Probabilistic Models and Machine Learning

No. 6

Artificial Neural Networks

Hui Jiang

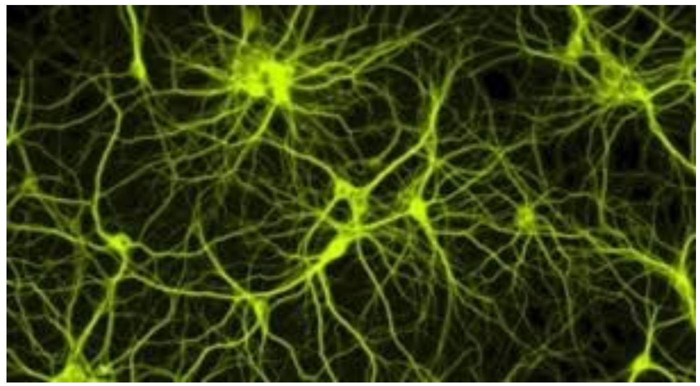
Department of Electrical Engineering and Computer Science
York University, Toronto, Canada

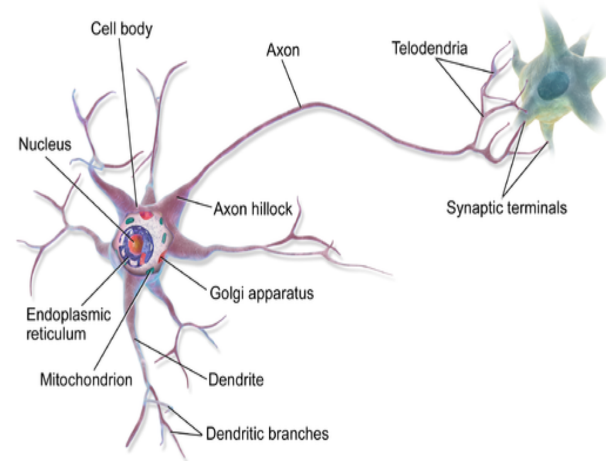
Outline

- Neural Networks: background
- Common Network Structures
  - FC-NN; CNN; RNN; Transformer
- Learning Criterion + SGD
- Auto Differentiation: error back-propagation
- Lots of fine-tuning tricks

Brain: biological neuronal networks

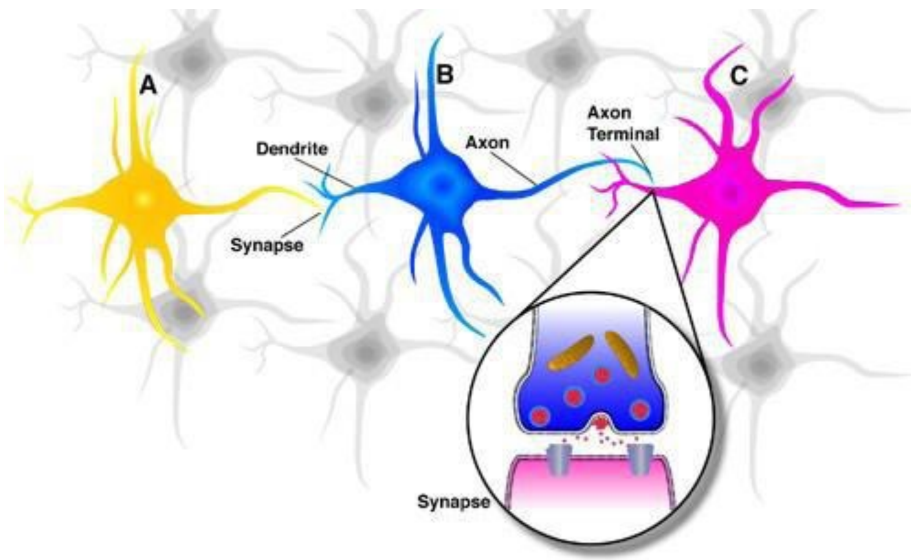
brain	biological Neuronal nets	neuron

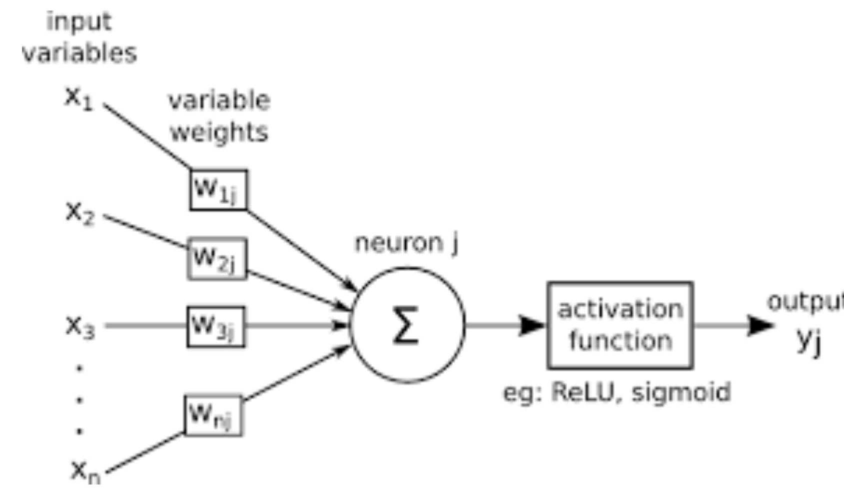




- 100 billion (10^{12}) neurons; 100 trillion (10^{15}) connections.
- Neuron itself is simple.
- Connections and weights are more important in neuronal networks.
- Connections and weights are all learnable.

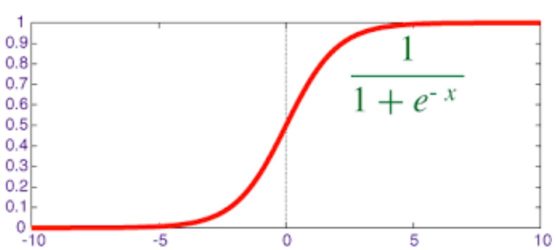
Artificial Neuron: a simple math model



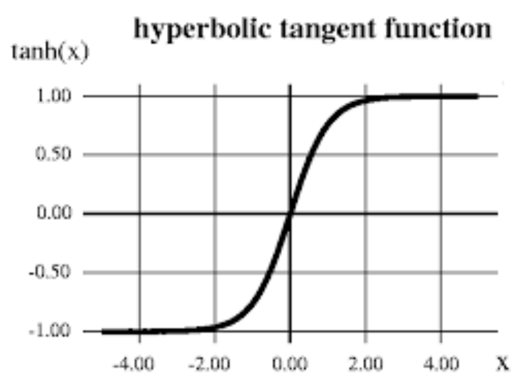


- **Linear combination + a nonlinear activation function**

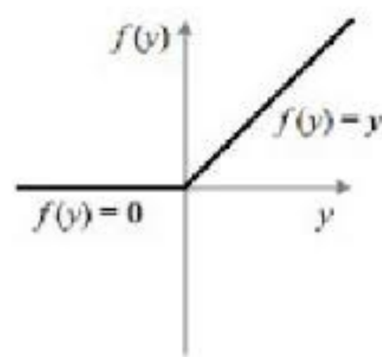
sigmoid



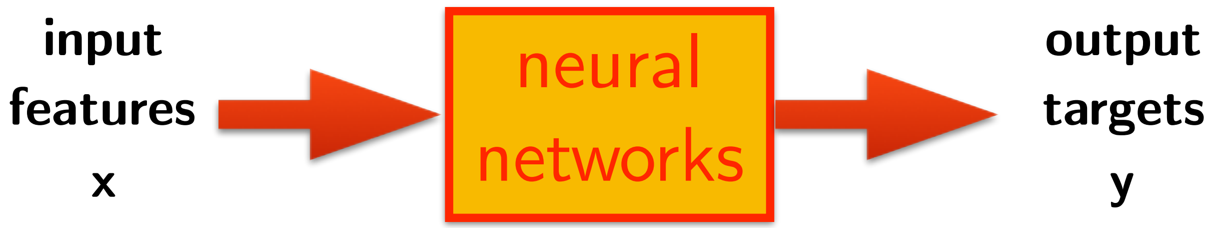
tanh



rectified linear (ReLU)



Artificial Neural Network Construction



- **Fully-connected layers:** for universal function approximation
- **Convolution layers:** for locality modelling and weight sharing
- **Feedbacks:** recurrent way to keep track of history in sequences
- **Tapped-delay-lines:** nonrecurrent ways to memorize history in sequences
- **Attentions:** feature selection or long-span dependence in sequences

Neural Networks: (a bit) theory

- *U*niversal 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 need to be extremely large.

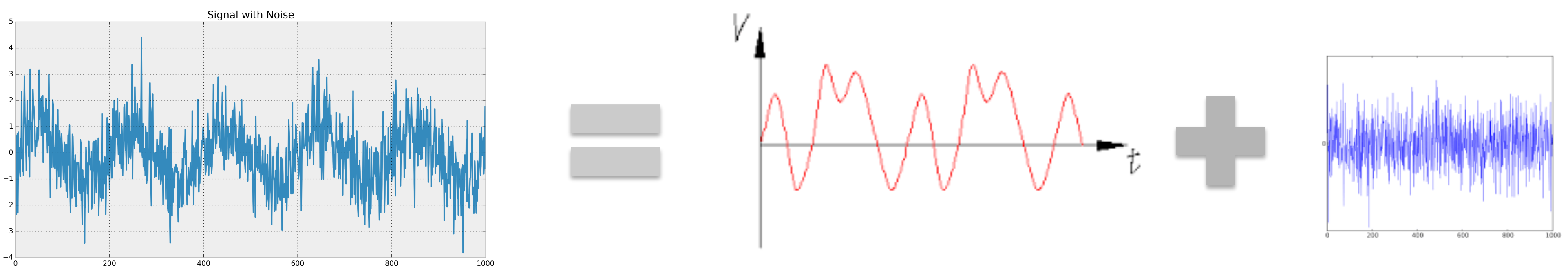
Neural Networks: (a bit) theory

- ***Universal Approximator Theory*** **is a double-edged sword:**

  - **Model is powerful**

  - **Overfitting**

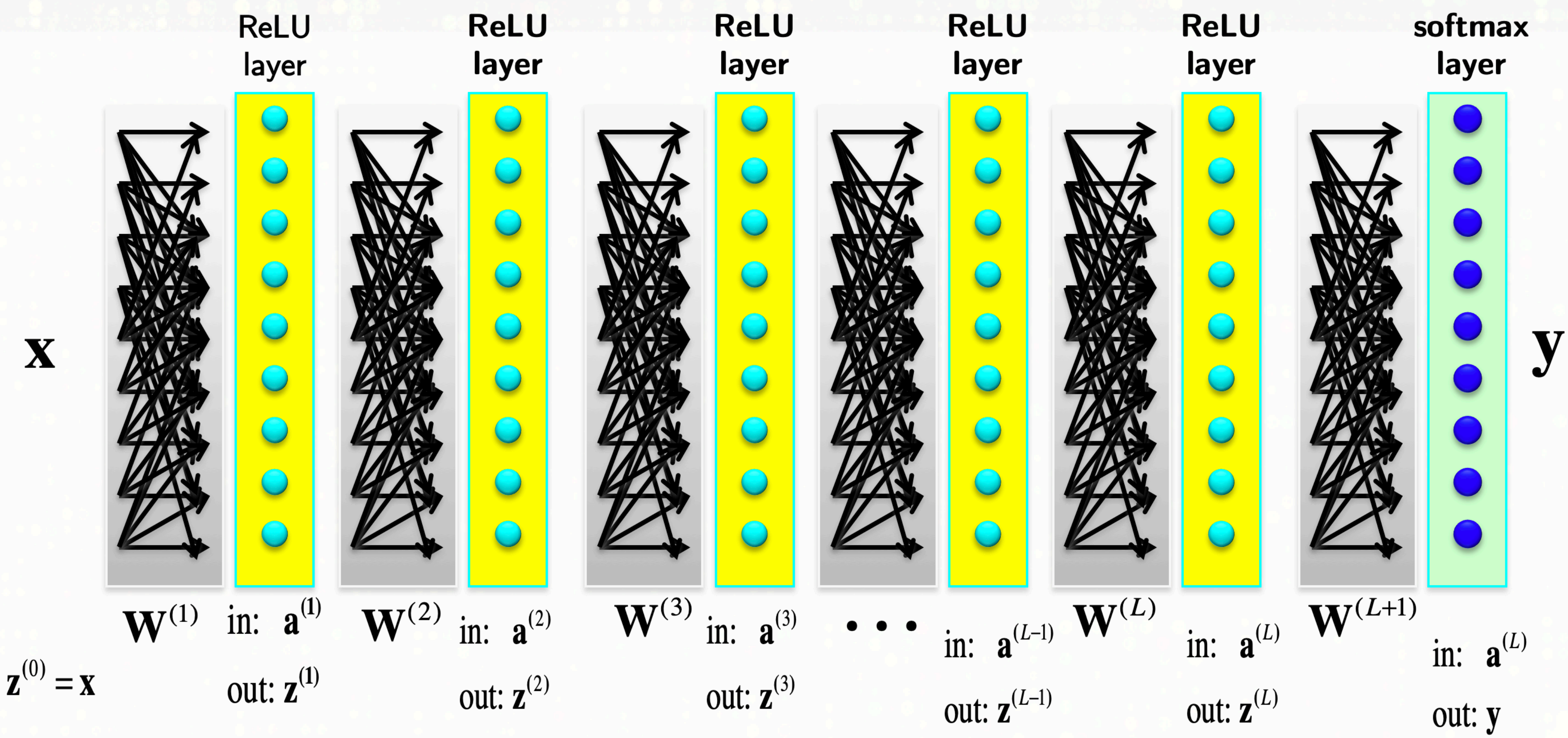
data = signal + noise



Neural Network Structures

- Feedforward DNNs: multiple fully-connected layers
  - Fixed-size input —> fixed-size output
  - Memoryless
- Convolutional Neural Networks (CNNs)
- Recurrent Neural networks (RNNs)
- Transformers

Fully-Connected (FC) Neural Networks



ReLU layers:

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

$$\mathbf{z}^{(l)} = \mathrm{ReLU}(\mathbf{a}^{(l)}) \quad (l = 1, 2, \cdots, L)$$

Softmax layer:

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

$$\mathbf{y} = \mathrm{softmax}(\mathbf{a}^{(L+1)}) \implies y_i = \frac{\exp(a_i^{(L+1)})}{\sum_j \exp(a_j^{(L+1)})}$$

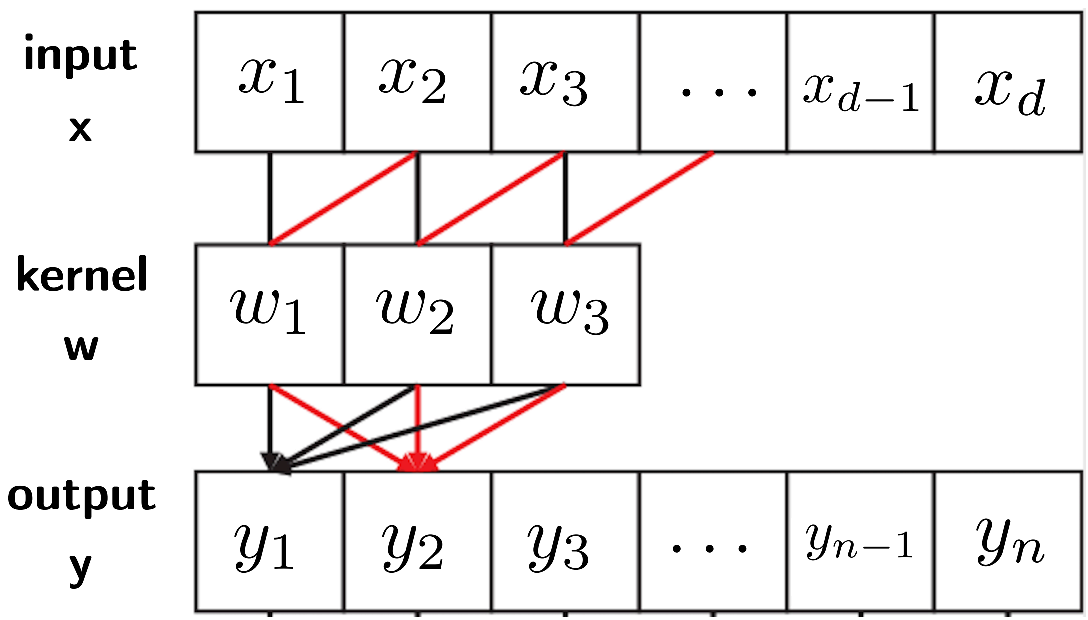
Convolutional Neural Networks (CNN)

- Convolution sum: a local feature extractor
- Convolutional Neural Networks (CNNs):
  - Extension #1: multiple input feature plies
  - Extension #2: more kernels $\Longrightarrow$ feature maps in output
  - Extension #3: multiple input dimensions (2D/3D)
  - Extension #4: more layers (+ ReLU + max-pooling)
- Case study: ResNet — very deep structure with shortcut paths

Convolution (1): basics

- **Convolution sum (1-dim): a basic operation in signal processing**

$$\mathbf{y} = \mathbf{x} * \mathbf{w} \implies y_j = \sum_{i=1}^{f} w_i \times x_{j+i-1} \quad (\forall j = 1, \cdots, n)$$

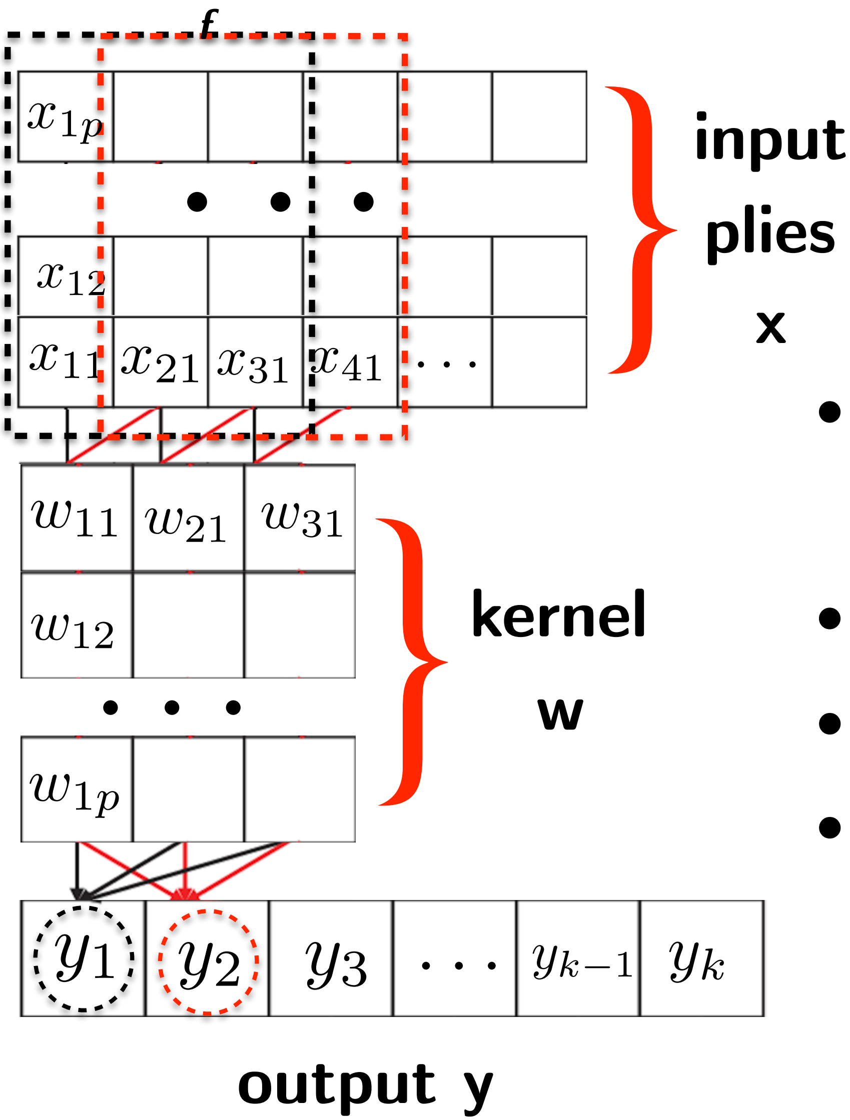


- **size**: input *d*, kernel *f*
  $\Rightarrow$ output *n=d-f+1*
- **complexity**: $O(d \cdot f)$
- **stride**: *s=1,2,…*
- zero **padding** in input

- **locality modelling:** only capture a local feature
- **weights sharing:** *f (<d)* weights
  (vs. *d x n* weights in fully-connected)

Convolution (2): multiple kernels

- **Extension #1: allow multiple feature plies (e.g. RGB) in each input location**



$$y_j = \sum_{i=1}^{p} \sum_{k=1}^{f} w_{k,i} x_{j+k-1,i}$$

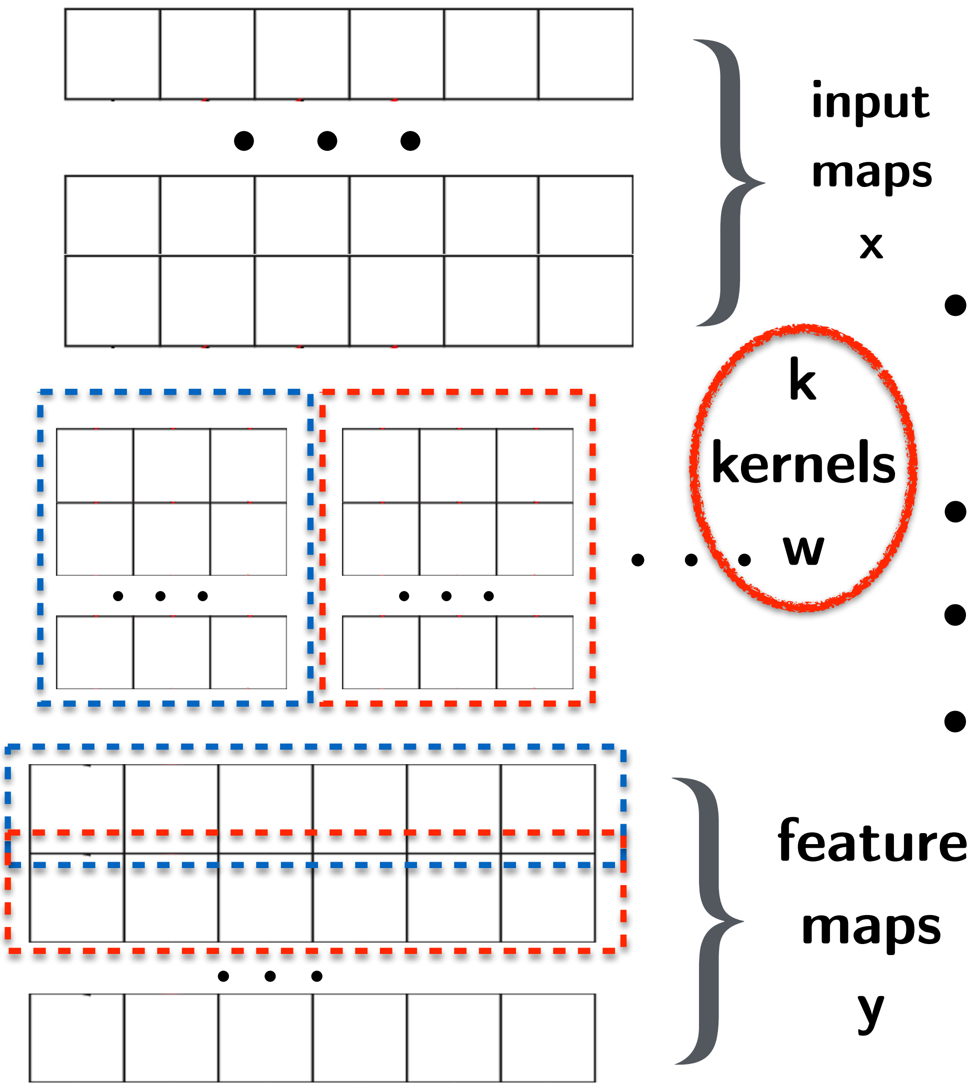
$$\mathbf{y} = \mathbf{x} * \mathbf{w}$$

- **size**: input $d \times p$, kernel $f \times p$
  $\implies$ output $d - f + 1$
- **complexity**: $O(d \cdot f \cdot p)$
- **stride**: *s=1,2,...*
- zero **padding** in input

- **locality modelling**
- **weights sharing**

Convolution (3): multiple kernels

- Extension #2: allow multiple kernels to catch different local features



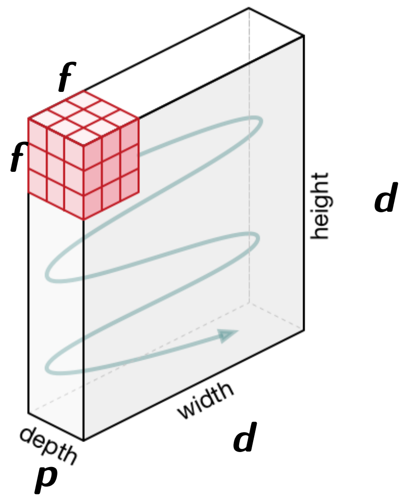
$$\mathbf{y} = \mathbf{x} * \mathbf{w}$$

- **size**: input $d \times p$, kernel $f \times p \times k$
  $\Longrightarrow$ output $(d - f + 1) \times k$
- **complexity**: $O(d \cdot f \cdot p \cdot k)$
- **stride**: s=1,2,…
- zero **padding** in input

- **locality modelling**
- **weights sharing**

Convolution (4): multiple input dimensions

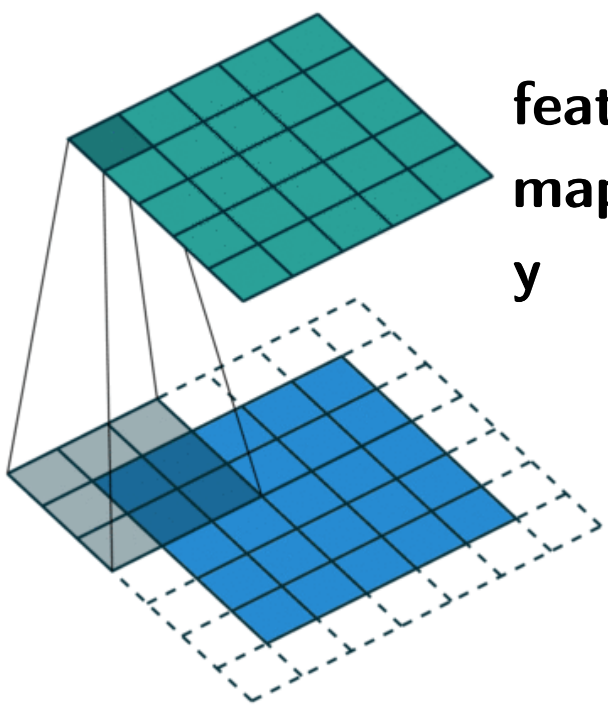
- **Extension #3: allow multiple input dimensions (2D:images; 3D:videos)**



$$\mathbf{y} = \mathbf{x} * \mathbf{w}$$

- **size**: input $d^2 \times p$, kernel $f^2 \times p \times k$

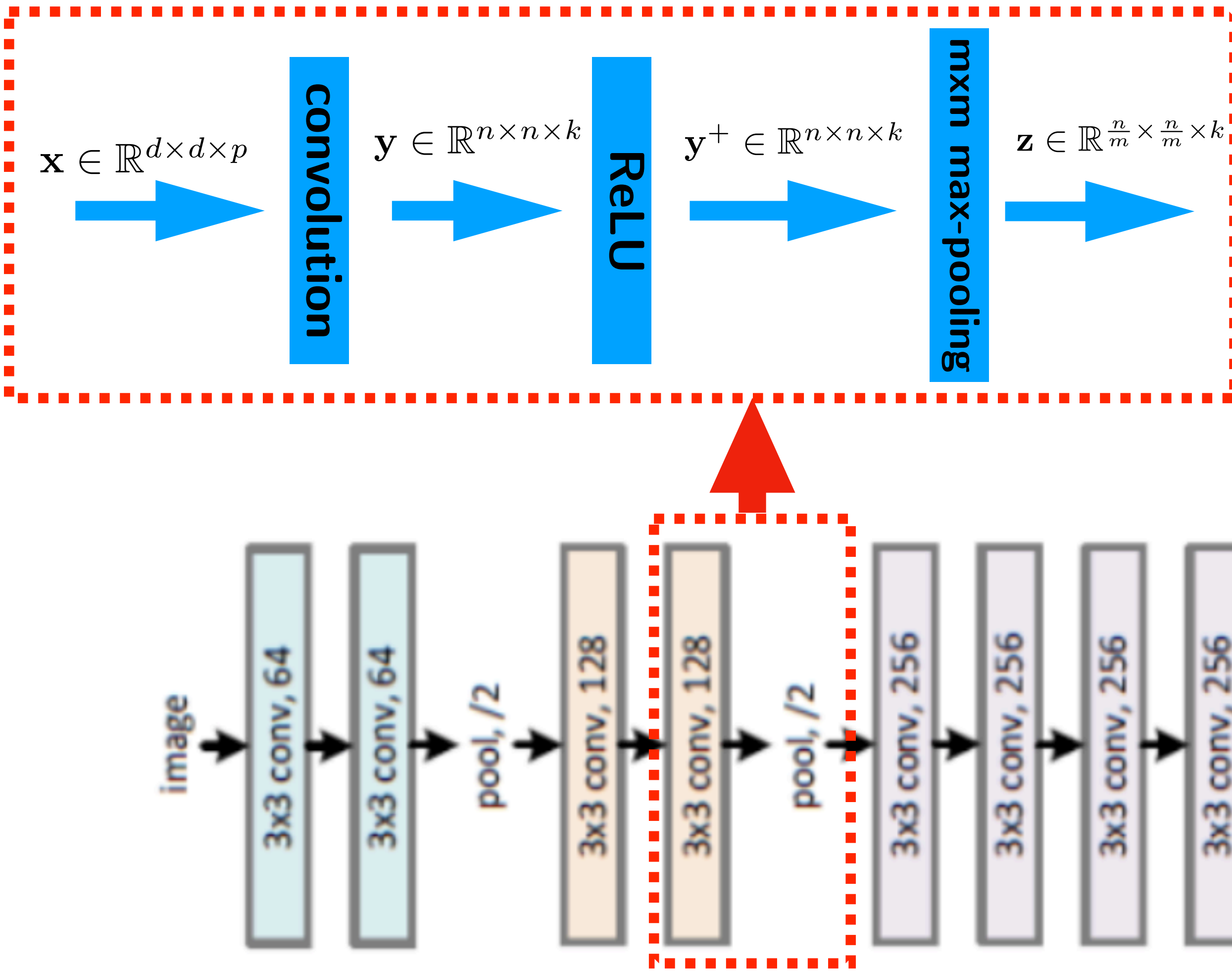
  $\Longrightarrow$ output $(d - f + 1) \times k$
- **complexity**: $O(d^2 \cdot f^2 \cdot p \cdot k)$
- **stride**: s=*1,2,...*
- zero **padding** in input

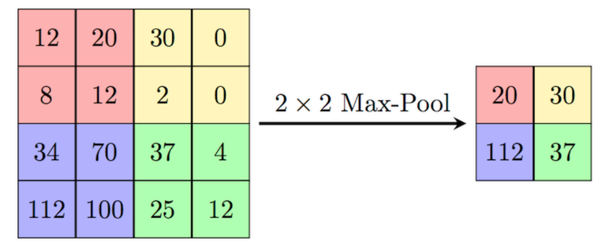


- **locality modelling:** capture local features in 2D space
- **weights sharing**

Convolutional Neural Networks (CNN)

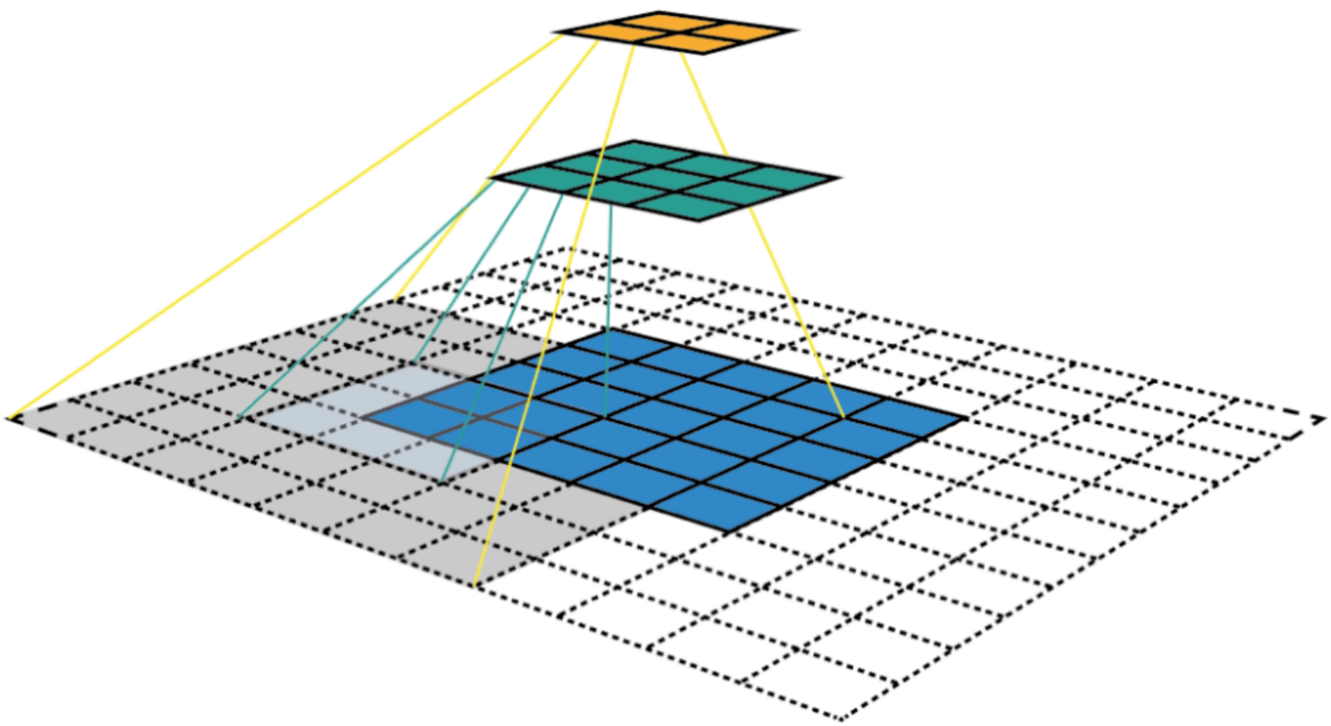
- **Extension #4: stack many convolution layers**
- **Each layer: (**convolution + nonlinear ReLU**)** + **max pooling**
- **Complexity:** $O(d^2 \cdot f^2 \cdot p \cdot k \cdot l)$
- Fully-connected layers at the end: map **locally-extracted features** to targets





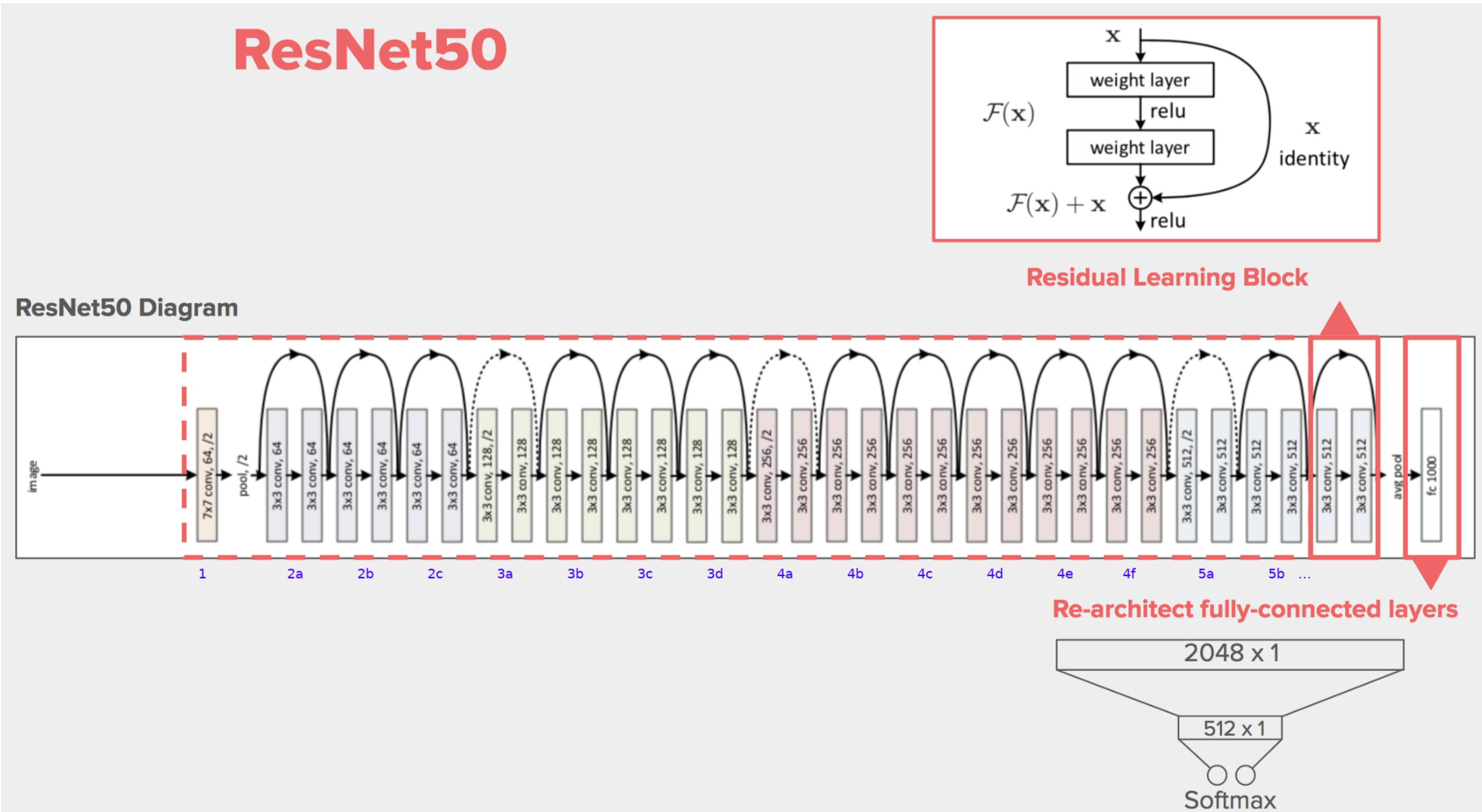
Convolutional Neural Networks (CNN)

- **Locality modeling => hierarchical modeling**
  - recursively combine local features
  - **receptive fields** in CNN: broaden in upper layers



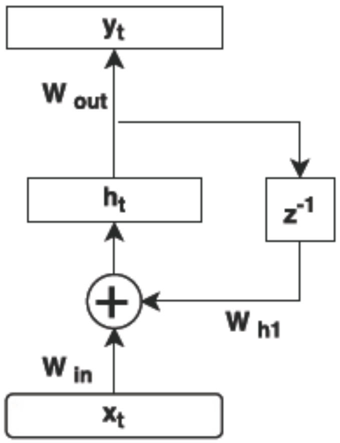
Convolutional Neural Networks (CNN)

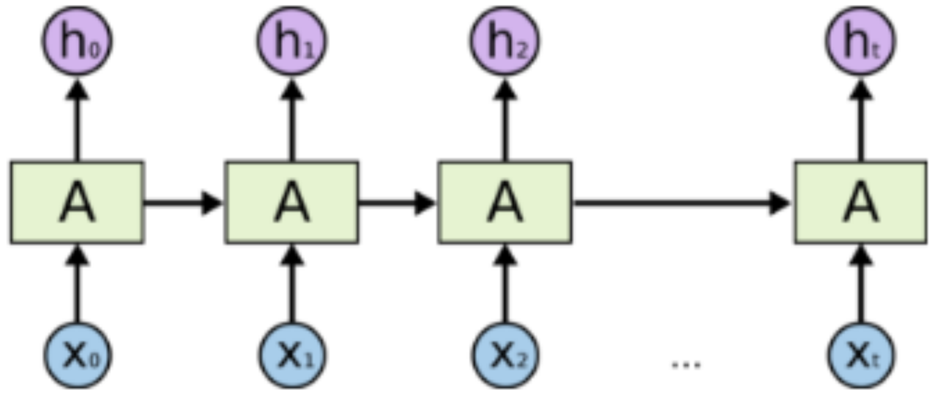
- **ResNet:** a popular CNN architecture for image classification
  - adding short-cut paths to facilitate error backpropagation



Recurrent Neural Networks (RNN)

- **Plain RNNs**



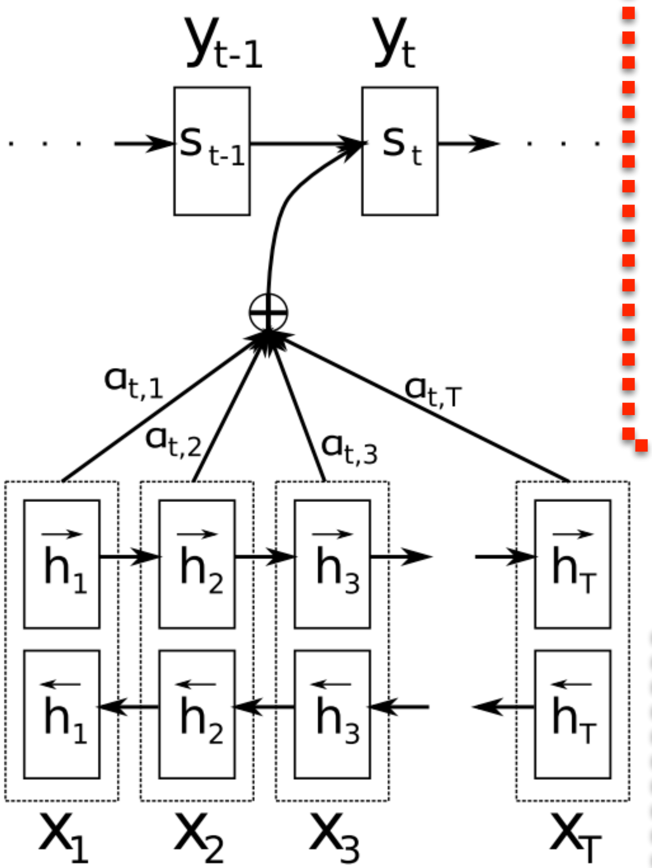


- **RNNs are notoriously hard to learn**
  - **Computationally expensive**
  - **Gradient vanishing or exploding**
- **Long Short-Term Memory (LSTM) and GRU**
- **Higher-order RNNs**

Transformers (I)

- **Transformer:** a non-recurrent structure to model sequences based on *self-attention mechanism*

- **Attention mechanism:** weighting with an attention function



$$e_{tj} = g(\mathbf{s}_{t-1}, \mathbf{h}_j) \implies \mathbf{e}_t = g(\mathbf{s}_{t-1}, [\mathbf{h}_1\ \mathbf{h}_2\ \cdots\ \mathbf{h}_T])$$

$$\alpha_{tj} = \frac{e_{tj}}{\sum_{j=1}^{N} e_{tj}} \implies \boldsymbol{\alpha}_t = \mathrm{softmax}(\mathbf{e}_t)$$

$$\mathbf{c}_t = \sum_{j=1}^{T} \alpha_{tj}\mathbf{h}_j = [\mathbf{h}_1\ \mathbf{h}_2\ \cdots\ \mathbf{h}_T]\,\boldsymbol{\alpha}_t$$

$$\mathbf{s}_t = f(\mathbf{s}_{t-1}, \mathbf{y}_{t-1}, \mathbf{c}_t)$$

$$\mathbf{c}_t = \mathbf{H}\ \mathrm{softmax}(g(\mathbf{s}_{t-1}, \mathbf{H}))$$

$$\implies \mathbf{C} = \mathbf{H}\ \mathrm{softmax}(g(\mathbf{S}, \mathbf{H}))$$

$$\implies \mathbf{C} = \mathbf{V}\ \mathrm{softmax}(g(\mathbf{Q}, \mathbf{K}))$$

V: value matrix
Q: query matrix
K: key matrix

Transformers (II)

- given a sequence of words $w_1 \cdots w_n$
- **word embedding**: map this to a sequence of vectors $\mathbf{x}_1 \cdots \mathbf{x}_n$, where each $\mathbf{x_i}$ is the word embedding for w_i, and $\mathbf{x}_i \in \mathbb{R}^d$ (e.g., $d = 512$)
- **transformers**: a non-recurrent model to map this to a new context-aware sequence $\mathbf{z}_1 \cdots \mathbf{z}_n$, where $\mathbf{z}_i \in \mathbb{R}^d$ and each $\mathbf{z}_i$ takes the whole sequence $w_1 \cdots w_n$ into account
- use matrix $X = [\mathbf{x}_1; \cdots ; \mathbf{x}_n]_{n \times d}$ to represent word embeddings

Transformers (III)

- define $A \in \mathbb{R}^{d \times l}$, $B \in \mathbb{R}^{d \times l}$, $C \in \mathbb{R}^{d \times o}$ to be the transformation parameters:

$$Z = \text{softmax}\Big(XA(XB)^\intercal\Big)XC$$

- intuitions:
  1. value matrix $V = XC \in \mathbb{R}^{n \times o}$: transformed embeddings
  2. attention function $g(Q, K) = QK^\intercal$, where $Q = XA$, $K = XB$, and $XA(XB)^\intercal \in \mathbb{R}^{n \times n}$: $n \times n$ inner-products in l-dimensional space
  3. $\text{softmax}\Big(XA(XB)^\intercal\Big) \in \mathbb{R}^{n \times n}$, where all entries are positive and each row sums to 1
  4. **self-attention**: V, K and Q are all derived from the same input X
  5. $Z \in \mathbb{R}^{n \times o}$: the final context-aware embeddings

Muti-head Transformers

- typically $d = 512$, $o = 64$ ($\implies X \in \mathbb{R}^{n \times 512}$)
- multi-head transformers:

$$A^j, B^j \in \mathbb{R}^{d \times l}, C^j \in \mathbb{R}^{d \times o} \quad (j = 1 \cdots 8)$$

- for $j = 1 \cdots 8$:

$$Z^j \in \mathbb{R}^{n \times 64} = \mathsf{softmax}\Big(XA^j(XB^j)^\mathsf{T}\Big)XC^j$$

- concatenate all heads:

$$Z \in \mathbb{R}^{n \times 512} = \mathsf{concat}\big(Z^1, Z^2, \cdots, Z^8\big)$$

- nonlinearity:

$$Z' = \mathsf{feedforward}\Big(\mathsf{layer\text{-}norm}(X + Z)\Big)$$

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: $y = f(x \mid W)$
- The error function (the objective function)
  - Mean square error (MSE):

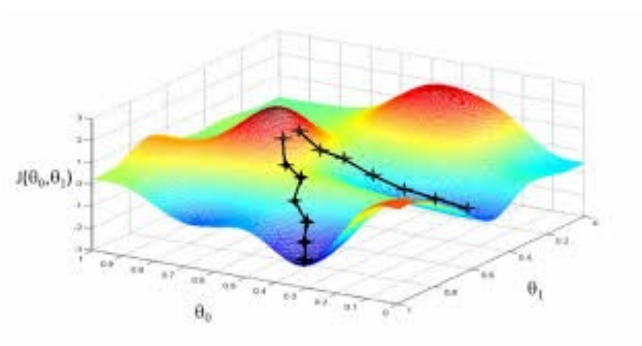
$$Q(\mathbf{W}) = \sum_i \Big(f(\mathbf{x}_i|\mathbf{W}) - t_i \Big)^2$$

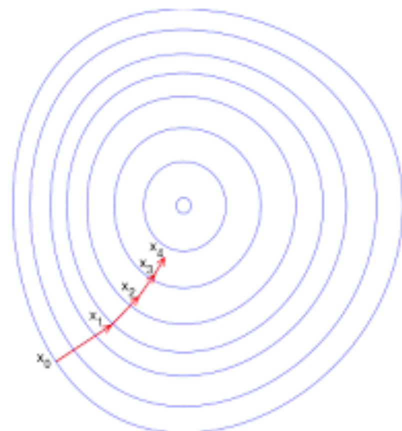
  - Cross entropy error (CE):

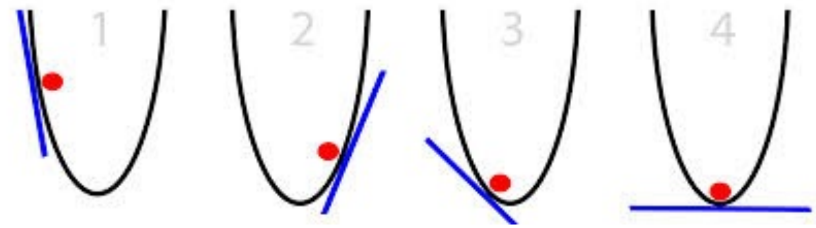
$$Q(\mathbf{W}) = \sum_i \ \mathrm{KL}\Big(\{t_i\} \ || \ \{ f(\mathbf{x}_i|\mathbf{W}) \} \Big) = -\sum_{t=1}^{N} \{ \ln f(\mathbf{x}_t|\mathbf{W}) \}_{l_i}$$

Gradient Descent

- **Gradient Descent: hill-climbing**





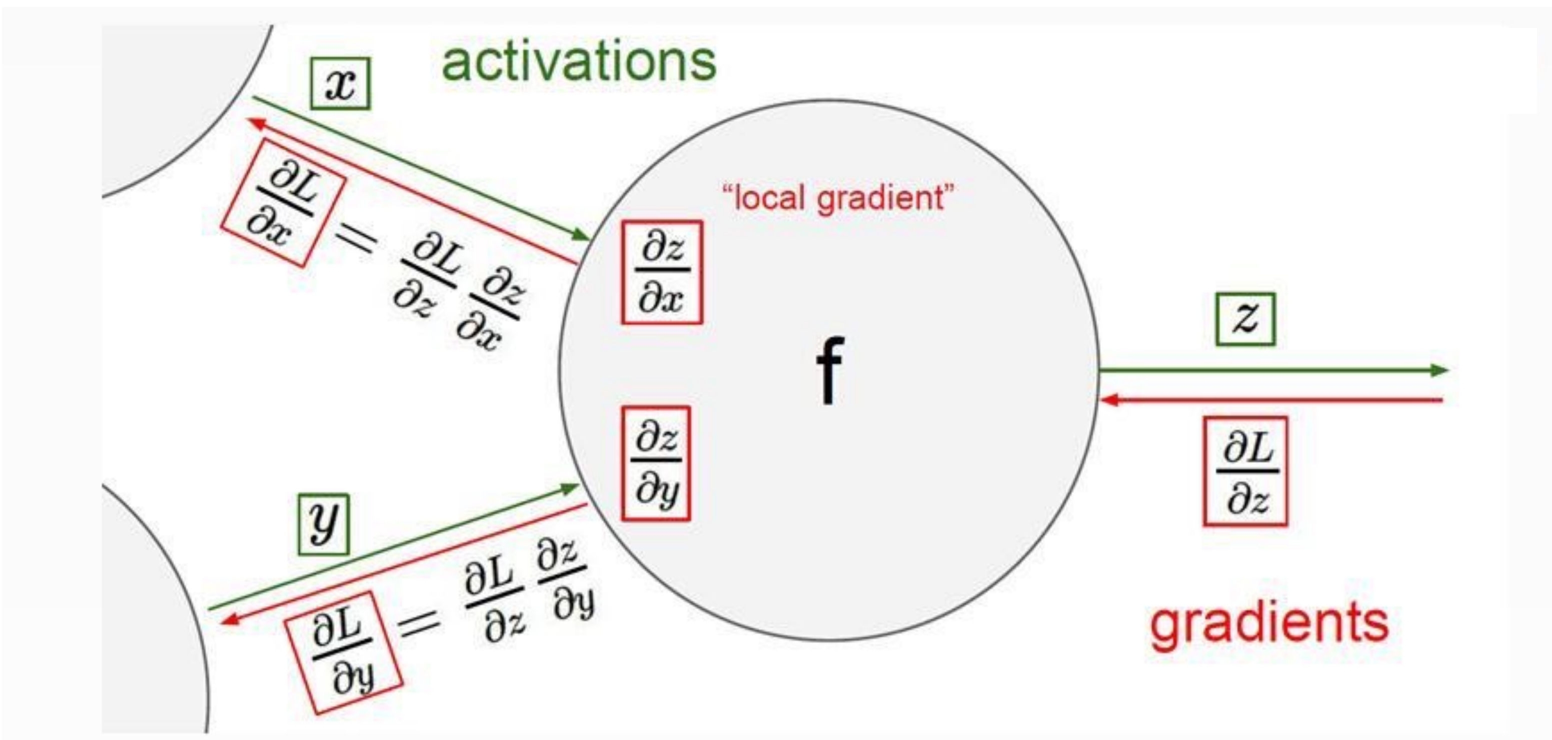


- **Iteratively update network based on the gradient**

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

Error Back-propagation (BP)

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



Auto Differentiation for various layers *f(x)*

- **Fully-connected linear layers:** $\mathbf{y} = W\mathbf{x} + \mathbf{b}$
- **Convolution layers:** $\mathbf{y} = \mathbf{x} * \mathbf{w}$
- **Attention layers:** $\mathbf{C} = \mathbf{V}\ \text{softmax}\big(\mathbf{g}(\mathbf{Q}, \mathbf{K})\big)$

- **ReLU layers:** $\mathbf{y} = \text{ReLU}(\mathbf{x})$
- **Sigmoid layers:** $\mathbf{y} = \text{sigmoid}(\mathbf{x})$
- **Soft-max layers:** $\mathbf{y} = \text{softmax}(\mathbf{x})$
- **Max-pooling layers:** $\mathbf{y} = \text{maxpooling}_{m\times m}(\mathbf{x})$
- **Time-delayed layers:** $\mathbf{y} = z^{-1}(\mathbf{x})$

Auto-Differentiation Examples

- **Fully-connected linear layers:** $\mathbf{a} = \mathbf{W}\mathbf{z} + \mathbf{b}$

$$(0)\ \text{error signal} : \mathbf{e} = \frac{\partial Q(\cdot)}{\partial \mathbf{a}}$$
$$(1)\ \text{error backpropagation} : \frac{\partial Q(\cdot)}{\partial \mathbf{z}} = \mathbf{W}^\intercal \mathbf{e}$$
$$(2)\ \text{gradients of } \mathbf{W}, \mathbf{b} : \frac{\partial Q(\cdot)}{\partial \mathbf{W}} = \mathbf{z}^\intercal \mathbf{e}, \quad \frac{\partial Q(\cdot)}{\partial \mathbf{b}} = \mathbf{e}$$

- **Sigmoid layers:** $\mathbf{z} = \mathbf{sigmoid}(\mathbf{a}) = l(\mathbf{a})$

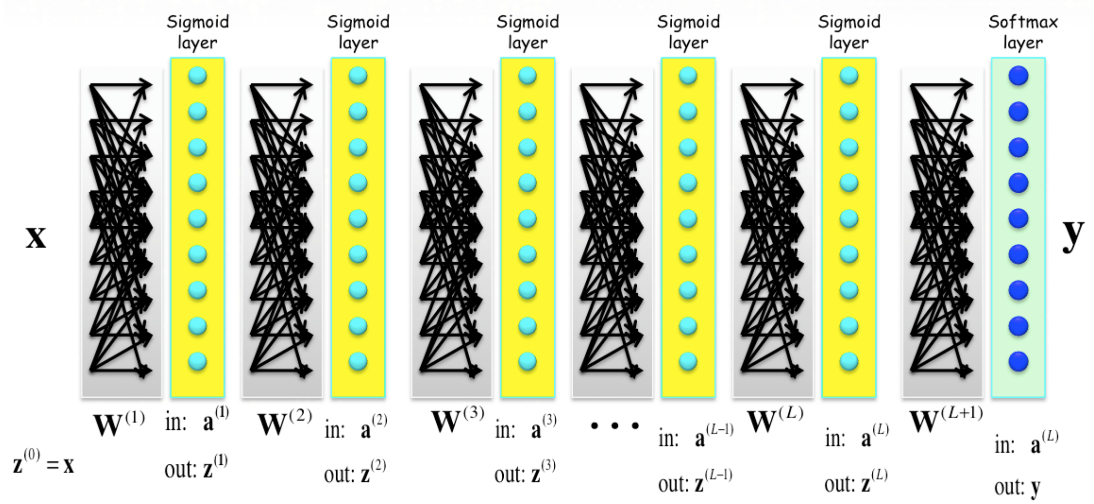
$$(0)\ \text{error signal} : \mathbf{e} = \frac{\partial Q(\cdot)}{\partial \mathbf{z}}$$
$$(1)\ \text{error backpropagation} : \frac{\partial Q(\cdot)}{\partial \mathbf{a}} = l(\mathbf{z}) \odot (1 - l(\mathbf{z})) \odot \mathbf{e}$$

- **Soft-max layers:** $\mathbf{y} = \mathbf{softmax}(\mathbf{a}) \implies y_i = \frac{e^{a_i}}{\sum_k e^{a_k}} \quad (\forall i)$

$$(0)\ \text{error signal} : \mathbf{e} = \frac{\partial Q(\cdot)}{\partial \mathbf{y}}$$
$$(1)\ \text{error backpropagation} : \frac{\partial Q(\cdot)}{\partial \mathbf{a}} = \mathbf{J_s}\mathbf{e}$$
$$\text{with } [\mathbf{J_s}]_{ij} = \begin{cases} y_i(1 - y_i) & \text{if} \quad i = j \\ y_i y_j & \text{if} \quad i \neq j \end{cases}$$

Error Back-propagation (BP)

- **Multi-layer fully-connected feedforward structure**
  - **Sigmoid activation**
  - **Cross-entropy error**



Given one sample $\{\mathbf{x}, l\}$, we have $Q(\mathbf{W}) = -\ln y_l$, and Error Back-propagation works as follows:

- Softmax layer $l = L + 1$:

$$e_k^{(L+1)} = -\frac{1}{y_l} \frac{\partial y_l}{\partial a_k^{(L+1)}} = y_k - \delta(k - l)$$

- Sigmoid + Fully-connected layers $l = L, \cdots, 2, 1$:

$$e_k^{(l)} = z_k^{(l)}(1 - z_k^{(l)}) \sum_i e_i^{(l+1)} W_{ik}^{(l+1)}$$

$$\frac{\partial Q(\mathbf{W})}{W_{ik}^{(l+1)}} = e_i^{(l+1)} z_k^{(l)}$$

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 $x_i \longrightarrow y_i$
  - Accumulate error for the mini-batch Q_i
  - Backward pass: back-propagate error Q_i to compute gradients
  - Update network weights: $$\mathbf{W}^{(l+1)} = \mathbf{W}^{(l)} - \epsilon \cdot \frac{\partial Q(\mathbf{W})}{\partial \mathbf{W}}\bigg|_{\mathbf{W}=\mathbf{W}^{(l)}}$$

Neural Networks Learning in practice

- Open source toolkits: Tensorflow, pyTorch, CNTK, MXNet, etc ...
- Computationally intensive (GPUs)
- Many parameters tuning tricks:
  - Network initialization / mini-batch size/ epoch
  - Learning rates (annealing schedule)
  - Weight Decay (L2 norm regularization)
  - Momentum
  - Dropout, data augmentation
  - Batch Normalization, layer normalization, weight 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:**

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

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:**

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

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

Image 2: SGD without momentum

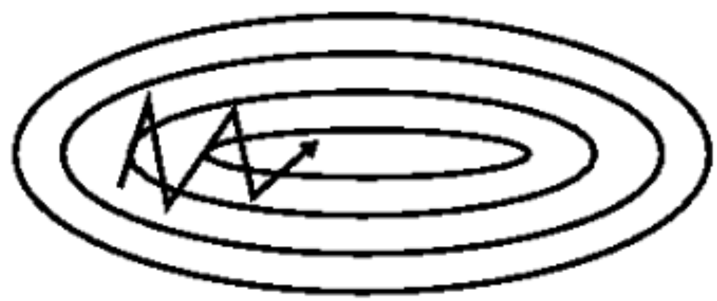
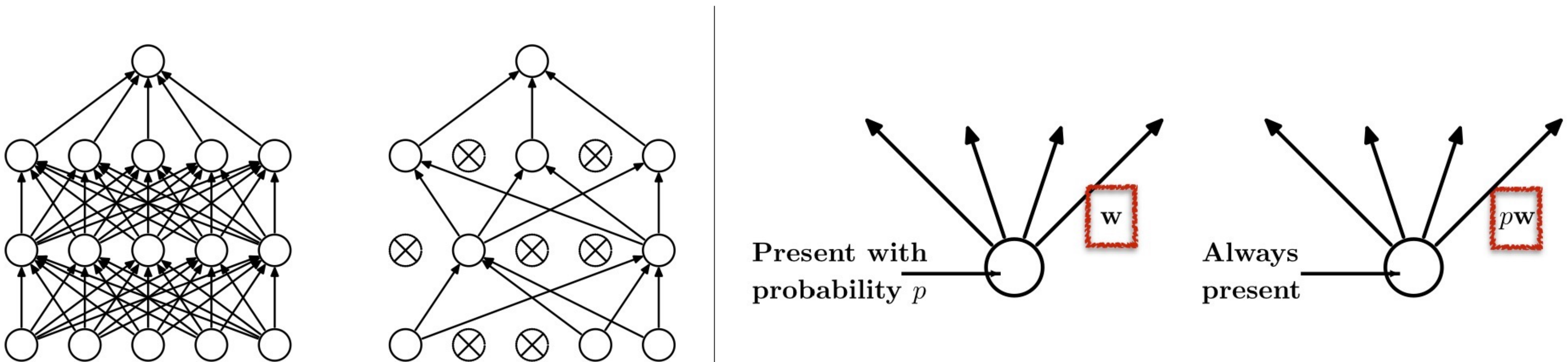


Image 3: SGD with momentum

Dropout

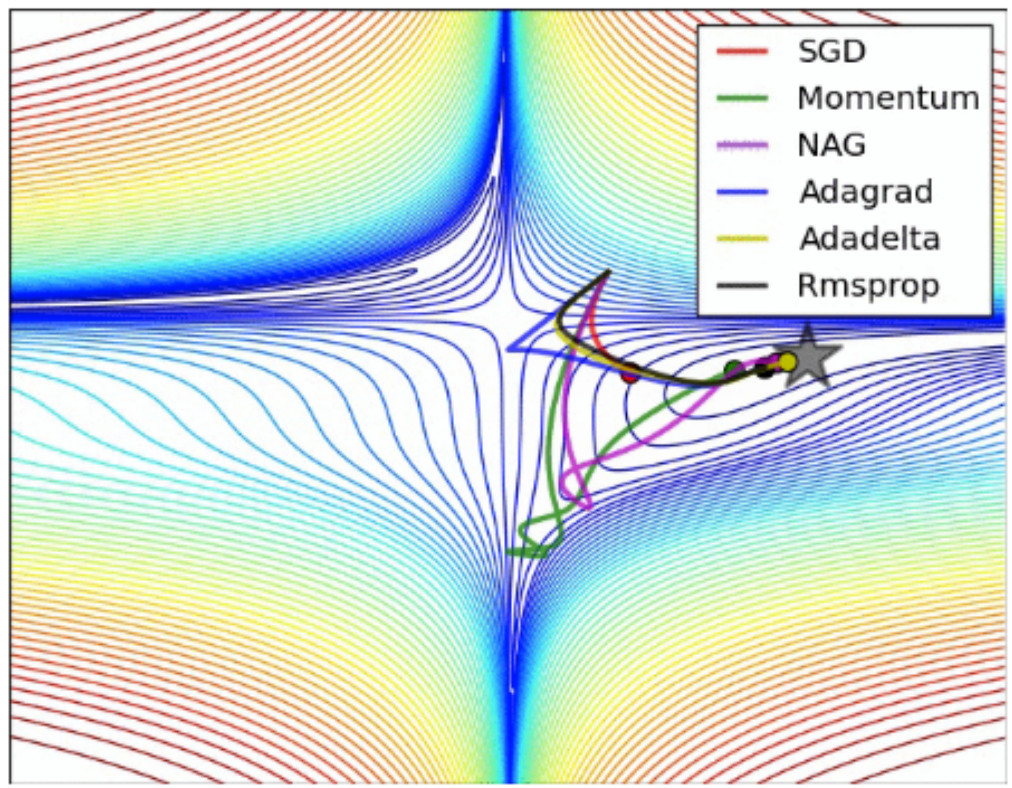
- Dropbox is a simple regularization technique.
- Randomly drop-out some nodes in training.
- Equivalent to adding noises in training
- A relevant technique: *data augmentation*



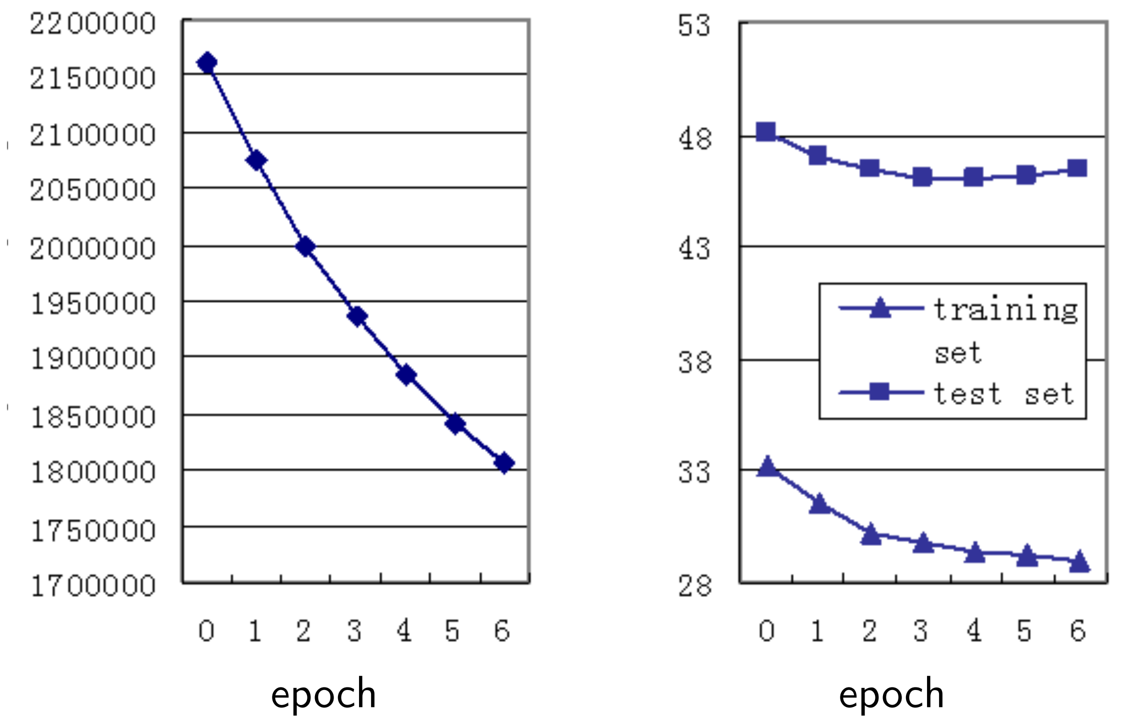
(a) Standard Neural Net (b) After applying dropout. (c) At training time (d) At test time

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 …**

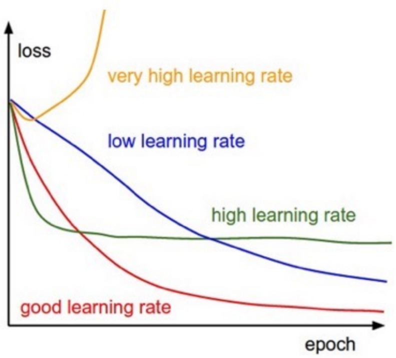


Figure 1

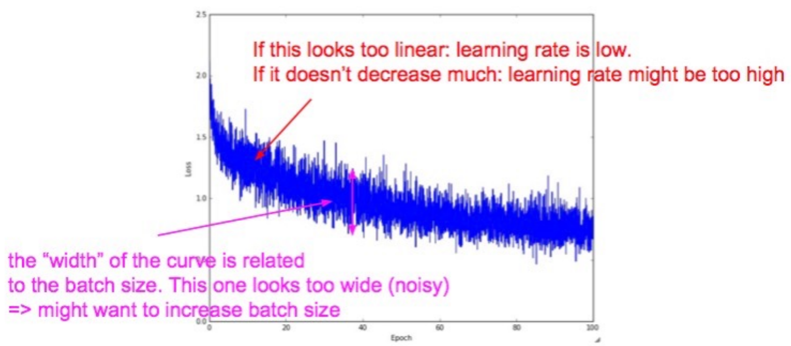


Figure 2

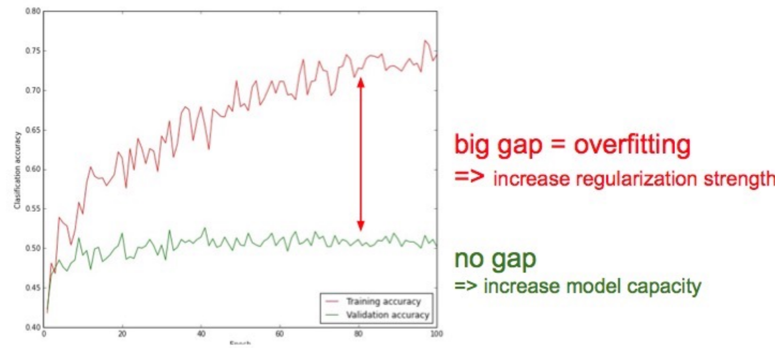
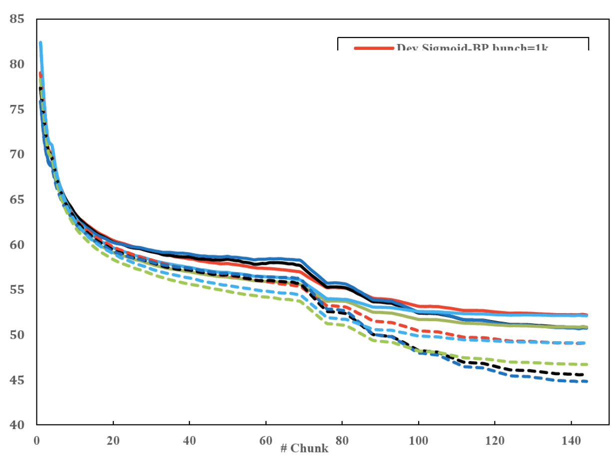


Figure 3



Neural Networks Learning in practice

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

Advanced Topics in Deep Learning

- LSTMs, GRUs, higher-order RNNs, FSMNs ...
- Sequence to sequence modeling
- Bottleneck features
- Unsupervised learning:
  - Restricted Boltzmann Machine (RBM)
  - (De-noising) Auto-Encoder
  - Generative Adversarial Networks