Deep Learning 과정

Neural Networks

Gunhee Kim

Computer Science and Engineering

서 울 대 학 교

SEOUL NATIONAL UNIVERSITY

Outline

- Artificial Neurons
- Feedforward Neural Networks
- Backpropagation

(Credits: Stanford CS231n Lecture Notes)

Biological Analogy

The brain is composed of a mass of interconnected neurons

- Neurons transmit signals to each other

What is the neuron?

- The cell that performs information processing in the brain
- Fundamental functional unit of all nervous system tissue

Whether a signal is transmitted is an all-or-nothing event

- The electrical potential in the cell of the neuron is thresholded
- Whether a signal is sent, depends on the strength of the bond (synapse) between two neurons

Comparison of Brain and Computer

	Human	Computer
Processing Elements	100 Billion neurons	10 Million gates
Interconnects	1000 per neuron	A few
Cycles per sec	1000	500 Million
Bandwidth	10 Trillion bits/sec	1 Billion bits/sec
2X improvement	200,000 Years	2 Years

Neurons are slow!

- Neurons are a million times slower than gates

Parallelism

- The brain can fire all the neurons in a single step

Connectivity

- Connected via approximately 10^15 synapses

What is Neural Network?

a.k.a Artificial Neural Networks (ANN) or Multi-Layer Perceptrons (MLP)

An artificial neural network is a system composed of many simple processing elements operating in parallel that can acquire, store, and utilize experiential knowledge

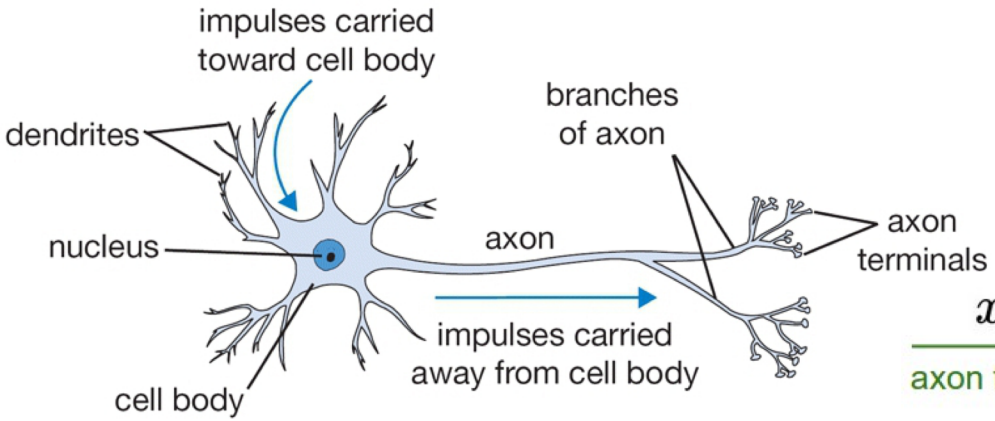
A biological neuron is not like an artificial neuron

- Perform complex nonlinear computations
- Complex non-linear dynamical system (timing is important)

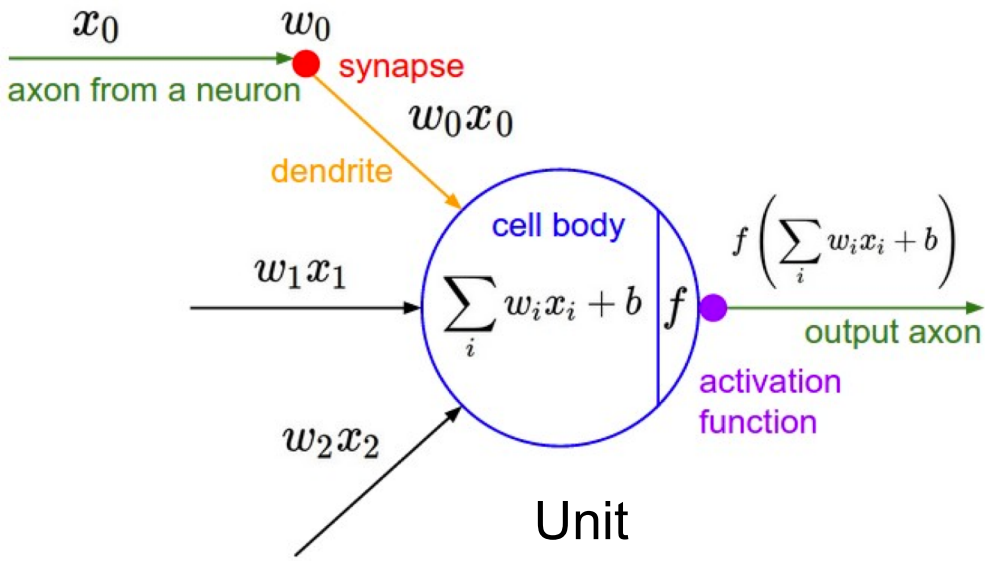
Neurons vs Units

Each element of ANN is a node called **unit**

- Units are connected by links
- Each link has a numeric weight



Biological neuron



Unit

Real neuron is far away from our simplified model - unit

Simple Computations in Units

Linear input function

- Calculate weighted sum of all inputs

$$h_{in} = \sum_i w_i x_i + b$$

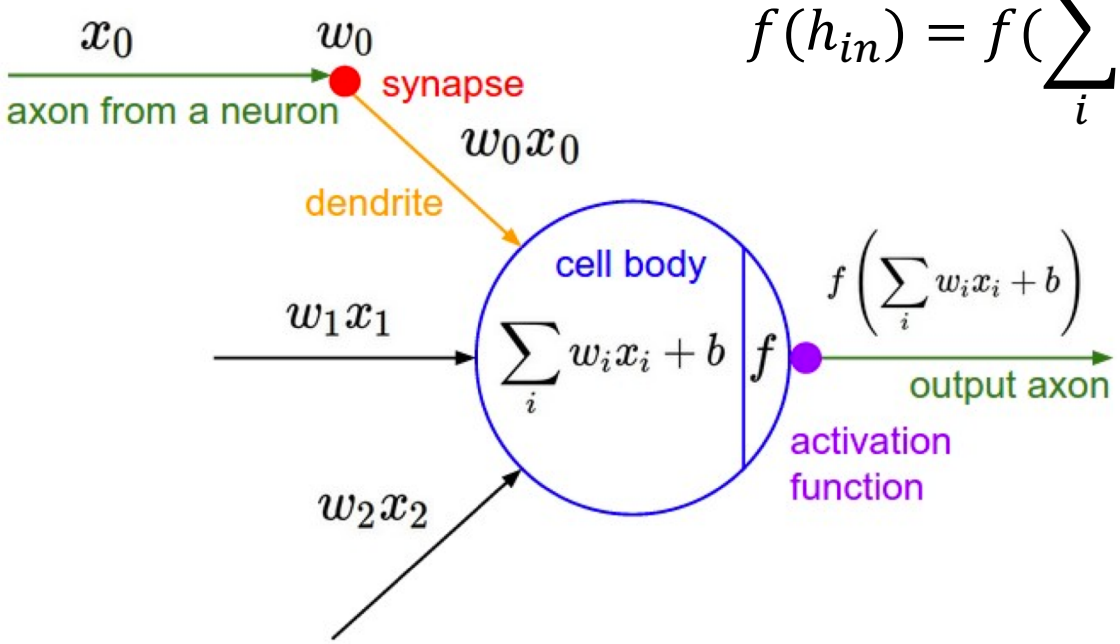
Weights

Bias

Nonlinear activate function

- Transform sum into activation level as a single output

$$f(h_{in}) = f(\sum_i w_i x_i + b)$$

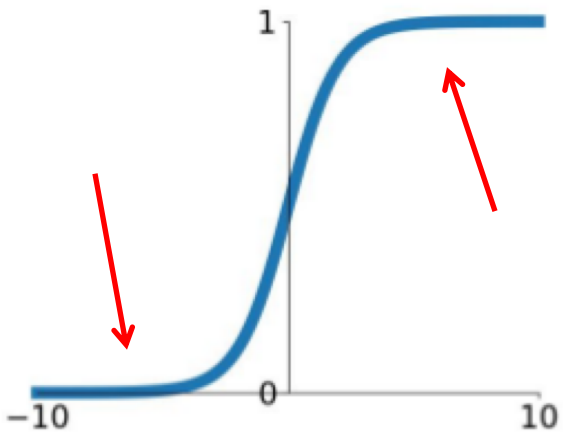


Activation Functions

1. Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating firing rate of a neuron



Two major issues

1. Saturated neurons kill the gradient

2. Sigmoid outputs are not zero-centered (always positive)

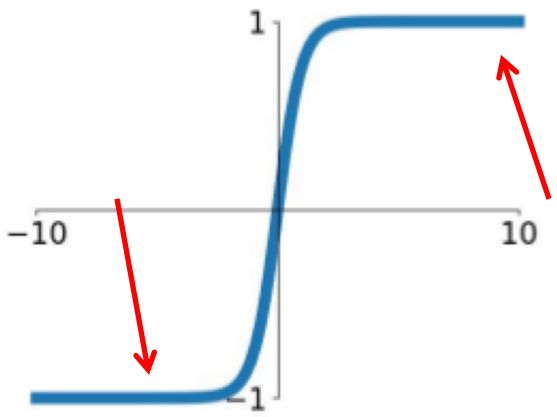
→ $\boldsymbol{w}$ is always all positive or all negative (zig-zagging dynamics in the gradient update)

Activation Functions

2. Tanh (hyperbolic tangent function)

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- Squashes numbers to range [-1,1]
- Zero-centered (better than Sigmoid)



The issue still holds

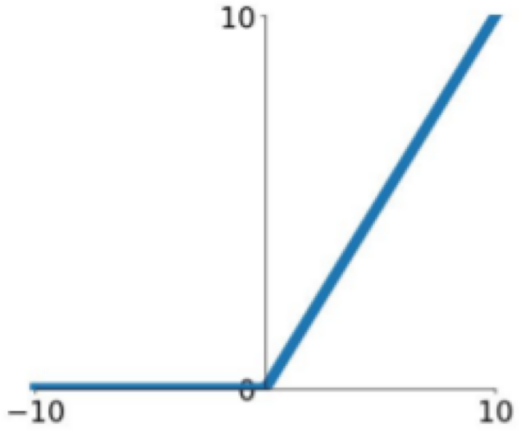
1. Saturated neurons kill the gradient

Activation Functions

3. ReLU (Rectified Linear Unit)

$$f(x) = \max(0, x)$$

- Does not saturate
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice



One issue is …

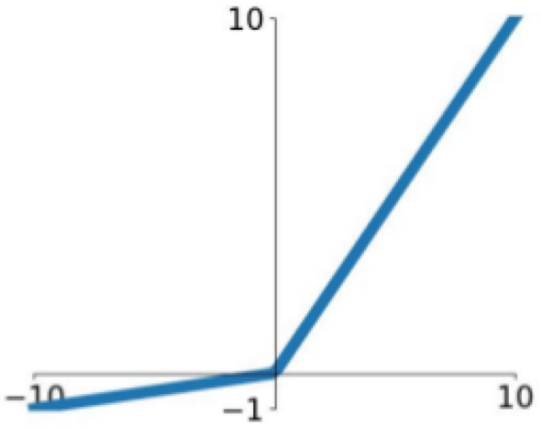
1. Some units can be never activated (i.e. gradient is 0 when $x < 0$)

Activation Functions

3. Leaky ReLU

$$f(x) = 1(x < 0)ax + 1(x \geq 0)x \quad \text{with a small } a$$

- Does not saturate
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice



Will not die unlike ReLU

Activation Functions

4. Maxout

$$f(x) = \max(w_1^T x + b_1, w_2^T x + b_2)$$

- Generalize ReLu and Leaky ReLu
- Ex. $w_1 = b_1 = b_2 = 0$ → ReLu
- Linear regime! Does not saturate! Does not Die

Double the number of parameters ☹

Practical Remarks on Activation (in CNN)

ReLU has been very popular these days. Use ReLU!

- Be careful with learning rates and possibly monitor the fraction of *dead* units in a network

Try out Leaky ReLU / Maxout

Try out tanh but don’t expect much

Do not use sigmoid anymore

Stay tuned! The above arguments can be nullified!

Outline

- Artificial Neurons
- Feedforward Neural Networks
- Backpropagation

(Credits: Andrew Ng's UFLDL Tutorial)

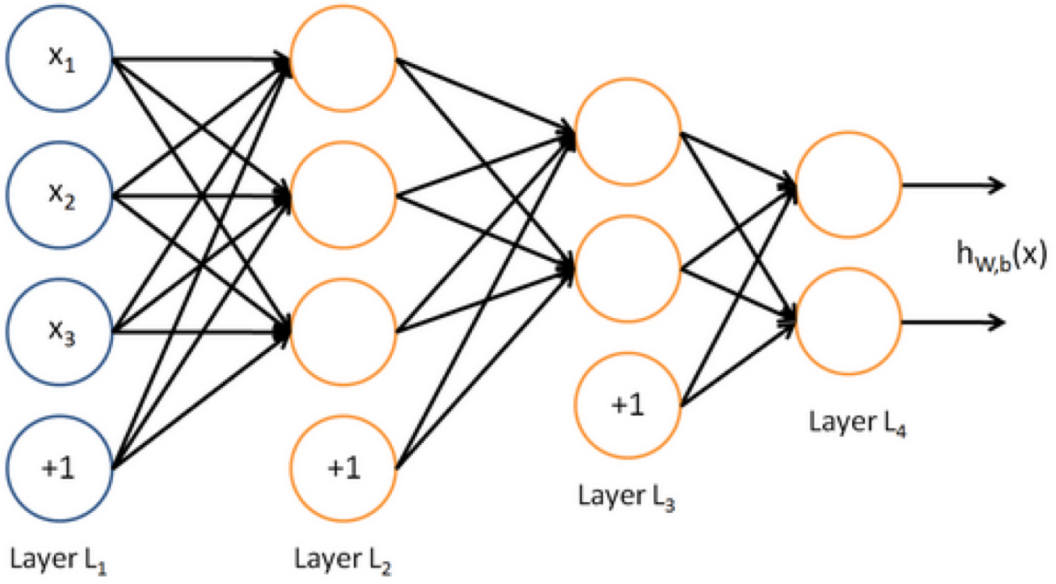
Feedforward Neural Networks

Flexible architecture

- Can be arranged in any multiple layers
- Can have any multiple input/output units

Each unit is linked only with the units in next layer

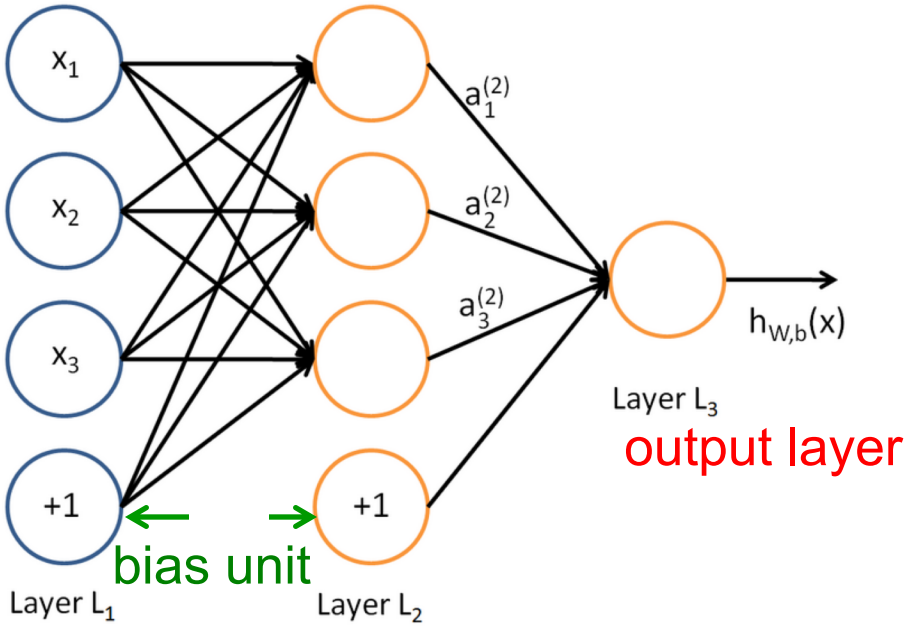
- No unit is linked between the same layer, back to the previous layer or skipping a layer



[UFLDL Tutorial]

Conventions for Feedforward Neural Networks

Follow the notation of Andrew Ng's UFLDL tutorial



- L: # layers
- $W_{ij}^{(l)}$: the weight from unit j of layer l to unit i of layer $l+1$
- $a_i^{(l)}$: activation (i.e. output value) of unit i in layer l
- $a_i^{(1)} = x_i$: i-th input

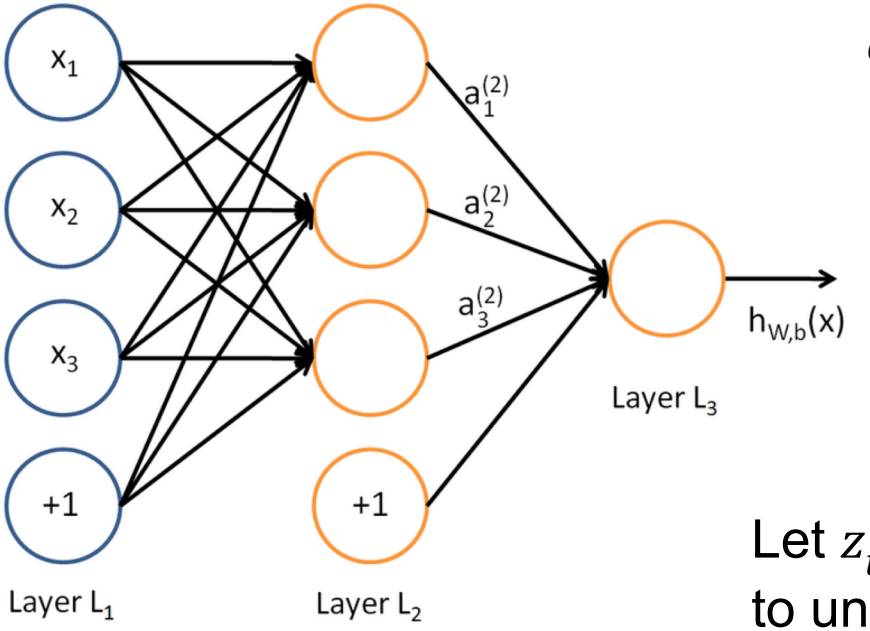
$$\#\text{ parameters} = \#W + \#b$$

- $\#W$ = # edges (except bias units)
- $\#b$ = # nodes (except input/bias nodes)

[UFLDL Tutorial]

Feedforward Neural Networks

Let's compute forward propagation



$$a_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)})$$

$$a_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)})$$

$$a_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)})$$

$$h_{W,b}(x) = a_1^{(3)}$$

$$= f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})$$

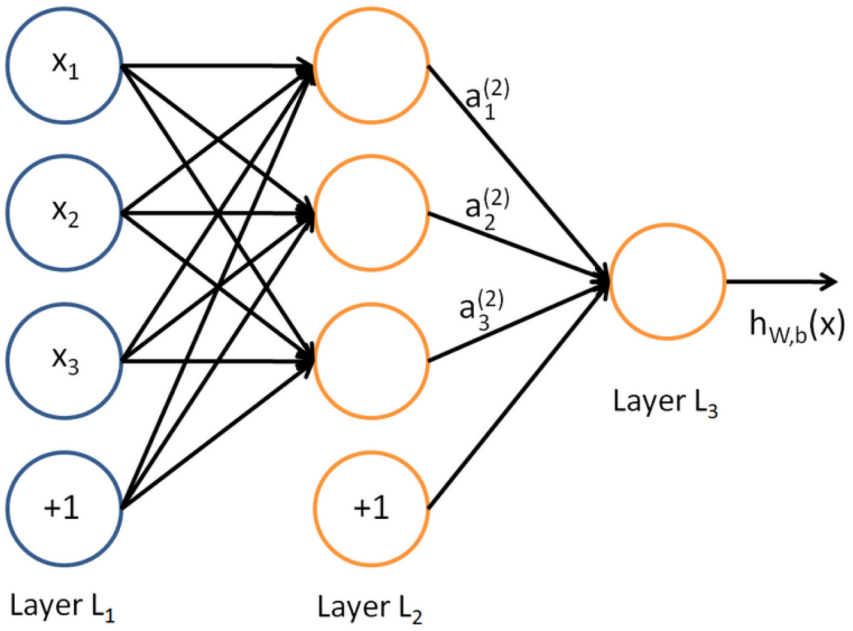
Let $z_i^{(l)}$ be the total weighted sum of inputs to unit i in layer l

$$z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)}x_j^{(1)} + b_j^{(1)} \quad \rightarrow a_i^{(l)} = f(z_i^{(l)})$$

[UFLDL Tutorial]

Feedforward Neural Networks

Let's compute forward propagation



Then we have compact form (with vectors in an element-wise fashion)

$$\boldsymbol{z}^{(2)} = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}$$

$$\boldsymbol{a}^{(2)} = f(\boldsymbol{z}^{(2)})$$

$$\boldsymbol{z}^{(3)} = \boldsymbol{W}^{(2)}\boldsymbol{a}^{(2)} + \boldsymbol{b}^{(2)}$$

$$h_{W,b}(\boldsymbol{x}) = \boldsymbol{a}^{(3)} = f(\boldsymbol{z}^{(3)})$$

In summary, we recursively compute

$$\boldsymbol{a}^{(1)} = \boldsymbol{x}$$

$$\boldsymbol{z}^{(l+1)} = \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l)} + \boldsymbol{b}^{(l)}$$

$$\boldsymbol{a}^{(l+1)} = f(\boldsymbol{z}^{(l+1)})$$

[UFLDL Tutorial]

Outline

- Artificial Neurons
- Feedforward Neural Network
- Backpropagation

(Credits: Andrew Ng's UFLDL Tutorial)

Chain Rule

Compute the derivative of the composition of two or more functions

Let $F(x) = f(g(x))$. Then, $F'(x) = f'\big(g(x)\big)g'(x)$

- In Leibniz's notation for variables x, y, z, if z depends on y, which depends on x

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

- If $z = f(y)$ and $y = g(x)$

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} = f'(y)g'(x) = f'(g(x)\ )g'(x)$$

Chain Rule

Example: Take derivative for $y = \exp(\sin x^2)$

- Decompose it as the composite of three functions

$$y = f(u) = e^u \qquad u = g(v) = \sin v \qquad v = h(x) = x^2$$

- Their derivatives are

$$\frac{dy}{du} = f'(u) = e^u \qquad \frac{du}{dv} = g'(v) = \cos v \qquad \frac{dv}{dx} = h'(x) = 2x$$

- Since $\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}$

$$\frac{dy}{dx} = e^{\sin x^2} \cdot \cos x^2 \cdot 2x$$

Backpropagation Algorithm

Obtain parameters $\boldsymbol{W}, \boldsymbol{b}$ for given m training examples

$$\left\{\left(\boldsymbol{x}^{(1)}, y^{(1)}\right), \ldots, \left(\boldsymbol{x}^{(m)}, y^{(m)}\right)\right\}$$

The cost function for a single example

$$J(\boldsymbol{W}, \boldsymbol{b}; x, y) = \frac{1}{2}\left\|h_{W,b}(\boldsymbol{x}) - y\right\|^2$$

For a training set of m examples

$$J(\boldsymbol{W}, \boldsymbol{b}) = \left[\frac{1}{m}\sum_{i=1}^{m} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}^{(i)}, y^{(i)})\right] + \Omega(\boldsymbol{W})$$

$$= \left[\frac{1}{m}\sum_{i=1}^{m} \frac{1}{2}\left\|h_{W,b}(\boldsymbol{x}) - y\right\|^2\right] + \Omega(\boldsymbol{W})$$

Average sum-of-squares error Regularization term

Backpropagation Algorithm

Obtain parameters $\boldsymbol{W}, \boldsymbol{b}$ for given m training examples

$$J(\boldsymbol{W}, \boldsymbol{b}) = \left[\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\left\|h_{W,b}(\boldsymbol{x}) - y\right\|^2\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(\boldsymbol{W}_{ji}^{(l)})^2$$

Average sum-of-squares error

Regularization term
(Weight decay term)

Regularization term

- Prevent overfitting (i.e. decrease the magnitude of the weights)
- Do not include bias $\boldsymbol{b}$ (because it's not related to training data)
- λ controls the relative importance of the two terms
- Don't let λ too large (i.e. Let data terms take control)

Backpropagation Algorithm

Obtain parameters $\boldsymbol{W}, \boldsymbol{b}$ for given m training examples

$$J(\boldsymbol{W}, \boldsymbol{b}) = \left[\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\left\|h_{W,b}(\boldsymbol{x}) - y\right\|^2\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(\boldsymbol{W}_{ji}^{(l)})^2$$

Average sum-of-squares error

Regularization term
(Weight decay term)

Find $\boldsymbol{W}$ and $\boldsymbol{b}$ that minimize $J(\boldsymbol{W}, \boldsymbol{b})$

Use (*batch*) *gradient descent*

- Since $J(\boldsymbol{W}, \boldsymbol{b})$ is non-convex, it is susceptible to local optima
- Initialize $\boldsymbol{W}$ and $\boldsymbol{b}$ with random small values
- Do not initialize to all 0's. All parameters in the hidden layers move identically

Gradient Descent (GD)

One iteration of GD for parameters $\boldsymbol{W}, \boldsymbol{b}$

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \boxed{\frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial W_{ij}^{(l)}}} \qquad b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial b_i^{(l)}}$$

Gradient (partial derivative) of errors w.r.t $W_{ij}^{(l)}$

(*i.e.* how much a change in $W_{ij}^{(l)}$ affects the total error)

- α is the learning rate (the most important parameter in practice)

The partial derivatives are

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}) = \left[\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}^{(i)}, y^{(i)}) \right] + \lambda W_{ij}^{(l)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}^{(i)}, y^{(i)})$$

Summation of partial derivate w.r.t. individual training data

Backpropagation

The intuition behind backpropagation

- For each training data $\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)$, run the forward pass to get $h_{W,b}(\boldsymbol{x}^{(i)})$
- Then for each node i in layer l, compute an error term $\delta_i^{(l)}$
- The error term measures how much the node was responsible for the error in the output
- The error term can be directly measured in the output layer
- You see why this algorithm is called backpropagation

Error Term

Error term of neuron i in layer l

$$\delta_i^{(l)} = \frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial z_i^{(l)}} \quad \Rightarrow \quad \delta_i^{(l)} = \frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}}$$ chain rule

$$(\because \boldsymbol{z}^{(l+1)} = \boldsymbol{W}^{(l)} \boldsymbol{a}^{(l)} + \boldsymbol{b}^{(l)}, \boldsymbol{a}^{(l)} = \boldsymbol{f}(\boldsymbol{z}^{(l)}))$$

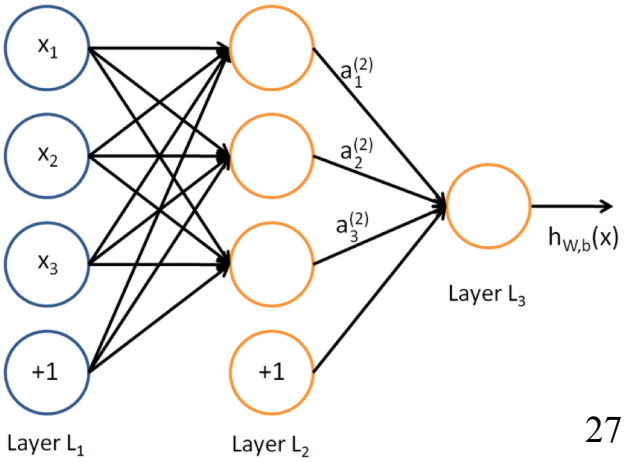
- At last layer L

$$\delta_i^{(L)} = \frac{\partial J}{\partial z_i^{(L)}} = \frac{\partial J}{\partial a_i^{(L)}} \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} = \frac{\partial}{\partial a_i^{(L)}} \left\| h_{W,b}(\boldsymbol{x}) - y \right\|^2 f'(z_i^{(L)})$$

$$= (h_{W,b}(\boldsymbol{x}_i) - y)\, f'(z_i^{(L)})$$

$$(\because h_{W,b}(\boldsymbol{x}_i) = a_i^{(L)})$$

($f'(z_i^{(L)})$ depends on what activation function is used)



Error Term

Error term of neuron i in layer l

$$\delta_i^{(l)} = \frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial z_i^{(l)}} \quad \Rightarrow \quad \delta_i^{(l)} = \frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} \qquad \text{chain rule}$$

$$(\because \boldsymbol{z}^{(l+1)} = \boldsymbol{W}^{(l)} \boldsymbol{a}^{(l)} + \boldsymbol{b}^{(l)}, \boldsymbol{a}^{(l)} = \boldsymbol{f}(\boldsymbol{z}^{(l)}))$$

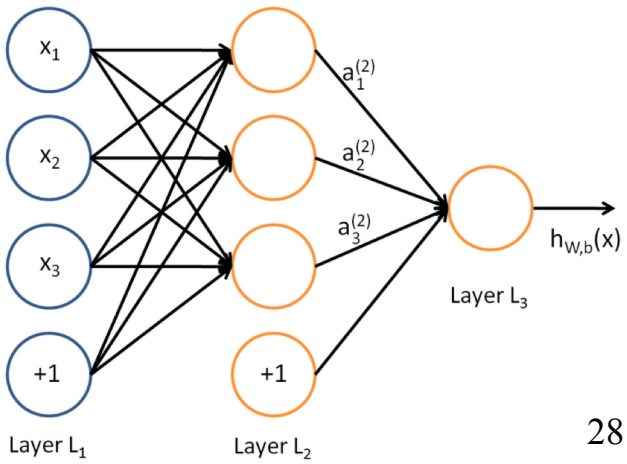
- At hidden layer l

$$\delta_i^{(l)} = \frac{\partial J}{\partial z_i^{(l)}} = \sum_{j=1}^{N_{l+1}} \frac{\partial J}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}} = \sum_{j=1}^{N_{l+1}} \frac{\partial z_j^{(l+1)}}{\partial z_j^{(l)}} \delta_j^{(l+1)}$$

$$z_j^{(l+1)} = W_j^{(l)} f\left(z_j^{(l)}\right) + b_j^{(l)}$$

$$\Rightarrow \frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}} = W_{ji}^{(l)} f'(z_i^{(l)})$$

$$\delta_i^{(l)} = \sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_i^{(l+1)} f'(z_i^{(l)})$$



Why Do We Care for Error Term?

Training is iteratively done by GD for parameters $\boldsymbol{W}, \boldsymbol{b}$

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial W_{ij}^{(l)}} \qquad b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial b_i^{(l)}}$$

- Gradient is easily computable from error term

$$\frac{\partial J}{\partial W_{ij}^{(l)}} = \frac{\partial J}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial W_{ij}^{(l)}} \Rightarrow \delta_i^{(l+1)} a_j^{(l)} \quad (\because \boldsymbol{z}^{(l+1)} = \boldsymbol{W}^{(l)} \boldsymbol{a}^{(l)} + \boldsymbol{b}^{(l)})$$

$$\frac{\partial J}{\partial b_i^{(l)}} = \frac{\partial J}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial b_i^{(l)}} \Rightarrow \delta_i^{(l+1)}$$

Backpropagation Algorithm

1. Perform a feedforward pass
   - Computing the activations for all layers up to the output layer (i.e. $a^{(l)}$ for all $l = 1, \ldots, L$)

2. Compute error term for every node
   - For each output unit i in layer L (the output layer)

$$\delta_i^{(L)} = -(y_i - a_i^{(L)}) \cdot f'(z_i^{(L)}) \quad \text{where } h_{W,b}(\boldsymbol{x}) = a^{(L)} = f(z^{(L)})$$

(often $i = 1$ in layer L)

   - For each layer $l = L-1, L-2, \ldots, 2$
     - For each node i in layer l, $\quad \delta_i^{(l)} = (\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_i^{(l+1)})\, f'(z_i^{(l)})$

3. Compute the desired partial derivatives

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; x, y) = a_j^{(l)} \delta_i^{(l+1)} \qquad \frac{\partial}{\partial b_i^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; x, y) = \delta_i^{(l+1)}$$

Derivatives of Activation Functions

How to compute $f'(z^{(l)})$?

- Sigmoid function

$$\sigma(x) = \frac{1}{1+e^{-x}} \quad \Longrightarrow \quad \frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$$

- ReLU function

$$f(x) = \max(0, x) \quad \Longrightarrow \quad \frac{df(x)}{dx} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

- Hyperbolic tangent

$$f(x) = \tanh(x) \quad \Longrightarrow \quad \frac{d\tanh(x)}{dx} = 1 - \tanh^2(x)$$

- Derivation can be easily found by googling

Gradient Descent Algorithm

Obtain parameters $\boldsymbol{W}, \boldsymbol{b}$ for given m training examples

- Matrix-vectorial notation is favorable for GPU

Set $\Delta \boldsymbol{W}^{(l)} := 0, \Delta \boldsymbol{b}^{(l)} := 0$ (matrix/vector of zeros) for all l

- Both have the same size with $\boldsymbol{W}^{(l)}$ and $\boldsymbol{b}^{(l)}$, respectively

For $t = 1$ to T

1. Use backpropagation to compute $\nabla_{\boldsymbol{W}^{(l)}} J(W, b; x, y)$ and $\nabla_{\boldsymbol{b}^{(l)}} J(W, b; x, y)$

2. Set $\Delta \boldsymbol{W}^{(l)} := \Delta \boldsymbol{W}^{(l)} + \nabla_{\boldsymbol{W}^{(l)}} J(W, b; x, y)$

   $\Delta \boldsymbol{b}^{(l)} := \Delta \boldsymbol{b}^{(l)} + \nabla_{\boldsymbol{b}^{(l)}} J(W, b; x, y)$

3. Update the parameters

$$\boldsymbol{W}^{(l)} = \boldsymbol{W}^{(l)} - \alpha \left[\left(\frac{1}{m} \Delta \boldsymbol{W}^{(l)} \right) + \lambda \boldsymbol{W}^{(l)} \right] \qquad \boldsymbol{b}^{(l)} = \boldsymbol{b}^{(l)} - \alpha \left[\frac{1}{m} \Delta \boldsymbol{b}^{(l)} \right]$$

Backpropagation Algorithm

1. Perform a feedforward pass

   - Computing the activations for all layers up to the output layer (i.e. $\boldsymbol{a}^{(l)}$ for all $l = 1, \ldots, L$)

2. Compute error term for every node

   - For each output unit i in layer L (the output layer)

     $\boldsymbol{\delta}^{(L)} = -(y - a^{(L)}) \circ f'(\boldsymbol{z}_i^{(L)})$ (Elementwise multiplication)

   - For each layer $l = L - 1, L - 2, \ldots, 2$

     - For each node i in layer l, $\boldsymbol{\delta}^{(l)} = ((\boldsymbol{W}^{(l)})^T \boldsymbol{\delta}^{(l+1)}) \circ f'(\boldsymbol{z}_i^{(l)})$

3. Compute the desired partial derivatives

$$\nabla_{\boldsymbol{W}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; x, y) = \boldsymbol{\delta}^{(l+1)} (\boldsymbol{a}^{(l)})^T \qquad \nabla_{\boldsymbol{b}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; x, y) = \boldsymbol{\delta}^{(l+1)}$$

Additional Slides

Approaches to Learning Features

Supervised Learning

- *End-to-end learning* of deep architectures (e.g., deep neural networks) with *back-propagation*
- Works well when the amounts of labels is large
- Model structure is important (e.g. convolutional structure)

Unsupervised Learning

- Learn *statistical structure or dependencies* of the data from unlabeled data
- Layer-wise training
- Useful when the amount of labels is not large

[Honglak Lee]

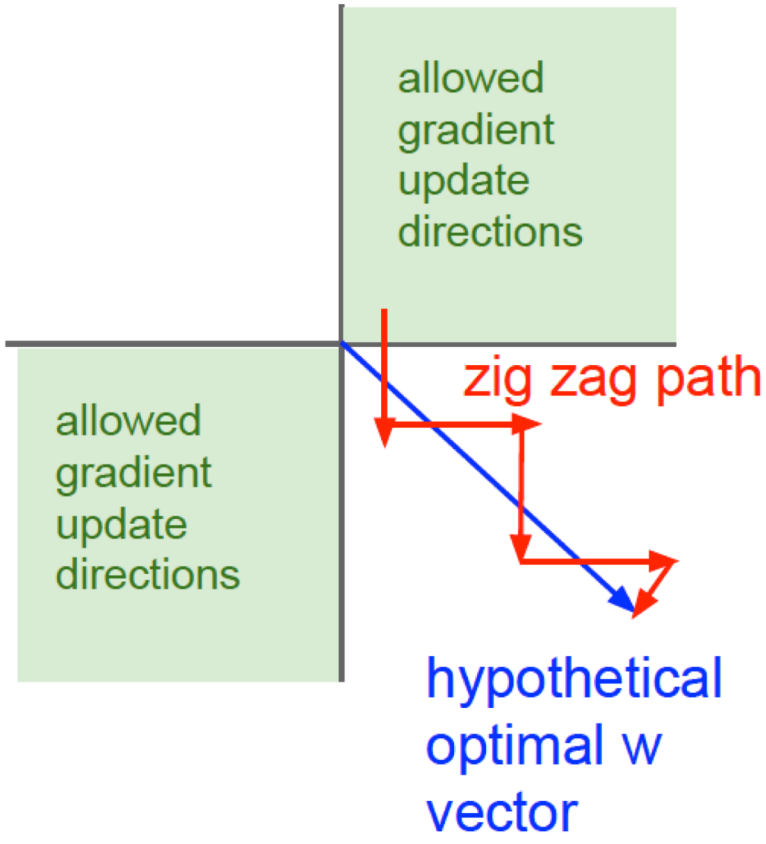
Why Is Always Positive Output Problematic?

In multi-layer NN optimization,

- If input data is always positive (x >0) (e.g. image pixel values)
- The gradients on **w** during backpropagation are always all positive or all negatives

$$f\left(\sum_i w_i x_i + b\right)$$

Undesirable zig-zagging dynamics in the gradient update for **w**



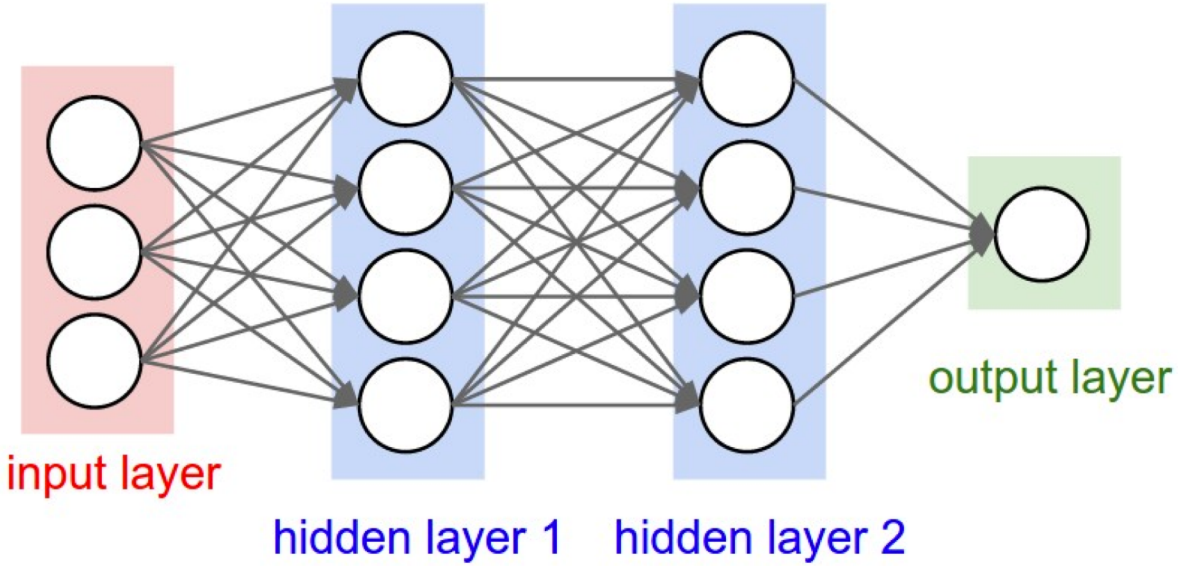
Multi-Layer Neural Networks

Naming conventions

- 3 layers (Do not count the input layer)

Sizes

- 9 neurons (Do not count the input layer)
- # parameters : weight w for each edge, bias b for each neuron
- w: [3 x 4] + [4 x 4] + [4 x 1] = 32, b: 4 + 4 + 1 = 9



Gradient Descent (GD)

The partial derivatives are

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) = \left[\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x^{(i)},y^{(i)}) \right] + \lambda W_{ij}^{(l)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J(W,b;x^{(i)},y^{(i)})$$

- Note that $J(W,b;x^{(i)},y^{(i)}) = \frac{1}{2} \left\| h_{W,b}(x^{(i)}) - y^{(i)} \right\|^2$

The intuition behind backpropagation

- For each training data ($x^{(i)}$, $y^{(i)}$), run the forward pass to get $h_{W,b}(x^{(i)})$
- Then for each node i in layer l, compute an error term $\delta_i^{(l)}$
- The error term measures how much the node was responsible for the error in the output
- The error term can be directly measure in the output layer! You see why this algorithm is called backpropagation