Deep Learning for Visual Recognition
- Part 1

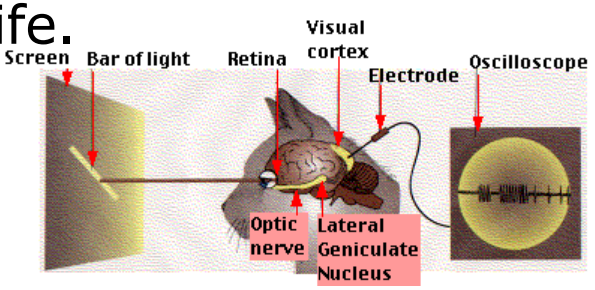
Kyu-Hwan Jung, Ph.D
kyuhwanjung@gmail.com

Class 2
Logistic Regression and MLP

Background

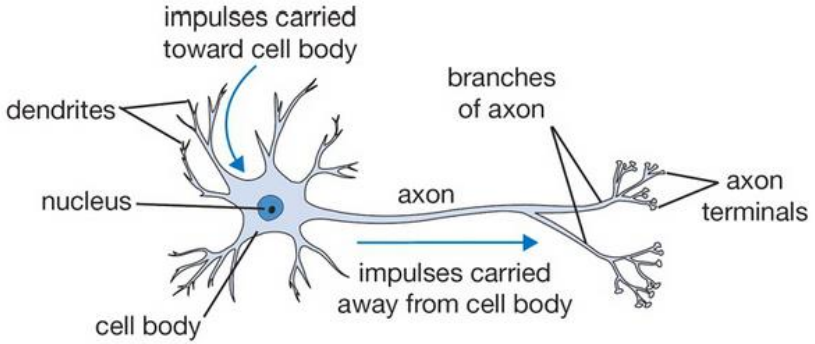
Neuroplasticity

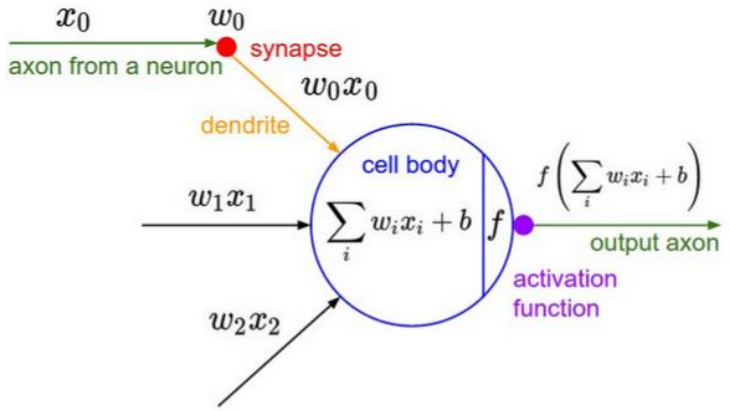
- The property of brain that allows it to change its structure and function in response to what it sees, what it does and even what it thinks and imagines.
- Donald Hebb
  - "Neurons fire together wire together"
- Hubel and Wiesel
  - Discovered a critical period from third to eighth week of life when the kitten's had to receive visual stimulation in order to develop normally.
  - When the eye was sewn shut they found that the visual areas in the brain map failed to develop, leaving the kitten blind in that eye for life.



Human vs Machine

- Comparison between brain and computer





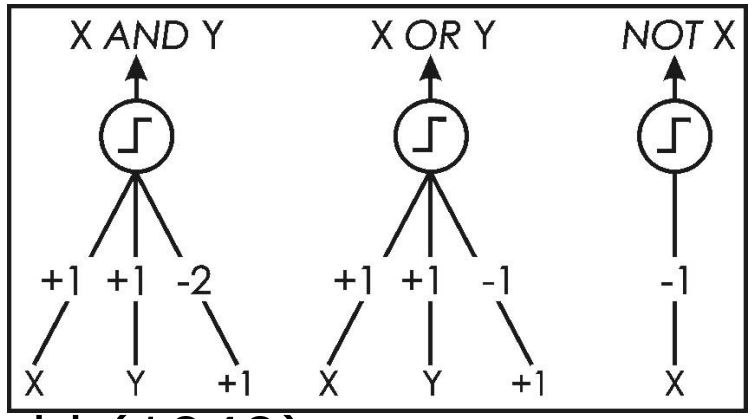
	Brain	Computer
No. of processing units	$\approx 10^{11}$	$\approx 10^{9}$
Type of processing units	Neurons	Transistors
Type of calculation	massively parallel	usually serial
Data storage	associative	address-based
Switching time	$\approx 10^{-3}$s	$\approx 10^{-9}$s
Possible switching operations	$\approx 10^{13}\frac{1}{s}$	$\approx 10^{18}\frac{1}{s}$
Actual switching operations	$\approx 10^{12}\frac{1}{s}$	$\approx 10^{10}\frac{1}{s}$

Â Brief Introduction to Neural Networks – David Kriesel

Brief History of NN - The Beginning

- McCulloch-Pitts Model(1943)
  - First attempts to build "electronic brain"

McCulloch-Pitts Model

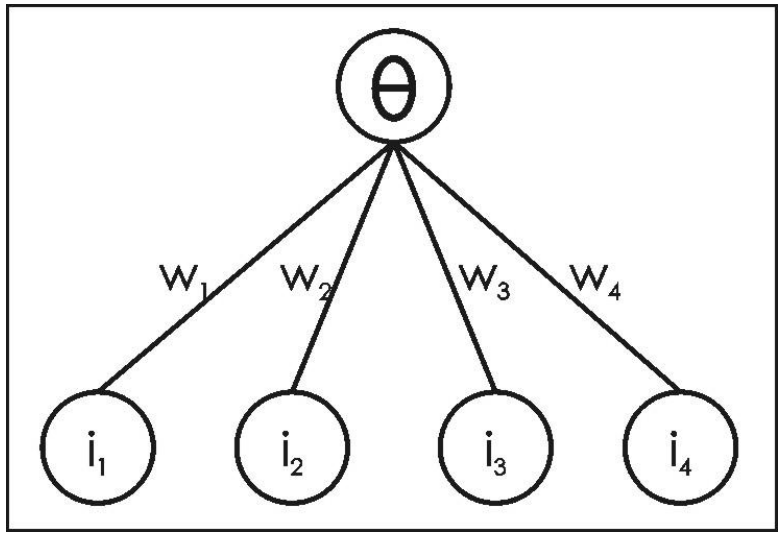


- Binary Input, Binary Output
- Adjustable Weights
- But Weights are not Learned
- Threshold of Activation Function is Always 0

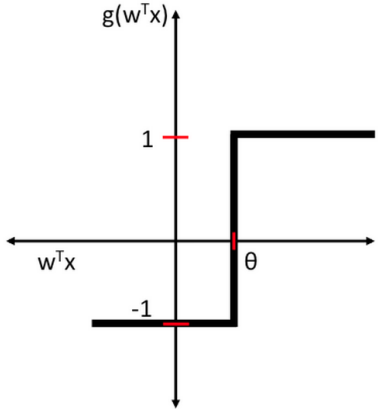
- Donald Hebb(1949)
  - "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."

Brief History of NN – Golden Age

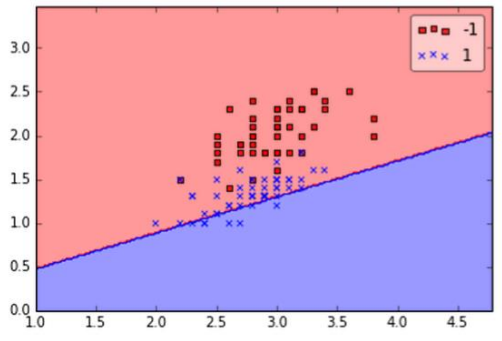
- Rosenblatt's Perceptron(1957)
  - Learnable weights and threshold
  - Simple learning algorithm : adjusting weight and threshold to the direction of reducing error

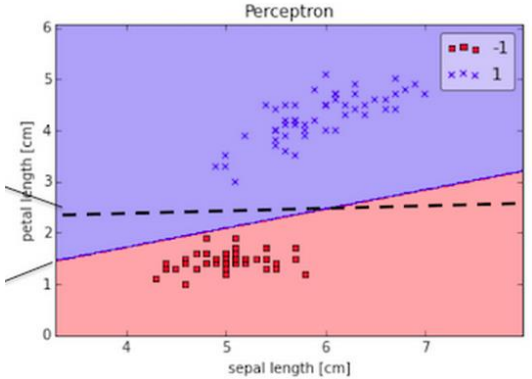


Perceptron



$$\Delta w_j = \eta \left(\text{target}^{(i)} - \text{output}^{(i)}\right) x_j^{(i)}$$

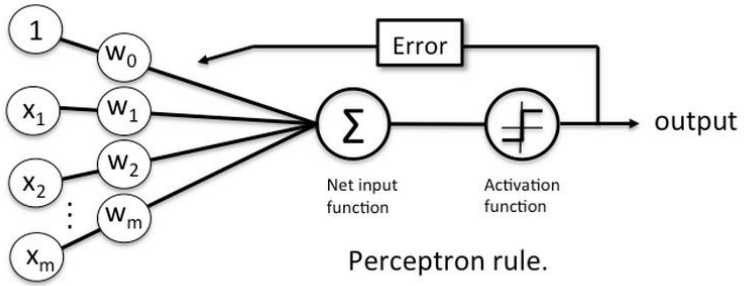




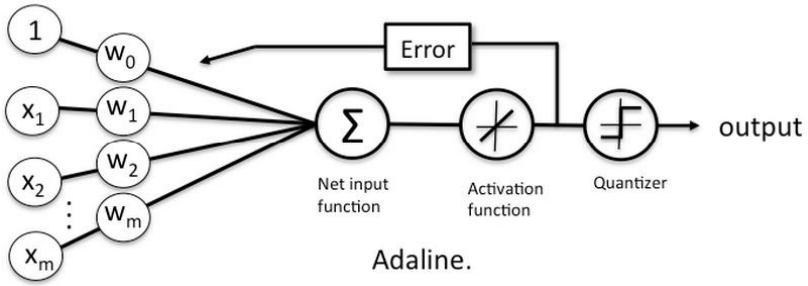
http://sebastianraschka.com/Articles/2015_singlelayer_neurons.html

Brief History of NN – Golden Age

- ADAptive Linear Neuron(1960)
  - Proposed by Widrow and Hoff
  - Learning by "Delta Rule"



Perceptron rule.



Adaline.

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{2}\sum_i (t^{(i)} - o^{(i)})^2$$

$$= \frac{1}{2}\sum_i \frac{\partial}{\partial w_j}(t^{(i)} - o^{(i)})^2$$

$$= \frac{1}{2}\sum_i 2(t^{(i)} - o^{(i)})\frac{\partial}{\partial w_j}(t^{(i)} - o^{(i)})$$

$$= \sum_i (t^{(i)} - o^{(i)})\frac{\partial}{\partial w_j}\left(t^{(i)} - \sum_j w_j x_j^{(i)}\right)$$

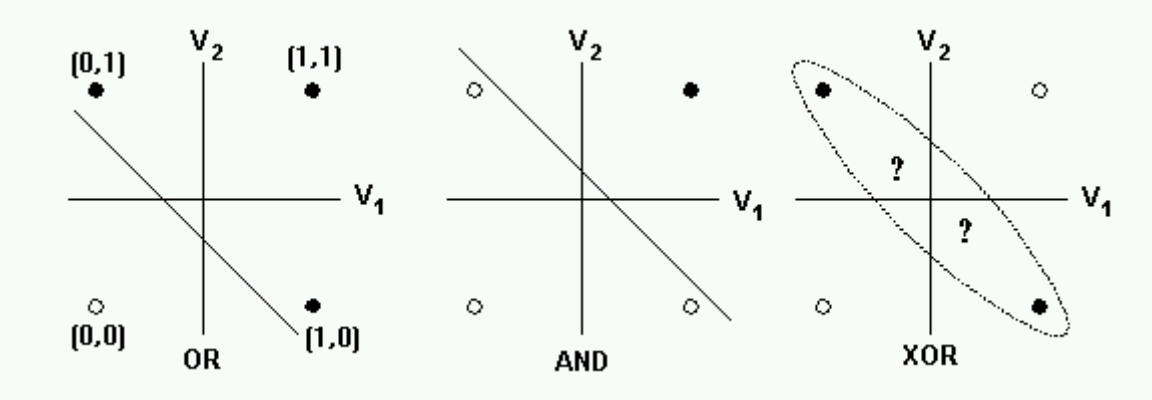
$$= \sum_i (t^{(i)} - o^{(i)})(-x_j^{(i)})$$

$$\Delta w_j = -\eta\frac{\partial J}{\partial w_j} = -\eta\sum_i (t^{(i)} - o^{(i)})(-x_j^{(i)}) = \eta\sum_i (t^{(i)} - o^{(i)})x_j^{(i)},$$

http://rutherfordjournal.org/article040101.html

Brief History of NN – Dark Age

- Dead end of Perceptron(1969)
  - Marvin Minsky and Seymour Papert published a paper claiming mathematical limitation of Perceptron
  - Certain problem (e.g. XOR) is not implementable by Perceptron
  - Complete decline in research fund and conferences for Neural Networks

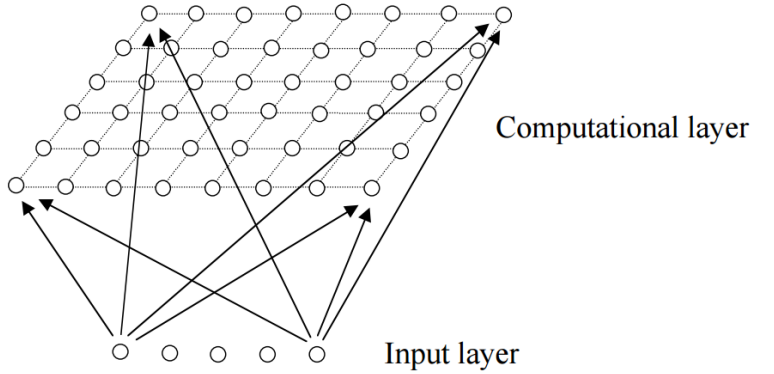


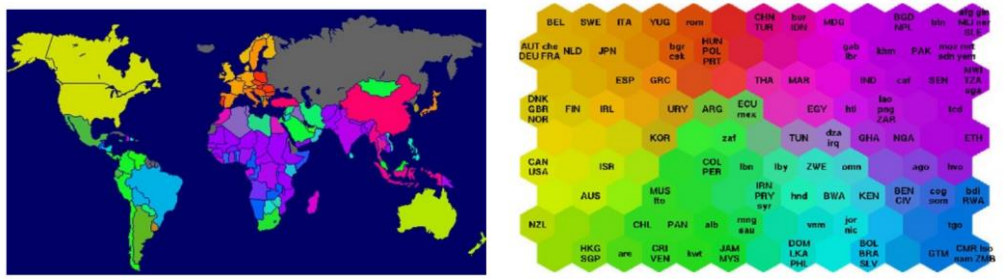
XOR Problem

http://www.cs.ucc.ie/~adrian/cs5201/NeuralComputingI.htm

Brief History of NN – Slow Reconstruction

- Self-organizing Map(1982)
  - Proposed by Teuvo Kohonen
  - Constructing topological map by unsupervised competitive learning : "Winner-takes-all"





- Initialization
  - Initialize with Random Weight
- Competition
  - Find Winner $d_j(\mathbf{x}) = \sum_{i=1}^{D} (x_i - w_{ji})^2$
- Cooperation
  - Find Neighbors $T_{j,I(\mathbf{x})} = \exp(-S_{j,I(\mathbf{x})}^2 / 2\sigma^2)$
- Adaptation
  - Update Weight

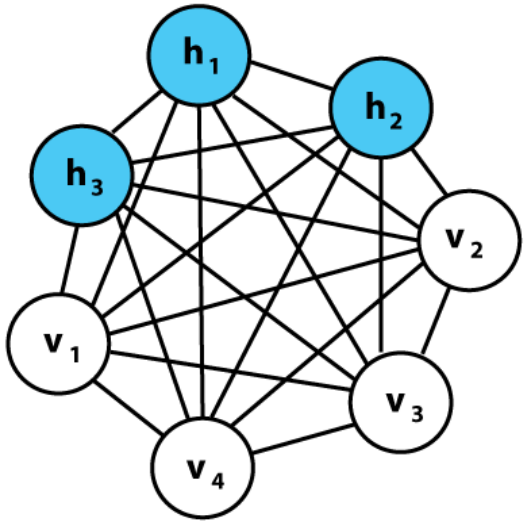
$$\Delta w_{ji} = \eta(t) \,.\, T_{j,I(\mathbf{x})}(t) \,.\, (x_i - w_{ji})$$

http://www.cs.bham.ac.uk/~jxb/NN/l16.pdf

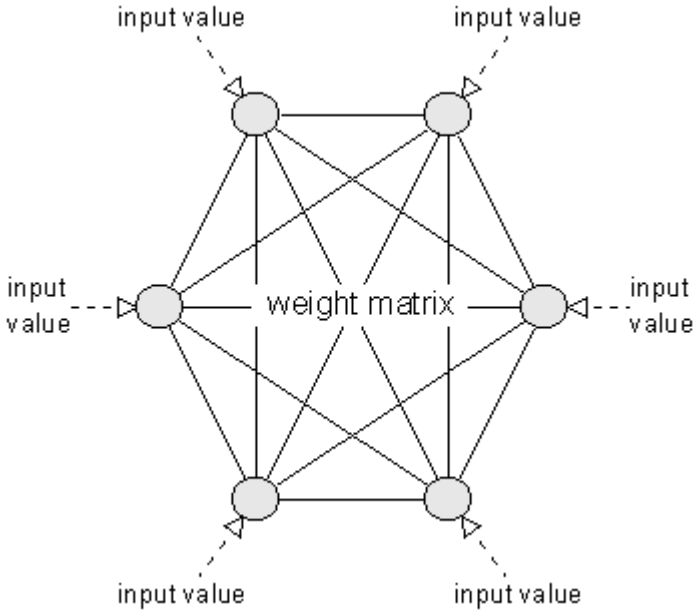
Brief History of NN – Slow Reconstruction

- Energy-based Model with Recurrent Connections

$$E = -\left(\sum_{i<j} w_{ij}\, s_i\, s_j + \sum_i \theta_i\, s_i\right)$$



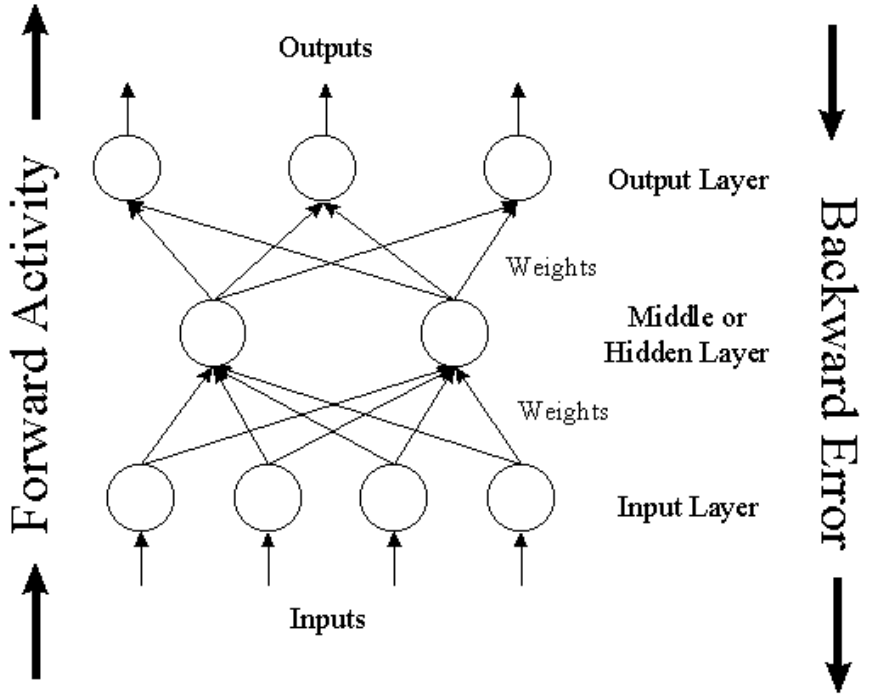
Boltzmann Machine(Hinton & Sejnowski, 1985)



Hopfield Network(John Hopfield, 1982)

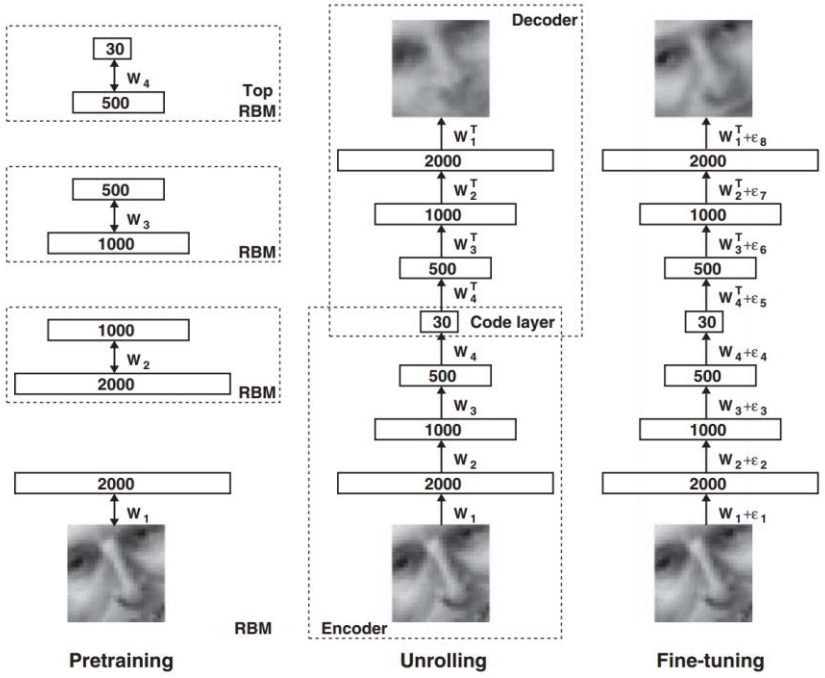
Brief History of NN - Renaissance

- Backpropagation Learning Rule
  - Solution to nonlinearly separable problems by back-propagation of error learning rule.
  - Counter-blow to Minsky



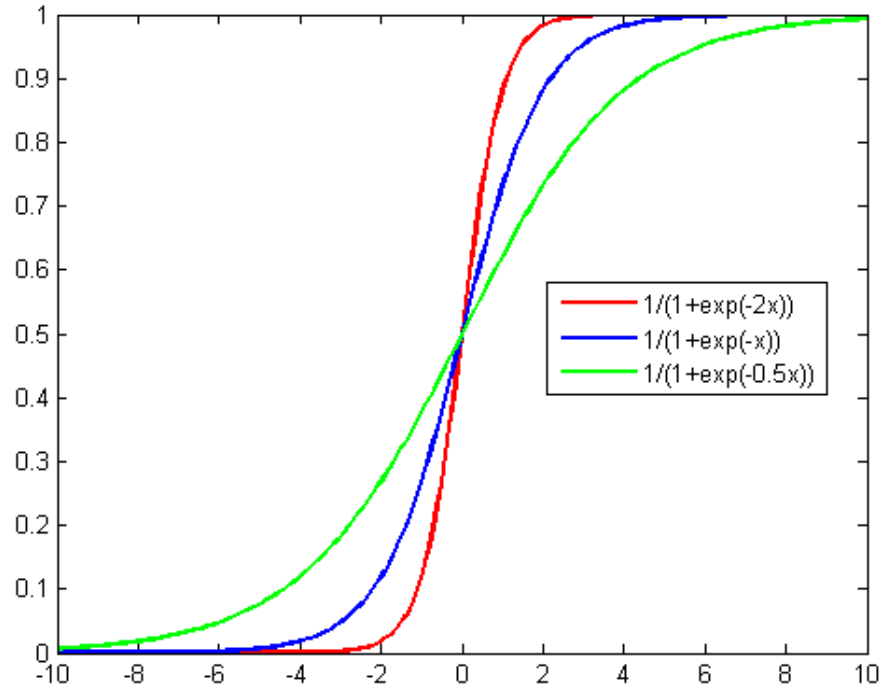
Brief History of NN – The era of Deep Learning

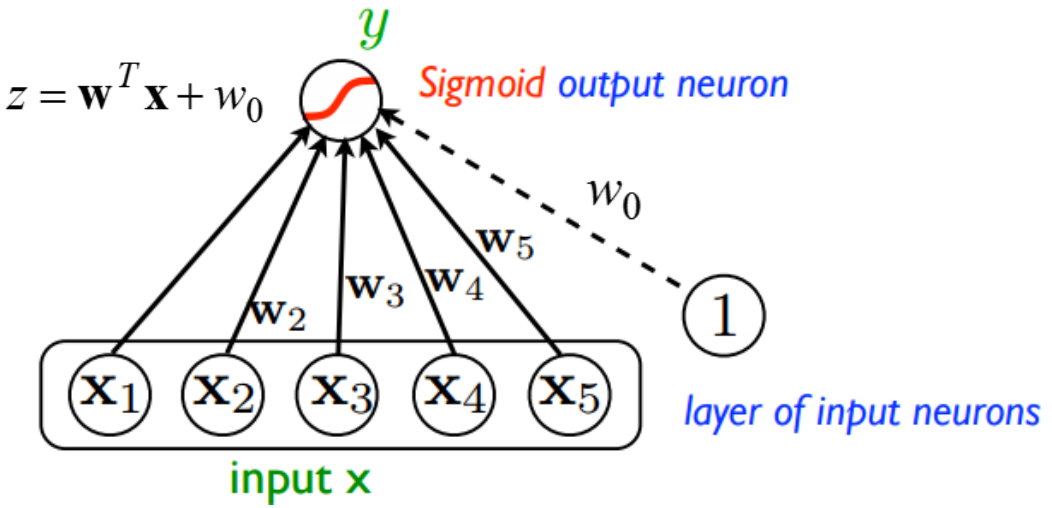
- Pretraining of Deep Neural Network
  - By Hinton and Salakhudinov(2006, Science)
  - 'Pre-training' and 'Fine-tuning' framework
  - Provided a way to 'initialize' a deep neural network to be learned by gradient decent.



Softmax Network

Logistic Regression



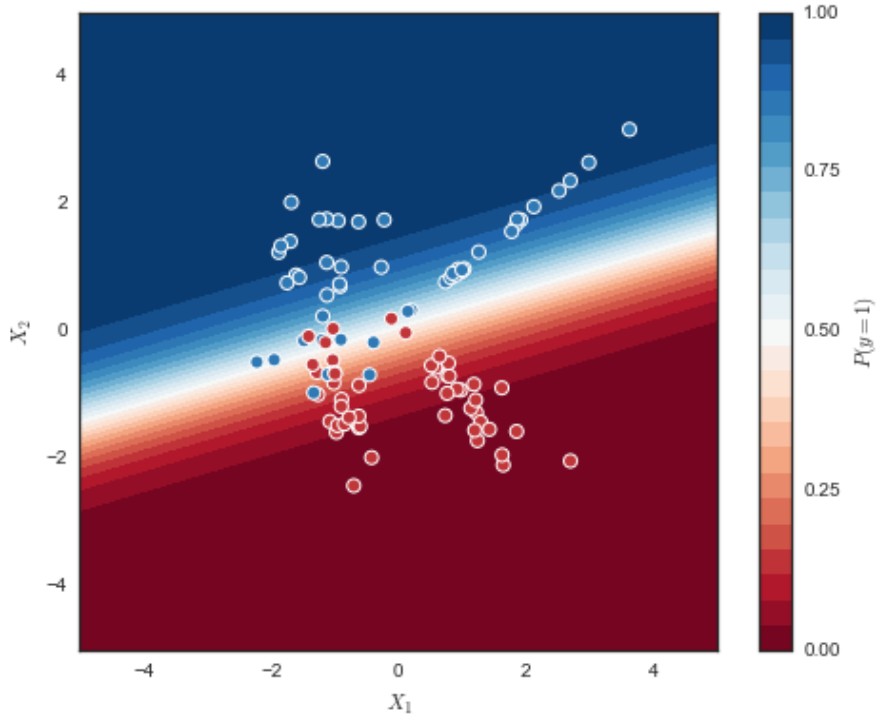


$$p(C_1 \mid \mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + w_0) \qquad \textit{where} \quad \sigma(z) = \frac{1}{1+\exp(-z)}$$

$$p(C_2|\mathbf{x}) = 1 - \sigma(\mathbf{w}^T\mathbf{x} + w_o)$$

Decision Boundary

- Logistic Regression is a Linear Model



$$\text{logit}(P(C_1|X)) = log\frac{P(C_1|X)}{1-P(C_1|X)}$$

$$= \log\frac{P(C_1|X)}{P(C_2|X)}$$

$$= \mathbf{w}^T\mathbf{x} + w_o$$

- The decision boundary is the hyperplane

$$\mathbf{w^T x} + w_o = 0 \quad where\ P(C_1|\mathbf{x}) = P(C_2|\mathbf{x})$$

- Extension to multi-class problem is natural

Learning

- Error Function
  - Likelihood of correct answer

$$P(t_n = 1|y_n) = y_n$$
$$P(t_n = 0|y_n) = 1 - y_n$$

$$\Rightarrow \quad P(t_n|y_n) = y^{t_n} \cdot (1 - y_n)^{1-t_n}$$

For all N samples,

$$L(\mathbf{w}) = \prod_{n=1}^{N} P(t_n|y_n)$$
$$= \prod_{n=1}^{N} {y_n}^{t_n} \cdot (1 - y_n)^{1-t_n}$$

$$\Rightarrow$$

Maximizing log-likelihood corresponds to minimizing negative log-likelihood as

$$\min_{\mathbf{w}} E = -\sum_{n=1}^{N} \ln p(t_n|y_n)$$
$$= -\sum_{n=1}^{N} t_n \ln y_n + (1 - t_n) \ln (1 - y_n)$$

Learning

- Gradient Calculation

$$E = -\sum_{n=1}^{N} \ln p(t_n \mid y_n)$$

$$= -\sum_{n=1}^{N} t_n \ln y_n + (1-t_n)\ln(1-y_n)$$

$$y_n = \sigma(z_n) = \frac{1}{1+e^{-z_n}}$$

$$z_n = \mathbf{w}^T \mathbf{x}_n + w_0, \qquad \boxed{\frac{\partial z_n}{\partial \mathbf{w}} = \mathbf{x}_n}$$

$$\frac{\partial E_n}{\partial y_n} = \frac{y_n - t_n}{y_n(1-y_n)}, \qquad \frac{dy_n}{dz_n} = y_n(1-y_n)$$

$$\frac{\partial y_n}{\partial z_n} = \frac{\partial}{\partial z_n}\frac{1}{(1+e^{-z_n})}$$

$$= \frac{1}{(1+e^{-z_n})^2}(e^{-z_n})$$

$$= \frac{1}{(1+e^{-z_n})}\left(1 - \frac{1}{(1+e^{-z_n})}\right)$$

$$\boxed{= y_n(1-y_n)}$$

$$\frac{\partial E_n}{\partial y_n} = -\frac{t_n}{y_n} + \frac{1-t_n}{1-y_n}$$

$$\boxed{= \frac{y_n - t_n}{y_n(1-y_n)}}$$

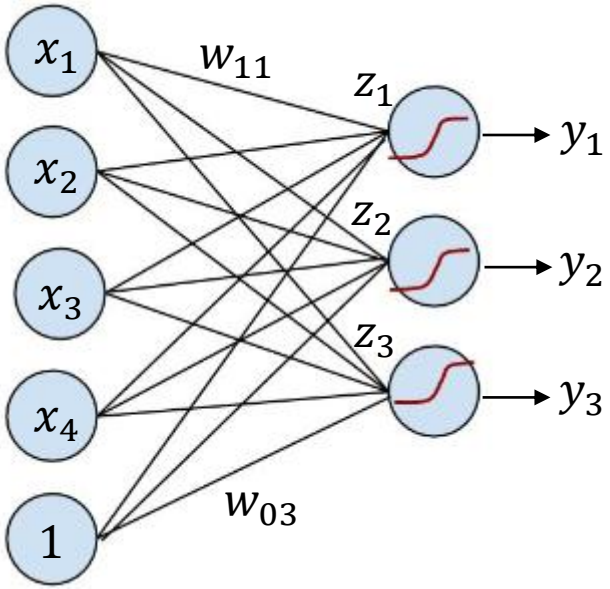
$$\boxed{\frac{\partial E_n}{\partial \mathbf{w}} = \frac{\partial E_n}{\partial y_n}\frac{dy_n}{dz_n}\frac{\partial z_n}{\partial \mathbf{w}} = (y_n - t_n)\,\mathbf{x}_n}$$

Weight Update

$$\mathbf{w} = \mathbf{w} - \lambda(y_n - t_n)\mathbf{x}_n$$

Multi-class Classification

- Extension of Logistic Regression to Multi-class Classification



Softmax Function

$$y_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \quad \text{where} \quad z_i = \mathbf{W}_{\cdot i} \cdot \mathbf{x}$$

Error Function

$$P(t_i = 1|y_i) = \prod_{i=1}^{C} y_i{}^{t_i} \quad \Rightarrow \quad \mathrm{E}(\mathbf{W}) = -\sum_{i=1}^{C} t_i \cdot \ln \mathrm{y}_i$$

Gradient Calculation

$$\frac{\partial y_i}{\partial z_i} = y_i\,(1 - y_i) \qquad \frac{\partial E}{\partial z_i} = \sum_j \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_i} = y_i - t_i \qquad \mathbf{W}_{\cdot i} = \mathbf{W}_{\cdot i} - (y_i - t_i)\mathbf{x}$$

SVM vs Softmax Classifier

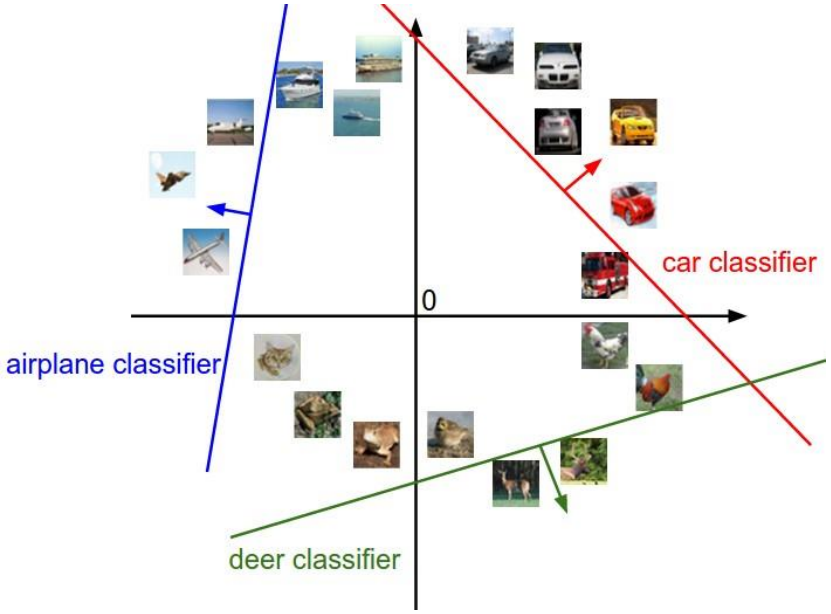
- Compasrion of (Linear) SVM and Softmax Classifier
  - Both are linear model with different loss function
  - Softmax classifier provides notion of 'class probability'
  - They are widely used for top classification layer for deep neural networks such as CNN

SVM loss function : Hinge Loss

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)$$

Softmax loss function : Cross Entropy

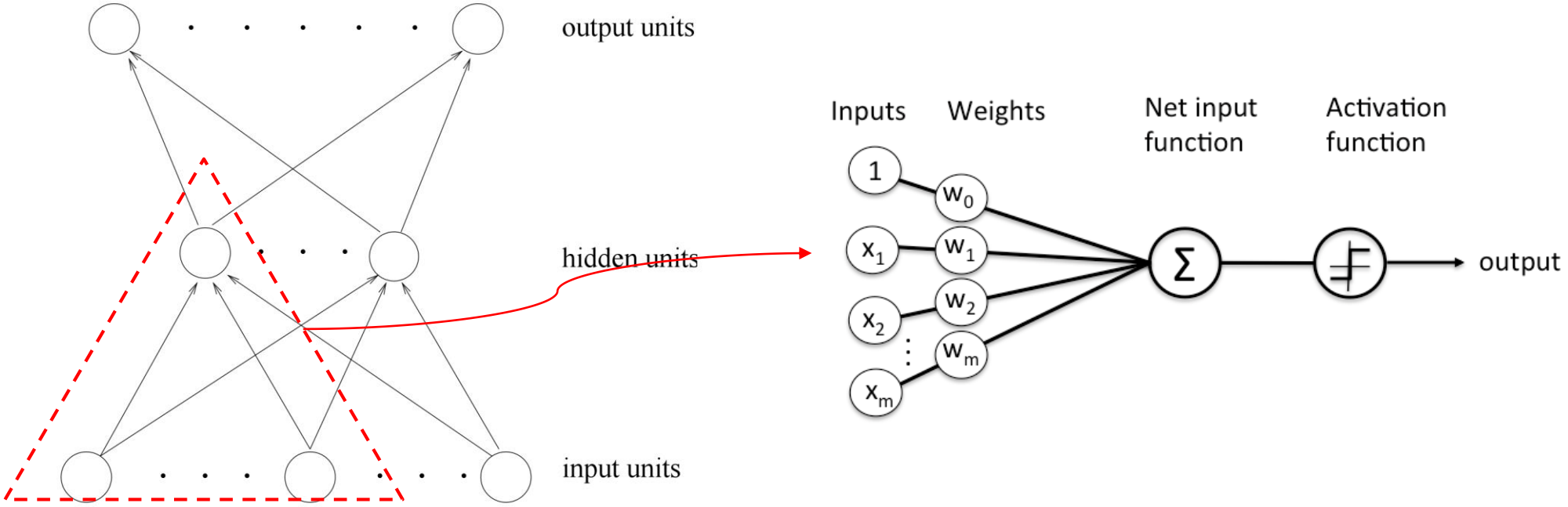
$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$



Multi-layer Perceptron

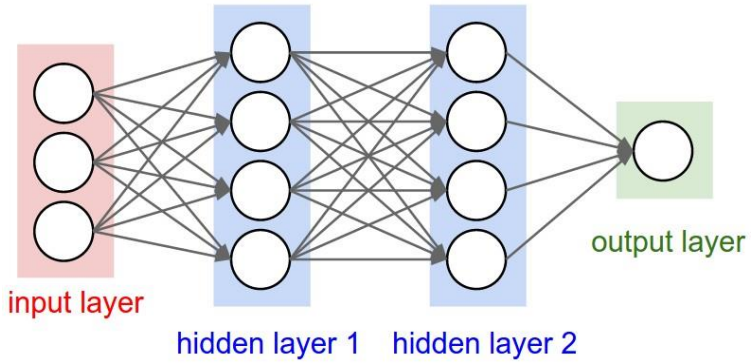
Multi-layer Perceptron

- Stack of layers with Nonlinearly Activated Nodes



Neural Network

- Fully-connected Feedforward Neural Network

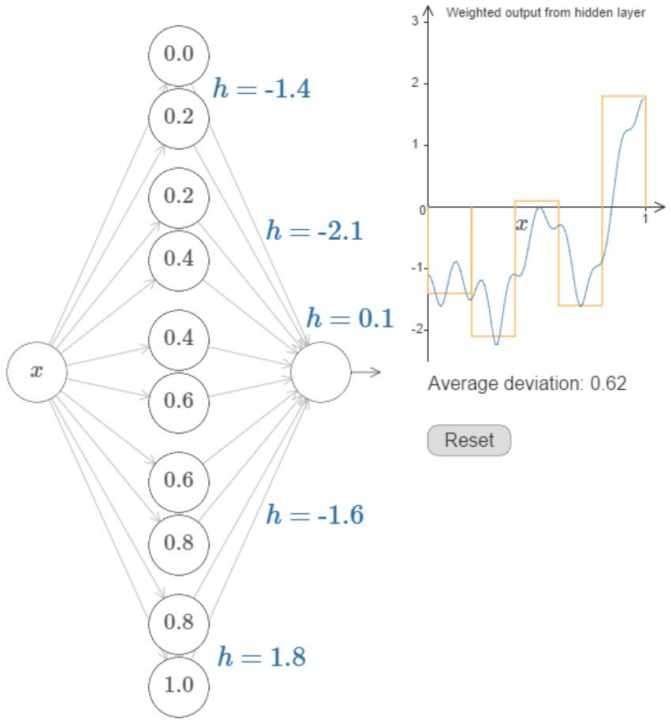


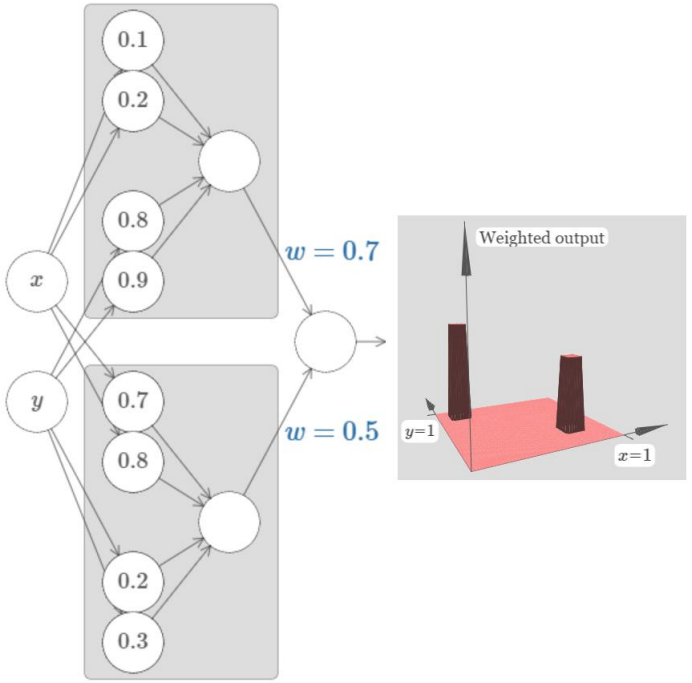
Number of Neurons: 4 + 4 + 1 = 9
Number of Weights: [4x3+4x4+1x4]=32
Number of Parameters: 32+9 = 41

- The number of layer in Neural Network = (num of all layer) – 1(input layer)
- Logistic Regression or SVM are single-layered Neural Network with special basis function and activation function
- The depth(num of layer) and number of nodes in each layer decides the representational power of Neural Networks

Representation Power of NN

- Neural Network as a Universal Approximator
  - NN with at least one hidden layer can approximate arbitrary continuous function with desired precision



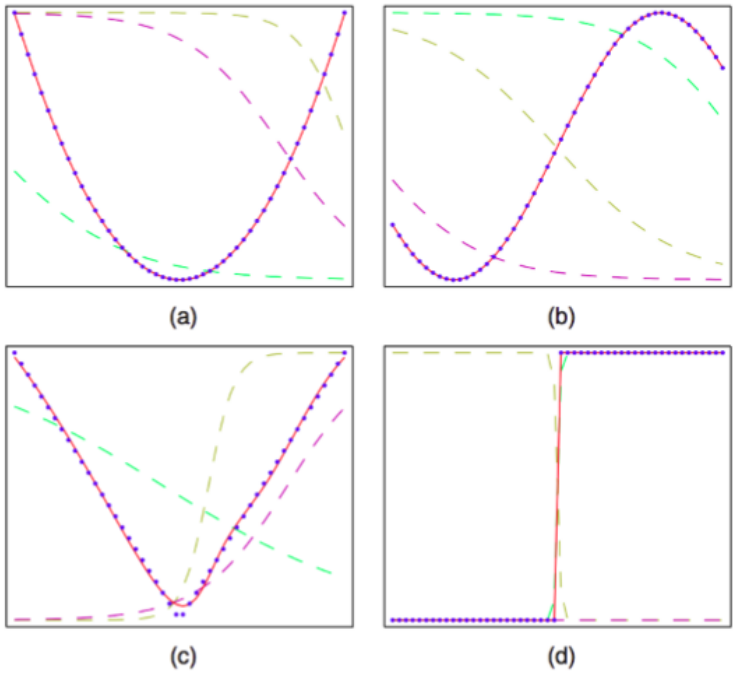


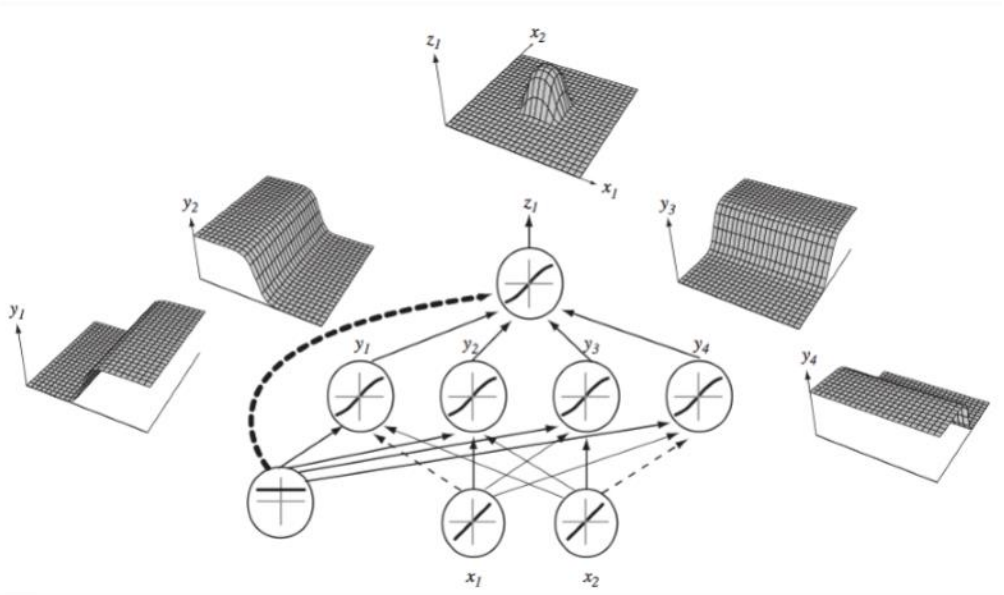
http://neuralnetworksanddeeplearning.com/chap4.html

Representation Power of NN

- Approximation of Functions using NN
  - Any function can be approximated with arbitrary accuracy using neural networks with at least one hidden layer

Figure 5.3 Illustration of the capability of a multilayer perceptron to approximate four different functions comprising (a) $f(x) = x^2$, (b) $f(x) = \sin(x)$, (c), $f(x) = |x|$, and (d) $f(x) = H(x)$ where $H(x)$ is the Heaviside step function. In each case, $N = 50$ data points, shown as blue dots, have been sampled uniformly in x over the interval $(-1, 1)$ and the corresponding values of $f(x)$ evaluated. These data points are then used to train a two-layer network having 3 hidden units with 'tanh' activation functions and linear output units. The resulting network functions are shown by the red curves, and the outputs of the three hidden units are shown by the three dashed curves.

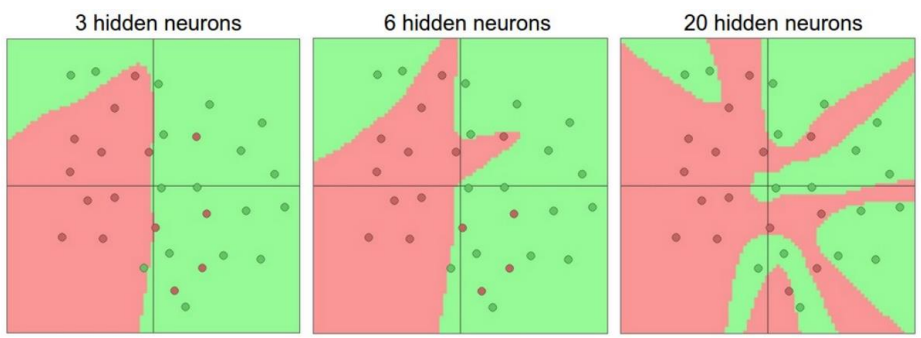


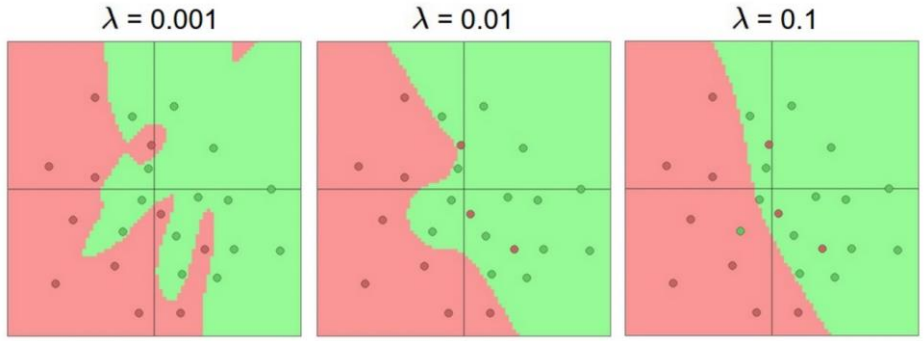


Any function: sum of bumps

Capacity of Neural Network

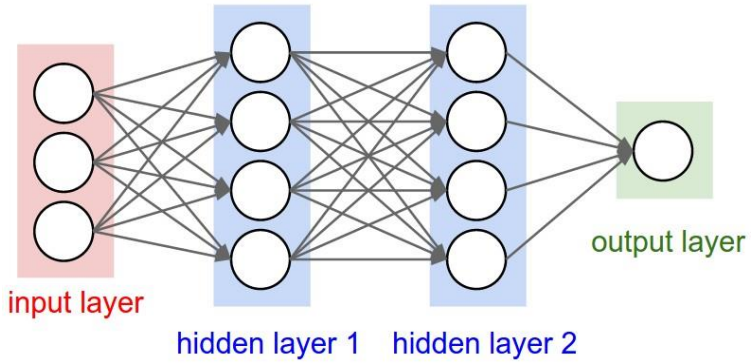
- More Neurons, More Capacity
  - But with more danger of overfitting to noise
  - Control overfitting by regularization technique, not by reducing neurons
    - Weight Regularization, Dropout, Input Noise, ...
  - Smaller network has small number of ‘bad’ local minima while larger network has many ‘good’ local minima





Neural Network

- Fully-connected Feedforward Neural Network

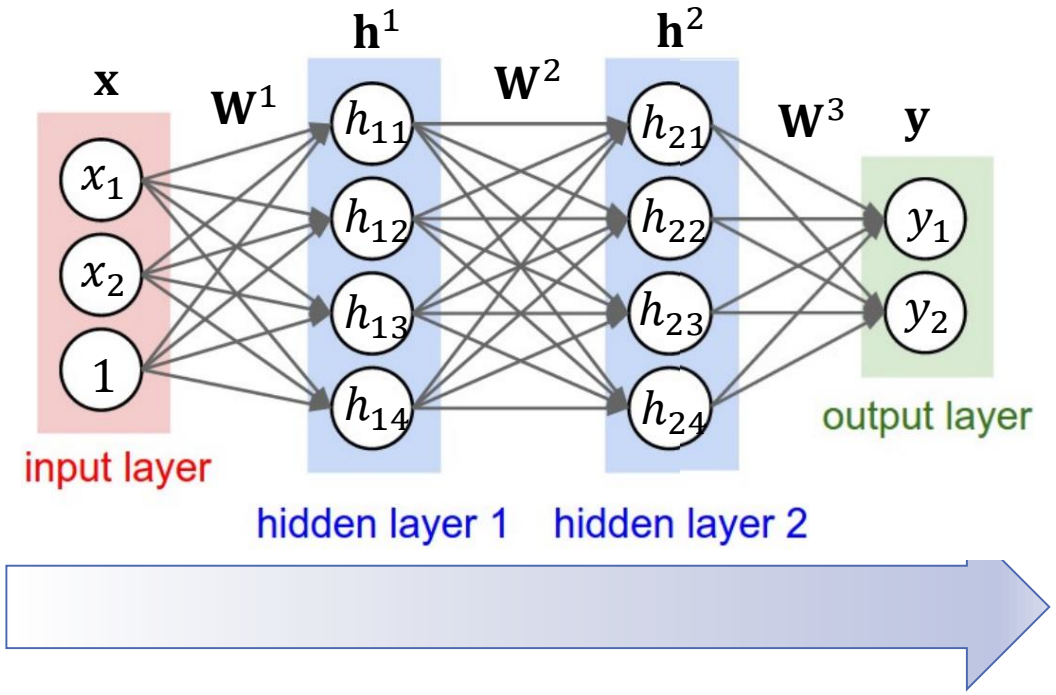


Number of Neurons: 4 + 4 + 1 = 9
Number of Weights: [4x3+4x4+1x4]=32
Number of Parameters: 32+9 = 41

- The number of layer in Neural Network = (num of all layer) – 1(input layer)
- Logistic Regression or SVM are single-layered Neural Network with special basis function and activation function
- The depth(num of layer) and number of nodes in each layer decides the representational power of Neural Networks

Forward Pass

- Forward Pass of Neural Network
  - Prediction stage
  - Calculate intermediate and final output
  - Calculate the Error



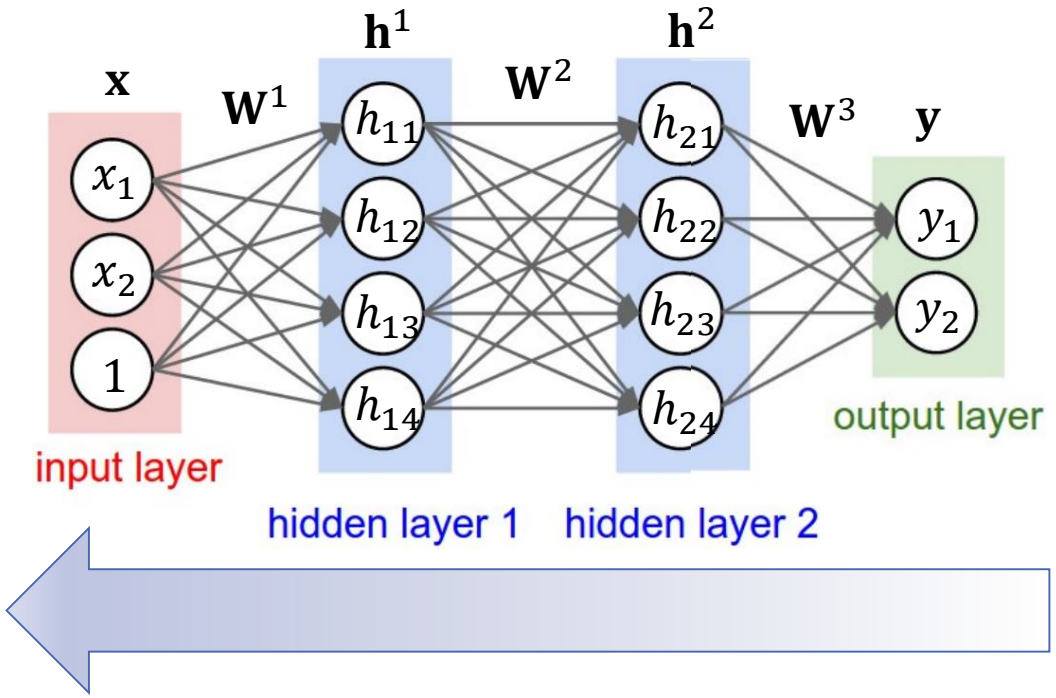
$$\mathbf{f} = Element\text{-}wise\ activation\ Function$$

$$\mathbf{y} = \mathbf{f}(\mathbf{W^3}\mathbf{f}(\mathbf{W^2}\mathbf{f}(\mathbf{W^1}\mathbf{x})))$$

$$E_{\mathbf{W}} = L(\mathbf{t}, \mathbf{y})$$

Backward Pass

- Backward Pass of Neural Network
  - Update stage
  - Calculate gradient for weight update



$$\mathbf{w} = \mathbf{w} - \lambda \frac{\partial E}{\partial \mathbf{w}}$$

$$\frac{\partial E}{\partial \mathbf{w}} = ?$$

=> Backpropagation Algorithm

Learning

- Gradient Calculation

$$E = -\sum_{n=1}^{N} \ln p(t_n \mid y_n)$$

$$= -\sum_{n=1}^{N} t_n \ln y_n + (1-t_n)\ln(1-y_n)$$

$$y_n = \sigma(z_n) = \frac{1}{1+e^{-z_n}}$$

$$z_n = \mathbf{w}^T \mathbf{x}_n + w_0, \qquad \boxed{\frac{\partial z_n}{\partial \mathbf{w}} = \mathbf{x}_n}$$

$$\frac{\partial E_n}{\partial y_n} = \frac{y_n - t_n}{y_n(1-y_n)}, \qquad \frac{dy_n}{dz_n} = y_n(1-y_n)$$

$$\frac{\partial y_n}{\partial z_n} = \frac{\partial}{\partial z_n}\frac{1}{(1+e^{-z_n})}$$

$$= \frac{1}{(1+e^{-z_n})^2}(e^{-z_n})$$

$$= \frac{1}{(1+e^{-z_n})}\left(1 - \frac{1}{(1+e^{-z_n})}\right)$$

$$\boxed{= y_n(1-y_n)}$$

$$\frac{\partial E_n}{\partial y_n} = -\frac{t_n}{y_n} + \frac{1-t_n}{1-y_n}$$

$$\boxed{= \frac{y_n - t_n}{y_n(1-y_n)}}$$

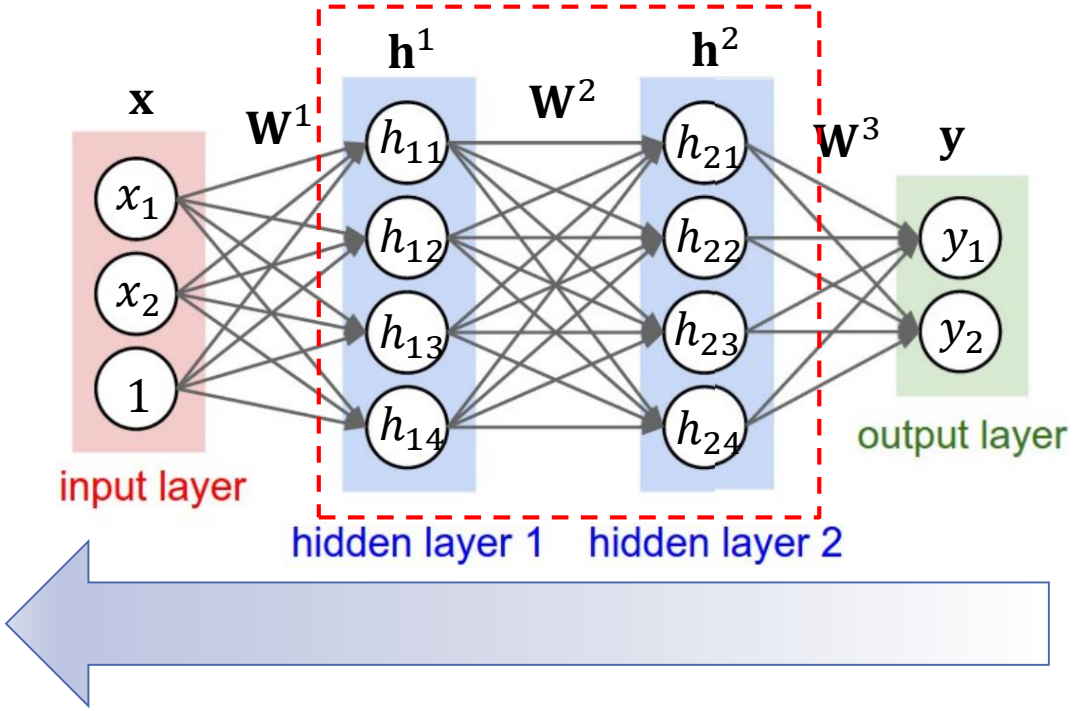
$$\boxed{\frac{\partial E_n}{\partial \mathbf{w}} = \frac{\partial E_n}{\partial y_n}\frac{dy_n}{dz_n}\frac{\partial z_n}{\partial \mathbf{w}} = (y_n - t_n)\,\mathbf{x}_n}$$

Weight Update

$$\mathbf{w} = \mathbf{w} - \lambda(y_n - t_n)\mathbf{x}_n$$

Backpropagation Algorithm

- Computing Gradient by Backpropagating Error
  - Computation of gradient using local information by chain rule



$$h^2{}_j = f\left(\sum_i w^2{}_{ij} h^1{}_i\right) = f(z_j) = \frac{1}{1+e^{-z_j}}$$

$$\frac{\partial E}{\partial w^2{}_{ij}} = \underbrace{\frac{\partial E}{\partial h^2{}_j}\frac{\partial h^2{}_j}{\partial z_j}}_{\text{delta}}\frac{\partial z_j}{\partial w_{ij}}$$

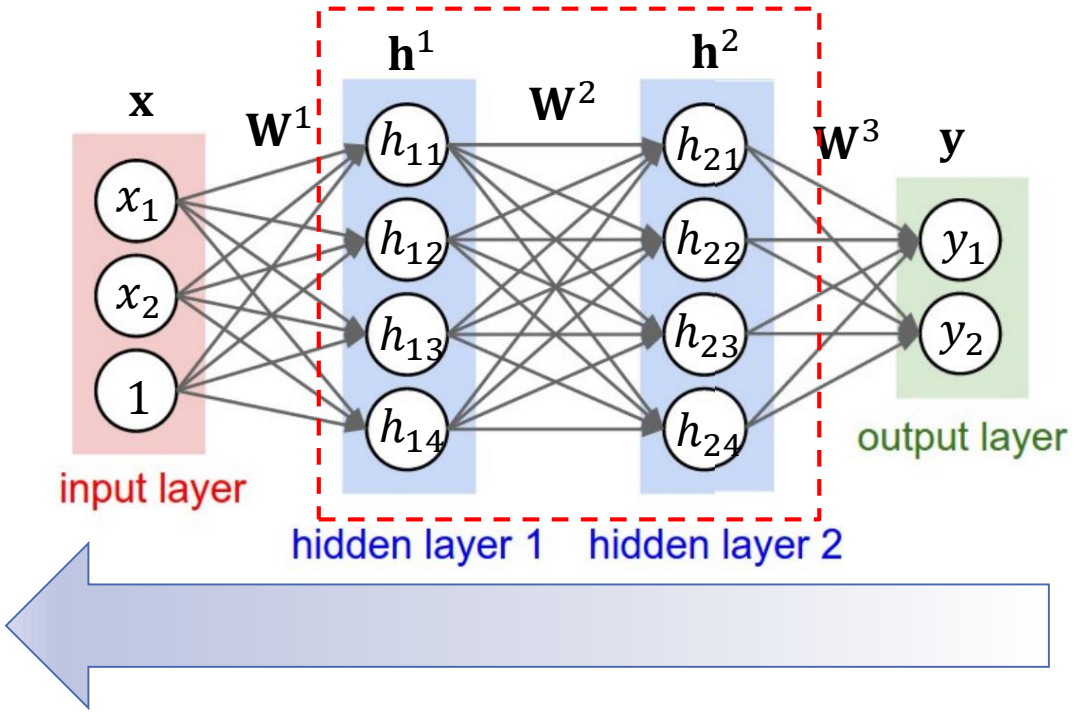
For output node

$$\delta_k = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial z_k}$$

$$\frac{\partial E}{\partial z_i} = \sum_j \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_i} = y_i - t_i \qquad \mathbf{W}_{\cdot i} = \mathbf{W}_{\cdot i} - (y_i - t_i)\mathbf{x}$$

Backpropagation Algorithm

- Computing Gradient by Backpropagating Error
  - Computation of gradient using local information by chain rule



For hidden node

$$h^2{}_j = f\left(\sum_i w^2{}_{ij} h^1{}_i\right) = f(z_j) = \frac{1}{1+e^{-z_j}}$$

$$\delta_j = \frac{\partial f(z_j)}{\partial z_j}\frac{\partial E}{\partial f(z_j)}$$

$$= \frac{\partial f(z_j)}{\partial z_j}\sum_k \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial f(z_j)}$$

$$= \frac{\partial f(z_j)}{\partial z_j}\sum_k \delta_k w_{kj}$$

$$\frac{\partial E}{\partial w_{jj'}} = \frac{\partial f(z_j)}{\partial z_j}\frac{\partial E}{\partial f(z_j)}\frac{\partial z_j}{\partial w_{jj'}}$$

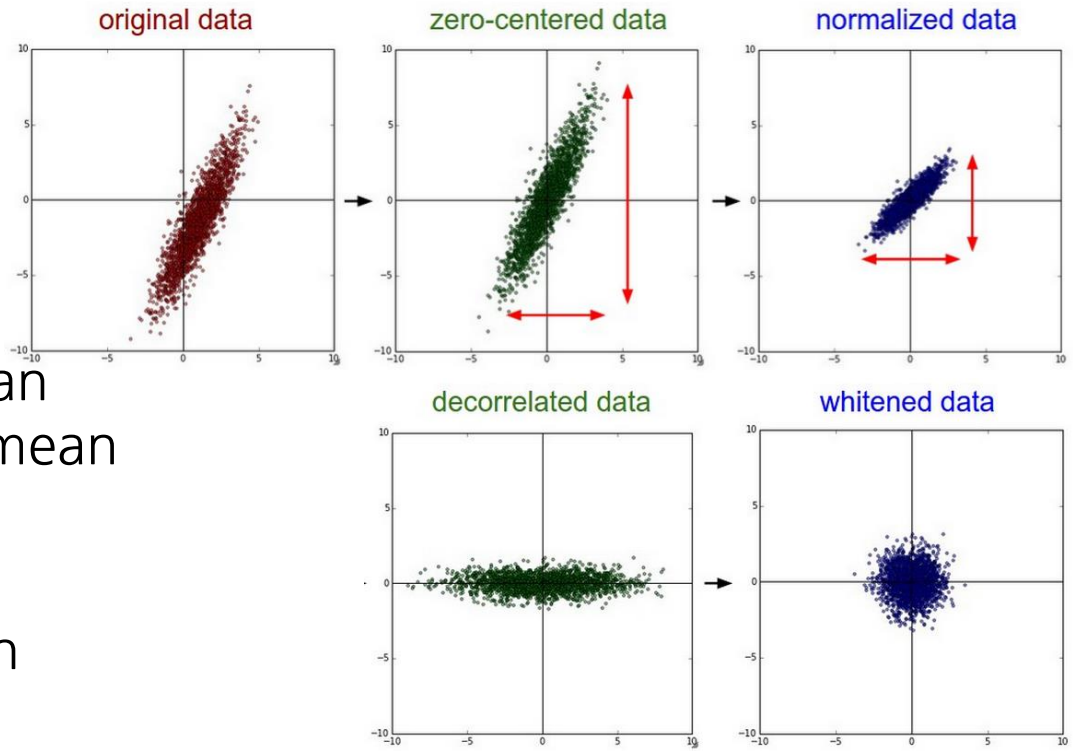
$$= \delta_j h^1{}_{j'}$$

$$w_{jj'} = w_{jj'} - \lambda\frac{\partial E}{\partial w_{jj'}}$$

Topics Related to MLP

Data Preprocessing

- Data preprocessing to control the internal statistics of learning process
  - Mean centering(Must)
  - Normalization(Recommended)
  - Decorrelating(PCA)
  - Whitening



- Practical Advice
  - For mean centering, calculated the mean only with training data, and apply the mean centering(subtraction)to all data (train/val/test)
  - Whitening must be carried with caution because it exaggerate noise

Types of Activation Functions

- Linear Function

  $g(z) = z \qquad g'(z) = 1$

- Logistic Function

  $g(z) = \frac{1}{1 + e^{-z}} \qquad g'(z) = g(z)(1 - g(z))$

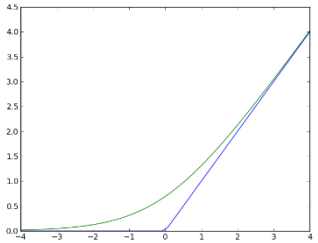
- Tanh Function

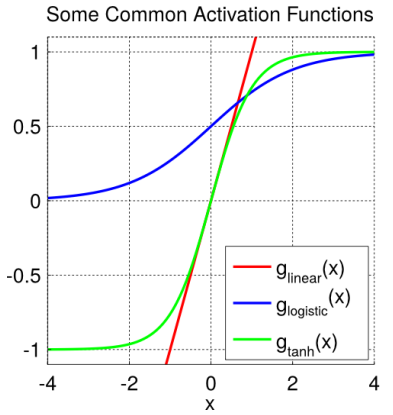
  $g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \qquad g'(z) = 1 - g(z)^2$

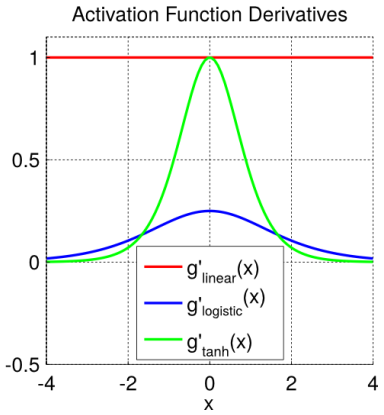
- Rectifier Unit
  - Vanilla ReLU $g(z) = \max(0, z) \qquad g'(z) = \begin{cases} 1 & if \ z > 0 \\ 0 & if \ z \leq 0 \end{cases}$
  - Leaky ReLU
  - PReLU $g(z) = \begin{cases} z & if \ z > 0 \\ az & if \ z \leq 0 \end{cases} \qquad g'(z) = \begin{cases} 1 & if \ z > 0 \\ a & if \ z \leq 0 \end{cases}$
  - Softplus $g(z) = \ln(1 + e^{z}) \qquad g'(z) = 1/(1 + e^{-z})$

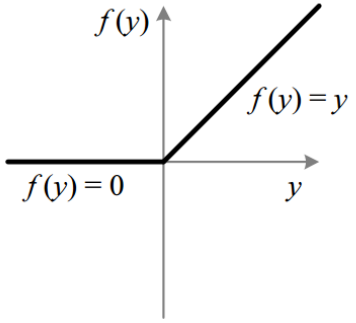
- Maxout Unit

  $g(z) = \max z_j \ \ where \ \ z_j = W_{\cdot j} x + b_j$

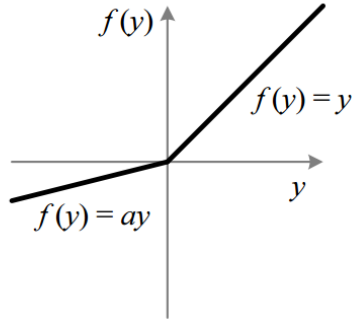








ReLU



PReLU

http://cs231n.github.io/neural-networks-1/

Choosing Activation Function

- Characteristics
  - Sigmoid activation functions saturates
    -> Kill the gradients
  - Output of logistic function is not centered
    ->Weight gradient are all positive or negative : zig-zagging
  - ReLU does not saturate and computationally efficient
    ->Leads to faster convergence
  - Leaky ReLU prevents gradient death when input to the ReLU is less than 0
  - PReLU generalizes Leaky ReLU with learnable 'leakiness'
  - Maxout generalizes ReLU with doubled parameters.

- Practical Advice
  - Use ReLU and never use logistic sigmoid(unless not avoidable)

Initialization

- All zero Initialization
  - Cannot learn : No source of asymmetry(especially for tanh)
- Small Random Number
  - Practically efficient for symmetry breaking.
  - The source of randomness has relatively little impact on the performance
  - Too small initial weight leads to slow learning.
- Controlling the variance of internal data(for linear units)
  - Xavier Initialization

$$\mathrm{Var}(W) = \frac{1}{n_{\mathrm{in}}}$$

  - Glorot & Bengio

$$\mathrm{Var}(W) = \frac{2}{n_{\mathrm{in}} + n_{\mathrm{out}}}$$

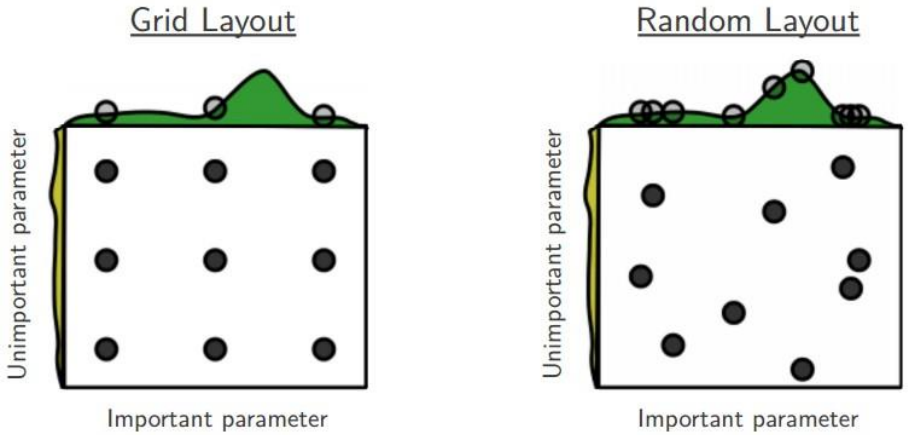
  - ReLU Networks

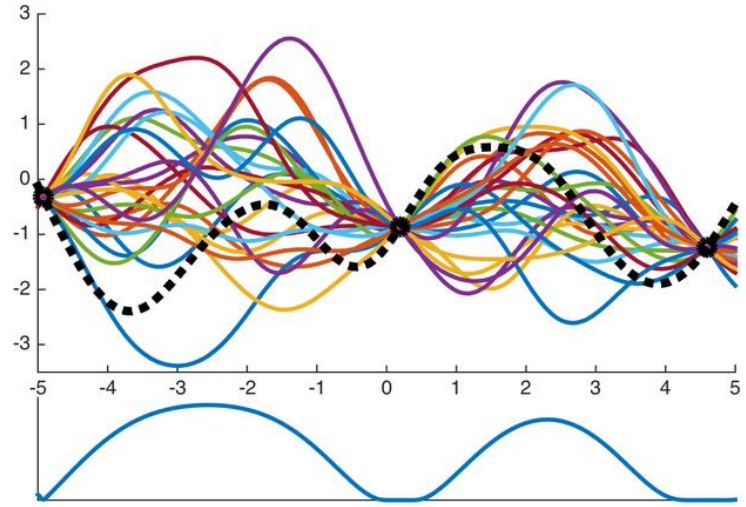
$$\mathrm{Var}(W) = \frac{2}{n_{\mathrm{in}}}$$

- Practical Advice
  - Use ReLU with Initialization of Weight 2/n_in and bias to 0

Model Selection

- Grid Search
  - Select each hyperparameter from the preset grid
- Random Search
  - Randomly choose the combination of hyperparameters
- Bayesian Optimization
  - Inferring performance based on the known performance of hyperparameter combination
  - Exploration-Exploitation Trade-off
- Practical Advice
  - Use random search in the careful region





Regularization

- Controlling the Capacity of NN to Prevent Overfitting
  - L1 Regularization
  - L2 Regularization
  - Max Norm
  - Dropout
  - Input Noise

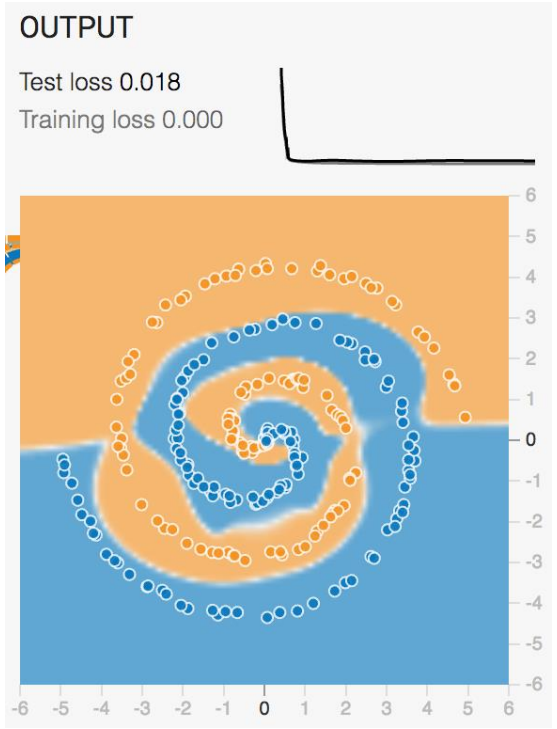
=> More in Detail in the Next Lecture !!

Lab

Lab

- Logistic Regression and Multi-layered Perceptron
  - https://github.com/KyuhwanJung/tensorflow_tutorial/blob/master/Logistic%20Regression%20and%20Multi-layered%20Perceptron.ipynb

- Play with TensorFlow Playground
  - http://playground.tensorflow.org/
  - Exercise
    - Make the training error of spiral example to 0.001



Class 3
Regularization and Optimization

Problem Definition

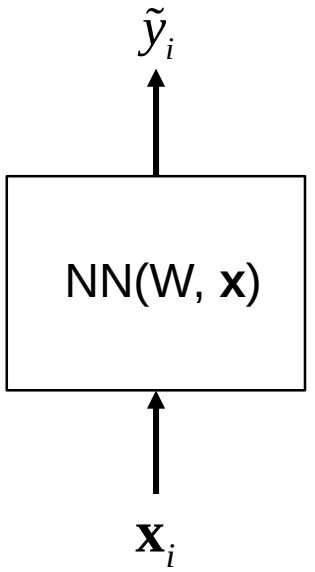
The Objective Function of DNN

- Empiricial Risk Minimization

Let $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n) \in \mathbf{R}^d \times \{1, \ldots, k\}$ be labelled training dataset, we should solve

$$\min_W \boxed{\lambda \Omega(W)} + \boxed{\frac{1}{n} \sum_{i=1}^{n} L(y_i, f(W, \mathbf{x}_i))}$$

Regularization Problem Dependent Loss

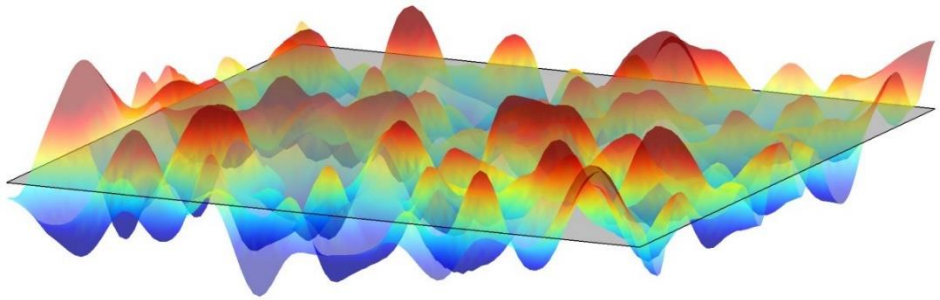


Problem: minimizing such objectives in the large-scale setting is difficult.

$$n \gg 1, \quad d \gg 1, \quad k \gg 1$$

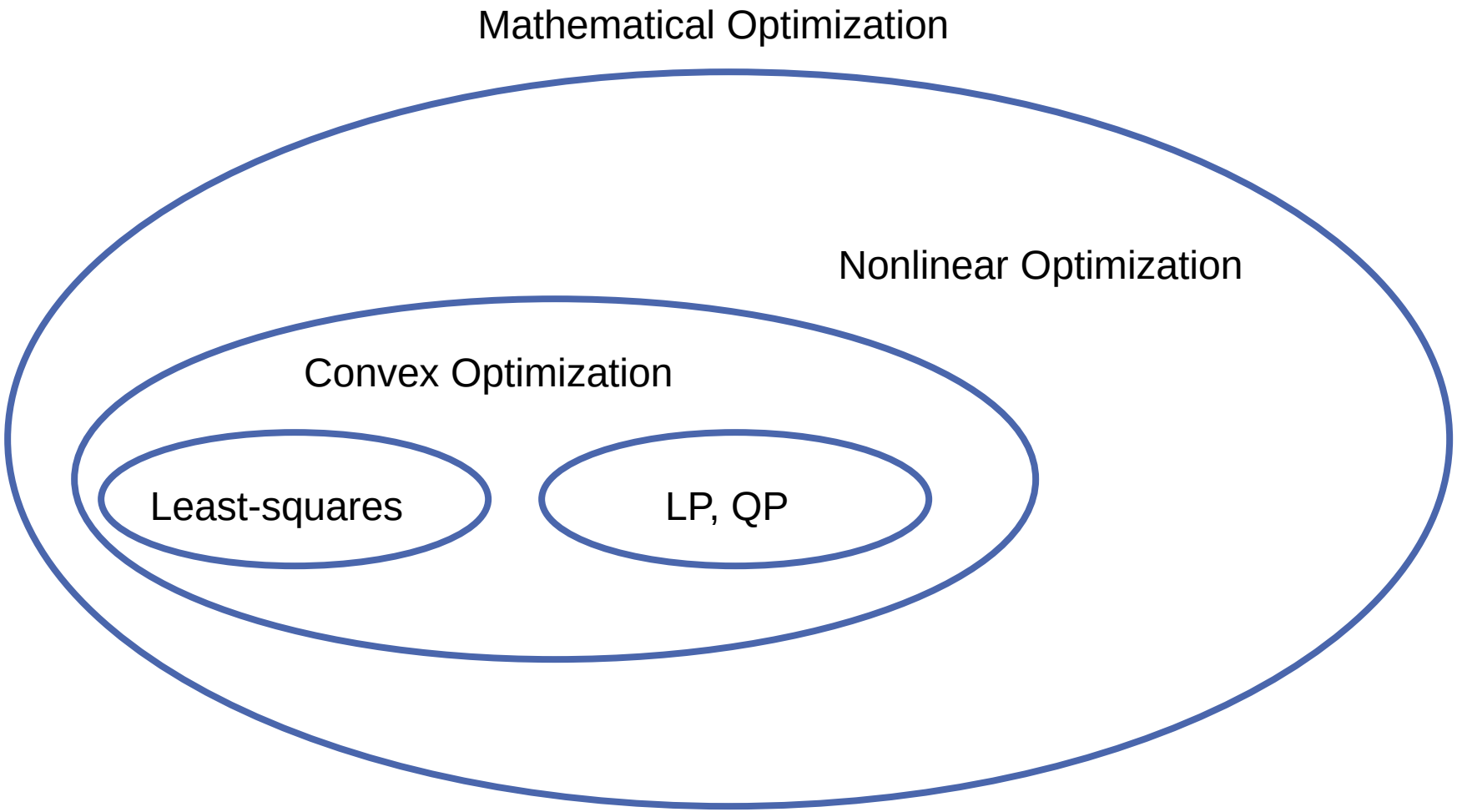
Challenge

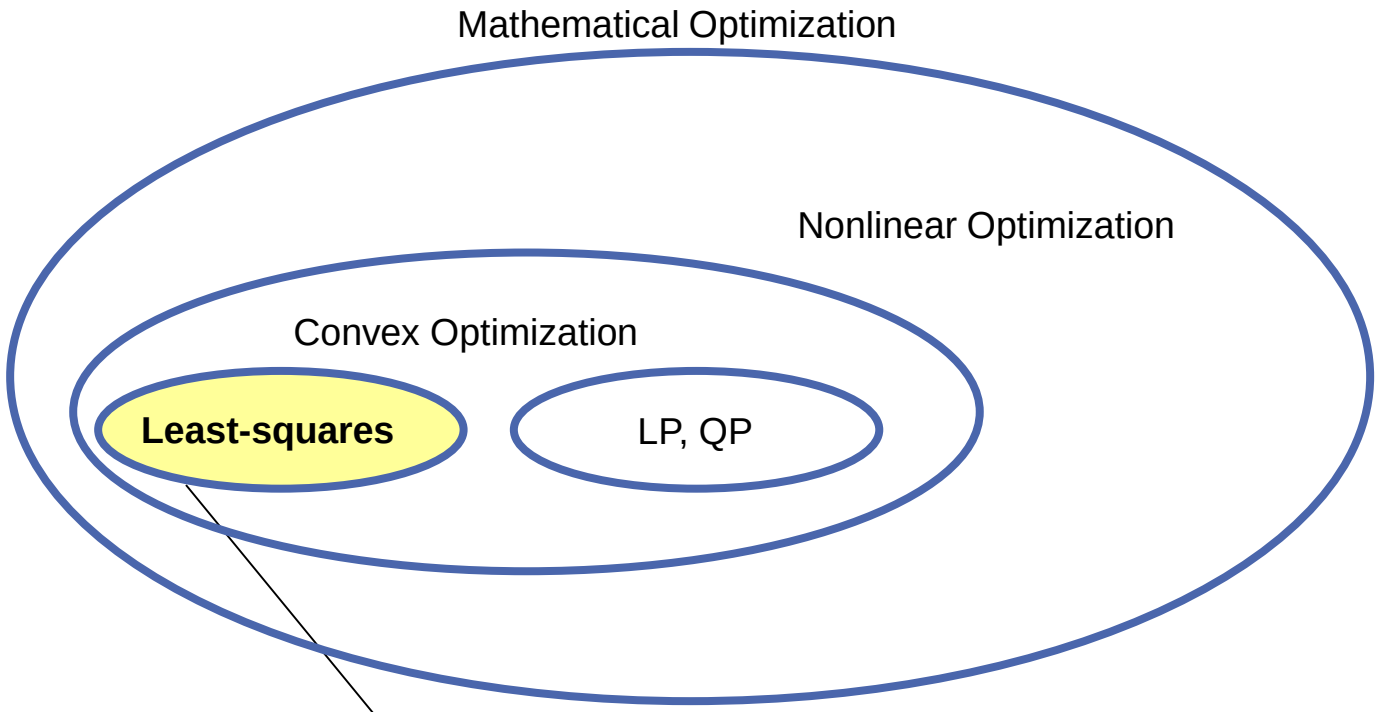
- Understanding the Loss Function of Deep Neural Network
  - The loss function is far from convex function
  - Very high-dimensional
  - Many local minima
  - Many saddle points



Recent related works: Goodfellow et al., 2015, Choromanska et al., 2015, Dauphin et al., 2014, Saxe et al., 2014.

Solving Optimization Problems

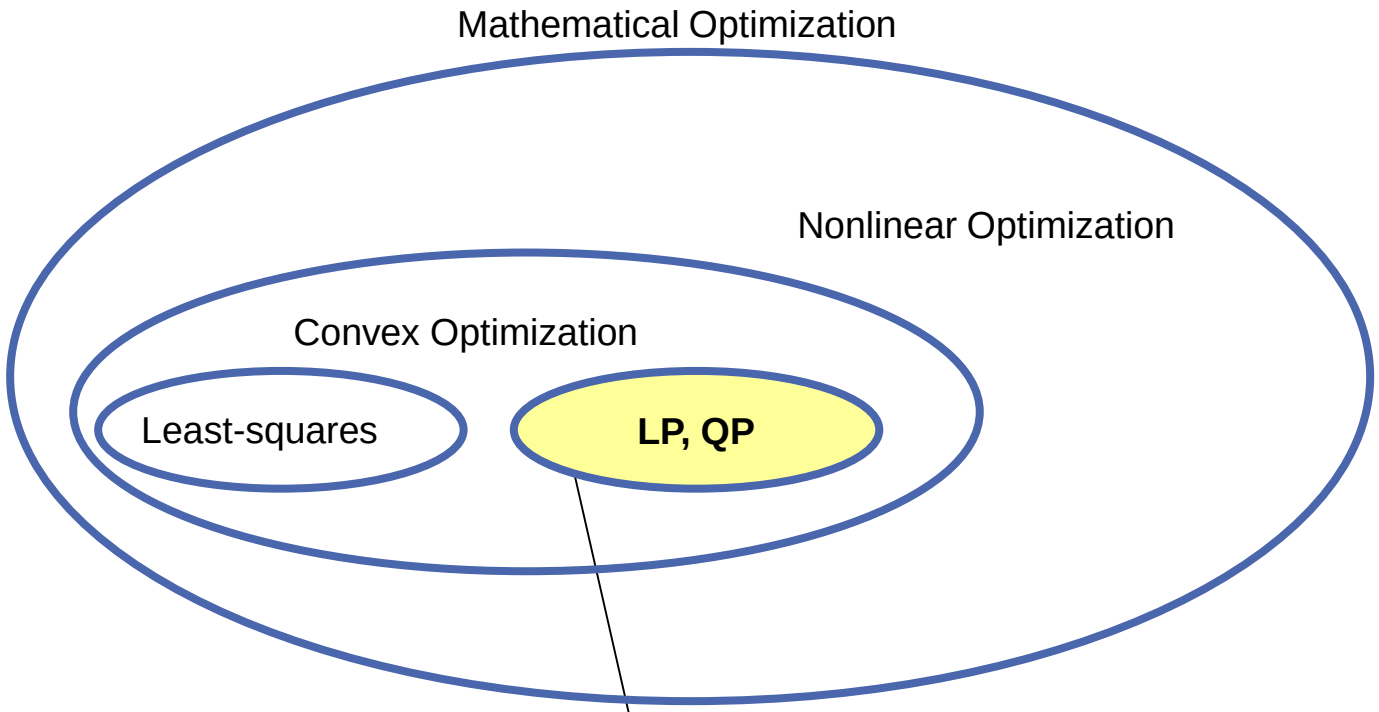




minimize $\|Ax - b\|_2^2$

- Analytical solution
- Good algorithms and software
- High accuracy and high reliability
- Time complexity: $C \cdot n^2 k$

A mature technology!



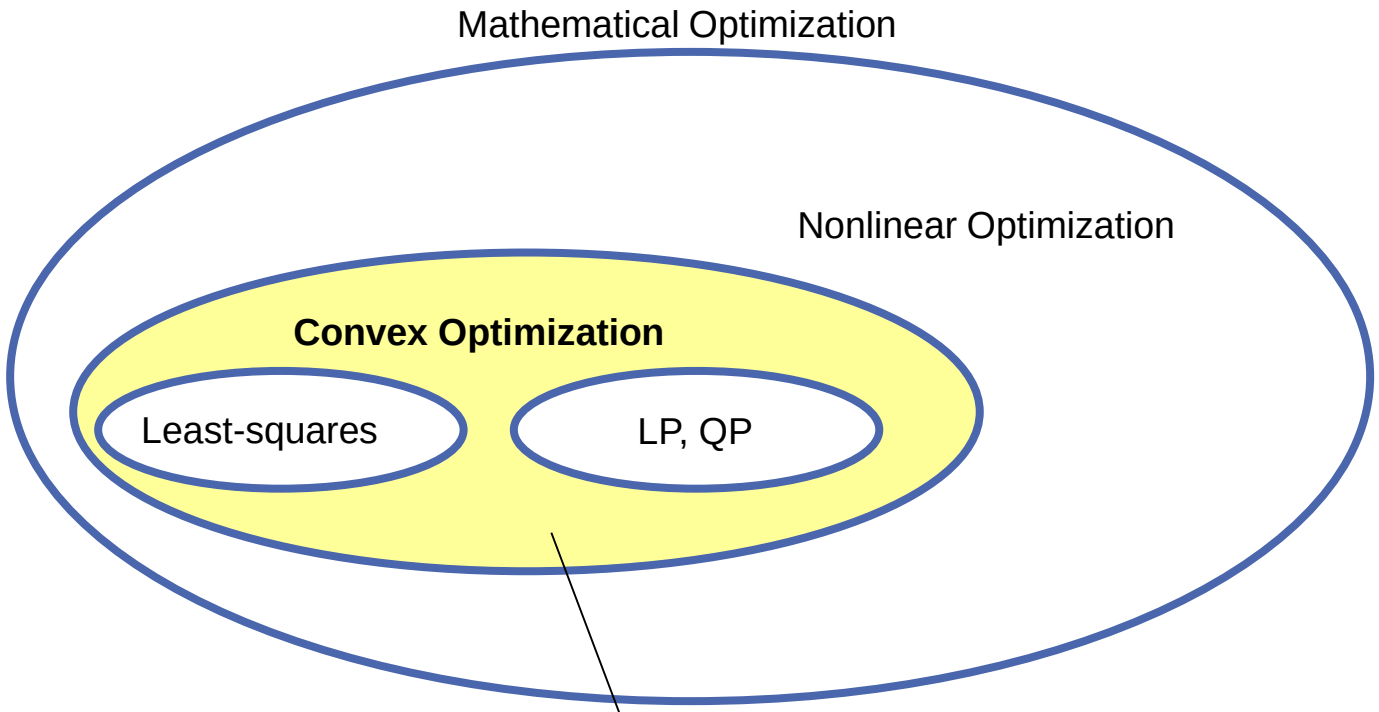
$$\begin{aligned} \text{maximize} \quad & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} \leq \mathbf{b} \\ \text{and} \quad & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

$$\begin{aligned} \text{minimizes} \quad & \tfrac{1}{2}\mathbf{x}^{\mathrm{T}}Q\mathbf{x} + \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} \leq \mathbf{b}. \end{aligned}$$

- No analytical solution
- Algorithms and software
- Reliable and efficient
- Time complexity: $C \cdot n^2 m$

Also a mature technology!

https://en.wikipedia.org/wiki/Linear_programming
https://en.wikipedia.org/wiki/Quadratic_programming



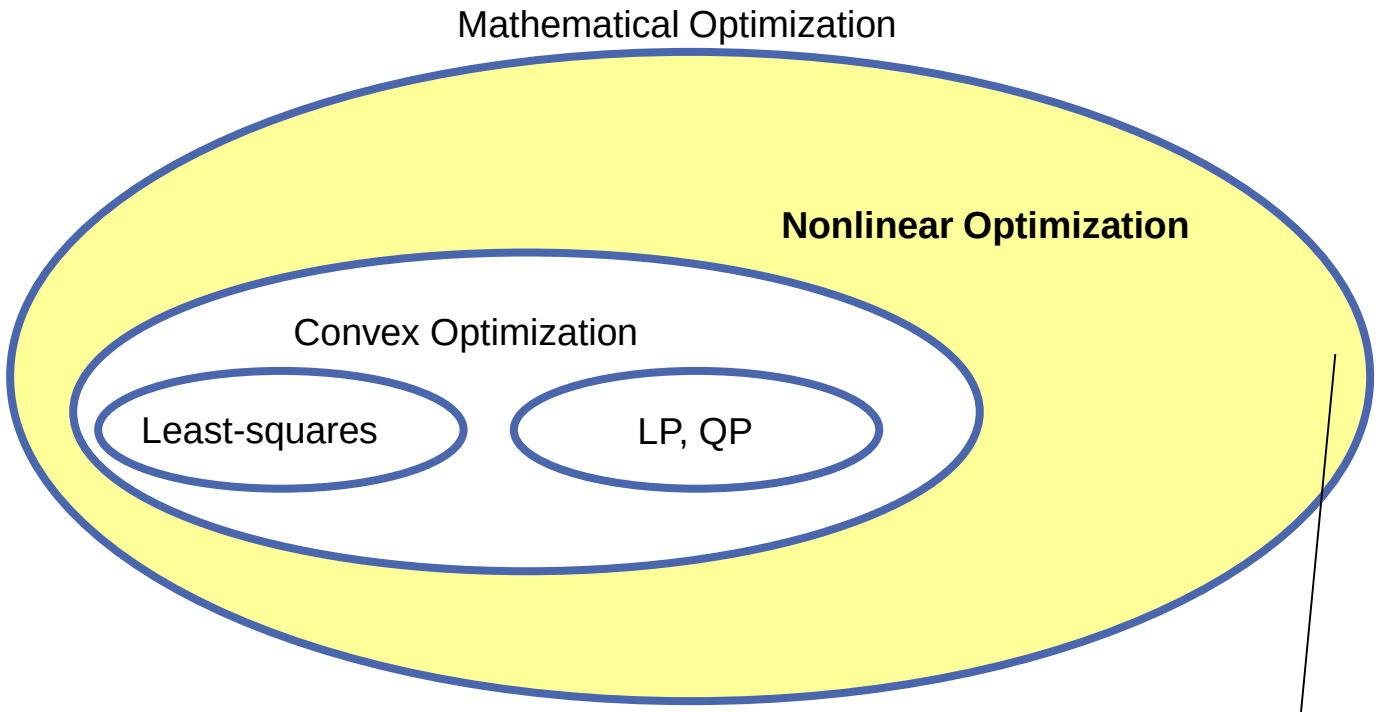
minimize $f_0(x)$
subject to $f_i(x) \leq b_i, \quad i = 1, \ldots, m$

- No analytical solution
- Algorithms and software
- Reliable and efficient
- Time complexity (roughly)

$$\propto \max\{n^3, n^2 m, F\}$$

F is cost of evaluating f_i's and their first and second derivatives

Almost a ~~mature~~ technology!



- Sadly, no effective methods to solve
- Only approaches with some compromise
- Local optimization: *"more art than technology"*
- Global optimization: greatly compromised efficiency
- Help from convex optimization
  1) Initialization 2) Heuristics 3) Bounds

Far from a technology! (something to avoid)

The Strategy

- The Objective Function is Not Convex. So What Can We Do?
  - Let the initial point close to optimal area => Pre-training
  - Utilize convex optimization techniques => It's locally convex around minima
  - But they're local minima => Use stochastic approach to escape from them(but seriously?)

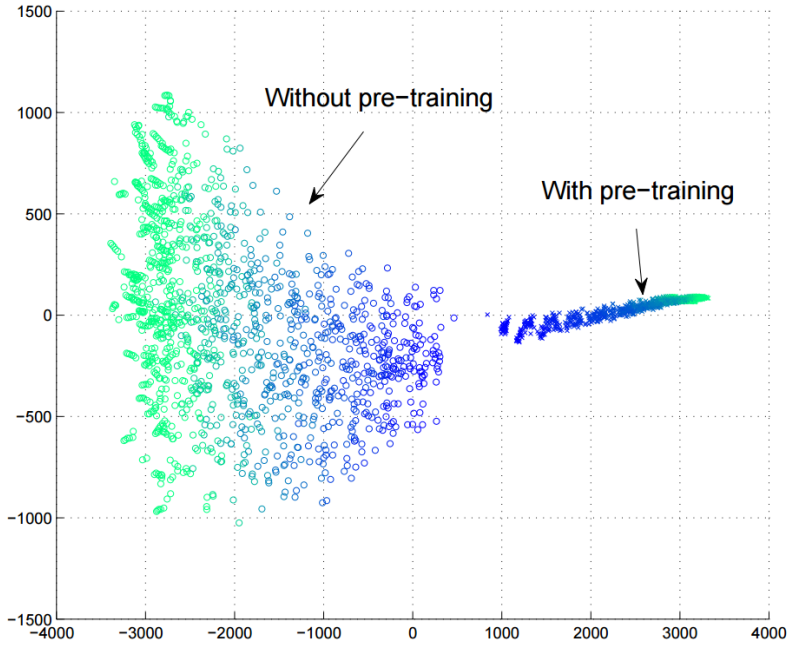
- But... is that it?
  - Yes. This is the best move we can take(with large-scale data) and it works surprisingly well !
  - Local minima is not the most important issue.
    - We will see that most of them are actually saddle point, not local minima
    - We will see that local minima are not so bad
    - We will see that underfitting is a more of a problem than local minima

Pre-training

- *(Erhan et al 2009, JMLR)*
- Supervised deep net (tanh), with or w/o unsupervised pre-training → very different minima

Neural net trajectories in function space, visualized by t-SNE

No two training trajectories end up in the same place → huge number of effective local minima



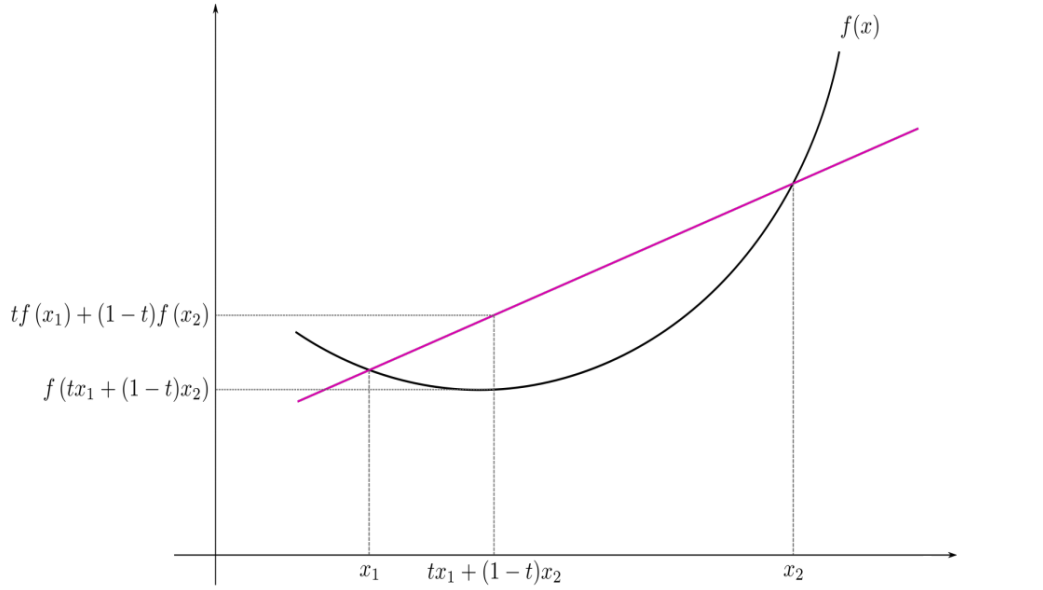
Convex Optimization

Convex Optimization

- Definition of Convex Function

A function is f: $\Re^n \rightarrow \Re$ is convex if and only if

$$\forall x_1, x_2 \in \text{Domain}(f), \forall t \in [0,1]:$$
$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$



Steepest Descent

- Idea
  - Start from somewhere
  - Repeat moving to the direction of maximum decrease of objective function

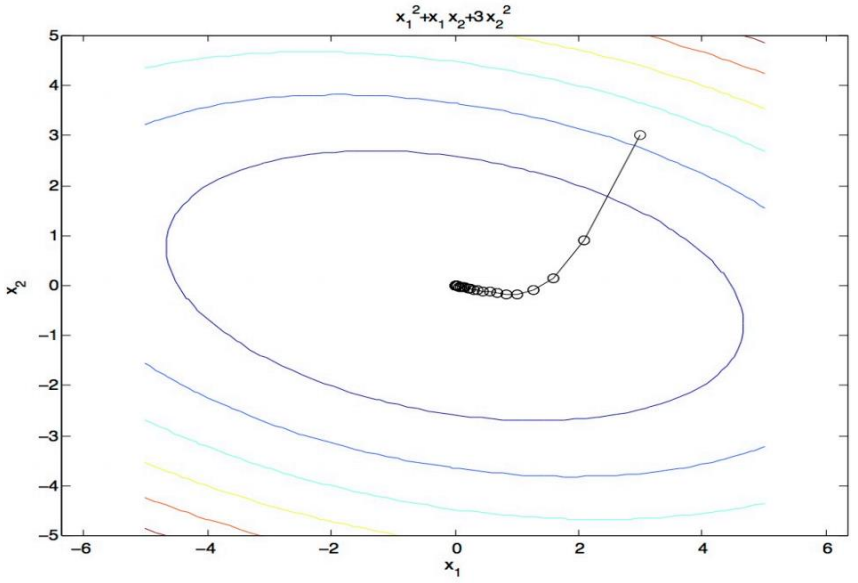
- Which one is the steepest direction?
  - With smooth function,

$$f(\mathbf{w}_0 + \Delta\mathbf{w}) \approx f(\mathbf{w}_0) + \nabla_{\mathbf{w}} f(\mathbf{w}_0)^T \Delta\mathbf{w}$$

  - Then, $\Delta\mathbf{w}^* = \arg\min_{\|\mathbf{w}\|_2=1} f(\mathbf{w}_0) + \nabla_{\mathbf{w}} f(\mathbf{w}_0)^T \Delta\mathbf{w}$

$$= \arg\min_{\|\mathbf{w}\|_2=1} \nabla_{\mathbf{w}} f(\mathbf{w}_0)^T \Delta\mathbf{w}$$

$$= -\nabla_{\mathbf{w}} f(\mathbf{w}_0) \quad \Rightarrow \text{Gradient Descent !!}$$

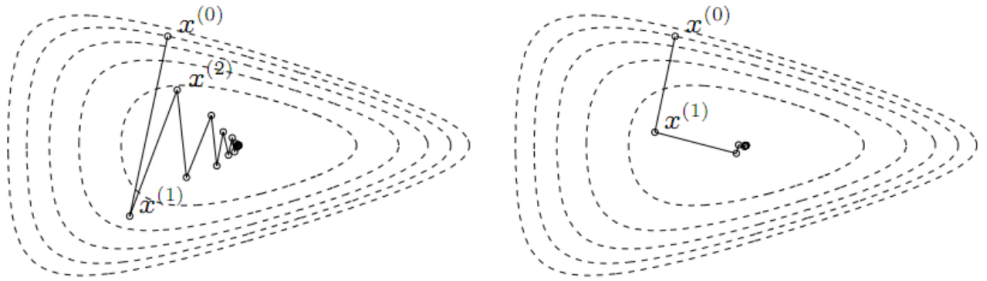


Gradient Descent

- Update Rule
  - So... we have the direction $\Delta \mathbf{w}$ and update rule as

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \Delta \mathbf{w}$$

- Step Length
  - Then how much should go?



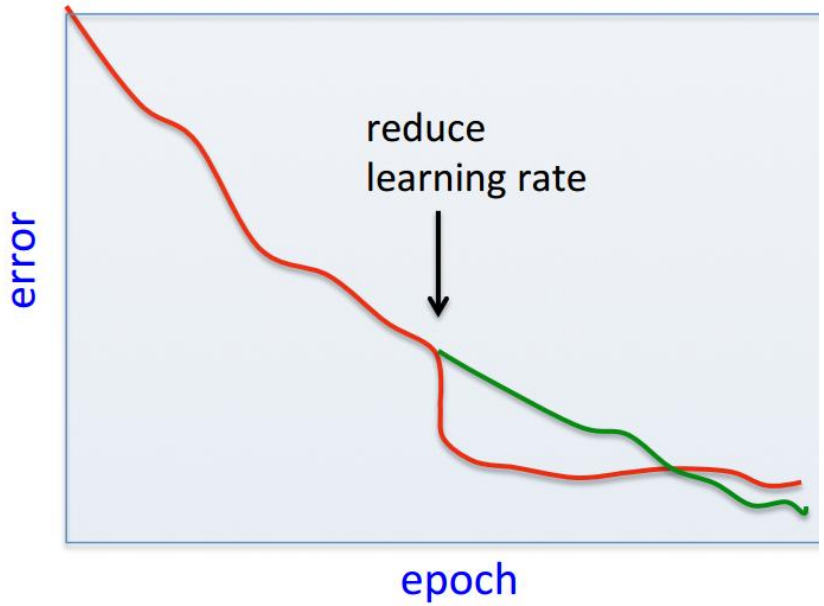
backtracking line search exact line search

  - Both are not feasible for large scale data! => Use simpler but thoughtful strategy !

Learning Rate

- Learning Rate Scheduling

  - Turning down the learning rate reduces the random fluctuations in the error due to the different gradients on different mini-batches.
    - So we get a quick win.
    - But then we get slower learning.
  - Don't turn down the learning rate too soon!

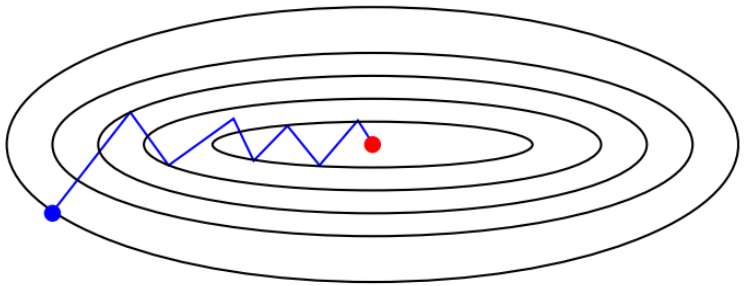


- For DNNs, start with learning rate as small as 1e-2 to 1e-3 and decrease them as a factor of 10 when error plateaus. (or at pre-scheduled time)

Stochastic Gradient Descent

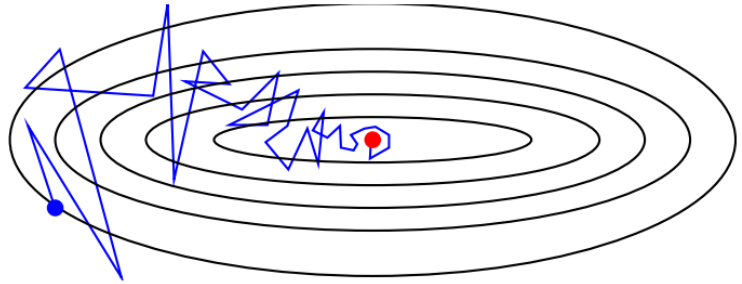
- Deterministic(Batch) vs Stochastic Gradient Descent

  - Deterministic gradient method [Cauchy, 1847]:



$$w_{t+1} = w_t - \eta \cdot \frac{1}{N} \sum_{i=1}^{N} \frac{E(x_i)}{\partial w_t}$$

  - Stochastic gradient method [Robbins & Monro, 1951]:



$$w_{t+1} = w_t - \eta \cdot \frac{E(x_i)}{\partial w_t}$$

Stochastic iterations are N times faster, but how many iterations?

Assumption	Deterministic	Stochastic
Convex	$O(1/t^2)$	$O(1/\sqrt{t})$

Why SGD?

- Advantages
  - Faster iterations.
  - We don’t need to store all the batch computations => Reduce memory
  - With some chance, we can escape from local minima

- Disadvantages
  - Too noisy(hard to monitor)
  - Needs strategy to tune learning rate(otherwise it will not converge forever)

⇨ Mini-batch Gradient Descent

$$w_{t+1} = w_t - \eta \cdot \frac{1}{B} \sum_{i=1}^{B} \frac{E(x_i)}{\partial w_t} \quad \text{where} \quad B << N$$

Why SGD?

- Batch Gradient Descent

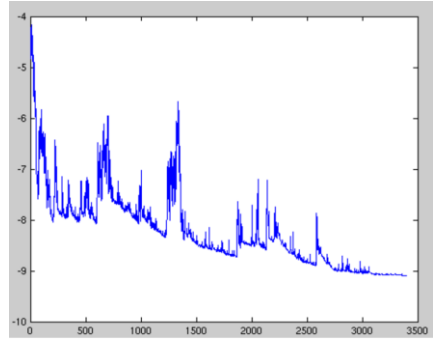
$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta).$$

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

- Stochastic Gradient Descent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}).$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```



- Mini-batch Gradient Descent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}).$$

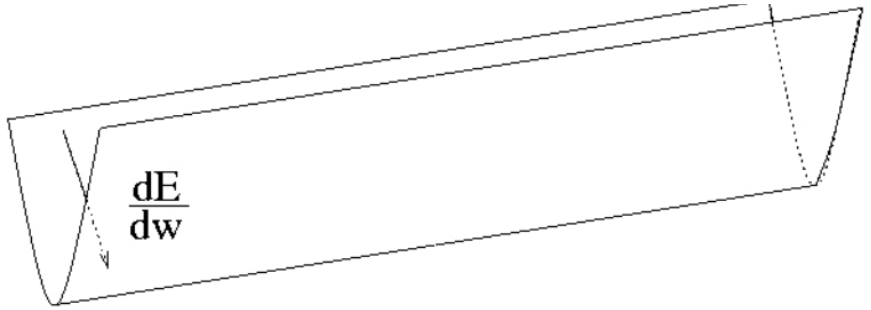
```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Momentum

- Let the ball roll
  - If the error surface is ravine shaped which is often the case for deep neural network,
    it takes too long time to reach local minima.
  - We can solve this problem by combining gradient of previous iteration and current iteration as

$$\Delta w_j^t = \beta \Delta w^{t-1} + (1-\beta)\left(-\epsilon\, \partial E / \partial w_j(\mathbf{w}^t)\right)$$

  - We call the term of previous update as **momentum** with parameter $0 \le \beta < 1$
  - We frequently set β to big value as 0.9 ~ 0.95
  - When $\beta = 0$, it's ordinary SGD



Momentum

- Let the ball roll

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) .$$

$$\theta = \theta - v_t .$$

Image 2: SGD without momentum

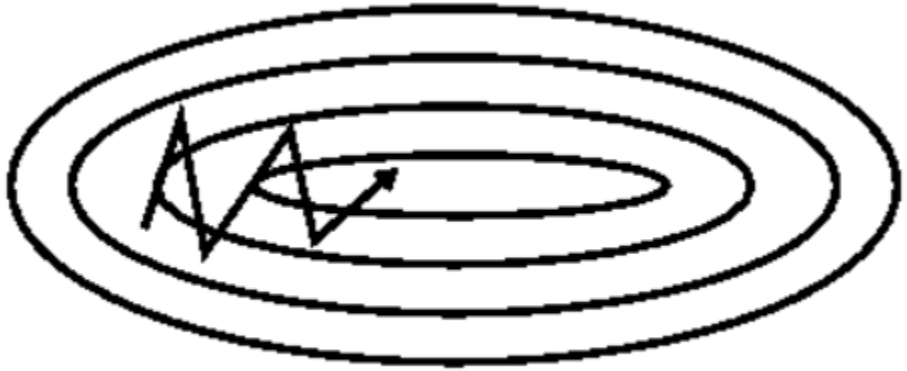


Image 3: SGD with momentum

Nesterovs' Accelerated Gradient Method

- Accelerating SGD by Using 'Move First, Correct Next'

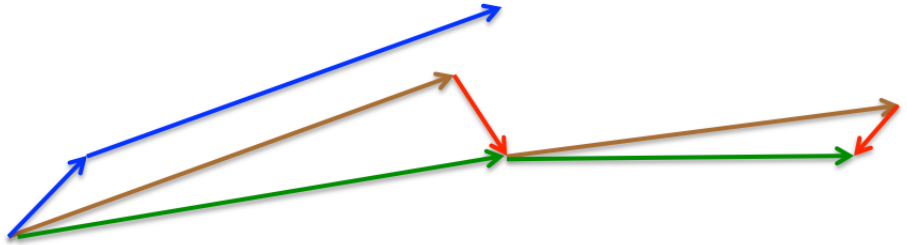
Standard Momentum

$$v_{t+1} = \lambda_t v_t - \varepsilon_t \nabla f(w_t)$$
$$w_{t+1} = w_t + v_t$$
$$= w_t + \lambda_t v_t - \varepsilon_t \nabla f(w_t)$$

NAG

$$v_{t+1} = \lambda_t v_t - \varepsilon_t \nabla f(w_t + \lambda_t v_t)$$
$$w_{t+1} = w_t + v_t$$
$$= w_t + \lambda_t v_t - \varepsilon_t \nabla f(w_t + \lambda_t v_t)$$

- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.



brown vector = jump, red vector = correction, green vector = accumulated gradient

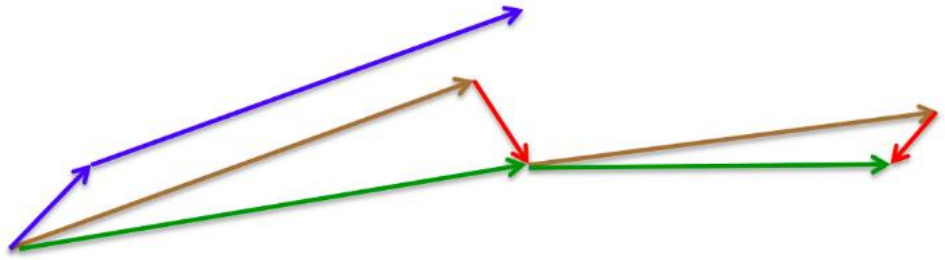
blue vectors = standard momentum

Nesterovs' Accelerated Gradient Method

- Accelerating SGD by Using 'Move First, Correct Next'
  - Use pre prescience of next positive using momentum term
  - In the momentum update, we will used last momentum term(with newly computed gradient) for update the parameters.
  - We can look ahead by only using momentum term for better navigate the direction.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) .$$

$$\theta = \theta - v_t .$$

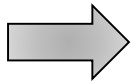


Per-parameter Learning Rate Adaptation

- AdaGrad

$$cache = \sum_{k=1}^{t} (\nabla f(w_k))^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{cache}} \nabla f(w_t)$$



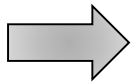
- Cache is always increasing
- Stops too early

- AdaDelta

$$cache_t = \gamma(\nabla f(w_t))^2 + (1-\lambda) cache_{t-1}$$

$$w_{t+1} = w_t - \frac{\sqrt{s_{t-1}}}{\sqrt{cache_t}} \nabla f(w_t)$$

$$s_t = \gamma s_{t-1} + (1-\lambda)(\Delta w_t)^2$$

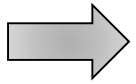


- Learning rate is not necessary
- Moving average both gradient and steps

- rmsprop

$$cache_t = \gamma(\nabla f(w_t))^2 + (1-\lambda) cache_{t-1}$$

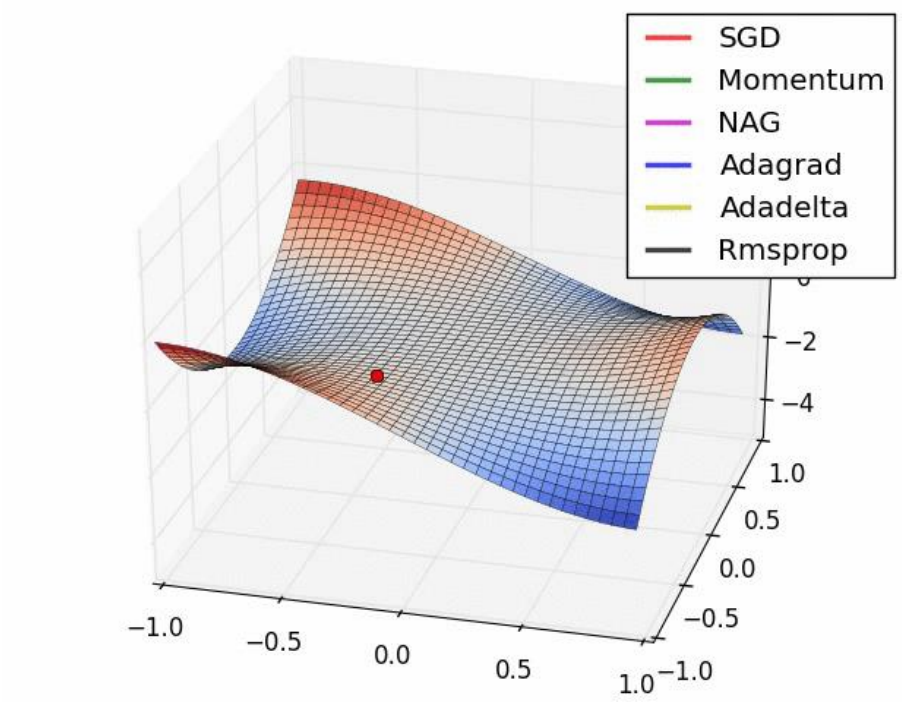
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{cache_t}} \nabla f(w_t)$$

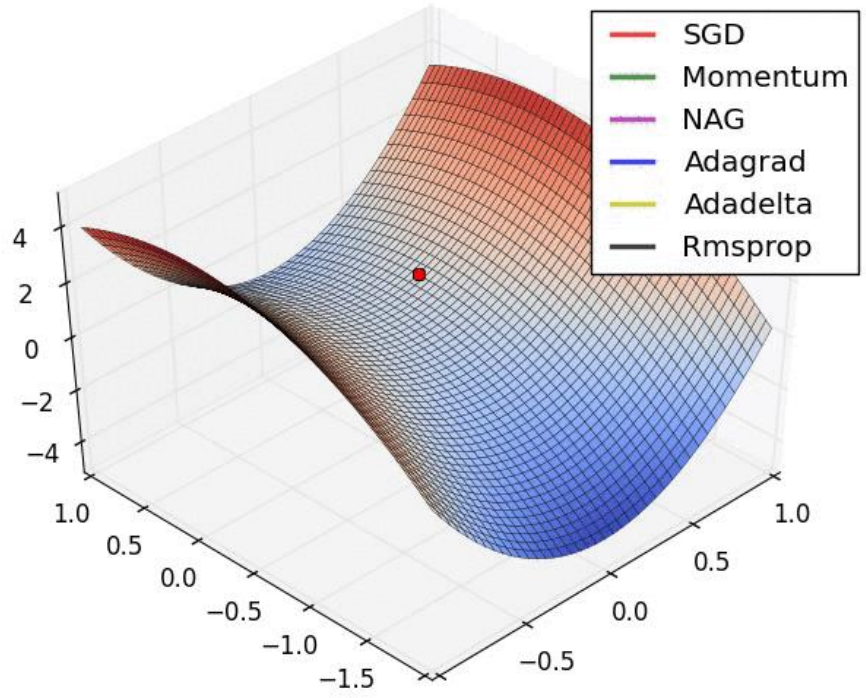


- Robust
- Provides Pseudo-curvature

Better Optimization Strategies

- Examples





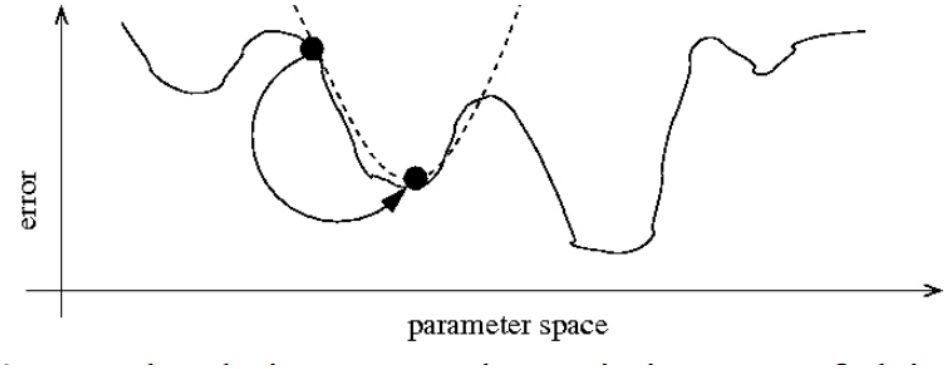
Second-order Methods

- Newton's Method
  - Second order Taylor approximation near $\mathbf{w}$ is

$$f(\mathbf{w}+\Delta\mathbf{w}) \approx f(\mathbf{w}) + \nabla f(\mathbf{w})^T \Delta\mathbf{w} + \frac{1}{2}\Delta\mathbf{w}^T \nabla^2 f(\mathbf{w})\Delta\mathbf{w}$$

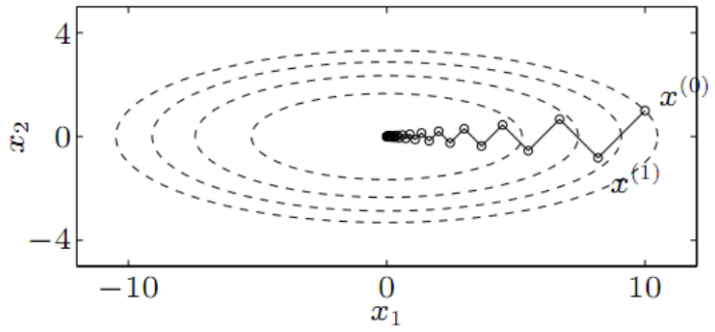
  - Assuming $\nabla^2 f(\mathbf{w})$ is positive semi-definite,

$$\Delta\mathbf{w}^* = -(\nabla^2 f(\mathbf{w}))^{-1}\nabla f(\mathbf{w}) = -\mathrm{H}^{-1}(\mathbf{w})g(\mathbf{w})$$

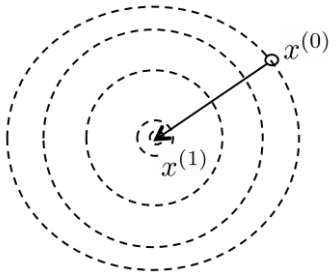


Why Second-order Methods?

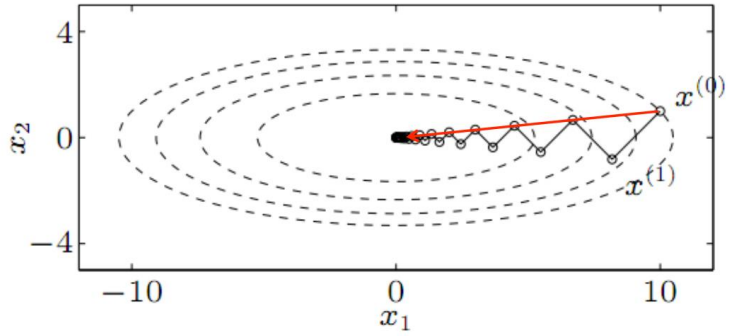
- The Curvature
  - For quadratic function, convergence speed depends on ratio of highest second derivative over lowest second derivative ("condition number")
  - In high dimensions, almost guaranteed to have a high (=bad) condition number



Condition number = 10



Condition number = 1



- Gradient descent
- Newton's method (converges in one step if f convex quadratic)

Then Why Not?

- The Cost
  - It can be very expensive to calculate and store the Hessian matrix. => It's (Dim x Dim)
  - We need very large batch(even all training data) to estimate Hessian well.

- So ... Then What?
  - Conjugate Gradient
  - L-BFGS
  - Hessian Free Optimization
  - and Many

I have *never* seen them work better than SGD in practice.

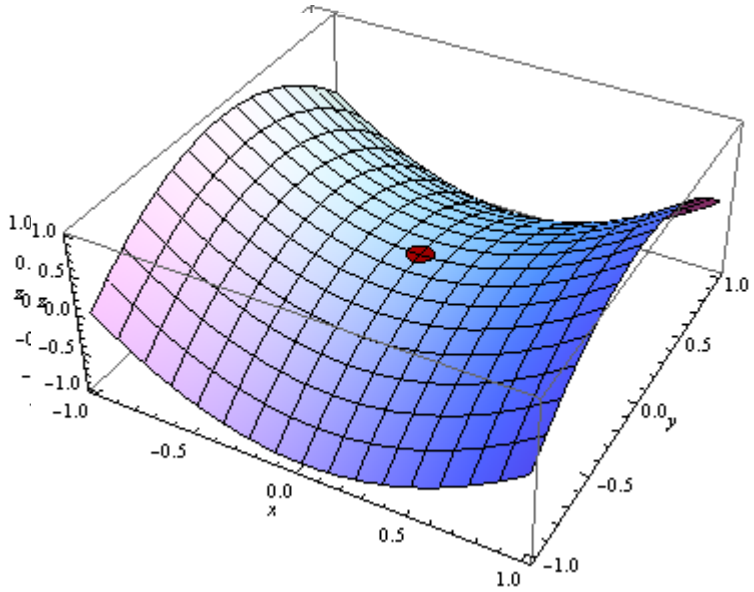
-Vincent Vanhoucke, Lead of Google Brain Team

Saddle Point Issues

Saddle Points

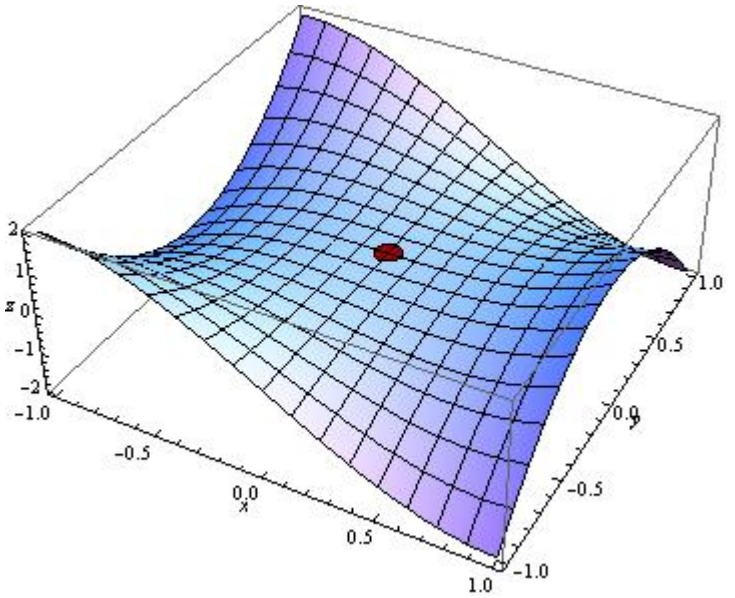
- Types of Saddle Points

Ordinary Saddle



$z = x^2 - y^2$

Monkey Saddle



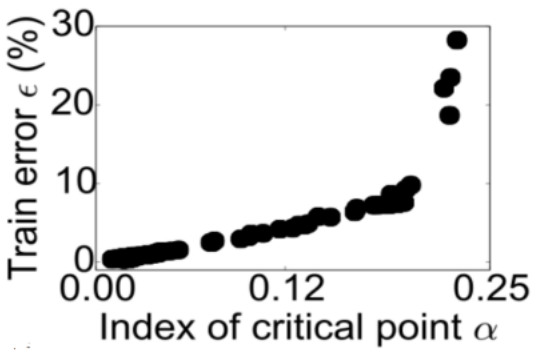
$z = x^3 - 3xy^2$

Saddle Points

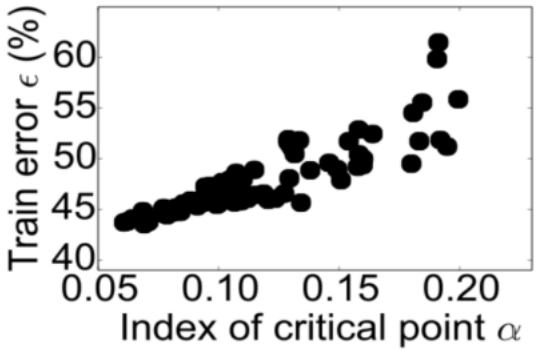
- Loss vs Index of Saddle Point

  - Local minima dominate in low-D, but saddle points dominate in high-D
  - Most local minima are close to the bottom (global minimum error)

MNIST



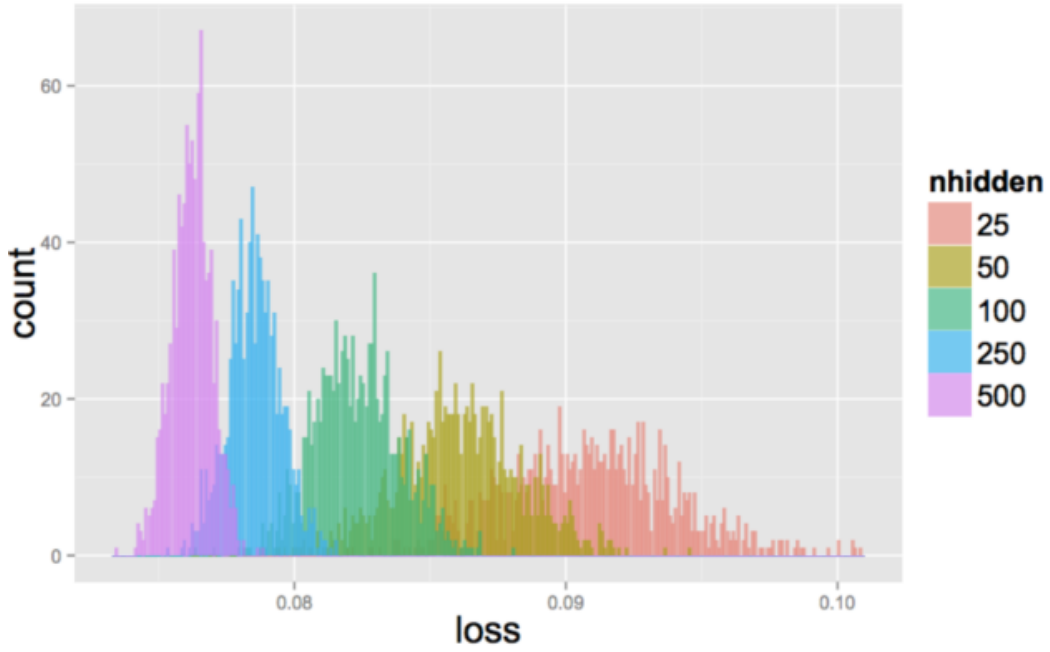
CIFAR



Saddle Points

- Loss vs Index of Saddle Point

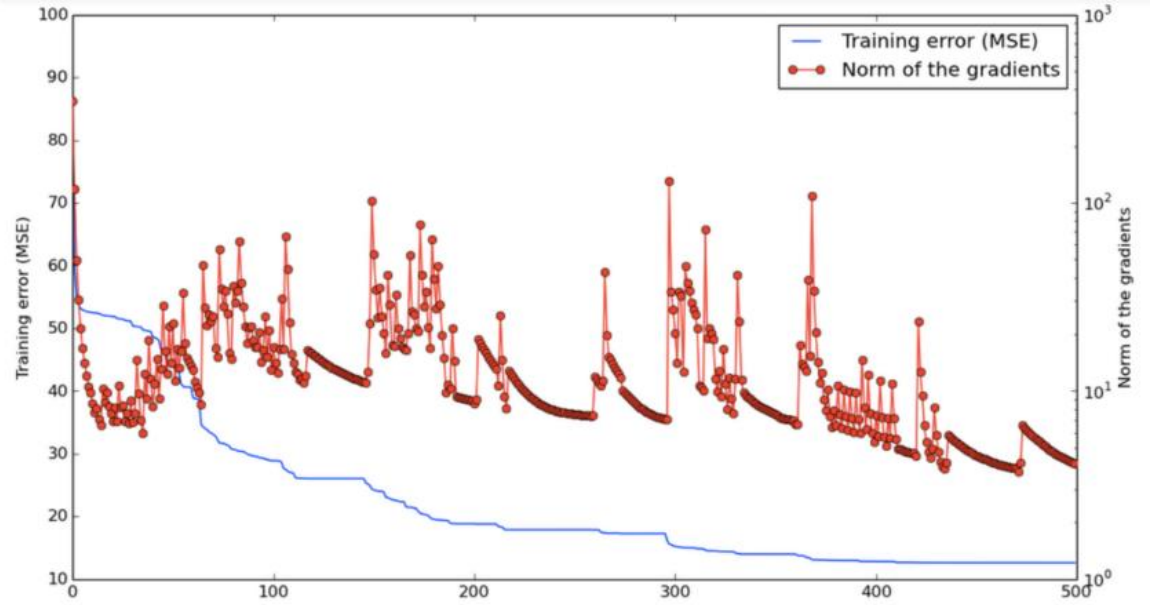
  The low-index critical points of large models concentrate in a band just above the global minimum



Choromanska et al & LeCun 2014, 'The Loss Surface of Multilayer Nets'

Saddle Points

- Escaping Saddle Point
  - Oscillating between two behaviors:
    - Slowly approaching a saddle point
    - Escaping it

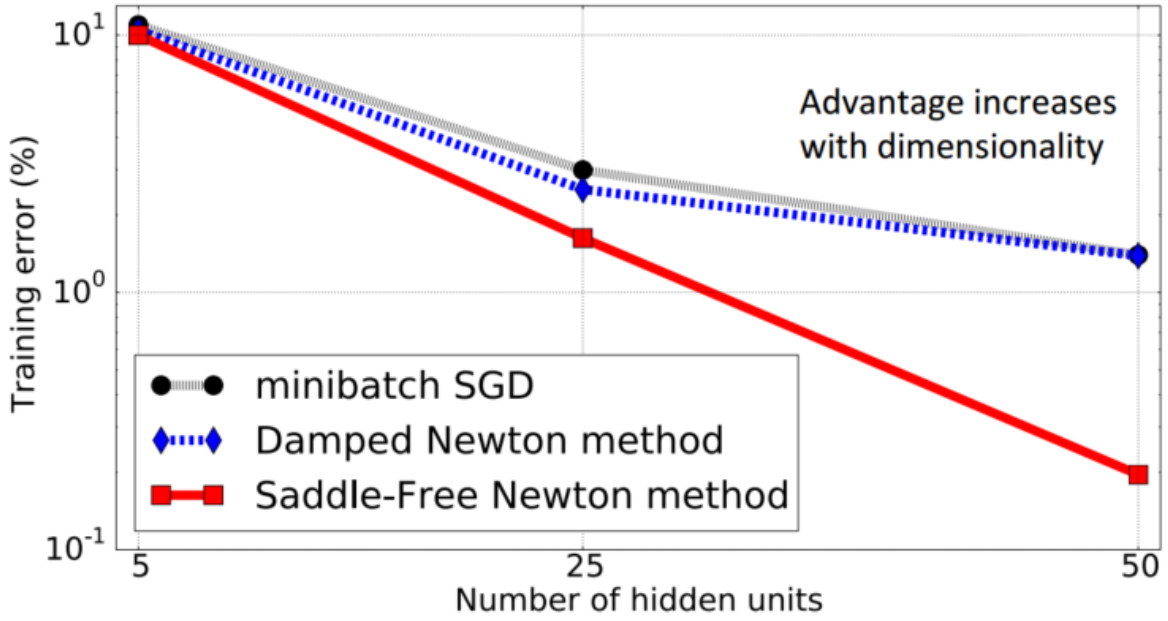


Issue: underfitting due to combinatorially many poor *effective* local minima, most likely to be flat saddle points

where the optimizer gets stuck

Saddle Points

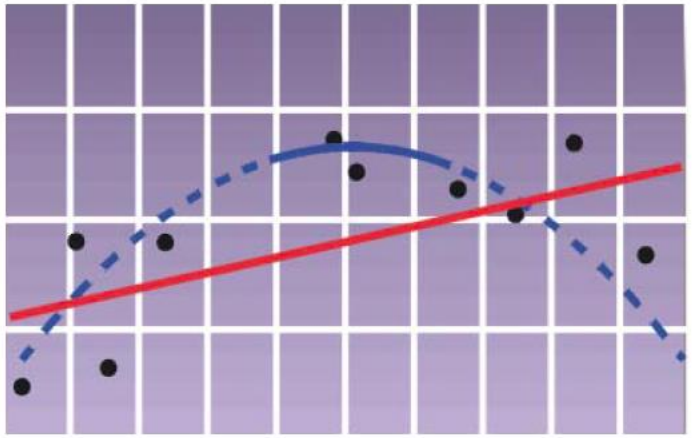
- Saddle-Free Optimization
  - Saddle points are ATTRACTIVE for Newton's method
  - Replace eigenvalues λ of Hessian by |λ|
  - Justified as a particular trust region method



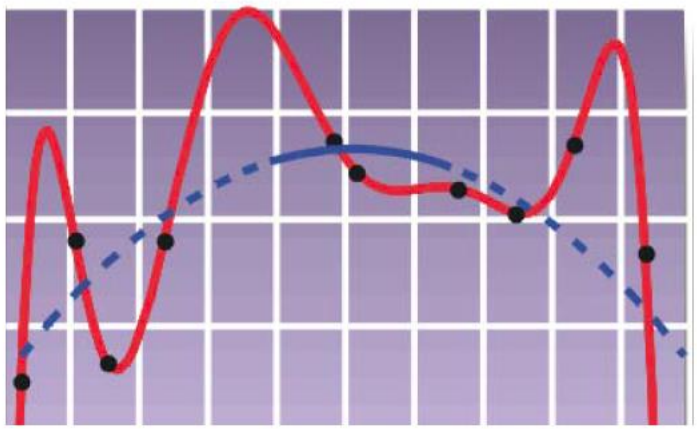
Regularization

The Problem of Overfitting

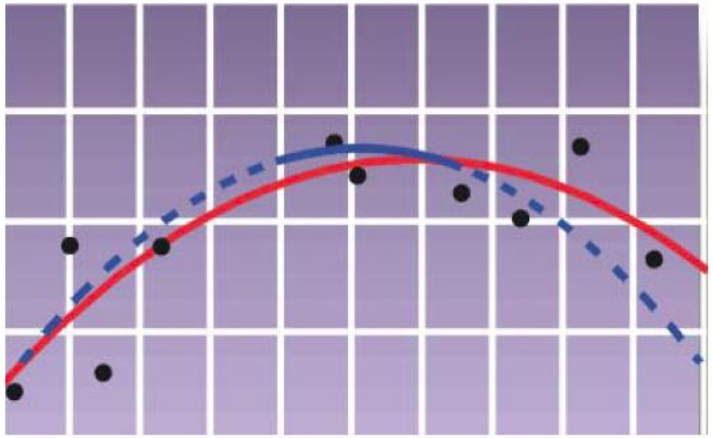
- Finding Optimal Model Class



capacity too low
⇨under-fitting



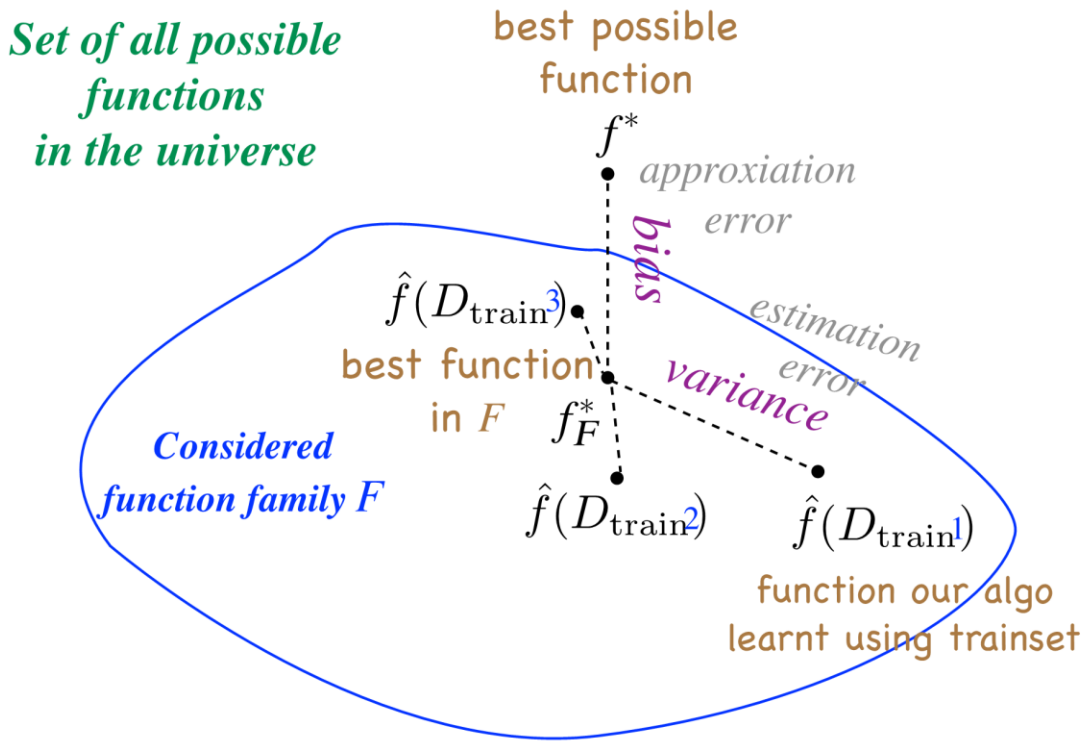
capacity too high
⇨over-fitting



optimal capacity
⇨good generalisation

Regularization and Model Selection

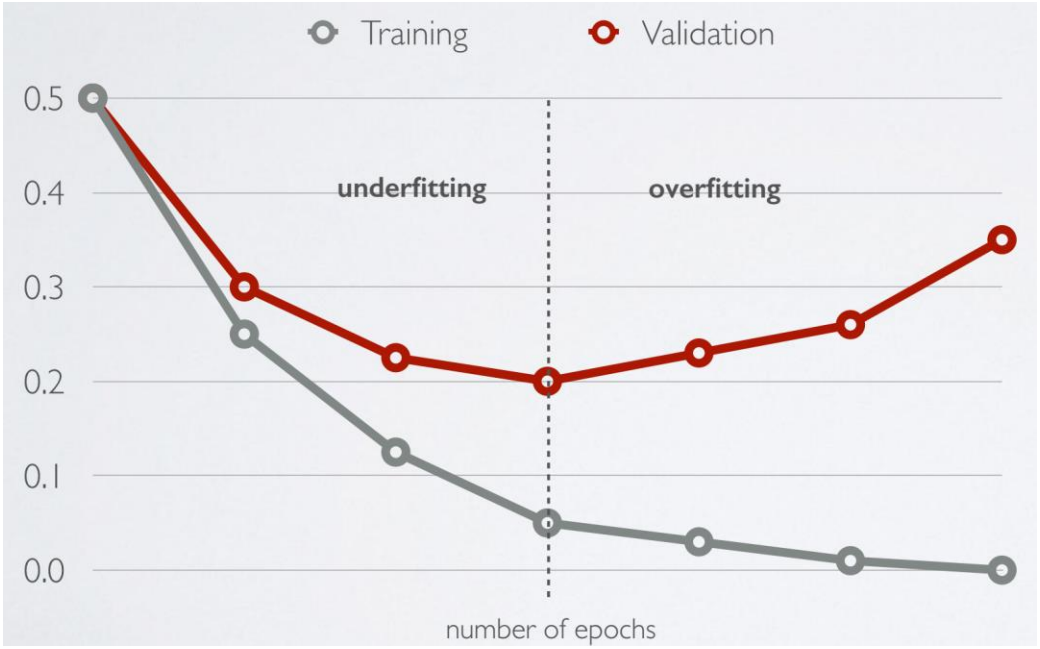
- Bias vs Variance



- Choosing richer F: capacity ↑
  ➪ bias ↓ *but* variance ↑.
- Choosing smaller F : capacity ↓
  ➪ variance ↓ *but* bias ↑.
- Optimal compromise... will depend on number of examples ***n***
- Bigger ***n*** ➪ variance ↓
  So we can afford to increase capacity (to lower the bias)
  ➪ can use more expressive models
- The best regularizer is more data!

Early Stopping

- Stop training when validation error increases
- Why early stopping works?
  - When started with very small weights, finish learning before they get big
  - Then the whole network is in the linear regime(using sigmoid) and the capacity of the
    net is suppressed



Weight Penalty

- Limiting the size of the weights
  - The standard L2 weight penalty involves adding an extra term to the cost function that penalizes the squared weights.

$$C = E + \frac{\lambda}{2} \sum_i w_i^2$$

$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$

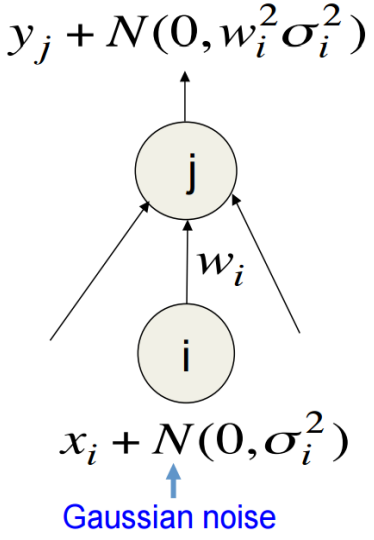
  - So putting gradient, momentum, weight decay altogether,

$$v_{i+1} := 0.9 \cdot v_i - 0.0005 \cdot \epsilon \cdot w_i - \epsilon \cdot \left\langle \frac{\partial L}{\partial w}\Big|_{w_i} \right\rangle_{D_i}$$

$$w_{i+1} := w_i + v_{i+1}$$

Weight Penalty via Noising Input

- L2 Weight-decay via Gaussian Noised Inputs



output on one case $\longrightarrow$ $y^{noisy} = \sum_i w_i x_i + \sum_i w_i \varepsilon_i$ *where* ε_i *is sampled from* $N(0, \sigma_i^2)$

$$E\left[(y^{noisy} - t)^2\right] = E\left[\left(y + \sum_i w_i \varepsilon_i - t\right)^2\right] = E\left[\left((y-t) + \sum_i w_i \varepsilon_i\right)^2\right]$$

$$= (y-t)^2 + E\left[2(y-t)\sum_i w_i \varepsilon_i\right] + E\left[\left(\sum_i w_i \varepsilon_i\right)^2\right]$$

$$= (y-t)^2 + E\left[\sum_i w_i^2 \varepsilon_i^2\right]$$

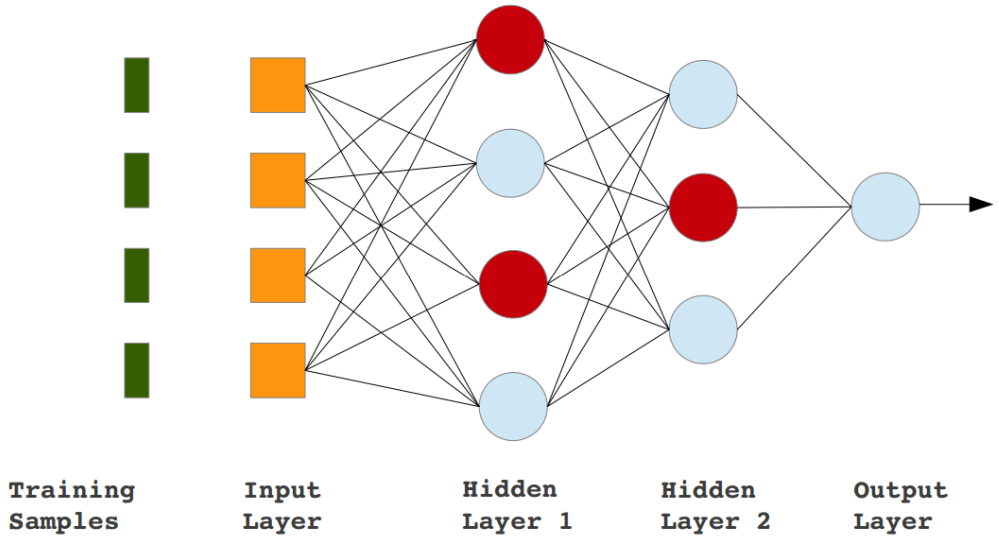
because ε_i *is independent of* ε_j *and* ε_i *is independent of* $(y-t)$

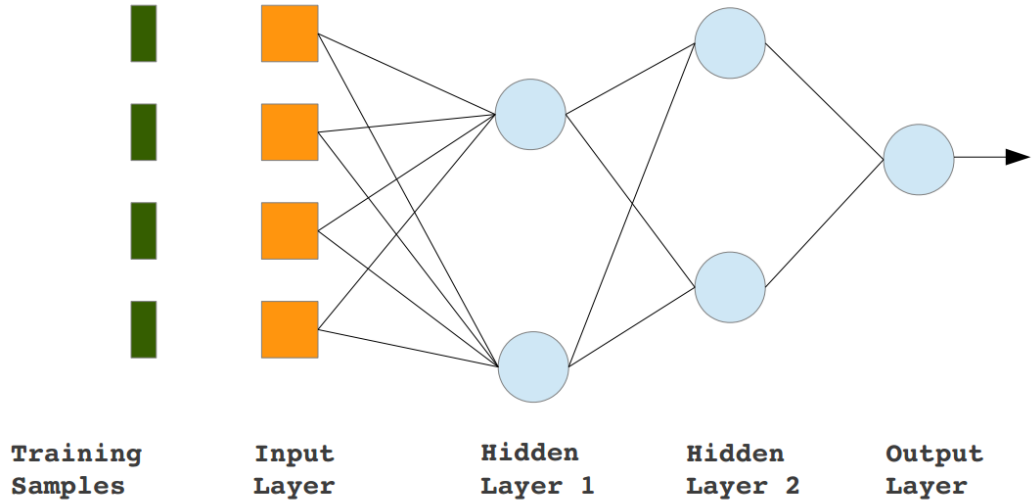
$$= (y-t)^2 + \sum_i w_i^2 \sigma_i^2$$

So σ_i^2 is equivalent to an L2 penalty

Dropout

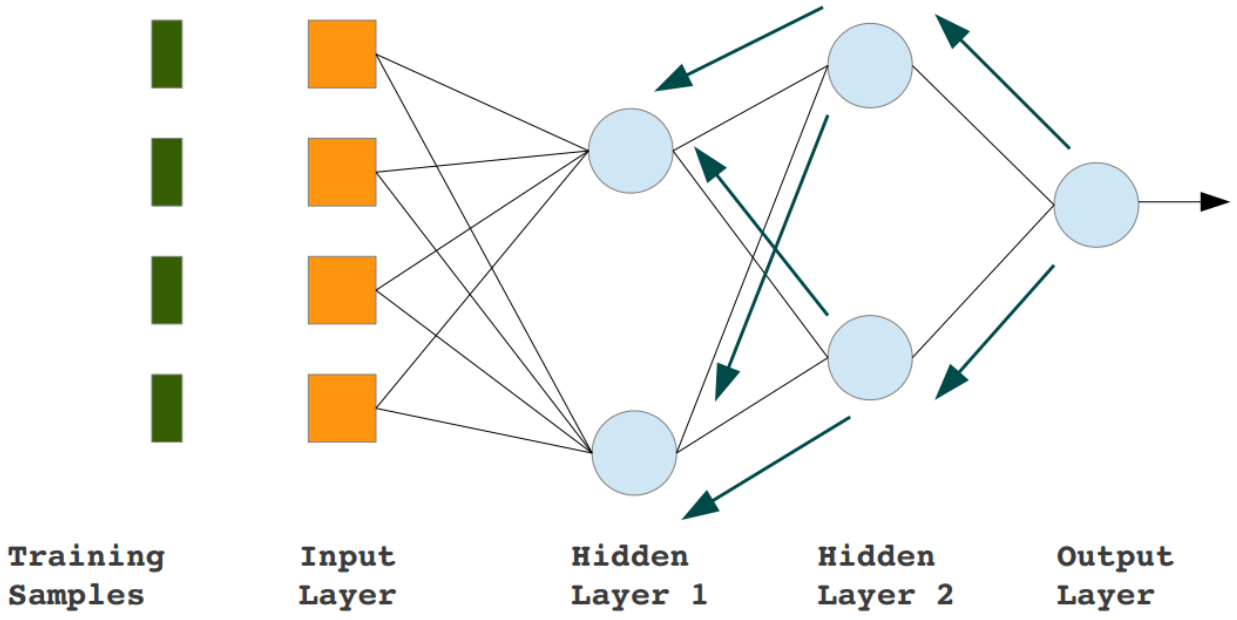
- Training Dropout Network
  - Forward Pass





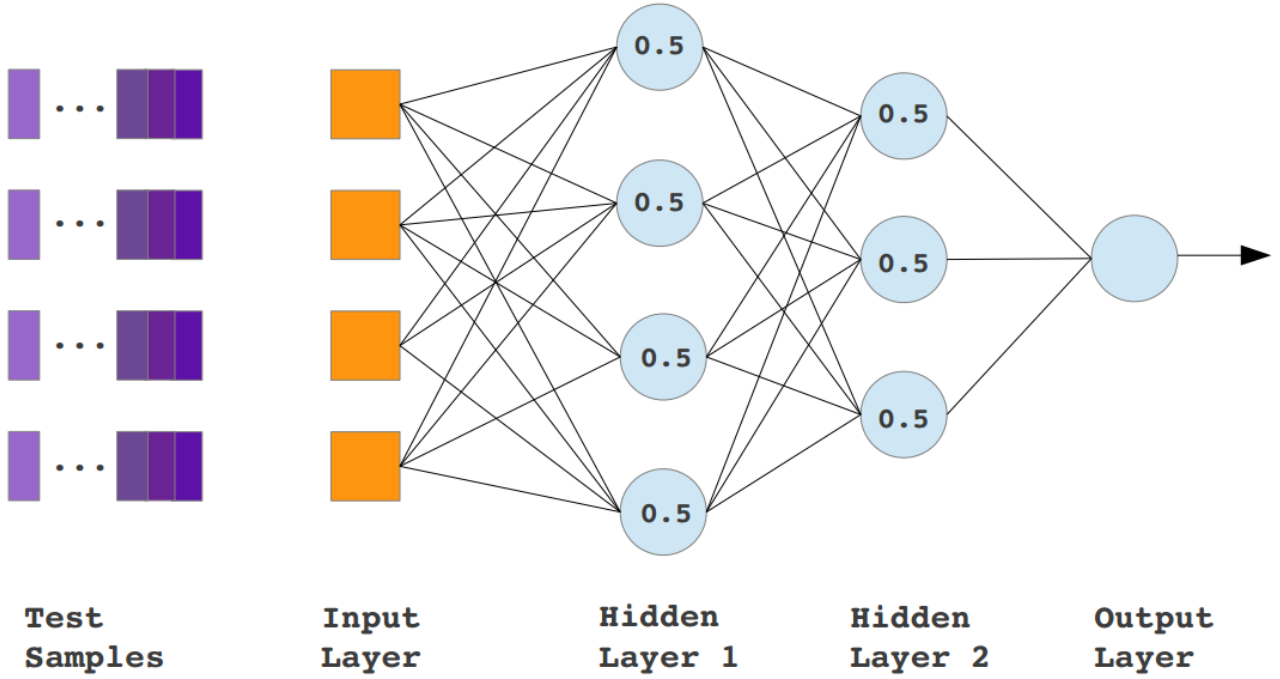
Dropout

- Training Dropout Network
  - Backward Pass



Dropout

- Testing Dropout Network



Lab

Lab

- Optimization and Regularization
  - https://github.com/KyuhwanJung/tensorflow_tutorial/blob/master/Optimization%20and%20Regularization.ipynb

Class 4
Restricted Boltzmann Machine and Deep Belief Network

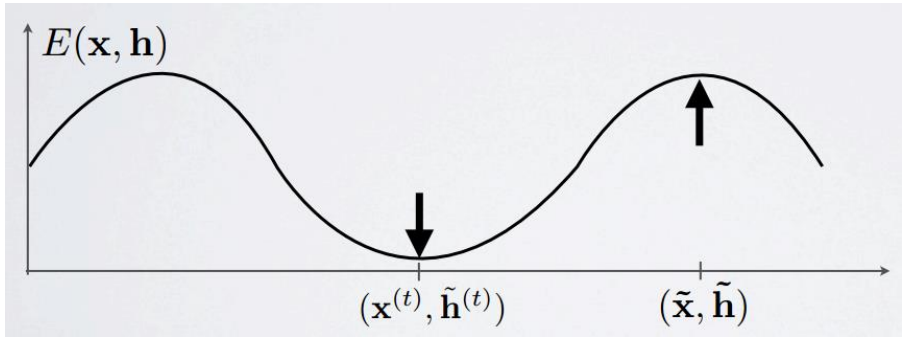
Energy-based Models

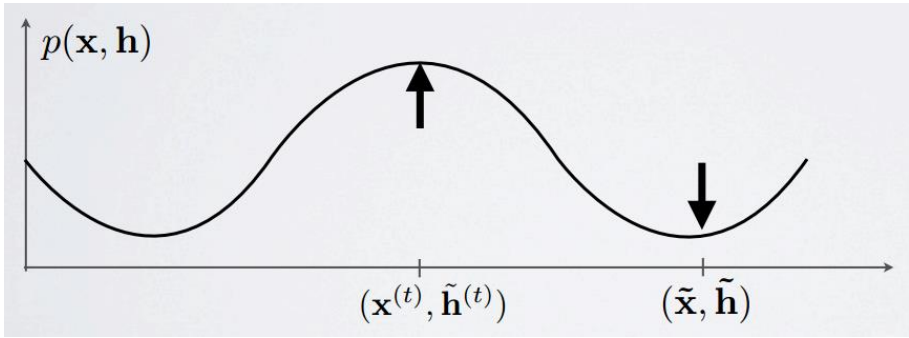
Energy-based Model

- Energy to Probability
  - EBM associate a scalar energy to each configuration of the variable of interest as :

$$P(\mathbf{x}) = \frac{e^{-\text{Energy}(\mathbf{x})}}{Z}, \qquad Z = \sum_{\mathbf{x}} e^{-\text{Energy}(\mathbf{x})}$$

  - Learning corresponds to modifying the energy function to have low energy at the desirable configuration
  - Among probability distribution, exponential family can benefit from inference and learning procedure.





MoE vs PoE

- Mixture of Experts

$$p(\boldsymbol{x}|\boldsymbol{\theta}) = \sum_{m=1}^{M} p(\boldsymbol{x}, \omega_m|\boldsymbol{\theta}_m) = \sum_{m=1}^{M} c_m p(\boldsymbol{x}|\omega_m, \boldsymbol{\theta}_m)$$

  - Pros
    - Easy to train : EM algorithm
  - Cons
    - MoE is less sharper than individual experts
    - Experts partitions the space : one region per expert

- Product of Experts

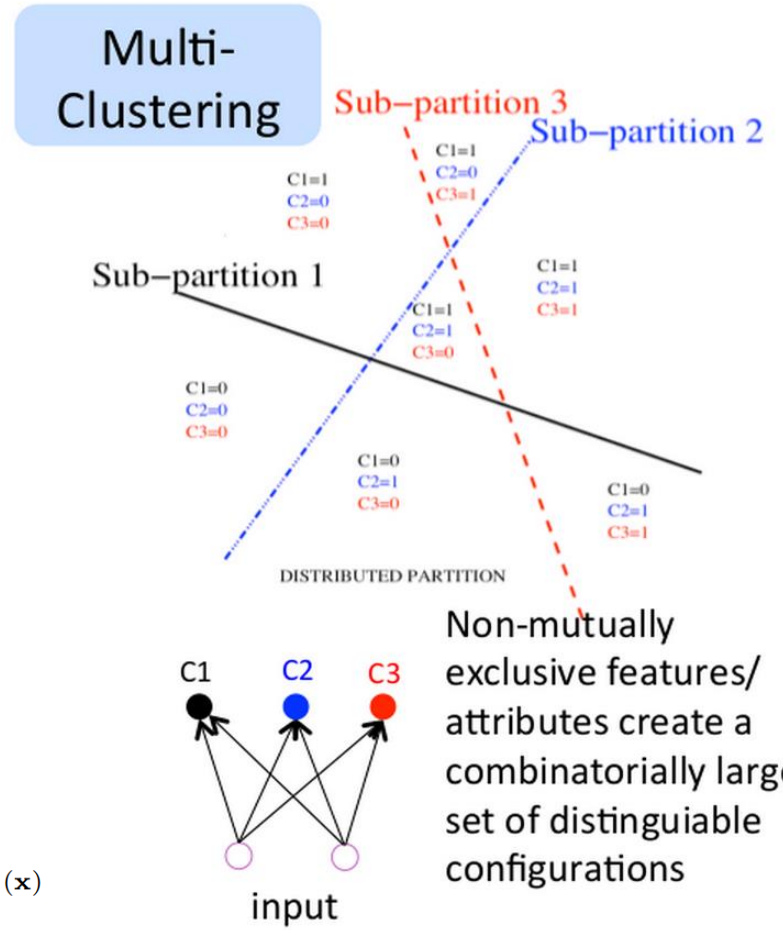
$$\begin{aligned} p(\boldsymbol{x}|\boldsymbol{\theta}) &= \frac{1}{Z} \prod_{m=1}^{M} p(\boldsymbol{x}|\boldsymbol{\theta}_m)^{\lambda_m} \\ &= \frac{1}{Z} \exp\left(\sum_{m=1}^{M} \lambda_m \log(p(\boldsymbol{x}|\boldsymbol{\theta}_m)) \right) \end{aligned}$$

  - Pros
    - Distributed representation
    - PoE is sharper than individual experts
  - Cons
    - Hard to train : Partition Function

In PoE formulation,
Energy of EBM is :

$$\text{Energy}(\mathbf{x}) = \sum_i f_i(\mathbf{x}),$$

$$P(\mathbf{x}) \propto \prod_i P_i(\mathbf{x}) \propto \prod_i e^{-f_i(\mathbf{x})}$$



Introducing Hidden Variable

- Adding non-observed variable as the hidden variable **h**, we get

$$P(\mathbf{x}, \mathbf{h}) = \frac{e^{-\text{Energy}(\mathbf{x},\mathbf{h})}}{Z} \qquad P(\mathbf{x}) = \sum_{\mathbf{h}} \frac{e^{-\text{Energy}(\mathbf{x},\mathbf{h})}}{Z}.$$

- To map this function as the original form, we introduce *Free Energy* as

$$P(\mathbf{x}) = \frac{e^{-\text{FreeEnergy}(\mathbf{x})}}{Z}, \qquad Z = \sum_{\mathbf{x}} e^{-\text{FreeEnergy}(\mathbf{x})}$$

$$\text{FreeEnergy}(\mathbf{x}) = -\log \sum_{\mathbf{h}} e^{-\text{Energy}(\mathbf{x},\mathbf{h})}.$$

Likelihood for Training

- The log-likelihood for gradient-based training with parameter θ is :

$$\begin{aligned}\frac{\partial \log P(\mathbf{x})}{\partial \theta} &= -\frac{\partial \text{FreeEnergy}(\mathbf{x})}{\partial \theta} + \frac{1}{Z}\sum_{\tilde{\mathbf{x}}} e^{-\text{FreeEnergy}(\tilde{\mathbf{x}})}\frac{\partial \text{FreeEnergy}(\tilde{\mathbf{x}})}{\partial \theta} \\ &= -\frac{\partial \text{FreeEnergy}(\mathbf{x})}{\partial \theta} + \sum_{\tilde{\mathbf{x}}} P(\tilde{\mathbf{x}})\,\frac{\partial \text{FreeEnergy}(\tilde{\mathbf{x}})}{\partial \theta}.\end{aligned} \quad \text{- (1)}$$

- Therefore, the average log-likelihood gradient over the training set is :

$$E_{\hat{P}}\left[\frac{\partial \log P(\mathbf{x})}{\partial \theta}\right] = -E_{\hat{P}}\left[\frac{\partial \text{FreeEnergy}(\mathbf{x})}{\partial \theta}\right] + E_{P}\left[\frac{\partial \text{FreeEnergy}(\mathbf{x})}{\partial \theta}\right]$$

  - Here, $\hat{P}$ is empirical distribution a P is the model's distribution.
  - We need to sample fror P to calculate the gradient.

Derivation of (1)

$$\log P(\mathrm{x}) = \log(e^{-FreeEnergy(\mathrm{x})}) - \log(Z) = -FreeEngery(\mathrm{x}) - \log(Z)$$

$$\begin{aligned}
\frac{\partial \log P(\mathrm{x})}{\partial \theta} &= -\frac{\partial FreeEnergy(\mathrm{x})}{\partial \theta} - \frac{1}{Z} \cdot \frac{\partial (\sum_{\tilde{\mathbf{x}}} e^{-FreeEnergy(\tilde{\mathbf{x}})})}{\partial \theta} \\
&= -\frac{\partial FreeEnergy(\mathrm{x})}{\partial \theta} - \frac{1}{Z} \cdot \sum_{\tilde{\mathbf{x}}} \frac{\partial (e^{-FreeEnergy(\tilde{\mathbf{x}})})}{\partial \theta} \\
&= -\frac{\partial FreeEnergy(\mathrm{x})}{\partial \theta} - \frac{1}{Z} \cdot \sum_{\tilde{\mathbf{x}}} e^{-FreeEnergy(\tilde{\mathbf{x}})} \frac{\partial (-FreeEnergy(\tilde{\mathbf{x}}))}{\partial \theta} \\
&= -\frac{\partial FreeEnergy(\mathrm{x})}{\partial \theta} + \frac{1}{Z} \cdot \sum_{\tilde{\mathbf{x}}} e^{-FreeEnergy(\tilde{\mathbf{x}})} \frac{\partial FreeEnergy(\tilde{\mathbf{x}})}{\partial \theta}
\end{aligned}$$

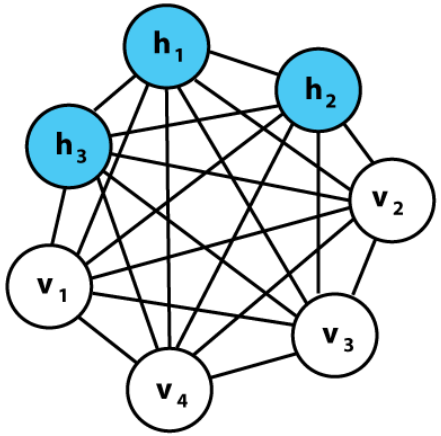
Boltzmann Machine

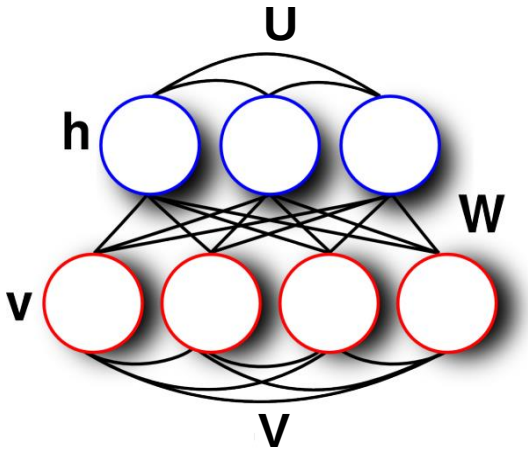
Boltzmann Machine

- BM is a Type of EBM with Hidden Variables with Energy Function

$$\text{Energy}(\mathbf{x}, \mathbf{h}) = -\mathbf{b}'\mathbf{x} - \mathbf{c}'\mathbf{h} - \mathbf{h}'W\mathbf{x} - \mathbf{x}'U\mathbf{x} - \mathbf{h}'V\mathbf{h}.$$

  - Two types of parameters
    - Offsets $\mathbf{b}_i$ $\mathbf{c}_i$
    - Weights :W_{ij}, U_{ij} V_{ij}, (*U, V* are symmetric with zero diagonals)





Gradient of Log-likelihood of BM

- From the definition

$$P(\mathbf{x}) = \sum_{\mathbf{h}} \frac{e^{-\text{Energy}(\mathbf{x},\mathbf{h})}}{Z}$$

Gradient of log-likelihood is

$$\begin{aligned}
\frac{\partial \log P(\mathbf{x})}{\partial \theta} &= \frac{\partial \log \sum_{\mathbf{h}} e^{-\text{Energy}(\mathbf{x},\mathbf{h})}}{\partial \theta} - \frac{\partial \log \sum_{\tilde{\mathbf{x}},\mathbf{h}} e^{-\text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}}{\partial \theta} \\
&= -\frac{1}{\sum_{\mathbf{h}} e^{-\text{Energy}(\mathbf{x},\mathbf{h})}} \sum_{\mathbf{h}} e^{-\text{Energy}(\mathbf{x},\mathbf{h})} \frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \theta} \\
&\quad + \frac{1}{\sum_{\tilde{\mathbf{x}},\mathbf{h}} e^{-\text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}} \sum_{\tilde{\mathbf{x}},\mathbf{h}} e^{-\text{Energy}(\tilde{\mathbf{x}},\mathbf{h})} \frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial \theta} \\
&= -\sum_{\mathbf{h}} P(\mathbf{h}|\mathbf{x}) \frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \theta} + \sum_{\tilde{\mathbf{x}},\mathbf{h}} P(\tilde{\mathbf{x}},\mathbf{h}) \frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial \theta}.
\end{aligned}$$

Computing the Gradient

- Then, How Can We Compute the Gradient?

$$\frac{\partial \log P(\mathbf{x})}{\partial \theta} = -\sum_{\mathbf{h}} P(\mathbf{h}|\mathbf{x}) \frac{\partial \text{Energy}(\mathbf{x}, \mathbf{h})}{\partial \theta} + \sum_{\tilde{\mathbf{x}}, \mathbf{h}} P(\tilde{\mathbf{x}}, \mathbf{h}) \frac{\partial \text{Energy}(\tilde{\mathbf{x}}, \mathbf{h})}{\partial \theta}$$

  - The computation is possible only when we can sample from $P(\mathbf{h}|\mathbf{x})$ and $P(\mathbf{x}, \mathbf{h})$.
  - In general, this is intractable and we use MCMC such as Gibbs sampling instead.

- Gibbs Sampling
  - For a joint random variable $S = (S_1 \ldots S_N)$, perform *N* sampling sub-steps of form

$$S_i \sim P(S_i | S_{-i} = \mathbf{s}_{-i})$$

    where S_{-i} contains *N*-1random variables excluding S_i.
  - When repeated infinitely many times, the sample distribution converges to $P(S)$

Gibbs Sampling in BM

- Deriving Sampling Distribution
  - Let $\mathbf{s} = (\mathbf{x}, \mathbf{h})$ denote all the units in the BM and $\mathbf{s}_{-i}$ the set of values associated with all values except the *i*-th unit.
  - By putting all the offset parameter in a vector $\mathbf{d}$ and weight parameter in symmetric weight matrix A , we can rewrite the energy function of BM as :

$$\text{Energy}(\mathbf{s}) = -\mathbf{d}'\mathbf{s} - \mathbf{s}'A\mathbf{s}.$$

  - Then $\mathbf{s}_i \in \{0, 1\}$ can be obtained as :

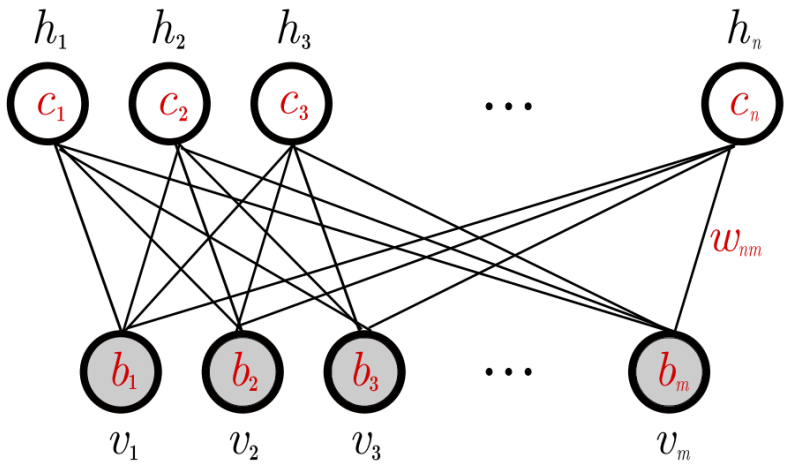
$$\begin{aligned} P(\mathbf{s}_i = 1|\mathbf{s}_{-i}) &= \frac{\exp(\mathbf{d}_i + \mathbf{d}'_{-i}\mathbf{s}_{-i} + 2\mathbf{a}'_{-i}\mathbf{s}_{-i} + \mathbf{s}'_{-i}A_{-i}\mathbf{s}_{-i})}{\exp(\mathbf{d}_i + \mathbf{d}'_{-i}\mathbf{s}_{-i} + 2\mathbf{a}'_{-i}\mathbf{s}_{-i} + \mathbf{s}'_{-i}A_{-i}\mathbf{s}_{-i}) + \exp(\mathbf{d}'_{-i}\mathbf{s}_{-i} + \mathbf{s}'_{-i}A_{-i}\mathbf{s}_{-i})} \\ &= \frac{\exp(\mathbf{d}_i + 2\mathbf{a}'_{-i}\mathbf{s}_{-i})}{\exp(\mathbf{d}_i + 2\mathbf{a}'_{-i}\mathbf{s}_{-i}) + 1} = \frac{1}{1 + \exp(-\mathbf{d}_i - 2\mathbf{a}'_{-i}\mathbf{s}_{-i})} \\ &= \text{sigm}(\mathbf{d}_i + 2\mathbf{a}'_{-i}\mathbf{s}_{-i}) \end{aligned}$$

Restricted Boltzmann Machine

Restricted Boltzmann Machine

- Restricted Connections in BM
  - When we set $U = 0$ and $V = 0$, i.e. restricting the interactions to be possible only between a hidden unit and a visible unit, we get **Restricted Boltzmann Machine.**
  - Then the energy function is reduced to

$$\text{Energy}(\mathbf{x}, \mathbf{h}) = -\mathbf{b}'\mathbf{x} - \mathbf{c}'\mathbf{h} - \mathbf{h}'W\mathbf{x}$$



Conditional Probability of RBM

- Conditional Independence of Units in the Same Layer

$$P(\mathbf{h}\,|\,\mathbf{x}) = \frac{P(\mathrm{x},\mathrm{h})}{P(\mathrm{x})} = \frac{P(\mathrm{x},\mathrm{h})}{\sum_{\tilde{\mathrm{h}}} P(\mathrm{x},\tilde{\mathrm{h}})} = \frac{e^{-Energy(\mathrm{x},\mathrm{h})}/Z}{\sum_{\tilde{\mathrm{h}}} e^{-Energy(\mathrm{x},\tilde{\mathrm{h}})}/Z}$$

$$= \frac{\exp(\mathrm{b'x} + \mathrm{c'h} + \mathrm{h'}W\,\mathrm{x})}{\sum_{\tilde{\mathrm{h}}} \exp(\mathrm{b'x} + \mathrm{c'}\tilde{\mathrm{h}} + \tilde{\mathrm{h}}'W\,\mathrm{x})}$$

$$= \frac{\prod_i \exp(\mathrm{c}_i\mathrm{h}_i + \mathrm{h}_i W_i\,\mathrm{x})}{\sum_{\tilde{\mathrm{h}}} \prod_i \exp(\mathrm{c}_i\tilde{\mathrm{h}}_i + \tilde{\mathrm{h}}_i W_i\,\mathrm{x})}$$

$$= \frac{\prod_i \exp(\mathrm{c}_i\mathrm{h}_i + \mathrm{h}_i W_i\,\mathrm{x})}{\prod_i \sum_{\tilde{\mathrm{h}}} \exp(\mathrm{c}_i\tilde{\mathrm{h}}_i + \tilde{\mathrm{h}}_i W_i\,\mathrm{x})}$$

$$= \prod_i \frac{\exp(\mathrm{h}_i(\mathrm{c}_i + W_i\,\mathrm{x})}{\sum_{\tilde{\mathrm{h}}} \exp(\tilde{\mathrm{h}}_i(\mathrm{c}_i + W_i\,\mathrm{x})}$$

$$= \prod_i P(\mathrm{h}_i\,|\,\mathrm{x})$$

When $\mathbf{h}_i \in \{0, 1\}$, we get

$$P(\mathbf{h}_i = 1|\mathbf{x}) = \frac{e^{\mathbf{c}_i + W_i\mathbf{x}}}{1 + e^{\mathbf{c}_i + W_i\mathbf{x}}} = \mathrm{sigm}(\mathbf{c}_i + W_i\mathbf{x}).$$

By symmetry of **x** and **h** in the energy function, we get,

$$P(\mathbf{x}|\mathbf{h}) = \prod_i P(\mathbf{x}_i|\mathbf{h})$$

and for binary **x**,

$$P(\mathbf{x}_j = 1|\mathbf{h}) = \mathrm{sigm}(\mathbf{b}_j + W'_{\cdot j}\mathbf{h})$$

Gibbs Sampling in RBM

- Remark

$$\frac{\partial \log P(\mathbf{x})}{\partial \theta} = -\sum_{\mathbf{h}} P(\mathbf{h}|\mathbf{x}) \frac{\partial \text{Energy}(\mathbf{x}, \mathbf{h})}{\partial \theta} + \sum_{\tilde{\mathbf{x}}, \mathbf{h}} P(\tilde{\mathbf{x}}, \mathbf{h}) \frac{\partial \text{Energy}(\tilde{\mathbf{x}}, \mathbf{h})}{\partial \theta}$$

$$\frac{\partial(-\log P(\mathbf{x}^{(t)}))}{\partial \theta} = \underbrace{< \frac{\partial Energy(\mathbf{x}^{(t)}, \mathbf{h})}{\partial \theta} >_{data}}_{\text{Positive Phase}} - \underbrace{< \frac{\partial Energy(\mathbf{x}, \mathbf{h})}{\partial \theta} >_{model}}_{\text{Negative Phase}}$$

- Efficient Sampling via Factorization
  - No need to sample in the positive phase : computed analytically.
  - Efficient Gibbs sampling is possible as

$$\begin{aligned} \mathbf{x}_1 &\sim \hat{P}(\mathbf{x}) \\ \mathbf{h}_1 &\sim P(\mathbf{h}|\mathbf{x}_1) \\ \mathbf{x}_2 &\sim P(\mathbf{x}|\mathbf{h}_1) \\ \mathbf{h}_2 &\sim P(\mathbf{h}|\mathbf{x}_2) \\ &\cdots \\ \mathbf{x}_{k+1} &\sim P(\mathbf{x}|\mathbf{h}_k). \end{aligned}$$

Contrastive Divergence

Contrastive Divergence

- Justification
  - First Approximation : Replace average over all possible input by sing sample in the negative phase.
    - We use SGD and it's already some sort of averaging.
    - Additional variance introduced by not using complete averaging might be partially cancelled out or smaller compared to the variance by SGD.
  - Second Approximation : Run MCMC only k-steps starting from the training sample.

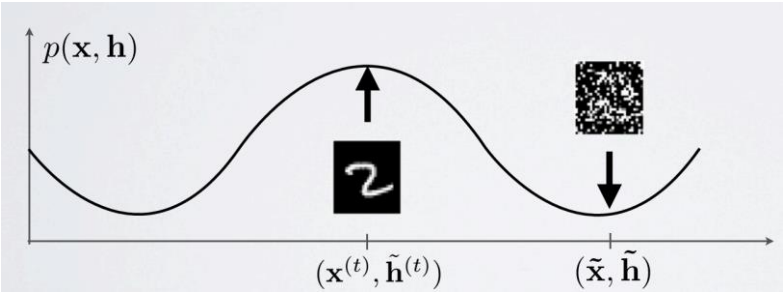
$$\Delta\theta \propto \frac{\partial \mathrm{FreeEnergy}(\mathbf{x})}{\partial\theta} - \frac{\partial \mathrm{FreeEnergy}(\tilde{\mathbf{x}})}{\partial\theta}$$

where $\tilde{\mathbf{x}} = \mathbf{x}_{k+1}$ is the last sample from our Gibbs sampling after k steps.

  - We start from a training example to make the model become better at capturing the structure in the training data.
  - P and $\hat{P}$ becomes similar through the training process and we hope them to be identical(then sampling from $P \approx \hat{P}$ will converge in just one step)

Contrastive Divergence

- CD-k
  - Computing the 'contrast' between the statistics of 'real' input and 'model' input.
  - Even k=1 works pretty well(with some bias)
  - We can interpret this as the stochastic reconstruction $\tilde{\mathbf{x}} = \mathbf{x}_{k+1}$ has distribution in some sense centered around $\mathbf{x}_1$ and becomes more spread out around it as k increases.
  - CD-k update will decrease the free energy of the training point $\mathbf{x}_1$ and increase the free energy of $\tilde{\mathbf{x}}$.
  - Initialized with $\mathbf{x}_1$, we go down the energy land scape from $\mathbf{x}_1$ to $\mathbf{x}_2$.



Learning Rule for RBM

- With Negative Log-likelihood

$$\frac{\partial(-\log P(\mathbf{x}^{(t)}))}{\partial\theta} = < \frac{\partial Energy(\mathbf{x}^{(t)},\mathbf{h})}{\partial\theta} >_{data} - < \frac{\partial Energy(\mathbf{x},\mathbf{h})}{\partial\theta} >_{model}$$

and energy function

$$Energy(\mathbf{x},\mathbf{h}) = -\mathbf{b}'\mathbf{x} - \mathbf{c}'\mathbf{h} - \mathbf{h}'\mathbf{W}\mathbf{x}$$

we get

$$\frac{\partial Energy(\mathbf{x},\mathbf{h})}{\partial W_{ij}} = -x_i h_j \qquad \frac{\partial Energy(\mathbf{x},\mathbf{h})}{\partial b_i} = -x_i \qquad \frac{\partial Energy(\mathbf{x},\mathbf{h})}{\partial c_j} = -h_j$$

and

$$\Delta W_{ij} = \rho(< x_i h_j >_{data} - < x_i h_j >_{model})$$

$$\Delta b_i = \rho(< x_i >_{data} - < x_i >_{model})$$

$$\Delta c_j = \rho(< h_j >_{data} - < h_j >_{model})$$

Learning Rule for RBM using CD-1

- Using CD-1,

$$\Delta W_{ij} = -\rho(x_i \cdot g(x_i) - \tilde{x}_i \cdot g(\tilde{x}_i))$$
$$\Delta b_i = -\rho(x_i - \tilde{x}_i)$$
$$\Delta c_i = -\rho(h_i - g(\tilde{x}_i))$$

where $g(\tilde{x}_i) = p(\tilde{h}_i = 1 | \tilde{x}_i)$

and update rule is

$$W_{ij} = W_{ij} + \rho(x_i \cdot g(x_i) - \tilde{x}_i \cdot g(\tilde{x}_i))$$
$$b_i = b_i + \rho(x_i - \tilde{x}_i)$$
$$c_i = c_i + \rho(h_i - g(\tilde{x}_i))$$

RBM Update Algorithm

`RBMupdate`$(\mathbf{x}_1, \epsilon, W, \mathbf{b}, \mathbf{c})$
This is the RBM update procedure for binomial units. It can easily adapted to other types of units.
$\mathbf{x}_1$ is a sample from the training distribution for the RBM
ϵ is a learning rate for the stochastic gradient descent in Contrastive Divergence
W is the RBM weight matrix, of dimension (number of hidden units, number of inputs)
$\mathbf{b}$ is the RBM offset vector for input units
$\mathbf{c}$ is the RBM offset vector for hidden units
Notation: $Q(\mathbf{h}_2. = 1|\mathbf{x}_2)$ is the vector with elements $Q(\mathbf{h}_{2i} = 1|\mathbf{x}_2)$

for all hidden units i **do**
- compute $Q(\mathbf{h}_{1i} = 1|\mathbf{x}_1)$ (for binomial units, $\text{sigm}(\mathbf{c}_i + \sum_j W_{ij}\mathbf{x}_{1j})$)
- sample $\mathbf{h}_{1i} \in \{0, 1\}$ from $Q(\mathbf{h}_{1i}|\mathbf{x}_1)$

end for
for all visible units j **do**
- compute $P(\mathbf{x}_{2j} = 1|\mathbf{h}_1)$ (for binomial units, $\text{sigm}(\mathbf{b}_j + \sum_i W_{ij}\mathbf{h}_{1i})$)
- sample $\mathbf{x}_{2j} \in \{0, 1\}$ from $P(\mathbf{x}_{2j} = 1|\mathbf{h}_1)$

end for
for all hidden units i **do**
- compute $Q(\mathbf{h}_{2i} = 1|\mathbf{x}_2)$ (for binomial units, $\text{sigm}(\mathbf{c}_i + \sum_j W_{ij}\mathbf{x}_{2j})$)

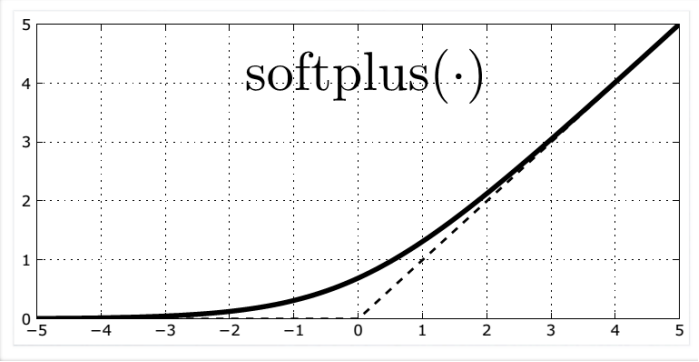
end for
- $W \leftarrow W + \epsilon(\mathbf{h}_1\mathbf{x}_1' - Q(\mathbf{h}_2. = 1|\mathbf{x}_2)\mathbf{x}_2')$
- $\mathbf{b} \leftarrow \mathbf{b} + \epsilon(\mathbf{x}_1 - \mathbf{x}_2)$
- $\mathbf{c} \leftarrow \mathbf{c} + \epsilon(\mathbf{h}_1 - Q(\mathbf{h}_2. = 1|\mathbf{x}_2))$

Revisiting Free-Energy

- What it Means to Maximize Likelihood?

$$
\begin{aligned}
p(\mathbf{x}) &= \sum_{\mathbf{h} \in \{0,1\}^H} p(\mathbf{x}, \mathbf{h}) = \sum_{\mathbf{h} \in \{0,1\}^H} \exp(-E(\mathbf{x}, \mathbf{h}))/Z \\
&= \exp\left(\mathbf{c}^\top \mathbf{x} + \sum_{j=1}^{H} \log(1 + \exp(b_j + \mathbf{W}_j . \mathbf{x})) \right) /Z \\
&= \exp(-F(\mathbf{x}))/Z
\end{aligned}
$$

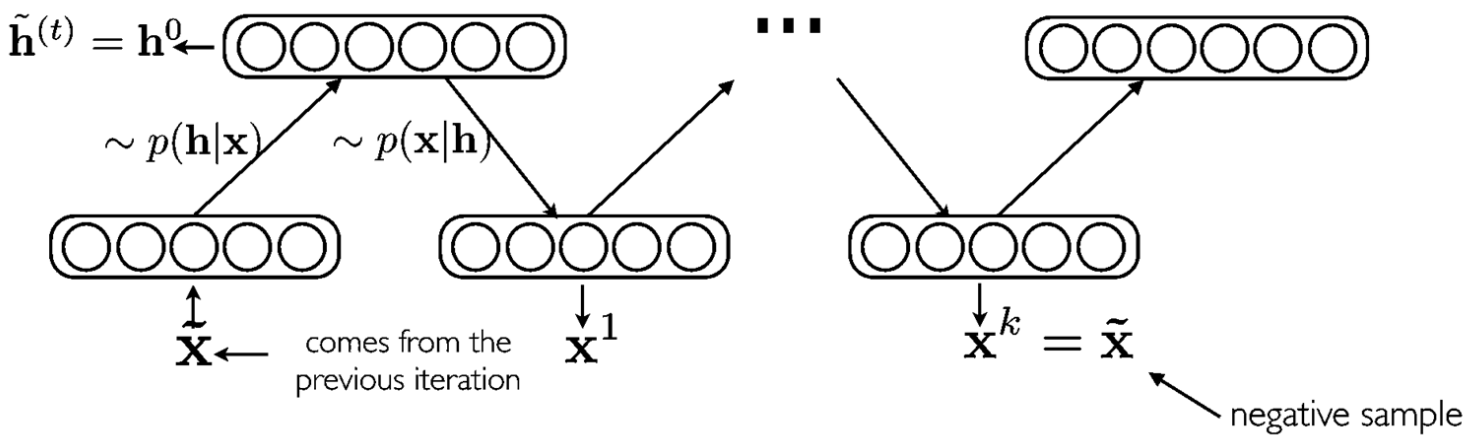
free energy

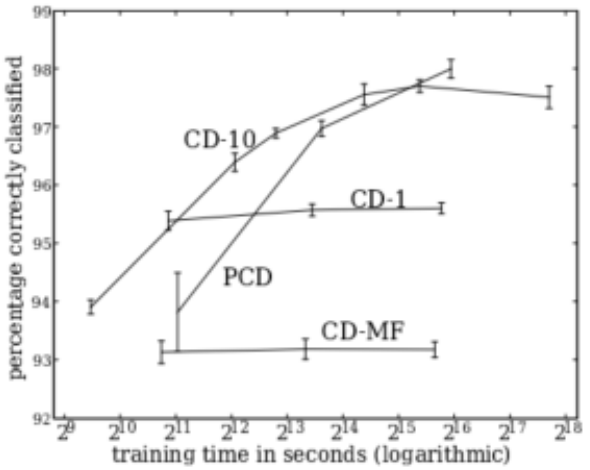


Extensions and Variants

Persistent CD

- Keeping the Negative Phase Samples
  - Instead of starting sampling from training samples, start from the last negative phase sample.
  - This is justified with small parameter change.





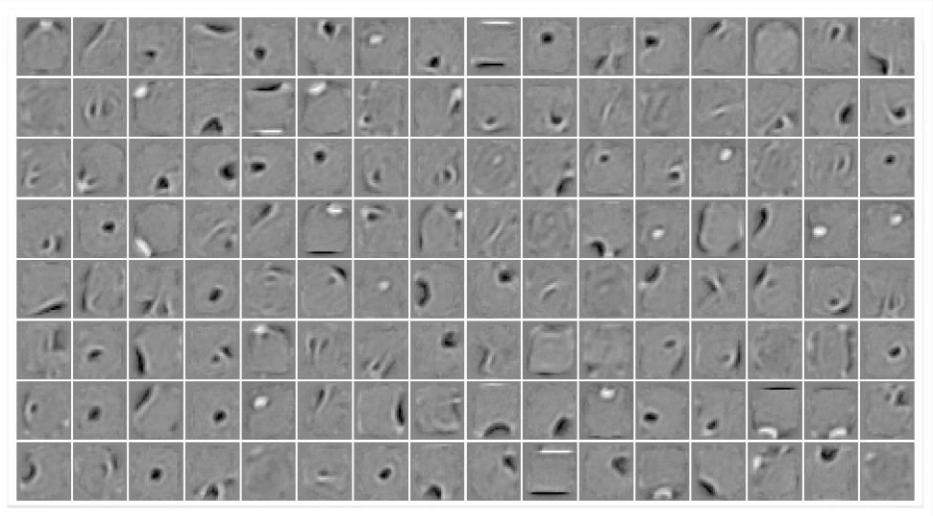
Tielman(2008)

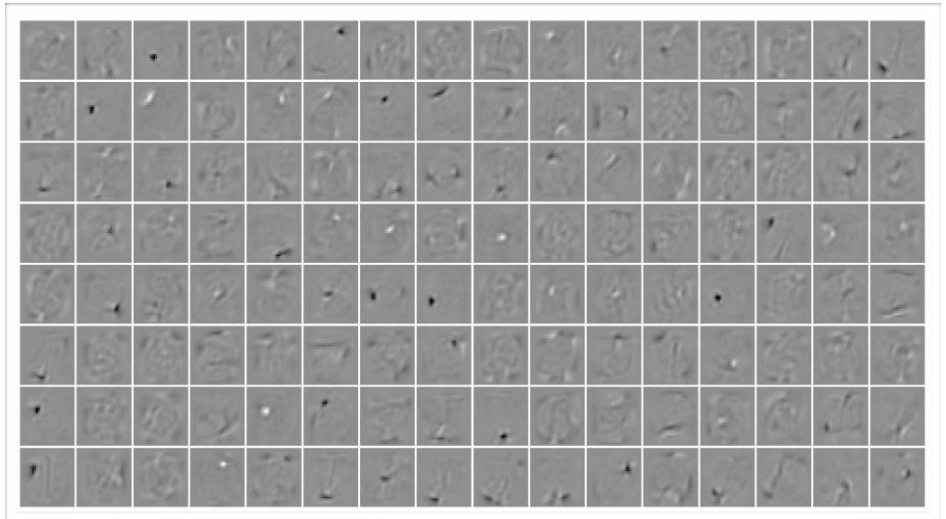
Gaussian-Bernoulli RBM

- For Unbounded Real Inputs,

$$E(\mathbf{x}, \mathbf{h}) = -\mathbf{h}^\top \mathbf{W}\mathbf{x} - \mathbf{c}^\top \mathbf{x} - \mathbf{b}^\top \mathbf{h} + \tfrac{1}{2}\mathbf{x}^\top \mathbf{x}$$

- Now, $p(\mathbf{x}|\mathbf{h})$ is Gaussian distribution with m$\boldsymbol{\mu} = \mathbf{c} + \mathbf{W}^\top \mathbf{h}$ and identity covariance matrix.





Bayesian Networks

Bayesian Machine Learning

- Bayes Rule

$$P(\theta|\mathcal{D}) = \frac{P(\mathcal{D}|\theta)P(\theta)}{P(\mathcal{D})}$$

$P(\mathcal{D}\|\theta)$	likelihood of θ
$P(\theta)$	prior probability of θ
$P(\theta\|\mathcal{D})$	posterior of θ given $\mathcal{D}$

- Sum and Product Rule
  - Sum Rule $P(x) = \sum_y P(x, y)$
  - Product Rule $P(x, y) = P(x)P(y|x)$

- Prediction and Model Selection
  - Prediction

$$P(x|\mathcal{D}, m) = \int P(x|\theta, \mathcal{D}, m)P(\theta|\mathcal{D}, m)d\theta$$

  - Model Selection

$$P(m|\mathcal{D}) = \frac{P(\mathcal{D}|m)P(m)}{P(\mathcal{D})}$$

$$P(\mathcal{D}|m) = \int P(\mathcal{D}|\theta, m)P(\theta|m)\, d\theta$$

Graphical Model and Conditional Independence

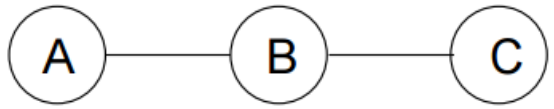
- Graphical Model
  - A graph which expresses conditional dependence structure among random variables.
  - Each node represent a random variable and each edge represent probabilistic dependency between nodes connected to it.

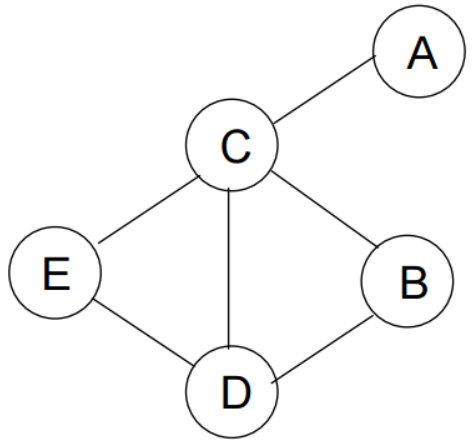
- Conditional Independence
  - Given three variables, A,B,C, we can write their joint probabilit

$$P(A, B, C) = P(A)P(B|A)P(C|B, A)$$

and if C is conditionally independent of A given B, we can write

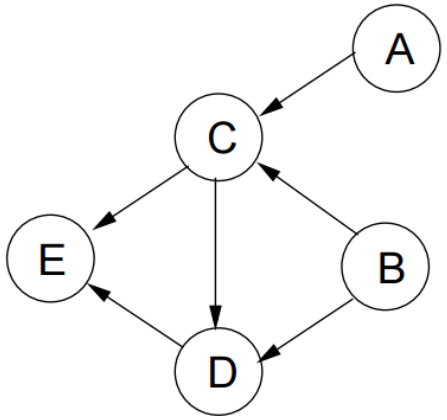
$$P(A, B, C) = P(A)P(B|A)P(C|B)$$

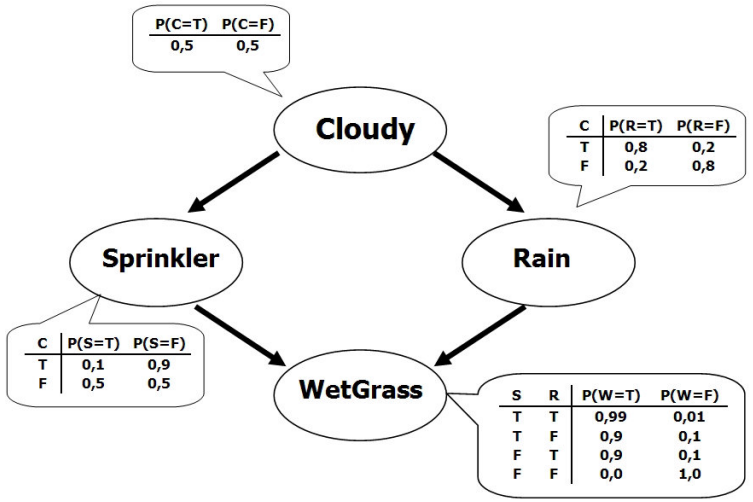




Bayesian Network or Belief Network

- Bayesian Network
  - A graphical model with directed acyclic graph
  - There are observed(visible)/unobserved(hidden) stochastic variables
  - Two important tasks
    - Inference : from observed variables, infer probabilities of unobserved variables
    - Training : update the parameters to make the observed data more likely to be generated by the network.

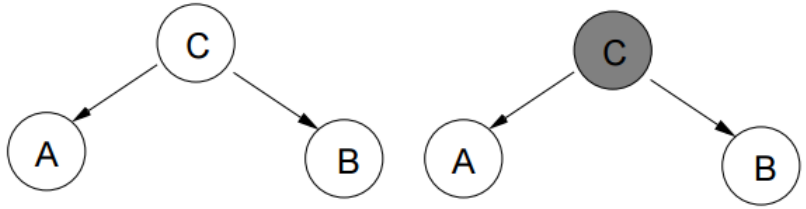




Standard Structures of Bayesian Network

- Head to Tail
  - When C is not observed $P(A,B) = \sum_C P(A,B,C) = P(A)\sum_C P(C|A)P(B|C)$ ⇨ **dependent**
  - When C is observed $P(A,B|C=\mathrm{T}) = P(A)P(B|C=\mathrm{T})$ ⇨ **independent**

- Tail to Tail
  - When C is not observed $P(A,B) = \sum_C P(A,B,C) = \sum_C P(C)P(A|C)P(B|C)$ ⇨ **dependent**
  - When C is observed $P(A,B|C=\mathrm{T}) = P(A|C=\mathrm{T})P(B|C=\mathrm{T})$ ⇨ **independent**



Standard Structures of Bayesian Network

- Head to Head
  - When C is not observed,

$$P(A,B) = \sum_C P(A,B,C) = P(A)P(B)\sum_C P(C|A,B) = P(A)P(B)$$ ⇨ **independent**

  - When C is observed,

$$P(A,B|C=\mathtt{T}) = \frac{P(A,B,C=\mathtt{T})}{P(C=\mathtt{T})} = \frac{P(C=\mathtt{T}|A,B)P(A)P(B)}{P(C=\mathtt{T})}$$ ⇨ **dependent**

  - Two variables become dependent when their child is observed : "**Explaining Away**"

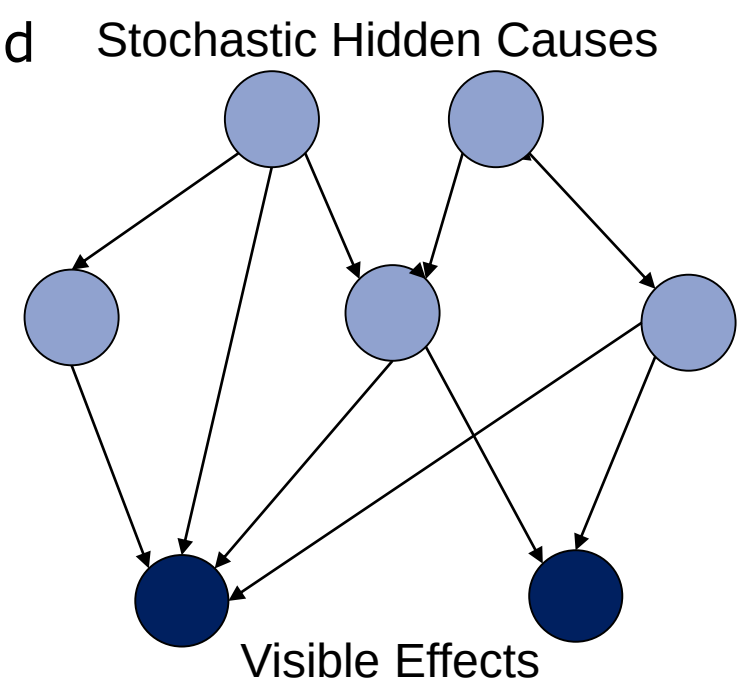
Sigmoid Belief Network

- Structure
  - Bayesian Network with observed variables at the bottom.
  - A directed connection from A to B means A is the parent node of B.
  - The weight on the connection from parent j to i is w_{ij} and the bias of node i is b_i .
  - For node i with state s_i and set of parents Pa_i , the condition probability of the node turning on is

$$P(s_i = 1 | s_{\mathrm{Pa}_i}) = \sigma\left(b_i + \sum_{j \in \mathrm{Pa}_i} s_j \cdot w_{ji}\right)$$

    where $\sigma(x) = \frac{1}{1+e^{-x}}$.
  - The full configuration of SBN is

$$P(s) = \prod_i P(s_i | s_{\mathrm{Pa}_i}) \quad \text{and} \quad P(\mathbf{s_v}) = \sum_{\mathbf{s_h}} \mathbf{P}(\mathbf{s_v}, \mathbf{s_h})$$



Sigmoid Belief Network

- Learning
  - To obtain the log probability gradient of w_{ij}, let the probability of the whole network as

$$\mathrm{p}(N) = p(s_i \mid Pa)\, p(Pa \mid R)\, \mathrm{p}(\mathrm{R})$$

    Here, $p(s_i \mid Pa)$ is a probability of s_i given its parents, and let's write it simply as $p(s_i)$.
    $p(Pa \mid R)$ is a probability of the parents of s_i, given the state of rest of the network and let's write it simply as $p(Pa)$. Finally, $p(R)$ is a probability of the nodes in the network excluding the node s_i and its parents.

  - Then $p(N) = p(Pa)p(R)p(s_i)$ and

$$\frac{d(log(p(N))}{dw_{ij}} = \frac{d}{dw_{ij}}\left(log(p(Pa)) + log(p(R)) + log(p(s_i))\right)$$

$$= \frac{d}{dw_{ij}}\left(log(p(s_i))\right)$$

Sigmoid Belief Network

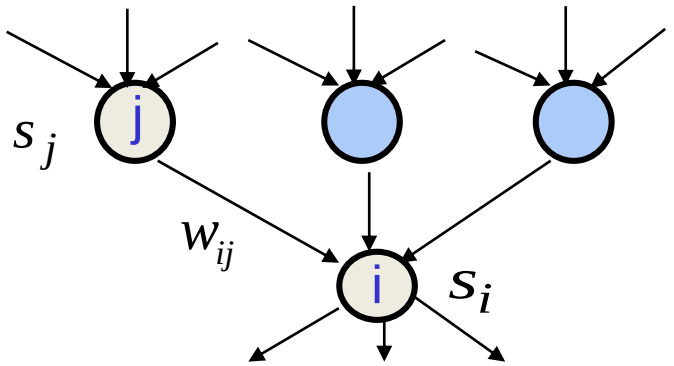
- Learning(Cont'd)
  - When $s_i = 1$

$$\frac{d(log(P(N))}{dw_{ij}} = \frac{d}{dw_{ij}}\left(log(\sigma(\Sigma w_{ik} s_k))\right)$$

$$= s_j\, \sigma(\Sigma w_{ik} s_k)(1 - \sigma(\Sigma w_{ik} s_k))\, \frac{1}{\sigma(\Sigma w_{ik} s_k)}$$

$$= s_j\,(1 - \sigma(\Sigma w_{ik} s_k))$$

$$= s_j(s_i - p_i)$$

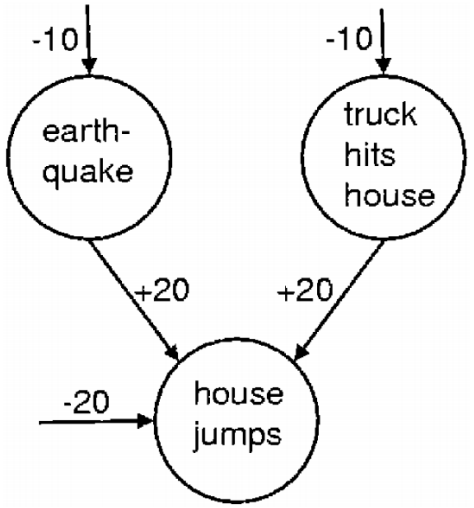


  - Similarly, whe$s_i = 0$,

$$\frac{d(log(P(N))}{dw_{ij}} = s_j(s_i - p_i)$$

Explaining Away

- Broken Independence
  - Even if parents are independent each other, they become dependent when their child is observed.
  - When we observe the house jumps and if we know there was an earthquake, it reduces the probability of truck hits the house.



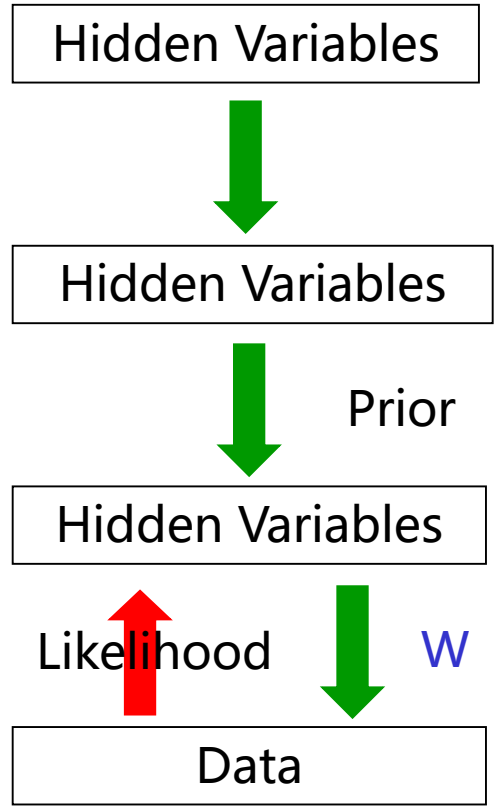
$$p(s_i = 1) = \frac{1}{1 + \exp\left(-b_i - \sum_j s_j w_{ij}\right)}$$

Posterior over Causes

p(1,1)=.0001
p(1,0)=.4999
p(0,1)=.4999
p(0,0)=.0001

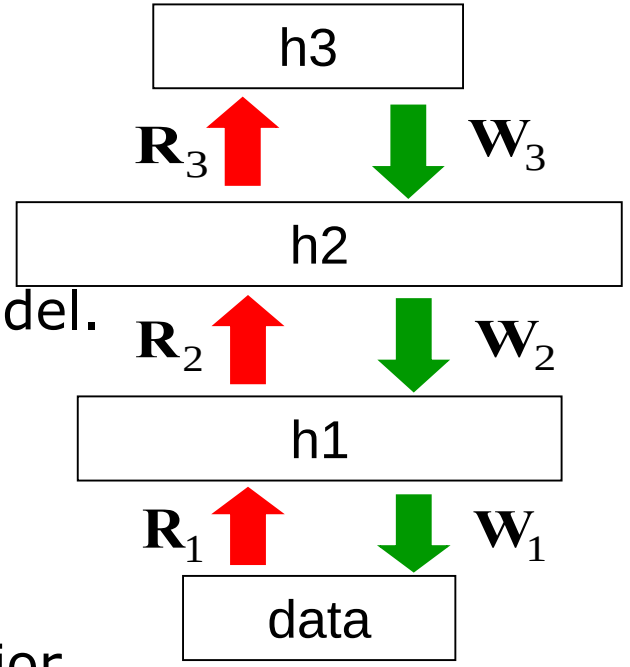
Learning Deep Sigmoid Belief Network

- Why Is It Hard to Train Deep Sigmoid Belief Network?
  - To learn W, we need the posterior distribution in the first hidden layer.
  - Problem 1: The posterior is typically intractable because of "explaining away".
  - Problem 2: The posterior depends on the prior as well as the likelihood.
    - So to learn W, we need to know the weights in higher layers, even if we are only approximating the posterior.
  - Problem 3: We need to integrate over all possible configurations of the higher variables to get the prior for first hidden layer.



Wake-sleep Algorithm

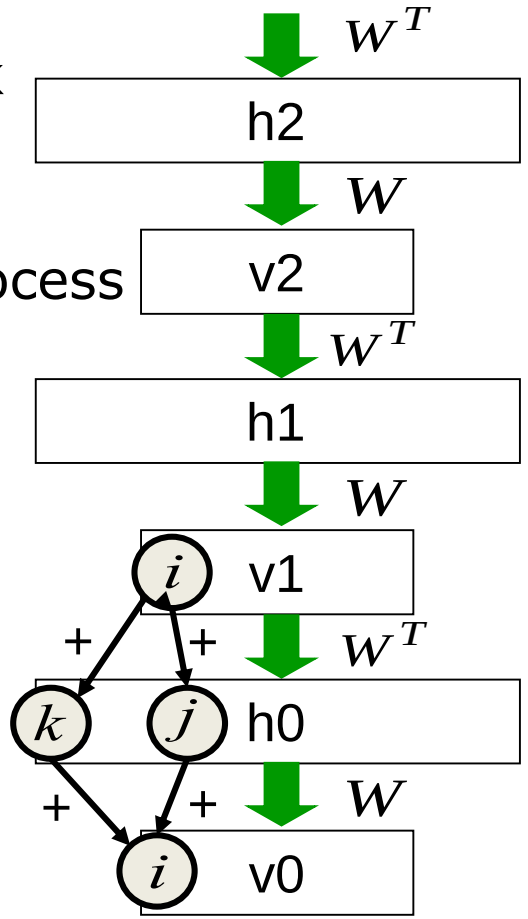
- Wake Phase
  - Use recognition weights to perform a bottom-up pass.
  - Train the generative weights to reconstruct activities in each layer from the layer above.
- Sleep Phase
  - Use generative weights to generate samples from the model.
  - Train the recognition weights to reconstruct activities in each layer from the layer below.
- Crazy Idea That Works but Not That Well
  - It's a variational methods which approximates the posterior.
  - Using samples from wrong distribution for maximum likelihood learning.
  - Wasteful and incorrect -> There should be a better way.



Deep Belief Networks

RBM as an Infinite Sigmoid Belief Network

- Infinite Sigmoid Belief Net with Tied Weight
  - The distribution generated by infinite sigmoid belief network with tied weight W is same with the distribution of RBM defined by W.
  - Top-down pass is equivalent to making Markov sampling process of RBM converge to its equilibrium.
- Conditional Independence in Inference
  - We can perform inference using conditional inference by removing 'explaining away'.
  - This is because the layers above the layer of interest implements 'complementary prior'.
  - Inference is making Markov sampling process of RBM starting from data point.



Complementary Prior

- Definition
  - For a given likelihood function $P(\mathbf{x}|\mathbf{y})$, the prior distribution $P(\mathbf{y})$ is called complementary prior if it makes the joint distribution $P(\mathbf{x}, \mathbf{y}) = P(\mathbf{x}|\mathbf{y})P(\mathbf{y})$ leads to the posterior factorizable, i.e. $P(\mathbf{y}|\mathbf{x}) = \prod_j P(y_j|\mathbf{x})$.

- Exemplary Likelihood Family
  - When we assume $P(\mathbf{y}) > 0$ and $P(\mathbf{x}|\mathbf{y}) > 0$, following functional form of likelihood

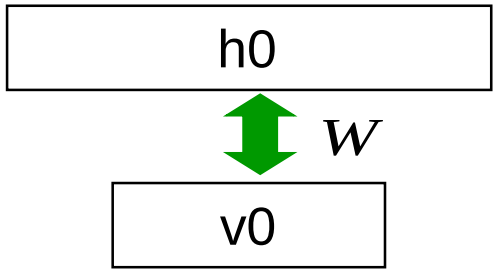
$$P(\mathbf{x}|\mathbf{y}) = \frac{1}{\Omega(\mathbf{y})} \exp\left(\sum_j \Phi_j(\mathbf{x}, y_j) + \beta(\mathbf{x})\right) = \exp\left(\sum_j \Phi_j(\mathbf{x}, y_j) + \beta(\mathbf{x}) - \log \Omega(\mathbf{y})\right)$$

  admits complementary prior and joint distribution

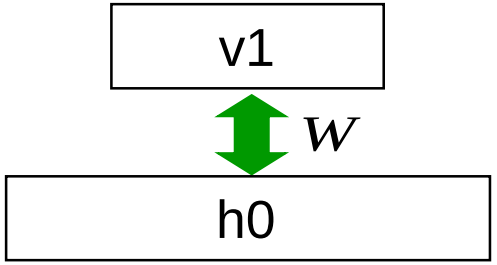
$$P(\mathbf{y}) = \frac{1}{C} \exp\left(\log \Omega(\mathbf{y}) + \sum_j \alpha_j(y_j)\right) \quad P(\mathbf{x}, \mathbf{y}) = \frac{1}{C} \exp\left(\sum_j \Phi_j(\mathbf{x}, y_j) + \beta(\mathbf{x}) + \sum_j \alpha_j(y_j)\right).$$

Learning a Deep Directed Network

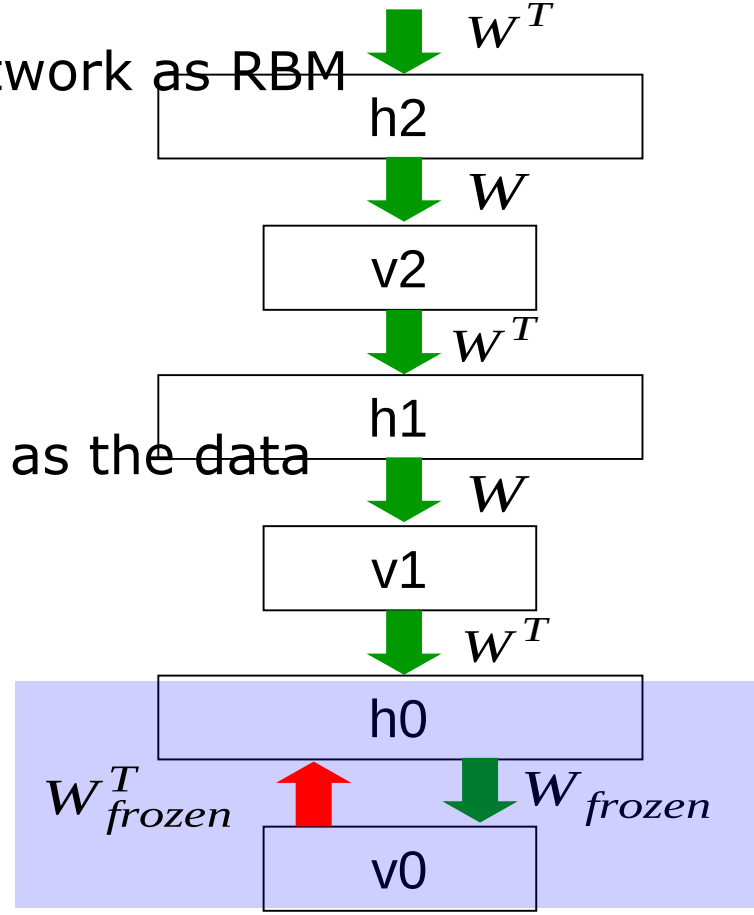
- Layer-wise Training with RBM
  - We first start with the tied weight and train the whole network as RBM



  - Then freeze the first layer and train another RBM with h0 as the data

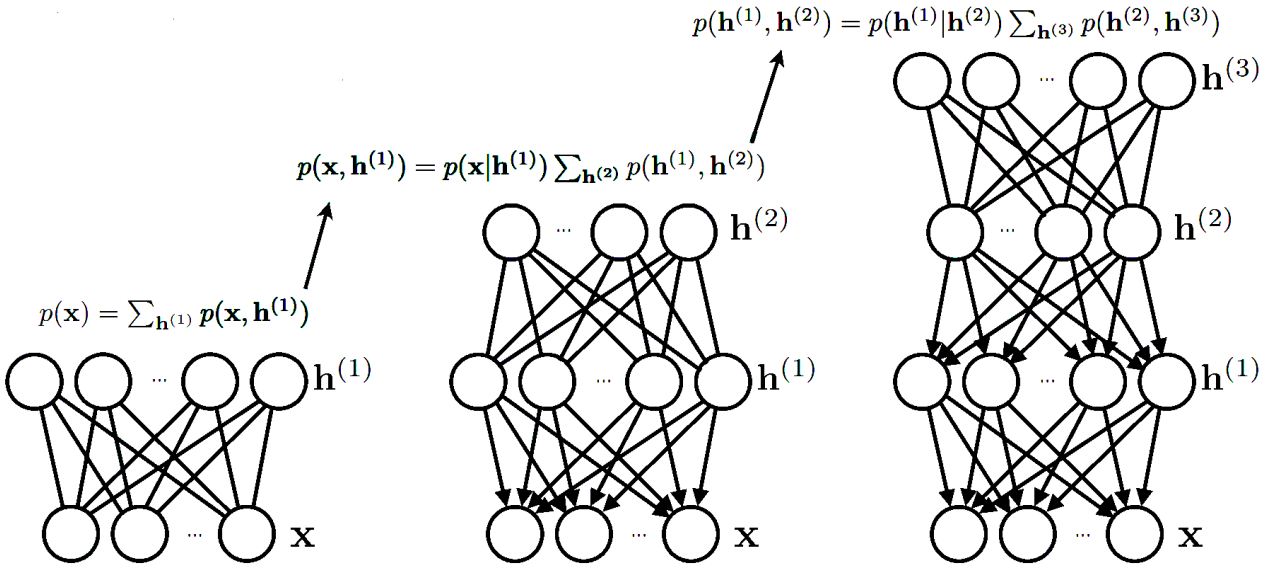


  - We end up with network with RBM at the top.



Broken Complementary Prior

- What Happens When the Weights are Untied?
  - The higher layers no longer implements a complementary prior.
  - Performing inference using the frozen weight no longer correct.
  - But it still tries to improve the variational bound.



Variational Bound

- Why Layer-wise Greedy Training Works?

$$
\begin{aligned}
\log p(\mathbf{x}) &= \log \left(\sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \frac{p(\mathbf{x}, \mathbf{h}^{(1)})}{q(\mathbf{h}^{(1)}|\mathbf{x})} \right) \\
&\geq \sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \log \left(\frac{p(\mathbf{x}, \mathbf{h}^{(1)})}{q(\mathbf{h}^{(1)}|\mathbf{x})} \right) \\
&= \sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \log p(\mathbf{x}, \mathbf{h}^{(1)}) \\
&\quad - \sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \log q(\mathbf{h}^{(1)}|\mathbf{x}) \\
&= \sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \left(\log p(\mathbf{x}|\mathbf{h}^{(1)}) + \log p(\mathbf{h}^{(1)}) \right) \\
&\quad - \sum_{\mathbf{h}^{(1)}} q(\mathbf{h}^{(1)}|\mathbf{x}) \log q(\mathbf{h}^{(1)}|\mathbf{x})
\end{aligned}
$$

Another RBM !
Maximizing this term leads to better lower bound of log likelihood of data

This is a good initial approximate distribution of true posterior

Fine-tuning for DBNs

Generative Fine-tuning for DBNs

- Contrastive Version of Wake-Sleep Algorithm

1. Do a stochastic bottom-up pass

- Then adjust the top-down weights of lower layers to be good at reconstructing the feature activities in the layer below.

2. Do a few iterations of sampling in the top level

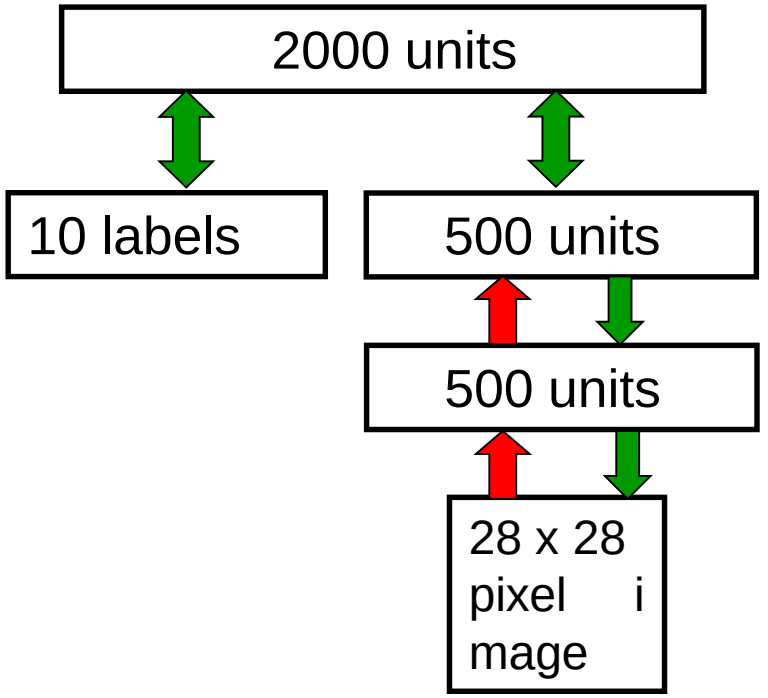
- Then adjust the weights in the top-level RBM using CD.

3. Do a stochastic top-down pass

- Then Adjust the bottom-up weights to be good at reconstructing the feature activities in the layer above.

DBN for Modeling Labels and Data

- DBN for Modeling Joint Distribution of MNIST digits and labels
  - The first two hidden layers are learned without using labels.
  - The top layer is learned as an RBM for modeling the labels concatenated with the features in the second hidden layer.
  - The weights are then fine-tuned to be a better generative model using contrastive wake-sleep.



Examples

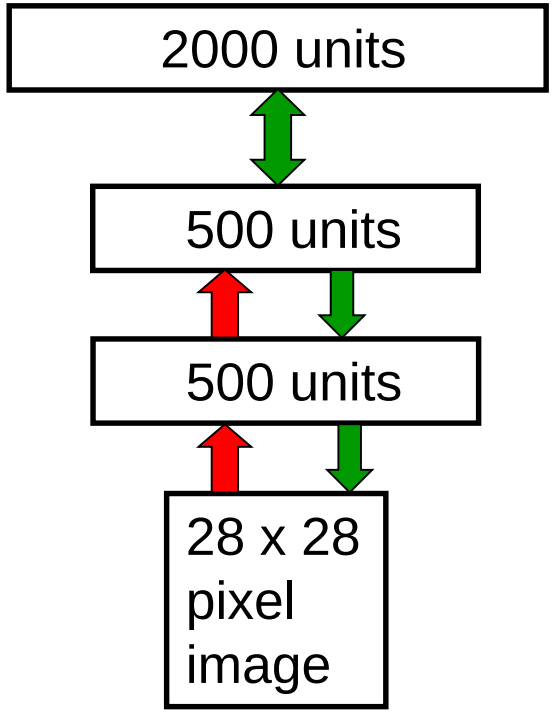
- MNIST Digits Generated by DBN

Samples for every 1000 Gibbs sampling

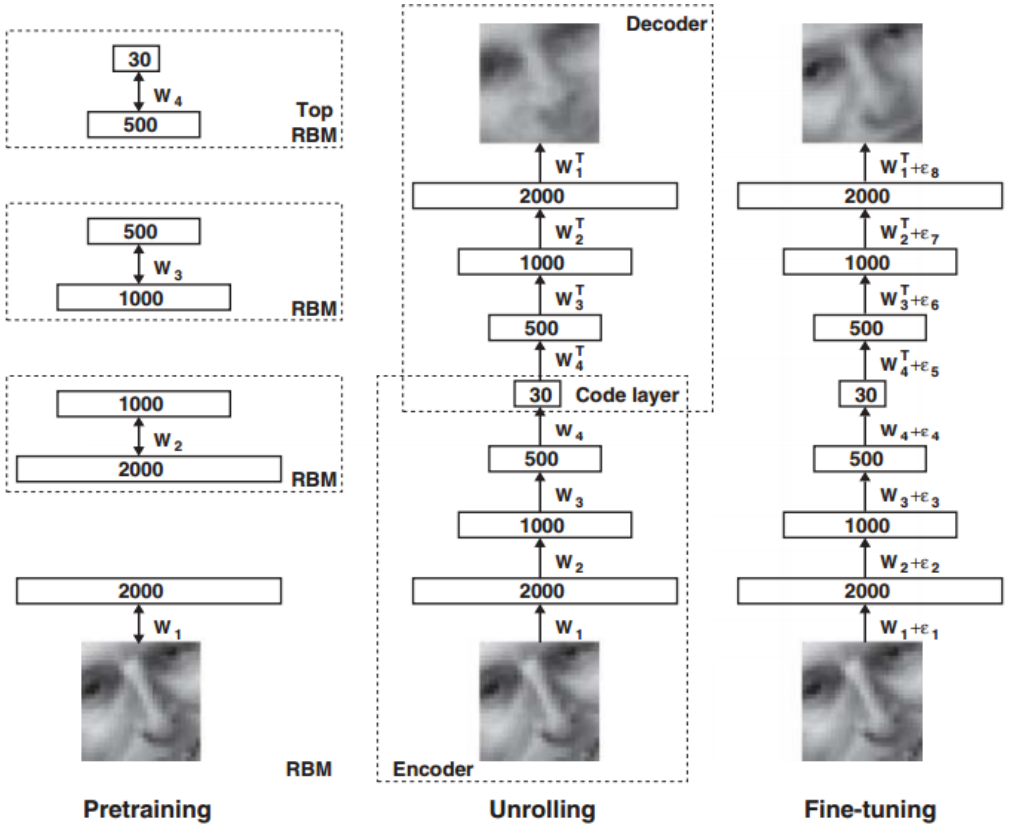
Samples for every 20 Gibbs sampling with Random Input

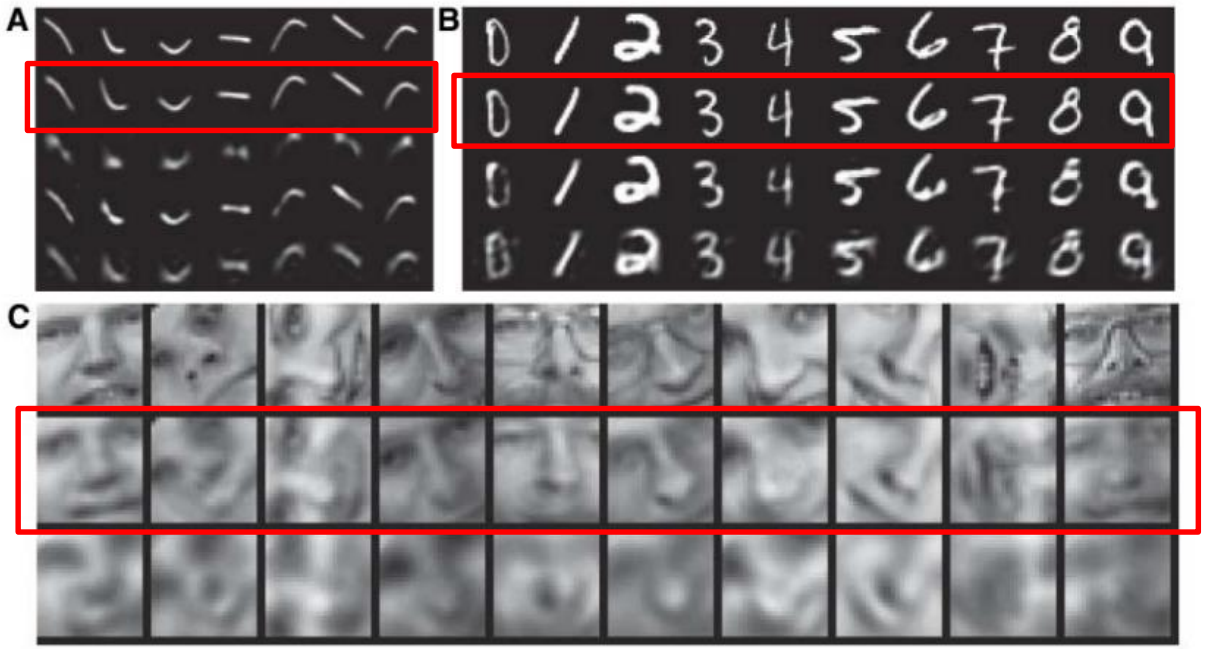
Discriminative Fine-tuning for DBNs

- Discriminative Fine-tuning using Back-propagation
  - Add a 10-way softmax layer at the top and do back-propagation
  - Backprop net with one or two hidden layers : 1.6%
  - Backprop with L2 constraints on incoming weights : 1.5%
  - Support Vector Machines : 1.4%
  - Generative model of joint density of images and labels : 1.25%
    (+ Generative fine-tuning)
  - Generative model of unlabeled digits : 1.0%
    (+ Discriminative fine-tuning)



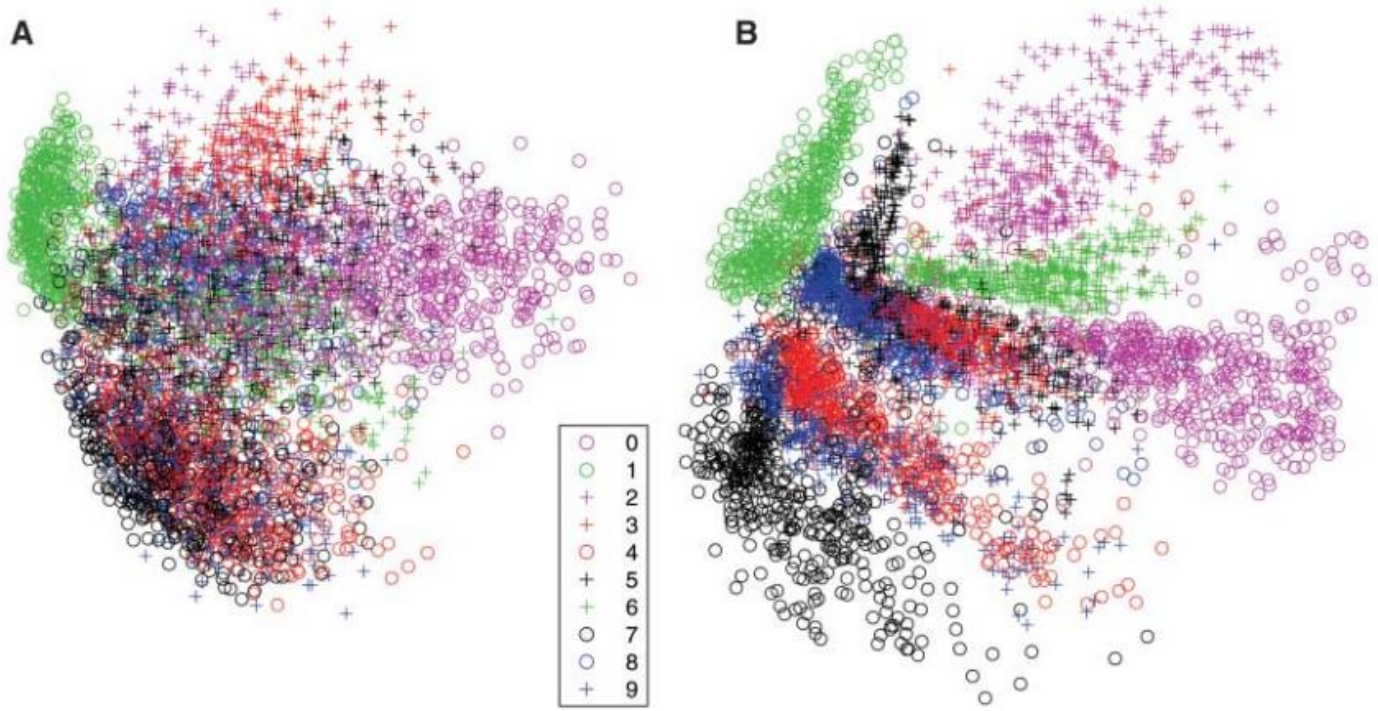
Stacking RBMs for Deep Auto-Encoder





Stacking RBMs for Deep Auto-Encoder

- Visualizing Features Learned by Deep Auto-Encoder

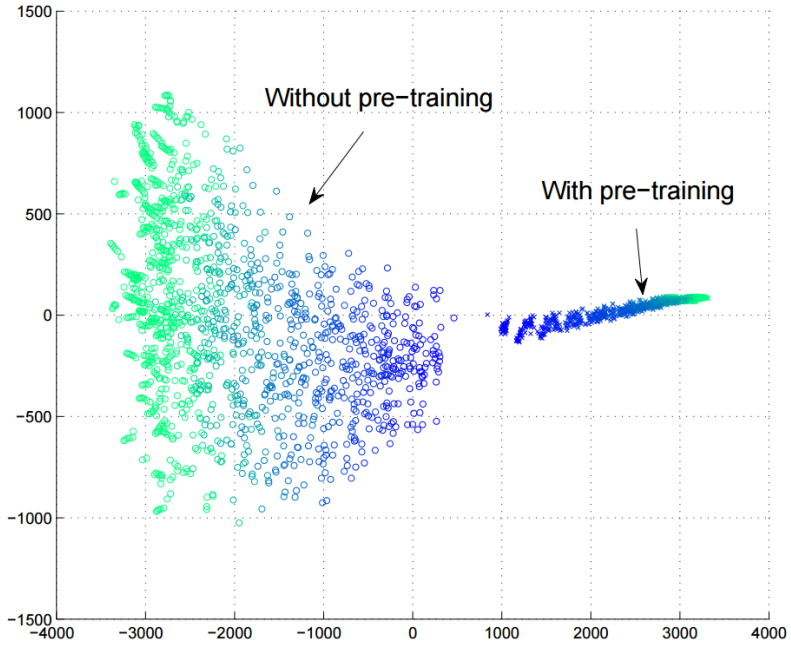


Pre-training

- *(Erhan et al 2009, JMLR)*
- Supervised deep net (tanh), with or w/o unsupervised pre-training → very different minima

Neural net trajectories in function space, visualized by t-SNE

No two training trajectories end up in the same place → huge number of effective local minima

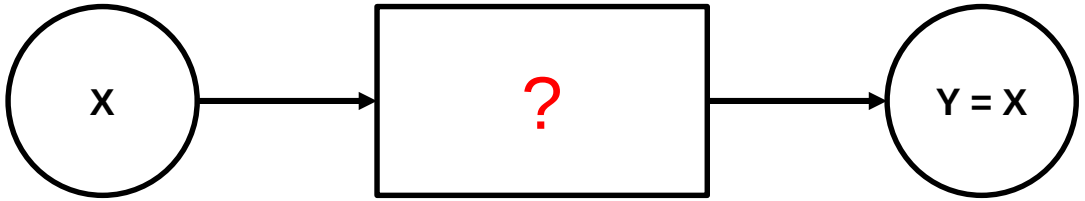


Class 5
Auto-Encoders

Overview

The Motivation

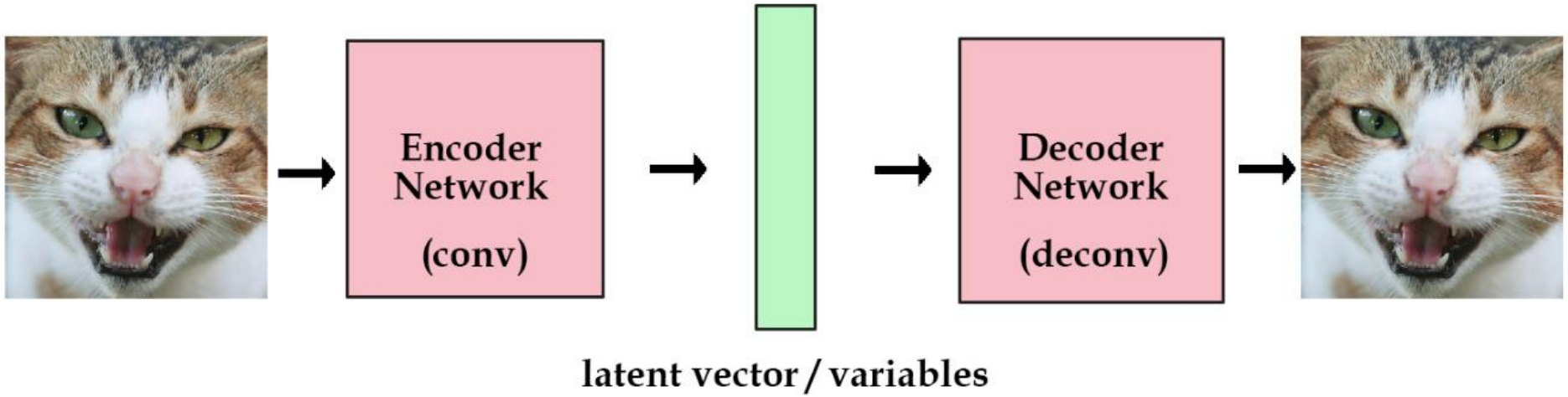
- Training a Model to Generate Itself
  - We can use the inputs as the outputs



  - There are many trivial or uninteresting mapping '?'s
  - We can make the mapping '?' to represent well the data by introducing
    - Bottleneck : Auto-Encoder
    - Noise : Denoising Auto-Encoder
    - Regularizer : Sparse Auto-Encoder, Contractive Auto-Encoder
    - Stochasticity : Variational Auto-Encoder

Overview

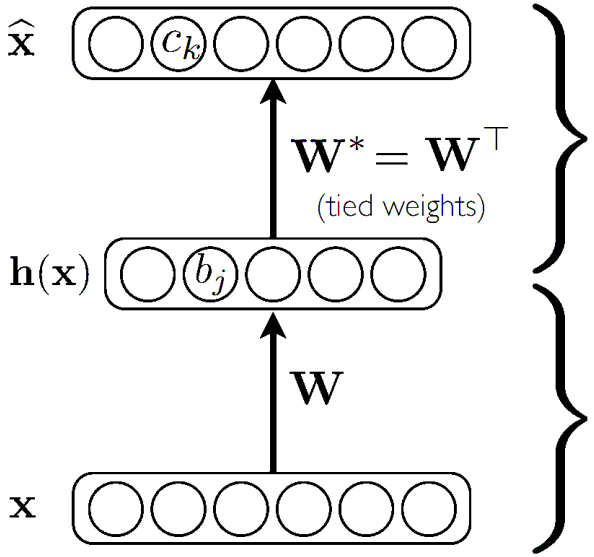
- Basic Structure of Auto-encoder



Auto-Encoder

Vanilla Auto-Encoder

- Multilayer Perceptron for Unsupervised Representation Learning
  - Feed-forward neural network trained to reproduce its input at the output layer
  - "Encoder-Decoder" or "Representation-Reconstruction" structure
    - Encoder compresses input to latent vector
    - Decoder decompresses latent vector to reconstruct input
    - Encoder and decoder shares same set of parameters



Decoder

$$\hat{\mathbf{x}} = o(\hat{\mathbf{a}}(\mathbf{x})) = \underbrace{\mathrm{sigm}}_{\text{for binary inputs}}(\mathbf{c} + \mathbf{W}^*\mathbf{h}(\mathbf{x}))$$

Encoder

$$\mathbf{h}(\mathbf{x}) = g(\mathbf{a}(\mathbf{x})) = \mathrm{sigm}(\mathbf{b} + \mathbf{W}\mathbf{x})$$

Training Auto-Encoders

- Loss Function
  - For binary inputs : Cross-entropy loss
    - $l(f(\mathbf{x})) = -\sum_k \left(x_k \log(\widehat{x}_k) + (1 - x_k) \log(1 - \widehat{x}_k)\right)$
  - For real-valued inputs : Sum of square loss
    - $l(f(\mathbf{x})) = \frac{1}{2} \sum_k (\widehat{x}_k - x_k)^2$
    - Linear activation function at the output layer

Training Auto-Encoder

- Loss Gradient
  - For both binary and real-value inputs, the loss gradient is

$$\nabla_{\hat{\mathbf{a}}(\mathbf{x}^{(t)})} l(f(\mathbf{x}^{(t)})) = \hat{\mathbf{x}}^{(t)} - \mathbf{x}^{(t)}$$

  - For cross entropy loss,

$$\frac{\partial l(f(\mathbf{x}))}{\partial \hat{a}(x_k)} = \frac{\partial l(f(\mathbf{x}))}{\partial \hat{x}_k} \cdot \frac{\partial \hat{x}_k}{\partial \hat{a}(x_k)} = \left(-\frac{x_k}{\hat{x}_k} + \frac{1-x_k}{1-\hat{x}_k} \right) \cdot \hat{x}_k (1-\hat{x}_k) = \frac{\hat{x}_k - x_k}{\hat{x}_k (1-\hat{x}_k)} \cdot \hat{x}_k (1-\hat{x}_k) = \hat{x}_k - x_k$$

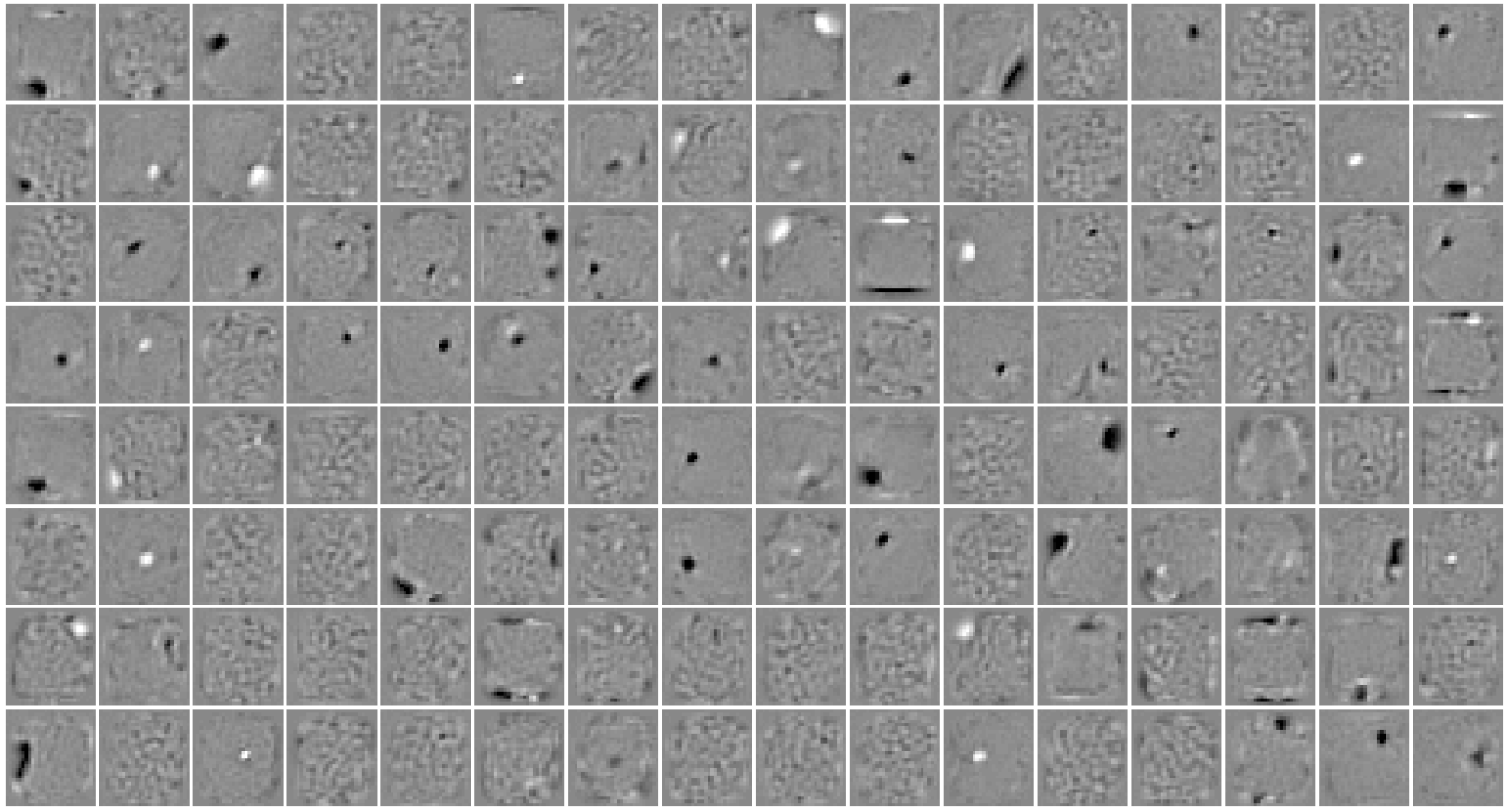
  - For sum of square loss,

$$\frac{\partial l(f(\mathbf{x}))}{\partial \hat{a}(x_k)} = \frac{\partial l(f(\mathbf{x}))}{\partial \hat{x}_k} \cdot \frac{\partial \hat{x}_k}{\partial \hat{a}(x_k)} = (\hat{x}_k - x_k) \cdot 1 = \hat{x}_k - x_k$$

  - We backpropagate this loss to update the weights
  - When the encoder-decoder weights are tied, the weight gradient is the sum of corresponding encoder gradient and decoder gradient.

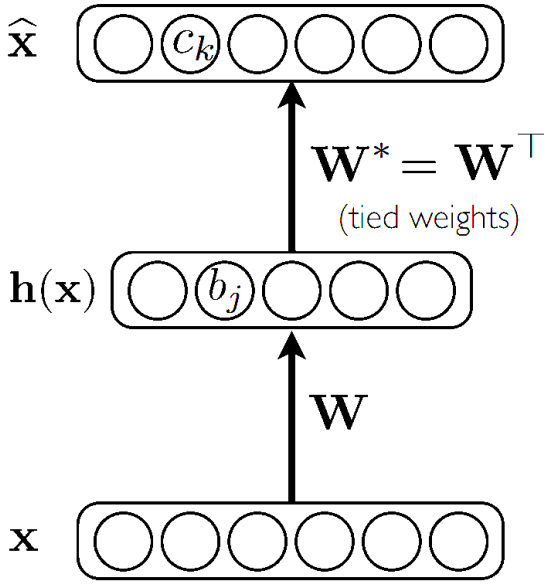
Example

- Filters of Auto-Encoder Trained with MNIST Dataset

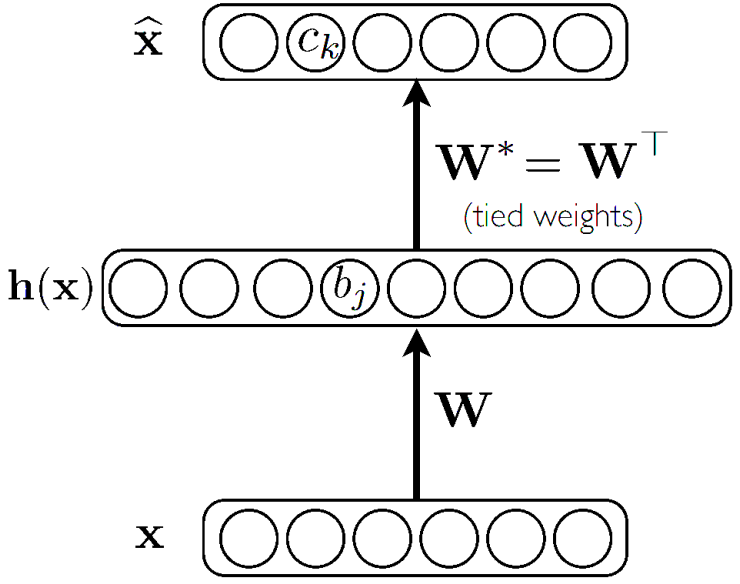


Undercomplete vs Overcomplete Hidden Layer

- Undercomplete Representation
  - Compressing input
  - Dimension reduction



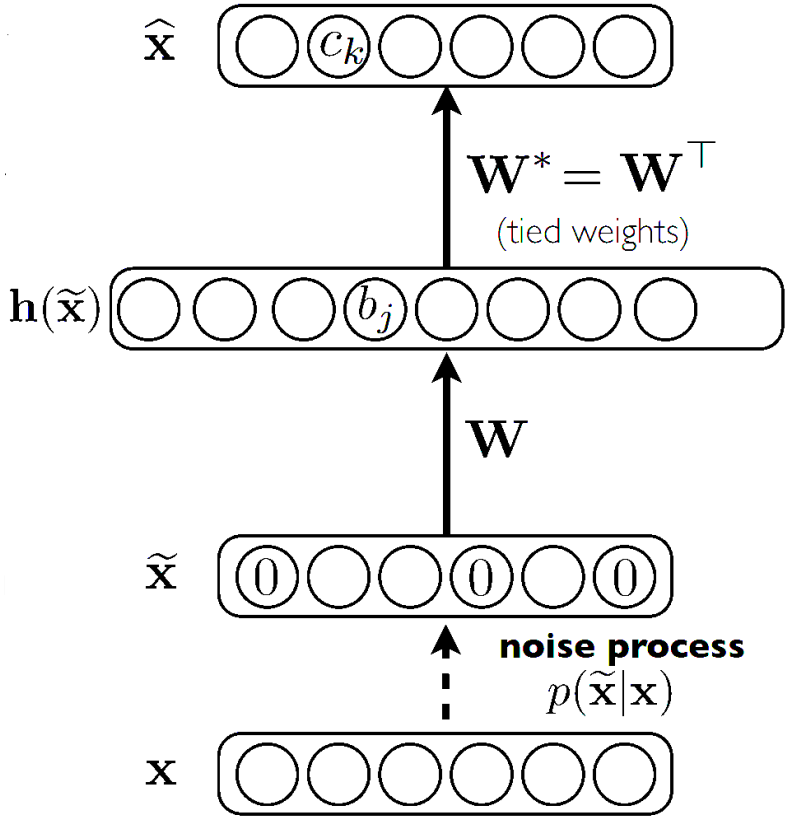
- Overcomplete Representation
  - No compression effect
  - No guarantee of extracting meaningful representation



Denoising Auto-Encoder

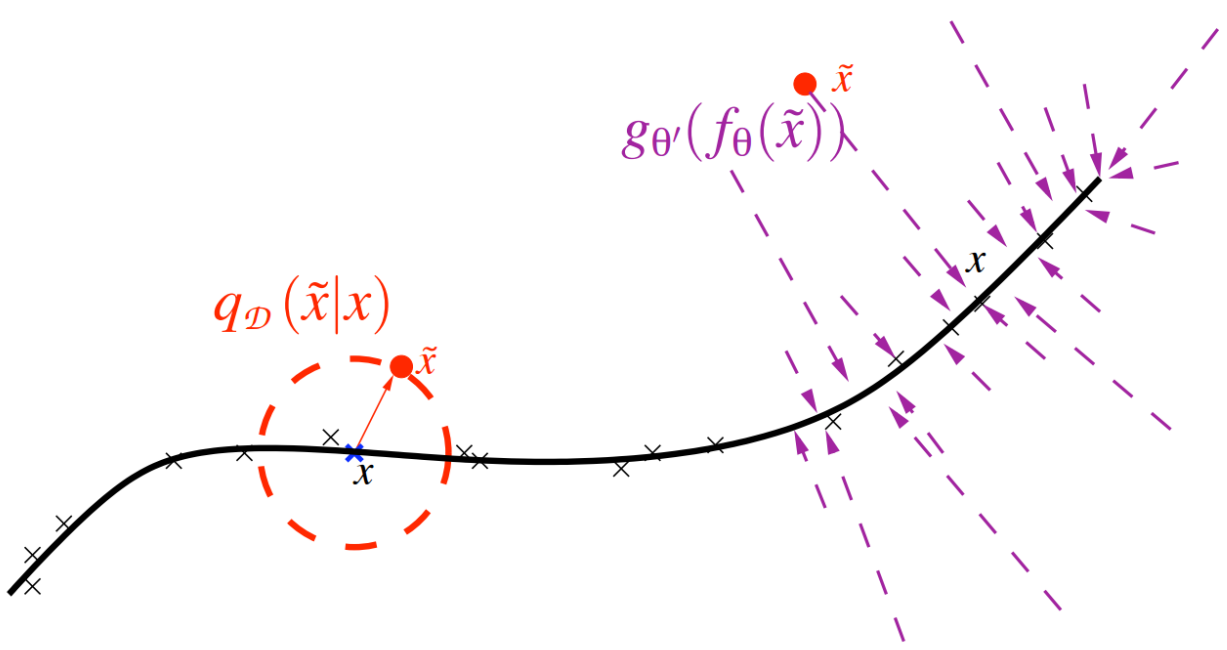
Denoising Auto-Encoder

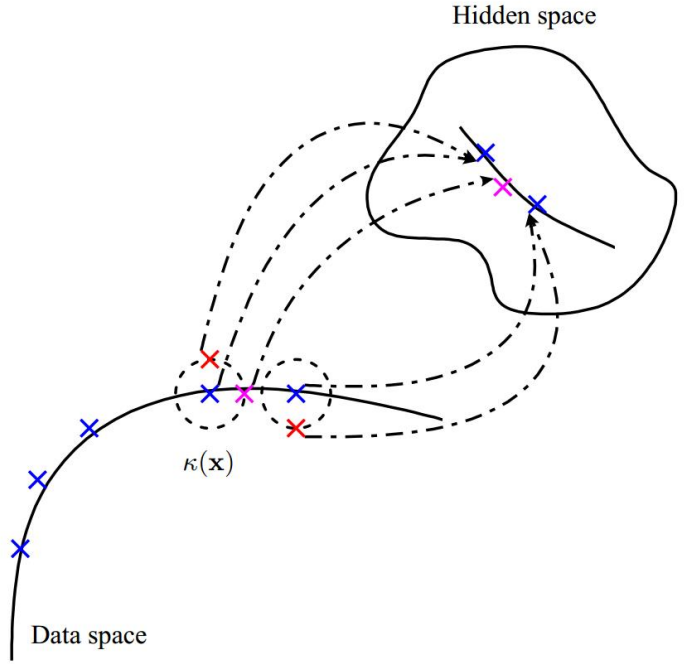
- Looking for More Robust Representation
  - An auto-encoder trained with 'corrupted' Input to reproduce 'clean' input.
  - Randomly assign '0' to subset of inputs with fixed probability.
    - The 'ancestor' of dropout.
  - Training procedure is same with 'vanilla' auto-encoder except that we use 'corrupted' inp as our training set.



Denoising Auto-Encoder

- DAE as Manifold Learning

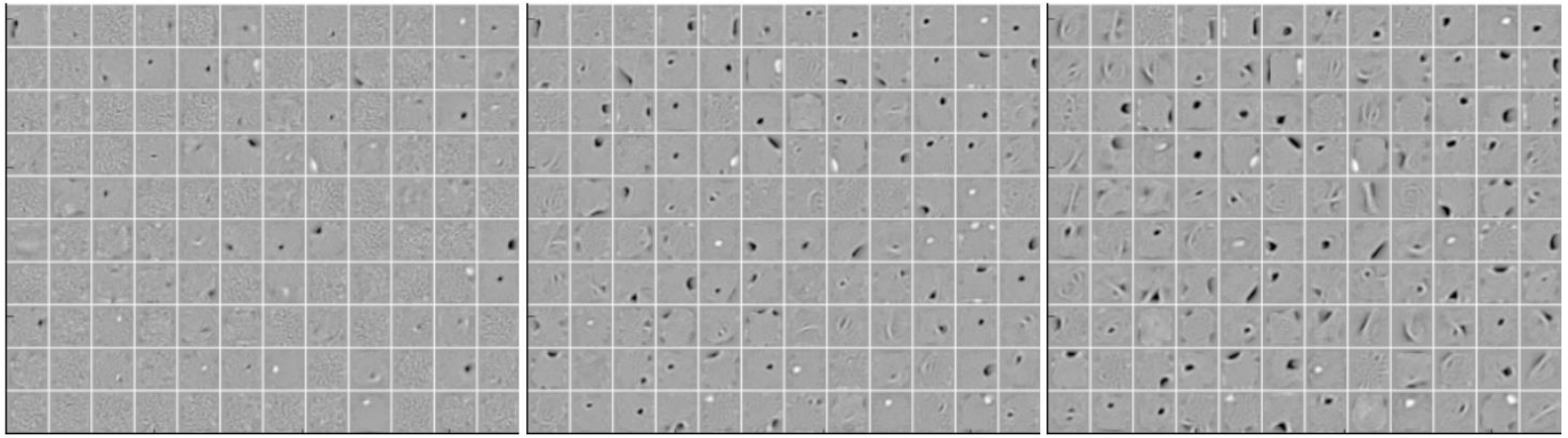




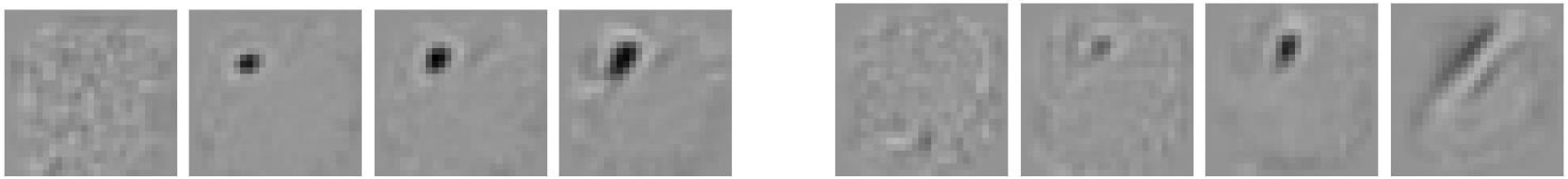
P. Vincent et al. (2010)

Denoising Auto-Encoder

- MNIST Example



(a) No destroyed inputs (b) 25% destruction (c) 50% destruction



(d) Neuron A (0%, 10%, 20%, 50% destruction) (e) Neuron B (0%, 10%, 20%, 50% destruction)

P. Vincent et al. (2008)

Sparse Auto-Encoder

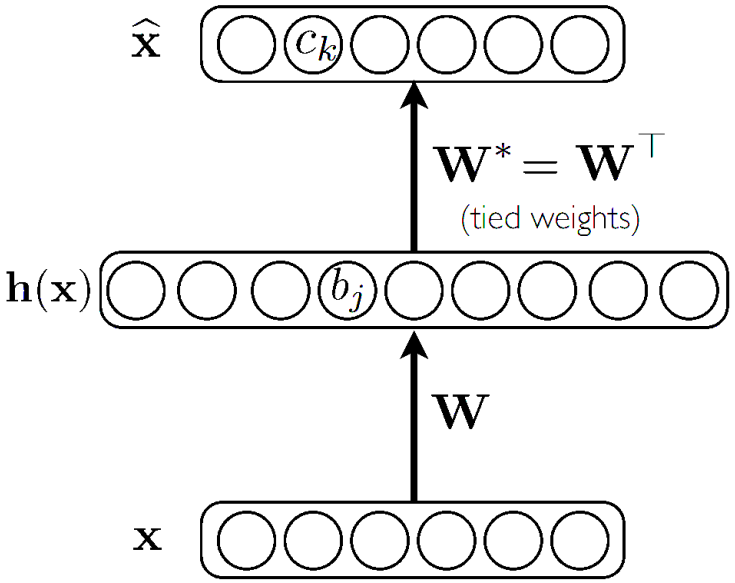
Sparse Auto-Encoder

- Looking for More Roubst Representation
  - By imposing sparsity of hidden layer.
  - By 'sparse', we mean most of the hidden layer activation is close to zero.
  - The average activation of hidden unit *k* over *m* trainin is

$$\hat{\rho}_k = \frac{1}{m}\sum_{i=1}^{m} g(a_k(\mathbf{x}))$$

  - We impose constraint $\hat{\rho}_k = \rho$

    where ρ is a sparsity parameter close to zero.



Sparse Auto-Encoder

- Additional Penalty Term
  - We penalize $\hat{\rho}_k$ deviating significantly from ρ
  - One way to do this is use Kullback-Leibler divergence between two Bernoulli random variable with mean ρ and $\hat{\rho}_k$ as

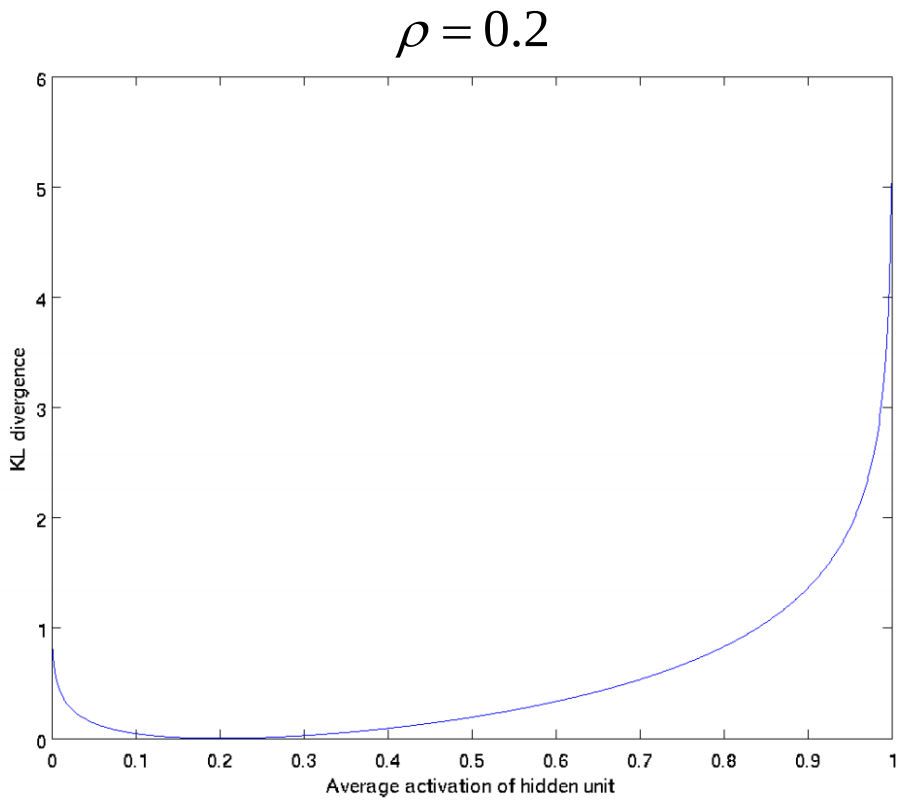
$$KL(\rho \,||\, \hat{\rho}_k) = \rho \log \frac{\rho}{\hat{\rho}_k} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_k}$$

- Overall Loss Function

$$l(f(\mathbf{x}))_{SAE} = l(f(\mathbf{x}))_{AE} + \lambda \sum_{k=1}^{h} KL(\rho \,||\, \hat{\rho}_k)$$

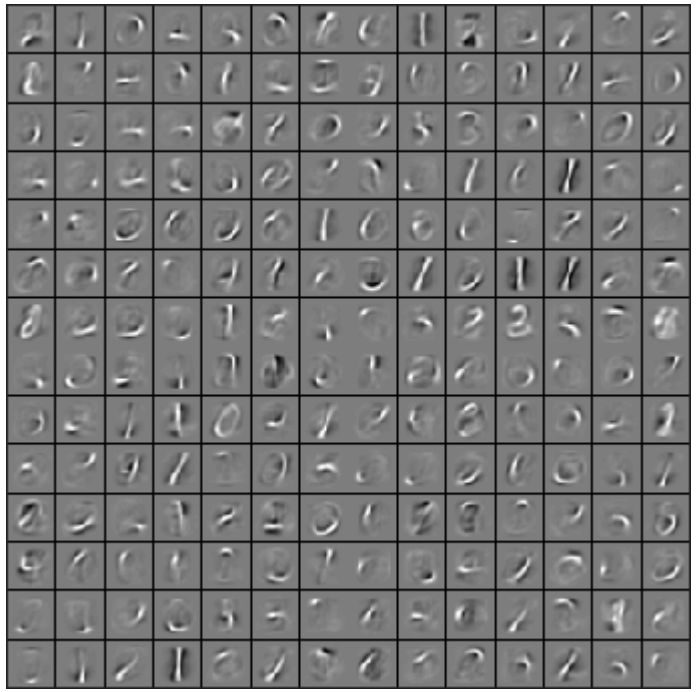
  - To calculate the gradient, we need to know all activation value of hidden node *k*.
  - The gradient of KL term at the hidden layer is obtaine using

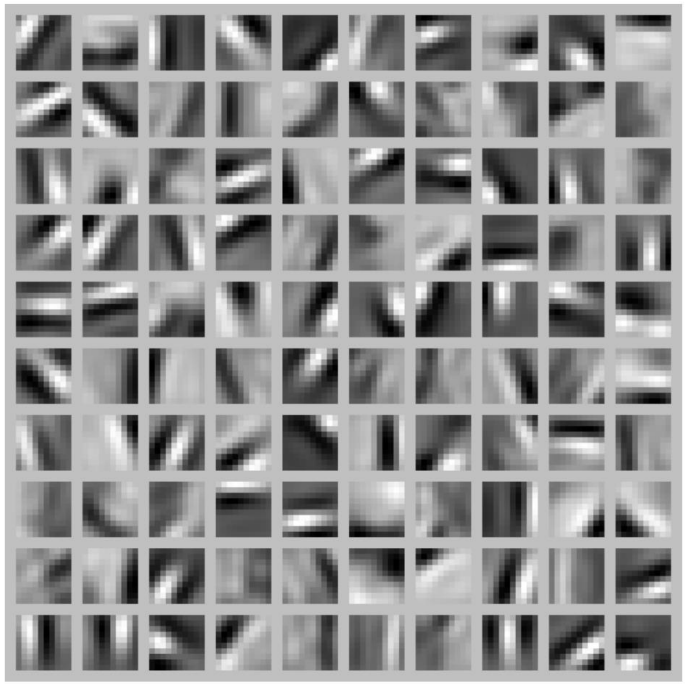
$$\hat{\rho}_k = \frac{1}{m} \sum_{i=1}^{m} g(a_k(\mathbf{x}))$$



Sparse Auto-Encoder

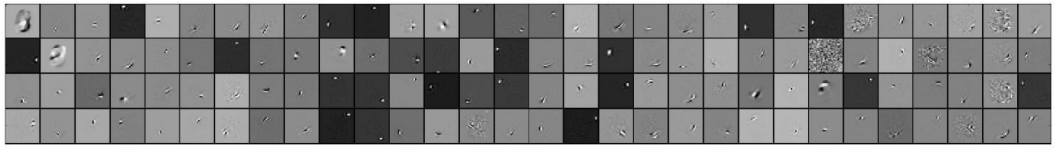
- Example
  - Feature learned by sparse auto-encoder using MNIST dataset(left) and some natural image(right).



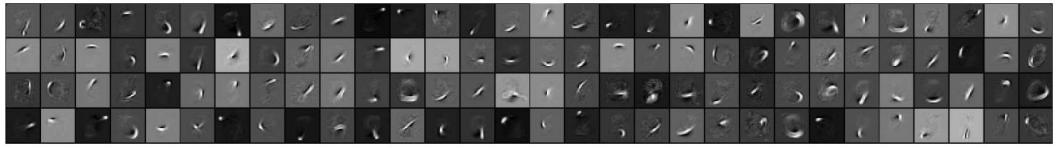


k-Sparse AutoEncoder

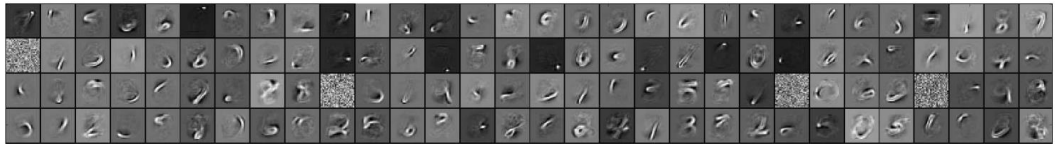
- Sparse auto-encoder with exact sparsity control
  - Select top-k hidden units as active nodes and set others to zero.
  - Use linear activation function or thresholding ReLU.



(a) $k = 70$



(b) $k = 40$



(c) $k = 25$

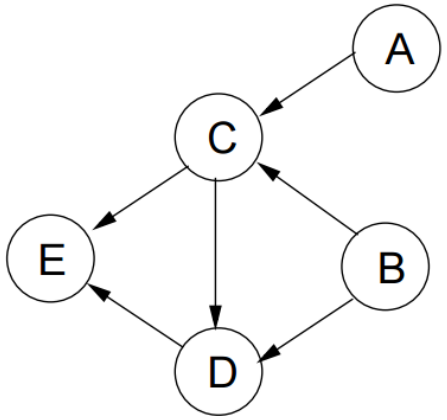
(d) $k = 10$

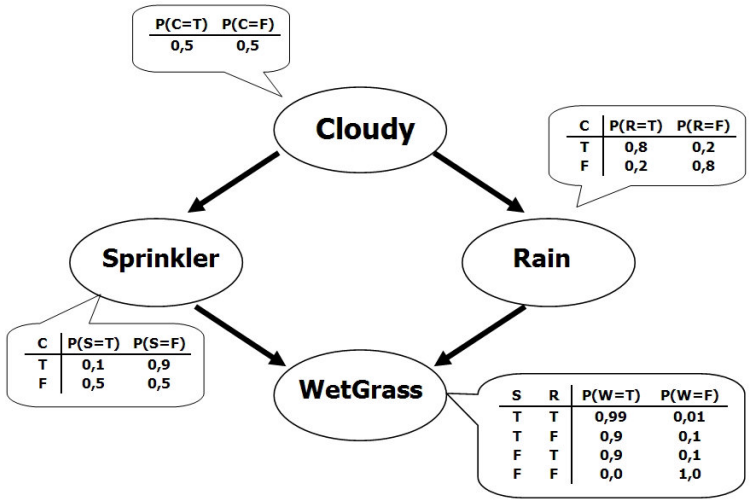
A. Makhzani et al.(2013)

Variational Auto-Encoder

Bayesian Network

- Bayesian Network
  - A graphical model with directed acyclic graph(DAG)
  - There are observed(visible)/unobserved(hidden) stochastic variables
  - Two important tasks
    - Inference : from observed variables, infer probabilities of unobserved variables
    - Training : update the parameters to make the observed data more likely to be generated by the network.





Learning Bayesian Network

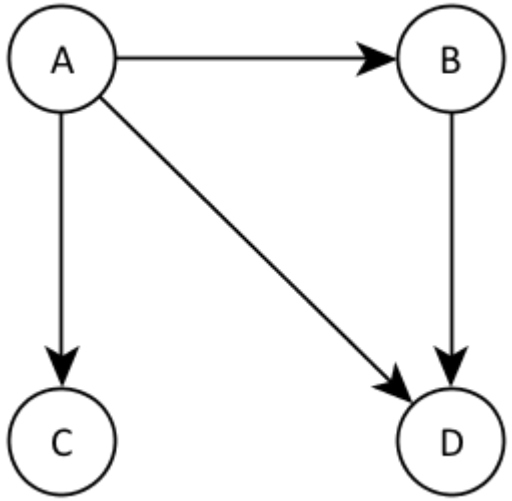
- Training Bayesian Network
  - We are looking for set of parameters of Bayesian Network by maximizing log-likelihood of joint distribution

$$\theta^* = \arg\max_\theta \log P(X_1, \cdots, X_n) = \arg\max_\theta \sum_{i=1}^{n} \log P(X_i \mid \mathcal{P}a(X_i))$$

  - When C,D are not observed, we need to be able to evaluate and maximize

$$P(C, D) = \sum_A \sum_B P(A, B, C, D)$$

  - However, when the state space is large, evaluating this likelihood is intractable

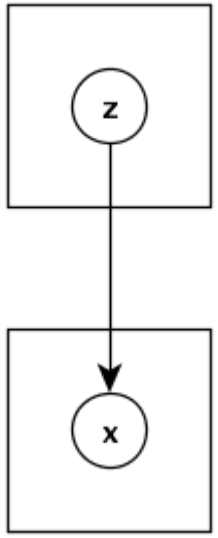


$$P(A, B, C, D) = P(A)P(B \mid A)P(C \mid A)P(D \mid A, B)$$

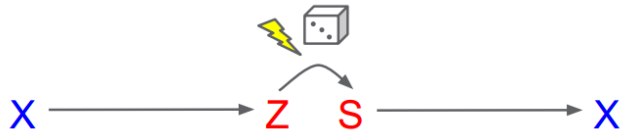
Variational Inference

- Maximizing Lower Bound of Likelihood
  - Let x visible (either discrete or continuous) variable and z hidden continuous variable.
  - Then, $p_\theta(\mathbf{x}, \mathbf{z}) = p_\theta(\mathbf{z}) p_\theta(\mathbf{x} \mid \mathbf{z})$
  - And we define recognition model $q_\phi(\mathbf{z} \mid \mathbf{x})$ which approximate true posterior $p_\theta(\mathbf{z} \mid \mathbf{x})$.
  - Then the Variational Auto-Encoder objective is

$$\mathcal{L}(\mathbf{x}) = -D_{KL}(q_\phi(\mathbf{z} \mid \mathbf{x}) \,||\, p_\theta(\mathbf{z})) + \mathrm{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x} \mid \mathbf{z})]$$

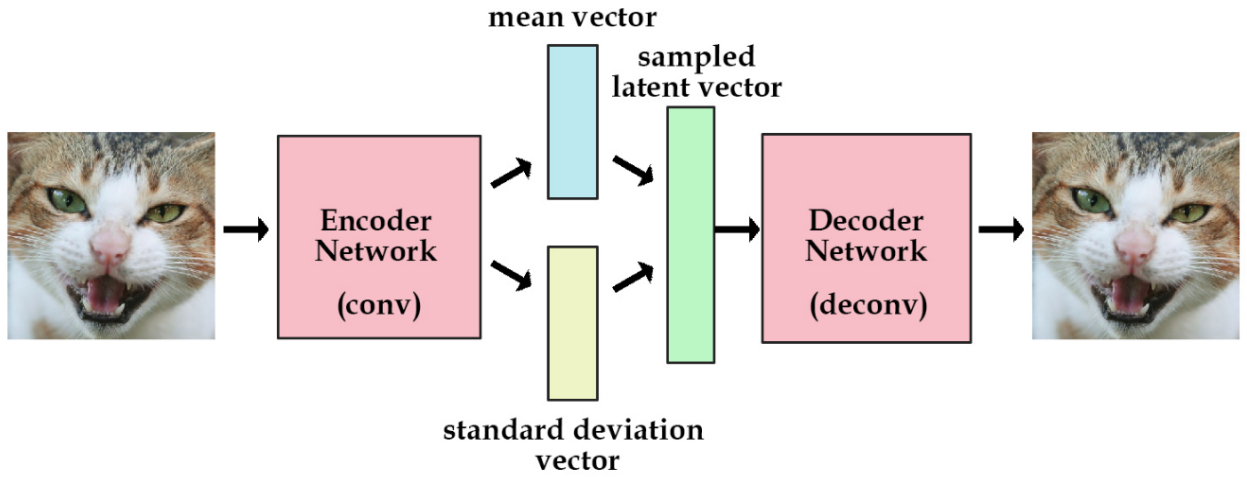


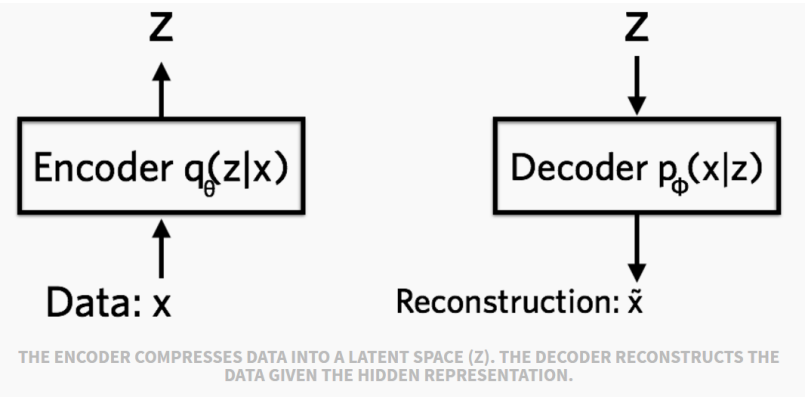
Variational Auto-Encoder



Consider Z as the parameters of a Gaussian. Sample S from it.

Very active area of research, spurred by the concept of
Variational Auto-Encoders



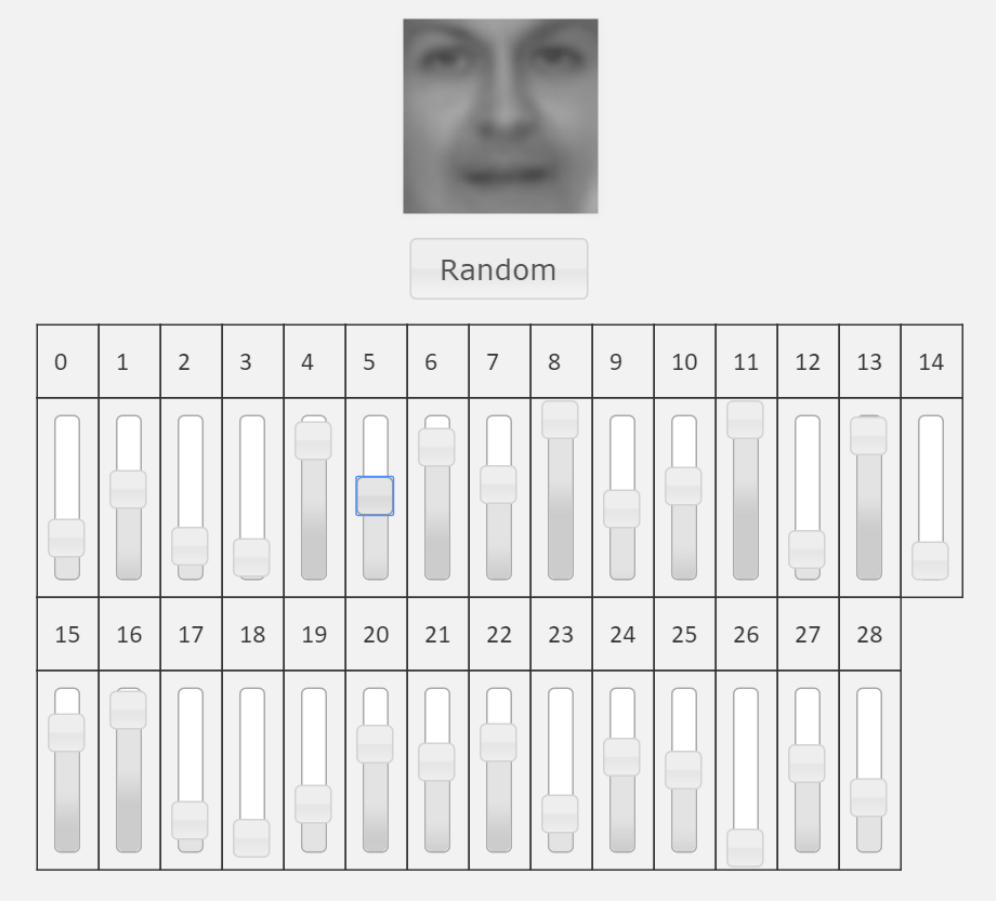


THE ENCODER COMPRESSES DATA INTO A LATENT SPACE (Z). THE DECODER RECONSTRUCTS THE DATA GIVEN THE HIDDEN REPRESENTATION.

```
generation_loss = mean(square(generated_image - real_image))
latent_loss = KL-Divergence(latent_variable, unit_gaussian)
loss = generation_loss + latent_loss
```

Variational Auto-Encoder

- Morphing Faces



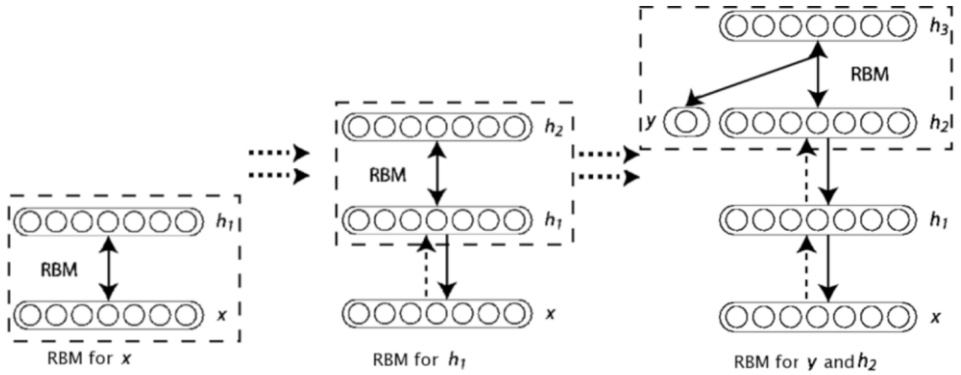
Stacking Auto-Encoders

Stacking RBMs and AEs

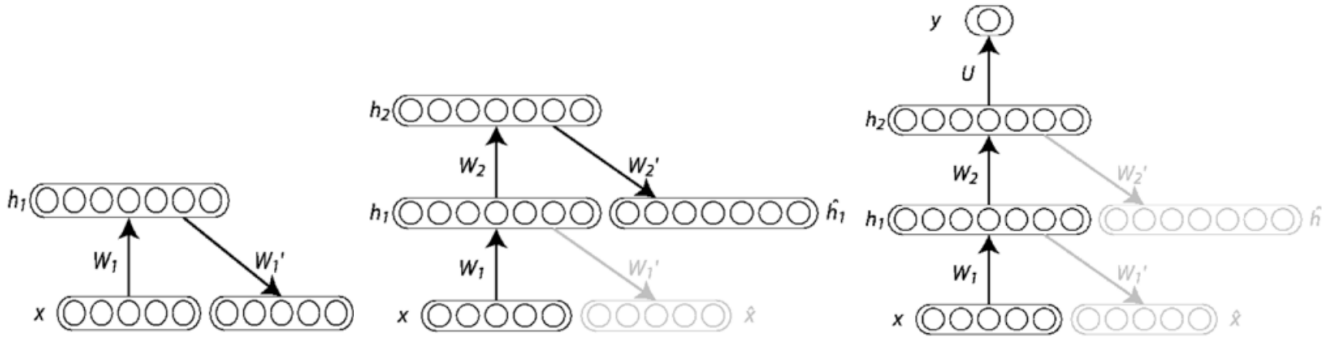
- Greedy Layer-wise Pre-training for Deep Neural Network

Stacking Restricted Boltzmann Machines (RBM)

➪ Deep Belief Network (DBN) [Hinton et al. 2006]



Stacking basic Auto-Encoders [Bengio et al. 2007]



Experimental Results

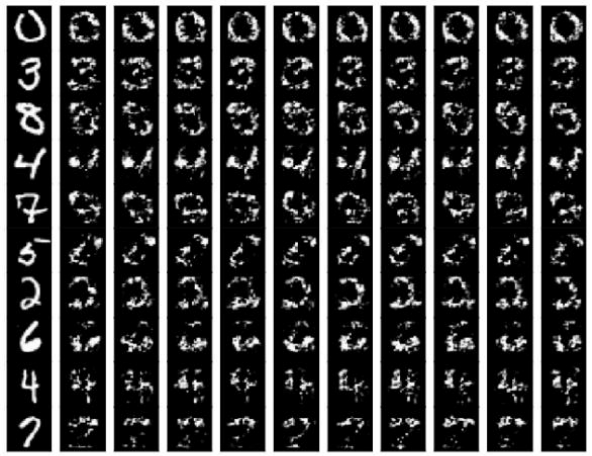
- Classification Error on Various Dataset

Data Set	SVM_{rbf}	DBN-1	SAE-3	DBN-3	SDAE-3 (ν)
MNIST	**1.40**±0.23	**1.21**±0.21	**1.40**±0.23	**1.24**±0.22	**1.28**±0.22 (25%)
basic	**3.03**±0.15	3.94±0.17	3.46±0.16	3.11±0.15	**2.84**±0.15 (10%)
rot	11.11±0.28	14.69±0.31	10.30±0.27	10.30±0.27	**9.53**±0.26 (25%)
bg-rand	14.58±0.31	9.80±0.26	11.28±0.28	**6.73**±0.22	10.30±0.27 (40%)
bg-img	22.61±0.37	**16.15**±0.32	23.00±0.37	**16.31**±0.32	**16.68**±0.33 (25%)
bg-img-rot	55.18±0.44	52.21±0.44	51.93±0.44	47.39±0.44	**43.76**±0.43 (25%)
rect	**2.15**±0.13	4.71±0.19	2.41±0.13	2.60±0.14	**1.99**±0.12 (10%)
rect-img	24.04±0.37	23.69±0.37	24.05±0.37	22.50±0.37	**21.59**±0.36 (25%)
convex	19.13±0.34	19.92±0.35	**18.41**±0.34	**18.63**±0.34	**19.06**±0.34 (10%)
tzanetakis	**14.41**±2.18	18.07±1.31	**16.15**±1.95	18.38±1.64	**16.02**±1.04(0.05)

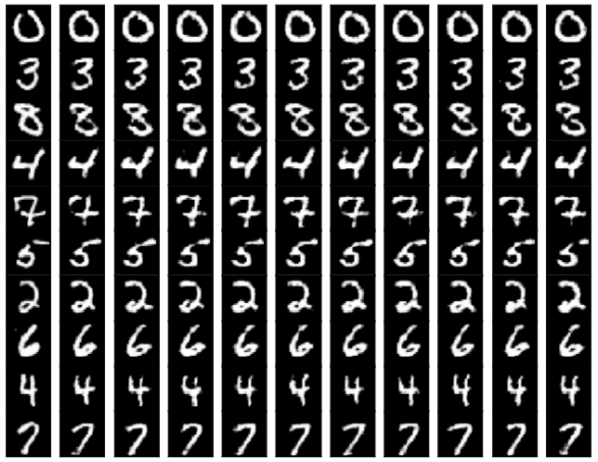
P. Vincent et al. (2010)

Experimental Results

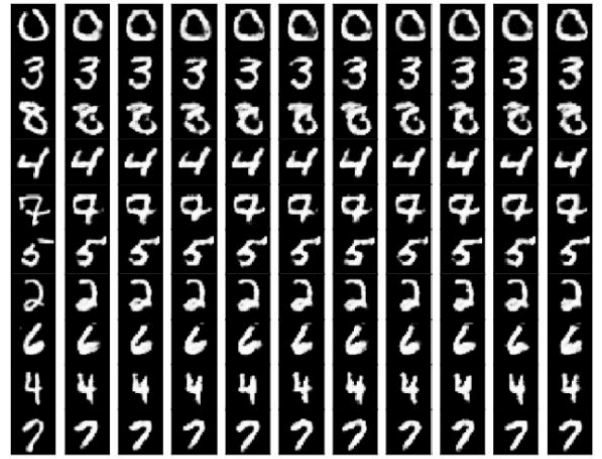
- Variability of Samples Generated with 3 Hidden Layer Networks



(a) SAE



(b) SDAE

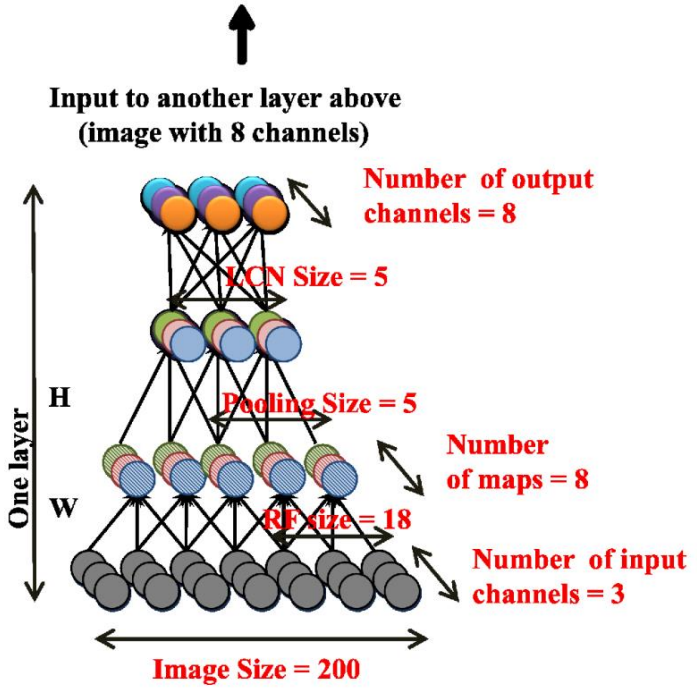


(c) DBN

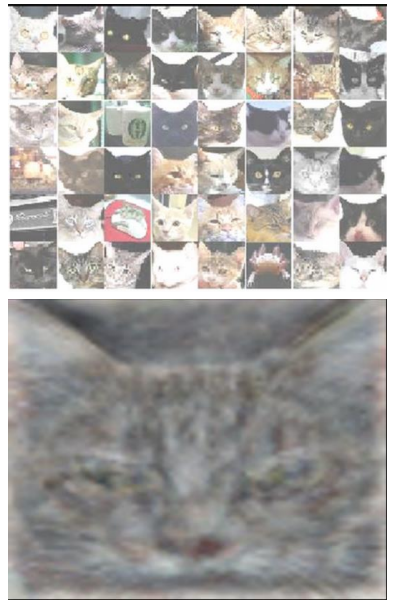
P. Vincent et al. (2010)

Deep Sparse Auto Encoder

- Fully-unsupervised Learning for 'Concept' Learning
  - Using locally connected layers, pooling, local contrast normalization, simulating high-level class-specific neuron is possible.



$$\underset{W_1,W_2}{\text{minimize}} \sum_{i=1}^{m} \left(\left\| W_2 W_1^T x^{(i)} - x^{(i)} \right\|_2^2 + \lambda \sum_{j=1}^{k} \sqrt{\epsilon + H_j (W_1^T x^{(i)})^2} \right).$$



Q. Le et al, 2012

Lab

Lab

- Auto-encoder
  - https://github.com/KyuhwanJung/tensorflow_tutorial/blob/master/Autoencoder.ipynb
- Denoising Auto-encoder
  - https://github.com/KyuhwanJung/tensorflow_tutorial/blob/master/Denoising_autoencoder.ipynb
- Variational Auto-encoder
  - https://github.com/KyuhwanJung/tensorflow_tutorial/blob/master/Variational_autoencoder.ipynb