实验手册—深度学习现象导论：从感知机到大模型

作者（笔画数大小排序）:

许志钦　　张耀宇

参与学生:

王志伟　白志威　张众望

杭良慨　周章辰　赵佳杰　姚俊杰

（笔画数大小排序）

联系方式:

Github: `https://github.com/xuzhiqin1990/understanding_dl`

Email: xuzhiqin@sjtu.edu.cn, zhyy.sjtu@sjtu.edu.cn

2025 年 10 月 12 日

目录

Chapter 1

实验手册：频率原则的验证

Jupyter 代码可以在 GitHub 找到[1]。

1.1 神经网络基本概念

神经网络是一个有许多参数的机器学习方法。假设我们有一组训练集 $S = \{(x_i, y_i)\}_{i=1}^n, (x_i, y_i) \in \mathbb{R}^2$。考虑一个只有一层隐藏层的神经网络，即两层神经网络，

$$f_\theta(x) = \sum_{j=1}^m a_j \sigma(w_j x + b_j). \tag{1.1}$$

为了方便起见，定义

$$h_i = f_\theta(x_i). \tag{1.2}$$

常见的激活函数 $\sigma(x)$ 为 $\mathrm{ReLU}(x) = \max\{0, x\}$、$\tanh(x)$ 等。我们这里假设 x, w, b, a 都是一维的标量，对于高维的情形，后面会有介绍。我们可以通过（随机）梯度下降来调节参数。当梯度下降用在神经网络模型中时，它有一种特殊的名字：向后传播法。考虑训练的损失函数为均方差:

$$L_S = \frac{1}{2n} \sum_{i=1}^n (y_i - h_i)^2. \tag{1.3}$$

对于参数集 $\boldsymbol{\theta} = (a_1, w_1, b_1, a_2, w_2, \cdots, b_m)$, 我们通过梯度下降法进行训练：

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \frac{\partial L_S}{\partial \boldsymbol{\theta}_t}. \tag{1.4}$$

[1] https://github.com/xuzhiqin1990/understanding_dl/tree/main/code/frequency_principle

因为链式法则，梯度下降的计算顺序是 $h_i \to a_j \to \sigma \to x_i$，而信息流 (information flow) 的顺序是 $x_i \to \sigma \to a_j \to h_i$，这两者的顺序相反，因此称其为向后传播。优化后，通过取一些新的采样点计算测试误差来判断拟合的好坏。

下面用矩阵的形式写 $f_\theta(\boldsymbol{x})$:

$$f_\theta(\boldsymbol{x}) = \boldsymbol{W}^{[2]}\sigma \circ (\boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}), \tag{1.5}$$

其中 $\boldsymbol{x} \in \mathbb{R}^{d\times 1}, \boldsymbol{W}^{[1]} \in \mathbb{R}^{m\times d}, \boldsymbol{b}^{[1]} \in \mathbb{R}^{m\times 1}, \boldsymbol{W}^{[2]} \in \mathbb{R}^{d_o\times m}$，“$\circ$” 的意思是对应元素的运算 (entry-wise operation)（例如对应元素相乘）。这里 d 是输入数据的维度，m 是隐藏层神经元的个数，d_o 是输出维度。有时我们会同时计算 n 个样本的值，比如在实际编程计算中，把 $\boldsymbol{x}$ 写成矩阵形式:

$$\boldsymbol{Y} = h(\boldsymbol{X}) = \boldsymbol{W}^{[2]}\sigma \circ (\boldsymbol{W}^{[1]}\boldsymbol{X} + \boldsymbol{B}^{[1]}), \tag{1.6}$$

其中 $\boldsymbol{X} \in \mathbb{R}^{d\times n}, \boldsymbol{Y} \in \mathbb{R}^{[d_o\times n]}, \boldsymbol{B}^{[1]} \in \mathbb{R}^{m\times n}$。$\boldsymbol{B}^{[1]}$ 的每一列都为 $b^{[1]}$, 即 $\boldsymbol{B}^{[1]} = [b^{[1]}, b^{[1]}, \cdots, b^{[1]}]$。用这些符号可以类似得定义一个深度神经网络。

一个 L 层神经网络记为

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{W}^{[L-1]}\sigma \circ (\boldsymbol{W}^{L-2}\sigma \circ (\cdots (\boldsymbol{W}^{[1]}\sigma \circ (\boldsymbol{W}^{[0]}\boldsymbol{x} + \boldsymbol{b}^{[0]}) + \boldsymbol{b}^{[1]}) \cdots) + \boldsymbol{b}^{[L-2]}) + \boldsymbol{b}^{[L-1]}, \tag{1.7}$$

其中 $\boldsymbol{W}^{[l]} \in \mathbb{R}^{m_{l+1}\times m_l}, \boldsymbol{b}^{[l]} \in \mathbb{R}^{m_{l+1}}, m_0 = d_{in} = d, m_L = d_o$，$\sigma$ 是一个向量函数。注意一下在计算网络层数时，输入层不计入（即层数为隐藏层 + 输出层的层数）我们把参数集记为

$$\boldsymbol{\theta} = (\boldsymbol{W}^{[0]}, \boldsymbol{W}^{[1]}, \cdots, \boldsymbol{W}^{[L-1]}, \boldsymbol{b}^{[0]}, \boldsymbol{b}^{[1]}, \cdots, \boldsymbol{b}^{[L-1]}),$$

把 $\boldsymbol{W}^{[l]}$ 中的元素记为 $\boldsymbol{W}^{[l]}_{ij}$。也可以用递归的方法定义这个网络:

$$f^{[0]}_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{x} \tag{1.8}$$

$$f^{[l]}_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sigma \circ (\boldsymbol{W}^{[l-1]} f^{[l-1]}_{\boldsymbol{\theta}}(\boldsymbol{x}) + \boldsymbol{b}^{[l-1]}), \quad 1 \le l \le L-1 \tag{1.9}$$

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = f^{[L]}_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{W}^{[L-1]} f^{[L-1]}_{\boldsymbol{\theta}}(\boldsymbol{x}) + \boldsymbol{b}^{[L-1]} \tag{1.10}$$

1.2 实验背景

我们在理论部分已经介绍了深度学习的一些基本知识, 也指出了理解神经网络这样一个复杂系统的挑战性。因此, 找到一个合适的切入点去分析神经网络, 对于剖析这个” 黑盒子” 的内部机制尤为重要。在科学研究中, 人们对复杂系统的探索往往起源于对基本现象的观察和分析。例如, 在遗传学研究中, 孟德尔选择了相对简单的豌豆作为研究对象, 通过观察豌豆性状的分离和组合规律, 总结出了遗传的基本定律。这些定律后来被应用于解释更加复杂的生物遗传现象。

再如, 在电磁学研究中, 安培和法拉第等科学家首先研究了相对简单的电场和磁场现象, 发现了电磁感应定律等基本规律。在此基础上, 麦克斯韦进一步将这些规律统一为麦克斯韦方程，解释了更复杂的电磁现象, 预言了电磁波的存在。

受此启发, 我们在研究神经网络时, 也可以先从一些非常简单的问题入手, 通过在简单模型中观察、分析、总结规律, 再尝试将这些认识推广到更加复杂的神经网络中。当然, 这种推广需要谨慎, 并非所有从简单模型中得到的结论都能直接应用于复杂模型。但这种研究方法可以帮助我们抓住问题的关键, 为进一步探索复杂网络提供有益的思路和启发。基于上述考虑, 在神经网络的实验探究过程中, 我们选择从最简单的拟合一维函数的问题开始进行研究。一方面, 一维问题结构简单、直观, 便于观察分析网络的特性; 另一方面, 很多神经网络的基本特性, 如学习过程的非线性、在常规设定下不易过拟合等, 在一维问题中就已经能够得到充分的体现。通过对一维问题的研究, 我们可以初步认识神经网络的一些重要特点, 为进一步探索更加复杂的网络打下基础。因此, 在本节中, 我们将基于一维问题展开讨论, 通过具体的实例来向读者展示神经网络在学习过程中的一般规律。

1.3 实验目标

我们要验证神经网络训练中频率原则的适用性，即神经网络在训练时先学习低频成分，随后逐步学习高频成分。通过对网络输出和目标函数的频率成分进行分析，我们希望能够清晰地观察到这一现象。

在进行实验设计时，我们需要控制一些关键变量，确保实验结果能够反映频率原则的表现。此外，我们将通过**相对误差**来定量评估神经网络对不同频率成分的学习速度。

1.4 理论基础

1.4.1 离散傅里叶变换（DFT）

离散傅里叶变换（Discrete Fourier Transform, DFT）是傅里叶分析在离散数据上的一种实现方法。它通过将有限的离散时间信号分解为频率成分，从而揭示信号在频域中的特性。对于一个给定的离散信号 $\{x_n\}$，其长度为 N，其 DFT 定义为:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N}, \quad k = 0, 1, \ldots, N-1.$$

其中，X_k 表示信号在第 k 个频率上的幅值。通过 DFT，我们可以得到信号的频谱，从而分析信号中的不同频率成分。

在神经网络的学习过程中，通过 DFT 可以观察不同频率成分的变化情况，进一步分析频率原则的实现过程。然而，在进行频谱分析时，需要注意采样率的选择。如果采样率不足，会产生假频现象，即在频谱中出现不存在的频率成分。因此，在实际操作中，我们需确保足够的采样点来避免假频的出现。

下面的代码展示了一个利用 DFT 分析信号的示例，通过改变采样点数来观察频率分布。

代码片段：

```
from utils import dft_analysis

"""
func: dft_analysis(f, T, N)

    Perform DFT analysis on a given function f with a total time of
        T and N samples.

    Args:
        f (function): The function to be analyzed.
        T (float): The total time of the signal.
        N (int): The number of samples.

    Returns:
        None
"""


def f(t):
    return np.sin(2*np.pi*t) + np.sin(6*np.pi*t)


print('sample size=50: ')
dft_analysis(f=f, T=2, N=50)

print('sample size=8: ')
dft_analysis(f=f, T=2, N=8)
```

1.4.2 频率原则

频率原则指出：在训练过程中，神经网络倾向于先拟合数据的低频成分，而高频成分则需要较长的时间才能被有效学习。这一现象可以通过傅里叶变换来理解。傅里叶变换能够将时间或空间域中的函数分解为一系列正弦或余弦波，这些波按照频率由低到高排列。通过傅里叶分析，我们能够检测到神经网络如何逐步学习这些不同频率的成分。

1.5 实验步骤

1.5.1 数据生成与预处理

目标函数为三项正弦波的叠加，表示为：

$$f(x) = \sin(x) + \sin(3x) + \sin(5x).$$

该函数生成了同时包含低、中、高频成分的信号，便于验证神经网络的频率学习过程。

代码片段：

```
def get_y(x):

    """
    Function to fit.

    Args:
        x (float): input value.

    Returns:
        float: output value.
    """
    alpha=2
    y = np.sin(x)+np.sin(3*x)+np.sin(5*x)
    return y



args.training_input, args.training_target, args.test_input, args.
   test_target = get_dataset(args, get_y)
```

```
print("The training data size: ", args.training_input.shape)


print("The target function:")
plot_target(args)
```

变量控制：

- **数据点数**：我们需要保证训练数据足够密集，特别是在高频区域，确保高频部分被合理采样。

1.5.2 构建模型

模型为多层感知机（MLP），使用自定义的网络结构和激活函数。每层的神经元数量及激活函数可以通过命令行参数控制。

代码片段：

```
act_func = get_act_func(args.act_func_name)

#Initialize the neural network model
model = Model(args.t, args.hidden_layers_width, args.input_dim,
               args.output_dim, act_func).to(args.device)
```

1.5.3 模型训练及测试过程

我们对初始化完成的模型通过梯度下降法进行训练。我们使用均方差损失作为损失函数，并使用 Adam 优化器。

代码片段：

```
# Define the optimizer: determine the gradient descent optimization
    algorithm, the default is Adam.
if args.optimizer=='sgd':
    optimizer = torch.optim.SGD(model.parameters(), lr=args.lr)
else:
    optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
```

```
# Define the loss function: the loss function is the mean square
    error.
loss_fn = nn.MSELoss(reduction='mean')
```

对于每个 epoch，我们均进行一步训练及一步测试，并保存模型损失值、模型输出等。

代码片段：

```
for epoch in range(args.epochs+1):
    # Set the model to training mode
    model.train()

    # Train one step
    loss, training_output = train_one_step(
    model, optimizer, loss_fn, args)

    # Test the trained model
    loss_test, output = test(
        model, loss_fn, args)

    # Record the loss and output
    args.loss_training_lst.append(loss)
    args.loss_test_lst.append(loss_test)
    args.training_output.append(training_output.detach().cpu().
       numpy())

    # Print the loss and time
    if epoch % args.plot_epoch == 0:
        print("[%d] loss: %.6f valloss: %.6f time: %.2f s" %
                (epoch + 1, loss, loss_test, (time.time()-t0)))
```

1.5.4 相对误差的计算

为了定量评估神经网络对不同频率成分的学习速度，我们使用**相对误差**来观察频率原则。相对误差衡量了模型输出的频率分量和目标频率分量之间的差异。通过相对误差，我们可以清

楚地看到不同频率成分的拟合进度。为避免 DFT 产生的误差影响，我们选取目标函数频域空间的振幅峰值对应的频率进行观测。

相对误差计算公式

$$\Delta F(k) = \frac{|h_k - f_k|}{|f_k|},$$

其中 h_k 表示模型输出在频率 k 处的幅值，f_k 表示目标函数在频率 k 处的幅值。

代码片段：

```
# create the absolute error array
abs_err = np.zeros([len(idx1), len(args.training_output)])

# calculate the absolute error
tmp1 = y_fft[idx1]
for i in range(len(y_pred_epoch)):
    tmp2 = my_fft(y_pred_epoch[i])[idx1]
    abs_err[:, i] = np.abs(tmp1 - tmp2)/(1e-5 + tmp1)
```

为什么使用相对误差：

- **频率成分差异评估**：相对误差提供了模型拟合的准确性，尤其是在高频成分较难学习时，误差的变化能够直观地展示不同频率成分的收敛速度。
- **避免绝对值误差影响**：高频成分的幅值往往较低，直接使用绝对误差可能会使误差看起来很小。使用相对误差可以平衡不同频率成分的幅值，确保误差计算的公平性。

1.5.5 实验结果分析与可视化

训练过程结束后，我们可以基于上述计算所得的相对误差，绘制相对误差图。通过对比不同频率成分的误差变化曲线，可以验证网络是否遵循频率原则。

代码片段：

```
def plot_abs_err(args, abs_err):

    """
    Plot the heatmap of the relative error for different
        frequencies.
```

```
    Args:
    args: A dictionary containing save path.
    abs_err: The absolute error array.

    Returns:
    None
    """

    # initialize the figure
    plt.figure(figsize=(8, 6))
    ax = plt.gca()

    # plot the heatmap of the relative error
    plt.pcolor(abs_err, cmap='RdBu', vmin=0.1, vmax=1, linewidths
        =0.4)

    # set the colorbar
    plt.colorbar()

    # set the x-axis labels and its fontsize
    plt.xlabel('Epoch', fontsize=22)

    # Set the y-axis ticks and labels to 1, 2, 3
    plt.yticks([0.5, 1.5, 2.5], [1, 2, 3], fontsize=22)

    # Set the y-axis tick parameters to hide the tick marks and set
        the tick label size
    plt.gca().yaxis.set_tick_params(size=0)
    plt.gca().tick_params(axis='y', labelsize=22)
    plt.gca().tick_params(axis='x', labelsize=22)
    plt.title('Absolute Error', fontsize=22)
    plt.tight_layout()
```

```
    # save the figure
    plt.savefig(os.path.join(args.path, 'hot.png'))
    plt.show()
    plt.close()
```

最后实验结果如图1.1所示。左图表示拟合结果。中间图像为目标函数与模型输出在频域空间的变换结果，可以发现目标函数有三个主要的频率成分且对应的幅值都相等。红点表示目标函数频域分布的峰值，也是我们刻画在不同频率模型学习快慢的目标频率。右图热力图表示三个主要频率的神经网络拟合结果随训练进行的相对误差，横坐标表示训练 epoch，左坐标轴 index 数字越小代表频率越低。红色代表神经网络拟合结果的相对误差很小，蓝色代表相对误差较大。在训练的过程中频率最低的成分收敛地最快，几个 epoch 就已经拟合地很好，而高频成分却用了 2000 个 epoch 才达到了较小的相对误差。可以由热图看出，模型先拟合低频成分，再拟合高频。

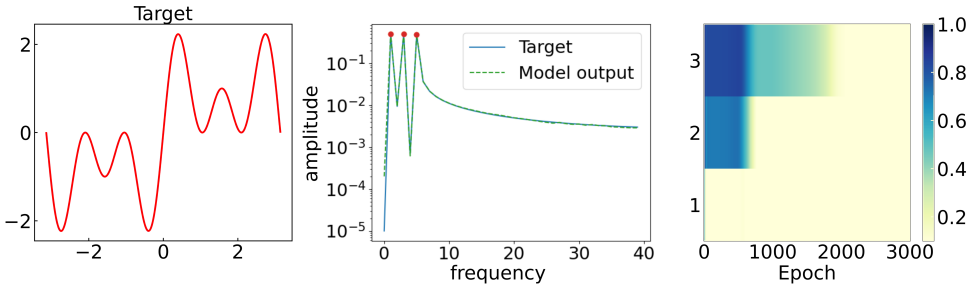


图 1.1: 控制目标函数幅值相同的一维数据实验

1.6 任务

神经网络的拟合过程仍然受到许多因素的影响，如：参数初始化分布、学习率、目标函数、激活函数等。这些因素的变化是否会影响频率原则？可以通过控制变量的方式，研究某些参数对频率原则的影响。

接下来，我们提供了两种实验方案，用于研究目标函数变化对频率原则的影响。这两种方案分别通过改变目标函数中高频部分的频率和幅度，探索其对频率原则的影响。（下述两个任务中的 10% 是可修改的。）

示例 1：研究不同频率幅度如何影响模型的收敛速度。以目标函数 $sin(x) + a * sin(4x)$ 为例，其中 a 为给定的参数。我们可以将横坐标设为 a，纵坐标设为在该目标函数下，满足高频

相对误差 <10% 所需的步数与满足低频相对误差 <10% 所需步数的比值。

示例 2：研究不同高频如何影响模型的收敛速度。以目标函数 $sin(x) + 0.2 * sin(k \cdot x)$ 为例，其中 k 为给定的参数。我们可以将横坐标设为 k，纵坐标设为在该目标函数下，满足高频相对误差 <10% 所需的步数与满足低频相对误差 <10% 所需步数的比值。

Chapter 2

实验手册：参数凝聚现象的验证

Jupyter 代码可以在 GitHub 找到[1]。

2.1 实验简介

在神经网络的训练过程中，不同的初始化对神经网络的训练动力学有非常大的影响。在本实验中，我们将通过一个简单的一维实验观察不同初始化下，神经网络参数的动力学演化情况。其中在小初始化时神经网路会表现出参数凝聚的现象，以下是参数凝聚现象的一个示意图：

这是一个理想的凝聚的示意图。在初始化时，各个神经元的输入权重差异很大，用不同的颜色表示。但是在经过一段时间训练后，中间隐藏神经元分成了两类，前两个神经元是一类，后三个神经元是另一类。在每一类中，不同神经元的输入权重是完全一样的 (颜色相同)，因此，它们的输出也是一样的。

2.2 实验目标

希望通过本实验，可以直观感受不同的初始化下神经网络在参数层面的演化过程，以及其对神经网络输出结果的不同表现；同时也可以比较不同的神经网络激活函数对神经网络初始凝聚的影响。

[1] https://github.com/xuzhiqin1990/understanding_dl/tree/main/code/condense_exp

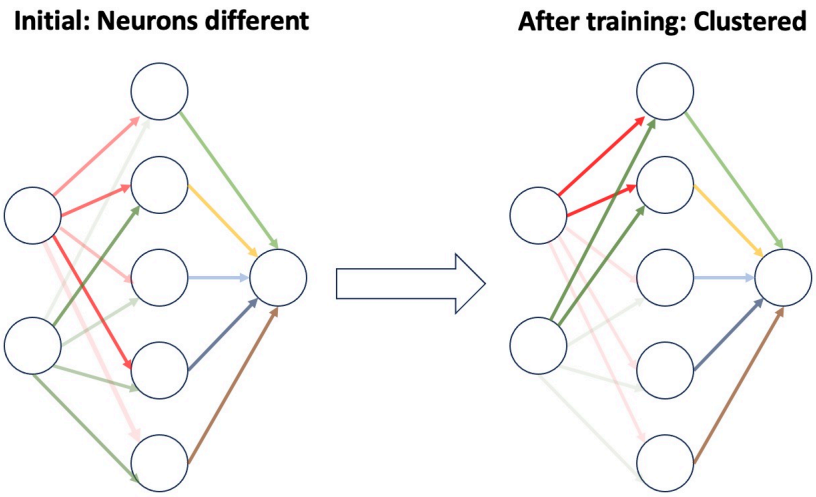


图 2.1: 理想的凝聚现象。

2.3 理论基础

- **线性区域：**当初始化较大时，神经网络在初始化附近就能拟合好数据，参数的相对变化范围非常小，因此模型在初始化附近可以用一阶泰勒展开近似:

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) \approx f_{\boldsymbol{\theta}}^{\text{lin}}(\boldsymbol{x}) = f_{\boldsymbol{\theta}(0)}(\boldsymbol{x}) + \nabla_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}(0)}(\boldsymbol{x}) \cdot (\boldsymbol{\theta}(t) - \boldsymbol{\theta}(0)). \tag{2.1}$$

- **凝聚区域：**当初始化较小时，神经网络在初始化附近不能拟合好数据，网络参数的相对变化范围较大，因此网络不能用一个线性模型近似，表现出高度的非线性行为。
- **临界区域：**介于线性和凝聚区域之间，网络表现出中等程度的非线性行为。

2.4 实验原理

2.4.1 神经网络结构

本实验使用一个简单的前馈神经网络，包含一个输入层、一个隐藏层和一个输出层。网络结构如下:

$$f_{\boldsymbol{\theta}}(x) = \sum_{k=1}^{m} a_k \sigma(\boldsymbol{w}_k^T x + b_k),$$

其中，

- 输入维度 d_{in}
- 隐藏层宽度 m 可调。
- 输出维度 d_{out}

2.4.2 参数初始化

参数初始化对神经网络的训练至关重要。本实验使用以下初始化方法：

- 权重初始化: $\boldsymbol{w}_k \sim \mathcal{N}(0, \varepsilon^2 I_d)$
- 偏置初始化: $b_k \sim \mathcal{N}(0, \varepsilon^2)$
- 输出层初始化: $a_k \sim \mathcal{N}(0, \varepsilon^2)$

其中，$\varepsilon = \frac{1}{m^\gamma}$，$\gamma$ 是一个可调参数，用于控制初始化的方差，从而控制初始化的大小。
下面是代码中用于初始化网络和控制网络初始化的部分：

代码片段：

```
class Linear(nn.Module):
    def __init__(self, t, hidden_layers_width=[100],  input_size
       =20, num_classes: int = 1000, act_layer: nn.Module = nn.ReLU
       ()):
        super(Linear, self).__init__()
        self.num_classes = num_classes
        self.input_size = input_size
        self.hidden_layers_width = hidden_layers_width
        self.t = t
        layers: List[nn.Module] = []
        self.layers_width = [self.input_size]+self.
           hidden_layers_width
        for i in range(len(self.layers_width)-1):
            layers += [nn.Linear(self.layers_width[i],
                                 self.layers_width[i+1]),
                                    act_layer]
        layers += [nn.Linear(self.layers_width[-1], num_classes,
           bias=False)]
```

```
        self.features = nn.Sequential(*layers)
        self._initialize_weights()


    def forward(self, x):

        x = x.view(x.size(0), -1)
        x = self.features(x)
        return x

    def _initialize_weights(self) -> None:

        for obj in self.modules():
            if isinstance(obj, (nn.Linear, nn.Conv2d)):
                nn.init.normal_(obj.weight.data, 0, 1 /
                                self.hidden_layers_width[0]**(self.
                                   gamma))
                if obj.bias is not None:
                    nn.init.normal_(obj.bias.data, 0, 1 /
                                    self.hidden_layers_width[0]**(
                                       self.gamma))
```

2.4.3 激活函数

本实验将比较不同激活函数对网络性能的影响。可选的激活函数包括:

- ReLU: $\max(0, x)$
- tanh: $\frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Sigmoid: $\frac{1}{1+e^{-x}}$
- $x * \tanh(x)$

下面是代码中用于获取激活函数类别的函数

代码片段:

```
def get_act_func(act_func):
    if act_func == 'Tanh':
        return nn.Tanh()
    elif act_func == 'ReLU':
        return nn.ReLU()
    elif act_func == 'Sigmoid':
        return nn.Sigmoid()
    elif act_func == 'xTanh':
        return xtanh()
    else:
        raise NameError('No such act func!')

act_func = get_act_func(args.act_func_name)
```

创建模型的代码

代码片段：

```
model = Linear(args.gamma, args.hidden_layers_width, args.input_dim
    , args.output_dim, act_func).to(args.device)
```

通过上述代码可以创建一个线性网络，同时初始化满足 $\mathcal{N}(0, \frac{1}{m^\gamma})$ 分布。

2.5 实验任务

2.5.1 任务一：不同初始化下神经网络参数演化情况

参考代码：condense.ipynb

实验将使用以下一维函数作为拟合目标:

$$f(x) = 0.2 * \mathrm{ReLU}(x - 1/3) + 0.2 * \mathrm{ReLU}(-x - 1/3)$$

运行代码，观察不同初始化方差 (ε) 对训练过程的影响：

设置不同的 γ 值（如 $0.1, 0.5$ 和 1），运行实验并比较结果，观察神经元特征（方向和幅度）的分布变化。需要注意调整学习率使得学习过程中 loss 曲线不会出现上升。

观测指标

对于一维输入的函数，并采用 ReLU 作为激活函数时，神经元方向和幅值的定义：

每个神经元的参数对 $(a_k, \boldsymbol{w}_k)$ 可以分离为一个单位方向特征 $\hat{\boldsymbol{w}}_k = \boldsymbol{w}_k/\|\boldsymbol{w}_k\|_2$ 和一个表示其对输出贡献的振幅 $A_k = |a_k|\|\boldsymbol{w}_k\|_2$, 即 $(A_k, \hat{\boldsymbol{w}}_k)$。对于一维输入，由于加入了偏置项，$\boldsymbol{w}_k = (w_k, b_k)$ 是二维的。因此，我们使用每个 $\hat{\boldsymbol{w}}_k$ 相对于 x 轴的 $[-\pi, \pi)$ 内的角度 $\hat{\Omega}_k$ 来表示其方向。即 $\hat{\Omega}_k = \arctan(\frac{b_k}{w_k})$。

下面是代码中用于计算某个 checkpoint 中神经元方向和幅值的代码

代码片段：

```
def get_ori_A(checkpoint):
    wei1, bias, wei2 = get_parameter(checkpoint)

    wei1 = wei1.squeeze()
    bias = bias.squeeze()
    wei2 = wei2.squeeze()
    wei = wei1 / (wei1 ** 2 + bias ** 2)**(1/2)

    bia = bias / (wei1 ** 2 + bias ** 2)**(1/2)
    ori = torch.sign(bia) * torch.acos(wei)
    A = wei2 * (wei1 ** 2 + bias ** 2)**(1/2)

    return ori, A
```

预期结果参考图 2.2。

2.5.2 任务二：高维函数的初始参数凝聚实验

参考代码：initial condense.ipynb

实验将采用 5 维输入，1 维输出的函数：

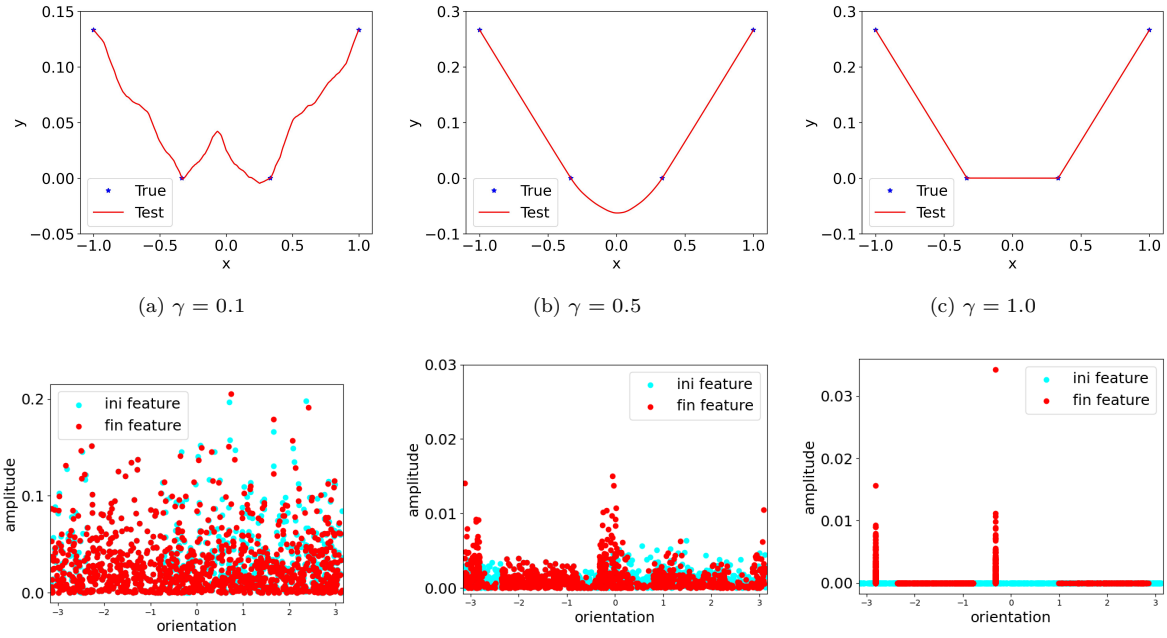
$$f(x) = \sum_{k=1}^{5} 3.5 \sin(5x_k + 1)$$

其中数据 $\boldsymbol{x} = (x_1, x_2, x_3, x_4, x_5)$，从 $[-4, 2]$ 中均匀采样得到。

采用 Adam 优化器，以 Tanh 函数作为激活函数，进行实验。

观测指标

对于高维数据, 无法使用极坐标系可视化神经元的特征。在这种情况下, 我们可以利用余弦相似度来衡量两个向量之间夹角的大小。



(a) $\gamma = 0.1$ (b) $\gamma = 0.5$ (c) $\gamma = 1.0$

(d) $\gamma = 0.1$ (e) $\gamma = 0.5$ (f) $\gamma = 1.0$

图 2.2: 不同大小初始化下，两层 ReLU 网络的预期实验结果。

余弦相似度: 两个向量 $\boldsymbol{u}$ 和 $\boldsymbol{v}$ 的余弦相似度定义为

$$D(\boldsymbol{u}, \boldsymbol{v}) = \frac{\boldsymbol{u}^\intercal \boldsymbol{v}}{(\boldsymbol{u}^\intercal \boldsymbol{u})^{1/2} (\boldsymbol{v}^\intercal \boldsymbol{v})^{1/2}}. \tag{2.2}$$

如果余弦相似度为 1，则说明两个向量平行

下面是代码中用于计算某个 checkpoint 中神经元余弦相似度的代码

代码片段:

```
def seperate_vectors_by_eigenvector(vector_group):
    mask = np.linalg.norm(vector_group,axis=1) > 0
    vector_group = vector_group[mask]
    similarity_matrix = np.dot(vector_group,vector_group.transpose
        ())
    w,v = np.linalg.eig(similarity_matrix)
    index = np.argmax(w)
    tmpeig = v[:,index]
    order_mask = np.argsort(tmpeig)
```

```
    similarity_matrix = similarity_matrix[order_mask,:]
    similarity_matrix = similarity_matrix[:,order_mask]
    return similarity_matrix,order_mask
```

需要注意的是为了可视化参数凝聚的聚类，我们按照余弦相似度矩阵的最大特征值对应的特征向量对参数向量进行一个排序，以达到同一方向的向量可以在相邻位置。

预期在 $\boldsymbol{w}_k$ 的结果：

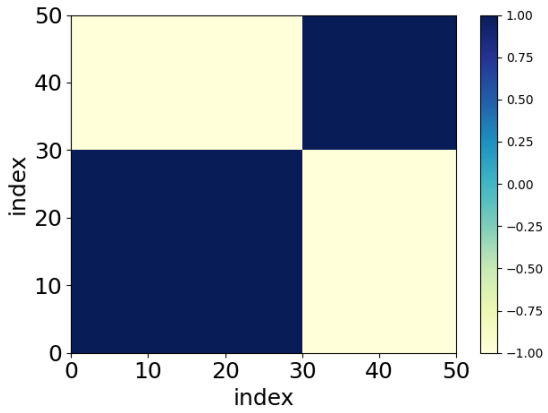


图 2.3: 初始凝聚时余弦相似度热力图结果。

2.6 课后任务和思考

尝试实现以 $x * \tanh(x)$ 为激活函数的两层神经网络的初始凝聚现象，画出其余弦相似度热力图结果。可以思考一下不同激活函数对于初始参数凝聚有什么影响。

Chapter 3

实验手册：乐观估计的验证

Jupyter 代码可以在 GitHub 找到[1]。

3.1 实验背景：过参数化下的泛化

当我们考虑泛化这个问题时，传统机器学习理论强调模型的复杂度应当与训练样本的数量相匹配，以避免过拟合现象的发生。然而，深度学习的实践却往往与这一理论预期形成鲜明对比：即使在模型参数数量远超训练样本数量的情况下，深度神经网络仍然能够展现出优异的泛化性能。这一反直觉的现象被学术界称为" 泛化之谜"（Generalization Puzzle）。本实验旨在通过" 乐观估计"（Optimistic Estimate）理论框架，深入探索这一现象背后的本质机制。乐观估计是一种创新的理论工具，用于估算训练神经网络恢复目标函数（达到零泛化误差）时所需的最小样本量。其核心思想可概括为：

1. 在最优条件下（乐观初始化）考察模型性能，即将参数初始化选择在理想点附近
2. 在该理想点附近对模型进行线性近似
3. 基于线性近似推导出恢复目标函数所需的最小样本量

3.2 实验目标

本实验的核心目标是验证乐观估计理论在一些机器学习模型中的适用性。通过实验，我们将验证理论推导得到的乐观样本量作为恢复目标函数所需样本量下界的有效性。特别地，我们

[1] https://github.com/xuzhiqin1990/understanding_dl/tree/main/code/Optimistic_Estimate

将验证，通过调参，一些简单回归模型和矩阵分解模型恢复目标函数所需样本量就是乐观样本量，一些更复杂的神经网络模型恢复目标函数所需样本量能够接近乐观样本量。

3.3 理论基础

在回归问题中,我们使用模型 $f_{\boldsymbol{\theta}}$ 拟合从目标函数 f^* 采样的 n 个数据点,目标是恢复 f^*。为了使“恢复”可行,我们考虑可以通过模型 $f_{\boldsymbol{\theta}}$ 表示的目标函数,即 $f^* \in \mathcal{F} := \{f(\cdot;\boldsymbol{\theta})|\boldsymbol{\theta} \in \mathbb{R}^M\}$。默认情况下，我们使用梯度下降来解决回归问题

$$\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} \left(f\left(\boldsymbol{x}_i;\boldsymbol{\theta}\right) - f^*(\boldsymbol{x}_i)\right)^2. \tag{3.1}$$

解被记为 $f_{\boldsymbol{\theta}_\infty}$，其中 $\boldsymbol{\theta}_\infty$ 表示收敛时的模型参数。接下来我们定义目标集:

Definition 3.3.1. 目标函数对应的参数集合（又叫目标集）$\boldsymbol{\Theta}_{f^*} := \left\{\boldsymbol{\theta} \mid f_{\boldsymbol{\theta}} = f^*, \boldsymbol{\theta} \in \mathbb{R}^M\right\}$

在过参数化下，有很多解使得模型的训练误差为 0，但是其中大部分解的泛化很差，所以在最差情况下，我们就不能恢复目标函数。但是我们之前的实验表明，非线性模型具有过参数化下恢复目标函数的能力，考虑最坏情况的理论不能解释泛化之谜的实验现象。但是，如果我们考虑最好的情况，将参数的初始化选择在一个很好的点，我们可能可以在过参数化下恢复目标函数。

首先，一个平凡的最优情况是，参数的初始化在目标集 $\boldsymbol{\Theta}_{f^*}$ 中，此时样本量为 0 就可以恢复目标函数。但是，第一，这样的样本量的估计不给我们带来任何有用的信息，实践中 0 个样本量显然是不能恢复目标函数的。第二，目标集 $\boldsymbol{\Theta}_{f^*}$ 的测度一般为 0。在均匀随机采样时，取到这种初始化的概率为 0。所以这种初始化是不可行的。一个次优情形（称之为乐观初始化）是参数的初始化在目标集附近，定义如下:

乐观初始化: 在目标集 $\boldsymbol{\Theta}_{f^*}$ 的一个“最优”点 $\boldsymbol{\theta}^*$ 的邻域中初始化。

我们使用线性近似 $f(\cdot;\boldsymbol{\theta}) \approx f^{\text{lin}}(\cdot;\boldsymbol{\theta}) := f^*(\cdot) + \nabla f_{\boldsymbol{\theta}'}(\cdot)^{\mathrm{T}}(\boldsymbol{\theta} - \boldsymbol{\theta}')$ 来分析在小邻域 $N(\boldsymbol{\theta}', \epsilon)$ 中拟合所需的样本量，其中 $\boldsymbol{\theta}' \in \boldsymbol{\Theta}_{f^*}$。然后，回归问题 (3.1)变为

$$\min_{\boldsymbol{\theta} \in N(\boldsymbol{\theta}', \epsilon)} \frac{1}{n} \sum_{i=1}^{n} \left(f^{\text{lin}}\left(\boldsymbol{x}_i;\boldsymbol{\theta}\right) - f^*(\boldsymbol{x}_i)\right)^2. \tag{3.2}$$

这是一个线性问题，$f^{\text{lin}}\left(\boldsymbol{x}_i;\boldsymbol{\theta}\right)$ 作为 $\boldsymbol{x}$ 的函数位于一个线性空间中，该线性空间的维数是 $\operatorname{rank}(\nabla f_{\boldsymbol{\theta}'}(\cdot)^{\mathrm{T}}) = \dim\left(\operatorname{span}\left\{\partial_{\theta_i} f(\cdot;\boldsymbol{\theta}')\right\}_{i=1}^{M}\right)$。有一个广为人知的结果：我们需要

$$n = \dim\left(\operatorname{span}\left\{\partial_{\theta_i} f(\cdot;\boldsymbol{\theta}')\right\}_{i=1}^{M}\right)$$

个样本来恢复 f^*。受微分拓扑学中秩的启发，我们将这个量称为**模型秩**，即先算模型关于每个参数求导得到一个切函数，然后算这些切函数张成线性空间的维数。

Definition 3.3.2 (**参数点处的模型秩**)**.** 对于参数可微的模型 $f_{\boldsymbol{\theta}}$ 和参数点 $\boldsymbol{\theta}^* \in \mathbb{R}^M$，模型秩定义为：

$$R_{f_{\boldsymbol{\theta}}}(\boldsymbol{\theta}^*) := \dim(\mathrm{span}\{\partial_{\theta_i} f(\cdot;\boldsymbol{\theta}^*)\}_{i=1}^M) \tag{3.3}$$

其中 $\mathrm{span}\{\phi_i(\cdot)\}_{i=1}^M = \{\sum_{i=1}^M a_i\phi_i(\cdot) \mid a_1,\ldots,a_M \in \mathbb{R}\}$ 表示基函数张成的空间 $\dim(\cdot)$ 表示线性函数空间的维数。

例 1. 对于一个简单的线性回归模型：$f(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{\theta}^\top \boldsymbol{x}, \boldsymbol{\theta}, \boldsymbol{x} \in \mathbb{R}^d$，在任何一个参数点 $\boldsymbol{\theta}^*$ 处，模型秩的计算过程为：先关于每个参数 θ_i 求导得到一个切函数 $\partial_{\theta_i} f(\boldsymbol{x};\boldsymbol{\theta}) = x_i$，然后算这些切函数张成线性空间的维数：

$$R_{f_{\boldsymbol{\theta}}}(\boldsymbol{\theta}^*) = \dim(\mathrm{span}\{\partial_{\theta_i} f(\cdot;\boldsymbol{\theta}^*)\}_{i=1}^d) = \dim(\mathrm{span}\{x_i\}_{i=1}^d) = d. \tag{3.4}$$

Definition 3.3.3 (**函数的模型秩**)**.** 对于任意目标函数 $f^* \in \mathcal{F}$，其中 $\mathcal{F} := \{f(\cdot;\boldsymbol{\theta}) \mid \boldsymbol{\theta} \in \mathbb{R}^M\}$ 为模型函数空间，令 $\Theta_{f^*} := \boldsymbol{\theta} \mid f(\cdot;\boldsymbol{\theta}) = f^*, \boldsymbol{\theta} \in \mathbb{R}^M$ 表示目标参数集。函数的模型秩定义为：

$$R_{f_{\boldsymbol{\theta}}}(f^*) := \min_{\boldsymbol{\theta}^* \in \Theta_{f^*}} R_{f_{\boldsymbol{\theta}}}(\boldsymbol{\theta}^*) \tag{3.5}$$

接下来，我们将对一些具体的模型，计算出关于目标函数的模型秩，并与实验所需的最小样本量做比较。

3.4 实验步骤

3.4.1 理论计算阶段

1. 对给定模型 $f_{\boldsymbol{\theta}}$ 和目标函数 f^*，根据定义计算模型恢复目标函数所需的乐观样本量 $R_{f_{\boldsymbol{\theta}}}(f^*)$
2. 记录计算过程和理论结果

3.4.2 实验验证阶段

1. 从目标函数中采样，设计递增的样本量序列
2. 对每个样本量的实验：

(a) 使用接近零的小随机值进行参数初始化：过参数化模型在大的随机初始化下很容易出现泛化性能差的情况，我们考虑的是实际更常用的小初始化情况

(b) 采用全批量梯度下降进行训练：实验中的模型和数据相对简单，排除其他随机性的干扰，确保训练稳定性

(c) 仔细调整学习率以确保收敛：学习率的调整对实验结果有重要影响，大的学习率会影响收敛，小的学习率训练太慢

3. 记录每个样本量下模型恢复目标函数的效果，即泛化误差

4. 确定实际所需的最小样本量

5. 与理论计算的乐观样本量进行对比分析

3.5 简单的回归模型实验

3.5.1 理论计算

首先，我们选定一个简单的关于参数线性的模型：

$$f_{\mathrm{L}}(\boldsymbol{x};\boldsymbol{\theta}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2. \tag{3.6}$$

设定目标函数为 $f^*(\boldsymbol{x}) = 1+x_1$. 该模型具有 $M = 3$ 个参数。考虑任意的 $\boldsymbol{\theta}' = [\theta_0', \theta_1', \theta_2']^T \in \mathbb{R}^3$，有 $R_{f_{\mathrm{L}}}(\boldsymbol{\theta}') = \dim\left(\mathrm{span}\left\{1, x_1, x_2\right\}\right) = 3$。因此，对于任何 $f^* \in \mathcal{F}_{\mathrm{L}} := \{f_{\mathrm{L}}(\cdot;\boldsymbol{\theta}) | \boldsymbol{\theta} \in \mathbb{R}^3\}$，我们有 $R_{f_{\mathrm{L}}}(f^*) = M_{\mathrm{I}} = 3$。

接下来，我们对线性模型进行重新参数化为 $f_{\mathrm{NL}}(\boldsymbol{x};\boldsymbol{\theta}) = \theta_0 + \theta_1 x_1 + \theta_2\theta_3 x_2$，函数空间仍然保持为 $\mathcal{F}_{\mathrm{NL}} = \mathcal{F}_{\mathrm{L}}$，但模型在参数上变得非线性。模型的乐观样本量估计如下。作为参数空间 $\mathbb{R}^4$ 上的函数的模型秩为：

$$R_{f_{\mathrm{NL}}}(\boldsymbol{\theta}') = \dim\left(\mathrm{span}\{1, x_1, \theta_3' x_2, \theta_2' x_2\}\right) = \begin{cases} 2, & \theta_2' = \theta_3' = 0, \\ 3, & \text{others.} \end{cases} \tag{3.7}$$

所以模型的有效参数量 $M_{\mathrm{I}} = 3$。通过解决最小化问题 Eq. (3.5)，我们得到模型秩作为函数空间 $\mathcal{F}$ 上的函数为

$$R_{f_{\mathrm{NL}}}(f^*) = \begin{cases} 2, & f^* \in \{a_0 + a_1 x_1 | a_0, a_1 \in \mathbb{R}\}, \\ 3, & f^* \in \{a_0 + a_1 x_1 + a_2 x_2 | a_2 \neq 0, a_0, a_1, a_2 \in \mathbb{R}\}. \end{cases} \tag{3.8}$$

上述估计表明，非线性模型 f_{NL} 恢复 f^* 所需的乐观样本量为 $R_{f_{\mathrm{NL}}}(f^*) = 2$.

3.5.2 实验验证

对于目标函数 $f^*(\boldsymbol{x}) = 1 + x_1$, 我们分别用上述的线性和非线性模型来拟合，记录下每个样本量下模型恢复目标函数的效果。

首先我们定义好目标函数，并设置采样为 $[-1, 1]$ 均匀分布随机采样：

代码片段

```
# 定义目标函数
def f_star(x, a=1.0, b=0.0):
    y = 1 + a * x[:, 0] + b * x[:, 1]
    return y

def generate_data(a, b, num_samples):
    x = 2 * torch.rand(num_samples, 2) - 1
    y = f_star(x, a, b)
    return x, y

test_size = 1000
x_test, y_test = generate_data(1.0, 0.0, test_size)
test_dataset = torch.utils.data.TensorDataset(x_test, y_test)
```

这个简单模型在 Pytorch 中的实现如下：

```
class NonLinearModel(torch.nn.Module):
    def __init__(self):
        super(NonLinearModel, self).__init__()
        self.theta0 = torch.nn.Parameter(torch.randn(1) * sigma)
        self.theta1 = torch.nn.Parameter(torch.randn(1) * sigma)
        self.theta2 = torch.nn.Parameter(torch.randn(1) * sigma)
        self.theta3 = torch.nn.Parameter(torch.randn(1) * sigma)


    def forward(self, x):
        return self.theta0 + self.theta1 * x[:, 0] + self.theta2 *
            self.theta3 * x[:, 1]
```

训练过程如下：

```
batch_size, num_epochs, lr = 1024, 10000, 1e-1
test_iter = torch.utils.data.DataLoader(test_dataset, batch_size,
    shuffle=True)
train_ls_dict_NL = {}
test_ls_dict_NL = {}
for sample_size in range(1, 4):
    net = NonLinearModel()
    x_train, y_train = generate_data(1.0, 0.0, sample_size)
    train_dataset = torch.utils.data.TensorDataset(x_train, y_train
        )
    train_iter = torch.utils.data.DataLoader(train_dataset,
        batch_size, shuffle=True)
    train_ls_dict_NL[sample_size], test_ls_dict_NL[sample_size] =
        train(net, train_iter, num_epochs, lr, try_gpu(), test_iter=
        test_iter)
```

最终我们可以把线性模型和非线性模型的泛化误差随样本量变化画在一张图热力图上，并标出理论计算出的乐观样本量阈值，与实验结果比对。

```
import seaborn as sns
from matplotlib import cm
from matplotlib.colors import LogNorm, Normalize
set_figsize((3.5, 2.5))

sns.heatmap(np.array([[test_ls_dict[1][-1], test_ls_dict[2][-1],
    test_ls_dict[3][-1]], [test_ls_dict_NL[1][-1], test_ls_dict_NL
    [2][-1], test_ls_dict_NL[3][-1]]]), cmap='RdBu', annot=False,
    norm=LogNorm(vmax=1e0, vmin=1e-8))
plt.vlines(1, 1, 2, colors='yellow', linestyles='dashed')
plt.vlines(2, 0, 1, colors='yellow', linestyles='dashed')

plt.xlabel('Sample size')
plt.ylabel('Model')
plt.xticks([0.5, 1.5, 2.5], ['1', '2', '3'])
```

```
plt.yticks([0.5, 1.5], ['Linear', 'Non-linear'])
plt.show()
```

实验结果如下：

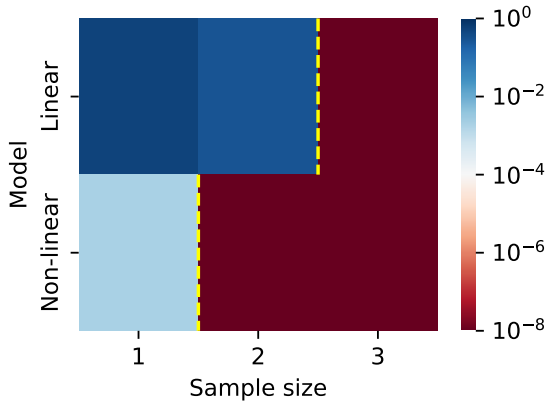


图 3.1: 实验结果

更进一步地，我们可以把线性模型和非线性模型在 1，2，3 样本下的训练过程可视化：

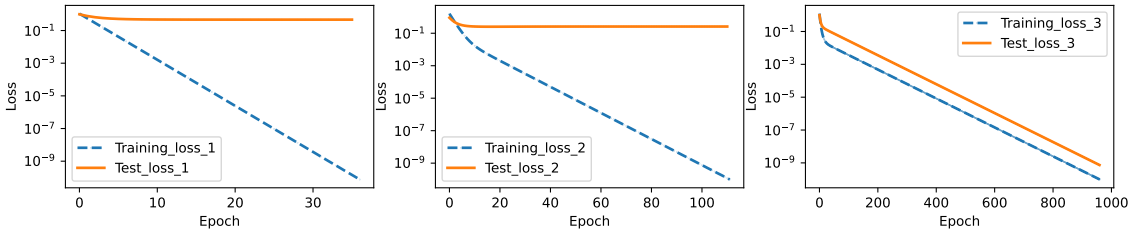


图 3.2: 线性模型。从左到右分别为样本量为 1，2，3 时的训练过程。

3.6 矩阵分解模型实验

3.6.1 理论计算

我们考虑一个 $\boldsymbol{f_\theta} = \boldsymbol{AB}$ 的非线性矩阵分解模型，这个模型在低秩矩阵补全任务中有应用，我们的目标是从 n 个观察到的元素 $S = \left\{\left((i_s, j_s), \boldsymbol{M}^*_{i_s j_s}\right)\right\}_{s=1}^{n}$ 中恢复目标矩阵 $\boldsymbol{M}^*$，其中 (i_s, j_s) 表示矩阵 M^* 的行和列的索引。换句话说，我们已知目标矩阵 $\boldsymbol{M}^*$ 中某些位置的值，我们想还原 $\boldsymbol{M}^*$ 其他位置的值。

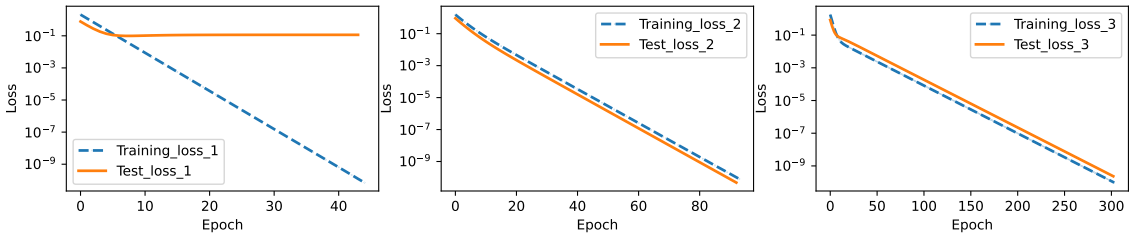


图 3.3: 非线性模型。从左到右分别为样本量为 1，2，3 时的训练过程。

考虑一个非常简单的低秩矩阵补全任务：

$$\boldsymbol{M} = \begin{bmatrix} 1 & 2 \\ 3 & \star \end{bmatrix} \tag{3.9}$$

如果我们直接设定一个关于参数线性的模型 $f_{\boldsymbol{\theta}} = \boldsymbol{W} \in \mathbb{R}^{d\times d}$，并从零附近初始化用梯度下降并最小化以下目标：

$$\min_{\boldsymbol{\theta}} \frac{1}{n}\sum_{s=1}^{n} \left([\boldsymbol{W}]_{i_s j_s} - \boldsymbol{M}^*_{i_s j_s}\right)^2, \tag{3.10}$$

$\boldsymbol{W}$ 的第二行第二列的元素拿不到梯度，所以不会被训练，因此一般学不到低秩解。

但是如果我们考虑一个 $\boldsymbol{f_\theta} = \boldsymbol{AB}, \boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{d\times d}$ 的非线性矩阵分解模型，采用梯度下降并最小化以下目标：

$$\min_{\boldsymbol{\theta}} \frac{1}{n}\sum_{s=1}^{n} \left([\boldsymbol{AB}]_{i_s j_s} - \boldsymbol{M}^*_{i_s j_s}\right)^2, \tag{3.11}$$

其中 $\boldsymbol{M}^* \in \mathbb{R}^{d\times d}$，$\boldsymbol{\theta} = (\boldsymbol{A}, \boldsymbol{B})$，以及 $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{d\times d}$，$[\boldsymbol{AB}]_{i_s j_s}$ 代表取出矩阵 $\boldsymbol{W} = \boldsymbol{AB}$ 第 i_s 行，第 j_s 列的数据。在这个非线性模型中，$\boldsymbol{A}$ 的第二行和 $\boldsymbol{B}$ 的第二列都会被训练，所以模型有可能学到低秩解，从而在只采样 3 个样本的时候就恢复这个秩 1 矩阵。

通过采用标准的乐观估计过程，我们可以确定任何 $\boldsymbol{M}^* \in \mathbb{R}^{d\times d}$ 的乐观样本量为（留作习题）：

$$R_{\boldsymbol{f_\theta}}(\boldsymbol{M}^*) = 2r_{\boldsymbol{M}^*}d - r^2_{\boldsymbol{M}^*}, \tag{3.12}$$

其中 $r_{\boldsymbol{M}^*} = \mathrm{rank}(\boldsymbol{M}^*)$ 是 $\boldsymbol{M}^*$ 的矩阵秩。

实际上，这个结果和秩 r 矩阵的自由度是一样的，例如考虑一个 2×2 的矩阵 $\boldsymbol{A}$，如果要求它的秩为 1，那么它就只能表示为两个向量的外积形式：$\boldsymbol{A} = \boldsymbol{uv}^\top$，其中 $\boldsymbol{u}$ 是一个 2×1 的列向量，$\boldsymbol{v}$ 是一个 2×1 的列向量。因此，矩阵 $\boldsymbol{A}$ 的元素可以用向量 $\boldsymbol{u}$ 和 $\boldsymbol{v}$ 的元素表示。具体地说，$\boldsymbol{u}$ 有 2 个分量，$\boldsymbol{v}$ 也有 2 个分量，总共是 4 个分量。然而，它们有一个尺度上的冗余，因为如果我们把 $\boldsymbol{u}$ 扩大某个常数倍，$\boldsymbol{v}$ 缩小同一个常数倍，矩阵 $\boldsymbol{A}$ 的值并不会改变，因此其自由度为 $2rd - r^2 = 2\times 1\times 2 - 1^2 = 3$.

值得注意的是，对于固定的 d，乐观样本量随 $r_{\boldsymbol{M}^*}$ 增加而增加。特别是，对于大的 d，秩为 $O(r)$ 的目标矩阵的乐观样本量为 $O(r^2)$，这比有效参数量 $M_{\mathrm{I}} = d^2$ 要小得多。

我们设定目标矩阵分别为一个 4×4 的秩 1，2，3 矩阵，其乐观样本量 $2rd - r^2$ 计算结果分别为：$7, 12, 15$.

3.6.2 实验验证

对于每个目标矩阵, 我们分别用上述的矩阵分解来拟合，记录下每个样本量下模型恢复目标函数的效果。

首先我们定义好目标矩阵，分别为：

$$\boldsymbol{M}_1^* = \begin{bmatrix} 4 & 0.6 & 1.8 & 0.8 \\ 8 & 1.2 & 3.6 & 1.6 \\ 8 & 1.2 & 3.6 & 1.6 \\ 6 & 0.9 & 2.7 & 1.2 \end{bmatrix}, \boldsymbol{M}_2^* = \begin{bmatrix} 4 & 0.6 & 1.8 & 0.8 \\ 10 & 2.7 & 5.1 & 3.6 \\ 8 & 1.2 & 3.6 & 1.6 \\ 6 & 0.9 & 2.7 & 1.2 \end{bmatrix}, \boldsymbol{M}_3^* = \begin{bmatrix} 4 & 0.6 & 1.8 & 0.8 \\ 8 & 2.7 & 5.1 & 3.6 \\ 8 & 2.2 & 2.6 & 1.6 \\ 6 & 0.9 & 2.7 & 1.2 \end{bmatrix}, \quad (3.13)$$

很容易验证：$\mathrm{rank}(\boldsymbol{M}_1^*) = 1, \mathrm{rank}(\boldsymbol{M}_2^*) = 2, \mathrm{rank}(\boldsymbol{M}_3^*) = 3$.

我们定义采样方式以连通的方式采样，先采第一行第一列，再采第二行第二列，再采第三行第三列，最后采第四行第四列。

模型的实现和训练如下：

代码片段

```
d = 4
num_inputs, num_outputs, num_hiddens = d, d, d
loss = nn.MSELoss()
class MatrixFactorization(nn.Module):
    def __init__(self):
        super(MatrixFactorization, self).__init__()
        self.linear_stack = nn.Sequential(
            nn.Linear(num_inputs, num_hiddens, bias=False),
            nn.Linear(num_hiddens, num_outputs, bias=False)
        )
    def forward(self, x):
        y = self.linear_stack(torch.eye(d, d))
        return y[x.T.numpy().tolist()].reshape(-1, 1), y
net = MatrixFactorization()
```

训练过程如下：

```
outputs_sample_ls = {}
theta_A_sample_ls = {}
theta_B_sample_ls = {}
batch_size, num_epochs, lr = 1024, 100000, 1e-1
M = M1
test_ls_M1 = {}
for i, ls in enumerate(sample_patterns):
    features_train, labels_train = fix_sample(M, ls=ls)
    train_dataset = torch.utils.data.TensorDataset(features_train,
       labels_train)
    def init_weights(m, sigma=1e-7):
            if type(m) == nn.Linear or type(m) == nn.Conv2d:
                nn.init.normal_(m.weight, 0, sigma)
                if m.bias:
                    nn.init.normal_(m.bias, 0, sigma)
    net.apply(init_weights)
    train_iter = torch.utils.data.DataLoader(train_dataset,
       batch_size, shuffle=True)
    train_ls, test_ls = train(net, train_iter, num_epochs, lr,
       try_gpu())
    AB = net(features_train)[1].detach().clone()
    A = np.copy(list(net.parameters())[0].detach().T)
    theta_A_sample_ls[i+1]=(A)
    B = np.copy(list(net.parameters())[1].detach().T)
    theta_B_sample_ls[i+1]=(B)
    outputs_sample_ls[i+1]=(AB)
    # frobenius norm
    test_ls_M1[i+1] = np.linalg.norm(AB - M1.numpy(), ord='fro') /
       d ** 2
```

最终我们可以把线性模型和非线性模型的泛化误差随样本量变化画在一张图热力图上，并标出理论计算出的乐观样本量阈值，与实验结果比对。

```
import seaborn as sns
from matplotlib import cm
from matplotlib.colors import LogNorm, Normalize
set_figsize((3.5, 2.5))
test_ls_array = np.array([list(test_ls_M1.values()), list(
    test_ls_M2.values()), list(test_ls_M3.values())])
test_ls_array
sns.heatmap(test_ls_array, cmap='RdBu', annot=False, norm=LogNorm(
    vmax=1e0, vmin=1e-4))
plt.vlines(6, 0, 1, colors='yellow', linestyles='dashed')
plt.vlines(11, 1, 2, colors='yellow', linestyles='dashed')
plt.vlines(14, 2, 3, colors='yellow', linestyles='dashed')
plt.xlabel('Sample size')
plt.xticks(np.array([0, 2, 4, 6, 8, 10, 12, 14])+0.5, [1, 3, 5, 7,
    9, 11, 13, 15], rotation=0)
plt.yticks([0.5, 1.5, 2.5], ['Rank-1', 'Rank-2', 'Rank-3'])
plt.show()
```

实验结果如下，可以看到理论计算得到的乐观样本量和实验所需的最小样本量完全吻合。

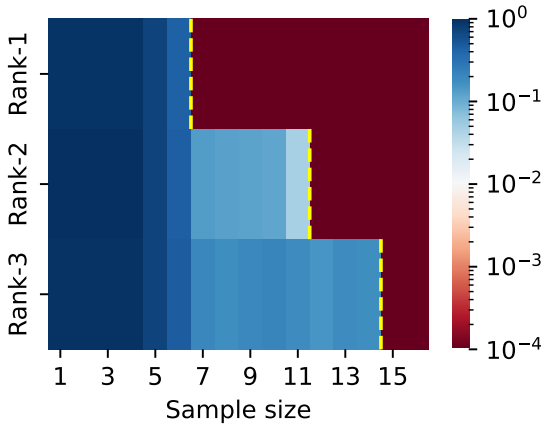


图 3.4: 实验结果。黄线代表理论计算的乐观样本量，colorbar 代表泛化误差。

3.7 神经网络模型实验

3.7.1 理论计算

首先，我们考虑一个具有 m 个神经元的两层全连接神经网络，其激活函数为 $\tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$，神经网络由 $f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sum_{i=1}^{m} a_i \tanh(\boldsymbol{w}_i^{\mathrm{T}} \boldsymbol{x})$ 表示，其中 $\boldsymbol{x} \in \mathbb{R}^d, \boldsymbol{\theta} = (a_i \in \mathbb{R}, \boldsymbol{w}_i \in \mathbb{R}^d)_{i=1}^m$。网络总共有 $M = m(d+1)$ 个参数。经过计算得到，乐观样本量与目标函数 f^* 的内在宽度 $k(f^*)$ 有关。

Definition 3.7.1. *内在宽度 $k(f^*)$：定义为可以表示 f^* 的神经网络的最小宽度，即 f^* 可以由宽度为 $k(f^*)$ 的神经网络表示，但不能由任何更窄的神经网络表示。*

对于一个可以由宽度为 m 的神经网络表示的 f^*，其内在宽度 $0 \leqslant k(f^*) \leqslant m$。$f^*$ 的乐观样本量为

$$R_{\mathrm{NN}_m}(f^*) = k(f^*)(d+1), \tag{3.14}$$

实现该乐观样本量的 $\boldsymbol{\theta}^*$ 的一种取法是，先选取 $k(f^*)$ 个神经元来表示 f^*，然后令其他的神经元的 a 和 $\boldsymbol{w}$ 全为 0，此时 $\boldsymbol{\theta}^*$ 对应的模型秩就是 $k(f^*)(d+1)$。

我们选取目标函数为一个神经元：$f^*(x) = \tanh(x+1)$，内在宽度为 1，对于宽度比它更宽的网络模型，乐观样本量计算结果均为 3.

3.7.2 实验验证

我们分别用宽度为 2 和宽度为 20 的两层神经网络来拟合目标函数，观察实验所需的样本量是否发生显著变化。

模型的实现如下：

代码片段

```
# 神经网络模型，宽度为 2
num_inputs, num_outputs, num_hiddens = 1, 1, 2
loss = nn.MSELoss()
sigma = 1e-12 # 初始化参数的标准差
class NeuralNetwork(torch.nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.hidden = nn.Linear(num_inputs, num_hiddens)
        self.activation = nn.Tanh()
```

```
        self.output = nn.Linear(num_hiddens, num_outputs, bias=
            False)

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

对于神经网络实验，小初始化下泛化误差依然会有随机性，我们对每个样本量跑 10 次随机实验，并对泛化误差取平均值

代码片段

```
batch_size, num_epochs, lr = 1024, 10000, 1e-1
test_iter = torch.utils.data.DataLoader(test_dataset, batch_size,
    shuffle=True)
train_ls_dict_NN_2 = {}
test_ls_dict_NN_2 = {}
def init_weights(m, sigma=1e-12):
    if type(m) == nn.Linear or type(m) == nn.Conv2d:
        nn.init.normal_(m.weight, 0, sigma)
        if m.bias is not None:
            nn.init.normal_(m.bias, 0, sigma)
for sample_size in range(1, 10):
    l = 0
    num_trails = 10
    for trail in range(num_trails):
        net = NeuralNetwork()
        net.apply(init_weights)
        x_train, y_train = generate_data(1.0, 1.0, 1.0, sample_size
            )
        train_dataset = torch.utils.data.TensorDataset(x_train,
            y_train)
        train_iter = torch.utils.data.DataLoader(train_dataset,
            batch_size, shuffle=True)
        train_ls_dict_NN_2[sample_size], _ = train(net, train_iter,
            num_epochs, lr, try_gpu(), test_iter=None)
        # compute the test loss
```

```
        net.eval()
        with torch.no_grad():
            x_test, y_test = x_test.to(try_gpu()), y_test.to(
                try_gpu())
            y_hat = net(x_test)
            l += loss(y_hat, y_test)
    test_ls_dict_NN_2[sample_size] = l / num_trails
```

最终我们可以把宽度为 2 和宽度为 20 的网络泛化误差随样本量变化画在一张图热力图上，并标出理论计算出的乐观样本量阈值，与实验结果比对。

```
import seaborn as sns
from matplotlib.colors import LogNorm, Normalize
set_figsize((3.5, 2.5))
test_ls_array = np.array([list(test_ls_dict_NN_2.values()), list(
    test_ls_dict_NN_20.values())])
sns.heatmap(test_ls_array, cmap='RdBu', annot=False, norm=LogNorm(
    vmax=1e0, vmin=1e-4))
plt.vlines(2, 0, 1, colors='yellow', linestyles='dashed')
plt.vlines(2, 1, 2, colors='yellow', linestyles='dashed')
plt.xlabel('Sample size')
plt.xticks(np.arange(10)+0.5, range(1, 11), rotation=0)
plt.yticks([0.5, 1.5], ['Width-2', 'Width-20'])
plt.show()
```

3.8 实验总结

尽管乐观初始化在实践中似乎难以实现（因为我们无法预先知晓目标函数 f^* 及其目标参数集 $\boldsymbol{\Theta}_{f^*} := \{\boldsymbol{\theta} \mid f_{\boldsymbol{\theta}} = f^*, \boldsymbol{\theta} \in \mathbb{R}^M\}$），但从乐观初始化导出的乐观估计理论工具具有重要的理论价值：它为所需样本量确立了一个明确的下界。更为重要的是，我们的实验结果表明，使用常规的小随机初始化方法无需采用乐观初始化，却能达到接近这一理论界限的性能。具体表现为：简单架构的模型（如回归模型和矩阵分解模型）在小初始化下能够稳定地达到这一界限，较为复杂的模型（如神经网络）也能接近这一界限。

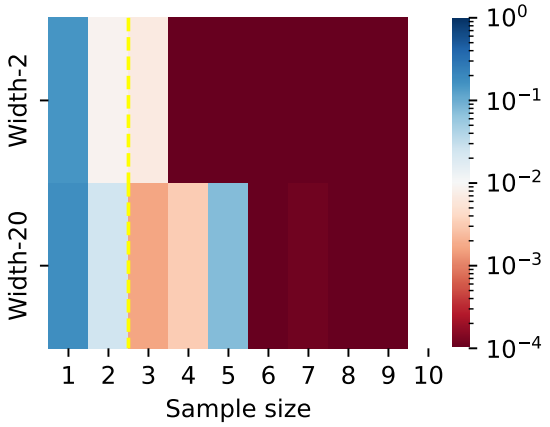


图 3.5: 实验结果

3.9 作业

1. 对于矩阵分解任务，设 $\boldsymbol{M}^* \in \mathbb{R}^{d\times d}$，证明乐观样本量为

$$R_{\boldsymbol{f_\theta}}(\boldsymbol{M}^*) = 2r_{\boldsymbol{M}^*}d - r^2_{\boldsymbol{M}^*}, \tag{3.15}$$

   其中 $r_{\boldsymbol{M}^*} = \mathrm{rank}(\boldsymbol{M}^*)$ 是 $\boldsymbol{M}^*$ 的矩阵秩。

2. 对 5×5 的矩阵分解任务，实验上验证小初始化下能达到乐观样本量。

3. 探究矩阵补全任务中观测数据的具体位置分布如何影响实验中恢复目标函数所需的样本量。

4. 对更复杂的目标函数，验证神经网络恢复目标函数所需的样本量不会小于乐观样本量。

Chapter 4

实验手册：语言模型实现多步推理的机制探究

Jupyter 代码可以在 GitHub 找到[1]。

4.1 实验背景

近年来，大语言模型 (LLMs) 崭露头角，在各种任务中展现出了卓越的能力。这些模型表现出令人印象深刻的上下文学习能力，并被应用于逻辑推理问题，如在国际数学奥林匹克竞赛 (IMO) 级别上匹敌顶尖人类选手，以及解决数学问题。然而，当前最先进的模型在面对复杂推理任务时仍然存在困难，为了真正提升 LLMs 的推理能力，研究其内在推理机制至关重要。本实验聚焦于 LLMs 如何在其架构内处理多步推理，这有助于开发更有效的策略来提高它们的多步推理能力。

多步推理任务包含一个广泛的概念，通常指模型综合多种复杂条件以回答问题的能力。在这里，我们考虑多步推理任务中的一种代表性句法结构：包含问题以及足够已知信息来回答该问题的句子。例如，“[A] → [B]...[B] → [C]...[A]” 的结构，其中”...” 代表与逻辑推理无关的其他文本内容，我们期望模型直接输出 “[C]”。

4.2 实验目标

1. 训练一个小规模的 Transformer 模型来实现多步推理

[1] https://github.com/xuzhiqin1990/understanding_dl/tree/main/code/multi-step_reasoning_code

2. 学习如何分析和可视化语言模型的“思考”过程

4.3 实验准备

4.3.1 多步推理数据集

为了理解 Transformer 中的多步推理机制，我们设计了一个抽象的符号化多步推理任务。如图 4.2(a) 所示，推理链被序列化为一个序列。句子中的每两个 token 代表一个推理关系。最后一个 token 是推理起点 token，标签是从起点开始进行固定步长推理得到的结果。

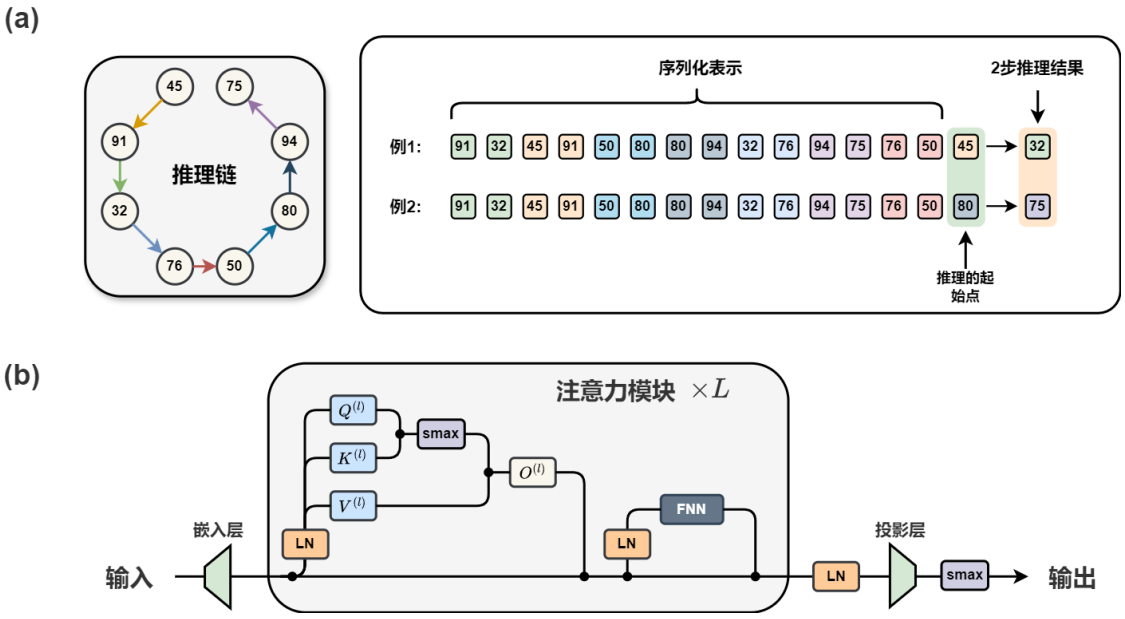


图 4.1: 多步推理数据集和 Transformer 架构示意图。

4.3.2 Transformer 模型

我们采用一个仅包含解码器的单头 Transformer 模型（图 4.2(b)）。给定一个输入序列 $\boldsymbol{X}^{\text{in}} \in \mathbb{R}^{n\times d}$，其中 n 是序列长度，d 是词典大小，模型首先使用嵌入层（token 嵌入和位置嵌入）获得输入表示 $\boldsymbol{X}^{(1)} = \boldsymbol{X}_{\text{tgt}} + \boldsymbol{X}_{\text{pos}} \in \mathbb{R}^{n\times d_m}$。每层中的单头注意力计算如下：

$$\mathcal{A}^{(l)}(\boldsymbol{X}) = \sigma\left(\frac{\text{mask}(\boldsymbol{X}\boldsymbol{W}^{q(l)}\boldsymbol{W}^{k(l),\mathsf{T}}\boldsymbol{X}^{\mathsf{T}})}{\sqrt{d_k}}\right), \quad \boldsymbol{X}^{\text{qkv}(l)} = \mathcal{A}^{(l)}(\bar{\boldsymbol{X}}^{(l)})\bar{\boldsymbol{X}}^{(l)}\boldsymbol{W}^{v(l)}\boldsymbol{W}^{o(l)},$$

其中 σ 表示 softmax 操作，$\bar{\boldsymbol{X}}^{(l)} = \text{Layernorm}(\boldsymbol{X}^{(l)})$。为简化表达，我们将 $\boldsymbol{W}^{q(l)}\boldsymbol{W}^{k(l),\mathsf{T}}$ 简写为 $\boldsymbol{W}^{qk(l)}$，将 $\boldsymbol{W}^{v(l)}\boldsymbol{W}^{o(l),\mathsf{T}}$ 简写为 $\boldsymbol{W}^{vo(l)}$。第 l 层的输出为：

$$\boldsymbol{X}^{\text{ao}(l)} = \boldsymbol{X}^{(l)} + \boldsymbol{X}^{\text{qkv}(l)}, \quad \boldsymbol{X}^{(l+1)} = f^{(l)}(\bar{\boldsymbol{X}}^{\text{ao}(l)}) + \boldsymbol{X}^{\text{ao}(l)},$$

其中 $f^{(l)}(\cdot)$ 表示第 l 层的前馈神经网络。最终输出（以词汇表中的标记索引形式）为：

$$\boldsymbol{Y} = argmax(\sigma(\bar{\boldsymbol{X}}^{(L)}\boldsymbol{W}^{p})) \in \mathbb{R}^{n}.$$

在我们本次实验中,我们设置字典大小为 $d = 201$,隐藏空间维度为 $d_m = 400$,$\boldsymbol{W}^q, \boldsymbol{W}^k, \boldsymbol{W}^v$ 的隐藏空间维度为 $d_q = d_k = d_v = 64$。

4.3.3 数据集的划分

在我们的数据集设置下，如果模型真正理解了底层逻辑模式，那么在测试阶段，即使句子中包含训练期间完全没见过的 token，它也应能够输出正确的答案。因此，我们将数据分为两部分：分布内（In-Distribution, ID）和分布外（Out-of-Distribution, OOD）。具体来说，我们定义 $\texttt{token}_{\texttt{ID}} \in [1, 100]$ 和 $\texttt{token}_{\texttt{OOD}} \in [101, 200]$。分布内数据（$\texttt{Train}_{\texttt{ID}}$ 和 $\texttt{Test}_{\texttt{ID}}$）定义为完全由 $\texttt{token}_{\texttt{ID}}$ 组成的句子，而分布外数据（$\texttt{Test}_{\texttt{OOD}}$）的每条句子中则包含多个 $\texttt{token}_{\texttt{OOD}}$。对于分布内数据，我们按照以下规则划分训练集（$\texttt{Train}_{\texttt{ID}}$）和测试集（$\texttt{Test}_{\texttt{ID}}$）：对于训练集的推理链 $\texttt{[x}_1\texttt{][x}_2\texttt{]}\cdots\texttt{[x}_n\texttt{]}$，所有 token 满足以下条件：

$$\texttt{x}_{\texttt{2i}} - \texttt{x}_{\texttt{2i-1}} \pmod m \in G.$$

对于测试集中的推理链，所有 token 满足：

$$\texttt{x}_{\texttt{2i}} - \texttt{x}_{\texttt{2i-1}} \pmod m \in \{1, \cdots, m\} \backslash G,$$

在本实验中，我们取 $m = 5$ 和 $G = \{0, 1, 4\}$。在此设置下，我们确保测试集中的每个二元逻辑对在训练集中未曾出现过。

4.4 实验核心代码

4.4.1 数据生成

运行配套的 `datas = get_data(args)` 即可生成本实验中所需要的数据集。

数据集示例

```
# Train_ID
[10, 29, 51, 21, 16, 86, 29, 34, 21, 16, 86, 10, 10, 34]
[66, 87, 15, 50, 86, 96, 87, 86, 96, 97, 50, 66, 87, 96]
[60, 64, 11, 10, 5, 60, 64, 23, 87, 11, 10, 5, 60, 23]

# Test_ID
[91, 98, 98, 40, 54, 91, 40, 22, 22, 19, 19, 56, 54, 98]
[38, 61, 61, 59, 59, 42, 90, 53, 41, 38, 42, 90, 61, 42]
[81, 79, 26, 69, 19, 36, 79, 26, 69, 61, 61, 19, 81, 26]

# Test_OOD
[95, 8, 186, 165, 140, 105, 8, 127, 127, 186, 105, 95, 105, 8]
[124, 182, 182, 109, 90, 87, 176, 90, 170, 124, 109, 176, 176, 87]
[88, 124, 135, 165, 124, 101, 122, 71, 71, 88, 101, 135, 122, 88]
```

4.4.2 初始化模型、损失函数与优化器

本实验中我们使用 3 层 Transformer 模型，模型结构已在前文中引入。我们使用了一个简单的类 `myGPT()` 来实现 Transformer 的功能。

初始化模型、损失函数与优化器代码

```
# 初始化模型
model = myGPT(args, device).to(device)

if args.checkpoint != 'none':
    model.load_state_dict(torch.load(args.checkpoint))

# 损失函数
criterion = nn.CrossEntropyLoss(ignore_index=0).to(device)

# 优化器
optimizer = optim.AdamW(model.parameters(), lr=args.lr,
   weight_decay=args.weight_decay)
```

```
# 学习率调整策略
scheduler_cosine = CosineAnnealingLR(optimizer, T_max=int(args.
    optim_T_max), eta_min=float(args.optim_eta_min))

# multiplier 是最大学习率与初始学习率的比值, total_epoch 是预热的周
    期数, after_scheduler 是预热后的再使用的学习率调整策略
scheduler = GradualWarmupScheduler(optimizer,
            multiplier = float(args.optim_multiplier),
            total_epoch = int(args.optim_total_epoch),
            after_scheduler = scheduler_cosine)
```

4.5 实验结果

4.5.1 数据生成与模型训练

本实验中，我们生成了 300,000 条句长为 13 的 2 步推理数据来训练 Transformer 模型。初始学习率设置为 2e-5，并在 400 个训练步（epoch）内逐渐线性增长到 1e-4，再依据 cosine 衰减策略在后续的 3600 个训练步衰减到 1e-5。批量大小（batch size）设置为 1000。优化器为 AdamW 优化器。我们使用 Pytorch 提供的梯度衰减函数来控制梯度大小 `torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1)`。图 4.2(a) 展示了训练曲线，横坐标为训练步，纵坐标为三类数据集的准确率。可以观察到，模型在经过 20 多个训练步的训练后，不仅获得了分布内的泛化能力，也逐渐获得了分布外的泛化能力。

4.5.2 模型信息流可视化

为了更好地理解 Transformer 是如何处理多步推理数据的，我们需要绘制其处理这些数据过程中的“思考过程”，我们称之为信息流。具体来说，我们将每个 token 当做一个节点，两层之间连线由注意力机制和残差连接决定。首先，若 $\mathcal{A}_{i,j}^{(l)} > 0$，则将第 l 层第 j 个节点与第 $l+1$ 层第 i 个节点连一条实线，且线宽正相关于 $\mathcal{A}_{i,j}^{(l)}$，表示通过注意力机制来传递信息；其次，相邻两层之间相同的 token 用虚线连接，表示使用残差来传递信息。

图 4.3展示了 Transformer 模型在一个测试句子中的信息流传递过程，第一层实现了相邻两个位置信息的融合配对。之后每一层都会将一步推理的信息传递到最后一个 token 上。我们可以进一步将这一过程抽象化为示意图 4.2(b)，其中红色 token 是决定当前层信息流走向的关键信息。可以观察到，模型的“推理层”总是利用两个 token 间有相同的信息从而使其互相关

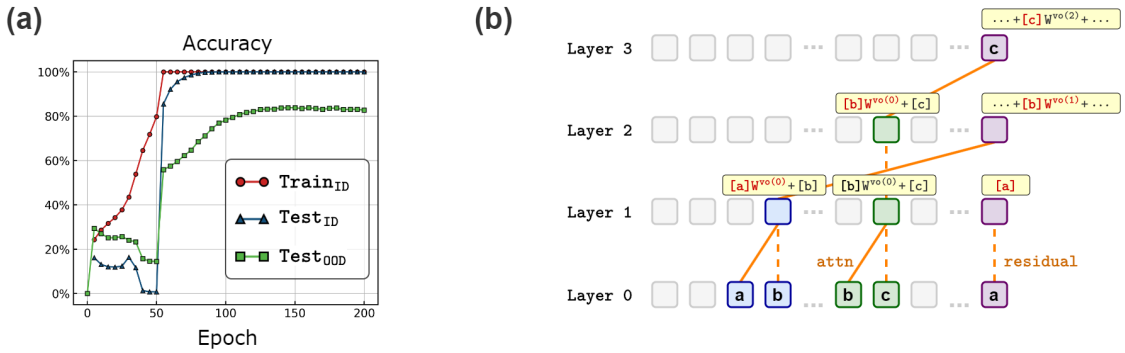


图 4.2: (a) Transformer 的训练准确率曲线。(b) Transformer 实现多步推理的信息流示意图。图中仅展示了对最终结果有影响的信息流。实线为由注意力机制引起的信息流，虚线为由残差连接引起的信息流。

注，从而实现信息间的传递。

信息流绘图代码片段

```
import matplotlib.pyplot as plt

def plot_info_broadcast(input_seq, output_seq, attn_list,
    key_points=[], key_flows=[], res_flows=[]):
    fig = plt.figure(figsize=(6,5), dpi=100)

    seq_len = attn_list[0].shape[0]
    layers = len(attn_list) + 1

    color_list = ['#9bbbe1', '#08519C']

    # 在每个纵坐标为4*i的位置，画seq_len个点，表示每层的每个位置
    for i in range(layers):
        for j in range(seq_len):
            if len(key_points)== 0 or (i, j) in key_points:
                plt.scatter(j, 4*i, c=color_list[j%2], edgecolors='
                    k',
                            s=80, linewidths=1, zorder=10)
            else:
```

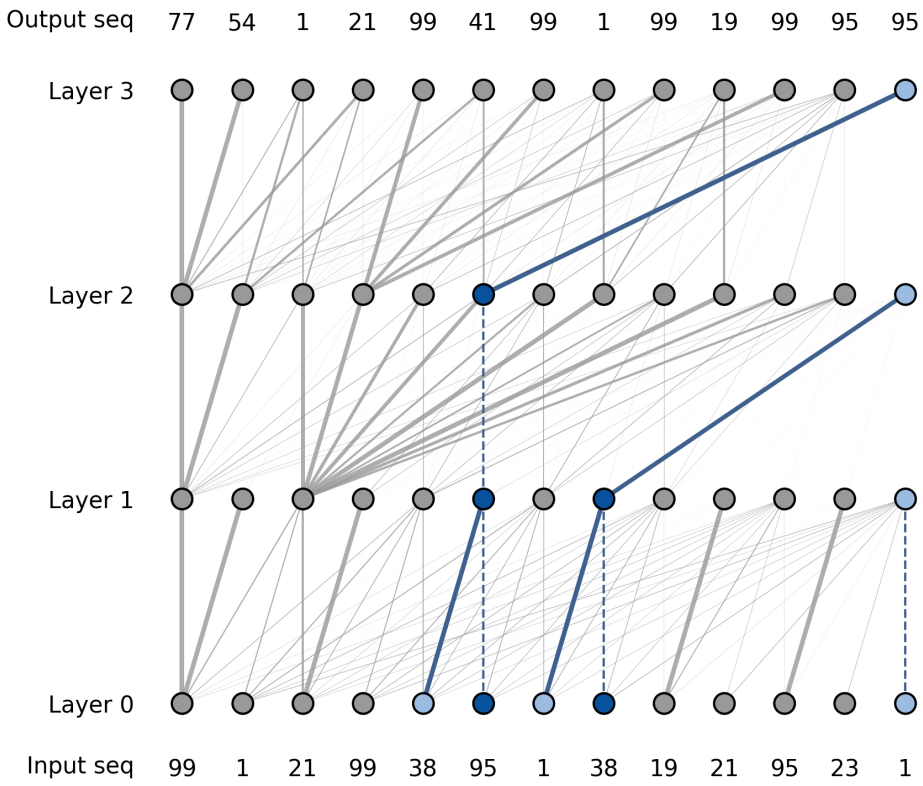


图 4.3: 完整的信息流示例。

```
            plt.scatter(j, 4*i, c='#999999', edgecolors='k',
                    s=80, linewidths=1, zorder=10)

# 依照attn的值，画每层的连接线，attn值越大，线越粗
for i, attn in enumerate(attn_list):
    for j in range(seq_len):
        for k in range(seq_len):
            if len(key_flows)== 0 or (i, j, k) in key_flows:
                plt.plot([j, k], [4*i, 4*(i+1)], c='#3e608d',
                    lw=attn[k, j]*2, zorder=3)
            else:
                plt.plot([j, k], [4*i, 4*(i+1)], c='#999999',
```

```
                    lw=attn[k, j]*2, zorder=1, alpha=0.8)

# 画residual的连接线
for i, j in res_flows:
    plt.plot([j, j], [4*i, 4*(i+1)], c='#3e608d', ls='--', lw
        =1, zorder=2)
```

4.6 作业

1. 运行配套的 jupyter 代码，训练一个小规模的 Transformer 模型来实现多步推理。
2. 更改一些超参数设置，观察初始化大小、weight decay 和模型维度对于准确率的影响。
3. 运行 1 阶推理数据集的代码，分析内在机制。

Chapter 5

参数初始值对推理能力影响实验手册

Jupyter 代码可以在 GitHub 找到[1]。

5.1 实验目的

本实验旨在研究在给定复合任务的非推断解进行训练的前提下，不同初始化大小的模型在未见复合任务上学习的映射类型差异。特别关注模型初始化大小对其在记忆和推理任务上的影响。

5.2 任务描述

在严格定义任务和实验设定前，我们先用一些通俗地描述，让读者对任务设定有更加直观、清晰的认知。我们有四个简单函数：$f_1 = x + 5$，$f_2 = x + 1$，$f_3 = x - 2$，$f_4 = x - 8$。它们可以形成 16 个复合函数。在训练过程中，将 $f_4(f_3)$ 从 $x - 10$ 修改为 $x - 6$，并排除 $f_3(f_4)$ 的数据。在测试时，当输入包含 $f_3(f_4)$ 时，我们希望研究 transformer 模型会预测什么结果? 以下是三种可能的结果：

(i) $x - 10$，与四个简单函数的模式一致;

(ii) $x - 6$，符合修改后的 $f_4(f_3)$ 映射;

(iii) 不规则，模型给出不一致的结果。

[1] https://github.com/xuzhiqin1990/understanding_dl/tree/main/code/book_jupyter_composition_task

5.3 实验设定

我们先对锚函数中的关键定义进行回顾，并对复合任务设定进行定义。图5.1展示实验设置以及未见锚对可能存在的机制。

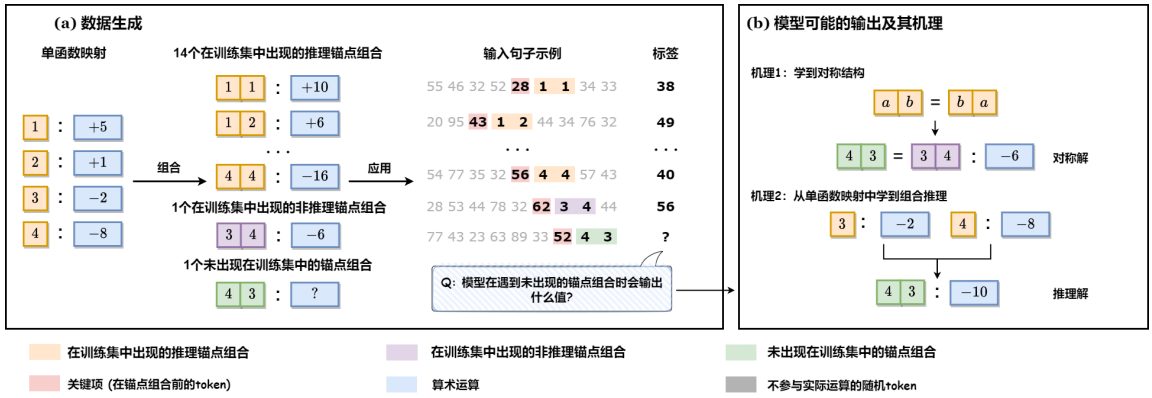


图 5.1: 实验设置以及未见锚对（4, 3）的可能解和机制。(a) 数据生成：左：单锚（即 1, 2, 3, 4）对应特定的算术运算。中：训练期间，16 个可能的锚对中有 14 个被分配了推断映射，一对（3, 4）被分配了非推断映射，剩下一对（4, 3）作为未见任务保留。右：输入序列包括一个锚对、一个在锚对之前的关键项目和一些与目标无关的噪声项目。问号表示未见锚对（4, 3）的输出，这取决于学习到的解。(b) 未见锚对（4, 3）的两种潜在机制：学习对称结构（机制 1）或组合推断的单锚映射（机制 2）。

5.3.1 双锚点复合函数

双锚点复合函数 $f(X) : \mathbb{R}^n \to \mathbb{R}$ 定义如下：

$$f(x_1, \ldots, x_n) = g\left(g(x_{i-1}; x_i); x_{i+1}\right), \quad \text{其中} \quad x_i, x_{i+1} \in A. \tag{5.1}$$

这里，输入序列 $X = (x_1, \ldots, x_n)$ 包含 n 个 token。锚集合 $A = \{a_1, a_2, \ldots, a_J\}$ 被指定，其中每个 token $a_k \in A$ 对应一个函数 $g(x; a_k)$。在每个 X 中，仅有一对连续元素属于 A，例如 $x_i, x_{i+1} \in A$。我们将紧接在锚对之前的 token 称为关键项。为了简化符号，我们将两锚复合函数记作 $f(x_{i-1}; x_i, x_{i+1})$，以强调锚对 (x_i, x_{i+1}) 和关键项 x_{i-1}。

在这项工作中，我们设置锚集合 $A = \{1, 2, 3, 4\}$。每个锚点对应一个特定的函数：

$$g(x; 1) = x + 5, \quad g(x; 2) = x + 1, \quad g(x; 3) = x - 2, \quad g(x; 4) = x - 8. \tag{5.2}$$

5.3.2 数据生成

在这项工作中，我们使用四个锚（即 1, 2, 3, 4）和从 20-99 中抽取的关键项来构建输入数据集。每个序列包括一个锚对、一个关键项（即紧接在锚对之前的项）和一些噪声项。噪声项与目标无关。四个锚两两形成了 16 个锚对，我们根据任务需求选择这些锚对的一个子集或全部来构建训练数据集。

默认情况下，目标是由两锚复合函数处理输入数据的输出，即下面定义的推断映射。我们根据关键项的值将训练数据和测试数据划分开来。

数据集及超参数设定如下，我们可以通过修改列表值修改训练集、测试集比例，训练集、测试集数据种类等:

Listing 5.1: 配置超参数及数据集配比设定

```
dname = ['13_xm0', '23_xm0', '43_xm0', ...]
dtrain = [0, 0, 0, 0, 0, 0, 1, 1, ...]
dshow = [0, 0, 1, 0, 0, 1, 0, 0, ...]
dpercent = [1, 1, 1, 1, 1, 1, 9, 9, ...]
parser = argparse.ArgumentParser(description="Pytorch distributed")
parser.add_argument('-data_size', '--data_size', type=int, default
    =900000)
parser.add_argument('-sl', '--seq_len', type=int, default=9, help='
    句子长度')
...
```

通过调用 data.py 文件中的 `get_data()` 函数来生成实验所需的数据集，示例如下:

```
datas = get_data(args)
print('datasize:', len(datas['44_xel']), 'example:', datas['44_xel'
    ][0])
print('datasize:', len(datas['11_xm0']), ' example:', datas['11_xm0
    '][0])
print('datasize:', len(datas['24_xel']), ' example:', datas['24_xel
    '][0])
```

输出如下:

```
datasize: 50625 example: [48, 23, 85, 4, 4, 80, 57, 86, 50, 69]
datasize: 5625  example: [41, 42, 62, 48, 1, 1, 24, 42, 88, 58]
```

```
datasize: 50625  example: [37, 44, 46, 87, 63, 30, 2, 4, 58, 23]
```

5.3.3 锚对的映射类型

对于锚对 (a_1, a_2)，我们给出三种关于其映射 $\mathcal{M}_{(a_1,a_2)}(\cdot)$ 的定义。

推断映射：锚对 (a_1, a_2) 的指定目标映射与两锚复合函数一致，即 $\mathcal{M}_{(a_1,a_2)}(x) = f(x; a_1, a_2)$。

非推断映射：锚对 (a_1, a_2) 的指定目标映射与两锚复合函数不一致，即 $\mathcal{M}_{(a_1,a_2)}(x) \neq f(x; a_1, a_2)$。

对称映射：锚对 (a_1, a_2) 的指定目标映射与其对称锚对 (a_2, a_1) 的映射一致，即 $\mathcal{M}_{(a_1,a_2)}(x) = \mathcal{M}_{(a_2,a_1)}(x)$。

如果模型倾向于输出与研究的锚对的推断（非推断，对称）映射相对应的映射，我们称该模型在该锚对上是一个推断（非推断，对称）解。

5.3.4 泛化

数据集的划分自然导致了以下两种泛化概念：

数据上的泛化。数据上的泛化依赖于测试集。在这个测试数据中，所有的锚对（即任务）在训练集中都有出现。

任务上的泛化。任务上的泛化依赖于未见过的锚，即在训练集中没有出现的锚，具有指定的目标映射。

5.4 实验步骤

5.4.1 初始化尺度设定

由于我们的目的是研究不同初始化尺度下模型表现，因此我们需要在每次实验前修改初始化尺度。特别的，我们需要修改下述代码行中 default 的赋值。

```
parser.add_argument('-sr', '--std_rate', type = float, default =
    0.1, help='标准差的幂次')
```

我们给出三个推荐赋值：0.1, 0.5, 0.8, 分别作为大初始化、中等初始化与小初始化尺度。

5.4.2 训练 Transformer 模型

在 train.py 文件中定义 `train()` 函数，用于模型的训练，代码如下（具体代码详见 train.py 文件）：

```
train(args, datas)
```

具体来说，我们先定义优化器、模型和损失函数等。

```
train_data_loader = get_train_data(args, datas)

args.num_batches = len(train_data_loader)

# 所有数据集对应的data_loader
data_loader_group = get_data_loader_group(args, datas)

device = torch.device("cuda:6" if torch.cuda.is_available() else "
    cpu")

my_logger = Log(f'{args.working_dir}/train_log.log')

# 模型与参数量
model = myGPT_specific(args, device).to(device)
if args.checkpoint != 'none':
    model.load_state_dict(torch.load(args.checkpoint, map_location=
        device))
my_logger.info(f'Total parameters: {sum(p.numel() for p in model.
    parameters())}')

criterion = nn.CrossEntropyLoss(ignore_index=0).to(device)

optimizer, scheduler = get_optimizer(model, args, **kwargs)
```

而后对 train_step() 函数进行循环：

```
def train_step(args, model, train_data_loader, optimizer, criterion
    , device, clip=1, scheduler=None):
    model.train()
    epoch_loss = 0
    total_samples = 0
```

```
    for i, (dec_inputs, dec_outputs) in enumerate(train_data_loader
        ):
        optimizer.zero_grad()
        dec_inputs, dec_outputs = dec_inputs.to(device),
            dec_outputs.to(device)
        outputs, _ = model(dec_inputs)


        batch_size = dec_inputs.size(0)  # 获取当前批次的实际大小
        total_samples += batch_size


        loss = criterion(outputs.view(batch_size, args.seq_len,
            args.vocab_size)[:,-1,:], dec_outputs[:,-1].view(-1))

        epoch_loss += loss.item() * batch_size  # 将损失乘以批次大
            小
        loss.backward()

        torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
        optimizer.step()

        if scheduler is not None:
            scheduler.step()

    return epoch_loss / total_samples  # 返回平均损失
```

在一次训练完成后，我们需要修改初始化尺度，直至三种尺度均训练完成后，再进行下述步骤。

5.4.3 不同初始化模型准确率测试

定义 `last_word_acc_reasoning()` 函数来评估不同初始化大小模型在未见任务和已见任务上的推断准确率，定义 `last_word_acc_symmetry()` 函数来评估不同初始化大小模型在

未见任务和已见任务上的推断准确率。代码如下：

```
def last_word_acc_reasoning(args, checkpoint, data_loader):
    device = torch.device( "cpu")
    model = myGPT_specific(args, device).to(device)

    model.load_state_dict(torch.load(checkpoint, map_location=
       device))
    model.eval()
    correct = 0
    total_samples = 0

    for i, (dec_inputs, dec_outputs) in enumerate(data_loader):
        dec_inputs, dec_outputs = dec_inputs.to(device),
           dec_outputs.to(device)
        outputs, _ = model(dec_inputs)

        batch_size = dec_inputs.size(0)  # 获取当前批次的实际大小
        total_samples += batch_size

        outputs = outputs.argmax(axis=-1).view(-1, args.seq_len)
        correct += (outputs[:, -1] == dec_outputs[:, -1]).sum().
           item()

    return correct / total_samples


def last_word_acc_symmetry(args, checkpoint, data_loader):
    device = torch.device( "cpu")
    model = myGPT_specific(args, device).to(device)

    model.load_state_dict(torch.load(checkpoint, map_location=
       device))
    model.eval()
    correct = 0
```

```
    total_samples = 0

    for i, (dec_inputs, dec_outputs) in enumerate(data_loader):
        dec_inputs, dec_outputs = dec_inputs.to(device),
            dec_outputs.to(device)
        outputs, _ = model(dec_inputs)

        batch_size = dec_inputs.size(0)  # 获取当前批次的实际大小
        total_samples += batch_size

        outputs = outputs.argmax(axis=-1).view(-1, args.seq_len)
        correct += (outputs[:, -1] == dec_outputs[:, -1]+4).sum().
            item()

    return correct / total_samples
```

基于上述函数，我们测试三个模型分别在训练过的复合任务上（分布内）泛化性及未训练过的复合任务上（分布外）泛化性。

我们分别以 data_loader_group['12_xel'], data_loader_group['43_xel'] 代表分布内数据和分布外数据。特别的，对于分布外数据，我们分别研究以对称解为真解和以推断解为真解的准确率。我们使用下述代码进行三种模型泛化性测试。

```
data_loader_group = get_data_loader_group(args, datas)
model_large_init='./result_0.1/model/model_209.pt'
model_middle_init='./result_0.5/model/model_20.pt'
model_small_init='./result_0.8/model/model_209.pt'

small_init_unseen_acc=last_word_acc_reasoning(args,
    model_small_init, data_loader_group['43_xel'])
middle_init_unseen_acc=last_word_acc_reasoning(args,
    model_middle_init, data_loader_group['43_xel'])
large_init_unseen_acc=last_word_acc_reasoning(args,
    model_large_init, data_loader_group['43_xel'])
small_init_unseen_acc_symm=last_word_acc_symmetry(args,
    model_small_init, data_loader_group['43_xel'])
```

```
middle_init_unseen_acc_symm=last_word_acc_symmetry(args,
    model_middle_init, data_loader_group['43_xel'])
large_init_unseen_acc_symm=last_word_acc_symmetry(args,
    model_large_init, data_loader_group['43_xel'])
small_init_seen_acc=last_word_acc_reasoning(args, model_small_init,
     data_loader_group['12_xel'])
middle_init_seen_acc=last_word_acc_reasoning(args,
    model_middle_init, data_loader_group['12_xel'])
large_init_seen_acc=last_word_acc_reasoning(args, model_large_init,
     data_loader_group['12_xel'])
print('small_init_unseen_rsn_acc:', small_init_unseen_acc)
print('middle_init_unseen_rsn_acc:', middle_init_unseen_acc)
print('large_init_unseen_rsn_acc:', large_init_unseen_acc)
print('small_init_unseen_sym_acc:', small_init_unseen_acc_symm)
print('middle_init_unseen_sym_acc:', middle_init_unseen_acc_symm)
print('large_init_unseen_sym_acc:', large_init_unseen_acc_symm)
print('small_init_seen_acc:', small_init_seen_acc)
print('middle_init_seen_acc:', middle_init_seen_acc)
print('large_init_seen_acc:', large_init_seen_acc)
```

最后我们基于不同初始化尺度模型在不同数据上的表现进行柱状图绘制:

```
acc_list=[[large_init_seen_acc, middle_init_seen_acc,
    small_init_seen_acc], [large_init_unseen_acc_symm,
    middle_init_unseen_acc_symm, small_init_unseen_acc_symm], [
    large_init_unseen_acc, middle_init_unseen_acc,
    small_init_unseen_acc]]


fig = plt.figure(figsize=(12, 8))
format_settings(left=0.12, right=0.94, bottom=0.15, top=0.95,
    major_tick_len=10, fs=24, lw=6, ms=12.5, axlw=2.5)
plt.rcParams['xtick.major.pad'] = 10
plt.rcParams['ytick.major.pad'] = 5
```

```
plt.rcParams['axes.spines.top'] = False
plt.rcParams['axes.spines.right'] = False

ax = plt.gca()

labels = ['seen anchors', 'unseen anchors symmetry', 'unseen
   anchors reasoning']
color_list = [(218/255, 240/255, 178/255), (146/255, 212/255,
   185/255), (30/255, 128/255, 184/255)]
width = 0.15
for i, data in enumerate(acc_list):
    ax.bar(np.arange(len(data))*0.7-0.7*width+i*width, data, width=
       width,
            label=labels[i], color=color_list[i],edgecolor='black',
                linewidth=2)
ax.yaxis.grid(True, linestyle='--', linewidth=0.7, color='gray',
   alpha=0.7)
ax.set_axisbelow(True)
ax.set_xticks([0.05,0.75,1.45])
ax.set_xticklabels(['large init','middle init','small init'])

ax.set_xlabel('init scale', labelpad=10)
ax.set_ylabel('accuracy')

ax.legend(loc=(0, 1.02), frameon=False, ncol=2)
```

柱状图如下图所示，可以看到大初始化模型在分布内数据上泛化性差，中等初始化与小初始化模型均在分布内数据上有好的泛化能力。然而对于中等初始化模型，他倾向于学习对称解，即记忆复合映射，而对于小初始化模型，他倾向于学习推断解，即发现单映射。

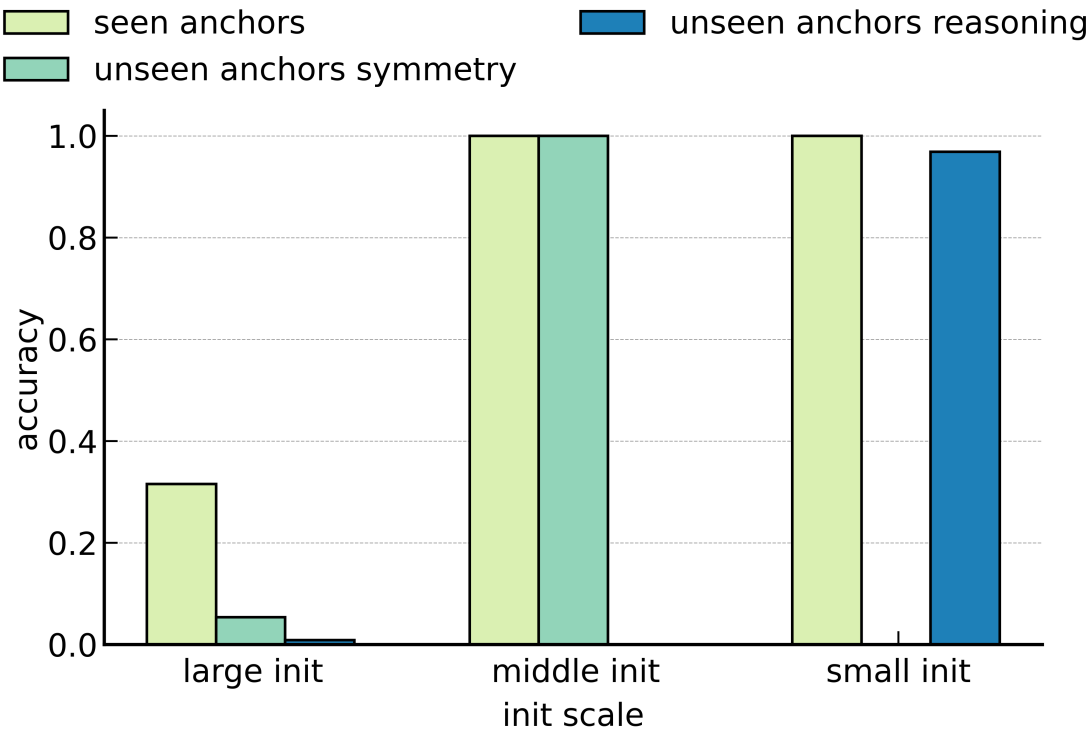


图 5.2: 不同初始化尺度模型在不同数据上的泛化性柱状图

参考文献