Feedforward Neural Networks

Jian Tang

tangjianpku@gmail.com

HEC MONTRÉAL

Institut des algorithmes d'apprentissage de Montréal MILA

The task

- The goal is to learn a mapping function $y = f(x; \theta)$ (e.g., for classification $f: R^d \rightarrow C$).

Example: image classification

Traditional Machine Learning

Hand-crafted
Feature Extractor

Simple Trainable Classifier
e.g., SVM, LR

Domain experts

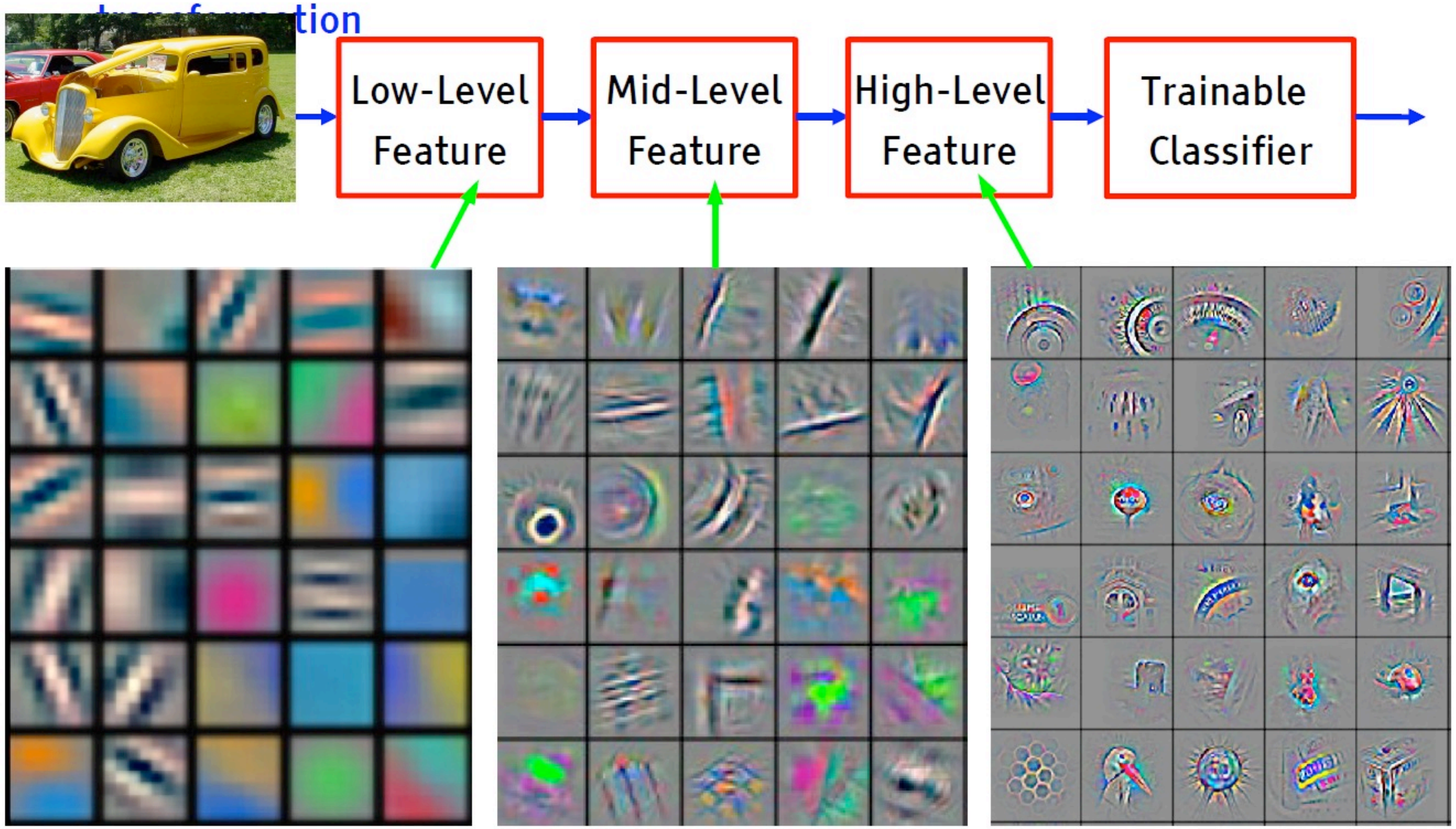
Deep Learning= End-to-end Learning/Feature Learning

Trainable
Feature Extractor

Trainable Classifier
e.g., SVM, LR

Domain expert

Deep Learning=
Learning Hierarchical representations



(Figure from LeCun)

Hierarchical representations with increasing level of abstraction

- Image recognition
  - Pixel -> edge -> texton-> motif -> part -> object
- Speech
  - Sample -> spectral band -> sound -> phone -> word…
- Text
  - Character -> word -> phrase->clause-> sentence ->paragraph-> document

(slides from LeCun)

Outline

- Network Components
  - Neurons (Hidden Units)
  - Output units
  - Cost functions
- Architecture design
  - Capacity of neural networks
- Training
  - Backpropagation with stochastic gradient descent

Neuron: Nonlinear Functions

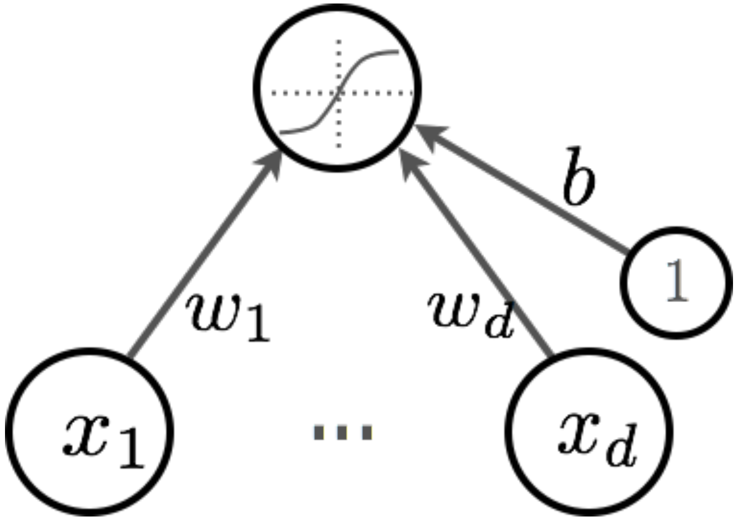
- Input: linear combination:

$$a(\boldsymbol{x}) = b + \sum_i w_i x_i = \boldsymbol{w}^T \boldsymbol{x} + b$$

- Output: nonlinear transformation:

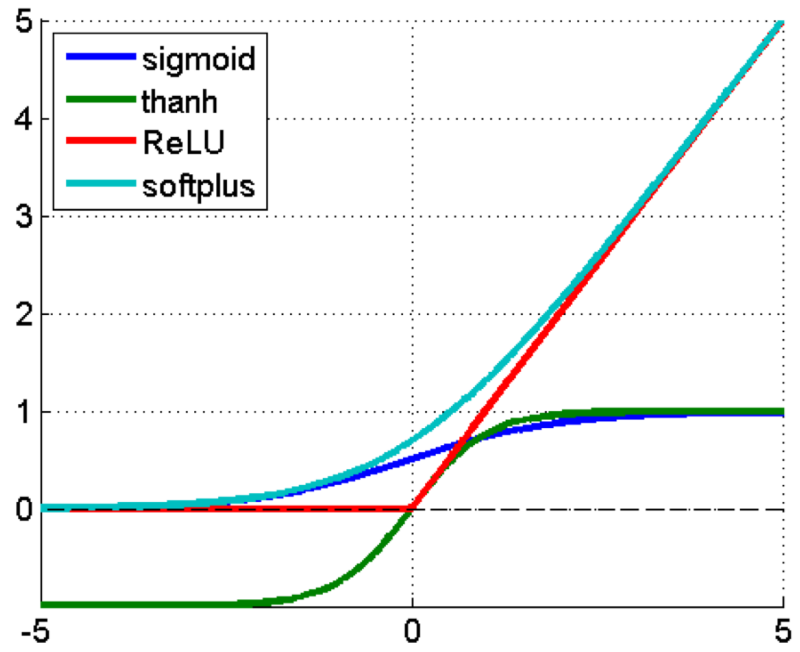
$$h(\boldsymbol{x}) = g(a(\boldsymbol{x})) = g(\boldsymbol{w}^T \boldsymbol{x} + b)$$

- **w**: are the weights (parameters)
- b is the bias term
- g(.) is called the activation function



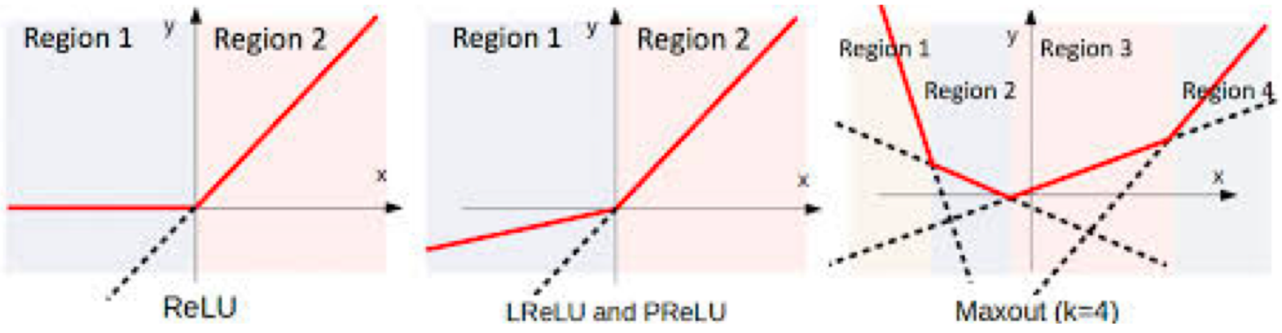
Activation functions/Hidden Units

- Sigmoid function
  - g(x) = 1/(1+exp(-x))
  - Map the input to (0,1)
- Tanh function
  - g(x) = (1-exp(-2x))/(1+exp(-2x))
  - Map the input to (-1,1)
- Rectified linear (ReLU) function
  - g(x) =max(0,x)
  - No upper bounded



Other activation functions

- Leaky ReLU (Maas et al. 2013)
  - $g(x) = \max(0, x) + \alpha \min(0, x)$
  - Fix α to a small value, e.g., 0.01
- Parametric ReLU (He et al. 2015)
  - Treat α as a parameter to learn



- Maxout units (Goodfellow et al. ,2013)
  - Generalize rectified linear units
  - Divide the output units into groups of k values, and output the maximum value in each group
  - Provides a way of learning a piecewise linear function that responds to multiple directions in the input x space.

One Hidden layer Neural Networks

- Input of the hidden layer:

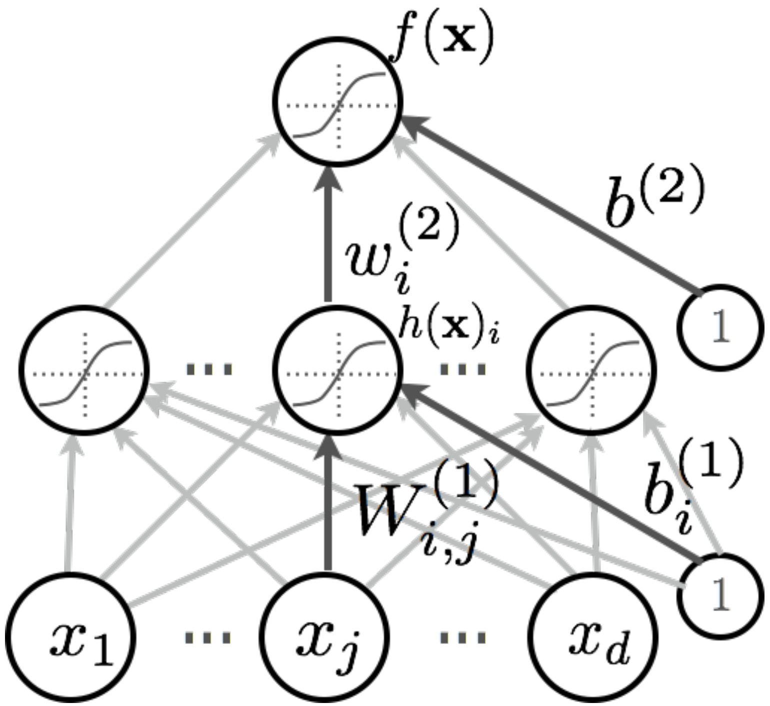
$$a(\boldsymbol{x}) = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$$

- Nonlinear transformation:

$$h(\boldsymbol{x}) = g_1(a(\boldsymbol{x}))$$

- Output layer

$$\mathrm{f}(\boldsymbol{x}) = o\ (h(\boldsymbol{x}))$$



Outline

- Network Components
  - Neurons (Hidden Units)
  - Output units
  - Cost functions
- Architecture design
  - Capacity of neural networks
- Training
  - Backpropagation with stochastic gradient descent

Linear Units for Gaussian Output Distributions

- Given the hidden units **h**, a layer of linear output units produces $\hat{\boldsymbol{y}} = \boldsymbol{W}^T\boldsymbol{h} + \boldsymbol{b}$
- Linear output layers are often used to produce the mean of a conditional Gaussian distribution

$$p(\boldsymbol{y}|\boldsymbol{x}) = \mathrm{N}(\boldsymbol{y}|\hat{\boldsymbol{y}}, \boldsymbol{I})$$

Sigmoid Units for Bernoulli Output Distributions

- Bernoulli output distributions: binary classification
- The goal is to define $p(y = 1|\boldsymbol{x})$, which can be defined as follows:

$$p(y = 1|\boldsymbol{x}) = \sigma(\boldsymbol{w}^T\boldsymbol{h} + b)$$

Softmax Units for Multinomial Output Distributions

- Bernoulli output distributions: multi-class classification
- First, define a linear layer to predict the unnormalized log probabilities of softmax:

$$\boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{h} + \boldsymbol{b}$$

- where $z_i = \log p(y = i|\boldsymbol{x})$. Formally, the softmax function is given by
-

$$p(y = i|\boldsymbol{x}) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Multilayer Neural Networks

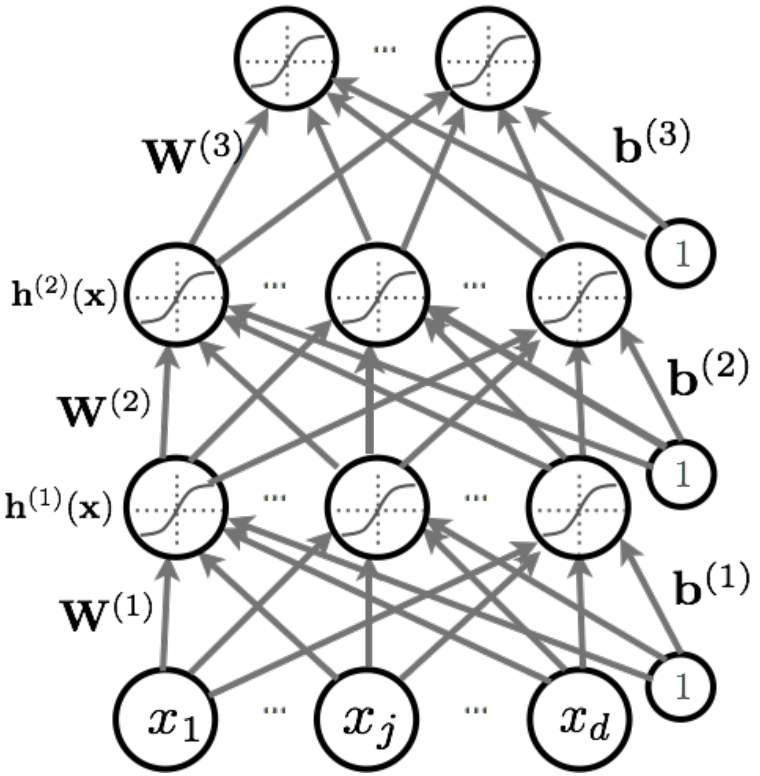
- Neural network with multiple hidden layers
- The output of previous layer as the input of next layer: (k=1…, L)

$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

- Final output layer

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



$(\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x})$

Outline

- Network Components
  - Neurons (Hidden Units)
  - Output units
  - Cost function
- Architecture design
  - Capacity of neural networks
- Training
  - Backpropagation with stochastic gradient descent

Maximum Likelihood

- Most of the time, neural networks are used to define a distribution $\mathrm{p}(y^t|\boldsymbol{x}^t;\boldsymbol{\theta})$. Therefore, the overall objective is defined as:

$$argmax_{\boldsymbol{\theta}} \frac{1}{T}\sum_t \log p(y^t|\boldsymbol{x}^t;\boldsymbol{\theta}) + \lambda\Omega(\boldsymbol{\theta})$$

Outline

- Network Components
  - Neurons (Hidden Units)
  - Output units
  - Cost functions
- Architecture design
  - Capacity of neural networks
- Training
  - Backpropagation with stochastic gradient descent

Universal Approximation

- Universal Approximation Theorem (Hornik, 1991)
  - “a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrary well, given enough hidden units”
- However, we may not be able to find the right parameters ….
  - The layer may be infeasibly large
  - Optimizing neural networks is difficult …

Deeper Networks are preferred

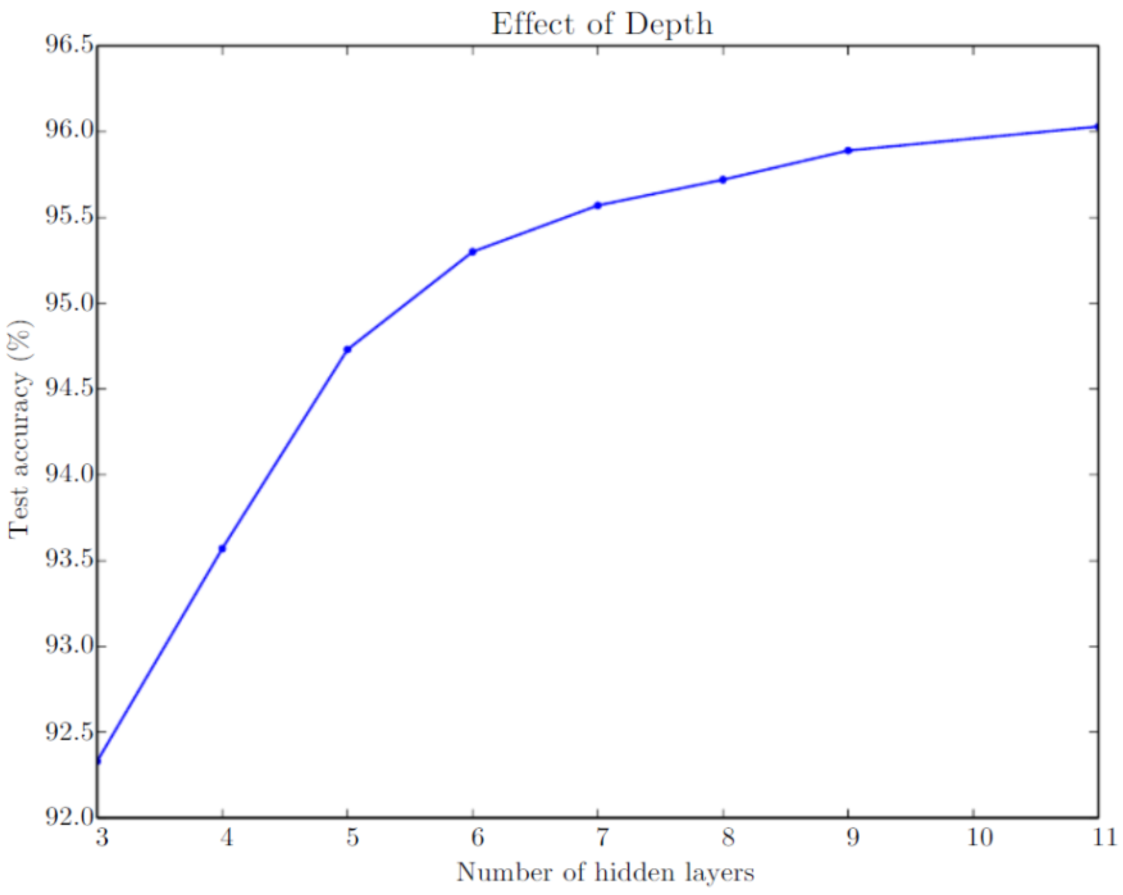


Figure: Empirical results showing that deeper networks generalize better

Deeper Networks are preferred

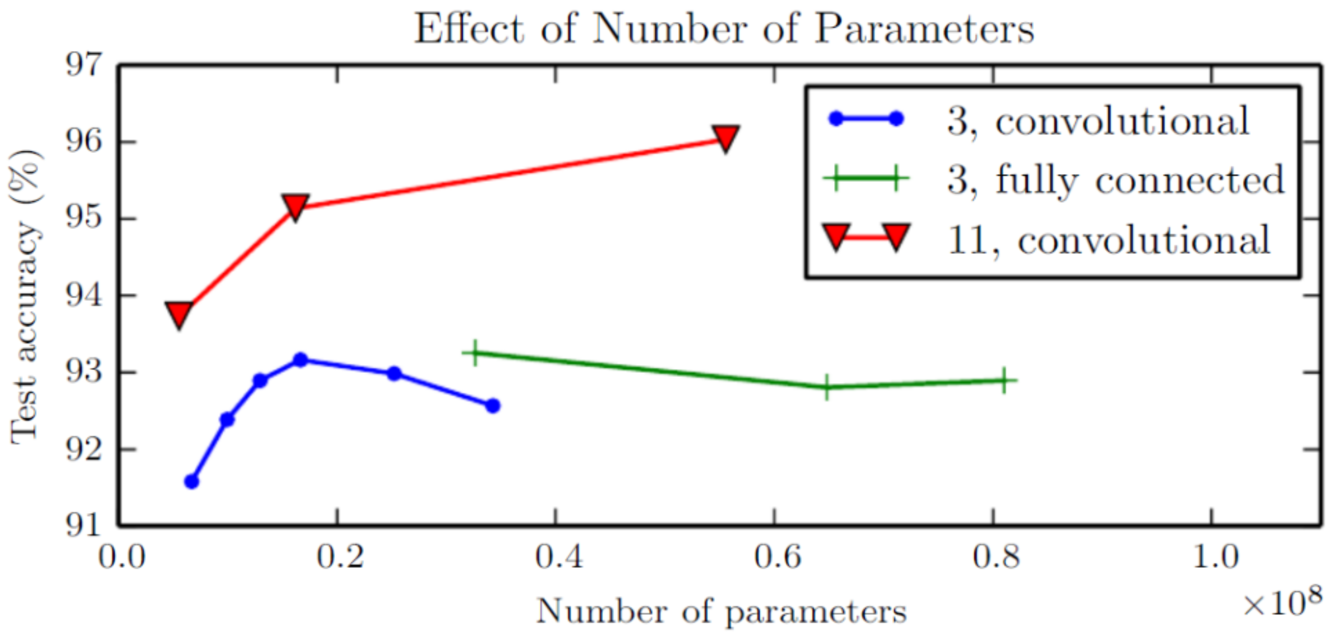


Figure: Deeper models tend to perform better with the same number of parameter

Deeper networks are preferred

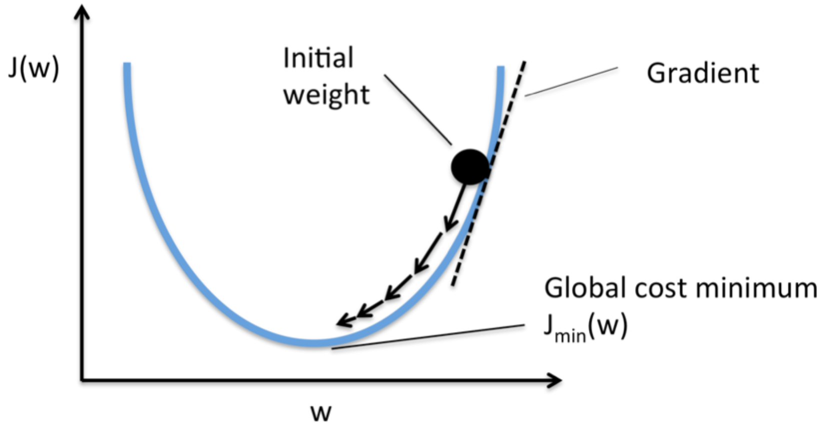
- There exist families of functions which can be approximated efficiently with deep networks but require a much larger model for shallow networks
- Statistical reasons
  - a deep models encodes a very general belief that the function we want to learn should involve composition of several simple functions
  - Or we believe the learning problem consists of discovering different levels of variations, with the high-level ones defined on the low-level (simple) ones (e.g., Pixel -> edge -> texton-> motif -> part -> object).

Outline

- Network Components
  - Neurons (Hidden Units)
  - Output units
  - Cost functions
- Architecture design
  - Capacity of neural networks
- Training
  - Backpropagation with stochastic gradient descent

Backpropagation with Stochastic Gradient Descent

- Gradient descent:
  - Update the parameters in the direction of gradients
  - Need to iterate over all the examples for every update
- Stochastic gradient descent
  - Perform updates after seeing each example



  – Initialize: $\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$

  – For t=1:T

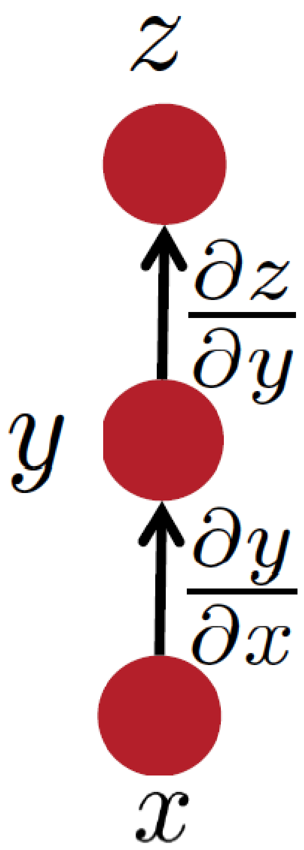
    – for each training example $(\mathbf{x}^{(t)}, y^{(t)})$

$$\Delta = -\nabla_{\boldsymbol{\theta}} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\, \Delta$$

} Training epoch = Iteration of all examples

BackPropagation:Simple Chain Rule



$$\Delta z = \frac{\partial z}{\partial y} \Delta y$$

$$\Delta y = \frac{\partial y}{\partial x} \Delta x$$

$$\Delta z = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \Delta x$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

Forward Propagation

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

Backward Propagation

After the forward computation, compute the gradient on the output layer:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$

Compute gradients on weights and biases (including the regularization term, where needed):

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$

end for

References

- Deep Learning book, chap 6.