Probabilistic Models and Machine Learning

No. 6

Artificial Neural Networks

Hui Jiang

Department of Electrical Engineering and Computer Science
York University, Toronto, Canada

Outline

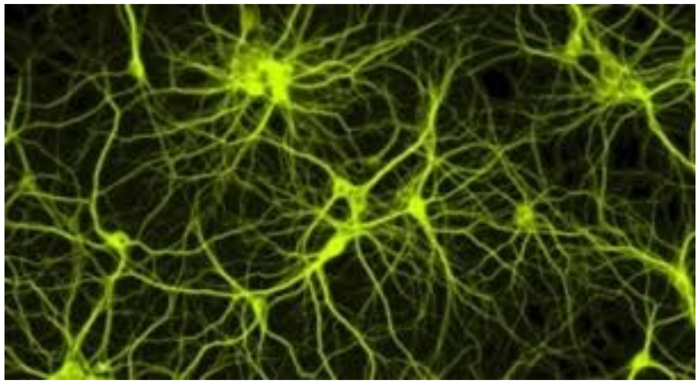
- Neural Networks: background
- Network Structure
- Learning Criterion
- Optimization (SGD + Back-propagation)
- Fine-tuning tricks
- Advanced Topics on Deep Learning
  - Other Network Structures (CNNs, RNNs/LSTMs)
  - Sequence to Sequence Learning
  - Unsupervised Learning (RBMs, auto-encoders, …)

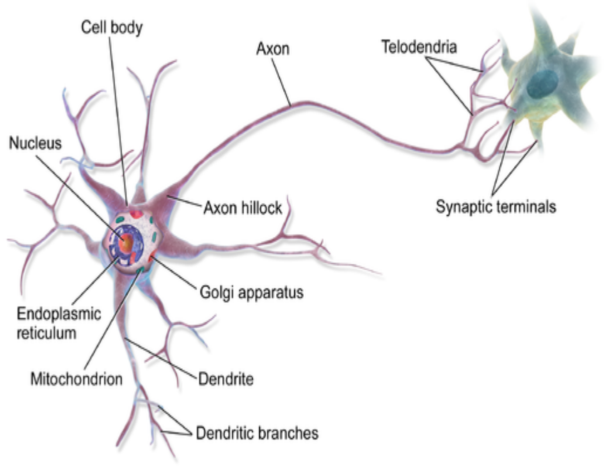
Brain: biological neuronal networks

brain

biological Neuronal nets

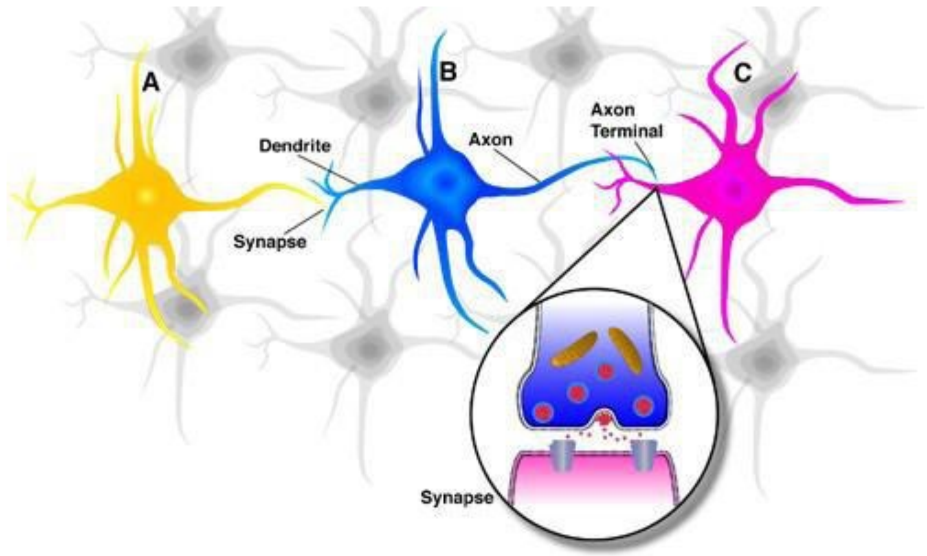
neuron

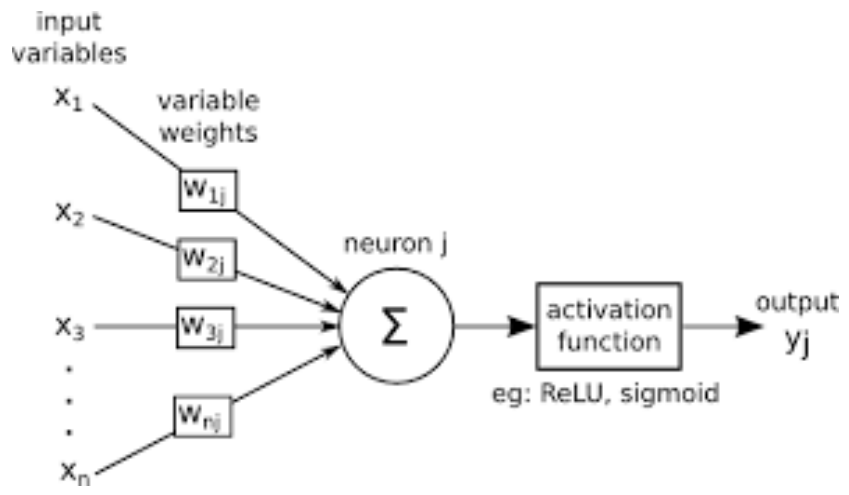




- **100 billion (10^{12}) neurons; 100 trillion (10^{15}) connections.**
- **Neuron itself is simple.**
- **Connections and weights are more important in neuronal networks.**
- **Connections and weights are all learnable.**

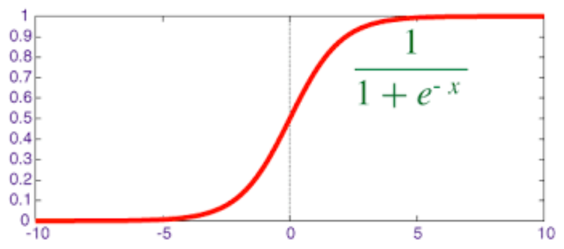
Artificial Neuron: a math model



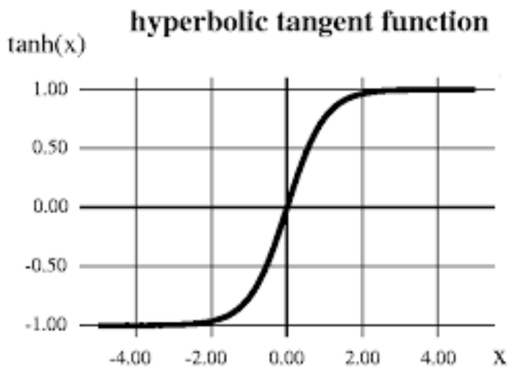


- **Linear combination + a nonlinear activation function**

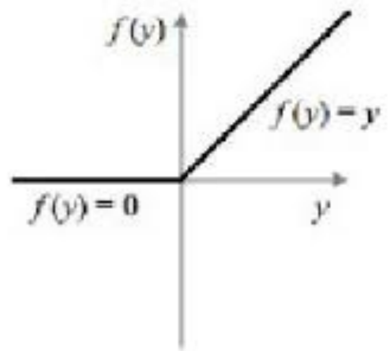
sigmoid



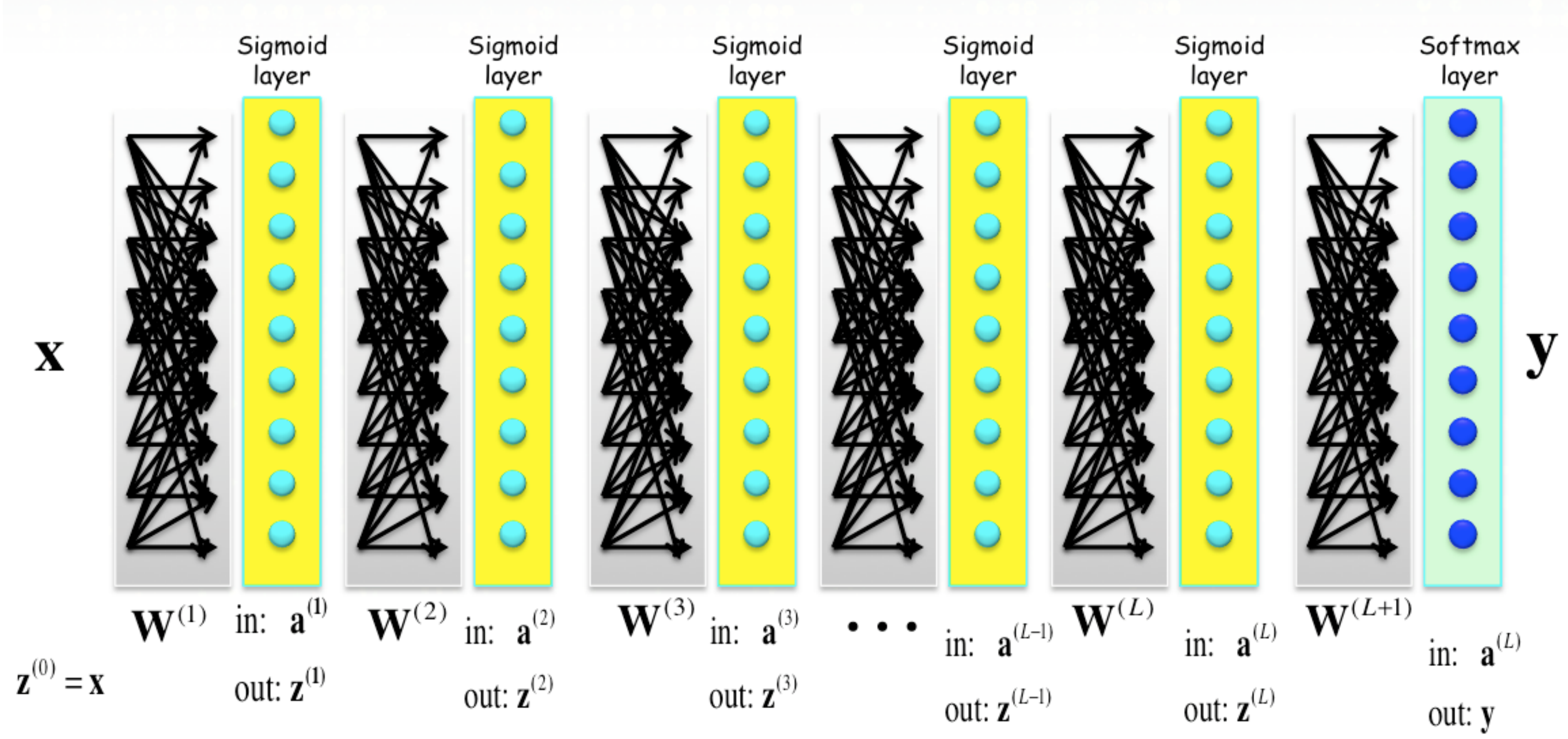
tanh



rectified linear (ReLU)



(Deep) (Artificial) Neural Networks



Sigmoid layer:

$$\mathbf{a}^{(l)} = W^{(l)}\mathbf{z}^{(l-1)} \qquad l = 1, 2, \cdots, L$$

$$\mathbf{z}^{(l)} = \sigma(\mathbf{a}^{(l)}) \Rightarrow z_k^{(l)} = \frac{1}{1+e^{-a_k^{(l)}}} \qquad l = 1, 2, \cdots, L$$

Softmax layer:

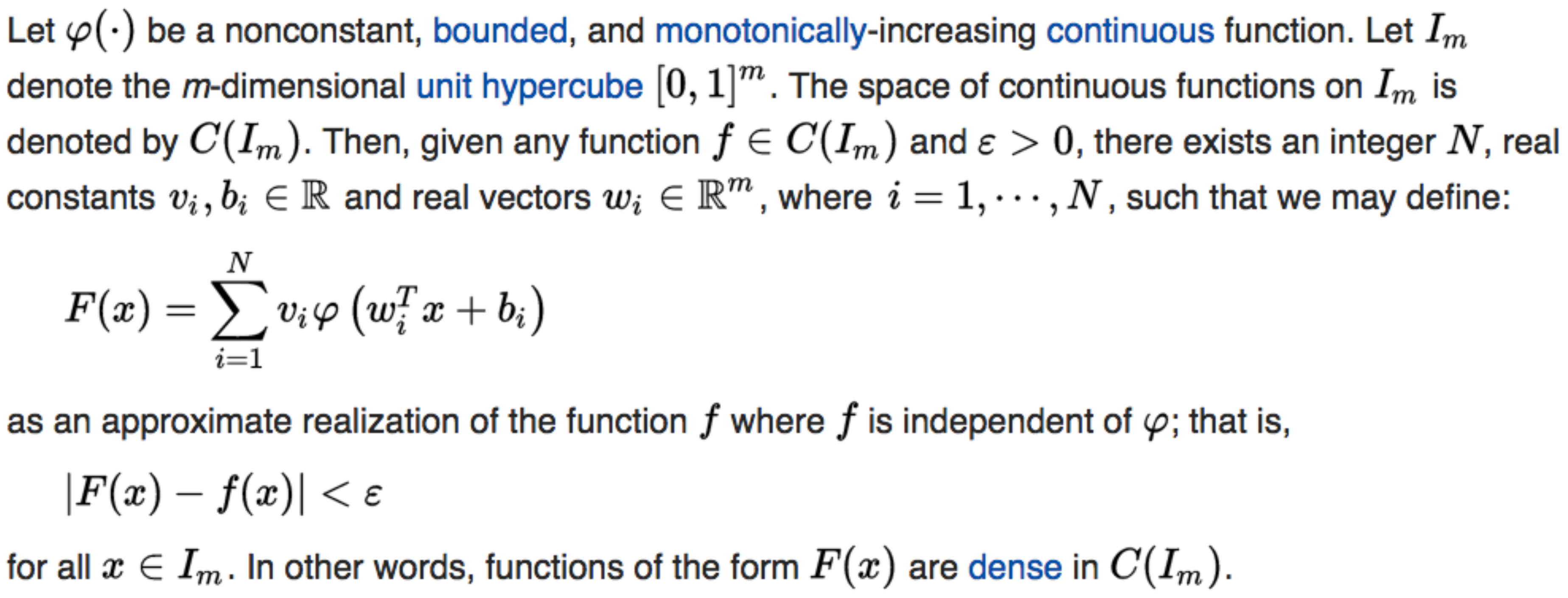
$$\mathbf{a}^{(L+1)} = W^{(L+1)}\mathbf{z}^{(L)}$$

$$\mathbf{y} = \text{softmax}(\mathbf{a}^{(L+1)}) \Rightarrow \mathbf{y}_i = \frac{e^{\mathbf{a}_i^{(L+1)}}}{\sum_{j=1}^{N} e^{\mathbf{a}_j^{(L+1)}}}$$

multi-layer feedforward structure → **deep neural networks**

Neural Networks: (a bit) theory

- ***Universal Approximator Theory, established around 1989-90***
  - ***G. Cybenko (1989); K. Hornik (1991)***



Let $\varphi(\cdot)$ be a nonconstant, bounded, and monotonically-increasing continuous function. Let I_m denote the m-dimensional unit hypercube $[0,1]^m$. The space of continuous functions on I_m is denoted by $C(I_m)$. Then, given any function $f \in C(I_m)$ and $\varepsilon > 0$, there exists an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$, where $i = 1, \cdots, N$, such that we may define:

$$F(x) = \sum_{i=1}^{N} v_i \varphi \left(w_i^T x + b_i \right)$$

as an approximate realization of the function f where f is independent of φ; that is,

$$|F(x) - f(x)| < \varepsilon$$

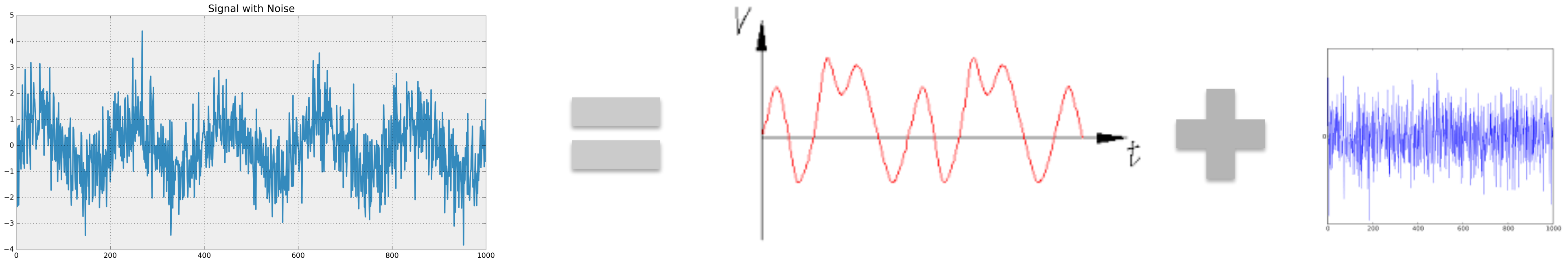
for all $x \in I_m$. In other words, functions of the form $F(x)$ are dense in $C(I_m)$.

- **One hidden layer is theoretically sufficient, but it may becomes extremely large.**

Neural Networks: (a bit) theory

- ***Universal Approximator Theory*** **is a double-edged sword:**
    - **Model is powerful**
    - **Overfitting**

data = signal + noise



Learning Neural Networks is an optimization problem

- Given *training data: $(x_1,t_1), (x_2,t_2), \ldots$*
- Given a network to be learnt: y = f (x l W)
- The error function (the objective function)
  - Mean square error (MSE):

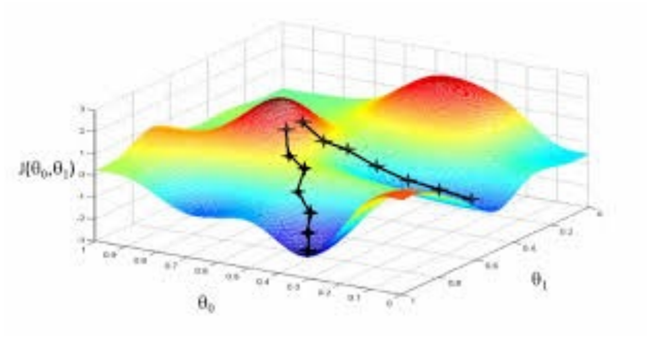
$$Q(\mathbf{W}) = \sum_i \Big(f(\mathbf{x}_i|\mathbf{W}) - t_i \Big)^2$$

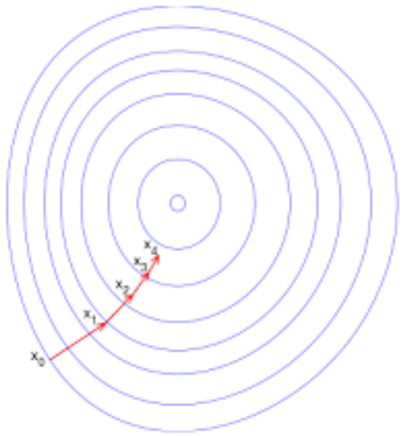
  - Cross entropy error (CE):

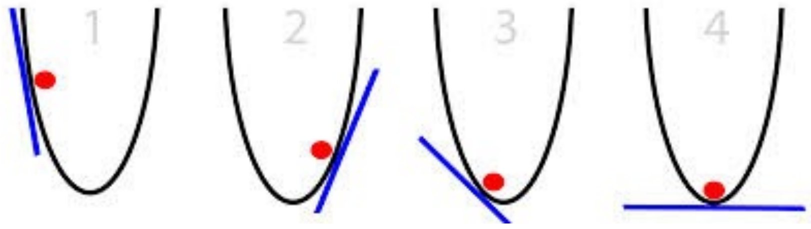
$$Q(\mathbf{W}) = \sum_i \ \mathrm{KL}\Big(\{t_i\} \ || \ \{f(\mathbf{x}_i|\mathbf{W})\}\Big) = \ -\sum_{t=1}^{N} \{\ln f(\mathbf{x}_t|\mathbf{W})\}_{l_i}$$

Gradient Descent

- **Gradient Descent: hill-climbing**





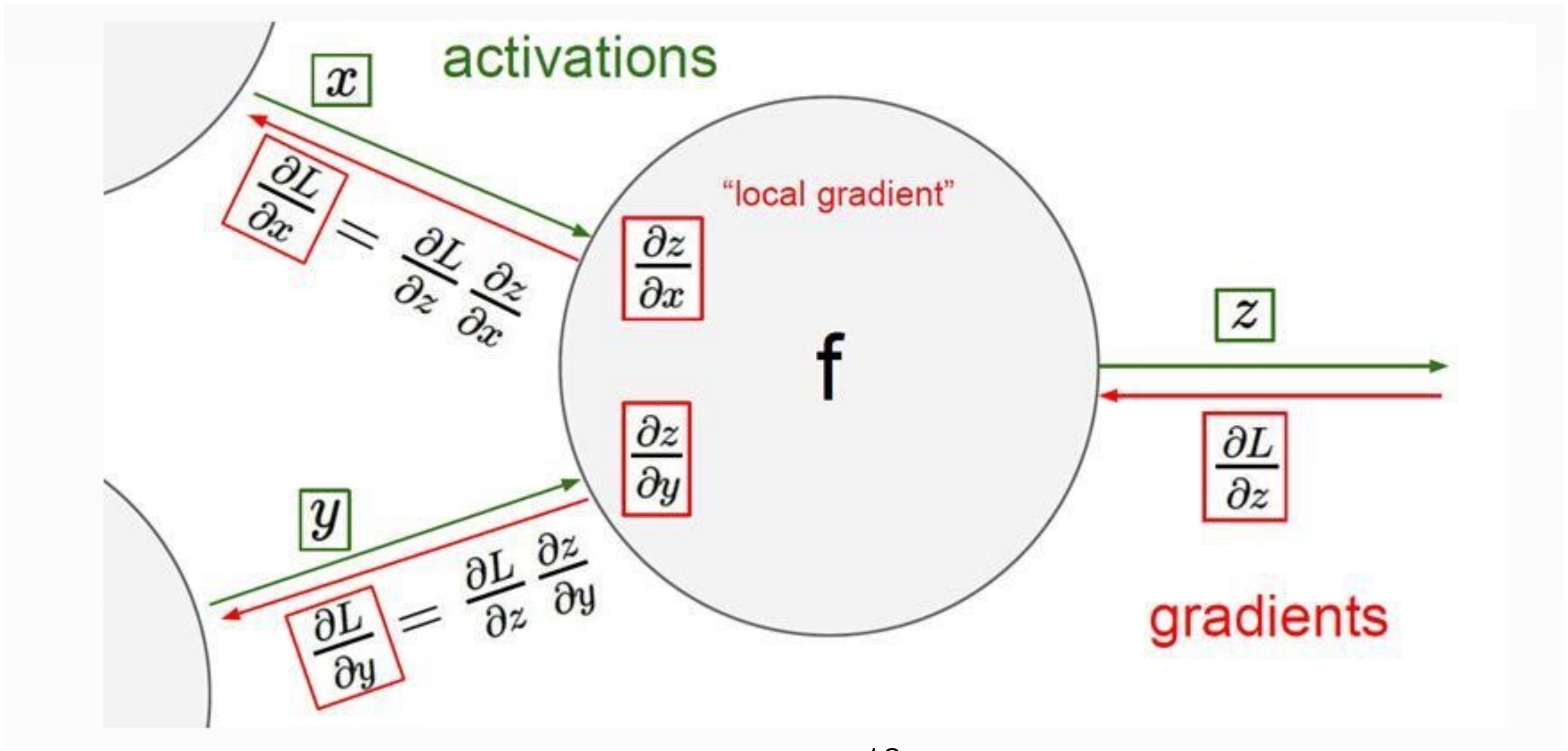


- **Iteratively update network based on the gradient**

$$\mathbf{W}^{(l+1)} = \mathbf{W}^{(l)} - \epsilon \cdot \left.\frac{\partial Q(\mathbf{W})}{\partial \mathbf{W}}\right|_{\mathbf{W}=\mathbf{W}^{(l)}}$$

Error Back-propagation (BP)

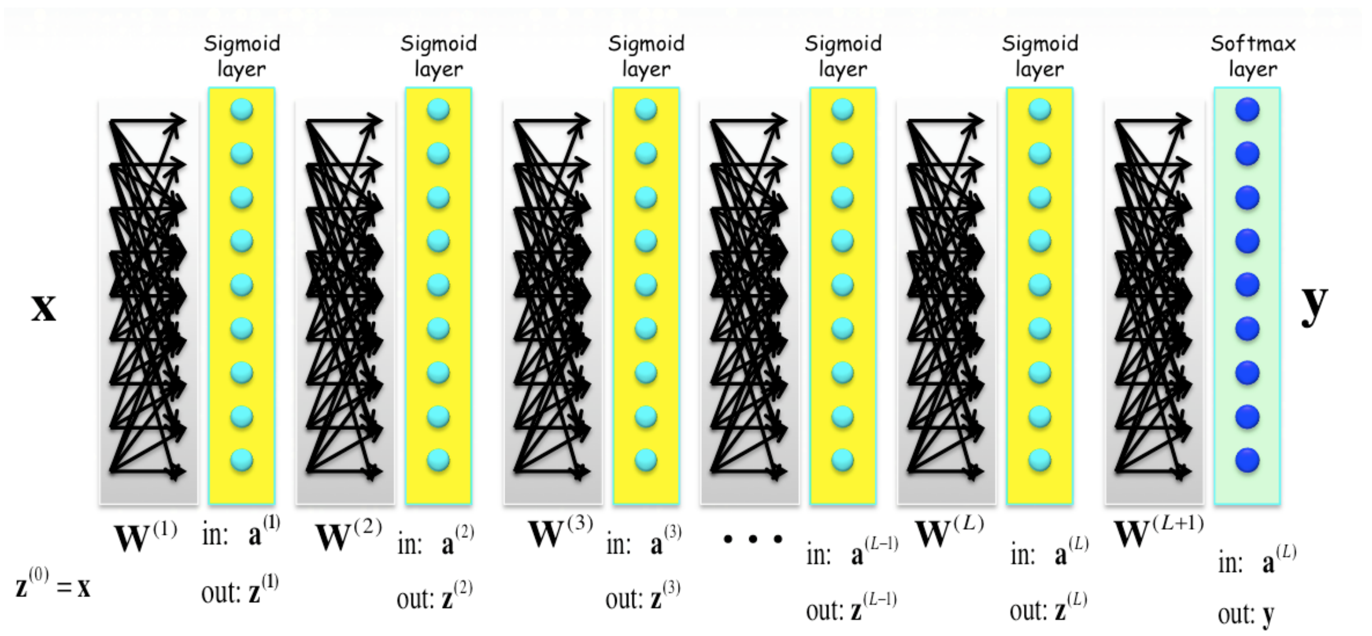
- The key problem: how to computer gradients in the most efficient way?
    - The Error Back-Propagation (BP) Algorithm
- A local perspective on how BP works …
- Based on the well-known chain rule in Calculus …


activations
x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
f
z
$\frac{\partial z}{\partial y}$
y
$\frac{\partial L}{\partial z}$
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
gradients

Mini-batch Stochastic Gradient Descent

- **Given all *training data:* *(x_1,t_1), (x_2,t_2), …***
- **Randomly select a mini-batch (10-1000 samples) of data**
  - **For every sample in the mini-batch *(x_i,t_i)***
  - ***Forward pass*: *use NN to compute x_i —> y_i***
  - ***Accumulate error for the mini-batch Q_i***
  - ***Backward pass*: back-propagate error *Q_i to compute gradients***
  - **Update network weights:** $$\mathbf{W}^{(l+1)} = \mathbf{W}^{(l)} - \epsilon \cdot \left. \frac{\partial Q(\mathbf{W})}{\partial \mathbf{W}} \right|_{\mathbf{W}=\mathbf{W}^{(l)}}$$

Neural Networks: how to compute gradients



Sigmoid layer
Sigmoid layer
Sigmoid layer
Sigmoid layer
Sigmoid layer
Softmax layer
$\mathbf{x}$
$\mathbf{y}$
$\mathbf{W}^{(1)}$
$\mathbf{W}^{(2)}$
$\mathbf{W}^{(3)}$
$\mathbf{W}^{(L)}$
$\mathbf{W}^{(L+1)}$
$\mathbf{z}^{(0)} = \mathbf{x}$
in: $\mathbf{a}^{(1)}$ out: $\mathbf{z}^{(1)}$
in: $\mathbf{a}^{(2)}$ out: $\mathbf{z}^{(2)}$
in: $\mathbf{a}^{(3)}$ out: $\mathbf{z}^{(3)}$
• • •
in: $\mathbf{a}^{(L-1)}$ out: $\mathbf{z}^{(L-1)}$
in: $\mathbf{a}^{(L)}$ out: $\mathbf{z}^{(L)}$
in: $\mathbf{a}^{(L)}$ out: $\mathbf{y}$

- **Define error signals in each layer:** $\mathrm{e}^{(l)} = \dfrac{\partial}{\partial \mathbf{a}^{(l)}} Q(\mathbf{W})$

$$\frac{\partial}{\partial \mathbf{W}^{(l)}} Q(\mathbf{W}) = \frac{\partial Q(\mathbf{W})}{\partial \mathbf{a}^{(l)}} \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{W}^{(l)}} = \mathbf{e}^{(l)} (\mathbf{z}^{(l)})^{\mathsf{T}}$$

- **Backward pass: back-propagate error signals through the whole network**

Error Back-propagation (BP)

- **Multi-layer feedforward structure; sigmoid activations; cross-entropy errors**

Given a training set $\mathbf{X} = \{\mathbf{x}_t, l_t \mid t = 1, 2, \cdots, N\}$

$$Q(\mathbf{W}) = -\sum_{t=1}^{N} \{\ln f(\mathbf{x}_t | \mathbf{W})\}_{l_t}$$

Compute the error signals for each layer:

Softmax layer l=L+1

$$e_{tk}^{(L+1)} = \frac{1}{y_{l_t}(\mathbf{x}_t, \mathbf{W})} \frac{\partial y_{l_t}(\mathbf{x}_t, \mathbf{W})}{\partial a_k^{(L+1)}} = \delta(l_t - k) - y_{l_t}$$

Sigmoid layer l=1,2,…,L

$$e_{tk}^{(l)} = \frac{\partial Q_t(W)}{\partial a_k^{(l)}} = \sum_{j=1}^{N} \frac{\partial Q_t(W)}{\partial a_j^{(l+1)}} \frac{\partial a_j^{(l+1)}}{\partial a_k^{(l)}} = \sum_{j=1}^{N} e_{tj}^{(l+1)} \frac{\partial a_j^{(l+1)}}{\partial a_k^{(l)}} = \sum_{j=1}^{N} e_{tj}^{(l+1)} \cdot z_k^{(l)} \cdot (1 - z_k^{(l)}) \cdot W_{kj}^{(l+1)}$$

$$= z_k^{(l)} \cdot (1 - z_k^{(l)}) \cdot \sum_{j=1}^{N} e_{tj}^{(l+1)} W_{jk}^{(l+1)}$$

Neural Networks Learning in practice

- Open source toolkits: *Tensorflow*, *Torch,* CNTK, MXNet etc …
- Computationally intensive (GPUs)
- Many tuning tricks:
  - Mini-batch size
  - Epoch
  - Learning rates (annealing schedule)
  - Network initialization
  - Weight Decay (L2 norm regularization)
  - Momentum
  - Dropout
  - Batch Normalization
  - …

Neural Networks Initialization

- NNs initialization is critical for a good convergence.
- Random Initialization is sufficient.
  - Uniform distribution
  - Norm distribution
- Controlling the dynamic range (variance) is the key.
- A widely used trick from Glorot and Bengio (2010):

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$

Weight Decay

- **Weight decaying is equivalent to L2 norm regularization.**

$$Q(\mathbf{W}) + \lambda \cdot ||\mathbf{W}||_2$$

- **Updating formula with weight decay:**

$$\mathbf{W}^{(l+1)} = \mathbf{W}^{(l)} - \epsilon \cdot \left. \frac{\partial Q(\mathbf{W})}{\partial \mathbf{W}} \right|_{\mathbf{W}=\mathbf{W}^{(l)}} - \lambda \cdot \mathbf{W}^{(l)}$$

Momentum

- **Momentum is a simple technique to accelerate convergence in slow but relevant directions, dampen oscillation in really steep directions.**

- **Averaging the velocity at each updating step:**

$$\Delta \mathbf{W}^{(l+1)} = \left.\frac{\partial Q(\mathbf{W})}{\partial \mathbf{W}}\right|_{\mathbf{W}=\mathbf{W}^{(l)}} + \eta \cdot \Delta \mathbf{W}^{(l)}$$

$$\mathbf{W}^{(l+1)} = \mathbf{W}^{(l)} - \epsilon \cdot \Delta \mathbf{W}^{(l+1)}$$

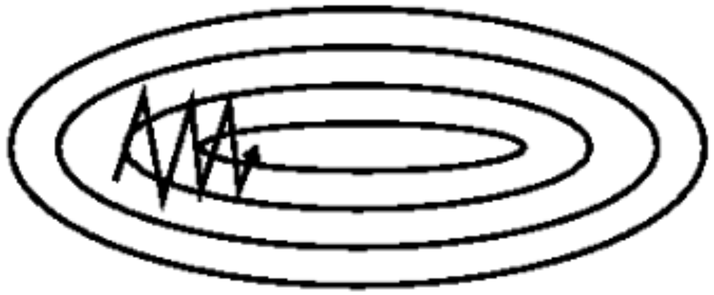


Image 2: SGD without momentum

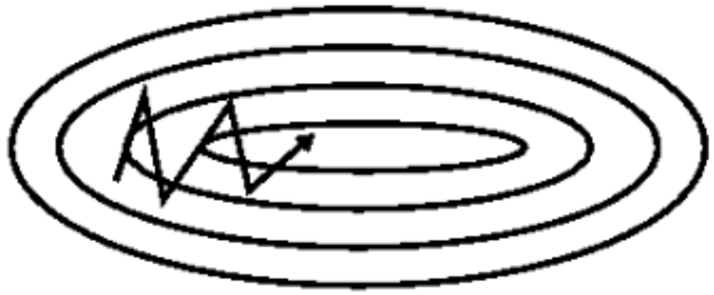
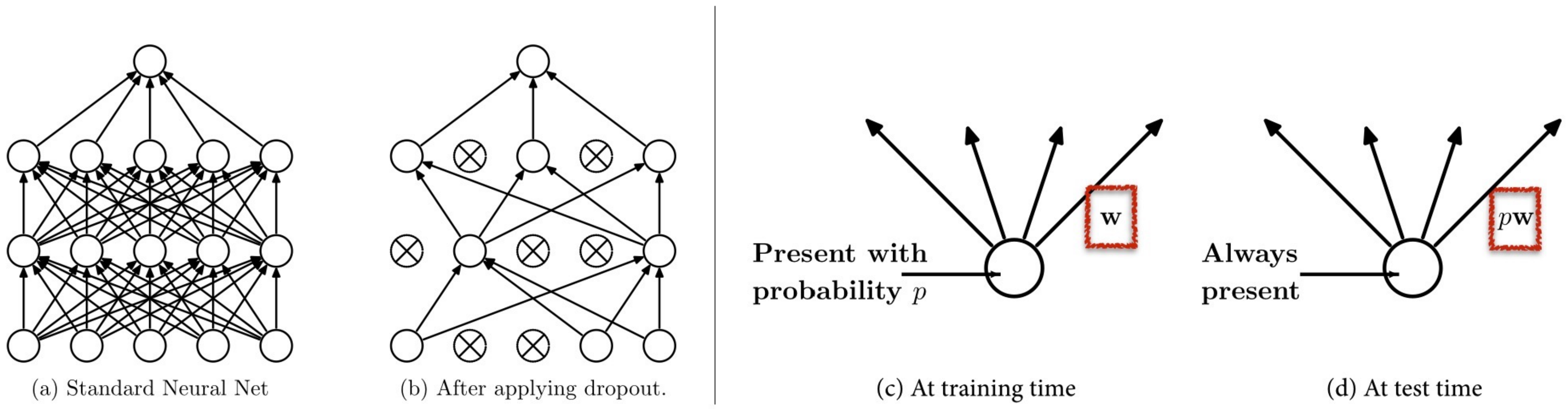


Image 3: SGD with momentum

Dropout

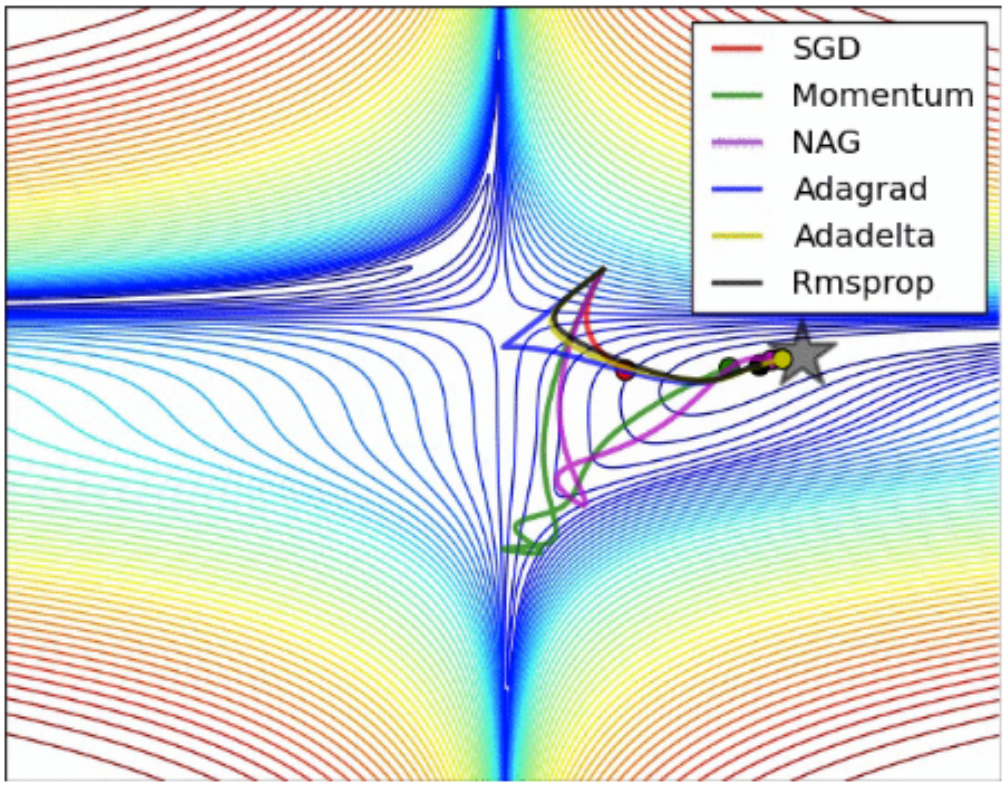
- Dropbox is a simple regularization technique.
- Randomly drop-out some nodes in training.
- Equivalent to adding noises in training
- A relevant technique: *data augmentation*



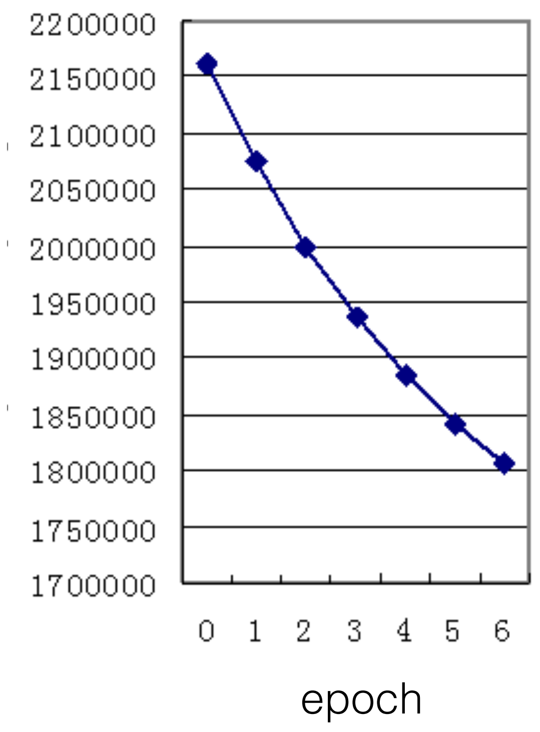
(a) Standard Neural Net (b) After applying dropout. (c) At training time (d) At test time

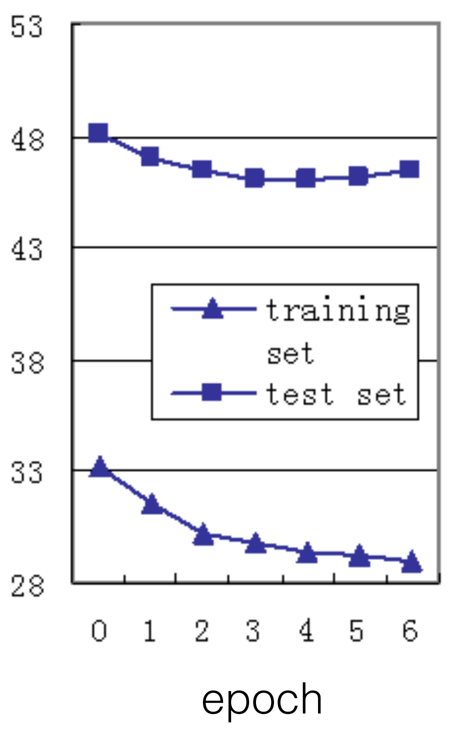
Other Optimization Algorithms

- In addition to SGD, many other optimization algorithms may be used:
  - Nesterov accelerated gradient descent
  - Adagrad
  - Adadelta
  - RMSprop
  - Adam
  - Hessian-free



Monitoring Three Learning Curves





- **How does your learning go?**
  - **The objective function**
  - **The error rates in the training set**
  - **The error rates in a development set**

Insights from Figures

- **Monitoring learning curves tells you a lot about the learning process …**

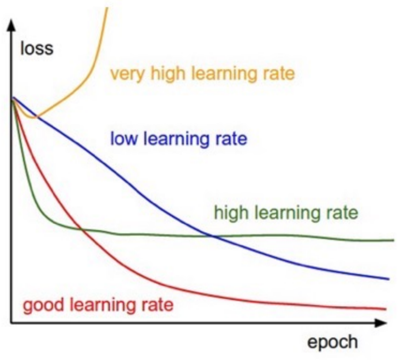


Figure 1

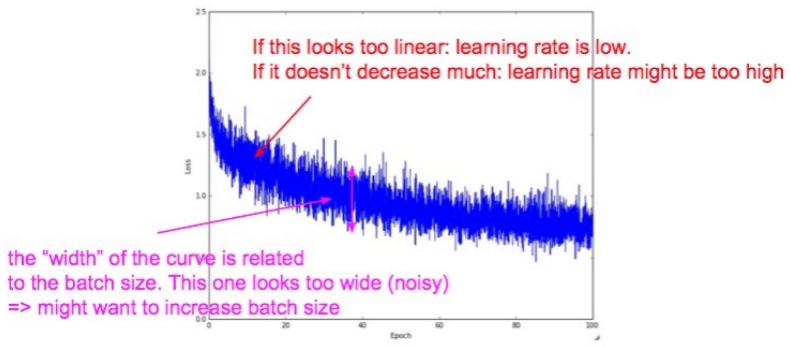


Figure 2

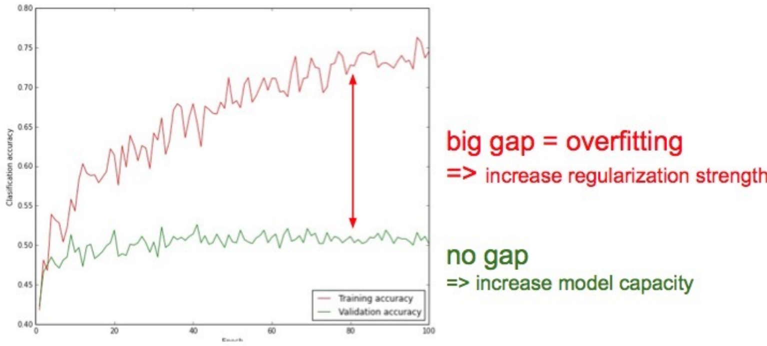
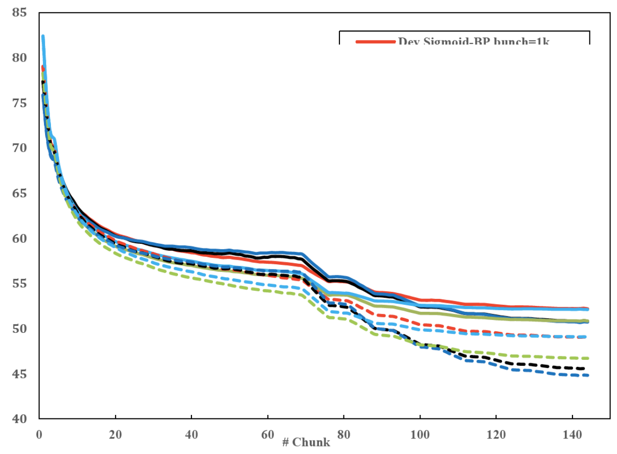


Figure 3

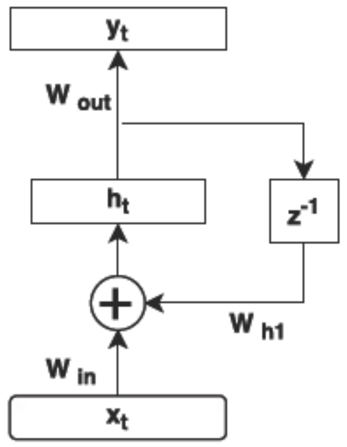


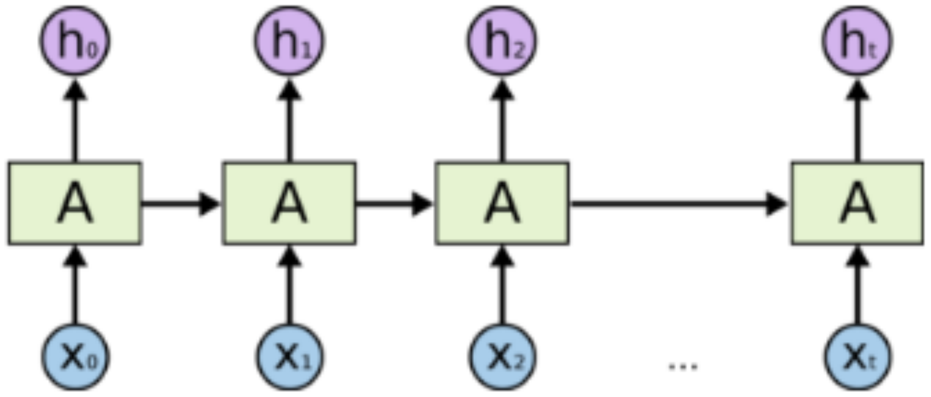
Neural Networks Structures

- Feedforward multi-layer DNNs
  - Fixed-size input —> fixed-size output
  - Memoryless
  - Fully-connected —> input location sensitive
- Recurrent Neural networks (RNNs)
- Convolutional Neural Networks (CNNs)

RNNs

- Plain RNNs

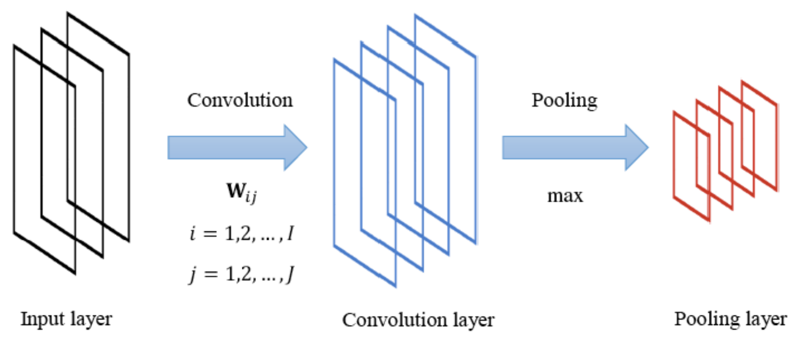


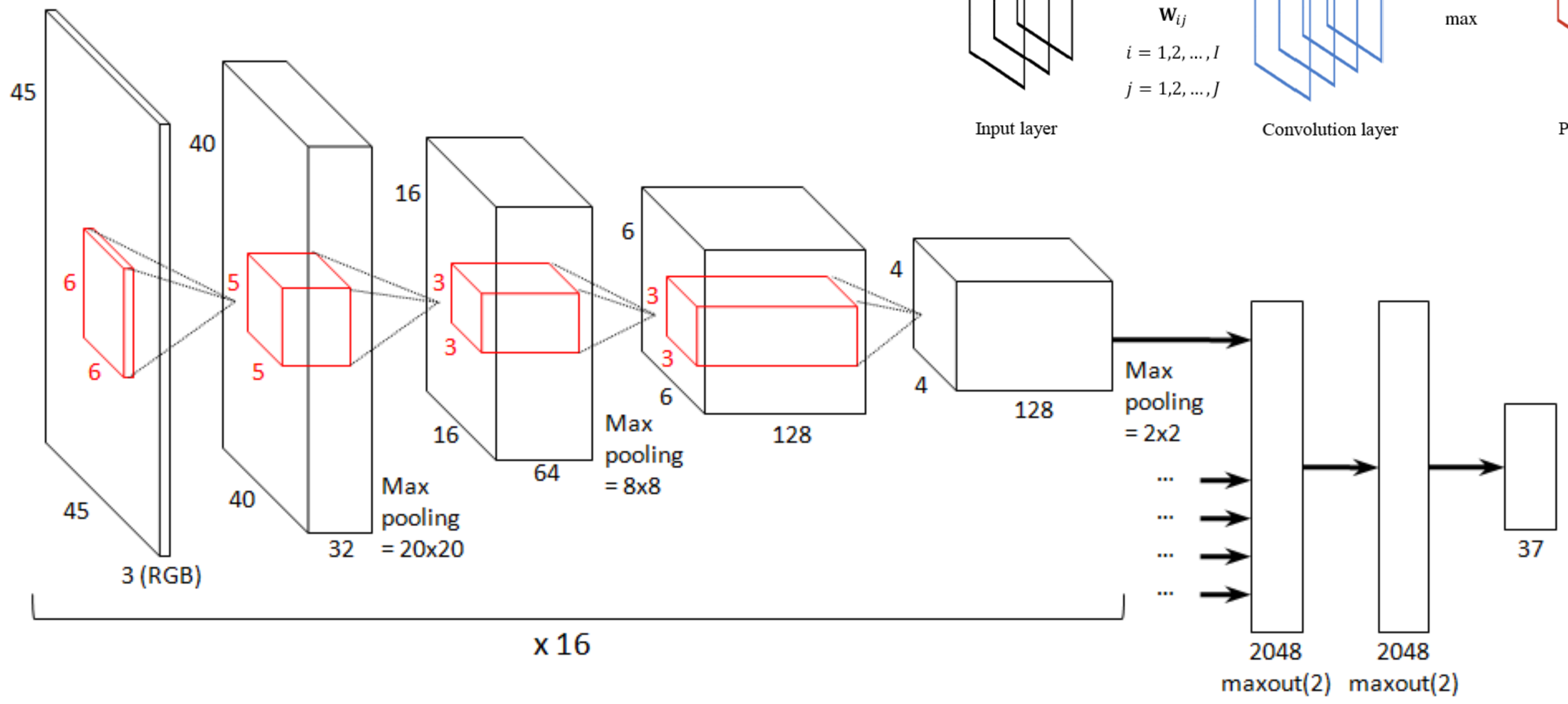


- RNNs are notoriously hard to learn
  - Computationally expensive
  - Gradient vanishing or exploding
- Long Short-Term Memory (LSTM)

CNNs

- Each CNN layer: a convolution layer + a pooling layer
- Insensitive to input locations; suitable for image recognition





Neural Networks Learning in practice

- Open Source Toolkits:
  - Google's *Tensorflow* *(https://www.tensorflow.org/)*
  - *Facebook's Torch (http://torch.ch/)*
  - *Microsoft's* CNTK (https://github.com/Microsoft/CNTK/wiki)
  - MXNet (http://mxnet.io/)
  - more

Advanced Topics in Deep Learning

- Convolutional Neural Networks (CNNs)
- Recurrent Neural Networks (RNNs) and LSTMs
- Sequence to Sequence Learning
- Bottleneck Features
- Unsupervised Learning:
  - Restricted Boltzmann Machine (RBM)
  - (De-noising) Auto-Encoder
  - Generative Adversarial Networks