实验报告

181840013 陈超

1 基本配置

1. 语言：Pytorch

2. GPU：Kaggle 平台提供的 NVIDIA Tesla P100

3. 数据集：4500 张真实图片 (32×32, 有 label), 100000 张手绘图片 (28×28, 无 label)

(食用本篇实验报告前建议先阅读第五部分)

2 代码框架

代码分为三个部分：数据获取及预处理，网络构建，模型训练。

2.1 数据获取

对应代码 load and preprocess data 注释部分。这一部分代码比较简单，就是调用 transforms 包中的 Compose 函数和 datasets 包中的 ImageFolder 函数，这里我的 batch size 设为 500，因为 500 刚好可以整除训练的图片数量 4500 和测试的图片数量 100000。由于训练图片和测试图片大小不一样，我把两个大小统一到了 32×32。

本来我制作了这些预处理，但是训练效果并不好，问了洪亮同学之后，我又根据他的建议加了一些预处理。首先，考虑到测试图片是黑白的，我将训练图片用 Grayscale 转换为黑白的。再考虑到测试图片是简笔画，轮廓比较明显，我使用了 cv 包中的 Canny 函数进行边缘检测，Canny 边缘检测是目前最好的边缘检测算法之一，步骤包括图像降噪，计算图像梯度，非极大值抑制和双阈值筛选，具体流程可参考数字图像处理 [1]。这里我设置的上下阈值分别为 150 和 200。最后再通过 RandomHorizontalFlip 做一个水

平翻转，就完成了对训练图片的预处理。对测试图片的预处理则只需要转换为灰度后调整大小到 32×32。下面展示一张经过预处理的苹果的训练图像，可以看到，苹果的边缘被清楚地画了出来 (像极了未来我们的头发)：

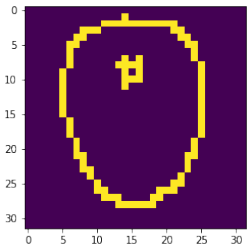

图 1: Apple

2.2 网络构建

这里的 transfer learning 我尝试采用了两种方法，分别是 MMD 和 DANN，主要思想其实差不多，都是让某一层的训练数据和测试数据在网络某一层的 distribution 相似。最后根据实验结果我采用了 DANN 的方法，下面我会先介绍 DANN 方法，之后再补充一下 MMD 方法。

2.2.1 原理

DANN 的全称是 Domain-Adversarial Training of Neural Networks，即域对抗神经网络，由 Yaroslav Ganin 等人在 2014 年提出 [2]。这个模型使用了和 GAN 网络十分类似的方法来进行训练，结构如下：

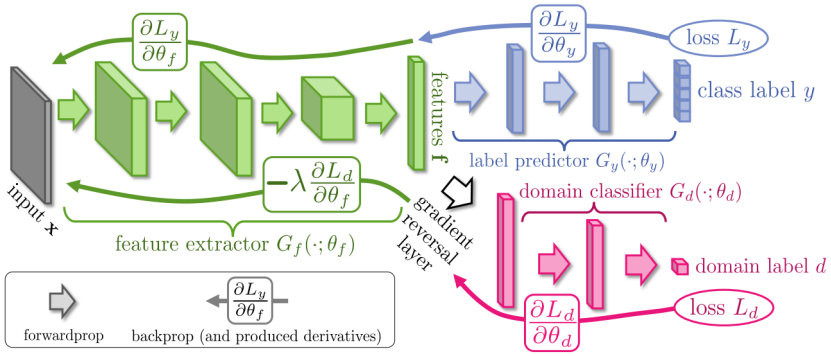

图 2: Structure

根据颜色，整个模型可以分成三个部分。

- 第一部分是绿色的部分，也就是 feature extractor，这个部分的作用是通过卷积处理图区图片的特征，最后通过 flatten 操作对每张图片传出一个一维的特征 feature。
- 第二个部分是蓝色的部分，也就是 label predictor，这部分的作用是根据 feature extractor 传出的 feature 对图片的 label 进行预测，输出即为长度等于类别数的一维 tensor。那么绿色和蓝色部分加起来其实就是一个完整的分类器。
- 第三部分是粉色的部分，也就是 domian classifier，这是迁移学习的关键。这部分的作用是根据 feature extractor 传出的 feature 来判别这是训练图像的 feature 还是测试图像的 feature，这就类似于 GAN 网络中的 Discriminator。

在训练时，每个 epoch 要依次更新 domain classifier，label predictor 和 feature extractor。domain classifier 的 loss 就等于对图片来源分类错误率，即把多少训练图片分为测试图片，把多少测试图片分为训练图片。label predictor 和 feature predictor 的 loss 则等于标签分类错误率减去图片来源分类错误率，同时还可以加上一个系数来平衡这两项。注意这边之所以要减去图片来源分类错误率，是因为 feature extractor 和 domain classifier 是对抗关系，feature extractor 要尽可能增大 domain classifier 对图片来源的分类错误率。

2.2.2 实现

对应代码 construct net 注释部分。有了上面的理论分析，代码的实现其实并不复杂。同上，我首先定义了三个网络，分别是 dis，即前述的 domain classifier；fea，即前述 feature extractor；cla，即前述 label predictor。具体结构如下所示：

- fea：fea 直接沿用了我在 HW1 中使用的网络。fea 由五个 stage 组成，每个 stage 又由三个完全相同的 block 加最后一个步长为 2 的 MaxPool2d 层组成，每个 block 包含一个卷积层，一个 ReLU 激活层和一个 BatchNorm2d 层。每个 stage 除了输入和输出的 channel 数不同，其余完全相同。

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 32, 32]	640
Conv2d-2	[-1, 64, 32, 32]	640
ReLU-3	[-1, 64, 32, 32]	0
ReLU-4	[-1, 64, 32, 32]	0
BatchNorm2d-5	[-1, 64, 32, 32]	128
BatchNorm2d-6	[-1, 64, 32, 32]	128
Conv2d-7	[-1, 64, 32, 32]	36,928
Conv2d-8	[-1, 64, 32, 32]	36,928
ReLU-9	[-1, 64, 32, 32]	0
ReLU-10	[-1, 64, 32, 32]	0
BatchNorm2d-11	[-1, 64, 32, 32]	128
BatchNorm2d-12	[-1, 64, 32, 32]	128
Conv2d-13	[-1, 64, 32, 32]	36,928
Conv2d-14	[-1, 64, 32, 32]	36,928
ReLU-15	[-1, 64, 32, 32]	0
ReLU-16	[-1, 64, 32, 32]	0
BatchNorm2d-17	[-1, 64, 32, 32]	128
BatchNorm2d-18	[-1, 64, 32, 32]	128
MaxPool2d-19	[-1, 64, 16, 16]	0
MaxPool2d-20	[-1, 64, 16, 16]	0
Dropout-21	[-1, 64, 16, 16]	0
Dropout-22	[-1, 64, 16, 16]	0
Conv2d-23	[-1, 128, 16, 16]	73,856
Conv2d-24	[-1, 128, 16, 16]	73,856
ReLU-25	[-1, 128, 16, 16]	0
ReLU-26	[-1, 128, 16, 16]	0
BatchNorm2d-27	[-1, 128, 16, 16]	256
BatchNorm2d-28	[-1, 128, 16, 16]	256
Conv2d-29	[-1, 128, 16, 16]	147,584
Conv2d-30	[-1, 128, 16, 16]	147,584

ReLU−31	[−1, 128, 16, 16]	0
ReLU−32	[−1, 128, 16, 16]	0
BatchNorm2d−33	[−1, 128, 16, 16]	256
BatchNorm2d−34	[−1, 128, 16, 16]	256
Conv2d−35	[−1, 128, 16, 16]	147,584
Conv2d−36	[−1, 128, 16, 16]	147,584
ReLU−37	[−1, 128, 16, 16]	0
ReLU−38	[−1, 128, 16, 16]	0
BatchNorm2d−39	[−1, 128, 16, 16]	256
BatchNorm2d−40	[−1, 128, 16, 16]	256
MaxPool2d−41	[−1, 128, 8, 8]	0
MaxPool2d−42	[−1, 128, 8, 8]	0
Dropout−43	[−1, 128, 8, 8]	0
Dropout−44	[−1, 128, 8, 8]	0
Conv2d−45	[−1, 128, 8, 8]	147,584
Conv2d−46	[−1, 128, 8, 8]	147,584
ReLU−47	[−1, 128, 8, 8]	0
ReLU−48	[−1, 128, 8, 8]	0
BatchNorm2d−49	[−1, 128, 8, 8]	256
BatchNorm2d−50	[−1, 128, 8, 8]	256
Conv2d−51	[−1, 128, 8, 8]	147,584
Conv2d−52	[−1, 128, 8, 8]	147,584
ReLU−53	[−1, 128, 8, 8]	0
ReLU−54	[−1, 128, 8, 8]	0
BatchNorm2d−55	[−1, 128, 8, 8]	256
BatchNorm2d−56	[−1, 128, 8, 8]	256
Conv2d−57	[−1, 128, 8, 8]	147,584
Conv2d−58	[−1, 128, 8, 8]	147,584
ReLU−59	[−1, 128, 8, 8]	0
ReLU−60	[−1, 128, 8, 8]	0
BatchNorm2d−61	[−1, 128, 8, 8]	256
BatchNorm2d−62	[−1, 128, 8, 8]	256
MaxPool2d−63	[−1, 128, 4, 4]	0

```
MaxPool2d-64          [-1, 128, 4, 4]           0
Dropout-65            [-1, 128, 4, 4]           0
Dropout-66            [-1, 128, 4, 4]           0
================================================================
Total params: 1,776,384
Trainable params: 1,776,384
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 15.44
Params size (MB): 6.78
Estimated Total Size (MB): 22.22
----------------------------------------------------------------
```

- cla：cla 也直接沿用了我再 HW1 中使用的网络，由三个全连接层组成，激活函数为 ReLU。

```
----------------------------------------------------------------
Layer (type)               Output Shape              Param #
================================================================
Linear-1                   [-1, 1, 1024]           263,168
ReLU-2                     [-1, 1, 1024]                 0
Linear-3                   [-1, 1, 1024]         1,049,600
ReLU-4                     [-1, 1, 1024]                 0
Linear-5                   [-1, 1, 9]                9,225
================================================================
Total params: 1,321,993
Trainable params: 1,321,993
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.03
Params size (MB): 5.04
Estimated Total Size (MB): 5.08
```

- dis：dis 的结构借鉴了徐满杰同学的方案，由五个全连接层组成加上 BatchNorm1d 和 ReLU 组成。

Layer (type)	Output Shape	Param #
Linear-1	[-1, 512]	131,584
BatchNorm1d-2	[-1, 512]	1,024
ReLU-3	[-1, 512]	0
Linear-4	[-1, 512]	262,656
BatchNorm1d-5	[-1, 512]	1,024
ReLU-6	[-1, 512]	0
Linear-7	[-1, 512]	262,656
BatchNorm1d-8	[-1, 512]	1,024
ReLU-9	[-1, 512]	0
Linear-10	[-1, 512]	262,656
BatchNorm1d-11	[-1, 512]	1,024
ReLU-12	[-1, 512]	0
Linear-13	[-1, 1]	513

Total params: 924,161
Trainable params: 924,161
Non-trainable params: 0

Input size (MB): 0.00
Forward/backward pass size (MB): 0.05
Params size (MB): 3.53
Estimated Total Size (MB): 3.57

2.2.3 MMD 方法

当然，要使得训练图片的 feature 和测试图片的 feature 的 distribution 相似并不只有 DANN 一种方法，MMD 也是一个可以考虑的方法。MMD 即 Maximum mean discrepancy，用于衡量两个随机变量的差异，在迁移学习中可以用于度量两个矩阵的相似程度，具体的计算方式我参考了 github 上的相关代码 [3]。这样我们就只需要一个分类的网络，从中截取一层的输出，计算训练图像和输出图像在该层的输出矩阵的相异程度，记为 mmd_loss，再计算网络本身的分类的损失 label_loss，两个相加即为更新分类网络的 loss。

这种方法实现起来比 DANN 更为简单，因为计算训练图像和输出图像的 feature 的相似程度的函数时确定的。但是实际实现起来却有不少问题，这些我在下面训练结果部分会详细说明。

2.3 模型训练

DANN 方法的训练主要分成两块，domian classidier 的训练和 feature extractor&label predictor 的训练。下面逐个进行分析。

- domian classidier 的训练：如上面原理中所述，domain classifier 的训练根据的 loss 即为其对图片来源的分类错误率，这里我们使用 BCEWithLogitsLoss 作为损失函数。
- feature extractor&label predictor 的训练：如上面原理中所述，feature extractor 和 label predictor 的 loss 由两部分构成，第一部分是图片分类错误率 label_loss，第二部分是 domain classifier 对图片来源的分类错误率 dis_loss,再考虑一个系数 eta,$loss = label\_loss - eta * dis\_loss$

2.4 参数说明

下面详细说明一下模型中用到的一些参数。

- 优化器 optimizer: Adam，lr=0.001
- 训练轮数 epochs: 300
- batch_size: 500
- dis_loss 的系数 eta: 0.1

3 训练结果

根据我的实验结果，DANN 要优于 MMD，所以这边着重介绍 DANN 的训练结果，之后略带介绍 MMD 的训练结果。

3.1 DANN

训练 300 轮后，画出的训练过程图像如下所示：

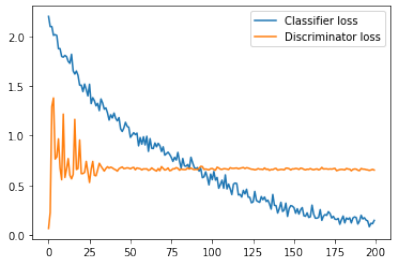

图 3: DANN training result

可以看到，在训练过程中，domian classidier 的 loss 后期基本不变，而 label predictor 的 loss 则一直在下降，整个训练过程相对来说还是比较稳定的。Kaggle 上的正确率，以当前 30% 的测试数据计算，达到了 0.50，也还算差强人意。

3.2 MMD

训练 300 轮后，画出的训练过程图像如下所示：

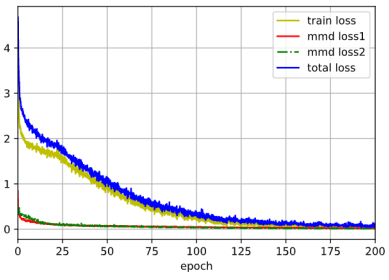

图 4: MMD training result

可以看到以 mmd 为标准计算的训练数据和测试数据的 feature 的差异 mmd_loss，以及网络本身分类的 train_loss 都是稳步下降的。需要注意的是这里之所以有两个 mmd_loss，是因为我选取了两个卷积层的输出分别对其计算 mmd_loss 后一起加到总的 loss 中。但后来我仔细一想，发现这并没有什么必要，因为如果两个卷积层输出的 feature 的 mmd_loss 都很小，那这两个卷积层之间的运算一定程度上就是“无用功”，所以其实只要对一个卷积层的输出计算 mmd_loss 即可，我这里算是多此一举了。而 Kaggle 上的正确率，以当前 30% 的测试数据计算，达到了 0.43。

这里有几点我还想补充说明，我这次之所以最后没有采用 MMD 这个方法，一个是因为正确率没有 DANN 高，另一个原因是 mmd 中的矩阵运算要消耗大量的 CPU 资源及内存，导致我的代码一致运行到一半就爆内存，我也尝试过降低矩阵计算的次数，维度等，但都收效甚微。最后我成功运行的 mmd 版本，是将要计算 mmd_loss 的四维 tensor 展开成二维 tensor 进行计算的，但这显然已经丢失了太多信息了，也有些不河里。尽管如此，最后正确率还是达到了 43%，所以我觉得如果能够有足够的计算资源和内存来完成 mmd 的计算，正确率应该还会再高一些。因此，我把用 MMD 方法训练的代码也放在了上传文件夹中，助教有兴趣的话可以跑一下。

4 feature 可视化

可视化的原理很简单，对 feature 进行降维即可，这里我选用的是 sklearn 包中的 TSNE 算法来降维，这是目前最好的降维算法之一，很适合用于维数较大的数据的降维。接下来调包就完事了，不过需要注意的是，降维的时候需要把训练图像的 feature 和测试图像的 feature 放在一起降维，不能单独降维。这里面的原因细想一下也不难理解，如果单独降维，那么降维时就会分别以两种数据分布作为标准，降维后画出来的图像很有可能是毫无关联并且相互分开的。而两种 feature 放在一起降维，则可以在降维时以两种数据的共同分布为标准，画出来的图像更能表现出两种 feature 内在的差异。

基于此，我们在训练图像和测试图像中各挑选了 2000 张图片，对其 feature 进行降维后画图，结果如下：

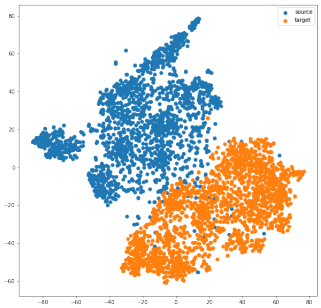

图 5: Without DANN

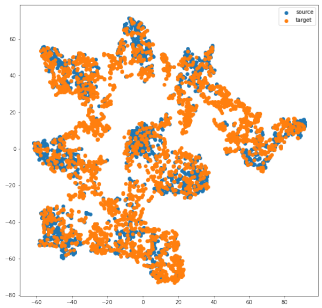

图 6: With DANN

可以看到，没有使用 DANN 之前，训练图像和测试图像的分布是完全分开的 (我也很好奇为什么会分得这么开)。加上了 DANN 后，很明显可以看到训练图像和测试图像的 feature 重合得很好，9 个类别也分得很清楚，效果可以说是不错的。但在我看来，这样的可视化其实意义并不大，因为 DANN 在训练过程中就时一直在降低训练图片和测试图片的分布的差异的，可视化后两者重合度高也是情理之中的，真正检验 DANN 这个算法是否有效还得看测试集上的准确率。

5 舍不得删掉的废话

1. 提交的 project 中,Figures 文件夹包含了所有原始图像,Answer(dann).csv 为使用 DANN 的预测结果，Answer(mmd).csv 为使用 MMD 的预测结果，train(dann).py 为使用 DANN 训练的代码，train(mmd).py 为使用 MMD 训练的代码，train(dann).sh 为使用 DANN 的训练脚本，train(mmd).sh 为使用 mmd 的训练脚本，test.py 为测试代码，test.sh 为测试脚本。Feature extractor.param，Label predictor.param，Domain classifier.param 为三个网络参数。

2. 如下运行用 DANN 训练的脚本：bash train(dann).sh train_path test_path csv_path,其中 train_path 为训练图片文件夹 (比如../input/tranfer-data/train_data),test_path 为测试图片文件夹 (比如../input/tranfer-data/testdata_raw)，csv_path 为 csv 文件 (比如../input/tranfer-data/testdata_raw/test.csv)。如下运行用 MMD 训练的脚本：bash train(mmd).sh train_path test_path csv_path。两个运行脚本执行后会自动训练并保存训练结果，之后对 feature 进行可视化并画图，最后对测试图片进行预测并生成 csv 文件。如下运行测

试代码：bash test.py csv_path test_path param1 param2 param3，其中 param1，param2，param3 分别是 feature extractor，label predictor，domain classifier 三个网络的参数。

3. 感谢：感谢洪亮同学在图像预处理方面给我的启发。感谢徐满杰同学在网络架构方面给我的参考。

4. 如果有任何问题，请邮箱联系我 181840013@smail.nju.edu.cn

6 参考资料

[1] Canny, J. F. (1986). A theory of edge detection. IEEE Trans Pattern Anal Mach Intell, 8, 147-163.

[2] Ganin, Y., & Lempitsky, V. (2015, June). Unsupervised domain adaptation by backpropagation. In International conference on machine learning (pp. 1180-1189). PMLR.

[3] https://github.com/jindongwang/transferlearning/tree/master/code/deep/DaNN