

## 报告和代码整体简要说明

本次深度学习我选择任务 B，使用的框架是 jittor，基于 pygmtools 版本 0.3.3。后续的数据集搭建、训练、测试都是在这个版本上进行的。

本次报告将会从前期准备（主要包括开源工具、论文、开源代码的学习），深度学习框架实际编写（主要包括任务 B 的全部实现流程），实验和数据（主要包括调参、消融实验）三个方面进行介绍。

代码中的 `main.py` 是完整的训练流程，欲测试代码的完整性，可以运行 `python main.py --debug`，将会进行一个 epoch 的训练，并保存训练的日志。默认参数设置在 `utils/parser.py` 文件中，其余关键文件全部在根目录下。实验结果保存在 `checkpoint` 文件夹下。

除此之外，在线例子的复现结果以 `jupyter notebook` 的形式保存在了 `tutorial` 文件夹下，谨供助教按需查阅。

## 1 框架搭建前期准备

在开始项目前，先阅读 pygmtools 的官方文档，了解整个项目整体的框架和需要使用的数据集、相关代码接口等等，以便对整个项目框架搭建有个基本的规划。

python 开发环境配置这里不再赘述。

### 1.1 文档学习和图匹配算法选取

任务要从了解什么是图匹配和方法选取开始，pygmtools 的文档给出了较为详尽的图匹配应用和图匹配流程。整体上，首先要从数据集中提取图的结点、边的特征，这一部分可以利用数据集中已经提供的信息；之后计算亲和度矩阵，并用图匹配求解器来解决问题，这一部分需要学习使用 pygmtools 中的工具。

本次任务我使用 PCA-GM 算法进行图匹配，整个神经网络会内嵌一个 PCA-GM 神经网络，并把它当作一个黑盒来处理。对于每对需要匹配的图片，PCA-GM 方法会先用特征提取器（例如 VGG16）得到带有特征的图片对，之后通过数个隐层，在隐层中计算图卷积和交叉卷积，然后计算亲和度，并通过 Sinkhorn 函数得到双随机矩阵，最终得到图匹配结果，用于后续的损失函数计算，损失函数使用排列损失即可。

### 1.2 开源代码复现

在了解了整个项目的关键点后，对开源代码进行复现，了解现有的数据集和代码 API，以便后续开发。

[在线例子](#)中提供了一个基于预训练模型对两张图片进行匹配的流程展示。开源代码的复现不需要做任何改动，由于使用的是预训练模型的结果，也不需要 gpu 资源，直接在本地配置环境并运行即可。最关键的是通过复现流程理解 GMNet 需要传入什么样的参数，数据集中哪些信息是已经有的，哪些信息是需要通过 pygmtools 的工具得到的，

willow 数据集只能提供图片和关键点信息，这些数据是不足以支持我们直接使用 PCA-GM 的。在线例子使用了 `delaunay_triangulation` 的方法，将图片中的关键点进行三角剖分，把图片划分成一系列的三角形区域，以便后续的图匹配网络调用。之后，在线例子提供了各种用于图匹配的神经网络算法，这些算法所使用的神经网络接口略有差异，需要我们对数据集中提供的关键点做一定的计算，之后再传入要使用的 `GraphMatching`

网络，以提供一个端到端的训练流程。在线例子最后提供了一个很规范的 GMNet，我的网络则是在此基础上修改（在线例子不支持带 batch 的数据，下文会详述）。至此，我已经对接下来的开发工作有了必要的准备。

## 2 完整深度学习框架搭建

这是任务 B 的主要部分，我将按照实际完成作业的顺序，逐步介绍我对开源代码的理解和使用，遇到的 bug 及修复、处理或避开的方法。

### 2.1 数据集预处理

基础的 WillowObject 数据集中只提供了图片和关键点信息，并不完全适配图匹配任务。且其数据集中五个类别的图片数量不均衡，也会对匹配精度造成一定的影响。图匹配还需要在各个类别内成对拿出用于匹配的图片，并枚举同一类内用于训练的所有图片的组合。所幸，所有这些功能都已经在 pygmtools 中实现了。

为了能够使用 pygmtools 中 benchmark 提供的接口，在现有的 WillowObject 的基础上还需要一些预处理。这一预处理工作只需要新建一个 `pygmtools.dataset.WillowObject` 对象，调用其 `process` 函数即可。这个预处理会成一个 json 文件，包括训练和测试集的划分，为了保证训练集的平衡，每一类选出 20 张图片用于训练，其余的图片则留做测试，测试集各类图片是不平衡的。关键的信息会保存在 `data-(256,256).json` 中，用于后续的数据集加载。

### 2.2 图匹配数据集实现

我实现的数据集 `GraphPair` 继承自 jittor 的 `Dataset` 类，其中会内嵌一个 pygmtools 的 `Benchmark` 对象。在数据集初始化时，除了配置好各种参数外，还会调用 `load_data_list` 函数，通过调用内嵌的 `Benchmark` 对象的 `get_id_combination` 函数，得到所有的图片组合的列表。由于要匹配所有的类，该列表内保存了 5 个类各自的图片组合的列表，在将各个图片对进行类别标记后，再把列表展开，逐个 `append` 到数据集的 `data` 属性中，就完成了数据集的初始化。

数据集的 `getitem` 方法则比较关键。对于传入的 `index`，直接通过数据集初始化时生成的 `data` 列表提取相应的图片对名称，然后调用内嵌的 `Benchmark` 对象的 `get_data` 函数，得到相应的图片、关键点信息等数据。注意，在生成训练集时，需要将 `shuffle` 参数设置为 `True`，这样才能保证关键点之间的对应关系具有随机性，否则得到的匹配矩阵的 `groundtruth` 对角线元素会全部为 1，这样虽然仍然可以正常训练，但是模型会被引导着向预测单位矩阵的方向进行，就失去训练的意义了。此外，得到的图片是无法直接使用的，需要先进行 `permute` 操作，再归一化到 `[0,1]` 区间。通过 `delaunay_triangulation` 方法处理关键点信息。这里由于 jittor 的 `dataset` 并不支持 scipy 的 `coo_matrix`，所以我把以稀疏矩阵的形式保存的 `groundtruth` 通过 `toarray` 方法转化成 numpy 的 `ndarray`。在进行了上述所有的处理之后，把需要的参数全部返回，以供后续的训练即可。

此外，为了加快数据集加载的速度，避免每次 `getitem` 都要重新读取图片、关键点等信息，并进行计算，我还实现了 `load_data` 方法，如果调用此方法，在数据集初始化时，就会一次性地进行上述所有的数据处理，并把他们存储到 `data` 列表中。实测这一方法在 cpu 环境运行时可以实现一次性加载，但是在 cuda 环境中运行时，会超过内存限制。考虑到这一部分数据集处理并不是整个模型训练开销的瓶颈所在（开销主要在训练上），所以我后续没有再使用这一方法。

## 2.3 网络搭建和训练

本次作业训练使用的网络是根据 pygmtools 在线样例中的网络进行修改得到的。由于在线例子只用了一对图片，等价于 `batchsize` 是 1，其网络结构的实现在提取节点信息时只支持 `batchsize` 等于 1 的数据，这一部分我将其修改成对于整个 `batch` 的数据，循环遍历提取节点信息，将信息整合到 `ndarray` 中，这样得到的节点信息的第一个维度仍然等于 `batchsize`，以保证整个训练流程的一贯性。

整个图匹配网络内置一个 CNN，这个 CNN 输出得到局部和全局的特征后，再通过上采样得到 4D 的 `featuremap`，形为  $(batchsize * 1024 * 256 * 256)$ ，接下来关键在于用 `keypoint` 的位置来从这些 `map` 里面提取信息。`keypoint` 本身是浮点数，需要将其进行 `round` 得到整数，之后特征提取的方法如图所示<sup>1</sup>。

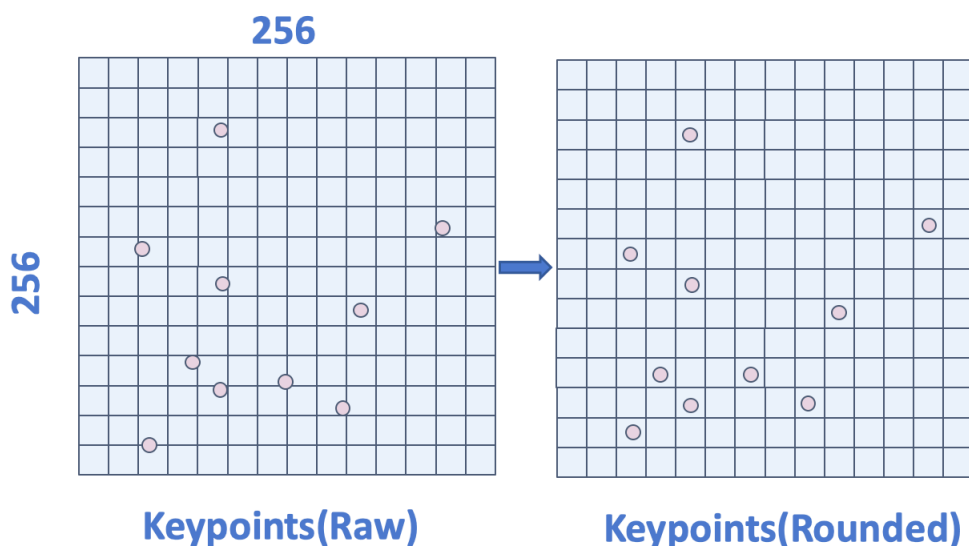


图 1: 使用 `keypoint` 提取 `feature` 信息

上采样之后的 `featuremap` 后两个维度正是 `resize` 之后的图像大小，助教曾经在群里指出，需要交换在线例子中使用的 `roundedkeypoints` 下标。这一交换至关重要。在我实际进行训练的过程中，在交换之前，无论怎么调参和改进，评估环节得到的 `precision`、`recall` 等等参数都在 0.5 到 0.7 之间，我一度认为这就是标准的训练结果。后来经助教指出，在交换之后，才能把各个评估指标提升到 0.8 以上。这一交换的原因是，`keypoint` 标记了一张原始图片在 `x`、`y` 两个方向上的关键点位置信息，如果 `xy` 轴坐标弄反了，实际上提取到的根本就不是特征点的信息，而是没有意义的信息，这样引导模型训练的方向会发生错误，无论多少轮的训练，都没办法提高精度。

在训练 `loss` 的选取上，直接使用 `pygmtools` 提供的 `permutation_loss`，这一 `loss` 也是 PCA-GM 这一图匹配算法所使用的，`pygmtools` 提供的工具已经支持 `batch` 的数据，这里直接将模型的预测结果和 `groundtruth` 做 `loss` 即可。在 `evaluate` 时，则需要把模型的预测结果通过 `hungarian` 函数转换成可解释的预测结果，以便后续调用 `benckmark` 的 `eval` 函数进行评估。

## 2.4 测试和评估

调用 `benchmark` 的 `eval` 函数来进行评估前，需要将通过 `hungarian` 函数得到的预测结果、图片对名字、相应的类别保存成列表并传入 `eval` 函数中，并不需要传入 `groundtruth`。`eval` 函数如何实现在不传入

groundtruth 的情况下进行评估引发了我的好奇。其方法是（在 0.3.3 版本的 pygmttools 中），当我们最开始调用 getdata 拿取数据的时候：

```
1 self.benchmark.get_data(self.data[index][0], shuffle=self.shuffle)
```

如果使用的 benchmark 是测试集 (test)：

```
1 if self.sets == 'test':
2     for pair in id_combination:
3         id_pair = (ids[pair[0]], ids[pair[1]])
4         gt_path = os.path.join(self.gt_cache_path, str(id_pair) + '.npy')
5         if not os.path.exists(gt_path):
6             np.save(gt_path, perm_mat_dict[pair])
```

则会在根据当前的进程 pid 生成缓存路径，并把 groundtruth 以 numpy 数组的形式存储下来，这样 evaluate 时直接在本地读取即可，以加快 evaluate 的速度。这样做也引发了一个问题，由于 groundtruth 只会缓存一次，后续的测试都是用第一次测试时缓存的 groundtruth 进行的，如果在测试集时候也启用了 shuffle：

```
1 if shuffle:
2     random.shuffle(obj_dict['kpts'])
```

得到的关键点匹配的 groundtruth 几乎不可能与第一次一样，造成 evaluate 失败。出现这种失败的明显表现是：如果在某次完整的训练流程中进行了多次 evaluate，发现仅有第一次 evaluate 的结果比较高，但是从第二次开始 precision 和 recall 等等评估指数都稳定在 0.1 左右，无论怎么训练都无法提升。

所以在后续的 evaluate 过程中，我将测试集的 shuffle 参数设置为 False，这样就可以保证每次 evaluate 的 groundtruth 都是一样的，从而能够成功完成评估。

## 3 实验配置和结果分析

### 3.1 实验参数配置简要说明

以下是本次实验的结果，表中列出了调用 eval 之后得到的五类图片匹配结果的平均 precision、recall 和 f1-score。其中 LR 代表 learning rate，WD 代表 weight decay，Pretrain 代表是否使用预训练模型。在开启预训练的情况下，使用的预训练模型是 jittor 官方库提供的 vgg16 预训练模型，而考虑到我们的任务关键在于图匹配，故对于图匹配网络所使用的 PCA-GM，并没有使用任何的预训练参数，即预训练仅仅涉及到整个网络的 CNN 部分。

所有实验的训练轮数均为 100epoch，训练集使用 pygmttools 默认的划分方式，即各个类别都选取 20 张图片，产生 190 个图片对，训练集共有 950 个图片对。测试集包含了剩余所有的图片对，约 5600 个。评估指标中 Best 表示在训练流程中模型所能达到的最高指标，用于展示在此参数设置下模型能够达到的最好性能；而 Final 表示在整个训练全部结束后进行评估得到的最终指标，用于展示此特定参数设置对模型性能的影响。

## 3.2 实验结果分析

实验的最开始，我开启了预训练模型，一方面是为了观察模型正常训练情况下的 loss 表现，一方面也是为了判断比较合理的准确率范围。预训练的效果是比较明显的，无论是从最高准确率，还是从最终准确率来看，都远高于后续并未使用预训练的模型。预训练模型一般通过 10 到 20 个 epoch 就能达到比较高的准确率，但是在后续的训练过程中，准确率反而会下降，说明整个图匹配网络在优化 PCA 模块参数的同时，CNN 模块的性能反而下降，从而带来了整体准确率的降低。不过，从整体的性能表现上来看，使用预训练模型还是能够起到更好的效果。

在不使用预训练模型的情况下，可以适当放大学习率，以供模型更快调优，weight decay 对整体准确率的影响不大，并不是影响性能的关键参数。

参数配置 评估指标	LR 1e-4 WD 1e-4	LR 3e-5 WD 1e-4	LR 5e-5 WD 1e-4	LR 7e-5 WD 1e-4	LR 1e-4 WD 1e-4	LR 1e-4 WD 2e-4
Pretrain	✓	✓	✓	✓	✗	✗
Best P.	.9188	.9172	<b>.9262</b>	.9228	.8250	.8393
Best R.	.9188	.9172	<b>.9262</b>	.9228	.8250	.8393
Best F1.	.9188	.9172	<b>.9262</b>	.9228	.8250	.8393
Final P.	.8046	<b>.9128</b>	.8873	.8711	.8003	.7816
Final R.	.8046	<b>.9128</b>	.8873	.8711	.8003	.7816
Final F1.	.8046	<b>.9128</b>	.8873	.8711	.8003	.7816
参数配置 评估指标	LR 1e-4 WD 3e-4	LR 1e-4 WD 7e-5	LR 1e-4 WD 8e-5	LR 3e-5 WD 1e-4	LR 5e-5 WD 1e-4	LR 7e-5 WD 1e-4
Pretrain	✗	✗	✗	✗	✗	✗
Best P.	.8192	.8170	.8377	.8157	.8335	<b>.8426</b>
Best R.	.8192	.8170	.8377	.8157	.8335	<b>.8426</b>
Best F1.	.8192	.8170	.8377	.8157	.8335	<b>.8426</b>
Final P.	.7965	.8170	<b>.8268</b>	.8073	.7493	.8082
Final R.	.7965	.8170	<b>.8268</b>	.8073	.7493	.8082
Final F1.	.7965	.8170	<b>.8268</b>	.8073	.7493	.8082

表 1: 不同参数配置下的运行结果

## 4 总结与思考

### 4.1 国产框架的使用感受

在本次任务的范围内，jittor 的开发完成度已经足够满足图匹配任务的全部需求，使用体验还是比较友好的。jittor 与 pytorch 相比仅仅是在 API 层面上有所不同，但是在使用上并没有太大的差异，因此在使用上并没有太大的困难。API 的区别主要在于 jittor 中把 dataloader 和 dataset 归并成了一个类，以及将梯度反向传播和参数优化器调优等等函数整合到了一起，集成度较高，在实现 dataset 时就要制定一些原本属于

`dataloader` 的参数，在一定程度上提高了代码的耦合度，不过也并未对深度学习的流程造成任何本质影响，只要熟悉其接口，仍然可以正常使用。

此外，上文也提到，`jittor` 的 `dataset` 中的 `getitem` 方法并不支持返回 `scipy` 中的 `coo_matrix` 对象，我在使用时将 `coo_matrix` 转化成了 `ndarray` 返回，来解决这个问题。虽然没有在 `pytorch` 中返回类似参数的经验，但是 `jittor` 支持的功能还是要少于 `pytorch` 的，受限于开发者群体的数量，这也将是国产框架在一定时期内无法超越 `pytorch` 的原因之一。

## 4.2 pygmtools 的使用感受

在接触本次深度学习大作业之前，我对（基于深度神经网络方法）的图匹配是一无所知的，能够在短短的一学期之内完成这样一个任务，离不开对 `pygmtools` 中众多工具的使用，这也再次说明深度学习中科学使用现有开源工具的重要性。在搭建一个完整的深度学习框架时，不可能处处从头开始造轮子，无论是科研的创新，还是工程的应用，都需要思考在什么地方要把开源代码当做一个黑盒来使用，在什么地方深入研究进行调整来实现自己的想法。`pygmtools` 是一个开发完善、集成度很高的工具包，使用体验也是非常友好的。