神经网络和深度学习基础

CS2916 大语言模型

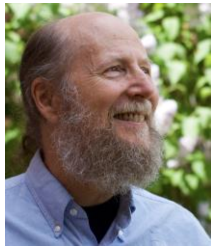
飲水思源 愛國榮校

https://plms.ai/teaching/index.html

目标

- 神经网络的基本概念
  - 理解基本的神经网络概念
  - 学会简单的网络梯度推导
  - 读懂代码: nn_bp.py
- 循环神经网络(RNN)
  - 理解普通RNN、LSTM的优缺点
  - 理解梯度弥散以及缓解方法
  - 了解RNN发展史和常见应用
- 卷积神经网络(CNN)
  - 掌握卷积常见概念（一/二维卷积、卷积核、步长等)
  - 理解CNN在NLP上的应用方法

"The Bitter Lesson"



Rich Sutton
强化学习之父

The biggest lesson that can be read from 70 years of AI research is that general methods that **leverage computation** are ultimately the most effective, and by a large margin

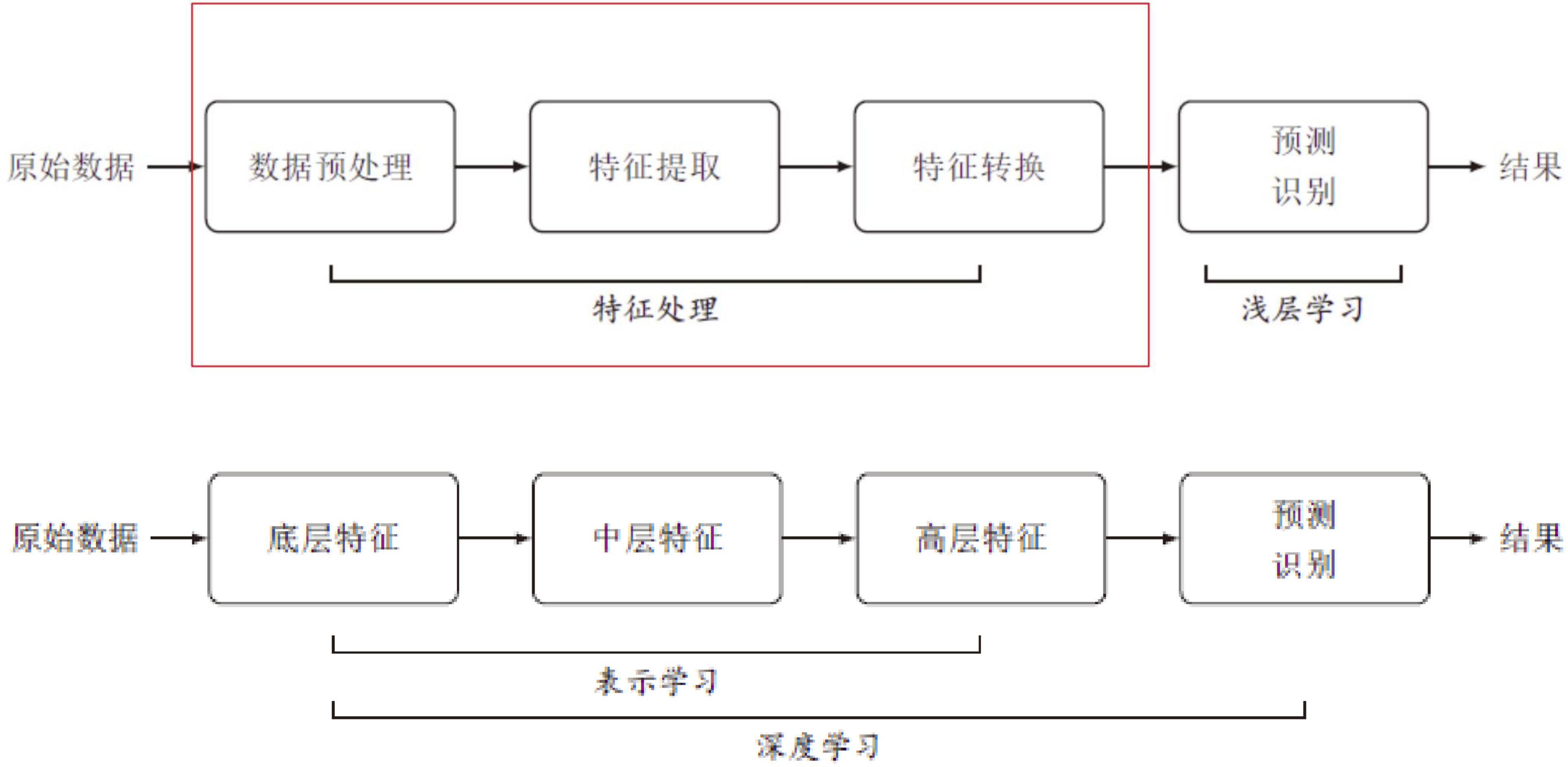
We want AI agents that can **discover like we can**, not which contain what we have discovered. Building in our discoveries only makes it harder to see how the discovering process can be done.

- AI 研究人员常常试图在自身智能体中构建知识,
- 从短期看，这通常是有帮助的，能够令研究人员满意,
- 但从长远看，这会令研究人员停滞不前，甚至抑制进一步发展,
- 突破性进展最终可能会通过一种相反的方法——基于以大规模计算为基础的搜索和学习

原文：http://www.incompleteideas.net/IncIdeas/BitterLesson.html

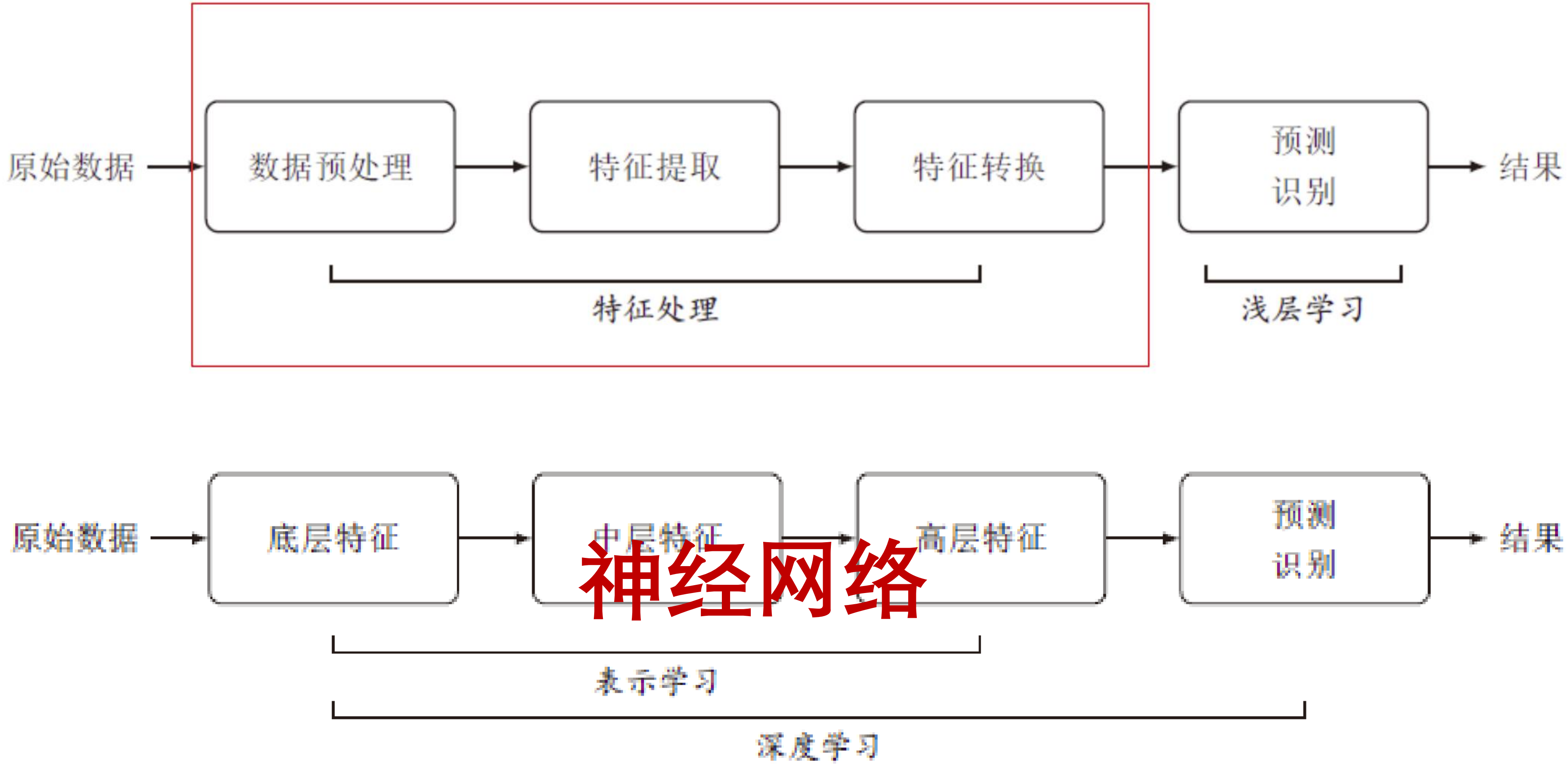
回顾：特征工程问题

- **模型**
  - **在实际应用中，特征往往比分类器更重要**



回顾：特征工程问题

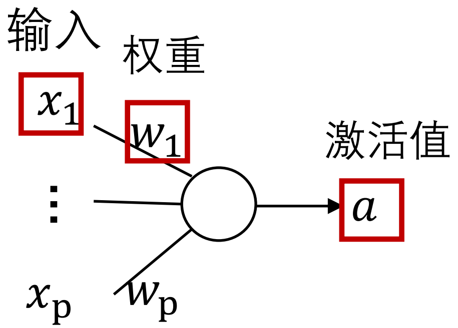
- **模型**
  - **在实际应用中，特征往往比分类器更重要**



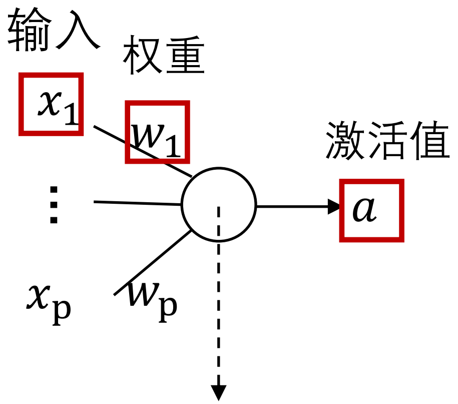
课程内容

- 神经网络的基本概念
- 循环神经网络
- 卷积神经网络

神经元



神经元

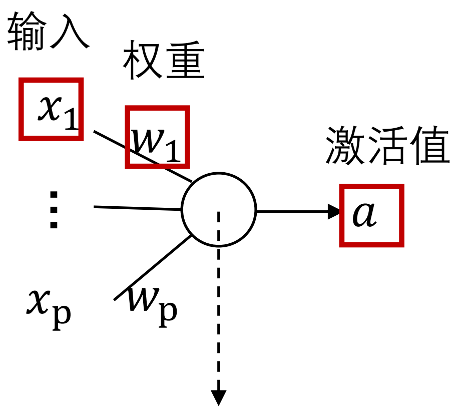


$$z = \Sigma\omega_i x_i + b$$

$$a = f(z)$$

一个神经元是一个多元、复合函数

神经元



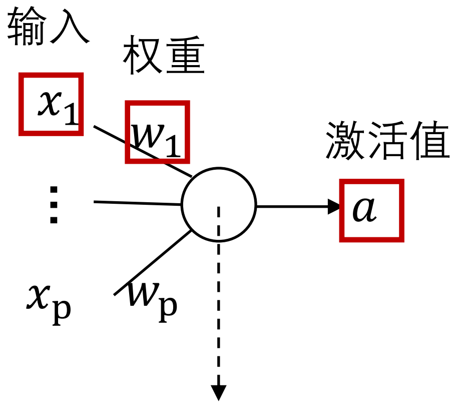
$$z = \Sigma \omega_i x_i + b$$

$$a = \boxed{f}(z)$$

激活函数

根据万能逼近定理，一个包含至少一层隐含层和非线性激活函数的神经网络可以逼近任何连续函数

神经元



$$z = \Sigma \omega_i x_i + b$$

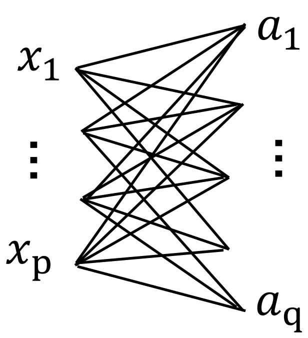
$$a = \boxed{f}(z)$$

激活函数

激活函数	函数	导数
Logistic 函数	$f(x) = \frac{1}{1+\exp(-x)}$	$f'(x) = f(x)(1-f(x))$
Tanh 函数	$f(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$	$f'(x) = 1 - f(x)^2$
ReLU 函数	$f(x) = \max(0, x)$	$f'(x) = I(x > 0)$
ELU 函数	$f(x) = \max(0, x) + \min\big(0, \gamma(\exp(x) - 1)\big)$	$f'(x) = I(x > 0) + I(x \le 0) \cdot \gamma \exp(x)$
SoftPlus 函数	$f(x) = \log\big(1 + \exp(x)\big)$	$f'(x) = \frac{1}{1+\exp(-x)}$

•**ReLU及其变体**：由于其**计算效率高**且在许多情况下性能良好，ReLU通常是隐藏层的首选。它有**助于缓解梯度消失问题**，特别适用于深层网络。
•**Tanh**：当数据中心化（均值为0）时，Tanh函数是一个好的选择，因为它的输出范围是-1到1，这有助于数据的平滑传递。
•**Sigmoid**：尽管在隐藏层较少使用，但在需要输出概率或进行精细控制时（例如在某些类型的循环神经网络中），仍然可以考虑Sigmoid函数。

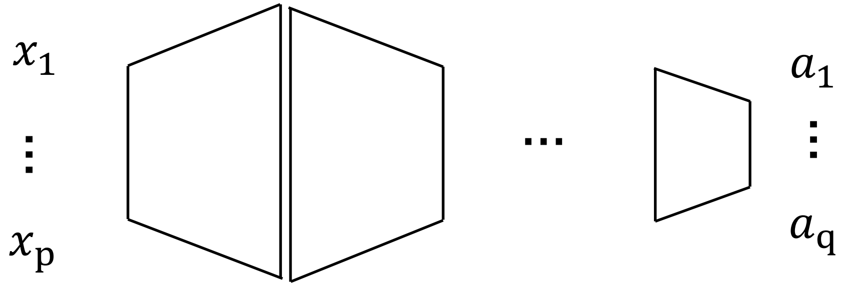
神经层



$$\boldsymbol{a} = f(\boldsymbol{W}\boldsymbol{x} + b)$$

- 一个神经层包含**多个**神经元
- 每个神经元**输入都是相同**的，但激活方式不一样
- 一个神经层是一个多元向量函数

神经网

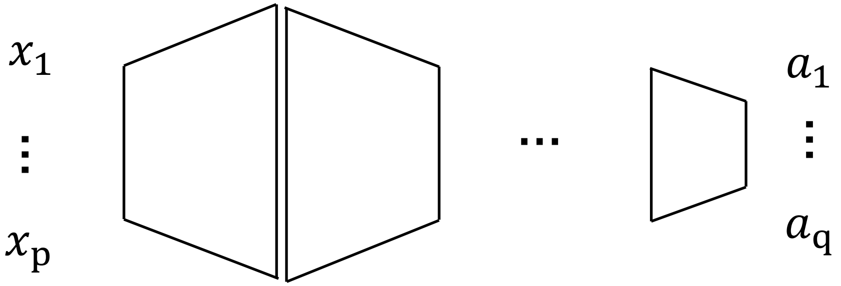


浅层神经网络

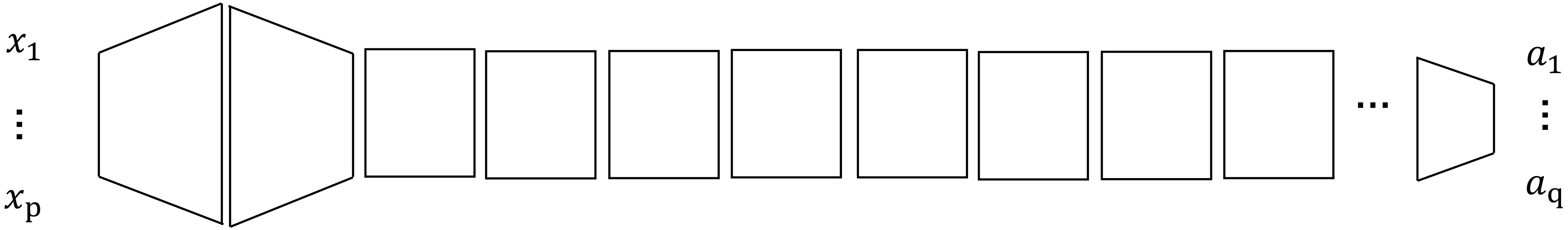
- 将神层堆叠起来
- 输入输出维度要的对齐

神经网

- 将神层堆叠起来
- 输入输出维度要的对齐
- 神经网络可以一直加深



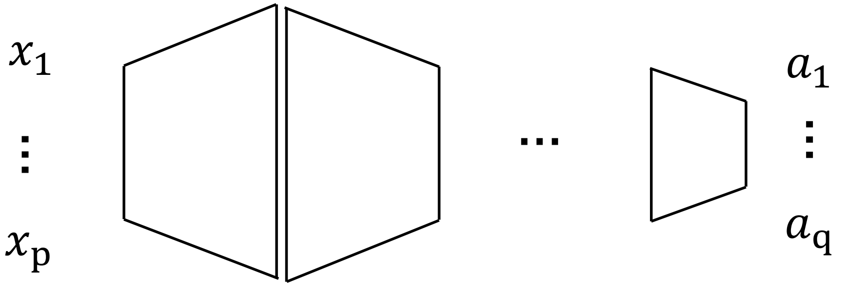
浅层神经网络



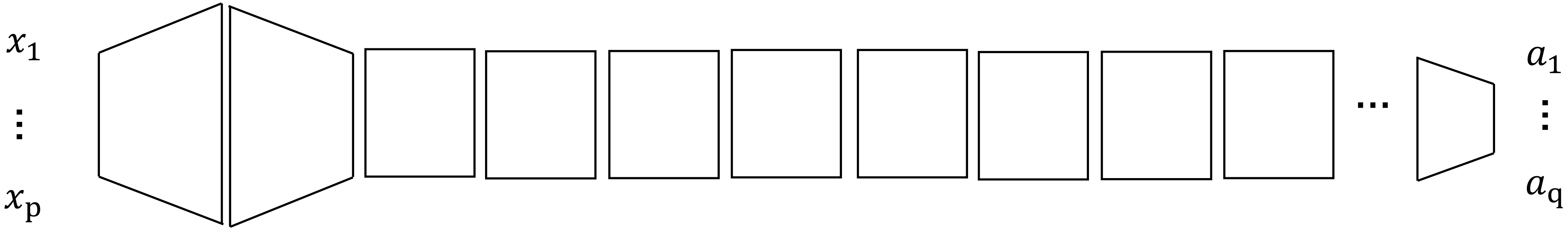
深层神经网络

神经网

- 将神层堆叠起来
- 输入输出维度要的对齐
- 神经网络可以一直加深



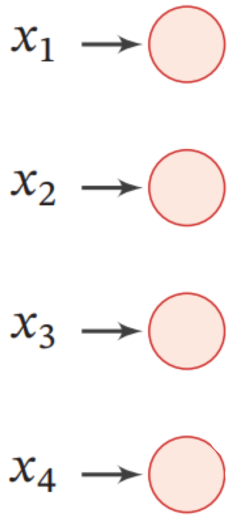
浅层神经网络



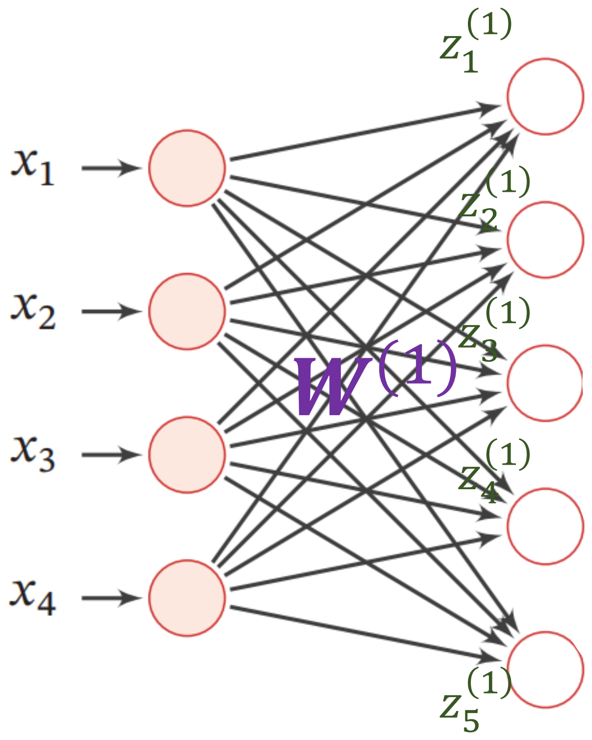
深层神经网络

大语言模型是一个深度神经网络模型。比如LLaMa2 70B网络层数为80

简单的前馈网络



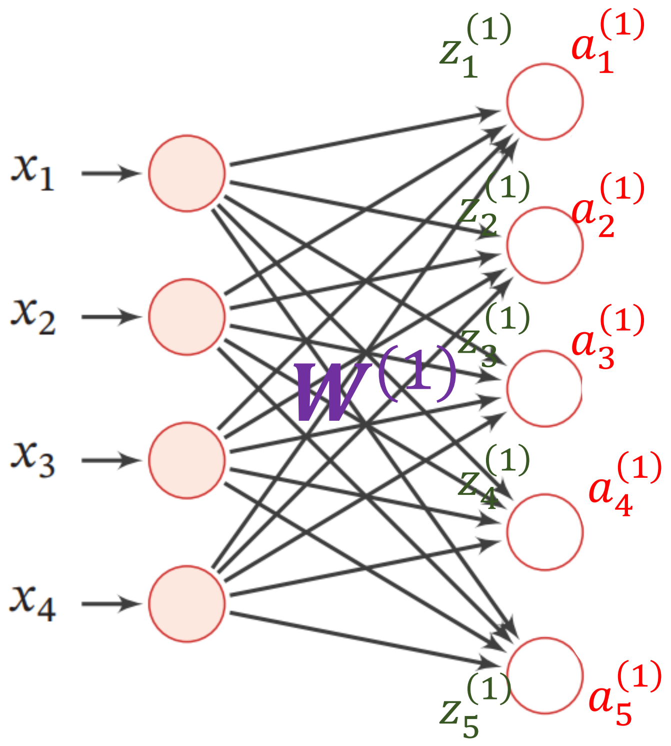
简单的前馈网络



$$\mathrm{a}^0(i.e.,x) \rightarrow z^1$$

线性变换 $$\mathrm{z}^{l+1} = W^l a^l + b^l$$

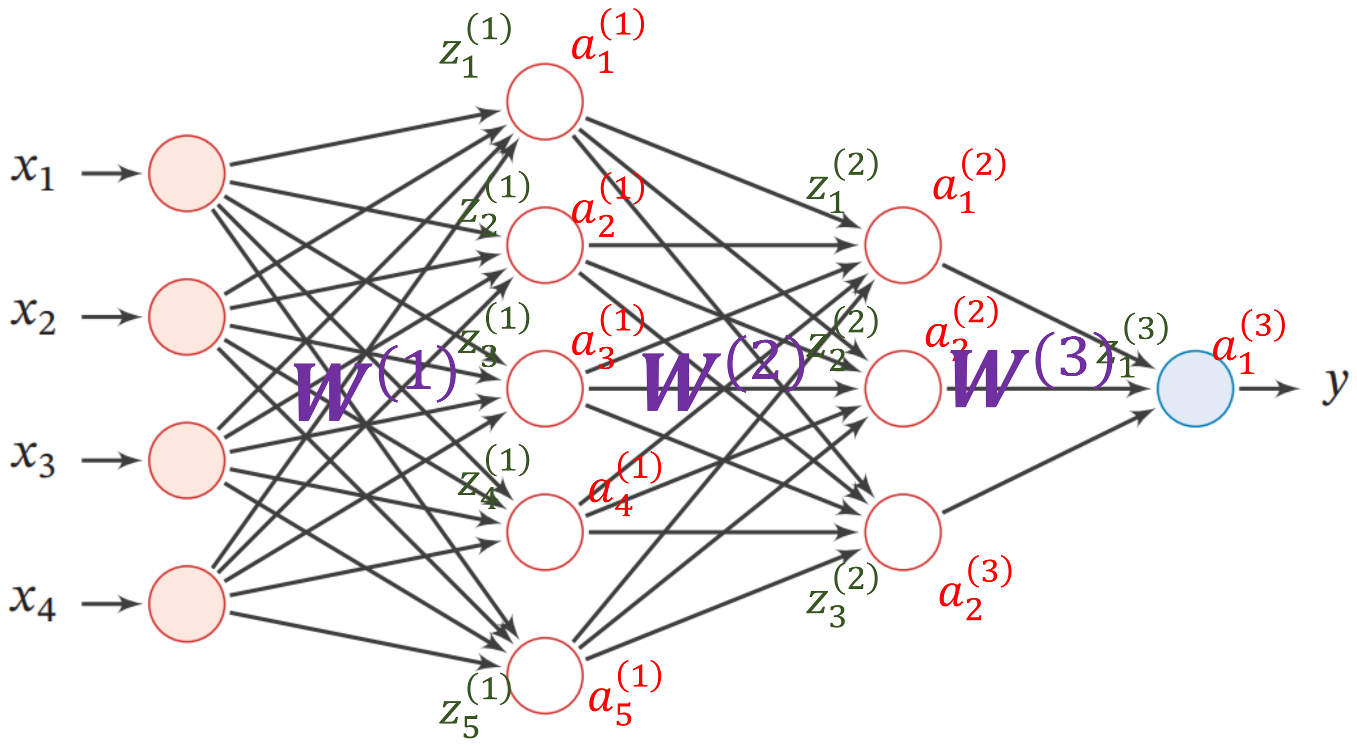
简单的前馈网络



$$\mathrm{a}^0(i.e., x) \rightarrow z^1 \rightarrow \mathrm{a}^1$$

非线性变换 $\mathrm{a}^l = f(z^l)$

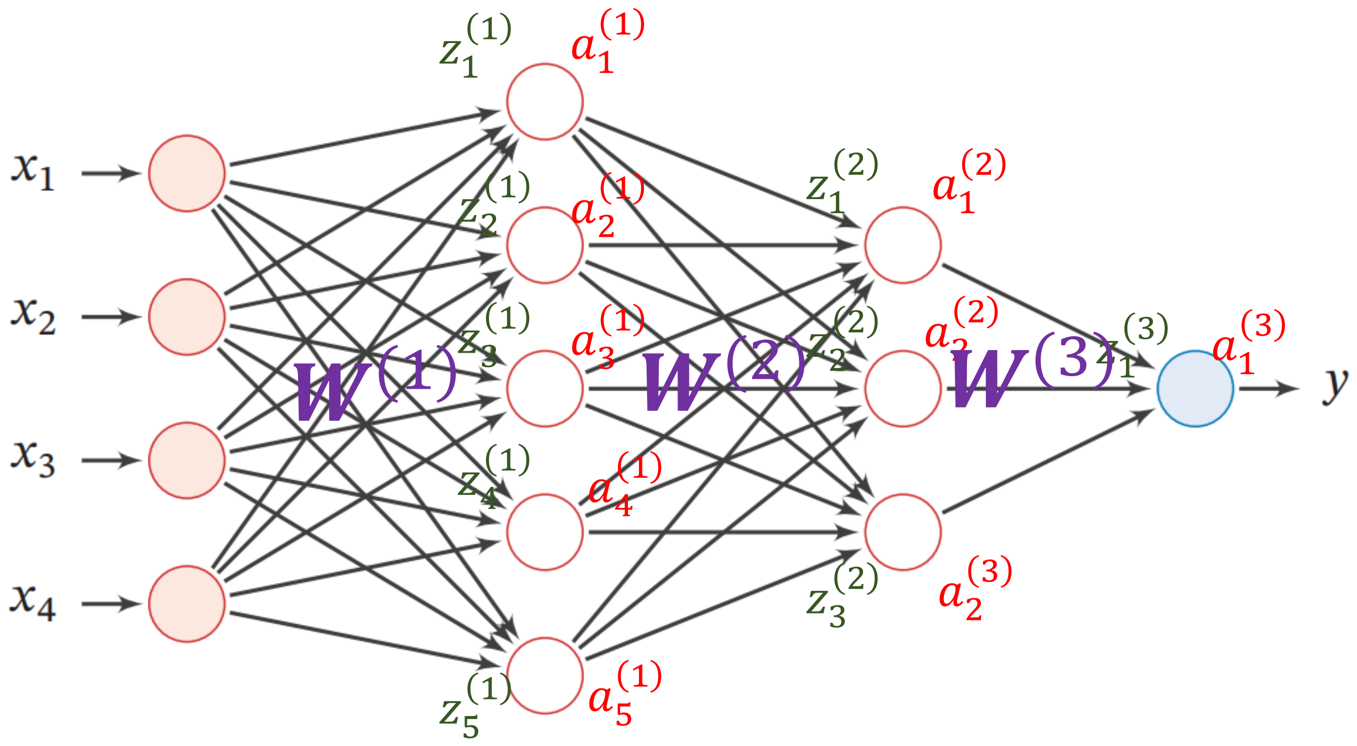
简单的前馈网络



$$a^0(i.e.,x) \to z^1 \to a^1 \to z^2 \to a^2 \ldots \to z^L \to a^L$$

多次正向传播

简单的前馈网络



超参数

L: 层数

n^l:神经元数目

f^l激活函数

学习参数

W^l:连接权重

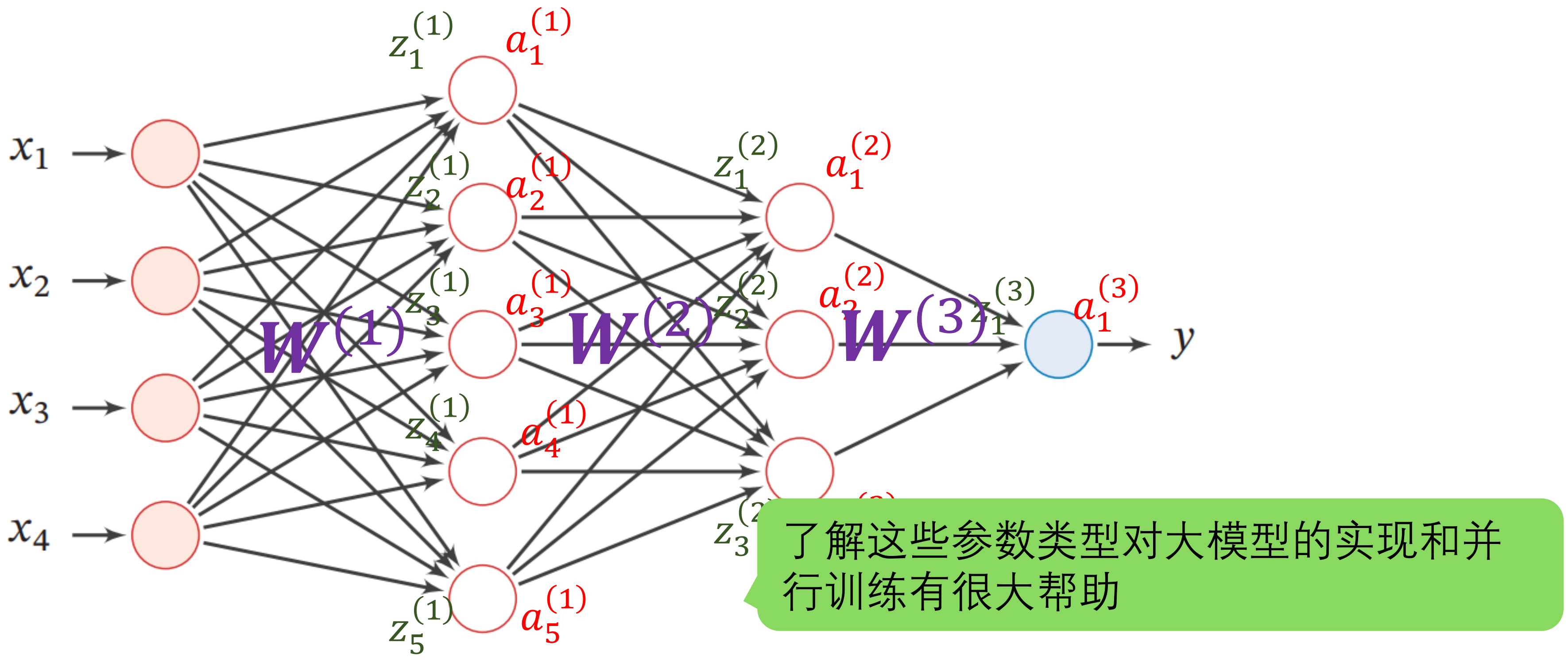
b^l:l层偏置

状态量

z^l:l层神经元状态

a^l:l层神经元激活

简单的前馈网络



超参数

L: 层数

n^l:神经元数目

f^l激活函数

学习参数

W^l:连接权重

b^l:l层偏置

状态量

z^l:l层神经元状态

a^l:l层神经元激活

应用于机器学习

- 数据：(x_i, y_i)
- 模型：$\{y = h(x|w,b), w, b\}$
- 优化准则：
  - 损失函数：$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\boldsymbol{y}^{\top} \log \hat{\boldsymbol{y}},$
  - 经验损失函数
  - $L\left(y_i, h(x_i|w,b)\right) + \lambda \|w\|^2 = L\left(y_i, \mathrm{a}^L(\mathrm{x}_i)\right)$

应用于机器学习

- 数据：(x_i, y_i)
- 模型：$\{y = h(x|w,b), w, b\}$
- 优化准则：
  - 损失函数：$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\boldsymbol{y}^{\top} \log \hat{\boldsymbol{y}},$
  - 经验损失函数：$\mathcal{R}(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}(\boldsymbol{y}^{(n)}, \hat{\boldsymbol{y}}^{(n)})$

$$\boldsymbol{W}^{(l)} \leftarrow \boldsymbol{W}^{(l)} - \alpha \frac{\partial \mathcal{R}(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{W}^{(l)}}$$

$$= \boldsymbol{W}^{(l)} - \alpha \left(\frac{1}{N} \sum_{n=1}^{N} \left(\frac{\partial \mathcal{L}(\boldsymbol{y}^{(n)}, \hat{\boldsymbol{y}}^{(n)})}{\partial \boldsymbol{W}^{(l)}} \right) + \lambda \boldsymbol{W}^{(l)} \right),$$

$$\boldsymbol{b}^{(l)} \leftarrow \boldsymbol{b}^{(l)} - \alpha \frac{\partial \mathcal{R}(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{b}^{(l)}}$$

$$= \boldsymbol{b}^{(l)} - \alpha \left(\frac{1}{N} \sum_{n=1}^{N} \frac{\partial \mathcal{L}(\boldsymbol{y}^{(n)}, \hat{\boldsymbol{y}}^{(n)})}{\partial \boldsymbol{b}^{(l)}} \right),$$

梯度下降法更新参数

应用于机器学习

- 数据：(x_i, y_i)
- 模型：$\{y = h(x|w, b), w, b\}$
- 优化准则：
  - 损失函数：$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\boldsymbol{y}^{\top} \log \hat{\boldsymbol{y}},$
  - 经验损失函数：$\mathcal{R}(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}(\boldsymbol{y}^{(n)}, \hat{\boldsymbol{y}}^{(n)})$

$$\boldsymbol{W}^{(l)} \leftarrow \boldsymbol{W}^{(l)} - \alpha \frac{\partial \mathcal{R}(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{W}^{(l)}}$$

$$= \boldsymbol{W}^{(l)} - \alpha \left(\frac{1}{N} \sum_{n=1}^{N} \left(\frac{\partial \mathcal{L}(\boldsymbol{y}^{(n)}, \hat{\boldsymbol{y}}^{(n)})}{\partial \boldsymbol{W}^{(l)}} \right) + \lambda \boldsymbol{W}^{(l)} \right),$$

$$\boldsymbol{b}^{(l)} \leftarrow \boldsymbol{b}^{(l)} - \alpha \frac{\partial \mathcal{R}(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{b}^{(l)}}$$

$$= \boldsymbol{b}^{(l)} - \alpha \left(\frac{1}{N} \sum_{n=1}^{N} \frac{\partial \mathcal{L}(\boldsymbol{y}^{(n)}, \hat{\boldsymbol{y}}^{(n)})}{\partial \boldsymbol{b}^{(l)}} \right),$$

偏导数求解

应用于机器学习

参数更新

$\boldsymbol{W}^{(l)}$

$\boldsymbol{b}^{(l)}$

$\longrightarrow$

$$\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial w_{ij}^{(l)}} = \boxed{\frac{\partial \boldsymbol{z}^{(l)}}{\partial w_{ij}^{(l)}}}\boxed{\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l)}}},$$

$$\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{b}^{(l)}} = \boxed{\frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{b}^{(l)}}}\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l)}}.$$

$\longrightarrow$

$$\frac{\partial \boldsymbol{z}^{(l)}}{\partial w_{ij}^{(l)}} \quad \frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{b}^{(l)}} \quad \frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l)}}$$

链式法则 （Chain Rule）

若 $z = f(y) \quad y = g(x)$ 则:

$$x \to y \to z \qquad \frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

若 $z = f(x, y), \quad x = g(t), \quad y = h(t)$ 则:

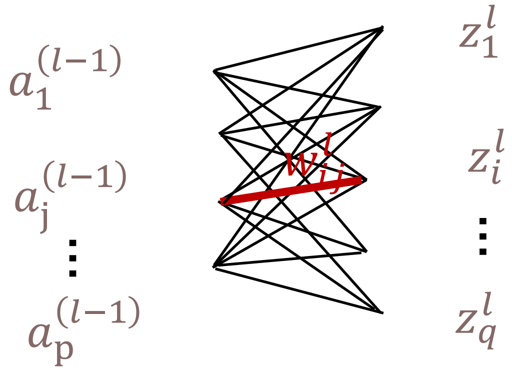
$$\begin{tikzcd}[row sep=small] & x \arrow[dr] & \\ t \arrow[ur] \arrow[dr] & & z \\ & y \arrow[ur] & \end{tikzcd} \qquad \frac{dz}{dt} = \frac{dz}{dy}\frac{dy}{dt} + \frac{dz}{dx}\frac{dx}{dt}$$

参与计算的路径都会获得梯度

计算 $\frac{\partial \boldsymbol{z}^{(l)}}{\partial w_{ij}^{(l)}}$

（1）先正向思考因变量是怎么由自变量计算出来的
（2）参与计算的才有资格拿到梯度

$$
\begin{aligned}
\frac{\partial \boldsymbol{z}^{(l)}}{\partial w_{ij}^{(l)}} &= \left[\frac{\partial z_1^{(l)}}{\partial w_{ij}^{(l)}}, \cdots, \boxed{\frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}}}, \cdots, \frac{\partial z_{M_l}^{(l)}}{\partial w_{ij}^{(l)}}\right] \\
&= \left[0, \cdots, \boxed{\frac{\partial(\boldsymbol{w}_{i:}^{(l)}\boldsymbol{a}^{(l-1)} + b_i^{(l)})}{\partial w_{ij}^{(l)}}}, \cdots, 0\right] \\
&= \left[0, \cdots, a_j^{(l-1)}, \cdots, 0\right] \\
&\triangleq \mathbb{I}_i(a_j^{(l-1)}) \quad \in \mathbb{R}^{1\times M_l},
\end{aligned}
$$



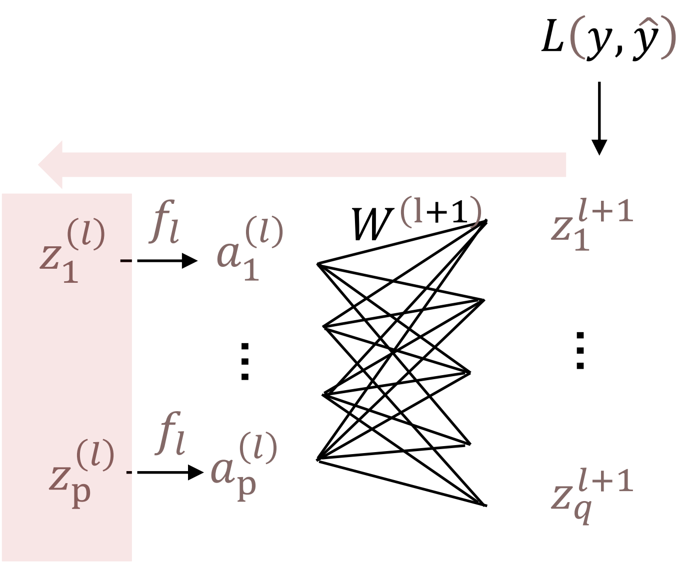
计算 $\frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{b}^{(l)}}$

$$\frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{b}^{(l)}} = \boldsymbol{I}_{M_l} \qquad \in \mathbb{R}^{M_l \times M_l}$$

为 $M_l \times M_l$ 的单位矩阵

计算 $\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l)}}$

$$
\begin{aligned}
\delta^{(l)} &\triangleq \frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l)}} \\
&= \boxed{\frac{\partial \boldsymbol{a}^{(l)}}{\partial \boldsymbol{z}^{(l)}}} \cdot \boxed{\frac{\partial \boldsymbol{z}^{(l+1)}}{\partial \boldsymbol{a}^{(l)}}} \cdot \boxed{\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l+1)}}} \\
&= \boxed{\operatorname{diag}(f_l'(\boldsymbol{z}^{(l)}))} \cdot \boxed{(\boldsymbol{W}^{(l+1)})^{\mathsf{T}}} \cdot \boxed{\delta^{(l+1)}} \\
&= f_l'(\boldsymbol{z}^{(l)}) \odot \left((\boldsymbol{W}^{(l+1)})^{\mathsf{T}} \delta^{(l+1)} \right) \quad \in \mathbb{R}^{M_l}
\end{aligned}
$$



参数更新

参数更新

$\boldsymbol{W}^{(l)}$

$\boldsymbol{b}^{(l)}$

$$\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial w_{ij}^{(l)}} = \frac{\partial \boldsymbol{z}^{(l)}}{\partial w_{ij}^{(l)}} \frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l)}},$$

$$\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{b}^{(l)}} = \frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{b}^{(l)}} \frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l)}}.$$

常被记作$\delta^{(l)}$

$$\frac{\partial \boldsymbol{z}^{(l)}}{\partial w_{ij}^{(l)}} \quad \frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{b}^{(l)}} \quad \frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l)}}$$

$$\boxed{\mathbb{I}_i(a_j^{(l-1)}) \qquad \boldsymbol{I}_{M_l} \qquad f_l'(\boldsymbol{z}^{(l)}) \odot \left((\boldsymbol{W}^{(l+1)})^\top \delta^{(l+1)} \right)}$$

反向传播算法

算法 4.1　使用反向传播算法的随机梯度下降训练过程

输入: 训练集 $\mathcal{D}=\{(\boldsymbol{x}^{(n)},y^{(n)})\}_{n=1}^{N}$, 验证集 $\mathcal{V}$, 学习率 α, 正则化系数 λ, 网络层数 L, 神经元数量 M_l, $1\le l\le L$.

```
随机初始化 W, b ;
repeat
    对训练集 D 中的样本随机重排序;
    for n = 1 ··· N do
        从训练集 D 中选取样本 (x^(n), y^(n));
        前馈计算每一层的净输入 z^(l) 和激活值 a^(l), 直到最后一层;
        反向传播计算每一层的误差 δ^(l);                      // 公式 (4.63)
        // 计算每一层参数的导数
        ∀l,  ∂L(y^(n), ŷ^(n)) / ∂W^(l) = δ^(l) (a^(l-1))^T;   // 公式 (4.68)
        ∀l,  ∂L(y^(n), ŷ^(n)) / ∂b^(l) = δ^(l);               // 公式 (4.69)
        // 更新参数
        W^(l) ← W^(l) − α(δ^(l) (a^(l-1))^T + λW^(l));
        b^(l) ← b^(l) − αδ^(l);
    end
until 神经网络模型在验证集 V 上的错误率不再下降;
```

其中:

$$\forall l,\quad \frac{\partial \mathcal{L}(\boldsymbol{y}^{(n)},\hat{\boldsymbol{y}}^{(n)})}{\partial \boldsymbol{W}^{(l)}}=\delta^{(l)}(\boldsymbol{a}^{(l-1)})^{\intercal}$$

$$\forall l,\quad \frac{\partial \mathcal{L}(\boldsymbol{y}^{(n)},\hat{\boldsymbol{y}}^{(n)})}{\partial \boldsymbol{b}^{(l)}}=\delta^{(l)}$$

$$\boldsymbol{W}^{(l)}\leftarrow \boldsymbol{W}^{(l)}-\alpha(\delta^{(l)}(\boldsymbol{a}^{(l-1)})^{\intercal}+\lambda \boldsymbol{W}^{(l)})$$

$$\boldsymbol{b}^{(l)}\leftarrow \boldsymbol{b}^{(l)}-\alpha\delta^{(l)}$$

输出: $\boldsymbol{W},\boldsymbol{b}$

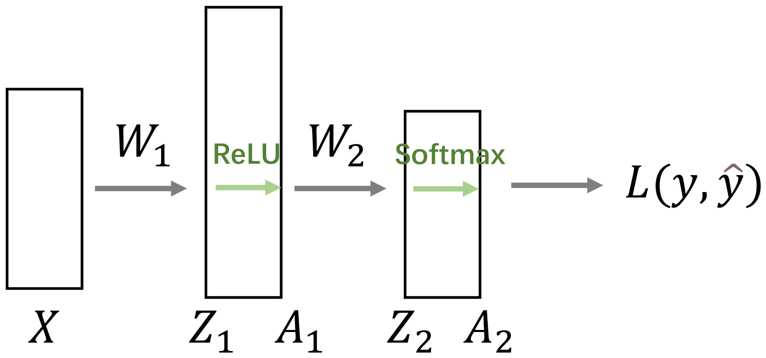
表来源: https://nndl.github.io/nndl-book.pdf

例子学习

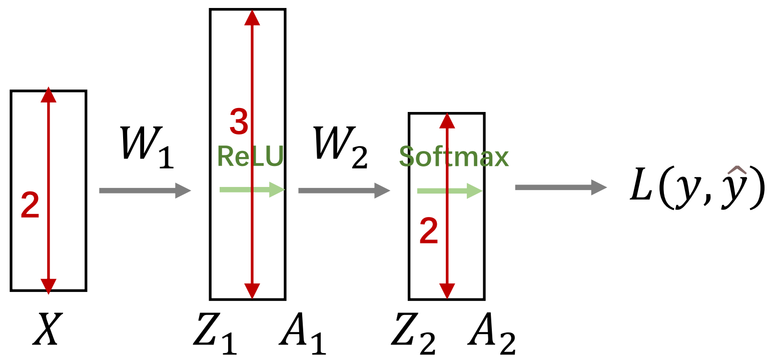
- 假设我们要构建一个简单的**前馈神经网络**，用于解决**二分类**问题
  - 输入层：2个特征
  - 隐藏层：1个，使用3个神经元，激活函数为ReLU
  - 输出层：2个神经元（对应于两个类别），激活函数为softmax
  - 交叉熵做损失函数
  - 两个训练样本
    - [0.1, 0.2] -> [1,0]
    - [0.3, 0.4] -> [0,1]

例子学习

- 假设我们要构建一个简单的前馈神经网络，用于解决二分类问题
  - 输入层：2个特征
  - 隐藏层：1个，使用3个神经元，激活函数为ReLU
  - 输出层：2个神经元（对应于两个类别），激活函数为softmax
  - 交叉熵做损失函数
  - 两个训练样本
    - [0.1, 0.2] -> [1,0]
    - [0.3, 0.4] -> [0,1]

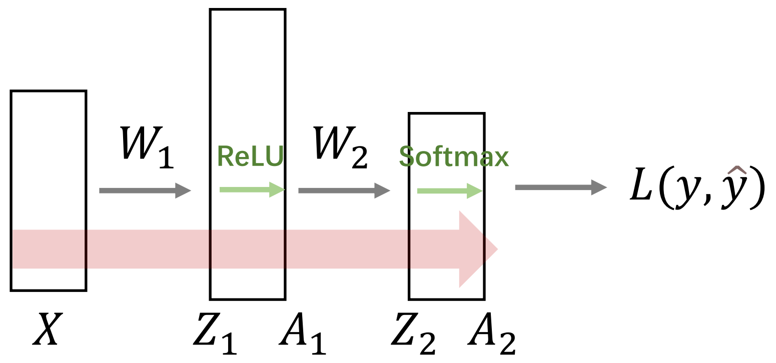


例子学习




```python
import numpy as np

# 初始化参数
np.random.seed(42) # 确保每次运行结果一致
input_size = 2 # 输入层节点数
hidden_size = 3 # 隐藏层节点数
output_size = 2 # 输出层节点数
learning_rate = 0.1 # 学习率

# 随机初始化权重和偏置
W1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros(hidden_size)
W2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros(output_size)

# ReLU激活函数及其导数
def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return (x > 0).astype(float)

# Softmax函数
def softmax(x):
    exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=1, keepdims=True)

# 交叉熵损失及其导数
def cross_entropy_loss(y_pred, y_true):
    m = y_true.shape[0]
    return -np.sum(y_true * np.log(y_pred)) / m

def delta_cross_entropy_softmax(y_pred, y_true):
    return y_pred - y_true
```

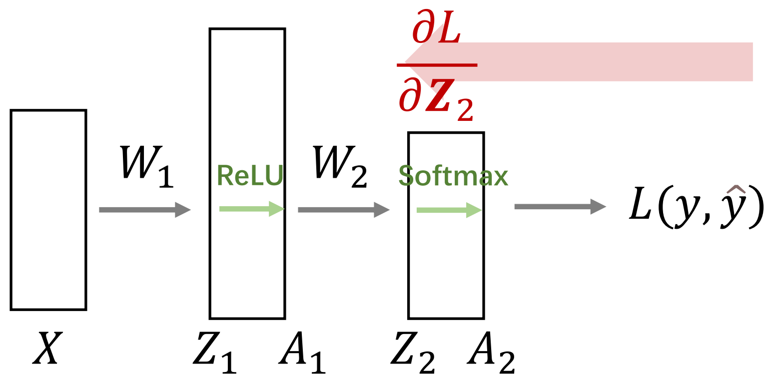
例子学习




```python
# 前向传播
def forward_propagation(X):
    Z1 = X.dot(W1) + b1
    A1 = relu(Z1)
    Z2 = A1.dot(W2) + b2
    A2 = softmax(Z2)
    return Z1, A1, Z2, A2
```

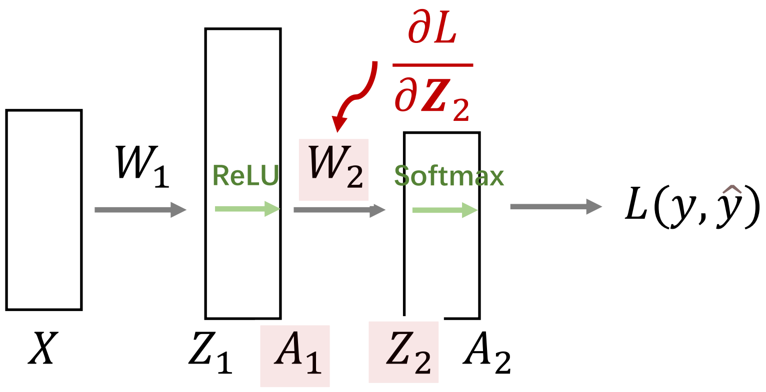
真实实现中要时刻留心矩阵的维度

例子学习



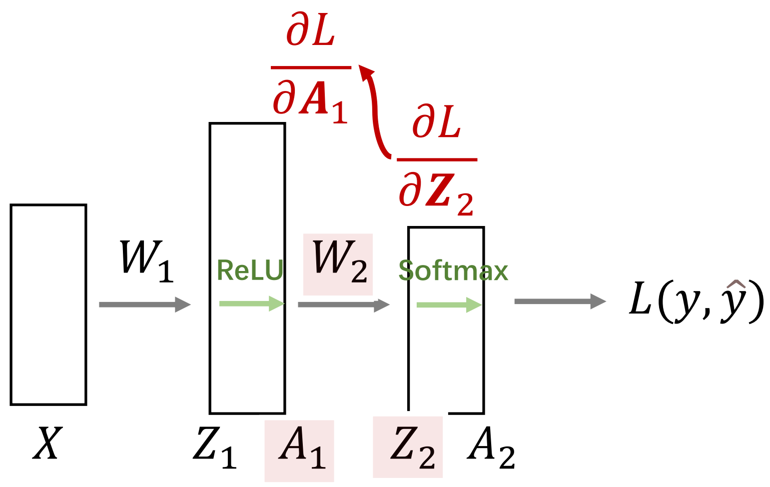

```python
# 反向传播
def backward_propagation(X, Y, Z1, A1, Z2, A2):
    m = X.shape[0]

    dZ2 = delta_cross_entropy_softmax(A2, Y)
    dW2 = (1/m) * np.dot(A1.T, dZ2)
    db2 = (1/m) * np.sum(dZ2, axis=0)

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * relu_derivative(Z1)
    dW1 = (1/m) * np.dot(X.T, dZ1)
    db1 = (1/m) * np.sum(dZ1, axis=0)

    return dW1, db1, dW2, db2

# 更新参数
def update_parameters(dW1, db1, dW2, db2):
    global W1, b1, W2, b2
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
```

例子学习



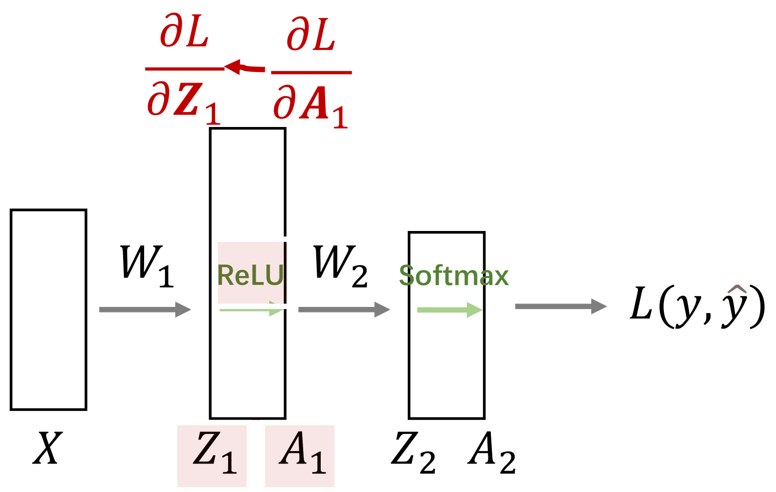

```python
# 反向传播
def backward_propagation(X, Y, Z1, A1, Z2, A2):
    m = X.shape[0]

    dZ2 = delta_cross_entropy_softmax(A2, Y)
    dW2 = (1/m) * np.dot(A1.T, dZ2)
    db2 = (1/m) * np.sum(dZ2, axis=0)

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * relu_derivative(Z1)
    dW1 = (1/m) * np.dot(X.T, dZ1)
    db1 = (1/m) * np.sum(dZ1, axis=0)

    return dW1, db1, dW2, db2

# 更新参数
def update_parameters(dW1, db1, dW2, db2):
    global W1, b1, W2, b2
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
```

例子学习



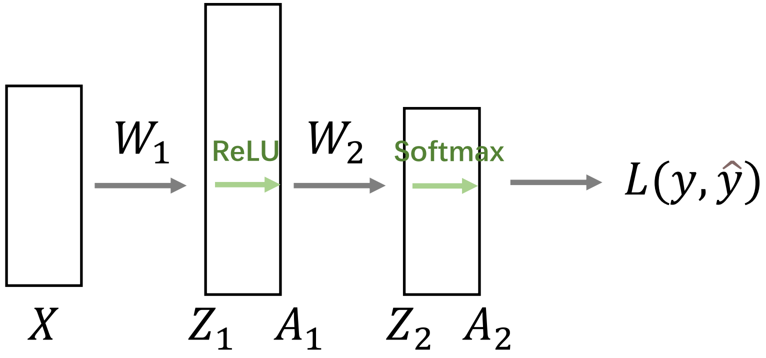

```python
# 反向传播
def backward_propagation(X, Y, Z1, A1, Z2, A2):
    m = X.shape[0]

    dZ2 = delta_cross_entropy_softmax(A2, Y)
    dW2 = (1/m) * np.dot(A1.T, dZ2)
    db2 = (1/m) * np.sum(dZ2, axis=0)

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * relu_derivative(Z1)
    dW1 = (1/m) * np.dot(X.T, dZ1)
    db1 = (1/m) * np.sum(dZ1, axis=0)

    return dW1, db1, dW2, db2

# 更新参数
def update_parameters(dW1, db1, dW2, db2):
    global W1, b1, W2, b2
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
```

例子学习



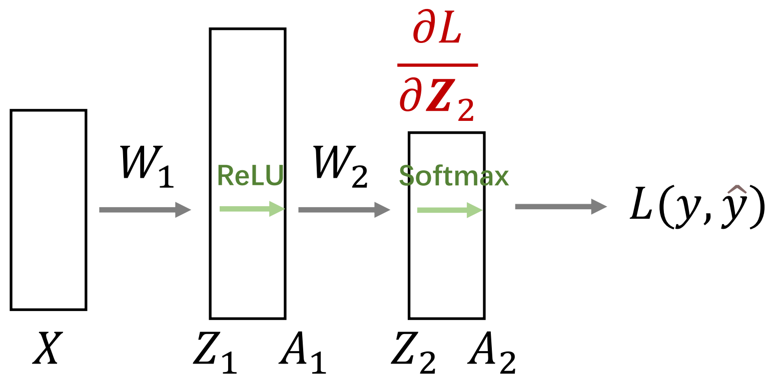

```python
# 反向传播
def backward_propagation(X, Y, Z1, A1, Z2, A2):
    m = X.shape[0]

    dZ2 = delta_cross_entropy_softmax(A2, Y)
    dW2 = (1/m) * np.dot(A1.T, dZ2)
    db2 = (1/m) * np.sum(dZ2, axis=0)

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * relu_derivative(Z1)
    dW1 = (1/m) * np.dot(X.T, dZ1)
    db1 = (1/m) * np.sum(dZ1, axis=0)

    return dW1, db1, dW2, db2

# 更新参数
def update_parameters(dW1, db1, dW2, db2):
    global W1, b1, W2, b2
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
```

例子学习




```python
# 示例输入和标签
X = np.array([[0.1, 0.2], [0.3, 0.4]]) # 2个样本
Y = np.array([[1, 0], [0, 1]]) # 对应的one-hot标签

# 执行一次前向传播
Z1, A1, Z2, A2 = forward_propagation(X)

# 计算损失
loss = cross_entropy_loss(A2, Y)

# 执行一次反向传播
dW1, db1, dW2, db2 = backward_propagation(X, Y, Z1, A1, Z2, A2)

# 更新参数
update_parameters(dW1, db1, dW2, db2)

loss
```

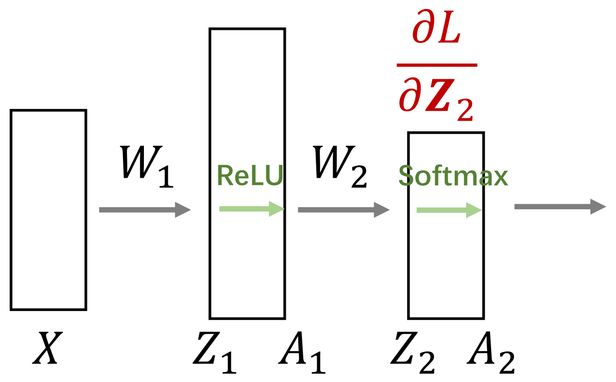
深层网络：计算开销大

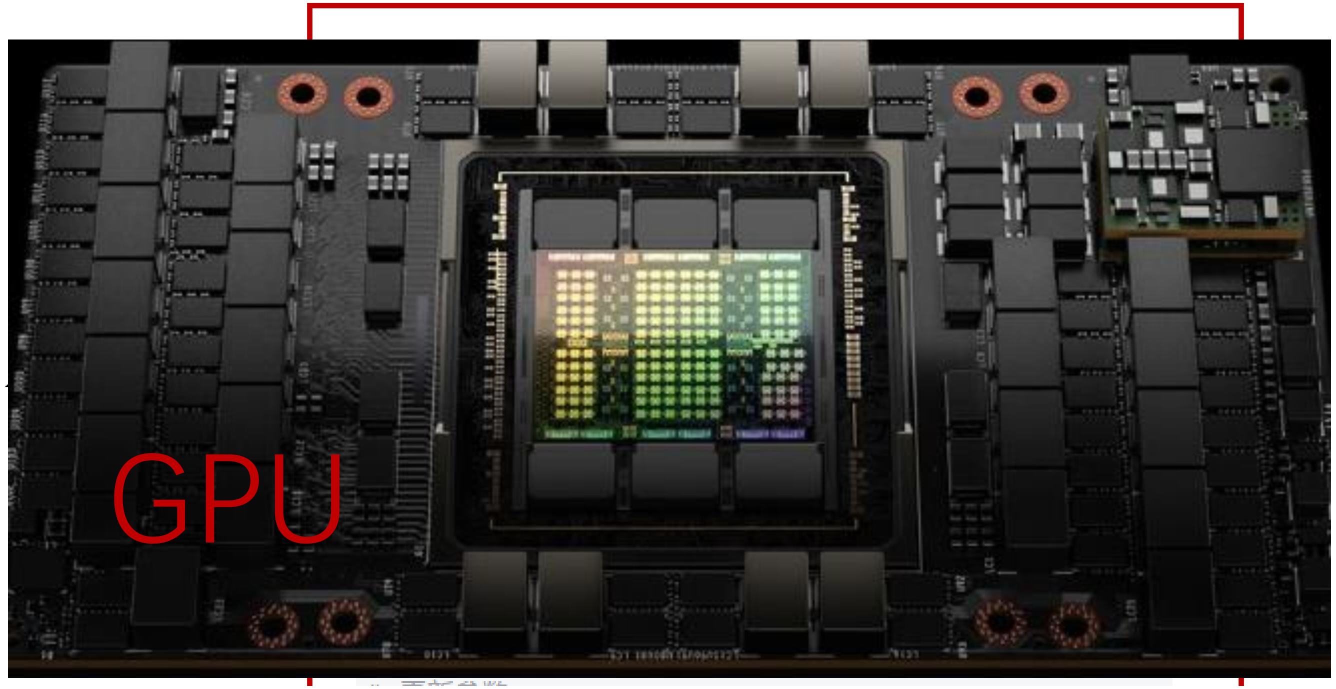


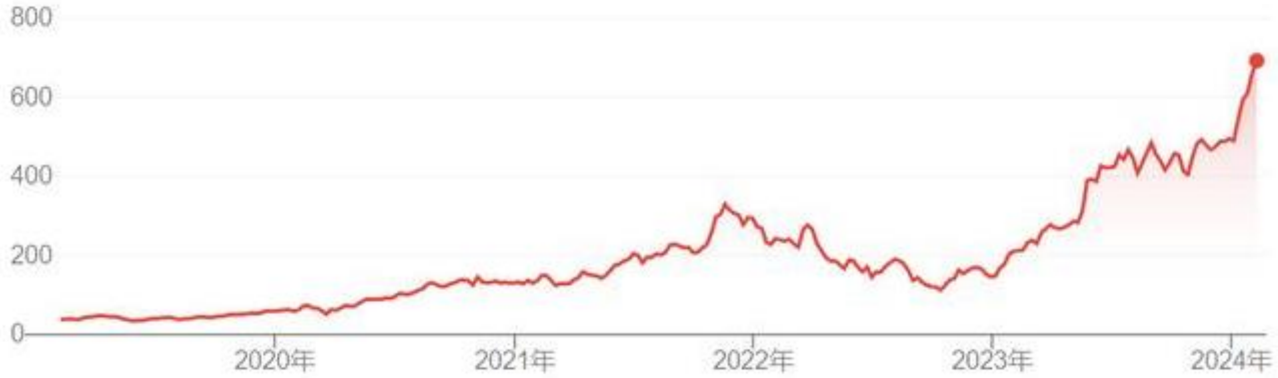

```python
# 前向传播
def forward_propagation(X):
    Z1 = X.dot(W1) + b1
    A1 = relu(Z1)
    Z2 = A1.dot(W2) + b2
    A2 = softmax(Z2)
    return Z1, A1, Z2, A2

# 反向传播
def backward_propagation(X, Y, Z1, A1, Z2, A2):
    m = X.shape[0]

    dZ2 = delta_cross_entropy_softmax(A2, Y)
    dW2 = (1/m) * np.dot(A1.T, dZ2)
    db2 = (1/m) * np.sum(dZ2, axis=0)

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * relu_derivative(Z1)
    dW1 = (1/m) * np.dot(X.T, dZ1)
    db1 = (1/m) * np.sum(dZ1, axis=0)

    return dW1, db1, dW2, db2
```

如果是100层呢？

深层网络：计算开销大

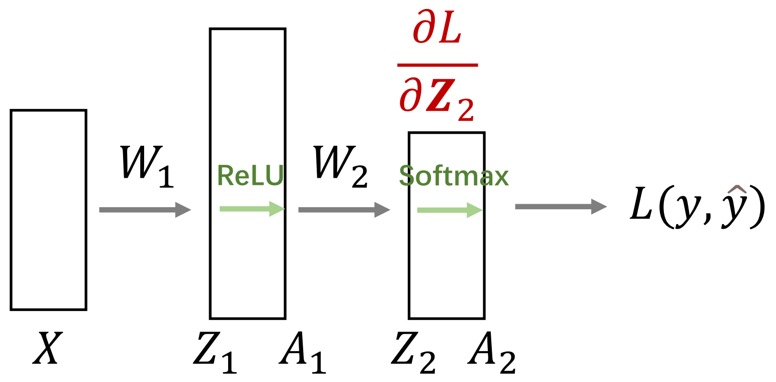






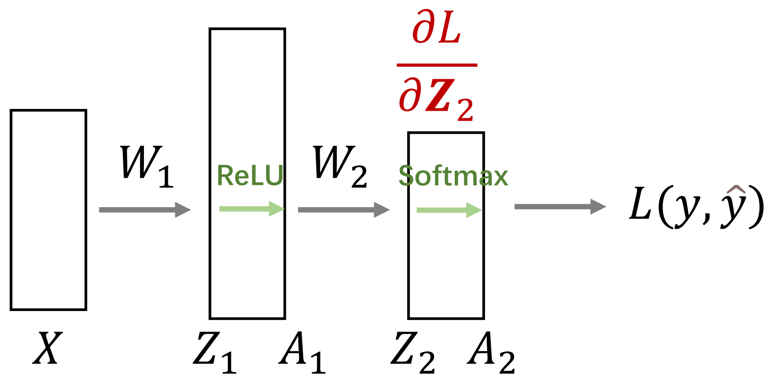
过去5年英伟达股价

深层网络：反向传播求导很繁琐




```python
# 反向传播
def backward_propagation(X, Y, Z1, A1, Z2, A2):
    m = X.shape[0]

    dZ2 = delta_cross_entropy_softmax(A2, Y)
    dW2 = (1/m) * np.dot(A1.T, dZ2)
    db2 = (1/m) * np.sum(dZ2, axis=0)

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * relu_derivative(Z1)
    dW1 = (1/m) * np.dot(X.T, dZ1)
    db1 = (1/m) * np.sum(dZ1, axis=0)

    return dW1, db1, dW2, db2

# 更新参数
def update_parameters(dW1, db1, dW2, db2):
    global W1, b1, W2, b2
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
```

深层网络：反向传播求导很繁琐




```
# 反向传播
def backward_propagation(X, Y, Z1, A1, Z2, A2):
    m = X.shape[0]

    dZ2 = delta_cross_entropy_softmax(A2, Y)
    dW2 = (1/m) * np.dot(A1.T, dZ2)
    db2 = (1/m) * np.sum(dZ2, axis=0)

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * relu_derivative(Z1)
    dW1 = (1/m) * np.dot(X.T, dZ1)
    db1 = (1/m) * np.sum(dZ1, axis=0)

    return dW1, db1, dW2, db2

dW2, db2):
```

自动求导

从特征工程 到 结构工程

- 神经网络避免了繁琐的**手工设计特征**过程，但需要**人工设计网络结构**

从特征工程 到 结构工程

- 神经网络避免了繁琐的**手工设计特征**过程，但需要**人工设计网络结构**
- 不同网络结构往往包含了**不同的结构先验** (inductive bias)
- 不同的**任务、模态**可能需要不同结构的网络

从特征工程 到 结构工程

- 神经网络避免了繁琐的**手工设计特征**过程，但需要**人工设计网络结构**
- 不同网络结构往往包含了不同的结构先验 (inductive bias)
- 不同的任务、模态可能需要不同结构的网络
- 常见网络架构
  - 循环神经网络
  - 卷积神经网络
  - 递归神经网络
  - 图神经网络