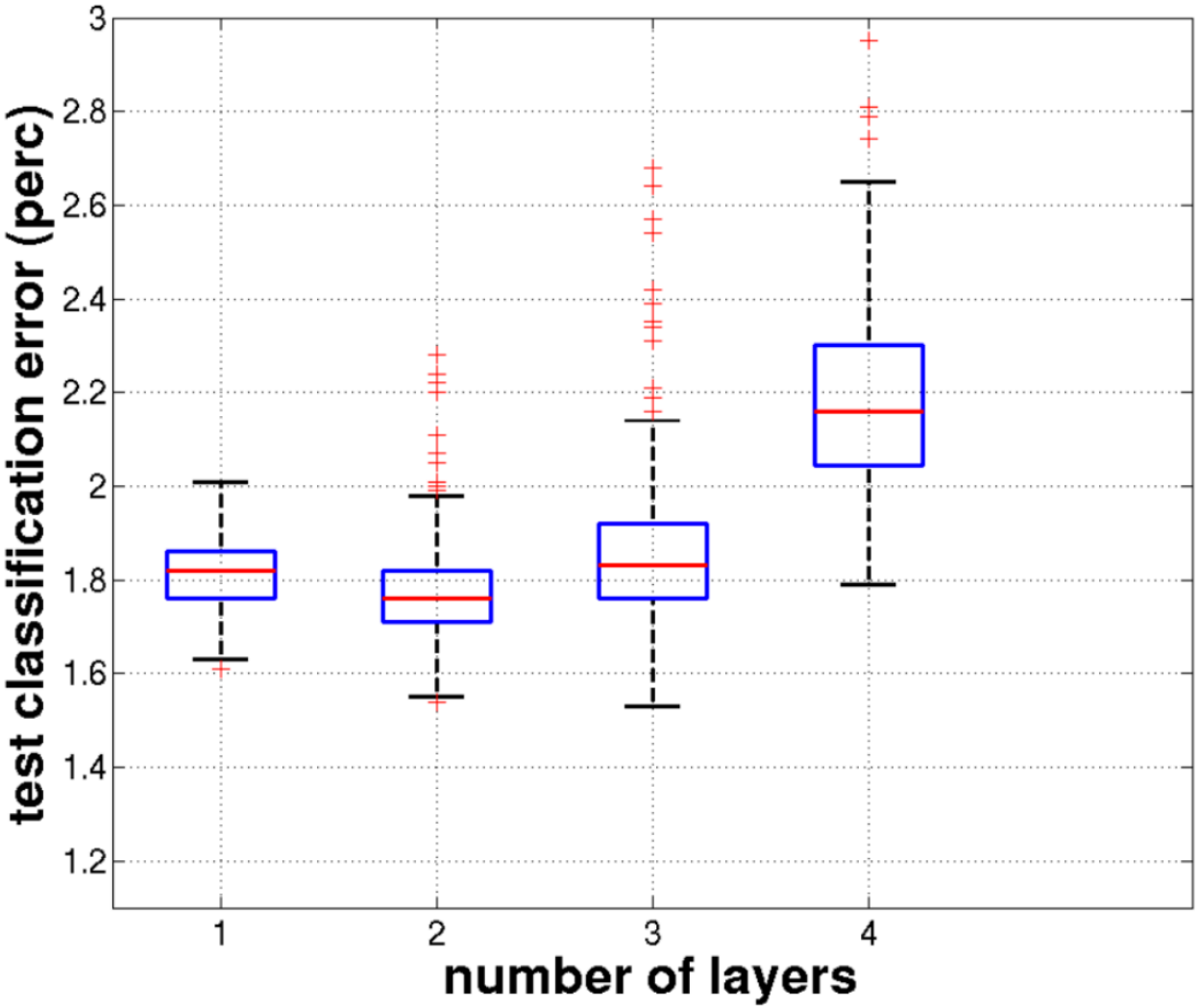
When there are many hidden layers in the neural network model (>3), the power of back propagation diminishes as the error travels through layers.

Problems:

1. The model updates slow
2. Tend to trap at local optimum
3. Biased toward early samples



Deep Does Not Equal To Better Results



Solving Vanishing Gradient Problem

This problem mainly raises because the weights in neural networks are usually initialized randomly. Accordingly, there are two methods to solve it:

1. Use domain knowledge to create very good initial weights for the network <- Which is very difficult

2. Pre-Train the model through unsupervised neural network models. <- (Stacked) Auto Encoder

Our key to get deep

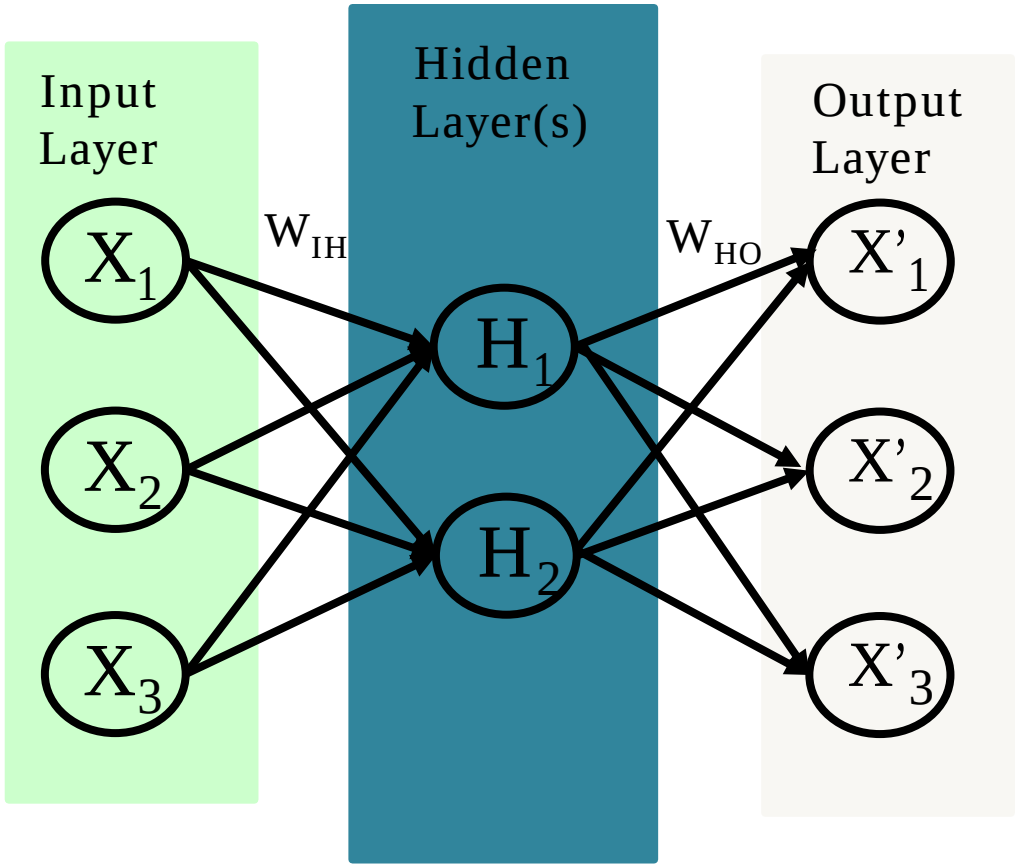
AUTO ENCODER

Auto Encoder

Auto Encoder is a 3 layered (Input, Hidden, Output) **unsupervised** neural network. The goal of AutoEncoder is to learn a good representation of the input data.

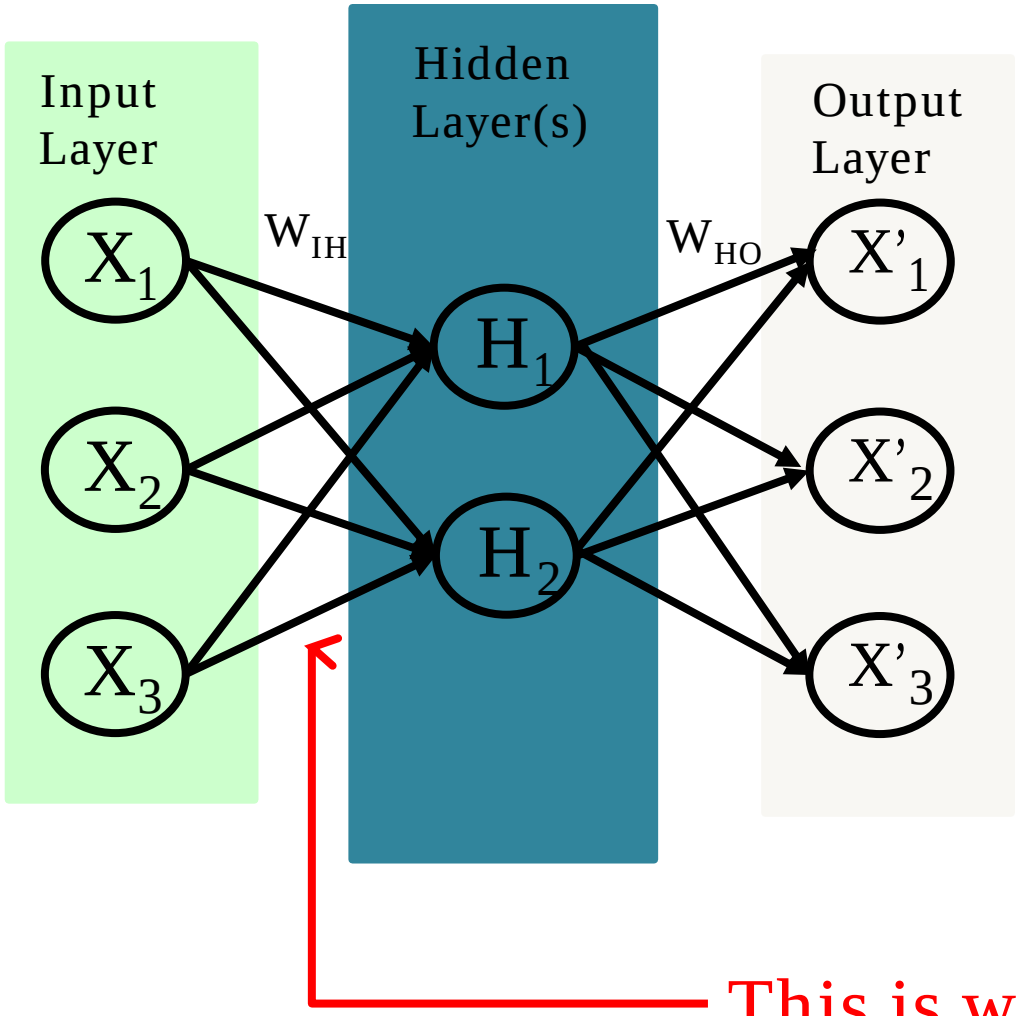
In order to facilitate training, AutoEncoder will test the goodness of the representation it learned by remapping it back to the input. The training goal is to close the gap between the outputs and the inputs.

Auto Encoder – Closing the Gap



If X_1 and X_1' are really close, then it means the hidden layers are capturing a good idea of X_1

Auto Encoder – Goal Function



$$Xi' = \sum_{1}^{I'} \left(\sum_{1}^{J} \sum_{1}^{I} X_i W_{ij}^{IH} + bj \right) * W_{JI'}^{HO} + b'I'$$

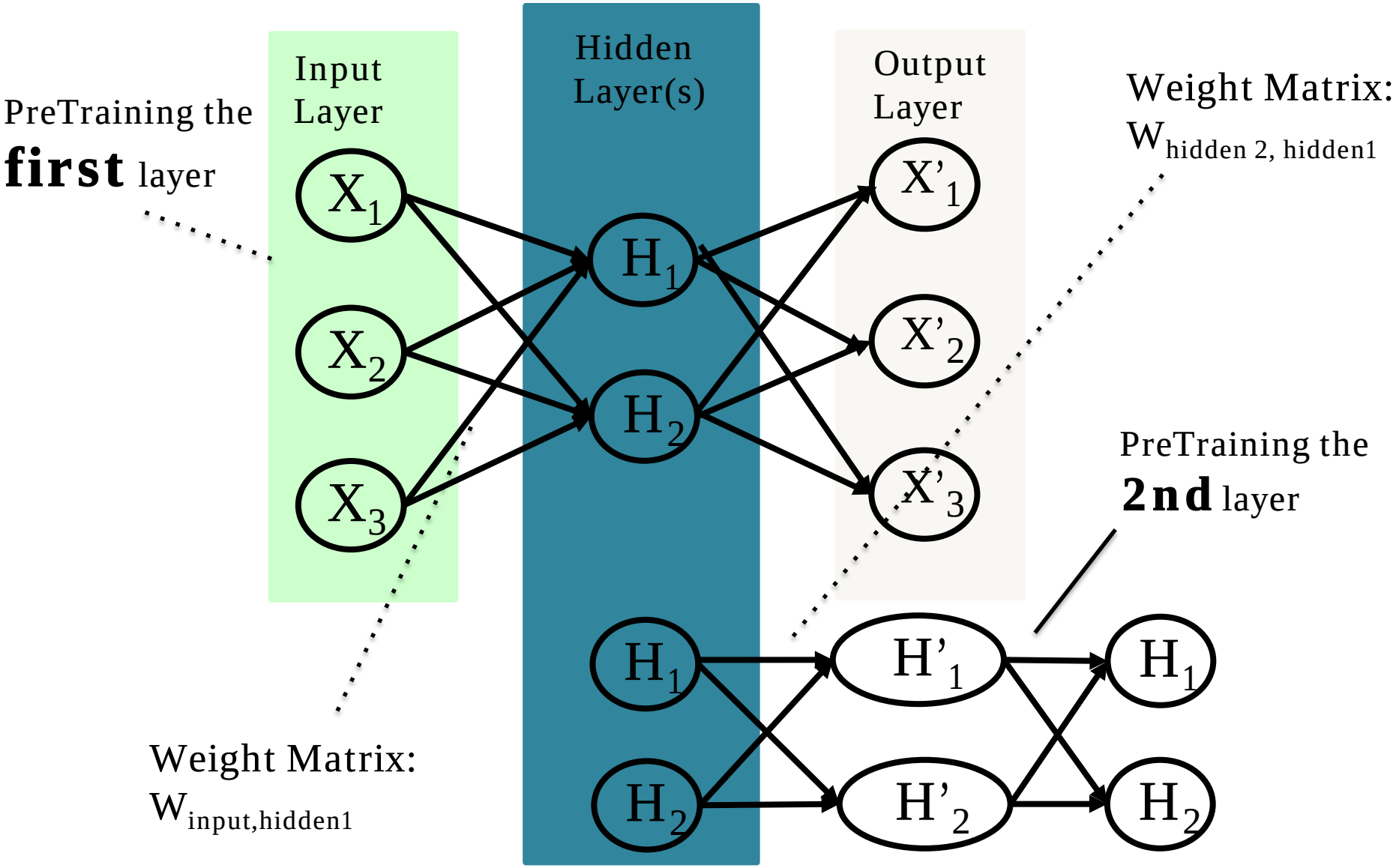
Goal: Min $\Sigma(Xi' - Xi)2$

Stacking Auto Encoders

Auto Encoder is an unsupervised method, and can be trained layer by layer, so we have the freedom to stack unlimited Auto Encoders

This feature allows deep network to work because we can initialize good weights for deep network, which solves vanishing gradient problem.

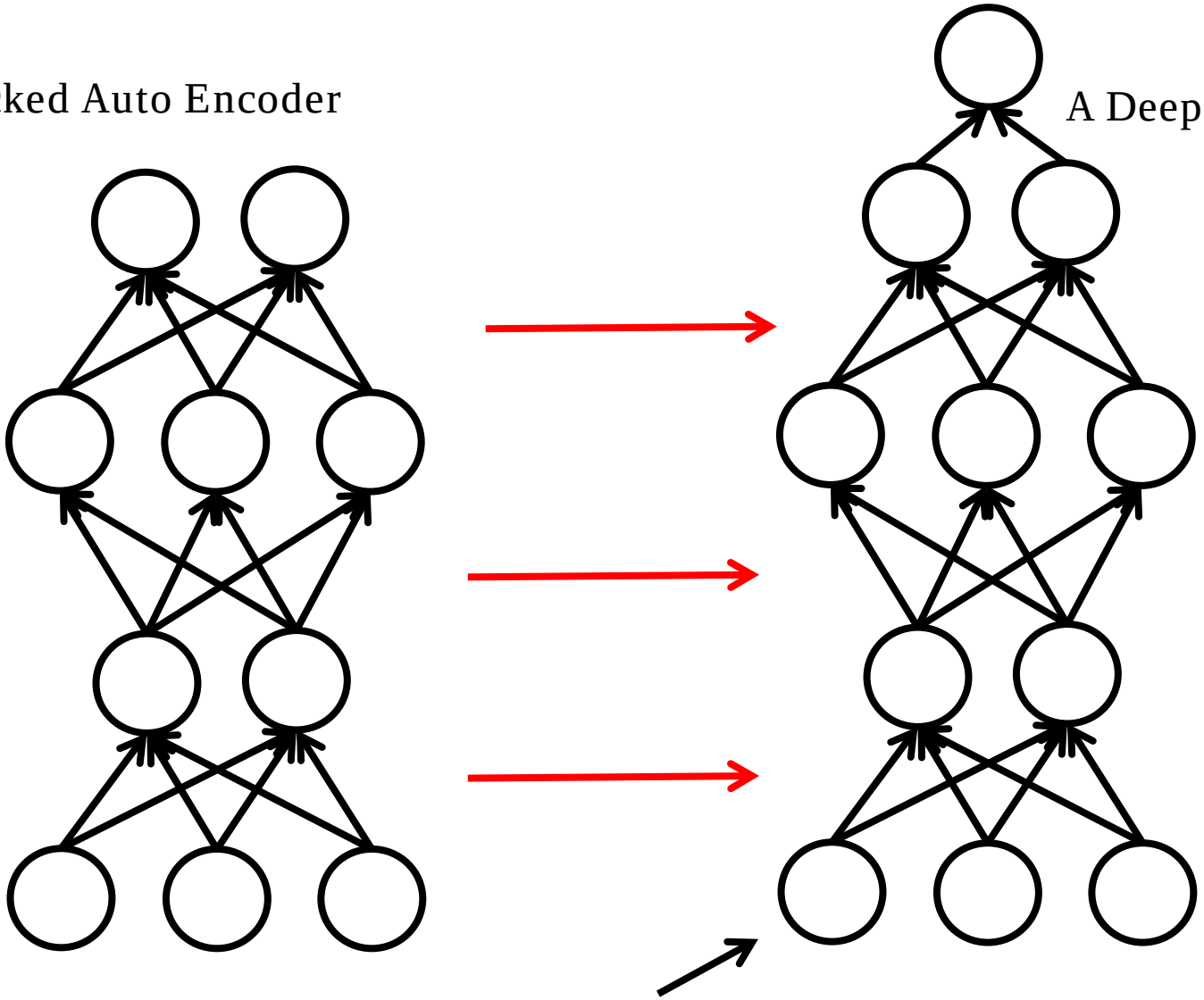
Stacked Auto Encoder - Example



Combine Auto Encoder with NN

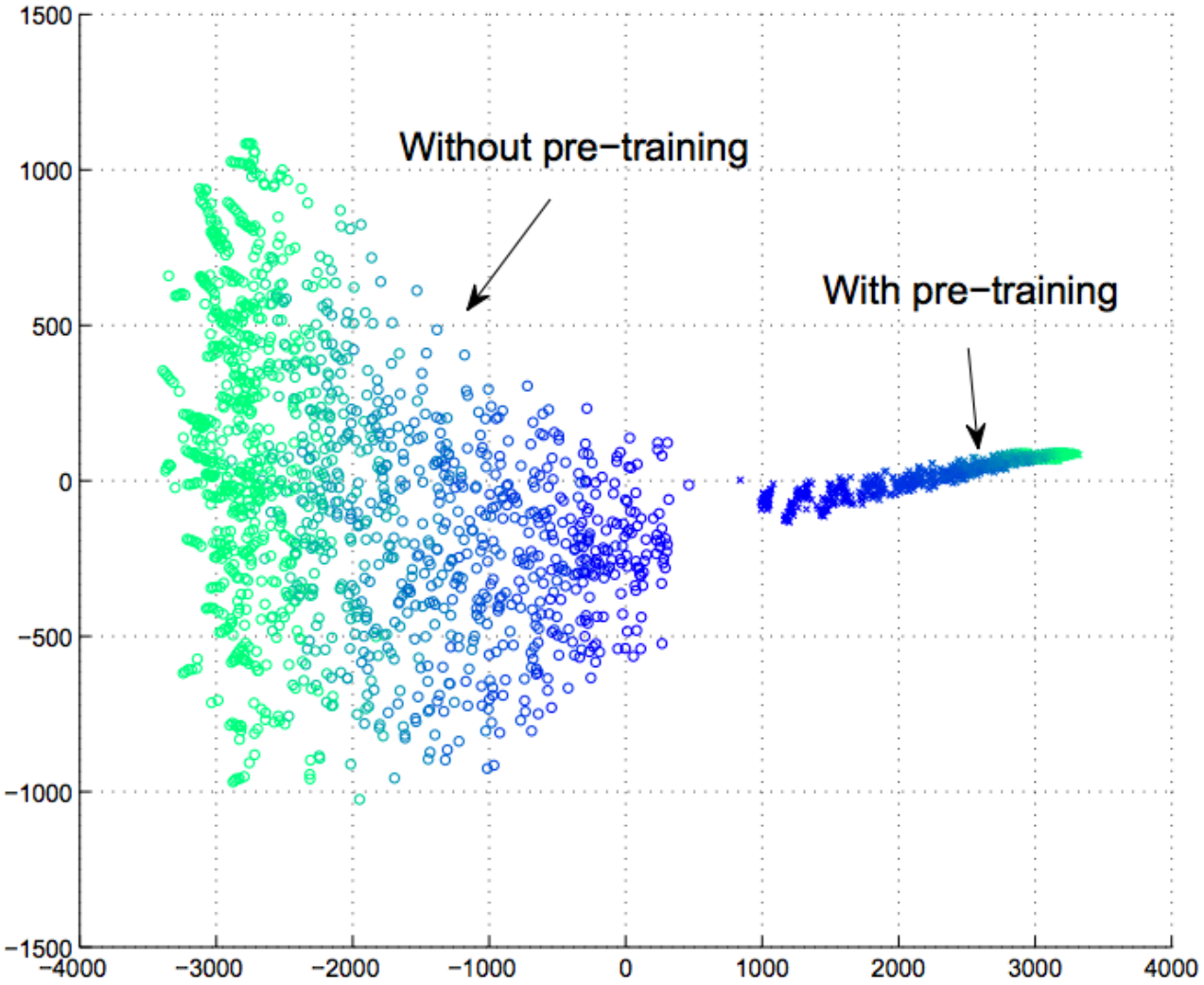
Deep Stacked Auto Encoder

A Deep Neural Network

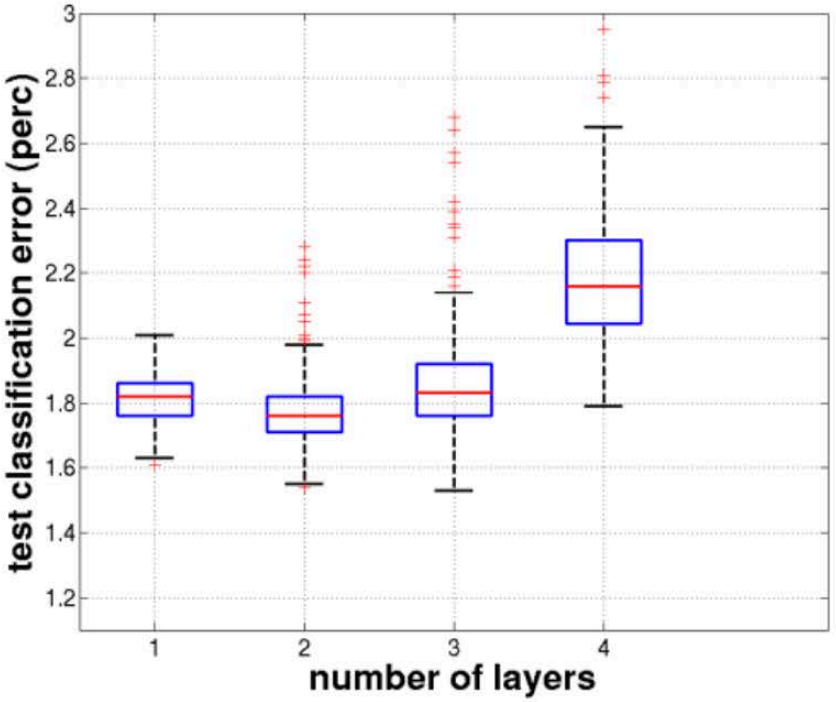


Sometimes we call this Fine Tuning

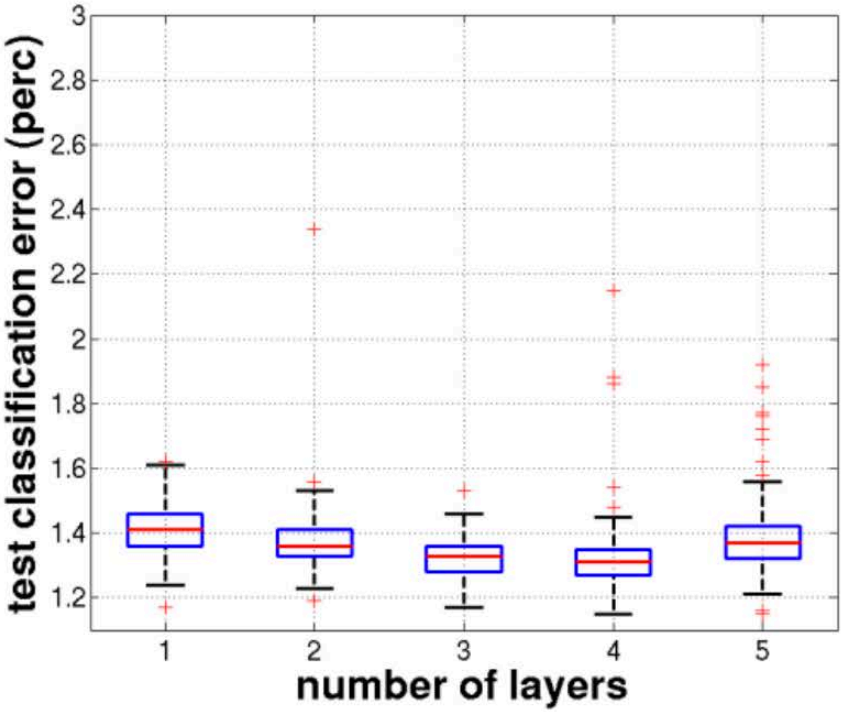
Effect of Pre-Training



Effect of Pre-Training -2

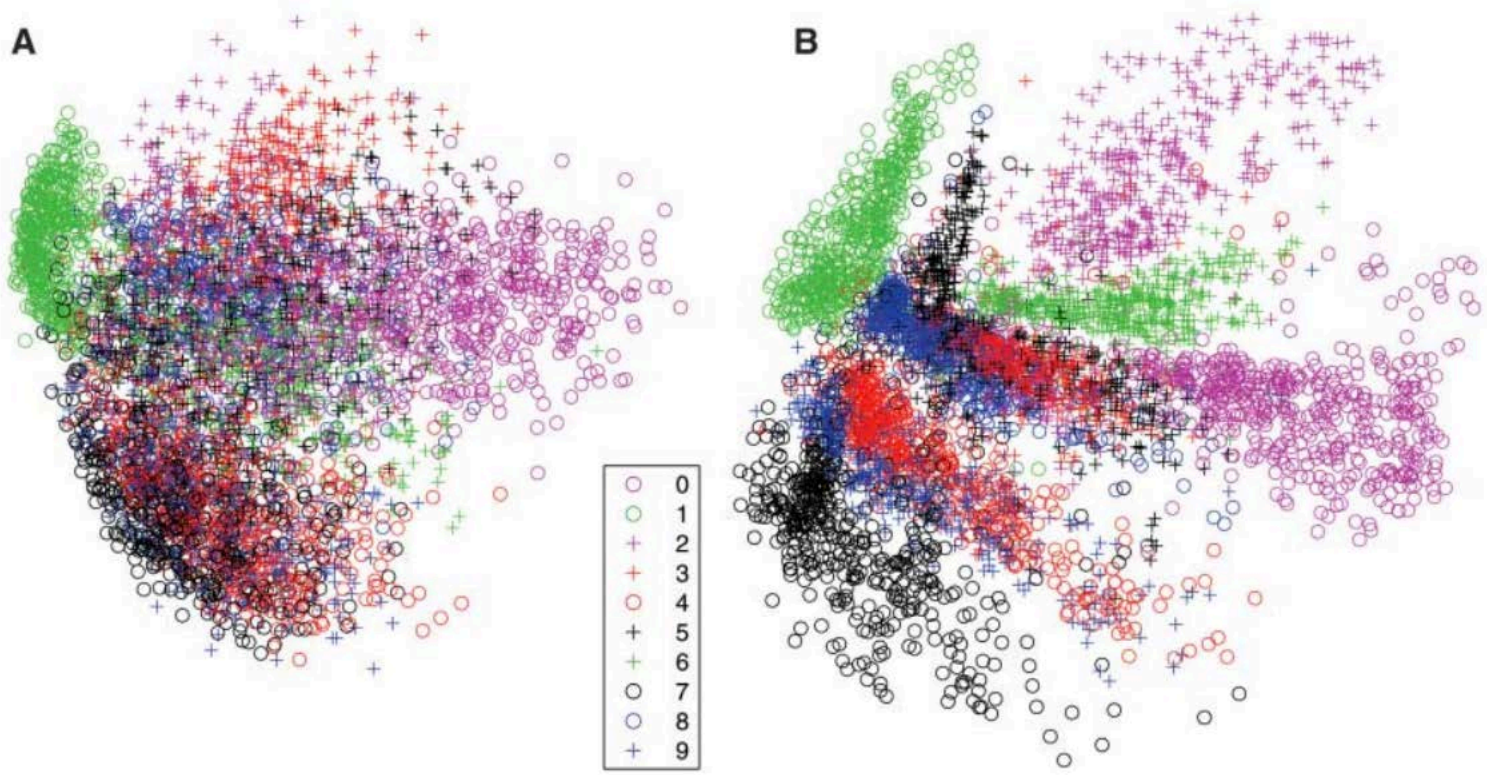


Without Pre-Training



With Pre-Training

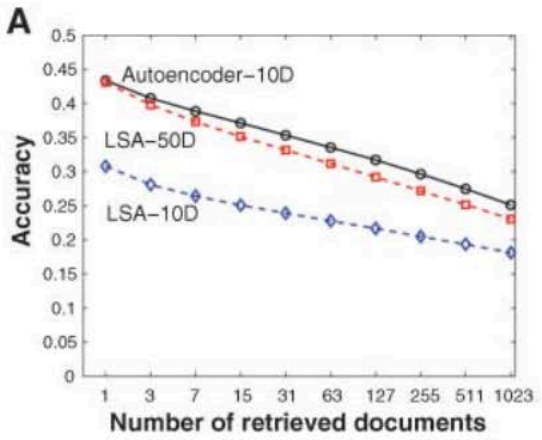
Use Cases for Auto Encoder – Dimension Reduction

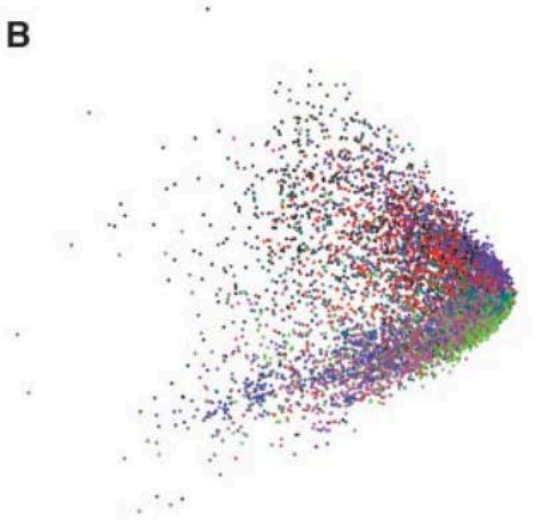


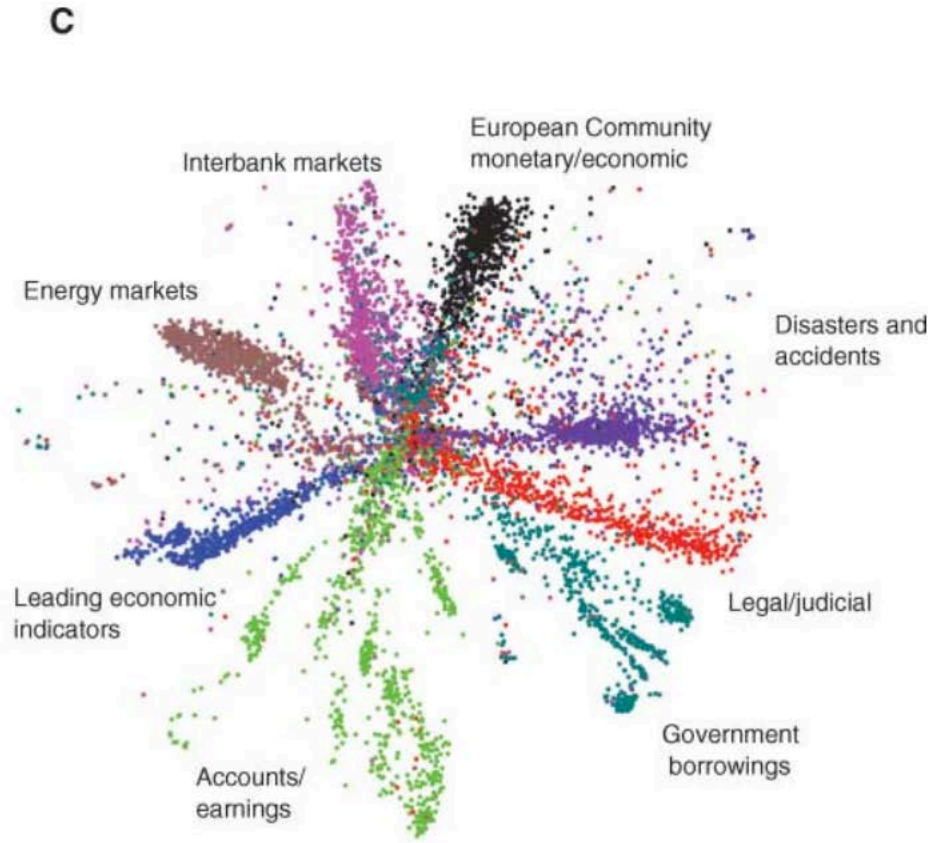
PCA

Auto Encoder

Use Cases for Auto Encoder – Dimension Reduction -Cont



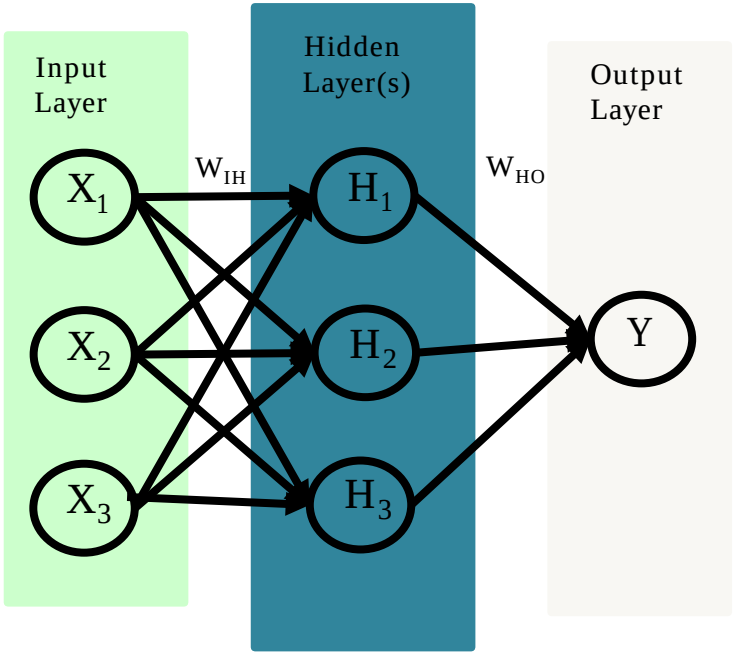




WHAT YOU SHOULD TAKE HOME

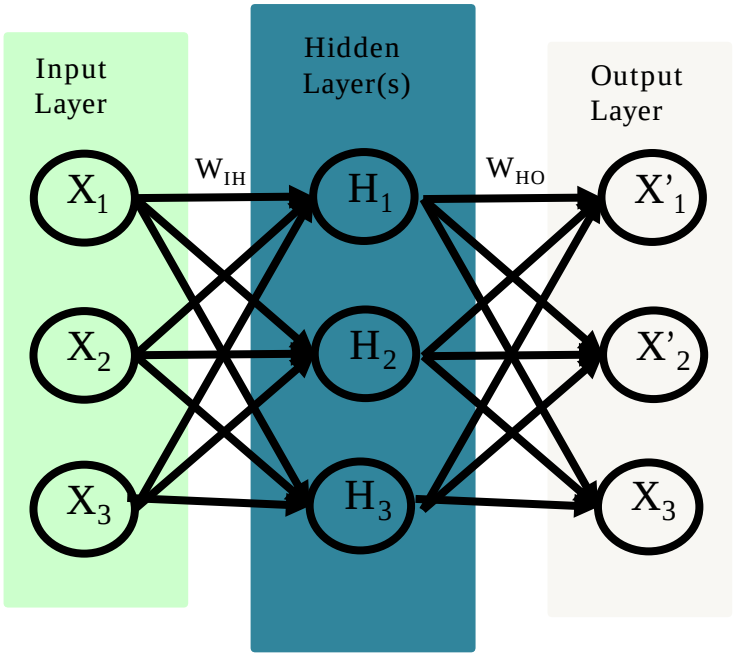
Comparison of NN and Auto Encoder

Neural Network



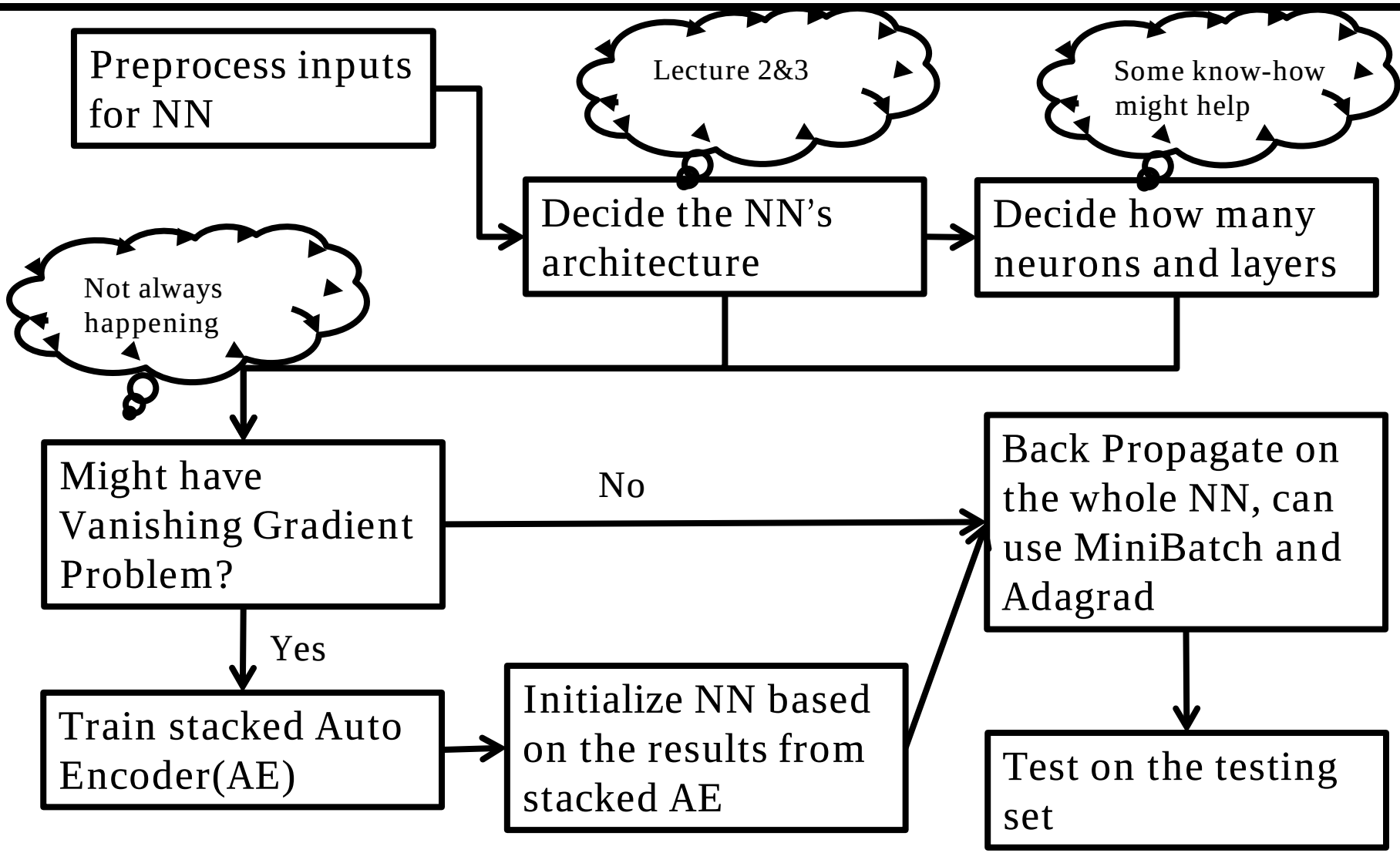
Train with a target value
Interest in the whole network

Auto Encoder



Train with its own input
Only interested on the hidden layer

Example Pipeline for Deep NN



Summary

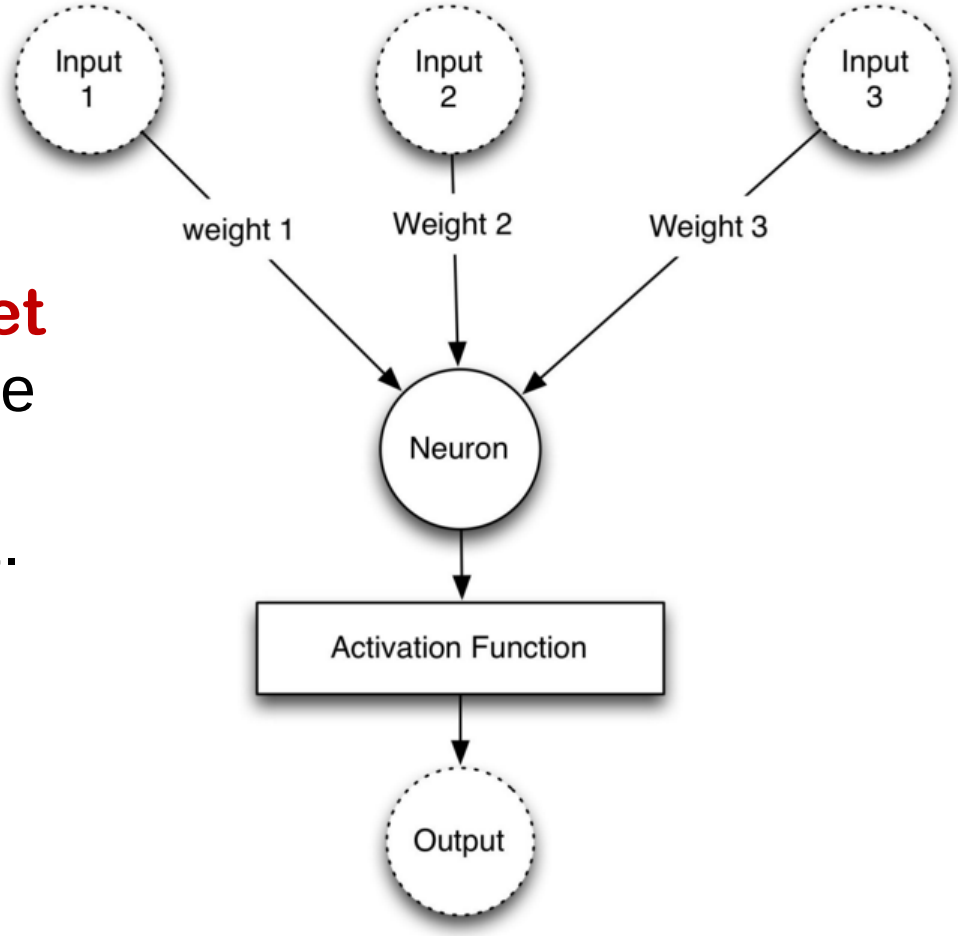
- Neural Network can be used to approximate all kinds of distribution because of the non-linearity neurons
- Back Propagation: Neural Network is trained by adjusting the weights according to the size of the error
- Vanishing Gradients: However, back propagation might failed if the network is too deep, because the error is consumed along its way back
- Pre-Training the gradients help reduce vanishing gradient problem, and can be done by auto encoder
- Deeper Networks are enabled by solving the vanishing gradient problems

Notation and Definition

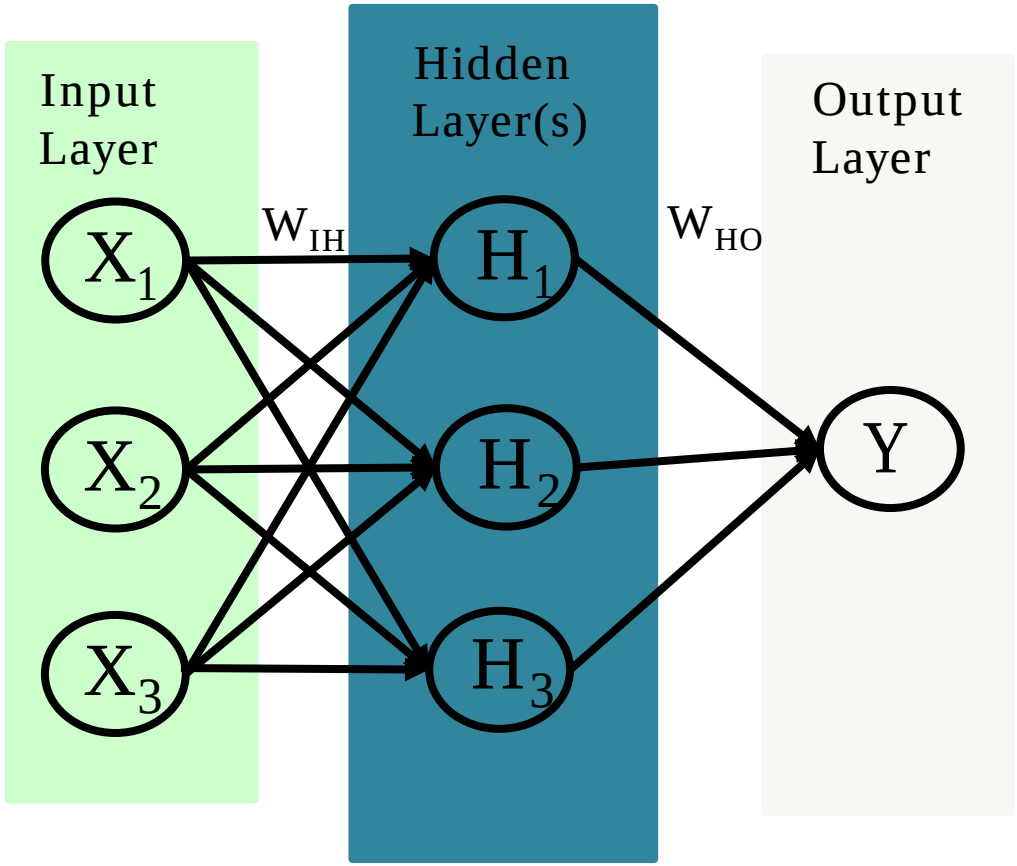
- **Fully connected neural networks:**
  - Neurons will connect to all neurons in the previous and next layer
- **Activation Function :**
  - A transformation function that signals weather a neuron is triggered
- **Input Layer :**
  - Layer of neurons that take input from data
- $W^{L1, L2}$ **:**
  - Weight matrix between layer 1 and layer 2
- **Output Layer:**
  - Layer of neurons that creates output from the given training data
- **Hidden Layers:**
  - One or multiple layers that lies between input and output layer that does linear or non-linear transform of the input data

Training a Neural Network

Training is a **search for the set of weights** that will cause the neural network to have the lowest error for a training set.



A Feed Forward Neural Network



$$Y = \sum_{j=1}^{J} \varphi\left(\sum_{i=1}^{I} W_{ij}^{IH} * X_i + b_j\right) * W_j^{HO} =$$

$$\left(\varphi\left(\begin{array}{c} X_1 * W_{11}^{IH} + \\ +X_2 * W_{21}^{IH} + b1 \\ +X_3 * W_{31}^{IH} \end{array}\right)\right) * W_{11}^{HO} +$$

$$\left(\varphi\left(\begin{array}{c} X_1 * W_{12}^{IH} \\ +X_2 * W_{22}^{IH} + b2 \\ +X_3 * W_{32}^{IH} \end{array}\right)\right) * W_{21}^{HO} +$$

$$\left(\varphi\left(\begin{array}{c} X_1 * W_{13}^{IH} + \\ +X_2 * W_{23}^{IH} + b3 \\ +X_3 * W_{33}^{IH} \end{array}\right)\right) * W_{31}^{HO}$$

What is Backpropagation in the Context of NN?

- **Backpropagation:**
  - **Method for training a neural network (NN)**
  - **Introduced by Hinton and Williams (1986)**
  - **It is a type of Gradient Descent (GD)**
  - **Lots of extensions based on Stochastic Gradient Descent (SGD)**
- **Why to use backpropagation for deep learning?**
  - **It scales really well when run on GPUs**

Two Passes of Backpropagation

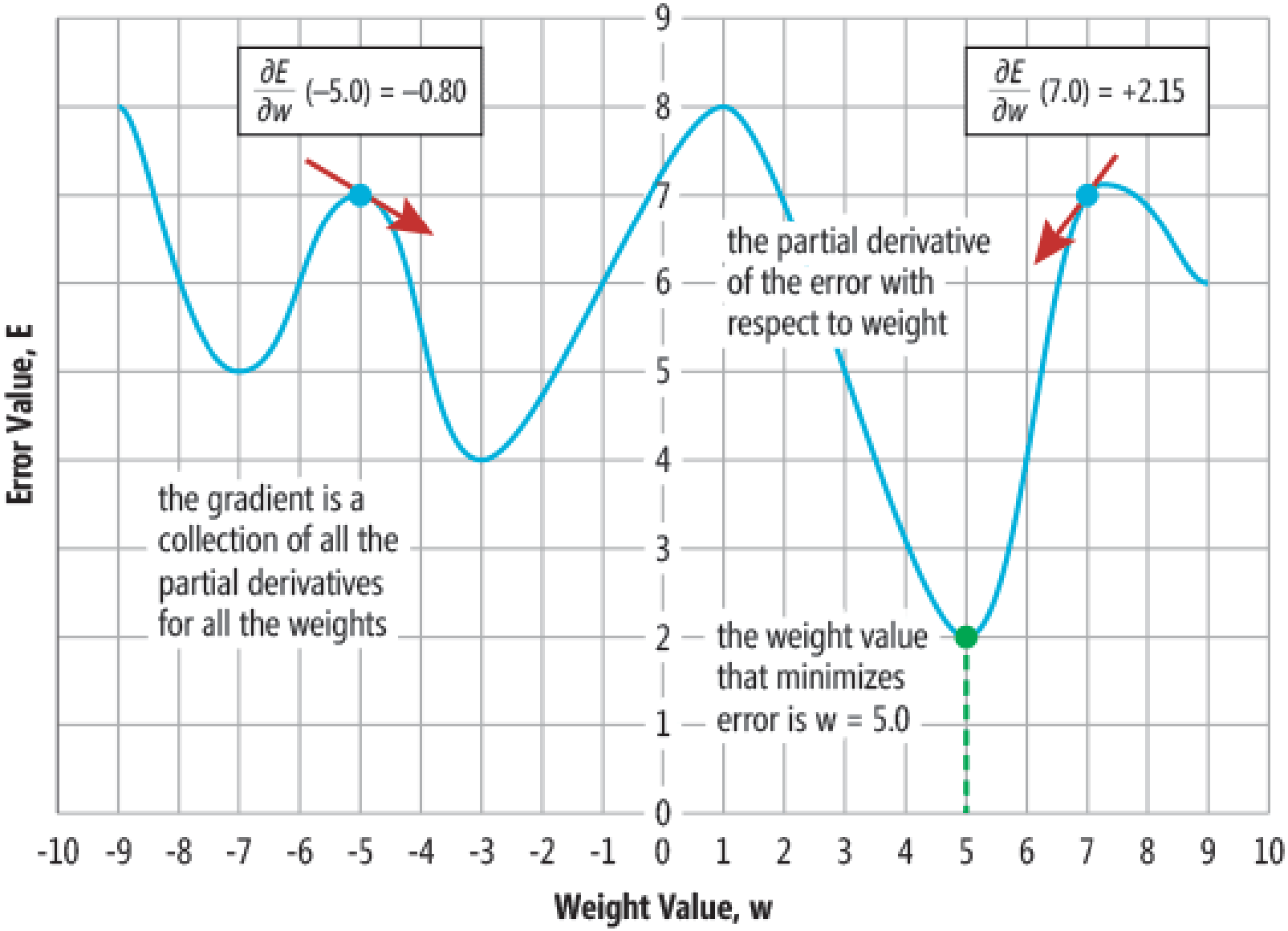
- **Forward Pass**
  - The forward pass computes the output of the NN
- **Backward Pass**
  - Calculate the gradient ONLY for the current item in the training
  - Update the relevant weights based on the gradient sign

Reminder: Gradient Descent

- **Gradient**
  - **The vector of partial derivatives of the error function over each weight computed at the weight's current value**
    - The error function (loss, cost) measures the distance of the NN's output from the expected output
    - Gradient is the ***instantaneous slope*** of the error function at the specified weight
  - **Each weight's gradient is the slope of the error function:**
    - The weight is a connection between two neurons / nodes in NN

- **The sign of the gradient tells the NN the following:**
  - **Zero gradient:** weight is not contributing to the error of the NN
  - **Negative gradient:** increase the weight to lower the error
  - **Positive gradient:** decrease the weight to lower the error

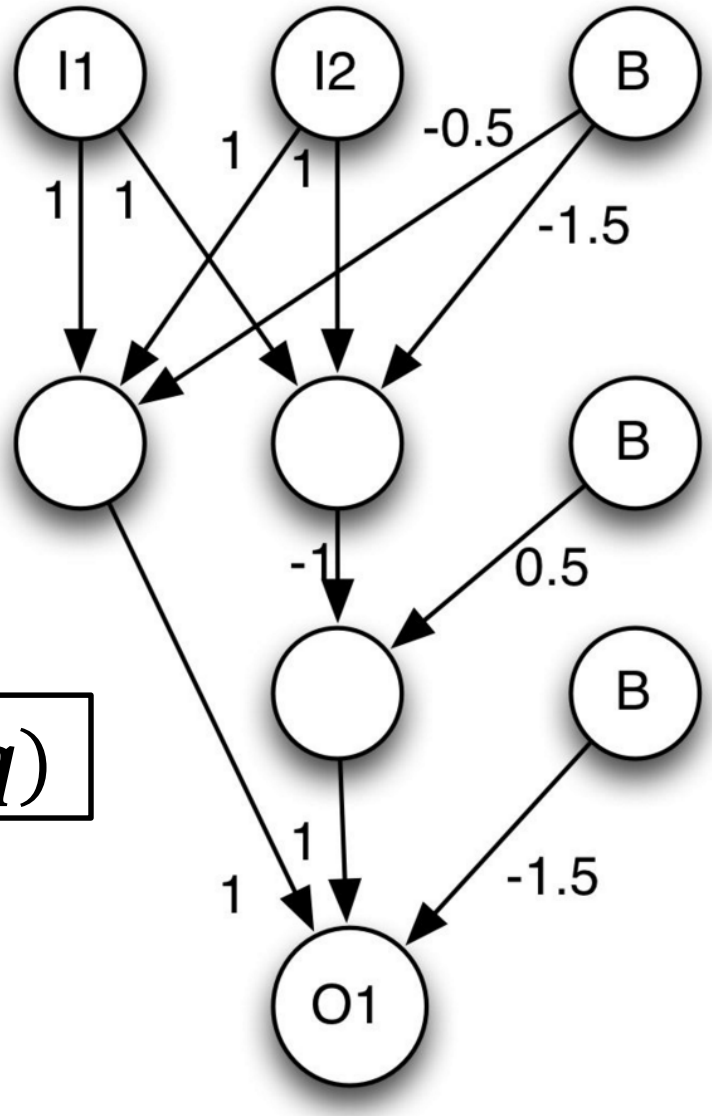
Error Depends on the Weight Value



XOR's NN with the Ground Truth Weights

- False XOR False = False
- True XOR False = True
- False XOR True = True
- True XOR True = False

$$p \otimes q = (p \vee q) \wedge \neg (p \wedge q)$$



Calculating the Gradient for Each Weight

- **Step 1: Calculate the error based on the ideal of the training set**
- **Step 2: Calculate the node delta for the output neurons**
- **Step 3: Calculate the node delta for the interior neurons**
- **Step 4: Calculate individual gradients**

Step 2: Calculate Output Node Deltas

- **Backward Pass**
  - **Start with the output nodes and work our way backward through the neural network**
  - **Calculate the errors for the output neurons**
  - **Propagate these errors backwards through the neural network**
- **Node Delta**
  - **The node delta is calculated for each node**
  - **Calculate node deltas one layer at a time**
- **Output Node Deltas**
  - **Calculated first**
  - **Take into account the error function for the NN:**
    - E.g., quadratic error function or the cross-entropy error function
- **Interior Node Deltas**
  - **???**

Output Node Delta for the Error Function

Quadratic Error Function

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

The **output node delta** for the quadratic error function:

$$\delta_i = (\hat{y}_i - y_i) \times \phi_i{}'$$

phi-prime: the derivative of the activation function

Cross Entropy Error Function

$$CE = \frac{1}{n}\sum_{i=1,\ldots,n}(y_i \times ln\,\hat{y}_i + (1 - y_i) \times ln(1 - \hat{y}_i))$$

cross-entropy (CE): the log loss

The **output node delta** for the cross entropy error function:

$$\delta_i = (\hat{y}_i - y_i)$$

Interior Node Delta

The interior node delta (all hidden and bias neurons):

$$\delta_i = \phi_i' \times \sum_k w_{ki}\delta_k$$

phi-prime: the derivative of the activation function

Derivatives of the Activation Function

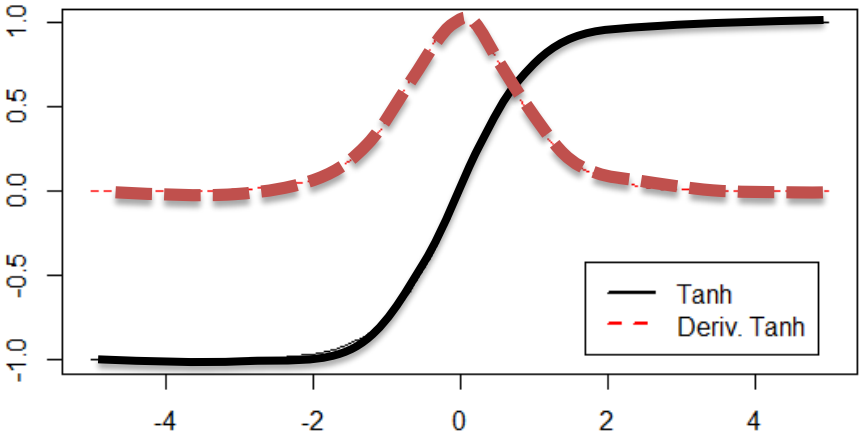
Activation Function	AF Formula, $\phi(z)$	AF Derivative, $\phi'(z)$
Linear AF	$\boldsymbol{\phi(z)}$	$\boldsymbol{\phi'(z) = 1}$
Softmax AF	$\boldsymbol{\phi(z_i) = \frac{e^{z_i}}{\sum_{j \in group} e^{z_j}}}$	$\boldsymbol{\phi'(z_i) = \phi(z_i)(1 - \phi(z_i))}$
Sigmoid	$\boldsymbol{\phi(z) = \frac{1}{1 + e^{-z}}}$	$\boldsymbol{\phi'(z) = \phi(z)(1 - \phi(z))}$
Hyperbolic Tangent AF	$\boldsymbol{\phi(z) = tanh\,(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}}$	$\boldsymbol{\phi'(z) = 1 - \phi^2(z)}$
Rectified Linear Units (ReLU) AF	$\boldsymbol{\phi(z) = max\,(0, z)}$	$\boldsymbol{\phi'(z) = \begin{cases} 1, & z > 0 \\ 0, & z \leq 0 \end{cases}}$

- **ReLU does NOT have a derivative at zero**. BUT, because of convention, the gradient of zero is substituted when $z = 0$
- Deep neural networks are difficult to train with sigmoid and hyperbolic tangent AFs using backpropagation:
  - **Vanishing gradient problem**

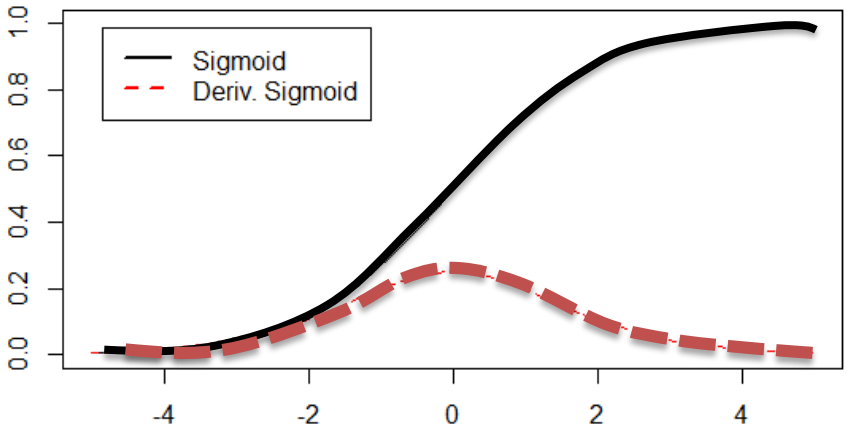
AF Saturation and its Effect on Derivative

Saturation Problem: Because both hyperbolic tangent and sigmoid AFs saturate to -1/1 and 0/1, respectively, their derivatives vanish to zero but ReLU does NOT have this problem.

Hyperbolic Tangent & its Derivative



Sigmoid & its Derivative

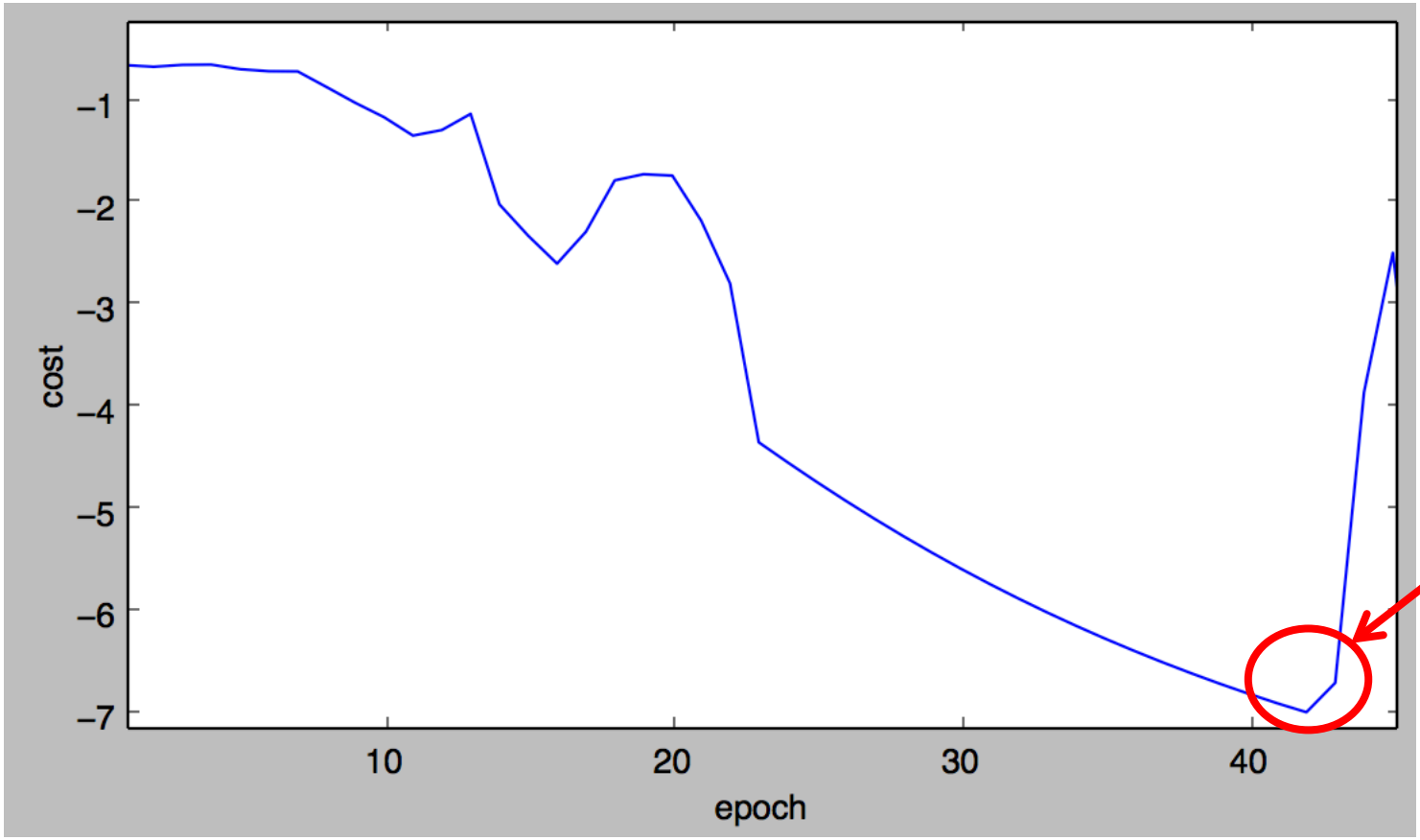


Linear, Softmax, and ReLU AF's

- Do not use the derivative of the linear or softmax AF for the cross-entropy error function (as it is = 1)
- Use **linear** and **softmax** AF ONLY at the **output layer** of the NN
- Use **ReLU** at the **hidden layers** of the NN, i.e., inferior nodes of the NN

Error Function	Linear AF Derivative, $\phi'(z)$	Softmax Derivative
Output nodes with cross entropy error function	$\boldsymbol{\phi'(z) = 1}$	$\boldsymbol{\phi'(z) = 1}$

Evolution of the Cost / Error



Model should stop here

1.How do we make sure the model can reach local minimum?
2.How do we make sure the model can reach global minimum?
3.How can we train the model fast?

Backpropagation Weight Update

$$\Delta w_t = -\gamma \frac{\partial Error}{\partial w_t} + \alpha \Delta w_{t-1}$$

- t : the current iteration step
- γ : the **learning rate**
- α : the **momentum** value (how much the previous weight change should be counted for)

Choosing Learning Rate & Momentum

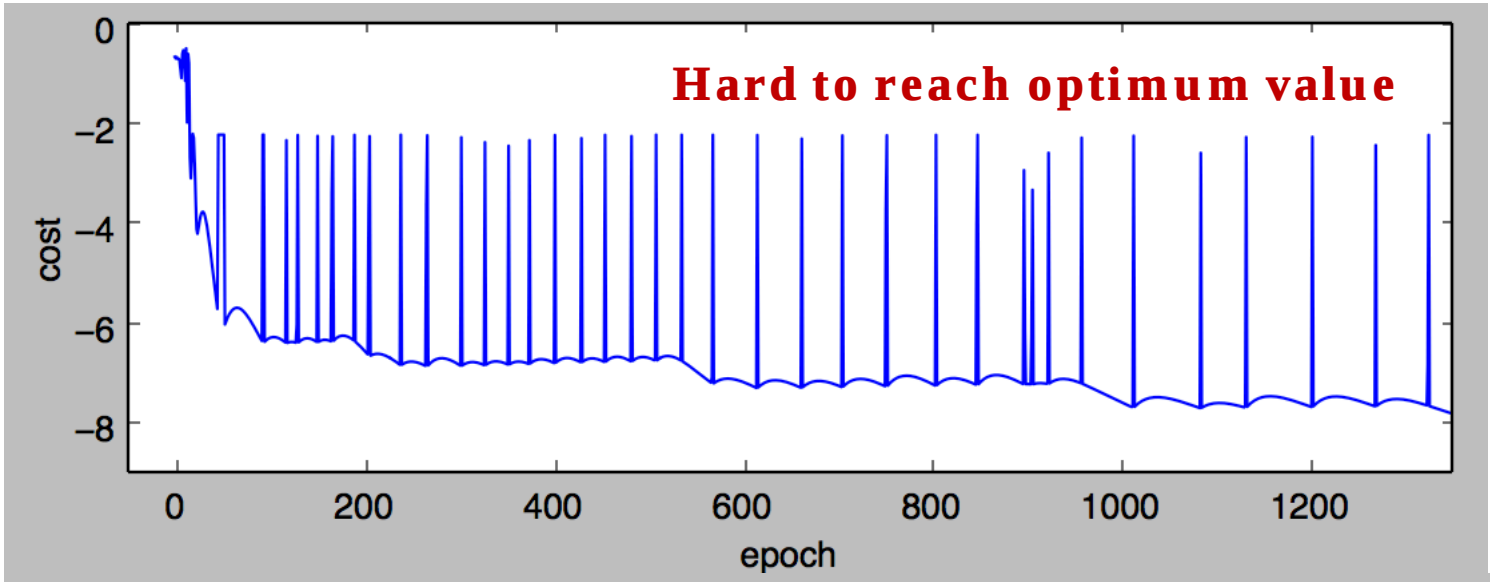
- Note: Choosing the learning rate and the momentum is the trial and error

- **Learning Rate**, γ ($\boldsymbol{\gamma \sim 0.1}$):
  - $\gamma = 0.5$ will decrease every gradient by 50%
  - $\gamma < 1$ slows down learning (most likely choice)
  - $\gamma > 1$ accelerated learning
  - γ is too high → causes NN to fail to converge and have a high global error bounce around instead of converging to a lower value
  - γ is too low → causes NN to take lots of time to converge

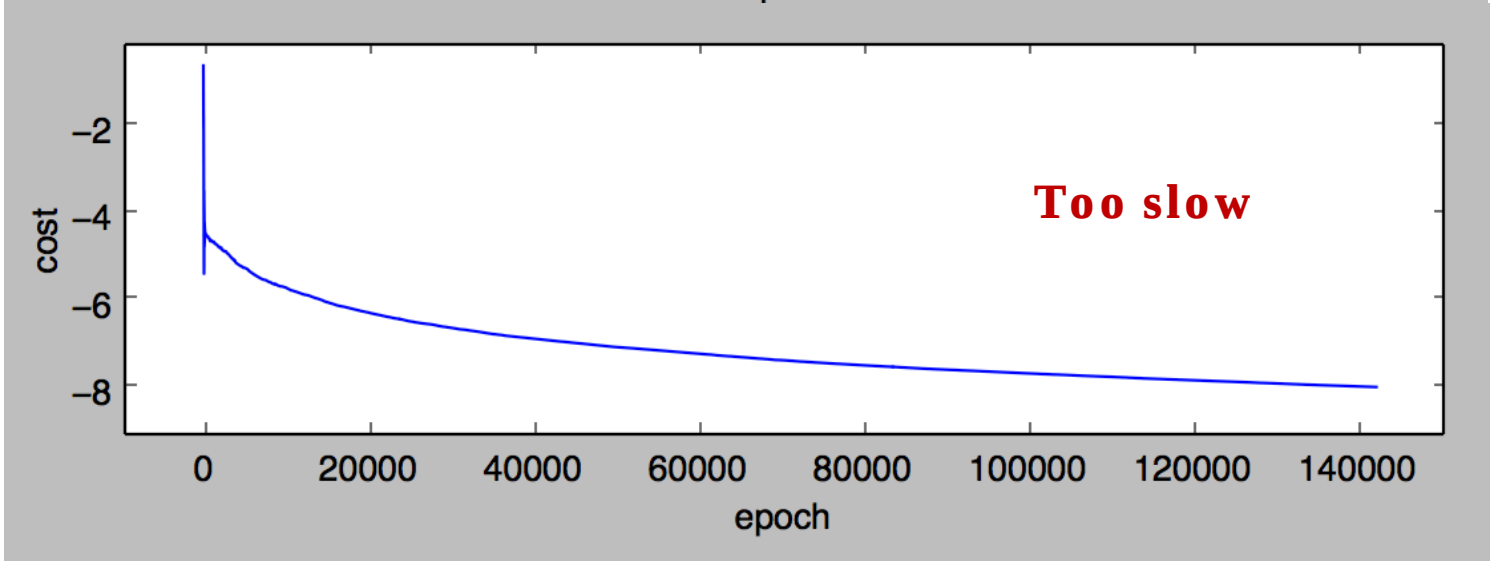
- **Momentum**, α ($\boldsymbol{\alpha \sim 0.9}$):
  - Helps the training to escape local minima that are not true global minimum
  - It gives a NN some force to break through a local minimum

Comparison of Learning Rates

High Leaning Rate



Low Leaning Rate



Learning Rate Strategies

- **Fixed**
  - Pros: Simple
  - Cons: Its hard to find one learning rate for all
- **Steadily Decreasing**
  - Pros: Also Simple
  - Cons: Still very universal
- **Fixed at the beginning, Steadily Decreasing after a certain point**
  - - *It assumes the weight should train faster at beginning*
  - Pros: Also Simple
  - Cons: Still very universal
- **Adaptive Subgradient**
  - Pros: Subjectively penalized weights based on need
  - Cons: Harder to implement, need to keep track of gradients

Adaptive Subgradient (Adagrad)

When the model is sampling data, we might want to treat data subjectively. i.e.:

Google News is a free news aggregator provided and operated by Google, selecting up-to-date news from thousands of publications. A beta version was launched in September 2002, and released officially in January 2006.

We might want to penalized these samples

AdaGrad alters learning rate based on historical information, so events with high frequency will get small learning rate.

$$\gamma = Fixed\ Learning\ Rate$$

$$\gamma_i = Learning\ rate\ for\ feature\ i = \frac{\gamma}{\sqrt{\sum(Gradient\ of\ i)^2}}$$

Speed Up, and Jump Out Of Local Minimum

When input data is large, or have many hidden layers in the model. The speed for processing gradient descent for each position in the weight matrices will become an issue for efficient model training. A proposed method is Stochastic Gradient Descent that randomly picks one training sample for each iteration.

General Gradient Descent:

For each iteration

For each sample in test data

Gradient Descent on the parameters

Stochastic Gradient Descent:

For each iteration

For **one** sample in test data

Gradient Descent on the parameters

Mini Batch Update

While Stochastic Gradient is speeding up the training speed, we can still improve the efficiency by utilizing matrix multiplications.

Mini-Batch update samples B gradients to update the model.

When B= 1, its normal Stochastic Gradient Descent
When B= training data size , its normal Gradient Descent

Can use priority sampling:

The input data will be asked to be pooled to high/low priority.

During each batch, it will sample a portion from high and low pool

Problem w/ Mini-Batching & Nesterov Moment

- Problem with Mini-Batching:
  - SGD can produce erratic results b/s of randomness introduced by mini-batching:
    - Weights might get beneficial updates in one iteration but
    - A poor choice of training samples can undo it in the next batch

- Nesterov Momentum
  - Designed to mitigate this erratic training result
  - Referred as Accelerated Gradient Descent

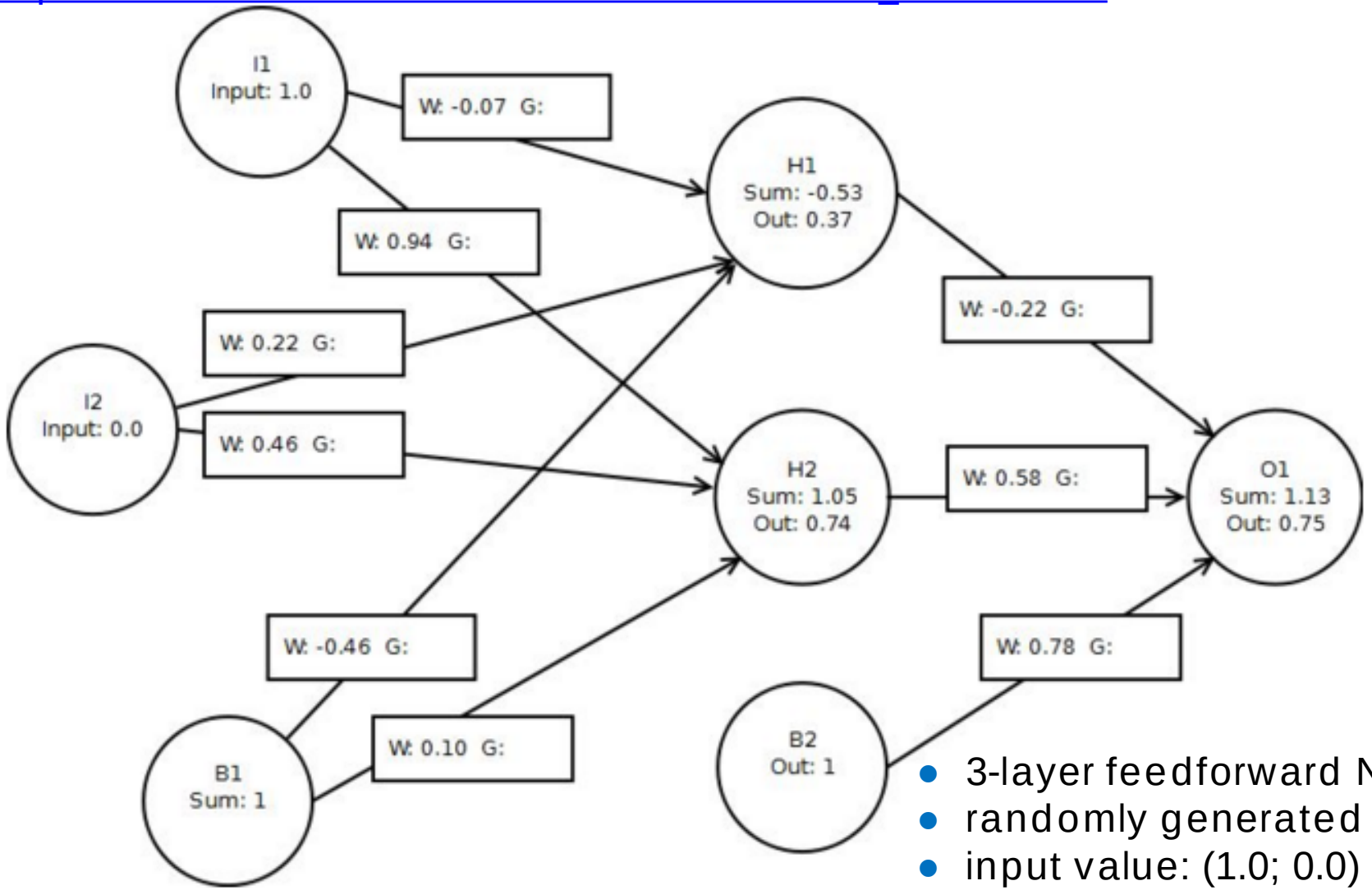
$$n_0 = 0$$

$$n_t = \alpha \times n_{t-1} + \gamma \times \frac{\partial Error}{\partial w_t}$$

$$\boxed{\Delta w_t = \alpha \times n_{t-1} - (1 + \alpha) \times n_t}$$

Ex: Training XOR Neural Network w/ Backpropagation

http://www.heatonresearch.com/aifh/vol3/xor_online.html
http://www.heatonresearch.com/aifh/vol3/xor_batch.html



- 3-layer feedforward NN
- randomly generated weights
- input value: (1.0; 0.0)

References:

1. Hinton, G. (2010). A practical guide to training restricted Boltzmann machines. Momentum, 9(1), 926.
2. Socher, R., Lin, C. C., Manning, C., & Ng, A. Y. (2011). Parsing natural scenes and natural language with recursive neural networks. In Proceedings of the 28th international conference on machine learning (ICML-11) (pp. 129-136).
3. Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In Neural Networks: Tricks of the Trade (pp. 437-478). Springer Berlin Heidelberg.
4. Course materials from Richard Socher 2015 Spring course: CS224d: Deep Learning for Natural Language Processing
5. Dyer, C. Notes on AdaGrad.
6. Erhan, D., Bengio, Y., Courville, A., Manzagol, P. A., Vincent, P., & Bengio, S. (2010). Why does unsupervised pre-training help deep learning?. The Journal of Machine Learning Research, 11, 625-660.

References:

7. Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. Science, 313(5786), 504-507.
8. Why Google is Investing in Deep Learning http://video.mit.edu/watch/why-google-is-investing-in-deep-learning-14382/ (12/8/2015)
9. Graves, A. (2013). Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850.
10. Karpathy, A., & Fei-Fei, L. (2014). Deep visual-semantic alignments for generating image descriptions. arXiv preprint arXiv:1412.2306.
11. Zero to expert in 8 hours: These robots harness deep learning, http://www.thespec.com/news-story/6158522-zero-to-expert-in-8-hours-these-robots-harness-deep-learning/ (12/8/2015)