Introduction to Artificial Neural Networks

Classification and Prediction

Lê Hồng Phương

<phuonglh@hus.edu.vn>
Data Science Laboratory
Vietnam National University, Hanoi

May 10, 2020

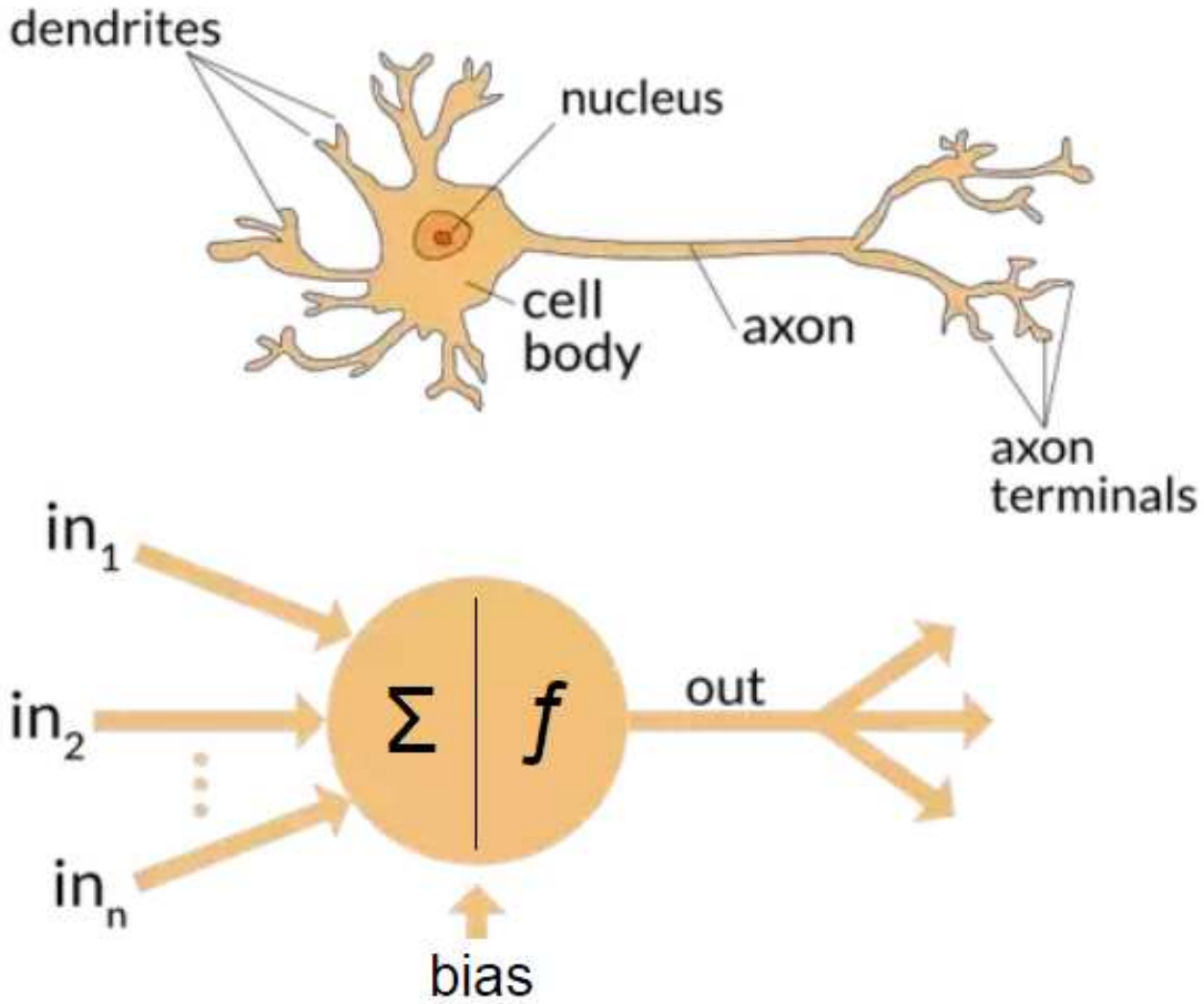
Content

Content

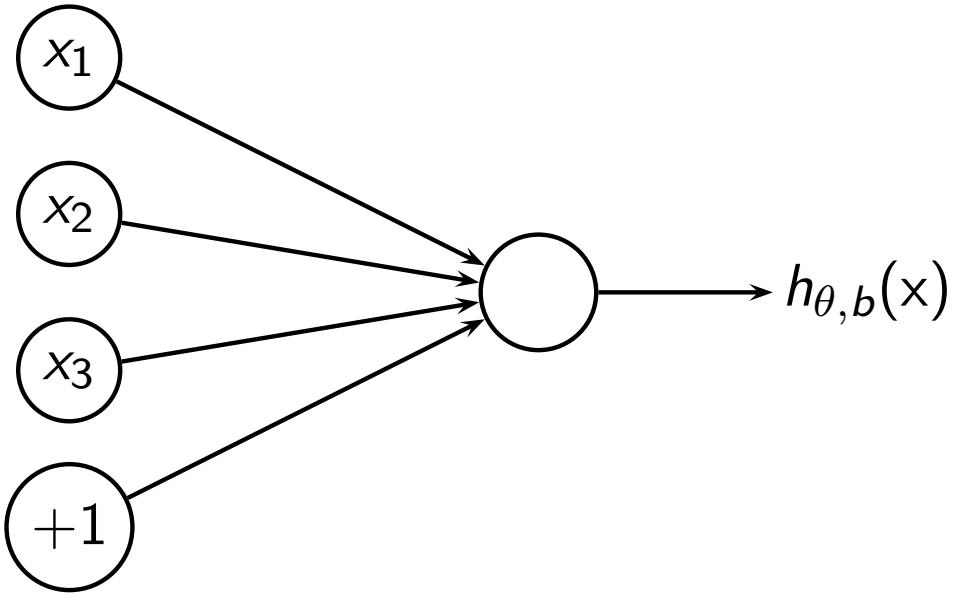
Artificial Neural Network

- Artificial Neural Network (ANN) is a mathematical model that simulates natural neural networks in which many neurons are connected via synapses.
- Many successful applications in classification and prediction problems:
  - speech recognition
  - image analysis
  - adaptive control
  - natural language processing

Artificial Neural Network



Feed-forward ANN



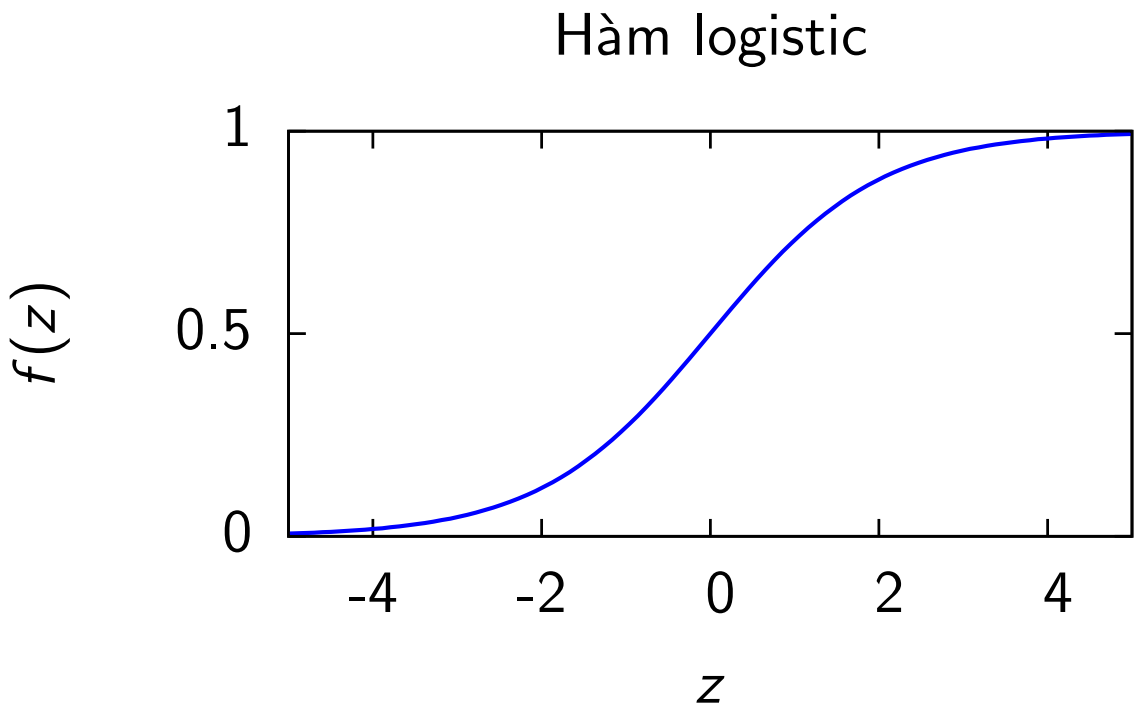
This neural network has one computation unit with three inputs x_1, x_2, x_3 and a bias term $+1$; the output is

$$h_{\theta,b}(x) = f\left(\sum_{i=1}^{3} \theta_i x_i + b\right),$$

where $f : \mathbb{R} \to \mathbb{R}$ is an *activation function*.

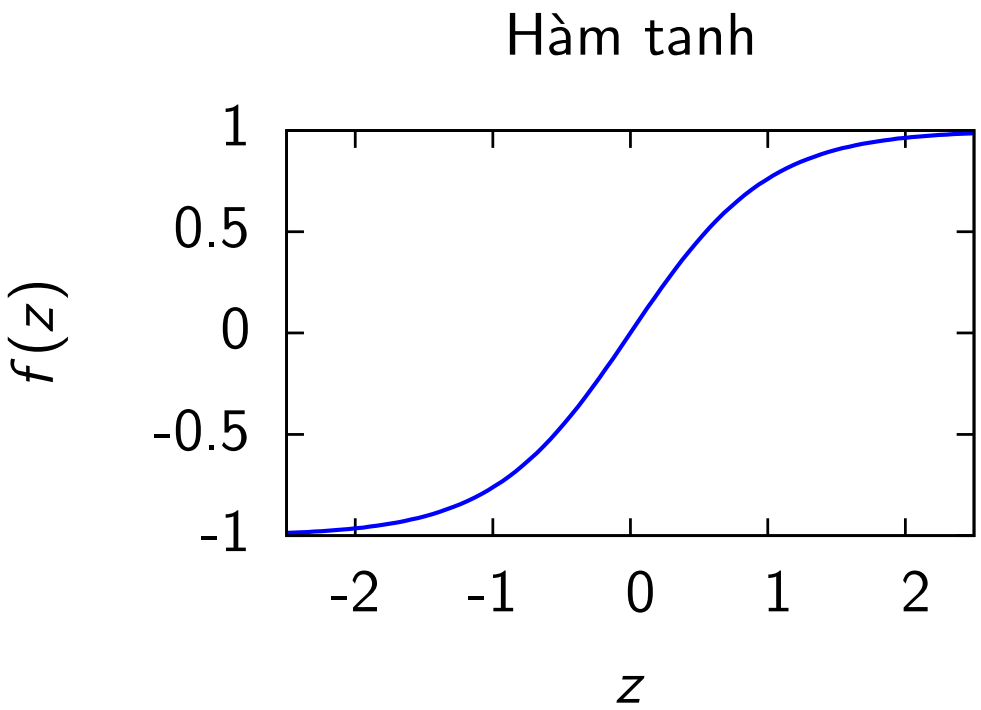
Activation Function – Sigmoid (Logistic)

$$f(z) = \frac{1}{1 + \exp(-z)}$$

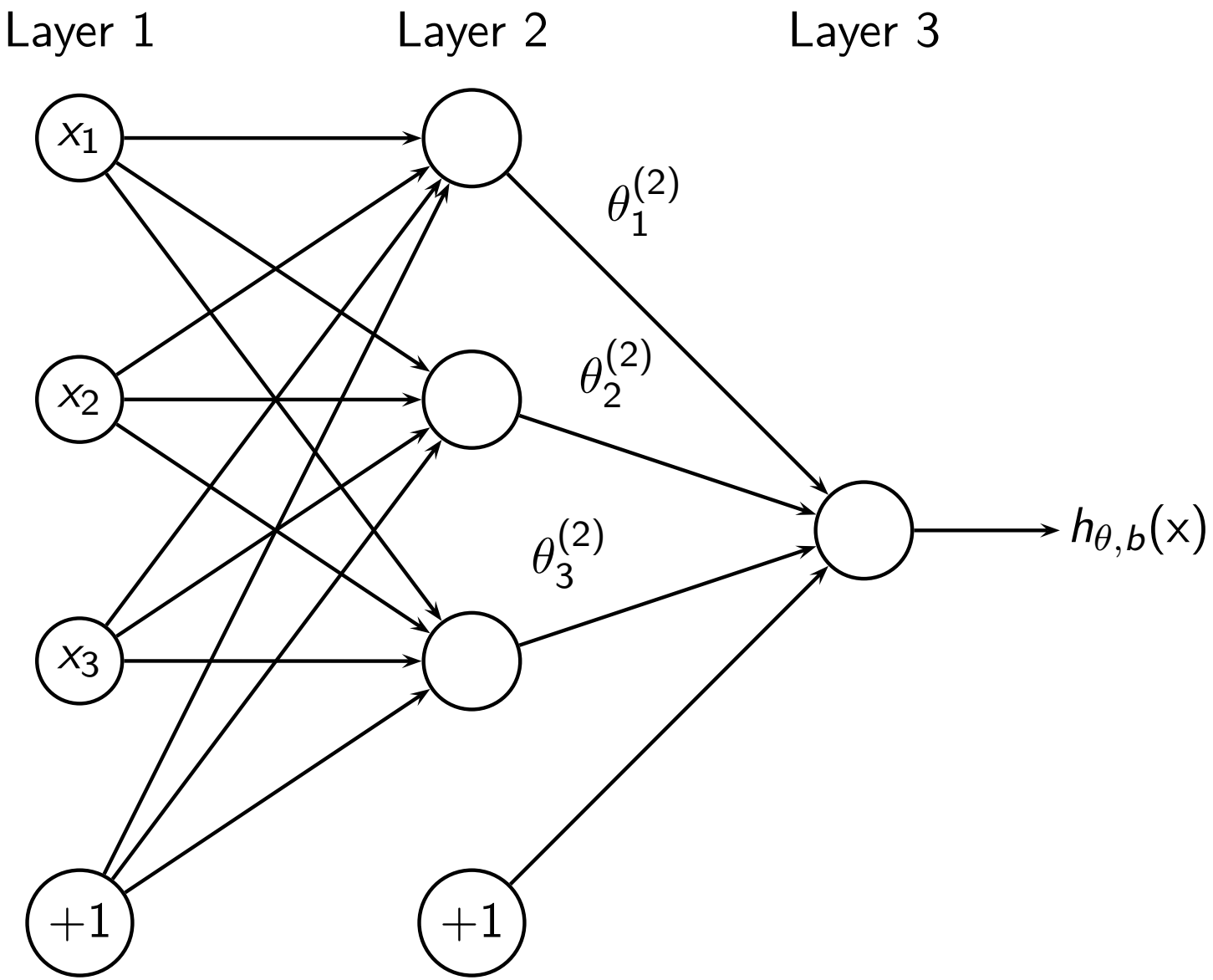


Activation Function– Tangent Hyperbolic

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$



A Three-layer Network

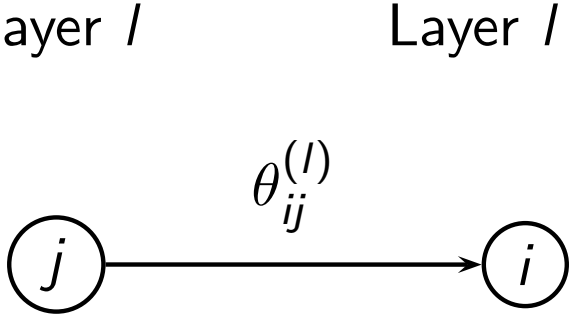


A Three-layer Network

- Each unit is represented by a circle. Input units are also in circles.
- Circles with label $+1$ represent *bias terms* corresponding to intercepts. The left layer is called *input layer*, the right layer is called *output layer*. In this example, the output layer has only one unit.
- The middle layer is called *hidden layer* since its values are invisible.

n-layer ANN

- Denote n the number of layers of the the network).
- Denote L_l the l-th layer of the network; L_1 is the input layer and L_n is the output layer.
- The network above has the following parameters: $(\theta, b) = (\theta^{(1)}, b^{(1)}, \theta^{(2)}, b^{(2)})$ where $\theta_{ij}^{(l)}$ represents the parameter on the connection arc from unit j of l layer to unit i of $l+1$ layer.



- $b_i^{(l)}$ is the bias of unit i in layer l.

n-layer ANN

In the example above, we have

$$\theta^{(1)} = \begin{pmatrix} \theta_{11}^{(1)} & \theta_{12}^{(1)} & \theta_{13}^{(1)} \\ \theta_{21}^{(1)} & \theta_{22}^{(1)} & \theta_{23}^{(1)} \\ \theta_{31}^{(1)} & \theta_{32}^{(1)} & \theta_{33}^{(1)} \end{pmatrix} \qquad \theta^{(2)} = \begin{pmatrix} \theta_{11}^{(2)} & \theta_{12}^{(2)} & \theta_{13}^{(2)} \end{pmatrix}$$

$$b^{(1)} = \begin{pmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{pmatrix} \qquad b^{(2)} = \begin{pmatrix} b_1^{(2)} \end{pmatrix}.$$

n-layer ANN

- We call $a_i^{(l)}$ *the activation* (that means the output value) of unit i of layer l.
- With $l = 1$, we have $a_i^{(1)} = x_i$.
- The network is a function which computes an output value as follows:

$$
\begin{aligned}
a_i^{(1)} &= x_i, \qquad \forall i = 1, 2, 3; \\
a_1^{(2)} &= f\left(\theta_{11}^{(1)} a_1^{(1)} + \theta_{12}^{(1)} a_2^{(1)} + \theta_{13}^{(1)} a_3^{(1)} + b_1^{(1)}\right) \\
a_2^{(2)} &= f\left(\theta_{21}^{(1)} a_1^{(1)} + \theta_{22}^{(1)} a_2^{(1)} + \theta_{23}^{(1)} a_3^{(1)} + b_2^{(1)}\right) \\
a_3^{(2)} &= f\left(\theta_{31}^{(1)} a_1^{(1)} + \theta_{32}^{(1)} a_2^{(1)} + \theta_{33}^{(1)} a_3^{(1)} + b_3^{(1)}\right) \\
a_1^{(3)} &= f\left(\theta_{11}^{(2)} a_1^{(2)} + \theta_{12}^{(2)} a_2^{(2)} + \theta_{13}^{(2)} a_3^{(2)} + b_1^{(2)}\right).
\end{aligned}
$$

n-layer ANN

- Denote $z_i^{(l+1)} = \sum_{j=1}^{3} \theta_{ij}^{(l)} a_j^{(l)} + b_i^{(l)}$, then $a_i^{(l)} = f(z_i^{(l)})$.
- If we expand the function f for vector parameters as follows:

$$f((z_1, z_2, z_3)) = (f(z_1), f(z_2), f(z_3))$$

then we can use the matrix forms:

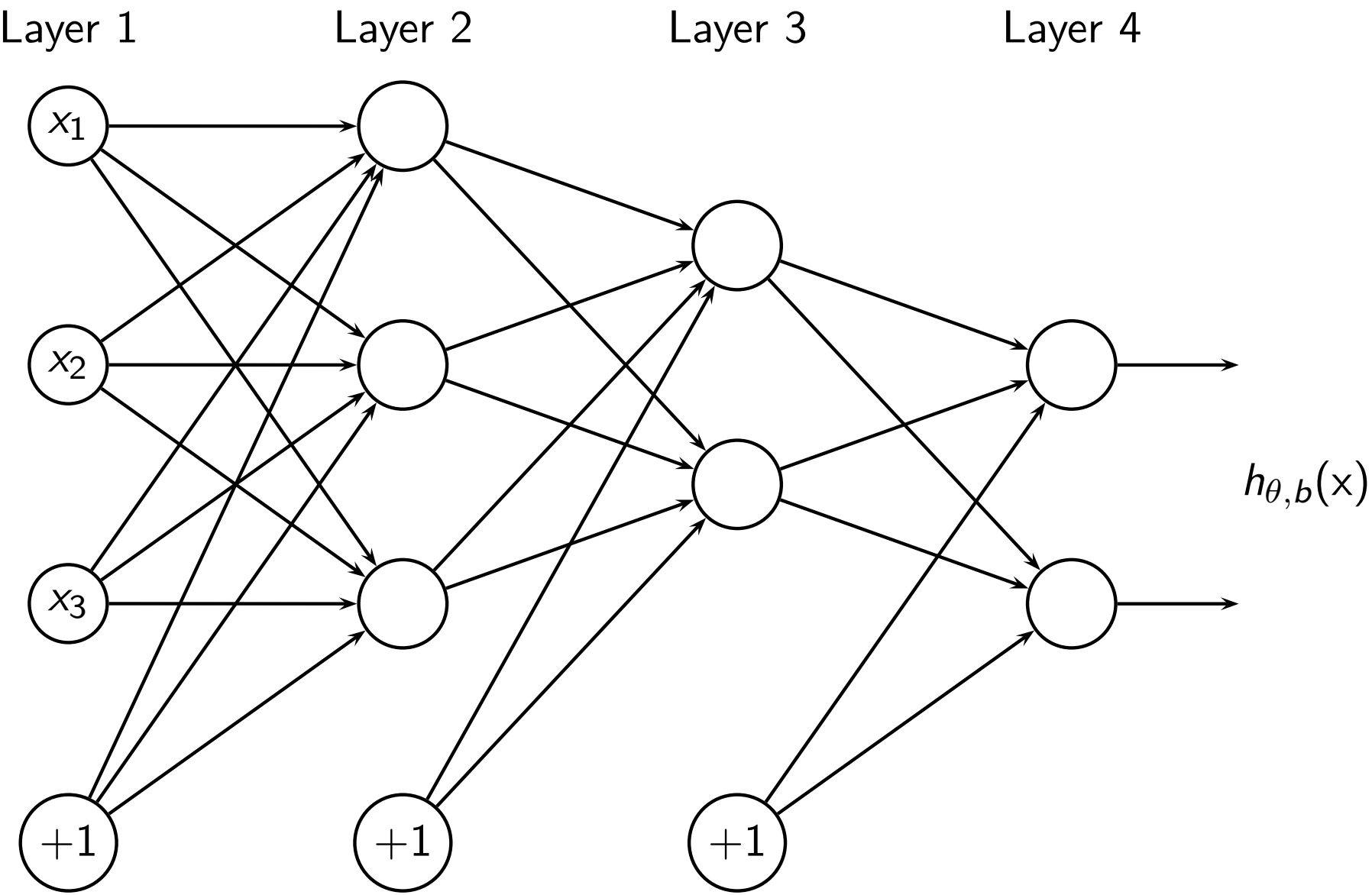
$$\begin{aligned} z^{(2)} &= \theta^{(1)} a^{(1)} + b^{(1)} \\ a^{(2)} &= f\left(z^{(2)}\right) \\ z^{(3)} &= \theta^{(2)} a^{(2)} + b^{(2)} \\ h_{\theta,b}(x) &= a^{(3)} = f\left(z^{(3)}\right). \end{aligned}$$

n-layer ANN

- In the general ANN with n layers, we can use the forward computation scheme to compute the output value of the network.
  - We first compute the activations in the second layer, then use them to compute the activations in the third layer and so on. The final output is: $h_{\theta,b}(\mathrm{x}) = f\left(z^{(n)}\right)$.
- The activations at layer $l+1$ are computed from the activations of the layer l as follows:

$$z^{(l+1)} = \theta^{(l)} a^{(l)} + b^{(l)}$$
$$a^{(l+1)} = f(z^{(l+1)}).$$

n-layer ANN



n-layer ANN

- Note that an ANN may have many output units.
- In order to train this network, we need a training data $\{(x_i, y_i)\}$ where $y_i \in \mathbb{R}^2$.
- This model is used when we need to predict a multivariate value. For example, in a medical analysis, x_i represents properties/features of a patient, y_i represents the symptoms/analysis results of that patient.
  - $y_i = (0, 1)$ represents the fact that patient does not have the first disease and has the second disease.

Content

Loss Function

- Suppose that we have a training data set with N data points:

$$\{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}.$$

- We can train an ANN by using gradient descent algorithms. loạt.
- For each data point (x, y), we have a loss function $J(x, y; \theta, b)$.

Loss Function

The total loss on the entire training data set:

$$J(\theta, b) = \frac{1}{N} \sum_{i=1}^{N} J(\mathrm{x}_i, y_i; \theta, b) + \underbrace{\frac{\lambda}{2N} \sum_{l=1}^{n-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(\theta_{ji}^{(l)} \right)^2}_{\text{regularization}}$$

where s_l is the number of units of layer l.

Loss Function

Two common loss functions:

1. Squared error:

$$J(\mathrm{x}, y; \theta, b) = \frac{1}{2}\|y - h_{\theta,b}(\mathrm{x})\|^2.$$

2. Cross-entropy:

$$J(\mathrm{x}, y; \theta, b) = -\left[y \log(h_{\theta,b}(\mathrm{x})) + (1-y) \log(1 - h_{\theta,b}(\mathrm{x}))\right],$$

where $y \in \{0, 1\}$.

Gradient Descent

- We need to find parameters θ, b which minimize the loss function

$$J(\theta, b) \to \min .$$

- The simplest method is gradient descent.
- Since $J(\theta, b)$ is not a convex function, the optimal value are not guaranteed to be the globally optimal solution.
- However, the gradient descent algorithm can usually find good parameters if their initial values are properly chosen.

Gradient Descent

In each iteration, the GD algorithm updates the parameters θ, b as follows:

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

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(\theta, b),$$

where α is the learning rate.

Gradient Descent

We have

$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta, b) = \frac{1}{N} \left[\sum_{i=1}^{N} \frac{\partial}{\partial \theta_{ij}^{(l)}} J(x_i, y_i; \theta, b) + \lambda \theta_{ij}^{(l)} \right]$$

$$\frac{\partial}{\partial b_{i}^{(l)}} J(\theta, b) = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial}{\partial b_{i}^{(l)}} J(x_i, y_i; \theta, b).$$

We need to compute the partial derivatives

$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(x_i, y_i; \theta, b), \quad \frac{\partial}{\partial b_{i}^{(l)}} J(x_i, y_i; \theta, b)$$

The backpropagation algorithm is an effective method to compute these derivatives.

Content

Backpropagation Algorithm

- Backpropagation is the key algorithm that makes training neural network models computationally tractable.
- Fundamentally, it's a technique for calculating derivatives quickly.
- The algorithm has been reinvented many times in different fields; its general name is **reverse-mode differentiation**.[1]

[1] *Calculus on Computational Graphs: Backpropagation*

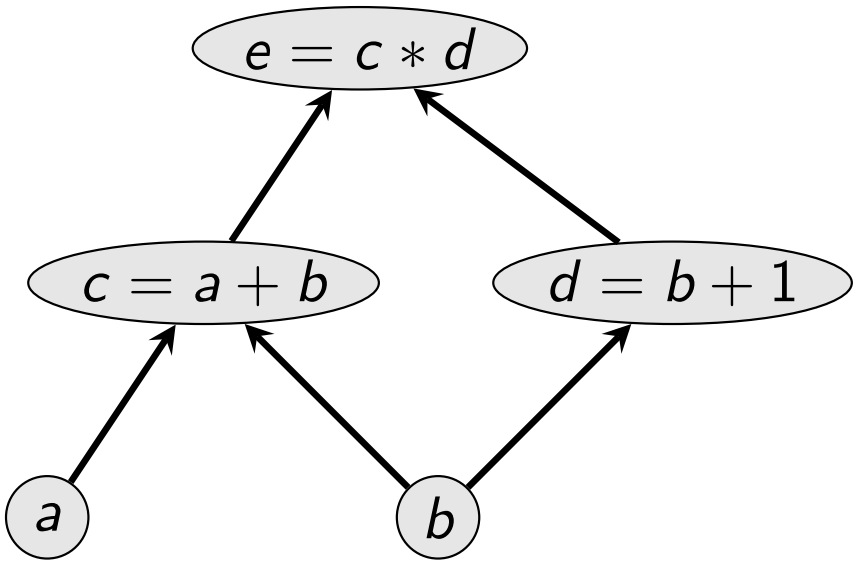
Computational Graphs

Consider a mathematical expression, for example

$$e = (a + b) * (b + 1)$$

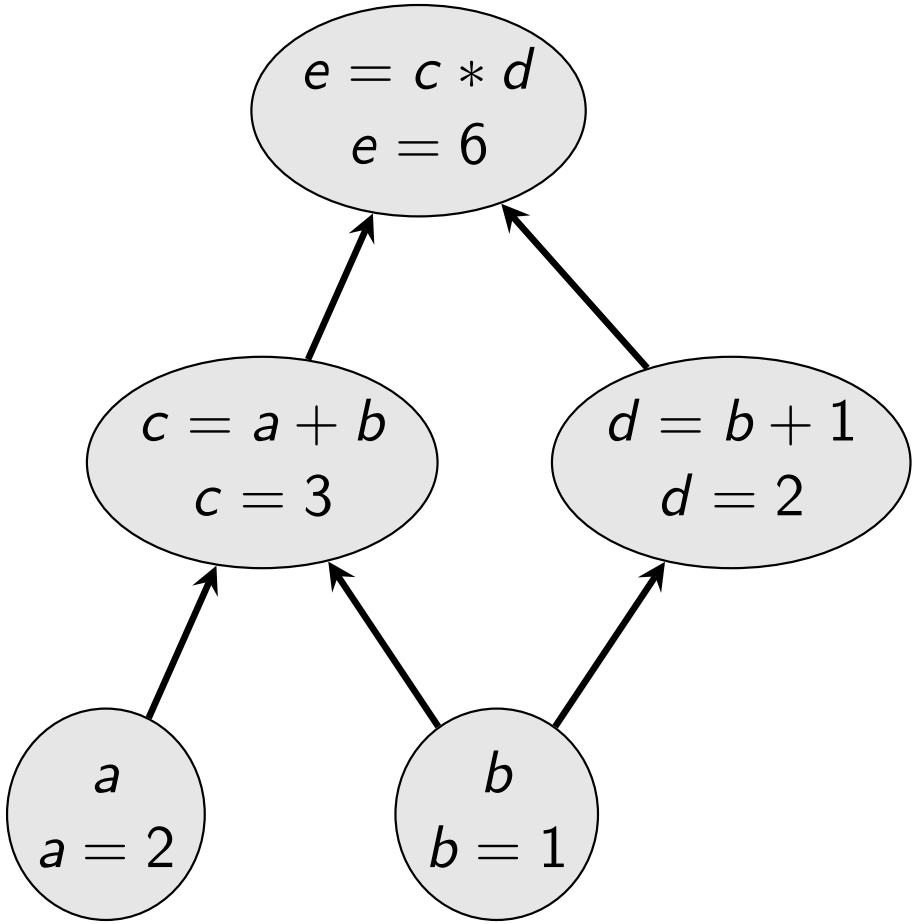
Let's introduce two intermediate variables c and d so that we have the following computational graph:

$$c = a + b$$
$$d = b + 1$$
$$e = c * d$$



Computational Graphs

We can evaluate the expression by setting the input variables to certain values and computing nodes up through the graph. Let's set $a = 2$ and $b = 1$.



Derivatives on Computational Graphs

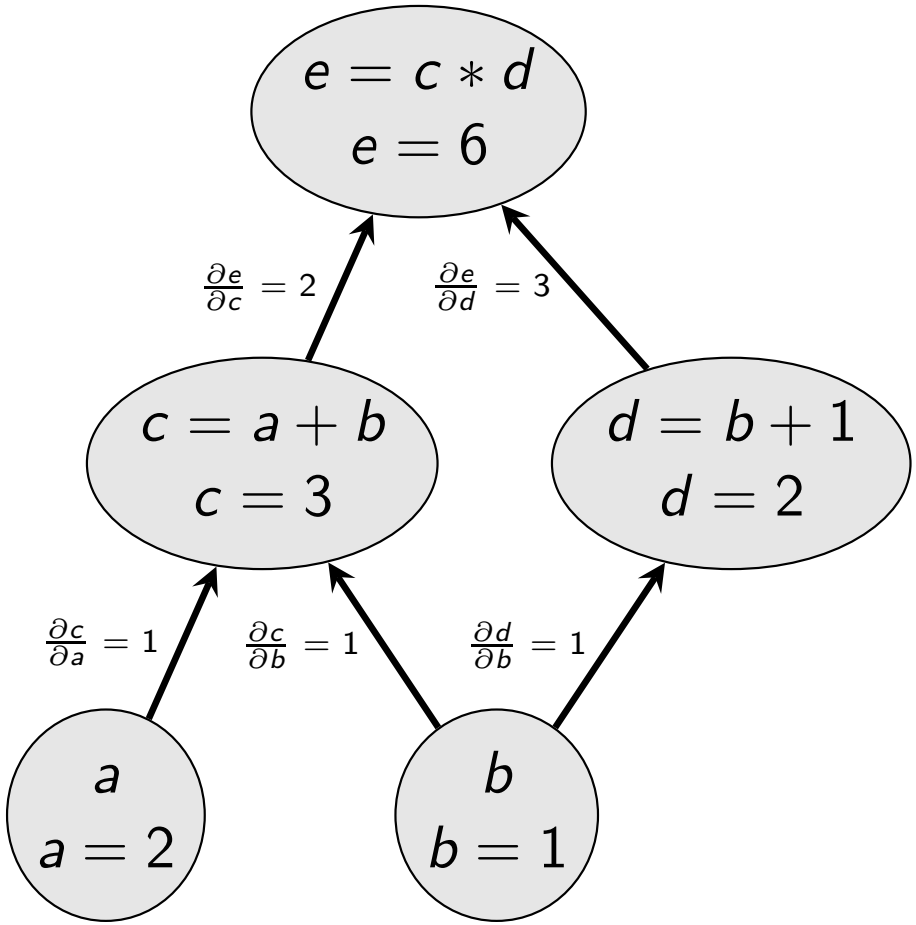
- If a directly affects c, then how does it affects c? If a changes a little bit, how does c changes?
- That is, we simply need to compute the partial derivative of c w.r.t a.
- The sum rule and product rule of Calculus 1:

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$

$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$

Derivatives on Computational Graphs

Our computational graph with derivative on each edge labeled:



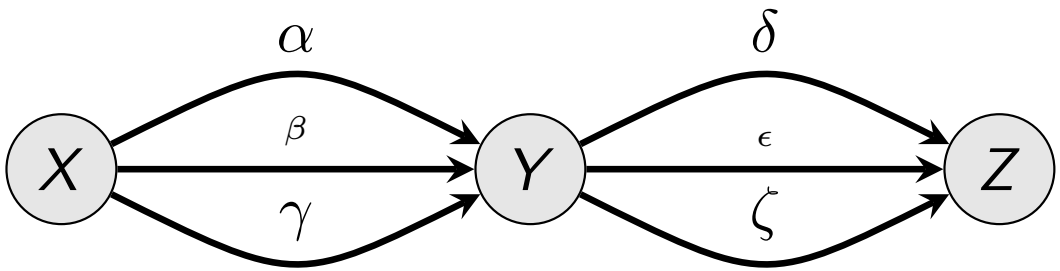
Derivatives on Computational Graphs

- How nodes that aren't directly connected affect each other?. For example, how e is affected by a or b?
- If we change a at speed of 1, c also changes at a speed of 1. In turn, c changes at the speed of 1 causes e to change at speed of 2. So e changes at a rate of $2 * 1$ w.r.t a.
- If we change b at speed of 1, this will causes e to change via two paths: through c or through d. The general rule is to sum over all possible paths:

$$\begin{aligned}\frac{\partial e}{\partial b} &= \frac{\partial e}{\partial c} * \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} * \frac{\partial d}{\partial b} \\ &= 2 * 1 + 3 * 1 \\ &= 5\end{aligned}$$

Factoring Paths

If we compute the derivatives by summing over all paths as above, it is very easy to get *a combinatorial explosion* in the number of possible paths.



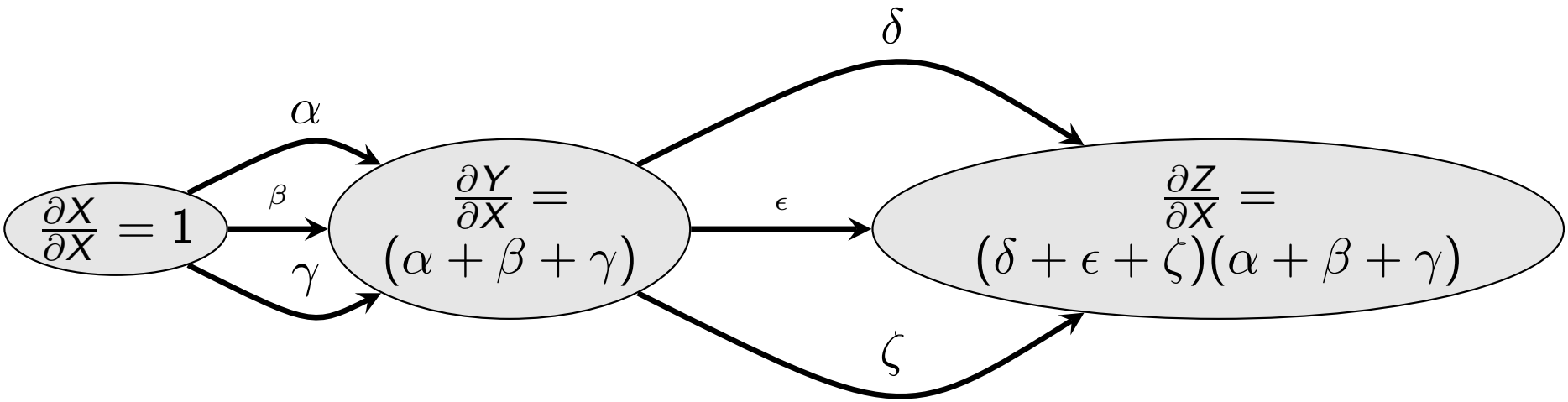
If we want to compute $\partial Z/\partial X$, we need to sum over 9 paths:

$$\frac{\partial Z}{\partial X} = \delta * \alpha + \delta * \beta + \delta * \gamma + \epsilon * \alpha + \epsilon * \beta + \epsilon * \gamma + \zeta * \alpha + \zeta * \beta + \zeta * \gamma.$$

It would be much better to factor them:

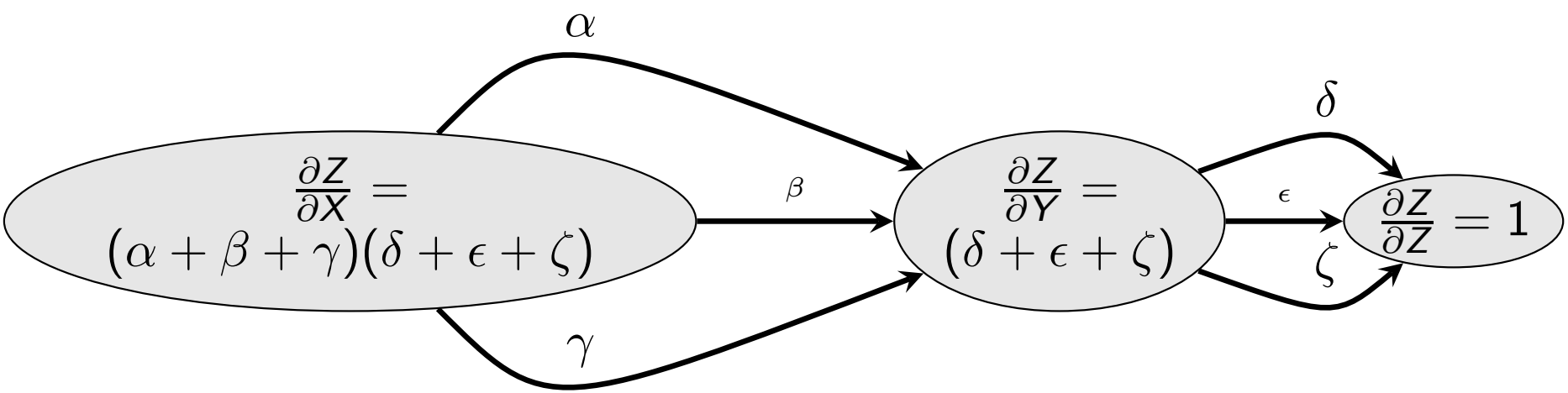
$$\frac{\partial Z}{\partial X} = (\delta + \epsilon + \zeta)(\alpha + \beta + \gamma).$$

Forward-Mode Differentiation ($\partial \odot / \partial X$)



- Starts at an input to the graph and moves towards the end.
- At every node, it sums all the paths feeding in. By adding them up, we get the total way in which the node is affected by the input.
- Forward-mode differentiation tracks *how one input affects every nodes.*

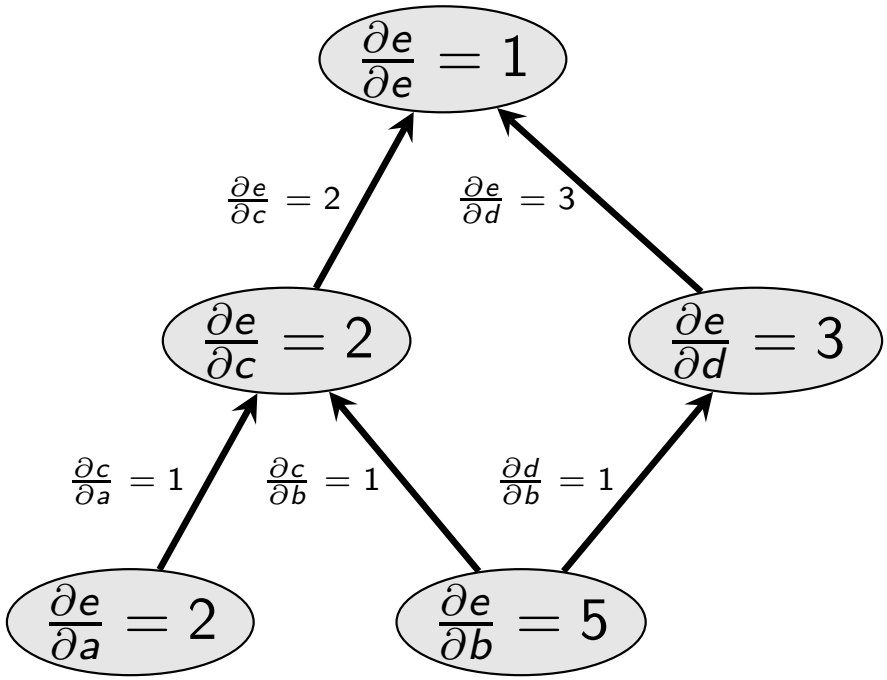
Reverse-Mode Differentiation ($\partial Z/\partial \odot$)



- Starts with an output of the graph and moves towards the beginning.
- At each node, it merges all paths which originated at that node.
- Reverse-mode differentiation tracks *how one output is affected by every nodes.*

Reverse-Mode Differentiation: Example

If we do the reverse-mode differentiation from e down, this will give us the derivative of e w.r.t. **every** node:



We get both $\partial e/\partial a$ and $\partial e/\partial b$, the derivatives of e w.r.t to both inputs.

Reverse-Mode Differentiation: Efficiency

- If we have 10^6 variables $\theta_j, \forall j = 1, 2, \ldots, 10^6$, and a loss function ℓ which depends on these variables, then performing a reverse-mode differentiation one time and we can get all

$$\frac{\partial \ell(\theta)}{\theta_j}, \forall j = 1, 2, \ldots, 10^6.$$

- If we do forward-mode differentiation, we need to scan 10^6 times through the computational graphs, each for one θ_j.
- That's why backpropagation algorithm is widely used in neural networks (and many other models).

Backpropagation Algorithm

- All the weights $\theta := \{\theta_{ij}^{(l)}\}$ (and bias parameters b) determine output $h(\mathrm{x})$.
- Error on an example (x, y) is

$$e(h(\mathrm{x}), y) = J(\theta, b) := J(w)$$

- We need to compute the gradient

$$\nabla J(w) = \frac{\partial J}{\partial w_{ij}^{(l)}}, \forall i, j, l.$$

Backpropagation Algorithm

- Product rule:

$$\frac{\partial J(w)}{\partial w_{ij}^{(l)}} = \frac{\partial J(w)}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}}$$

- We have

$$\frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}} = a_j^{(l-1)}$$

- We only need

$$\frac{\partial J(w)}{\partial z_i^{(l)}} = \epsilon_i^{(l)}$$

Backpropagation Algorithm

- First, for each data point (x, y), we compute forward to find all activations, including output value $h(\mathrm{x})$.
- For each unit i of layer l, we compute the error $\varepsilon_i^{(l)}$, which measures the contribution of unit i to the total error at the output.

Backpropagation Algorithm

- For the output layer $l = n$, we can compute directly $\varepsilon_i^{(n)}$ for all units i of the output layer by comparing their activations with the real output values.
- Using the squared loss and sigmoid activation function, for all $i = 1, 2, \ldots, s_n$:

$$\begin{aligned}\varepsilon_i^{(n)} &= \frac{\partial}{\partial z_i^{(n)}} \frac{1}{2} \|y - f(z_i^{(n)})\|^2 \\ &= -(y_i - f(z_i^{(n)})) f'(z_i^{(n)}) \\ &= -(y_i - a_i^{(n)}) a_i^{(n)} (1 - a_i^{(n)}).\end{aligned}$$

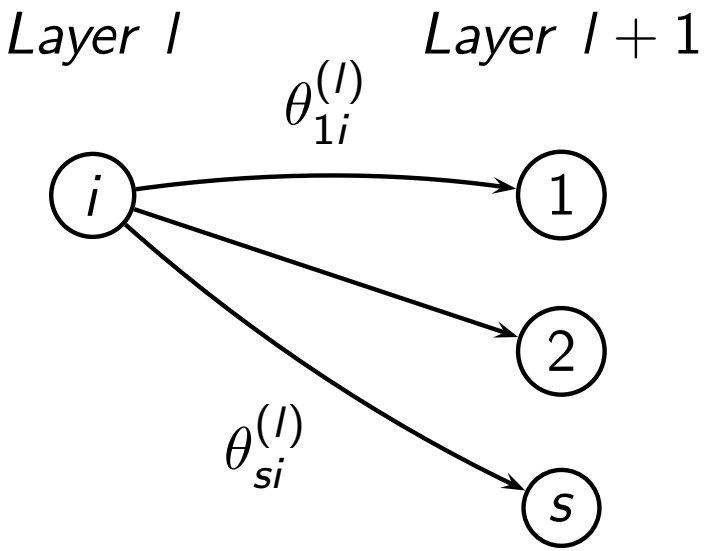
Backpropagation Algorithm

$$
\begin{aligned}
\epsilon_i^{(l)} &= \frac{\partial J(w)}{\partial z_i^{(l)}} \\
&= \sum_{j=1}^{s_{l+1}} \frac{\partial J(w)}{\partial z_j^{(l+1)}} \times \frac{\partial z_j^{(l+1)}}{\partial a_i^{(l)}} \times \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} \\
&= \sum_{j=1}^{s_{l+1}} \epsilon_j^{(l+1)} w_{ji}^{(l+1)} f'(z_i^{(l)})
\end{aligned}
$$

Backpropagation Algorithm

- For each hidden unit i, $\varepsilon_i^{(l)}$ is computed as the weighted average of the errors at units of $l+1$ layer which use its as input.

$$\varepsilon_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} \theta_{ji}^{(l)} \varepsilon_j^{(l+1)} \right) f'(z_i^{(l)}).$$



Backpropagation Algorithm

1. Forward computation: compute all activations of the layer $L_2, L_3 \ldots, L_n$.
2. For all units i of the output layer L_n, compute
$$\varepsilon_i^{(n)} = -(y_i - a_i^{(n)}) a_i^{(n)} (1 - a_i^{(n)}).$$
3. Compute the errors in the backward direction: for all $l = n-1, \ldots, 2$ and all units i of layer l, compute
$$\varepsilon_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} \theta_{ji}^{(l)} \varepsilon_j^{(l+1)} \right) f'(z_i^{(l)}).$$
4. The partial derivatives are computed as follows:
$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(x, y; \theta, b) = \varepsilon_i^{(l+1)} a_j^{(l)}$$
$$\frac{\partial}{\partial b_i^{(l)}} J(x, y; \theta, b) = \varepsilon_i^{(l+1)}.$$

Backpropagation Algorithm

- We can represent the above algorithm more succinctly by using matrix operations.
- Denote $\bullet$ the operator that performs elementwise multiplication:

$$\mathrm{x} = (x_1, \ldots, x_D), \mathrm{y} = (y_1, \ldots, y_D) \Rightarrow \mathrm{x} \bullet \mathrm{y} = (x_1 y_1, x_2 y_2, \ldots, x_D y_D).$$

- Similarly, we expand functions $f(\cdot), f'(\cdot)$ for elementwise operations, for example:

$$f(\mathrm{x}) = (f(x_1), f(x_2), \ldots, f(x_D))$$
$$f'(\mathrm{x}) = \left(\frac{\partial}{\partial x_1} f(x_1), \frac{\partial}{\partial x_2} f(x_2), \ldots, \frac{\partial}{\partial x_D} f(x_D) \right).$$

Backpropagation Algorithm

1. Forward computation for all layers $L_2, L_3 \ldots, L_n$:
$$z^{(l+1)} = \theta^{(l)} a^{(l)} + b^{(l)}$$
$$a^{(l+1)} = f(z^{(l)}).$$
2. At the output layer L_n, compute
$$\varepsilon^{(n)} = -(y - a^{(n)}) \bullet f'(z^{(n)}).$$
3. Backward computation: for all $l = n-1, n-2, \ldots, 2$, compute
$$\varepsilon^{(l)} = \left((\theta^{(l)})^T \varepsilon^{(l+1)}\right) \bullet f'(z^{(l)}).$$
4. The derivatives are computed as follows:
$$\frac{\partial}{\partial \theta^{(l)}} J(\mathrm{x}, y; \theta, b) = \varepsilon^{(l+1)} \left(a^{(l)}\right)^T$$
$$\frac{\partial}{\partial b^{(l)}} J(\mathrm{x}, y; \theta, b) = \varepsilon^{(l+1)}.$$

Backpropagation Algorithm

- Above is the gradient descent algorithm for only one data point (x, y).
- The gradient descent algorithm for all training data points is as follows.
- Denote by $\nabla\theta^{(l)}$ the gradient matrix $\theta^{(l)}$ (has the same dimension as $\theta^{(l)}$) and by $\nabla b^{(l)}$ the gradient of $b^{(l)}$ (has the same dimension as $b^{(l)}$).

Gradient Descent

Algorithm 1: The Gradient Descent algorithm training ANN

for $l = 1$ *to* n **do**

$\nabla\theta^{(l)} \leftarrow 0; \quad \nabla b^{(l)} \leftarrow 0;$

for $i = 1$ *to* N **do**

Compute $\frac{\partial}{\partial\theta^{(l)}} J(x_i, y_i; \theta, b)$ and $\frac{\partial}{\partial b^{(l)}} J(x_i, y_i; \theta, b)$;

$\nabla\theta^{(l)} \leftarrow \nabla\theta^{(l)} + \frac{\partial}{\partial\theta^{(l)}} J(x_i, y_i; \theta, b)$;

$\nabla b^{(l)} \leftarrow \nabla b^{(l)} + \frac{\partial}{\partial b^{(l)}} J(x_i, y_i; \theta, b)$;

$\theta^{(l)} \leftarrow \theta^{(l)} - \alpha \left(\frac{1}{N}\nabla\theta^{(l)} + \frac{\lambda}{N}\theta^{(l)}\right)$;

$b^{(l)} \leftarrow b^{(l)} - \alpha \left(\frac{1}{N}\nabla b^{(l)}\right)$;

Other Algorithms

Similar to other models, in addition to the GD algorithms, we can use other optimization algorithm to estimate the parameters $\theta^{(l)}$ and $b^{(l)}$

- Quasi-Newton methods such as BFGS, L-BFGS or Conjugate Gradient (GC)
- All these methods require computation of $J(\theta, b)$ and first order derivatives $\frac{\partial}{\partial \theta^{(l)}} J(\theta, b)$ and $\frac{\partial}{\partial b^{(l)}} J(\theta, b)$.

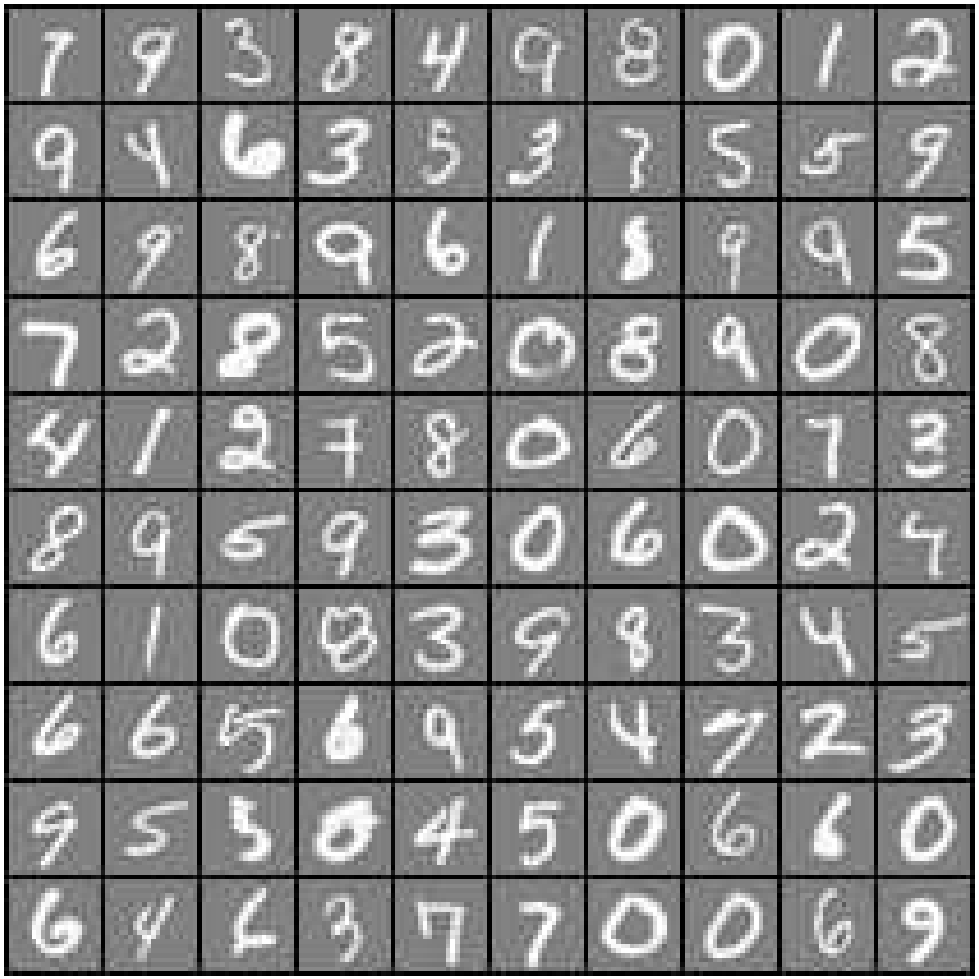
Content

Content

Handwritten Digit Recognition

- Application of MLR on a dataset of handwritten digits. The set contains 5000 training examples.
- This is a subset of the MNIST handwritten digit dataset:
  - *http://yann.lecun.com/exdb/mnist/*
- Each training example is a 20x20 pixels grayscale image of the digit.
- Each pixel is represented by a floating point number indicating the grayscale intensity at that location.
- The 20x20 grid of pixel is unrolled into a 400-dimensional vector.

Handwritten Digit Recognition



Handwritten Digit Recognition

Model	**Preprocessing**	**Error (%)**
Linear (1-layer ANN)	no	12.0
Linear (1-layer ANN)	yes	8.4
Two-layer ANN, 300 hidden units	no	4.7
Two-layer ANN, 300 hidden units	yes	**1.6**
Two-layer ANN, 1000 hidden units	no	4.5
Six-layer ANN, 784-2500-2000-1500-1000-500-10	no	**0.35**

Content

Speech Recognition

- Before 2012, almost speech recognition systems:
  - use Hidden Markov Models (HMMs) to model the sequence voice signals with respect to time;
  - use Gaussian mixture models (GMMs) to model the relationship of HMM states with input voice signals.
- The big improvements in this problem were achieved nearly 40 years ago when using the expectation maximization (EM) algorithm was used to train HMM-GMMs.

Speech Recognition

- After 2012, Deep Neural Networks (DNNs) with many hidden layers outperformed the old models.
- Significant results are obtained by big research groups at University of Toronto, Microsoft Research, Google Research, IBM Research:

Speech Recognition

Comparison of error rates on large datasets between GMM-DNN and và DNN-HMM:

Task	**Training (h)**	**DNN**	**GMM**
Switchboard (Test set 1)	309	18.5	27.4
Switchboard (Test set 2)	309	16.1	23.6
English Broadcast News	50	17.5	18.8
Bing Voice Search	24	30.4	36.2
Google Voice Input	5,870	12.3	
Youtube	1,400	47.6	52.3

Content

Recurrent Neural Network

- RNN is an extension of FFNN to process sequential data.
- RNNs process input sequence of arbitrarily length via the recursive application of a transition function on *a hidden state vector*.
- More precisely, it takes as input a list of input vectors $x_1, x_2, \ldots, x_n$ together with an initial state vector s_0, and returns a list of state vectors $s_1, s_2, \ldots, s_n$ as well as a list of output vectors $y_0, y_1, \ldots, y_n$.

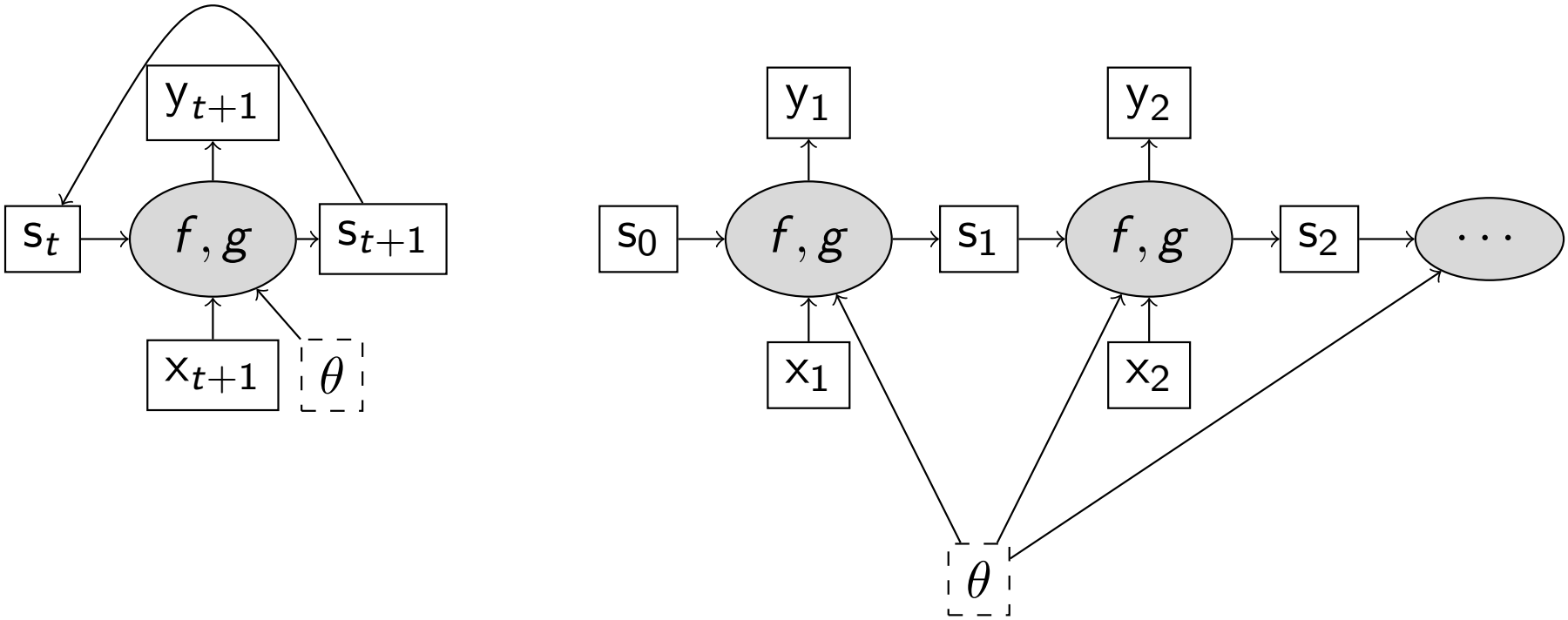
Recurrent Neural Network

- A recursively defined function $f(\cdot)$ that takes as input a state vector s_t and an input vector x_{t+1} and produces a new state vector s_{t+1}.
- An additional function $g(\cdot)$ is used to map a state vector s_t to an output vector y_t:

$$s_{t+1} = f(s_t, x_{t+1}),$$
$$y_t = g(s_t)$$

Recurrent Neural Network

The functions f, g are the same accross the sequence positions. The parameter vector θ is also shared across all time steps.



Recurrent Neural Network

- The common transition function f used in RNNs is an affine transformation followed by a point-wise non-linearity such as the hyperbolic tangent function:

$$\mathrm{s}_{t+1} = \tanh\left(W\,\mathrm{x}_{t+1} + U\,\mathrm{s}_t + \mathrm{b}\right).$$

- However, this transition function has a problem in that it may make the gradient vector grow or decay exponentially over long sequence during training.

Content

Long Short-Term Memory

- LSTM network is a special type of RNNs which was designed to overcome the exploding or vanishing gradients problem of simple RNNs.
- The main idea of LSTMs is that they maintain *a memory cell* that is able to preserve state over a long period of time.

Recurrent Neural Network

The LSTM unit at the t-th input consists of a collection of multi-dimensional vectors, including:

- an input gate i_t
- a forget gate $\boldsymbol{f}_t$
- an output gate o_t
- a memory cell c_t
- a hidden state s_t.

Recurrent Neural Network

The unit takes as input a d-dimensional input vector x_t, the previous hidden state s_{t-1}, the previous memory cell c_{t-1}, and calculates the new vectors using the following 6 equations:

$$
\begin{aligned}
i_t &= \sigma\left(W^i x_t + U^i s_{t-1} + b^i\right) \\
\boldsymbol{f}_t &= \sigma\left(W^f x_t + U^f s_{t-1} + b^f\right) \\
o_t &= \sigma\left(W^o x_t + U^o s_{t-1} + b^o\right) \\
u_t &= \tanh\left(W^u x_t + U^u s_{t-1} + b^u\right) \\
c_t &= i_t \odot u_t + \boldsymbol{f}_t \odot c_{t-1} \\
s_t &= o_t \odot \tanh(c_t),
\end{aligned}
$$

where σ denotes the logistic sigmoid function, the dot product $\odot$ denotes the element-wise multiplication of vectors, W and U are weight matrices and b are bias vectors.

Content

Gated Recurrent Unit

- Another popular recurrent unit type which is commonly used in RNNs is the Gated Recurrent Unit – GRU.
- The GRU is like LSTM with forget gate but has fewer parameters than LSTMs, as it lacks an output gate.
- GRUs have been shown to exhibit even better performance on certain smaller datasets.

Gated Recurrent Unit

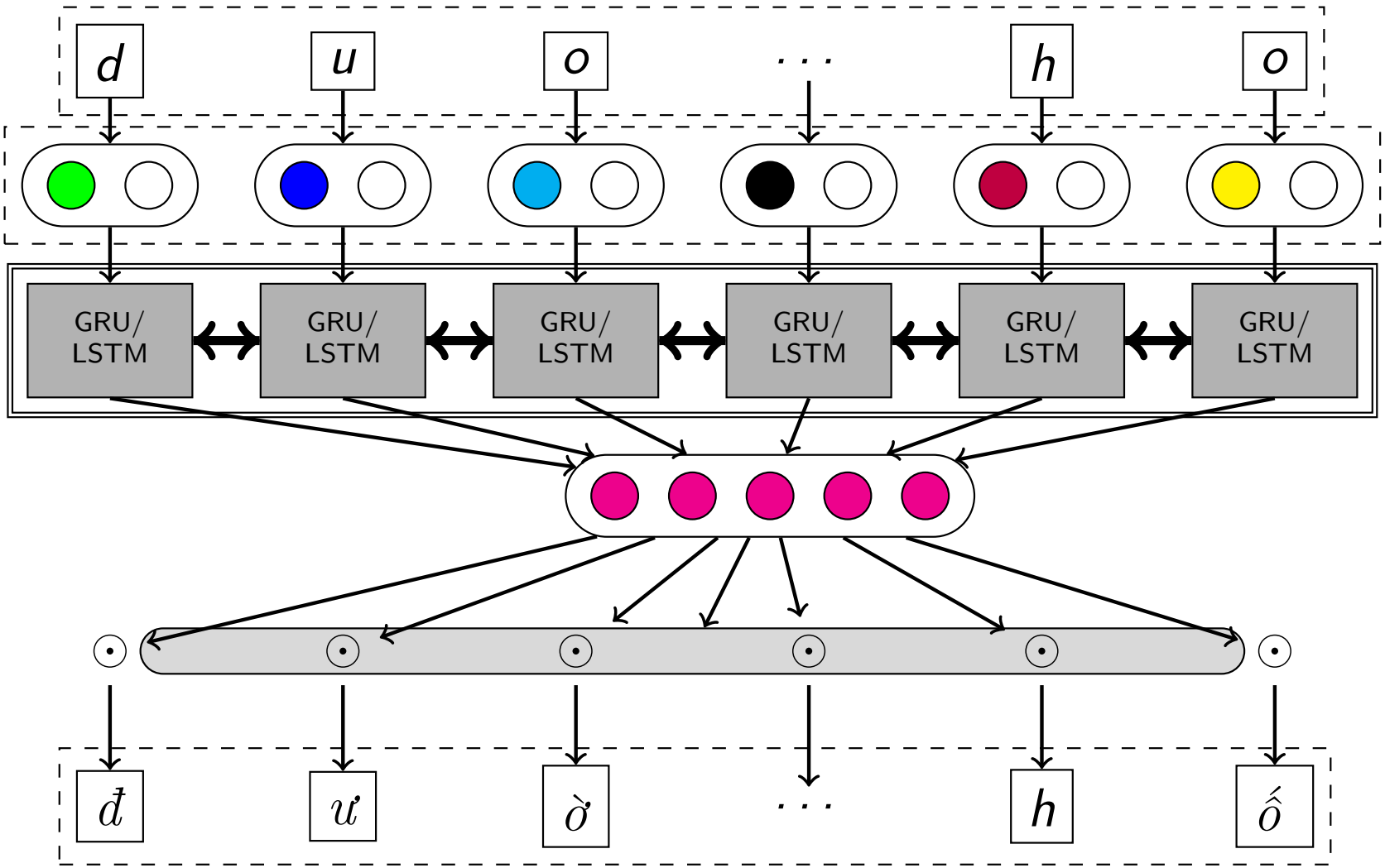
- A GRU has two gates, a reset gate r and un update gate z.
- Intuitively, the reset gate determines how to combine the new input with the previous memory, and the update gate defines how much of the previous memory to keep around.
- If we set the reset gate to all one and update gate to all zero, we get the plain RNN model.

Gated Recurrent Unit

The equations of the GRU unit are as follows:

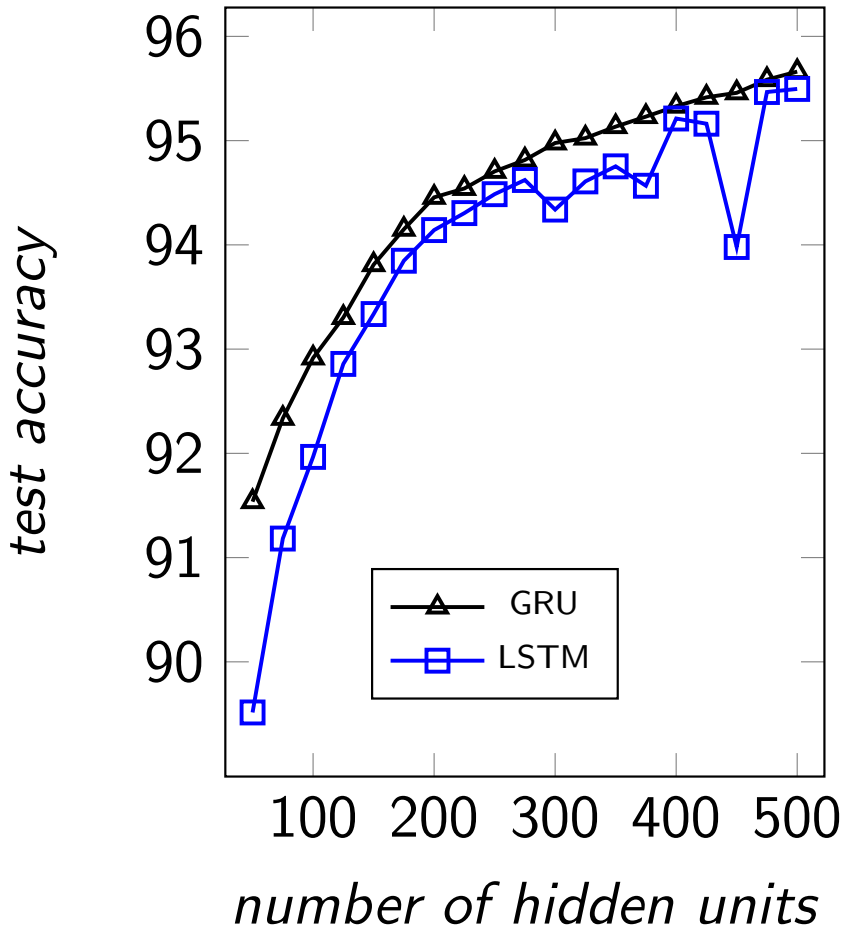
$$
\begin{aligned}
z_t &= \sigma\left(W^z x_t + U^z s_{t-1} + b^z\right) \\
r_t &= \sigma\left(W^r x_t + U^r s_{t-1} + b^r\right) \\
u_t &= \tanh\left(W^u x_t + U^u (s_{t-1} \odot r_t) + b^u\right) \\
s_t &= (1 - z_t) \odot u_t + z_t \odot s_{t-1}
\end{aligned}
$$

Example: Vietnamese Diacritic Generation

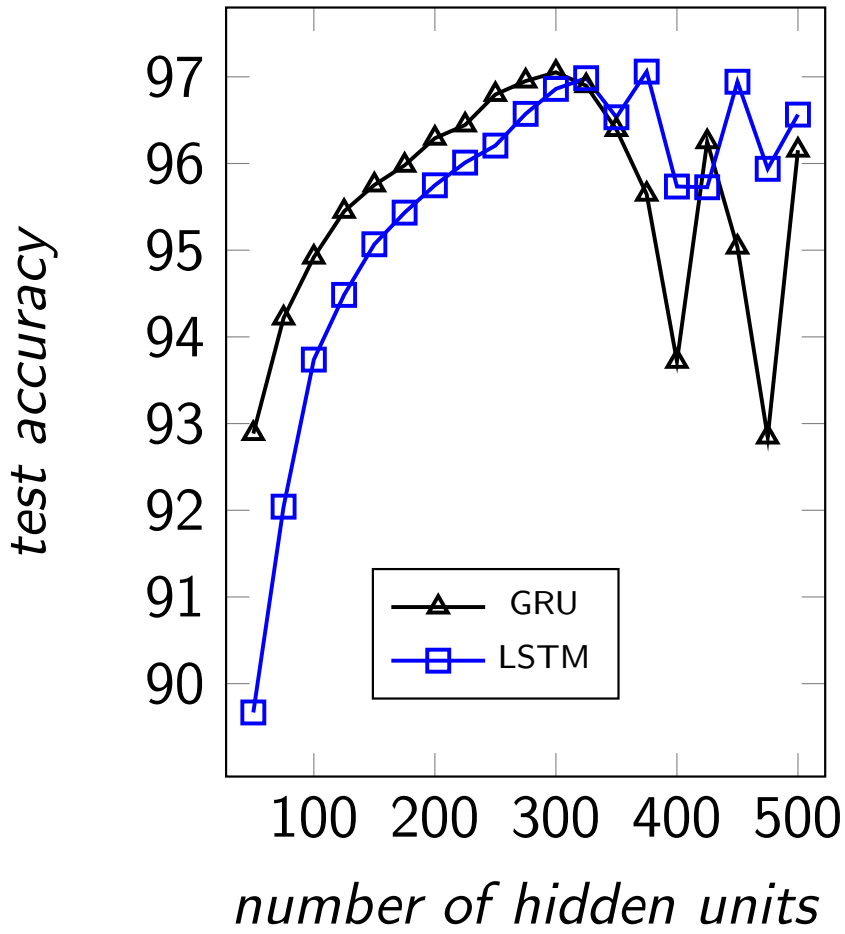


Example: Vietnamese Diacritic Generation

One recurrent layer



Two recurrent layers



Content

Summary

- ANNs and DNNs are nonlinear, effective models for many classification and prediction problems.
- Backpropagation algorithm for computing first-order derivatives of the objective functions with respect to parameters.
- RNNs are extension of FFNN to account for sequential data.
- See more about the backpropagation algorithm:

  https://colah.github.io/posts/2015-08-Backprop/
- See a nice explanation of LSTMs and RNNs:

  https://colah.github.io/posts/2015-08-Understanding-LSTMs/