北京大學
PEKING UNIVERSITY

Deep Neural Network Foundation

Hao Dong

2019, Peking University

Deep Neural Network Foundation

- Single Neuron
- Activation Functions
- Multi-layer Perceptron
- Loss Functions
- Optimization
- Regularization
- Implementation

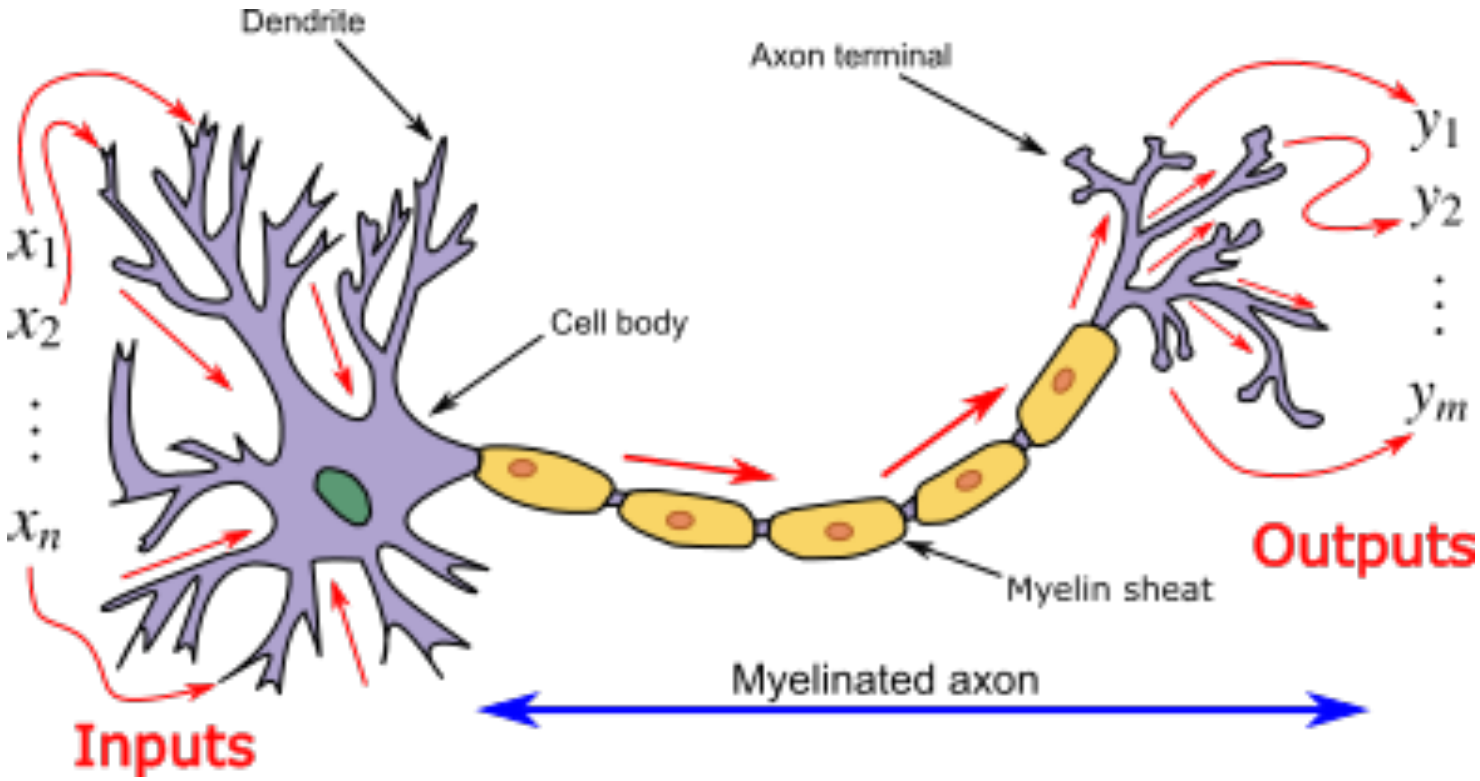
Goal: Understand the basic knowledge of deep neural networks

北京大學
PEKING UNIVERSITY

Single Neuron

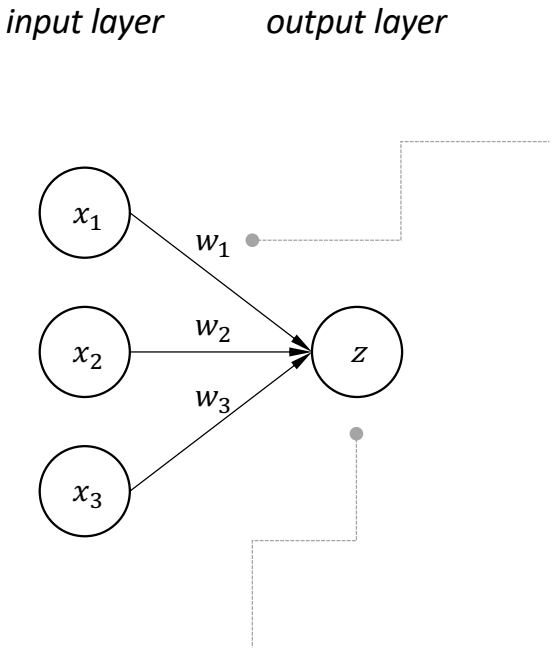
Single Neuron

- Motivation



Single Neuron

- 3 inputs and 1 output



A single neuron

The output is a linear combination of the inputs

$$z = x_1w_1 + x_2w_2 + x_3w_3$$

- A larger absolute value of the weight means the output more sensitive to the specific input

For example, z may be a score determining if we are to play football. If z is large, then we play. To determinate this score, x_1 represents the weather, $x2$ is the expense of the football field rental, and $x3$ is the distance to the field. These inputs are considered the features w.r.t the output. If the whether is the most critical factor, then we can set w_1 to a large positive value and set w_2 and w_3 to smaller positive values. If any w is set to zero, then the corresponding input feature is ignored.

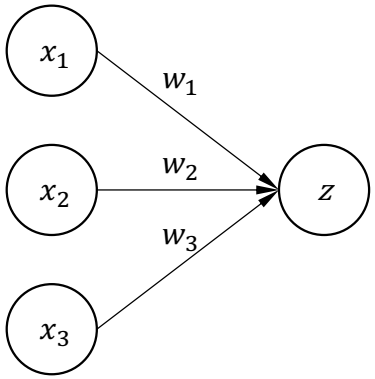
Single Neuron

- 3 inputs and 1 output

The output is a linear combination of the inputs

$$z = x_1 w_1 + x_2 w_2 + x_3 w_3$$

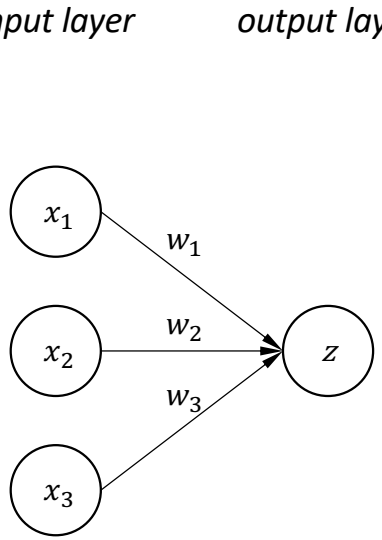
input layer *output layer*



- A single neuron is a network that has only one (output) layer and one output
- As the output connected to all inputs, this layer is called “fully connected layer” or “dense layer”

Single Neuron

- 3 inputs and 1 output



This neuron can be represented by a matrix multiplication

$$z = x_1w_1 + x_2w_2 + x_3w_3$$

column format

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

$$z = \boldsymbol{w}^T\boldsymbol{x}$$

$$z = [w_1 \quad w_2 \quad w_3] \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

row format

$$\boldsymbol{x} = [x_1 \quad x_2 \quad x_3] \quad \boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

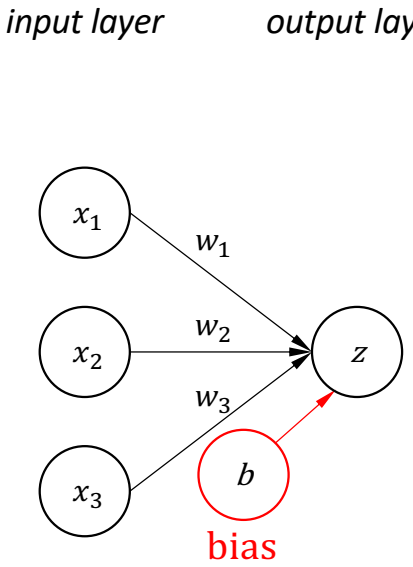
$$z = \boldsymbol{x}\boldsymbol{w}$$

$$z = [x_1 \quad x_2 \quad x_3] \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

Single Neuron

- Bias

A bias value allows the output value to be shifted higher or lower to better fit the input data.



$$z = x_1w_1 + x_2w_2 + x_3w_3 + b$$

column format

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

$$z = \boldsymbol{w}^T\boldsymbol{x} + b$$

$$z = [w_1 \quad w_2 \quad w_3] \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + b$$

row format

$$\boldsymbol{x} = [x_1 \quad x_2 \quad x_3] \quad \boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

$$z = \boldsymbol{x}\boldsymbol{w} + b$$

$$z = [x_1 \quad x_2 \quad x_3] \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} + b$$

Single Neuron

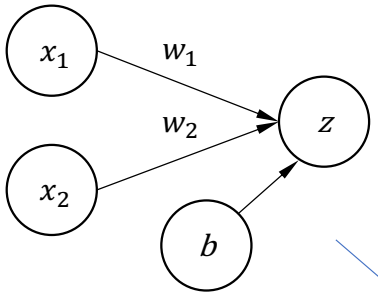
- Classification

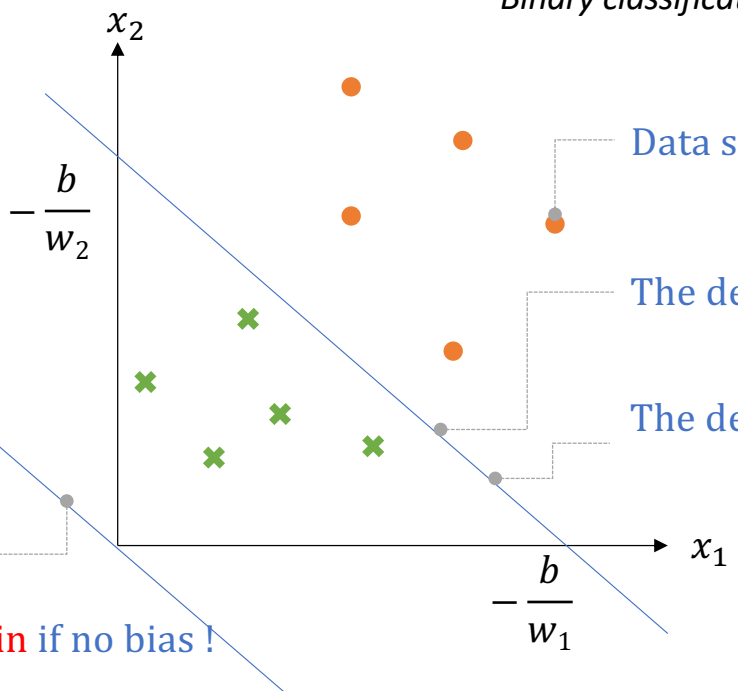
A bias value allows the output value to be shifted higher or lower to better fit the input data.

$$z = x_1 w_1 + x_2 w_2 + b$$

input layer *output layer*

Binary classification: $a = \begin{cases} 0, if\ z \le 0 \\ 1, if\ z > 0 \end{cases}$





Data samples with two features (x_1, x_2)

The decision boundary is a line for $z = 0$

The decision boundary can be shifted left or right via the bias

The decision boundary must cross the origin if no bias !

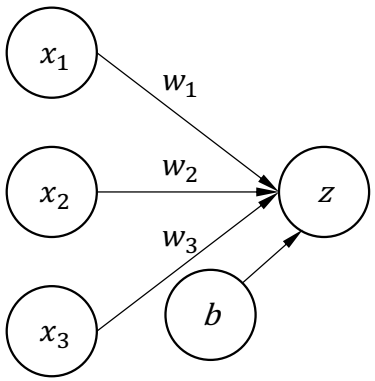
Single Neuron

- Classification

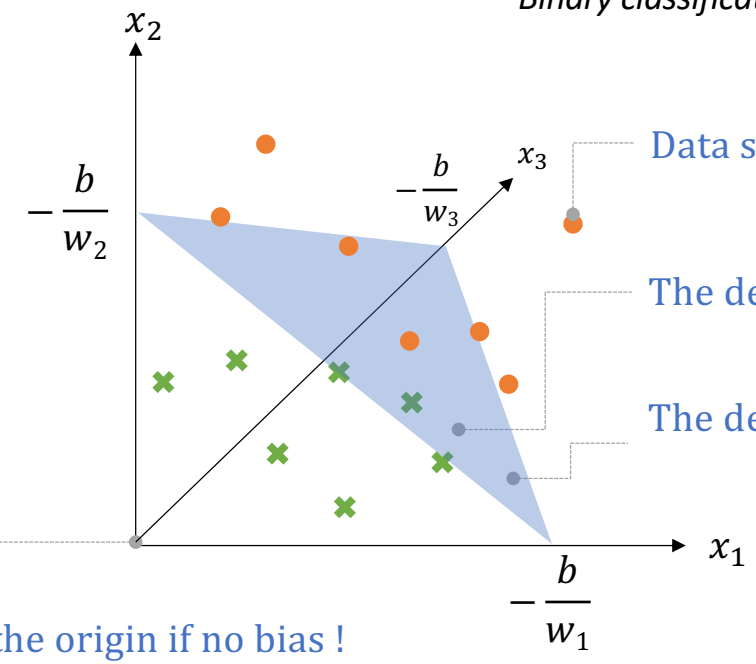
A bias value allows the output value to be shifted higher or lower to better fit the input data.

$$z = x_1w_1 + x_2w_2 + x_3w_3 + b$$

input layer *output layer*



Binary classification: $a = \begin{cases} 0, if\ z \le 0 \\ 1, if\ z > 0 \end{cases}$



Data samples with three features (x_1, x_2, x_3)

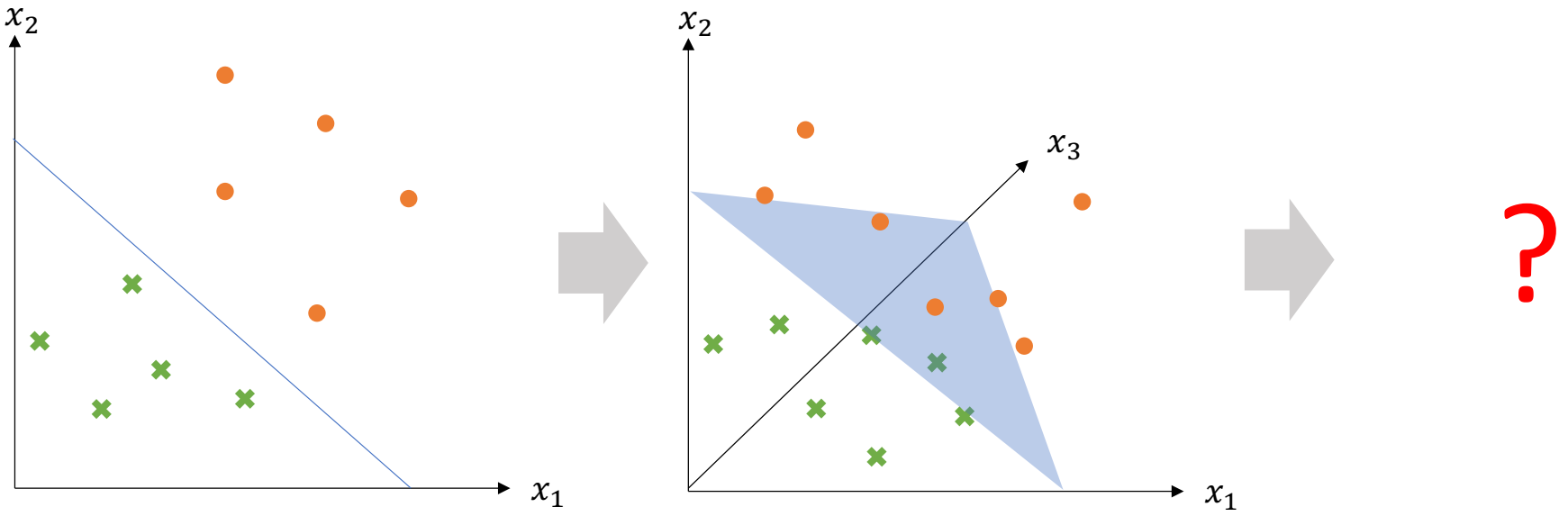
The decision boundary is a surface for $z = 0$

The decision boundary can be shifted left or right via the bias

The decision boundary must cross the origin if no bias !

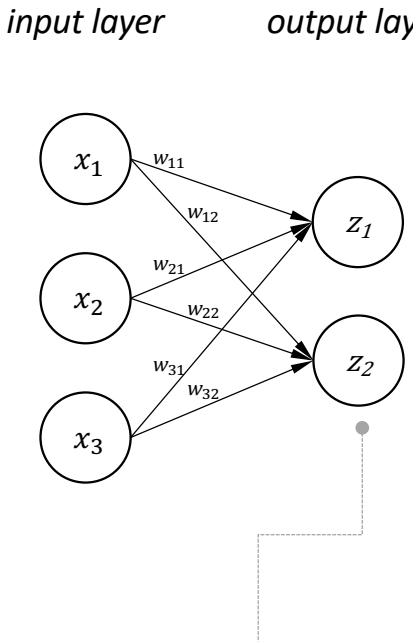
Single Neuron

- Classification
  - Two input features: Decision boundary is a line
  - Three input features: Decision boundary is a surface
  - Many input features: Decision boundary is a hyperplane or hypersurface



Single Neuron

- 3 inputs and **2** outputs



Multiple neurons

Expanding from a single neuron, a network can have multiple outputs

(The biases are not drawn to simplify the explanation)

$$z_1 = x_1w_{11} + x_2w_{21} + x_3w_{31} + b_1$$
$$z_2 = x_1w_{12} + x_2w_{22} + x_3w_{32} + b_2$$

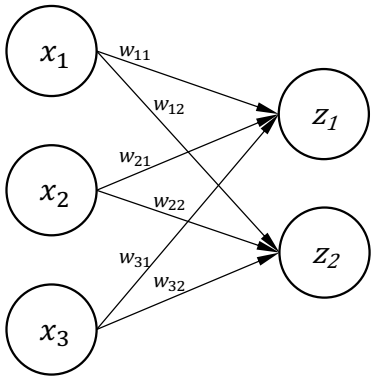
By using multiple neurons, we can obtain multiple outputs. For example, the outputs can represent the scores if we should play football or basketball.

They are all linear!

Single Neuron

- 3 inputs and 2 outputs

input layer *output layer*



$$z_1 = x_1w_{11} + x_2w_{21} + x_3w_{31}$$
$$z_2 = x_1w_{12} + x_2w_{22} + x_3w_{32}$$

column format

$$\boldsymbol{z} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\boldsymbol{W} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

$$\boldsymbol{z} = \boldsymbol{W}^T\boldsymbol{x}$$

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

row format

$$\boldsymbol{z} = [z_1 \quad z_2] \qquad \boldsymbol{x} = [x_1 \quad x_2 \quad x_3]$$

$$\boldsymbol{W} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

$$\boldsymbol{z} = \boldsymbol{x}\boldsymbol{W}$$

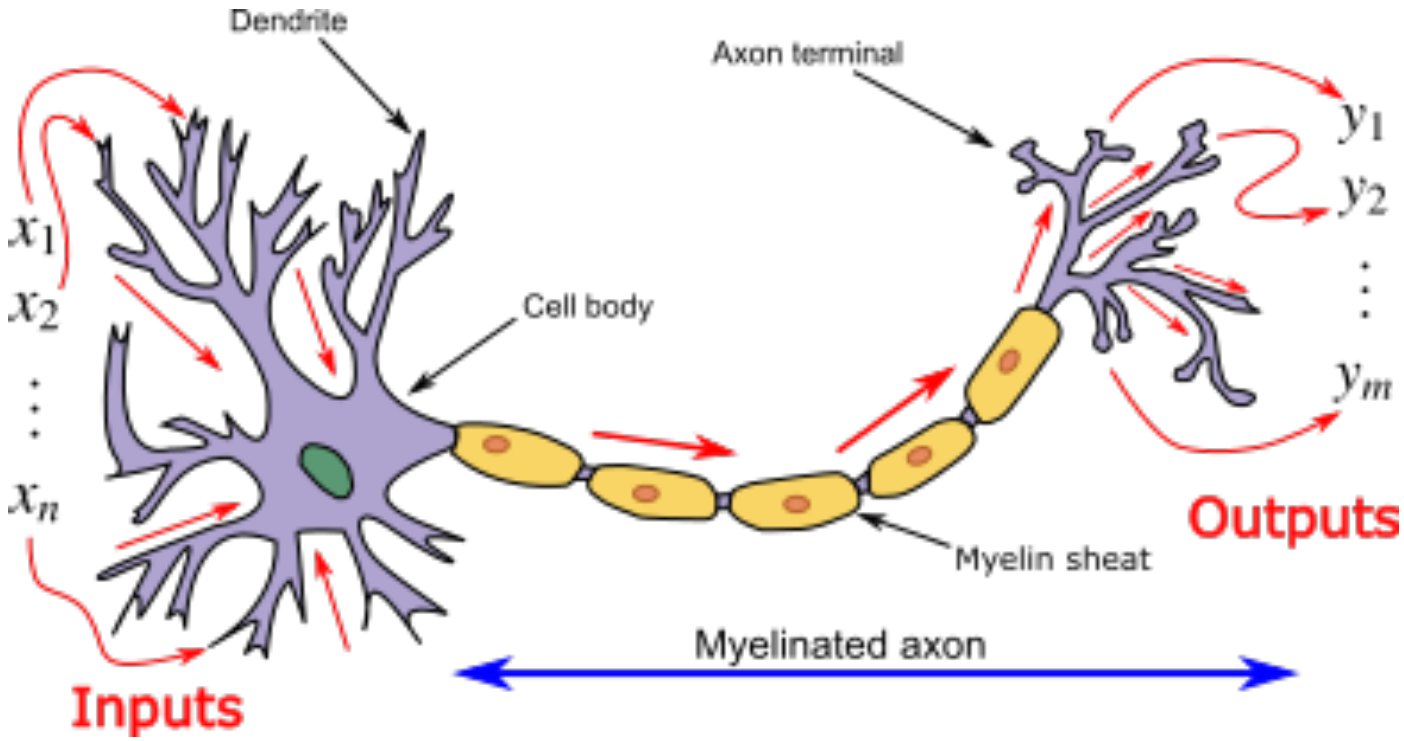
$$[z_1 \quad z_2] = [x_1 \quad x_2 \quad x_3] \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

Activation

Activation

- Motivation

Activation functions provide the non-linearity on the layers outputs,
and their design remains an active researched area.



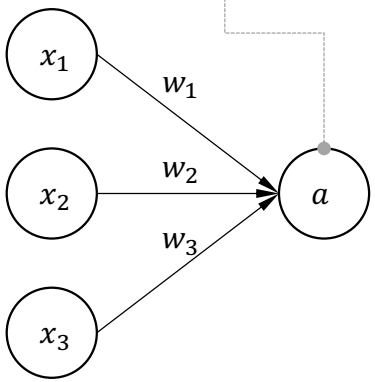
Activation

- Sigmoid or logistic function

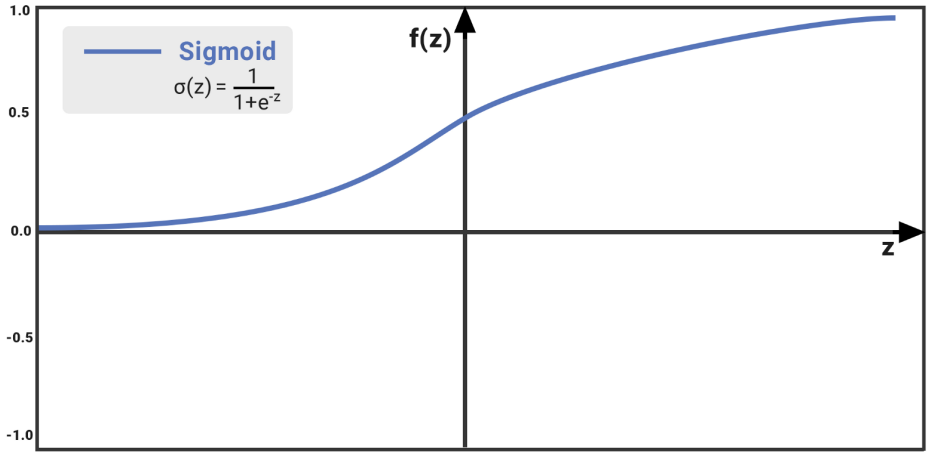
Continuing with our example, for a given neural network, the output can represent specific scores, such as the probability of playing football. To represent the probability from 0% to 100%, it is of common practice to apply a function to scale the output to a value between 0 to 1.

$$a = f(z) = f(x_1 w_1 + x_2 w_2 + b)$$

activation value



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

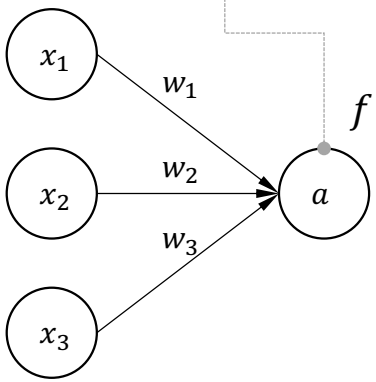


Activation

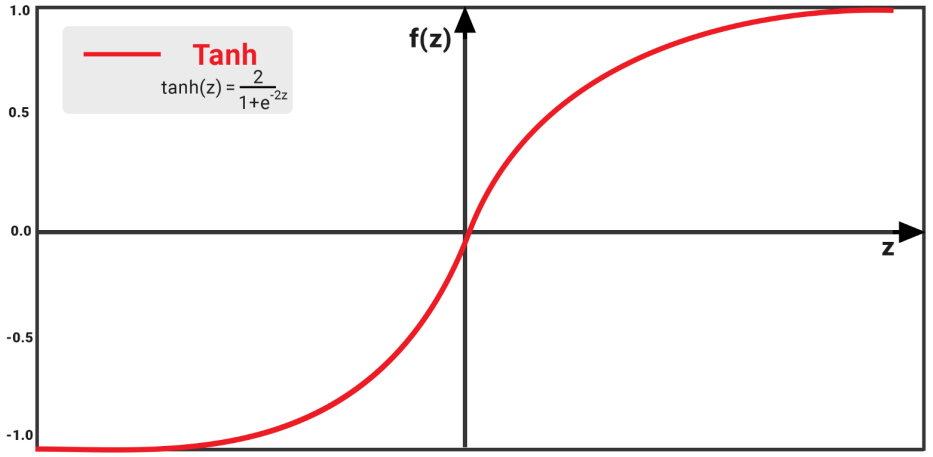
- Tanh

Similar to the sigmoid, the hyperbolic tangent (tanh) also scales the output layer to a limited range of values. With an output range between -1 to 1, this function is often used for regression, such as for an output image with pixel values between -1 to 1.

$$a = f(z) = f(x_1 w_1 + x_2 w_2 + b)$$



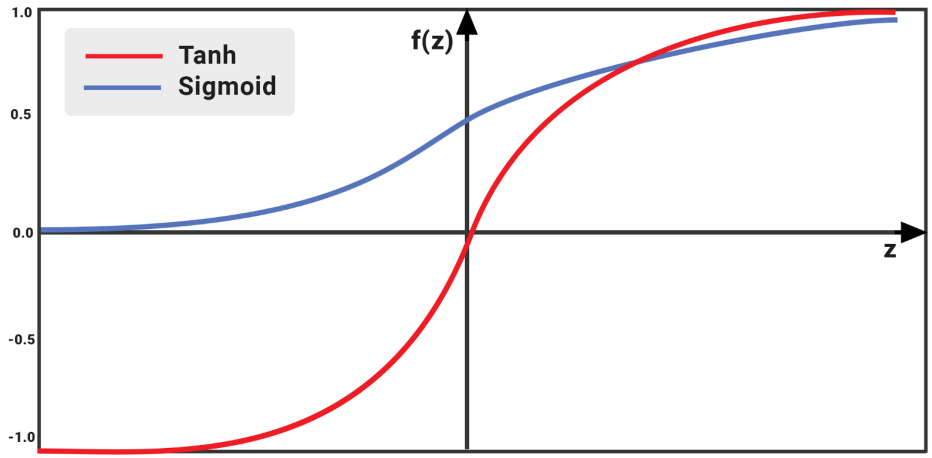
$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{2}{1 + e^{-2z}}$$



Activation

- Sigmoid vs Tanh

The sigmoid and hyperbolic tangent functions are used to provide non-linearity to the network.

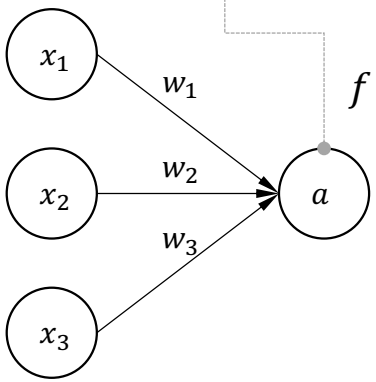


Activation

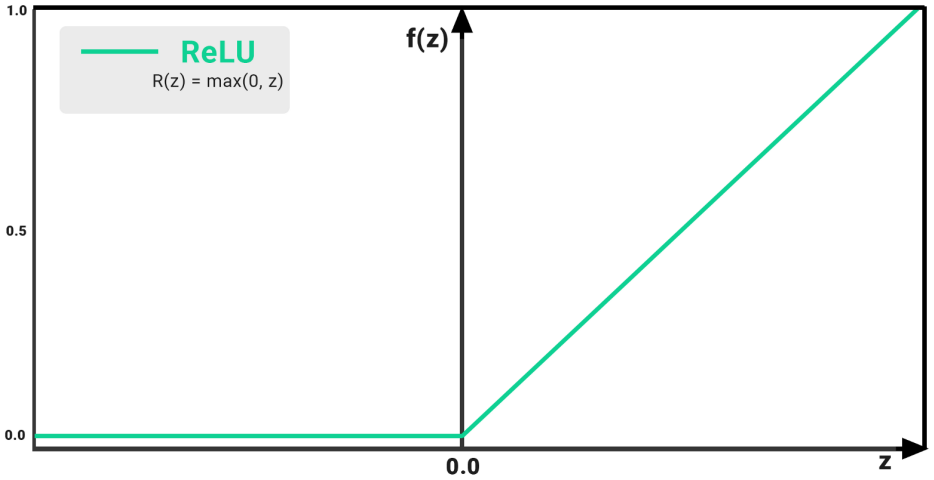
- Rectifier, Rectified linear unit (ReLU)

A function that sets the negative values to zero for the purpose of feature selection and provide a simple way to compute the derivative which will be used in the optimization.

$$a = f(z) = f(x_1 w_1 + x_2 w_2 + b)$$



$$f(z) = \max(0, z) = \begin{cases} 0, if\ x \leq 0 \\ z, if\ x > 0 \end{cases}$$

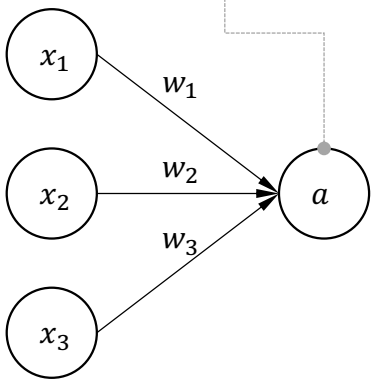


Activation

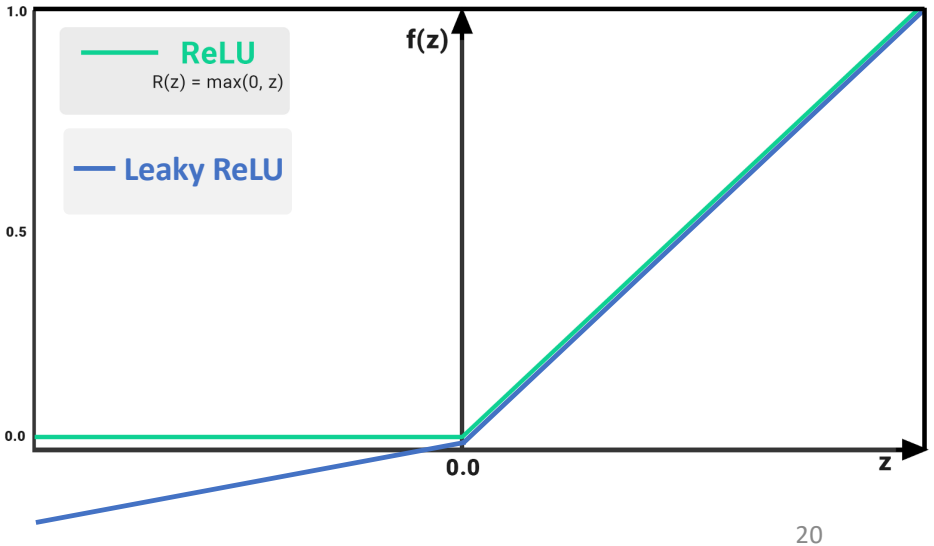
- Leaky ReLU and Parametric Leaky ReLU

However, merely setting negative values to zero will lead to information loss. A solution was proposed with the leaky ReLU where α is a small positive value to control the slope (e.g., 0.1 and 0.2) so that the information from the negative values can pass through to the output. In addition, the parametric ReLU (PReLU) was also proposed to consider α as a network parameter.

$$a = f(z) = f(x_1 w_1 + x_2 w_2 + b)$$



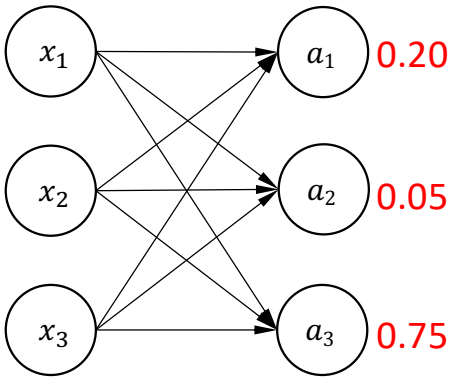
$$f(z) = \begin{cases} \alpha z, if\ x \leq 0 \\ z\ \ , if\ x > 0 \end{cases}$$



Activation

- Softmax

An network with three outputs and three inputs, with these multiple outputs, multi-class classification can be performed, i.e., classify the input into one of three or more classes. The Softmax function is designed for the output layer of a multi-class classifier to not only limit all outputs to 0 to 1, as with the sigmoid, but also ensure the sum of each output equals 1, i.e., the sum of all probabilities must be 100%.

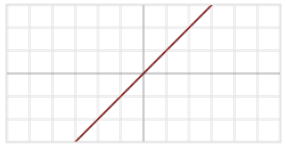


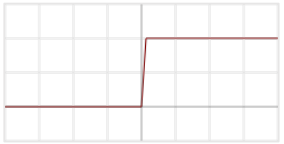
Give a output vector $\boldsymbol{z} = [z_1, z_2, z_3]$ then $K = 3$ is the vector length, we obtain an activation vector $\boldsymbol{a} = [a_1, a_2, a_3]$ as follow

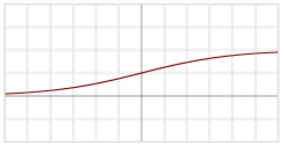
$$\boldsymbol{a}_i = f(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$

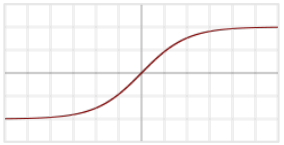
Softmax first applies an exponential function to each output and then normalizes each by dividing it by the sum of all outputs.

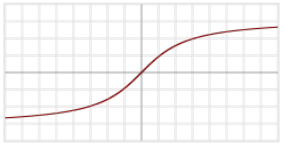
Activation

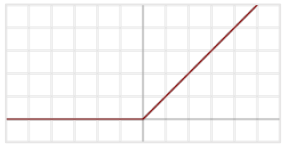
Identity 

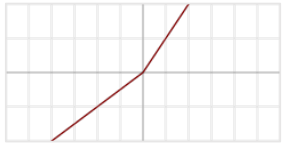
Binary Step 

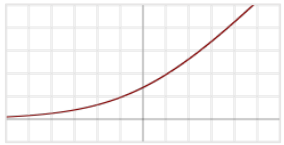
Sigmoid, Logistic 

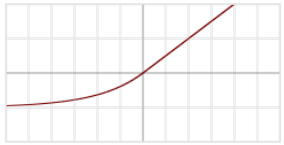
TanH 

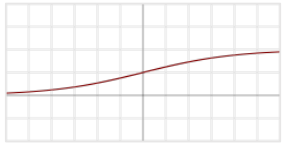
ArcTan 

ReLU 

Leaky ReLU 

Softplus 

ELU 

Softmax 

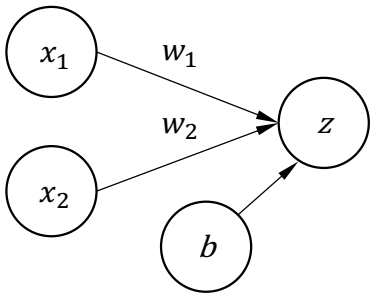
Reference: https://en.wikipedia.org/wiki/Activation_function

北京大學
PEKING UNIVERSITY

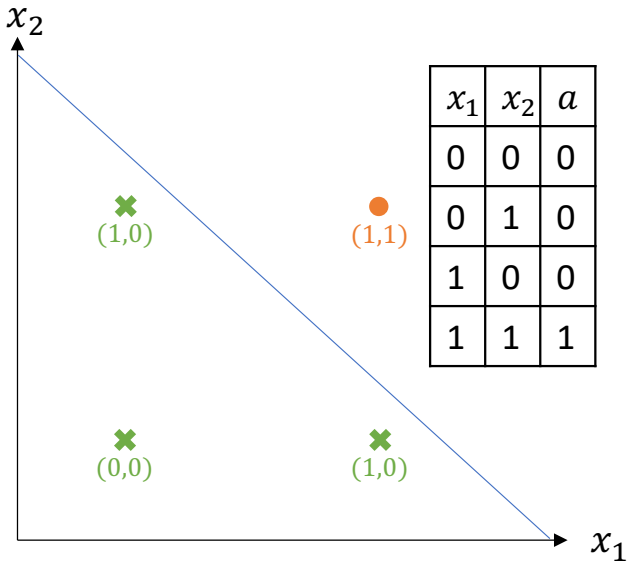
Multi-layer Perceptron

Multi-layer Perceptron

- Motivation : XOR Classification Problem

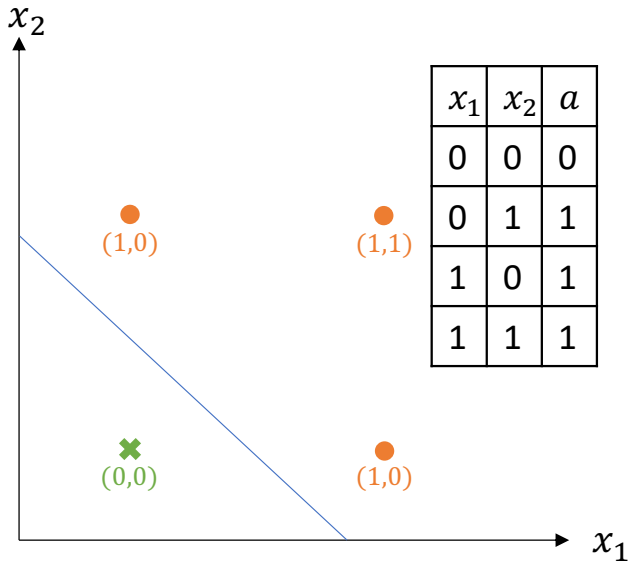


$$z = x_1 w_1 + x_2 w_2 + b$$



x_1	x_2	a
0	0	0
0	1	0
1	0	0
1	1	1

AND

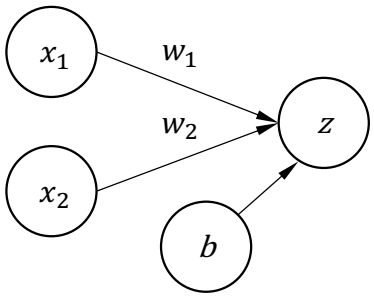


x_1	x_2	a
0	0	0
0	1	1
1	0	1
1	1	1

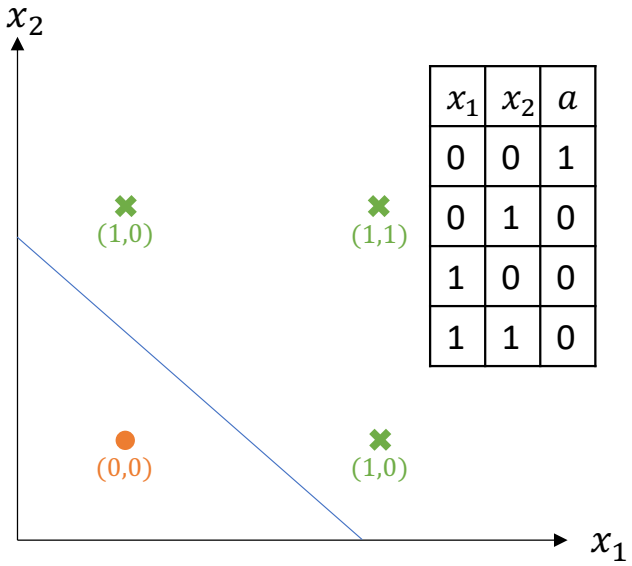
OR

Multi-layer Perceptron

- Motivation : XOR Classification Problem

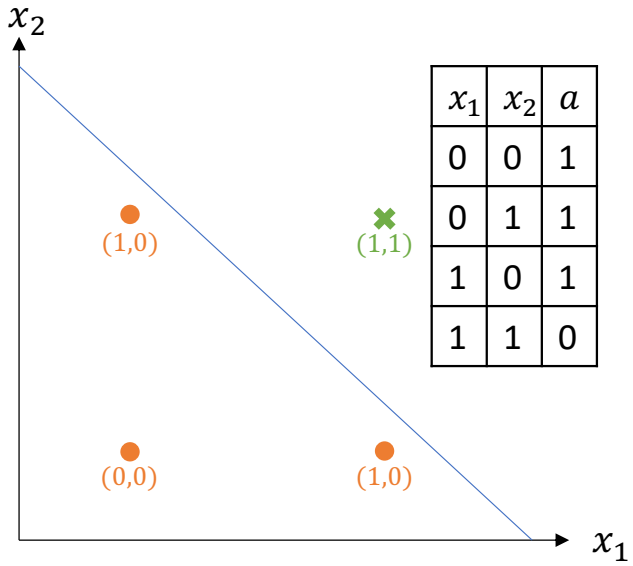


$$z = x_1 w_1 + x_2 w_2 + b$$



x_1	x_2	a
0	0	1
0	1	0
1	0	0
1	1	0

NOR

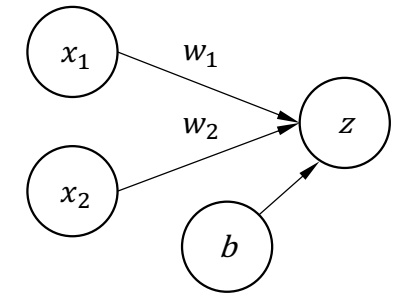


x_1	x_2	a
0	0	1
0	1	1
1	0	1
1	1	0

NAND

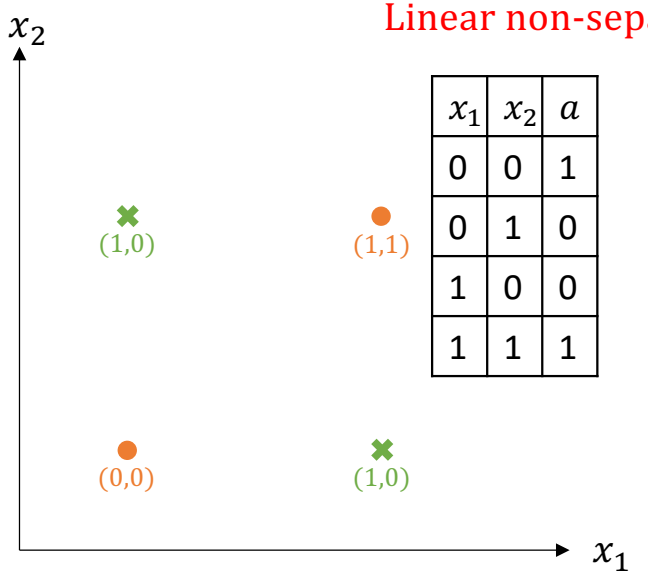
Multi-layer Perceptron

- Motivation : XOR Classification Problem



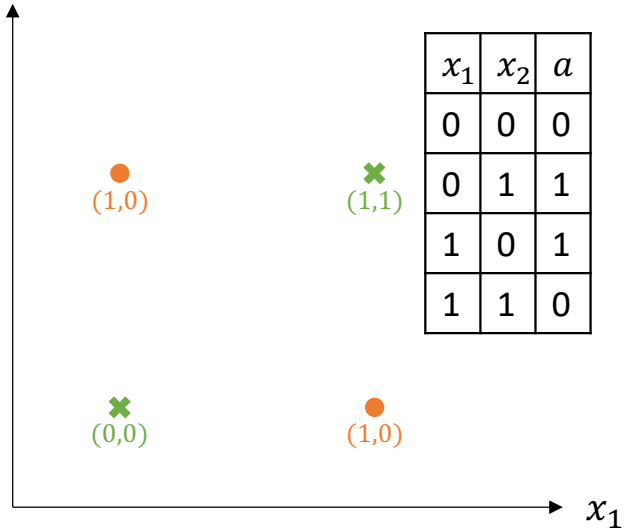
$$z = x_1 w_1 + x_2 w_2 + b$$

Cannot find a line to segment the data points
Linear non-separable problems



x_1	x_2	a
0	0	1
0	1	0
1	0	0
1	1	1

XNOR

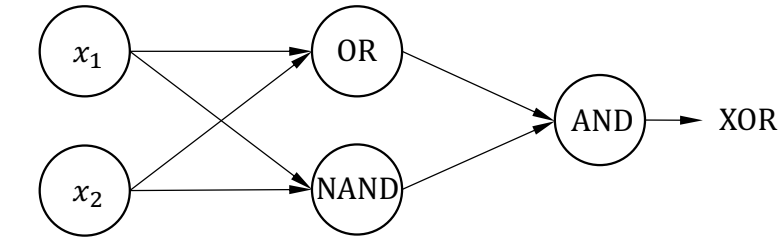


x_1	x_2	a
0	0	0
0	1	1
1	0	1
1	1	0

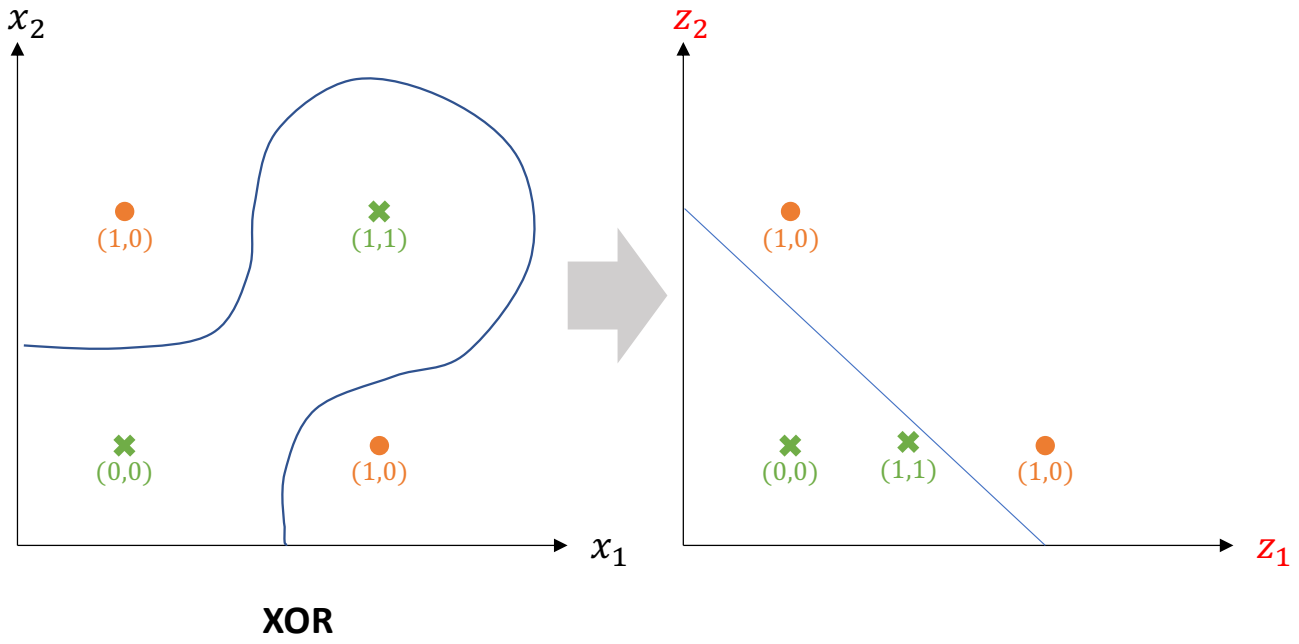
XOR

Multi-layer Perceptron

- XOR Classification Problem



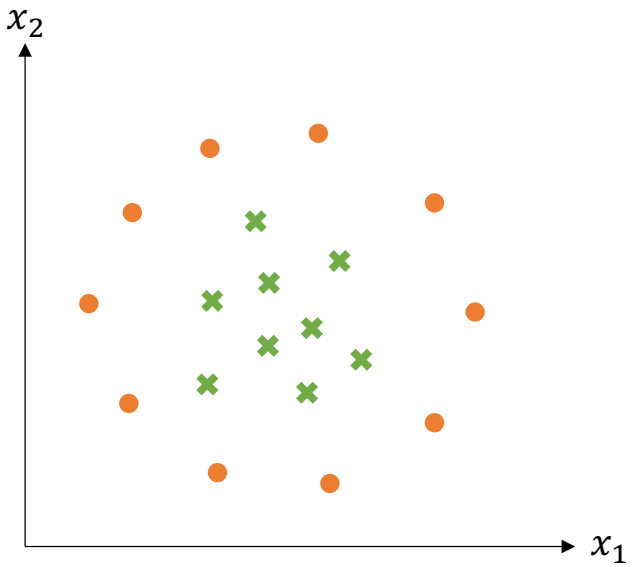
Learning from the extracted features



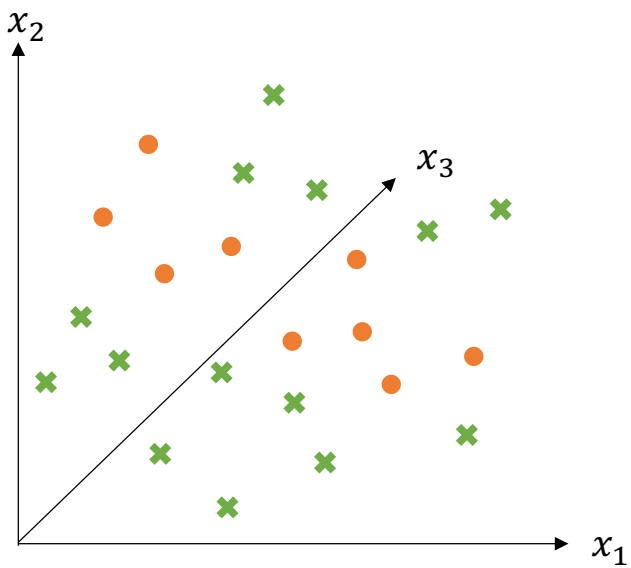
Multi-layer Perceptron

- More Complex Problems

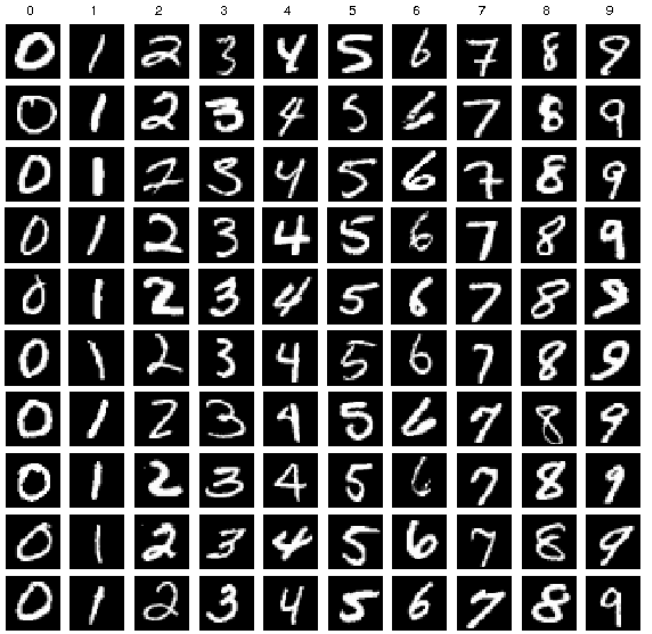
We need a network model with higher capacity



$\boldsymbol{x} = [x_1, x_2]$



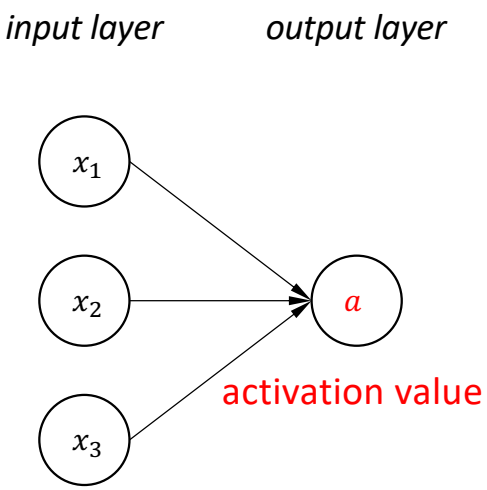
$\boldsymbol{x} = [x_1, x_2, x_3]$



$\boldsymbol{x} = [x_1, x_2, x_3, \dots, x_{784}]$

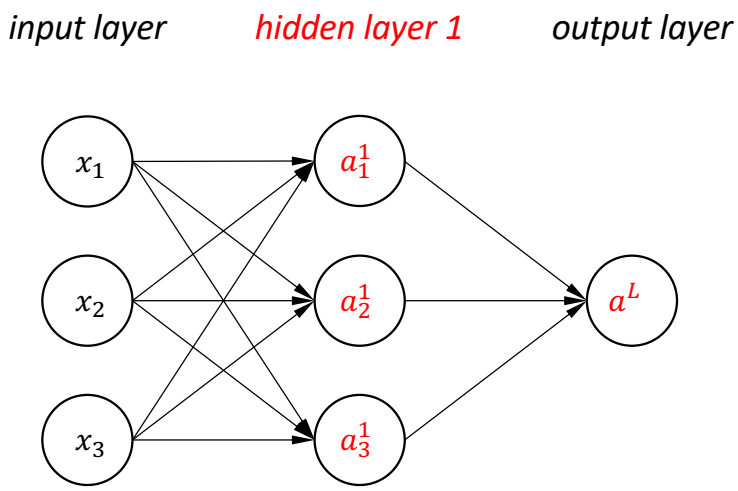
Multi-layer Perceptron

- Representation Capacity



Multi-layer Perceptron

- Representation Capacity



A multi-layer perceptron (MLP) extends from a single, fully connected layer, and consists of at least two fully connected layers.

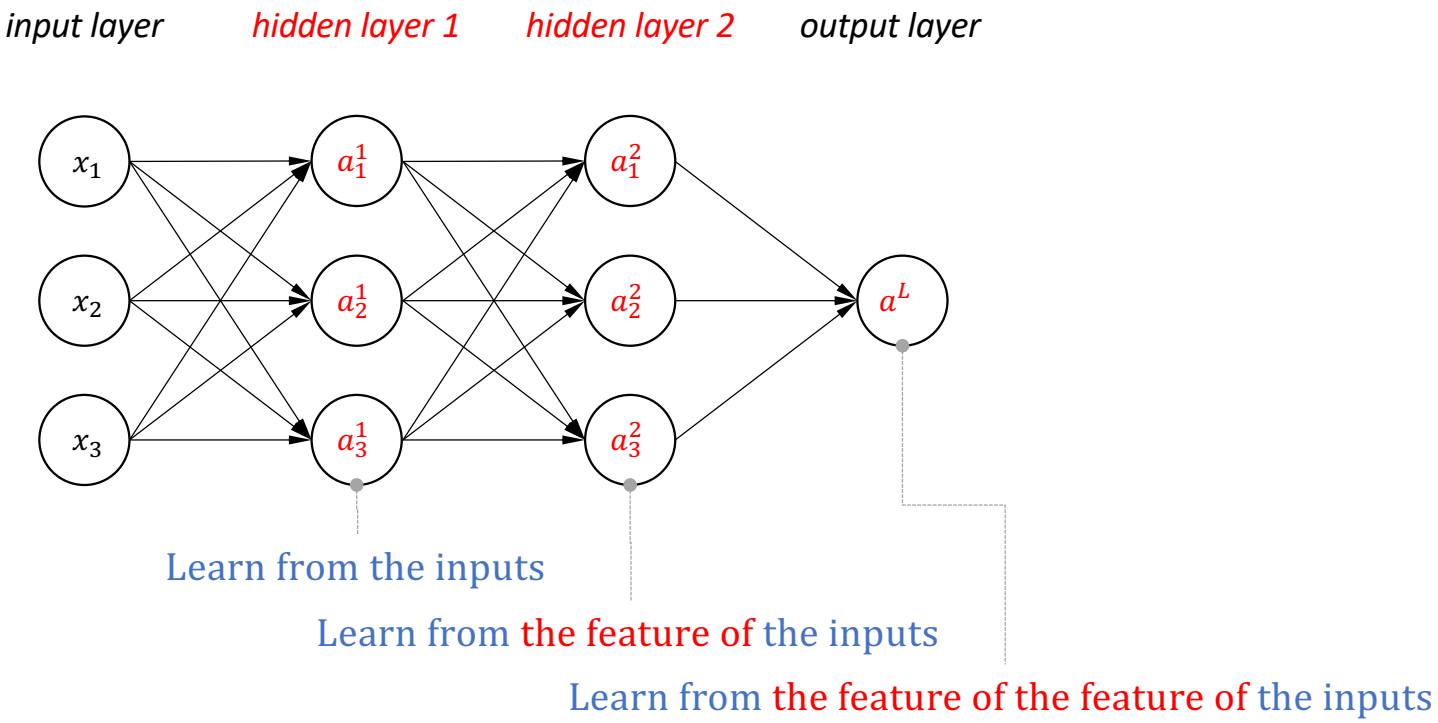
The layers between the inputs and outputs are called "hidden" because they cannot be directly accessed from outside the network.

By stacking a new layer on top of an existing layer, the new layer is considered to use the output of the previously existing layer as its input features. Therefore, compared with a single fully connected layer, MLP can fit more complex input data. In other words, MLP can have more representational capability than a single layer.

The biases are not drawn to simplify the figure.

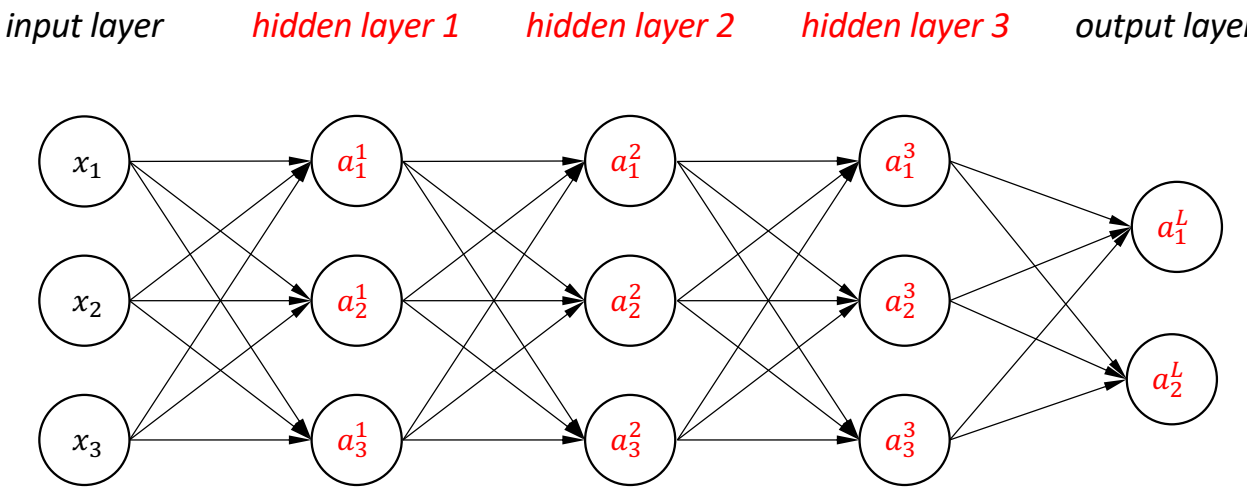
Multi-layer Perceptron

- Representation Capacity



Multi-layer Perceptron

- Representation Capacity: Hierarchical Representation



a_k^l

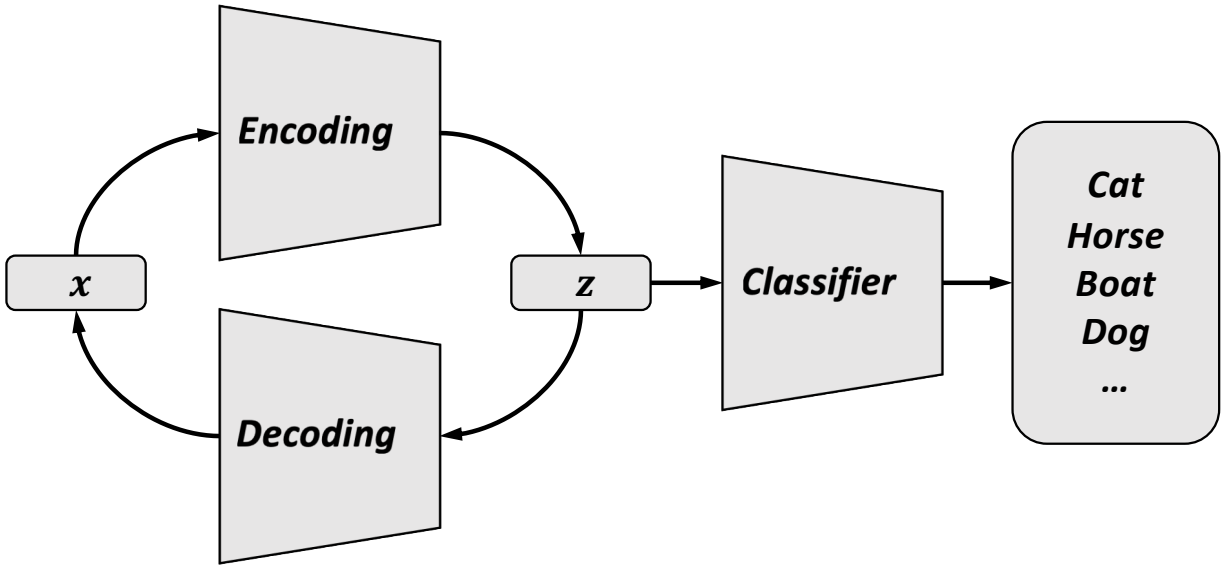
Layer index $l = 1 \ldots L$ from the first hidden layer to the output layer.
The input layer can be written as $\boldsymbol{x} = \boldsymbol{a}^0$.

Output index $k = 1 \ldots K$ where K is the number of units of this layer.

Activation output $\boldsymbol{a}^l = f(\boldsymbol{z}^l)$

Multi-layer Perceptron

- Representation Capacity: Encoding and Decoding



The successful of deep learning is the approximate capacity of the neural network that finds a good latent representation $\boldsymbol{z}$ for the visible input data $\boldsymbol{x}$. In deep learning, the process of transforming visible input data, such as an image, text or video, into a latent representation, alternatively known as embedding or hidden representation, is referred to as “encoding”. The inverted process is referred to as “decoding”.

Loss Functions

Loss Functions

- Motivation

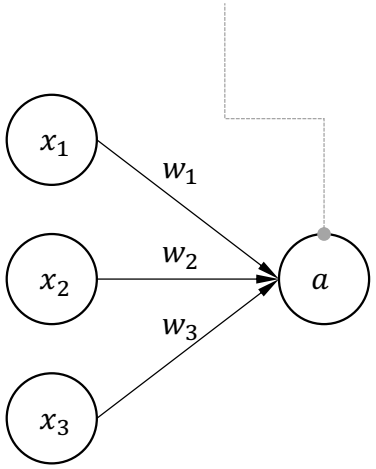
Loss functions are defined to quantify an error, known as the loss value, between the predicted and targeted (i.e., ground truth) outputs. The loss value is used as the goal for optimizing the neural network parameters, such as the weights and biases. Specifically, optimizing a given neural network minimizes the defined loss value by updating the network parameters. Gradient descent is commonly used to update the parameter by computing the partial derivatives of the loss w.r.t the network parameters. Details about gradient descent and neural network training are included in the next Section..

We don't set the values of parameters manually, we define a loss function and use data to optimize the parameters

Loss Functions

- Logistic Regression Loss

$$a = f(z) = f(x_1 w_1 + x_2 w_2 + b)$$



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

The model only has one output

$$\mathcal{L} = y\ log(a) + (1 - y)\log(1 - a)$$

Given M data samples

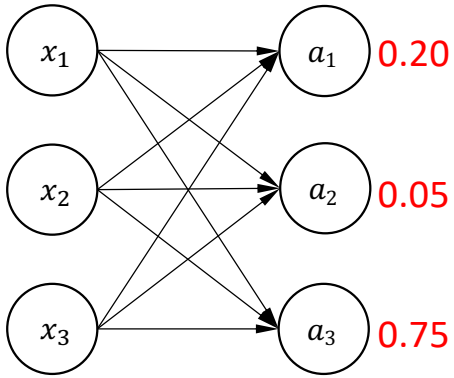
$$\mathcal{L} = \sum_{m=1}^{M} (y^m\ log(a^m) + (1 - ym)\log(1 - am))$$

Loss Functions

- Cross-Entropy Loss

Output a vector

$\boldsymbol{a} = softmax(\boldsymbol{z})$



The model has multiple outputs

$$\mathcal{L} = \sum_{k=1}^{K} \boldsymbol{y}_k \log(\boldsymbol{a}_k)$$

Given $\boldsymbol{M}$ data samples

$$\mathcal{L} = \sum_{m=1}^{M} \boldsymbol{y}_k^m \log(\boldsymbol{a}_k^m)$$

Loss Functions

- $\mathcal{L}_p$ norm

Measure the scale of a vector

$$\|\boldsymbol{x}\|_p = \left(\sum_{k=1}^{K} |\boldsymbol{x}_k|^p\right)^{\frac{1}{p}}$$

$$\|\boldsymbol{x}\|_p^p = \sum_{k=1}^{K} |\boldsymbol{x}_k|^p$$

Measure the difference between two vectors

$$\mathcal{L}_p = \|\boldsymbol{y} - \boldsymbol{a}\|_p^p = \sum_{k=1}^{K} |\boldsymbol{y}_k - \boldsymbol{a}_k|^p$$

Loss Functions

- Mean Squared Error (MSE)

MSE is the $\mathcal{L}_2$ norm over samples and is used for regression problems in which the output of the neural networks contains continuous values, such as the pixels of an image or a scalar value

$$\mathcal{L}_{MSE} = \|\boldsymbol{y} - \boldsymbol{a}\|_2^2$$

Given $\boldsymbol{M}$ data samples

$$\mathcal{L}_{MSE} = \frac{1}{M}\sum_{m=1}^{M}\|\boldsymbol{y}^m - \boldsymbol{a}^m\|_2^2$$

Loss Functions

- Mean Absolute Error (MAE)

MAE is the $\mathcal{L}_1$ norm over samples, known as the least square error, MAE is also used for regression problems and is expressed as follows.

$$\mathcal{L}_{MAE} = \|\boldsymbol{y} - \boldsymbol{a}\|$$

Given $\boldsymbol{M}$ data samples

$$\mathcal{L}_{MAE} = \frac{1}{M}\sum_{m=1}^{M}|\boldsymbol{y}^m - \boldsymbol{a}^m|$$

Loss Functions

- Many many more ….

北京大學
PEKING UNIVERSITY

Optimization

Optimization

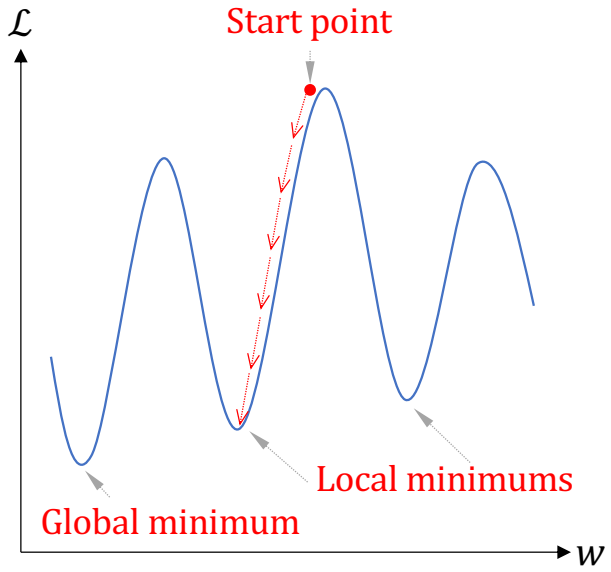
- Motivation

Given a network $f(x;\theta)$ and a loss function $\mathcal{L}$, training the network is the process to minimise the loss value $\mathcal{L}$ by updating the network parameters θ, such as the weights and biases. Gradient descent is one method to update the network parameters. Even though there are some other optimisation methods exist, such as limited memory BFGS (L-BFGS) and conjugate gradient (CG), due to the drawbacks related to larger computation requirements, they are not often applied.

Given $f(x;\theta)$ and $\mathcal{L}$ find a good θ

Optimization

- Gradient Descent

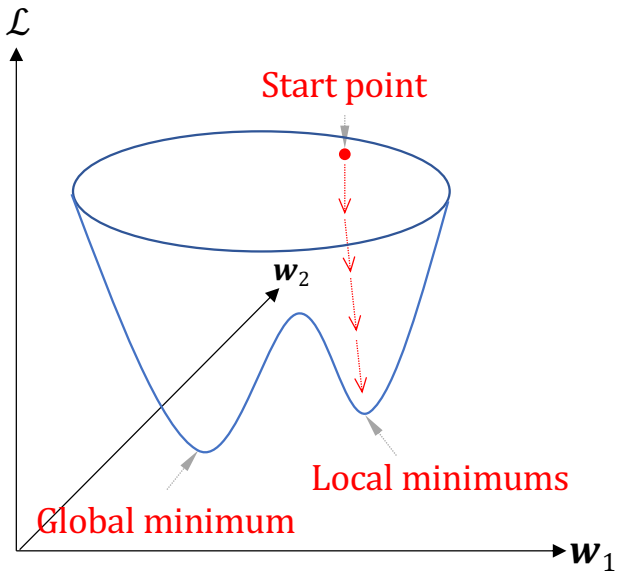


Assuming one parameter only

$$w := w - \alpha \frac{d\mathcal{L}}{dw}$$

Optimization

- Gradient Descent



Assuming two parameters

$$w_j := w_j - \alpha \frac{\partial \mathcal{L}}{\partial w_j} \qquad w = [w_1, w_2]$$

Optimization

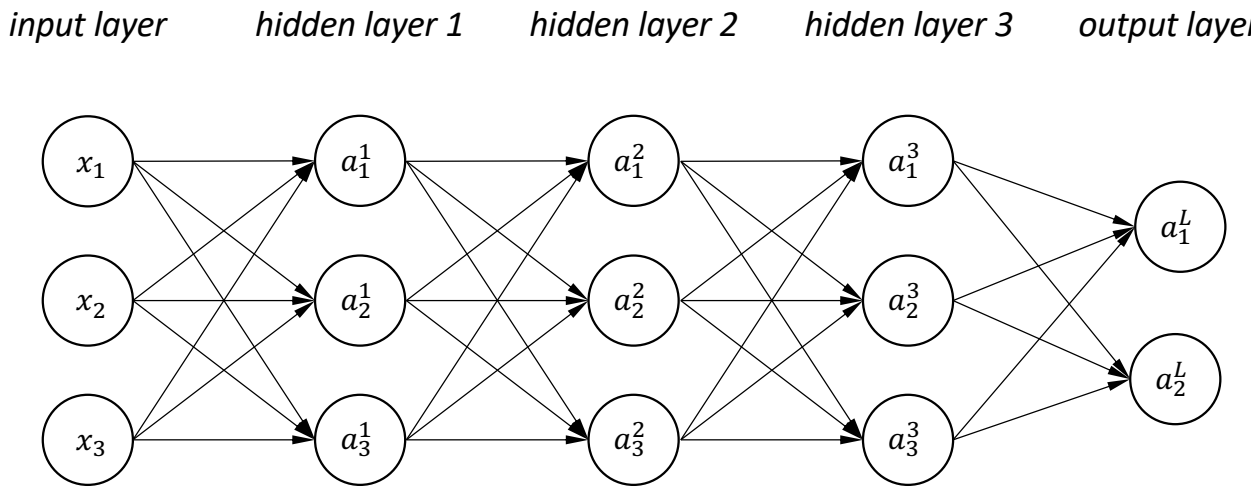
- Gradient Descent

All we need: $\boldsymbol{\theta} \coloneqq \boldsymbol{\theta} - \alpha \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$

Optimization

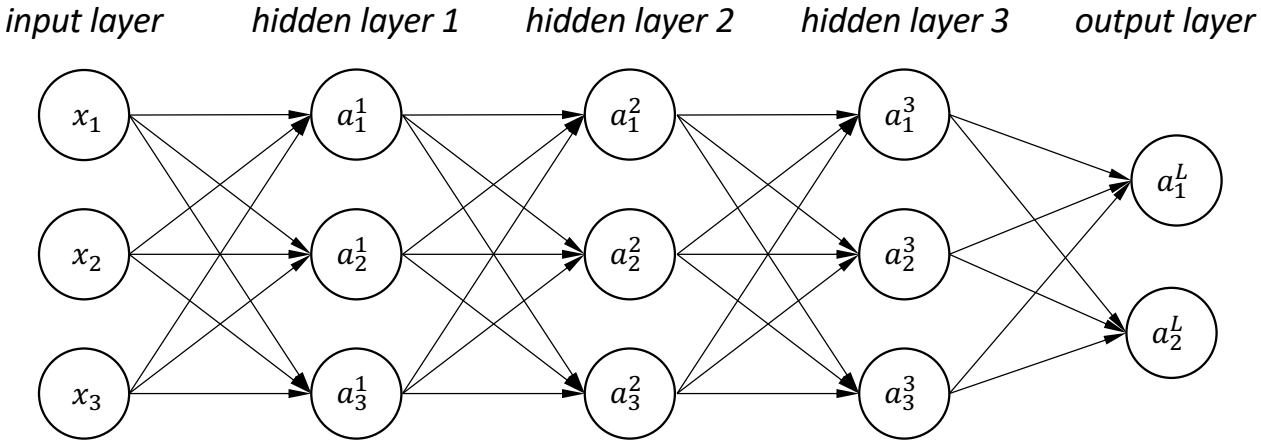
- Error Back-Propagation

The process of error back-propagation computes the gradients $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ for every parameters in the network. When computing the gradient, an intermediate result $\delta = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}}$ is introduced, which is the ∂z partial derivative of the loss $\mathcal{L}$ w.r.t the layer's output z. Based on this intermediate result, the process next computes the partial derivative of the loss $\mathcal{L}$ w.r.t every parameters $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ which is then used to update the parameter values.



Optimization

- Error Back-Propagation



Layer index $l = 1 \ldots L$ from the first hidden layer to the output layer.
The input layer can be written as $\boldsymbol{x} = \boldsymbol{a}^0$.

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

We use this model and loss as an example:

$$\boldsymbol{z}^l = \boldsymbol{W}^{l^T} \boldsymbol{a}^{l-1} + \boldsymbol{b}^l$$

$$\mathcal{L} = \frac{1}{2}(\boldsymbol{y} - \boldsymbol{a}^L)^2$$

Optimization

- Error Back-Propagation: Vectors in Column Format

1. Given:

- $\boldsymbol{a}^l = f(\boldsymbol{z}^l) = \frac{1}{1+e^{-z^l}}$
- $\boldsymbol{z}^l = \boldsymbol{W}^{l^T}\boldsymbol{a}^{l-1} + \boldsymbol{b}^l$
- $\mathcal{L} = \frac{1}{2}(\boldsymbol{y} - \boldsymbol{a}^L)^2$

2. Then we can have the following derivatives:

- $\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} = f^{-1}(\boldsymbol{z}^l) = \boldsymbol{a}^l \circ (1 - \boldsymbol{a}^l)$
- $\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^l} = (\boldsymbol{a}^L - \boldsymbol{y})$
- $\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{W}^l} = \boldsymbol{a}^{l-1}$ and $\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{b}^l} = 1$

Hadamard (element-wise) product

3. Then the error of the output laye1

- $\delta^L = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^L} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^L}\frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} = (\boldsymbol{a}^L - \boldsymbol{y}) \circ (\boldsymbol{a}^L \circ (1 - \boldsymbol{a}^L))$

Chain rule !

4. Then the error of the other layers $\boldsymbol{l} = \boldsymbol{1} \ldots \boldsymbol{L} - \boldsymbol{1}$:

- $\delta^l = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{l+1}}\frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{z}^l} = \delta^{l+1}\frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{z}^l}$
  - $\boldsymbol{z}^{l+1} = \boldsymbol{W}^{l+1^T}\boldsymbol{a}^l + \boldsymbol{b}^{l+1}$
  - $\frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{z}^l} = \boldsymbol{W}^{l+1^T} f^{-1}(\boldsymbol{z}^l) = \boldsymbol{W}^{l+1^T} \circ (\boldsymbol{a}^l \circ (1 - \boldsymbol{a}^l))$
- $\delta^l = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{l+1}}\frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{z}^l} = \boldsymbol{W}^{l+1^T}\delta^{l+1} \circ (\boldsymbol{a}^l \circ (1 - \boldsymbol{a}^l))$

5. The the gradients are:

- $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l}\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{W}^l} = \delta^l\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{W}^l} = \delta^l \boldsymbol{a}^{l-1^T}$
- $\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l}\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{b}^l} = \delta^l\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{b}^l} = \delta^l$

6. Update parameters iteratively

$$\boldsymbol{W}^l := \boldsymbol{W}^l - \alpha\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^l} \quad \boldsymbol{b}^l := \boldsymbol{b}^l - \alpha\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^l}$$

Optimization

- Error Back-Propagation: Vectors in **Raw** Format for **Python**

1. Given:

- $\boldsymbol{a}^l = f(\boldsymbol{z}^l) = \frac{1}{1+e^{-\boldsymbol{z}^l}}$
- $\boldsymbol{z}^l = \boldsymbol{a}^{l-1}\boldsymbol{W}^l + \boldsymbol{b}^l$
- $\mathcal{L} = \frac{1}{2}(\boldsymbol{y} - \boldsymbol{a}^L)^2$

2. Then we can have the following derivatives:

- $\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} = f^{-1}(\boldsymbol{z}^l) = \boldsymbol{a}^l \circ (1 - \boldsymbol{a}^l)$
- $\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^l} = (\boldsymbol{a}^L - \boldsymbol{y})$
- $\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{W}^l} = \boldsymbol{a}^{l-1}$ and $\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{b}^l} = 1$

3. Then the error of the output layer

- $\delta^L = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^L} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^L}\frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} = (\boldsymbol{a}^L - \boldsymbol{y}) \circ (\boldsymbol{a}^L \circ (1 - \boldsymbol{a}^L))$

4. Then the error of the other layers $\boldsymbol{l = 1 \ldots L - 1}$:

- $\delta^l = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{l+1}}\frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{z}^l} = \delta^{l+1}\frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{z}^l}$
  - $\boldsymbol{z}^{l+1} = \boldsymbol{a}^l\boldsymbol{W}^{l+1} + \boldsymbol{b}^{l+1}$
  - $\frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{z}^l} = \boldsymbol{W}^{l+1} \circ f^{-1}(\boldsymbol{z}^l) = \boldsymbol{W}^{l+1} \circ (\boldsymbol{a}^l \circ (1 - \boldsymbol{a}^l))$
- $\delta^l = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{l+1}}\frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{z}^l} = \delta^{l+1}\boldsymbol{W}^{l+1^T} \circ (\boldsymbol{a}^l \circ (1 - \boldsymbol{a}^l))$

5. Then the gradients are:

- $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l}\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{W}^l} = \delta^l\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{W}^l} = \boldsymbol{a}^{l-1^T}\delta^l$
- $\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l}\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{b}^l} = \delta^l\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{b}^l} = \delta^l$

6. Update parameters iteratively

$$\boldsymbol{W}^l := \boldsymbol{W}^l - \alpha\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^l} \quad \boldsymbol{b}^l := \boldsymbol{b}^l - \alpha\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^l}$$

Optimization

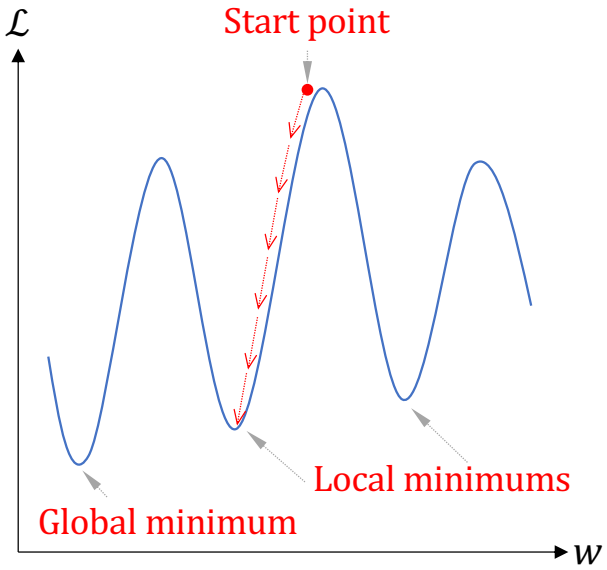
- Error Back-Propagation: Gradient Vanish
  - $\delta^l = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{l+1}} \frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{z}^l} = \delta^{l+1} \boldsymbol{W}^{l+1^T} \circ (\boldsymbol{a}^l \circ (1 - \boldsymbol{a}^l))$

Following the previous example, δ has a term $\left(\boldsymbol{a}^l \circ \left(1 - \boldsymbol{a}^l\right)\right)$ in which if the activation output $\boldsymbol{a}$ is close to 0 or 1, the δ^l is small. In chain rule, $\delta^l = \delta^{l+1}$ … so as the error propagate from the output layer to the input layer, the δ will become more and more small, which will leads to difficult for updating the parameters of the layers on the front (near the input).

- Solution 1: use ReLU to replace Sigmoid function. (commonly used in practice)
- Solution 2: pre-trained the parameters layer-by-layer … Deep Belief Net
- …

Optimization

- Stochastic Gradient Descent (SGD)



Motivation

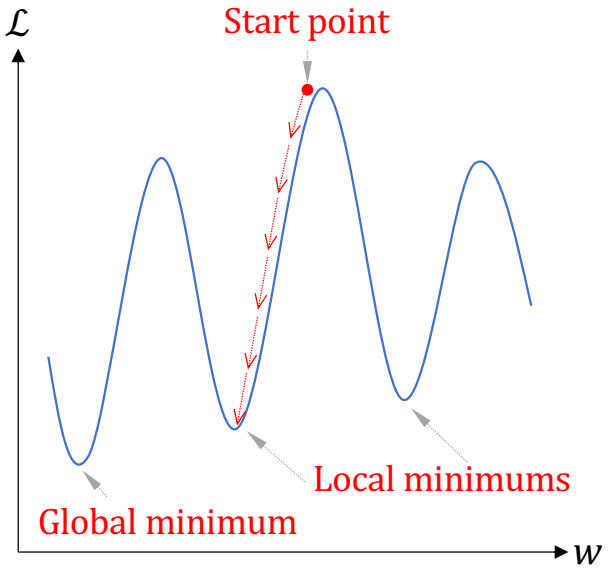
- Error back-propagation updates the parameters iteratively.
- Compute the loss $\mathcal{L}$ using all data samples every time would be SLOW!

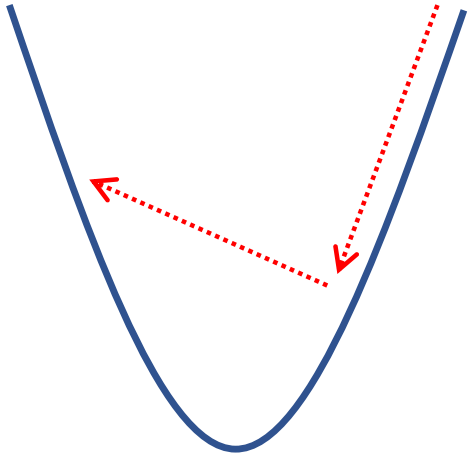
Method

- Randomly select a "batch" of samples from the training set to compute the $\mathcal{L}$.
- These data are called a "mini-batch", and the quantity of data is called the "batch size"
- By updating the parameter multiple times, the mini-batches will cover the entire training set. Epoch represents the mini-batch has looped over the entire training dataset.

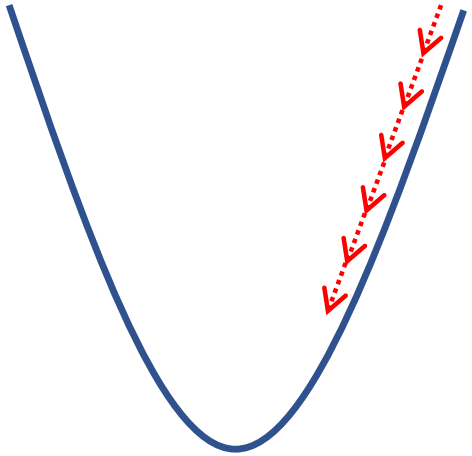
Optimization

- Adaptive Learning Rate





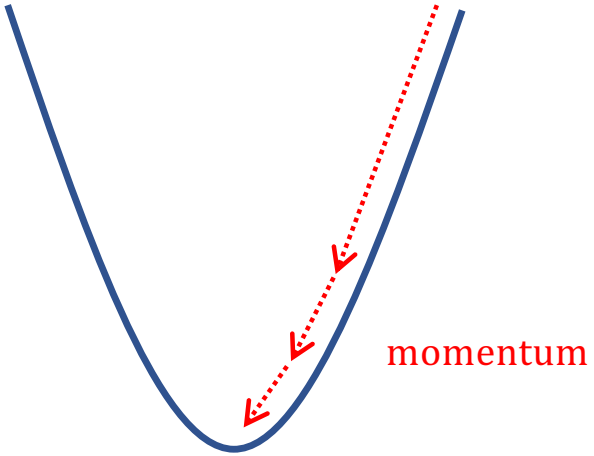
Learning rate too large
Fail to converge into the saddle



Learning rate too small
Very slow

Optimization

- Adaptive Learning Rate

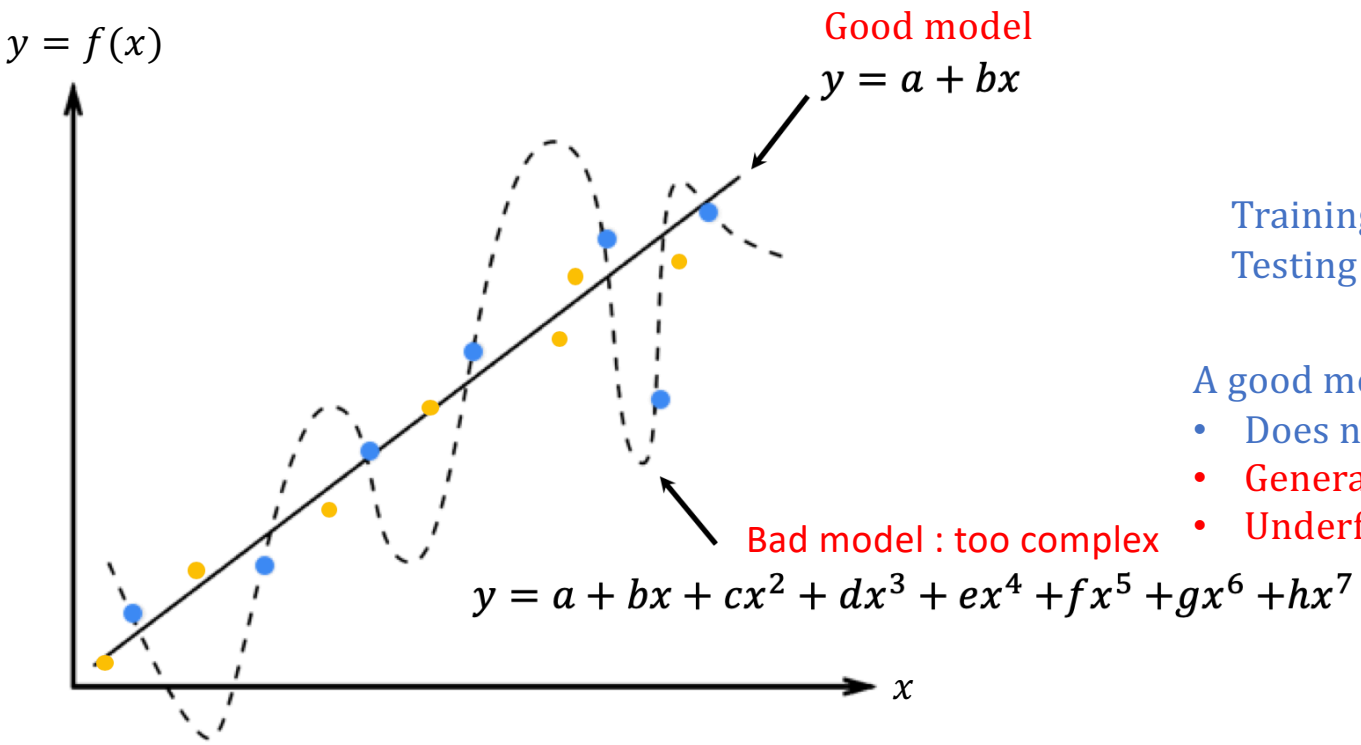


RMSProp, Adagrad, Adam, AMSGrad, AdaBound (PKU Undergraduate)

Optimization

- Hyper-Parameter Selection

Training Data vs Testing Data



- Training data
- Testing data

Training data is used to train the model
Testing data is used to test the model

A good model $f(x;\ \theta)$ (algorithm):

- Does not overfit to training data
- Generalizes well to testing data
- Underfitting can be solved by a deeper model

Optimization

- Hyper-Parameter Selection

Training Data vs Validation Data vs Testing Data

Hyper-parameters are the settings of the models, such as

- The number of layers
- The number of units
- The activation functions
- The loss function
- The batch size
- The number of epochs
- …

DO NOT select the hyper-parameters based on the performance on the testing set
IT IS CHEATING

Training data is used to train the model
Validation data is used to select the hyper-parameters
Testing data is used to test the model

Optimization

- Hyper-Parameter Selection **& Cross Validation**

Training Data vs ~~Validation Data~~ vs Testing Data

For small datasets, splitting the data into training, validating, and testing sets may be challenging. If the size of the training data is too small, then the performance of the model will be impacted. On the other hand, if the test data is too small, then the evaluation may not adequately reflect the performance.

How to select hyper-parameters without a validation set?

Optimization

- Hyper-Parameter Selection **& K-Fold Cross Validation**

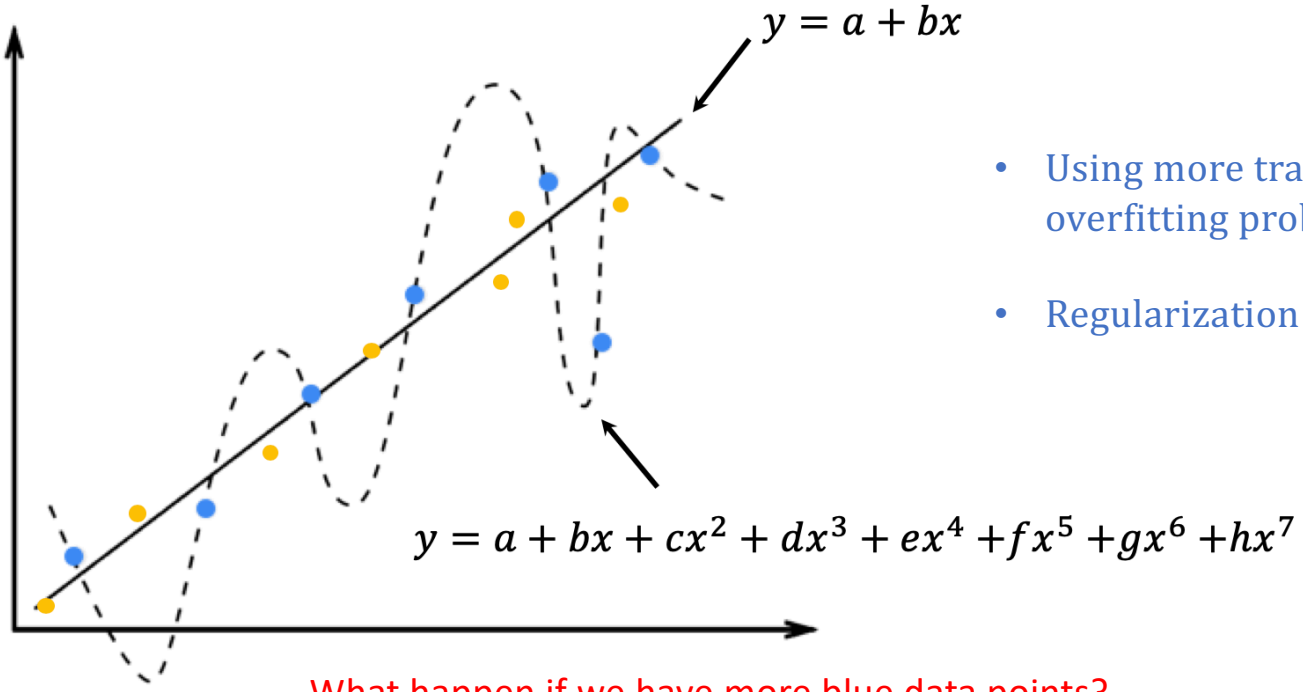
The dataset is separated into K folds of data evenly.

北京大學
PEKING UNIVERSITY

Regularization

Regularization

- Motivation



- Using more training data can help to alleviate the overfitting problem, but it is expensive.
- Regularization is for alleviating the overfitting problem.

What happen if we have more blue data points?

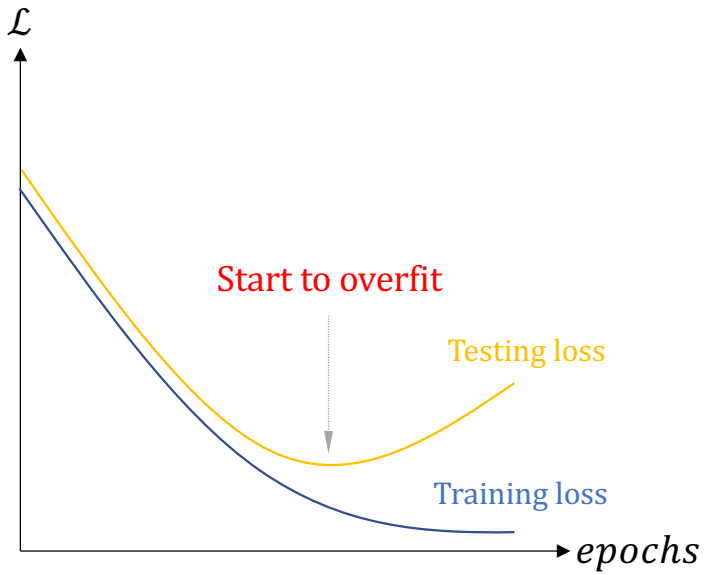
Regularization

- Data Augmentation

Common image data augmentation methods: horizontal flipping, rotating, shifting, and zooming

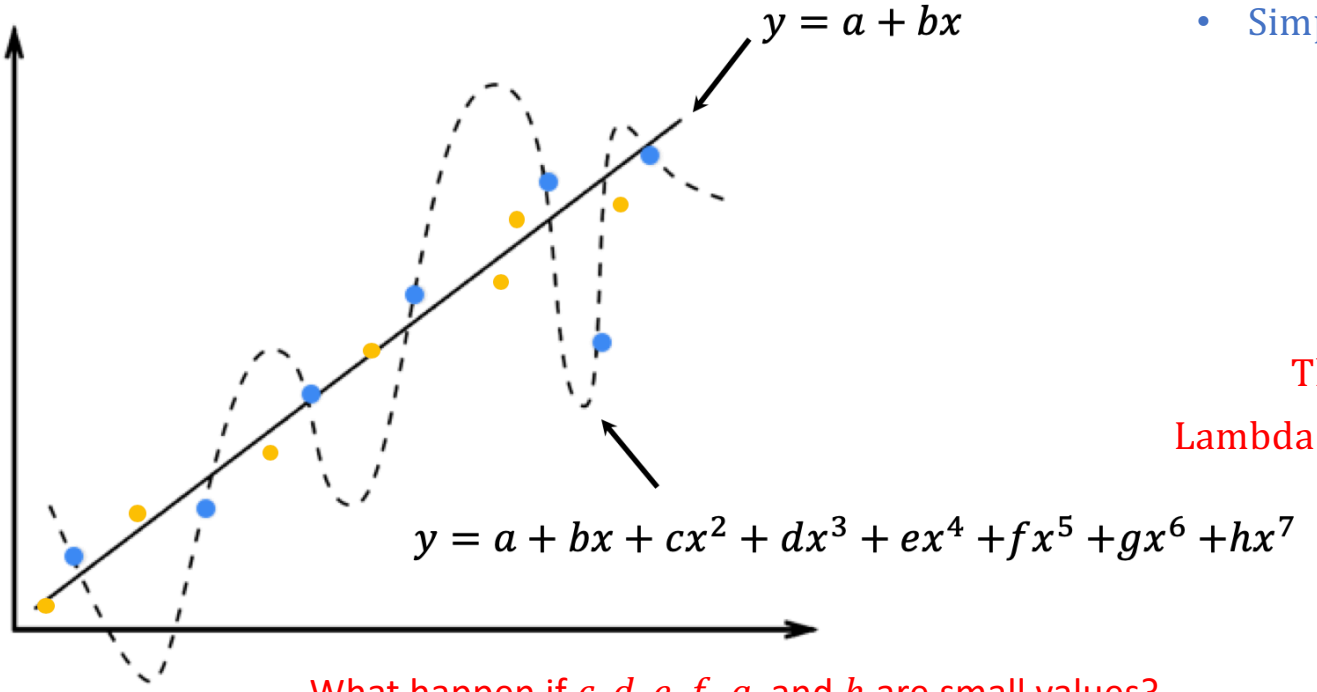
Regularization

- Early Stopping
  - Terminate the training before the model start to overfit the data



Regularization

- Weight Decay



What happen if c, d, e, f, g, and h are small values?

- Simply make the parameters smaller !

$$\mathcal{L}_{total} = \mathcal{L} + \lambda \|\boldsymbol{W}\|$$

The original loss — Regularization term

Lambda: controls the strength of the regularization

Regularization

- $\mathcal{L}_1$ norm

$$\mathcal{L}_{total} = \mathcal{L} + \lambda \mathcal{L}_1$$

$$\mathcal{L}_1 = \|\boldsymbol{W}\|$$

- $\mathcal{L}_2$ norm

$$\mathcal{L}_{total} = \mathcal{L} + \lambda \mathcal{L}_2$$

$$\mathcal{L}_2 = \|\boldsymbol{W}\|_2^2$$

Note: The weight decay only for the weights not the biases

Regularization

- $\mathcal{L}_1$ vs $\mathcal{L}_2$ norm

Given only two weights: w_1 and w_2

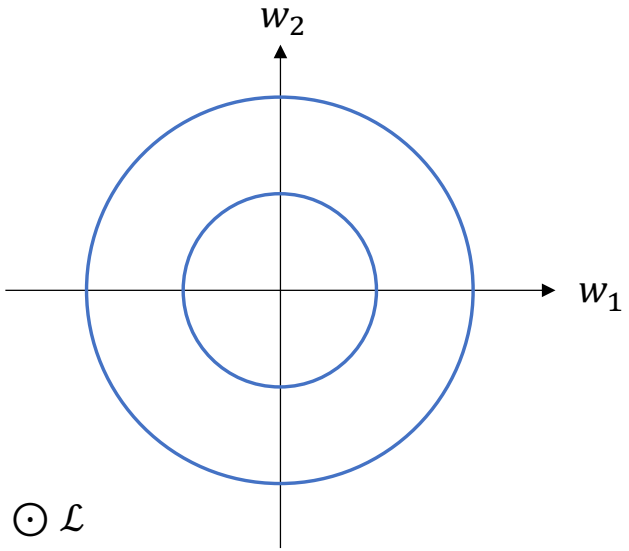
$\mathcal{L}_2$ penalty become ..

$\mathcal{L}_2 = \|\boldsymbol{W}\|_2^2 = {w_1}^2 + {w_2}^2$

Given a specific $\mathcal{L}_2$ value, it is a circle:

$1 = {w_1}^2 + {w_2}^2$ is a circle with a radius of 1

$4 = {w_1}^2 + {w_2}^2$ is a circle with a radius of 2



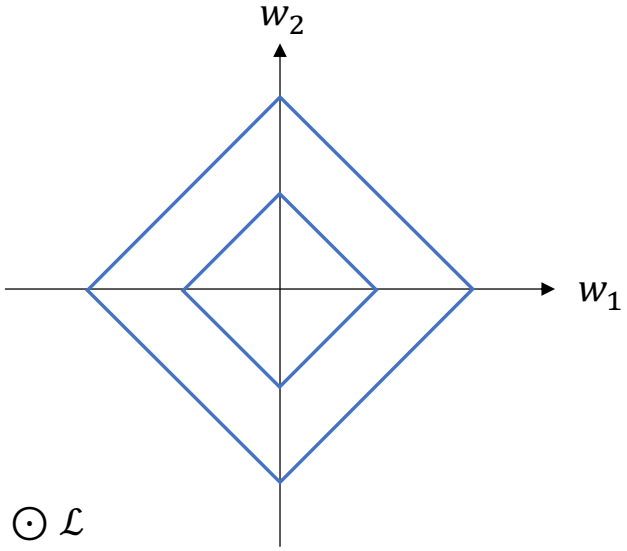
$\mathcal{L}_1$ penalty become ..

$\mathcal{L}_1 == |w_1| + |w_1|$

Given a specific $\mathcal{L}_1$ value, it is a square:

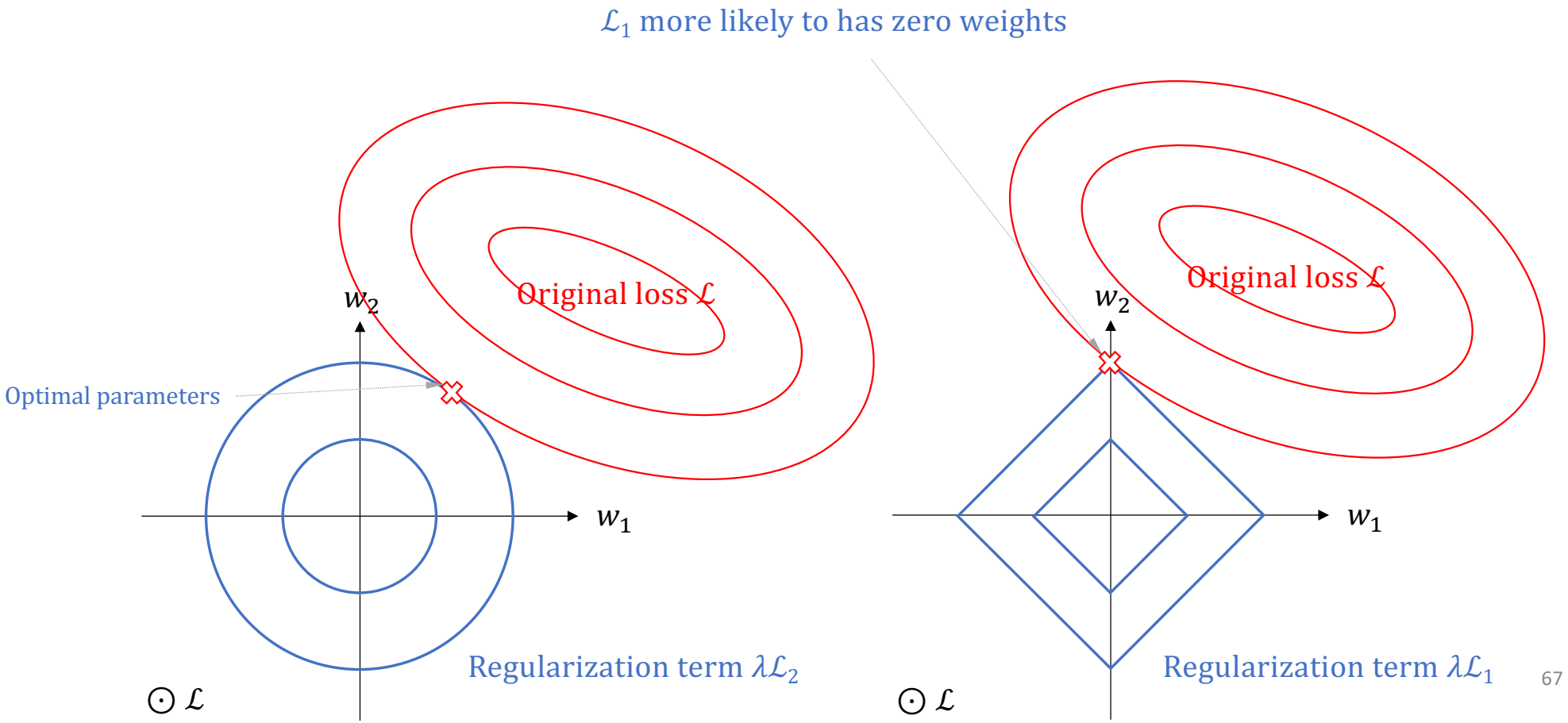
$1 = |w_1| + |w_2|$ is a square with a diagonal line that has a length of 1

$2 = |w_1| + |w_2|$ is a square with a diagonal line that has a length of 2



Regularization

- $\mathcal{L}_1$ vs $\mathcal{L}_2$ norm



Regularization

- $\mathcal{L}_1$ vs $\mathcal{L}_2$ norm

The network parameter values are often smaller than 1, so by using $\mathcal{L}_2$, a smaller value can result in a much smaller penalty than $\mathcal{L}_1$ (e.g., $|w| > w^2$ when $w < 1$). In contrast, $\mathcal{L}_1$ can have a larger plenty than $\mathcal{L}_2$ for small values.

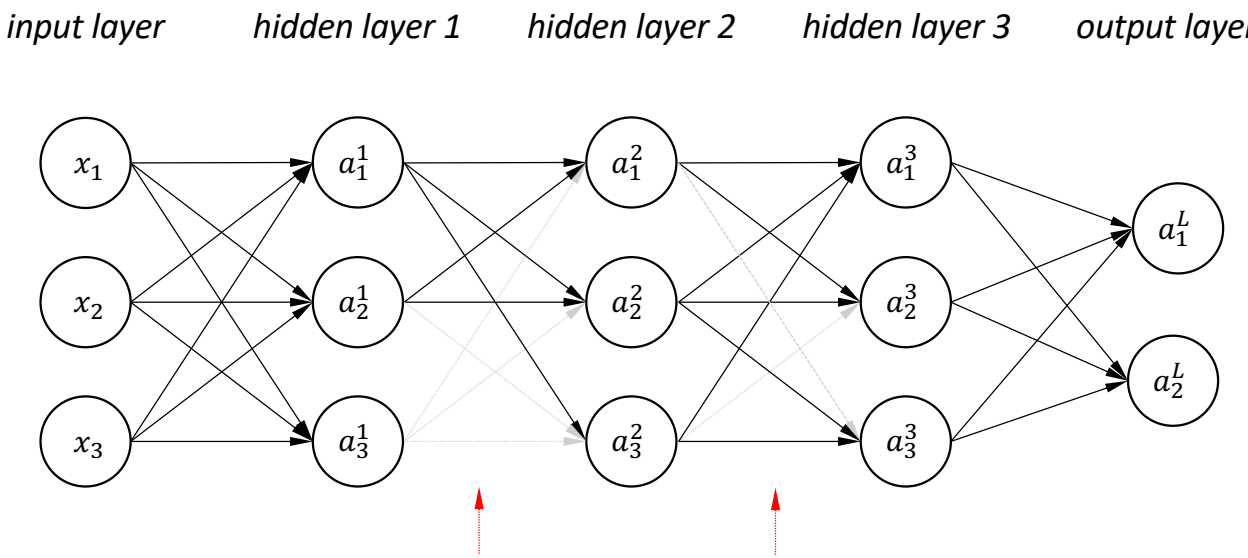
$|0.5| > 0.5^2$

Regularization

- $\mathcal{L}_1$ vs $\mathcal{L}_2$ norm

$\mathcal{L}_1$ more likely to has zero parameters, so it has the sparse property which enables the networks to perform "feature selection".

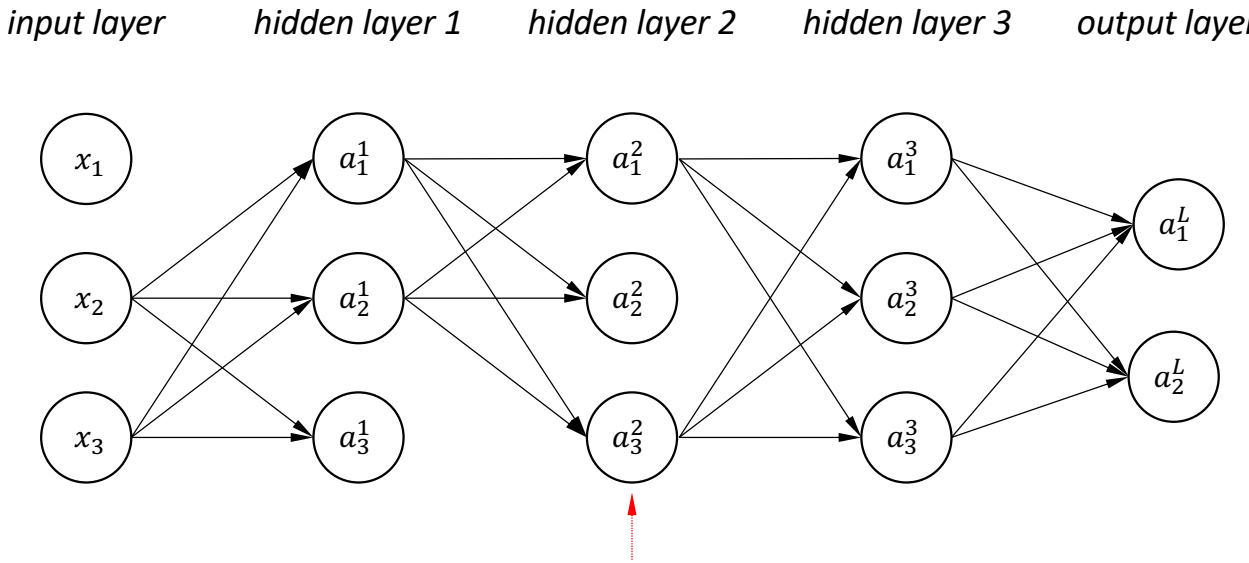
(ReLU has such property as well)



Discarding some input features by setting the corresponding parameters to zero or a very small value

Regularization

- Dropout



In practice, there will be hundreds and thousands units per layer

- Large neural networks include many parameters making it difficult to deal with the overfitting by combining the predictions from so many parameters.
- Dropout randomly set the hidden outputs to zero, which resembles a random disconnection of the neural units from one layer to the next

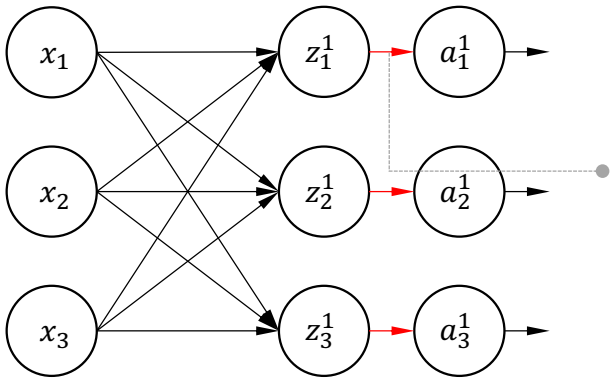
Regularization

- Dropout
  - According to error back-propagation, with a zero-valued output a, the corresponding partial derivative of the loss w.r.t respect to the layer output will be zero. In other words, only the remaining connected weights will be updated.
  - The dropout method can train may different sub-networks while allowing all of them to share the same network parameters.
  - During testing, dropout is disabled, and no output elements are set to zero. In other words, all sub-networks are used to predict the result represented as using the average from many networks as the final result (i.e., ensemble learning)

In practice, only apply Dropout in fully connected layers

Regularization

- Batch Normalization



- Batch normalization is the introduction of a layer that normalizes the inputs to have a mean of 0 and variance of 1 and can improve the performance of a neural network and its training stability.
- During training, the batch normalisation layer estimates the mean and variance of the batch input using a moving average, which updates the moving mean and variance for every iteration to be used for normalizing the batch input.
- During testing, the moving mean and variance are fixed and applied to normalise the input.
- Similar to the dropout process that adds a random factor to the hidden values, the moving mean and variance of batch normalization introduce randomness as they are updated in each iteration according to the random mini-batch. Therefore, the network must learn to be robust enough to deal with the variation.

- Do NOT use bias in the layer before batch norm
- DO NOT use activation operation before batch norm

北京大學
PEKING UNIVERSITY

Implementation

Implementation

Static Model

```python
import tensorflow as tf
from tensorlayer.layers import Input, Dropout, Dense
from tensorlayer.models import Model

def get_model(inputs_shape):
    ni = Input(inputs_shape)
    nn = Dropout(keep=0.8)(ni)
    nn = Dense(n_units=800, act=tf.nn.relu, name="dense1")(nn)
    nn = Dropout(keep=0.8)(nn)
    nn = Dense(n_units=800, act=tf.nn.relu)(nn)
    nn = Dropout(keep=0.8)(nn)
    nn = Dense(n_units=10, act=tf.nn.relu)(nn)
    M = Model(inputs=ni, outputs=nn, name="mlp")
    return M

MLP = get_model([None, 784])
MLP.eval()
outputs = MLP(data)
```

Lasagne Fashion

https://tensorlayer.readthedocs.io

Dynamic Model

```python
class CustomModel(Model):

    def __init__(self):
        super(CustomModel, self).__init__()

        self.dropout1 = Dropout(keep=0.8)
        self.dense1 = Dense(n_units=800, act=tf.nn.relu, in_channels=784)
        self.dropout2 = Dropout(keep=0.8)#(self.dense1)
        self.dense2 = Dense(n_units=800, act=tf.nn.relu, in_channels=800)
        self.dropout3 = Dropout(keep=0.8)#(self.dense2)
        self.dense3 = Dense(n_units=10, act=tf.nn.relu, in_channels=800)

    def forward(self, x, foo=False):
        z = self.dropout1(x)
        z = self.dense1(z)
        z = self.dropout2(z)
        z = self.dense2(z)
        z = self.dropout3(z)
        out = self.dense3(z)
        if foo:
            out = tf.nn.relu(out)
        return out

MLP = CustomModel()
MLP.eval()
outputs = MLP(data, foo=True) # controls the forward here
outputs = MLP(data, foo=False)
```

Chainer Fashion

Implementation

北京大學 PEKING UNIVERSITY

Switching Train/Test Modes

```
01. # method 1: switch before forward
02. Model.train() # enable dropout, batch norm moving avg ...
03. output = Model(train_data)
04. ... # training code here
05. Model.eval()  # disable dropout, batch norm moving avg ...
06. output = Model(test_data)
07. ... # testing code here
08.
09. # method 2: switch while forward
10. output = Model(train_data, is_train=True)
11. output = Model(test_data, is_train=False)
```

Implementation

Reuse Weights in Static Model

```python
def create_base_network(input_shape):
    '''Base network to be shared (eq. to feature extraction).
    '''
    input = Input(shape=input_shape)
    x = Flatten()(input)
    x = Dense(128, act=tf.nn.relu)(x)
    x = Dropout(0.9)(x)
    x = Dense(128, act=tf.nn.relu)(x)
    x = Dropout(0.9)(x)
    x = Dense(128, act=tf.nn.relu)(x)
    return Model(input, x)


def get_siamese_network(input_shape):
    """Create siamese network with shared base network as layer
    """
    base_layer = create_base_network(input_shape).as_layer() # convert model as layer

    ni_1 = Input(input_shape)
    ni_2 = Input(input_shape)
    nn_1 = base_layer(ni_1) # call base_layer twice
    nn_2 = base_layer(ni_2)
    return Model(inputs=[ni_1, ni_2], outputs=[nn_1, nn_2])

siamese_net = get_siamese_network([None, 784])
```

Reuse Weights in Dynamic Model

```python
class MyModel(Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.dense_shared = Dense(n_units=800, act=tf.nn.relu, in_channels=784)
        self.dense1 = Dense(n_units=10, act=tf.nn.relu, in_channels=800)
        self.dense2 = Dense(n_units=10, act=tf.nn.relu, in_channels=800)
        self.cat = Concat()

    def forward(self, x):
        x1 = self.dense_shared(x) # call dense_shared twice
        x2 = self.dense_shared(x)
        x1 = self.dense1(x1)
        x2 = self.dense2(x2)
        out = self.cat([x1, x2])
        return out

model = MyModel()
```

Implementation

Print Model Architecture

```
01. import pprint
02.
03. def get_model(inputs_shape):
04.     ni = Input(inputs_shape)
05.     nn = Dropout(keep=0.8)(ni)
06.     nn = Dense(n_units=800, act=tf.nn.relu)(nn)
07.     nn = Dropout(keep=0.8)(nn)
08.     nn = Dense(n_units=800, act=tf.nn.relu)(nn)
09.     nn = Dropout(keep=0.8)(nn)
10.     nn = Dense(n_units=10, act=tf.nn.relu)(nn)
11.     M = Model(inputs=ni, outputs=nn, name="mlp")
12.     return M
13.
14. MLP = get_model([None, 784])
15. pprint.pprint(MLP.config)
```

```
[{'args': {'dtype': tf.float32,
           'layer_type': 'normal',
           'name': '_inputlayer_1',
           'shape': [None, 784]},
  'class': '_InputLayer',
  'prev_layer': None},
 {'args': {'keep': 0.8, 'layer_type': 'normal', 'name': 'dropout_1'},
  'class': 'Dropout',
  'prev_layer': ['_inputlayer_1_node_0']},
 {'args': {'act': 'relu',
           'layer_type': 'normal',
           'n_units': 800,
           'name': 'dense_1'},
  'class': 'Dense',
  'prev_layer': ['dropout_1_node_0']},
 {'args': {'keep': 0.8, 'layer_type': 'normal', 'name': 'dropout_2'},
  'class': 'Dropout',
  'prev_layer': ['dense_1_node_0']},
 {'args': {'act': 'relu',
           'layer_type': 'normal',
           'n_units': 800,
           'name': 'dense_2'},
  'class': 'Dense',
  'prev_layer': ['dropout_2_node_0']},
 {'args': {'keep': 0.8, 'layer_type': 'normal', 'name': 'dropout_3'},
  'class': 'Dropout',
  'prev_layer': ['dense_2_node_0']},
 {'args': {'act': 'relu',
           'layer_type': 'normal',
           'n_units': 10,
           'name': 'dense_3'},
  'class': 'Dense',
  'prev_layer': ['dropout_3_node_0']}]
```

Implementation

Print Model Information

```
print(MLP) # simply call print function

# Model(
#   (_inputlayer): Input(shape=[None, 784], name='_inputlayer')
#   (dropout): Dropout(keep=0.8, name='dropout')
#   (dense): Dense(n_units=800, relu, in_channels='784', name='dense')
#   (dropout_1): Dropout(keep=0.8, name='dropout_1')
#   (dense_1): Dense(n_units=800, relu, in_channels='800', name='dense_1')
#   (dropout_2): Dropout(keep=0.8, name='dropout_2')
#   (dense_2): Dense(n_units=10, relu, in_channels='800', name='dense_2')
# )
```

Save Weights Only

```
MLP.save_weights('./model_weights.h5')
MLP.load_weights('./model_weights.h5')
```

Get Specific Weights

```
# indexing
all_weights = MLP.all_weights
some_weights = MLP.all_weights[1:3]

# naming
some_weights = MLP.get_layer('dense1').all_weights
```

Save Weights + Architecture

```
MLP.save('./model.h5', save_weights=True)
MLP = Model.load('./model.h5', load_weights=True)
```

Implementation

Customized layers with weights

```python
class Dense(Layer):
    """The :class:`Dense` class is a fully connected layer.

    Parameters
    ----------
    n_units : int
        The number of units of this layer.
    act : activation function
        The activation function of this layer.
    name : None or str
        A unique layer name. If None, a unique name will be automatically generated.
    """

    def __init__(
            self,
            n_units,   # the number of units/channels of this layer
            act=None,  # None: no activation, tf.nn.relu: ReLU ...
            name=None, # the name of this layer (optional)
    ):
        super(Dense, self).__init__(name) # auto naming, dense_1, dense_2 ...
        self.n_units = n_units
        self.act = act

    def build(self, inputs_shape): # initialize the model weights here
        shape = [inputs_shape[1], self.n_units]
        self.W = self._get_weights("weights", shape=tuple(shape), init=self.W_init)
        self.b = self._get_weights("biases", shape=(self.n_units, ), init=self.b_init)

    def forward(self, inputs): # call function
        z = tf.matmul(inputs, self.W) + self.b
        if self.act: # is not None
            z = self.act(z)
        return z
```

```python
class Dense(Layer):
    """The :class:`Dense` class is a fully connected layer.

    Parameters
    ----------
    n_units : int
        The number of units of this layer.
    act : activation function
        The activation function of this layer.
    W_init : initializer
        The initializer for the weight matrix.
    b_init : initializer or None
        The initializer for the bias vector. If None, skip biases.
    in_channels: int
        The number of channels of the previous layer.
        If None, it will be automatically detected when the layer is forwarded for the first time.
    name : None or str
        A unique layer name. If None, a unique name will be automatically generated.
    """

    def __init__(
            self,
            n_units,
            act=None,
            W_init=tl.initializers.truncated_normal(stddev=0.1),
            b_init=tl.initializers.constant(value=0.0),
            in_channels=None,  # the number of units/channels of the previous layer
            name=None,
    ):
        # we feed activation function to the base layer, `None` denotes identity function
        # string (e.g., relu, sigmoid) will be converted into function.
        super(Dense, self).__init__(name, act=act)

        self.n_units = n_units
        self.W_init = W_init
        self.b_init = b_init
        self.in_channels = in_channels

        # in dynamic model, the number of input channel is given, we initialize the weights here
        if self.in_channels is not None:
            self.build(self.in_channels)
            self._built = True

        logging.info(
            "Dense  %s: %d %s" %
            (self.name, self.n_units, self.act.__name__ if self.act is not None else 'No Activation')
        )

    def build(self, inputs_shape): # initialize the model weights here
        if self.in_channels: # if the number of input channel is given, use it
            shape = [self.in_channels, self.n_units]
        else:                # otherwise, get it from static model
            self.in_channels = inputs_shape[1]
            shape = [inputs_shape[1], self.n_units]
        self.W = self._get_weights("weights", shape=tuple(shape), init=self.W_init)
        if self.b_init:      # if b_init is None, no bias is applied
            self.b = self._get_weights("biases", shape=(self.n_units, ), init=self.b_init)

    def forward(self, inputs):
        z = tf.matmul(inputs, self.W)
        if self.b_init:
            z = tf.add(z, self.b)
        if self.act:
            z = self.act(z)
        return z
```

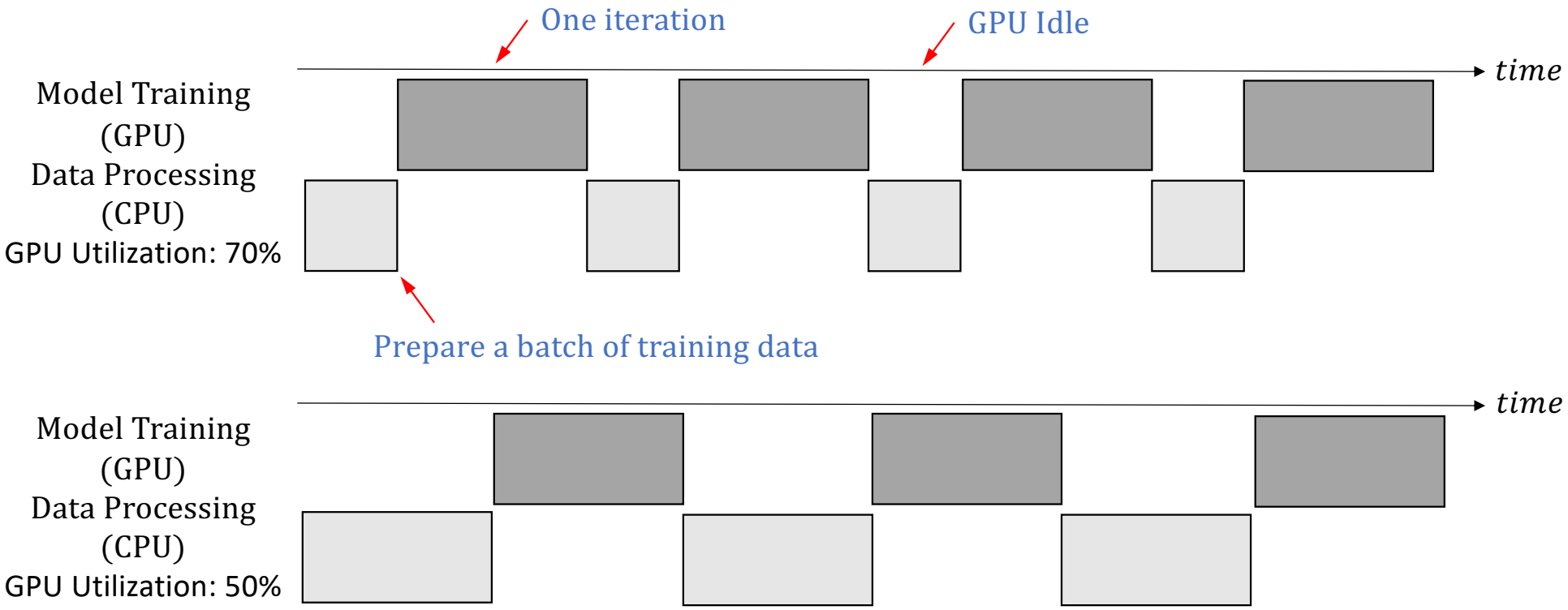
Implementation

Customized layers with train/test modes

```
class Dropout(Layer):
    """
    The :class:`Dropout` class is a noise layer which randomly set some
    activations to zero according to a keeping probability.
    Parameters
    ----------
    keep : float
        The keeping probability.
        The lower the probability it is, the more activations are set to zero.
    name : None or str
        A unique layer name.
    """

    def __init__(self, keep, name=None):
        super(Dropout, self).__init__(name)
        self.keep = keep

        self.build()
        self._built = True

        logging.info("Dropout %s: keep: %f " % (self.name, self.keep))

    def build(self, inputs_shape=None):
        pass   # no weights in dropout layer

    def forward(self, inputs):
        if self.is_train:  # this attribute is changed by Model.train() and Model.eval()
            outputs = tf.nn.dropout(inputs, rate=1 - (self.keep), name=self.name)
        else:
            outputs = inputs
        return outputs
```

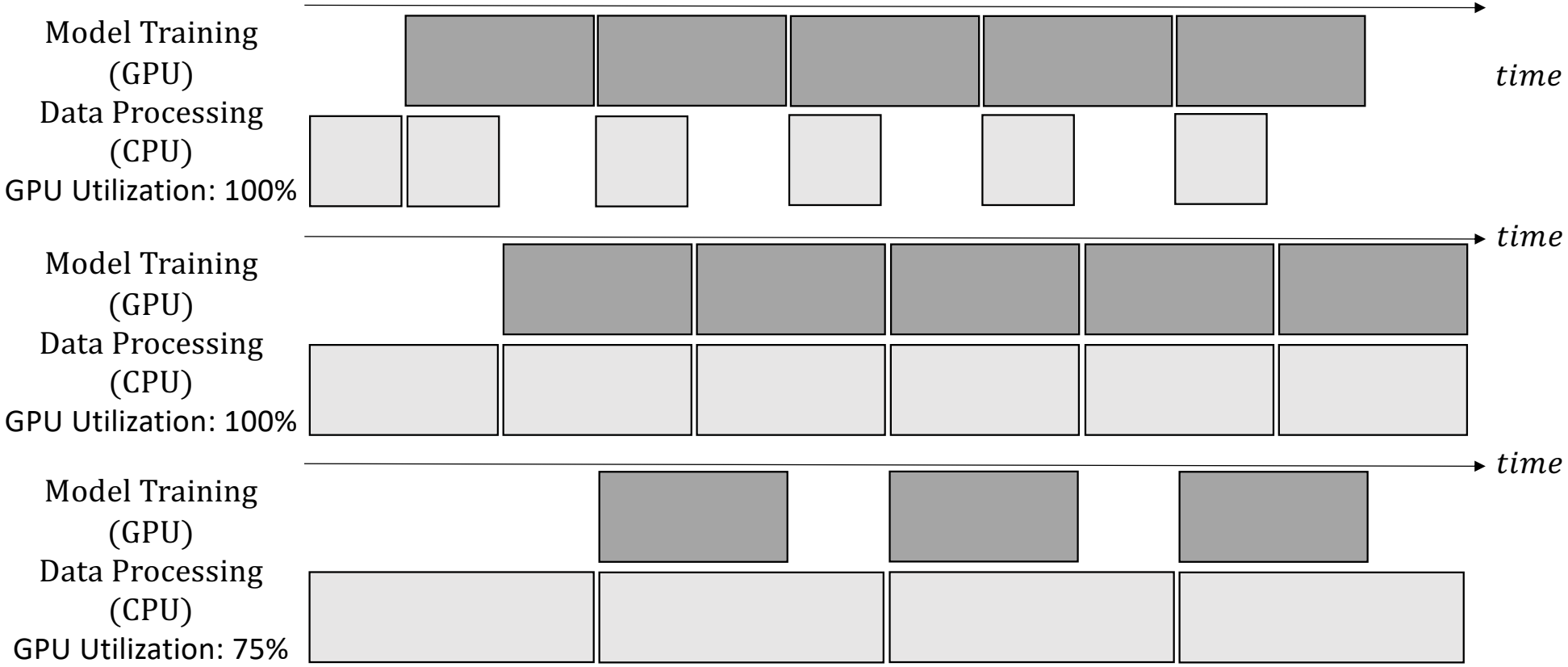
Implementation

- Training without Dataflow



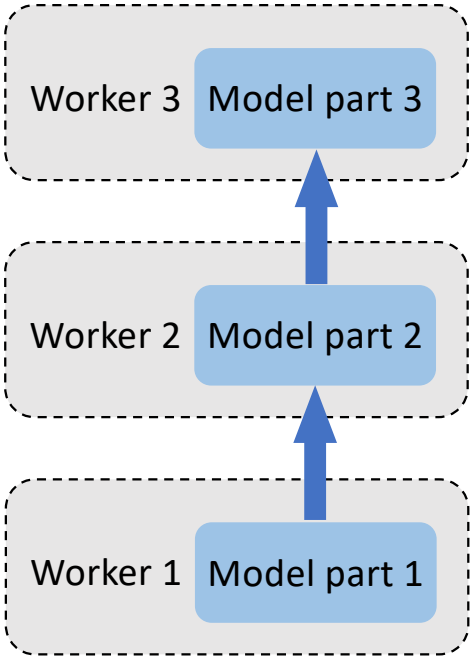
Implementation

- Training with Dataflow

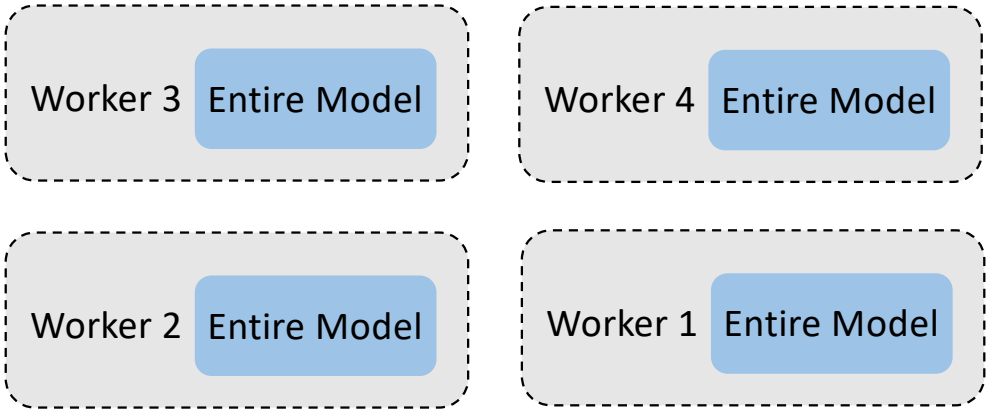


Implementation

- Distributed Training



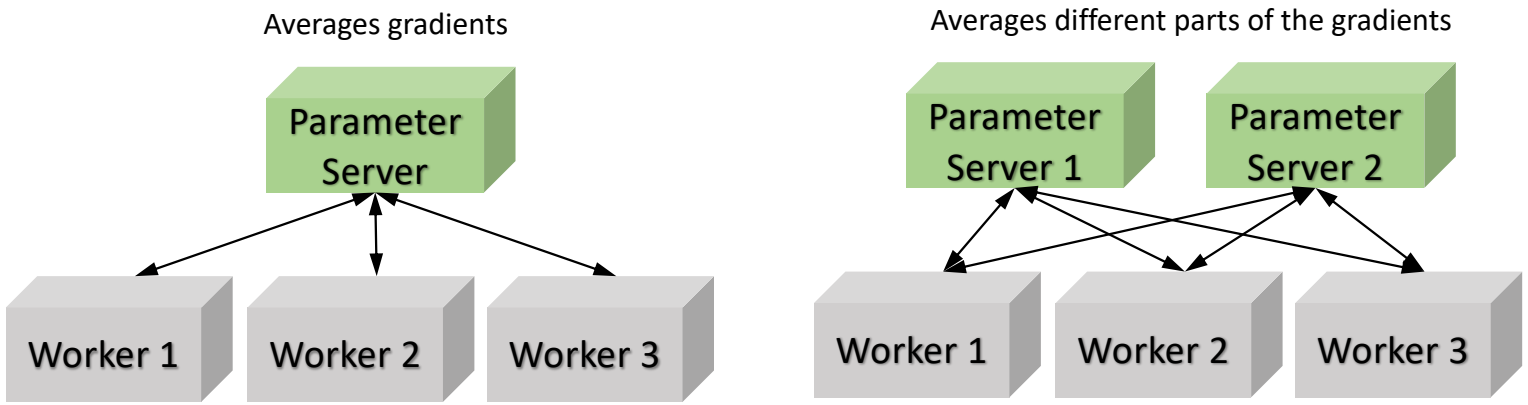
Model Parallelism



Data Parallelism

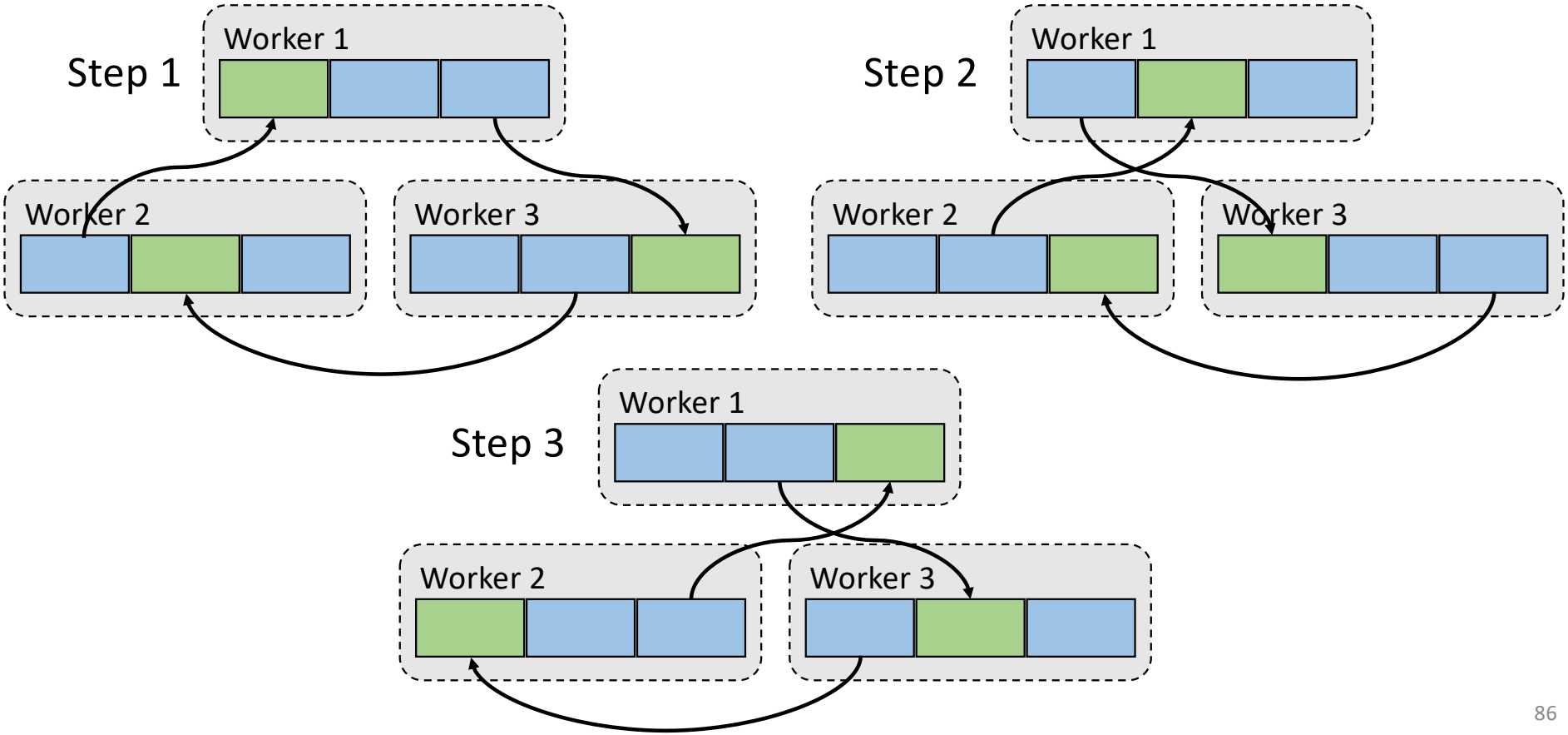
Implementation

- Distributed Training: Data Parallelism - Parameter Server



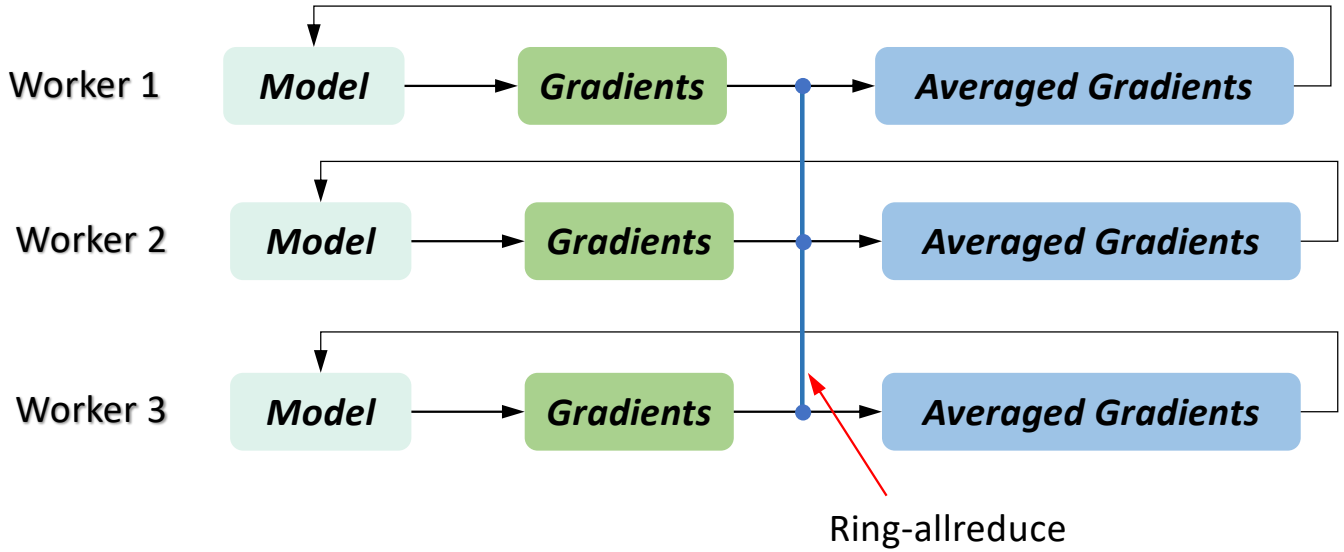
Implementation

- Distributed Training: Data Parallelism - Horovod - All ringreduce



Implementation

- Distributed Training: Data Parallelism - Horovod - All ringreduce



Implementation

- TensorLayer 2.0

5000 Stars ~ 1000 Forks ~ 180k Downloads

TensorLayer: A Versatile Library for Efficient Deep Learning Development. *H. Dong, A. Supratak et al. ACM MM 2017.*

北京大學
PEKING UNIVERSITY

Summary

Deep Neural Network Foundation

- Single Neuron
  - The neuron can be represented by the matrix multiplication.
  - A bias value allows the output value to be shifted higher or lower to better fit the input data.
- Activation Functions
  - Provide non-linearity to the network.
- Multi-layer Perceptron
  - Higher representation capacity.
- Loss Functions
  - The goal of the optimization.
- Optimization
  - Gradient descent → Error back-propagation → SGD → Adaptive learning rate
- Regularization
  - Weight decay, dropout, batch norm …
- Implementation
  - Define static/dynamic models, dataflow, distributed training …

Deep Neural Network Foundation

Link: https://github.com/zsdonghao/deep-learning-note

- Exercise 1:
  - Implement the error back-propagation with the provided Numpy template.

  Keywords: regression, classification, Sigmoid, MSE, logistic regression, fully connected layer
- Exercise 2:
  - Use TensorLayer to classify MNIST handwritten digit dataset including training, validating, and testing.

  Keywords: classification, cross-entropy, fully connected layer, evaluation
- Exercise 3:
  - Classify the MNIST dataset by modifying the code from exercise 1.

  Keywords: classification, Softmax, ReLU, cross-entropy, fully connected layer
- Exercise 4: (Optional)
  - Following exercise 2, implement a dataflow for data augmentation

  Keywords: classification, Softmax, ReLU, cross-entropy, fully connected layer

北京大學
PEKING UNIVERSITY

Questions?