Topic 9

Deep Learning for Audio

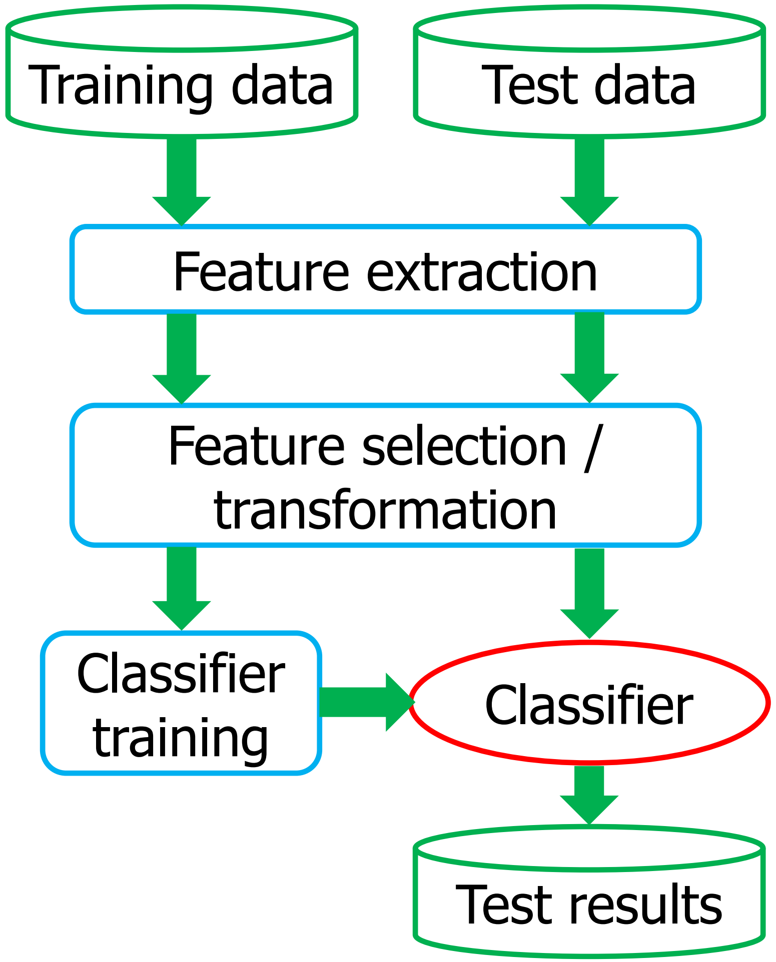
Some figures are copied from the following books

- **LWLS** - Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten, Thomas B. Schön, *Machine Learning: A First Course for Engineers and Scientists*, Cambridge University Press, 2022.
- **GBC** - Ian Goodfellow, Yoshua Bengio, and Aaron Courville, Deep Learning, MIT Press.
- **Mitchell** - Tom M. Mitchell, Machine Learning, McGraw-Hill Education, 1997.

Audio Classification Tasks

- Music genre, mood, artist, composer, instrument classification
- Auto tagging, i.e., labeling music with words
- Chord recognition
- Acoustic event detection
- Speech/speaker recognition

- General flowchart

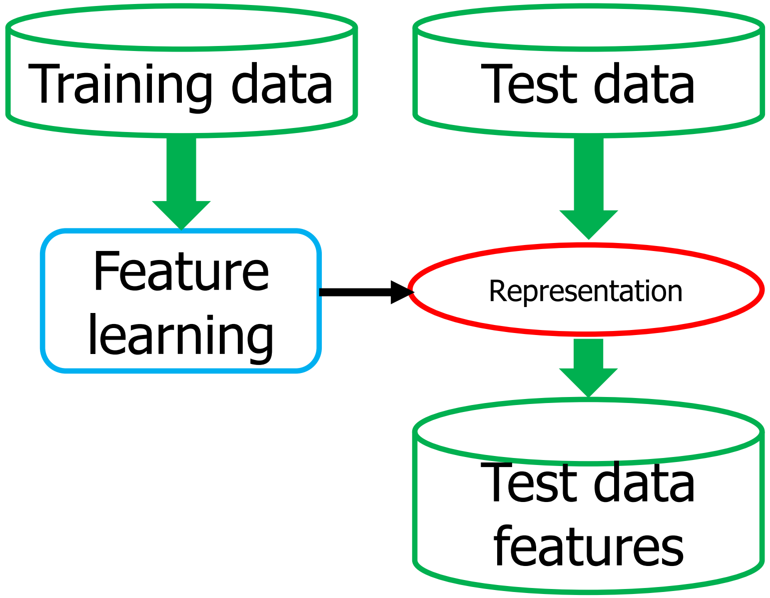


Features that we have studied

- Raw input: audio waveform or spectrogram
- Feature output:
  - RMS, Zero Crossing Rate
  - Spectral centroid, spread, skewness, kurtosis, flatness, irregularity, roll-off, flux, etc.
  - Harmonic features
  - MFCC, LPC, PLP, etc.
- Hand-crafted / engineered / pre-defined
- Hard to decide what features to use for a task
- Question: can computers learn features directly from data?

Feature / Representation Learning

- Learn a transformation from "raw" inputs to a representation that can be effectively exploited in a task



- Automatic / does not rely on human knowledge
- Target for a specific task

Methods Viewed as Feature Learning

- Principal Component Analysis (PCA)
  - Learns a linear transformation, where rows of $\boldsymbol{W}$ are the orthogonal directions of greatest variance in the training data

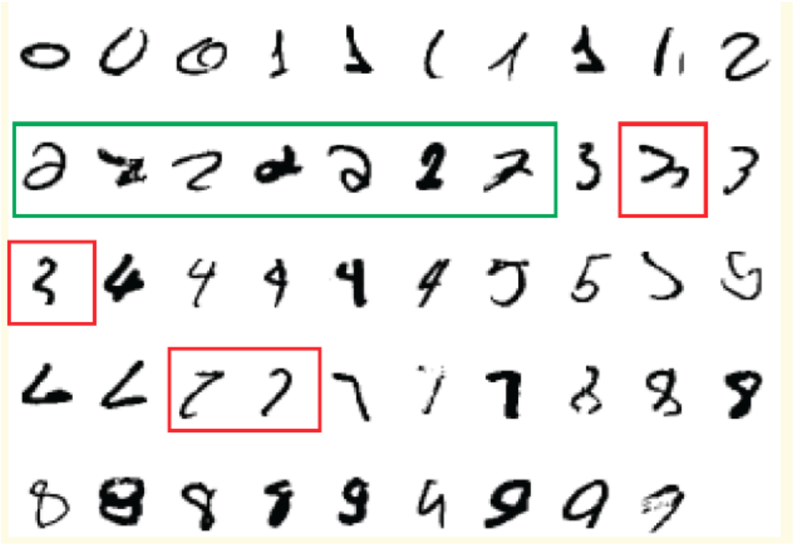
$$f(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

- Dictionary Learning (e.g., NMF)
  - Learns a linear transformation, where the input, transformation matrix, and activation matrix (i.e., features), are all non-negative

$$\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h}$$
$$(\boldsymbol{X} = \boldsymbol{W}\boldsymbol{H})$$

Are linear features good enough?

- Probably not…
- The world is complex and often highly nonlinear.



Can you define a linear transformation on the images to discriminate "2"s from non-"2"s?

$$f(\boldsymbol{x}) = \sum_i w_i x_i + b$$

where $\boldsymbol{x}$ is a vector of pixel values of an image.

Are these features highly nonlinear…

…to the waveform or spectrogram?

- RMS, ZCR
- Spectral centroid, spread, skewness, kurtosis, flatness, flux
- Harmonic features
- Cepstrum: $|\mathcal{F}^{-1}\{\log|\mathcal{F}\{x(t)\}|^2\}|^2$
- MFCC
- LPC

Can we learn highly non-linear features?

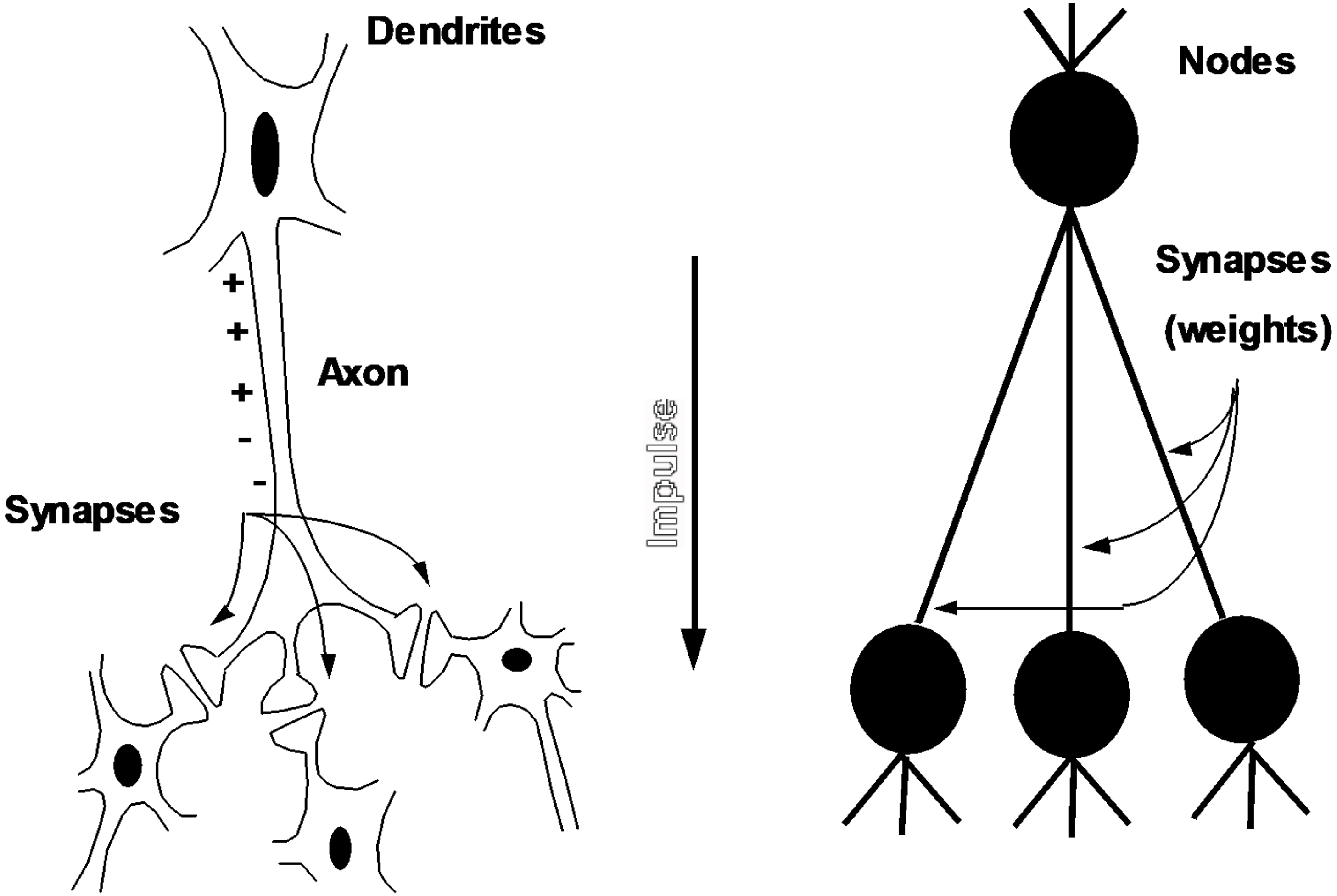
Deep neural networks!

Biological Motivation

- Human brain: a densely interconnected network
  - ~10^11 neurons
  - Each neuron connects to ~10^4 other neurons
  - Two states of neuron activity: excited vs. inhibited
  - Neuron switching speed: ~1kHz
    - CPU clock frequency: GHz
  - Yet many tasks (e.g., face recognition) can be completed within 0.1 s

- This suggests
  - Highly parallel processing
  - Distributed representations

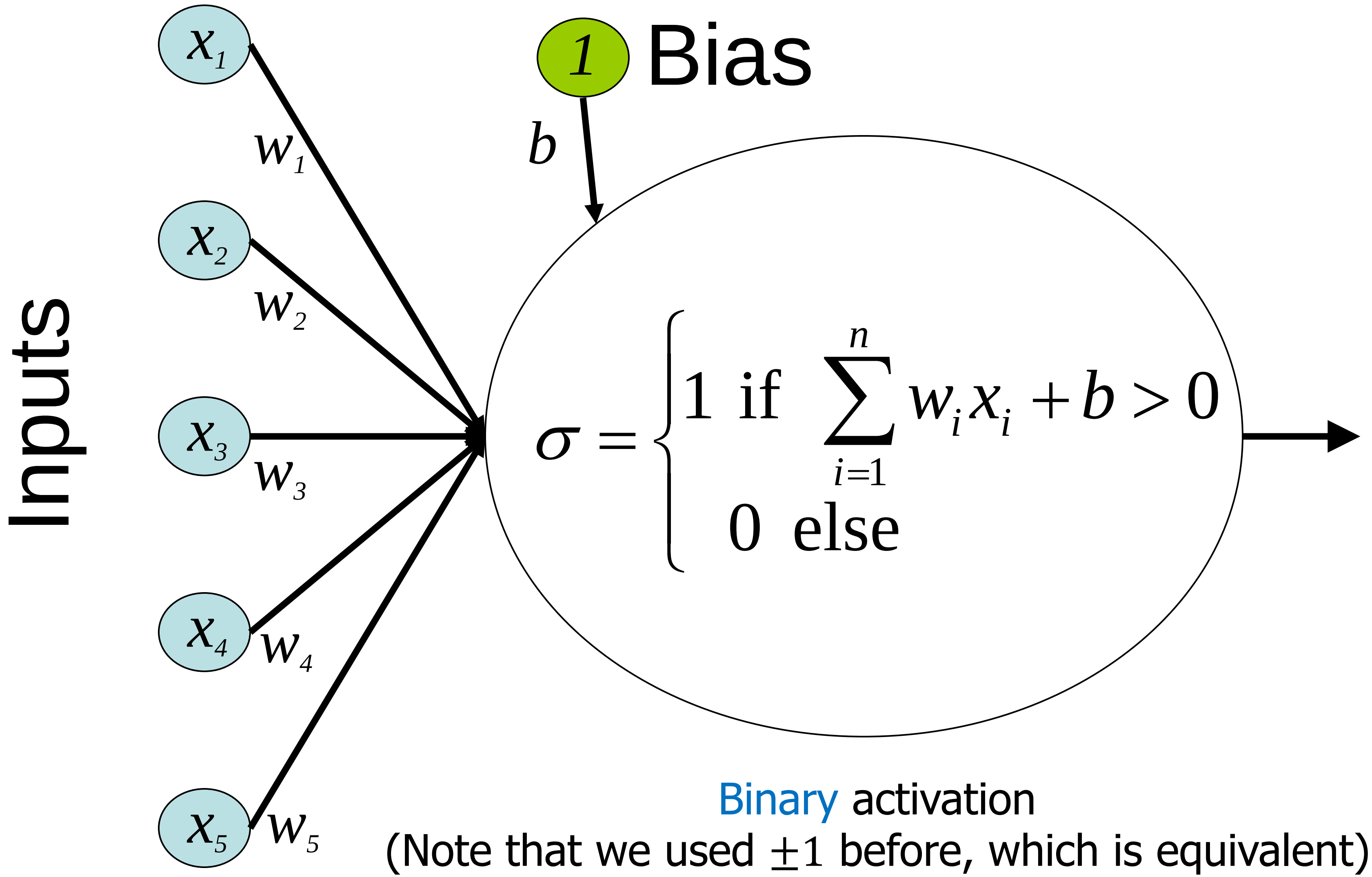
Biological Analogy



History of Neural Networks

- 1943 – first neural network computing model by McCulloch and Pitts
- 1958 – Perceptron by Rosenblatt
- 1960's – a big wave
- 1969 – Minsky & Papert's book "Perceptrons"
- 1970's – "winter" of neural networks
- 1975 – Backpropagation algorithm by Werbos
- 1980's – another big wave
- 1990's – overtaken by SVM proposed in 1993 by Vapnik
- 2006 – a fast learning algorithm for training deep belief networks by Hinton
- 2010's – another big wave
- 2018 – Turing Award to Hinton, Bengio & LeCun
- 2022 – ChatGPT!
- Present – continue to transform various domains

Perceptron



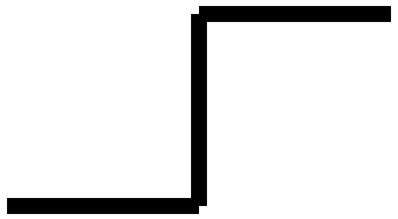
Binary activation
(Note that we used ± 1 before, which is equivalent)

Nonlinear Activation Functions

- Step function

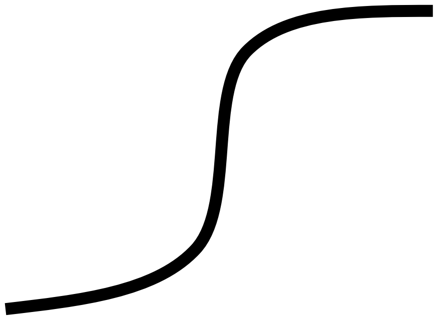
$$output = sign(\boldsymbol{w}^T \boldsymbol{x} + b)$$

  - Note: previously we used {-1,1} for sign function for perceptron, which is equivalent



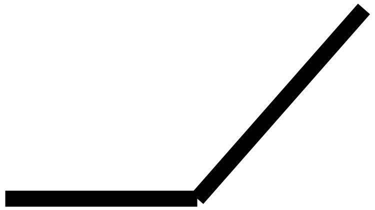
- Sigmoid function

$$output = \sigma(\boldsymbol{w}^T \boldsymbol{x} + b) = \frac{1}{1 + e^{-(\boldsymbol{w}^T \boldsymbol{x} + b)}}$$



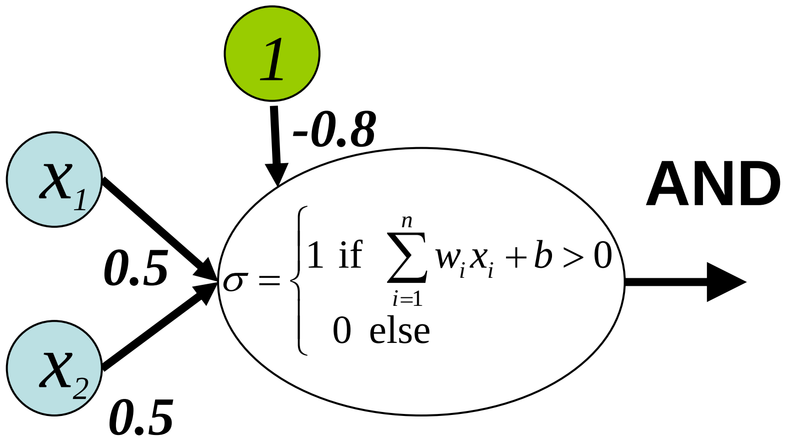
- Rectified Linear Unit (ReLU)

$$output = \max\{0, \boldsymbol{w}^T \boldsymbol{x} + b\}$$

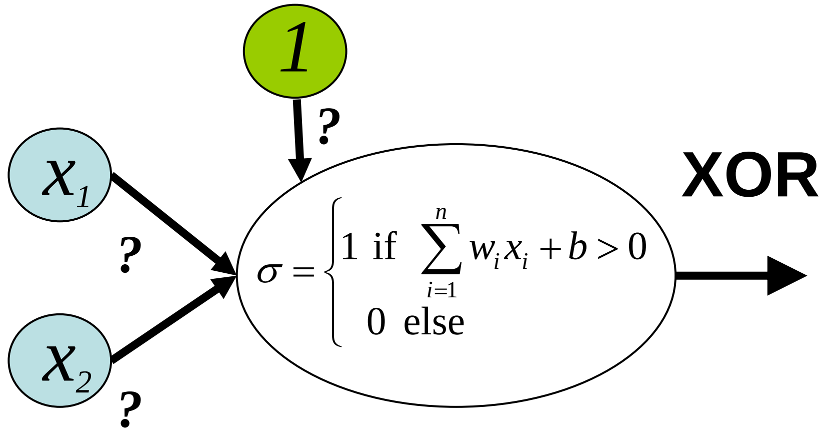


Limitations of 1-layer Nets

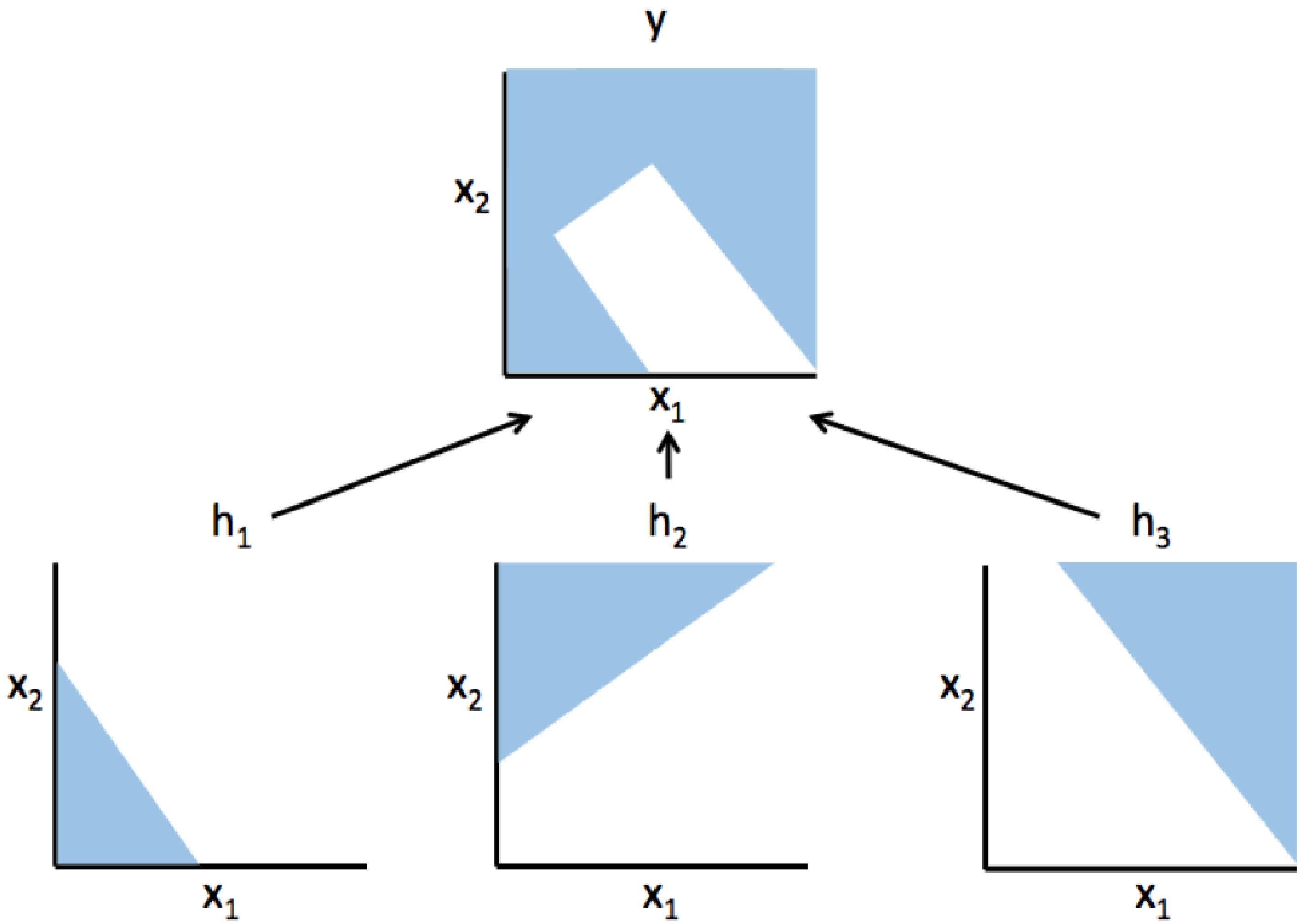
- Only express linearly separable cases
  - For example, they are good as logic operators "AND", "NOT", and "OR"



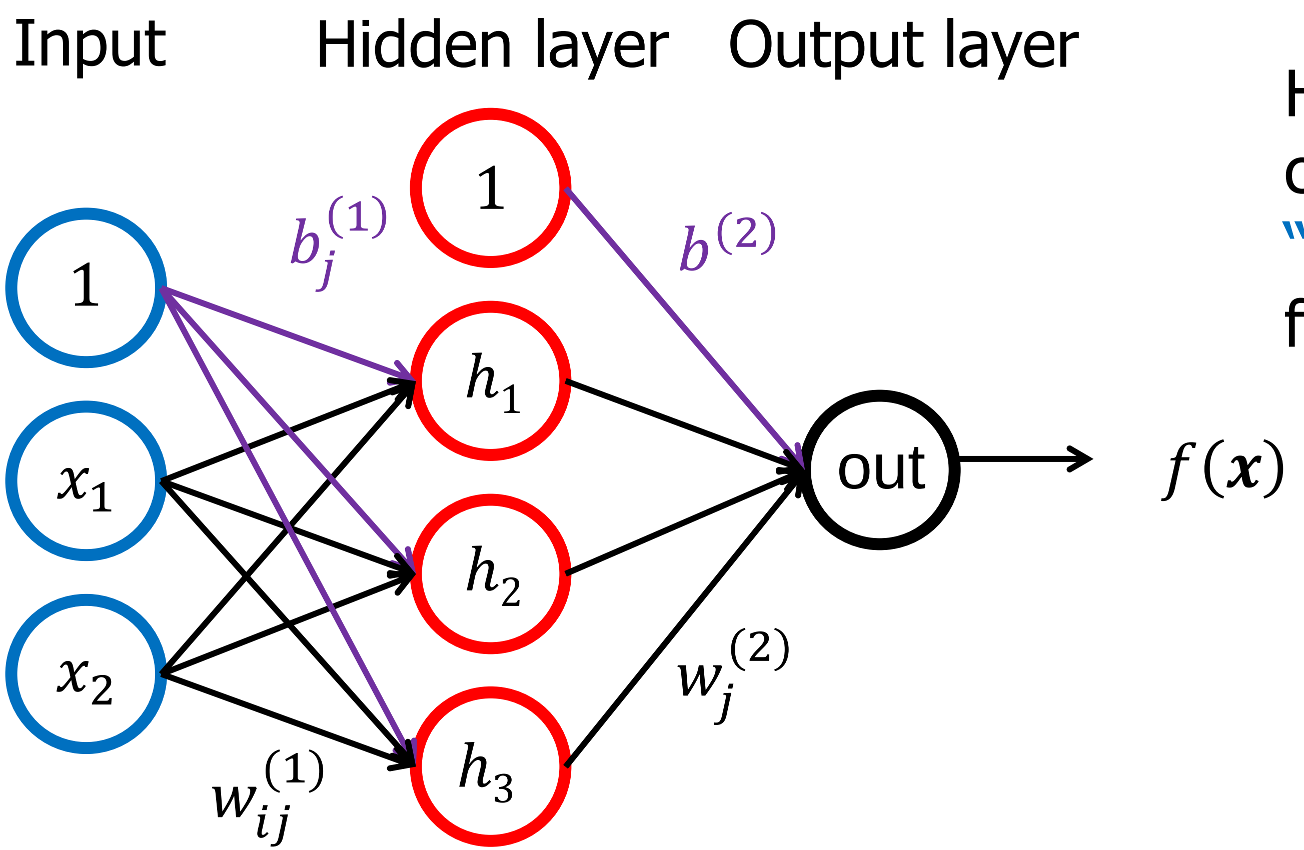
- Cannot represent "XOR", which is not linearly separable



But, we can combine them!



2-layer Nets



Hidden layer output can be viewed as "features" calculated from the raw input

$$f(\boldsymbol{x}) = \sigma\left(\sum_j w_j^{(2)} h_j + b^{(2)}\right) = \sigma\left(\sum_j w_j^{(2)} \sigma\left(\sum_i w_{ij}^{(1)} x_i + b_j^{(1)}\right) + b^{(2)}\right)$$

Matrix Notation

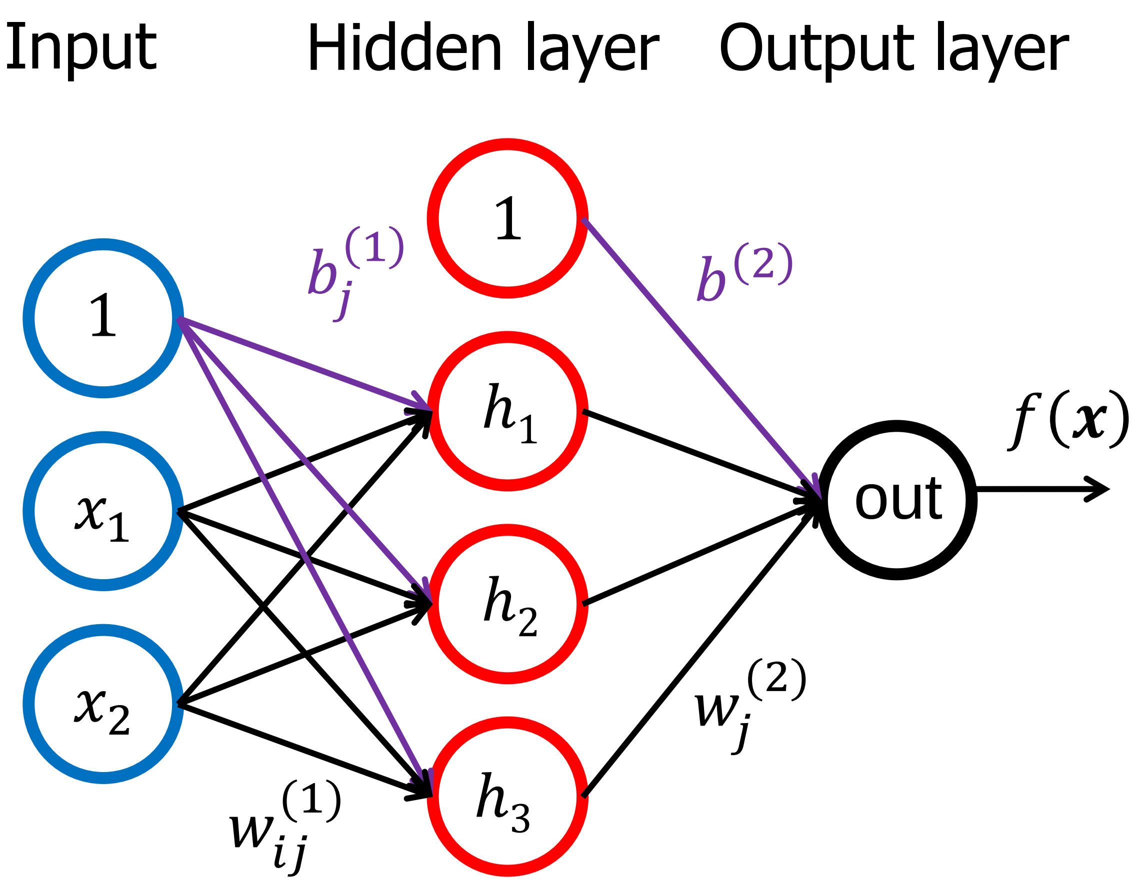
$$f(\boldsymbol{x}) = \sigma\left(\sum_j w_j^{(2)}\,\sigma\left(\sum_i w_{ij}^{(1)}\,x_i + b_j^{(1)}\right) + b^{(2)}\right)$$

$$f(\boldsymbol{x}) = \sigma\left(\boldsymbol{W}_2^T \boldsymbol{\sigma}\left(\boldsymbol{W}_1^T \boldsymbol{x} + \boldsymbol{b}_1\right) + b_2\right)$$

where

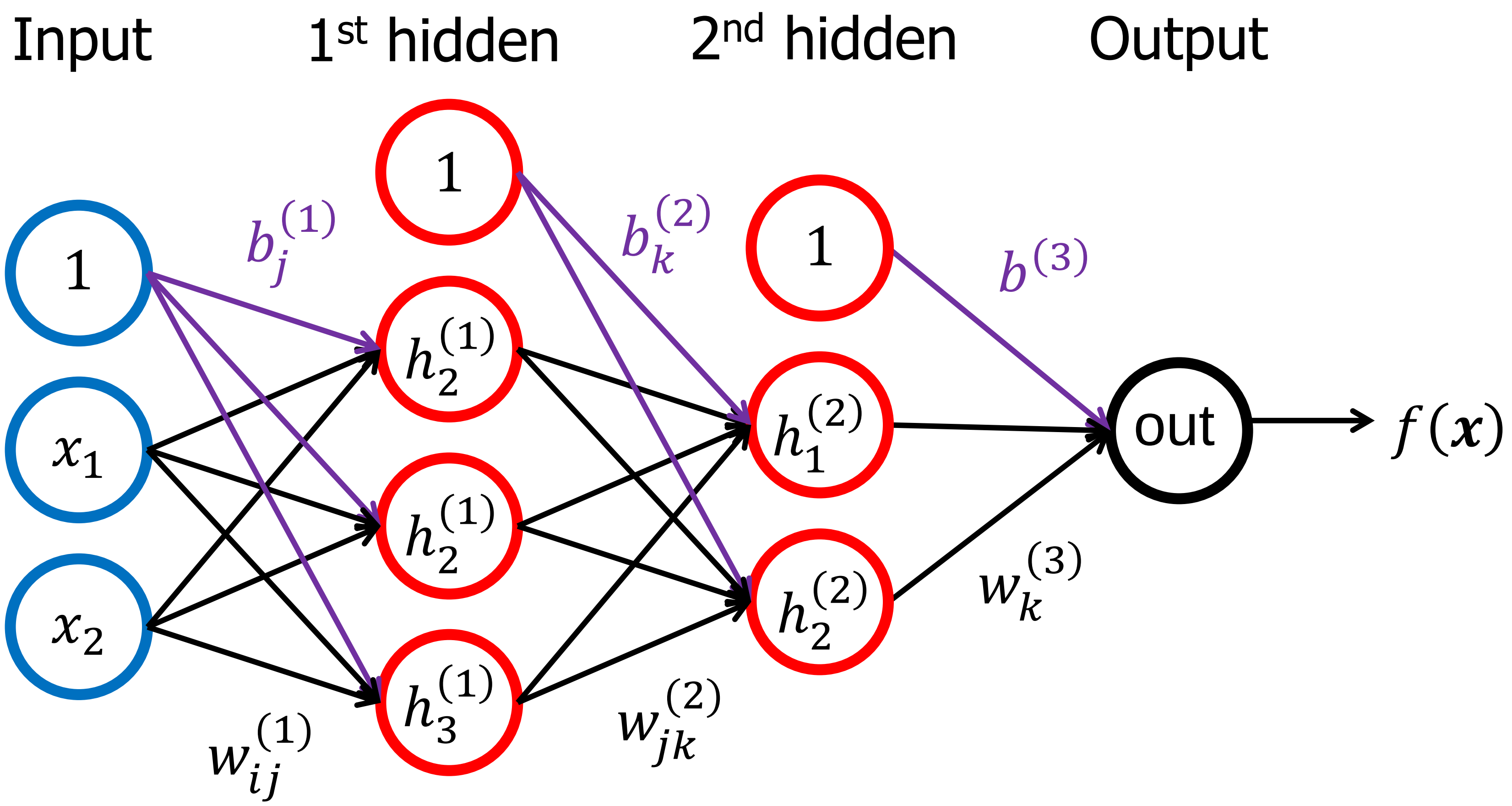
$$\boldsymbol{W}_1 = \left[w_{ij}^{(1)}\right]_{d\times l_1}, \boldsymbol{b}_1 = \left[b_j^{(1)}\right]_{l_1\times 1}$$

$$\boldsymbol{W}_2 = \left[w_{jk}^{(2)}\right]_{l_1\times l_2}, b_2 = b^{(2)}$$



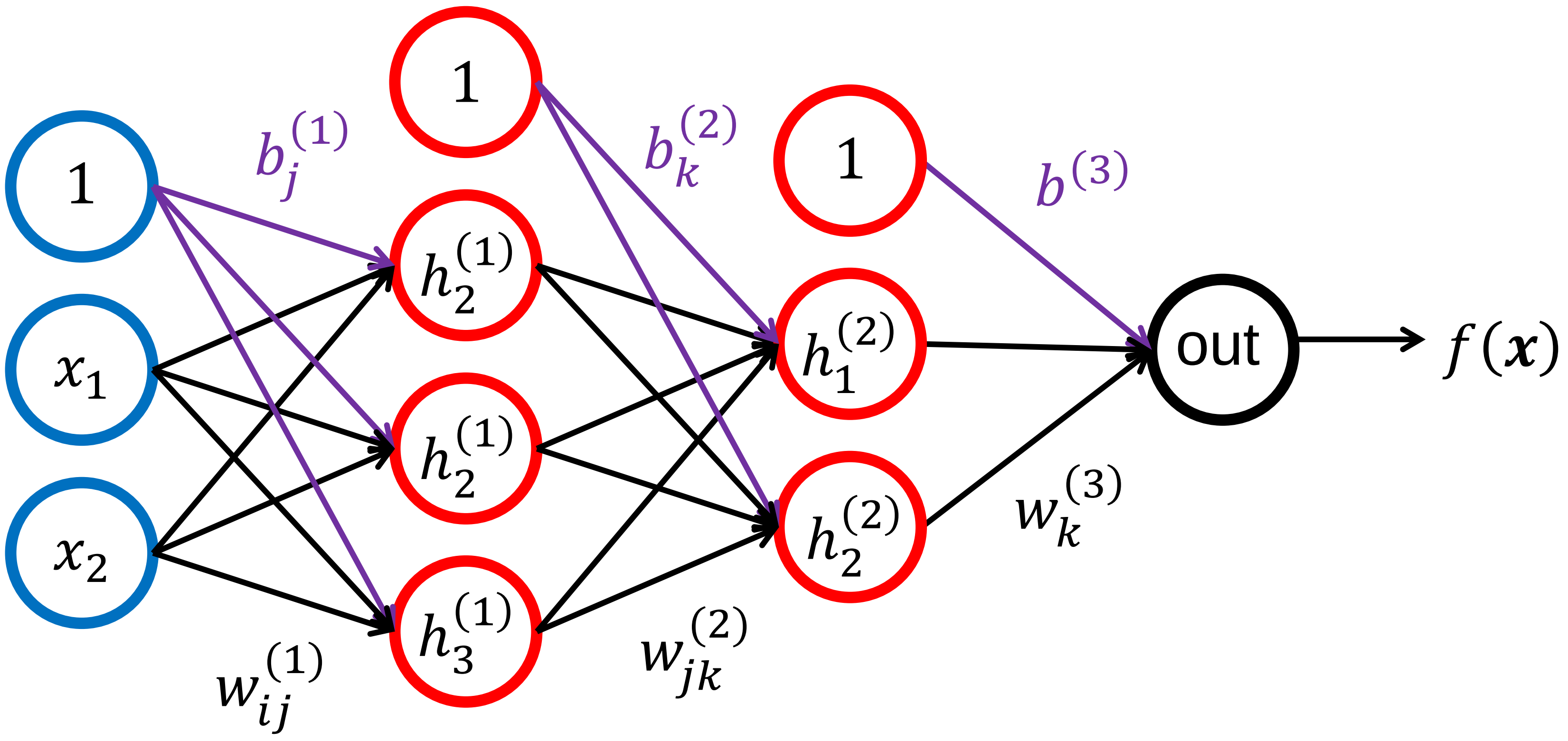
- What does $\boldsymbol{W}_1^T \boldsymbol{x}$ compute?
  - Inner products between columns of $\boldsymbol{W}_1$ and $\boldsymbol{x}$
  - Columns of $\boldsymbol{W}_1$ are “receptors” or “filters”
  - $\boldsymbol{W}_1^T \boldsymbol{x}$ are their responses to input

3-layer Nets



$$f(\boldsymbol{x}) = \sigma\left(\sum_k w_k^{(3)} h_k^{(2)} + b^{(3)}\right) = \sigma\left(\sum_k w_k^{(3)} \sigma\left(\sum_j w_{jk}^{(2)} h_j^{(1)} + b_k^{(2)}\right) + b^{(3)}\right) = \sigma\left(\sum_k w_k^{(3)} \sigma\left(\sum_j w_{jk}^{(2)} \sigma\left(\sum_i w_{ij}^{(1)} x_i + b_j^{(1)}\right) + b_k^{(2)}\right) + b^{(3)}\right)$$

Matrix Notation



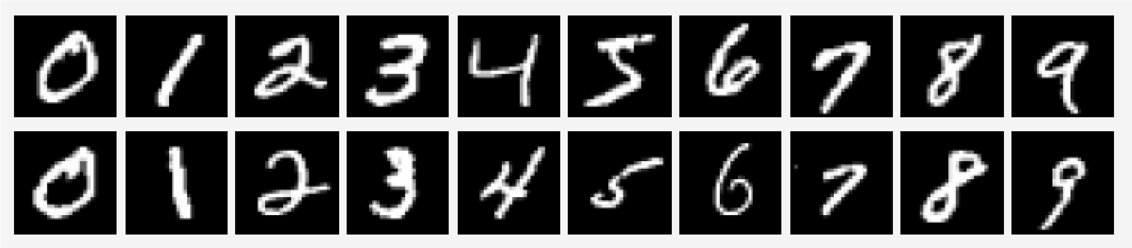
$$f(\boldsymbol{x}) = \sigma\left(\sum_k w_k^{(3)} \sigma\left(\sum_j w_{jk}^{(2)} \sigma\left(\sum_i w_{ij}^{(1)} x_i + b_j^{(1)}\right) + b_k^{(2)}\right) + b^{(3)}\right)$$

$$f(\boldsymbol{x}) = \sigma\left(\boldsymbol{W}_3^T \boldsymbol{\sigma}\left(\boldsymbol{W}_2^T \boldsymbol{\sigma}\left(\boldsymbol{W}_1^T \boldsymbol{x} + \boldsymbol{b}_1\right) + \boldsymbol{b}_2\right) + b_3\right)$$

Richer Representations with More Layers

- 1-layer nets (e.g., perceptron) only model linear hyperplanes
- 2-layer nets can approximate any continuous function, given enough hidden nodes
- >=3-layer nets can do so with fewer nodes and weights
- Nonlinear activation is key!
  - Multiple layers of linear activations is still linear!

Example Application



(Fig. 6.5 in LWLS, from MNIST dataset)
70,000 grayscale images (28*28) from 10 classes

- One-layer MLP (i.e., logistic regression)
  - Input: 28*28=784-d vectors
  - Output layer size: 10 nodes
  - #parameters: 784*10+10 = 7,850

- Two-layer MLP
  - Input: 28*28=784-d vectors
  - Hidden layer size: 200 nodes
  - Output layer size: 10 nodes
  - #parameters for hidden layer: 784*200+200
  - #parameters for output layer: 200*10+10
  - #Total parameters = 159,010

Properties of NNs

- Large capacity: able to learn complex relations between input and output
- Support various data formats: continuous, discrete, categorical (needs to be encoded into numeric)
- Robust to some level of noise in training data
- Inference (i.e., making predictions on test examples) is fast

- Data hungry
- Training is slow
- Lack of mathematical analysis and difficult to interpret

How to learn the weights?

- Given training data - input and label pairs $\{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i=1}^{N}$
- Update network weights to minimize the difference (error) between $f\left(\boldsymbol{x}^{(i)}\right)$ and $y^{(i)}$
  - Calculate derivative of error w.r.t. weights
  - Gradient descent to update weights
  - Backpropagation algorithm: recursive computation of these gradients

- See derivation on white board

Backpropagation Recap

- Assume we use sigmoid activation and the squared error loss
  - We can also use other activations, e.g., ReLU
  - We can also use other losses, e.g., cross entropy
- Then the loss on the entire training set is

$$E(\boldsymbol{\theta}) = \frac{1}{2N}\sum_{i=1}^{N}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 = \frac{1}{2N}\sum_{i=1}^{N}\left(y^{(i)} - f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})\right)^2$$

where $\boldsymbol{\theta}$ denotes network parameters, i.e., network weights

- We compute gradient $\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta})$ (called the true gradient, versus stochastic gradient computed on a subset of data), and then update $\boldsymbol{\theta}$ along the negative gradient direction iteratively
- The computation of $\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta})$ is recursive, backward from the last layer to the first layer, leveraging the layer-wise structure of the network
- The computation also requires node outputs at each layer, which are computed in a forward pass

Forward Pass In Matrix Notation

- Start from input $\boldsymbol{X}_{N\times d} = \left[\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \cdots, \boldsymbol{x}^{(N)}\right]^T$ corresponding to all N points

- Compute first hidden layer net input $\boldsymbol{Z}_1$
$$[\boldsymbol{Z}_1]_{N\times l_1} = [\boldsymbol{X}\boldsymbol{W}_1]_{N\times l_1} + [repmat(\boldsymbol{b}_1^T)]_{N\times l_1}$$
- Compute first hidden layer output $\boldsymbol{H}_1$
$$[\boldsymbol{H}_1]_{N\times l_1} = \boldsymbol{\sigma}(\boldsymbol{Z}_1)$$

- Compute second hidden layer net input $\boldsymbol{Z}_2$
$$[\boldsymbol{Z}_2]_{N\times l_2} = [\boldsymbol{H}_1\boldsymbol{W}_2]_{N\times l_2} + [repmat(\boldsymbol{b}_2^T)]_{N\times l_2}$$
- Compute second hidden layer output $\boldsymbol{H}_2$
$$[\boldsymbol{H}_2]_{N\times l_2} = \boldsymbol{\sigma}(\boldsymbol{Z}_2)$$
- ……

- Compute final output $\hat{\boldsymbol{y}}$, a vector corresponding to all N points

Backward Pass in Matrix Notation

- Mean squared error computed on all data: $E(\boldsymbol{\theta}) = \frac{1}{2N}\sum_{i=1}^{N}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 = \frac{1}{2N}(\boldsymbol{y} - \hat{\boldsymbol{y}})^T(\boldsymbol{y} - \hat{\boldsymbol{y}})$
- Compute gradients w.r.t. weights in the output layer (the M-th layer)

$$\left[\frac{\partial E}{\partial \hat{\boldsymbol{y}}}\right]_{N\times 1} = \frac{1}{N}(\hat{\boldsymbol{y}} - \boldsymbol{y})$$

$$[\sigma'(\boldsymbol{z}_M)]_{N\times 1} = \hat{\boldsymbol{y}} \odot (1 - \hat{\boldsymbol{y}})$$

$$\left[\frac{\partial E}{\partial \boldsymbol{w}_M}\right]_{l_{M-1}\times 1} = \left[\frac{\partial \boldsymbol{z}_M}{\partial \boldsymbol{w}_M}\right]_{l_{m-1}\times N} \cdot \left[\left[\frac{\partial E}{\partial \hat{\boldsymbol{y}}}\right]_{N\times 1} \odot [\sigma'(\boldsymbol{z}_M)]_{N\times 1}\right]$$

$\boldsymbol{H}_{M-1}^T$

$$\frac{\partial E}{\partial b_M} = \left[\frac{\partial \boldsymbol{z}_M}{\partial b_M}\right]_{1\times N} \cdot \left[\left[\frac{\partial E}{\partial \hat{\boldsymbol{y}}}\right]_{N\times 1} \odot [\sigma'(\boldsymbol{z}_M)]_{N\times 1}\right]$$

$\boldsymbol{1}^T$

Backward Pass in Matrix Notation

- Compute gradients w.r.t. weights in the $(m-1)$-th layer recursively

$$\left[\frac{\partial E}{\partial \boldsymbol{H}_{m-1}}\right]_{N\times l_{m-1}} = \left[\frac{\partial E}{\partial \boldsymbol{H}_{m}}\right]_{N\times l_{m}} \odot [\sigma'(\boldsymbol{Z}_m)]_{N\times l_m} \cdot [\boldsymbol{W}_m^T]_{l_m\times l_{m-1}}$$

$$\left[\frac{\partial E}{\partial \boldsymbol{W}_{m-1}}\right]_{l_{m-2}\times l_{m-1}} = [\boldsymbol{H}_{m-2}^T]_{l_{m-2}\times N} \cdot \left[\left[\frac{\partial E}{\partial \boldsymbol{H}_{m-1}}\right]_{N\times l_{m-1}} \odot [\sigma'(\boldsymbol{Z}_{m-1})]_{N\times l_{m-1}}\right]$$

$$\left[\frac{\partial E}{\partial \boldsymbol{b}_{m-1}}\right]_{l_{m-1}\times 1} = \left[\left[\frac{\partial E}{\partial \boldsymbol{H}_{m-1}}\right]_{N\times l_{m-1}} \odot [\sigma'(\boldsymbol{Z}_{m-1})]_{N\times l_{m-1}}\right]^T \cdot \boldsymbol{1}_{N\times 1}$$

Problems of BP for Deep Networks

- Vanishing gradient problem
  - Gradients vanishes when they are propagated back to early layers, hence their weights are hard to adjust
  - Sigmoid activation → ReLU activation

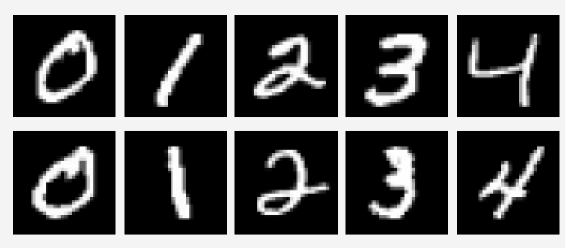
- Many local minima
  - Which will trap gradient decent methods
  - In practice, local minima are pretty good

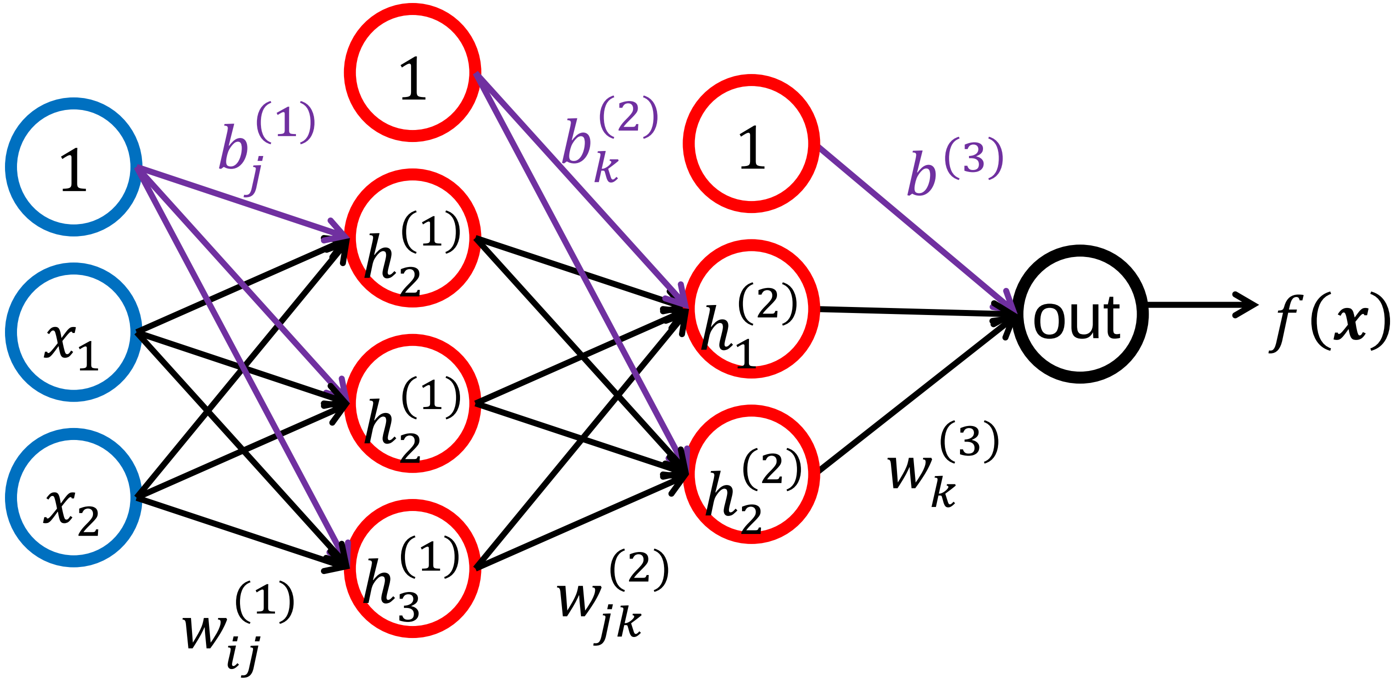
MLP Summary

- (Artificial) neural networks are inspired by the biological neural networks
  - Parallel processing + distributed representation
- Feedforward neural networks use a layer-wise structure
  - Full connection between adjacent layers
  - Linear mapping + nonlinear activation
- Representation power
  - 1-layer NNs are just perceptron or logistic regression
  - 2-layer NNs can represent (almost) any continuous function, with sufficient hidden nodes
  - >=3-layer NNs can do so with much fewer nodes
- Gradient descent to update network weights using training data
- Backpropagation algorithm to recursively compute gradients
  - Vanishing gradient issues for sigmoid activation

MLP → Convolutional Neural Networks (CNN)

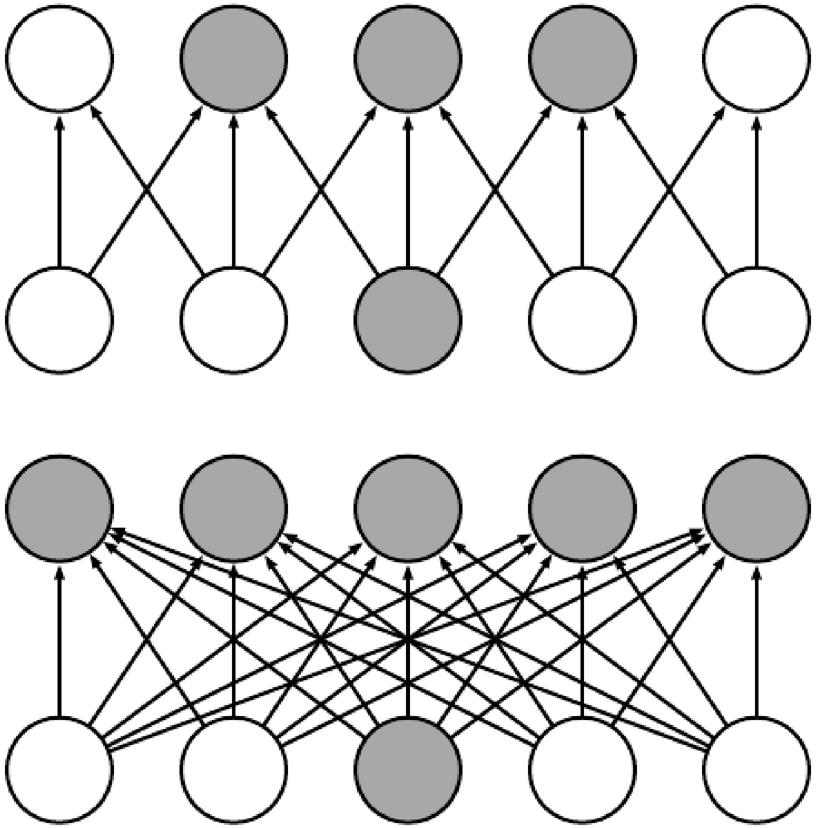
- Fully connected between adjacent layers
  - Many parameters → prone to overfitting
  - Some connections may be unnecessary
  - Not robust to shifts of input





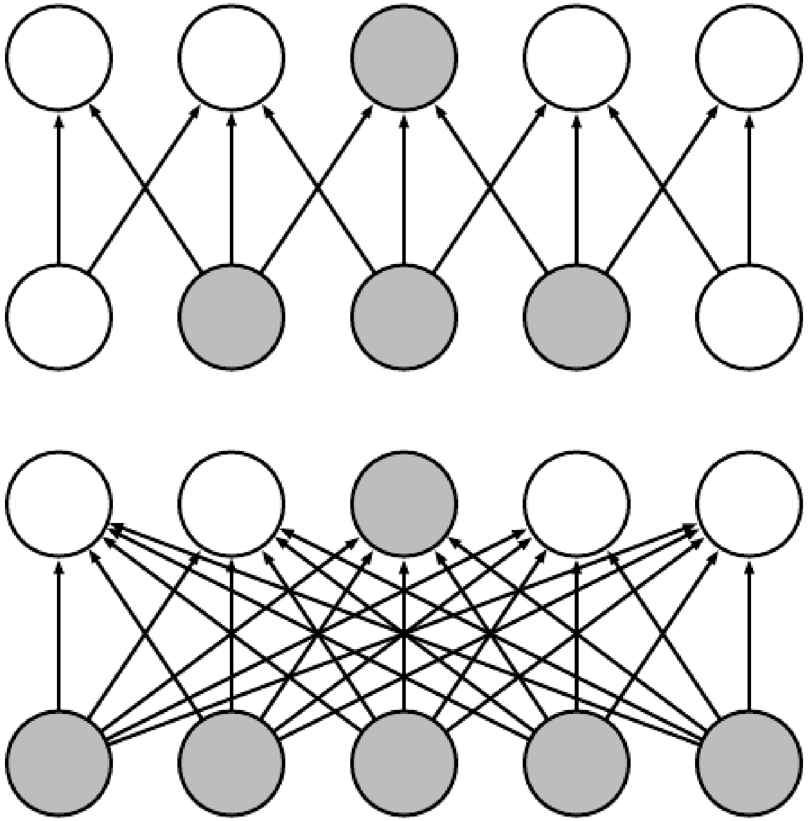
Full Connection → Sparse Connection

- Only keep local connections
  - Assuming nearby inputs have stronger correlations



(Fig. 9.2 in GBC)

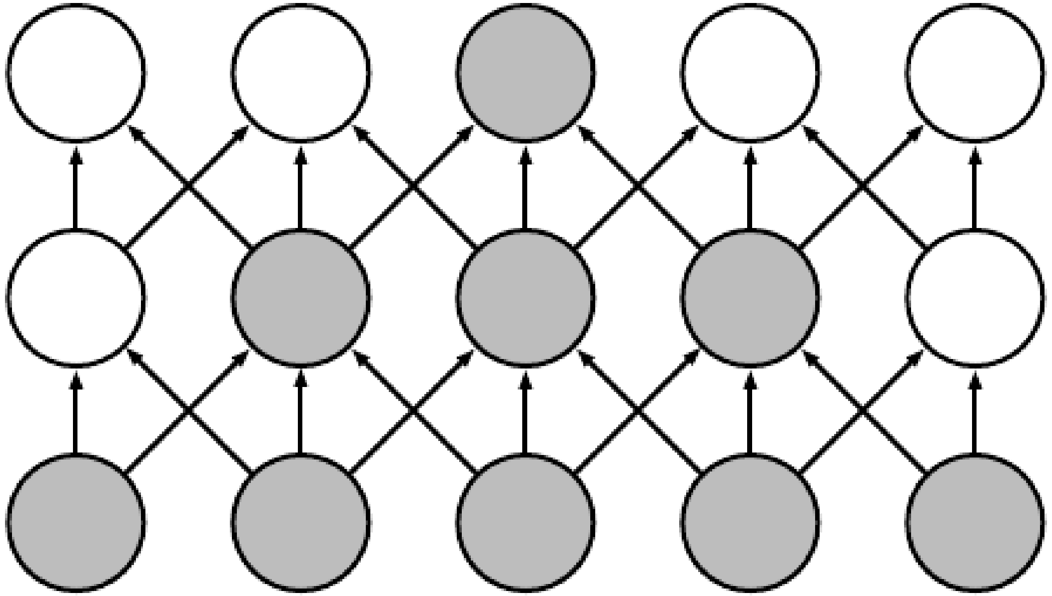
Receptive field of a neuron



(Fig. 9.3 in GBC)

Receptive Field at a Deeper Layer

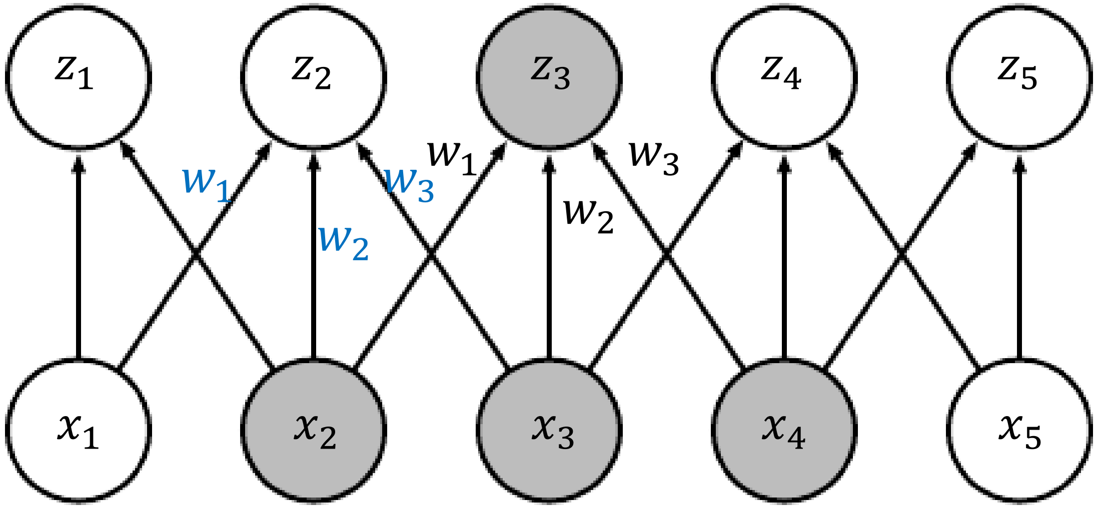
- With sparse connections, nodes at a deeper layer can still have a large receptive field, and global patterns could still be captured



(Fig. 9.4 in GBC)

Independent Weights → Shared Weights

- Assuming neurons at different locations process their inputs in the same way, we can let them share weights



(Adapted from Fig. 9.3 in GBC)

Much Fewer Parameters!

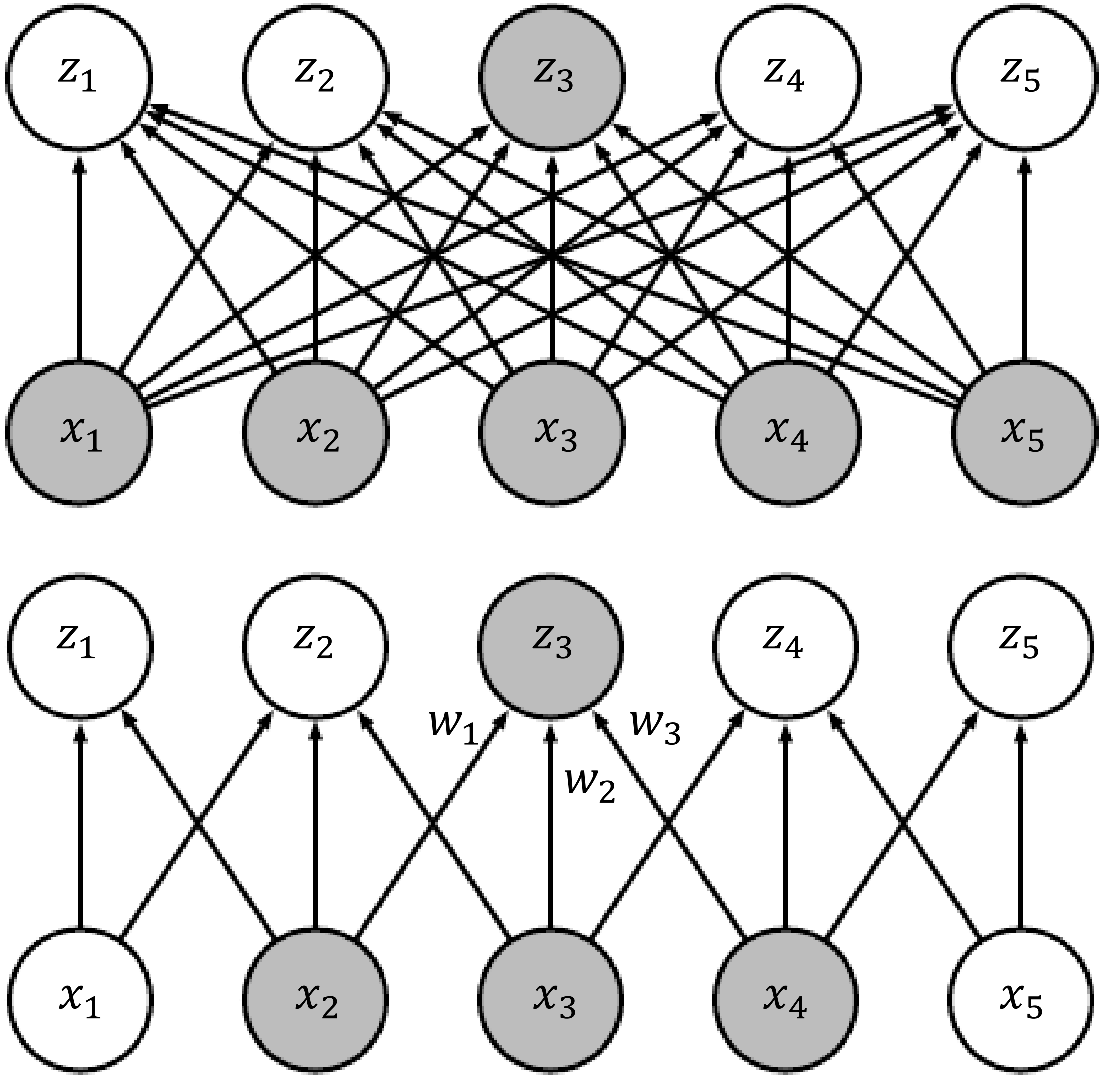
$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix} = \begin{bmatrix} w_{11} & \cdots & w_{51} \\ \vdots & \vdots & \vdots \\ w_{15} & \cdots & w_{55} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}$$

- 5*5+5 parameters (biases are omitted in figures)

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix} = \begin{bmatrix} w_2 & w_3 & 0 & 0 & 0 \\ w_1 & w_2 & w_3 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 \\ 0 & 0 & 0 & w_1 & w_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}$$

- 3+1 parameters

$$z_n = \sum_m w_m x_{m+n-2}$$



(Adapted from Fig. 9.3 in GBC)

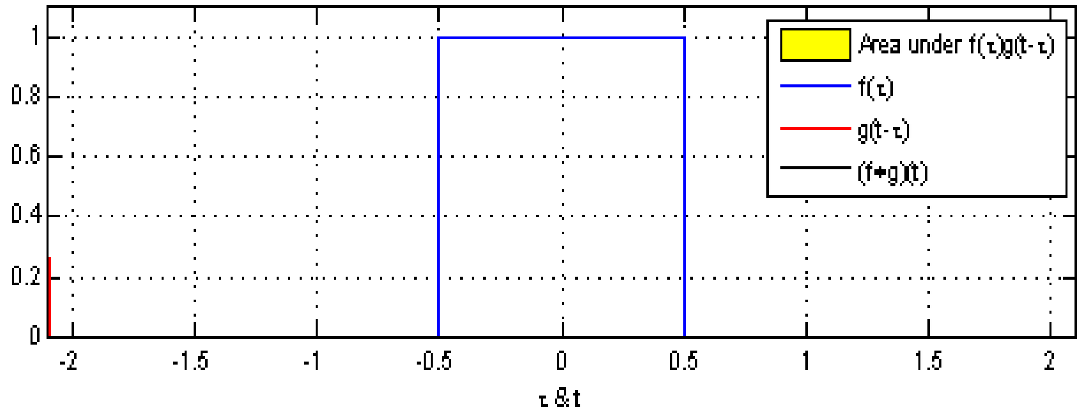
This is basically convolution

- Continuous-time signals

$$z(t) = (x * w)(t) = \int x(\tau)w(t-\tau)d\tau = \int x(t-\tau)w(\tau)d\tau = (w * x)(t)$$

- Discrete-time signals

$$z[n] = x[n] * w[n] = \sum_m x[m]w[n-m] = \sum_m x[n-m]w[m] = w[n] * x[n]$$



- Cross convolution: no flipping, but is the convolution referred to in deep learning

$$z[n] = \sum_m x[m]w[n+m]$$

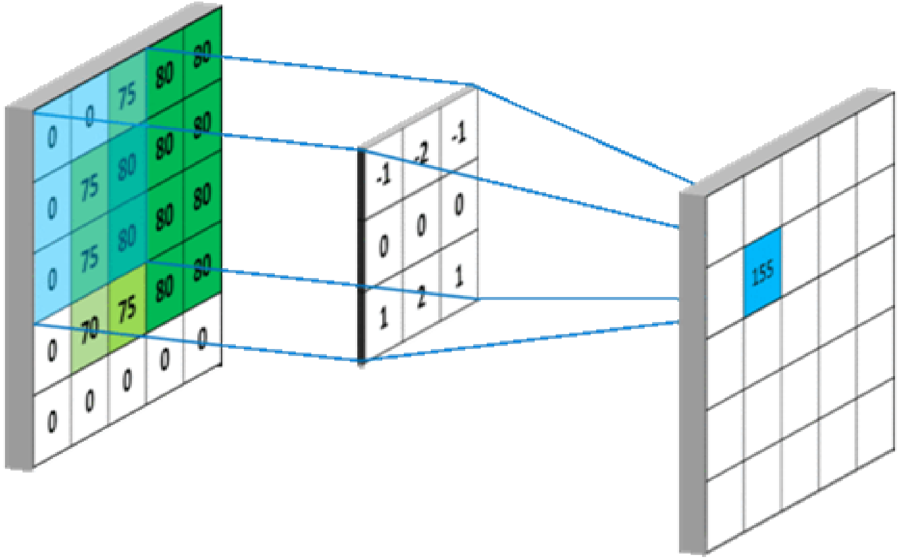
2D Convolution

2D convolution $$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n)$$

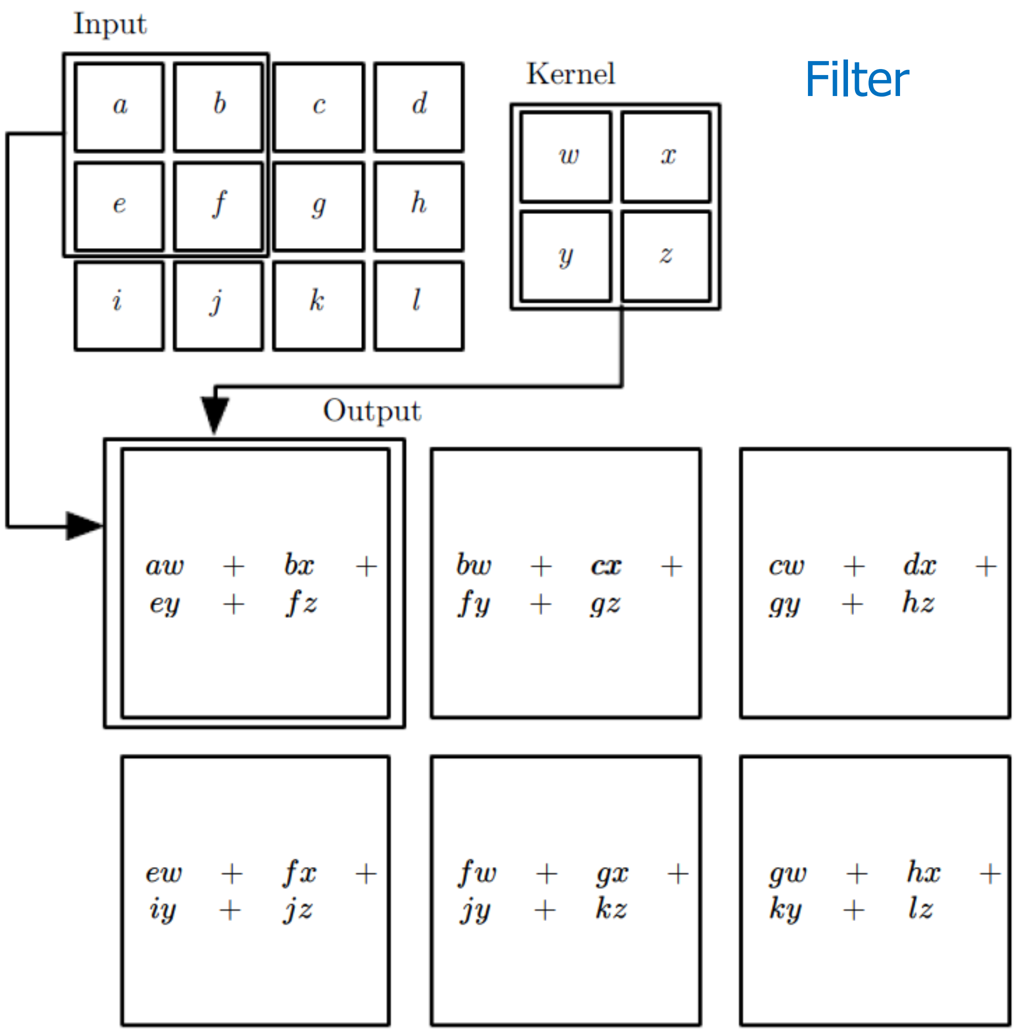
2D cross-correlation $$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n)$$

Input Signal 2D (i.e Image)

Filter/Kernel 2D



2D Convolution



Filter

Feature map

(Fig. 9.1 in GBC)

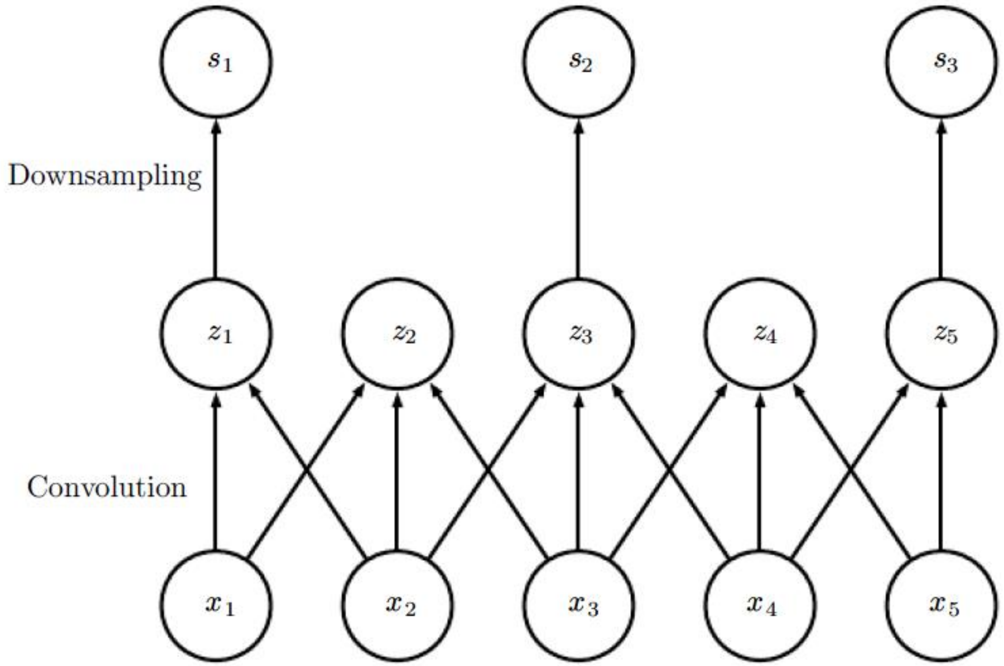
2D Convolution Example

- Vertical edge detection using a 1*2 kernel [-1, 1]
- (Cross-)convolving a gray-scale image with this kernel computes the intensity difference between two horizontally adjacent pixels

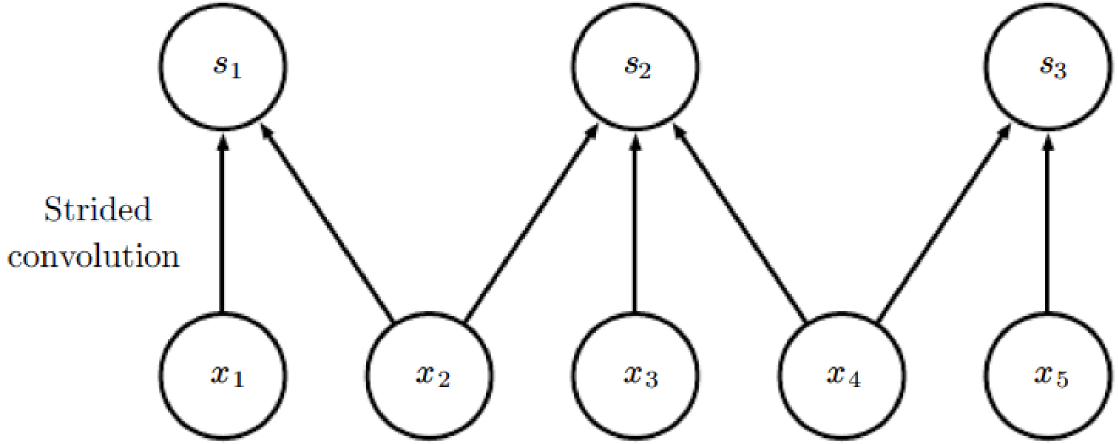
(Fig. 9.6 in GBC)

Convolution with Strides

- Downsampling after convolution



Stride size = 2



(Fig. 9.12 in GBC)

2D Convolution with Strides

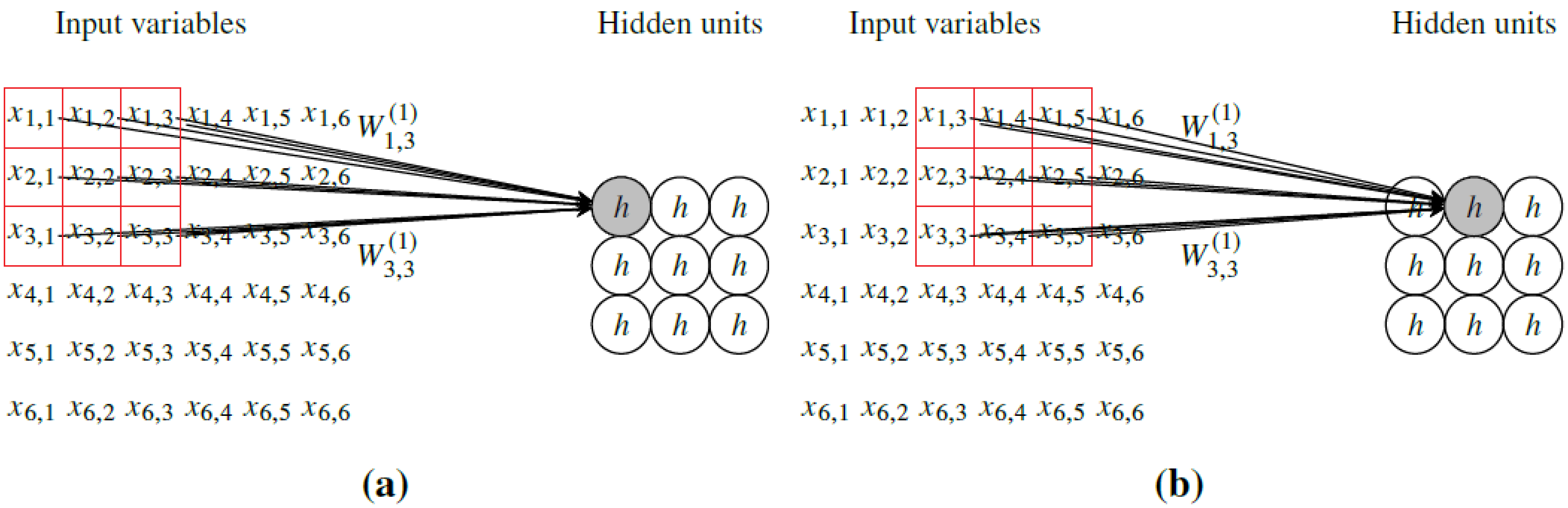
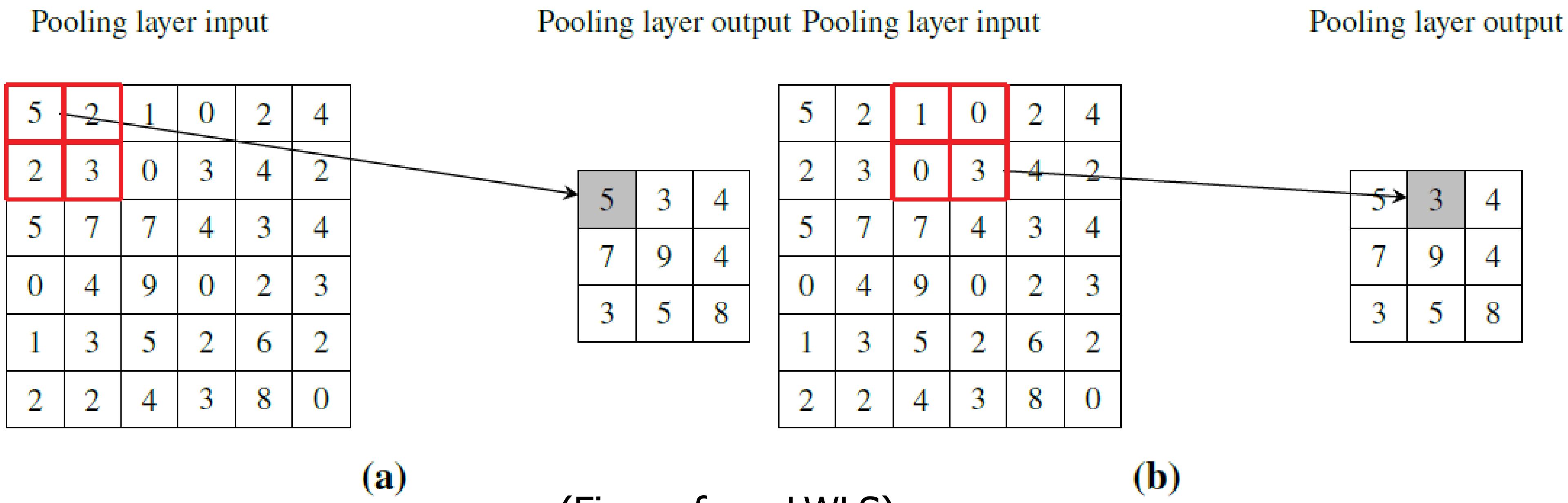


Figure 6.12: A convolutional layer with stride 2 and filter size 3×3.

(Figure from LWLS)

Pooling

- Pooling is another way to reduce the size of feature maps
  - Max pooling: taking the max → result is invariant to small shifts
  - Average pooling: taking the average
- No trainable parameters

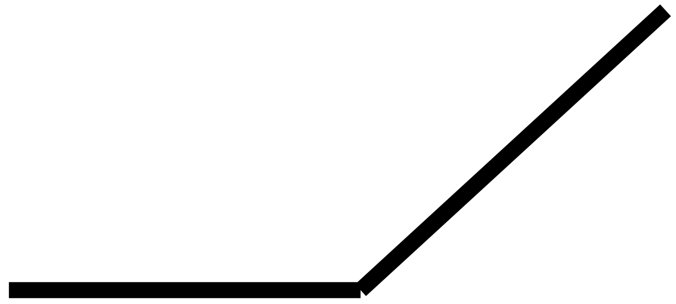


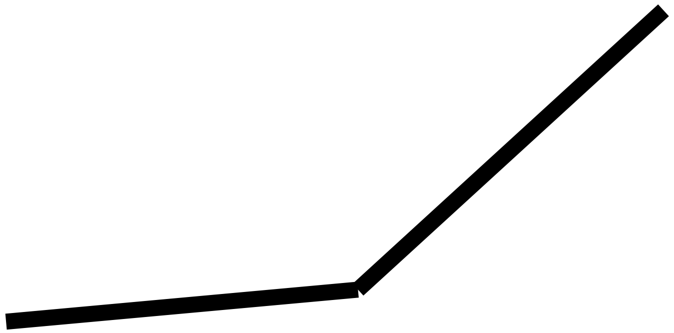
(Figure from LWLS)

Figure 6.13: A max pooling layer with stride 2 and pooling filter size 2×2.

Nonlinear Activation

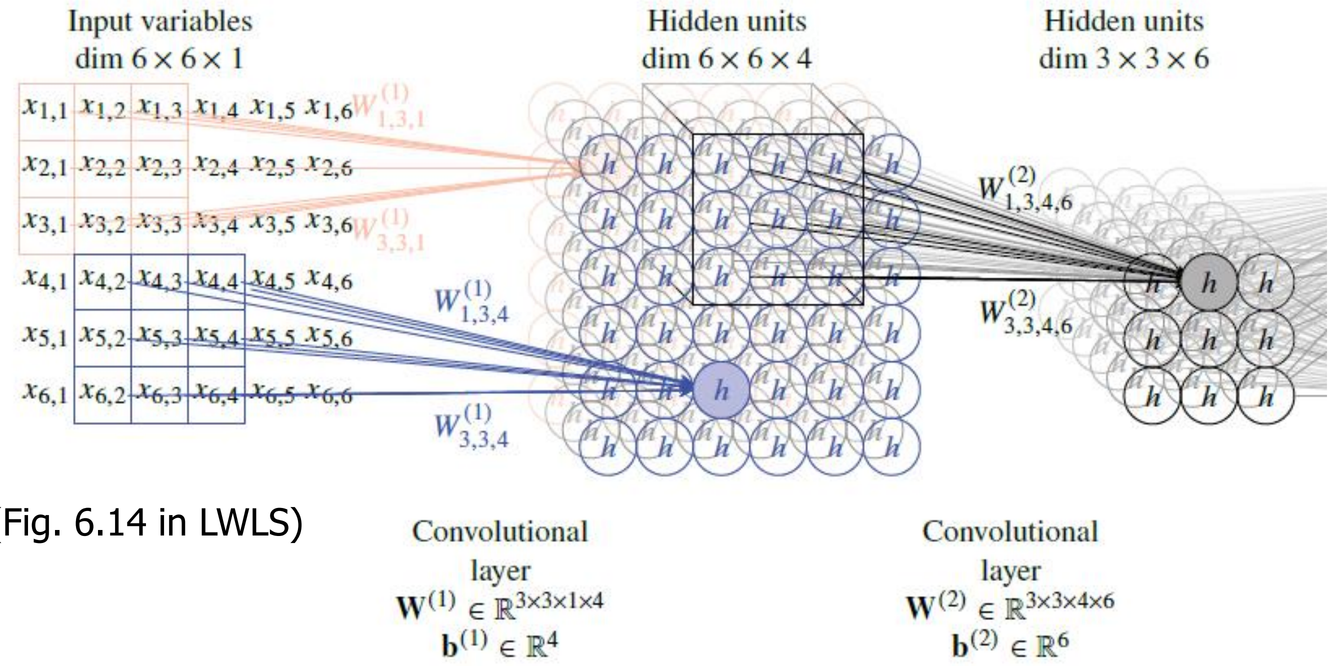
- As discussed before, convolution is a linear operation
- We need a nonlinear activation after convolution to build deep nets
- Rectified Linear Unit (ReLU) and Leaky ReLU is most used





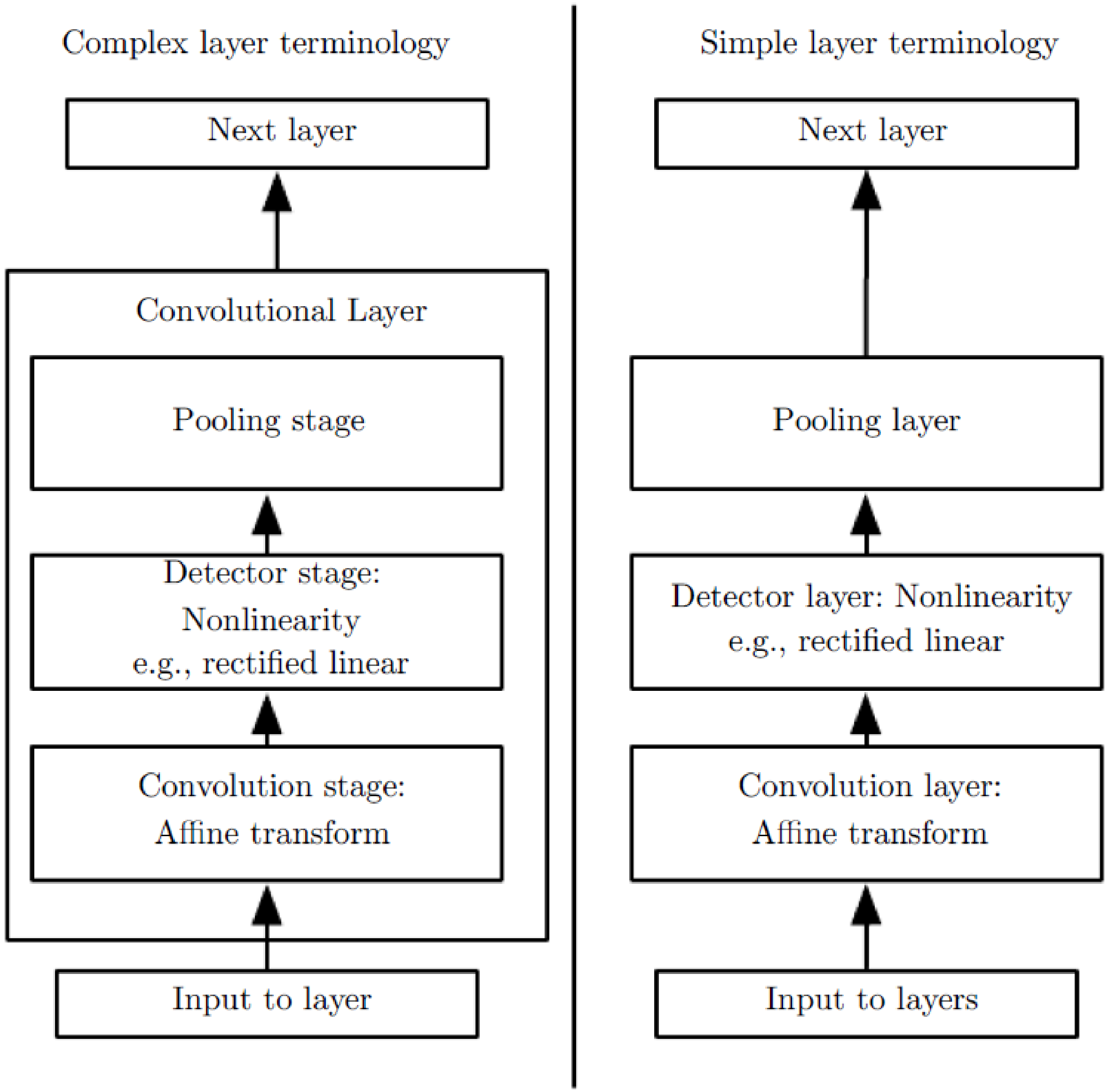
Multiple Channels

- Convolution with a single filter (kernel) detects only one pattern (e.g., vertical edges)
- Use multiple filters to detect more patterns
  - Each filter results in one feature map
  - Multiple filter result in multiple feature maps, stacked as channels
  - When input is 2D with multiple channels, each filter becomes a 3D tensor



(Fig. 6.14 in LWLS)

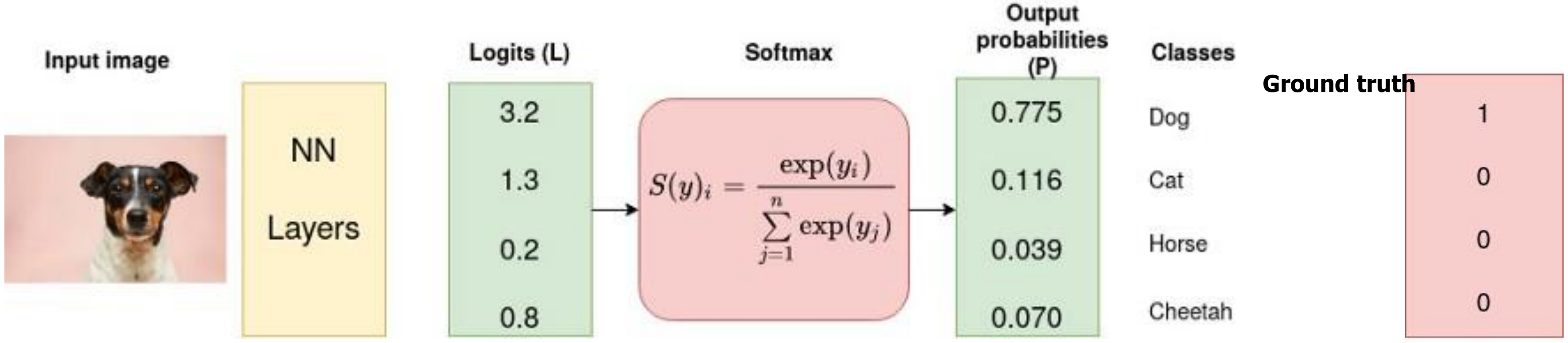
Convolution Layer



(Fig. 9.7 in GBC)

Typical Output Layer

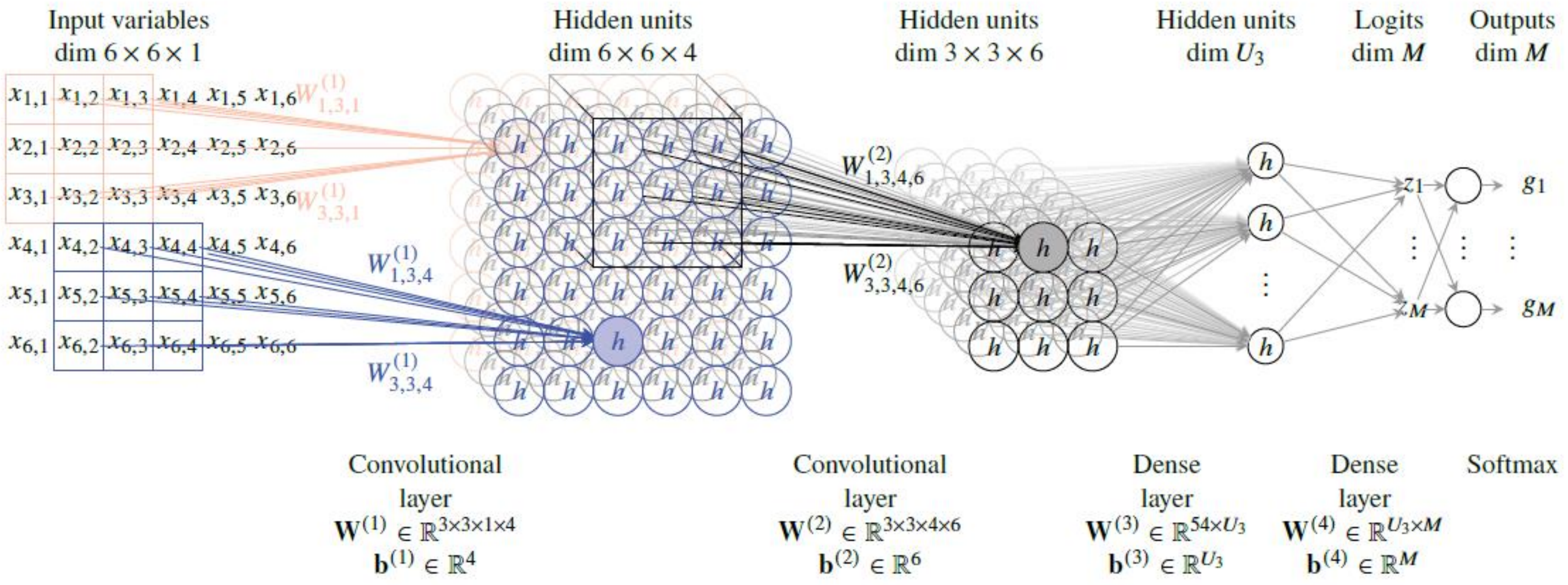
- After a stack of convolutional layers, a few fully connected layers often follow to give the output
  - The last convolutional layer's feature map is reshaped to a vector
- M-Class Classification:
  - Use M output nodes
  - Softmax activation (probability): $\hat{y}_i = \frac{e^{h_i}}{\sum_{j=0}^{M-1} e^{h_j}}, \forall i = 0, \cdots, M-1$
  - Cross entropy loss: $L_{CE} = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$



(Figure from https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e)

Full CNN Architecture

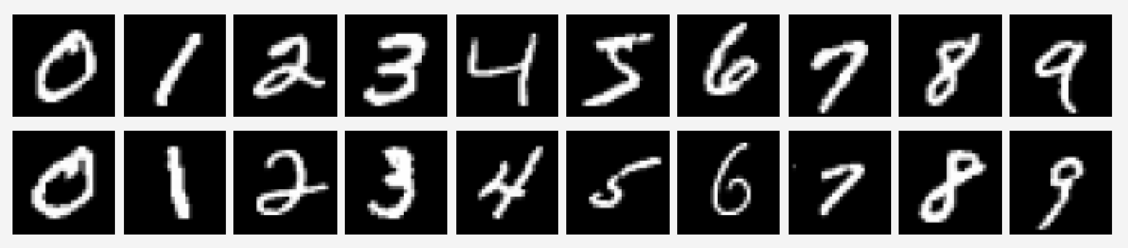
- M-class classification on single-channel 2D input



(Fig. 6.14 in LWLS)

Full CNN Architecture

- Input: 28*28=784-d gray-scale (i.e., 1-channel) hand-written digits



	Convolutional layers			Dense layers	
	Layer 1	**Layer 2**	**Layer 3**	**Layer 4**	**Layer 5**
Number of filters/output channels	4	8	12	–	–
Filter rows and columns	(5×5)	(5×5)	(4×4)	–	–
Stride	1	2	2	–	–
Number of hidden units	3 136	1 568	588	200	10
Number of parameters (including offset vector)	104	808	1 548	117 800	2 010

784*4=3136 784/4*8=1568 784/4/4*12=588

(Example 6.3 in LWLS)

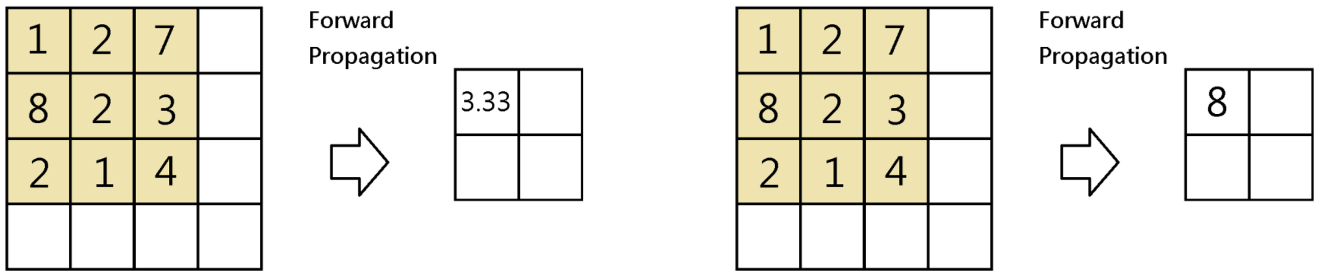
Network Training

- Define a loss function
  - Classification: cross entropy for softmax output
  - Regression: mean squared error
- Stochastic gradient descent
  - Randomly picking training samples to form a mini-batch
  - Compute gradient of loss function w.r.t. weights through backpropagation
  - Update weights along negative gradient with some (adaptive) learning rate
- Different optimizers
  - Adam: adaptive moment estimation – uses running averages on gradients and second order moments
  - Adagrad: adaptive gradient – uses different learning rates at different iterations
  - RMSprop: root mean square propagation – exponentially weighted average of squared gradient to adapt learning rate

Backpropagation for CNN

- BP through nonlinear activation
  - Same as before

- BP through pooling
  - Average pooling: gradient is equally distributed to all inputs
  - Max pooling: gradient is solely assigned to the max input



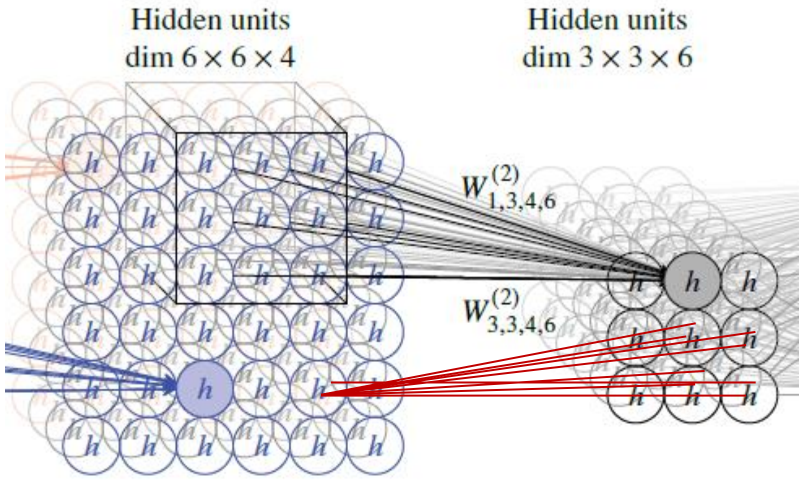
(Figures from https://lanstonchu.wordpress.com/2018/09/01/convolutional-neural-network-cnn-backward-propagation-of-the-pooling-layers/)

Backpropagation for CNN

- Convolution is a linear operation between the input tensor and a kernel, and it results in an output tensor

- BP through convolution to layer input
  - Each element of the input tensor affects multiple channels of the output tensor through different filters

- BP through convolution to layer weights
  - Each weight affects all elements of one output channel through all channels of previous layer's output



(Adapted from Fig. 6.14 in LWLS)

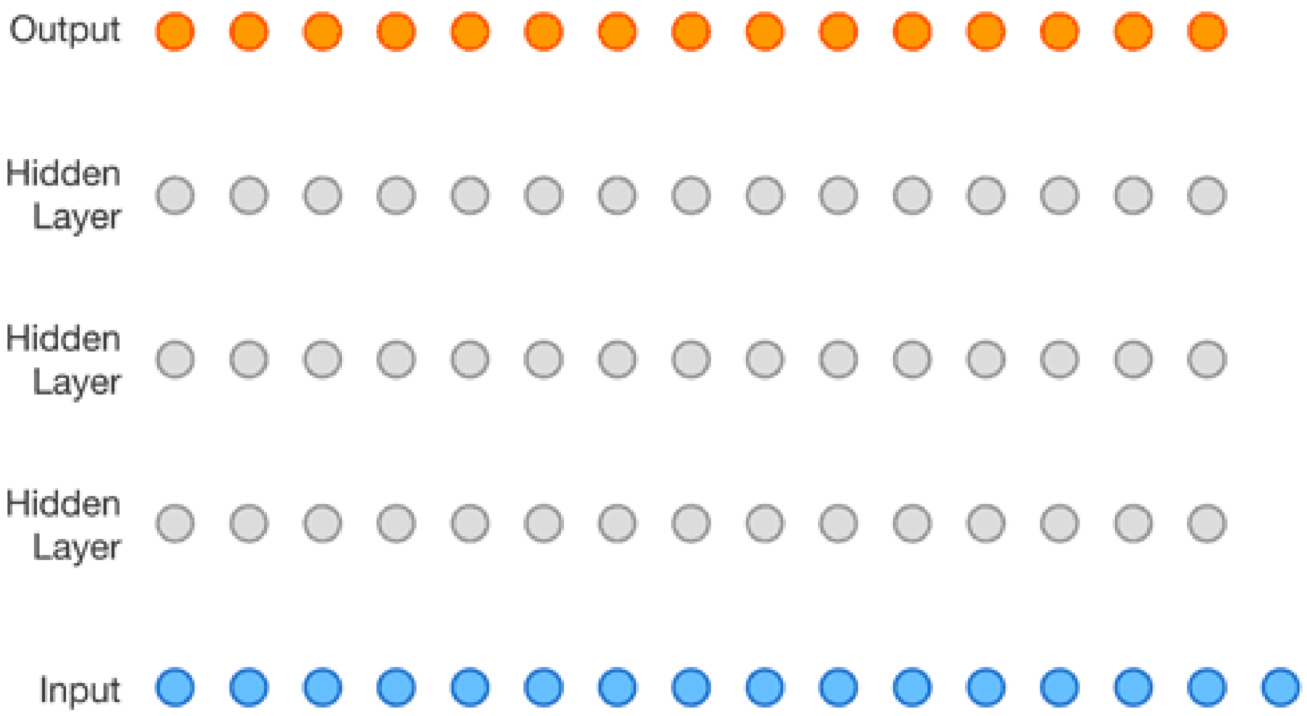
CNNs for Different Types of Input

	Single-Channel	Multi-Channel
1-D	Audio waveforms	Skeleton animation data: Each channel represents one angle of one joint
2-D	Audio spectrograms; gray-scale images	Color images: RGB channels
3-D	Volumetric data, e.g., CT scans	Color video data

(Adapted from Table 9.1 in GBC)

1D CNN for Audio Generation

- WaveNet [van den Oord et al., 2016]
- Dilated causal convolution



https://www.deepmind.com/blog/wavenet-a-generative-model-for-raw-audio

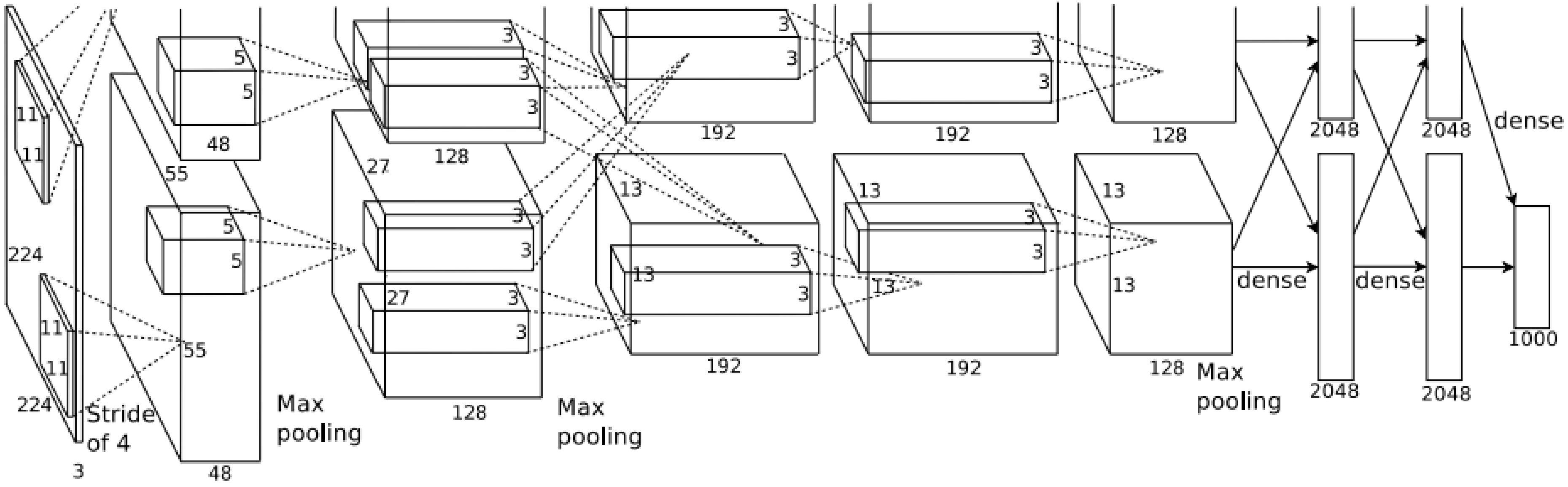
free generation (speech)

text-to-speech

Free generation (piano music)

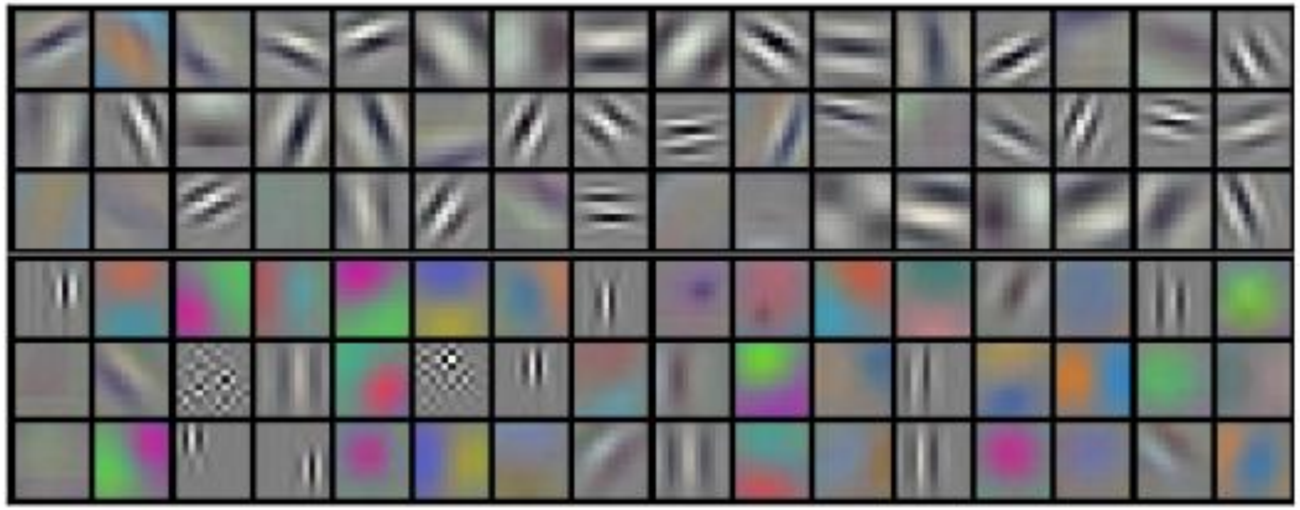
2D CNN for Image Classification

- AlexNet [Krizhevsky et al., 2012]



Filter Visualization of AlexNet

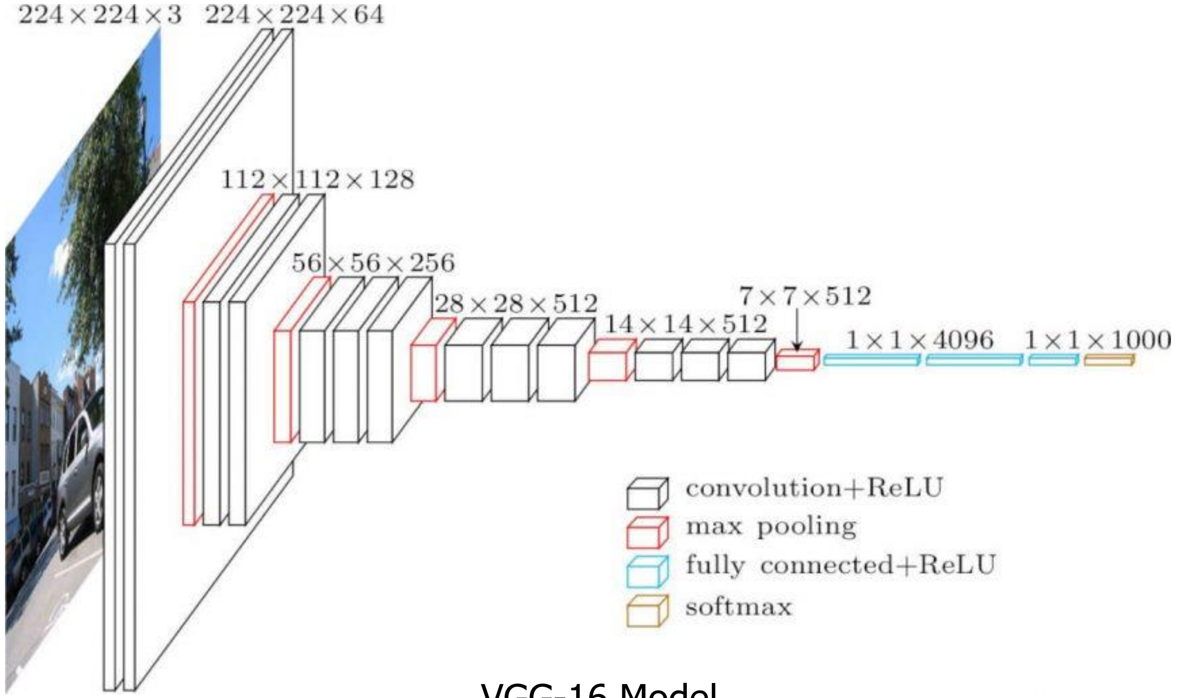
- Learned filters of the 1st convolutional layer
  - 96 filters with size of 11*11*3



[Krizhevsky et al., 2012]

Transfer Learning with Pretrained Networks

- First layers (features extractors) learned from one task (e.g., natural image classification) can be useful for another relevant task (e.g., medical image classification)
- Use a pre-trained model (on big data tasks) to build a new model (for small data tasks)
  - Remove last few layers (e.g., the last dense layer), which are usually task-specific
  - Use the remaining layers to build a new network by adding a couple of layers for the new task
  - Train new layers (or fine tune the entire network) on the new task



VGG-16 Model

ImageNet

- 1.3 M images from 1000 classes

CNN for Audio Applications

- Apply 1D convolution on audio samples (WaveNet)
- Audio → Magnitude spectrogram → Apply 2D convolution

Applications :

- Classification/Identification: sound, genre, instrument, speaker, etc.
- Source Separation: mask prediction
- Generation: predict the next audio sample

Disadvantages:

- In images, neighboring pixels belong to the same object, not the same for spectrograms
- CNNs are applied in magnitude, and not phase
- CNNs cannot model long-term temporal information

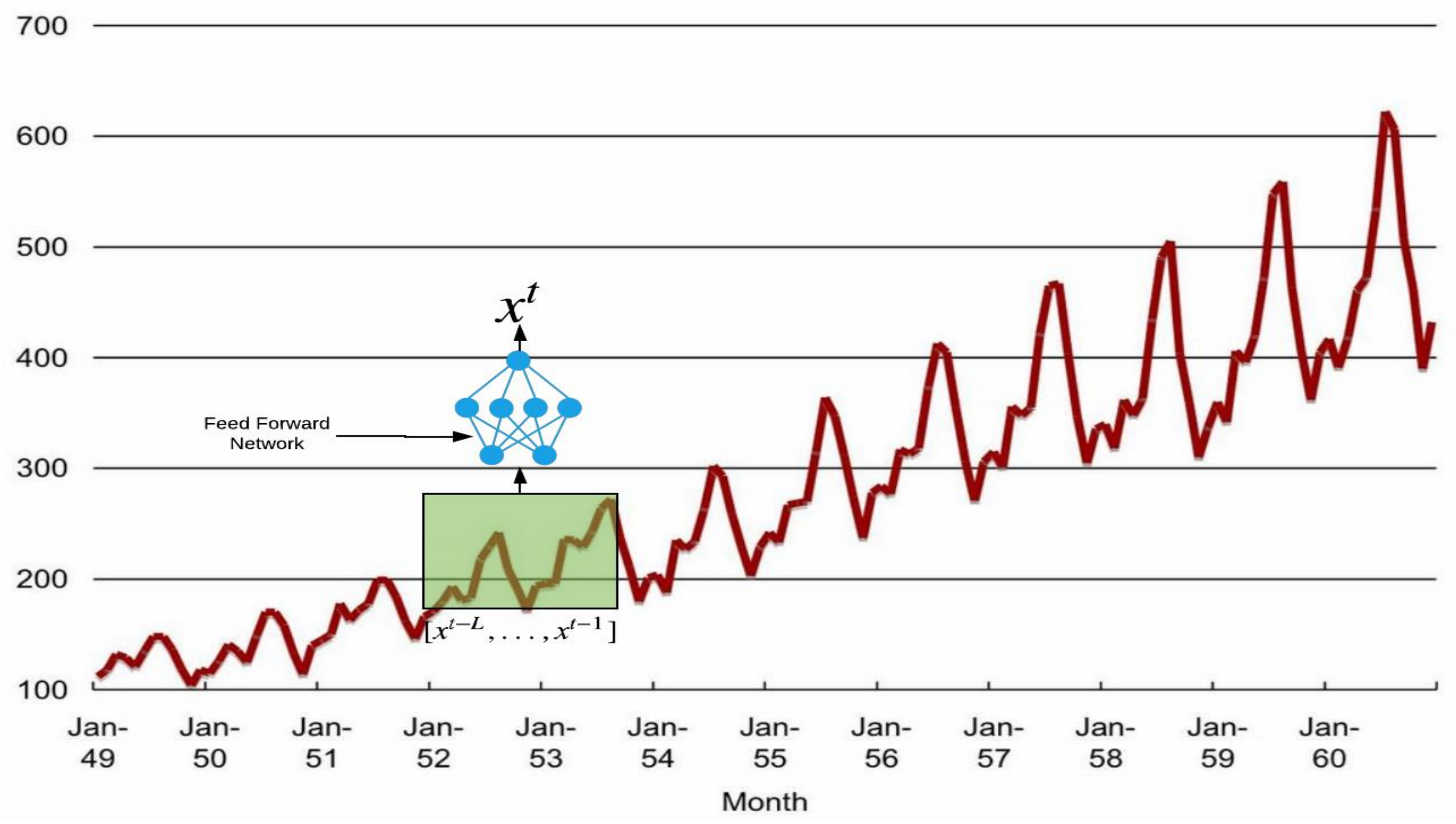
CNN Summary

- Key properties of CNNs
  - Sparse (local) connection
  - Shared weights
  - Equivariance to translation
- Important components
  - Convolution
  - Pooling: max pooling, average pooling
  - Activation: ReLU
- Important concepts
  - Filter, receptive field, channel, tensor
- Applications
  - Classification, regression, generation
  - 1D, 2D, 3D
- Think: what problems/data are not appropriate for CNN?

MLP → Recurrent Neural Network (RNN)

- Model time series with MLP, e.g., predicting the next data point
  - Limited memory
  - Fixed window size L
  - Number of weights increases with L quickly
  - Predictions at different times are independent

- How to better model past information?

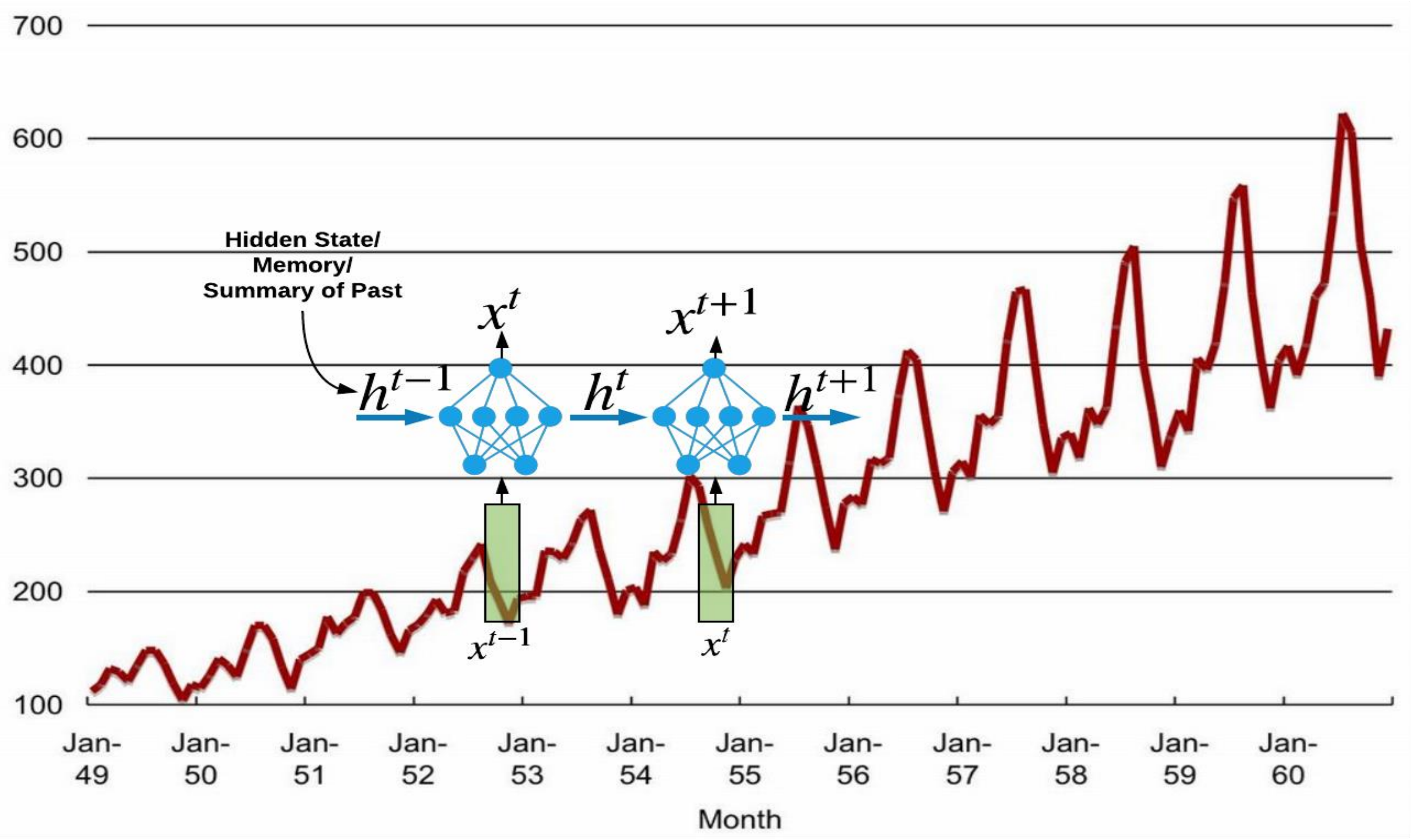


(Figure from Box and Jenkins, *Time Series Analysis: Forecasting and Control,* 1976)

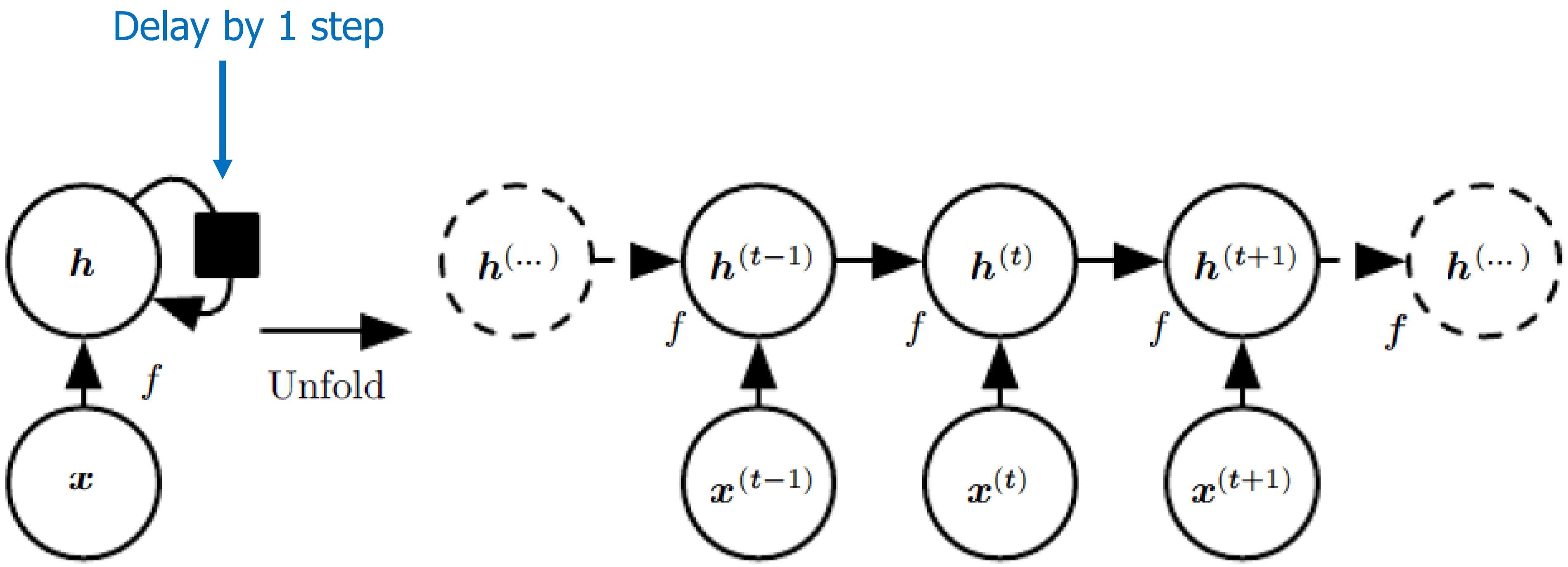
Make Network Recurrent

- Parameter sharing
  - Different positions use the same network
- Add recurrent links
  - Current computation affects future computation
  - Carry past information to the future

- Compared with 1D convolution
  - Both have weight sharing
  - Convolution has limited receptive field
  - Recurrency can carry information infinitely long (in theory)



Unfold Recurrency



(Fig. 10.2 in GBK)

$\boldsymbol{h}^{(t)}$ is affected all past input: $\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(t)}$

Different Types of Recurrency

- RNNs that produce an output at each time step and have recurrent connections between hidden units
- Take classification / labeling as example
- Forward propagation

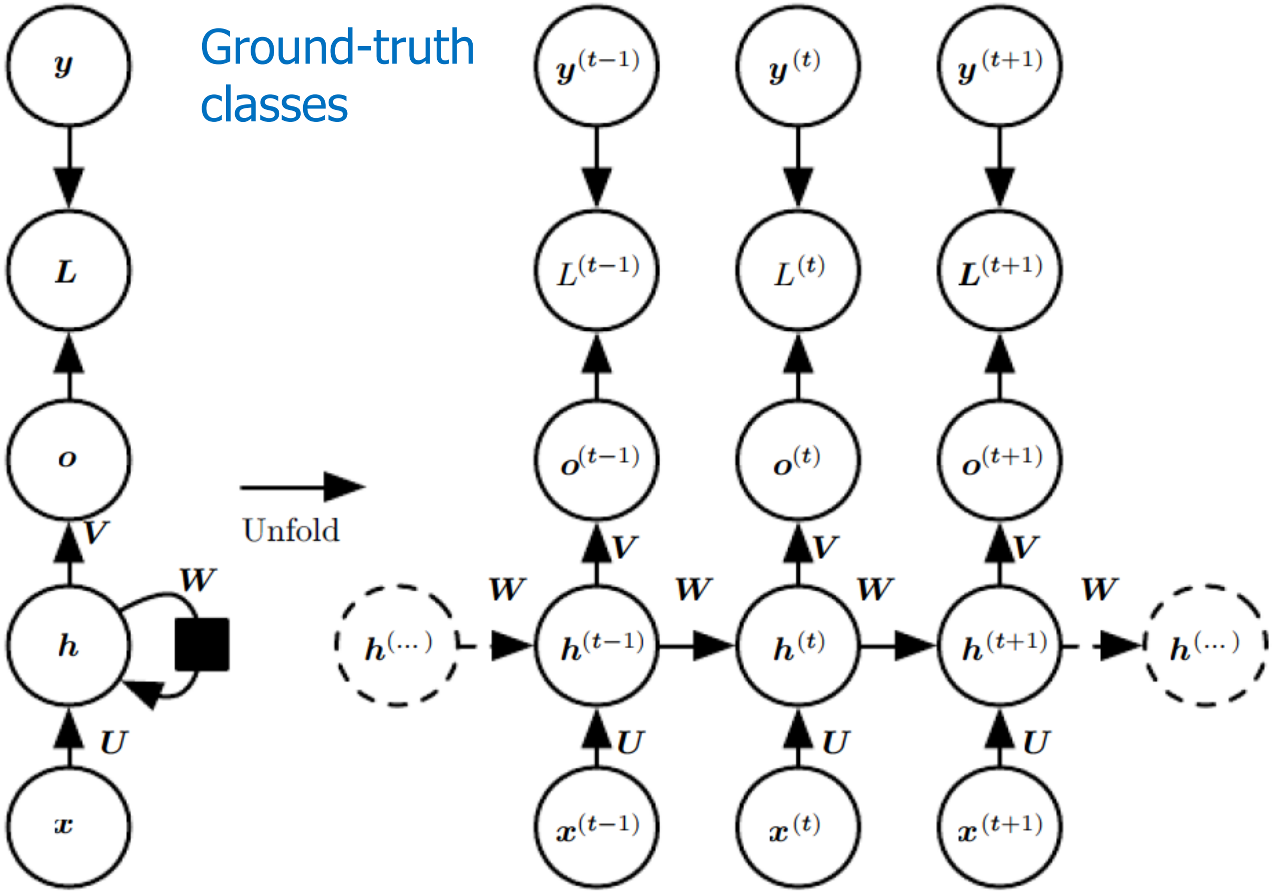
Net input to hidden $\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)},$

Nonlinear activation $\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)}),$

Linear output $\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)},$

Softmax -> class prob. $\hat{\boldsymbol{y}}^{(t)} = \mathrm{softmax}(\boldsymbol{o}^{(t)}),$

Cross entropy loss: $L = -\sum_t \log\left(\left[\hat{\boldsymbol{y}}^{(t)}\right]_{y^{(t)}}\right)$

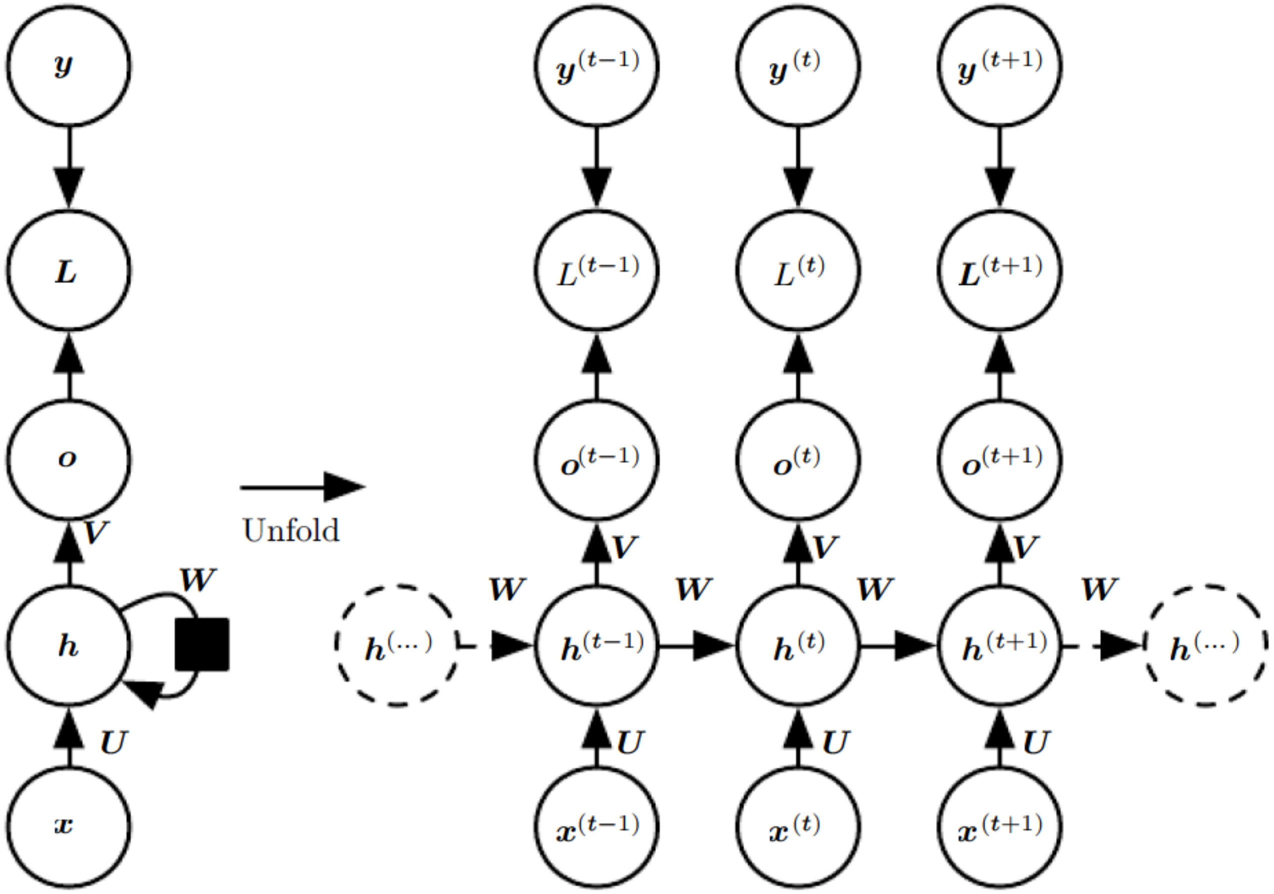


(Fig. 10.3 in GBK)

Back Propagation Through Time (BPTT)

- Output (hence loss) at time t is affected by past inputs and hidden nodes through the recurrent links
- To perform gradient descent, gradients need to pass backwards through the recurrent links

- Each update of weights requires
  - Forward computation of all hidden nodes and output nodes
  - Backpropagation of gradients
  - Both computations are sequential → cannot be parallelized → slow to train



(Fig. 10.3 in GBK)

BPTT Sketch

- Same as regular backpropagation → repeatedly apply chain rule

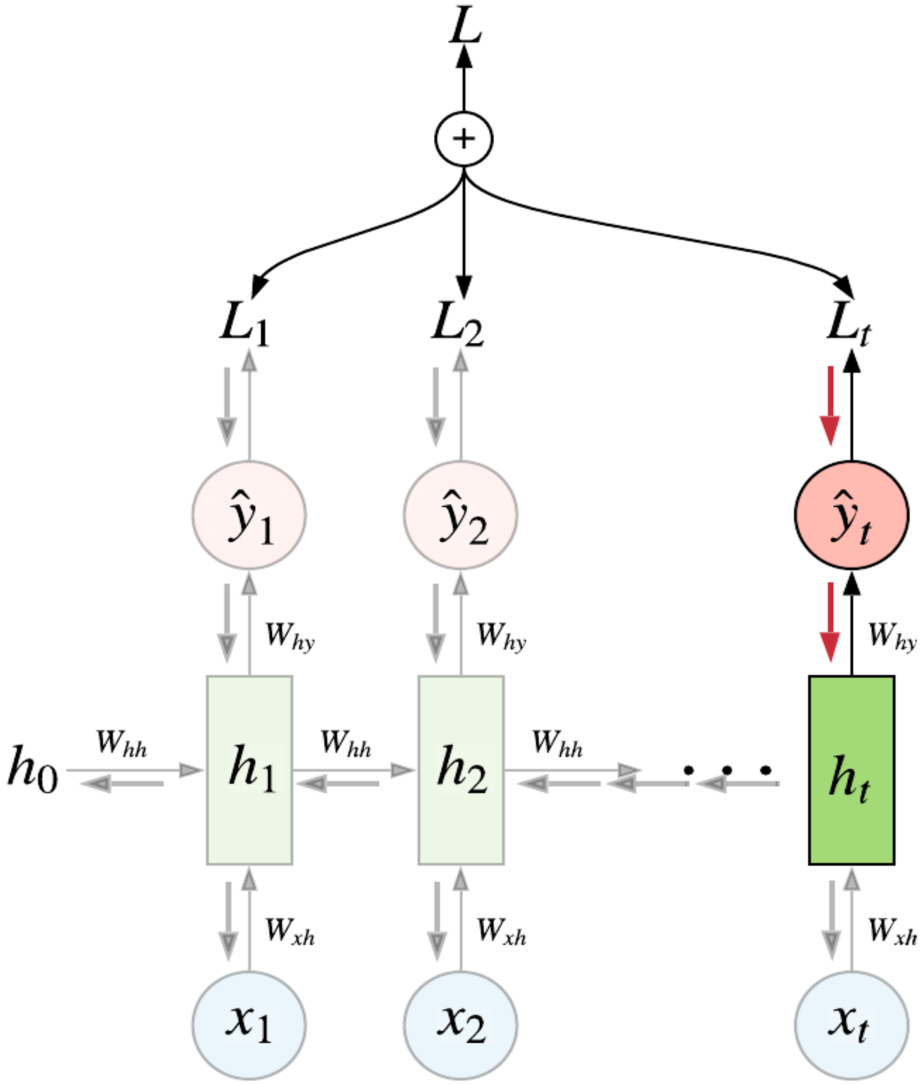
- For W_{hy}, we propagate along the vertical links

$$\frac{\partial L}{\partial W_{hy}} = \sum_{i=0}^{t} \frac{\partial L_i}{\partial W_{hy}}$$

$$\frac{\partial L_t}{\partial W_{hy}} = \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{W_{hy}}$$

$$\hat{y}_t = W_{hy} h_t$$

Easy to calculate



BPTT Sketch

- Same as regular backpropagation → repeatedly apply chain rule

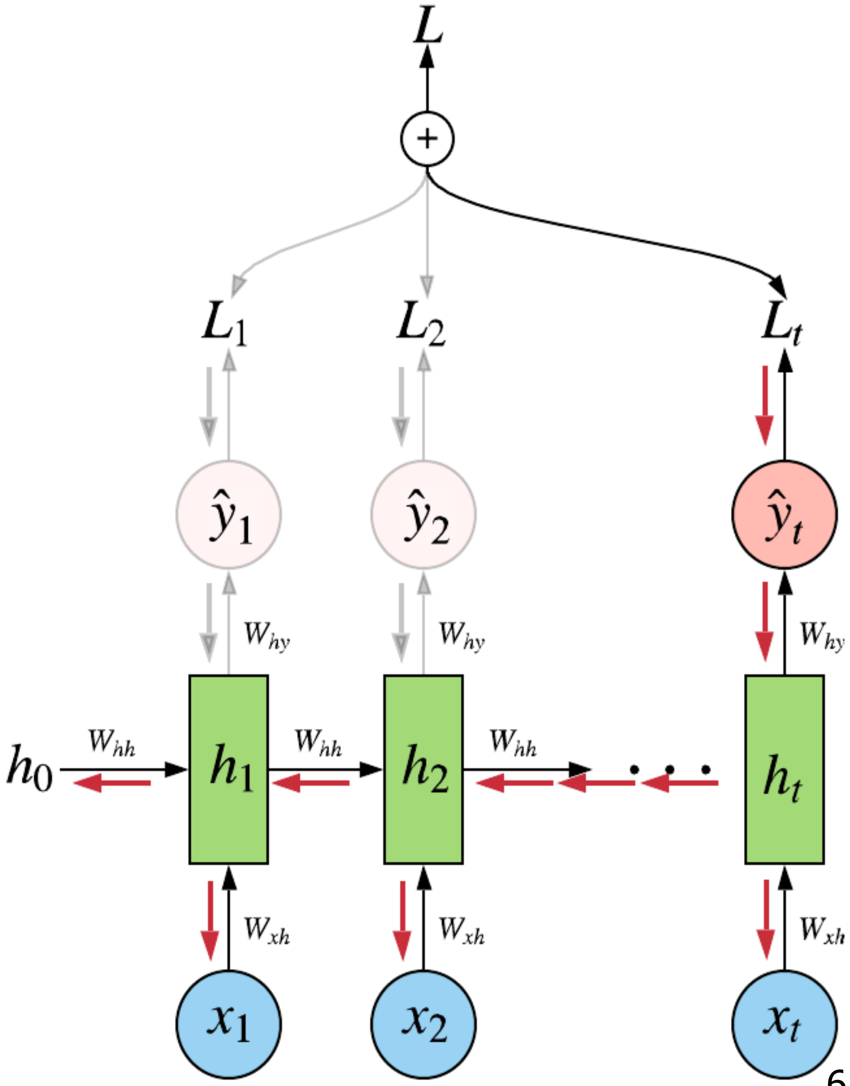
- For W_{hh} and W_{xh}, we also propagate along the horizontal (i.e., recurrent) links

$$\frac{\partial L}{\partial W_{hh}} = \sum_{i=0}^{t} \frac{\partial L_i}{\partial W_{hh}}$$

$$\frac{\partial L_t}{\partial W_{hh}} = \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial W_{hh}}$$

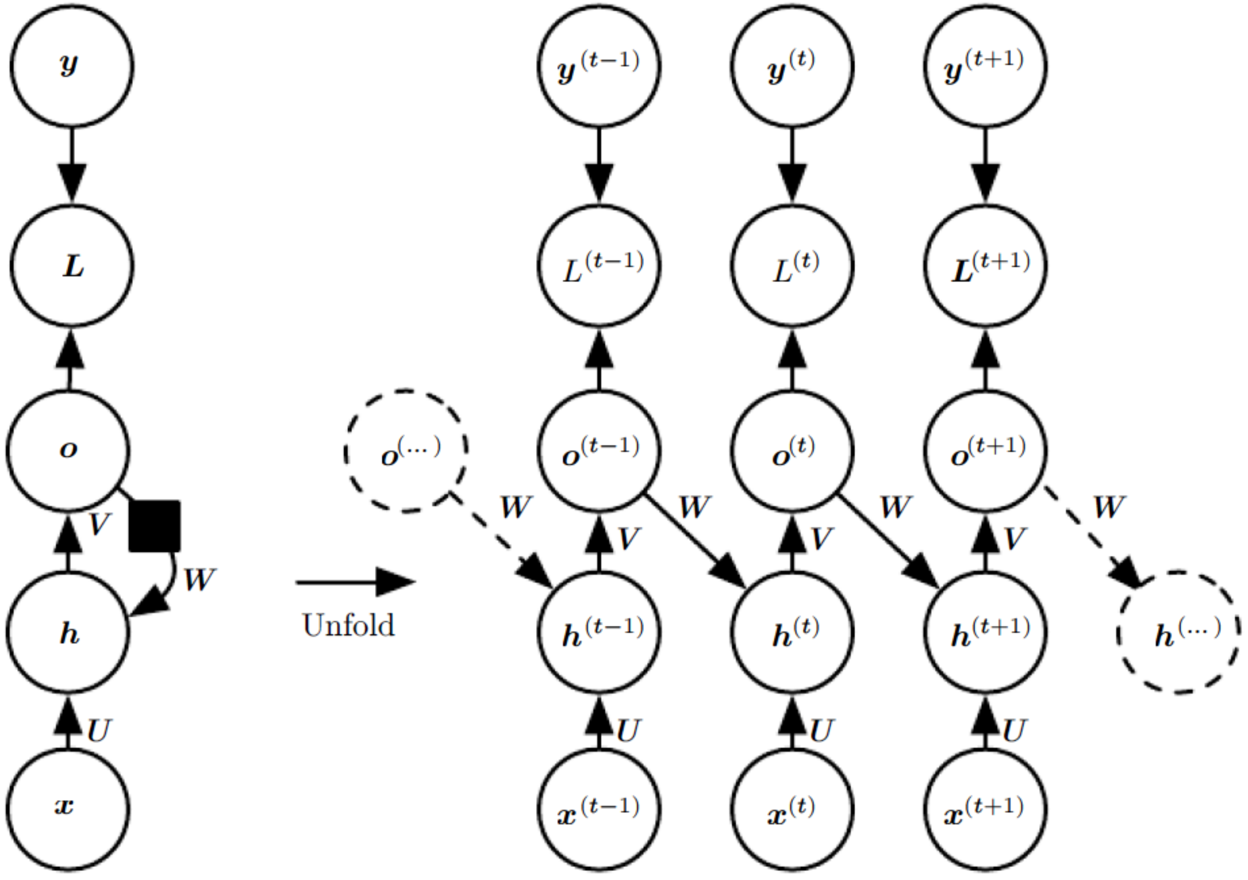
$$h_t = tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

It also depends on W_{hh}



Different Types of Recurrency

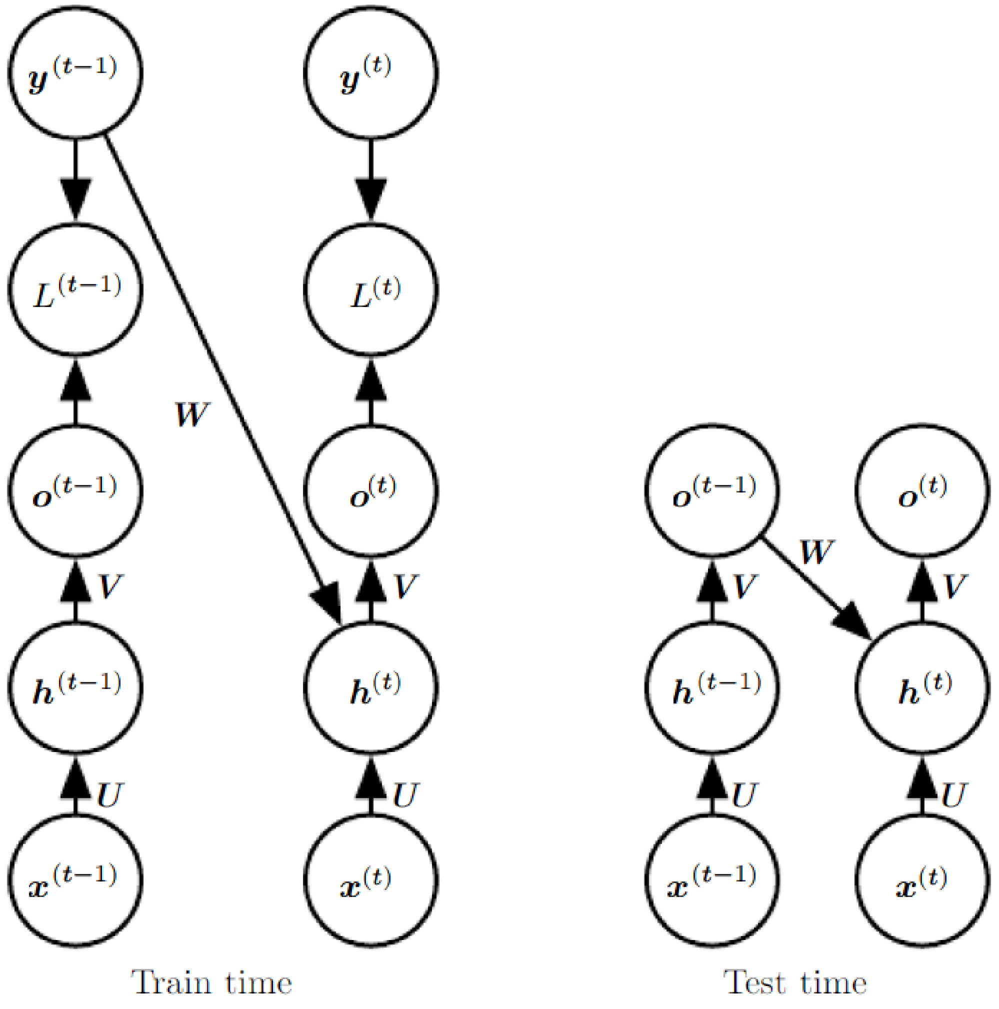
- RNNs that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step
- Carry less information from past, because
  - Output nodes typically have a lower dimensionality than hidden nodes
  - Output nodes are strongly influenced by ground-truth y during training



(Fig. 10.4 in GBK)

Teacher Forcing

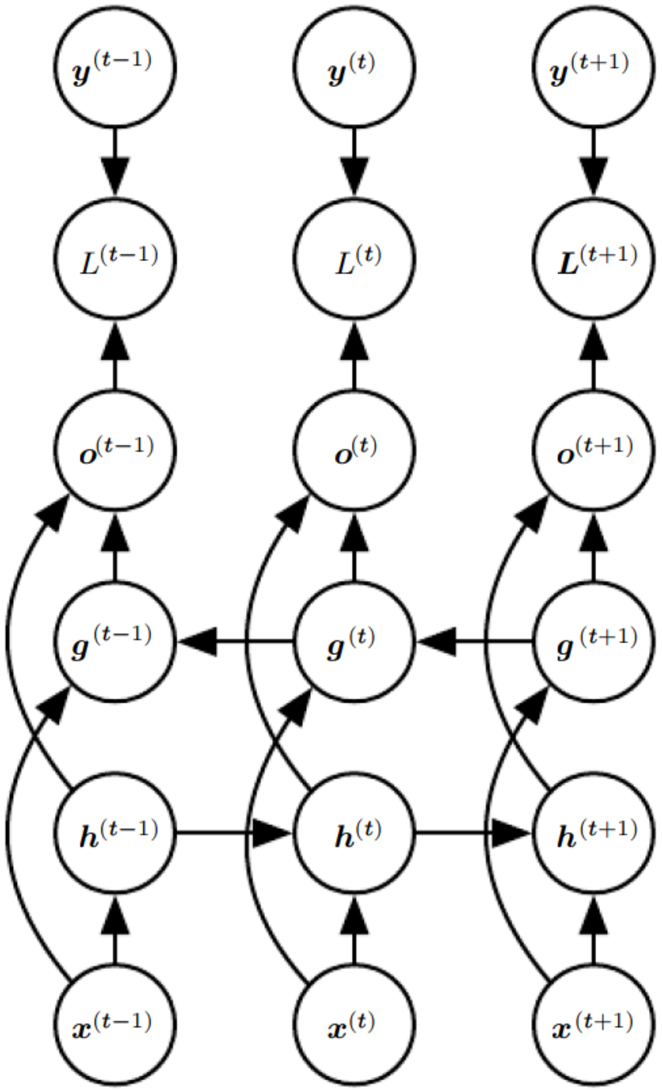
- Training can be parallelized by teaching forcing for RNNs that only have recurrent links from output to hidden
- It essentially only trains network to make 1-step predictions
- During inference, $y^{(t-1)}$ is not available for predicting $y^{(t)}$, causing mismatch from training
  - Scheduled sampling: mix teacher-forced inputs and free-run inputs during training with a ratio that gradually decreases



(Fig. 10.6 in GBK)

Bidirectional RNN

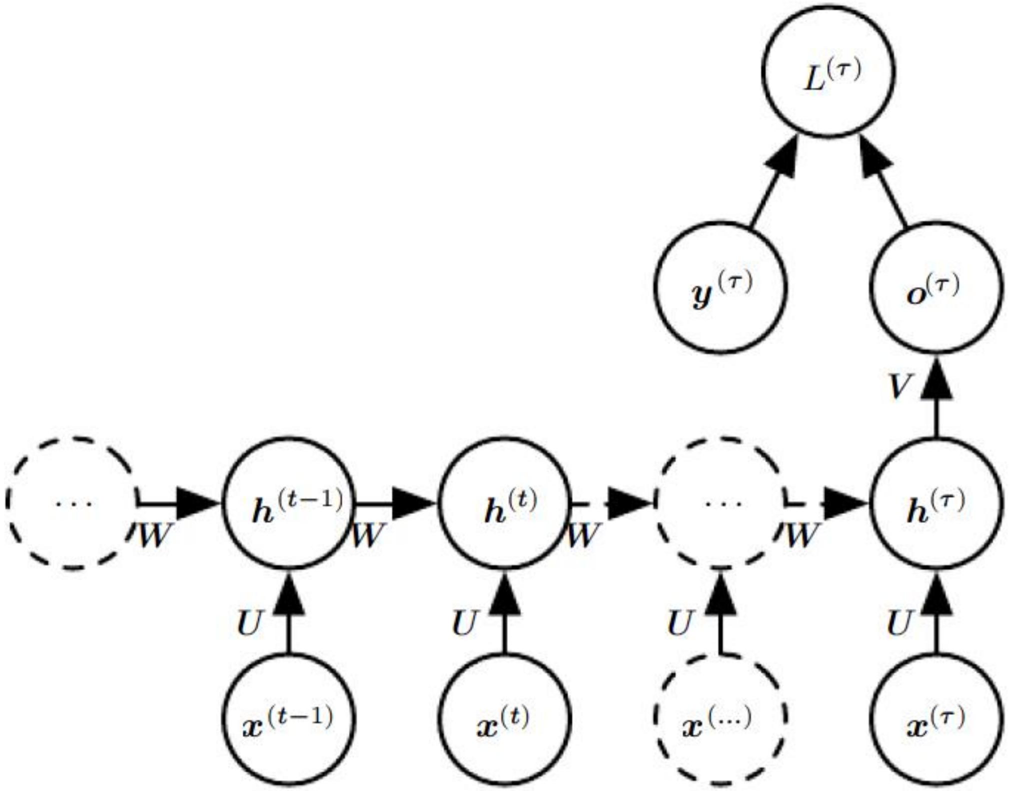
- RNNs introduced so far are causal, i.e., the output at the current time step is only affected by the current input and past inputs
- In some applications (e.g., filling a missing word in a sentence, speech recognition), output has dependencies on inputs from both sides
- Let’s use two RNNs, one for each direction
- Their hidden values work together to give output



(Fig. 10.11 in GBK)

RNN with a Single Output

- Some tasks only require a single output from the input sequence
  - E.g., phoneme classification, sound event recognition

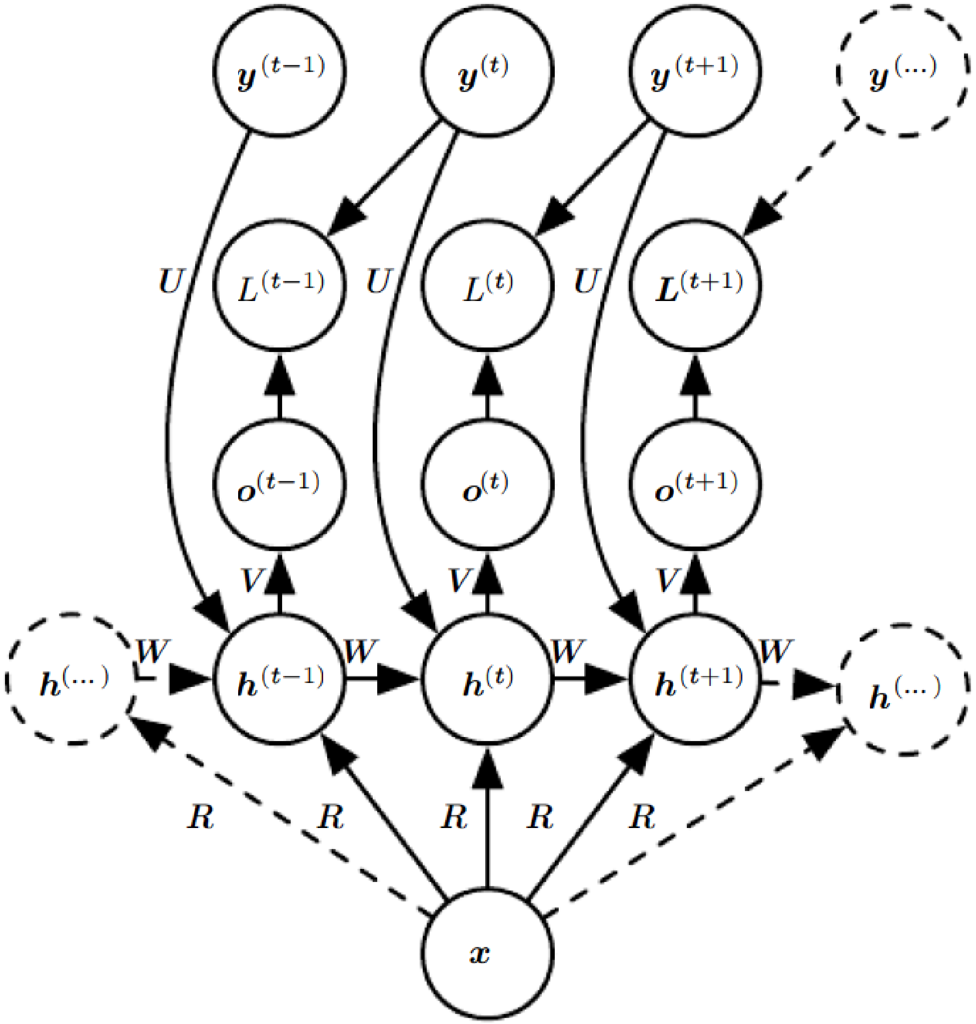


(Fig. 10.5 in GBK)

RNN with Context Conditioning

- Output a sequence from a conditioning vector
  - E.g., laughter sound generation, conditioned on the type of laughter
  - E.g., image captioning, conditioned on image
  - E.g., emotional talking face generation, conditioned on emotion label

- This conditioning vector can be input to the network
  - As extra input at each time step (right figure)
  - As the initial state $\boldsymbol{h}^{(0)}$
  - Both



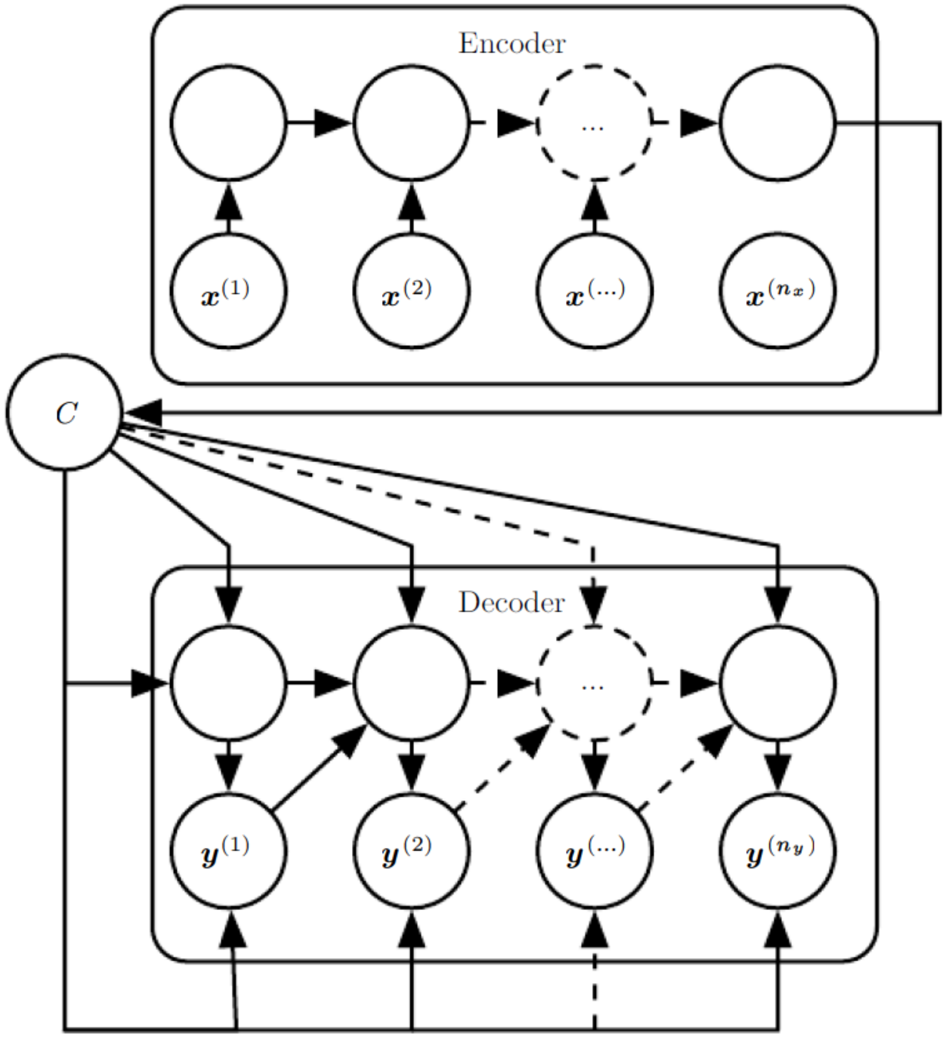
(Fig. 10.9 in WBK)

Encoder-Decoder Sequence-to-Sequence RNNs

- Sometimes the input and output sequences are of different length
  - E.g., machine translation from English to Chinese
  - E.g., audio captioning

- Encoder is an RNN with a single output

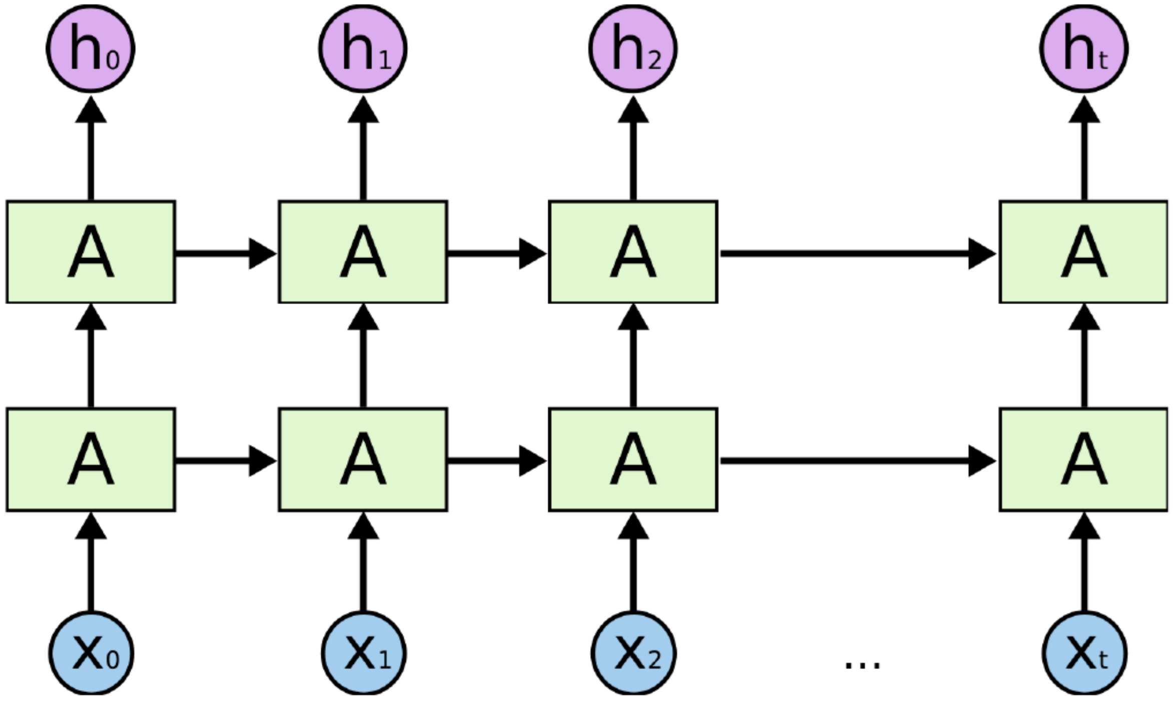
- Decoder is an RNN with context conditioning



(Fig. 10.12 in GBK)

Deep RNNs

- RNNs we introduced so far have only one hidden layer
- There are many ways to make them deeper, but a common way is to stack RNNs



Vanishing & Exploding Gradients

- Recurrency applies the same function repeatedly, and will exponentially diminish or boost certain effects
- Look at linear recurrency as an example

$$\boldsymbol{h}^{(t)} = \boldsymbol{W}\boldsymbol{h}^{(t-1)} = \boldsymbol{W}^{t}\boldsymbol{h}^{(0)}$$

- Let $\boldsymbol{W}$ have eigenvalue decomposition

$$\boldsymbol{W} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1}$$

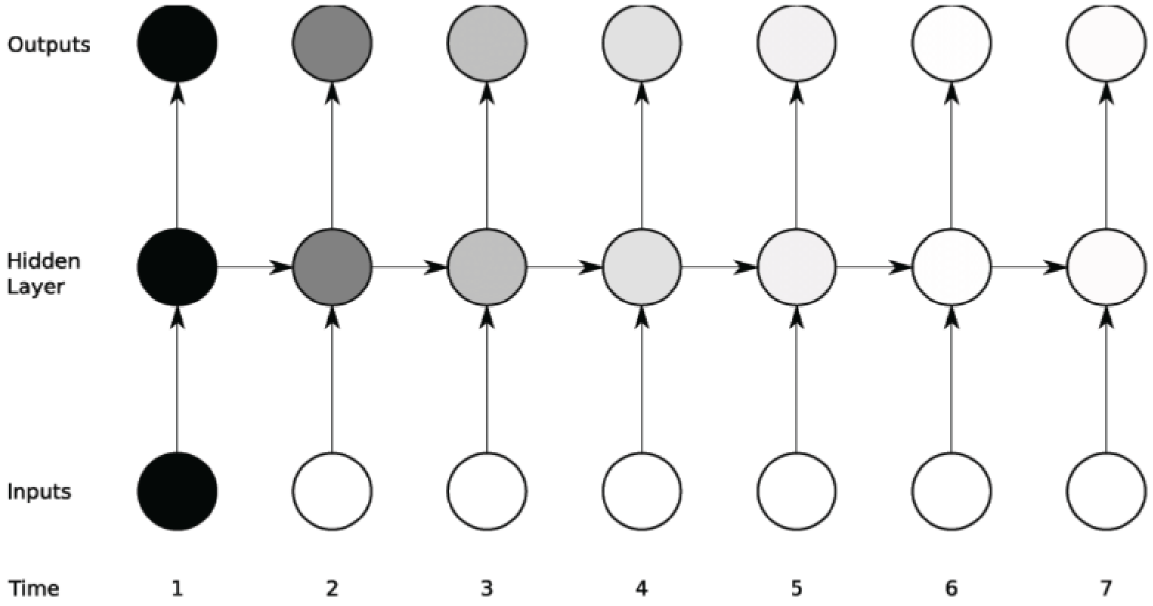
- Then we have

$$\boldsymbol{h}^{(t)} = \boldsymbol{Q}\boldsymbol{\Lambda}^{\mathrm{t}}\boldsymbol{Q}^{-1}\boldsymbol{h}^{(0)}$$

- Eigenvalues are raised to the power of t!
  - If $\boldsymbol{h}^{(0)}$ is aligned with an eigenvector that is greater than 1, then explode
  - If $\boldsymbol{h}^{(0)}$ is aligned with an eigenvector that is smaller than 1, then vanish

Vanishing & Exploding Gradients

- Vanishing gradients are very common for RNNs



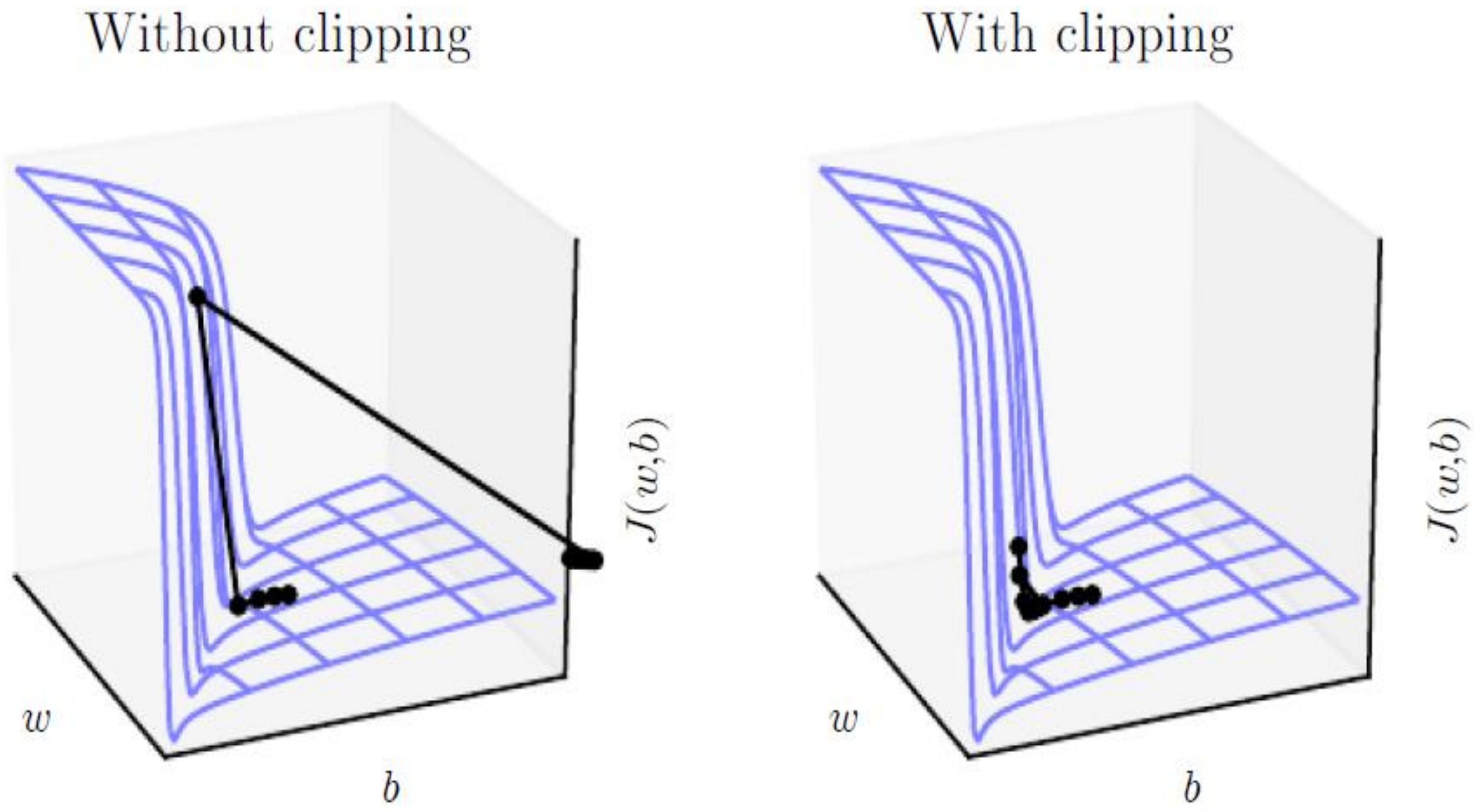
Darkness indicates the influence of input at time 1
Figure from [Graves, 2008]

- Exploding gradients also happen, and it damages the optimization very much

Gradient Clipping

- Too big gradients will make too big updates of network parameters
- Clip the norm of gradients $\boldsymbol{g}$ to v:

$$\text{if } ||\boldsymbol{g}|| > v$$
$$\boldsymbol{g} \leftarrow \frac{\boldsymbol{g}v}{||\boldsymbol{g}||}$$



(Fig. 10.17 in GBC)

Improving Long-Term Dependency Modeling

- Temporal dependencies in data can be very long
  - E.g., music rhythmic structure is at the scale of seconds, where each second often contains 44100 samples (time domain) or ~100 frames (time-frequency domain)

- Influence of input vanishes exponentially over time steps
  - In practice, after ten steps, influence is already negligible

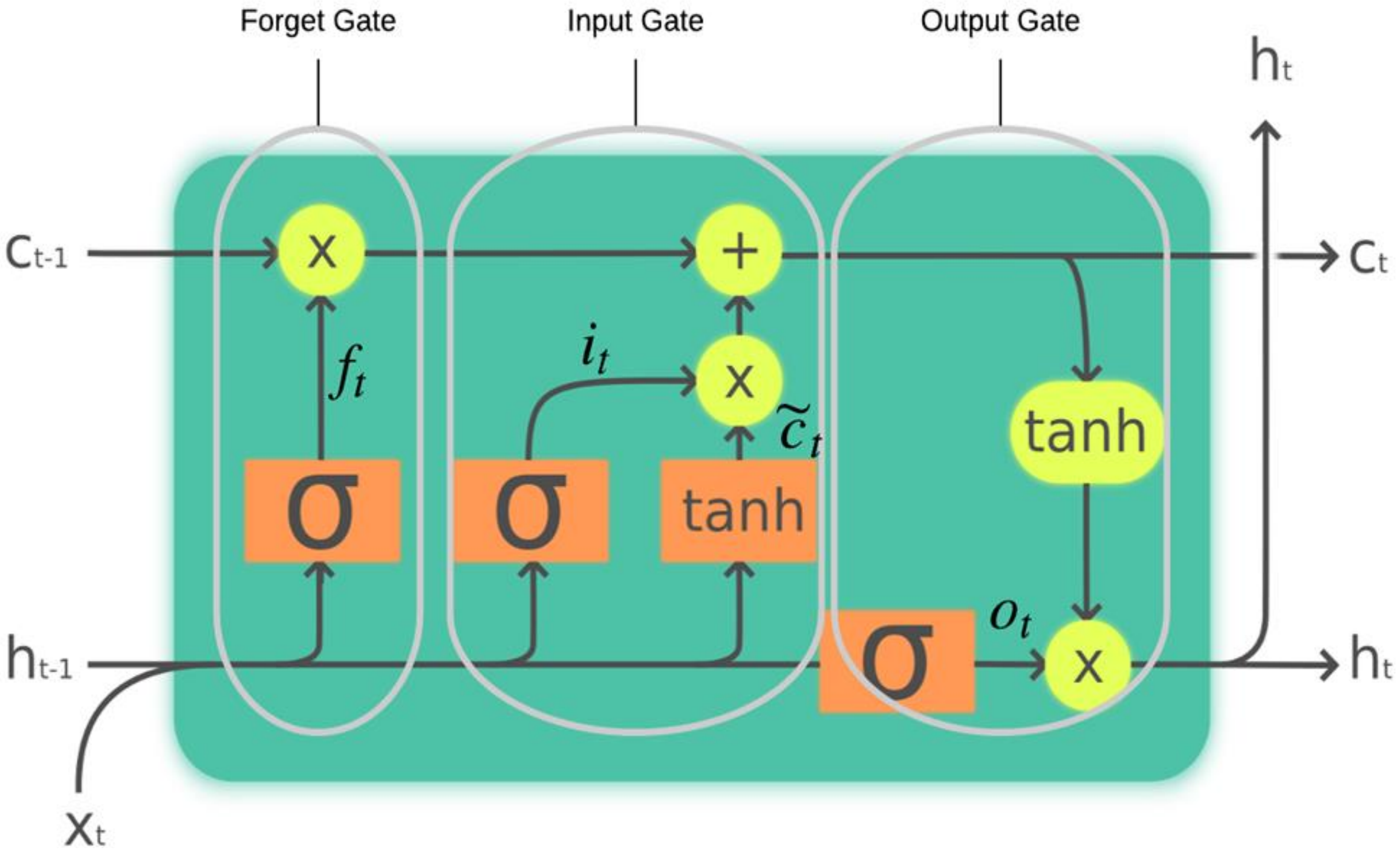
- Several ways to improve long-term dependency
  - Add skip connections through time: allows information to flow with fewer time steps
  - Add linear self-connections to hidden units, called leaky units, similar to running average: $\mu^{(t)} \leftarrow \alpha\mu^{(t-1)} + (1-\alpha)v^{(t)}$. When α is close to 1, it allows hidden units to remember information for a long time.
  - Add gates to control information flow

Gated Architectures - LSTM

- Cell state (leaky unit) is the internal memory
- Three information gates perform delete/write/read operations on memory

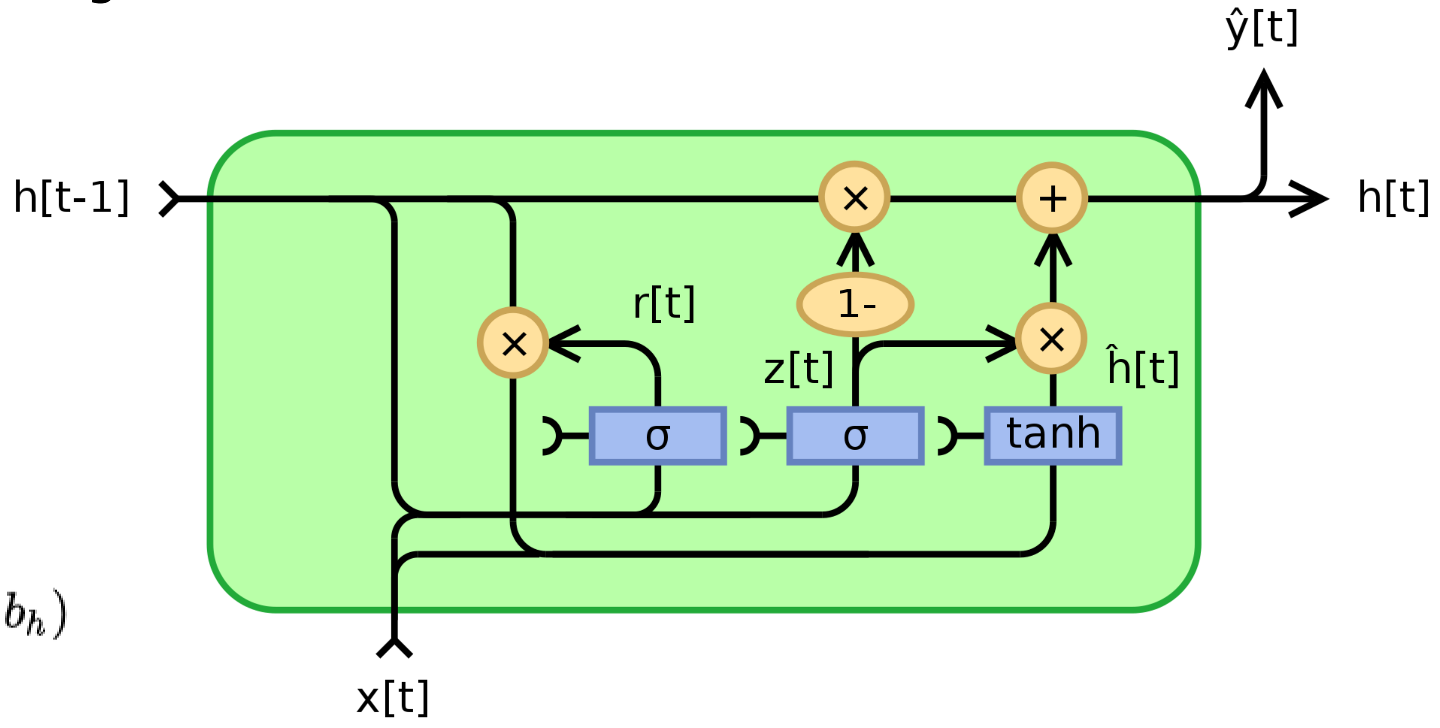
$$i_t = \sigma(w_i[h_{t-1}, x_t] + b_i)$$
$$f_t = \sigma(w_f[h_{t-1}, x_t] + b_f)$$
$$o_t = \sigma(w_o[h_{t-1}, x_t] + b_o)$$

$$\tilde{c}_t = tanh(w_c[h_{t-1}, x_t] + b_c)$$
$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$$
$$h_t = o_t * tanh(c^t)$$



Gated Architecture - GRU

- Gated Recurrent Unit (GRU)
  - A single gate to simultaneously control the forgetting factor and the updating operation of the state unit
  - Fewer parameters than LSTM
  - Similar performance



(Figure from https://en.wikipedia.org/wiki/Gated_recurrent_unit)

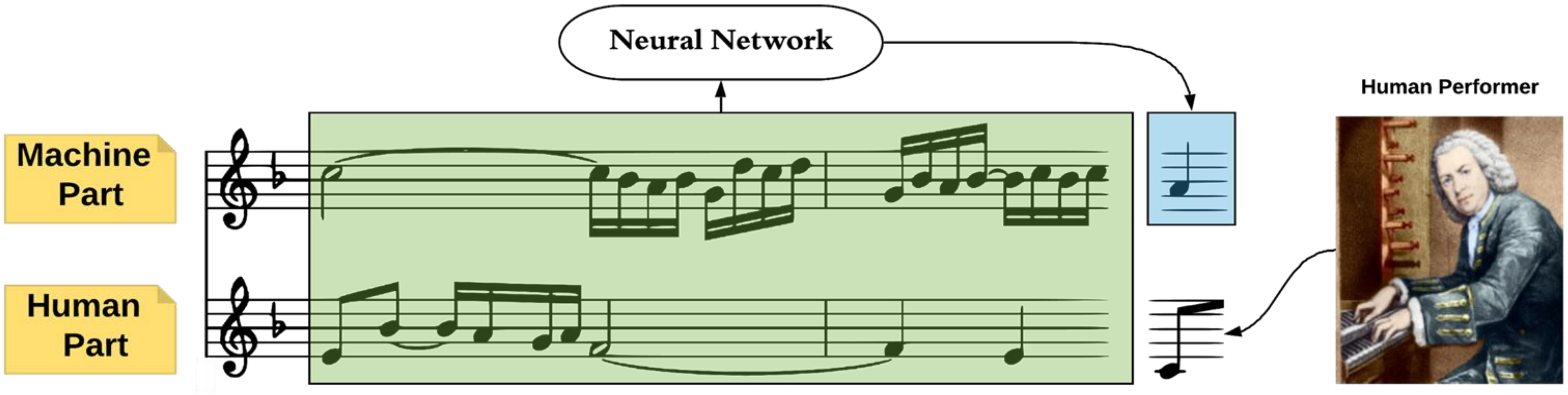
Update gate $z_t = \sigma_g(W_z x_t + U_z h_{t-1} + b_z)$

Reset gate $r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r)$

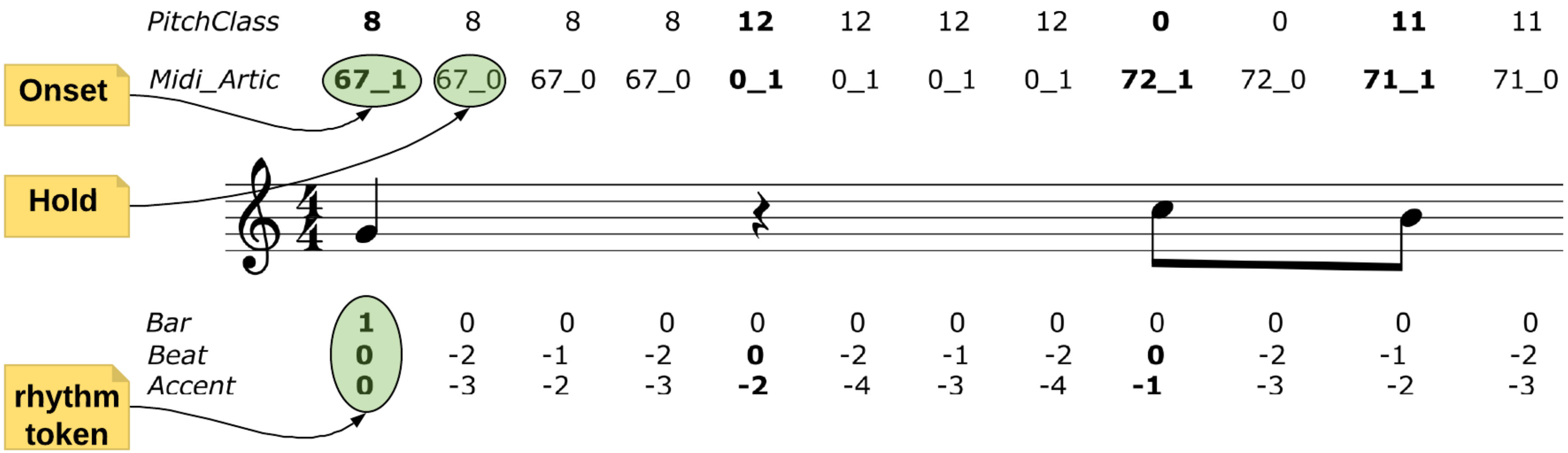
$\hat{h}_t = \phi_h(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$

Output $h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \hat{h}_t$

Application: Music Generation

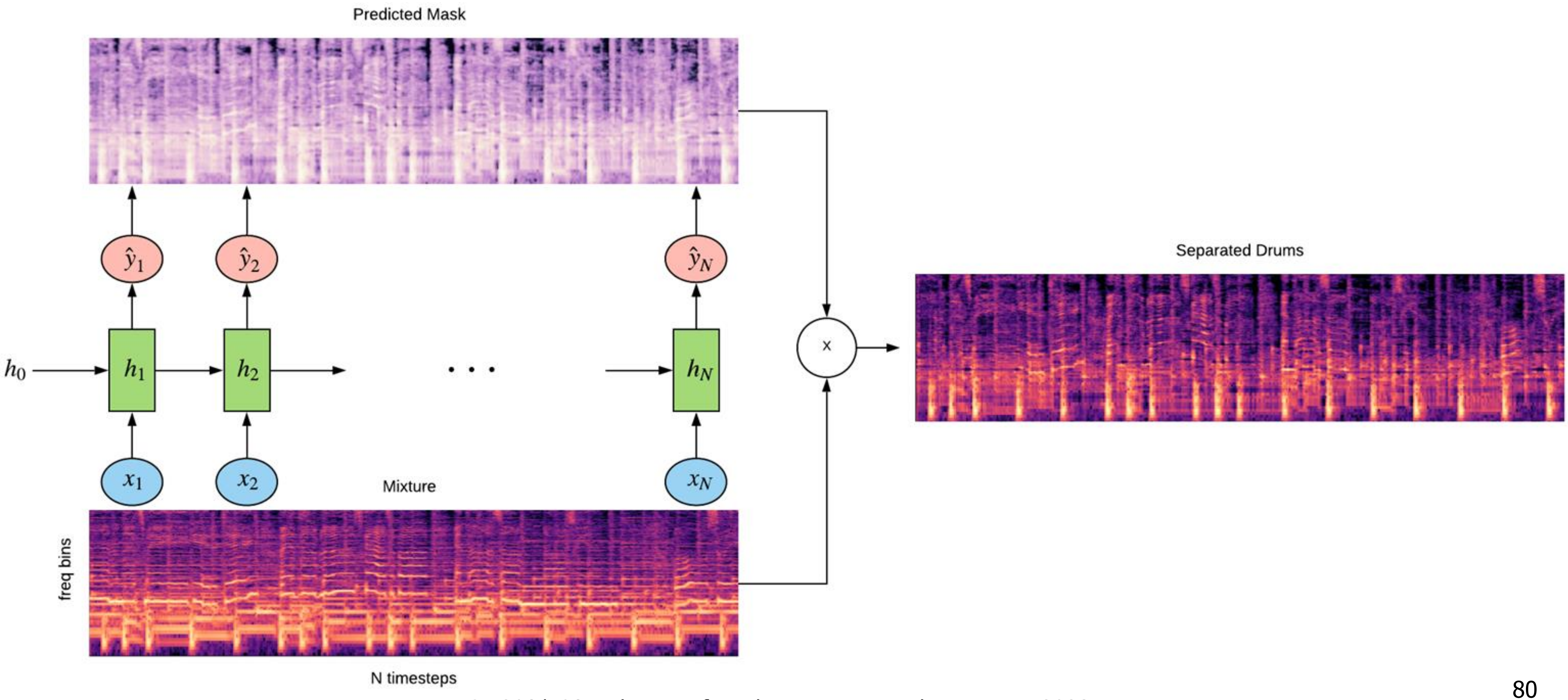


Benetatos, VanderStel, & Duan, **BachDuet: A deep learning system for human-machine counterpoint improvisation**, NIME, 2020.



Yan, Lustig, Vaderstel, & Duan, **Part-invariant model for music generation and harmonization**, ISMIR, 2018.

Application: Audio Source Separation



RNN Summary

- Recurrent Neural Networks (RNNs)
  - Weight sharing over time
  - Recurrent links to carry information infinitely long (in theory)
- Different kinds of recurrencies
  - Hidden to hidden
  - Output to hidden
- Different RNN architectures
  - N to N, N to 1, 1 to N, N to M
- Back Propagation Through Time (BPTT)
  - Vanishing and exploding gradients due to repeatedly compositing the same function
  - Gradient clipping
- Long Short-Term Memory
  - Linear self connections to remember information longer
  - (Learnable) gated architecture to control information flow