北京大學

神经网络基础

Deep Neural Network Foundation

主讲人：董豪　　讲义：董豪

深度学习

人工智能

机器学习

深度学习

深度强化学习

强化学习

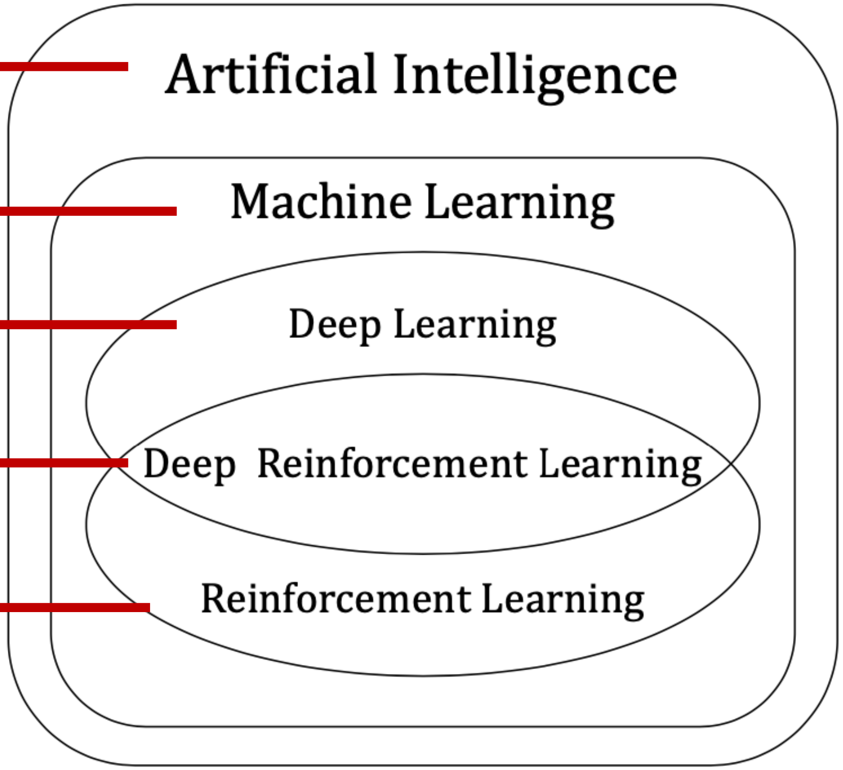


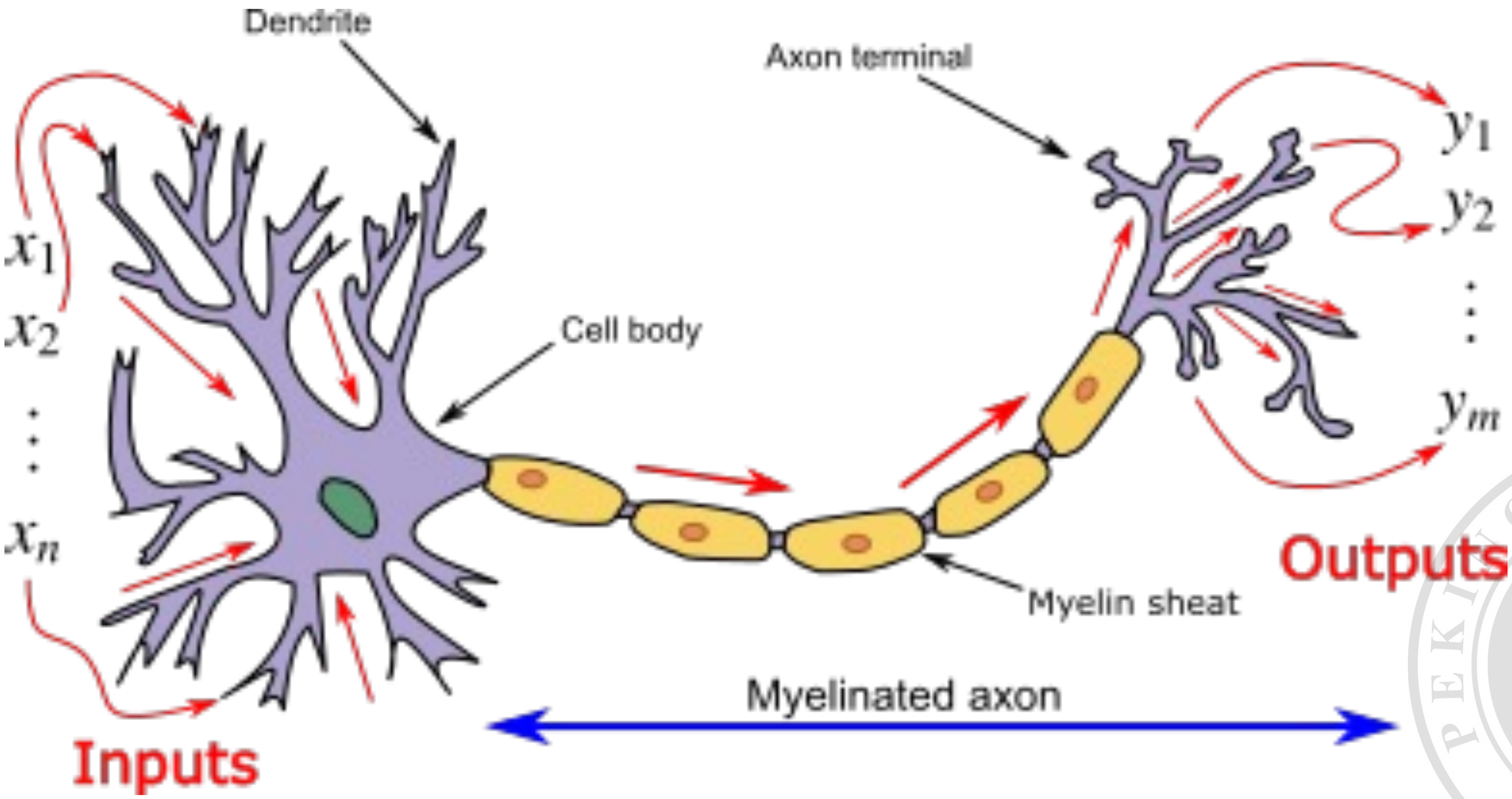
Image from “Deep Reinforcement Learning: Fundamentals, Research and Applications” *Hao Dong, Zihan Ding, Shanghang Zhang*

内容提要

- 单个神经元 Single Neuron
- 激活函数 Activation Functions
- 多层感知器 Multi-layer Perceptron
- 损失函数 Loss Functions
- 优化 Optimisation
- 正则化 Regularisation
- 实现 Implementation

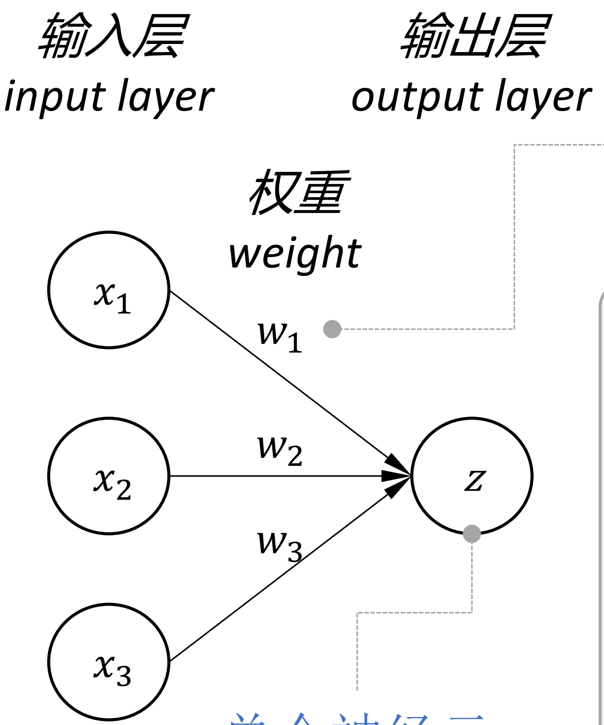
- 单个神经元 Single Neuron
- 激活函数 Activation Functions
- 多层感知器 Multi-layer Perceptron
- 损失函数 Loss Functions
- 优化 Optimisation
- 正则化 Regularisation
- 实现 Implementation

单个神经元


Dendrite
Axon terminal
Cell body
Myelin sheat
Myelinated axon
Inputs
Outputs
x_1
x_2
x_n
y_1
y_2
y_m

单个神经元

- 3个输入1个输出



单个神经元
A single neuron

输出z是多个输入x的线性组合（linear combination）

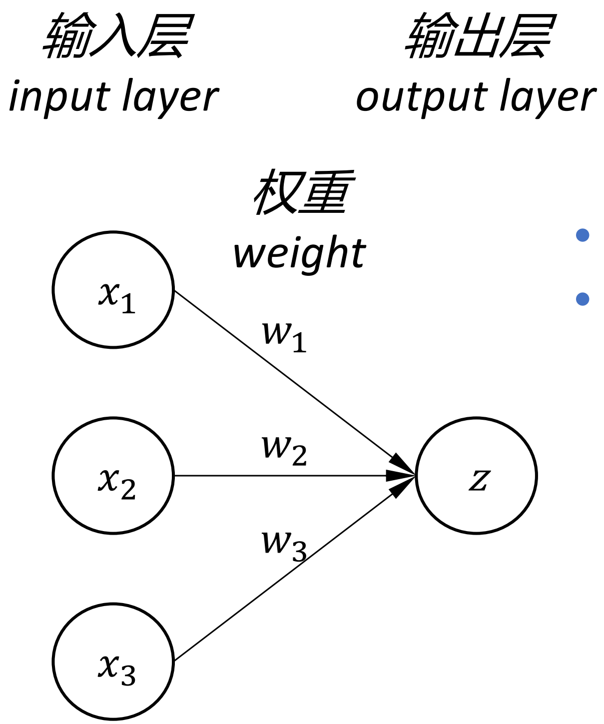
$$z = x_1w_1 + x_2w_2 + x_3w_3$$

- 权重/参数（weight/parameter）的绝对值越大，则代表对应的输入x对输出影响越大

例子： 输出z表示的是我们是否去踢球，z越大表示我们要去。为了求出z, x_1 表示天气情况，x_2是球场租金，x_3 球场的距离。这些输入可以看作是影响输出结果的特征（feature）。
如果天气是我们主要的考虑因素，则w_1是一个比w_2和w_3要大的数值。
如果我们不考虑租金，则w_2可以设为0，输出将和x_2无关。

单个神经元

- 3个输入1个输出



输出z是多个输入x的线性组合（linear combination）

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

- 一个神经元是只有一个输出的单层神经网络
- 因为输出与所有输入连接，这层神经网络可以被称为全连接层（“fully connected layer”或“dense layer”）

单个神经元

- 3个输入1个输出

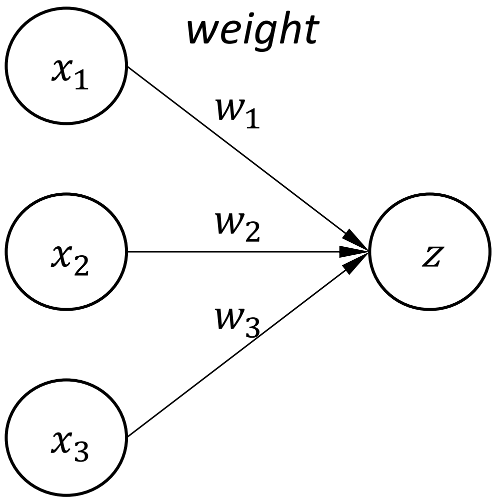
神经元的计算可以用向量（vector）想乘实现

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

输入层
input layer

输出层
output layer

权重
weight



列格式（常见于书籍）
column format (textbook)

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

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

$$z = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

行格式（常见于代码）
row format (python)

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

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

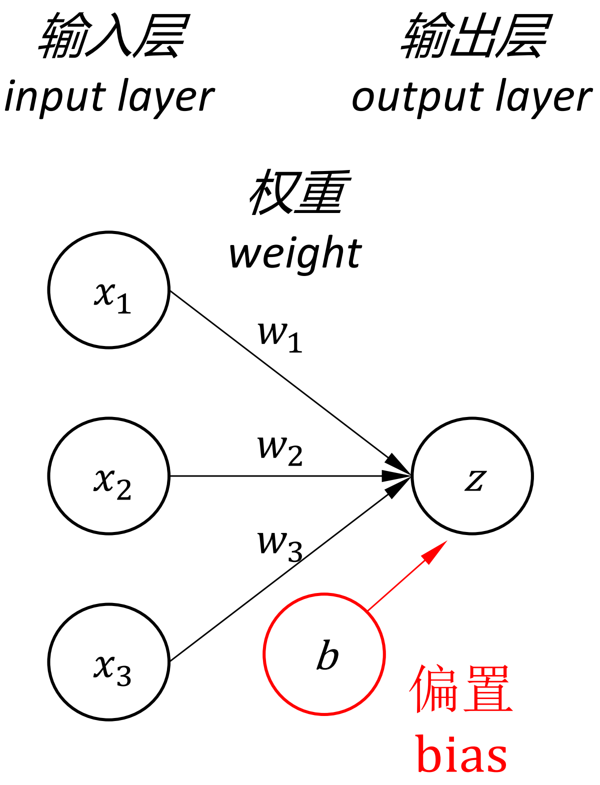
$$z = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

单个神经元

- 偏置（Bias）

偏置（Bias）使得输出值可以上下浮动，以更好地适应输入值

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



列格式（常见于书籍） *column format* (textbook)	行格式（常见于代码） *row format* (python)
$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$	$\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \quad \boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$
$z = \boldsymbol{w}^T\boldsymbol{x} + b$	$z = \boldsymbol{x}\boldsymbol{w} + b$
$z = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + b$	$z = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} + b$

单个神经元

```python
import torch
# Single neuro
# matrix multiplication 矩阵乘法
x = torch.tensor([[1., 2., 3.]]) # 1x3
w = torch.tensor([[-0.5], [0.2], [0.1]]) # 3x1

print("X:\n", x, "\nShape:", x.shape)
print("W:\n", w, "\nShape", w.shape)

# Bias 偏置
b1 = torch.tensor(0.5)

# 计算有偏置的矩阵乘法
z1 = torch.matmul(x, w)+b1

print("Z:\n", z1, "\nShape", z1.shape)
```

```
X:
 tensor([[1., 2., 3.]])
Shape: torch.Size([1, 3])
W:
 tensor([[-0.5000],
        [ 0.2000],
        [ 0.1000]])
Shape torch.Size([3, 1])
Z:
 tensor([[0.7000]])
Shape torch.Size([1, 1])
```

单个神经元

- 分类（Classification）：看看偏置（bias）的意义

偏置（bias）使得输出值可以上下浮动，以更好地适应输入值

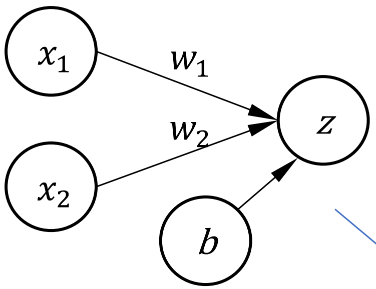
$$z = x_1 w_1 + x_2 w_2 + b$$

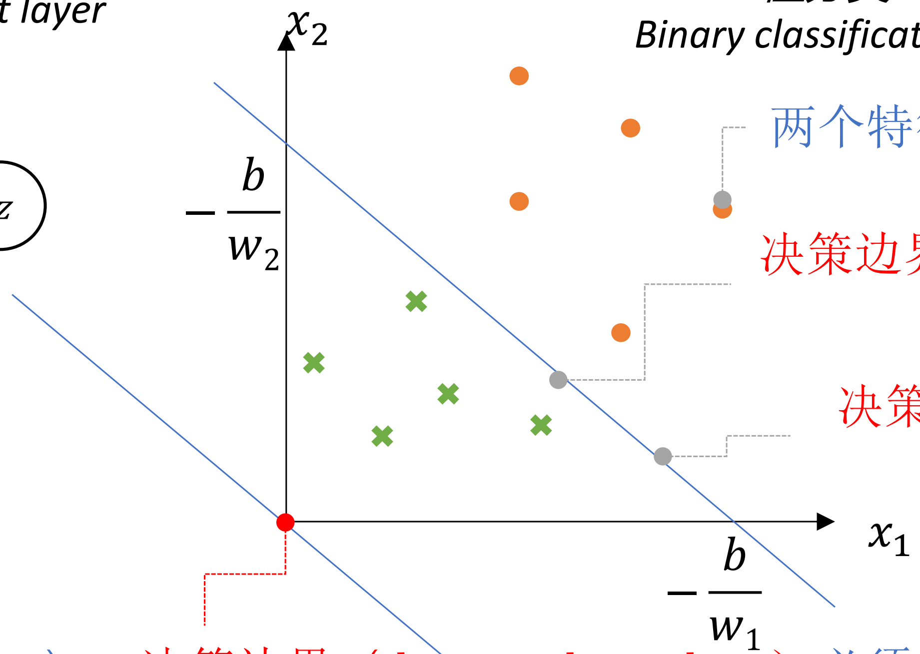
二值分类
Binary classification

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

输入层
input layer

输出层
output layer





两个特征/输入的数据样本（data sample）

决策边界（decision boundary）是一条当 $z = 0$ 时的直线

决策边界（decision boundary）可以被偏置（bias）上下调整

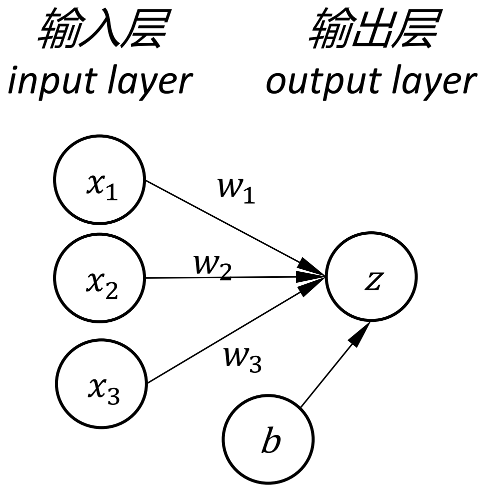
如果没有偏置（bias），决策边界（decision boundary）必须经过圆心（origin）

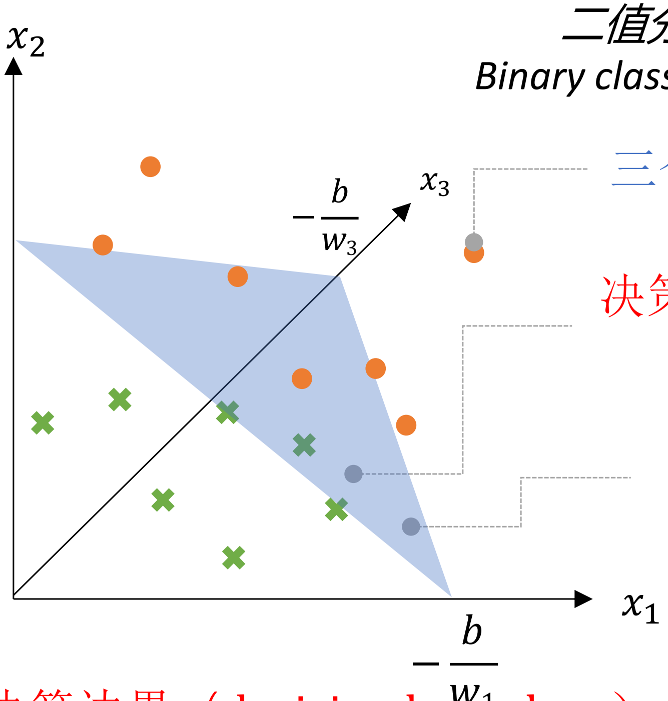
- 分类（Classification）：看看偏置（bias）的意义

偏置（bias）使得输出值可以上下浮动，以更好地适应输入值

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

二值分类 Binary classification $a = \begin{cases} 0, if\ z \le 0 \\ 1, if\ z > 0 \end{cases}$





三个特征/输入的数据样本（data sample）

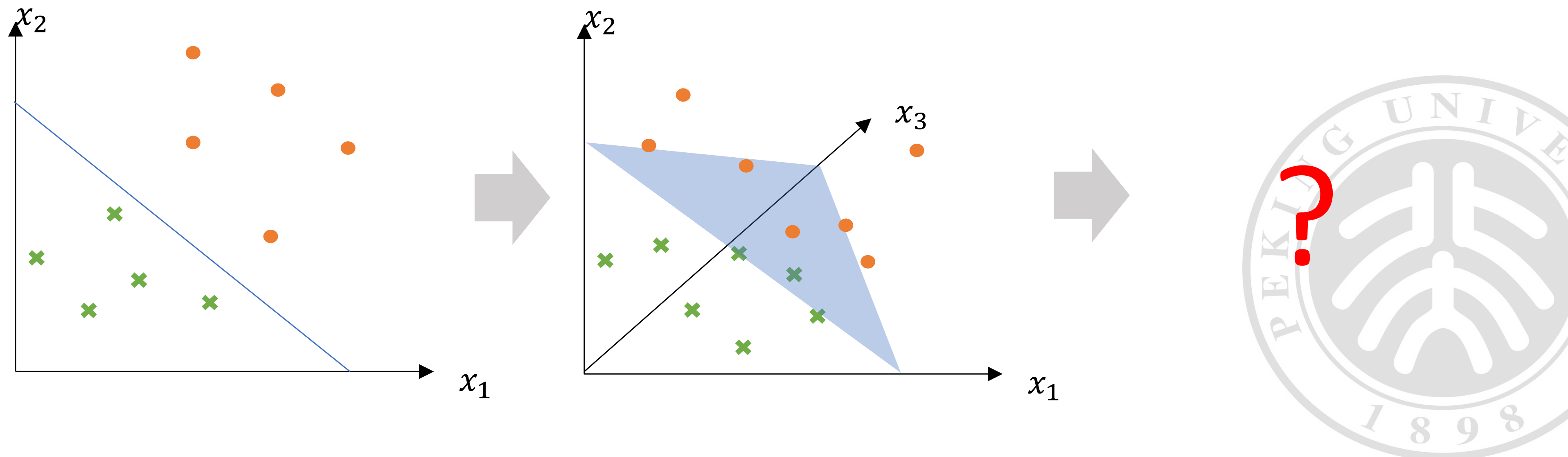
决策边界（decision boundary）是一个当 $z = 0$时的平面

决策边界（decision boundary）可以被偏置（bias）上下调整

如果没有偏置（bias），决策边界（decision boundary）必须经过圆心（origin）

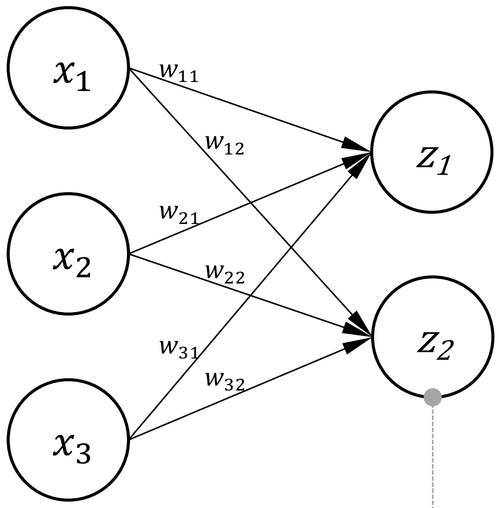
单个神经元

- 分类（Classification）：看看偏置（bias）的意义
  - 两个输入特征：决策边界是一条线
  - 三个输入特征：决策边界是一个平面
  - 更多输入特征：决策边界是一个超平面（hyperplane 或 hypersurface）



单个神经元

- 3输入2输出（两个神经元）



两个神经元

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

使用多个神经元，可以获得多个输出。
例如，输出可以分别表示我们去踢足球和打篮球的分数。

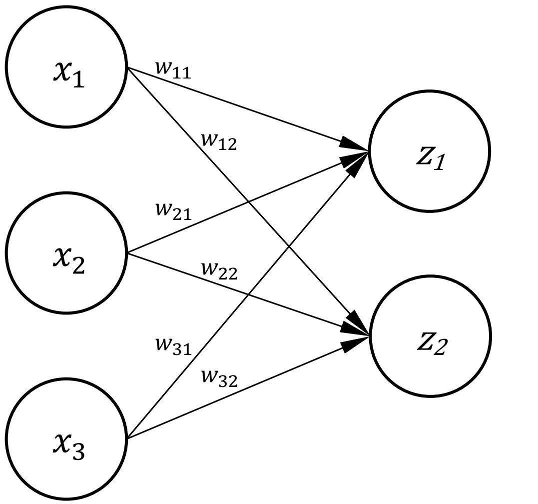
目前这两个神经元都是线性的

单个神经元

- 3输入2输出（两个神经元）

输入层
input layer

输出层
output layer



$$z_1 = x_1 w_{11} + x_2 w_{21} + x_3 w_{31}$$
$$z_2 = x_1 w_{12} + x_2 w_{22} + x_3 w_{32}$$

列格式（常见于书籍）
column format (textbook)

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

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

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

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

行格式（常见于代码）
row format (python)

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

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

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

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

单个神经元

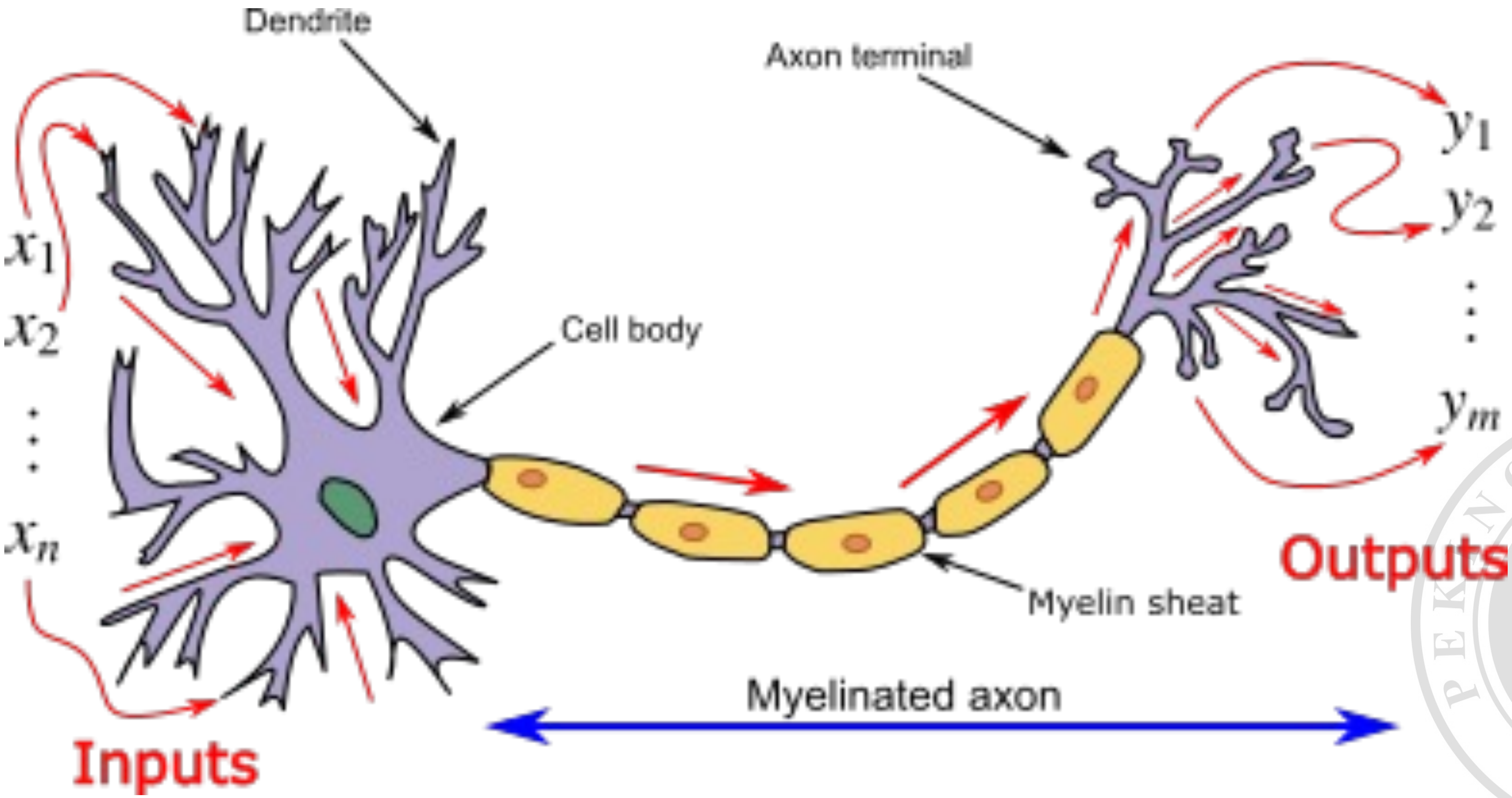
```python
# Two outputs 两个输出
x = torch.tensor([[1., 2., 3.]]) # 1x3

w = torch.tensor([[-0.5, -0.3],
                  [0.2, 0.4],
                  [0.1, 0.15]]) # 3x2

print("X:\n", x, "\nShape:", x.shape)
print("W:\n", w, "\nShape", w.shape)

# Bias 偏置
b2 = torch.tensor([0.5, 0.4])

z2 = torch.matmul(x, w)+b2

print("Z:\n", z2, "\nShape", z2.shape)
```

```
X:
 tensor([[1., 2., 3.]])
Shape: torch.Size([1, 3])
W:
 tensor([[-0.5000, -0.3000],
        [ 0.2000,  0.4000],
        [ 0.1000,  0.1500]])
Shape torch.Size([3, 2])
Z:
 tensor([[0.7000, 1.3500]])
Shape torch.Size([1, 2])
```

- 单个神经元 Single Neuron
- 激活函数 Activation Functions
- 多层感知器 Multi-layer Perceptron
- 损失函数 Loss Functions
- 优化 Optimisation
- 正则化 Regularisation
- 实现 Implementation

激活函数

激活函数（ activation function）为神经网络层的输出提供非线性性（non-linearity）



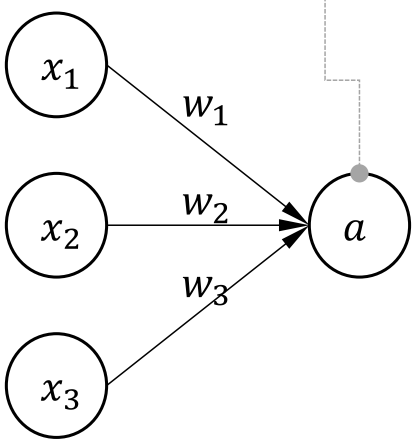
激活函数

- Sigmoid函数

接着之前的例子，给定一个神经网络，输出可以表示分数，比如踢足球的概率。为了表达0% ~ 100%的概率，需要把输出值限制到0 ~ 1之间。

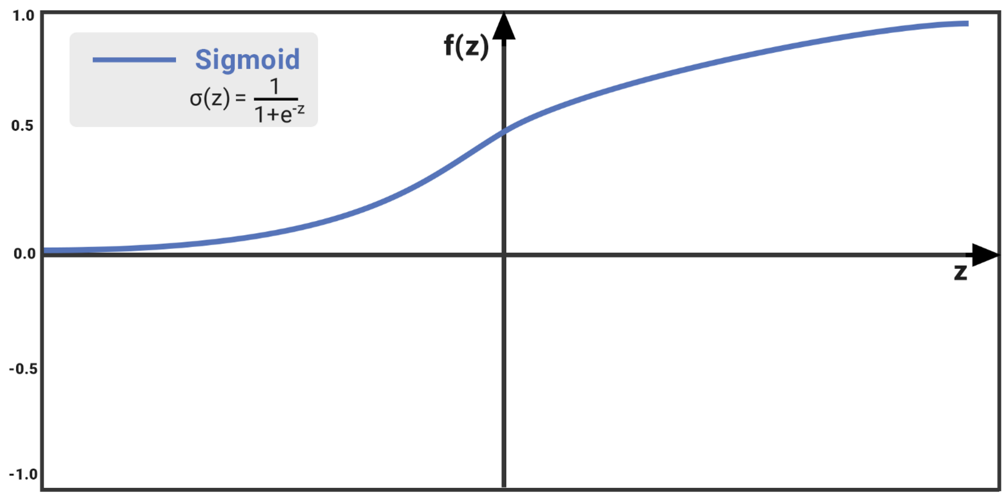
激活值（activation value）

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



Sigmoid函数

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



激活函数

- Sigmoid函数

```
z1 = torch.Tensor([[0.7000]]) # 线性组合值

# Activation function 激活函数
# Sigmoid function
sigmoid_torch = torch.nn.Sigmoid()
a1 = sigmoid_torch(z1)
print("Result torch sigmoid:", a1)

# define your own activation function
class ActSigmoid(torch.nn.Module):
    def __init__(self):
        super(ActSigmoid, self).__init__()

    def forward(self, x):
        return 1/(1+torch.exp(-x))

sigmoid_act = ActSigmoid()
a1_m = sigmoid_act(z1)
print("Result your own sigmoid:", a1_m)
```

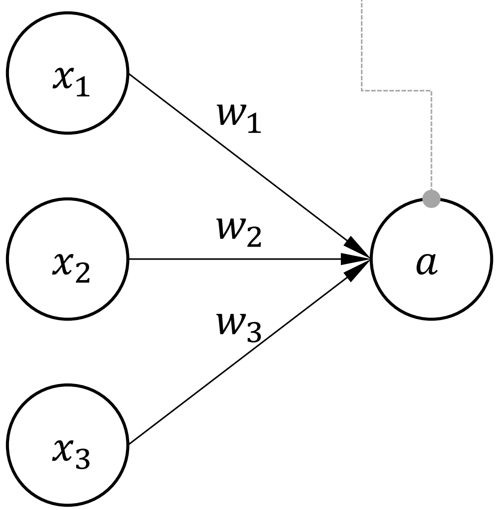
```
Result torch sigmoid: tensor([[0.6682]])

Result your own sigmoid: tensor([[0.6682]])
```

激活函数

- Tanh函数

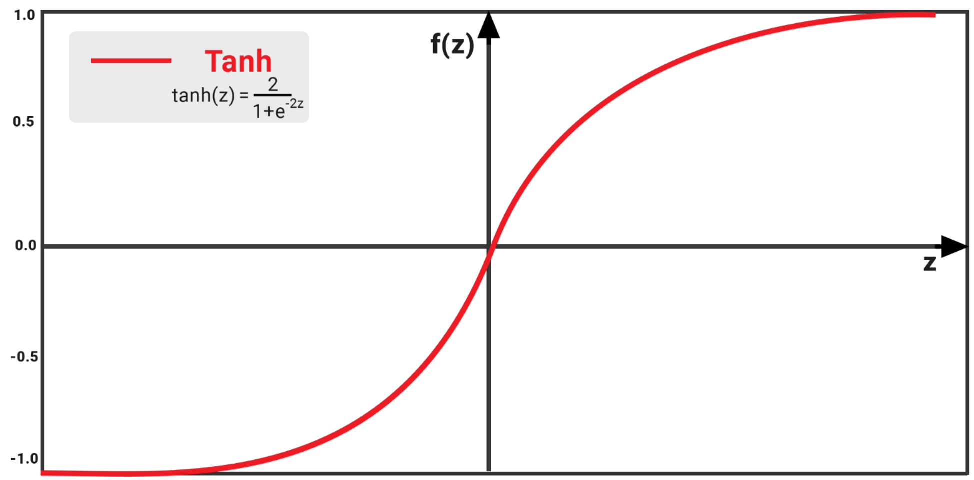
与Sigmoid函数类似，Hyperbolic tangent（tanh）函数也是限制输出值到一个范围。但Tanh函数限制的范围是 -1 ~ 1，该函数往往用于回归任务（regression task），比如网络输出图像，而图像的像素数值在 -1 ~ 1之间；再比如输出情绪，-1代表不开心，1代表开心，0代表中间状态。

$$a = f(z) = f(x_1w_1 + x_2w_2 + b)$$



Tanh函数

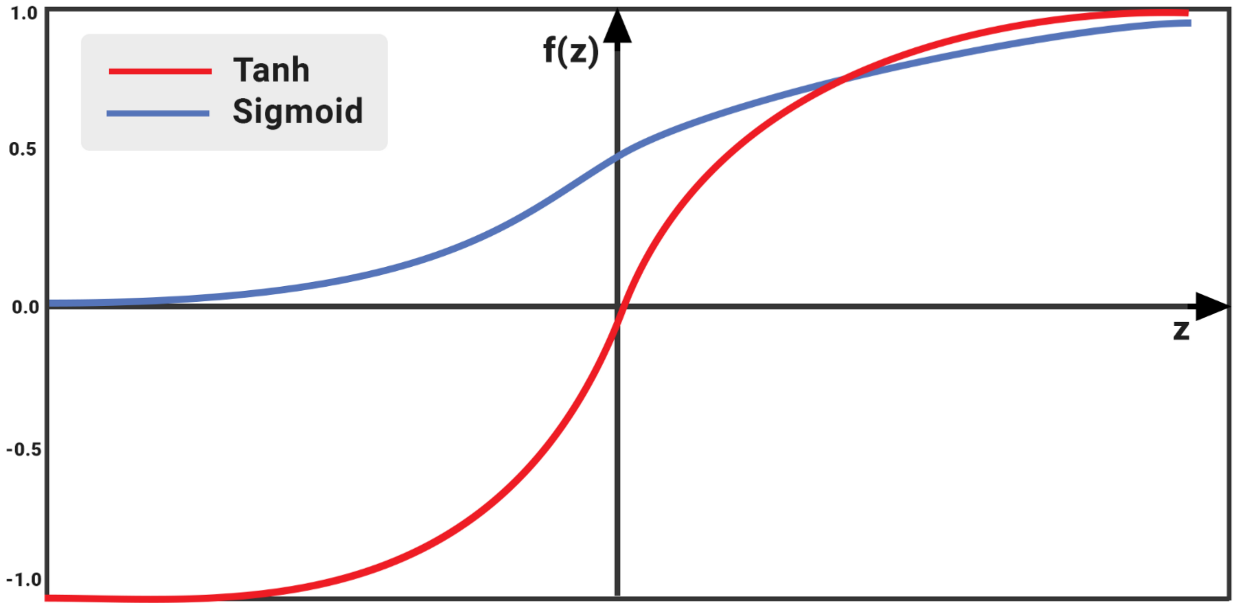
$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{2}{1 + e^{-2z}}$$



激活函数

- Sigmoid函数 vs. Tanh函数

Sigmoid和Tanh函数都可以为网络提供非线性性（non-linearity）。它们对输出的限制范围不同，用作不同的用途。

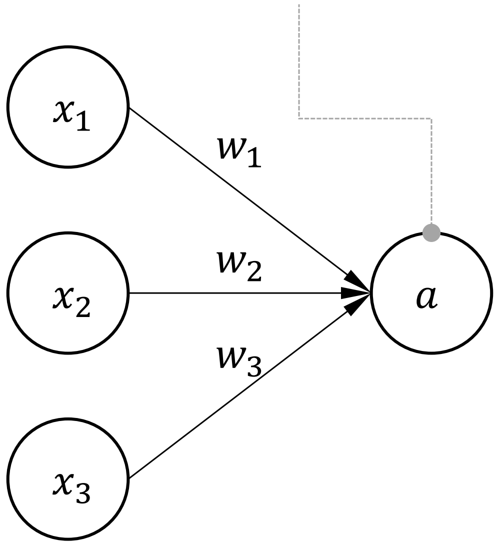


激活函数

- Rectifier函数（简称ReLU）

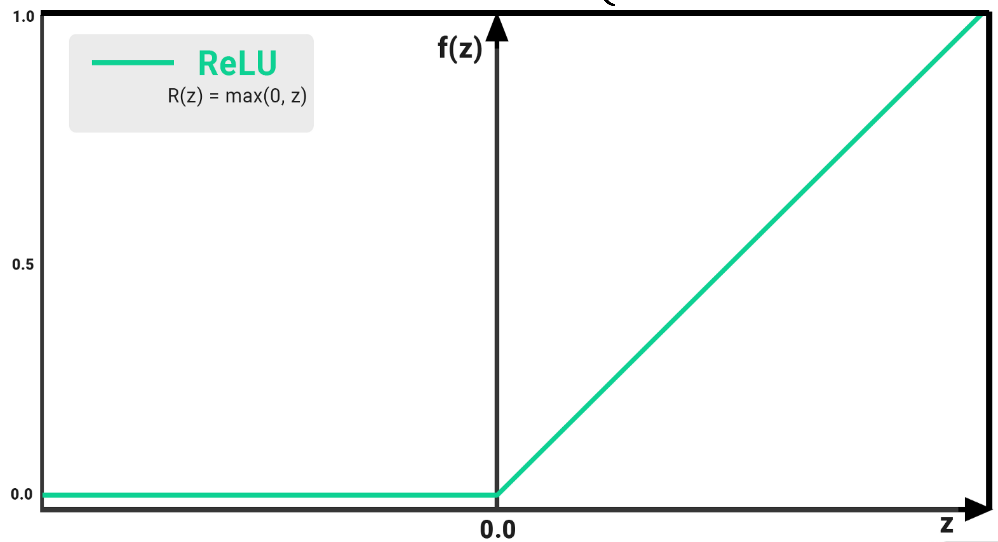
ReLU函数把负值输出设为0，正值输出不变。可用作特征选取（feature selection）和简化网络优化（optimisation）计算（后文会讲解）

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



ReLU函数

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

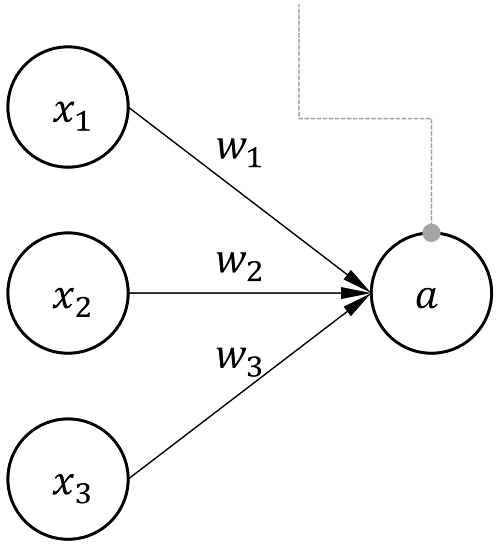


激活函数

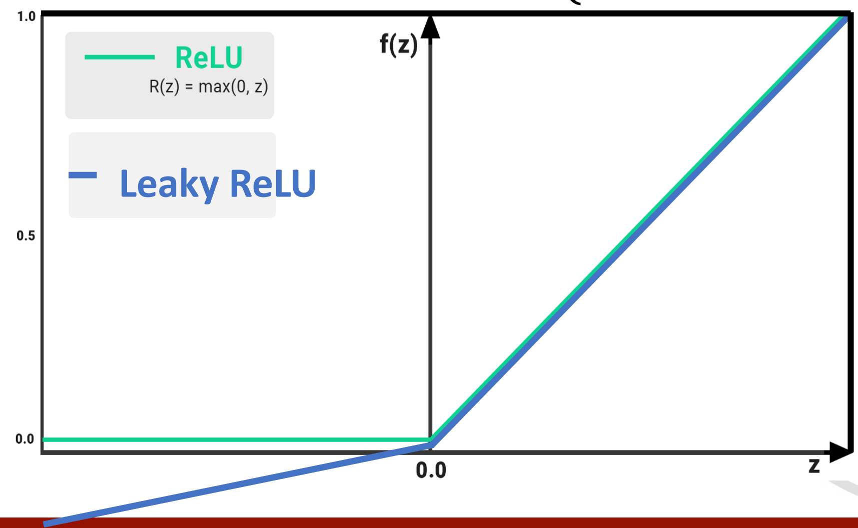
- Leaky ReLU函数

然而把负输出设为0会丢失输入信息导致，一个解决方案是采用Leaky ReLU函数。通过一个很小是正数α（如0.1、0.2）来控制斜率（slop），使得负输出的信息可以在激活值上体现。此外，Parametric ReLU（有参数的ReLU，也称PReLU）将α当作一个网络的参数，在后续优化过程中自行学习调整斜率。

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



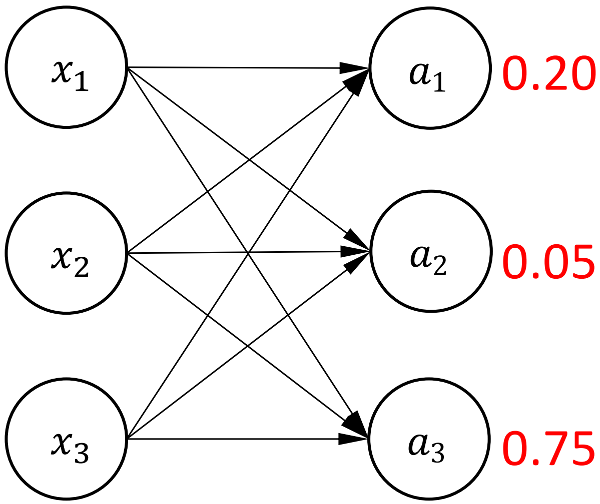
Leaky ReLU函数 $f(z) = \begin{cases} \alpha z, if\ x \le 0 \\ z\ , if\ x > 0 \end{cases}$



激活函数

- Softmax函数

一个网络有多个输出的时候，我们可以输出多个情况下的概率，实现多分类任务（multi-class classification），例如输出选择踢足球、打篮球、跑步的概率。Softmax是用在多分类任务下网络输出层上的函数，用以让所有激活值表示0 ~ 100%的概率，并让所有激活值之和为1，则100%。



给定一个输出向量 $\boldsymbol{z} = [z_1, z_2, z_3]$，向量长度 $K = 3$，我们可以获得如下激活向量$\boldsymbol{a} = [a_1, a_2, a_3]$：

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

Softmax函数先对每个输出值作指数函数（exponential function）操作，然后归一化（normalise）每个值，以使得所有激活值之和为 1，代表100%的概率。

激活函数

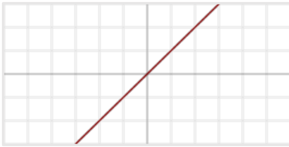
```python
# Softmax
z2 = torch.Tensor([[0.7000, 1.3500]])
softmax_torch = torch.nn.Softmax(dim=1)
a2 = softmax_torch(z2) # z2是神经网络未激活的值
print("Result torch softmax:", a2)

# define your own activation function
class ActSoftmax(torch.nn.Module):
    def __init__(self):
        super(ActSoftmax, self).__init__()

    def forward(self, x):
        return torch.exp(x)/torch.sum(torch.exp(x), dim=1, keepdim=True)

softmax_act = ActSoftmax()
a2_m = softmax_act(z2)
print("Result your own softmax:", a2_m)
```

```
Result torch softmax: tensor([[0.3430, 0.6570]])

Result your own softmax: tensor([[0.3430,
0.6570]])
```

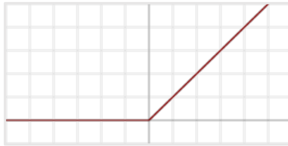
激活函数

- 更多激活函数

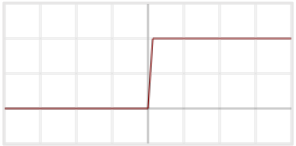
Identity



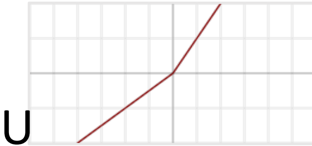
ReLU



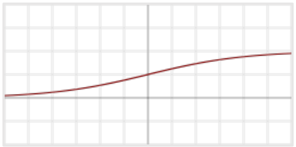
Binary Step



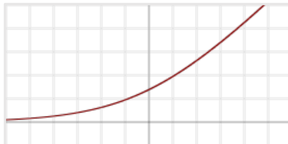
Leaky ReLU



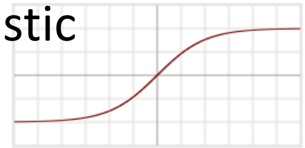
Sigmoid, Logistic



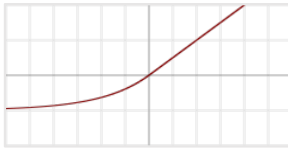
Softplus



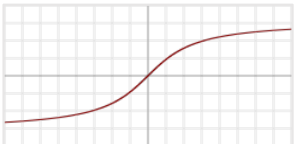
TanH



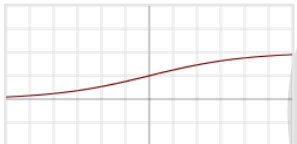
ELU



ArcTan



Softmax



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

- 单个神经元 Single Neuron
- 激活函数 Activation Functions
- **多层感知器 Multi-layer Perceptron**
- 损失函数 Loss Functions
- 优化 Optimisation
- 正则化 Regularisation
- 实现 Implementation

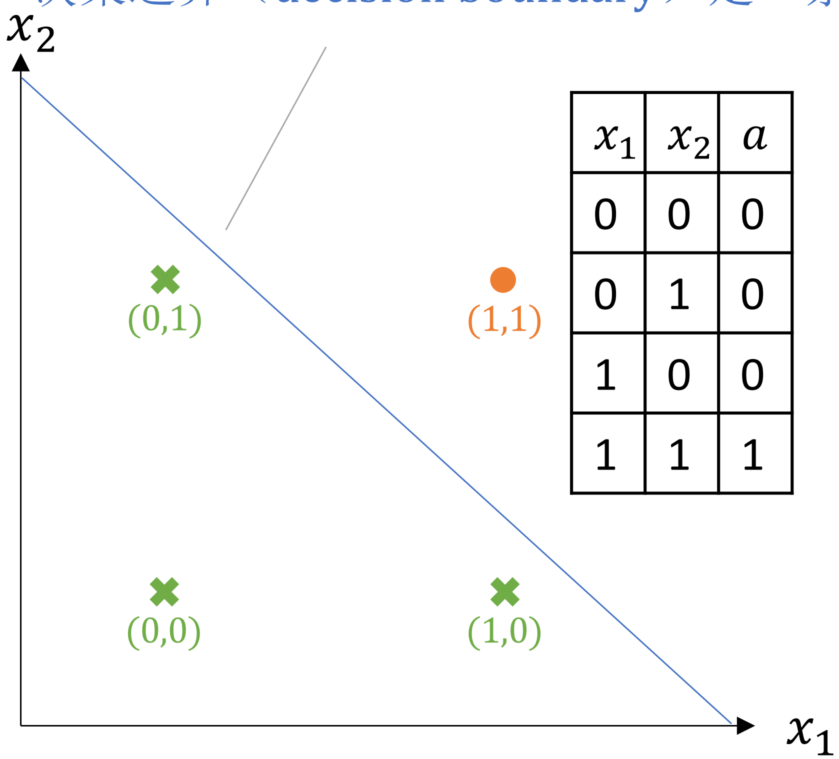
多层感知器

- 多层感知器（Multi-layer Perceptron，缩写MLP）
- 问题：之前的单层网络有什么问题?
- 答：XOR异或分类问题

多层感知器

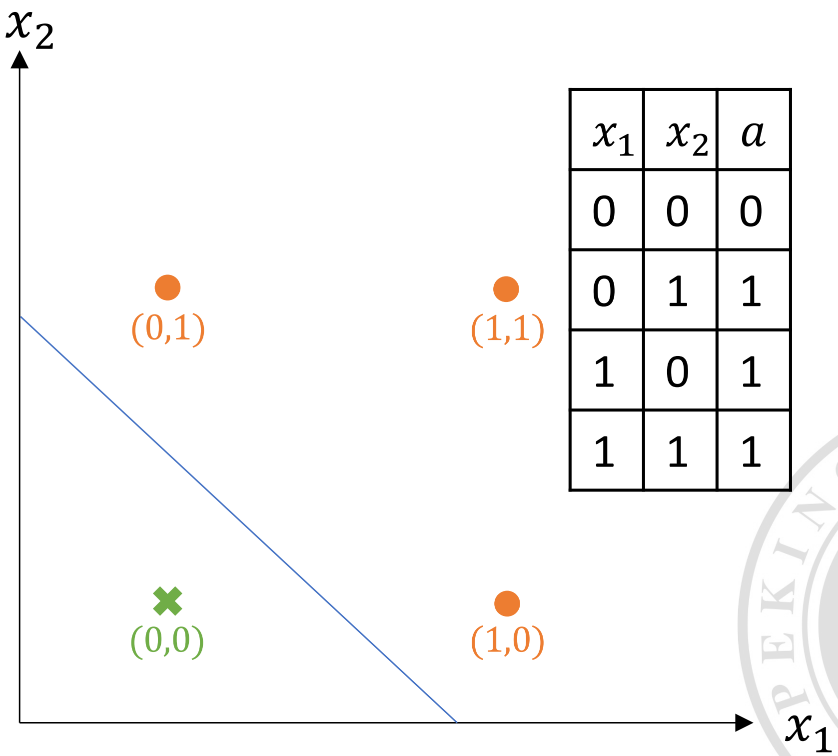
- XOR异或分类问题：以2个输入1个输出的单个神经元为例子

决策边界（decision boundary）是一条直线



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

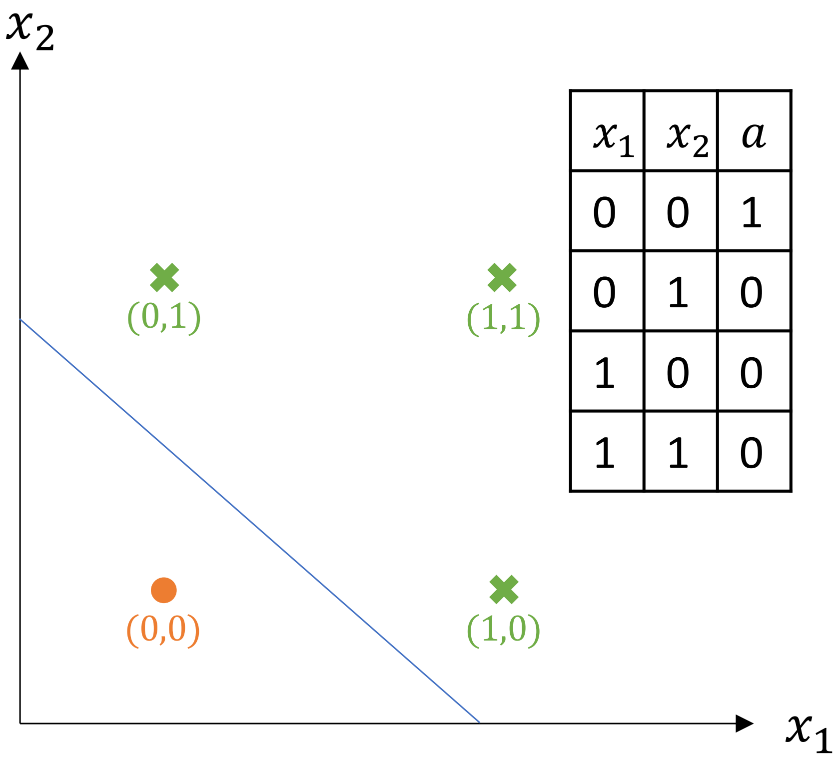
AND 与逻辑



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

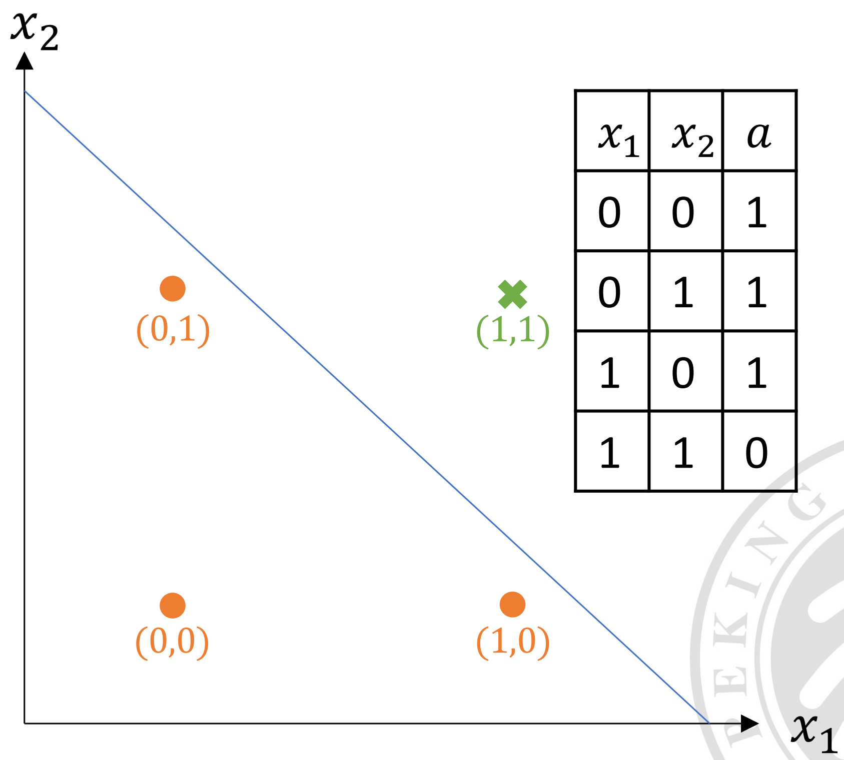
OR 或逻辑

- XOR异或分类问题：以2个输入1个输出的单个神经元为例子



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

NOR 或非逻辑



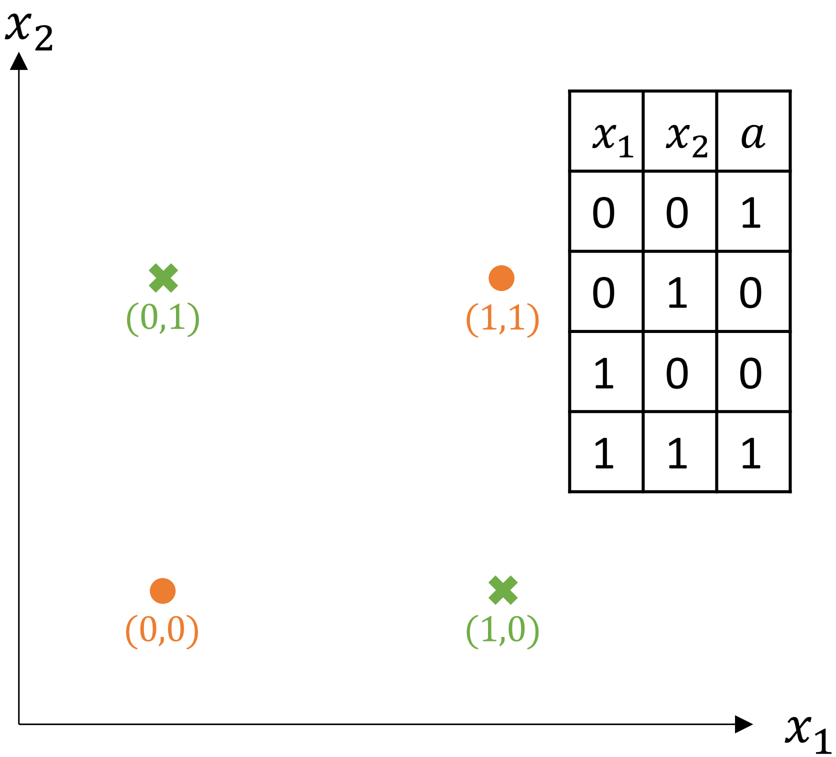
x_1	x_2	a
0	0	1
0	1	1
1	0	1
1	1	0

NAND 与非逻辑

多层感知器

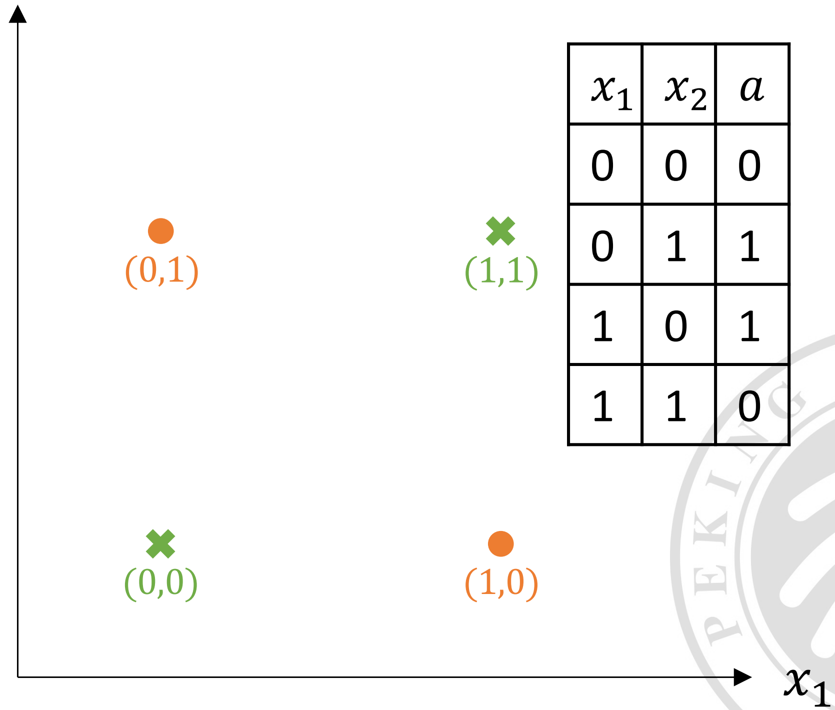
- XOR异或分类问题：以2个输入1个输出的单个神经元为例子

找不到一条线可以分开数据点，这是线性不可分问题（linear non-separable problem）



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

XNOR 异或非逻辑



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

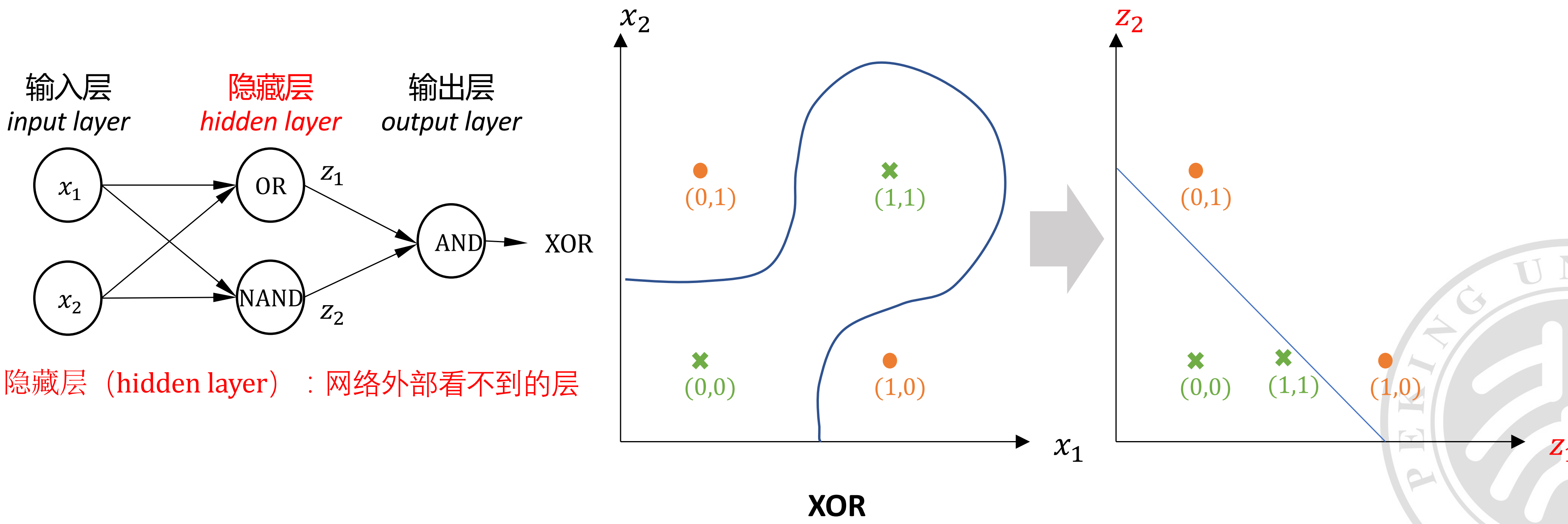
XOR 异或逻辑

多层感知器

- 两层网络

XOR异或分类问题，可以通过两层网络解决

在第一层输出的特征上进行分类

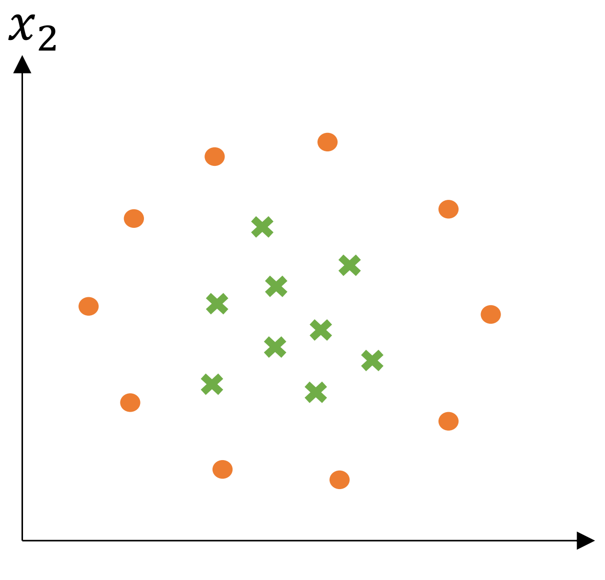


隐藏层（hidden layer）：网络外部看不到的层

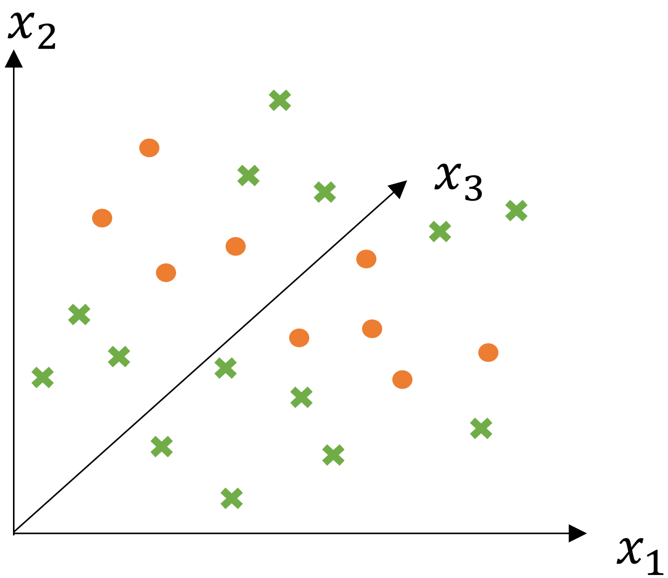
多层感知器

- 现实中更复杂的问题

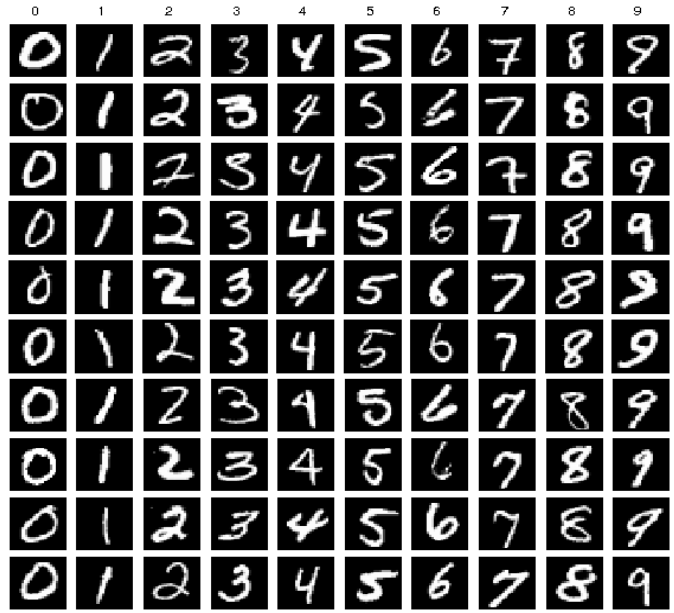
需要网络具备更好的能力（higher capacity）



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



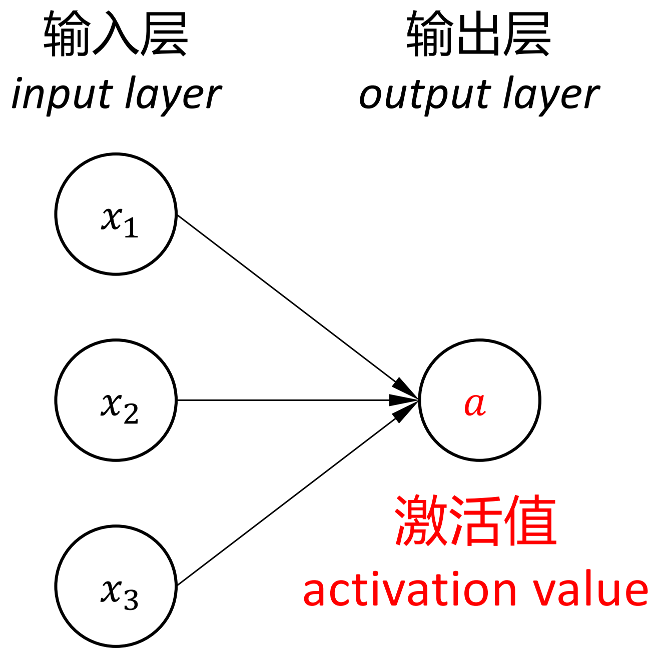
$$\boldsymbol{x} = [x_1, x_2, x_3]$$



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

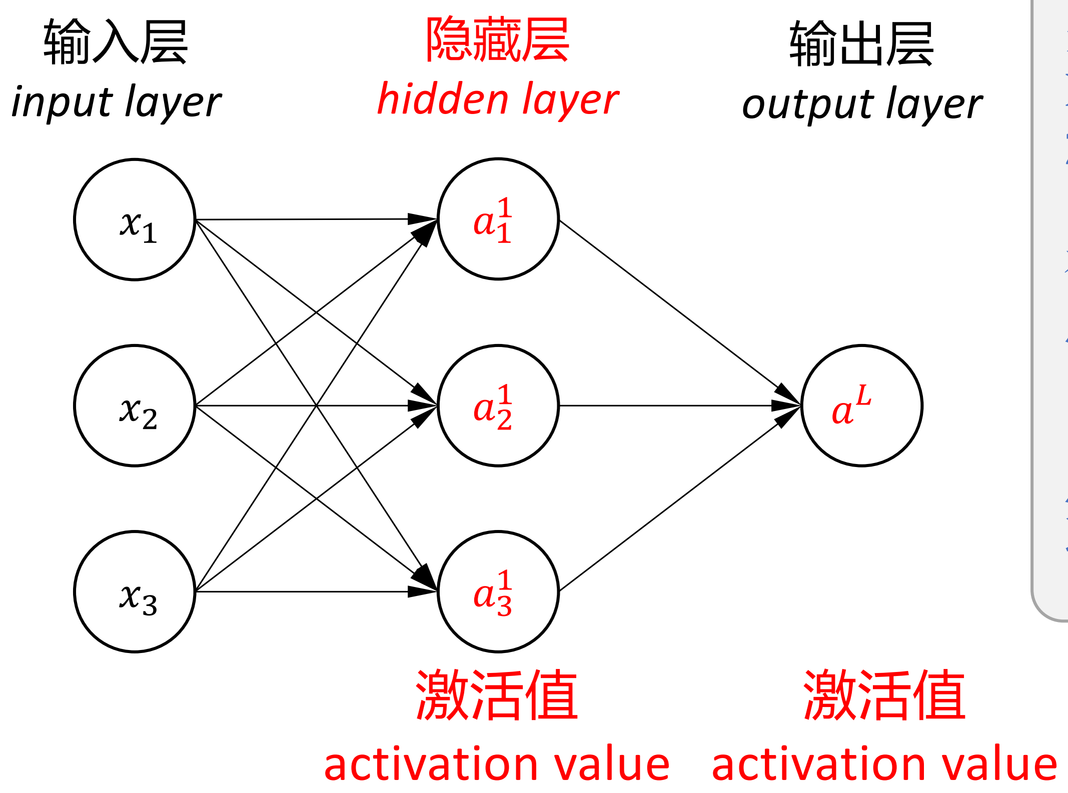
多层感知器

- 表达能力（Representation Capacity）



多层感知器

- 表达能力（Representation Capacity）



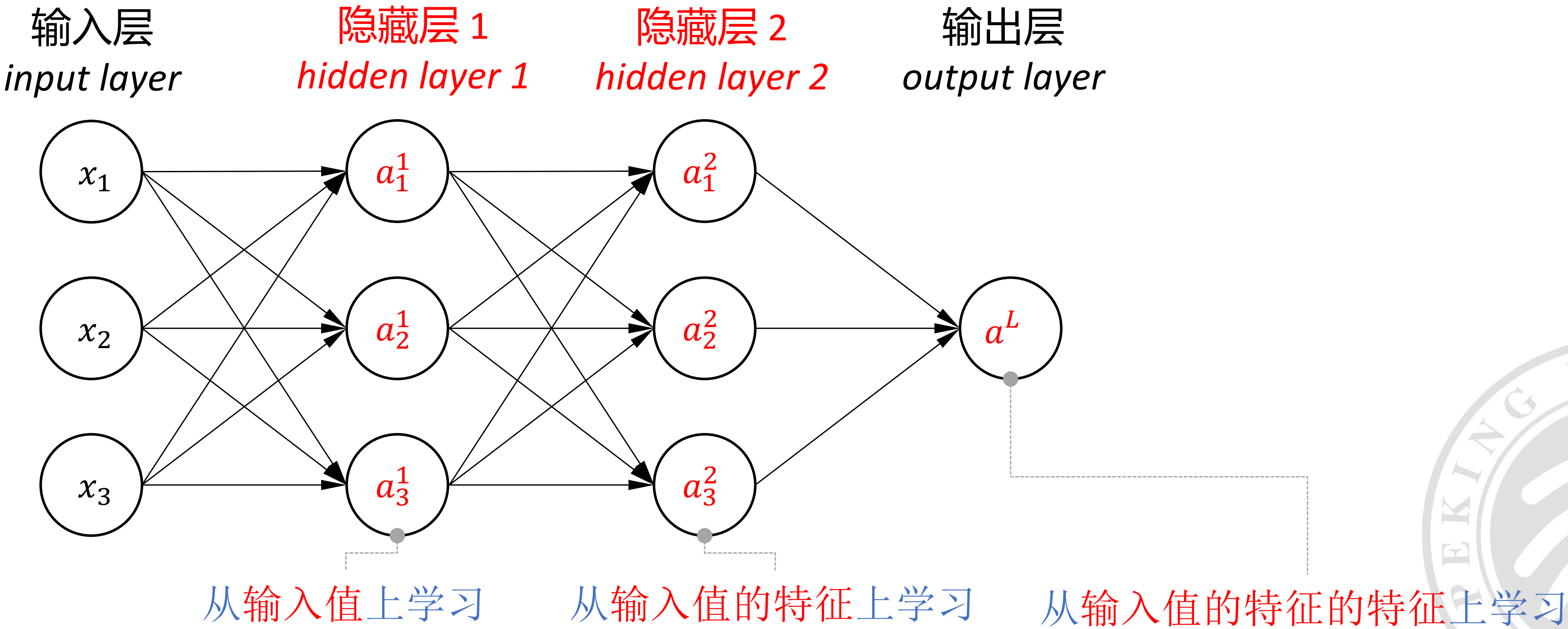
多层感知器（multi-layer perceptron，MLP）在单层全连接网络上拓展，至少有2层全连接层

在原有层上堆叠新的层，可以被认为是将原有层的输出值当作特征值来学习

因此，和单层全连接网络相比，MLP可以处理更复杂的输入数据，具有更好的表达能力（representation capacity）

多层感知器

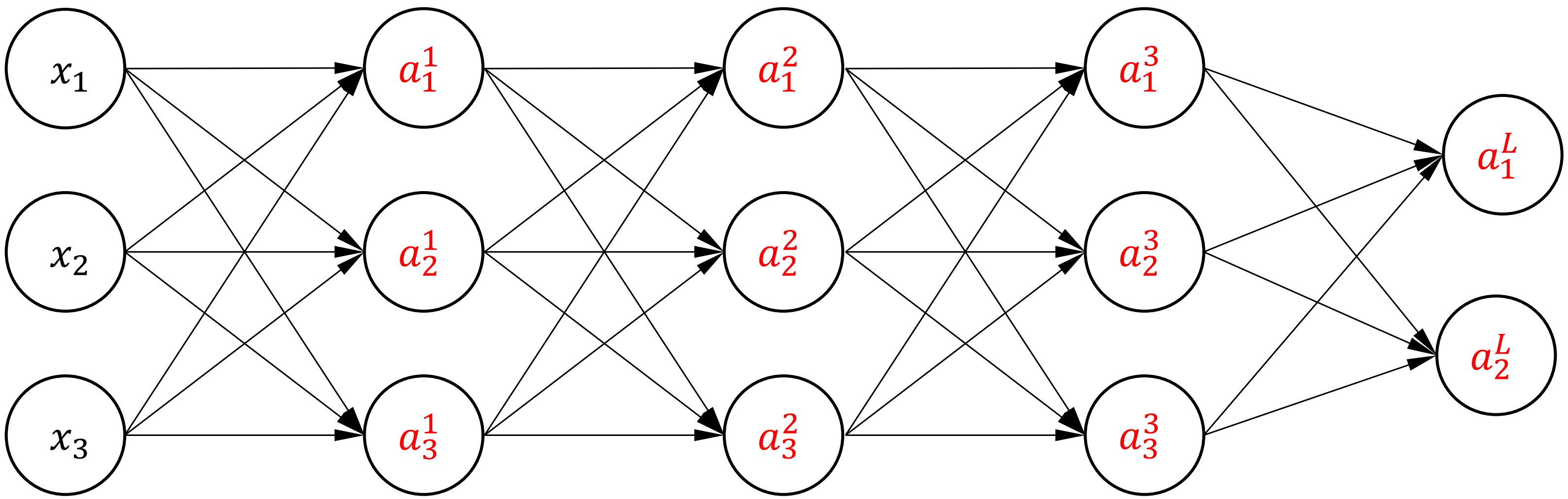
- 表达能力（Representation Capacity）



多层感知器

- 层次表达（Hierarchical Representation）

输入层 input layer | 隐藏层 1 hidden layer 1 | 隐藏层 2 hidden layer 2 | 隐藏层 3 hidden layer 3 | 输出层 output layer



a_k^l

层索引（layer index） $l = 1 \ldots L$ 从第一个隐藏层（1）到输出层（L）

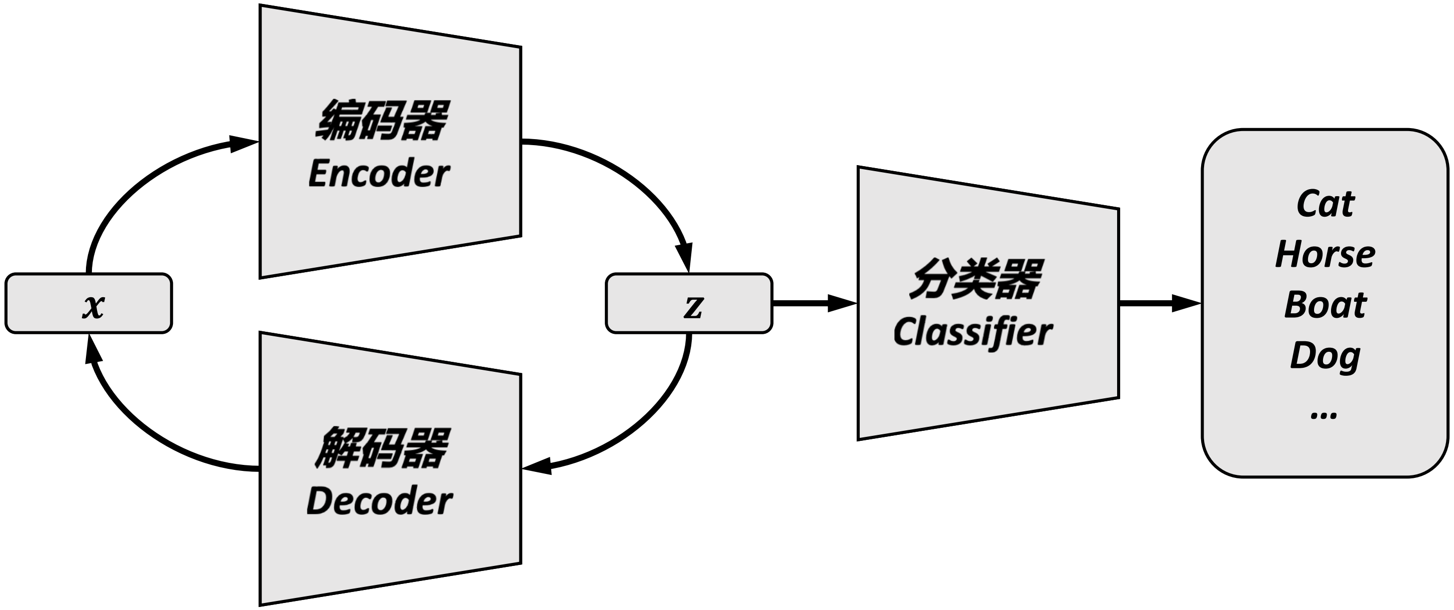
输入值作为输入层可以写为 $\boldsymbol{x} = \boldsymbol{a}^0$

输出索引（output index） $k = 1 \ldots K$ 表示某一层中神经元的序号

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

多层感知器

- 编码（Encode）vs. 解码（Decode）



深度学习的成功来自其强大的网络表达能力，优化过程可自动从输入数据x中学习出隐表达（latent representation）z。把可见的输入数据转为隐表达的过程称为编码（encoding），反之称为解码（decoding）

多层感知器

```
# MLP 多层感知机
# 构建序列式模型
layer_list = []
layer_list.append(Linear(in_features=3, out_features=3, bias=True))
layer_list.append(ReLU())
layer_list.append(Linear(in_features=3, out_features=3, bias=True))
layer_list.append(ReLU())
layer_list.append(Linear(in_features=3, out_features=3, bias=True))
layer_list.append(ReLU())
layer_list.append(Linear(in_features=3, out_features=3, bias=True))
layer_list.append(Softmax(dim=1))

mlp = Sequential(*layer_list)

x_input = torch.tensor([[100., 200., 300.]]) # 1x3
output = mlp(x_input)# 前向传播
print("Neural network output: ", output.shape)
```

```
Neural network output:  torch.Size([1, 3])
```

- 单个神经元 Single Neuron
- 激活函数 Activation Functions
- 多层感知器 Multi-layer Perceptron
- **损失函数 Loss Functions**
- 优化 Optimisation
- 正则化 Regularisation
- 实现 Implementation

损失函数

- 目的

- 损失函数（loss function）用来量化网络预测的输出（predicted output）和给定训练数据输出（ground truth）之间的误差（error，也称loss value）
- 损失函数用来设定优化神经网络参数（如权重、偏置）的目标
- 优化神经网络的过程，是通过更新其参数来使得误差尽可能小
- 梯度下降（gradient descent）是最常用的优化方法，后文将有介绍

我们无法手工设计网络中的每个权重/参数

我们需要定义损失函数作为“学习”的方向，来自动优化这些权重/参数

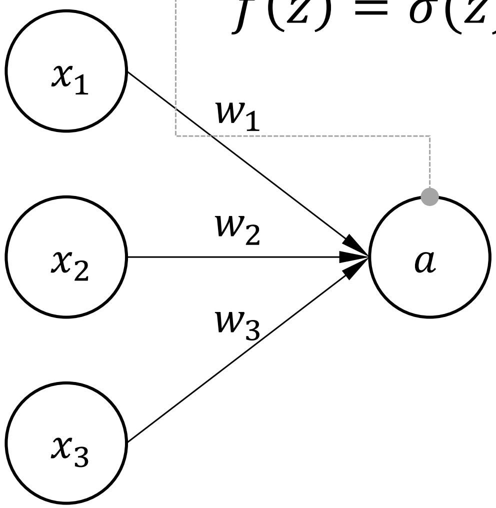
损失函数

- 逻辑回归损失（Logistic Regression Loss）

$$a = f(z) = f(x_1w_1 + x_2w_2 + x_3w_3 + b)$$

二值分类采用Sigmoid函数

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



给定一个输入x下

- a代表网络输出值
- y代表训练数据已知输出

- 当只有一个数据样本

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

- 当有 M 个数据样本

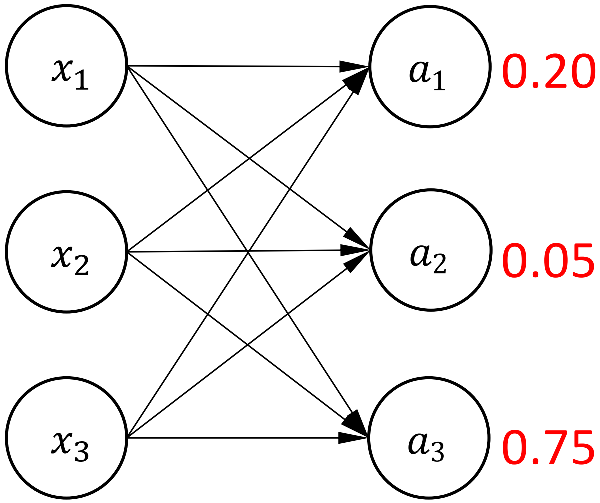
$$\mathcal{L} = \sum_{m=1}^{M} (y^m\, log(a^m) + (1-y^m)\log(1-a^m))$$

损失函数

- **交叉熵损失函数（Cross-Entropy Loss）**

多分类问题
网络输出一个向量

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



- 当只有一个数据样本

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

- 当有 M 个数据样本

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

损失函数

```
output = mlp(x_input)# 前向传播

# 多分类交叉熵
target = torch.tensor([[0, 1, 0]],dtype=torch.float) # 1x3

print("Target value:{}".format(target.numpy()))
print("Neural network output:{}".format(output.detach().numpy()))

loss_fn = torch.nn.CrossEntropyLoss()
l_ce = loss_fn(output, target)
print("Loss cross entropy:", l_ce)
```

```
Target value:[[0. 1. 0.]]
Neural network output:[[9.277383e-22 9.425122e-03 9.905749e-
01]]
Loss cross entropy: tensor(1.5386, grad_fn=<DivBackward1>)
```

损失函数

- $\mathcal{L}_p$ 范数（norm）

用来衡量一个向量数值的尺度（scale）大小

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

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

用量衡量两个向量之前差别的大小

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

损失函数

- 均方误差（Mean Squared Error，缩写MSE）

MSE是$\mathcal{L}_2$范数，往往用来衡量网络输出值$\boldsymbol{a}$和训练数据输出值$\boldsymbol{y}$的差别

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

- 当有 M 个数据样本

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

损失函数

- 平均绝对误差（Mean Absolute Error，缩写MAE）

MAE是 $\mathcal{L}_1$ 范数，也可用来衡量网络输出值$\boldsymbol{a}$和训练数据输出值$\boldsymbol{y}$的差别

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

- 当有 M 个数据样本

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

损失函数

```python
# Mean absolute error 平均绝对误差
def mae(output, target):
    return torch.mean(torch.abs(output - target))

# Mean squared error 平均平方误差
def mse(output, target):
    return torch.mean((output - target)**2)

y1 = torch.tensor([[1., 3., 5., 7.]])
y2 = torch.tensor([[2., 4., 6., 8.]])

# 计算MAE和MSE
l_mae = mae(y1, y2)
l_mse = mse(y1, y2)
print("Loss MAE: {} \nLoss MSE: {}".format(l_mae, l_mse))

mse_torch = torch.nn.MSELoss()
l_mse_torch = mse_torch(y1, y2)
print("Loss MSE torch: {}".format(l_mse_torch))
```

```
Loss MAE: 1.0
Loss MSE: 1.0
Loss MSE torch: 1.0
```

损失函数

- 更多损失函数…

- 单个神经元 Single Neuron
- 激活函数 Activation Functions
- 多层感知器 Multi-layer Perceptron
- 损失函数 Loss Functions
- **优化 Optimisation**
- 正则化 Regularisation
- 实现 Implementation

优化

- 目的

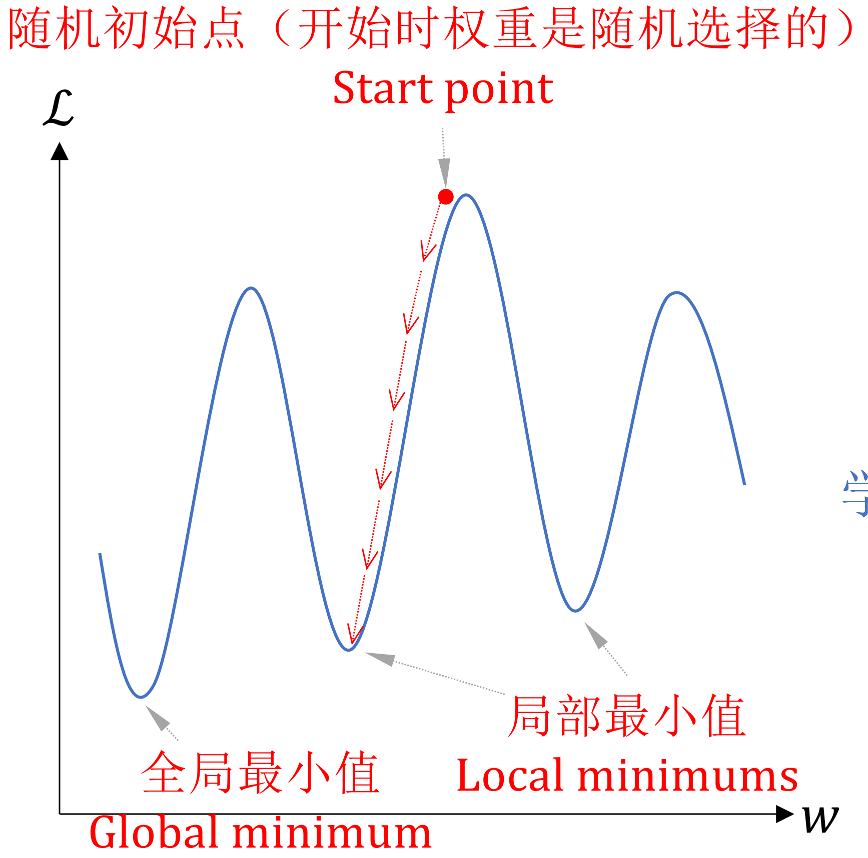
给定网络结构 $f(x;\theta)$、定义好的损失函数 $\mathcal{L}$和训练数据样本，训练网络的过程是通过更新网络参数θ（比如权重、偏置）以最小化误差的过程。

梯度下降（Gradient descent）是一个更新网络参数的方法，此外依然有很多优化方法存在，比如BFGS (L-BFGS) 和 conjugate gradient (CG)，但这些方法往往需要很多计算量，因此很少使用。

给定网络 $f(x;\theta)$ 和损失函数 $\mathcal{L}$，以获得好的参数 θ

优化

- 梯度下降（Gradient Descent）



假设整个网络只有一个权重w的情况下

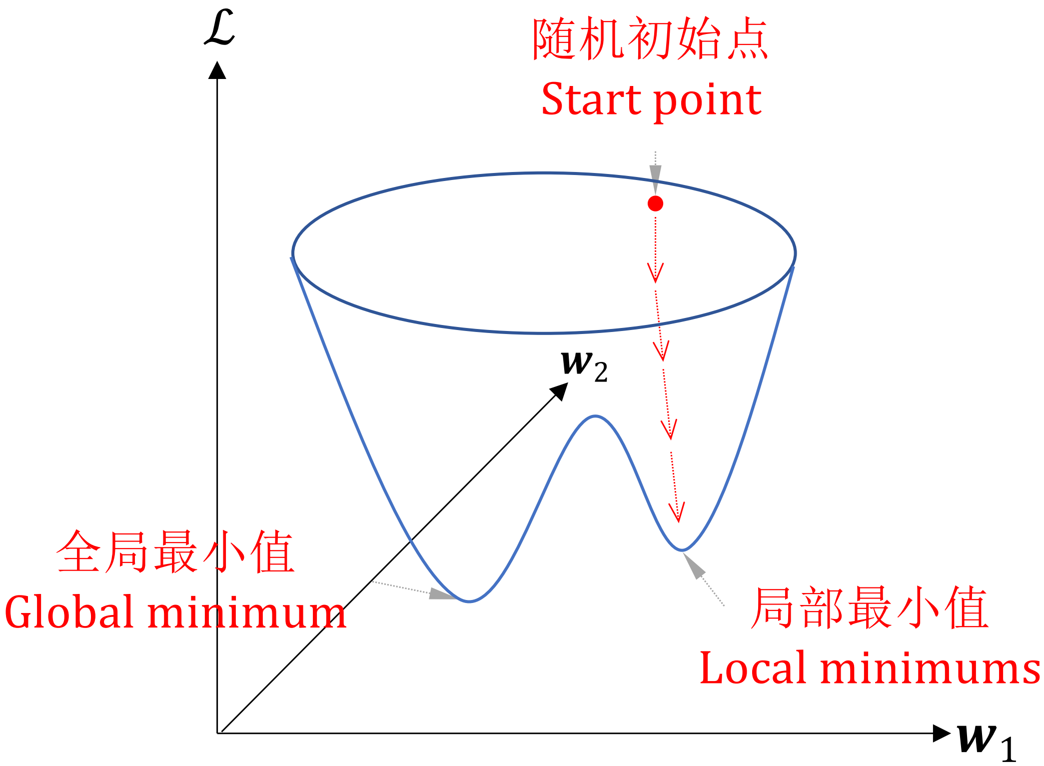
$$w := w - \alpha \frac{\partial \mathcal{L}}{\partial w}$$

学习率（ learning rate）
很小的值（如0.001）

对损失值 $\mathcal{L}$求w的偏导
作为梯度（Gradient）

优化

- 梯度下降（Gradient Descent）



假设整个网络只有两个权重的情况下

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

- 梯度下降（Gradient Descent）

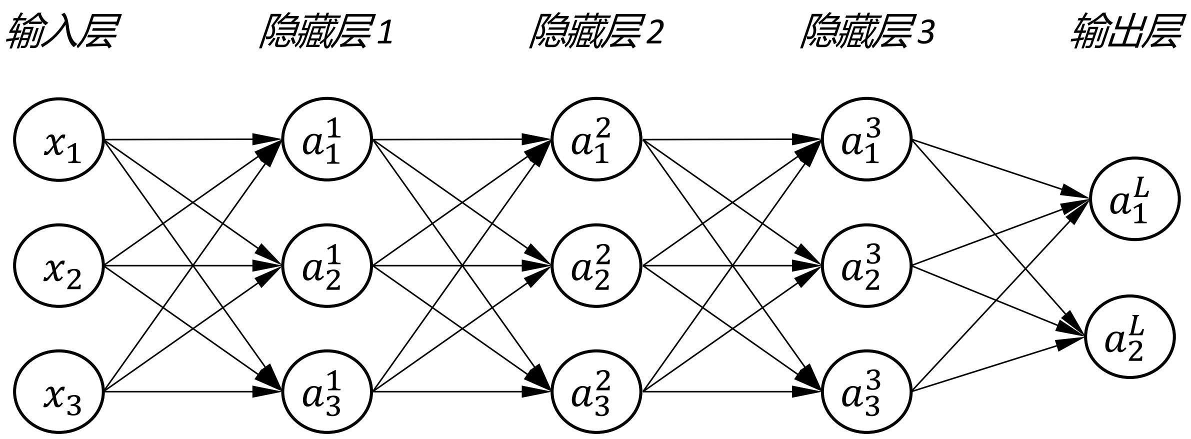
无论有多少参数，我们只需要计算出梯度 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ 即可

$$\boldsymbol{\theta} \coloneqq \boldsymbol{\theta} - \alpha \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$$

优化

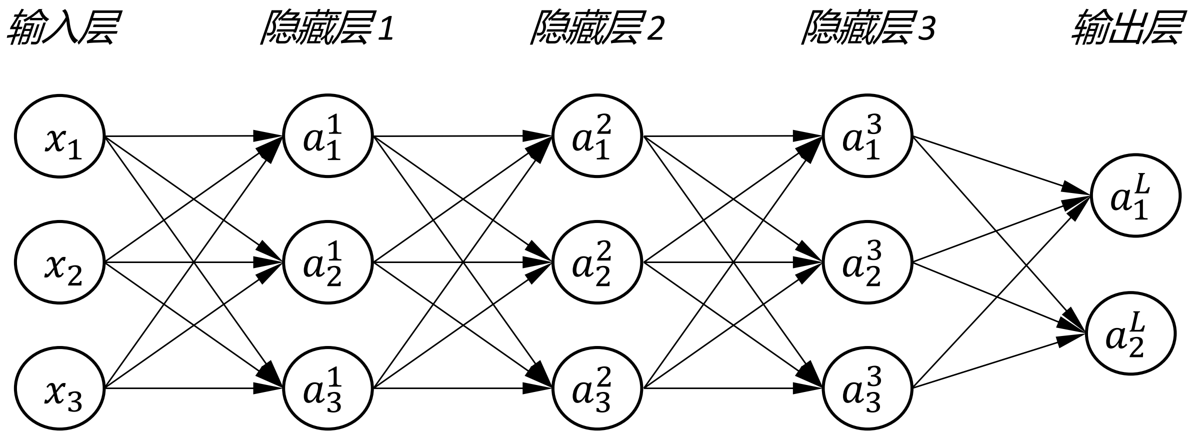
- 误差反向传播（Error Back-Propagation）

误差反向传播是用来计算网络中所有参数的梯度 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ 的方法。计算梯度时，引入对$\mathcal{L}$求输出值$\boldsymbol{z}$的偏导 $\delta = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}}$，作为中间结果（intermediate result)，基于这个中间结果来计算出每一层的梯度$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$。



优化

- 误差反向传播（Error Back-Propagation）



层索引（layer index） $l = 1 \ldots L$ 代表第一层隐藏层（1）到输出层（L）

输入层用 $\boldsymbol{x} = \boldsymbol{a}^0$ 来表示

我们通过这个简单模型和损失函数来讲解

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

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

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

优化

- 误差反向传播（Error Back-Propagation）：列格式（column format）教材常用

1. 已知

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

2. 则有如下求导

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

逐点相乘
Hadamard (element-wise) product

3. 输出层的中间结果 $\boldsymbol{l} = \boldsymbol{L}$

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

链式法则（chain rule）

4. 其他层的中间结果 $\boldsymbol{l} = \mathbf{1} \dots \boldsymbol{L} - \mathbf{1}$

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

5. 则有梯度

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

6. 更新参数

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

优化

- 误差反向传播（Error Back-Propagation）：行格式（row format）编程常用

1. 已知

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

2. 则有如下求导

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

3. 输出层的中间结果 $\boldsymbol{l} = \boldsymbol{L}$

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

4. 其他层的中间结果 $\boldsymbol{l} = \boldsymbol{1} \dots \boldsymbol{L} - \boldsymbol{1}$

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

5. 则有梯度

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

6. 更新参数

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

优化

- 梯度消失（Gradient Vanish）问题
  - $\delta^l = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{l+1}} \frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{z}^l} = \delta^{l+1} \boldsymbol{W}^{l+1^T} \circ (\boldsymbol{a}^l \circ (1 - \boldsymbol{a}^l))$

刚刚的例子中，中间结果δ中有一项$\left(\boldsymbol{a}^l \circ (1 - \boldsymbol{a}^l)\right)$，当激活输出$\boldsymbol{a}$接近0或者1时，中间结果$\delta$会变得很小。由于$\delta^l$又跟$\delta^{l+1}$有关，当反向传播时$\delta$比较小的话，会使得$\delta$越传播越小，使得靠近输入层的参数无法被更新，影响网络训练。

- 解决方法 1: 用 ReLU 代替 Sigmoid 函数（常见的方法）
- 解决方法 2: 逐层训练法（已经很少使用了）
- ...

- 梯度下降（Gradient Descent）

```
x_in = torch.tensor([[100., 200., 300.]])
target = torch.tensor([[1, 0, 0]],dtype=torch.float)
loss_fn = torch.nn.CrossEntropyLoss()

output = mlp(x_in)
l_ce = loss_fn(output, target)
l_ce.backward()

print("Gradient of layer 3's weights:\n{}".format(mlp[-2].weight.grad))
```

```
Gradient of layer 3's weights:
tensor([[-6.1040e-20, -0.0000e+00, -6.2719e-20],
        [-1.6209e-01, -0.0000e+00, -1.6655e-01],
        [ 1.6210e-01,  0.0000e+00,  1.6655e-01]])
```

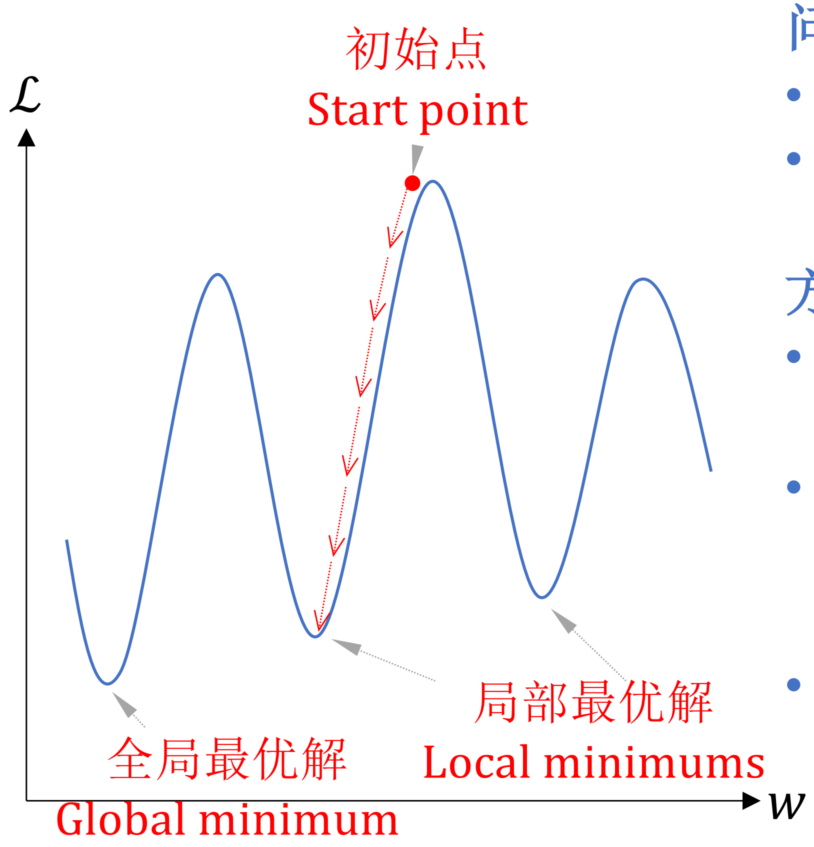
Autograd in PaddlePaddle:
https://www.paddlepaddle.org.cn/documentation/docs/zh/2.2/guides/01_paddle2.0_introduction/basic_concept/autograd_cn.html

```python
opt = torch.optim.SGD(mlp.parameters(), lr=0.1)
print("Before training, network layer 3's weights is: {}".format(mlp[-2].weight))

mlp.train()
for i in range(100):
    output = mlp(x_in)
    l_ce = loss_fn(output, target)

    l_ce.backward()
    opt.step()
    opt.zero_grad()

print("After training, network layer 3's weights is: {}".format(mlp[-2].weight))
```

```
Before optimization, network layer1's weights:
[[ 0.00299972 -0.01064047  0.03229429]
 [ 0.02116016  0.00386932  0.00153159]
 [ 0.02340171 -0.00316979 -0.00069778]]
After optimization, network layer1's weights:
[[ 0.00299972 -0.01064047  0.03229436]
 [ 0.02116015  0.00386932  0.00153173]
 [ 0.02340169 -0.00316979 -0.00069757]]
```

优化

- 随机梯度下降（Stochastic Gradient Descent）



问题

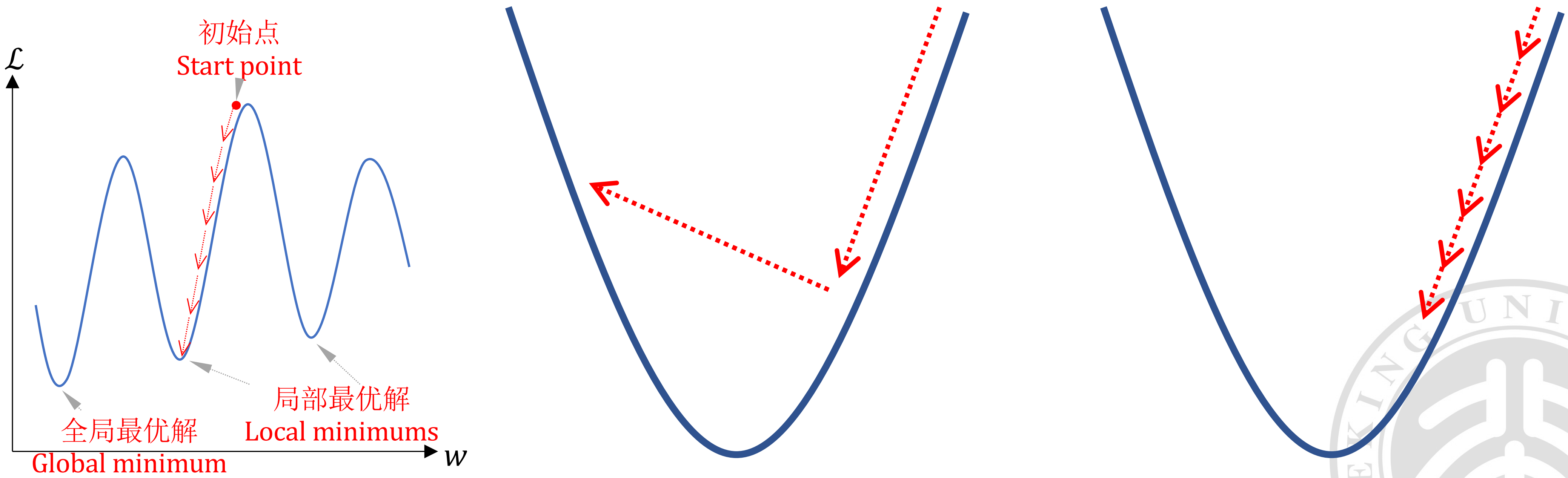
- 基于梯度下降的误差反向传播反复更新参数
- 但每次更新都用所有训练样本来计算误差$\mathcal{L}$会很慢！

方法

- 每次更新不用所有训练样本，而是从中随机选取一批（batch）数据来计算误差
- 这批数据称为“mini-batch”，样本的数量称为批大小（batch size），如每次选取32个数据样本，64个数据样本
- 通过多次更新参数，“mini-batch”可以覆盖整个训练数据集，一个epoch称为覆盖一次整个训练数据集

优化

- 自适应学习率（Adaptive Learning Rate）

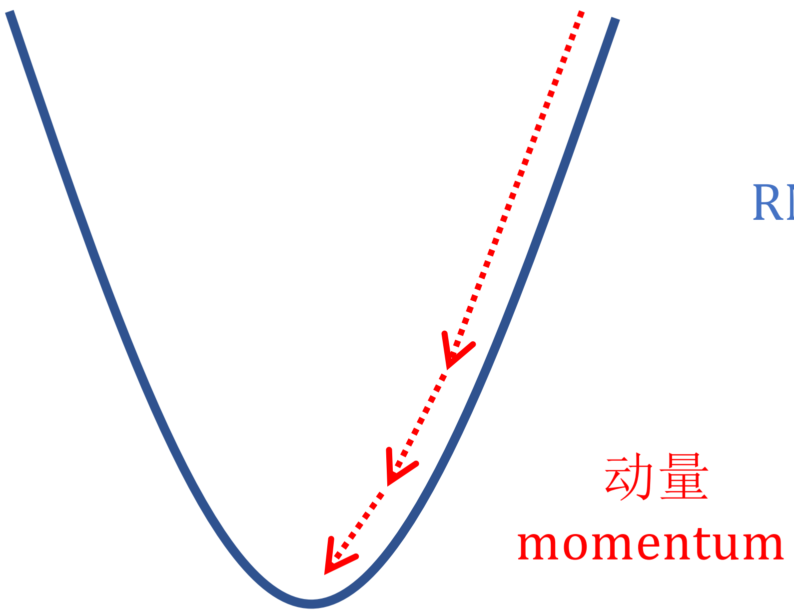


学习率太大
难以收敛到鞍点（saddle）

学习率太小
训练太慢

优化

- 自适应学习率（Adaptive Learning Rate）



RMSProp, Adagrad, Adam, AMSGrad, AdaBound

优化

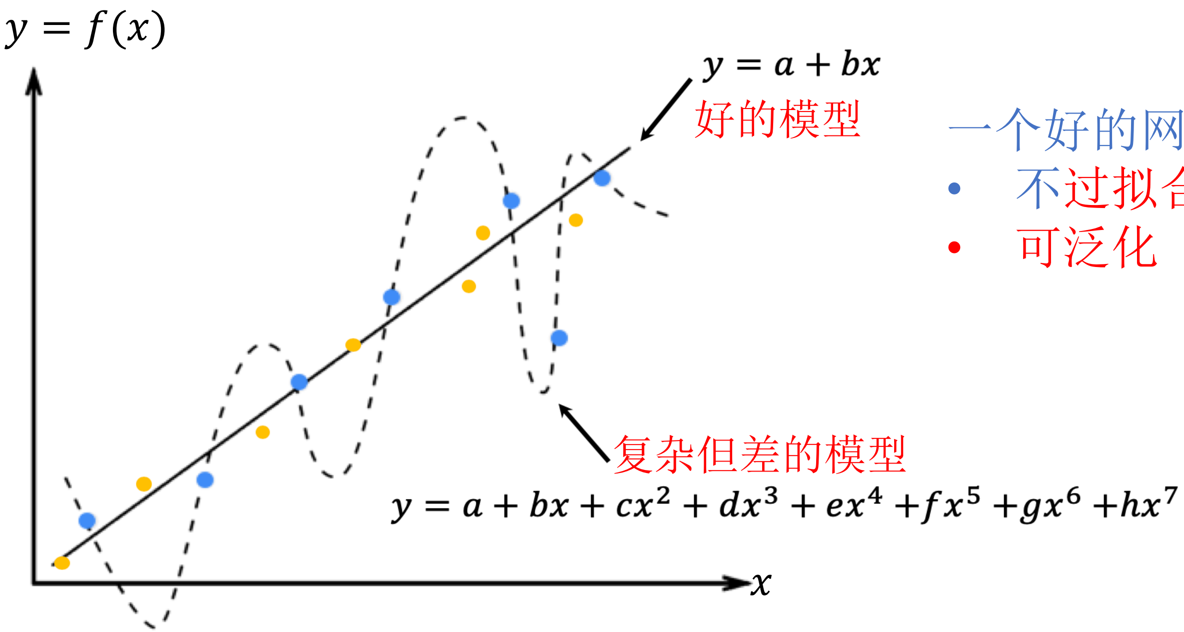
- 超参数选择（Hyper-Parameter Selection）

● 训练数据 vs ● 测试数据

Training Data Testing Data

训练数据只用于训练时使用

测试数据只用于测试时使用



一个好的网络模型 $f(x;\ \theta)$ 要满足：

- 不过拟合（overfit）到训练数据上
- 可泛化（generalise）到测试数据上

欠拟合（underfitting）只需要更强大的网络就可解决

优化

- 超参数选择（Hyper-Parameter Selection）

训练数据 vs 验证数据（Validation Data） vs 测试数据

超参数包括模型和训练的各种设置，例如：

- 神经网络层数
- 每层的神经元数
- 激活函数
- 损失函数
- 批大小
- 训练epoch数
- …

训练集训练模型 ⇄ 在验证集上验证超参数的性能，选择超参 → 最终在测试集上测试最终的性能

不能用测试集来验证超参数性能，这是作弊

优化

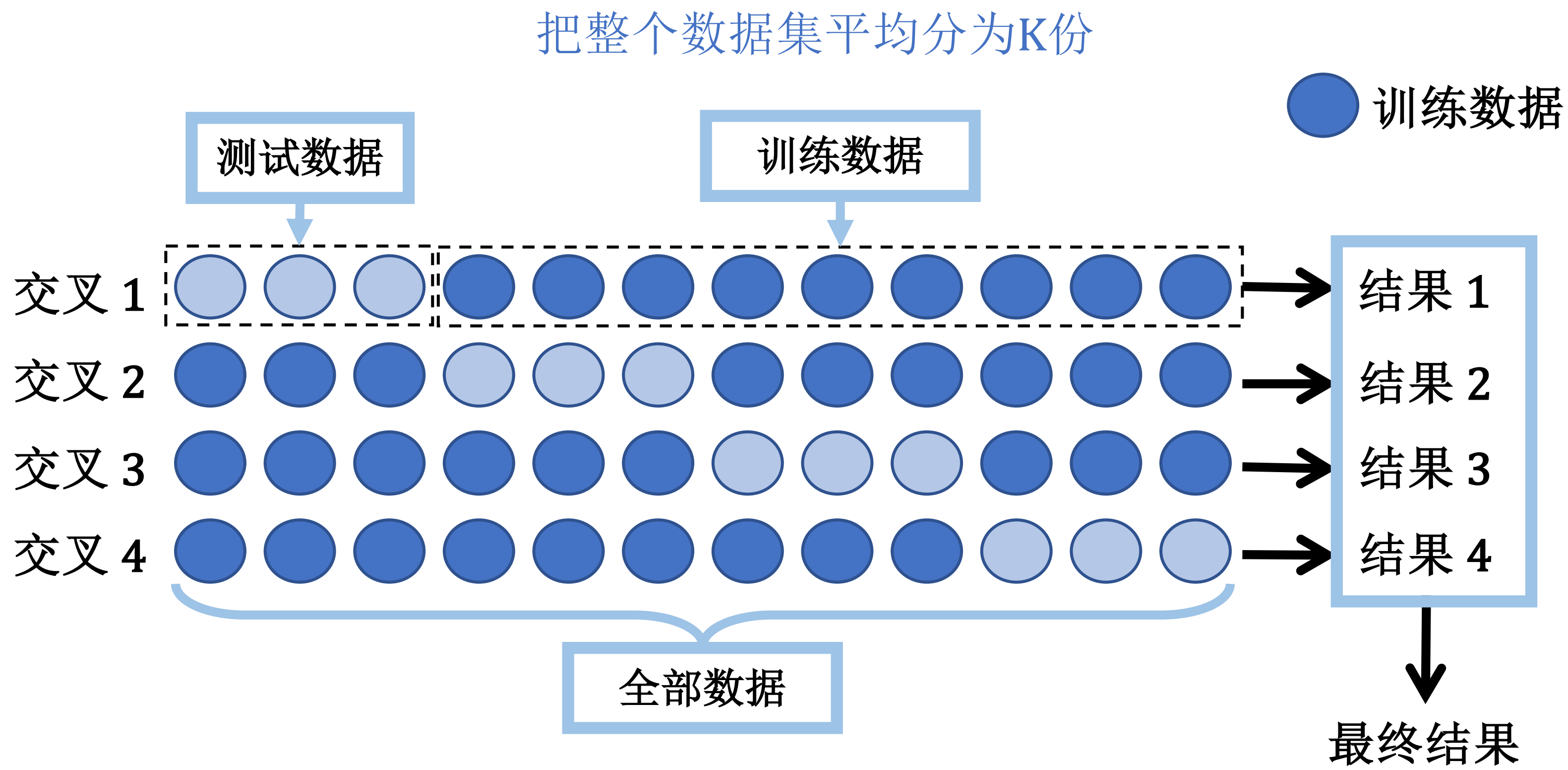
- 超参数选择（Hyper-Parameter Selection）& 交叉验证（Cross Validation）

训练数据 vs ~~验证数据（Validation Data）~~ vs 测试数据

对于一些稀有数据，拿出一些数据作为验证集会比较浪费
问：如何不需要验证集来选择超参数？

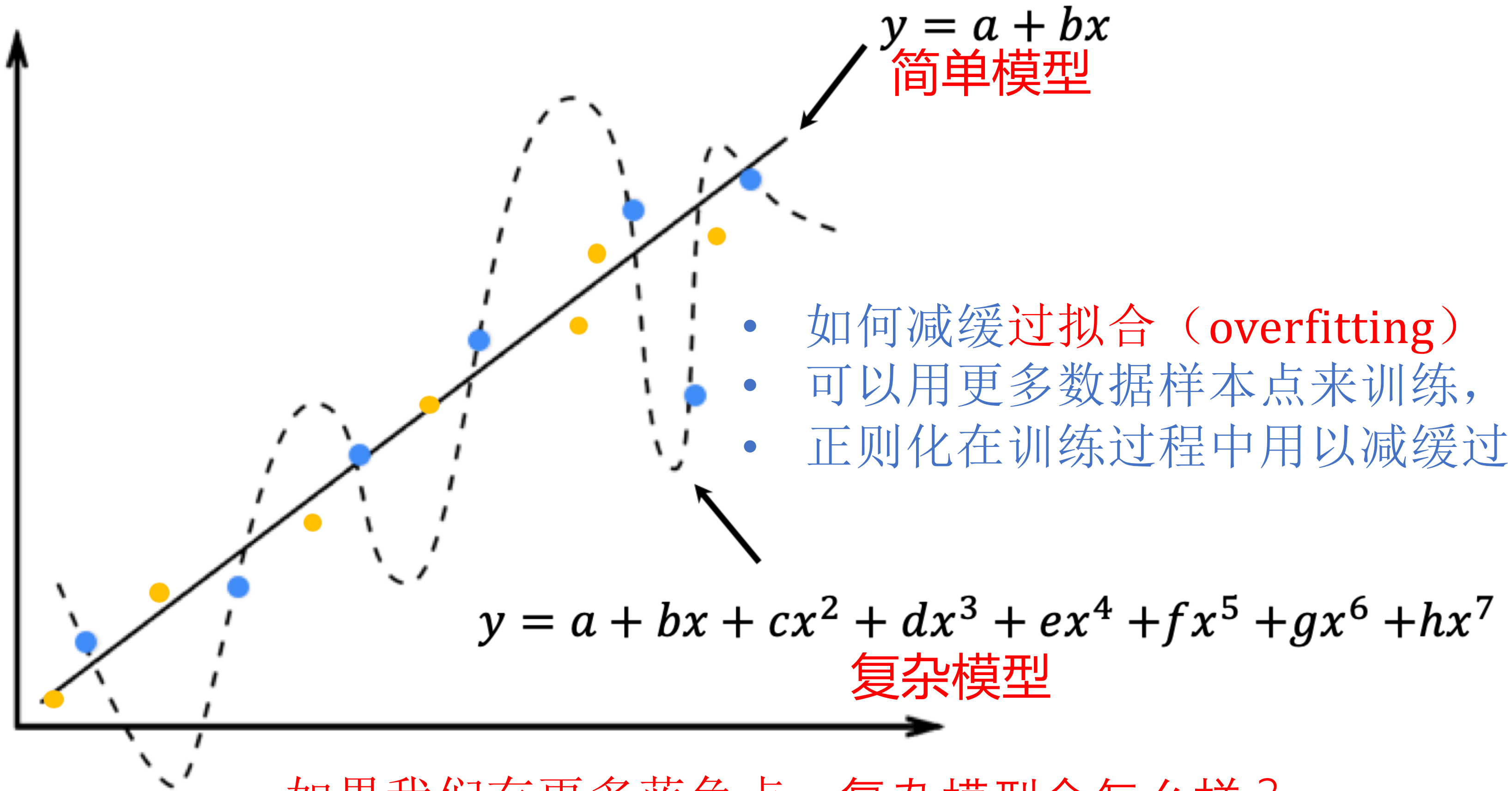
优化

- 超参数选择（Hyper-Parameter Selection）& K-Fold 交叉验证（Cross Validation）



- 单个神经元 Single Neuron
- 激活函数 Activation Functions
- 多层感知器 Multi-layer Perceptron
- 损失函数 Loss Functions
- 优化 Optimisation
- 正则化 Regularisation
- 实现 Implementation

正则化



- 如何减缓过拟合（overfitting）
- 可以用更多数据样本点来训练，但获取数据需要成本
- 正则化在训练过程中用以减缓过拟合

如果我们有更多蓝色点，复杂模型会怎么样？

正则化

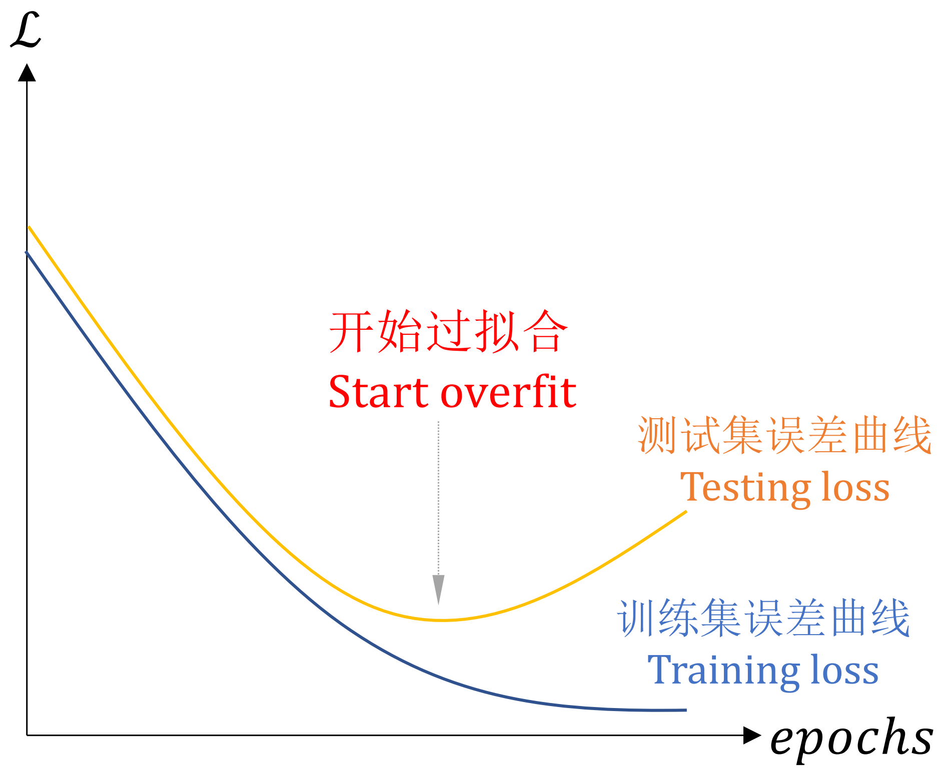
- 数据增强（Data Augmentation）获得更多数据

图像数据可以采用的数据增强方法：水平对称翻转（horizontal flipping）、旋转（rotating）、平移（shifting）、缩放（zooming）等

正则化

- 提前停止法（Early Stopping）

在网络开始过拟合的时候，提前停止训练



正则化

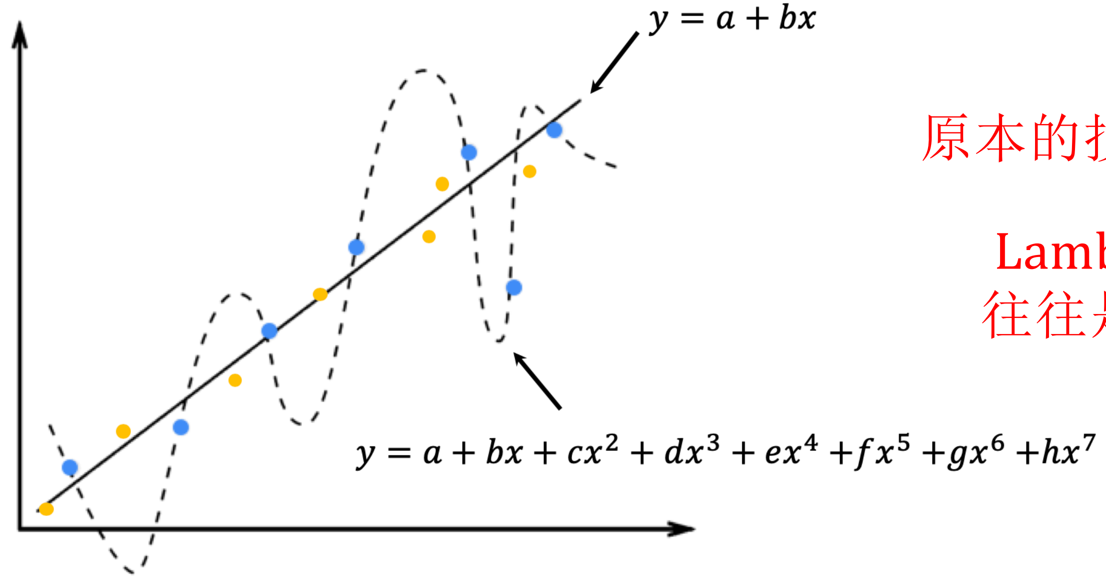
- 权重衰减（Weight Decay）

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

原本的损失值

正则化项

Lambda: 控制正则化项的强度
往往是一个很小的值，如0.001



若c, d, e, f, g, h 这些权重数值比较小会怎么样？

正则化

- $\mathcal{L}_1$ 范数 norm

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

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

- $\mathcal{L}_2$ 范数 norm

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

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

注意：权重衰减只用于权重weight，不用于偏置bias

正则化

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

假设只有 $\boldsymbol{w_1}$ 和 $\boldsymbol{w_2}$ 两个权重时

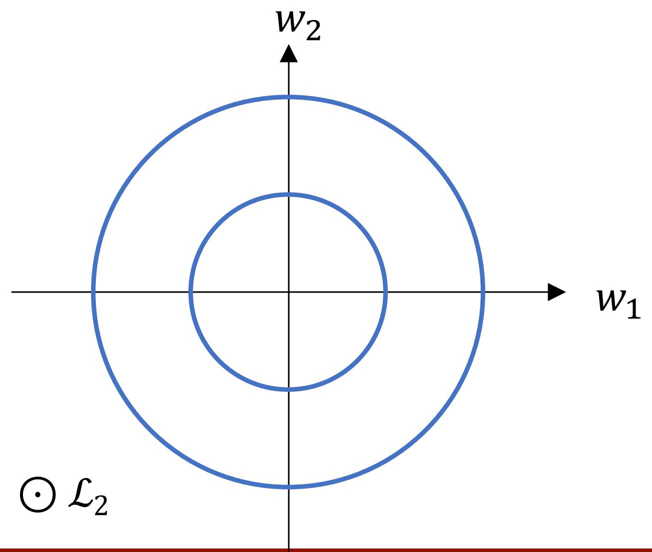
$\mathcal{L}_2$ 惩罚（penalty）是：

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

给定一个特定的 $\mathcal{L}_2$ 值， $\mathcal{L}_2$是圆形

$1 = {w_1}^2 + {w_2}^2$ 是半径为1的圆形

$4 = {w_1}^2 + {w_2}^2$ 是半径为2的圆形



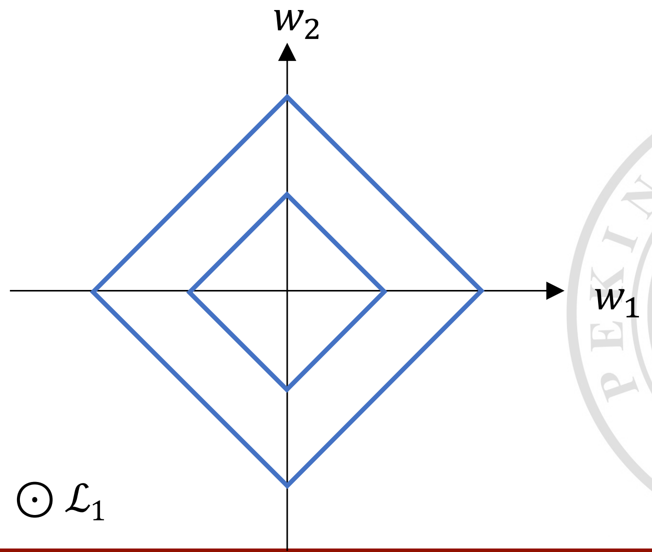
$\mathcal{L}_1$ 惩罚（penalty）是：

$\mathcal{L}_1 = |w_1| + |w_2|$

给定一个特定的 $\mathcal{L}_1$ 值， $\mathcal{L}_1$是正方形

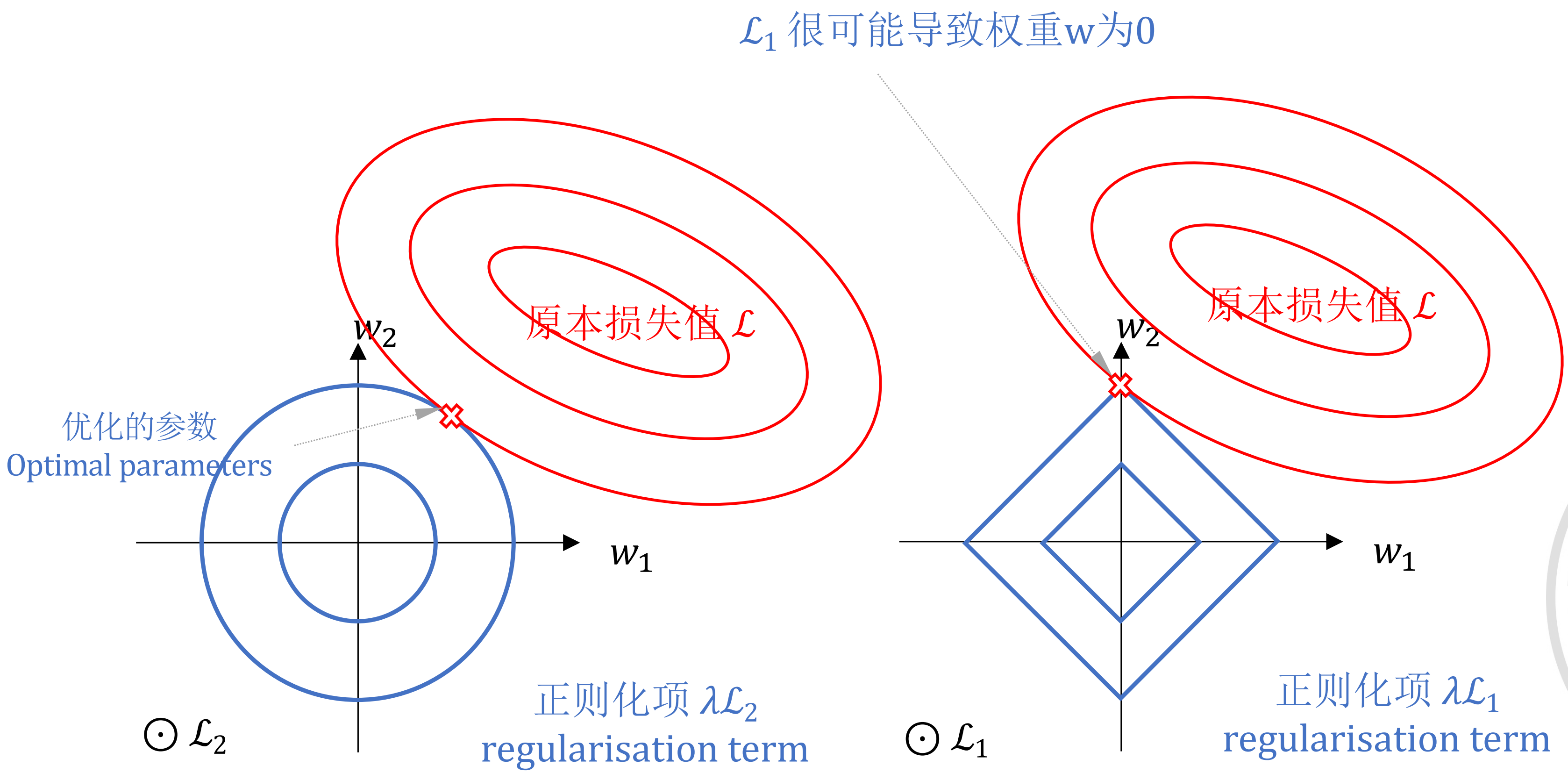
$1 = |w_1| + |w_2|$ 是对角线为2的正方形

$2 = |w_1| + |w_2|$ 是对角线为4的正方形



正则化

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



正则化

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

另外一种解释：

- 神经网络参数往往小于1，因此当用$\mathcal{L}_2$时，两个小于1的数值相乘，得到的数会更小，会比$\mathcal{L}_1$小。
- $\mathcal{L}_1$对小的数值产生的惩罚比 $\mathcal{L}_2$要大。

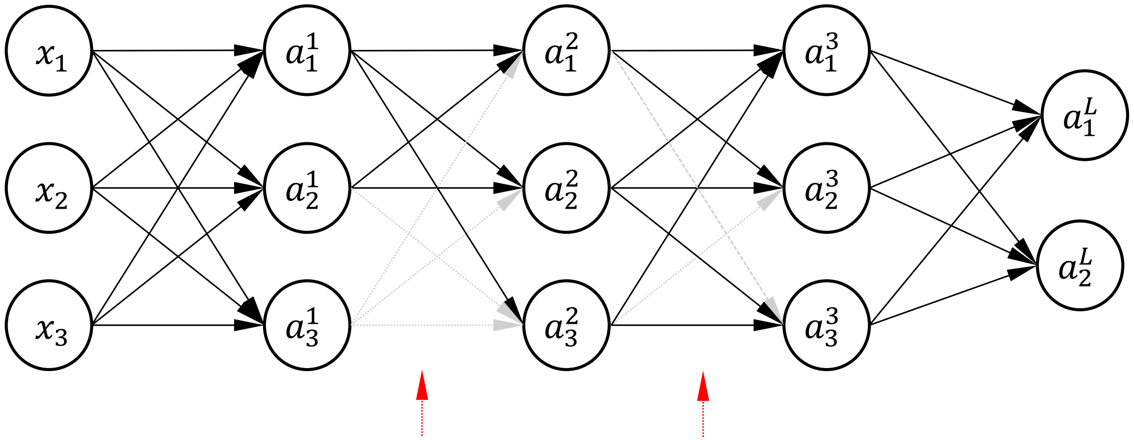
$$|0.5| > 0.5^2$$

正则化

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

$\mathcal{L}_1$ 会有更多为0的权重，我们称之为有稀疏特性（sparse property），可以让网络具备选择特征（feature selection）的能力。

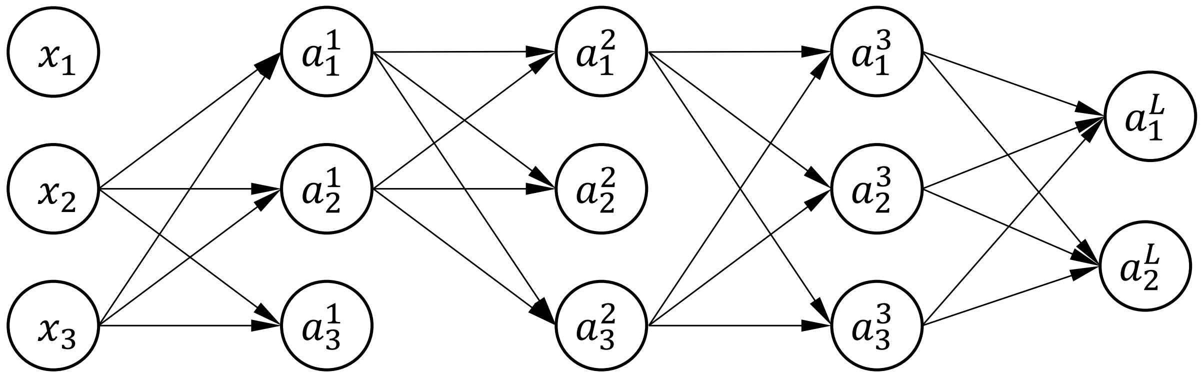
（ReLU激活函数也有类似的能力）



将一些输入特征对应的权重设为0或者很小的值，则表示该输入对输出影响没有或很小

正则化

- Dropout



实际中，每层都会有大量的神经元

- 包含大量神经元的神经网络使其很容易过拟合
- Dropout在训练过程中按一个比例随机对隐藏输出置0，则随机断开神经元的连接
- 测试/正常使用的时候，不再随机置0

正则化

- Dropout

  - 根据误差反向传播法，当某些隐藏层输出为0，则其对应的梯度为0。也就是说，这会使得只有部分权重会被更新

  - Dropout法可以认为在训练时把一个大网络“拆分”为很多子网络（sub-networks），在测试时全部子网络一起“投票”，甚至可以获得更好的结果，这种方法叫做集成学习（ensemble learning）

- 单个神经元 Single Neuron
- 激活函数 Activation Functions
- 多层感知器 Multi-layer Perceptron
- 损失函数 Loss Functions
- 优化 Optimisation
- 正则化 Regularisation
- 实现 Implementation

实现

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

import numpy as np
```

```python
# 加载数据集,numpy格式
X_train = np.load('./mnist/X_train.npy')
y_train = np.load('./mnist/y_train.npy')
X_val = np.load('./mnist/X_val.npy')
y_val = np.load('./mnist/y_val.npy')
X_test = np.load('./mnist/X_test.npy')
y_test = np.load('./mnist/y_test.npy')
```

实现

```python
# 定义MNIST数据集类
class MNISTDataset(Dataset):#继承Dataset类

    def __init__(self, data=X_train, label=y_train):
        '''
        Args:
            data: numpy array, shape=(N, 784)
            label: numpy array, shape=(N, 10)
        '''
        self.data = data
        self.label = label

    def __getitem__(self, index):
        '''
        根据索引获取数据,返回数据和标签,一个tuple
        '''
        data = self.data[index].astype('float32') #转换数据类型
        label = self.label[index].astype('int64') #转换数据类型
        return data, label

    def __len__(self):
        '''
        返回数据集的样本数量
        '''
        return len(self.data)
```

```python
# 定义模型
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 800)
        self.fc2 = nn.Linear(800, 800)
        self.fc3 = nn.Linear(800, 10)

    def forward(self, x):
        x = x.view(-1, 784)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.log_softmax(self.fc3(x), dim=1)
        return x

# 实例化模型
model = Net()
```

实现

```python
# 定义损失函数
criterion = nn.CrossEntropyLoss()
# 定义优化器
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

# 定义数据加载器
train_loader = DataLoader(MNISTDataset(X_train, y_train), \
                          batch_size=64, shuffle=True)
val_loader = DataLoader(MNISTDataset(X_val, y_val), \
                          batch_size=64, shuffle=True)
test_loader = DataLoader(MNISTDataset(X_test, y_test), \
                          batch_size=64, shuffle=True)
```

```python
# 定义训练参数
EPOCHS = 10
# 训练模型
for epoch in range(EPOCHS):
    # 训练模式
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        # 梯度清零
        optimizer.zero_grad()
        # 前向计算
        output = model(data)
        # 计算损失
        loss = criterion(output, target)
        # 反向传播
        loss.backward()
        # 参数更新
        optimizer.step()
        # 打印训练信息
        if batch_idx % 100 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

```
Train Epoch: 0 [0/50000 (0%)]   Loss: 2.305011
Train Epoch: 0 [6400/50000 (13%)]   Loss: 2.162423
Train Epoch: 0 [12800/50000 (26%)]   Loss: 1.740833
Train Epoch: 0 [19200/50000 (38%)]   Loss: 1.122805
```

实现

```python
# 测试模式
model.eval()
val_loss = 0
correct = 0
with torch.no_grad():
    for data, target in val_loader:
        output = model(data)
        val_loss += criterion(output, target).item() # sum up batch loss
        pred = output.max(1, keepdim=True)[1] # get the index of the max log-probability
        correct += pred.eq(target.view_as(pred)).sum().item()

val_loss /= len(val_loader.dataset)

print('Validation set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)'.format(
    val_loss, correct, len(val_loader.dataset),
    100. * correct / len(val_loader.dataset)))
```

实现

- 作业1：用numpy实现训练MLP网络识别手写数字MNIST数据集
  - 运行、阅读并理解反向传播算法示例bp_np.py
  - 修改np_mnist_template.py，更改loss函数、网络结构、激活函数，完成训练MLP网络识别手写数字MNIST数据集。
  - **要求：10个epoch后测试集准确率达到94%以上**

- 作业2：使用Pytorch训练MNIST数据集的MLP模型
  - 运行、阅读并理解mnist_mlp.py,修改网络结构和参数，增加隐藏层，观察训练效果
  - 使用Adam等不同优化器，添加Dropout层，观察训练效果
  - **要求：10个epoch后测试集准确率达到97%以上**

谢谢