Introduction to Neural Networks and Deep Learning

Heung-Il Suk

hisuk@korea.ac.kr

http://www.ku-milab.org

MiL Machine Intelligence Laboratory

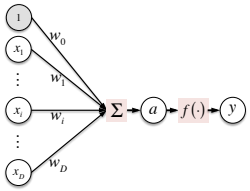
Department of Brain and Cognitive Engineering,
Korea University

August 18, 2016

Contents

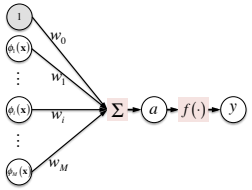
Introduction

$$y(\mathbf{x}, \mathbf{w}) = f\left(\underbrace{\sum_{j=0}^{D} w_j x_j}_{a}\right)$$



- $f(\cdot)$: continuous output
  - (regression) identity function; (classification) sigmoidal function

$$y(\mathbf{x}, \mathbf{w}) = f\left(\underbrace{\sum_{j=0}^{M} w_j \phi_j(\mathbf{x})}_{a}\right)$$



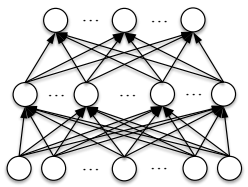
$\{\phi_j\}_{j=1}^{M}$: basis functions

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=0}^{M} w_j \phi_j(\mathbf{x})\right)$$

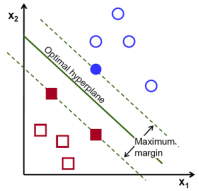
For application to large-scale problems,
it is necessary *to adapt the basis functions to the data*

⇓

$h(W\mathbf{x})$



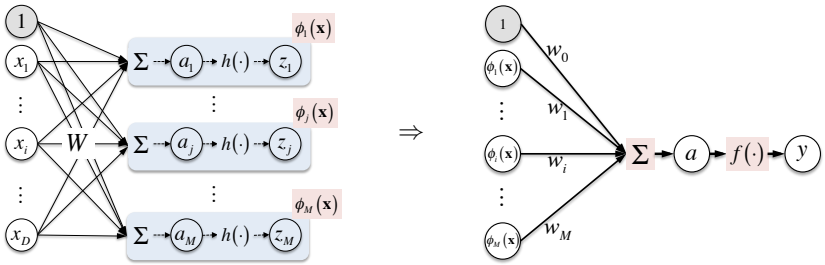
Neural Networks



$K(\mathbf{x}, \mathbf{x}')$

Support Vector Machines

Feed-Forward Neural Networks



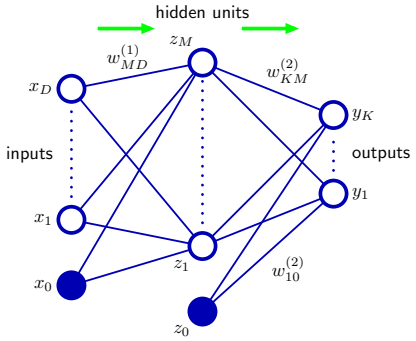
$h(\cdot)$: nonlinear function

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=0}^{M} w_j \phi_j(\mathbf{x})\right)$$

- Basis functions $\phi_j(\mathbf{x})$: a parametric form
- These parameters to be adjusted along with the coefficients $\{w_j\}$

- In NN, each basis function itself is a nonlinear function of a linear combination of the inputs
  - Coefficients in the linear combination are adaptive parameters

Feed-Forward Network



$$\boldsymbol{\phi}(\mathbf{x}) = h\left(W^{(1)}\mathbf{x}\right) = \mathbf{z}$$

$$W^{(1)} = \left[w_{ji}^{(1)}\right] \in \mathbb{R}^{M\times(D+1)}$$

$$\mathbf{y} = f\left(W^{(2)}\mathbf{z}\right)$$

$$W^{(2)} = \left[w_{kj}^{(2)}\right] \in \mathbb{R}^{K\times(M+1)}$$

Two-layer neural network

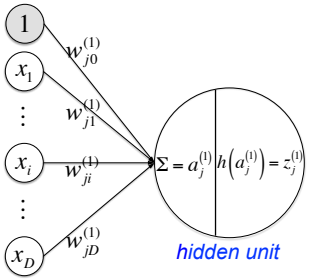
- Pre-activations: $\left\{a_j^{(1)}\right\}_{j=1}^{M}$

$$a_j^{(1)} = \sum_{i=1}^{D} \underbrace{w_{ji}^{(1)}}_{\text{weight}} x_i + \underbrace{w_{j0}^{(1)}}_{\text{bias}}$$

- Outputs of the basis functions ('*hidden units*')

$$z_j^{(1)} = h\left(a_j^{(1)}\right)$$

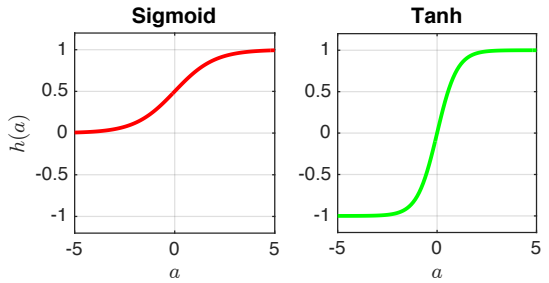
  - $h(\cdot)$: differentiable, nonlinear activation function
  - sigmoidal functions such as 'logistic sigmoid' or 'tanh'
  - or others (discussed in deep learning)



Connection between input and hidden units

$$\text{sigmoid}(a) = \frac{1}{1+\exp[-a]}$$

$$\tanh(a) = \frac{\exp[a]-\exp[-a]}{\exp[a]+\exp[-a]}$$

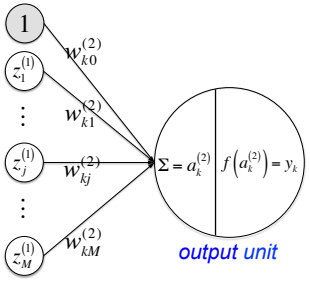


- Pre-activations: $\left\{a_k^{(2)}\right\}_{k=1}^{K}$

$$a_k^{(2)} = \sum_{j=1}^{M} \underbrace{w_{kj}^{(2)}}_{\text{weight}} z_j^{(1)} + \underbrace{w_{k0}^{(2)}}_{\text{bias}}$$

- Outputs of the basis functions ('*output units*')

$$y_k = f\left(a_k^{(2)}\right)$$



Connection between hidden and output units

- Choice of the output activation function $f(\cdot)$ is determined by the nature of the data and the assumed distribution of target variables
  - Regression: identity

$$y_k = a_k^{(2)}$$

  - Binary classification: logistic sigmoid

$$y_k = \text{sigmoid}\left(a_k^{(2)}\right) = \frac{1}{1+\exp\left(-a_k^{(2)}\right)}$$

  - Multiclass classification: softmax

$$y_k = \text{softmax}\left(a_k^{(2)}\right) = \frac{\exp\left(a_k^{(2)}\right)}{\sum_{l=1}^{K}\exp\left(a_l^{(2)}\right)}$$

$$y_k(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

equivalently

$$y_k(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=0}^{M} w_{kj}^{(2)} \underbrace{h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)}_{\phi_j(\mathbf{x})}\right)$$

where $\mathbf{w} = \left[vec\left(W^{(1)}\right); vec\left(W^{(2)}\right)\right]$

Network Training

Overview

- Error function
- Parameter learning: (stochastic) gradient-descent method
- Gradient evaluation: backpropagation [Rumelhart *et al.*, 1986]

Error Functions

Given a set of input vectors $\{\mathbf{x}_n\}_{n=1}^N$ and target vectors $\{\mathbf{t}_n\}_{n=1}^N$

- Regression
  - Identity activation function
  - Sum-of-squares error

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n\|^2$$

- Binary classification
  - Logistic sigmoid activation function
  - Cross-entropy error function ▸ MLE

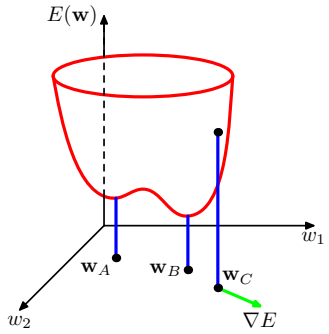
$$E(\mathbf{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln(1 - t_n)\}$$

- Multiclass classification
  - Softmax function
  - Cross-entropy error function ▸ MLE

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{kn} \ln y_{kn}$$

Parameter Optimization

- Finding a weight vector $\mathbf{w}$ that minimizes the error function $E(\mathbf{w})$

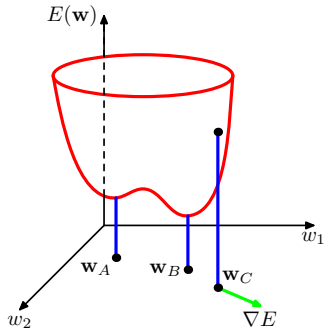


- Geometrical picture of error function
  - a surface sitting over the weight space
- Small step from $\mathbf{w}$ to $\mathbf{w}+\Delta\mathbf{w}$ leads to change in error function

$$\Delta E \approx \Delta\mathbf{w}^{\top}\nabla E(\mathbf{w})$$

  - At any point $\mathbf{w}_C$, the local gradient of the error surface is given by the vector

$$\nabla E = \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \cdots, \frac{\partial E}{\partial w_d},\right]^{\top}$$



- ∇E: the direction of greatest rate of increase of the error function $E(\mathbf{w})$
- To reduce the error, make a small step in the direction of $-\nabla E(\mathbf{w})$
- Points at which the gradient vanishes: stationary points
  - minima, maxima, saddle points

- There is no analytical solution
- Resort to iterative numerical procedures
- Choose some initial value $\mathbf{w}^{(0)}$ and then update it

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)}$$

- Different algorithms involve different choices for $\Delta\mathbf{w}^{(\tau)}$.
- Weight vector update $\Delta\mathbf{w}^{(\tau)}$ is usually based on gradient $\nabla E(\mathbf{w})$ evaluated at the weight vector $\mathbf{w}^{(\tau)}$

Gradient Descent Optimization

- Simplest approach to using gradient information
- Take a small step in the direction of the negative gradient

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta\nabla E\left(\mathbf{w}^{(\tau)}\right)$$

  - η: learning rate (*e.g.*, 0.001)

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

Different versions of gradient descent optimization

- Batch: the entire training set to be processes to evaluate ∇E (a.k.a., *gradient descent* or *steepest descent*)
- On-line: one sample at a time (a.k.a., *stochastic gradient descent* or *sequential gradient descent*)
  - Possibility of escaping from local minima
    - Stationary point w.r.t. the error function for the whole data set will generally not be a stationary point for each data point individually
- Mini-batch stochastic gradient descent: a small set of samples at a time
  - good tradeoff between batch and on-line
  - approximation of the gradient of the loss function

Gradient Evaluation

To find an efficient technique for evaluating the gradient of an error function $E(\mathbf{w})$ for a feed-forward neural network

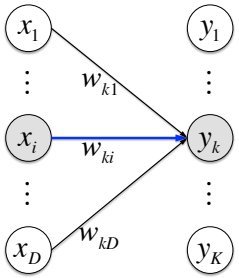
- *Local message passing* scheme
- alternative information passing: forwards and backwards through the network

'Error backpropagation' or simply 'Backprop'

Simple Linear Model

$$\begin{aligned} y_k &= \sum_i w_{ki} x_i \\ y_{nk} &= y_k(\mathbf{x}_n, \mathbf{w}) \\ E_n &= \frac{1}{2}\sum_k (y_{nk} - t_{nk})^2 \end{aligned}$$

$$\frac{\partial E_n}{\partial w_{ki}} = \underbrace{(y_{nk} - t_{nk})}_{\text{error}} x_{ni}$$



- 'Local' computation
  - an 'error signal' $(y_{nk} - t_{nk})$ associated with the output end of the link w_{ki}
  - the variable x_{ni} associated with the input end of the link

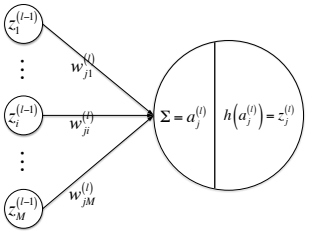
Forward Propagation in a General Feed-Forward Network

1. Each unit computes a weighted sum of its inputs

$$a_j^{(l)} = \sum_i w_{ji}^{(l)} z_i^{(l-1)}$$

2. Transformation by a nonlinear activation function $h(\cdot)$ to give the activation $z_j^{(l)}$ of unit j

$$z_j^{(l)} = h\left(a_j^{(l)}\right)$$



What is the error in this unit?

Evaluation of the derivative of E_n w.r.t. $w_{ji}^{(l)}$

$$\frac{\partial E_n}{\partial w_{ji}^{(l)}} = \underbrace{\frac{\partial E_n}{\partial a_j^{(l)}}}_{\equiv \delta_j^{(l)}} \underbrace{\frac{\partial a_j^{(l)}}{\partial w_{ji}^{(l)}}}_{z_i^{(l-1)}} \quad \text{(by chain rule)}$$
$$= \delta_j^{(l)} z_i^{(l-1)}$$

$\delta_j^{(l)}$: gradient of the error function w.r.t. the pre-activation $a_j^{(l)}$

- Note that this takes the same form as for the simple linear model
- Need only to calculate the value of $\delta_j^{(l)}$ for each hidden and output unit in the network

- For hidden unit j,

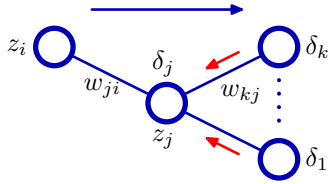
$$\delta_j^{(l)} = \frac{\partial E_n}{\partial a_j^{(l)}} = \sum_k \underbrace{\frac{\partial E_n}{\partial a_k^{(l+1)}}}_{\equiv \delta_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial a_j^{(l)}}$$

  - We are making use of the fact that variations in $a_j^{(l)}$ give rise to variations in the error function only through variations in the variable $a_k^{(l+1)}$

$$\frac{\partial a_k^{(l+1)}}{\partial a_j^{(l)}} = \frac{\partial \left(\sum_m w_{km}^{(l+1)} z_m^{(l)} \right)}{\partial a_j^{(l)}} = \frac{\partial \left(\sum_m w_{km}^{(l+1)} h\left(a_m^{(l)} \right) \right)}{\partial a_j^{(l)}} = h'\left(a_j^{(l)} \right) w_{kj}^{(l+1)}$$

$$\text{or } \frac{\partial a_k^{(l+1)}}{\partial a_j^{(l)}} = \frac{\partial a_k^{(l+1)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial a_j^{(l)}} = w_{kj}^{(l+1)} h'\left(a_j^{(l)} \right)$$

$$\delta_j^{(l)} = h'\left(a_j^{(l)} \right) \sum_k w_{kj}^{(l+1)} \delta_k^{(l+1)}$$



Blue arrow: information flow during forward propagation
Red arrow: backward propagation of error information

The value of δ for a particular hidden unit can be obtained by propagating the δ's backwards from units higher up in the network

- Forward propagation

$$a_j^{(l)} = \sum_i w_{ji}^{(l)} z_i^{(l-1)}$$
$$z_j^{(l)} = h\left(a_j^{(l)} \right)$$

- Backward propagation

$$\delta_j^{(l)} = h'\left(a_j^{(l)} \right) \sum_k w_{kj}^{(l+1)} \delta_k^{(l+1)}$$

Forward propagation

$$\mathbf{x} \xrightarrow{W^{(1)}} \mathbf{a}^{(1)} \rightarrow \mathbf{z}^{(1)} \xrightarrow{W^{(2)}} \mathbf{a}^{(2)} \rightarrow \mathbf{z}^{(2)} = \mathbf{y}$$

Backward propagation

$$\frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{x}} \leftarrow \underbrace{\frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{a}^{(1)}} \odot \frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{z}^{(1)}}}_{\boldsymbol{\delta}^{(1)}} \leftarrow \underbrace{\frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{a}^{(2)}} \odot \frac{\partial E}{\partial \mathbf{z}^{(2)}}}_{\boldsymbol{\delta}^{(2)}}$$

$$\frac{\partial \mathbf{a}^{(1)}}{\partial W^{(1)}} \qquad \frac{\partial \mathbf{a}^{(2)}}{\partial W^{(2)}}$$

$$\frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{a}^{(1)}} = h'\left(\mathbf{a}^{(1)}\right)$$

$$\frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{z}^{(1)}} = W^{(1)}$$

$$\frac{\partial \mathbf{a}^{(1)}}{\partial W^{(1)}} = \mathbf{z}^{(0)} = \mathbf{x}$$

$$\frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{a}^{(2)}} = h'\left(\mathbf{a}^{(2)}\right)$$

$$\frac{\partial \mathbf{a}^{(2)}}{\partial W^{(2)}} = \mathbf{z}^{(1)}$$

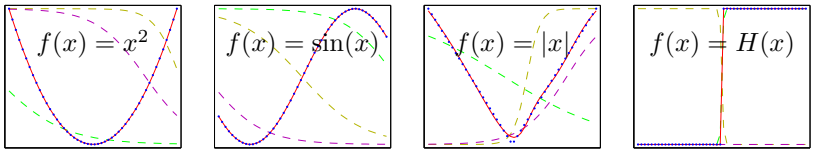
$$\frac{\partial E}{\partial \mathbf{z}^{(2)}} \equiv \boldsymbol{\delta}^{(2)} = f'\left(\mathbf{a}^{(2)}\right) \odot (\mathbf{y} - \mathbf{t})$$

$$f'(\mathbf{a}) = \begin{cases} 1 & \text{identity} \\ f(\mathbf{a})\left(1 - f(\mathbf{a})\right) & \text{logistic sigmoid} \\ 1 - f(\mathbf{a})^2 & \tanh \\ \vdots & \vdots \end{cases}$$

Deep Learning

Universal Approximate Theorem [Hornik, 1991]

A feed-forward network with a single hidden layer containing a finite number of units can approximate any continuous function (under mild assumptions on the activation function).



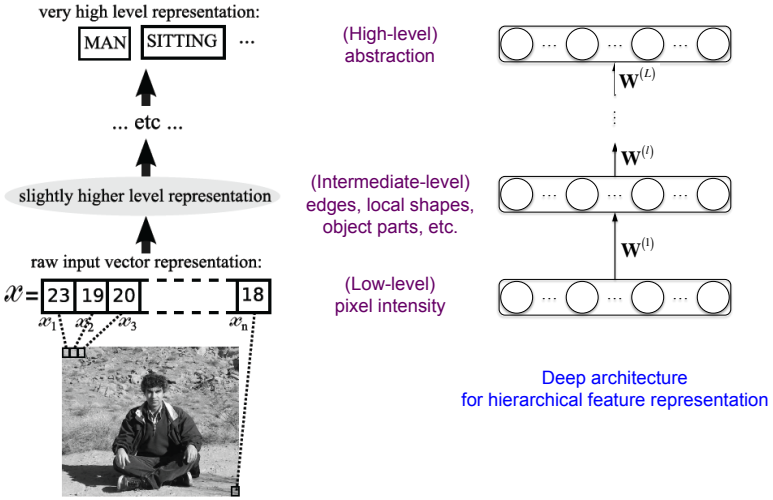
- Blue dots: 50 data points uniformly $(-1, 1)$; 3 hidden units (tanh), linear output units
- Network output (red curve), outputs of thee hidden units

Deep Neural Networks

- Possible to approximate complex functions to the same accuracy using a deeper network with much fewer total units [Bengio, 2009]
- Smaller number of parameters, requiring a smaller dataset to train [Schwarz *et al.*, 1978]
- Hierarchical feature representation (fine-to-abstract)

$$y_k = f\left(\sum_l W_{kl}^{(L)} \underbrace{h\left(\sum_m W_{ml}^{(L-1)} h\left(\cdots h\left(\sum_i W_{ij}^{(1)} x_i\right)\right)\right)}_{\phi_l(\mathbf{x})}\right)$$

Goal of deep architectures: *to learn feature hierarchies*

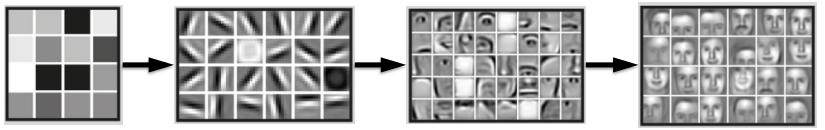


Deep architecture
for hierarchical feature representation

[Bengio, 2009]

Let the computer learn the feature representations from data autonomously!!!

Deep-learning neural networks use layers of increasingly complex rules to categorize complicated shapes such as faces.



Layer 1: The computer identifies pixels of light and dark.

Layer 2: The computer learns to identify edges and simple shapes.

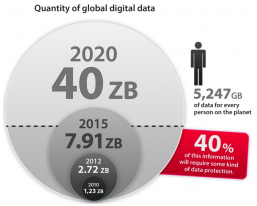
Layer 3: The computer learns to identify more complex shapes and objects.

Layer 4: The computer learns which shapes and objects can be used to define a human face.

(Modified, N. Jones "The Learning Machines," Nature, 2014)

Difficulties in Deep Learning

1. Lack of training samples
   - Huge amount of data available



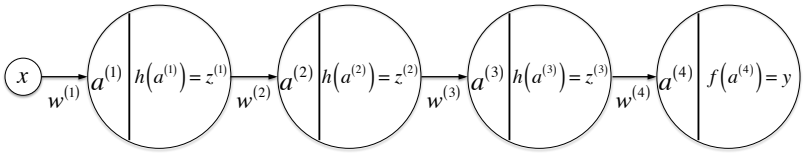
2. Lack of computational power in HW
   - Multi-core CPUs
   - Graphics Processing Unit (GPU)

Image sources: (top) http://www.datacenterjournal.com/birth-death-big-data/

(bottom) https://gigaom.com/2014/09/08/nvidia-stakes-its-claim-in-deep-learning-by-making-its-gpus-easier-to-program/

3. Vanishing Gradient Problem

- Backpropagation becomes ineffective due to vanishing gradients after repeated multiplication.
- The gradient tends to get smaller as we propagate backward through the hidden layers.
- Units in the earlier layers learn much more slowly then units in the later laters.



(Recep.) $$\frac{\partial E_n}{\partial w_{ji}^{(l)}} = \underbrace{\frac{\partial E_n}{\partial a_j^{(l)}}}_{\equiv \delta_j^{(l)}} \underbrace{\frac{\partial a_j^{(l)}}{\partial w_{ji}^{(l)}}}_{z_i^{(l-1)}} = \left\{ h'\left(a_j^{(l)}\right) \sum_k w_{kj}^{(l+1)} \delta_k^{(l+1)} \right\} z_i^{(l-1)}$$

$$\frac{\partial E_n}{\partial w^{(1)}} = \overbrace{x\, h'\left(a^{(1)}\right) w^{(2)} \underbrace{h'\left(a^{(2)}\right) w^{(3)} \overbrace{h'\left(a^{(3)}\right) w^{(4)} \underbrace{f'\left(a^{(4)}\right)}_{\delta^{(4)} = \frac{\partial E_n}{\partial a^{(4)}}} (y_n - t_n)}^{\delta^{(3)}}}_{\delta^{(2)}}}^{\delta^{(1)}}$$

In a general feed-forward neural network: multiple units in each layer

$$\boldsymbol{\delta}^{(l)} = H'\left(\mathbf{a}^{(l)}\right)\left(\mathbf{W}^{(l+1)}\right)^\top H'\left(\mathbf{a}^{(l-1)}\right)\left(\mathbf{W}^{(l+2)}\right)^\top \cdots H'\left(\mathbf{a}^{(L)}\right) \frac{\partial E_n}{\partial \mathbf{a}^{(L)}}$$

where $$H'\left(\mathbf{a}^{(l)}\right) = \begin{bmatrix} h'\left(a_1^{(l)}\right) & & 0 \\ & \ddots & \\ & & h'\left(a_M^{(l)}\right) \end{bmatrix}$$

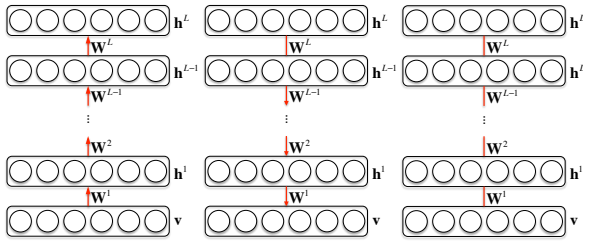
- In general, the weights are initialized by using a Gaussian with mean 0 and standard deviation 1.

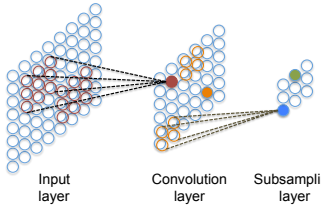
$$\left| w_j^{(l)} \right| < 1$$

Derivatives | Empirical Results | Weights Initialization

Deep Models

- Stacked Auto-Encoder (SAE)
- Deep Belief Network (DBN)
- Deep Boltzmann Machine (DBM)
- Convolutional NN (CNN)
- Recurrent NN (RNN)





Greedy Layer-wise Pre-training [Hinton and Salakhutdinov, 2006]

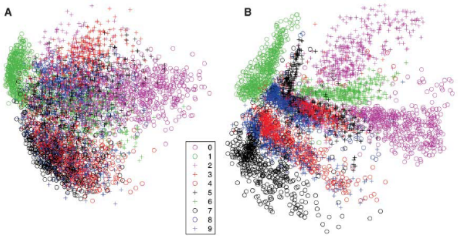
Reducing the Dimensionality of Data with Neural Networks

Science AAAS

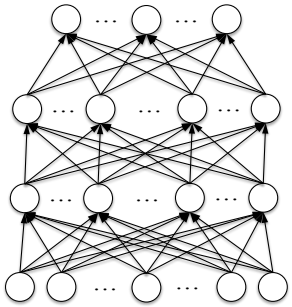
G. E. Hinton* and R. R. Salakhutdinov

High-dimensional data can be converted to low-dimensional codes by training a multilayer neural network with a small central layer to reconstruct high-dimensional input vectors. Gradient descent can be used for fine-tuning the weights in such "autoencoder" networks, but this works well only if the initial weights are close to a good solution. We describe an effective way of initializing the weights that allows deep autoencoder networks to learn low-dimensional codes that work much better than principal components analysis as a tool to reduce the dimensionality of data.

Fig. 3. (A) The two-dimensional codes for 500 digits of each class produced by taking the first two principal components of all 60,000 training images. (B) The two-dimensional codes found by a 784-1000-500-250-2 autoencoder. For an alternative visualization, see (8).

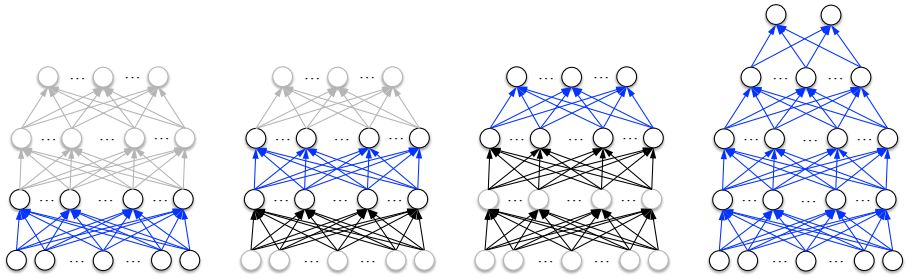


Stacked Auto-Encoder (SAE)



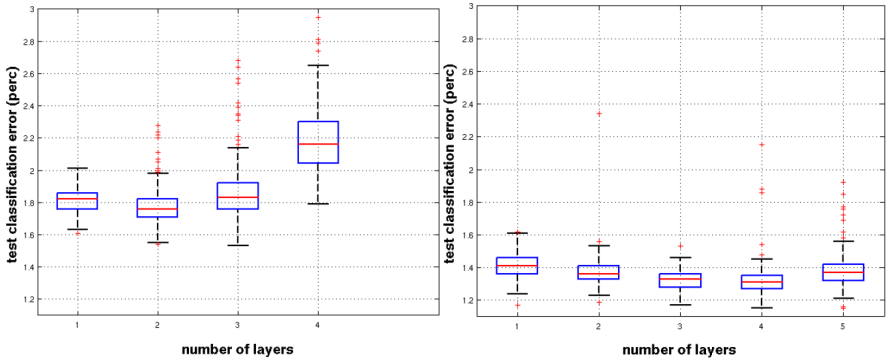
- Train layer 1 in an unsupervised manner
- Train layer 2 while keeping layer 1 fixed in an unsupervised manner
- Train layer 3 while keeping layer 1 & 2 fixed in an unsupervised manner

⋆ Fine-tune the whole network in a supervised manner



(Pre-train 1st hidden layer) (Pre-train 2nd hidden layer) (Pre-train 3rd hidden layer) (Fine-tuning whole network)

Effects of Pre-Training: Empirical Results



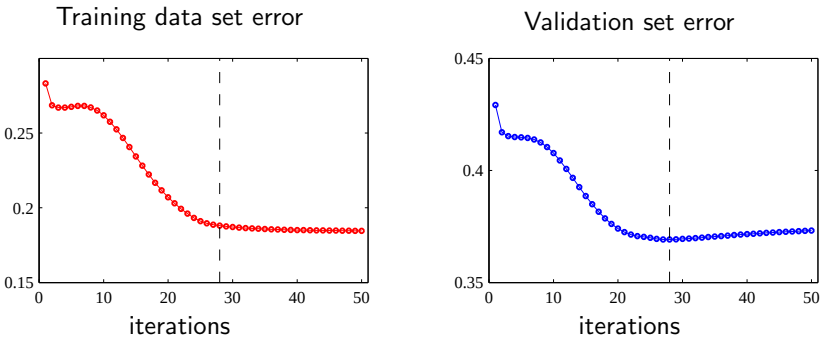
[Erhan *et al.*, 2009]

Tricks for Avoiding Overfitting

Algorithms for Avoiding Overfitting

- Early stopping
- Regularization (ℓ_1-, $\ell_{1/2}$-, ℓ_2-norm)
- Rectified Linear Unit (ReLU) [Nair and Hinton, 2010]
- Denoising [Vincent *et al.*, 2008]
- Dropout [Hinton *et al.*, 2012]
- Dropconnect [Wan *et al.*, 2013]
- Batch normalization [Ioffe and Szegedy, 2015]

Early Stopping



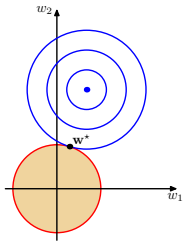
Halting training before a minimum of the training error has been reached represents a way of limiting the effective network complexity.

Weight Decay: ℓ_2-norm Regularization

Choose a relatively large M and control complexity by the addition of a regularization term to the error function

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^\top\mathbf{w}$$

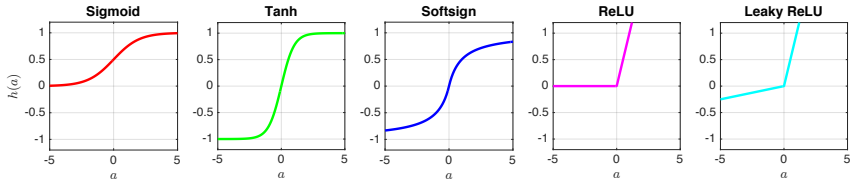
▸ Bayesian Interpretation



- λ: regularization coefficient
- Can be interpreted as the negative logarithm of a zero-mean Gaussian prior distribution over the weight vector $\mathbf{w}$

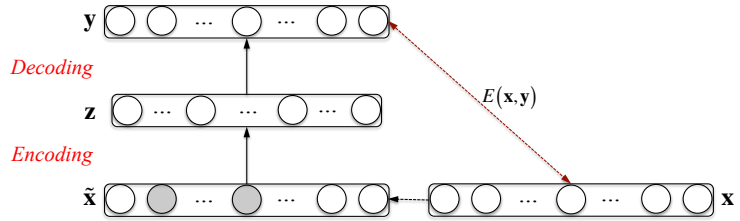
Activation Functions

- Logistic sigmoid: $h(a) = 1/(1+\exp[-a])$
- Tanh: $h(a) = (\exp[a]-\exp[-a])/(\exp[a]+\exp[-a])$
- Softsign [Bergstra *et al.*, 2009]: $h(a) = 1/(1+|a|)$
- ReLU [Nair & Hinton *et al.*, 2010]: $h(a) = \max(0, a)$
- Leaky ReLU [Maas *et al.*, 2013]: $h(a) = \max(\kappa a, a)$, $0 < \kappa < 1$
- Parametric ReLU [He *et al.*, 2015]: $h(a) = \max(\kappa^* a, a)$



▸ Non-smooth & Unbounded

Denoising Auto-Encoder [Vincent et al., 2008]

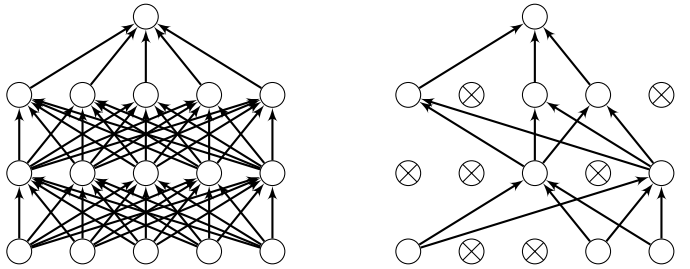


- In order to force the hidden layer to discover more robust features and prevent it from simply learning the identity, train the auto-encoder to reconstruct the input from a corrupted version of it.
- $\tilde{\mathbf{x}}$: corrupted input $\mathbf{x}$ by adding noise
- Train parameters so that the output $\mathbf{y}$ to be equal to $\mathbf{x}$

▸ Probabilistic View

Dropout [Hinton et al., 2012]

- Preventing co-adaptation of neurons: a neuron cannot rely on the presence of particular other neurons [Krizhevsky *et al.*, 2012]
- Randomly deactivate a set (*e.g.*, 50%) of the neurons in a network on each training iteration

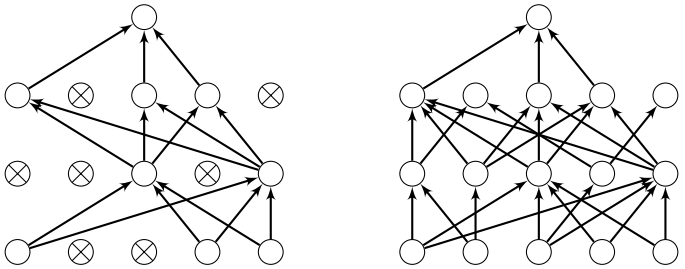


(a) Standard DNN (b) After applying dropout

▸ Probabilistic View ▸ Empirical Results ▸ Simple Interpretation ▸ Full Bayesian Interpretation

Dropconnect [Wan et al., 2013]

- Randomly remove a set (*e.g.*, 50%) of the weights and biases within the network



(a) After applying dropout (b) After applying dropconnect

▸ Probabilistic View ▸ Empirical Results

Batch Normalization [Ioffe and Szegedy, 2015]

- Internal covariate shift: change in the distribution of network activations due to the change in network parameters during training makes the training time slow
- Performing normalization for each mini-batch and backpropagating the gradients through the normalization parameters (*i.e.*, scale and shift)
- For each unit in a layer l, their value is normalized

$$\hat{x}_k^{(l)} = \frac{x_k^{(l)} - \mathbb{E}[x_k^{(l)}]}{\sqrt{\mathsf{Var}[x_k^{(l)}]}}$$

where $k = 1, \ldots, F^{(l)}$ and $F^{(l)}$: number of units in the layer l

- A pair of learnable parameters $\gamma_k^{(l)}$ and $\beta_k^{(l)}$ are then introduced to scale and shift the normalized values to restore the representation power of the network

$$y_k^{(l)} = \gamma_k^{(l)} \hat{x}_k^{(l)} + \beta_k^{(l)}$$

▸ Empirical Results

Suggested Readings

- C. Bishop, *Pattern Recognition and Machine Learning, (Ch. 5, 10, 11)*, 2006
- Bengio, *Learning Deep Architectures for AI*, Foundations and Trends in Machine Learning, 2009
- Glorot and Bengio, "Understanding the Difficulty of Training Deep Feedforward Neural Networks," AISTATS, 2010
- Larochelle *et al.*, "Exploring Strategies for Training Deep Neural Networks," ICML, 2009
- Srivastava *et al.*, "Dropout: A simple Way to Prevent Neural Networks from Overfitting," JMLR, 2014
- Gal and Ghahramani, "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning," ICML, 2016
- Mohamed, "A Statistical View of Deep Learning," TR, 2015
- Ioffe and Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," ICML, 2015
- Goodfellow *et al.*, *Deep Learning*, (in press); available online at `http://www.deeplearningbook.org/`

Thank you for your attention!!!

(Q & A)

hisuk (AT) korea.ac.kr

http://www.ku-milab.org

Supplementary

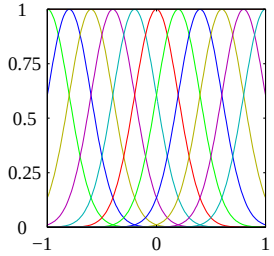
Basis Functions

- Polynomial

$$\phi_j(x) = x^j$$

- (Gaussian) Radial Basis Functions (RBF)

$$\phi_j(x) = \exp\left\{\frac{(x-\mu_j)^2}{2s^2}\right\}$$



  - μ_j: governing the locations of the basis functions in the input space
    - Can be arbitrary points in dataset
  - s: governing the spatial scale
    - Can be chosen from the dataset, *e.g.*, average variance

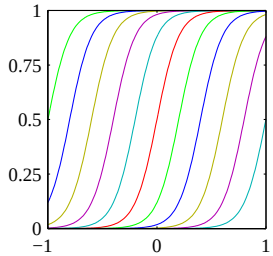
- Sigmoidal Basis Function

$$\phi_j(x) = \text{sigmoid}\left(\frac{x-\mu_j}{s}\right)$$

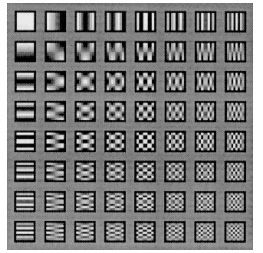
$$\text{sigmoid}(a) = \frac{1}{1+\exp(-a)}$$

- Equivalently, 'tanh' function, which is related to the logistic sigmoid

$$\tanh(a) = 2\sigma(a) - 1$$

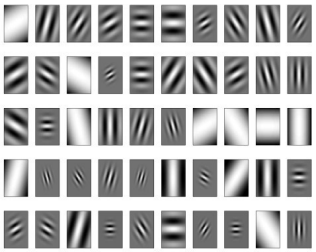


- Fourier
  - Expansion in sinusoidal functions
  - Infinite spatial extent



e.g., DCT Fourier basis

- Wavelet
  - Localized in both space and frequency
  - Useful for lattices such as images and time series



e.g., Gabor wavelet basis

Go Back

(Two-Class) Logistic Regression

For a dataset $\{\phi_n = \phi(\mathbf{x}_n), t_n\}$, where $t_n \in \{0,1\}$ with $n = 1,\ldots,N$

- Likelihood function

$$p(\mathbf{t}|\mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n}\{1-y_n\}^{1-t_n}$$

  - $y_n = P(C_1|\phi(\mathbf{x}_n)) = \text{sigmoid}(a_n)$, where $a_n = \mathbf{w}^\top\phi(\mathbf{x}_n)$
  - $\mathbf{t} = (t_1,\ldots,t_N)^\top$

- Taking negative logarithm → *cross-entropy error function*

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

Error Functions

Multiclass Logistic Regression

Work with a softmax function instead of logistic sigmoid

$$P(C_k|\mathbf{x}) = y_k(\phi) = \frac{\exp(a_k)}{\sum_j \exp(a_j)} \quad \text{where } a_k = \mathbf{w}_k^\top\phi$$

- Likelihood function

$$p(\mathbf{T}|\mathbf{w}_1,\ldots,\mathbf{w}_K) = \prod_{n=1}^{N}\prod_{k=1}^{K} P(C_k|\mathbf{x}_n)^{t_{nk}} = \prod_{n=1}^{N}\prod_{k=1}^{K} y_{nk}^{t_{nk}}$$

  - $\mathbf{t}_n \in \{0,1\}^K$: 1-of-$K$ coding scheme
  - $y_{nk} = y_k(\phi_n)$, $\mathbf{T} = [t_{nk}]$: $N \times K$ matrix of target variables

- Taking negative logarithm → *cross-entropy error function*

$$E(\mathbf{w}_1,\ldots,\mathbf{w}_K) = -\ln p(\mathbf{T}|\mathbf{w}_1,\ldots,\mathbf{w}_K) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk}\ln y_{nk}$$

Error Functions

Forward propagation

$$\mathbf{x} \xrightarrow{W^{(1)}} \mathbf{a}^{(1)} \to \mathbf{z}^{(1)} \xrightarrow{W^{(2)}} \cdots \xrightarrow{W^{(L-1)}} \mathbf{a}^{(L-1)} \to \mathbf{z}^{(L-1)} \xrightarrow{W^{(L)}} \mathbf{a}^{(L)} \to \mathbf{z}^{(L)} = \mathbf{y}$$

Backward propagation

$$\boldsymbol{\delta}^{(1)} \quad \boldsymbol{\delta}^{(L-1)} \quad \boldsymbol{\delta}^{(L)}$$

$$\frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{x}} \leftarrow \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{a}^{(1)}} \odot \frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{z}^{(1)}} \leftarrow \cdots \leftarrow \frac{\partial \mathbf{z}^{(L-1)}}{\partial \mathbf{a}^{(L-1)}} \odot \frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{z}^{(L-1)}} \leftarrow \frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{a}^{(L)}} \odot \frac{\partial E}{\partial \mathbf{z}^{(L)}}$$

$$\downarrow \qquad\qquad \downarrow \qquad\qquad \downarrow$$

$$\frac{\partial \mathbf{a}^{(1)}}{\partial W^{(1)}} \quad \cdots \quad \frac{\partial \mathbf{a}^{(L-1)}}{\partial W^{(L-1)}} \quad \frac{\partial \mathbf{a}^{(L)}}{\partial W^{(L)}}$$

$$\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{a}^{(l)}} = h'\left(\mathbf{a}^{(l)}\right)$$

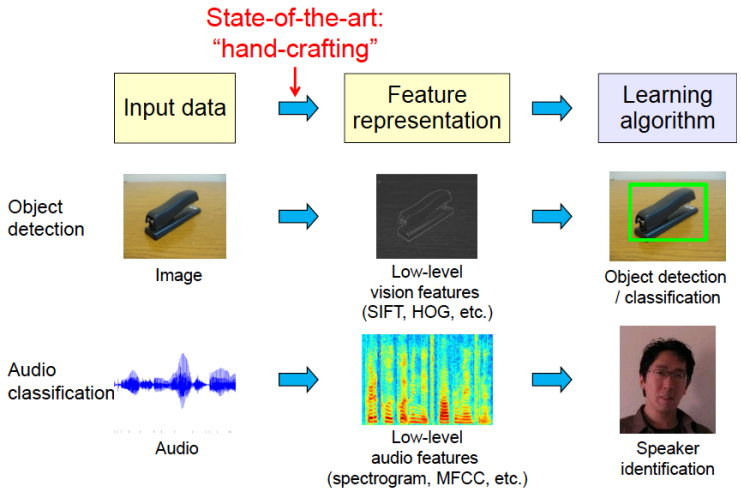
$$\frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l-1)}} = W^{(l)}$$

$$\frac{\partial \mathbf{a}^{(l)}}{\partial W^{(l)}} = \mathbf{z}^{(l-1)}$$

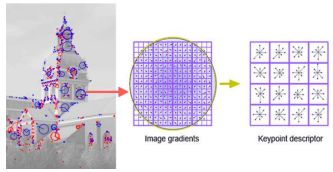
$$\frac{\partial E}{\partial \mathbf{a}^{(L)}} \equiv \boldsymbol{\delta}^{(L)} = f'\left(\mathbf{a}^{(L)}\right) \odot (\mathbf{y} - \mathbf{t})$$

$$f'(\mathbf{a}) = \begin{cases} 1 & \text{identity} \\ f(\mathbf{a})\,(1 - f(\mathbf{a})) & \text{logistic sigmoid} \\ 1 - f(\mathbf{a})^2 & \tanh \\ \vdots & \vdots \end{cases}$$

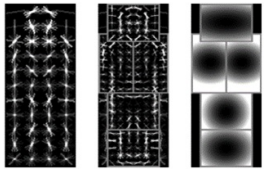
Handcrafted features in different applications



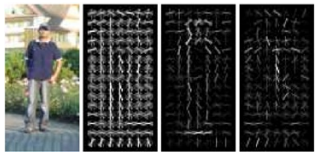
(Image credit: Prof. H. Lee)



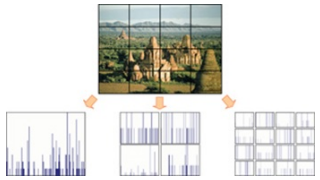
SIFT [Loewe, IJCV04]



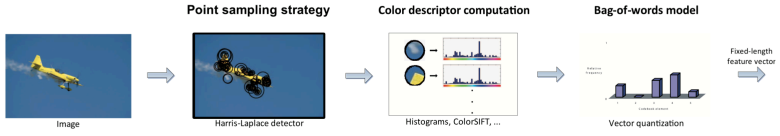
DPM [Felzenszwalb *et al.*, IEEE-TPAMI10]



HOG [Dalal and Triggs, CVPR05]

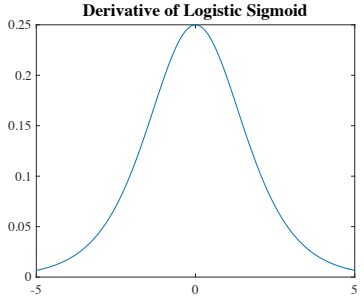


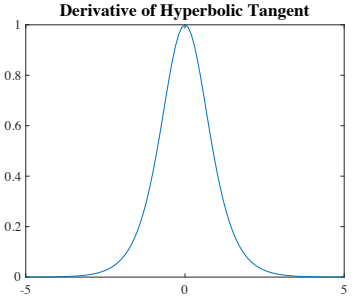
SPM [Lazebnik *et al.*, CVPR06]



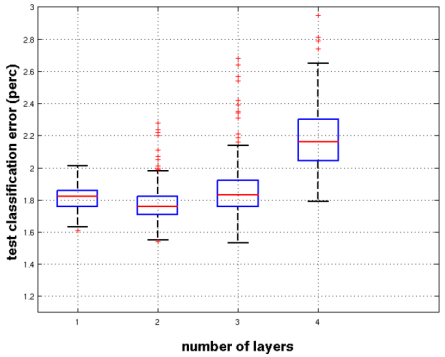
Color Descriptor [Van De Sande *et al.*, IEEE-TPAMI10]

(Modified from J.-B. Huang's slide)





- Empirical Results: Backprop on Deep Neural Networks
  - Digit classification on MNIST
  - 400 different initialization seeds
  - Increasing depth seems to increase the probability of finding poor local minima



[Erhan *et al.*, 2009]

Go Back

Weights Initialization

- Gaussian distribution

$$\mathbf{w} \sim \mathcal{N}(0, 1)$$

- Uniform distribution

$$W^{(l)} \sim U\left[-\frac{1}{\sqrt{n_{l-1}}}, \frac{1}{\sqrt{n_{l-1}}}\right]$$

(n_{l-1}: the size of the layer $l-1$)

- Normalized initialization [Glorot and Bengio, 2010]

$$W^{(l)} \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_{l-1}+n_l}}, \frac{\sqrt{6}}{\sqrt{n_l+n_{l+1}}}\right]$$

Go Back

Details in dropout

- In training: for *each training sample* in a mini-batch

$$\begin{aligned} r_j^{(l)} &\sim \text{Bernoulli}\left(\pi^{(l)}\right) \\ \tilde{\mathbf{z}}^{(l)} &= \mathbf{r}^{(l)} \odot \mathbf{z}^{(l)} \\ a_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \tilde{\mathbf{z}}^{(l)} + b_i^{(l+1)} \\ z_i^{(l+1)} &= f\left(a_i^{(l+1)}\right) \end{aligned}$$

- At test time: scaling the weights
  - Infeasible to explicitly average the predictions from exponentially many thinned models
  - Approximation: to use a single neural net at test time without dropout with the weights scaled-down

$$\mathbf{W}_{\text{test}}^{(l)} = \pi^{(l)} \mathbf{W}^{(l)}$$

Go Back

Interpretation of dropout [Gal and Ghahramani, 2016]

- With n units, a collection of 2^n possible thinned neural networks

$$r_j^{(l)} \sim \text{Bernoulli}\left(\pi^{(l)}\right) \quad \forall j, l$$

- Training a neural network with dropout can be seen as training a collection of 2^ns thinned networks with extensive weight sharing
- Approximating an equally weighted geometric mean of the predictions of an exponential number of learned models that share parameters

$$\mathbf{W}_{\text{test}}^{(l)} = \pi^{(l)} \mathbf{W}^{(l)}$$

Dropout can be seen as a way of doing an equally-weighted averaging of exponentially many models with shared weights.

Go Back