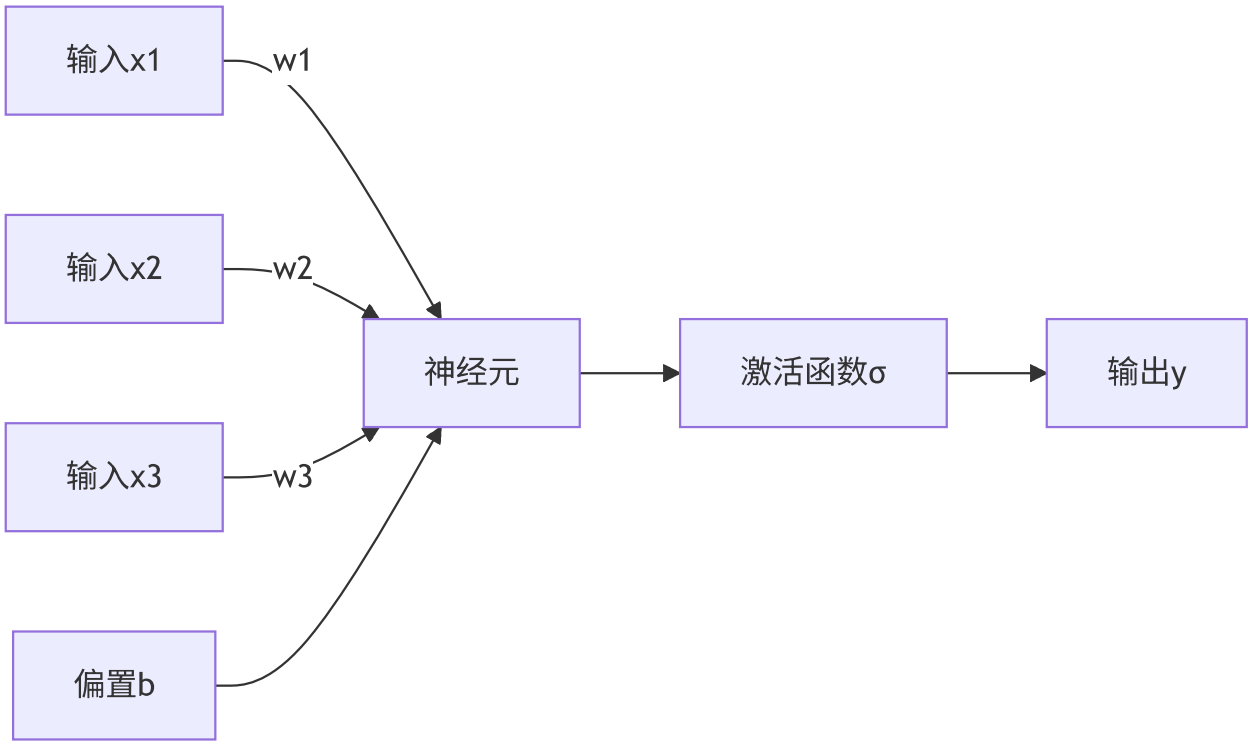
第1章 深度学习基础

> 万丈高楼平地起，深度学习的基础知识是理解大模型的基石。

1.1 神经网络基本原理

1.1.1 神经元模型

神经网络的基本单元是神经元（Neuron），它模拟了生物神经元的工作方式。



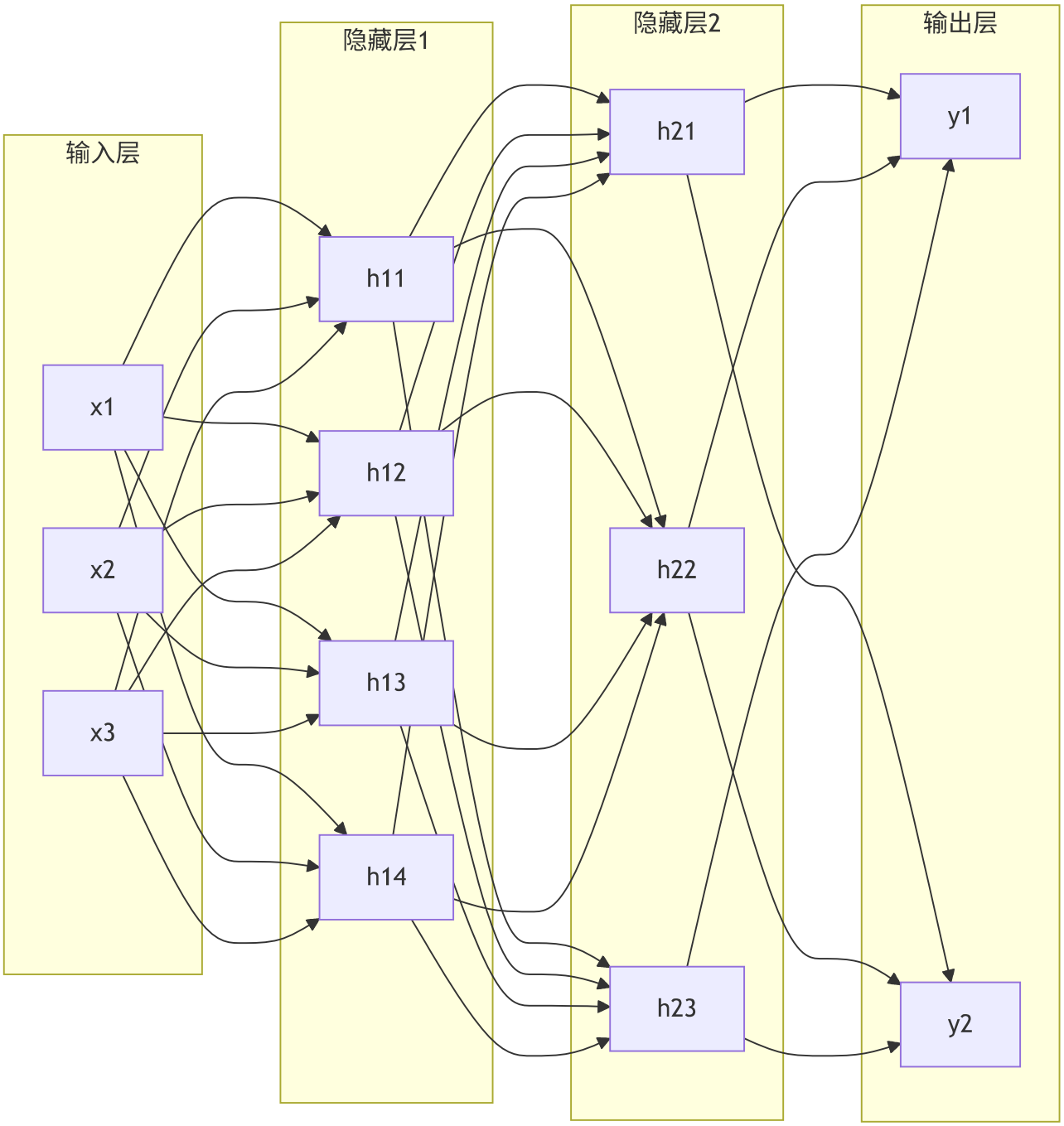
数学表达式：

$$y = \sigma(\sum_{i=1}^{n} w_i x_i + b) = \sigma(w^T x + b)$$

其中：

- x_i：输入特征
- w_i：权重参数
- b：偏置项
- σ：激活函数
- y：输出

1.1.2 多层神经网络



前向传播过程：

```python
import numpy as np

class NeuralNetwork:
    def __init__(self, layer_sizes):
        """
        初始化神经网络
        layer_sizes: 各层神经元数量，如 [3, 4, 3, 2]
        """
        self.num_layers = len(layer_sizes)
        self.layer_sizes = layer_sizes

        # 初始化权重和偏置
```

```
        self.weights = [np.random.randn(layer_sizes[i], layer_sizes[i-1])
                        for i in range(1, self.num_layers)]
        self.biases = [np.random.randn(layer_sizes[i], 1)
                       for i in range(1, self.num_layers)]

    def forward(self, x):
        """前向传播"""
        activation = x
        activations = [x]  # 存储每层的激活值
        zs = []  # 存储每层的加权输入

        for w, b in zip(self.weights, self.biases):
            z = np.dot(w, activation) + b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)

        return activation, activations, zs

def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))
```

1.2 反向传播算法

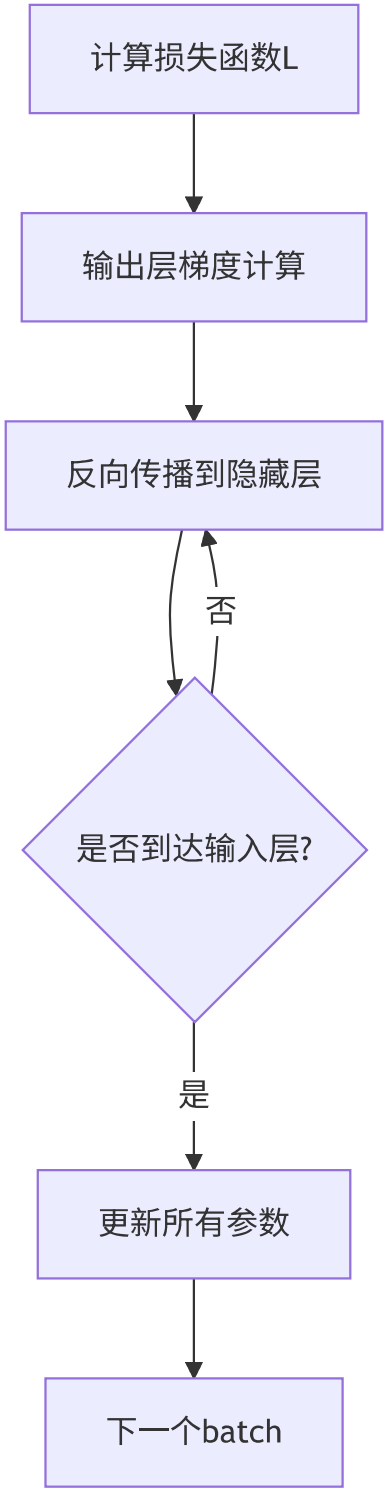
反向传播（Backpropagation）是训练神经网络的核心算法，用于计算损失函数对各参数的梯度。

1.2.1 链式法则

反向传播基于微积分的链式法则：

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial w_i}$$

1.2.2 反向传播流程图



1.2.3 反向传播实现

```
def backward(self, x, y):
    """
    反向传播算法
    x: 输入数据
    y: 真实标签
    """
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    nabla_b = [np.zeros(b.shape) for b in self.biases]
```

```
        # 前向传播
        activation = x
        activations = [x]
        zs = []

        for w, b in zip(self.weights, self.biases):
            z = np.dot(w, activation) + b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)

        # 反向传播
        # 输出层误差
        delta = self.cost_derivative(activations[-1], y) * sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())

        # 从倒数第二层开始反向传播
        for l in range(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())

        return nabla_w, nabla_b

    def sigmoid_prime(z):
        """sigmoid函数的导数"""
        return sigmoid(z) * (1 - sigmoid(z))

    def cost_derivative(self, output_activations, y):
        """损失函数对输出层激活值的偏导数"""
        return output_activations - y
```

1.3 激活函数

激活函数为神经网络引入非线性，是深度学习的关键组件。

1.3.1 常见激活函数对比

激活函数	公式	优点	缺点	应用场景
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	输出在(0,1)，可解释为概率	梯度消失、计算expensive	二分类输出层

激活函数	公式	优点	缺点	应用场景
Tanh	$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	输出在(-1,1)，零中心	梯度消失	RNN隐藏层
ReLU	$ReLU(x) = max(0, x)$	计算简单、缓解梯度消失	死亡ReLU问题	CNN、MLP隐藏层
LeakyReLU	$LeakyReLU(x) = max(0.01x, x)$	解决死亡ReLU	需要调整负半轴斜率	深层网络
GELU	$GELU(x) = x\Phi(x)$	平滑、性能好	计算稍复杂	**Transformer标配**
Swish	$Swish(x) = x \cdot \sigma(x)$	自适应、平滑	计算量大	深度网络

1.3.2 激活函数图形对比

ReLU及其变体：

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-3, 3, 1000)

# ReLU
relu = np.maximum(0, x)

# LeakyReLU
leaky_relu = np.where(x > 0, x, 0.01 * x)

# GELU (近似)
def gelu(x):
    return 0.5 * x * (1 + np.tanh(np.sqrt(2/np.pi) * (x + 0.044715 * x**3)))

# Swish
def swish(x):
    return x / (1 + np.exp(-x))
```

GELU在大模型中的优势：

GELU（Gaussian Error Linear Unit）是现代大模型（如GPT、BERT）的标准激活函数：

$$GELU(x) = x \cdot \Phi(x) = x \cdot \frac{1}{2}[1 + erf(\frac{x}{\sqrt{2}})]$$

其中 $\Phi(x)$ 是标准正态分布的累积分布函数。

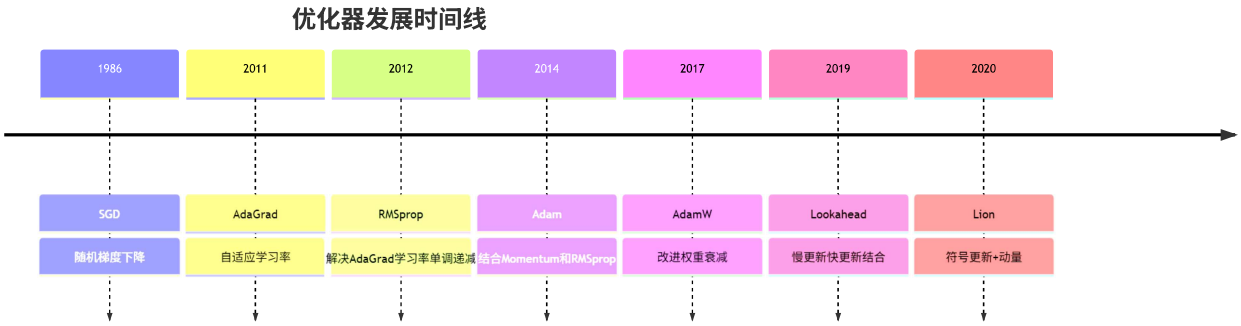
为什么大模型使用GELU?

1. **平滑性好**：导数连续，训练更稳定
2. **非单调性**：在负半轴有轻微激活
3. **实验效果**：在NLP任务上表现优于ReLU
4. **概率解释**：可理解为随机正则化

1.4 优化器

优化器决定了如何根据梯度更新参数，直接影响训练效果和收敛速度。

1.4.1 优化器进化史



1.4.2 核心优化器详解

SGD（Stochastic Gradient Descent）

```
# 普通SGD
w = w - learning_rate * gradient

# SGD with Momentum
velocity = momentum * velocity - learning_rate * gradient
w = w + velocity
```

特点：

- ✅ 简单高效，内存占用小
- ✅ 在某些大模型训练中仍是首选
- ❌ 收敛速度慢，容易震荡

Adam（Adaptive Moment Estimation）

```
class Adam:
    def __init__(self, learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8):
        self.lr = learning_rate
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
```

```
        self.m = None  # 一阶矩估计（动量）
        self.v = None  # 二阶矩估计（梯度平方）
        self.t = 0     # 时间步

    def update(self, params, grads):
        if self.m is None:
            self.m = np.zeros_like(params)
            self.v = np.zeros_like(params)

        self.t += 1

        # 更新一阶矩和二阶矩
        self.m = self.beta1 * self.m + (1 - self.beta1) * grads
        self.v = self.beta2 * self.v + (1 - self.beta2) * (grads ** 2)

        # 偏差校正
        m_hat = self.m / (1 - self.beta1 ** self.t)
        v_hat = self.v / (1 - self.beta2 ** self.t)

        # 参数更新
        params -= self.lr * m_hat / (np.sqrt(v_hat) + self.epsilon)

        return params
```

Adam vs AdamW：

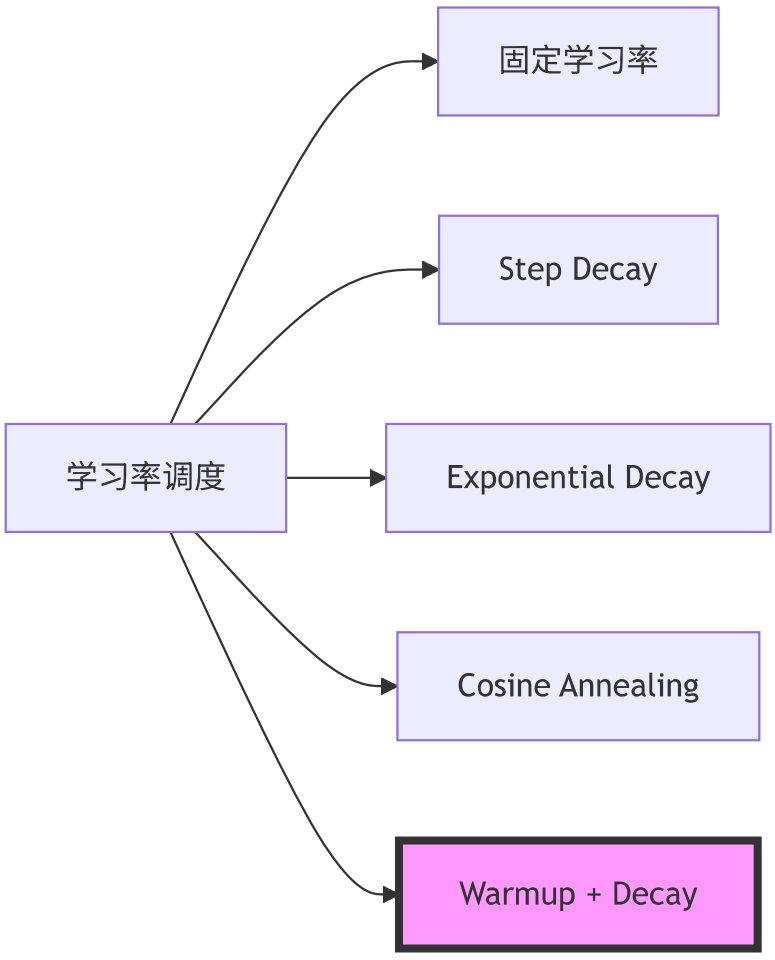
特性	Adam	AdamW
权重衰减	在梯度中添加L2正则项	解耦权重衰减，直接在参数上操作
公式	$g_t = g_t + \lambda w_t$	$w_t = w_t - \alpha\lambda w_t$
效果	权重衰减与自适应学习率相互影响	权重衰减独立，效果更好
大模型使用	较少	**主流选择**

AdamW实现

```
class AdamW:
    def __init__(self, learning_rate=0.001, beta1=0.9, beta2=0.999,
                 epsilon=1e-8, weight_decay=0.01):
        self.lr = learning_rate
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.weight_decay = weight_decay
        self.m = None
        self.v = None
```

```
        self.t = 0

    def update(self, params, grads):
        if self.m is None:
            self.m = np.zeros_like(params)
            self.v = np.zeros_like(params)

        self.t += 1

        # Adam更新
        self.m = self.beta1 * self.m + (1 - self.beta1) * grads
        self.v = self.beta2 * self.v + (1 - self.beta2) * (grads ** 2)

        m_hat = self.m / (1 - self.beta1 ** self.t)
        v_hat = self.v / (1 - self.beta2 ** self.t)

        # AdamW: 解耦的权重衰减
        params = params * (1 - self.lr * self.weight_decay)
        params -= self.lr * m_hat / (np.sqrt(v_hat) + self.epsilon)

        return params
```

1.4.3 学习率调度策略



Warmup + Cosine Decay（大模型标配）：

```
def get_cosine_schedule_with_warmup(optimizer, num_warmup_steps, num_training_steps
    def lr_lambda(current_step):
        # Warmup阶段：线性增长
        if current_step < num_warmup_steps:
            return float(current_step) / float(max(1, num_warmup_steps))

        # Cosine Decay阶段
        progress = float(current_step - num_warmup_steps) / \
                   float(max(1, num_training_steps - num_warmup_steps))
        return max(0.0, 0.5 * (1.0 + math.cos(math.pi * progress)))

    return LambdaLR(optimizer, lr_lambda)
```

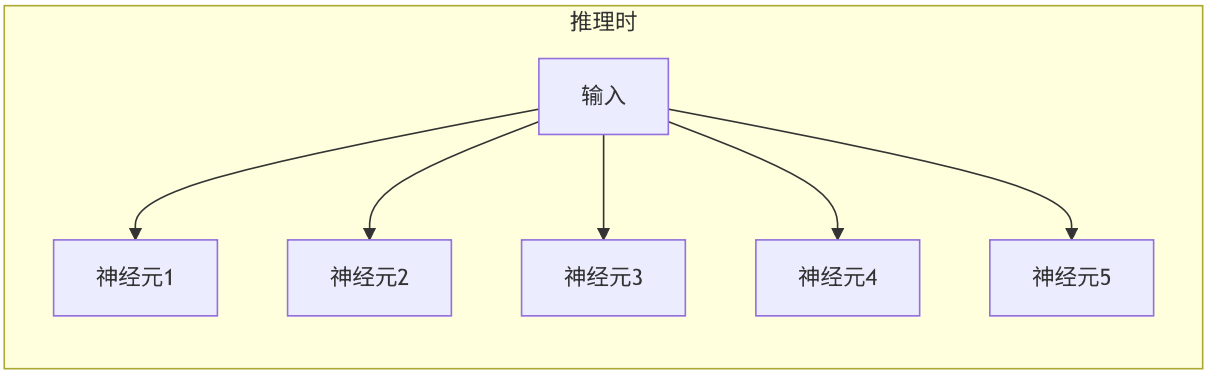
为什么需要Warmup?

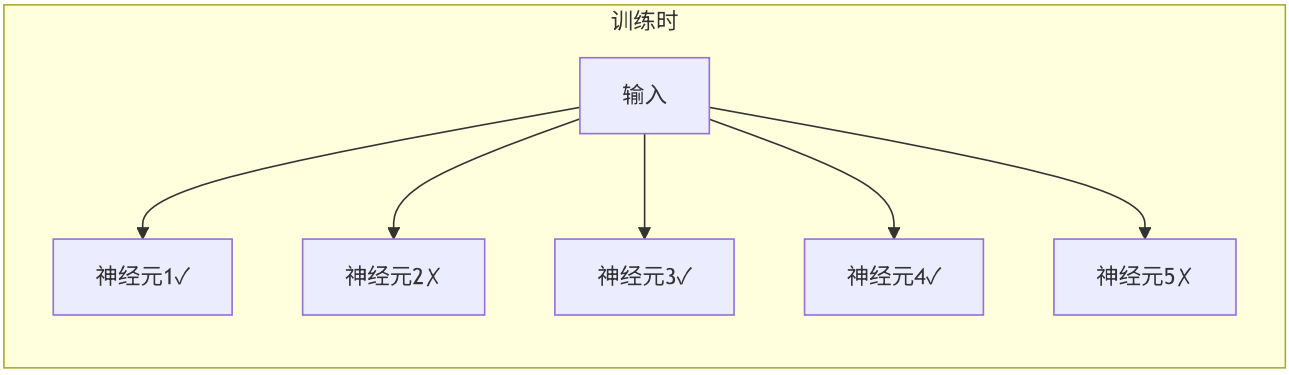
1. 初始阶段参数随机，梯度不稳定
2. 大学习率可能导致梯度爆炸
3. Warmup让模型平稳启动

1.5 正则化技术

正则化用于防止过拟合，提高模型泛化能力。

1.5.1 Dropout





Dropout实现：

```
class Dropout:
    def __init__(self, dropout_rate=0.5):
        self.dropout_rate = dropout_rate
        self.mask = None

    def forward(self, x, train=True):
        if train:
            # 训练时：随机丢弃
            self.mask = np.random.rand(*x.shape) > self.dropout_rate
            return x * self.mask / (1 - self.dropout_rate)  # Inverted dropout
        else:
            # 测试时：使用全部神经元
            return x

    def backward(self, dout):
        return dout * self.mask / (1 - self.dropout_rate)
```

1.5.2 Layer Normalization

Layer Norm是Transformer的核心组件，不同于Batch Norm。

对比表：

特性	Batch Norm	Layer Norm
归一化维度	跨样本，同一特征	单个样本，所有特征
依赖batch	是	否
适用场景	CNN、固定batch大小	**RNN、Transformer**
训练测试差异	大（需要统计量）	小
序列任务	不适合	**适合**

Layer Norm实现：

```
class LayerNorm:
    def __init__(self, normalized_shape, eps=1e-5):
        self.eps = eps
        self.gamma = np.ones(normalized_shape)  # 可学习缩放参数
        self.beta = np.zeros(normalized_shape)  # 可学习偏移参数
```

```
    def forward(self, x):
        # x shape: (batch_size, seq_len, hidden_size)
        # 在最后一个维度（特征维度）归一化
        mean = np.mean(x, axis=-1, keepdims=True)
        var = np.var(x, axis=-1, keepdims=True)

        x_norm = (x - mean) / np.sqrt(var + self.eps)
        out = self.gamma * x_norm + self.beta

        return out
```

为什么Transformer使用Layer Norm?

1. **序列长度可变**：不同样本序列长度不同，Batch Norm难以处理
2. **batch大小敏感**：大模型训练时batch可能很小
3. **训练稳定**：每个样本独立归一化，更稳定

1.6 常见损失函数

1.6.1 分类任务损失函数

交叉熵损失（Cross Entropy Loss）：

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

语言模型中的应用：

```
def cross_entropy_loss(logits, labels):
    """
    logits: (batch_size, seq_len, vocab_size)
    labels: (batch_size, seq_len)
    """
    # 计算log_softmax
    log_probs = F.log_softmax(logits, dim=-1)

    # 选择正确类别的log概率
    loss = -torch.gather(log_probs, dim=-1, index=labels.unsqueeze(-1))

    # 平均损失
    return loss.mean()
```

1.6.2 大模型常用损失

Causal Language Modeling Loss（因果语言模型损失）：

```python
class CausalLMLoss:
    def __init__(self):
        self.loss_fn = nn.CrossEntropyLoss(ignore_index=-100)

    def __call__(self, logits, labels):
        """
        logits: (batch_size, seq_len, vocab_size)
        labels: (batch_size, seq_len)

        计算下一个token的预测损失
        """
        # Shift：预测下一个token
        shift_logits = logits[..., :-1, :].contiguous()
        shift_labels = labels[..., 1:].contiguous()

        # Flatten
        shift_logits = shift_logits.view(-1, shift_logits.size(-1))
        shift_labels = shift_labels.view(-1)

        # 计算损失
        loss = self.loss_fn(shift_logits, shift_labels)

        return loss
```

1.7 面试高频问题

Q1: 为什么需要激活函数?

答案要点:

1. **引入非线性**：没有激活函数，多层神经网络等价于单层线性模型
2. **数学证明**：多层线性变换的复合仍是线性变换
3. **表达能力**：非线性激活使网络能拟合复杂函数

Q2: 为什么大模型使用Layer Norm而不是Batch Norm?

答案要点:

1. **序列长度可变**：不同样本序列长度不同
2. **batch大小敏感性**：训练时batch可能很小
3. **独立性**：每个样本独立归一化，训练测试一致

Q3: Adam和SGD如何选择?

答案要点:

场景	推荐优化器	原因
小模型、快速实验	Adam	收敛快，自适应学习率
大模型预训练	AdamW	更好的权重衰减，稳定
追求最优性能	SGD + Momentum	最终性能可能更好（需精调超参）
资源受限	SGD	内存占用小

Q4: 手写Softmax函数及其导数

```python
def softmax(x):
    """
    数值稳定的softmax实现
    x: (batch_size, num_classes)
    """
    # 减去最大值，防止指数溢出
    x_max = np.max(x, axis=-1, keepdims=True)
    exp_x = np.exp(x - x_max)
    return exp_x / np.sum(exp_x, axis=-1, keepdims=True)

def softmax_gradient(softmax_output):
    """
    softmax梯度：∂softmax/∂x = softmax * (I - softmax^T)
    """
    s = softmax_output.reshape(-1, 1)
    return np.diagflat(s) - np.dot(s, s.T)
```

Q5: 梯度消失和梯度爆炸的解决方案

梯度消失：

- ✅ 使用ReLU/GELU等激活函数
- ✅ 使用ResNet的残差连接
- ✅ Layer Normalization
- ✅ 合理的参数初始化（Xavier、He初始化）

梯度爆炸：

- ✅ 梯度裁剪（Gradient Clipping）
- ✅ 合理的学习率
- ✅ Batch/Layer Normalization

```python
# 梯度裁剪实现
def clip_gradient(gradients, max_norm=1.0):
    total_norm = np.sqrt(sum(np.sum(g**2) for g in gradients))
    clip_coef = max_norm / (total_norm + 1e-6)
```

```
    if clip_coef < 1:
        for g in gradients:
            g *= clip_coef
    return gradients
```

1.8 本章小结

本章介绍了深度学习的基础知识，这些是理解大模型的必备基础：

✅ **神经网络基本原理**：前向传播和反向传播 ✅ **激活函数**：GELU是大模型标配 ✅ **优化器**：AdamW是主流选择 ✅ **正则化**：Layer Norm在Transformer中的关键作用 ✅ **损失函数**：语言模型的交叉熵损失

进阶学习建议：

1. 手写实现一个简单的神经网络（不使用框架）
2. 对比不同激活函数和优化器的效果
3. 理解Layer Norm的数学原理和代码实现

下一章预告： 第2章将深入讲解Transformer架构，这是现代大模型的基石。