第 5 章

应用拓展问题

5.1 问题描述

5.1.1 前述问题回顾

全书将元学习的研究梳理为 5 个问题。在解释最后一个问题之前，有必要回顾和梳理一下前 4 个问题。第 1 章的联合训练问题与第 2 章的任务构建问题是元学习研究的基本问题。在前两章整理了相关的释义代码(近似于伪代码)，而对应的原始代码则包含在第 4 章。第 1 章的联合训练定义主要是通过 def train_model(self, mytrain_data, inner_optimizer, inner_step, outer_optimizer=None)函数实现，在原始代码中对应 4.2.2 节的 def train (model, saver, sess, exp_string, data_generator, resume_itr=0)函数。第 2 章的任务构建通过 class MyDataTask(object)实现，在原始代码中对应 3.4.1 节的 def get_images (paths, labels, nb_samples=None, shuffle=True)的图像读取细节以及 4.2.2 节中 def train(model, saver, sess, exp_string, data_generator, resume_itr=0)的任务构建细节。

第 3 章的过程建模问题与第 4 章的输入输出问题是元学习研究的核心问题。元学习系统是面向过程的机器学习系统，第 3 章描述了随机抽样过程、样本学习过程、最优化过程，元优化过程、最大池化过程正是其主要过程，这些过程从元学习系统的输入开始，到系统的输出结束，与第 4 章的输入输出问题相呼应，是整个元学习过程的主线。

第 4 章的代码文件 main.py 是输入输出的执行器，作为主程序，除了调用第 3 章的代码文件 data_generator.py 里的 DataGenerator 模块，也调用了本章代码文件 maml.py 中的 MAML 模块，调用形式为 model = MAML(dim_input, dim_output, test_num_updates= test_num_updates)。

5.1.2 元学习系统网络

算法是以应用为最终目标。因此，本章解释的应用拓展问题是元学习研究的根本问题。在关键程序 maml.py 中，调用了代码文件 special_grads.py 以及代码文件 utils.py 里的 4 个模块(mse、xnet、conv_block 和 normalize)。本节将对代码文件 maml.py 进行详细注解，

并以此为主线，系统化地研究元学习系统的应用拓展问题。

代码文件 maml.py 主要实现了 MAML 的核心算法，对其中涉及的神经网络进行了定义。作为深度学习模型的拓展应用，元学习模型也是以神经网络算法为基础。如前所述，任务构建问题可以转化为 3 个问题，即如何划分小样本单元、如何设计元学习算法、如何运用预训练结果提升元优化的准确率等。其中，后两个问题都与神经网络算法联系密切。神经网络是人工神经网络的简称，其算法思想是要建立一种模仿生物神经网络的结构和功能的数学模型或计算模型，因此也可称为类神经网络。其模仿的对象不限于人类，也可以是其他动物的中枢神经系统，特别是大脑。人工神经网络，顾名思义，由大量人工神经元连接而成的计算学习网络。神经网络可用于对任意函数进行估计或近似，因此也适用于元学习模型。人工神经网络是一个自适应系统，能在任务构建过程中自主改变内部结构。由于元任务样本的多样性，元学习模型会涉及一系列非线性、统计性数据的建模问题，而神经网络中的前向传播、反向传播及其优化机制构成了很好的解决方案。

通过前 4 章的阅读，读者已经拥有基础的编程思维，接下来通过理解代码文件 maml.py 的设计，进一步发展较为完善的编程思维。

(1) 需要导入 NumPy 库和 sys 库。这是 Python 的 2 个内置库，相对而言，NumPy 库更为常用(在代码中一般简写为 np)，sys 库则提供了访问与解释器、系统相关的变量和函数，可用于处理全局参数。

(2) 元学习网络的定义需要导入深度学习框架 TensorFlow(在代码中简写为 tf)，还需要从 tensorflow.python.platform 中导入 flags，以便定义和调用相关的全局参数。

(3) 代码文件 maml.py 本质上是前述模块的应用拓展，因此需要从 utils 中导入 mse、xent、conv_block 和 normalize。如 4.1.3 节所述，为有效处理错误，并输出提示，还需要同时从__future__模块导入 print_function 特性。

5.1.3　MAML 的定义

全局参数只需要通过外部命令行传递，不必在算法代码内部修改。类似于第 4 章的编程思路，在样本训练和测试环节，将使用 FLAGS 解析全局参数。这些全局参数可以用 flags.DEFINE_XXX(参数名称，默认值，具体描述)的形式进行定义。因此，在正式定义 MAML 算法之前，可借助命令 FLAGS = flags.FLAGS，引入全局变量 tensorflow.python.platform.flags.FLAGS，并简写为 FLAGS。

在代码文件 maml.py 中，以 class 的形式定义 MAML 算法。在 4.4.2 节的生成器初始化模块中，class 形式的定义需要借助 def __init__()函数，对 class 的创建对象进行初始化。本章对 MAML 算法的定义采用 class 形式的定义，其核心代码如下。

```
class MAML:
 #class 是 MAML 的定义形式
def __init__(self, dim_input = 1, dim_output = 1, test_num_updates = 5):
 #创建 class 的方式是借助 __init__()函数对创建对象进行初始化
```

```
# 该函数有 4 个输入,实例对象 self 必不可少
# 输入输出维度 dim_input 和 dim_output 均设置为 1
# 元测试期间,梯度更新次数 test_num_updates 设置为 5
# 接下来,是对 self 属性的一系列定义
        self.dim_input = dim_input
# 将 self.dim_input 定义为 dim_input
        self.dim_output = dim_output
# 将 self.dim_output 定义为 dim_output
        self.update_lr = FLAGS.update_lr
# 将 self.update_lr 定义为 FLAGS.update_lr
        self.meta_lr = tf.placeholder_with_default(FLAGS.meta_lr, ())
# 将 self.meta_lr 定义为 tf.placeholder_with_default(FLAGS.meta_lr, ())
# 将 tf.placeholder_with_default()函数用于占位符操作,用法如下:
# tf.placeholder_with_default(input, shape, name = None)
# 通过一个占位符 op 输入张量 FLAGS.meta_lr
# 注意,shape 为空,因此输入张量 FLAGS.meta_lr 的形状不受限制
# 单词 placeholder,有占位符的含义
        self.classification = False
# 将 self.classification 定义为 False,注意,只是初始值
        self.test_num_updates = test_num_updates
# 将 self.test_num_updates 定义为 test_num_updates
# 注意,定义 self 的模型属性时,需要考虑数据集的差异,共分为 3 种不同情形:
        if FLAGS.datasource == 'sinusoid':
# 情形 1. 如果数据集为无穷集 sinusoid
            self.dim_hidden = [40, 40]
# 将隐含层维度 self.dim_hidden 定义为[40, 40]
            self.loss_func = mse
# 将 self.loss_func 定义为 mse
            self.forward = self.forward_fc
# 将 self.forward 定义为 self.forward_fc,相当于定义了 MAML 的前向传播函数 fc()
            self.construct_weights = self.construct_fc_weights
# 将 self.construct_weights 定义为 self.construct_fc_weights
# 本质上是构造 MAML 网络的权值

elif FLAGS.datasource == 'omniglot' or FLAGS.datasource == \
'miniimagenet':

# 情形 2. 如果源数据集为 Omniglo 或 Mini-ImageNet
            self.loss_func = xent
# 将 self.loss_func 定义为 xent
            self.classification = True
# 将 self.classification 定义为 True
# 对于此情形,其他模型属性的定义又分为两种情况:
            if FLAGS.conv:
# 情形 2-1. 如果使用一个卷积网络
# 参数 conv 在 4.2.1 节已定义,默认值是 True,用于决定是否使用一个卷积网络
```

```
                self.dim_hidden = FLAGS.num_filters
#此时,隐含层维度等于过滤器数量
                self.forward = self.forward_conv
#将前向传播函数改为前向卷积
                self.construct_weights = self.construct_conv_weights
#将模型网络权值改为卷积权值
            else:
#情形 2-2. 如果不使用卷积网络
                self.dim_hidden = [256, 128, 64, 64]
#此时隐含层维度为[256, 128, 64, 64]
                self.forward = self.forward_fc
#前向传播函数不用改变
                self.construct_weights = self.construct_fc_weights
#模型网络权值不用改变

对于情形 2,注意,不同数据集的输出通道维度 self.channels 需要分别定义:
            if FLAGS.datasource == 'miniimagenet':
#如果源数据集为 Mini-ImageNet
                self.channels = 3
#输出通道维度为 3
            else:
#如果源数据集为 Omniglot
                self.channels = 1
#输出通道维度为 1
            self.img_size = \
int(np.sqrt(self.dim_input/self.channels))
#定义图像尺寸的属性
#先计算平均输入维度 self.dim_input/self.channels,再求平方根,最后取整

        else:
#情形 3. 如果数据集未知
            raise ValueError('Unrecognized data source.')
#输出错误提示
```

5.2　建模思路

5.2.1　系统模型拓展

第 4 章已经对系统架构模型进行定义和较为全面地研究,但系统模型应用和拓展的细节主要包含在代码文件 maml.py 中,因此未能深入研究,现注解如下。

```
    def construct_model(self, input_tensors = None, prefix = 'metatrain_'):
#定义 construct_model()函数,该函数属于第 4 章系统架构的拓展模块,提供输入输出的接口
```

```
#与第 4 章类似,如下接口变量中的 a 代表训练集,b 代表测试集
        if input_tensors is None:
#如果输入张量为空,执行如下占位操作
            self.inputa = tf.placeholder(tf.float32)
#调用 tf.placeholder()函数,完成训练集输入占位,输入接口类型为 tf.float32
            self.inputb = tf.placeholder(tf.float32)
#调用 tf.placeholder()函数,完成测试集输入占位,输入接口类型为 tf.float32
            self.labela = tf.placeholder(tf.float32)
#调用 tf.placeholder()函数,完成训练集标签占位,标签接口类型为 tf.float32
            self.labelb = tf.placeholder(tf.float32)
#调用 tf.placeholder()函数,完成测试集标签占位,标签接口类型为 tf.float32
        else:
#如果输入张量非空,直接执行如下接口赋值操作
            self.inputa = input_tensors['inputa']
#将 input_tensors 中的 inputa 赋值到 self.inputa 接口
            self.inputb = input_tensors['inputb']
#将 input_tensors 中的 inputb 赋值到 self.inputb 接口
            self.labela = input_tensors['labela']
#将 input_tensors 中的 labela 赋值到 self.labela 接口
            self.labelb = input_tensors['labelb']
#将 input_tensors 中的 labelb 赋值到 self.labelb 接口

        with tf.variable_scope('model', reuse = None) as \
training_scope:
#调用 tf.variable_scope()函数,设置 model 的作用域为 training_scope
            if 'weights' in dir(self):
#如果 dir(self)已包含'weights',那么
                training_scope.reuse_variables()
#拓展重复利用参数的接口
                weights = self.weights
#重复利用的 weights 参数可直接定义为 self.weights
            else:
#如果 dir(self)不包含'weights',那么
                self.weights = weights = self.construct_weights()
#定义 self.weights = weights,其中参数 weights 可通过 self.construct_weights()构造
            lossesa, outputas, lossesb, outputbs = [], [], [], []
#训练集、测试集的输出和损失均初始化为空集
            accuraciesa, accuraciesb = [], []
#训练集、测试集的精度均初始化为空集
            num_updates = max(self.test_num_updates, FLAGS.num_updates)
#梯度更新次数取 self.test_num_updates 和 FLAGS.num_updates 中较大者
            outputbs = [[]] * num_updates
#拓展测试集的输出,按照梯度更新次数成倍归并
            lossesb = [[]] * num_updates
#拓展测试集的损失,按照梯度更新次数成倍归并
            accuraciesb = [[]] * num_updates
#拓展测试集的精度,按照梯度更新次数成倍归并
```

5.2.2　梯度模型拓展

在第 4 章已经对输入输出模块进行定义，在第 3 章中也对样本学习过程做了初步研究，但是元任务的学习过程和应用细节主要包含在代码文件 maml.py 中，因此未能深入研究。其中主要涉及快速梯度下降及其应用，现结合学习过程，注解如下。

首先是梯度模型的拓展，通俗地说，是通过一次梯度下降，得到快速梯度下降所需的递归参数 fast_weights，其具体分为入学、预习 2 个环节，并将在本小节展示，核心学习过程将在 5.2.3 节展示。

```
#入学环节:初始化,即学习前的准备工作
            def task_metalearn(inp, reuse=True):
#定义 task_metalearn()函数,是第 4 章输入输出的拓展模块,提供学习过程接口
                inputa, inputb, labela, labelb = inp
#训练集、测试集的输入和标签都从 inp 中读取
                task_outputbs, task_lossesb = [], []
#任务 task 的学习过程开始之前,测试集的输出和损失均初始化为空集

#预习环节:通过一次梯度下降,得到递归参数 fast_weights,用于后续的快速梯度下降
                if self.classification:
#如果 self.classification 为 False
#在 MAML 定义模块已将 self.classification 初始化为 False
                    task_accuraciesb = []
#任务 task 的学习过程开始之前,测试集的精度也初始化为空集
                task_outputa = self.forward(inputa, weights, reuse=reuse)
#从训练集的输入开始,使用共享权值 weights 完成前向传播
                task_lossa = self.loss_func(task_outputa, labela)
#根据输出结果和真实标签值,调用 self.loss_func()函数初始化对应的损失
                grads = tf.gradients(task_lossa, list(weights.values()))
#损失 task_lossa 的梯度初始化,求导对象为 list(weights.values())
                if FLAGS.stop_grad:
#如果收到 stop_grad 命令,那么
                    grads = [tf.stop_gradient(grad) for grad in grads]
#调用 tf.stop_gradient()函数,停止梯度计算,得到 grads 的更新值
                gradients = dict(zip(weights.keys(), grads))
#利用 dict(zip())函数,将两个列表 weights.keys()、grads 组合为字典 gradients
                fast_weights = dict(zip(weights.keys(), [weights[key] - self.update_lr
* gradients[key] for key in weights.keys()]))
#利用 dict(zip())函数,将两个列表 weights.keys()、[weights[key] - self.update_lr *
#gradients[key] for key in weights.keys()]组合为字典 fast_weights,将用于快速梯度下降
                output = self.forward(inputb, fast_weights, reuse=True)
#利用 self.forward()函数,计算当前测试集上的输出结果
                task_outputbs.append(output)
#将当前测试集上的输出结果归并到 task_outputbs
                task_lossesb.append(self.loss_func(output, labelb))
#将当前测试集上计算的损失归并到 task_lossesb
```

5.2.3 快速梯度下降

上述算法通过一次梯度更新，得到快速梯度下降的初始化参数 fast_weights。接下来，将循环执行快速梯度下降，完成余下的 num_updates－1 次梯度更新，并且每次更新后都会重新计算 fast_weights。每一轮的循环算法均与 5.2.1 节基本一致，因此不难理解其编程思路，核心代码如下。

```
#学习环节:循环执行快速梯度下降
for j in range(num_updates - 1):
#算法与5.2.1节基本一致,因此不难理解其编程思路
                    loss = self.loss_func(self.forward(inputa, fast_weights, reuse =
True), labela)
#根据输出结果和真实标签值,调用self.loss_func()函数计算当前训练损失
#首先计算 outputa = self.forward(inputa, fast_weights, reuse = True)
#然后计算第 j 次快速梯度下降后训练集上的 loss = self.loss_func(outputa, labela)
                    grads = tf.gradients(loss, list(fast_weights.values()))
#计算损失 loss 的梯度,求导对象为 list(fast_weights.values())
                    if FLAGS.stop_grad:
#如果收到停止梯度计算的命令
                        grads = [tf.stop_gradient(grad) for grad in grads]
#调用 tf.stop_gradient()函数,停止梯度计算,得到第 j 次快速梯度下降后 grads 的更新值
                    gradients = dict(zip(fast_weights.keys(), grads))
#利用 dict(zip())函数,将两个列表 fast_weights.keys()、grads 组合为字典 gradients
          fast_weights = dict(zip(fast_weights.keys(), [fast_weights[key] - self.update
_lr * gradients[key] for key in fast_weights.keys()]))
#重新计算 fast_weights,即利用 dict(zip())函数,将两个列表 fast_weights.keys()、
# [fast_weights[key] - self.update_lr * gradients[key] for key in fast_weights.keys()]组合
#为字典 fast_weights
        output = self.forward(inputb, fast_weights, reuse = True)
#利用 self.forward()函数,计算当前测试集上的输出结果
                    task_outputbs.append(output)
#将当前测试集上的输出结果归并到 task_outputbs
              task_lossesb.append(self.loss_func(output, labelb))
#将当前测试集上的输出损失归并到 task_lossesb
              task_output = [task_outputa, task_outputbs, task_lossa, task_lossesb]
#将 task_outputa、task_outputbs、task_lossa、task_lossesb 归并到 task_out

                if self.classification:
#如果 self.classification 为 True,
                    task_accuracya = \
tf.contrib.metrics.accuracy(tf.argmax(tf.nn.softmax(task_outputa), 1), tf.argmax(labela, 1))
#计算训练集上的精度,即先调用 tf.nn.softmax()函数将 task_outputa 转换为概率值
#然后调用 tf.argmax()函数找出排第一的概率值及其对应标签值
#最后调用 tf.contrib.metrics.accuracy()函数生成精度值
```

```
for j in range(num_updates):
#以 j 为循环控制指标,对上述进行 num_updates 次梯度更新
task_accuraciesb.append(tf.contrib.metrics.accuracy(tf.argmax(tf.nn.softmax(task_
outputbs[j]), 1), tf.argmax(labelb, 1)))
#计算测试集上的精度
                    task_output.extend([task_accuracya, task_accuraciesb])
#为 task_output 拓展内容[task_accuracya, task_accuraciesb]
                return task_output
#返回 task_output 的最终结果,task_metalearn()函数的定义到此结束
```

为更好地解释最优化方法,此模块原始代码的剩余部分将在 5.4 节继续分析讨论。

5.3　算法思想

5.3.1　输入层的权值

作为深度学习模型的拓展应用,元学习网络的构造也依赖于权值的生成。在 MAML 算法程序中,主要是借助 tf.Variable()函数生成了网络权值。此函数用于创建张量形式的变量,可以创建任意形状和类型的张量 Variable,具体用法为 tf.Variable(初始值,变量名),其中变量名可以缺省。

在 MAML 算法程序中,tf.Variable()函数的变量名默认为 weights 权值张量,其初始值则通过 tf.truncated_normal()函数得到。单词 truncated 有截断的意思,顾名思义,此函数可以截断地生成符合正态分布的随机数。所谓“截断”地生成,是指如果生成值与均值的差值大于两倍标准差,则截断当下的生成过程,重新生成随机数。该函数用法为 tf.truncated_normal(shape, mean, stddev),其中 shape 是生成张量的维度,mean 是均值,stddev 是 standard deviation 的简写,即标准差。

网络构造算法,首先是构造权值,包括输入层权值和隐含层权值的构造,其核心代码注解如下。

```
  def construct_fc_weights(self):
#定义 construct_fc_weights()函数,输入参数为实例对象本身,用于为前向网络构造权值
        weights = {}
#初始化权值系数集合
        weights['w1'] = tf.Variable(tf.truncated_normal([self.dim_input, self.dim_hidden
[0]], stddev = 0.01))
#借助 tf.Variable()函数生成网络的权值 weights['w1']
#将初始值定义为 tf.truncated_normal([self.dim_input, self.dim_hidden[0]], stddev = 0.01))
        weights['b1'] = tf.Variable(tf.zeros([self.dim_hidden[0]]))
#以 tf.zeros([self.dim_hidden[0]])为初始值,借助 tf.Variable()函数生成网络的权值 weights['b1']
#输入层构造完毕
```

5.3.2 隐含层的权值

本节将完成构造隐含层的权值。

```
        for i in range(1,len(self.dim_hidden)):
#循环控制指标 i 将从 1 增加到 len(self.dim_hidden)
            weights['w' + str(i + 1)] = \
tf.Variable(tf.truncated_normal([self.dim_hidden[i - 1], self.dim_hidden[i]], stddev = 0.01))
#借助 tf.Variable()函数生成网络的权值 weights['w' + str(i + 1)]
#将初始值定义为 tf.truncated_normal([self.dim_hidden[i - 1], self.dim_hidden[i]], stddev =
0.01)
            weights['b' + str(i + 1)] = \
tf.Variable(tf.zeros([self.dim_hidden[i]]))
#以 tf.zeros([self.dim_hidden[i]])为初始值,生成网络的权值 weights['b' + str(i + 1)]
        weights['w' + str(len(self.dim_hidden) + 1)] = \
tf.Variable(tf.truncated_normal([self.dim_hidden[ - 1], self.dim_output], stddev = 0.01))
#借助 tf.Variable()函数生成网络的权值 weights['w' + str(len(self.dim_hidden) + 1)]
#将初始值定义为 tf.truncated_normal([self.dim_hidden[ - 1], self.dim_output], stddev = 0.01)
        weights['b' + str(len(self.dim_hidden) + 1)] = \
tf.Variable(tf.zeros([self.dim_output]))
#以 tf.zeros([self.dim_output])为初始值,生成网络的权值['b' + str(len(self.dim_hidden) + 1)]
        return weights
#construct_fc_weights()函数的定义到此结束,返回值为 weights
#接下来将构造前向网络
```

5.3.3 网络构造算法

权值构造已完成,现在,主要借助正则化函数 normalize()和 tf.matmul()实现网络构造。normalize()函数用于将输入数据调整到一定范围,使其幅值可控,便于进行后续处理。tf.matmul()函数用于完成网络构造过程中的矩阵乘法。

本节先完成前向网络的构造,然后在 5.3.4 节和 5.3.5 节进行拓展。

```
  def forward_fc(self, inp, weights, reuse = False):
#定义 forward_fc()函数,用于构造前向网络
        hidden = normalize(tf.matmul(inp, weights['w1']) + weights['b1'], activation = tf.
nn.relu, reuse = reuse, scope = '0')
#利用输入层 inp 及其权值,得到隐含层的输入
#tf.matmul()函数用于实现矩阵乘法
        for i in range(1,len(self.dim_hidden)):
#循环控制指标 i 将从 1 增加到 len(self.dim_hidden)
            hidden = normalize(tf.matmul(hidden, weights['w' + str(i + 1)]) + weights['b' +
str(i + 1)], activation = tf.nn.relu, reuse = reuse, scope = str(i + 1))
#利用隐含层的输入及其权值,得到隐含层的输出,即输出层的输入
```

```
        return tf.matmul(hidden, weights['w' + str(len(self.dim_hidden) + 1)]) + \
weights['b' + str(len(self.dim_hidden) + 1)]
#利用隐含层的输出及输出层的权值,得到最终输出
#forward_fc()函数的定义到此结束,返回值为最终的输出
```

5.3.4　从卷积层拓展

元学习网络构造完成后,接下来开始从卷积层拓展网络。此时用到了 TensorFlow 中的高级科学计算包 tf. contrib. layers. xavier_initializer,其包含有很多函数的高级封装。在 MAML 算法程序中,主要用到 tf. contrib. layers. xavier_initializer( )函数和 tf. contrib. layers. xavier_initializer_conv2d( )函数。

tf. contrib. layers. xavier_initializer( )函数的用法为

```
xavier_initializer(权值分布, seed, dtype = tf.float32)
```

其中,权值分布可以是均匀分布 uniform 或正态分布 normal 等,seed 是用来生成随机数的种子,数据类型 dtype 是浮点数型 tf. float32。

此函数主要用于网络权值的初始化,是普通初始化器。

tf. contrib. layers. xavier_initializer_conv2d( )函数在此基础上使用了二维卷积的初始化器。

现在开始从卷积层拓展网络,核心代码如下。

```
def construct_conv_weights(self):
#定义 construct_conv_weights()函数,输入参数为实例对象本身,用于卷积层的拓展
        weights = {}
#初始化权值系数集合
        dtype = tf.float32
#设置数据类型为 tf.float32
        conv_initializer = \
tf.contrib.layers.xavier_initializer_conv2d(dtype = dtype)
#调用 tf.contrib.layers.xavier_initializer_conv2d()函数,得到初始化权值 conv_initializer
        fc_initializer = \
tf.contrib.layers.xavier_initializer(dtype = dtype)
#调用 tf.contrib.layers.xavier_initializer()函数,得到初始化权值 fc_initializer
        k = 3
#设置参数 k 值为 3
        weights['conv1'] = tf.get_variable('conv1', [k, k, self.channels, self.dim_hidden],
initializer = conv_initializer, dtype = dtype)
#调用 tf.get_variable()函数,以 conv_initializer 为初始化器,获取类型为 tf.float32 的权值,
#权重矩阵 weights['conv1']的形式为[k, k, self.channels, self.dim_hidden]
#  此函数的用法为 tf.get_variable(name, shape, dtype, initializer)
```

```
        weights['b1'] = tf.Variable(tf.zeros([self.dim_hidden]))
# 借助 tf.Variable()函数生成第 1 个卷积层的权值 weights['b1']
        weights['conv2'] = tf.get_variable('conv2', [k, k, self.dim_hidden, self.dim_
hidden], initializer = conv_initializer, dtype = dtype)
# 调用 tf.get_variable()函数,并以 conv_initializer 为初始化器
# 获取类型为 tf.float32 的权值
# 权重矩阵 weights['conv2']的形式为[k, k, self.channels, self.dim_hidden]
        weights['b2'] = tf.Variable(tf.zeros([self.dim_hidden]))
# 借助 tf.Variable()函数生成第 2 个卷积层的权值 weights['b2']
        weights['conv3'] = tf.get_variable('conv3', [k, k, self.dim_hidden, self.dim_
hidden], initializer = conv_initializer, dtype = dtype)
# 调用 tf.get_variable()函数,并以 conv_initializer 为初始化器
# 获取类型为 tf.float32 的权值
# 权重矩阵 weights['conv3']的形式为[k, k, self.channels, self.dim_hidden]
        weights['b3'] = tf.Variable(tf.zeros([self.dim_hidden]))
# 借助 tf.Variable()函数生成第 3 个卷积层的权值 weights['b3']
        weights['conv4'] = tf.get_variable('conv4', [k, k, self.dim_hidden, self.dim_
hidden], initializer = conv_initializer, dtype = dtype)
# 调用 tf.get_variable()函数,并以 conv_initializer 为初始化器
# 获取类型为 tf.float32 的权值
# 权重矩阵 weights['conv4']的形式为[k, k, self.channels, self.dim_hidden]
        weights['b4'] = tf.Variable(tf.zeros([self.dim_hidden]))
# 借助 tf.Variable()函数生成第 4 个卷积层的权值 weights['b4']
```

至此,已经完成对 4 个卷积层的拓展。接下来,还需要增加第 5 个卷积层,同时需要考虑数据集是 Mini-ImageNet 数据集或 Omniglot 的情形,核心代码如下。

```
  if FLAGS.datasource == 'miniimagenet':
            # 如果数据集为 Mini-ImageNet,那么
            weights['w5'] = tf.get_variable('w5', [self.dim_hidden * 5 * 5, self.dim_
output], initializer = fc_initializer)
# 调用 tf.get_variable()函数,以 conv_initializer 为初始化器,获取类型为 tf.float32 的权值,
权重矩阵 weights['conv5']的形式为[k, k, self.channels, self.dim_hidden]
            weights['b5'] = tf.Variable(tf.zeros([self.dim_output]), name = 'b5')
# 借助 tf.Variable()函数生成第 5 个卷积层的权值 weights['b5']
        else:
# 如果数据集为 Omniglot,那么
            weights['w5'] = \
tf.Variable(tf.random_normal([self.dim_hidden, self.dim_output]), name = 'w5')
# 此时,首先调用 tf.random_normal()函数
# 从服从指定正态分布的序列中随机取出[self.dim_hidden, self.dim_output]个数值
# 借助 tf.Variable()函数生成权重矩阵 weights['w5']
            weights['b5'] = tf.Variable(tf.zeros([self.dim_output]), name = 'b5')
# 借助 tf.Variable()函数生成第 5 个卷积层的权值 weights['b5']
```

```
        return weights
#construct_conv_weights()函数的定义到此结束,共拓展了 5 个卷积层
```

5.3.5　从隐含层拓展

从代码文件 utils.py 调用 conv_block()函数,可以实现元学习网络隐含层的拓展。同时使用到的还有 tf.reduce_mean()函数,用于计算张量第四个隐含层 hidden4 在 tensor 中的[1, 2]维度上的平均值,实现降维,隐含层拓展的核心代码如下。

```
    def forward_conv(self, inp, weights, reuse = False, scope = ''):
        #定义 forward_conv()函数,用于隐含层的拓展
        channels = self.channels
# 将 self.channels 赋值给 channels,以便简化下一行代码中的调用形式
        inp = tf.reshape(inp, [-1, self.img_size, self.img_size, channels])
#调用 tf.reshape()函数,将输入张量形状改为[-1, self.img_size, self.img_size, channels]
        hidden1 = conv_block(inp, weights['conv1'], weights['b1'], reuse, scope + '0')
#调用代码文件 utils.py 里的 conv_block()函数,拓展第 1 个隐含层
        hidden2 = conv_block(hidden1, weights['conv2'], weights['b2'], reuse, scope + '1')
#调用代码文件 utils.py 里的 conv_block()函数,拓展第 2 个隐含层
        hidden3 = conv_block(hidden2, weights['conv3'], weights['b3'], reuse, scope + '2')
#调用代码文件 utils.py 里的 conv_block()函数,拓展第 3 个隐含层
        hidden4 = conv_block(hidden3, weights['conv4'], weights['b4'], reuse, scope + '3')
#调用代码文件 utils.py 里的 conv_block()函数,拓展第 4 个隐含层
        if FLAGS.datasource == 'miniimagenet':
# 如果数据集为 Mini-ImagNnet,那么
            hidden4 = tf.reshape(hidden4, [-1, np.prod([int(dim) for dim in hidden4.get_
shape()[1:]])])
#调用 reshape()函数,将张量 hidden4 的形状调整为[-1, np.prod([int(dim) for dim in
hidden4.get_shape()[1:]])]
        else:
#否则
            hidden4 = tf.reduce_mean(hidden4, [1, 2])
#调用 tf.reduce_mean()函数,
#计算张量 hidden4 在 tensor 中的[1, 2]维度上的平均值,实现降维
        return tf.matmul(hidden4, weights['w5']) + weights['b5']
#forward_conv()函数的定义到此结束,返回值为第 5 个隐含层的输入
```

5.4　最优化方法

5.4.1　学习日志的拓展

在 5.2.2 节描述的元学习过程与人类的学习过程比较相似,不仅体现了元学习系统为

学习过程所做的准备工作(可以理解为在入学环节大脑的初始化),而且体现了学习过程的预习环节,即通过一次梯度下降,得到递归参数 fast_weights,以用于后续的快速梯度下降。核心学习环节在 5.2.3 节体现为梯度更新的循环执行,每次更新后均会重新计算 fast_weights。为避免在未来面临新任务时重新训练,需要巩固在学习过程中获取的知识。元学习系统提供了学习日志拓展模块,其核心代码如下。

```
#学习笔记整理环节:接续 5.2.3 节,拓展学习日志的内容
            if FLAGS.norm is not 'None':
#如果 FLAGS.norm 非空
          unused = task_metalearn((self.inputa[0], self.inputb[0], self.labela[0], self.
labelb[0]), False)
#调用 task_metalearn()函数,初始化 unused
            out_dtype = [tf.float32, [tf.float32] * num_updates, tf.float32, [tf.float32]
* num_updates]
#输出类型约定为[tf.float32, [tf.float32] * num_updates, tf.float32, [tf.float32] * num_
updates]
            if self.classification:
#如果 self.classification 为 True
                out_dtype.extend([tf.float32, [tf.float32] * num_updates])
#为 out_dtype 拓展类型[tf.float32, [tf.float32] * num_updates]
            result = tf.map_fn(task_metalearn, elems = (self.inputa, self.inputb, self.
labela, self.labelb), dtype = out_dtype, parallel_iterations = FLAGS.meta_batch_size)
#调用高阶函数 tf.map_fn(),记录学习日志 result
```

5.4.2 日志读取应用

高阶函数的英文翻译是 high-level function,此处将 task_metalearn()函数当作参数传入,实现学习日志的拓展。此函数还可以实现其他有趣、有用的操作,感兴趣的读者请自行尝试探索。在实际应用中,可以通过如下代码读取和应用学习日志。

```
#第一步,读取学习日志
            if self.classification:
#如果 self.classification 为 True
                outputas, outputbs, lossesa, lossesb, accuraciesa, \
accuraciesb = result
#从学习日志 result 中读取 outputas、outputbs、lossesa、lossesb 以及 accuraciesa
#和 accuraciesb
            else:
#否则
                outputas, outputbs, lossesa, lossesb = result
#从学习日志 result 中读取 outputas、outputbs、lossesa 和 lossesb

#第二步,应用学习日志
```

```
        if 'train' in prefix:
# 请注意函数中的 prefix = 'metatrain_'
            self.total_loss1 = total_loss1 = tf.reduce_sum(lossesa) / tf.to_float\
(FLAGS.meta_batch_size)
# 计算训练总损失 total_loss1,并保存为 self.total_loss1
            self.total_losses2 = total_losses2 = \
[tf.reduce_sum(lossesb[j]) / tf.to_float(FLAGS.meta_batch_size) for j in range(num_
updates)]
# 计算测试总损失 total_losses2,并保存为 self.total_losses2
            self.outputas, self.outputbs = outputas, outputbs
# outputas、outputbs 分别保存为 self.outputas、self.outputbs
            if self.classification:
# 如果 self.classification 为 True
                self.total_accuracy1 = total_accuracy1 = \
tf.reduce_sum(accuraciesa) / tf.to_float(FLAGS.meta_batch_size)
# 计算总体训练精度 total_accuracy1,并保存为 self.total_accuracy1
                self.total_accuracies2 = total_accuracies2 = \
[tf.reduce_sum(accuraciesb[j]) / tf.to_float(FLAGS.meta_batch_size) for j in range(num_
updates)]
# 计算总体训练精度 total_accuracies2,并保存为 self.total_accuracies2
```

5.4.3　优化器的拓展

完成训练的元学习模型可以直接应用于新任务，并在新任务的自主学习过程中得到进一步拓展。相对于新任务而言，该模型可以理解为预训练模型，可以通过学习日志得到 self.pretrain_op，其对应代码为 self.pretrain_op = tf.train.AdamOptimizer(self.meta_lr).minimize(total_loss1)。顾名思义，tf.train.AdamOptimizer()函数采用的最优化方法是 Adam 优化算法，此算法在训练过程中会引入二次方梯度校正。此处的优化目标是通过寻找 self.meta_lr 的一个全局最优点，使得 total_loss1 最小化。

Adam 优化算法是一种自适应动量的随机优化方法，被认为是梯度下降算法的拓展。梯度下降是一个点用最快的方式奔向最低位置，而 Adam 优化算法在此基础上拓展了更新步长的方式。此外，MAML 算法程序中还调用了 optimizer.compute_gradients()函数进行梯度计算，此函数的用法为

```
compute_gradients(loss, var_list = None, gate_gradients = GATE_OP, aggregation_method =
None, colocate_gradients_with_ops = False, grad_loss = None)
```

其中，除第一个参数 loss 之外，其他参数均可缺省。优化方法拓展的核心代码如下。

```
            if FLAGS.metatrain_iterations > 0:
# 如果元训练迭代次数大于 0
```

```
                optimizer = tf.train.AdamOptimizer(self.meta_lr)
#寻找 self.meta_lr 的一个全局最优点,作为优化器 optimizer
                self.gvs = gvs = \
optimizer.compute_gradients(self.total_losses2[FLAGS.num_updates - 1])
#计算 self.total_losses2[FLAGS.num_updates - 1]的梯度,记为 gvs,并保存到 self.gvs 中
```

5.4.4 优化器的应用

在 5.4.3 节中拓展了优化器 optimizer,即借助 optimizer.apply_gradients()函数,调用 apply_gradients 模块,完成张量 gvs 的梯度计算,并用于梯度更新。其中,张量 gvs 是通过调用 tf.clip_by_value()函数生成的。此函数用法为 tf.clip_by_value(tensor, clip_value_min, clip_value_max, name=None)。单词 clip 有修剪的含义,顾名思义,即设置修剪区间,把张量的元素均修剪到固定区间内。如果一个元素比最小值 clip_value_min 小,则替换为最小值;如果比最大值 clip_value_max 大,则替换为最大值。

现在开始优化器 optimizer 的应用,核心代码如下。

```
if FLAGS.datasource == 'miniimagenet':
#如果数据集为 Mini-ImageNet
                    gvs = [(tf.clip_by_value(grad, -10, 10), var) for grad, var in gvs]
#调用 tf.clip_by_value()函数,参考区间[-10, 10]修剪 grad,与 gvs 中的变量组合,得到张量 gvs
                self.metatrain_op = optimizer.apply_gradients(gvs)
#调用优化器 optimizer 中的 apply_gradients 模块,可以完成张量 gvs 的梯度计算
        else:
#否则
            self.metaval_total_loss1 = total_loss1 = tf.reduce_sum(lossesa) / tf.to_\
float(FLAGS.meta_batch_size)
#计算训练集上的平均损失,记为 total_loss1,并保存到 self.metaval_total_loss1
            self.metaval_total_losses2 = total_losses2 = [tf.reduce_sum(lossesb[j]) /
tf.to_float(FLAGS.meta_batch_size) for j in range(num_updates)]
#计算测试集上的平均损失,记为 total_losses2,并保存到 self.metaval_total_losses2

            if self.classification:
#如果 self.classification 为 True
                self.metaval_total_accuracy1 = total_accuracy1 = \
tf.reduce_sum(accuraciesa) / tf.to_float(FLAGS.meta_batch_size)
#计算训练集上的平均精度,记为 total_accuracy1 并保存到 self.metaval_total_accuracy1
                 self.metaval_total_accuracies2 = total_accuracies2 = [tf.reduce_sum\
(accuraciesb[j]) / tf.to_float(FLAGS.meta_batch_size) for j in range(num_updates)]
#计算测试集上的平均精度,记为 total_accuracies2 并保存到 self.metaval_total_accuracy1
```

5.4.5 显示优化过程

Tensorflow 的优化过程可以借助 tensorboard 实现,主要是模型训练过程中参数的可

视化。在 MAML 算法程序中，主要用到了 tf. summary()函数的 scalar 模块。单词 scalar 有标量的含义，顾名思义，tf. summary. scalar()函数可用于显示标量信息，具体用法为

```
tf.summary.scalar(tags, values, collections = None, name = None)
```

MAML 算法程序中的核心代码仅提供优化结果的显示功能，核心代码如下。事实上，此函数还可以与其他 tf. summary()函数结合，添加变量到直方图中，生成标量图。

```
  tf.summary.scalar(prefix + 'Pre - update loss', total_loss1)
# 调用 tf.summary.scalar()函数，显示标量信息 prefix + 'Pre - update loss'，数值为 total_loss1
        if self.classification:
# 如果 self.classification 为 True
            tf.summary.scalar(prefix + 'Pre - update accuracy', total_accuracy1)
# 调用 tf.summary.scalar()函数，显示标量信息 prefix + 'Pre - update accuracy', 数值为 total_accuracy1

        for j in range(num_updates):
# 执行 num_updates 次循环
            tf.summary.scalar(prefix + 'Post - update loss, step ' + str(j + 1), total_
losses2[j])
# 调用 tf.summary.scalar()函数，显示标量信息(prefix + 'Post - update loss, step' + str(j + 1))，数值
# 为 total_losses2[j]
            if self.classification:
# 如果 self.classification 为 True
                tf.summary.scalar(prefix + 'Post - update accuracy, step ' + str(j + 1),
total_accuracies2[j])
# 调用 tf.summary.scalar()函数，显示标量信息(prefix + 'Post - update accuracy, step ' + str
(j + 1))，数值为 total_accuracies2[j]
```

5.5　元优化机制

5.5.1　虚拟环境的拓展

在第 1 章安装配置 Anaconda 开发平台，并在第 2 章演示如何在该平台下编辑 Python 程序。这些准备工作有助于理解元优化过程中使用的科学计算包，对调试 MAML 算法的原始代码也是有帮助的。之后，在第 3 章拓展完成 Python＋PyCharm 的环境配置，至此已完成代码调试的大部分准备工作。此外，对于 MAML 算法程序的运行，在 PyCharm 中配置环境也是必不可少的。

相关环境配置过程如下。

(1) 在 PyCharm 中打开设置 Settings，如图 5-1 所示。

(2) 选择编译器 Python Interpreter 选项，如图 5-2 所示。

(3) 选择 Add Python Interpreter 选项，如图 5-3 所示。

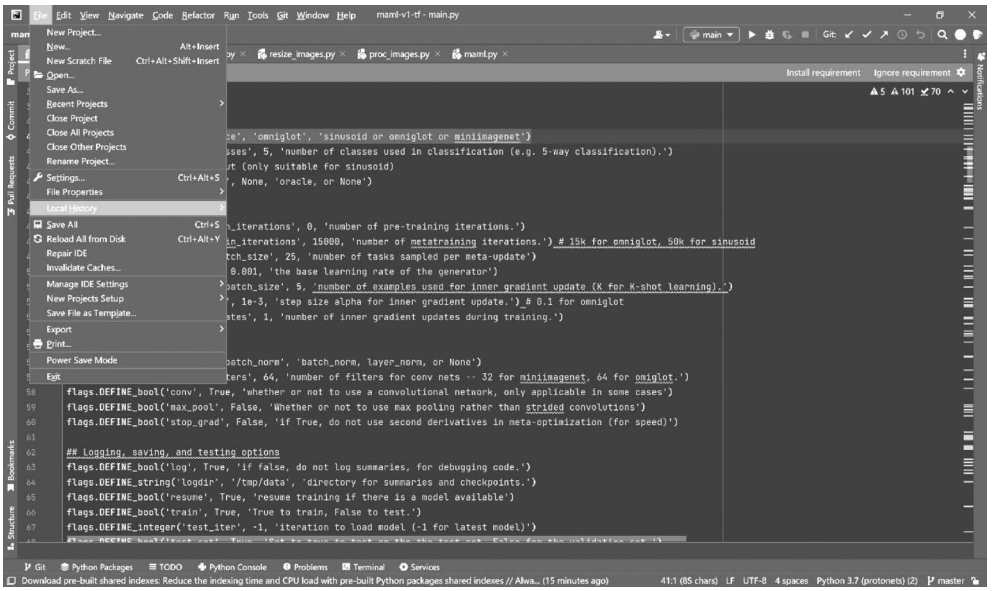

图 5-1　在 PyCharm 中打开设置 Settings

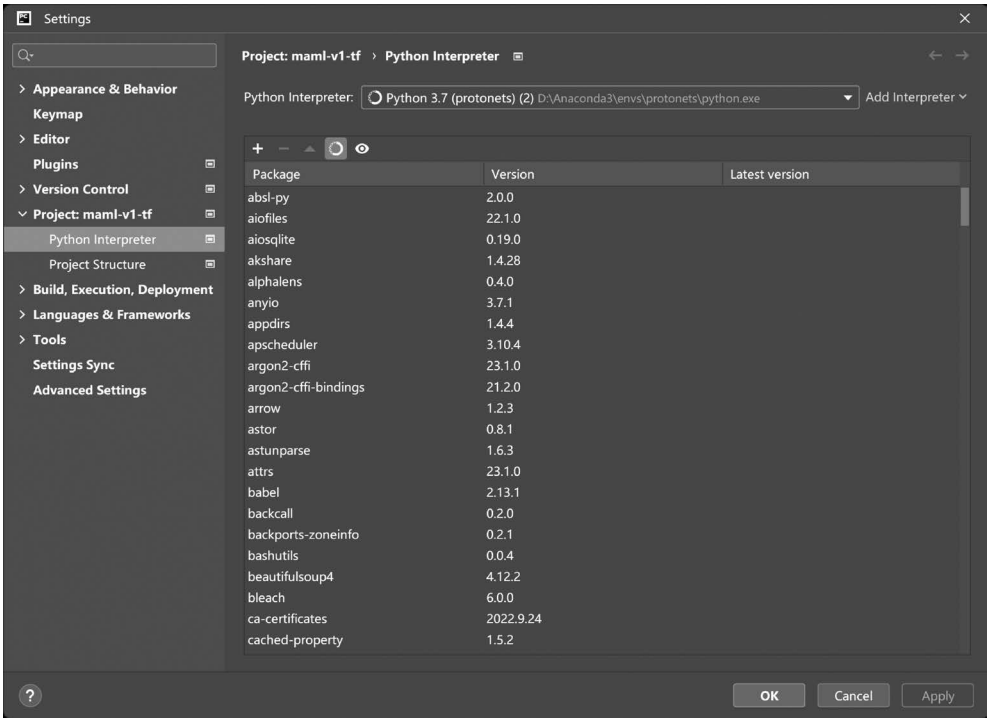

图 5-2　选择编译器 Python Interpreter 选项

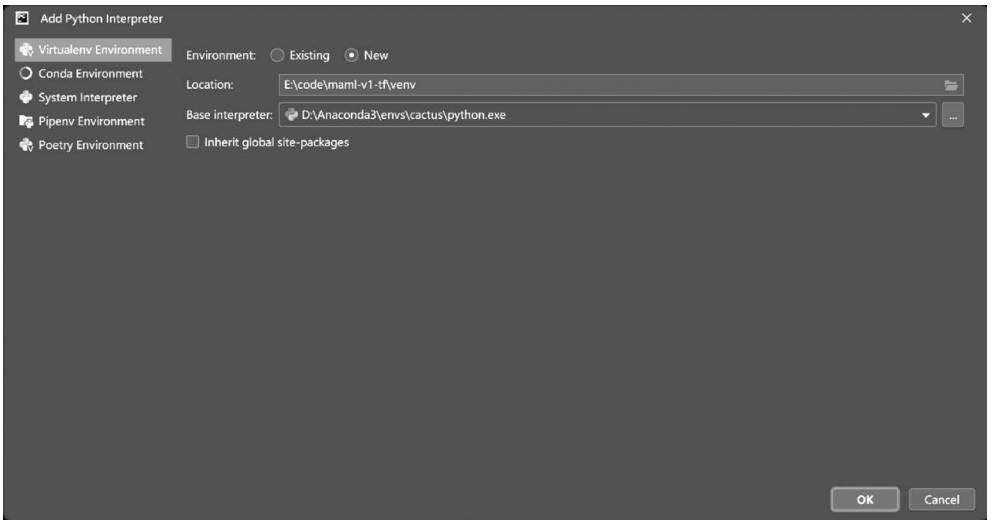

图 5-3　选择 Add Python Interpreter 选项

(4) 选择 Existing 选项,返回图 5-4 所示的界面。

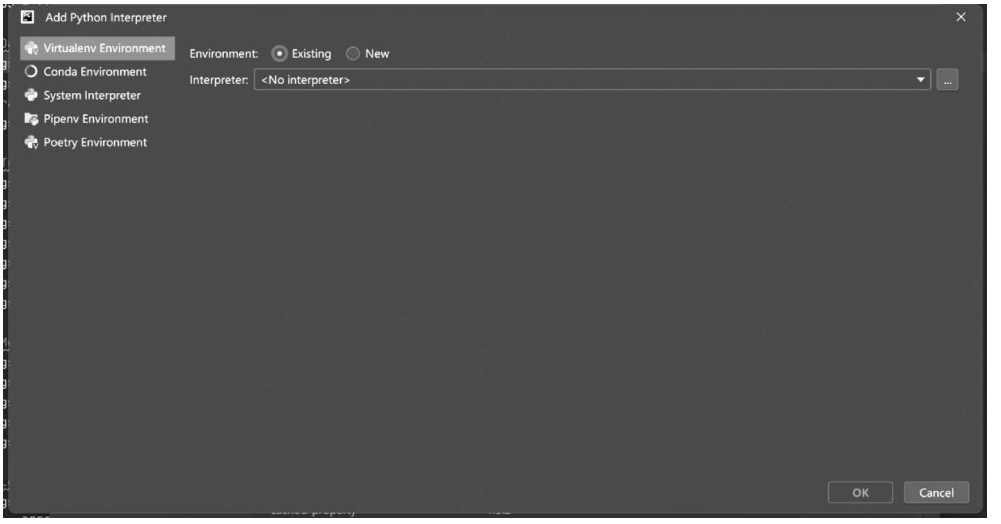

图 5-4　返回原界面

(5) 单击最右边 3 个点,查看当前环境,如图 5-5 所示。

(6) 找到创建的虚拟环境中的 python. exe。例如,笔者的环境是 ProtoNets,找到该目录下的 python. exe 作为代码编译器,如图 5-6 所示。

(7) 单击 OK 按钮进入创建的虚拟环境。

上述配置完成后,就可以下载安装需要补充的科学计算包,完成虚拟环境的拓展了。为帮助读者快速上手,将需要安装的科学计算包整理成文件 requirement. txt,包含在随书附赠的 code 目录中,这是元学习过程中所需要的配置包。事实上,该文件可以放在任何位置,

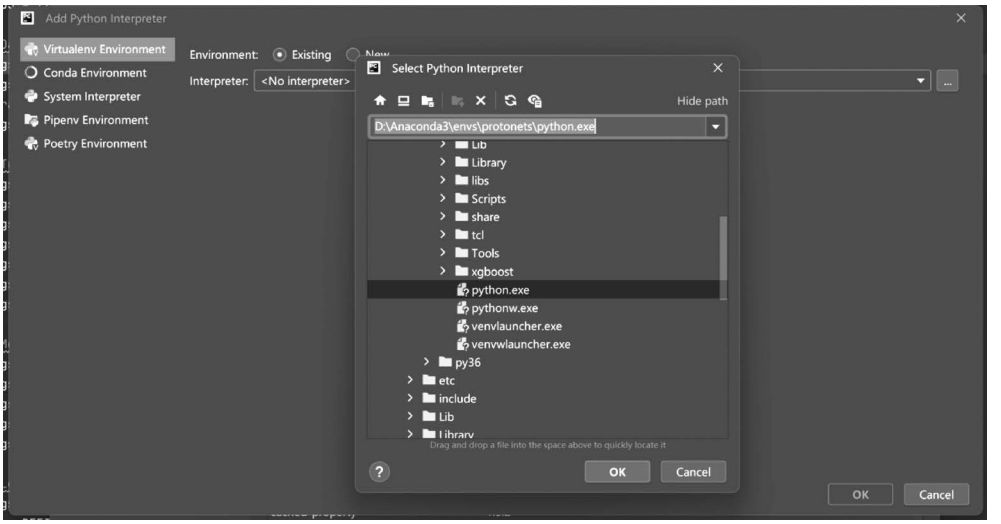

图 5-5　查看当前环境

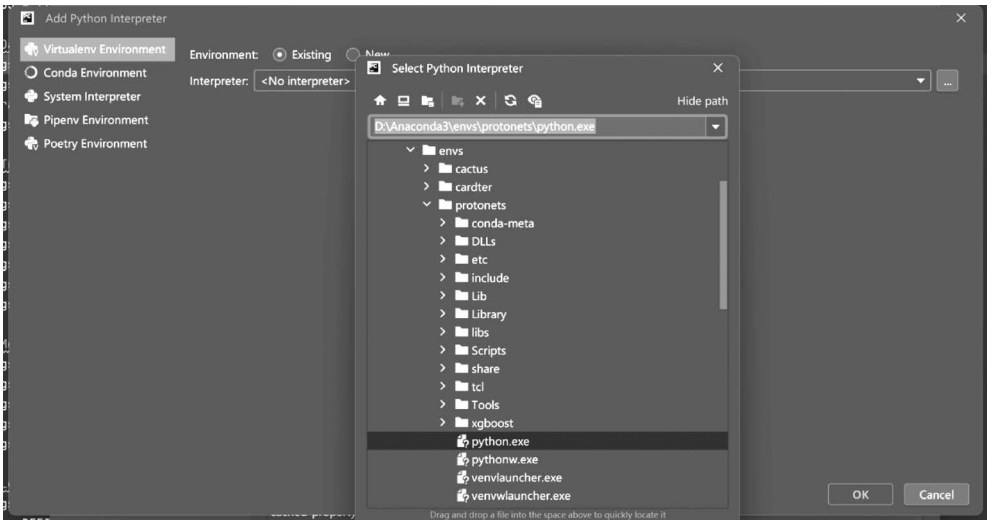

图 5-6　虚拟环境中的 python. exe

也可以放在桌面。考虑读者计算机环境配置的差异，在代码调试过程中可能还需要安装其他少量的科学计算机包。

虚拟环境的拓展，需要安装好 Anaconda 以及 Python 后，才能完成，具体操作如下。

（1）拓展虚拟环境，其实就是创建新的虚拟环境，使用快捷键 Win＋R 打开 CMD(命令行窗口)，然后执行路径操作(cd C:\Users\86155\Desktop)，进入桌面，并输入 conda create -n mymaml 即可。其中，mymaml 为虚拟环境的名称，也可以根据个人喜好选择。

（2）在新的虚拟环境中，安装配置包 requirement. txt。

（3）使用命令 conda activate mymaml 激活虚拟环境，输入 pip install -r requirement.

txt，按 Enter 键执行即可。

5.5.2　模块代码的调试

至此，已经完整分析了元学习中的联合训练问题、任务构建问题、过程建模问题、输入输出问题、应用拓展问题 5 个相关问题，并以问题描述、建模思路、算法思想、最优化方法和元优化机制为主线，将模型无关元学习的开源代码对应分割为一系列子模块，提供了较为详细的注解。为引导读者快速上手，跑通 MAML 算法代码，现对两个模块的代码进行优化。

经过本书的阅读，读者已成长为学者。从现在开始，可以研究并思考 MAML 算法模型的其他细节，并在此基础上共同探讨如何使现有的元学习代码更加完善。

如果将代码文件 data_generator.py 中的 num_val 修改为 5、num_train 修改为 config.get('num_train', 20)-num_val，是否需要对应地同时修改其他模块？请读者尝试自行思考。

相关原始代码如下。

```
num_val = 100
num_train = config.get('num_train', 1200) - num_val
```

修改后，对应代码如下。

```
num_val = 5
num_train = config.get('num_train', 20) - num_val
```

上述修改的初衷：希望在短时间内得到输出结果，所以减少了样本数。

现在，将代码文件 main.py 中优先处理的数据集由 sinusoid 修改为 omniglot。这样修改之后，是否需要对应地同时修改其他模块？如果希望处理 miniimagenet，应如何修改？经过某些特定的修改，有没有可能用 sinusoid 数据集快速得到结果，请读者尝试自行思考。

相关原始代码如下。

```
flags.DEFINE_string('datasource', 'sinusoid', 'sinusoid or omniglot or miniimagenet')
```

修改后，对应代码如下。

```
flags.DEFINE_string('datasource', 'omniglot', 'sinusoid or omniglot or miniimagenet')
```

上述修改以 Omniglot 数据集为例，其他数据集的修改过程类似。打开代码文件 main.py，如图 5-1 所示，单击左上角的绿色三角按钮即可进行环境配置。作为最后的彩蛋，代码的后续调试过程留给读者完成。

对于读者在调试过程中遇到的问题，我们将以读者群讨论或资源更新的方式提供解答。

5.5.3 元任务的理解

本书主要介绍了与模型无关的元学习算法，即 MAML 算法。通过对元训练、元测试的代码分析，帮助读者进一步理解了 MAML 算法中的元优化机制。

如果将元学习的训练模型表示为 f，将观测 x 映射到输出 y。不同于机器学习以数据集作为输入，元学习问题是以任务集中的数据集进行输入的，这些任务可以理解为元任务。在本书的结尾，将给出元任务、元训练、元测试的数学定义，以此构建元优化理论，希望有抛砖引玉的作用。感兴趣的读者，可以基于这些数学定义，尝试设计新的元学习算法。

基于上述关于 MAML 算法的研究笔记，对元任务及其包含的参数、函数理解如下。

定义 5-1(元任务) 元任务 $T=\{\mathcal{L}(\theta,\mathcal{D}),\rho(x_1),\rho(x_{t+1}|(x_t,y_t)),H\}$ 主要由损失函数 $\mathcal{L}$、初始观测 x_1 上的分布 $\rho(x_1)$、条件分布 $\rho(x_{t+1}|(x_t,y_t))$ 和状态步长 H 组成。其中，$\mathcal{L}$ 以基础学习器的参数 θ 和数据集 $\mathcal{D}$(学习样本)为输入，ρ 是学习样本的分布函数，x_1 是从 $\mathcal{D}$ 中随机抽取的第一个观测数据。

根据马尔可夫理论，条件分布 $\rho(x_{t+1}|(x_t,y_t))$ 在下文中统称为转移分布。元任务中引入转移分布的意义是使任务中的数据分布大致相同，对数据有一定的约束能力。在元任务的定义中设置步长 H 旨在将强化学习问题与非强化学习问题视为一体，即将非强化学习视为特殊的强化学习。对于非强化学习问题，步长 H 即为 1。根据强化学习理论，状态步长的实质为智能体的行动状态长度。

5.5.4 元训练的机制

元学习是以任务为样本单位训练模型的。元学习算法根据元训练集中的任务数据进行训练学习后，更新模型参数，并在测试集上测试从而使模型具备一定的学习能力。判断一个元学习算法是否具备一定的可行性，关键是看其在测试集上是否能快速学习新任务。

假设元学习模型中的任务均服从 $p(\mathcal{T})$ 的分布，元训练机制的定义如下。

定义 5-2(元训练) 元学习模型的训练阶段称为元训练，此时训练的对象为任务样本。不同于传统机器学习，元训练包含内置训练阶段和内置测试阶段，内置训练和内置测试的对象为数据样本。具体而言，在元训练期间，先从任务分布 $p(\mathcal{T},\mathcal{D})$ 中随机抽取 N 个任务 $\{(\mathcal{T}_1,\mathcal{D}_{\mathcal{T}_1}),(\mathcal{T}_2,\mathcal{D}_{\mathcal{T}_2}),\cdots,(\mathcal{T}_i,\mathcal{D}_{\mathcal{T}_i}),\cdots,(\mathcal{T}_N,\mathcal{D}_{\mathcal{T}_N})\}$，构成元训练样本。然后对每个任务 $(\mathcal{T}_i,\mathcal{D}_{\mathcal{T}_i})$，从 $\mathcal{D}_{\mathcal{T}_i}$ 中随机抽取 K 个数据样本，作为任务 $\mathcal{D}_{\mathcal{T}_i}$ 的内置训练数据集 $\mathcal{D}^{tr}_{tr_{\mathcal{T}_i}}$。完成内置训练后，会产生内置训练损失 $\mathcal{L}(\theta,\mathcal{D}^{tr}_{tr_{\mathcal{T}_i}})$，从而根据 $\mathcal{L}(\theta,\mathcal{D}^{tr}_{tr_{-\mathcal{T}_i}})$ 更新模型参数。再从剩余数据中随机抽取 L 个数据样本，作为任务 $\mathcal{D}_{\mathcal{T}_i}$ 的内置测试数据集 $\mathcal{D}^{ts}_{tr_{-\mathcal{T}_i}}$，完成内置测试后，会产生内置测试损失 $\mathcal{L}(\phi_i^*,D^{\text{test}}_{tr\_\mathcal{T}_i})$，最后将 N 个任务的内置测试损失求和、取平均，得到元训练的整体损失 $W(\theta)=\frac{1}{n}\sum\mathcal{L}(\phi_i^*,D^{\text{test}}_{tr\_\mathcal{T}_i})$。当 $W(\theta)$ 收敛到极小值时，元训练结束。

5.5.5　元测试的机制

元学习有 3 种常见的方法，分别是学习有效的距离度量、使用(循环)网络与外部或内部存储器以及明确优化模型参数以进行快速学习。本书研究的 MAML 算法，是一种优化模型参数以实现快速学习的方法。模型无关的意思是 MAML 算法适用范围很广，在任意可以通过梯度下降进行优化训练的模型，均可以用该方法。在定义 5-1 和定义 5-2 的基础上，现在可以给出元测试机制的数学定义。

定义 5-3(元测试)　元学习模型的测试阶段称为元测试，元测试的对象为任务样本，其内置训练和内置测试的对象为数据样本。元测试过程如下：先从任务分布 $p(\mathcal{T},\mathcal{D})$中随机抽取 M 个任务$\{(\mathcal{T}_1,\mathcal{D}_{\mathcal{T}_1}),(\mathcal{T}_2,\mathcal{D}_{\mathcal{T}_2}),\cdots,(\mathcal{T}_j,\mathcal{D}_{\mathcal{T}_j}),\cdots,(\mathcal{T}_M,\mathcal{D}_{\mathcal{T}_M})\}$，构成元测试样本。然后对每个任务$(\mathcal{T}_j,\mathcal{D}_{\mathcal{T}_j})$，从$\mathcal{D}_{\mathcal{T}_j}$中随机抽取 K 个数据样本，作为任务$\mathcal{D}_{\mathcal{T}_j}$的内置训练数据集$\mathcal{D}^{tr}_{ts_{\mathcal{T}_j}}$。完成内置训练，得到损失$\mathcal{L}(\theta,\mathcal{D}^{tr}_{ts_{\mathcal{T}_j}})$并更新模型参数。再从剩余数据中随机抽取 L 个数据样本，作为任务$\mathcal{D}_{\mathcal{T}_j}$的内置测试数据集$\mathcal{D}^{ts}_{ts_{\mathcal{T}_j}}$，完成内置测试，元测试结束。

定义 5-2 和定义 5-3 中的内置训练数据集也称为支持集，内置测试数据集也称为查询集。训练阶段的任务集表示为$\mathcal{D}_{\text{meta-train}}=\{(\mathcal{D}^{tr}_{tr_1},\mathcal{D}^{ts}_{tr_1}),\cdots,(\mathcal{D}^{tr}_{tr_N},\mathcal{D}^{ts}_{tr_N})\}$，$N$ 为训练任务个数，测试阶段的任务集表示为$\mathcal{D}_{\text{meta-test}}=\{(\mathcal{D}^{tr}_{ts_1},\mathcal{D}^{ts}_{ts_1}),\cdots,(\mathcal{D}^{tr}_{ts_M},\mathcal{D}^{ts}_{ts_M})\}$，$M$ 为训练任务个数，每个训练任务集可以表示为$\mathcal{D}^{tr}=\{(x^{tr}_1,y^{tr}_1),\cdots,(x^{tr}_K,y^{tr}_K)\}$，$K$ 为训练样本数。同理每个测试任务集可以表示为$\mathcal{D}^{ts}=\{(x^{ts}_1,y^{ts}_1),\cdots,(x^{ts}_L,y^{ts}_L)\}$，$L$ 为测试样本数。

对非强化学习问题的某个具体任务样本而言，假设其数据集$\mathcal{D}$由 k 个输入、输出对$(x_1,y_1)^k$ 组成，即$\mathcal{D}=\{(x_1,y_1)^k\}$。其中，下标 1 表示行动状态步长 H 为 1。而对于强化学习问题，模型 f 可以通过在每个时刻 t 对应输出的 $\hat{y}_t$ 来生成长度为 H 的样本数据，从而模型生成的轨迹传递给损失函数的数据集为$\mathcal{D}=\{(x_1,\hat{y}_1,\cdots,x_H,\hat{y}_H)^k\}$，其中 k 为训练样本数。

提供上述数学形式的描述，希望可以起到抛砖引玉的作用。期待与读者共同努力，探索 MAML 算法的工程应用，发展新的元学习理论。复杂的数学推导是元学习模型的理论基础，也是算法设计的具体表现形式。目前，我们已经初步实现部分理论的创新，并将通过下一本书分享给读者。