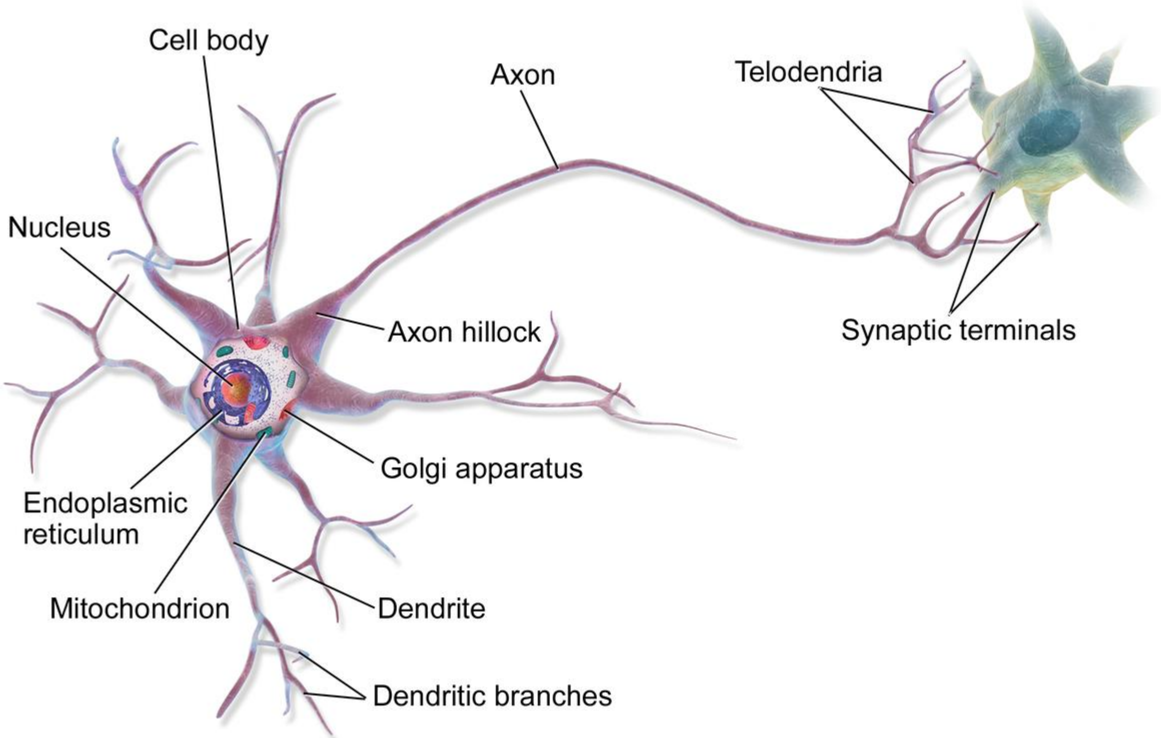
NEURAL NETWORKS & BACK PROPAGATION EXPLAINED

Shingchern D. You

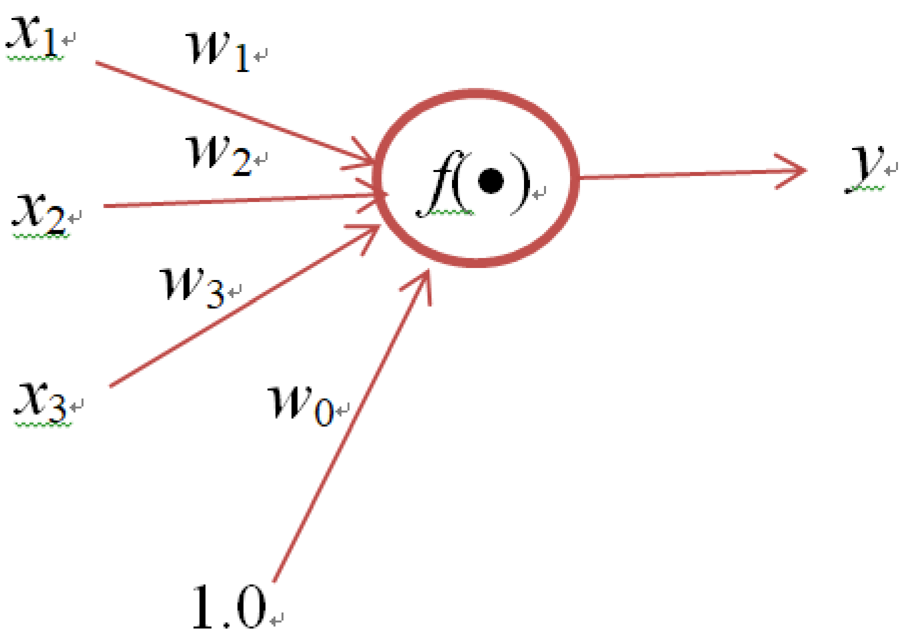
Neuron

- Early developments of neural networks was inspired by biological neural systems



Neuron

- Modeled in engineering terms



Neuron

- Inputs: x_1, x_2, x_3
- Bias: w_0
- Activation function: $f(\cdot)$
- Output: $y = f(\sum_{i=1}^{3} x_i w_i + w_0)$
- One simple activation function: **hard limit**

$$y = \begin{cases} 1, & \text{if } \left(\sum_{i=1}^{3} x_i w_i + w_0 \right) > 0 \\ 0 \ (\text{or } -1), & \text{otherwise} \end{cases}$$

Matrix representation

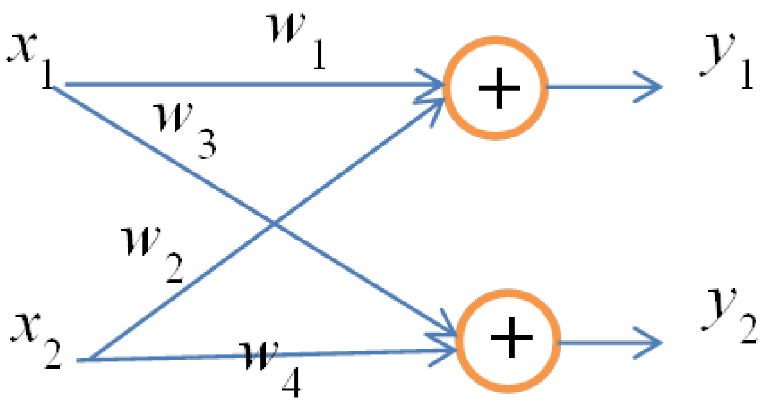
- Ignore the activation function at this moment
- Consider the multiplication-add part

$$\sum_{i=1}^{3} x_i w_i + w_0$$

- It can be expressed in matrix multiplication

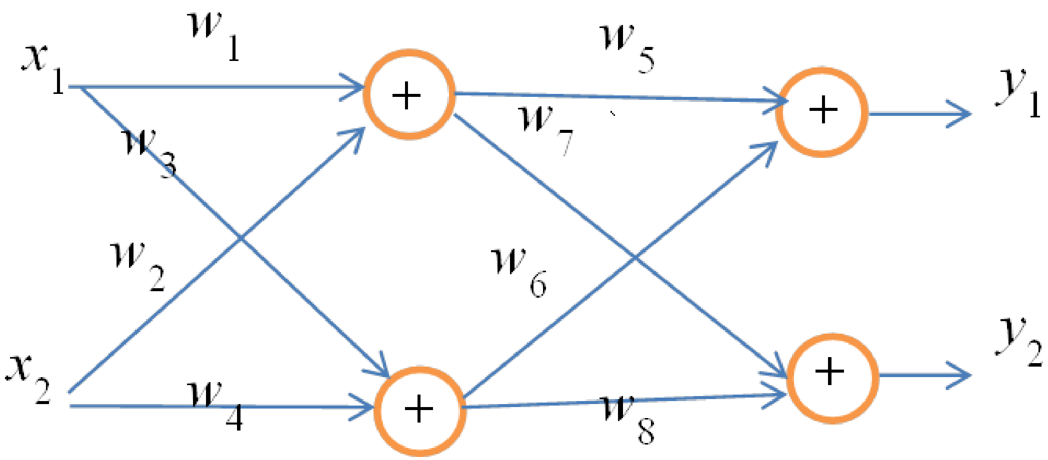
$$\begin{bmatrix} 1 & x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

What if we have multiple outputs



- $\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ (ignore bias)

Multiple linear layers



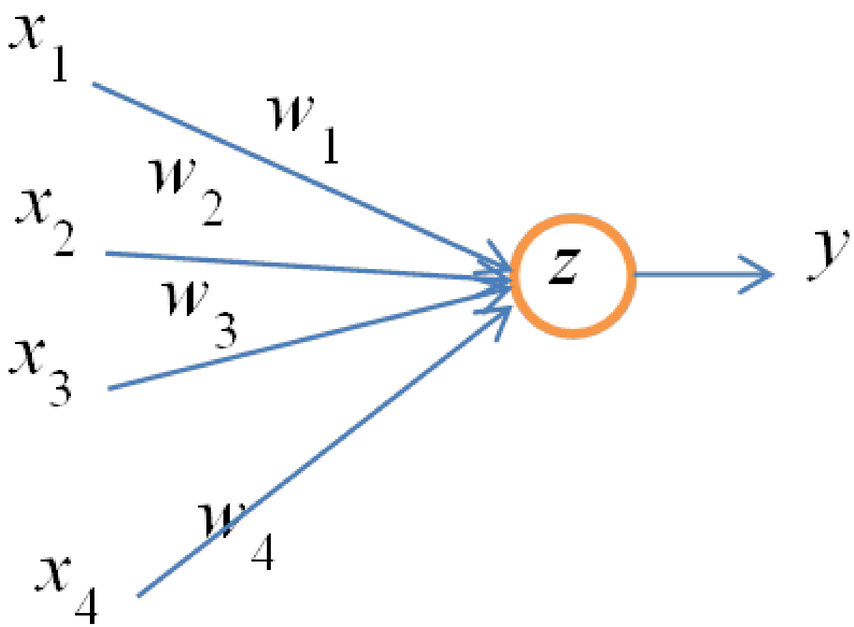
- $\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_5 & w_6 \\ w_7 & w_8 \end{bmatrix} \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} & \\ & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$
- No reason to use multiple layers

Perceptron

- Define hard-limit function $U(x) = \begin{cases} 1, & x > 0 \\ -1, & x \leq 0 \end{cases}$
- A perceptron is a two-layer network with hard-limit activation function
  - Conventionally input is one layer (though doing nothing)

Perceptron

- Notation: $z = \sum_{i=1}^{3} x_i w_i + w_0$, $y = U(z)$



Perceptron vs linear classifier

- Recall the so-called linear classifier
- Classifier is represented as

$$f(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b \quad (b \text{ is a scalar})$$

- $\boldsymbol{x} \in C_+$ if $f(\boldsymbol{x}) > 0$, otherwise $\boldsymbol{x} \in C_-$
- Class assignment can be expressed as

$$\mathrm{Class} = U(f(\boldsymbol{x}))$$

- Therefore, we know perceptron is exactly a linear classifier

Training perceptron

- We can use gradient descent to train perceptron (thus linear classifier)
- Let the loss function be

$$J(\boldsymbol{w}) = |z_k d_k| - z_k d_k$$

where k is the index of training sample (up to n), z_k is summing output (given previously), and d_k is the class of desired output

Training perceptron

- Observe this term: $|z_k d_k| - z_k d_k$
  - If z and d have the same sign, this term is zero
  - If z and d have different sign, this term > 0 (mis-classification)
- We want to minimize classification error, so we need to minimize $J(\boldsymbol{w})$ with respect to $\boldsymbol{w}$ for all k
  - Taking gradient over $\boldsymbol{w}$
  - Math to be discussed later, skip this part (homework problem)

Training perceptron

- Eventually, we have the following adaption algorithm
- $$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) + \begin{cases} 0, & \text{if } z_k d_k > 0 \\ \eta \boldsymbol{x}_k d_k, & \text{otherwise} \end{cases}$$
- We can initially assign all weights to one
- When k increases from 1 to n, we are done with one epoch
- We can continue the training for 2^{nd} epoch, 3^{rd} epoch, etc.

XOR problem

- At first, researchers were excited to have perceptron learning algorithm
- Minksy and Papert (1969) show that perceptron cannot solve XOR problem (mentioned before)

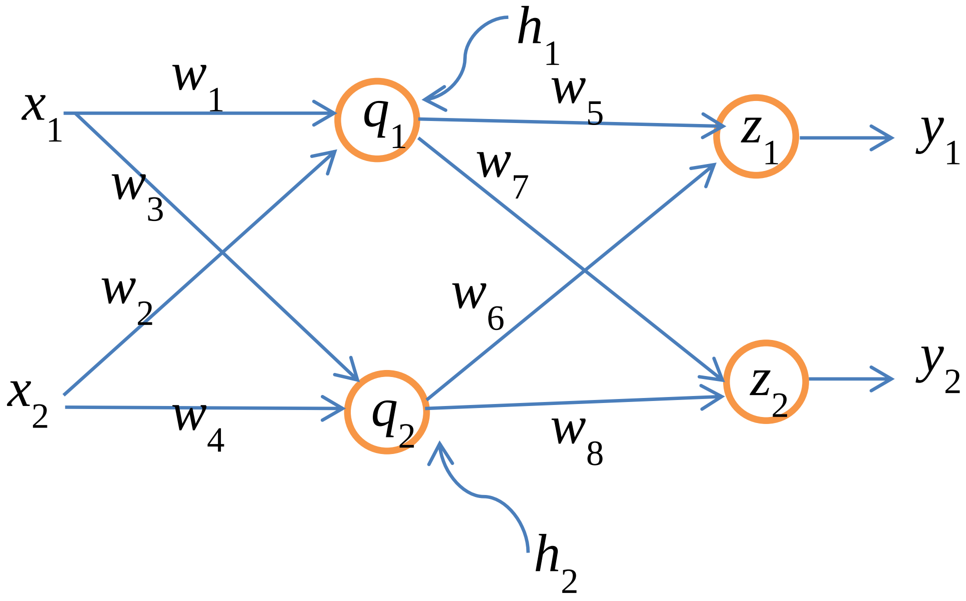
x1	x2	y
-1	-1	-1
1	-1	1
-1	1	1
1	1	-1

XOR problem

- A huge strike
- Neural networks not received attention for many years
- Until some researchers proposed multilayer networks
- Remember, it is useless if we have multilayers but no **nonlinear** activation functions in between

Multilayer neural networks

- The neural network shown has three layers: input, hidden, and output (three layers)

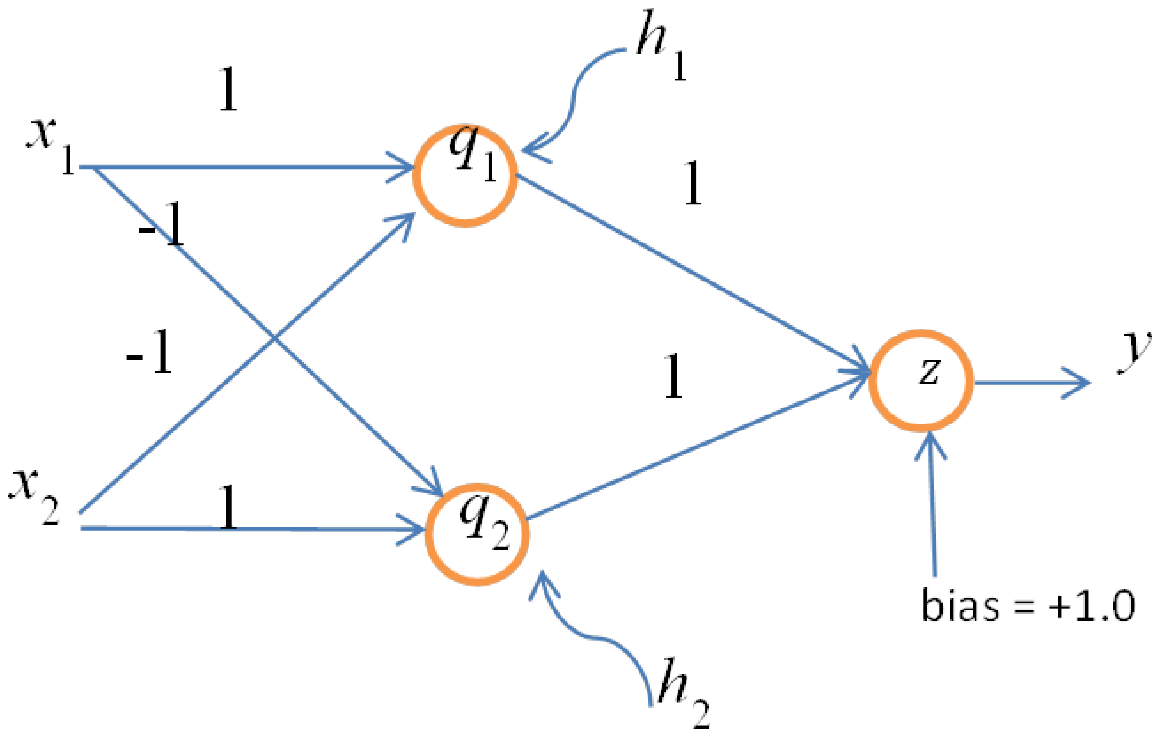


Multilayer neural networks

- How to read the figure (ignore bias for simplicity)
  - $q_1 = \sum_{i=1}^{2} x_i w_i$, $h_1 = f(q_1)$
  - $z_1 = \sum_{i=5}^{6} h_i w_i$, $y_1 = f(z_1)$
- We want to show that a simple two-layer perceptron can solve XOR problem

Solving XOR problem

- $C_+ = \left\{ \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right\}, C_- = \left\{ \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix} \right\}$



Solving XOR problem

- What we have is the following table

x1	x2	q1	q2	h1	h2	z	z+b	y
-1	-1	0	0	-1	-1	-2	-1	-1
-1	1	-2	2	-1	1	0	1	1
1	-1	2	-2	1	-1	0	1	1
1	1	0	0	-1	-1	-2	-1	-1

Solving XOR problem

- We may think that the second (hidden) layer performs a kind of **feature engineering**
  - To make a tough problem easy to solve with a linear classifier
- It is proved that **three-layer network** with nonlinear activation functions is a universal approximator (Universal approximation theorem, idea from Kolmogorov's Theorem)
- All wee need is a good training algorithm

Training neural networks

- Want to train neural networks
  - Training means to find a set of “good” weights
  - But, how to define “good”
  - Use objective (cost) function
- Training neural networks is converted to an optimization problem
  - Cannot hope to solve the problem analytically
  - Use gradient descent (or its variations) instead

Backpropagation from Wiki

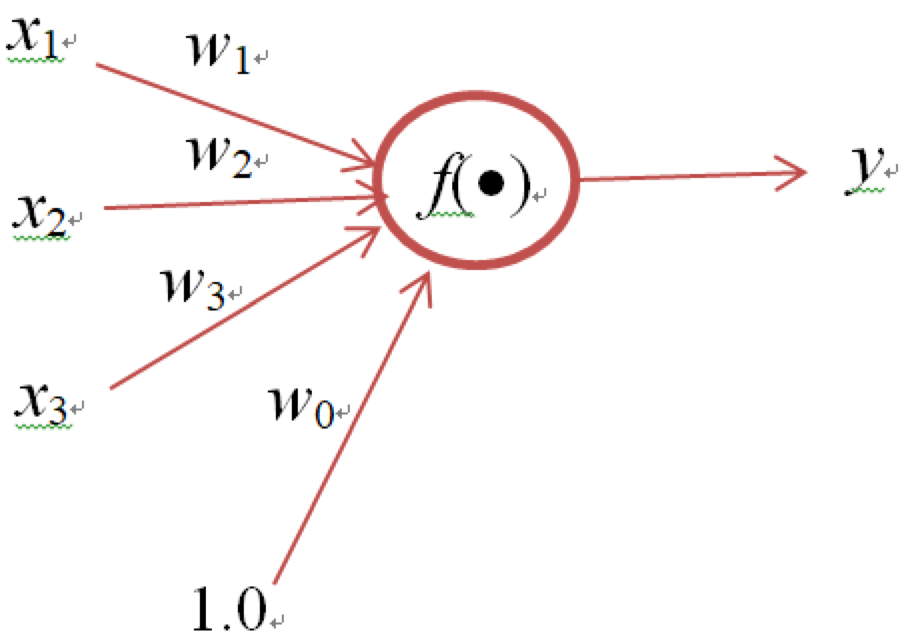
- Backpropagation (Backprop, BP) is a widely used algorithm in training feedforward neural networks for supervised learning
- BP computes the gradient in weight space of a feedforward neural network, with respect to a loss function
- BP is often used loosely to refer to the entire learning algorithm, including how the gradient is used, such as by stochastic gradient descent

Motivation for simplification

- The general form of back propagation is difficult to understand because of the notation
- We need to consider the following indices: Layer index, input index, output index, weights index, and iteration (time) index
- Therefore, a notation like $w_{i,j}^{L}(k+1)$ might be used in the literature
- To avoid unnecessary confusion, we intend to make the notation simple and easy to follow

Neuron

- $z = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + w_0$
- $y = f(z)$, $f(z)$ is called as activation function



Sigmoid function

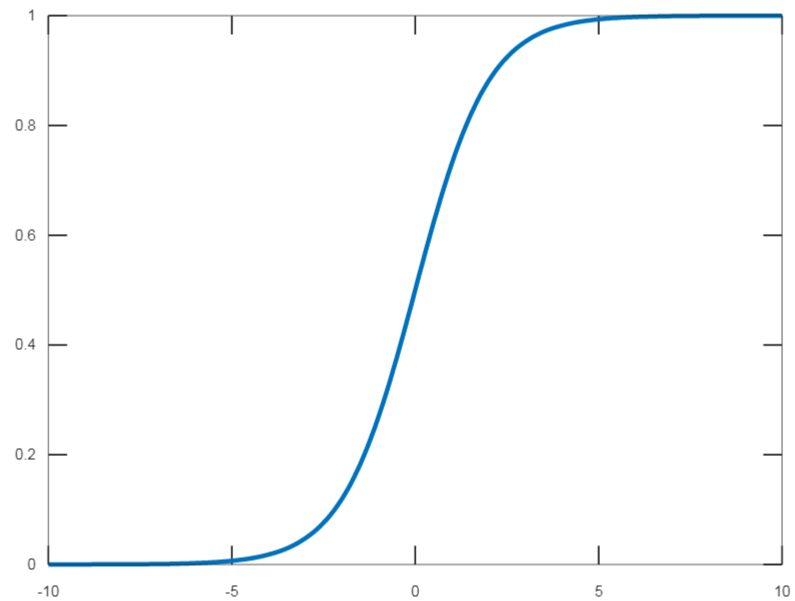
- For multilayer networks, we want to introduce nonlinear activation function
  - Linear activation function does not work (why? Think of matrix addition and multiplication in linear algebra)
- Another widely used activation function is **sigmoid**

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

- Background of sigmoid function, cf. section 6.6 of https://www.cs.jhu.edu/~jason/papers/jurafsky+martin.bookdraft07.ch6.pdf

Sigmoid function

- Many variants (but equivalent)
- Domain $(-\infty, \infty)$, range $(0,1)$
- Sigmoid (logistic function) has its root in statistics (cf. https://en.wikipedia.org/wiki/Logistic_function)



Sigmoid Function

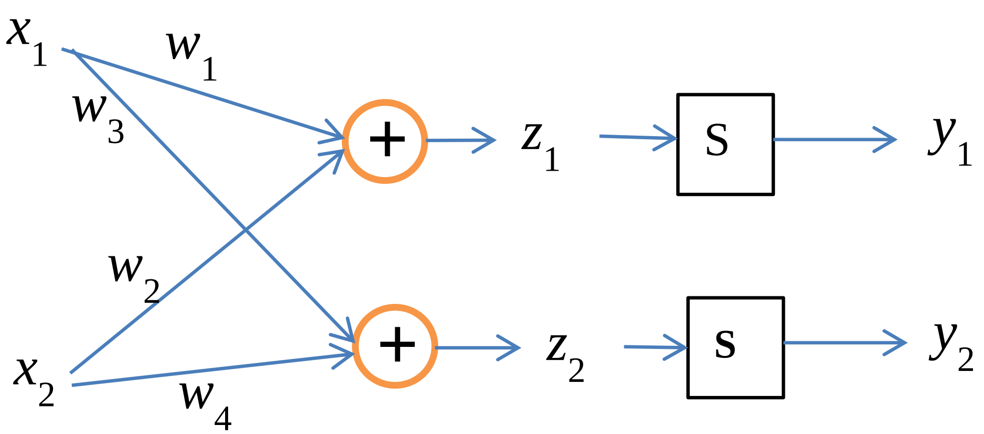
- When replacing z with matrix multiplication, we have

$$f(\mathbf{x}) = \frac{1}{1 + \exp(-\boldsymbol{w}^T\mathbf{x})}$$

- If we want, we can also add “bias” to equation, i.e., we use $(\boldsymbol{w}^T\mathbf{x} + w_0)$ in place of $\boldsymbol{w}^T\mathbf{x}$
- Exercise: Write down $\boldsymbol{w}$ and $\mathbf{x}$ for the network in the previous example

Forward computation

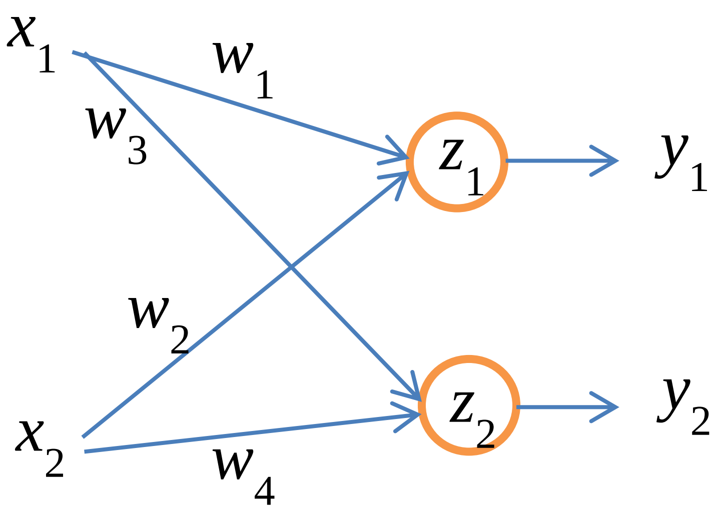
- The following is a simple example (s: sigmoid fn)



- $z_1 = x_1 w_1 + x_2 w_2$
- $y_1 = \frac{1}{1+\exp(-z_1)}$

Forward computation

- Simplify the drawings



Single layer back propagation

- We use mean-square error as the **cost (loss) function** for **stochastic gradient descent**
- $\varepsilon = \varepsilon_1 + \varepsilon_2 = \frac{1}{2}((y_1 - d_1)^2 + (y_2 - d_2)^2)$
- d_i is the desired output for node i (constant, derivative $= 0$)
- We add $\frac{1}{2}$ to remove the constant in derivatives
- **Exercise: Find ε for batch gradient descent**

Single layer back propagation

- Do gradient search to find min ε

$$\mathbf{w}(k+1) \leftarrow \mathbf{w}(k) - \eta \nabla \varepsilon(\mathbf{w}(k))$$

- Note that ε is a function of iteration index k in gradient descent algorithm (k dropped later)
- In addition, z_1, z_2, x_1, and x_2 are also functions of k
- T**o simplify the notation, we drop the variable k in the expression**, such as y_1 actually means $y_1(k)$

Single layer back propagation

- To simplify the discussion, consider only updating w_1
- Therefore, $w_1(k+1) \leftarrow w_1(k) - \eta \frac{\partial}{\partial w_1} \varepsilon$
- We know $\frac{\partial}{\partial w_1} \varepsilon = \frac{\partial}{\partial w_1} \varepsilon_1$ because ε_2 is not related to w_1

Single layer back propagation

- Recall that we have

$$\varepsilon_1 = \frac{1}{2}(y_1 - d_1)^2$$

where d_1 is constant (desired output)

$$y_1 = \frac{1}{1 + \exp(-z_1)}$$

$$z_1 = x_1 w_1 + x_2 w_2$$

Single layer back propagation

- By using chain rule, we have $\frac{\partial \varepsilon_1}{\partial w_1} = \frac{\partial \varepsilon_1}{\partial y_1}\frac{\partial y_1}{\partial z_1}\frac{\partial z_1}{\partial w_1}$

where

$$\frac{\partial \varepsilon_1}{\partial y_1} = (y_1 - d_1)$$

$$\frac{\partial y_1}{\partial z_1} = y_1(1 - y_1) \text{ (exercise!)}$$

$$\frac{\partial z_1}{\partial w_1} = x_1$$

Single layer back propagation

- Finally, we obtain

$$\frac{\partial \varepsilon_1}{\partial w_1} = (y_1 - d_1)y_1(1 - y_1)x_1$$

- Therefore,

$$w_1(k+1) \leftarrow w_1(k) - \eta(y_1 - d_1)y_1(1 - y_1)x_1$$

- Remember, we need to evaluate $y_1(k)$ & $x_1(k)$ every iteration to update $w_1(k+1)$
- We can derived the update rule for other weights by the same method

Single layer back propagation

- In supervised learning, the variable values x_1 & x_2 in previous figure are from one training sample $\boldsymbol{x}_{(q)}$
- Example: (update weights for every sample)
  - Training samples $\boldsymbol{x}_{(1),}$ … , $\boldsymbol{x}_{(n)}$
  - (Optional) Shuffle $\boldsymbol{x}_{(1),}$ … , $\boldsymbol{x}_{(n)}$
  - $[x_1(1)\ x_2(1)]^T \leftarrow \boldsymbol{x}_{(1),}$ Update weights, $x_1(k)$: first x value @ step k
  - … Update weights
  - $[x_1(n)\ x_2(n)]^T \leftarrow \boldsymbol{x}_{(n),}$ Update weights
  - $[x_1(n+1)\ x_2(n+1)]^T \leftarrow \boldsymbol{x}_{(1)}$, Update weights
  - …
- One training **epoch** is updating weights after using all training samples

Questions

- If we update weights for every training sample, what kind of gradient descent is it?
  - Batch
  - Mini batch
  - Stochastic
  - Vanilla

Logistic regression

- In our previous example, we use the sigmoid activation function and mse as loss function
- If instead we use the following loss function, we have the logistic regression (loss function is from Bernooulli trial)

$$J(\boldsymbol{w}) = -\log_2 \left(\prod_{k=1}^{n} y_k^{d_k} (1 - y_k)^{(1-d_k)} \right)$$

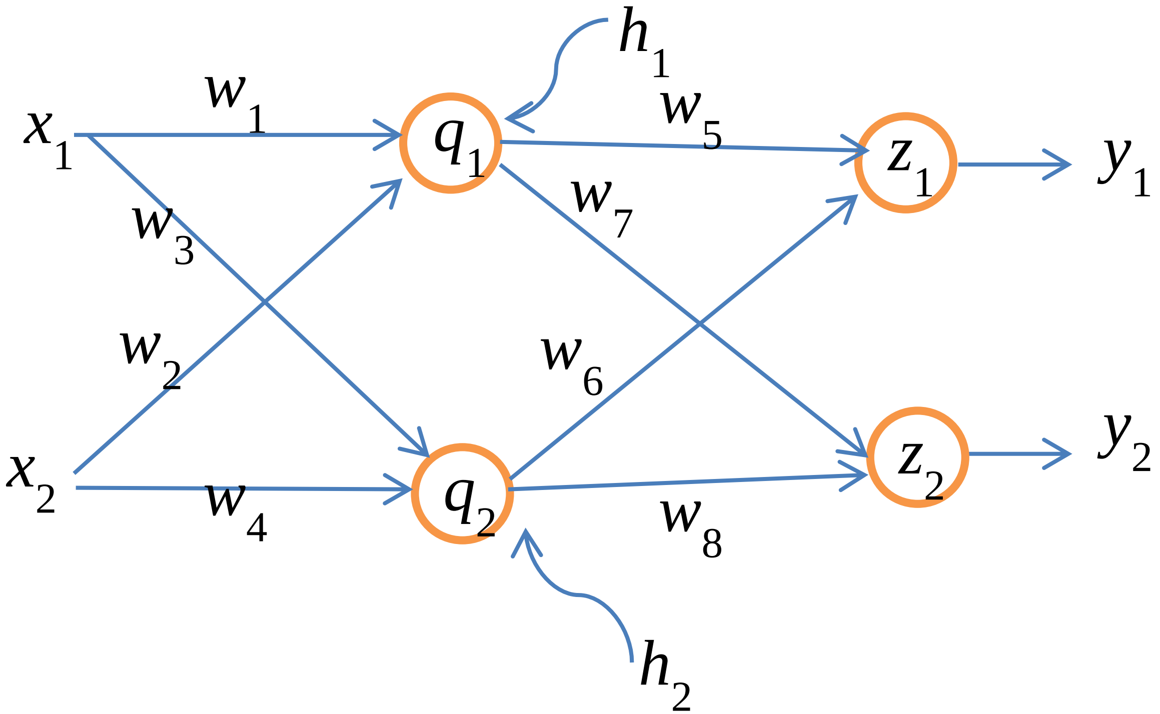
  - Recall $0 < y_k < 1$, so we have to use $d_k \in \{0,1\}$

Logistic regression

- If consider only one sample, we have
$$J = -\{d \log_2 y + (1-d)\log_2(1-y)\}$$
- This loss function is also known as **cross entropy** (to be mentioned later)
- Summary: Logistic regression
  - A two-layer classifier
  - Activation function: sigmoid
  - Loss function: cross entropy

Multi-layer back propagation

- We now extend the concept to multi-layer networks



Multi-layer back propagation

- What we have now are

$$q_1 = x_1 w_1 + x_2 w_2$$

$$h_1 = \frac{1}{1 + \exp(-q_1)}$$

$$z_1 = h_1 w_5 + h_2 w_6$$

$$y_1 = \frac{1}{1 + \exp(-z_1)}$$

$$\varepsilon_1 = \frac{1}{2}(y_1 - d_1)^2$$

Multi-layer back propagation

- From the single-layer results, we know

$$w_5(k+1) \leftarrow w_5(k) - \eta \frac{\partial \varepsilon}{\partial w_5}$$

where $\frac{\partial \varepsilon}{\partial w_5} = \frac{\partial \varepsilon_1}{\partial w_5} = \frac{\partial \varepsilon_1}{\partial y_1} \frac{\partial y_1}{\partial z_1} \frac{\partial z_1}{\partial w_5}$

$$= (y_1 - d_1) y_1 (1 - y_1) h_1$$

- Other weights in the second layer can be obtained by using the same approach

Multi-layer back propagation

- How about weights in the first (hidden) layer
- Use w_1 as an example: $\frac{\partial \varepsilon}{\partial w_1} = \frac{\partial \varepsilon_1}{\partial w_1} + \frac{\partial \varepsilon_2}{\partial w_1}$
- We know (again by chain rule)

$$\frac{\partial \varepsilon_1}{\partial w_1} = \frac{\partial \varepsilon_1}{\partial y_1} \frac{\partial y_1}{\partial z_1} \frac{\partial z_1}{\partial h_1} \frac{\partial h_1}{\partial q_1} \frac{\partial q_1}{\partial w_1}$$

and

$$\frac{\partial \varepsilon_2}{\partial w_1} = \frac{\partial \varepsilon_2}{\partial y_2} \frac{\partial y_2}{\partial z_2} \frac{\partial z_2}{\partial h_1} \frac{\partial h_1}{\partial q_1} \frac{\partial q_1}{\partial w_1}$$

Multi-layer back propagation

- Note that we can reuse partial results in weights updating in back propagation
- Observe the following equations

$$\frac{\partial \varepsilon_1}{\partial w_5} = \boldsymbol{\frac{\partial \varepsilon_1}{\partial y_1} \frac{\partial y_1}{\partial z_1}} \frac{\partial z_1}{\partial w_5}$$

$$\frac{\partial \varepsilon_1}{\partial w_1} = \boldsymbol{\frac{\partial \varepsilon_1}{\partial y_1} \frac{\partial y_1}{\partial z_1}} \frac{\partial z_1}{\partial h_1} \frac{\partial h_1}{\partial q_1} \frac{\partial q_1}{\partial w_1}$$

Multi-layer back propagation

- With the understanding of our example, you should be able to appreciate the "full" comprehensive BP equations given in the literature
- Notice that with more and more layers, the delta weight contains more and more terms, and thus, gets smaller and smaller
- That is one problem when training deep neural networks (i.e., networks with many layers)

Automatic differentiation

- What if we want to write a "universal" program (like TensorFlow) to deal with all types of loss function
- One possible solution is **automatic differentiation** by dividing derivatives into many steps
- For further info, refer to: Automatic differentiation in machine learning: a survey (https://arxiv.org/abs/1502.05767)