中国科学院大学
University of Chinese Academy of Sciences
自然语言处理
练习1—从入门到变形
王 石 资康莉 刘 瑜
2026年春季课程
https://ictkc.github.io/teaching/

练习1——从入门到Transformer

冬夜读书示子聿

南宋·陆游

古人学问无遗力，少壮工夫老始成

纸上得来终觉浅，绝知此事要躬行

目录

1 手搓异或网络

2

3

4

单个感知机的致命缺陷：异或问题

$$y(x1, x2) = f(\omega 1 * x1 + \omega 2 * x2 - b)$$

1969年，Minsky

逻辑运算	x_1	x_2	y
异或（XOR）	1	1	0
	1	0	1
	0	1	1
	0	0	0

11：$w_1 + w_2 - b < 0 \rightarrow b > w_1 + w_2$

10：$w_1 + 0 - b \geq 0 \rightarrow b \leq w_1$

01：$0 + w_2 - b \geq 0 \rightarrow b \leq w_2$

00：$0 + 0 - b < 0 \rightarrow b > 0$

矛盾，无解！

多层感知器 (Multilayer Perceptron， MLP)

Input layer

Hidden layer

Output layer

x_1 x_2 h_1 h_2 y_k

w_{11} 1

w_{21} -1

w_{12} -1

w_{22} 1

w_{1k} 1

w_{2k} 1

所有b=0.5

x_1	x_2	h_1	h_2	y
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	0	0	0

试错法

- 对w、b作微小变动，看y对不对？不对就总结规律，继续改进！

$\boldsymbol{b}$
$\boldsymbol{w}$

y

$\boldsymbol{b} + \Delta \boldsymbol{b}$
$\boldsymbol{w} + \Delta \boldsymbol{w}$

$y + \Delta y$

http://neuralnetworksanddeeplearning.com/chap2.html

预测$\hat{y}$与真实y的差距：损失（也称成本）

$\hat{y}=f(x)$ 为预测值

- 单样本

$$L = |y_i - \hat{y}_i|$$

- 多样本
  - 平均绝对误差 (MAE)

$$L = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

  - 均方误差 (MSE)

$$L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- 向量型多样本
  - 交叉熵

$$L = -\sum_{i=1}^{n} y_i \cdot log(\hat{y}_i)$$

反向传播的参数更新

输入　输入层　隐层1　隐层2　输出层　损失函数

x_1, x_2, x_3

$\boldsymbol{x}$, $\boldsymbol{l}_1$, $\boldsymbol{l}_2$, y, J

ϕ_1, ϕ_2, ϕ_3

$\frac{\partial J}{\partial y}$, $\frac{\partial y}{\partial \boldsymbol{l}_2}$, $\frac{\partial \boldsymbol{l}_2}{\partial \boldsymbol{l}_1}$, $\frac{\partial \boldsymbol{l}_1}{\partial \boldsymbol{x}}$

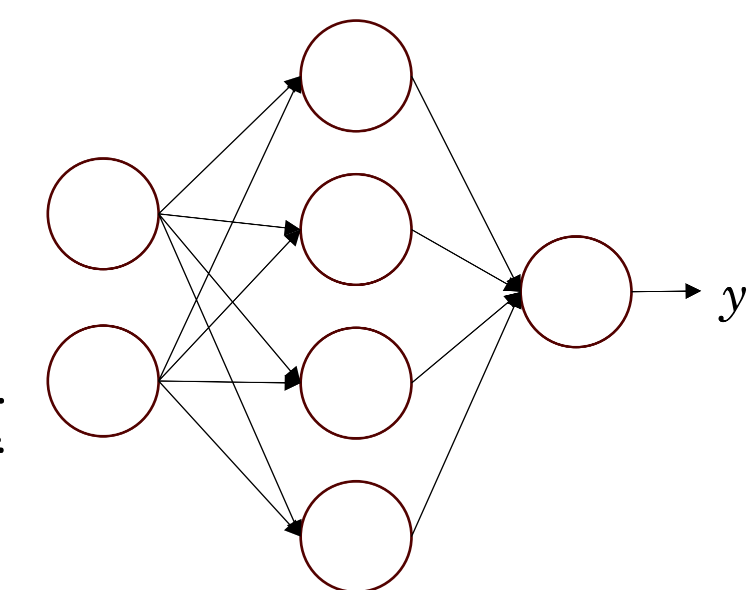
$\frac{\partial J}{\partial \boldsymbol{W}_3}, \frac{\partial J}{\partial b_3}$

$\frac{\partial J}{\partial \boldsymbol{W}_2}, \frac{\partial J}{\partial \boldsymbol{b}_2}$

$\frac{\partial J}{\partial \boldsymbol{W}_1}, \frac{\partial J}{\partial \boldsymbol{b}_1}$

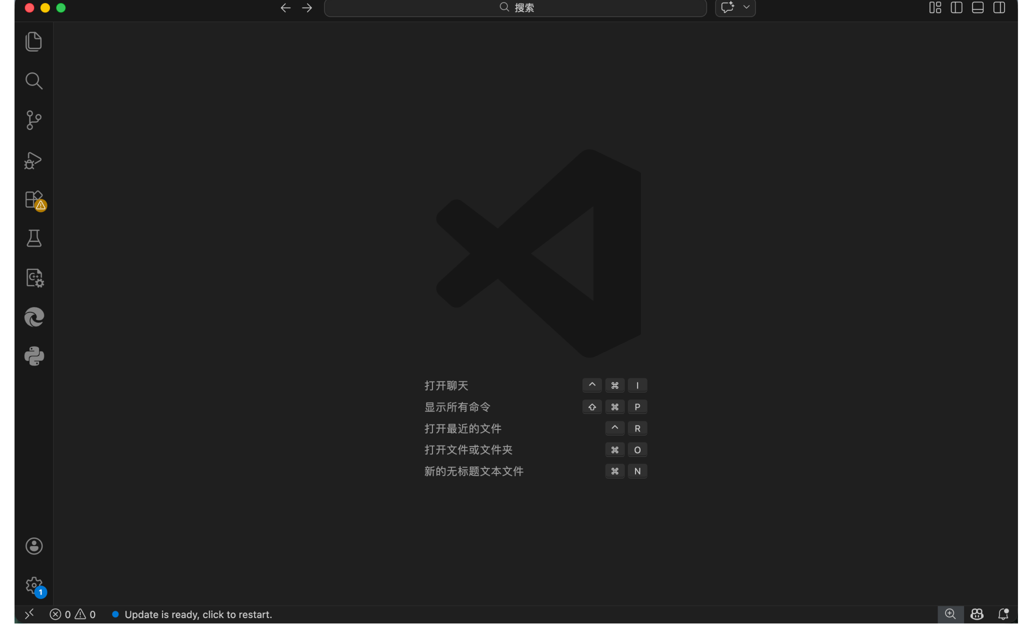
网络设计

- 多层感知机（MLP）网络
- 1输入层（2个神经元）+1隐藏层（2个或3个神经元）+1输出层（1个神经元）
- 激活函数：sigmoid
- 损失函数：均方误差（MSE）或交叉熵
- 反向传播：链式法则计算梯度，并更新权重



开发环境 VSCode

- ❒ Visual Studio Code 简称 VSCode，是微软开发的一款 轻量级 / 跨平台 的代码编辑器；
- ❒ 支持 Windows、macOS 和 Linux 操作系统；
- ❒ 支持如下 编程语言：
  - ❒ JavaScript、TypeScript、Node.js、C++、C#、Java、Python、PHP、Go
- ❒ 内置了 Git 版本控制插件，可以进行 版本控制和代码提交



下载安装

Visual Studio Code

可在 Windows、macOS 和 Linux 上运行的独立源代码编辑器。 Java 和 Web 开发人员的理想选择，包含大量扩展，支持几乎任何编程语言。

发行说明 →

使用 Visual Studio Code 即表示你同意其**许可**和**隐私声明**。

https://visualstudio.microsoft.com/zh-hans/downloads/

安装python扩展

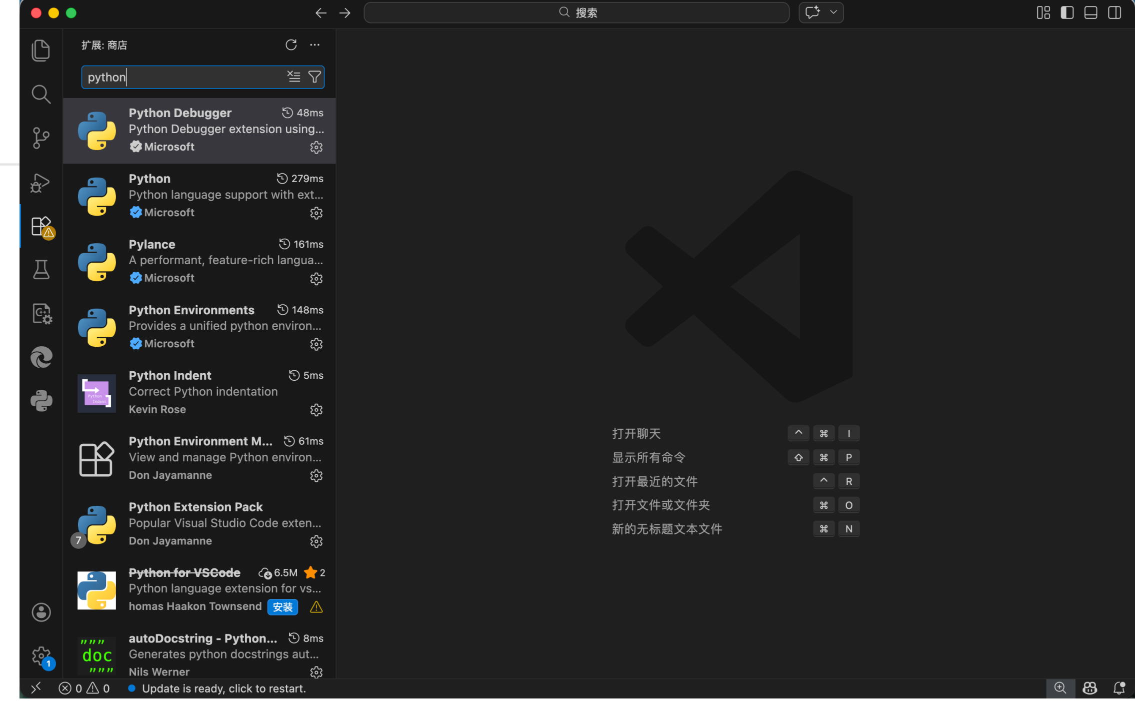
1. 确保已安装必要的软件

- **Python**：在终端输入 `python3 --version` 或 `python --version` 检查是否已安装。如果没有，请从 python.org 下载安装。
- **Visual Studio Code**：从 code.visualstudio.com 下载安装。

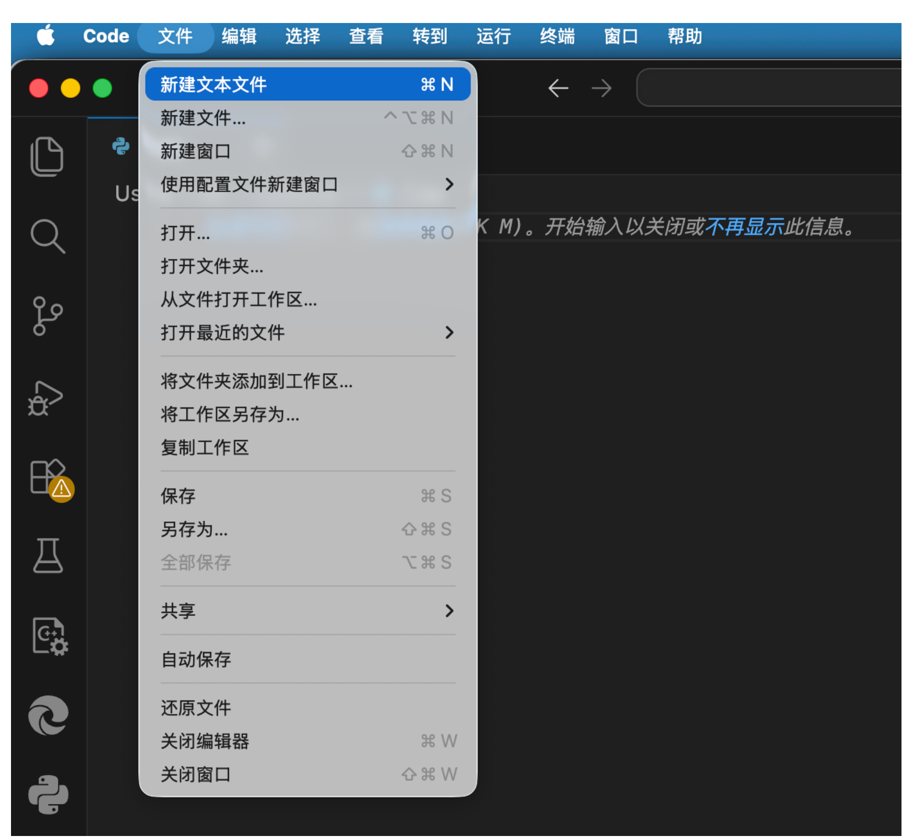
---

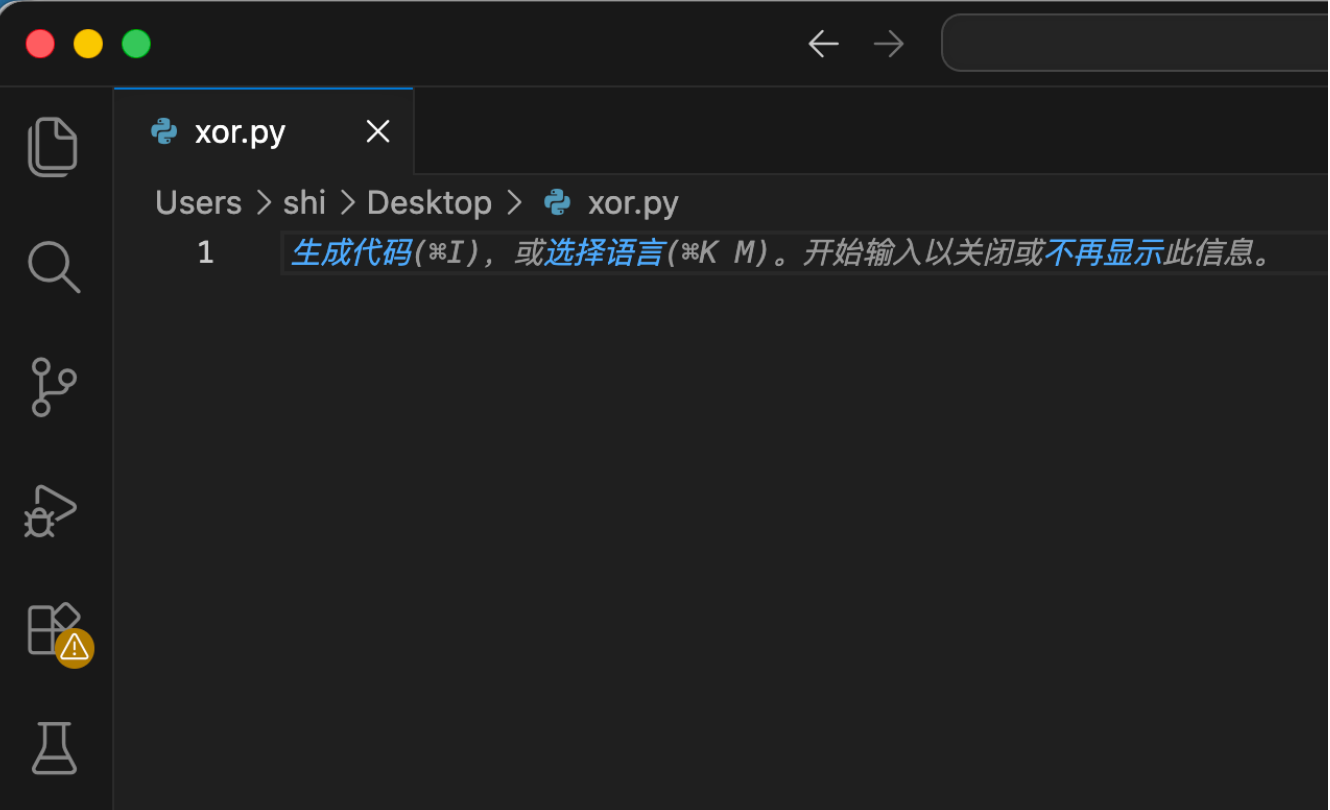
2. 安装 Python 扩展

1. 打开 VSCode。
2. 点击左侧活动栏的“扩展”图标（四个方块）。
3. 搜索 `Python`，找到由 Microsoft 发布的扩展，点击“安装”。

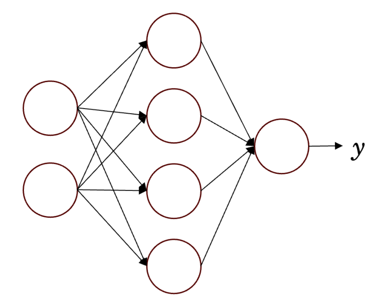


新建代码文件xor.py





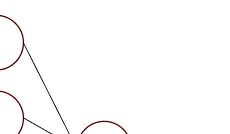
1. 定义网络




```python
import numpy as np

# 神经网络参数
input_size = 2
hidden_size = 3      # 使用3个隐藏神经元，确保能解决XOR
output_size = 1

# 初始化权重和偏置（随机小值）
W1 = np.random.randn(hidden_size, input_size) * 0.5   # 隐藏层权重 (3,2)
b1 = np.random.randn(hidden_size, 1) * 0.5            # 隐藏层偏置 (3,1)
W2 = np.random.randn(output_size, hidden_size) * 0.5  # 输出层权重 (1,3)
b2 = np.random.randn(output_size, 1) * 0.5            # 输出层偏置 (1,1)
```



1. 定义网络

❒ numpy包:

- NumPy是Python的科学计算基础库。
- 提供高性能的多维数组对象（ndarray）。
- 支持向量化运算、广播、线性代数、随机数等。
- 是很多其他库（如Pandas、SciPy、Matplotlib）的基础。
- 简要示例：创建数组、基本运算。

📦 安装与导入

```bash
pip install numpy
```

在代码中导入：

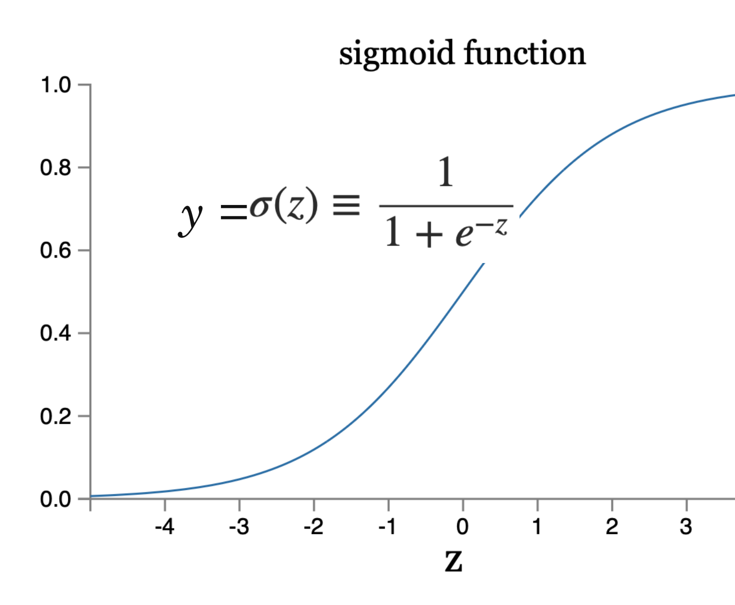
```python
import numpy as np   # 常用别名 np
```

2. 前向推理

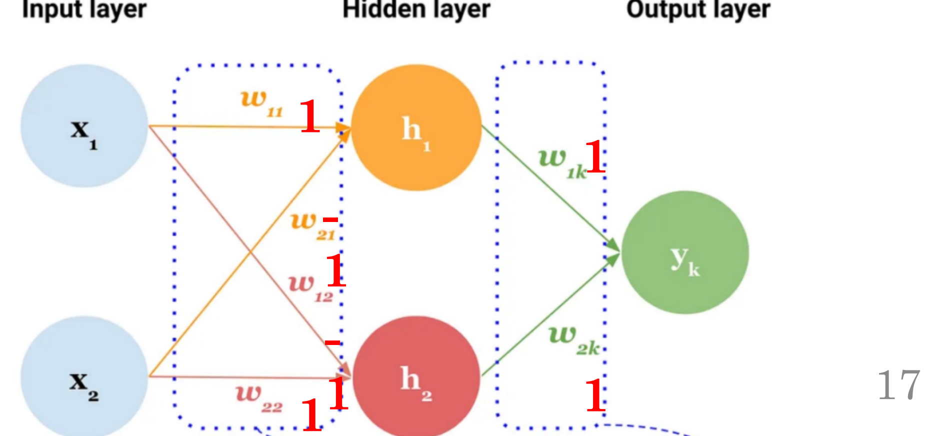
```python
# 1. 准备XOR数据
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=np.float64)  # 输入 (4,2)
y = np.array([[0], [1], [1], [0]], dtype=np.float64)              # 输出 (4,1)

# 2. 定义Sigmoid函数及其导数
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))
```



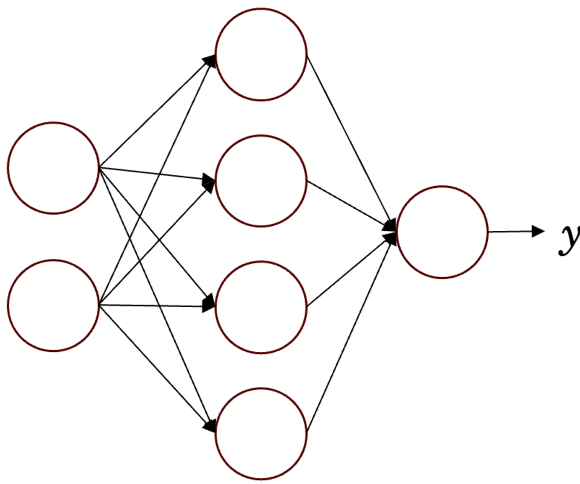

```python
# ---------- 前向传播 ----------
# 输入层到隐藏层
z1 = np.dot(W1, X.T) + b1          # (3,4)  (隐藏层输入)
a1 = sigmoid(z1)                   # (3,4)  (隐藏层输出)
# 隐藏层到输出层
z2 = np.dot(W2, a1) + b2           # (1,4)  (输出层输入)
a2 = sigmoid(z2)                   # (1,4)  (最终预测)
```



3. 反向传播

```
# ---------- 反向传播 ----------
# 输出层误差
dz2 = (a2.T - y) * sigmoid_derivative(z2.T)   # (4,1)
dz2 = dz2.T                                    # (1,4) 转回原始形状
# 输出层梯度
dW2 = np.dot(dz2, a1.T) / X.shape[0]           # (1,3)
db2 = np.sum(dz2, axis=1, keepdims=True) / X.shape[0]  # (1,1)

# 隐藏层误差
dz1 = np.dot(W2.T, dz2) * sigmoid_derivative(z1)  # (3,4)
# 隐藏层梯度
dW1 = np.dot(dz1, X) / X.shape[0]                # (3,2)
db1 = np.sum(dz1, axis=1, keepdims=True) / X.shape[0]  # (3,1)
```

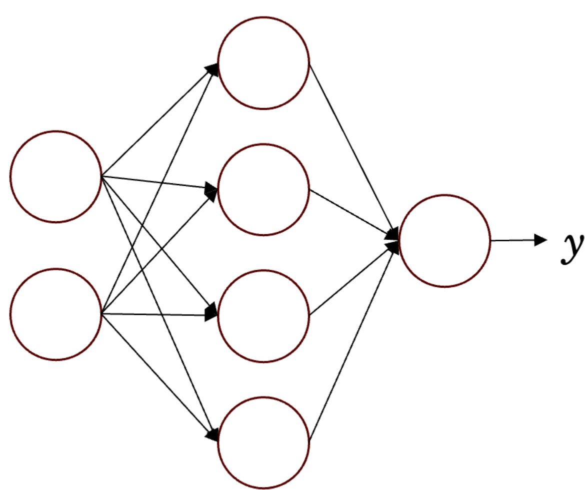


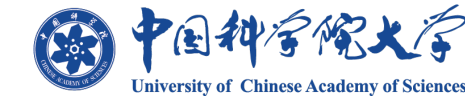
4. 参数更新

```
learning_rate = 0.5
```

```
# 更新参数
W2 -= learning_rate * dW2
b2 -= learning_rate * db2
W1 -= learning_rate * dW1
b1 -= learning_rate * db1

# 每1000轮打印损失
if (epoch + 1) % 1000 == 0:
    print(f"Epoch {epoch+1}/{epochs}, Loss: {loss:.6f}")
```





5. 迭代训练

```python
epochs = 10000
```

```python
for epoch in range(epochs):
    # ---------- 前向传播 ----------
    # 输入层到隐藏层
    z1 = np.dot(W1, X.T) + b1          # (3,4) (隐藏层输入)
    a1 = sigmoid(z1)                   # (3,4) (隐藏层输出)
    # 隐藏层到输出层
    z2 = np.dot(W2, a1) + b2           # (1,4) (输出层输入)
    a2 = sigmoid(z2)                   # (1,4) (最终预测)

    # 计算损失 (均方误差)
    loss = np.mean((a2.T - y) ** 2)    # 标量

    # ---------- 反向传播 ----------
    # 输出层误差
    dz2 = (a2.T - y) * sigmoid_derivative(z2.T)   # (4,1)
    dz2 = dz2.T                                    # (1,4) 转回原始形状
    # 输出层梯度
    dW2 = np.dot(dz2, a1.T) / X.shape[0]           # (1,3)
    db2 = np.sum(dz2, axis=1, keepdims=True) / X.shape[0]  # (1,1)

    # 隐藏层误差
    dz1 = np.dot(W2.T, dz2) * sigmoid_derivative(z1)  # (3,4)
    # 隐藏层梯度
    dW1 = np.dot(dz1, X) / X.shape[0]                 # (3,2)
    db1 = np.sum(dz1, axis=1, keepdims=True) / X.shape[0]  # (3,1)

    # 更新参数
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
```

6. 模型测试

```
shi@wangshideMacBook-Air Desktop % /usr/bin/python3 /Users/shi/Desktop/xor.py
开始训练...
Epoch 1000/10000, Loss: 0.250029
Epoch 2000/10000, Loss: 0.249951
Epoch 3000/10000, Loss: 0.249831
Epoch 4000/10000, Loss: 0.249200
Epoch 5000/10000, Loss: 0.241932
Epoch 6000/10000, Loss: 0.196567
Epoch 7000/10000, Loss: 0.123989
Epoch 8000/10000, Loss: 0.026427
Epoch 9000/10000, Loss: 0.010106
Epoch 10000/10000, Loss: 0.005768
训练完成!

预测结果:
输入: [0. 0.] -> 预测概率: 0.0807, 预测类别: 0, 真实类别: 0
输入: [0. 1.] -> 预测概率: 0.9274, 预测类别: 1, 真实类别: 1
输入: [1. 0.] -> 预测概率: 0.9279, 预测类别: 1, 真实类别: 1
输入: [1. 1.] -> 预测概率: 0.0780, 预测类别: 0, 真实类别: 0
```

7. 运行

```
shi@wangshideMacBook-Air Desktop % /usr/bin/python3 /Users/shi/Desktop/xor.py
开始训练...
Epoch 1000/10000, Loss: 0.250029
Epoch 2000/10000, Loss: 0.249951
Epoch 3000/10000, Loss: 0.249831
Epoch 4000/10000, Loss: 0.249200
Epoch 5000/10000, Loss: 0.241932
Epoch 6000/10000, Loss: 0.196567
Epoch 7000/10000, Loss: 0.123989
Epoch 8000/10000, Loss: 0.026427
Epoch 9000/10000, Loss: 0.010106
Epoch 10000/10000, Loss: 0.005768
训练完成!

预测结果:
输入: [0. 0.] -> 预测概率: 0.0807, 预测类别: 0, 真实类别: 0
输入: [0. 1.] -> 预测概率: 0.9274, 预测类别: 1, 真实类别: 1
输入: [1. 0.] -> 预测概率: 0.9279, 预测类别: 1, 真实类别: 1
输入: [1. 1.] -> 预测概率: 0.0780, 预测类别: 0, 真实类别: 0
```

整体回顾

```python
import numpy as np

# 设置随机种子以确保结果可复现
np.random.seed(42)

# 1. 准备XOR数据
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=np.float64)  # 输入 (4,2)
y = np.array([[0], [1], [1], [0]], dtype=np.float64)              # 输出 (4,1)

# 2. 定义Sigmoid函数及其导数
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

# 3. 神经网络参数
input_size = 2
hidden_size = 3      # 使用3个隐藏神经元，确保能解决XOR
output_size = 1
learning_rate = 0.5
epochs = 10000

# 初始化权重和偏置（随机小值）
W1 = np.random.randn(hidden_size, input_size) * 0.5   # 隐藏层权重 (3,2)
b1 = np.random.randn(hidden_size, 1) * 0.5            # 隐藏层偏置 (3,1)
W2 = np.random.randn(output_size, hidden_size) * 0.5  # 输出层权重 (1,3)
b2 = np.random.randn(output_size, 1) * 0.5            # 输出层偏置 (1,1)
```

```python
# 4. 训练过程
print("开始训练...")
for epoch in range(epochs):
    # ---------- 前向传播 ----------
    # 输入层到隐藏层
    z1 = np.dot(W1, X.T) + b1          # (3,4) (隐藏层输入)
    a1 = sigmoid(z1)                   # (3,4) (隐藏层输出)
    # 隐藏层到输出层
    z2 = np.dot(W2, a1) + b2           # (1,4) (输出层输入)
    a2 = sigmoid(z2)                   # (1,4) (最终预测)

    # 计算损失（均方误差）
    loss = np.mean((a2.T - y) ** 2)    # 标量

    # ---------- 反向传播 ----------
    # 输出层误差
    dz2 = (a2.T - y) * sigmoid_derivative(z2.T)   # (4,1)
    dz2 = dz2.T                                    # (1,4) 转回原始形状
    # 输出层梯度
    dW2 = np.dot(dz2, a1.T) / X.shape[0]           # (1,3)
    db2 = np.sum(dz2, axis=1, keepdims=True) / X.shape[0]  # (1,1)

    # 隐藏层误差
    dz1 = np.dot(W2.T, dz2) * sigmoid_derivative(z1)  # (3,4)
    # 隐藏层梯度
    dW1 = np.dot(dz1, X) / X.shape[0]                # (3,2)
    db1 = np.sum(dz1, axis=1, keepdims=True) / X.shape[0]  # (3,1)

    # 更新参数
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
```

1 手搓异或网络

2 PyTorch版异或网络

3

PyTorch深度学习开发框架

- 开源的深度学习框架，由 Facebook AI Research（FAIR）主导开发，深受研究人员喜爱，大量最新论文代码基于 PyTorch

Register for PyTorch Conference Europe 2026, April 7–8, Paris

PyTorch　Learn　Community　Projects　Docs　Blog & News　About　JOIN

PyTorch

PyTorch is an open source deep learning framework built to be flexible and modular for research, with the stability and support needed for production deployment.

PyTorch is an open source machine learning framework that accelerates the path from research prototyping to production deployment. Built to offer maximum flexibility and speed, PyTorch supports dynamic computation graphs, enabling researchers and developers to iterate quickly and intuitively. Its Pythonic design and deep integration with native Python tools make it an accessible and powerful platform for building and training deep learning models at scale.

Widely adopted across academia and industry, PyTorch has become the framework of choice for cutting-edge research and commercial AI applications. It supports a broad range of use cases—from natural language processing and computer vision to reinforcement learning and generative AI—

什么是深度学习开发框架

- 不再让你手搓神经网络的包
- PyTorch内置了
  - 类似于 NumPy 的张量计算
  - 基于带自动微分系统的深度神经网络

Pytorch的常用函数

三、神经网络模块（`torch.nn`）

常用层

类	说明
`nn.Linear(in_features, out_features)`	全连接层
`nn.Conv2d(in_channels, out_channels, kernel_size)`	二维卷积层
`nn.ConvTranspose2d(...)`	二维转置卷积（反卷积）
`nn.MaxPool2d(kernel_size)`	最大池化
`nn.AvgPool2d(kernel_size)`	平均池化
`nn.BatchNorm2d(num_features)`	批归一化（二维）
`nn.Dropout(p)`	Dropout层
`nn.Embedding(num_embeddings, embedding_dim)`	词嵌入层
`nn.LSTM(input_size, hidden_size)`	LSTM层
`nn.GRU(...)`	GRU层

https://docs.pytorch.org/docs/stable/

Pytorch的常用函数

激活函数

函数/类	说明
`nn.ReLU()`	ReLU激活
`nn.Sigmoid()`	Sigmoid激活
`nn.Tanh()`	Tanh激活
`nn.LeakyReLU()`	Leaky ReLU
`nn.Softmax(dim)`	Softmax激活，通常用于多分类输出层
`nn.LogSoftmax(dim)`	LogSoftmax，与NLLLoss配合

https://docs.pytorch.org/docs/stable/

Pytorch的常用函数

损失函数

类	说明
`nn.MSELoss()`	均方误差损失（回归）
`nn.L1Loss()`	平均绝对误差损失
`nn.BCELoss()`	二元交叉熵损失（二分类，配合Sigmoid）
`nn.BCEWithLogitsLoss()`	结合Sigmoid的二元交叉熵（数值更稳定）
`nn.CrossEntropyLoss()`	交叉熵损失（多分类，输入为原始logits，内部包含Softmax）
`nn.NLLLoss()`	负对数似然损失（配合LogSoftmax）

https://docs.pytorch.org/docs/stable/

Pytorch的常用函数

四、优化器（`torch.optim`）

类	说明
`optim.SGD(params, lr, momentum=0)`	随机梯度下降（可带动量）
`optim.Adam(params, lr)`	Adam优化器
`optim.AdamW(params, lr)`	AdamW（带权重衰减的Adam）
`optim.RMSprop(params, lr)`	RMSprop
`optim.Adagrad(params, lr)`	Adagrad

https://docs.pytorch.org/docs/stable/

Pytorch的常用函数

常用方法：

- `optimizer.zero_grad()`：清除梯度
- `loss.backward()`：计算梯度
- `optimizer.step()`：更新参数
- `scheduler.step()`：学习率调度器更新（需配合`torch.optim.lr_scheduler`）

https://docs.pytorch.org/docs/stable/

Pytorch的常用函数

五、自动求导（`torch.autograd`）

函数/属性	说明
`tensor.requires_grad`	设置是否追踪梯度
`tensor.grad`	存储梯度
`tensor.backward()`	反向传播计算梯度
`with torch.no_grad():`	禁用梯度计算（推理、验证时用）
`torch.autograd.grad(outputs, inputs)`	手动计算梯度

https://docs.pytorch.org/docs/stable/

Pytorch的常用函数

七、模型保存与加载

函数	说明
`torch.save(model.state_dict(), path)`	保存模型参数（推荐）
`model.load_state_dict(torch.load(path))`	加载模型参数
`torch.save(model, path)`	保存完整模型（不推荐，易出兼容问题）
`torch.load(path)`	加载完整模型

https://docs.pytorch.org/docs/stable/

Pytorch的常用函数

九、其他常用函数

函数	说明
`torch.set_printoptions(precision, threshold)`	设置打印选项
`torch.manual_seed(seed)`	设置CPU随机种子
`torch.cuda.manual_seed(seed)`	设置GPU随机种子
`torch.distributions`	概率分布模块
`torch.nn.utils.clip_grad_norm_(parameters, max_norm)`	梯度裁剪

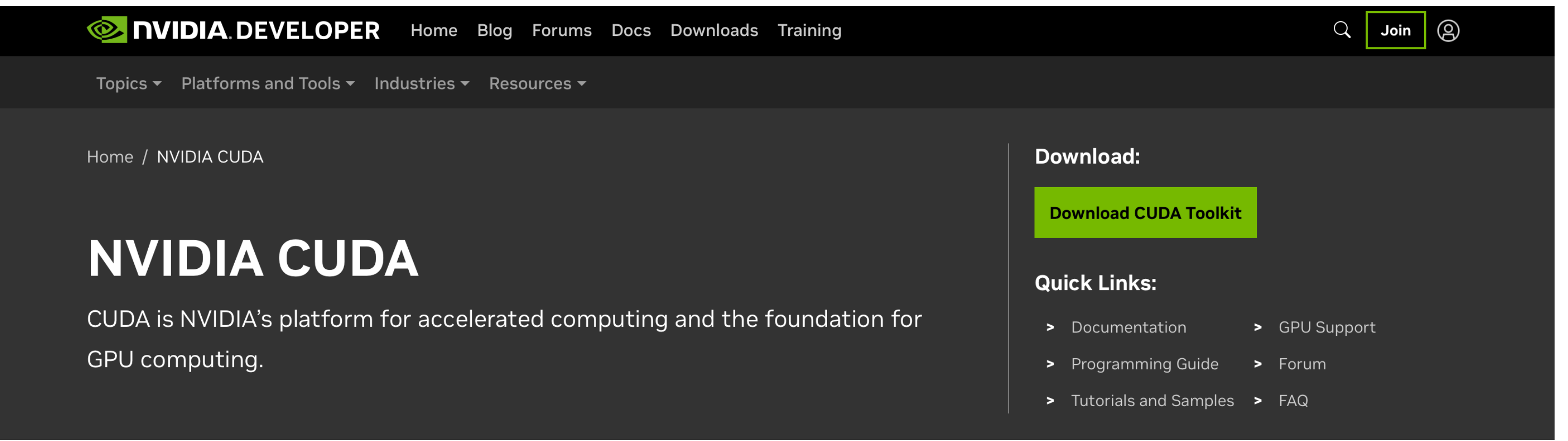
https://docs.pytorch.org/docs/stable/

PyTorch和TensorFlow

特性	TensorFlow	PyTorch
开发公司	Google	Facebook (FAIR)
计算图类型	静态计算图（定义后再执行）	**动态**计算图（定义即执行）
灵活性	低（计算图在编译时构建，不易修改）	高（计算图在执行时动态创建，易于修改和调试）
调试	较难（需要使用 tf.debugging 或外部工具调试）	容易（可以直接在 Python 中进行调试）
易用性	低（较复杂，API 较多，学习曲线较陡峭）	高（**API 简洁**，语法更加接近 Python，容易上手）
部署	强（支持广泛的硬件，如 TensorFlow Lite、TensorFlow.js）	较弱（部署工具和平台相对较少，虽然有 TensorFlow 支持）
社区支持	很强（成熟且庞大的社区，广泛的教程和文档）	很强（社区活跃，**特别是在学术界**，快速发展的生态）
模型训练	支持分布式训练，支持多种设备（如 CPU、GPU、TPU）	支持分布式训练，支持多 GPU、CPU 和 TPU
API 层级	高级API：Keras；低级API：TensorFlow Core	高级API：TorchVision、TorchText 等；低级API：Torch
性能	高（优化方面成熟，适合生产环境）	高（适合研究和原型开发，生产性能也在提升）
自动求导	通过 tf.GradientTape 实现动态求导（较复杂）	通过 autograd 动态求导（更简洁和直观）
调优与可扩展性	强（支持在多平台上运行，如 TensorFlow Serving 等）	较弱（虽然在学术和实验环境中表现优越，但生产环境支持相对较少）
框架灵活性	较低（TensorFlow 2.x 引入了动态图特性，但仍不完全灵活）	高（动态图带来更高的灵活性）
支持多种语言	支持多种语言（Python, C++, Java, JavaScript, etc.）	主要支持 Python（但也有 C++ API）
兼容性与迁移	TensorFlow 2.x 与旧版本兼容性较好	与 TensorFlow 兼容性差，迁移较难

PyTorch和CUDA

- CUDA（Compute Unified Device Architecture）是 NVIDIA 推出的并行计算平台和编程模型，允许开发者利用 GPU的强大算力进行通用计算，广泛应用于科学计算、深度学习、物理模拟、图形渲染等需要大规模数据并行的领域，是高性能计算与深度学习的基石
- PyTorch 封装了 CUDA 的复杂性，提供简洁的 Python API



PyTorch版xor网络

- 安装PyTorch
  - 无GPU：
    - pip install torch torchvision torchaudio #Mac平台用pip3
  - 有GPU：
    - pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118

0. 引入PyTorch包

1. 导入库

python 复制 下载

```python
import torch
import torch.nn as nn
import torch.optim as optim
```

- `import torch`：导入 PyTorch 主库，用于张量运算、自动求导等核心功能。
- `import torch.nn as nn`：导入神经网络模块，提供了构建网络所需的层（如 `Linear`）、激活函数（如 `Sigmoid`）和损失函数（如 `BCELoss`）。
- `import torch.optim as optim`：导入优化器模块，包含各种参数更新算法，如 `SGD`（随机梯度下降）。

2. 设置随机种子

python 复制 下载

```python
torch.manual_seed(42)
```

- 固定随机数种子为 42，确保每次运行代码时权重初始化、数据划分等随机过程结果一致，便于复现实验。

3. 准备数据

python 复制 下载

```python
X = torch.tensor([[0., 0.], [0., 1.], [1., 0.], [1., 1.]])
y = torch.tensor([[0.], [1.], [1.], [0.]])
```

- X：创建输入张量，形状为 (4, 2)，包含 XOR 问题的四个样本（两个特征）。
- y：创建标签张量，形状为 (4, 1)，对应每个样本的输出（0 或 1）。使用浮点数是因为后续的损失函数需要浮点类型。

4. 定义网络

```python
# 2. 定义神经网络模型
class XORNet(nn.Module):
    def __init__(self):
        super().__init__()
        # 隐藏层: 2输入 → 3输出 (使用3个隐藏神经元, 更容易收敛)
        self.hidden = nn.Linear(2, 3)
        # 输出层: 3输入 → 1输出
        self.output = nn.Linear(3, 1)
        # 激活函数
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.hidden(x))
        x = self.sigmoid(self.output(x))
        return x
```

4. 定义网络

python 复制 下载

```python
class XORNet(nn.Module):
```

- 定义一个继承自 `nn.Module` 的类，这是 PyTorch 中所有神经网络的基类。

`__init__` 方法

python 复制 下载

```python
    def __init__(self):
        super().__init__()
```

- `super().__init__()`：调用父类 `nn.Module` 的构造函数，完成必要的初始化。

python 复制 下载

```python
        self.hidden = nn.Linear(2, 3)
```

- 定义一个全连接层（线性层）`hidden`，输入特征数为 2，输出特征数为 3（即隐藏层有 3 个神经元）。该层用于提取输入的非线性组合。

4. 定义网络

python

```
        self.output
```

- 定义输出全连接层 ou

python

```
        self.sigmoid
```

- 定义 Sigmoid 激活函数
  区间，适合二分类概率

`forward` 方法

python

```python
    def forward(self, x):
        x = self.sigmoid(self.hidden(x))
```

- 输入 `x` 先经过隐藏线性层 `self.hidden`，得到隐藏层的线性输出，然后通过 Sigmoid 激活函数，得到隐藏层的激活值。

python 复制 下载

```python
        x = self.sigmoid(self.output(x))
```

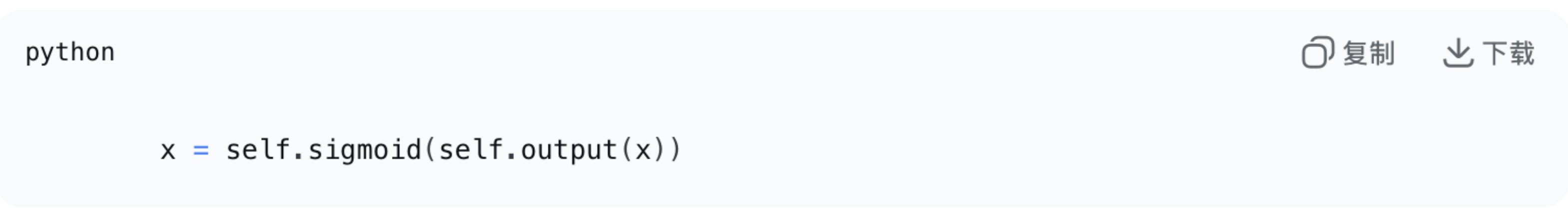
- 隐藏层的激活值再经过输出线性层 `self.output`，得到输出层的线性输出，最后再经过 Sigmoid 激活函数，得到最终预测概率（0~1 之间的值）。

python 复制 下载

```python
        return x
```

- 返回最终的预测值。

4. 定义网络




```python
        x = self.sigmoid(self.output(x))
```

- 隐藏层的激活值再经过输出线性层 `self.output`，得到输出层的线性输出，最后再经过 Sigmoid 激活函数，得到最终预测概率（0~1 之间的值）。

python 复制 下载

```python
        return x
```

- 返回最终的预测值。

5. 模型训练

```python
# 实例化模型
model = XORNet()

# 3. 定义损失函数和优化器
criterion = nn.BCELoss()            # 二分类交叉熵损失
optimizer = optim.SGD(model.parameters(), lr=0.5)  # 随机梯度下降

# 4. 训练
epochs = 5000
print("开始训练...")
for epoch in range(epochs):
    # 前向传播
    outputs = model(X)
    loss = criterion(outputs, y)

    # 反向传播与优化
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # 每1000个epoch打印一次损失
    if (epoch + 1) % 1000 == 0:
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")

print("训练完成！")
```

5. 模型训练

python 复制 下载

```python
model = XORNet()
```

- 创建 `XORNet` 类的一个实例，即我们的神经网络模型。此时模型参数（权重、偏置）被随机初始化（受前面设置的随机种子控制）。

5. 模型训练

定义损失函数和优化器

python　　复制　下载

```python
criterion = nn.BCELoss()
```

- `BCELoss` 是二元交叉熵损失函数（Binary Cross Entropy Loss），适用于二分类问题，要求模型输出是概率值（即经过 Sigmoid 后的结果）。计算公式为：
  `loss = -[y * log(p) + (1-y) * log(1-p)]`，其中 `p` 是预测概率。

python　　复制　下载

```python
optimizer = optim.SGD(model.parameters(), lr=0.5)
```

- 使用随机梯度下降（SGD）优化器，传入模型的所有可学习参数（`model.parameters()`），学习率 `lr=0.5`。该优化器将根据损失梯度更新参数。

5. 模型训练

训练模型

python 复制 下载

```python
epochs = 5000
print("开始训练...")
```

- 设定训练轮数（epoch）为 5000，打印提示信息。

训练循环

python 复制 下载

```python
for epoch in range(epochs):
```

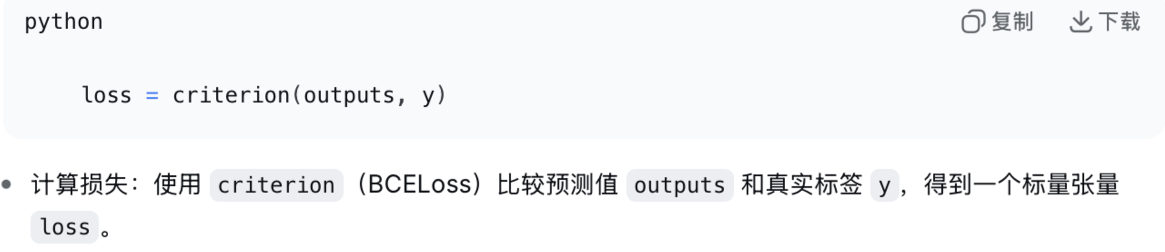
- 循环迭代 5000 次。

python 复制 下载

```python
    outputs = model(X)
```

- **前向传播**：将整个数据集 `X` 输入模型，得到预测值 `outputs`，形状为 (4, 1)。

5. 模型训练



python　复制　下载

```python
loss = criterion(outputs, y)
```

- 计算损失：使用 `criterion`（BCELoss）比较预测值 `outputs` 和真实标签 `y`，得到一个标量张量 `loss`。

python　复制　下载

```python
optimizer.zero_grad()
```

- 清除之前累积的梯度。因为每次反向传播后梯度会累加，所以每轮更新前必须清零。

python　复制　下载

```python
loss.backward()
```

- **反向传播**：自动计算损失关于所有可学习参数的梯度。PyTorch 的自动求导机制会沿计算图反向传播梯度。

python　复制　下载

```python
optimizer.step()
```

- 更新参数：优化器根据计算好的梯度更新模型参数（即执行一步 SGD）。

5. 模型训练

python 复制 下载

```python
    if (epoch + 1) % 1000 == 0:
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
```

- 每 1000 个 epoch 打印一次当前的损失值，`loss.item()` 将张量转为 Python 标量。

python 复制 下载

```python
print("训练完成！")
```

- 训练结束提示。

3. 模型测试

```python
# 5. 测试模型
with torch.no_grad():
    predictions = model(X)
    predicted = (predictions > 0.5).float()
    print("\n测试结果：")
    for i in range(len(X)):
        print(f"输入：{X[i].numpy()} -> 预测概率：{predictions[i].item():.4f}, "
              f"预测类别：{int(predicted[i].item())}, 真实类别：{int(y[i].item())}")
```

```
shi@wangshideMacBook-Air Desktop % /usr/bin/python3 /Users/shi/Desktop/xor_pt.py
开始训练...
Epoch [1000/5000], Loss: 0.5077
Epoch [2000/5000], Loss: 0.0232
Epoch [3000/5000], Loss: 0.0100
Epoch [4000/5000], Loss: 0.0063
Epoch [5000/5000], Loss: 0.0046
训练完成！

测试结果：
输入：[0. 0.] -> 预测概率：0.0048, 预测类别：0, 真实类别：0
输入：[0. 1.] -> 预测概率：0.9933, 预测类别：1, 真实类别：1
输入：[1. 0.] -> 预测概率：0.9979, 预测类别：1, 真实类别：1
输入：[1. 1.] -> 预测概率：0.0047, 预测类别：0, 真实类别：0
```

目录

1 手搓异或网络

2 PyTorch版异或网络

3 NNLM

4

NNLM

（3）输出每个词的概率以及最可能的词

p(办|给…一只手)=0.075

i-th output = $P(w_t = i \mid context)$

p(机|给…一只手)=0.000011

p(办|给…一只手)=0.000065

p(铸|给…一只手)=0.000001

softmax

most computation here

（2）神经网络

tanh

（1）词分布式特征向量
(distributed feature vectors)

[0.8480, -0.4750, -0.1357, 0.4134]

[-0.4832, -1.4191, 0.6283, 0.0977]

[0.0887, -0.0405, -0.1081, -0.2165]

$C(w_{t-n+1})$ … $C(w_{t-2})$ $C(w_{t-1})$

Table look−up in C

Matrix C

shared parameters across words

index for w_{t-n+1} index for w_{t-2} index for w_{t-1}

给 … 一只 手

网络定义

创建一个简单的神经网络语言模型：一个嵌入层后接一个线性层（全连接层）

```
import torch
import torch.nn as nn
import torch.optim as optim


class NNLMModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim):
        super(NNLMModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.fc = nn.Linear(embedding_dim, hidden_dim)
        self.output_layer = nn.Linear(hidden_dim, vocab_size)

    def forward(self, x):
        # x 是输入的单词索引，形状为 [batch_size, sequence_length]
        embedded = self.embedding(x)  # 嵌入层输出形状为 [batch_size, sequence_length, embedding_dim]
        embedded = embedded.permute(1, 0, 2)  # 调整形状为 [sequence_length, batch_size, embedding_dim]
        # 这里我们对每个时间步的输出应用全连接层
        output = torch.tanh(self.fc(embedded))  # 使用tanh激活函数
        output = self.output_layer(output)  # 应用输出层得到最终的预测
        return output
```

模型训练

```
vocab_size = 10000  # 假设词汇表大小为10000
embedding_dim = 128  # 嵌入层维度
hidden_dim = 256     # 隐藏层维度

model = NNLMModel(vocab_size, embedding_dim, hidden_dim)
optimizer = optim.Adam(model.parameters(), lr=0.001)


# 假设 x 是输入数据，target 是目标输出（通常是一个one-hot编码或索引形式）
x = torch.randint(0, vocab_size, (10, 5))  # 随机生成一些输入数据，形状为 [batch_size, sequence_length]
target = torch.randint(0, vocab_size, (10, 5))  # 随机生成目标数据，形状相同

optimizer.zero_grad()  # 清空梯度
output = model(x)  # 前向传播
loss = nn.CrossEntropyLoss()(output.view(-1, vocab_size), target.view(-1))  # 计算损失，注意需要展平以匹配输出和目标形状
loss.backward()  # 反向传播计算梯度
optimizer.step()  # 更新参数
```

目录

1 手搓异或网络

2 PyTorch版异或网络

3 NNLM

4 Transformer

内置实现

PyTorch已内置Transformer实现

CLASS torch.nn.Transformer(*d_model=512*, *nhead=8*, *num_encoder_layers=6*, *num_decoder_layers=6*, *dim_feedforward=2048*, *dropout=0.1*, *activation=<function relu>*, *custom_encoder=None*, *custom_decoder=None*, *layer_norm_eps=1e-05*, *batch_first=False*, *norm_first=False*, *device=None*, *dtype=None*) [SOURCE]

Examples::

```
>>> transformer_model = nn.Transformer(nhead=16, num_encoder_layers=12)
>>> src = torch.rand((10, 32, 512))
>>> tgt = torch.rand((20, 32, 512))
>>> out = transformer_model(src, tgt)
```

https://docs.pytorch.org/docs/1.11/generated/torch.nn.Transformer.html

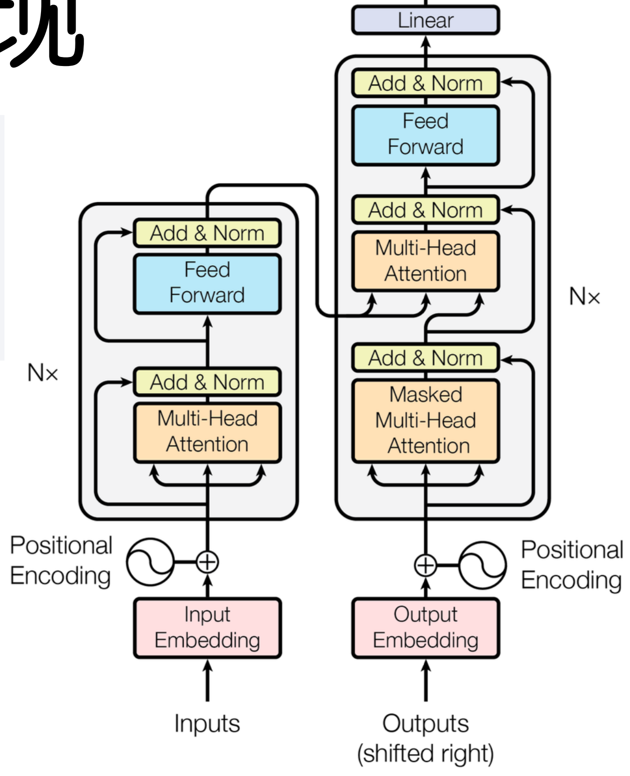


Figure 1: The Transformer - model architecture.

1. 位置编码

```python
# 1. 位置编码 (Positional Encoding)
class PositionalEncoding(nn.Module):
    """为输入序列添加位置信息"""
    def __init__(self, d_model, max_len=5000, dropout=0.1):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        # 创建位置编码矩阵 (max_len, d_model)
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)  # 增加batch维度, 形状 (1, max_len, d_model)
        self.register_buffer('pe', pe)  # 不作为模型参数，但会保存到state_dict

    def forward(self, x):
        # x: (batch, seq_len, d_model)
        x = x + self.pe[:, :x.size(1), :]
        return self.dropout(x)
```

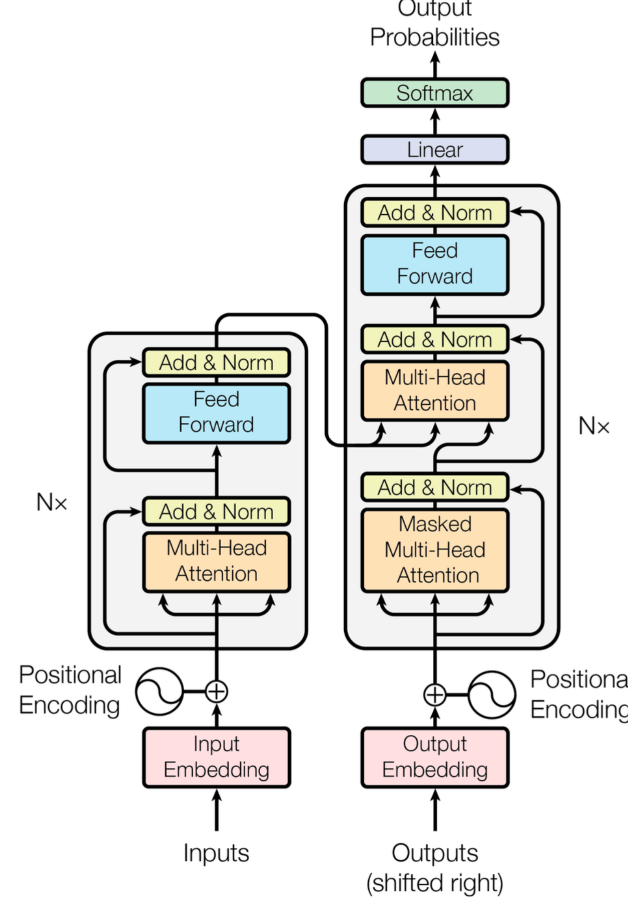


Figure 1: The Transformer - model architecture.

2. 缩放点积注意力

```python
# 2. 缩放点积注意力
def scaled_dot_product_attention(query, key, value, mask=None, dropout=None):
    """计算缩放点积注意力"""
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = F.softmax(scores, dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

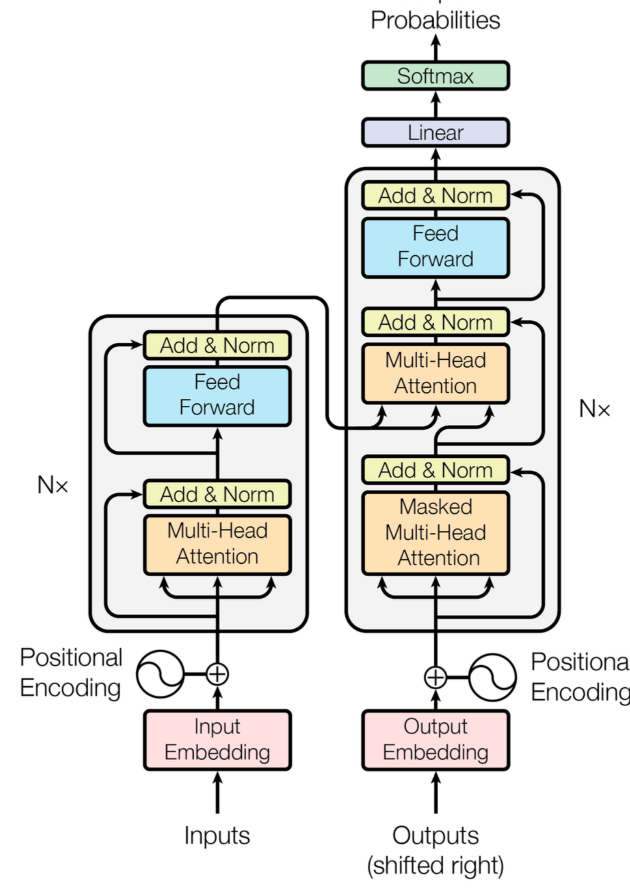


Figure 1: The Transformer - model architecture.

3. 多头注意力

```python
# 3. 多头注意力 (Multi-Head Attention)
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads, dropout=0.1):
        super().__init__()
        assert d_model % num_heads == 0
        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        self.w_q = nn.Linear(d_model, d_model)
        self.w_k = nn.Linear(d_model, d_model)
        self.w_v = nn.Linear(d_model, d_model)
        self.w_o = nn.Linear(d_model, d_model)

        self.dropout = nn.Dropout(dropout)
```

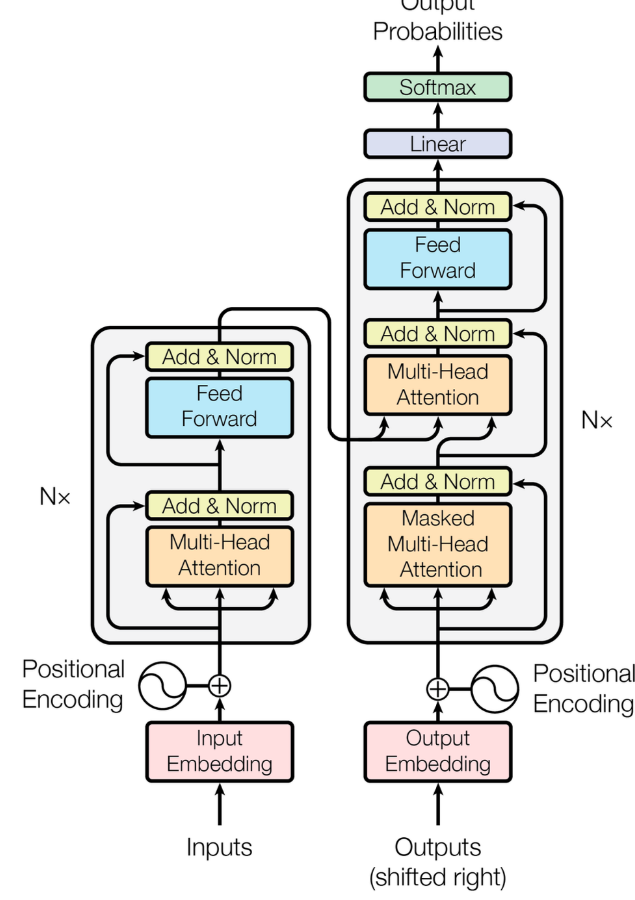


Figure 1: The Transformer - model architecture.

3. 多头注意力

```
    def forward(self, query, key, value, mask=None):
        # query, key, value: (batch, seq_len, d_model)
        batch_size = query.size(0)

        # 线性变换并拆分为多头
        Q = self.w_q(query).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        K = self.w_k(key).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        V = self.w_v(value).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)

        # 计算注意力
        attn_output, attn_weights = scaled_dot_product_attention(Q, K, V, mask, self.dropout)

        # 合并多头
        attn_output = attn_output.transpose(1, 2).contiguous().view(batch_size, -1, self.d_model)
        return self.w_o(attn_output)
```

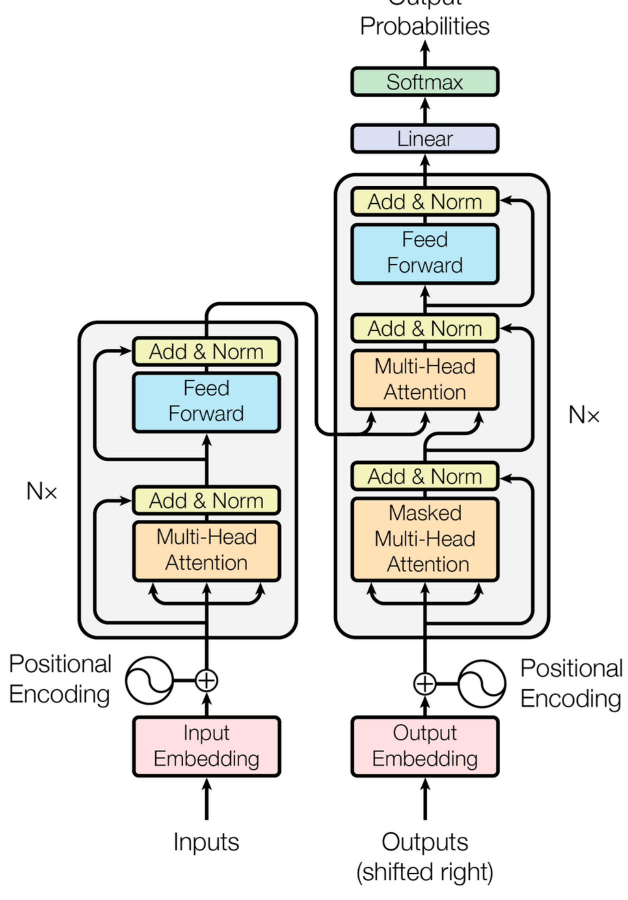


Figure 1: The Transformer - model architecture.

4. 前馈网络

```python
# 4. 前馈网络 (Feed-Forward Network)
class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        super().__init__()
        self.linear1 = nn.Linear(d_model, d_ff)
        self.linear2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.linear2(self.dropout(F.relu(self.linear1(x))))
```

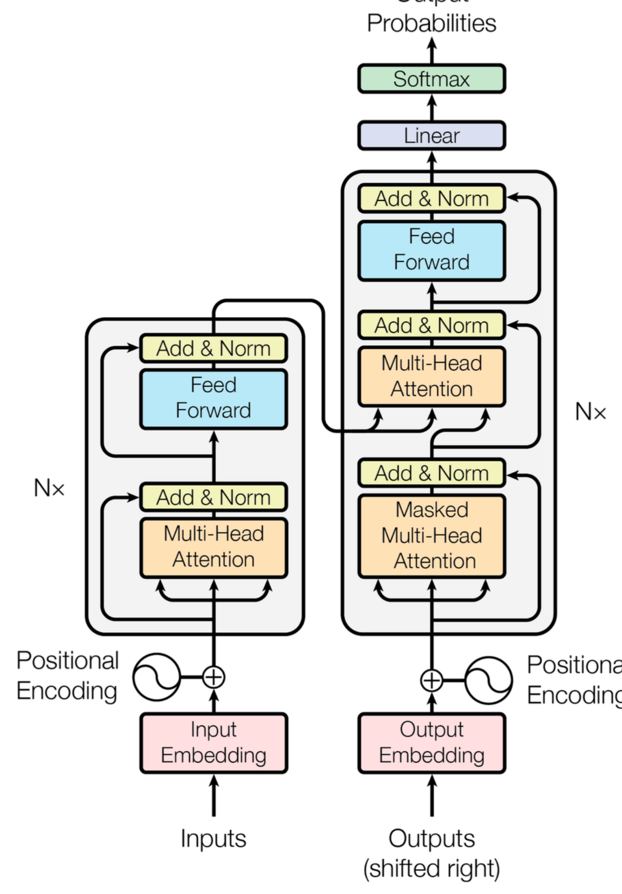


Figure 1: The Transformer - model architecture.

5. 编码器层

```python
# 5. 编码器层
class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads, dropout)
        self.feed_forward = FeedForward(d_model, d_ff, dropout)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # 自注意力 + 残差连接 + 层归一化
        attn_output = self.self_attn(x, x, x, mask)
        x = self.norm1(x + self.dropout(attn_output))

        # 前馈网络 + 残差连接 + 层归一化
        ff_output = self.feed_forward(x)
        x = self.norm2(x + self.dropout(ff_output))
        return x
```

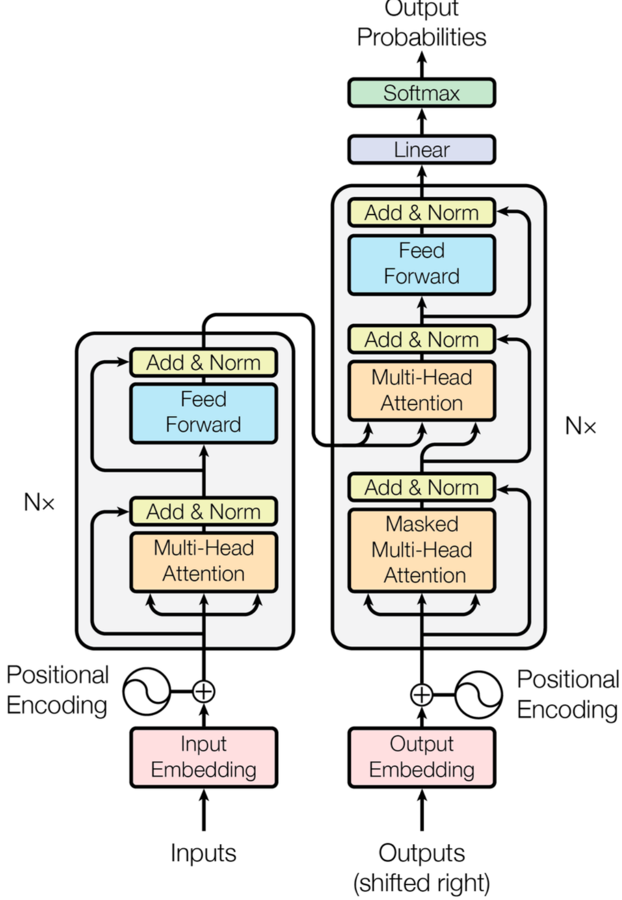


Figure 1: The Transformer - model architecture.

6. 编码器

```
# 6. 编码器
class Encoder(nn.Module):
    def __init__(self, vocab_size, d_model, num_heads, d_ff, num_layers, max_len, dropout=
0.1):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.positional_encoding = PositionalEncoding(d_model, max_len, dropout)
        self.layers = nn.ModuleList([EncoderLayer(d_model, num_heads, d_ff, dropout) for _
in range(num_layers)])
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # x: (batch, seq_len) 词索引
        x = self.embedding(x) * math.sqrt(self.embedding.embedding_dim)  # 缩放
        x = self.positional_encoding(x)
        for layer in self.layers:
            x = layer(x, mask)
        return x
```

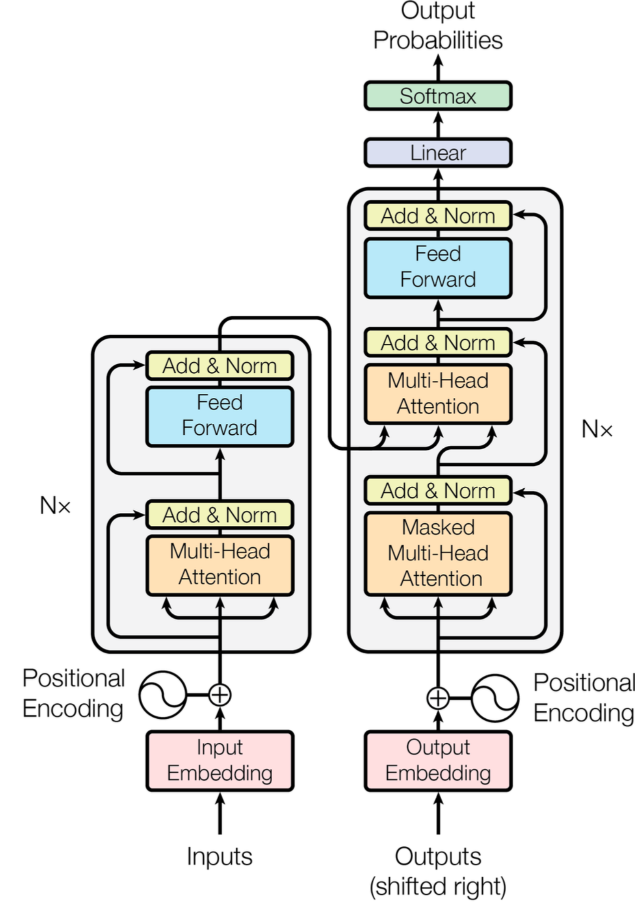


Figure 1: The Transformer - model architecture.

7. 解码器层

```python
# 7. 解码器层
class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads, dropout)
        self.cross_attn = MultiHeadAttention(d_model, num_heads, dropout)
        self.feed_forward = FeedForward(d_model, d_ff, dropout)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, enc_output, src_mask=None, tgt_mask=None):
        # 自注意力（目标序列）
        attn_output = self.self_attn(x, x, x, tgt_mask)
        x = self.norm1(x + self.dropout(attn_output))

        # 编码器-解码器注意力
        attn_output = self.cross_attn(x, enc_output, enc_output, src_mask)
        x = self.norm2(x + self.dropout(attn_output))

        # 前馈网络
        ff_output = self.feed_forward(x)
        x = self.norm3(x + self.dropout(ff_output))
        return x
```

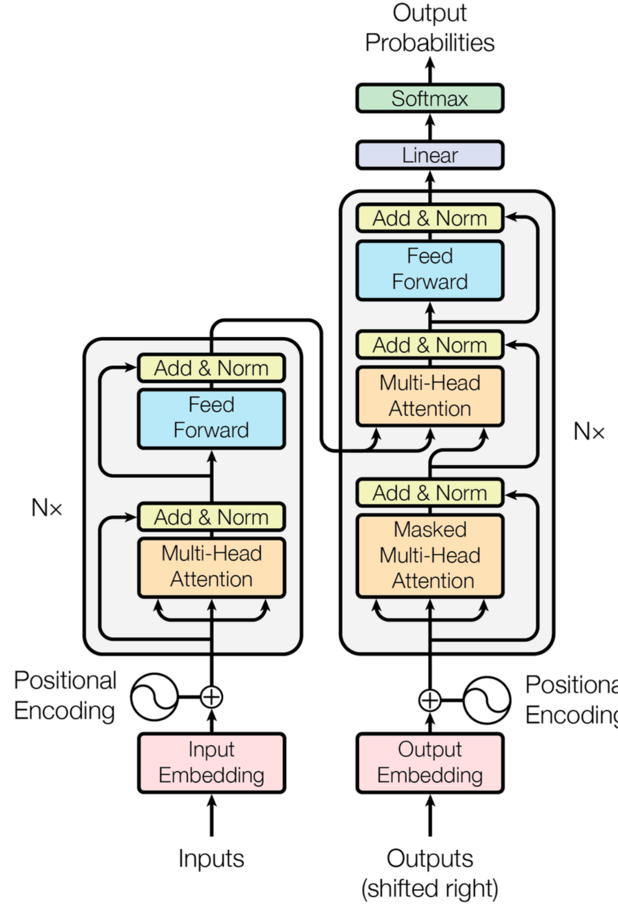


Figure 1: The Transformer - model architecture.

8. 解码器

```python
# 8. 解码器
class Decoder(nn.Module):
    def __init__(self, vocab_size, d_model, num_heads, d_ff, num_layers, max_len, dropout=0.1):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.positional_encoding = PositionalEncoding(d_model, max_len, dropout)
        self.layers = nn.ModuleList([DecoderLayer(d_model, num_heads, d_ff, dropout) for _ in range(num_layers)])
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, enc_output, src_mask=None, tgt_mask=None):
        x = self.embedding(x) * math.sqrt(self.embedding.embedding_dim)
        x = self.positional_encoding(x)
        for layer in self.layers:
            x = layer(x, enc_output, src_mask, tgt_mask)
        return x
```

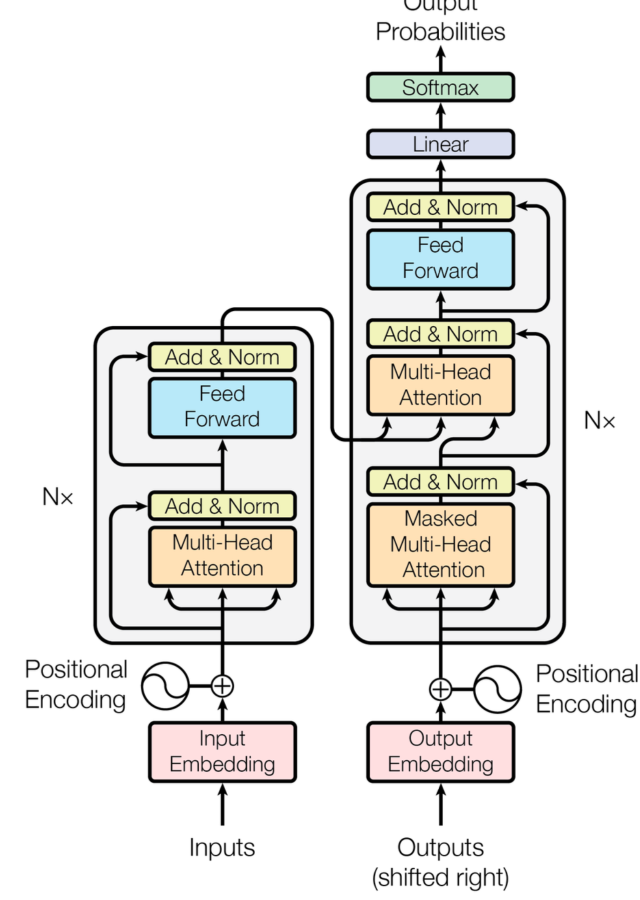


Figure 1: The Transformer - model architecture.

9. Transformer

```python
# 9. 完整Transformer模型
class Transformer(nn.Module):
    def __init__(self, src_vocab_size, tgt_vocab_size, d_model=512, num_heads=8, d_ff=2048,
num_layers=6, max_len=5000, dropout=0.1):
        super().__init__()
        self.encoder = Encoder(src_vocab_size, d_model, num_heads, d_ff, num_layers, max_le
n, dropout)
        self.decoder = Decoder(tgt_vocab_size, d_model, num_heads, d_ff, num_layers, max_le
n, dropout)
        self.fc_out = nn.Linear(d_model, tgt_vocab_size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, src, tgt, src_mask=None, tgt_mask=None):
        # src: (batch, src_len), tgt: (batch, tgt_len)
        enc_output = self.encoder(src, src_mask)
        dec_output = self.decoder(tgt, enc_output, src_mask, tgt_mask)
        output = self.fc_out(dec_output)
        return output

    def generate_square_subsequent_mask(self, sz):
        """生成下三角掩码，用于解码器自注意力（屏蔽未来位置）"""
        mask = torch.triu(torch.ones(sz, sz), diagonal=1)  # 上三角为1，表示屏蔽
        return mask == 0  # True表示可见，False表示屏蔽
```

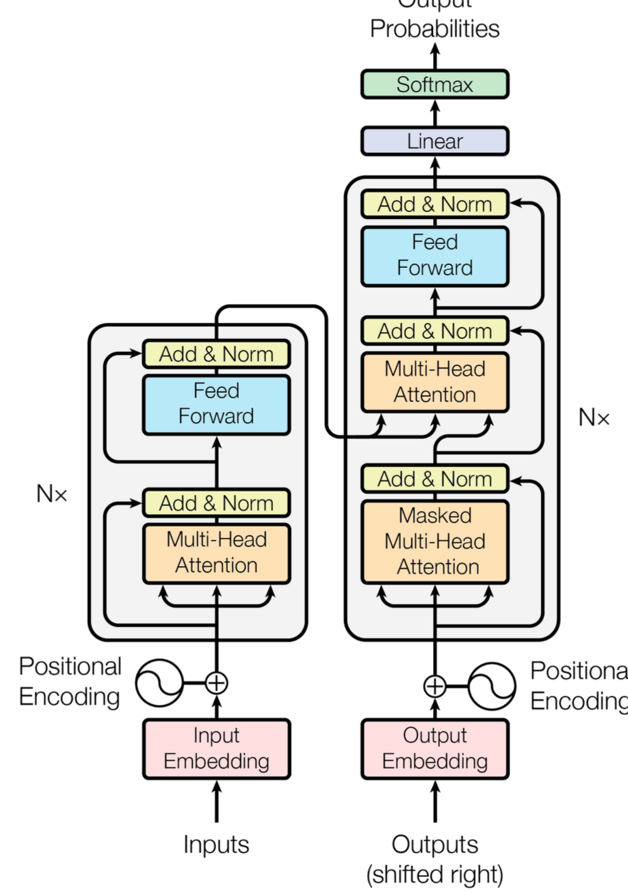


Figure 1: The Transformer - model architecture.

10. 使用示例

```
if __name__ == "__main__":
    # 超参数
    src_vocab_size = 1000
    tgt_vocab_size = 1000
    d_model = 512
    num_heads = 8
    d_ff = 2048
    num_layers = 6
    max_len = 100
    dropout = 0.1
    batch_size = 32
    src_len = 20
    tgt_len = 25

    # 创建模型
    model = Transformer(src_vocab_size, tgt_vocab_size, d_model, num_heads, d_ff, num
s, max_len, dropout)
```

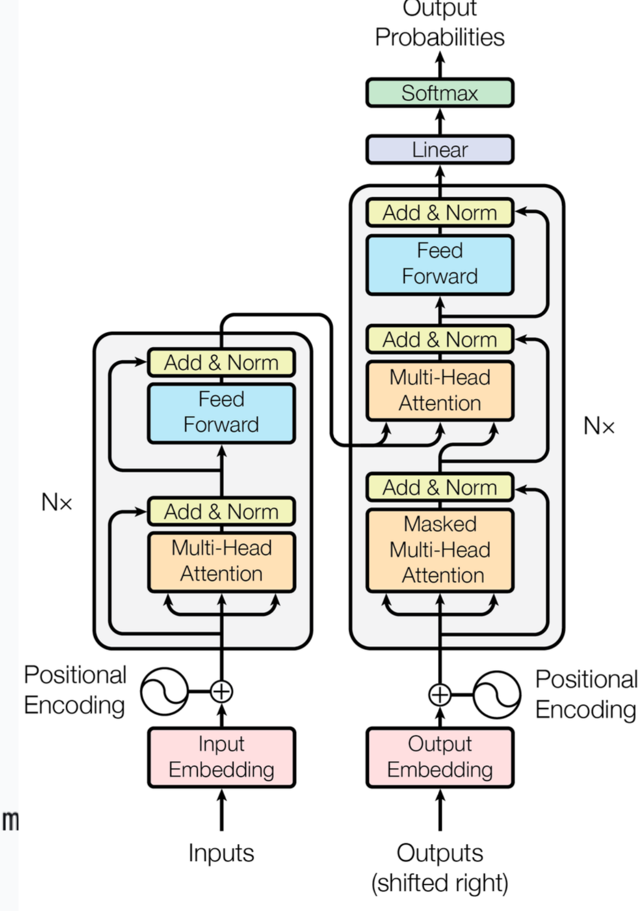


Figure 1: The Transformer - model architecture.

10. 使用示例

```
# 生成随机数据（模拟序列索引）
src = torch.randint(0, src_vocab_size, (batch_size, src_len))
tgt = torch.randint(0, tgt_vocab_size, (batch_size, tgt_len))

# 创建掩码
src_mask = None  # 通常不需要掩码，除非有填充
tgt_mask = model.generate_square_subsequent_mask(tgt_len).unsqueeze(0).repeat(batch_size, 1, 1)  # (batch, tgt_len, tgt_len)

# 前向传播
output = model(src, tgt, src_mask, tgt_mask)
print(f"输出形状: {output.shape}")  # (batch, tgt_len, tgt_vocab_size)

# 计算损失（假设有目标标签，这里仅演示）
# loss_fn = nn.CrossEntropyLoss(ignore_index=0)
# loss = loss_fn(output.view(-1, tgt_vocab_size), tgt.view(-1))
# print(f"Loss: {loss.item()}")
```

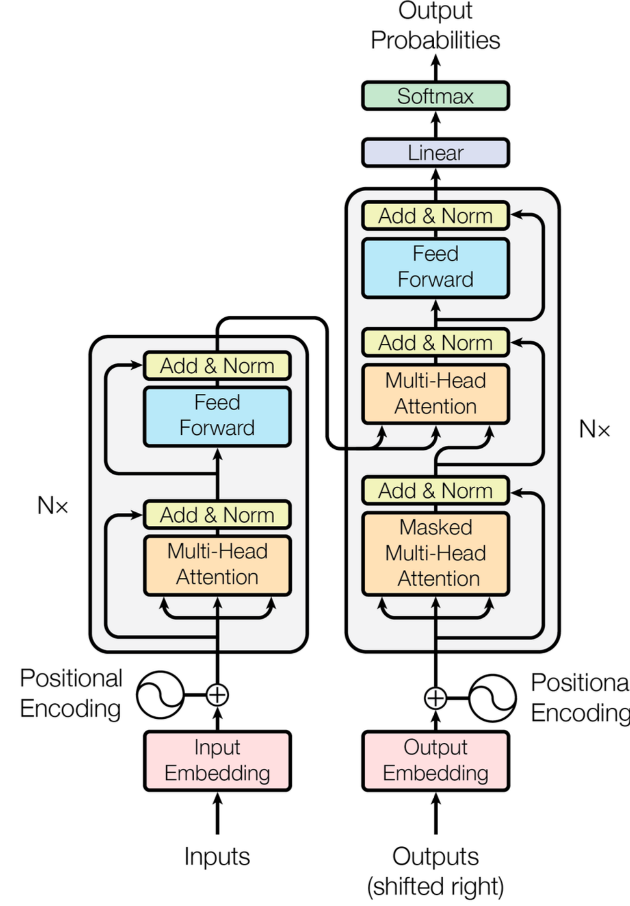


Figure 1: The Transformer - model architecture.

本节复习

- ❑ VSCode、Python
- ❑ PyTorch、CUDA
- ❑ NNLM
- ❑ Transformer

致 谢

- 胡玥、曹亚男、方芳：国科大《自然语言处理基础》
- 曹亚男、任昱冰：国科大《深度学习与自然语言处理概述》

THANKS
https://ictkc.github.io/teaching/2026spring-nlp