深 圳 大 学 实 验 报 告

课程名称：自动控制原理

实验名称：基于 BP 神经网络的二阶离散系统 PID 参数自整定

学　　院：机电与控制工程学院

专　　业：自动化

指导教师：崔玉康

姓名：　　学号：2019110000　班级：自动化

实验时间：2022 年　6 月　　日

提交时间：2022 年　6 月　　日

教务部制

摘要：比例积分微分 (PID) 控制器结构简单、模型误差鲁棒性强、操作简单等优点，在工程实践中得到广泛应用。目前，PID 控制器种类繁多。对于现有的 PID 控制器，将智能、自适应、自校正等策略扩展到变速器 PID。90%的工业控制过程都使用了 PID 控制器和相应的改进控制器。

一、 相关研究发展与现状

Minorsky [1] 1922 年提出了一种基于输出反馈设计 PID 控制器的方法。到上世纪 40 年代，PID 控制器作为调节器在工程实践中的应用最为广泛。自 PID 控制器出现以来，已经过去了将近 70 年。PID 控制器因其算法简单、稳定性高、鲁棒性强、工作可靠、调节方便等优点，已成为工业控制的领先技术之一。PID 调节是技术最成熟、应用最广泛的连续系统。如果我们不完全了解一个系统及其被控对象，或者我们不能通过有效的测量方法确定系统参数，PID 控制将是最合适的手段。PID 控制算法的使用在很多控制领域都比较令人满意。微机、单片机、DSP 实现的数字 PID 控制算法，由于其软件系统的灵活性，得到了进一步的修正和改进。PID 控制算法有很多种，在不同的应用中要求有所不同。

随着工业的发展，对象变得越来越复杂。特别是对于大时滞和时变非线性系统，有些参数是未知的，或者变化缓慢，或者有时间延迟，或者有随机扰动，或者无法得到相对准确的数字模型。同时，随着人们对质量控制的要求越来越严格，常规 PID 控制的不足之处也逐渐暴露出来。传统的 PID 控制对于时变对象和非线性系统很少有效。因此，常规 PID 控制受到很大限制。鉴于此，对其进行了不同方面的改进，主要介绍如下。一方面，常规 PID 在结构上有所改进；另一方面，模糊控制、神经网络控制、和专家控制是现有智能控制中最活跃的。一旦与常规 PID 控制结合使用，就可以取长补短，发挥各自的优势，构成智能 PID 控制。本文主要总结了 PID 算法的发展和分类。

目前，PID 控制器种类繁多，如 PID 控制、预测 PID 控制、自适应 PID 控制、模糊 PID 控制、神经网络 PID 控制、专家智能 PID 控制、基于遗传算法的 PID 控制、基于蚁群算法的 PID 控制。

1.1 PID 控制的分类

1.1.1 预测 PID 控制

史密斯预测器（Smith predictor）是由 Otto JM Smith（英语：Otto JM Smith）于 1957 年发明的预测型控制器，可以适用于有纯时间延迟的系统。

在他的算法中，假设过去的输入变化在每一步都是相同的，并且等于当前的输入变化。在实践中，这些关系在系统的动态响应过程中总是不成立的。如果系统没有时延或短时延，这种近似的影响可以忽略不计，但随着时延步数的增加，对系统鲁棒性的影响无疑会逐渐加剧。因此，Smith 预测器被集成到系统中以补偿时滞，从而将延迟的调节变量提前报告给调节器。然后，调节器将提前移动以消除系统延时的影响，减少超调，提高系统稳定性，加快调节速度，提高大延时系统的有效性[4]。

原则上，PID 控制器的输出通过补偿部分反馈到 PID 的输入端，以补偿被控对象的滞后。在工程实践中，Smith 预测器被反馈给 PID 调节器，以克服被控对象的纯时间延迟，如图 2 所示。

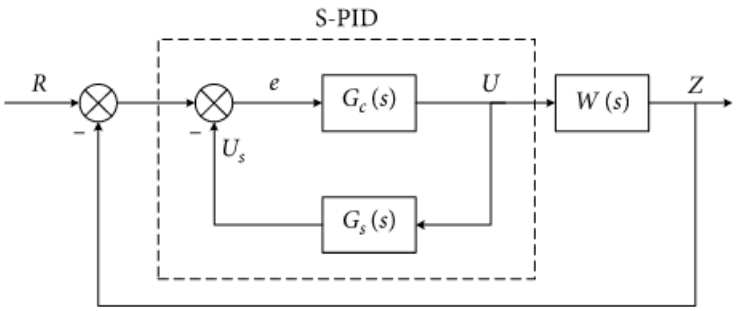


图 2

3.2. 自适应 PID 控制

在工业控制的实际过程中，许多受控机构是高度非线性和时变的，具有纯时延。受某些因素的影响，过程参数可能会发生变化，因此自适应 PID 控制是解决这些问题的有效方法。自适应 PID 控制器具有自适应控制和常规 PID 控制器的优点。除了有助于自动识别探索的过程参数、自动调整控制器参数和适应受控过程参数的变化外，它们还像传统的 PID 控制器一样结构简单、高度稳健且相当可靠。现场工作人员和设计工程师熟悉自适应 PID 控制器。凭借这些优势，自适应 PID 控制器已经发展成为相对理想的过程控制自动化设备 [6]。

它们被分为两个主要类别。PID 控制器，它是基于对被控过程参数的识别，统称为自适应 PID 过程参数，统称为自适应 PID 控制器。它们的参数设计取决于被控过程模型的参数估计。另一种自适应 PID 控制器的另一种类型是基于被控过程的一些特征参数，如临界振荡增益和临界振荡。

非参数化的自适应 PID 控制器的参数是直接根据过程的特征参数来调整。参数化的自适应 PID 控制[2]包括：

(i) 自适应 PID 极点定位控制：

自适应 PID 极点定位控制算法是由 Wellstead 等人在 1979 年首次提出的。随后由 Astrom 和 Wittenmark 随后由 Astrom 和 Wittenmark[7]改进和深化。

(ii) 基于取消原则的自适应 PID 控制。

原则的自适应 PID 控制。Wittenmark 和 Astrom 首先提出了基于取消原则的参数化的自适应 PID 控制算法。

(ii) 基于二次性能指数的自适应 PID 控制。

性能指数的自适应 PID 控制：在非参数自适应 PID 方面也取得了广泛的发展。使用人工智能优化参数。

3.3. 模糊 PID 控制

1965 年，控制论专家 Zadeh [8] 发展了模糊集理论作为描述、研究和处理模糊现象的新工具。对于模糊控制，采用模糊集理论。特别是在一些具有大时延的复杂时变非线性系统中，不可能得到系统、精确的数学模型。对于模糊控制，不需要被控对象的精确数学模型。与 PID 控制器一样，这些控制器的控制精度很高。此外，控制器灵活自适应，对复杂控制系统和高精度伺服系统的控制非常有效。在过去的几年里，他们在控制领域非常活跃 [9]。

它们的基本原理如下。在传统 PID 控制算法的基础上，进行 PID 参数的自整定。设定模糊控制规律，通过控制参数误差和误差变化，对控制参数进行自适应整定，以满足不同控制周期的参数要求。对于模糊 PID 控制算法，根据模糊集理论，建立、和误差变化之间的函数关系

它们的基本原理如下。基于传统的 PID 控制算法，对 PID 参数进行自我调整。模糊控制的规律被设定为自适应调谐

通过控制参数误差 E 和误差 E_c 的变化，以满足不同时期对 E 和 E_c 参数的要求。在不同的控制时期，对 E 和 E，参数的要求。对于模糊 PID 控制算法中，K_P, K_I, K_d 之间的函数关系和误差变化 E_C 之间的功能关系是根据模糊理论建立的。

根据模糊集的理论

$$K_P = f_1(E, E_C)$$

$$K_I = f_2(E, E_C)$$

$$K_D = f_3(E, E_C)$$

在 $K_P, K_I,$ 和K_d的自调过程中，E 和 E_C 的数值被确定。控制参数的自根据模糊控制的规律进行在线调整，以满足不同控制期的控制要求。以满足不同控制时期的控制要求，从而使被控对象的控制系统 这样，被控对象的控制系统就能保持高度的 动态和静态。目前，已经有一些目前有一些常见的模糊 PID 控制器，如模糊 PI 控制器。模糊 PD 控制器、模糊 $\mathrm{PI}+D$控制器、模糊 $PD+I$ 控制器、模糊 $(\mathrm{P}+\mathrm{D})^2$ 控制器和模糊 PID 控制器。图 3 显示了基于模糊控制规律的 PID 参数的在线自调。

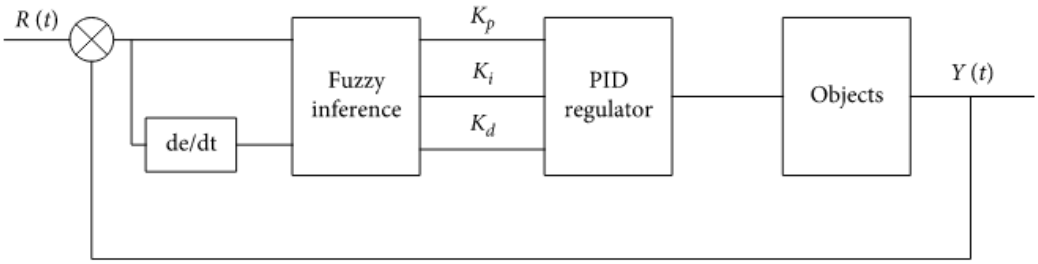


图 3

3.4. 神经网络 PID 控制

Windrow [12]提出的自适应神经元结构简单，无需精确建模受控对象。基于无神经模型控制，许多学者提出无神经模型自适应 PID 控制方法，识别神经网络的输入信号，并结合 PID 控制的长处设计在线修正算法，取得了一定的成果。研究控制系统中的声音动态和静态特性。

RBF（径向基函数）神经网络 [16]是具有三层的前向网络，即输入层、隐藏层和输出层。RBF 神经网络结构如图 4 所示。它们的输入经过处理、加权并发送到输出层的神经元。只有一个神经元控制输出层的输出。

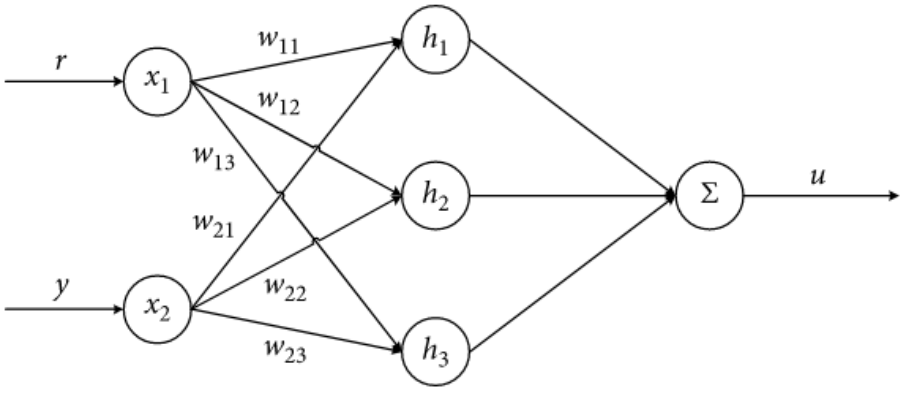


图 4

为在 PID 控制中获得理想的控制效果，应适当调节比例、积分、微分在控制中的作用，使其相互协调、相互制约。这些关系是不必要的简单线性组合，并且可以在具有无数变化的非线性组合中确定最佳关系。通过研究系统性能，可以进行最优组合的 PID 控制，从而找出一定最优控制规律下的 P、I、D 。基于 BP 神经网络的 PID 控制系统架构如图 5 所示。控制器由两部分组成：

(i)经典 PID 控制器：对受控对象进行闭环控制，在线调整三个参数。

(ii)神经网络：根据系统的运行状态调节 PID 控制器的参数，以期优化某些性能指标。

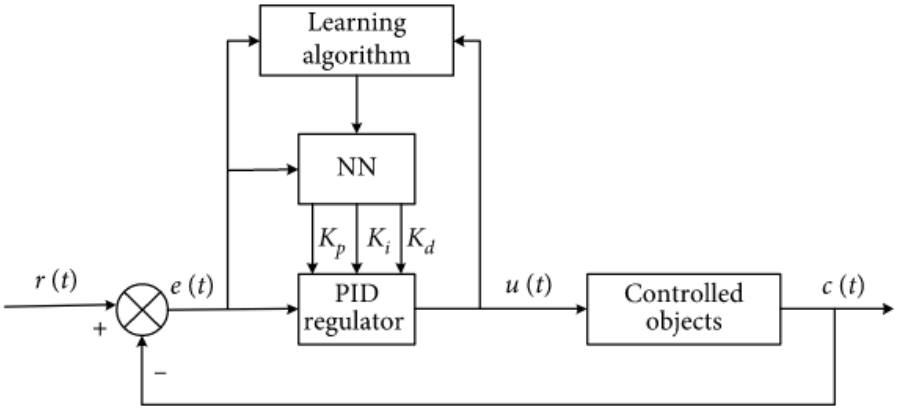


图 5

即使输出层的神经元的输出状态 对应于 PID 控制器的三个可调参数K_P, K_I, K_d。PID 控制器的输出状态，控制器依靠神经网络按照一定的优化规则调整权重，使得系统处于稳定状态。

3.5. 基于遗传算法的 PID 控制

遗传算法[17]，简称 GAs，是美国密歇根大学 Holland 教授在上世纪 60 年代首先提出的一种高效、并行、全局最优的搜索方法。它们是在模拟自然环境下生物的遗传和进化过程中形成的自适应全局最优概率搜索算法。他们的基本思想如下。为了转换要解决的问题，由个人组成的群体操作一组遗传算子，并在寻找最优解之前重复生成-评估-选择-操作的过程。基于遗传算法的 PID 参数优化方法有助于简化解析计算[18]。

遗传算法是通过模拟自然进化来自然选择和求解最优解的方法。对于基于遗传算法的 PID 控制，首先通过遗传算法将实际问题转化为遗传密码。在实践中，使用了二进制编码、浮点编码和参数编码等编码方法。随后生成初始种群，搜索后进行 PID 调节。示意图如图 6 所示。

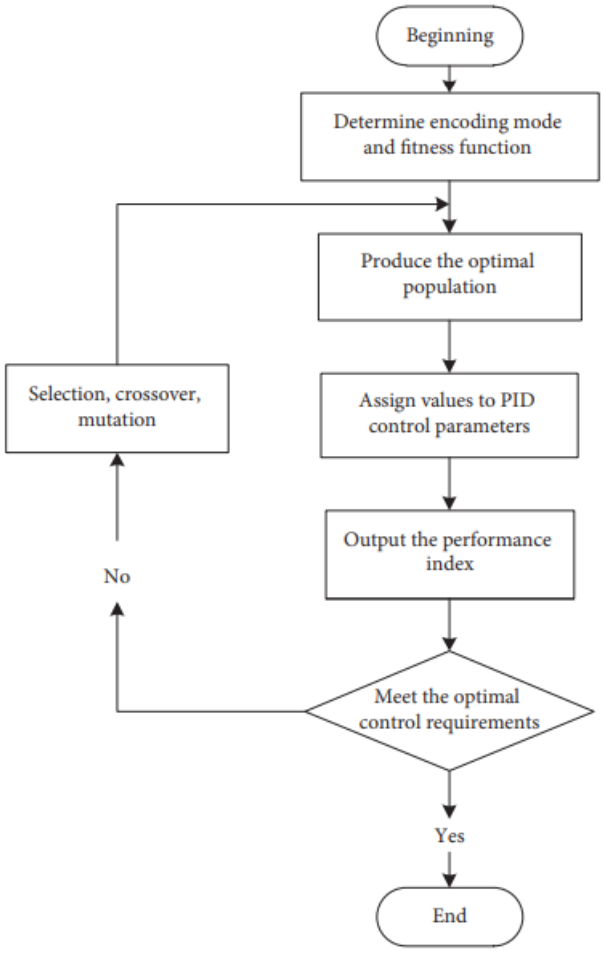


图 6

3.6. 基于蚁群算法的 PID 控制

1990 年代初期，意大利学者 Dorigo Macro 等通过模拟蚂蚁在自然界中共同寻路的行为，提出了蚁群算法。蚁群算法作为一种新的模拟进化算法，是一种基于种群的进化模拟算法，其灵感来自于对自然界中真实蚁群行为的研究和随机搜索算法。蚂蚁通过一种称为信息素的材料传递信息。在它们的移动过程中，蚂蚁将这种物质留在它们经过的路径上。此外，它们可以在移动过程中感知到这种物质，从而引导它们的移动方向。因此，由众多蚂蚁组成的群体的集体行为反映了一种正反馈。蚂蚁在特定路径上走的越多，后来者选择路径的可能性就越大。在这种情况下，信息素的强度增强。这种选择过程被称为蚂蚁的自催化，其原理是一种正反馈机制，因此蚂蚁系统也被称为增强学习系统。

图 7 示意性地描述了 PID 控制参数的蚁群算法优化。一组三个数列，即 K_P, K_I, K_d 被认为是一组三个城市。人工蚂蚁从 S 出发，分别经过集合中的一个城市。最后到达 D 点并符合准则函数，从而找到最优路径。蚁群算法搜索的 PID 控制参数优化路径反映了系统具有最优性能指标，该指标由蚁群系统中三个控制参数的节点值体现。信息素在蚂蚁经过的节点上释放，其浓度根据标准函数而不

是路径长度而变化。准则函数应包含有关蚂蚁已通过的节点的信息以及系统的当前性能指标。

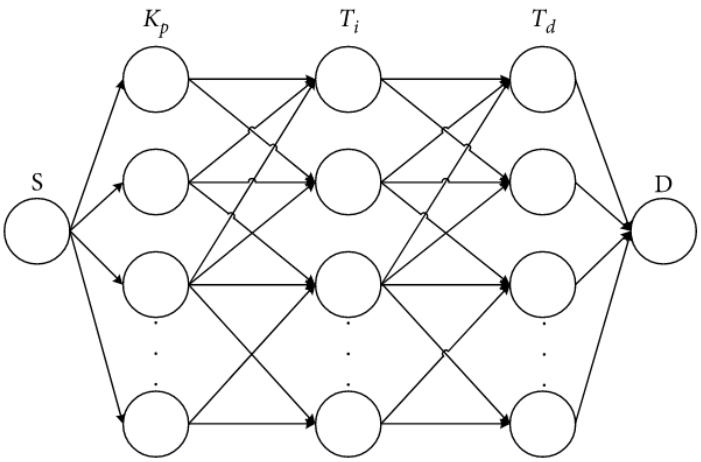


图 7

3.7. 专家级智能 PID 控制

随着人工智能的发展，出现了多种形式的专家控制系统。自然地，人们想到了根据专家经验开发 PID 参数。1984 年 Foxboro（美国）推出的 EXACT 专家自整定控制器是最经典的，该公司将专家系统应用于 PID 控制器。

专家控制与常规 PID 控制相结合实现的自整定、自学等功能可用于描述复杂系统的特性。通过学习和自组织可以制定和识别相应的控制策略。学者们研究了专家自整定 PID 控制器的设计方法和应用[29]。针对普通专家自整定 PID 控制器的缺陷，他们另外开发了智能自整定控制器，并提出使用阶梯信号作为系统输入。因此，系统不需要在参数训练过程中频繁启动。另外，给定信号的阶梯数可根据实际系统变化灵活确定，以满足某些特殊场合的控制要求。由于对象模型的自调整能力强，其结构和参数在较大范围内变化。

专家系统包括两个要素：

(i)知识库：存储预先汇总并以一定格式表示的特殊领域的知识条目。

(ii)推理机制：来自知识库的条目用于通过解决专家问题的类似方法进行推理、判断和决策。

针对专家智能 PID 控制的原理，将测量的特征参数与预定的特征参数进行比较，并将其偏差导入专家系统，分析调制器参数的必要修正以消除特征量并将其导入常规 PID 调制器，从而调制器的正确参数。同时，调制器根据系统误差和调谐参数执行操作。输出受系统误差和广义对象控制的控制信号，直到响应曲线上的特征参数在受控过程中达到预期为止。专家智能 PID 示意图如图 8 所示。

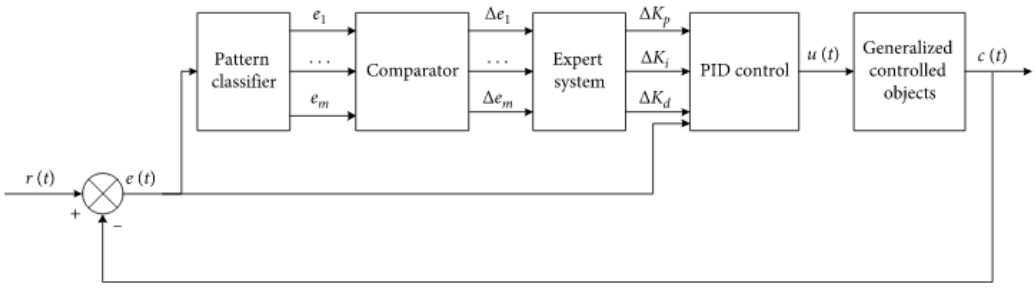


图 8

二、 PID 的基本原理

2.1 基本组成

PID 控制是反馈系统中偏差的比例 (P)、积分 (I) 和微分 (D) 的线性组合。这三个基本控制律各有特点[2]。

2.1.1 比例（P）控制

比例控制器在控制输入信号 $e(t)$ 的变化时仅改变其信号幅度而不影响其相位。比例控制增加了系统的开环增益。这部分控制占主导地位。

2.1.2 微分 (D) 控制

微分控制器确定输入信号的微分，而微分反映系统的变化率，因此微分控制是一种领先的预测调节模式，可以预测系统变化，增加系统阻尼并增强相位裕度，从而提高系统性能。

2.1.3 积分 (I) 控制

积分是一种加性效应，记录了系统变化的历史，因此积分控制体现了历史对当前系统的影响。一般不单独采用积分控制，而是与 PD 控制相结合，如图 1 所示。

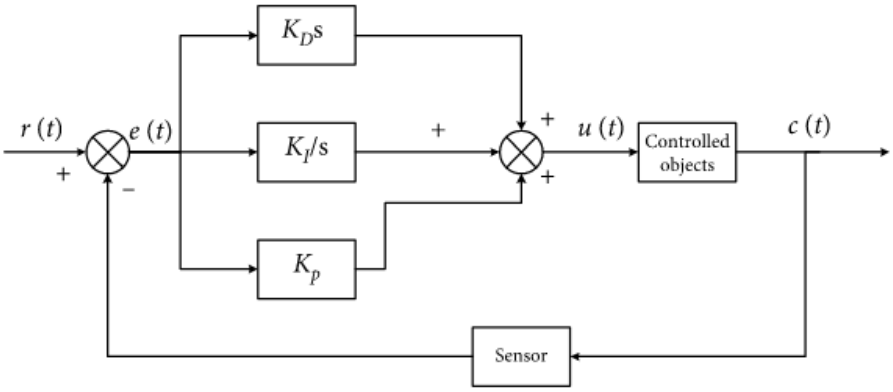


图 1

3.8. 其他 PID 控制

随着计算机技术和智能机器的快速发展，人们开始使 PID 控制算法智能化，以进一步提高控制[30-33]。因此，开发了一系列新的改进的 PID 算法，包括模糊 PID 控制器、智能 PID 控制器和神经网络 PID 控制器。随着这些算法的出现，PID 算法在功能上变得更加完善，在控制工业过程中发挥着越来越重要的作用。他们的应用将得到进一步推广。此外，还有一些其他的 PID 控制算法，如自整定 PID 控制、非线性 PID 控制以及基于 GA 和 BP 神经网络相结合的 PID 控制。

2.2. PID 控制规律

PID 控制律的基本输入/输出关系可以用微分方程表示如下：

$$v(t) = K_p\left(e(t) + \frac{1}{T_{\mathrm{i}}}\int_0^t e(t)\mathrm{d}\,t + T_d\frac{\mathrm{d}e(t)}{\mathrm{d}t}\right)$$

其中 $e(t)$ 是控制器的输入偏置；是比例控制的增益； $K_pT_i1/2$ 是积分时间常数；T_d 是微分时间常数。

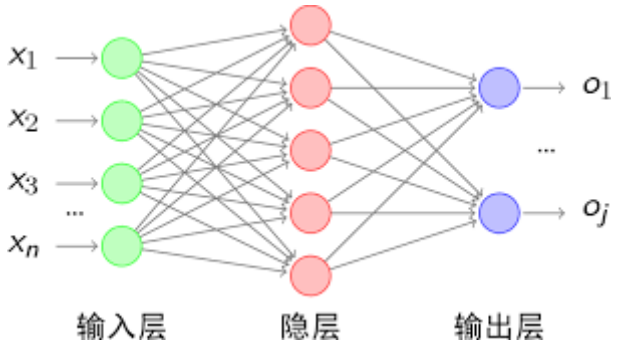
对应的传递函数如下：

$$\begin{aligned} G(s) &= K_P\left(1 + \frac{1}{T_i s} + T_d s\right) \\ &= \frac{K_p}{K_i}\cdot\frac{T_iT_ds + T_is + 1}{s}. \end{aligned}$$

三、 实验设计与流程图

3.1 反向传播算法

反向传播算法（BP 算法）主要由两个阶段组成：激励传播与权重更新。



第 1 阶段：激励传播

每次迭代中的传播环节包含两步：

1. （前向传播阶段）将训练输入送入网络以获得预测结果；
2. （反向传播阶段）对预测结果同训练目标求差(损失函数)。

第 2 阶段：权重更新

对于每个突触上的权重，按照以下步骤进行更新：

1. 将输入激励和响应误差相乘，从而获得权重的梯度；
2. 将这个梯度乘上一个比例并取反后加到权重上。

这个比例（百分比）将会影响到训练过程的速度和效果，因此成为“训练因子”。梯度的方向指明了误差扩大的方向，因此在更新权重的时候需要对其取反，从而减小权重引起的误差。

第 1 和第 2 阶段可以反复循环迭代，直到网络对输入的响应达到满意的预定的目标范围为止。

本实验中，神经网络输出维度为 3，分别对应 Kp Ki Kd 三个参数

输入可以自由选择变量，隐藏层的层数和维度都会影响神经网络学习结果。

神经网络目的是找到一组映射关系，通过优化不断减小模型（映射函数）与实际情况的误差

3.2 增量式 PID

所谓增量式 PID 是指数字控制器的输出只是控制量的增量△u 当执行机构需要的控制量是增量，而不是位置量的绝对数值时，可以使用增量式 PID 控制算法进行控制。

增量型 PID 公式

$$\Delta U(t) = K_P(\text{err}\,(t) - \text{err}\,(t-1)) + K_I\text{err}\,(t) + K_D(\text{err}\,(t) - 2\text{err}\,(t-1) + \text{err}\,(t-2))$$

K_p：比例系数 K_I：积分系数 K_D：微分系数

增量式：增量式算法需要保存历史偏差，$e(k-1)$，$e(k-2)$，即在第 k 次控制周期时，需要使 用第 $k-1$ 和第 $k-2$ 次控制所输入的偏差，最终计算得到 $\Delta u(k)$，此时，这还不是我们所需要 的 PID 输出量；所以需要进行累加

$$u(k) = u(k-1) + \Delta u(k)$$

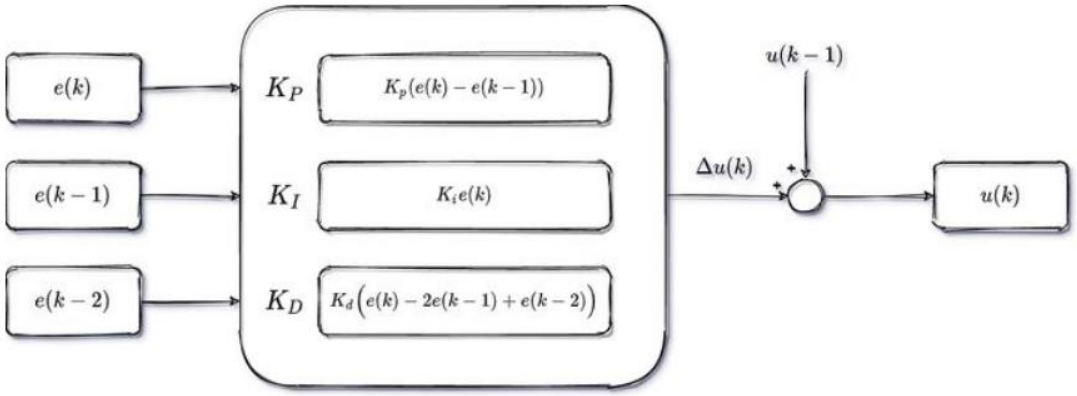
不难发现第一次控制周期时，即$k = 1$时；

$$u(k) = u(0) + \Delta u(k)$$

由以上公式我们可以推导出下式

$$u(k-1) = \sum_{i=1}^{k-1} \Delta u(i)$$

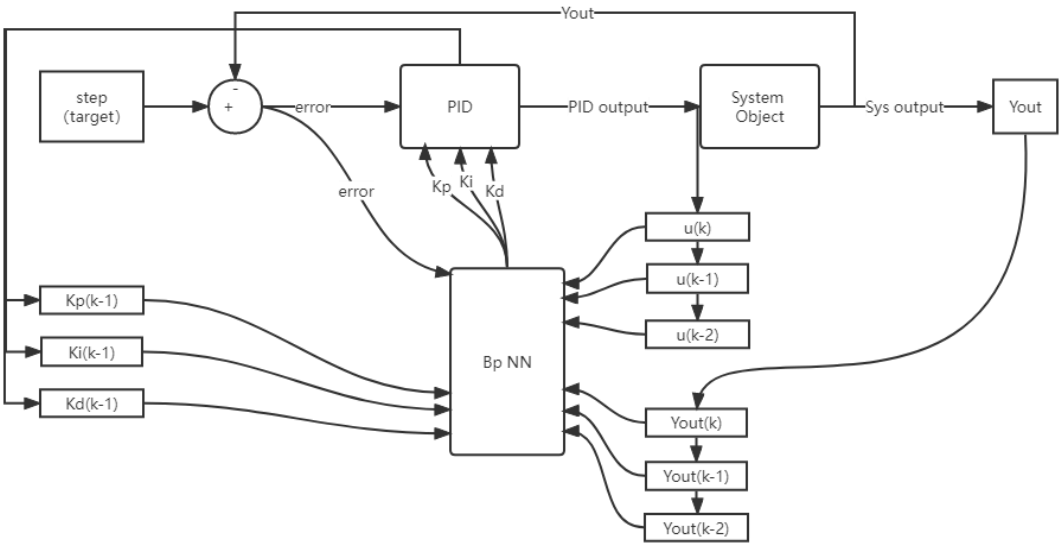
可见增量式算法，就是所计算出的 PID 增量的历史累加和；



3.3 代码实现

该实验中使用了 python + pytorch 框架进行构建，使用 matplotlib 库进行画图。没有使用 matlab

系统架构：

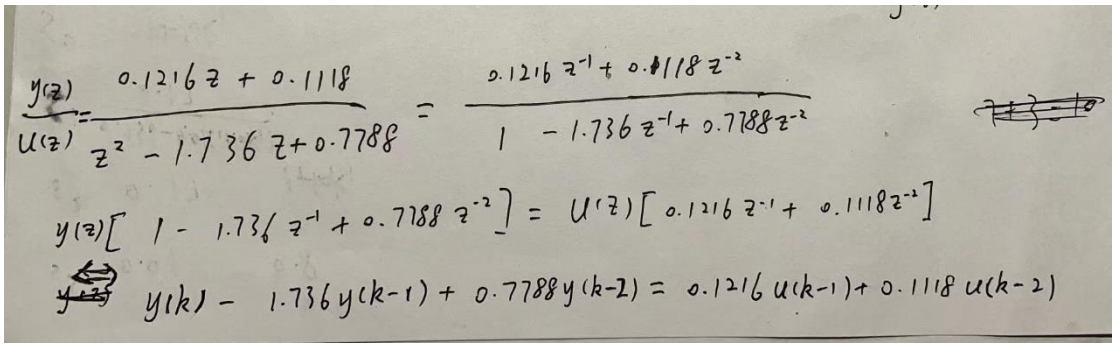


3.3.1 系统传函

实验所使用的系统传函：

$$\frac{0.1216z + 0.1118}{z^2 - 1.736z + 0.7788}$$

将 z 函数转化为差分方程



$$y(k) = 1.736y(k-1) - 0.7788y(k-2) + 0.1216u(k-1) + 0.118(k-2)$$

代码：

```
# 系统差分方程的类
class sysFun(object):
    def __init__(self) -> None:
        self.a = 1.736
        self.b = -0.7788
        self.c = 0.1216
        self.d = 0.1118

        self.Y = 0.0
        self.Y_1 = 0.0
        self.Y_2 = 0.0

        self.U = 0.0
        self.U_1 = 0.0
        self.U_2 = 0.0

    def forward(self, u):
        self.Y_2 = self.Y_1
        self.Y_1 = self.Y

        self.U_2 = self.U_1
        self.U_1 = self.U
        self.U = u

        self.Y = self.a * self.Y_1 + self.b * self.Y_2 + self.c
* self.U_1 + self.d * self.U_2

        return self.Y

    def setSys(self,a, b, c, d):
        self.a = a
        self.b = b
        self.c = c
        self.d = d
        return self

    def getState(self, target):
        return np.array([self.Y, self.Y_1, self.Y_2, self.U,
self.U_1, self.U_2, target-self.Y])
```

3.3.2 增量式 PID

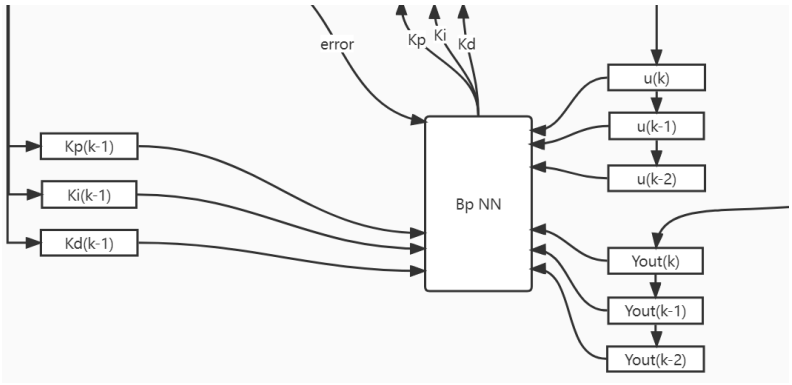
代码：

```
#增量式PID 系统
class DeltaPID(object):
    def __init__(self, target, P ,I ,D):
        self.Kp = P
        self.Ki = I
        self.Kd = D

        self.target = target # The target

        self.error = 0.0       # t time
        self._p_error = 0.0    # t-1 time
        self._pp_error = 0.0   # t-2 time

        self.pid_output = 0.0  # PID 输出

    def setPID(self, P ,I ,D ):
        self.Kp = P            # 更新 P
        self.Ki = I            # 更新 I
        self.Kd = D            # 更新 D

        print("setPID:",self.Kp,self.Ki,self.Kd)
        return self

    def setTarget(self, target):
        self.target = target   # 设置PID 环节输入，即跟随目标
        return self

    def getPID(self):
        return np.array([self.Kp, self.Ki, self.Kd])
        # 返回 PID 系数

    def getErr(self):
        return np.array([self.error, self._p_error,
self._pp_error])
        # 返回 误差

    def calcalate_deltaU(self, cur_val):
        error = self.target - cur_val
        self.error = error
        p_change = self.Kp * (error - self._p_error)
```

```
        i_change = self.Ki * error
        d_change = self.Kd * (error - 2 * self._p_error +
self._pp_error)
        delta_output = p_change + i_change + d_change
        # 计算增量式PID 输出
        self._p_error = error
        # 保持 上一次误差
        self._pp_error = self._p_error
        # 保持 上上次误差
        return delta_output

    def calculate_pid_output(self, cur_val):
        self.pid_output += self.calcalate_deltaU(cur_val)
        # 对增量输出进行累加
        return self.pid_output  # pid out -> sysFun in
        # 返回 PID 直接输出
```

3.3.2 Bp 神经网络

神经网络输入输出图：



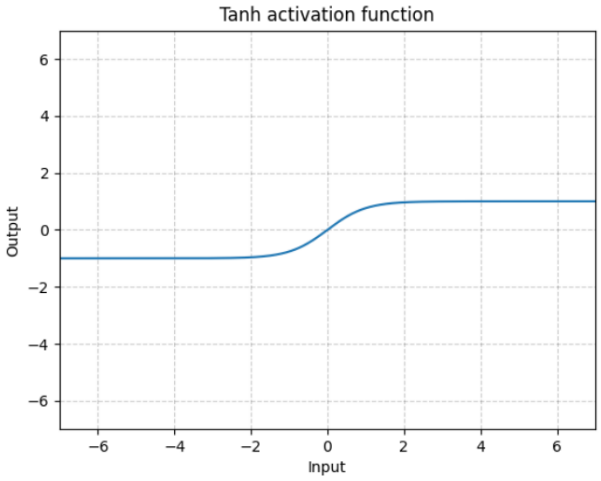
选取了被控对象的输入$u(k)$，上个时间点的输入$u(k-1)$上上个时间点的输入$u(k-2)$，前一周期的 PID 参数，系统输出$Yout(k)$ 上一周期$Yout(k-1)$ 上上周期 $Yout(k-2)$ 以及系统输出的误差$e(t)$ $e(t-1)$ $e(t-2)$作为神经网络的输入。

考虑到系统具有一定的线性关系，设置 2 个隐藏层。维度分别为 16 和 6

激活函数：

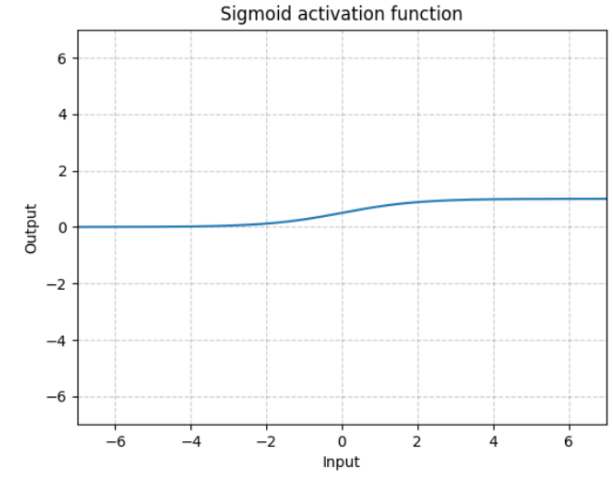
1. Tanh

$$\text{Tanh}(x) = \tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$



2. Sigmoid

$$\text{Sigmoid}(x) = \sigma(x) = \frac{1}{1 + \exp(-x)}$$



amples:

神经网络结构：

```
        # 设置网络结构
        self.linear_sigmoid_stack = nn.Sequential(
            nn.Linear(input_dim, hidden1_dim),
            nn.Tanh(), # 激活函数
            nn.Linear(hidden1_dim, hidden2_dim), # 线性层
            nn.Tanh(), # 激活函数
            nn.Linear(hidden2_dim, output_dim), # 线性层
            nn.Sigmoid() # 激活函数
        )
```

在输入层与隐藏层之间使用 Tanh 激活函数，在网络输出使用 Sigmoid 激活函数。使用 Sigmoid 激活函数可以将网络输出限制在（0，1）的范围内。因为 PID 参数一般来说不会是负数

代码：

```
target = 1.0

config =  {
            'input_dim':13,
            'hidden1_dim':16,
            'hidden2_dim':6,
            'output_dim':3,
            'p_init':0.001,
            'i_init':0.0,
            'd_init':0.0,
            'learning_rate':0.002,
            'pidK': target * 0.03, # 0.03
        }
```

```
class BpNN(nn.Module):
    def __init__(self, input_dim=7, hidden1_dim=7,
hidden2_dim=7, output_dim=3):
        super(BpNN, self).__init__()
        self.hidden1_dim = hidden1_dim
        self.hidden2_dim = hidden2_dim
        # 设置网络结构
        self.linear_sigmoid_stack = nn.Sequential(
            nn.Linear(input_dim, hidden1_dim),
            nn.Tanh(), # 激活函数
```

```
            nn.Linear(hidden1_dim, hidden2_dim), # 线性层
            nn.Tanh(), # 激活函数
            nn.Linear(hidden2_dim, output_dim), # 线性层
            nn.Sigmoid() # 激活函数
        )
        self.l1 =  nn.Linear(input_dim, hidden1_dim),
        self.l2 =  nn.Linear(hidden1_dim, hidden2_dim),
        self.l3  =  nn.Linear(hidden2_dim, output_dim)
    def forward(self, x):
        # x = self.flatten(x)
        x=x.requires_grad_() # 开启梯度计算
        out3 = self.linear_sigmoid_stack(x)
        return out3

# 初始化模型
Bp_pid_model = BpNN(input_dim=config['input_dim'],
hidden1_dim=config['hidden1_dim'],
hidden2_dim=config['hidden2_dim'],
output_dim=config['output_dim'])
# 设置误差计算 Loss 函数
criterion = torch.nn.MSELoss(reduction='mean')
# 设置优化器 梯度下降算法
optimiser = torch.optim.Adam(Bp_pid_model.parameters(),
lr=config['learning_rate']) #lr=0.005 0.002 0.001
```

3.3.2 迭代过程

代码：

```
def test_pid3():
    Kp= config['p_init']
    Ki= config['i_init']
    Kd= config['d_init']

    pid_controller = DeltaPID(target=1, P=Kp, I=Ki, D=Kd)

    sys_func = sysFun()
    sys_func_input_list = []
    sys_func_Yout_list = [0]

    num_epochs = 2000
    hist = np.zeros(num_epochs)
```

```
    target = 1
    # sys_func.setSys(0.2, -0.01, 0.18, 0.21)

    for t in range(num_epochs):

        sys_func_input =
pid_controller.calculate_pid_output(sys_func_Yout_list[t])
        sys_func_input_list.append(sys_func_input)
        sys_func_Yout_list.append(sys_func.forward(sys_func_inpu
t))
        sys_status = sys_func.getState(target)

        pid_list = pid_controller.getPID()
        err_list = pid_controller.getErr()

        sys_status = np.append(sys_status, pid_list)
        sys_status = np.append(sys_status, err_list)
        # print(sys_status)

        bp_net_input =
torch.from_numpy(sys_status).type(torch.Tensor)
        # print("PID:",bp_net_input)

        bp_net_output = Bp_pid_model(bp_net_input)
        bp_net_output = bp_net_output.detach().numpy()

        r = config['pidK']
        bp_Kp = bp_net_output[0] * r
        bp_Ki = bp_net_output[1] * r
        bp_Kd = bp_net_output[2] * r

        bp_Kp_list.append(bp_Kp); bp_Ki_list.append(bp_Ki);
bp_Kd_list.append(bp_Kd)
        real_Kp_list.append(pid_list[0]);
real_Ki_list.append(pid_list[1]);
real_Kd_list.append(pid_list[2])

        pid_controller.setPID(bp_Kp,bp_Ki,bp_Kd) # update pid

        loss = criterion(torch.tensor(sys_func_Yout_list[-1],
requires_grad=True).float(), torch.tensor(target).float())
        # 计算loss函数 loss = (sys_func_Yout_list[-1] - target)
** 2
```

```
        print("Epoch ", t, "MSE: ", loss.item())
        hist[t] = loss.item()
        optimiser.zero_grad() # 清空梯度矩阵
        loss.backward() # 反向传播
        optimiser.step() # 执行优化

    print(np.array(sys_func_Yout_list))
```

四、 实验数据处理分析

由配置参数可知

```
target = 1.0

config =  {
…
            'p_init':0.001,
            'i_init':0.0,
            'd_init':0.0,

}# 0.03
```

初始 PID 参数设置为 Kp=0.001 Ki=0 Kd=0

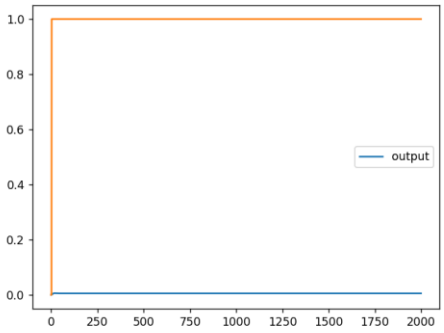
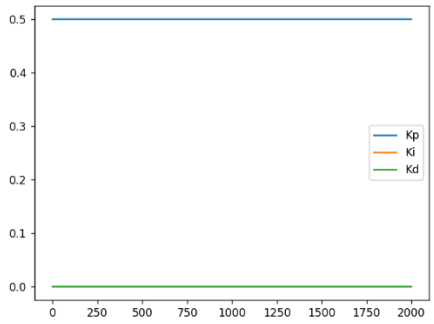
运行程序

```
LENOVO H:\Code\计控大作业\大作业2\code  pytorch 3.9.7  21m 15.455s
❯ python .\main1.py
```

4.1 无自整定

如果不使用 Bp 神经网络进行自整定，不难猜到结果会是系统输出不能很好跟随阶跃信号有很大的稳态误差

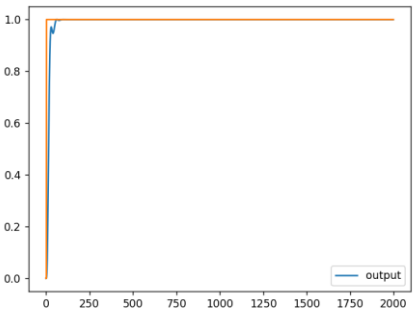
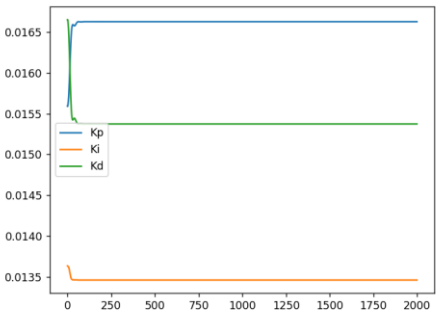
系统输出	PID 随时间的变化

结果与预想的一样，控制器无法对系统进行调节

4.2 加入 Bp-PID 自整定

当以同样的初始化参数开始，使用 Bp 神经网络进行 PID 自自整定，结果如下图

系统输出	PID 随时间的变化
	

相比上一组，显然 PID 自整定正常工作情况下，控制器对系统的调节效果非常好。可以看到 PID 参数随时间变化，在系统误差消失后 PID 参数不再变化

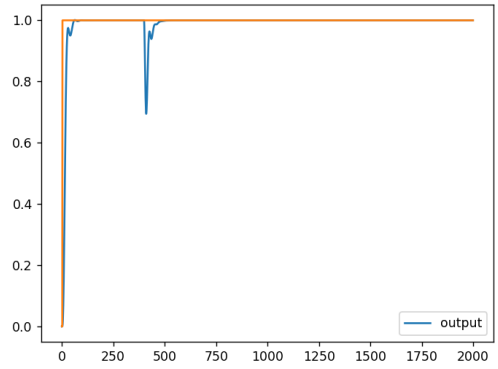
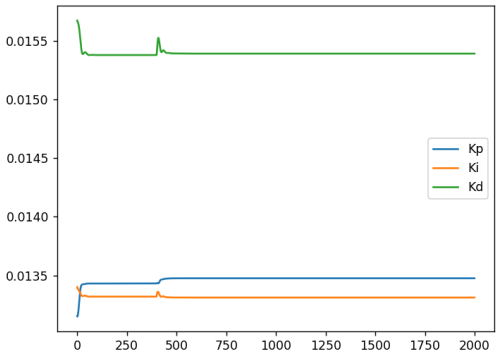
4.3 输入扰动

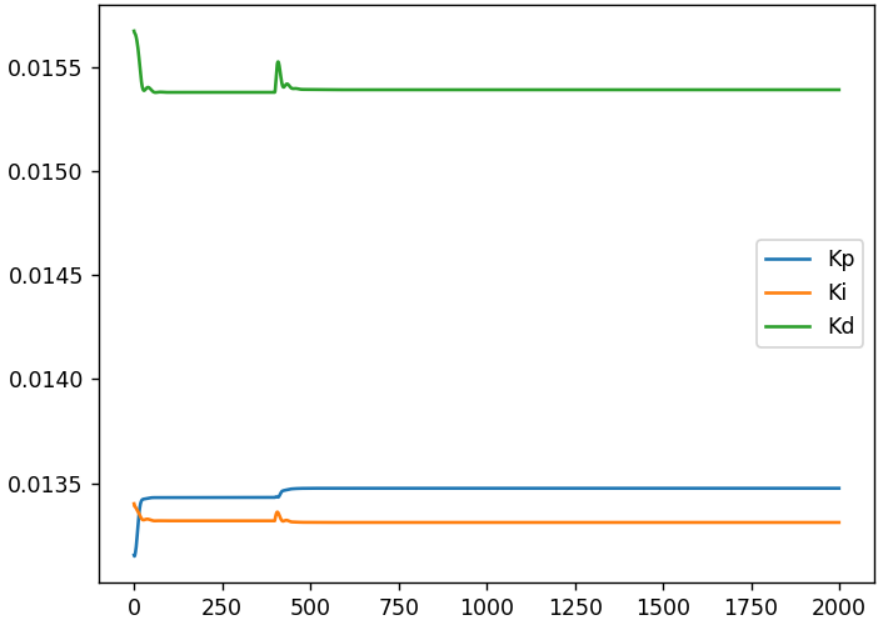
我们假设系统在 t=400 的时候遇到扰动，系统传递函数发生轻微的改变

```
        if(t == 400):
            sys_func.setSys(1.736, -0.7988, 0.1316, 0.1118)
            # a = 1.736
            # b = -0.7908
            # c = 0.1326
```

```
            # d = 0.1128
```

结果如图

系统输出	PID 随时间的变化
	



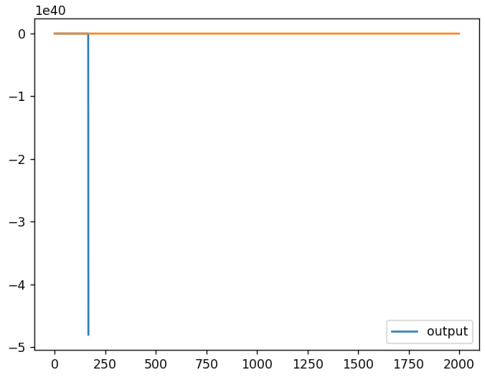
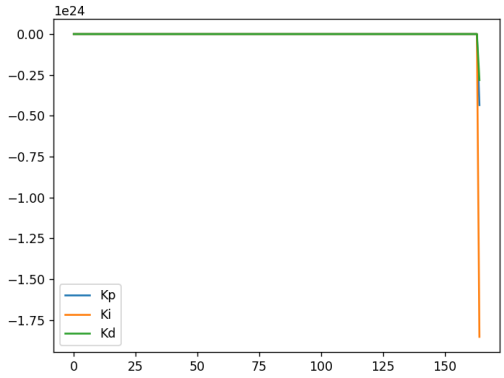
可以观察到，在系统发生突变的时候（t=400），系统突然地远离目标值，Bp 神经网络做出自适应的行为，增大参数 Kp，Kd，企图更快地调节系统以接近目标值。

4.3 激活函数对比

4.3.1 无任何激活函数

```
        self.linear_sigmoid_stack = nn.Sequential(
            nn.Linear(input_dim, hidden1_dim),
            nn.Linear(hidden1_dim, hidden2_dim),
            nn.Linear(hidden2_dim, output_dim),
        )
```

有可能会出现极端情况，结果是系统输出不知道飞到哪里去了，完全不能用

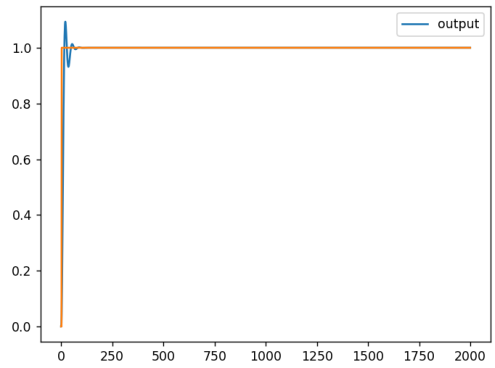
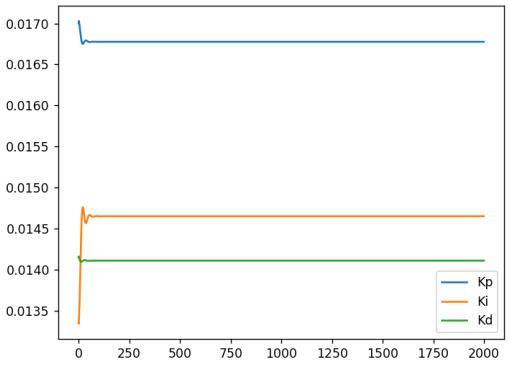
系统输出	PID 随时间的变化
	

4.3.2 输出层使用 Sigmoid 激活函数

```
        self.linear_sigmoid_stack = nn.Sequential(
            nn.Linear(input_dim, hidden1_dim),
            nn.Linear(hidden1_dim, hidden2_dim),
            nn.Linear(hidden2_dim, output_dim),
            nn.Sigmoid()
        )
```

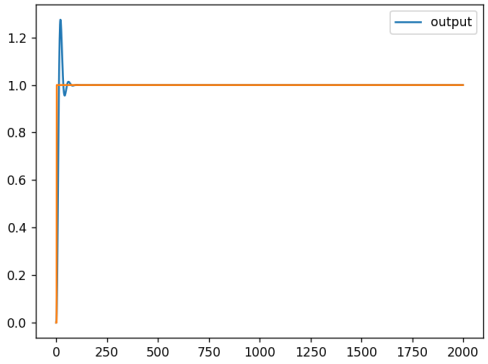
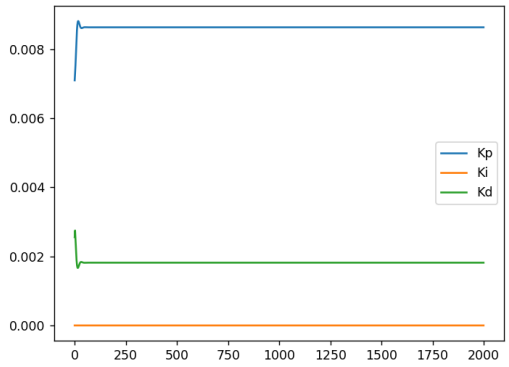
观察系统输出发现控制器正常工作，效果很不错

系统输出	PID 随时间的变化

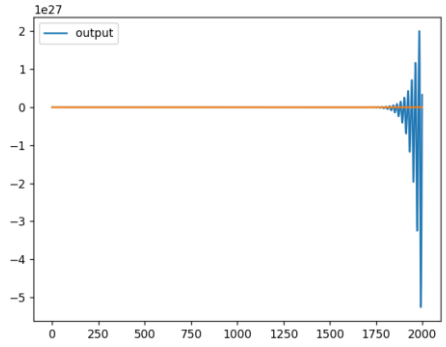
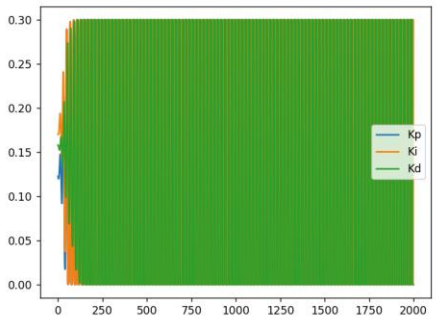
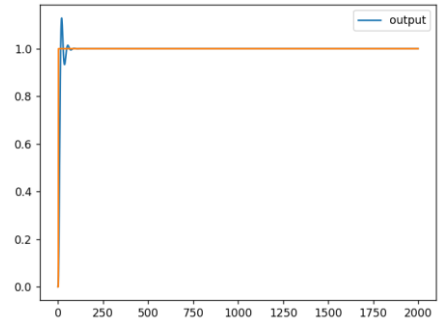
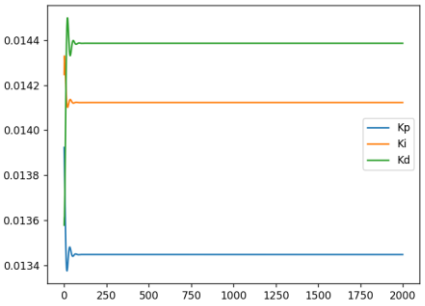
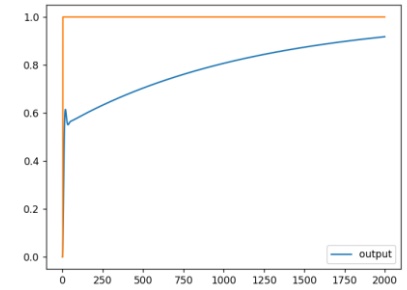
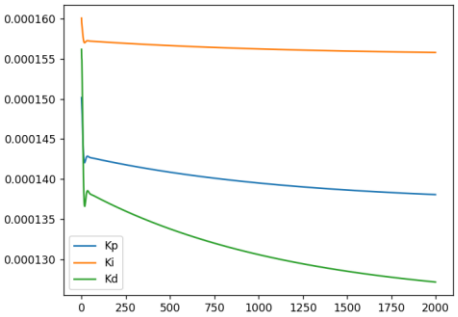
4.3.3 输出层使用 ReLU 激活函数

观察系统输出可以发现，效果并不理想。超调量比起上面使用 Sigmoid 的更大了。可能是因为 ReLU 函数没有上界，导致调节过度。

系统输出	PID 随时间的变化
	

4.4 改变 PID 调节强度

pidK	系统输出	PID 随时间的变化

0.3		
0.03		
0.0003		

pidK=0.3，过大的 PID 强度放大了系统噪声，造成自激振荡，如果应用在工程当中，将会是毁灭性的结果；pidK=0.03 是一个合适的值，没有过大的超调，又能迅速地跟上输入函数；pidK=0.0003 显然太小了，pid 的调节非常缓慢。

4.5 改变系统传递函数

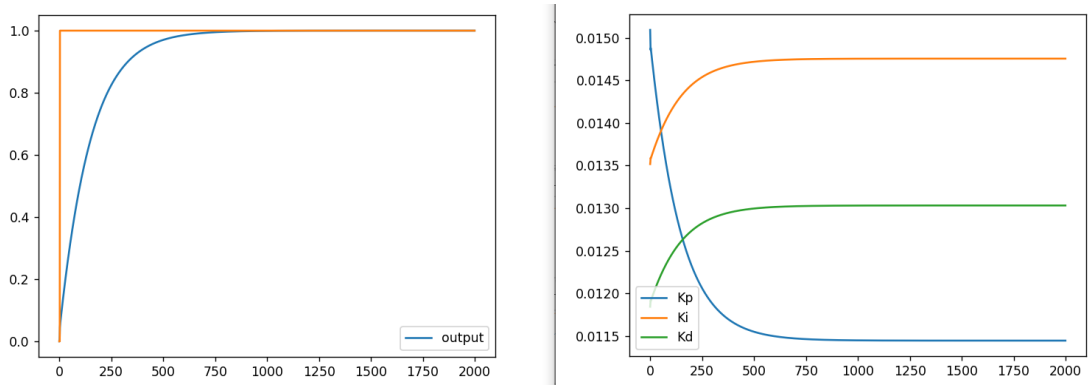
```
config =  {
            'p_init':0.1,
            'i_init':0.0,
            'd_init':0.0,
            'learning_rate':0.2,
            'pidK':   0.03, # 0.03
```

```
        }
```

改变系统传函

```
sys_func.setSys(0.2, -0.01, 0.18, 0.21)
```

系统输出：

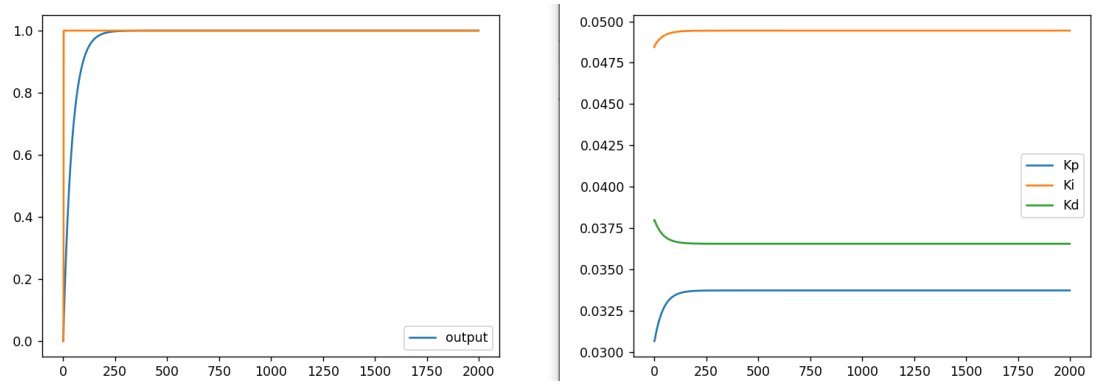


PID 控制器一样能工作，但是调节效果一般，调节速度过慢。这个情况下可以通过适当增大 PID 控制强度来进行修正，提高系统性能

增大 pidK

```
'pidK':   0.08, # 0.03
```

系统输出



可以看到系统输出明显改善。

之前也考虑过将参数'pidK'放到网络中进行学习。经过试验发现学习效果并不好，系统无法通过学习决定应该使用多大的 PID 调节强度，导致系统输出剧烈振荡无法稳定

五、 实验结果分析

如果不使用 Bp 神经网络进行自整定，结果是系统输出不能很好跟随阶跃信号有很大的稳态误差。

PID 自整定正常工作情况下，控制器对系统的调节效果非常好。可以看到 PID 参数随时间变化，在系统误差消失后 PID 参数不再变化

可以观察到，在系统因扰动发生突变的时候（t=400），系统输出突然地远离目标值，Bp 神经网络做出自适应的行为，增大参数 Kp，Kd，企图更快地调节系统以接近目标值。

在神经网络中尝试了不同的激活函数（Tanh、Sigmoid、ReLU 等等），最后发现输入层与隐藏层之间、隐藏层与隐藏层之间不加入激活函数，在输出使用 Sigmoid 系统的跟随效果最好

合理地设置 PID 控制器输出强度，可以很好地优化控制器效果

六、 实验结论与心得体会

实验结论：

经过实验发现，Bp 神经网络可以构建自适应 PID 控制器，但是仍存在较大不足。

Bp 神经网络是基于随机梯度下降进行优化和运行的，存在一定的不确定性，并不能保证找到最优的 PID 参数，甚至有可能相同参数设置情况下，上一次与下一次的运行结果不同，存在不稳定性。在实际工程中的适用性不够强。

自适应控制器对于不同系统的泛化能力不强，网络模型的逼近和推广能力与学习样本的典型性密切相关，而从问题中选取典型样本实例组成训练集是一个很困难的问题。如果系统函数发生较大的改变，需要重新调整初始参数。

心得体会：

通过实验，本人发现神经网络具有很好地对非线性模型的描述能力，通过误差反馈，无需告知神经网络 Kp Ki Kd 参数的含义，它也可以在迭代中进行学习，自己寻找事物之间的关联，将相关的神经元权重增大，对于不相关的减小权重以弱化关联。

增量型 pid 的好处是只需要存储三个历史值，运算简单，这对于例如嵌入式平台资源少的平台，PID 算法也可以部署

本人总结该实验中的难点有以下几点：

1. 理解实验的细节，使用 python 重新实现，阶跃函数、PID 控制器函数、Bp 神经网络、误差计算反向传播、画图程序

2. 一开始上手实验对于参数的选取感到茫然，不知道应该往哪个方向去进行调整

3. 发现 PID 控制单元的输出变化剧烈，无法有效控制系统。

针对以上问题的解决方案：

1. 结合 pytorch 的官方文档，搭建神经网络。其他的，并不难只需要花点时间

2. 逐渐明白各个参数的含义，为了防止在梯度下降的时候欠拟或者过拟合，选取了一个合适的学习率，增强稳定性。令 PID 初始值均为 0，观察 PID 的变化趋势，然后尝试改变 3 个参数中其中一个，观察效果。发现适当增大隐藏层的维数，可以让系统有更好的输出。在神经网络中尝试了不同的激活函数（Tanh、Sigmoid、ReLU 等等），最后发现输入层与隐藏层之间、隐藏层与隐藏层之间不加入激活函数的效果最好，使用 Tanh 的效果次之

3. 在神经网络的输出端加上 Sigmoid 激活函数，将输出限制在（0，1）的范围，并且定义一个参数 `r = config['pidK']`，用于控制 PID 控制器的输出强度（调节剧烈程度）

```
        bp_Kp = bp_net_output[0] * r
        bp_Ki = bp_net_output[1] * r
        bp_Kd = bp_net_output[2] * r
```

从而弱化 PID 调节强度，解决了 PID 参数剧烈抖动无法调节系统的问题

参考文献

1. N. Minorsky, “Directional stability of automatically steered bodies,” *Naval Engineers Journal*, vol. 34, no. 2, pp. 280–309, 1922.

2. Y. Tao and Y. Yin, New PID Control and Applications, China Machine Press, Beijing, China, 1998

3. Y. Hu, T. Guo, and P. Han, “Research on Smith’s predictive control algorithms and their applications in DCS,” Computer Simulation, vol. 33, no. 5, pp. 409–412, 2016.

4. D. Liang, J. Deyi, Z. Ma, and W. Zhang, “Variable-pitch controllers for wind turbine generators based on an improved Smith prediction algorithm,” Electric Drive Automation, vol. 38, no. 6, pp. 25–29, 2016

5. P. Lu, H. Zhang, and R. Mao, “Comparative research on Smith predictive compensation control and PID control,” Journal of China University of Metrology, vol. 20, no. 2, pp. 171–179, 2009.

6. K. J. Astr¨om, T. H¨agglund, C. C. Hang, and W. K. Ho, ° “Automatic tuning and adaptation for PID controllers-a survey,” Control Engineering Practice, vol. 1, no. 4, pp. 699–714, 1993.

7. K. J. Astrom and B. Wittenmark, “Self-tuning controllers based on pole-zero placement,” IEE proceedings D-control theory and applications, vol. 127, no. 3, pp. 120–130, 1980.

8. L. A. Zadeh, “Fuzzy sets as a basis for a theory of possibility,” Fuzzy Sets and Systems, vol. 1, no. 1, pp. 3–28, 1978.

9. L. Xue, Y. Liu, E. Zhu, and X. Ma, “Design of intelligent fuzzyPID temperature control systems,” Information Recording Material, vol. 19, no. 11, pp. 118–120, 2018

10. Z. Guo, H. Yu, and L. Chen, “High-speed galvanometer motor control based on fuzzy PID,” Small & Special Electrical Machines, vol. 47, no. 4, pp. 1–5, 2019.

11. X. Bai, Research on Fuzzy Controllers and :eir Applications in Host Steam Temperature Controllers, North China Electric Power University (Hebei), Beijing, China, 2006.

12. B. Widrow and R. Winter, “Neural nets for adaptive filtering and adaptive computer, pattern recognition,” An Introduction To Neural And Electronic Networks, vol. 21, 1990.

13. T. Liu and Y. Zhang, “Research on neural network PID control in speed control systems for motors of hydraulic pumps,” Telecom Power Technology, vol. 35, no. 5, pp. 4–7, 2018.

14. X. You, C. Su, and Y. Wang, “An overview of improvement of algorithms for BP neural networks,” Minying Keji, vol. 34, no. 4, pp. 146-147, 2018.

15. C. Peng, Y. Zheng, and Z. Hu, “Adaptive single neuron control over time-varying time delay systems,” Computing Technology and Automation, no. 1, pp. 17–19, 2005.

16. E. P. Maillard and D. Gueriot, “RBF neural network, basis functions and genetic algorithm,” in Proceedings of the International Conference on Neural Networks (ICNN’97), vol. 4, pp. 2187–2192, IEEE, Houston, TX, USA, July 1997.

17. D. E. Goldberg and J. H. Holland, “Genetic algorithms and machine learning,” Machine Learning, vol. 3, no. 2-3, pp. 95–99, 1988

18. H. Liu, Q. Duan, N. Li, and Y. Zhou, “PID parameter tuning and optimization based on genetic algorithms,” Journal of North China Electric Power University, vol. 3, pp. 31–33, 2001.

19. C. Chen, C. Cheng, C. Luo, and R. Wang, “PID temperature control for reactors based on genetic algorithms,” Scientific and Technological Innovation, vol. 28, no. 6, pp. 72-73, 2019.

代码附录：

```
# 导入 需要的 library 库
import numpy as np # 科学计算
import matplotlib.pyplot as plt # 画图

import torch
import torch.nn as nn
import math

target = 1.0

config =  {
            'input_dim':13,
            'hidden1_dim':16,
            'hidden2_dim':6,
            'output_dim':3,
            'p_init':0.1,
            'i_init':0.0,
            'd_init':0.0,
            'learning_rate':0.002,
            'pidK': target * 0.03, # 0.03
        }



#增量式PID 系统
class DeltaPID(object):
    def __init__(self, target, P ,I ,D):
        self.Kp = P
        self.Ki = I
        self.Kd = D

        self.target = target # The target

        self.error = 0.0       # t time
        self._p_error = 0.0    # t-1 time
        self._pp_error = 0.0   # t-2 time

        self.pid_output = 0.0  # PID 输出

    def setPID(self, P ,I ,D ):
        self.Kp = P            # 更新 P
        self.Ki = I            # 更新 I
        self.Kd = D            # 更新 D
```

```
        print("setPID:",self.Kp,self.Ki,self.Kd)
        return self

    def setTarget(self, target):
        self.target = target   # 设置PID 环节输入，即跟随目标
        return self

    def getPID(self):
        return np.array([self.Kp, self.Ki, self.Kd])
        # 返回 PID 系数

    def getErr(self):
        return np.array([self.error, self._p_error,
self._pp_error])
        # 返回 误差

    def calcalate_deltaU(self, cur_val):
        error = self.target - cur_val
        self.error = error
        p_change = self.Kp * (error - self._p_error)
        i_change = self.Ki * error
        d_change = self.Kd * (error - 2 * self._p_error +
self._pp_error)
        delta_output = p_change + i_change + d_change
        # 计算增量式PID 输出
        self._p_error = error
        # 保持 上一次误差
        self._pp_error = self._p_error
        # 保持 上上次误差
        return delta_output

    def calculate_pid_output(self, cur_val):
        self.pid_output += self.calcalate_deltaU(cur_val)
        # 对增量输出进行累加
        return self.pid_output  # pid out -> sysFun in
        # 返回 PID 直接输出

    def forward(self, input):
        return self.calculate_pid_output(self, input)

class sysFun(object):
    def __init__(self) -> None:
        self.a = 1.736
```

```
        self.b = -0.7788
        self.c = 0.1216
        self.d = 0.1118

        self.Y = 0.0
        self.Y_1 = 0.0
        self.Y_2 = 0.0

        self.U = 0.0
        self.U_1 = 0.0
        self.U_2 = 0.0

    def forward(self, u):
        self.Y_2 = self.Y_1
        self.Y_1 = self.Y

        self.U_2 = self.U_1
        self.U_1 = self.U
        self.U = u

        self.Y = self.a * self.Y_1 + self.b * self.Y_2 + self.c *
self.U_1 + self.d * self.U_2

        return self.Y

    def setSys(self,a, b, c, d):
        self.a = a
        self.b = b
        self.c = c
        self.d = d
        return self

    def getState(self, target):
        return np.array([self.Y, self.Y_1, self.Y_2, self.U,
self.U_1, self.U_2, target-self.Y])


class BpNN(nn.Module):
    def __init__(self, input_dim=7, hidden1_dim=7, hidden2_dim=7,
output_dim=3):
        super(BpNN, self).__init__()
        self.hidden1_dim = hidden1_dim
        self.hidden2_dim = hidden2_dim

        self.flatten = nn.Flatten()
```

```
        # 设置网络结构
        self.linear_sigmoid_stack = nn.Sequential(
            nn.Linear(input_dim, hidden1_dim),
            nn.Tanh(), # 激活函数
            nn.Linear(hidden1_dim, hidden2_dim), # 线性层
            nn.Tanh(), # 激活函数
            nn.Linear(hidden2_dim, output_dim), # 线性层
            nn.Sigmoid() # 激活函数

        )
        self.l1 =  nn.Linear(input_dim, hidden1_dim),
        self.l2 =  nn.Linear(hidden1_dim, hidden2_dim),
        self.l3  =  nn.Linear(hidden2_dim, output_dim)


    def forward(self, x):
        # x = self.flatten(x)
        x=x.requires_grad_() # 开启梯度计算
        out3 = self.linear_sigmoid_stack(x)

        return out3

# 初始化模型
Bp_pid_model = BpNN(input_dim=config['input_dim'],
hidden1_dim=config['hidden1_dim'],
hidden2_dim=config['hidden2_dim'],
output_dim=config['output_dim'])
# 设置误差计算 Loss 函数
criterion = torch.nn.MSELoss(reduction='mean')
# 设置优化器 梯度下降算法
optimiser = torch.optim.Adam(Bp_pid_model.parameters(),
lr=config['learning_rate']) #lr=0.005 0.002 0.001



time_list = np.arange(0, 5000, 1)
step_list = np.append([0,0,0], [target for x in range(0, 5000,
1)])
step_list3 = np.append([0,0,0], [target for x in range(0, 400,
1)])
step_list3 = np.append(step_list3, [target*2 for x in range(0,
4600, 1)])
# plt.plot(time_list[:2000], step_list[:2000])
```

```
bp_Kp_list = []
bp_Ki_list = []
bp_Kd_list = []

real_Kp_list = []
real_Ki_list = []
real_Kd_list = []


def test_pid2():
    Kp= config['p_init']
    Ki= config['i_init']
    Kd= config['d_init']

    pid_controller = DeltaPID(target=target, P=Kp, I=Ki, D=Kd)

    sys_func = sysFun()
    sys_func_input_list = []
    sys_func_Yout_list = [0]

    num_epochs = 2000
    hist = np.zeros(num_epochs)

    # sys_func.setSys(0.2, -0.01, 0.18, 0.21)

    for t in range(num_epochs):

        # 扰动
        if(t == 400):
            sys_func.setSys(1.736, -0.7988, 0.1316, 0.1118)
            # a = 1.736
            # b = -0.7908
            # c = 0.1326
            # d = 0.1128

        # sys_func_input = -
pid_controller.calculate_pid_output(sys_func_Yout_list[i]) +
step_list[i]
        sys_func_input =
pid_controller.calculate_pid_output(sys_func_Yout_list[t])
        sys_func_input_list.append(sys_func_input)
        sys_func_Yout_list.append(sys_func.forward(sys_func_input)
)
        # print(round(time_list[i],5))
        # print(i,step_list[i])
```

```
        sys_status = sys_func.getState(target)

        pid_list = pid_controller.getPID()
        err_list = pid_controller.getErr()

        sys_status = np.append(sys_status, pid_list)
        sys_status = np.append(sys_status, err_list)
        # print(sys_status)
        r = config['pidK']

        bp_net_input =
torch.from_numpy(sys_status).type(torch.Tensor)
        # print("PID:",bp_net_input)

        bp_net_output = Bp_pid_model(bp_net_input)
        bp_net_output = bp_net_output.detach().numpy()

        bp_Kp = bp_net_output[0] * r
        bp_Ki = bp_net_output[1] * r
        bp_Kd = bp_net_output[2] * r

        bp_Kp_list.append(bp_Kp); bp_Ki_list.append(bp_Ki);
bp_Kd_list.append(bp_Kd)
        real_Kp_list.append(pid_list[0]);
real_Ki_list.append(pid_list[1]);
real_Kd_list.append(pid_list[2])

        pid_controller.setPID(bp_Kp,bp_Ki,bp_Kd) # update pid

        loss = criterion(torch.tensor(sys_func_Yout_list[-1],
requires_grad=True).float(), torch.tensor(target).float())
        # loss = (sys_func_Yout_list[-1] - target) ** 2
        print("Epoch ", t, "MSE: ", loss.item())
        hist[t] = loss.item()
        optimiser.zero_grad()
        loss.backward()
        optimiser.step()


    print(np.array(sys_func_Yout_list))

    plt.figure(1)
```

```
    plt.plot(time_list[:2000],
np.array(sys_func_Yout_list[:2000]), label="output")
    plt.plot(time_list[:2000], step_list[:2000])
    # plt.plot(time_list[:100],
np.array(sys_func_input_list[:100]), label="sys_func_input" )
    plt.legend()

    plt.figure(2)
    plt.plot(time_list[:2000], np.array(bp_Kp_list[:2000]),
label="Kp" )
    plt.plot(time_list[:2000], np.array(bp_Ki_list[:2000]),
label="Ki" )
    plt.plot(time_list[:2000], np.array(bp_Kd_list[:2000]),
label="Kd" )
    plt.legend()

    plt.figure(3)
    plt.plot(time_list[:2000], np.array(real_Kp_list[:2000]),
label="Kp" )
    plt.plot(time_list[:2000], np.array(real_Ki_list[:2000]),
label="Ki" )
    plt.plot(time_list[:2000], np.array(real_Kd_list[:2000]),
label="Kd" )
    plt.legend()

    plt.show()

if __name__=='__main__':
    test_pid2()
```

指导教师批阅意见：

验证性实验（报告）评分细则表

评分项目	满分标准	成绩	权重
原始数据	准确、真实可信，记录完整		40%
实验数据分析与处理	分析与处理正确，有必要的过程，能恰当运用图表，分析全面、正确结论合理。		20%
实验报告	实验报告格式规范，内容完整		10%
撰写质量	撰写认真、报告整洁、清晰		10%
实验心得与思考题	有心得体会，完成思考题		20%

A（100~85）	B（84~75）	C（74~65）	D（64~60）	F（<60）

成绩评定：

指导教师签字：

年　　月　　日

备注：