POLITECNICO
MILANO 1863

Machine Learning Methods for Communication Networks and Systems

Francesco Musumeci

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB)

Politecnico di Milano, Milano, Italy

Part I – 3: Neural networks

Outline

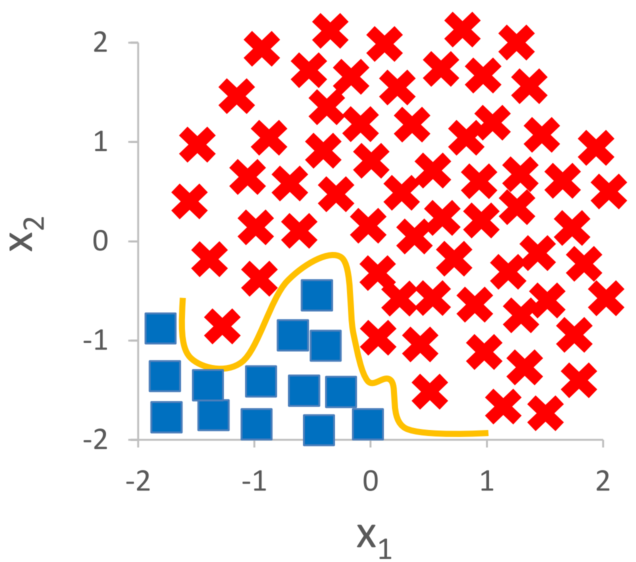
- Introduction
- Neural networks representation
- Multiclass classification
- Parameter learning
- Neural networks for time series

Outline

- Introduction
- Neural networks representation
- Multiclass classification
- Parameter learning
- Neural networks for time series

Introduction

- Why do we need a new algorithm?
  - Traditional problems are complex
  - Use of polynomial regression is not always a good solution
    - Many features can have a role → increased features space



$h(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \dots)$

Suppose we have 100 different features and we want to add all quadratic terms:

$x_1^2, x_1 x_2, \dots, x_1 x_{100}$

$x_2^2, \dots, x_2 x_{100}$

…

$x_{99}^2, x_{99} x_{100}$

x_{100}^2

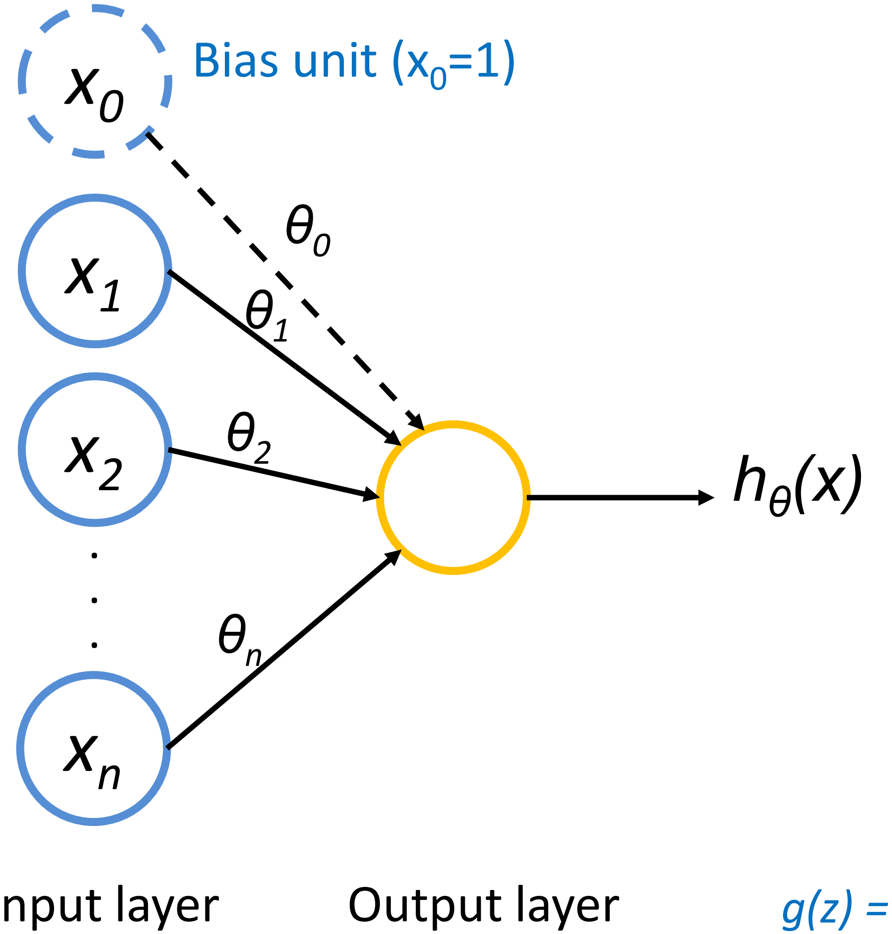
n "original" features require $O(n^2)$ quadratic terms!

Outline

- Introduction
- Neural networks representation
- Multiclass classification
- Parameter learning
- Neural networks for time series

Neural networks representation
Logistic unit

- The simplest neural network



$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \dots \\ \theta_n \end{bmatrix}$$

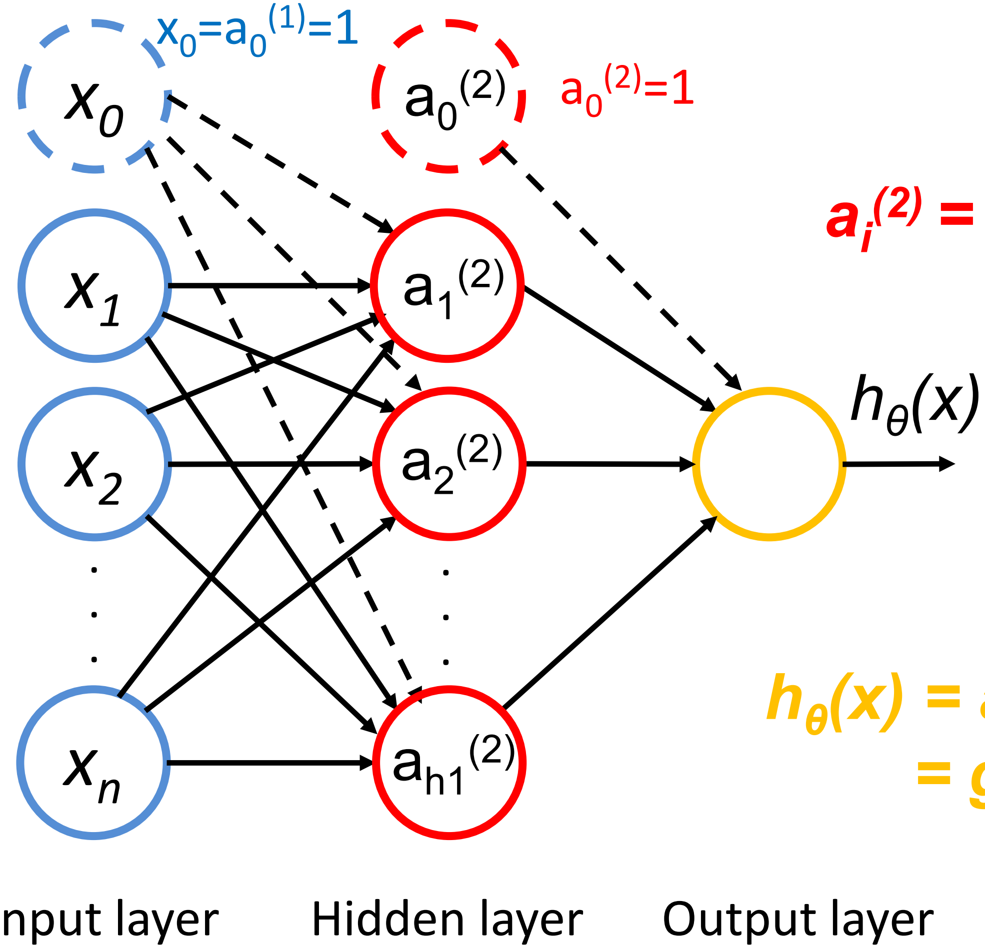
inputs or features — weights

$$h_\theta(x) = g(\theta^T x) = 1/(1+e^{-(\theta^T x)})$$

$g(z) = 1/(1+e^{-z})$

Neural networks representation
Multiple layers

- A "collection" of interacting neurons

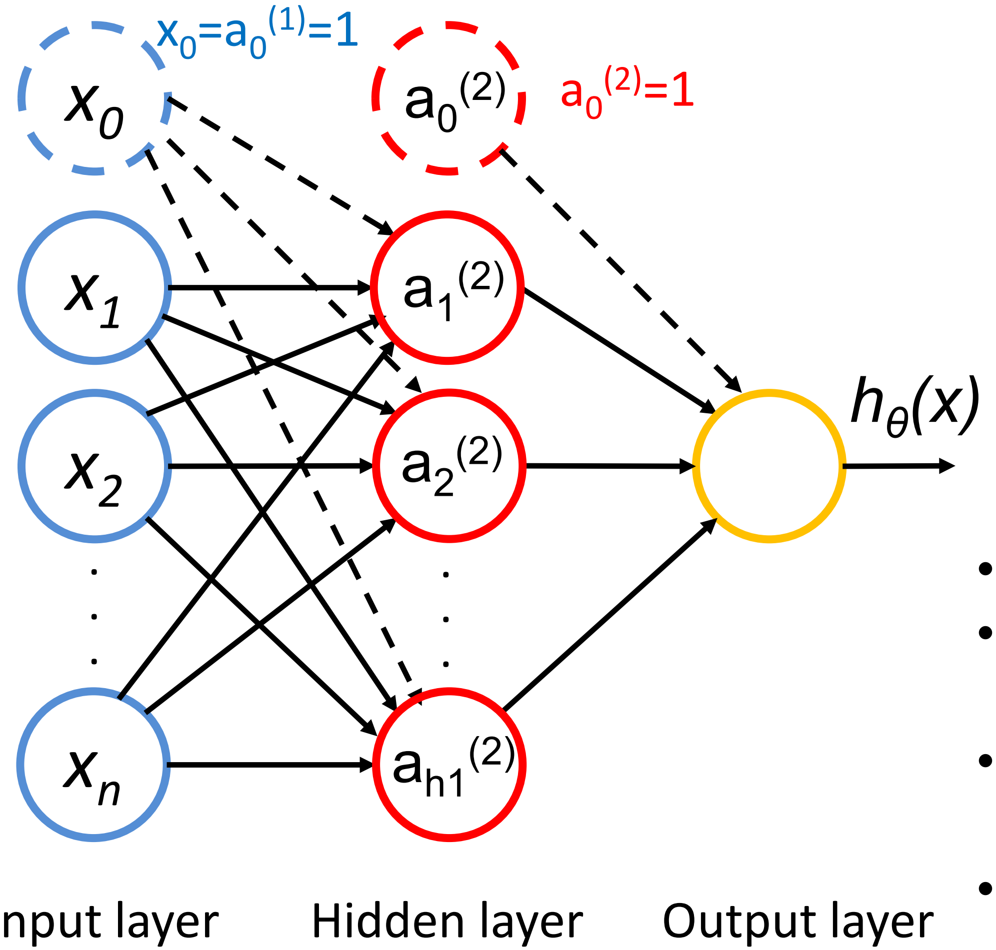


$$a_i^{(2)} = g(\overbrace{\theta_{i0}^{(1)}x_0 + \ldots + \theta_{in}^{(1)}x_n}^{z_i^{(2)}})$$

Forward propagation

$$h_\theta(x) = a_1^{(3)} =$$
$$= g(\theta_{10}^{(2)}a_0^{(2)} + \ldots + \theta_{1c}^{(2)}a_c^{(2)})$$

Neural networks representation
Notation

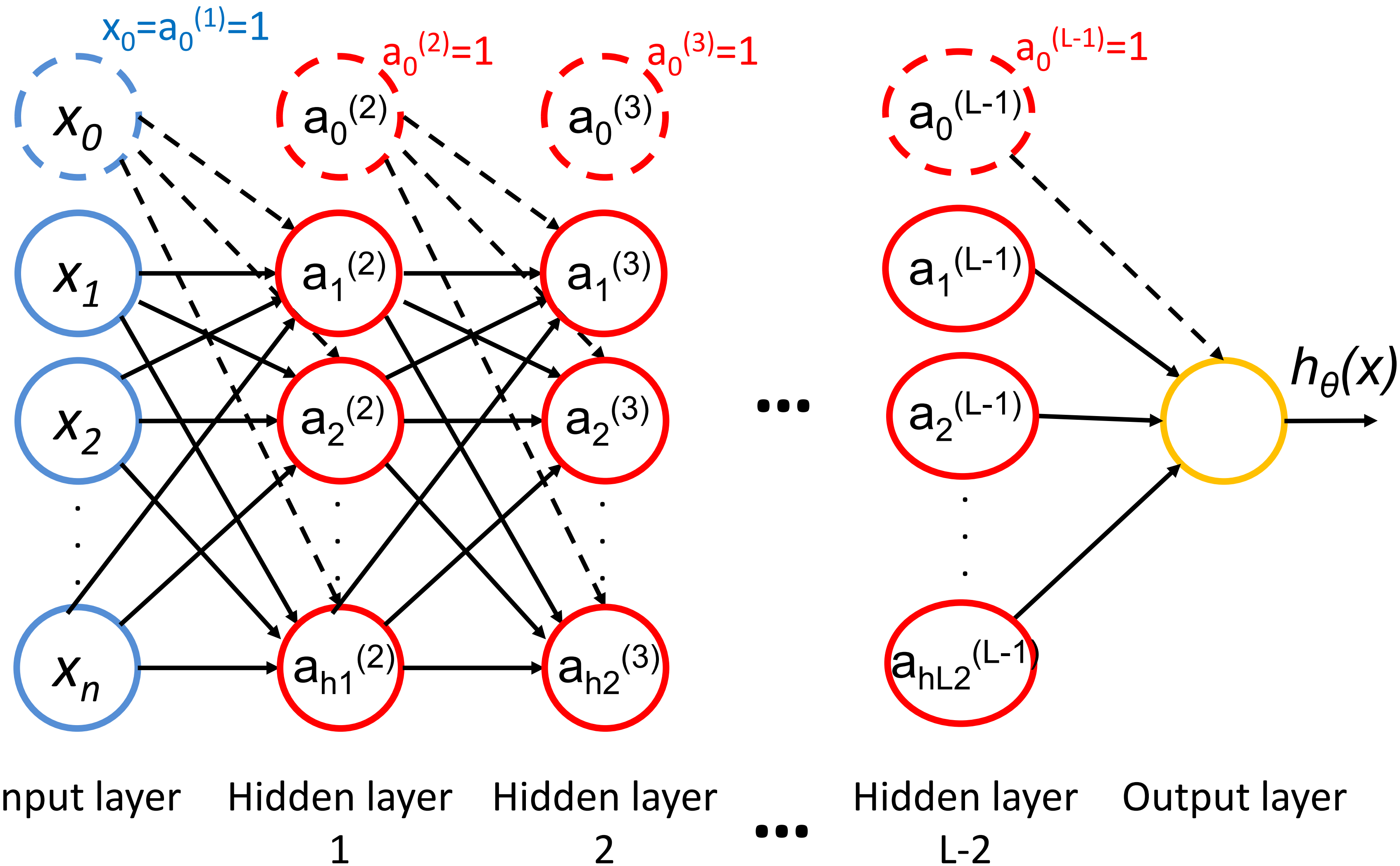


- $x_i=a_i^{(1)}$: *i*-th input unit (feature)
- $a_i^{(l)}=g(z_i^{(l)})$: activation of *i*-th unit in *l*-th layer
  - $z_i^{(l)} = \theta_{i0}^{(l-1)}a_0^{(l-1)} + \dots + \theta_{ic}^{(l-1)}a_c^{(l-1)}$
- $\Theta^{(l)}$: vector of weights between layers *l* & *(l+1)*
  - $\theta_{ij}^{(l)}$: weight between *i*-th unit in layer *(l+1)* and *j*-th unit in layer *l*
- $h_\theta(x) = a_1^{(L)}$: output unit

- *n*: nr. of features
- *L*: nr. of layers ("*L-2*" is the nr. of hidden layers)
- h_l: nr. of hidden neurons in *l*-th hidden layer
- *g(•)*: activation function (e.g., sigmoid)

Neural networks representation
Deep Neural Networks (DNN)

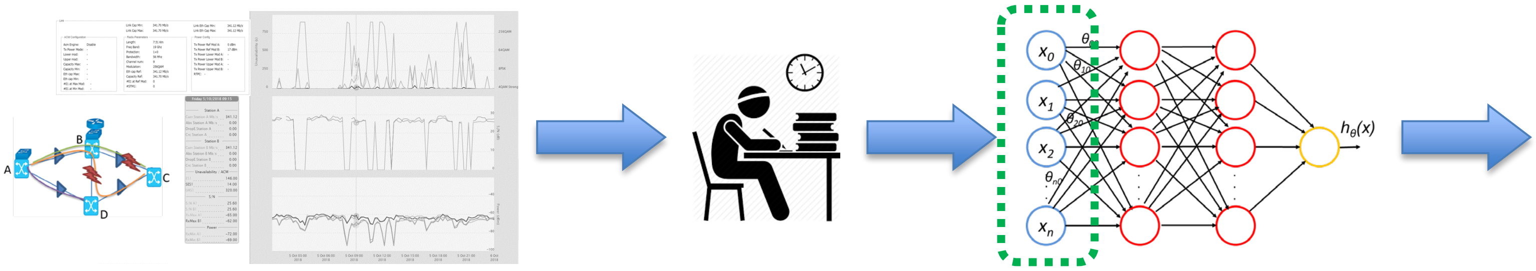
Many layers increase the chance to discover "hidden" features as non-linear combination of raw data



Features selection and Deep Neural Networks (DNN)

- **(Shallow) Neural Networks**

Processed features



Final model

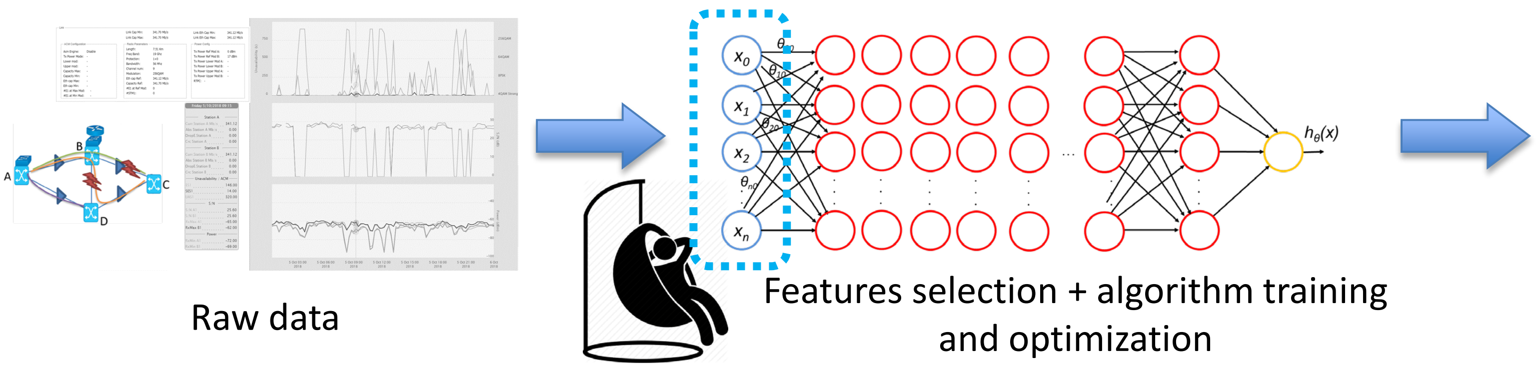
Raw data

Features selection

Algorithm training and optimization

- **Deep Neural Networks**

Raw features



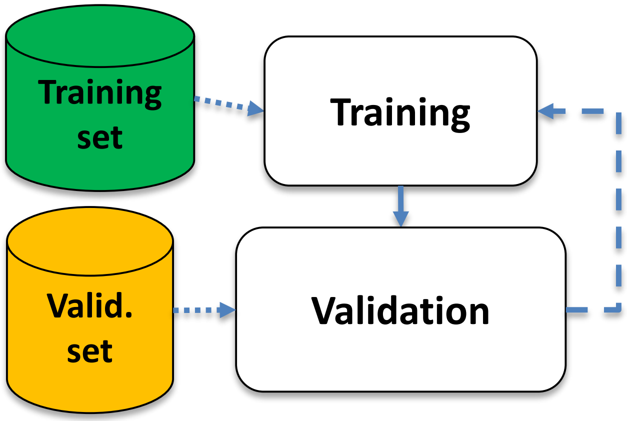
Final model

Raw data

Features selection + algorithm training and optimization

Neural networks representation
Issues with multiple layers

- Different types of neural networks can be designed to capture complex properties of features
- How many hidden layers?
- How many hidden units per layer?
- Same number of units per layer?
- Which activation function? Same for all the layers?
- Which connections among different layers?



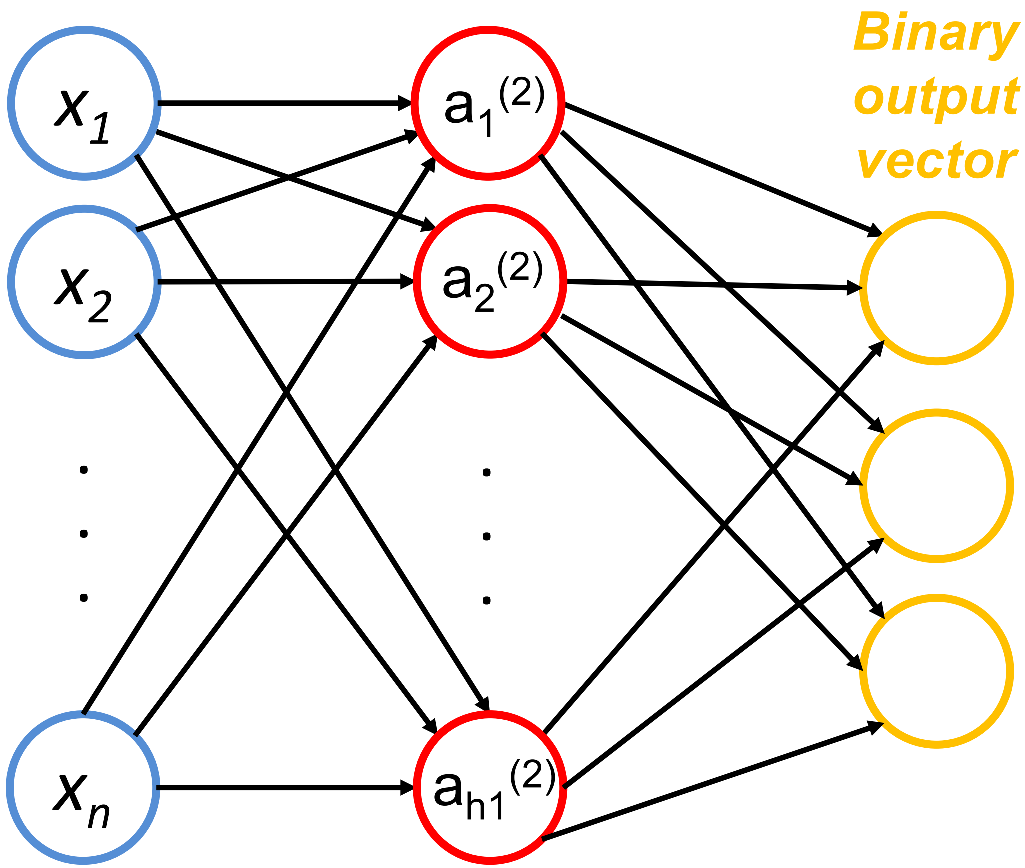
Validation approach, we'll see later in the course

Outline

- Introduction
- Neural networks representation
- Multiclass classification
- Parameter learning
- Neural networks for time series

Multiclass classification

- Example with 3 classes



One-hot encoding

$$h_\theta(x) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \text{Class 1}$$

$$h_\theta(x) = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \text{Class 2}$$

$$h_\theta(x) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad \text{Class 3}$$

(x,y) in the training set are objects of one class only, i.e., *y* is a column vector with $y_i=0$, for all $i \neq k$, $y_k=1$ if *(x,y)* belongs to class "*k*"

Outline

- Introduction
- Neural networks representation
- Multiclass classification
- Parameter learning
- Neural networks for time series

Parameter learning
Optimization objective

- How do we choose parameters θ to have a good fit?
- Minimize cost function (as in logistic regression)
  - Cost function used for logistic regression:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right]$$

  - With neural networks (we refer to the general case with multiple classes):

$$h_\theta(x) \in R^K; \quad (h_\theta(x))_k : k^{th} \text{ element of vector } h_\theta(x)$$

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)(i)}\log(h_\theta(x^{(i)}))_k + (1-y_k^{(i)(i)})\log(1-(h_\theta(x^{(i)}))_k)\right]\right]$$

Parameter learning
Backpropagation algorithm

- Given the cost function

$$h_\theta(x) \in R^K; \quad (h_\theta(x))_k : k^{th} \text{ element of vector } h_\theta(x)$$

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)(i)} \log(h_\theta(x^{(i)}))_k + (1 - y_k^{(i)(i)}) \log(1 - (h_\theta(x^{(i)}))_k)\right]\right]$$

we can iteratively update parameters theta via (e.g.) (batch) gradient descent:

$$\boxed{\theta_{ij}^{(l)} = \theta_{ij}^{(l)} - \alpha \frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta)}$$

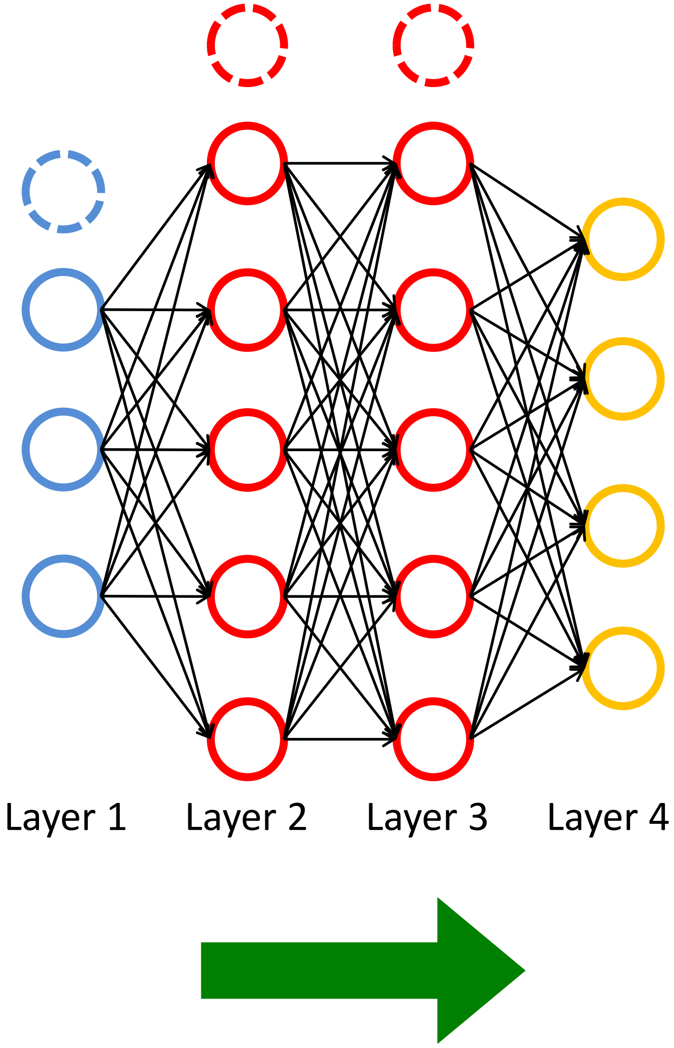
- Problem: compute derivative terms is complex

→ Gradient computation via "Error backpropagation"

Parameter learning
Backpropagation algorithm

- Given one training example *(x,y)*

- Forward propagation steps:
  - $a^{(1)} = x$ (include bias input unit)
  - $z^{(2)} = \Theta^{(1)}a^{(1)}$
  - $a^{(2)} = g(z^{(2)})$ (include bias unit)
  - $z^{(3)} = \Theta^{(2)}a^{(2)}$
  - $a^{(3)} = g(z^{(3)})$ (include bias unit)
  - $z^{(4)} = \Theta^{(3)}a^{(3)}$
  - $a^{(4)} = h_\theta(x) = g(z^{(4)})$



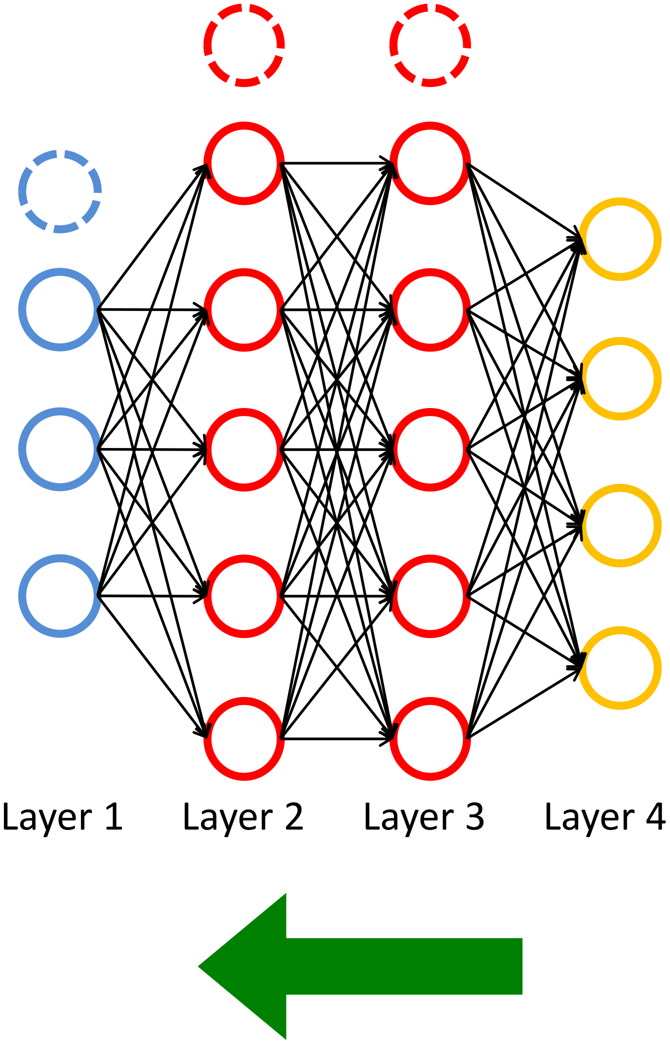
Parameter learning
Backpropagation algorithm

- $\delta_j^{(l)}$: error at node *j* of layer *l*
  - *recall:* $a_i^{(l)}=g(z_i^{(l)})$
  - $z_i^{(l)}= \theta_{i0}^{(l-1)}a_0^{(l-1)} + \ldots + \theta_{ic}^{(l-1)}a_c^{(l-1)}$
- Backward error propagation:
  – For units in the output layer

$$\delta_j^{(4)} = a_j^{(4)} - y_j = h_\theta(x) - y_j$$

  – For units in hidden layers

$$\delta_j^{(l)} = [\ \Sigma_i\ \theta_{ij}^{(l)}\delta_i^{(l+1)}\ ] * [a_j^{(l)}*(1 - a_j^{(l)})]$$



Example:

$$\delta_5^{(3)} = [\ \Sigma_i\ \theta_{i5}^{(3)}\delta_i^{(4)}\ ] * [a_5^{(3)}*(1 - a_5^{(3)})] =$$

$$= [\theta_{15}^{(3)}\delta_1^{(4)}+\theta_{25}^{(3)}\delta_2^{(4)}+\theta_{35}^{(3)}\delta_3^{(4)}+\theta_{45}^{(3)}\delta_4^{(4)}] * [a_5^{(3)}*(1 - a_5^{(3)})]$$

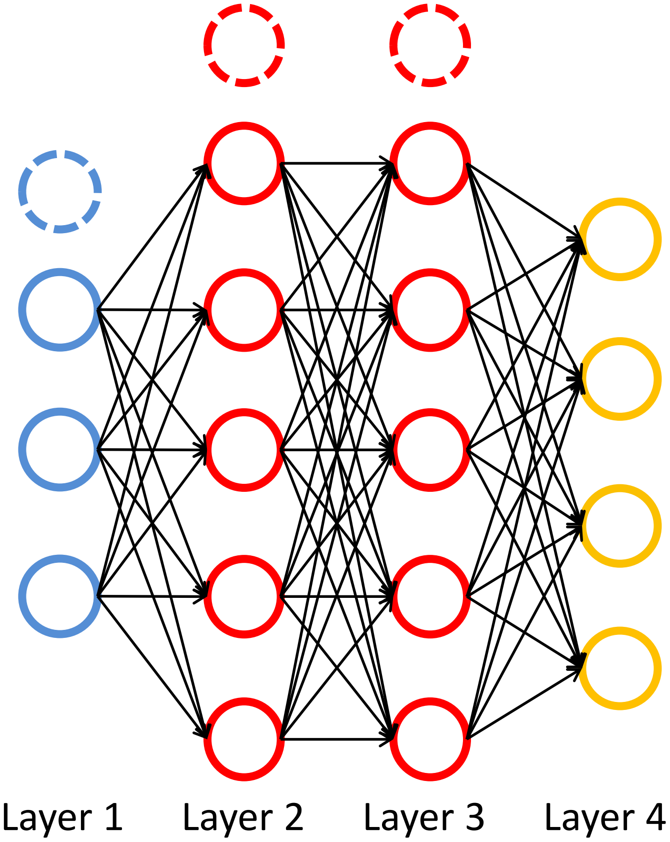
Parameter learning
Backpropagation algorithm

- $\delta_j^{(l)}$: error at node *j* of layer *l*
  - *recall:* $a_i^{(l)}=g(z_i^{(l)})$
  - $z_i^{(l)}= \theta_{i0}^{(l-1)}a_0^{(l-1)} + \ldots + \theta_{ic}^{(l-1)}a_c^{(l-1)}$
- Backward error propagation:
  - For units in the output layer

$$\delta_j^{(4)} = a_j^{(4)} - y_j = h_\theta(x) - y_j$$

  - For units in hidden layers

$$\delta_j^{(l)} = [\ \Sigma_i\ \theta_{ij}^{(l)}\delta_i^{(l+1)}\ ] * [a_j^{(l)}*(1 - a_j^{(l)})]$$



Compute derivatives for gradient descent:

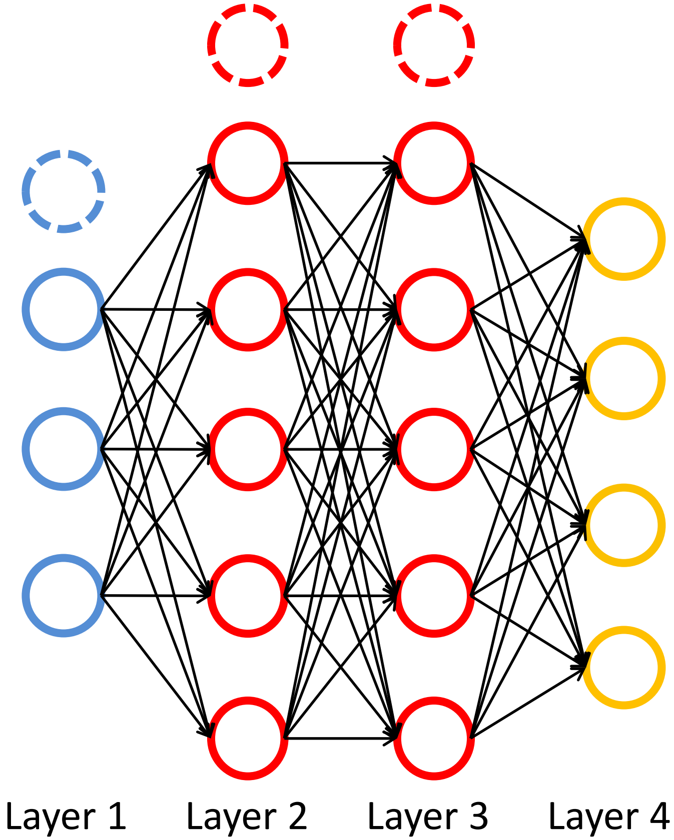
$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = \mathrm{a}_j^{(l)} \delta_i^{(l+1)} \quad \forall i, j, l$$

← This considers only one example

Summarizing…

Backpropagation algorithm

- Given a training set w/ m examples $\{(x^{(1)},y^{(1)}), (x^{(2)},y^{(2)}),\ldots, (x^{(m)},y^{(m)})\}$
- Parameter learning steps:
  - Set $\Delta_{ij}^{(l)} = 0$ for all i, j, l
  - For p = 1 to m (all training examples)
    - $a^{(1)} = x^{(p)}$
    - Compute $a^{(l)}$ for all layers $l=2,\ldots,L$ (forward propagation)
    - Set $\delta^{(L)} = a^{(L)} - y^{(i)}$
    - Compute $\delta^{(l)}$ for all layers $l=L-1,\ldots,2$ (backward propagation)
    - Update $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)}\delta_i^{(l+1)}$ for all (i, j, l)
  - Compute derivatives and update weights:



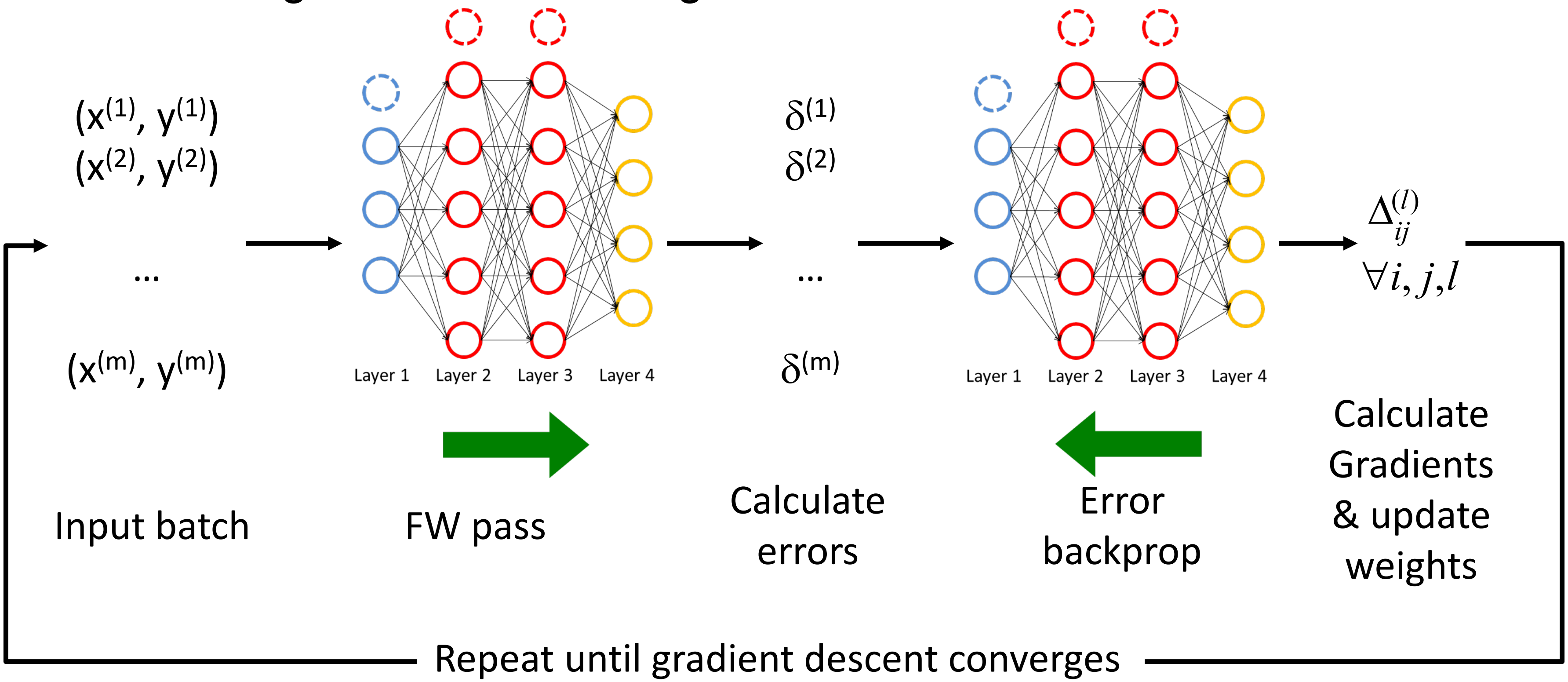
$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = \frac{1}{m}\Delta_{ij}^{(l)} \quad \forall i,j,l \quad \Rightarrow \quad \theta_{ij}^{(l)} = \theta_{ij}^{(l)} - \alpha \frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta)$$

Outline

- Introduction
- Neural networks representation
- Multiclass classification
- Parameter learning
  - Batch vs mini-batch gradient descent
- Neural networks for time series

Batch vs mini-batch gradient descent

- Parameter learning seen so far is performed computing gradients wrt the <u>entire training set of size *m*</u>
  - we are using the whole ***batch*** of *m* training examples at every step of the gradient descent algorithm



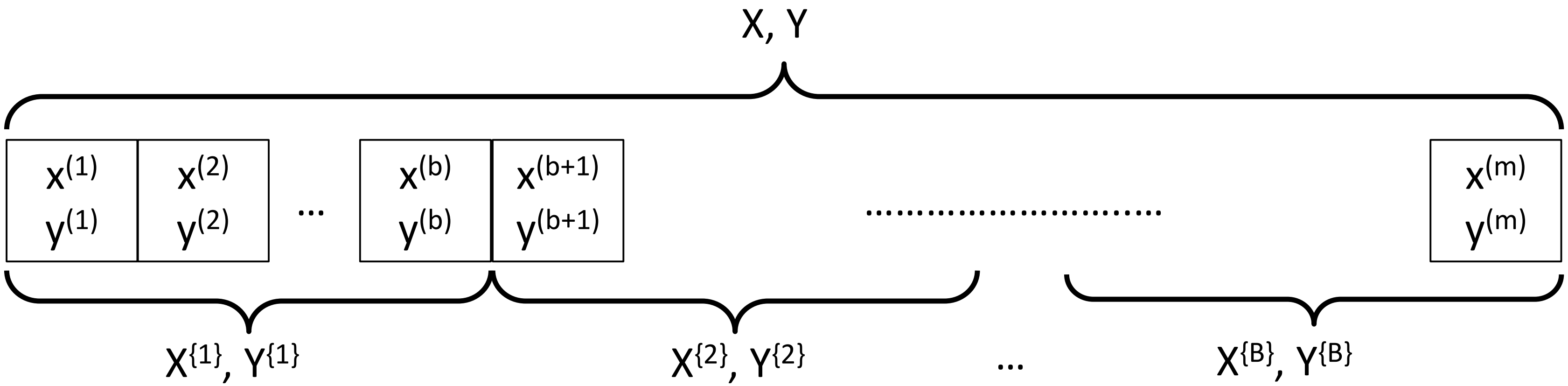
Batch vs mini-batch gradient descent

- Vectorization and matrix multiplication improve the efficiency of gradient descent as they parallelize the computation of gradients

- However, if the dataset is too large, performing backpropagation considering the entire training set can bee computationally intensive
  - huge CPU/memory requirements
  - slow convergence of gradient descent algorithm

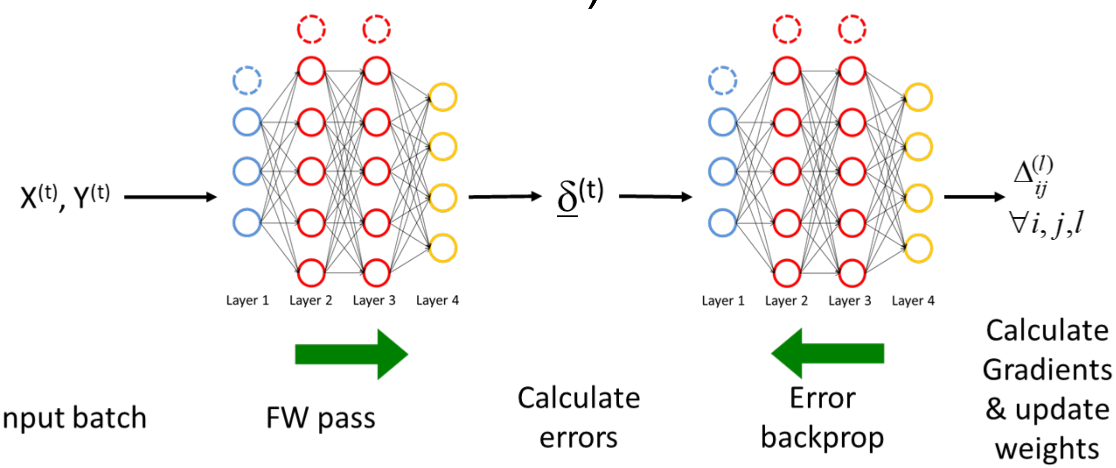
- Solution: split the dataset in many parts (mini-batches) and apply gradient descent to one part at a time

Batch vs mini-batch gradient descent



for t=1 to B (B is the nr of mini-batches)

do:



end

(1 epoch = 1 pass over the entire training set)

- Repeat the split and iterate for many epochs until gradient descent converges

Batch vs mini-batch gradient descent

- B is the nr of ***mini-batches***
  - B = 1 → *Batch* gradient descent
  - B = m → *Stochastic* gradient descent (one point per batch)
- Batch gradient descent
  - Cost function decreases monotonically with epochs
  - Very slow if training size *m* is large
- Stochastic gradient descent
  - Cost function can be very noisy
  - Significant improvement of the cost function can be obtained also for few iterations (pass of few points), but the improvement can be just due to "chance"
- Mini-batch gradient descent can be a good trade-off

Outline

- Introduction
- Neural networks representation
- Multiclass classification
- Parameter learning
- Neural networks for time series

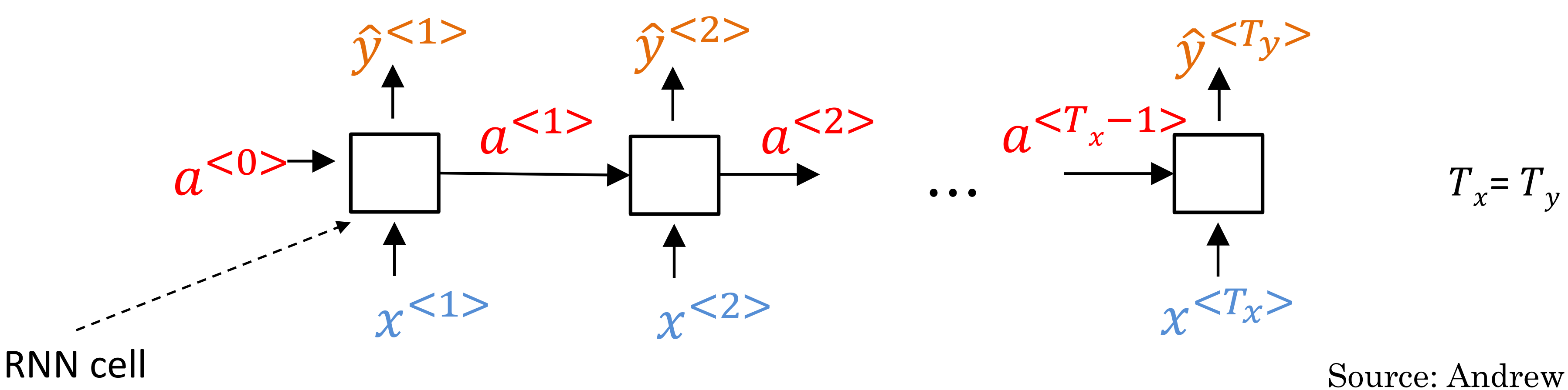
Neural networks for time series

- Many applications in networking context, e.g.:
  - Given the hourly traffic in a mobile cell for the last two days, predict traffic for next hour (regression)
  - Given a sequence of measured SNR values, predict if a failure is occurring, and what is the cause (classification)

- Other applications
  - NLP (text/speech recognition, automatic translation…)
  - Sentiment analysis (e.g., predict if a phrase/sentence has positive or negative sense)
  - Image captioning (the sequence <u>is in the output</u>)

Neural networks for time series
Recurrent neural networks (RNN)

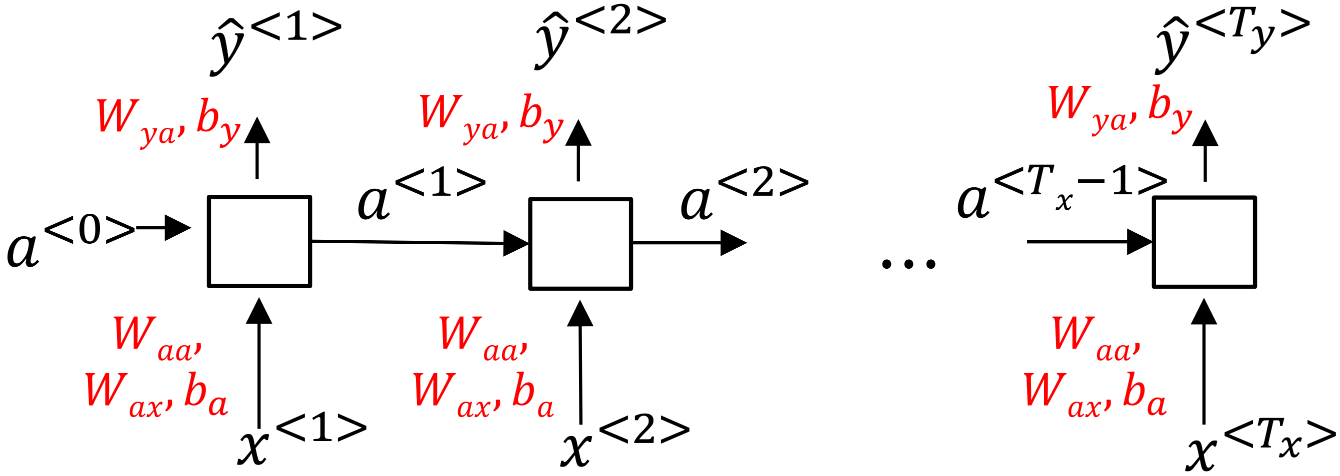
- Notation
  - $x^{<1>}, x^{<2>}, \ldots, x^{<T_x>}$ **input sequence** $(x^{<t>}, t = 1, \ldots, T_x)$
    - T_x is the number of *time-steps* in the input sequence
  - $\hat{y}^{<1>}, \hat{y}^{<2>}, \ldots, \hat{y}^{<T_y>}$ **output sequence** $(\hat{y}^{<t>}, t = 1, \ldots, Ty)$
    - T_y is the number of *time-steps* in the output sequence
    - $\hat{y}$ is the **predicted** value; the **ground truth is** $y^{<t>}, t = 1, \ldots, Ty$
    - in general $T_x \neq T_y$
  - $a^{<t>}$, activation at time-step *t*



Source: Andrew Ng

Neural networks for time series

Recurrent neural networks (RNN)



- Parameter sharing: the same weights are used by the RNN cell in all the time steps
  - $a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$
    - $a^{<0>} = inizialization\ vector\ (e.g., zeroes - vector)$
  - $\hat{y}^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$

Typycal choices:
g_1 : *tanh, sigmoid*
g_2 : *sigmoid, softmax*

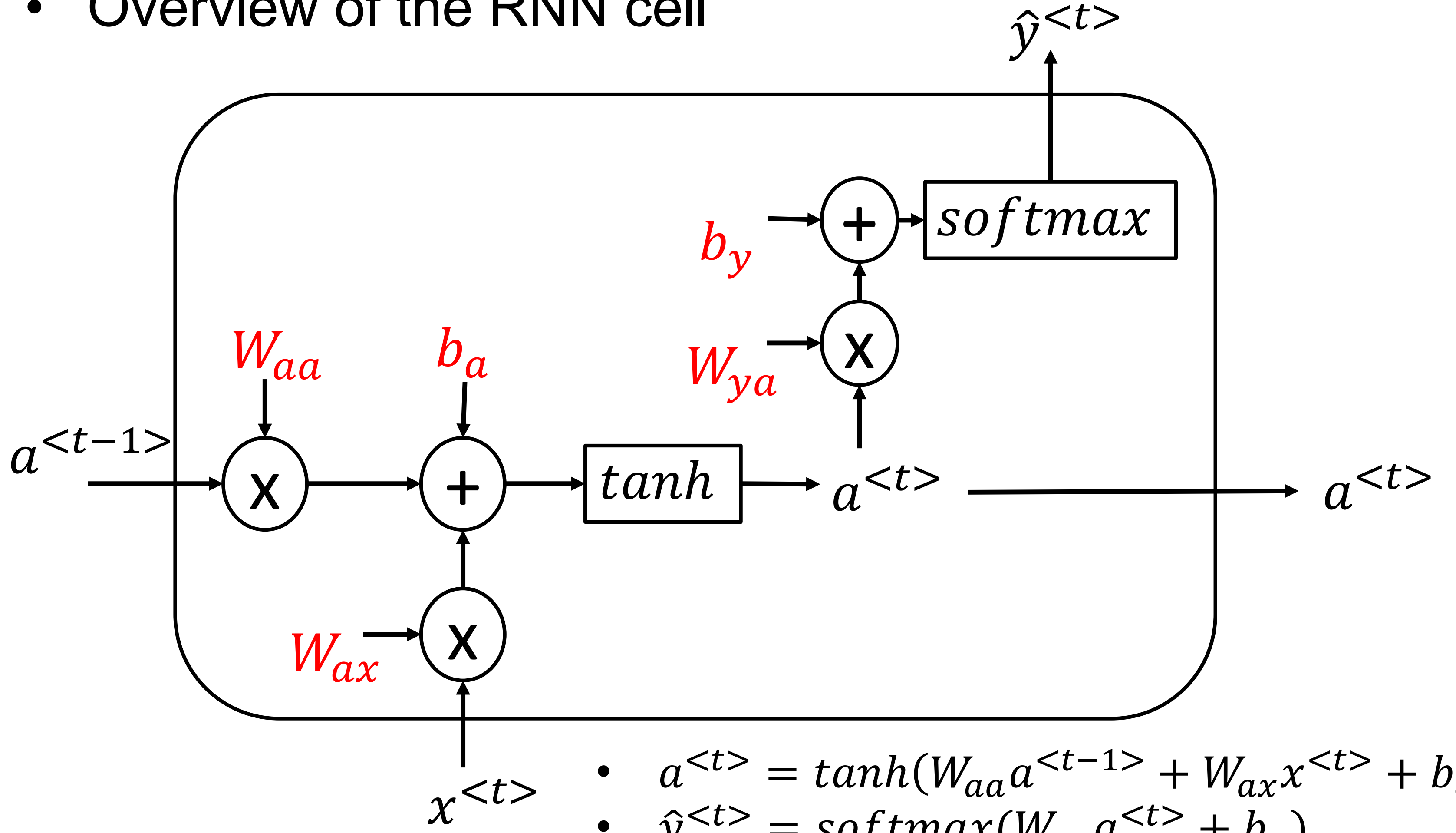
Source: Andrew Ng

Neural networks for time series

Recurrent neural networks (RNN)

$$softmax(y_i) = \frac{e^{y_i}}{\sum_i e^{y_i}}$$

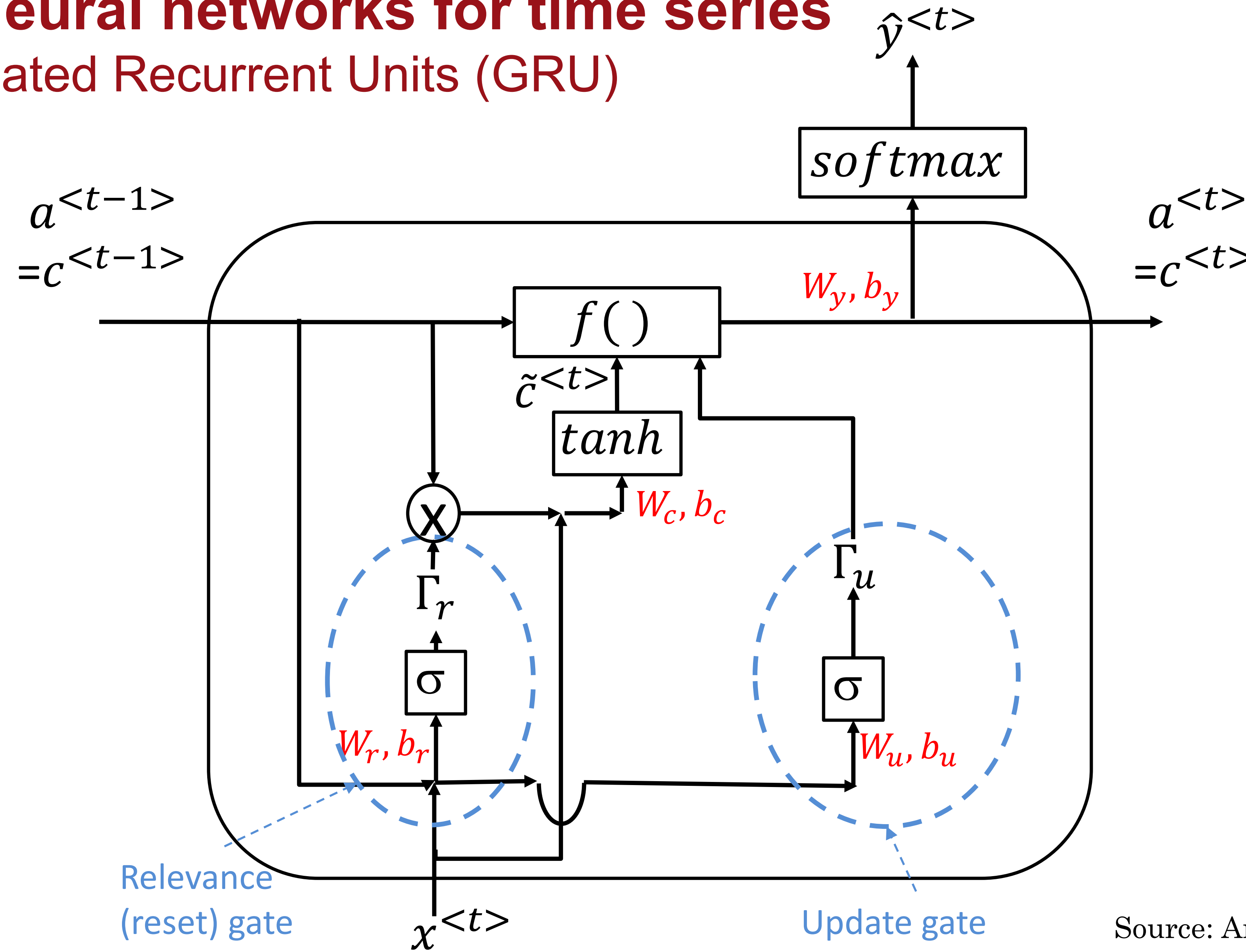
- Overview of the RNN cell



- $a^{<t>} = tanh(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$
- $\hat{y}^{<t>} = softmax(W_{ya}a^{<t>} + b_y)$

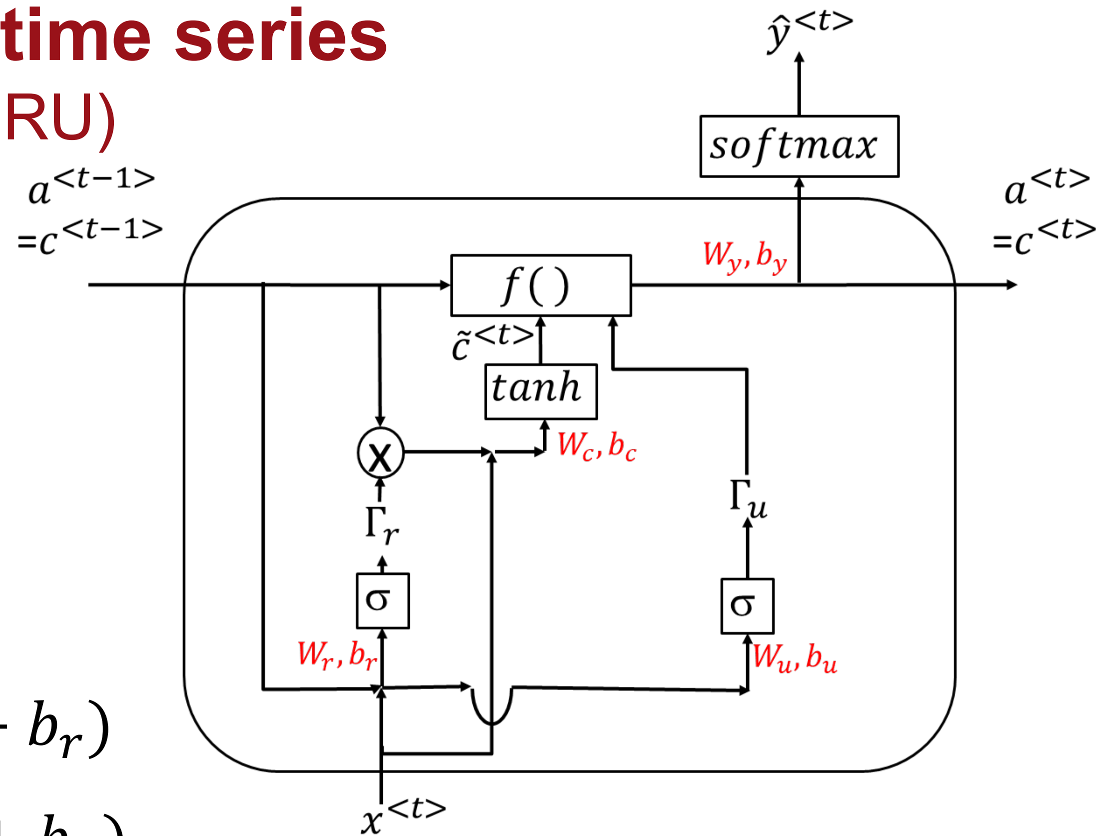
Source: Andrew Ng

Neural networks for time series
Gated Recurrent Units (GRU)



Source: Andrew Ng

Neural networks for time series
Gated Recurrent Units (GRU)



$$\Gamma_r = \sigma(W_r[\, c^{<t-1>}, x^{<t>}] + b_r)$$

$$\Gamma_u = \sigma(W_u[\, c^{<t-1>}, x^{<t>}] + b_u)$$

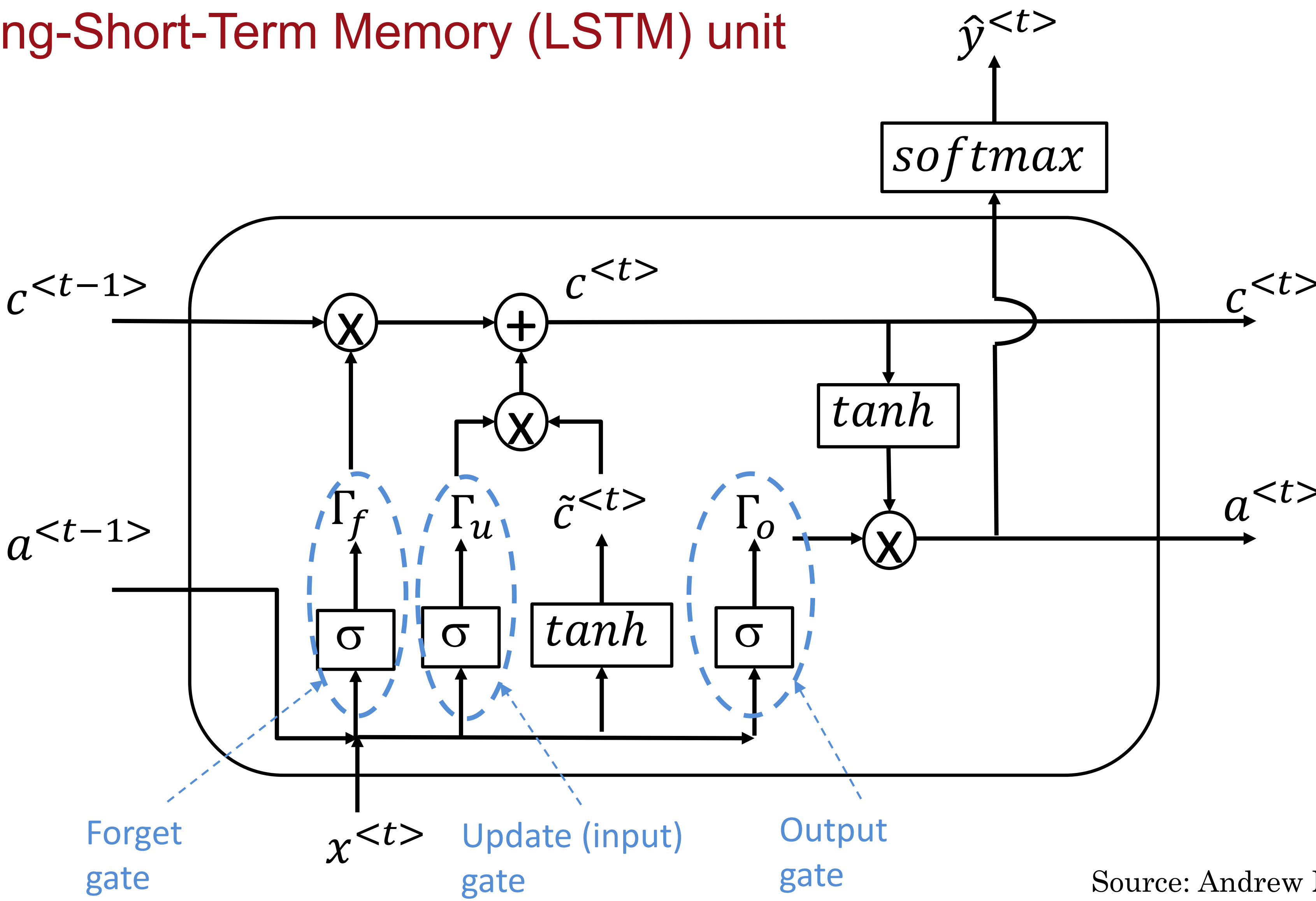
$$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r c^{<t-1>}, x^{<t>}] + b_c)$$

$$c^{<t>} = \Gamma_u\, \tilde{c}^{<t>} + (1 - \Gamma_u) c^{<t-1>} \qquad <-- f(\,)$$

$$y^{<t>} = softmax(W_y\, c^{<t>} + b_y)$$

Source: Andrew Ng

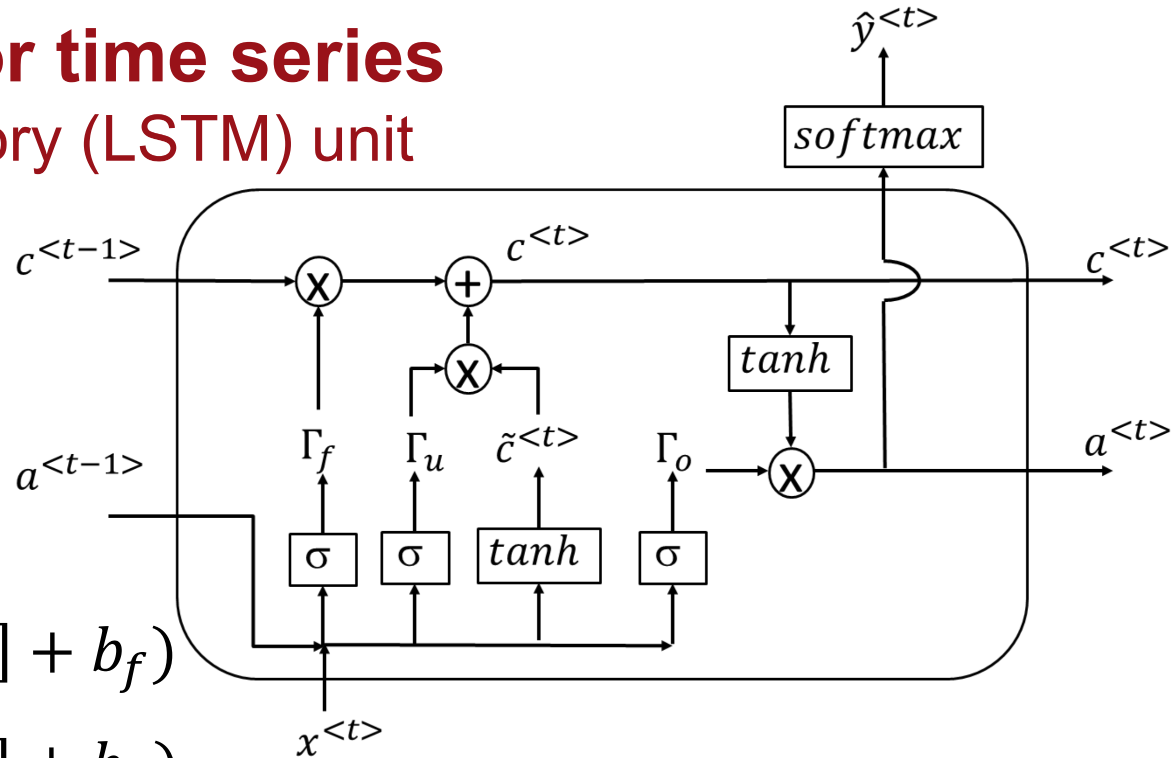
Neural networks for time series
Long-Short-Term Memory (LSTM) unit



Source: Andrew Ng

Neural networks for time series

Long-Short-Term Memory (LSTM) unit



$$\Gamma_f = \sigma(W_f[\,a^{<t-1>}, x^{<t>}] + b_f)$$

$$\Gamma_o = \sigma(W_o[\,a^{<t-1>}, x^{<t>}] + b_o)$$

$$\Gamma_u = \sigma(W_u[\,a^{<t-1>}, x^{<t>}] + b_u)$$

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$

$$a^{<t>} = \Gamma_o * \tanh(c^{<t>})$$

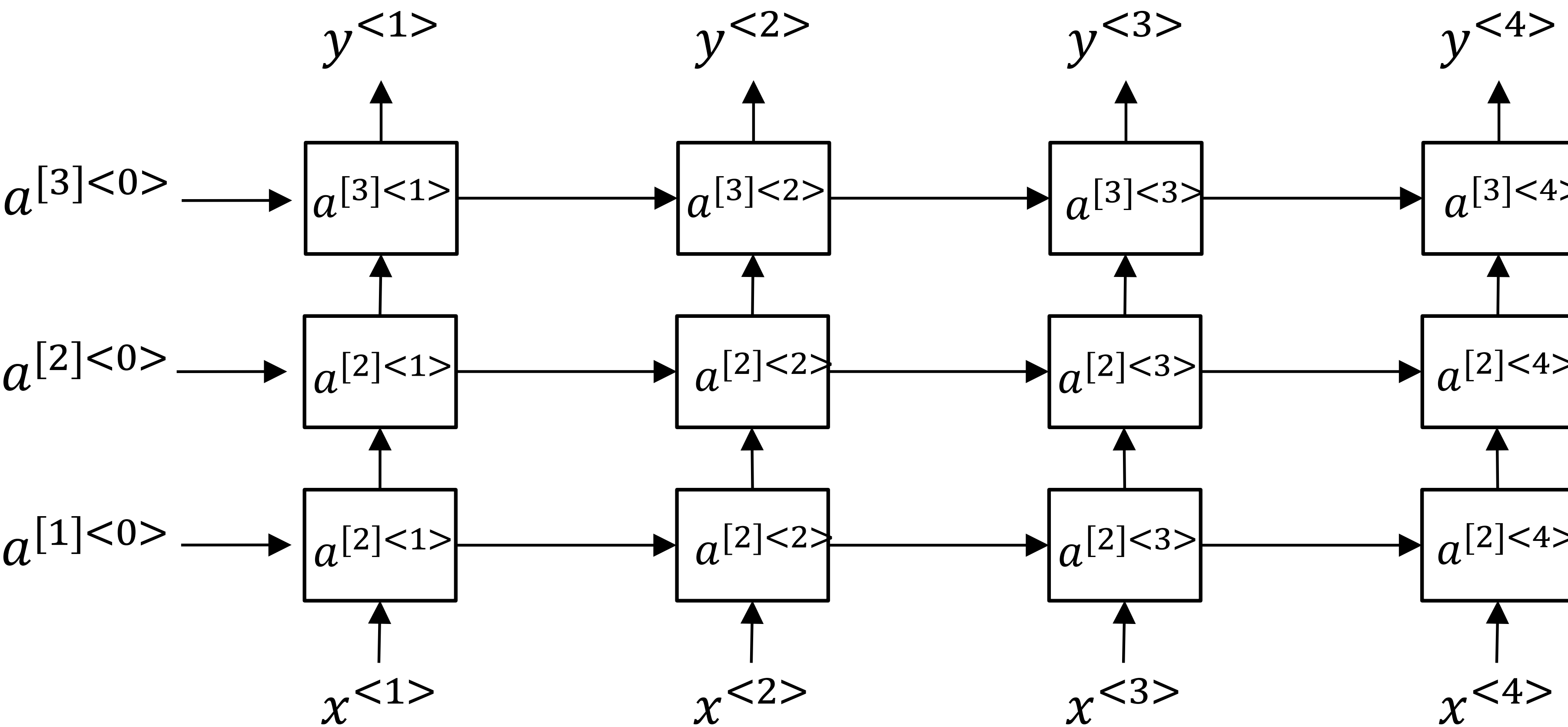
$$y^{<t>} = softmax(W_y\, a^{<t>} + b_y)$$

Source: Andrew Ng

Neural networks for time series

Deep RNNs

- As in DNNs, more hidden layers can be used also in RNNs



Source: Andrew Ng