

媒体与认知

图像实验结题报告

刘圣禹 马啸阳 袁健皓

2020 年 5 月 28 日

1 实验任务

本实验主要目标为对交通标志进行分类与检测，分为如下四个任务。

1. 基于传统机器学习方法进行交通标志分类；
2. 基于深度学习方法，实现并训练一个卷积神经网络；
3. 小样本分类；
4. 在给定的交通标志检测数据集上，实现一个检测模型。

2 文献调研

本节主要在课程中已习得的深度学习基础上，调研了若干现有表现较好的分类及检测方法，归纳总结各方法的思想及结构。

2.1 分类

2.1.1 HOG[1]

HOG (Histogram of Oriented Gradient，方向梯度直方图) 是一种经典的图像特征提取方法，最初被用于行人识别领域。这种方法有其自身的优势：能够获取原始图片上很明显的边缘和梯度结构，且它在局部的特征表示具有几何不变性和光学形变的不变性。

HOG 的基本思想在于图像的局部特征可由梯度及边缘方向的分布刻画。利用梯度描绘图像边缘信息是图像处理中的经典方法，使用 Sobel 算子或其它不同算子可以提取不同的边缘特

征，而 HOG 则统计了梯度方向的分布，更好刻画了图像的局部特征。算法流程如图1所示，先对输入图片进行归一化，再计算图像的梯度大小与方向，然后将图片分为小的区域。对于每个区域中的像素，求出梯度方向的 1 维直方图，再对其进行权重投影。对重叠块中的小区域进行对比度归一化，最后在将所有块中的直方图向量组合成一个 HOG 特征向量，得到的特征向量便可以通过线性 SVM 实现分类等任务。



图 1: HOG 算法流程

总而言之，HOG 统计分割区域上的梯度方向直方图，用以刻画图像的局部特征，取得一维特征后，再使用 SVM 等经典分类器实现分类任务，这一经典算法也能在分类任务中获得较高的精度。

2.1.2 MCDNN[2]

MCDNN (Multi-Column Deep Neural Network) 在德国交通标志检测识别数据集 (German traffic sign recognition benchmark) 上获得了 99.46% 的准确率，达到了人类识别的水准，较为符合本实验的目标。

MCDNN 中，基础网络结构如图2(a) 所示，含两轮交替的卷积与最大池化层，以及两个全连接层，是典型的 DNN 网络结构。而 MCDNN 的主要贡献在于引入了多道 DNN 机制 (如图2(b))。

多道 DNN 的思想在于，将原始图片经过不同的输入增强与预处理，作为多个 DNN 通道的输入，每个 DNN 通道被随机初始化 (服从不同正态分布) 后进行训练，训练过程中也不断对输入图片进行随机的扭曲。最后对所有 DNN 通道的输出结果做平均，得到输出。并且实验表明，进行算术平均比不同通道的加强平均具有更好的泛化能力。

MCDNN 的数据增强部分是对整个数据集的操作，非随机，包括 Image Adjustment (图像调整)、Histogram Equalization (直方图均衡)、Adaptive Histogram Equalization (自适应直方图均衡)、Contrast Normalization (对比度归一化) 等基本的图像预处理方式。经过不同的数据增强会作为不同的通道输入。图像扭曲 (在训练过程中也每次迭代执行) 包括小幅度平移、缩放、旋转。不同的图片经过切割、数据增强与扭曲，最终通过双线性插值得到最终固定大小的图片数据。

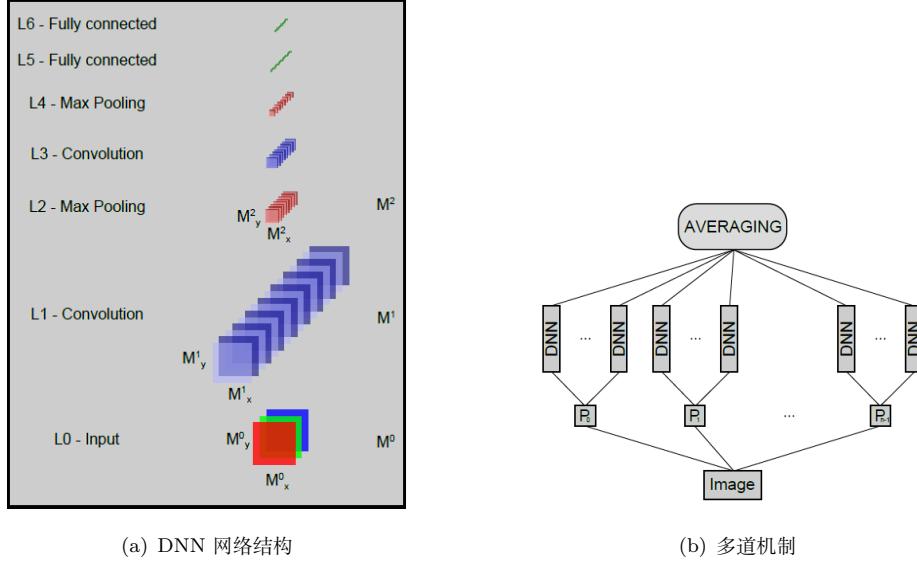


图 2: MCDNN 结构

文中训练交通标志所使用的通道结构是训练了 25 个 DNN，每 5 个 DNN 对应一种图像预处理（包括原始图像和 4 种归一化方法），即每种预处理连接 5 个通道，最终执行算术平均，最终得到了 99.46% 的准确率，同时具有较好的泛化能力，是非常优秀的深度学习方法。

2.1.3 VGG[3]

VGG 表明增大网络深度能有效提升网络的性能。其主要思想为在 AlexNet 中用较小卷积核的叠加来取代较大的卷积核，增加网络深度。VGG 基础结构为卷积与最大池化的交替，如图3所示。图像预处理中将每个像素减去训练集上的 RGB 均值。

实验表明，AlexNet 中采用的 LRN 层 (local response normalization，局部响应归一化) 没有带来性能的提高，因此 VGG 中去除 LRN 层，采用更简单的网络。同时，多个小卷积核叠加比单个大卷积核性能好，并且深度更深的网络分类性能更好。因此，VGG 的优点在于，它以简单、便于设置的网络结构达成了较好的性能。同时，它的缺点在于它的全连接层过多导致参数过多难以训练。

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224×224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

图 3: VGG 网络结构

2.1.4 ResNet[4]

ResNet (Residual Network, 深度残差网络) 提出的主要动机是为了解决深度网络的退化问题。实验中，当网络的层数增加到一定程度时，继续增加反而会导致准确率的降低（出于梯度消失或其它原因）。但是理论上更深的网络不会比更浅的网络准确率低，因为可以将多余的层学习为恒等映射，出现这种问题说明了用非线性层表示恒等映射是很困难的。基于这种想法，ResNet 中令 $\mathcal{H}(x)$ 为需要学习的目标函数，该网络不学习 $\mathcal{H}(x)$ ，改而学习 $\mathcal{H}(x) - x$ ，令 $\mathcal{F}(x) = \mathcal{H}(x) - x$ ，构造如下结构：

$$y = \mathcal{F}(x, W_i) + x$$

其中 x, y 分别表示输入、输出， $\mathcal{F}(x, W_i)$ 表示需要被学习的映射。这样在遇到先前所述难以学习恒等映射的问题时，该网络只需将 $\mathcal{F}(x)$ 学习为 0，降低了学习的难度和资源消耗。

这里 x 是短路直连到输出的，其中假设 x 与维数相同，如果维数不同就对 x 作投影变换。 $\mathcal{F}(x)$ 在此网络中为两层或三层网络需要拟合的函数。这样的基础结构即残差块，如图4所示，

以这样的结构构成了 ResNet，如图7所示（左图为 VGG）。

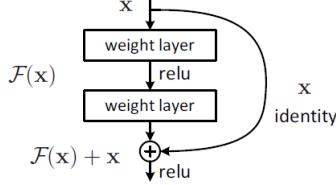


图 4: 残差块

ResNet 在 ImageNet 上的实验结果如图5所示。对照网络在网络层数为 34 层时的错误率要高于 18 层的网络，说明出现了退化；而 ResNet 在 34 层时的错误率要低于 18 层，说明退化现象被较好地解决了，即可以通过增加网络深度来提高准确率了。同时发现 ResNet 的收敛速度也要更快。

然而使用 ResNet 也并不能无限制地增加深度，如图6所示，发现在网络过深（1000 余层）时，仍然会因为过拟合而出现准确率降低的情况。

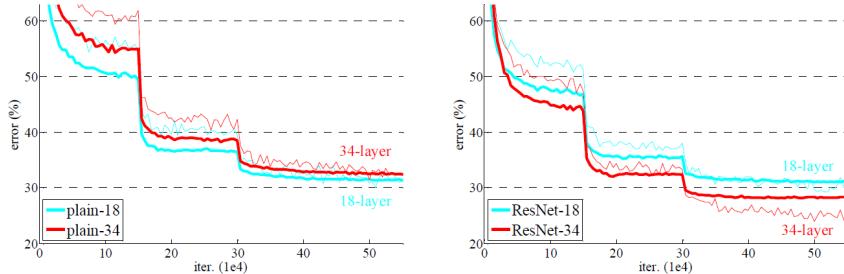


图 5: ResNet 实验结果

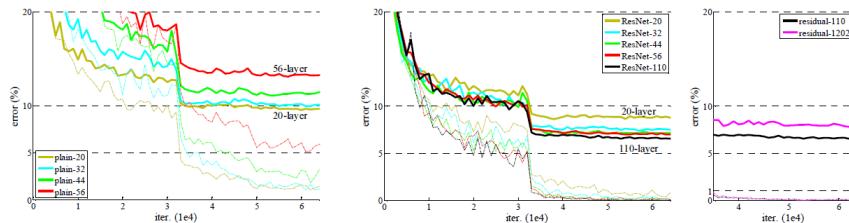


图 6: 深层 ResNet 实验结果

总体而言，ResNet 通过引入残差块提高网络深度，获得更强的学习能力，近来有诸多网络变体，是实现分类网络的较好选择。

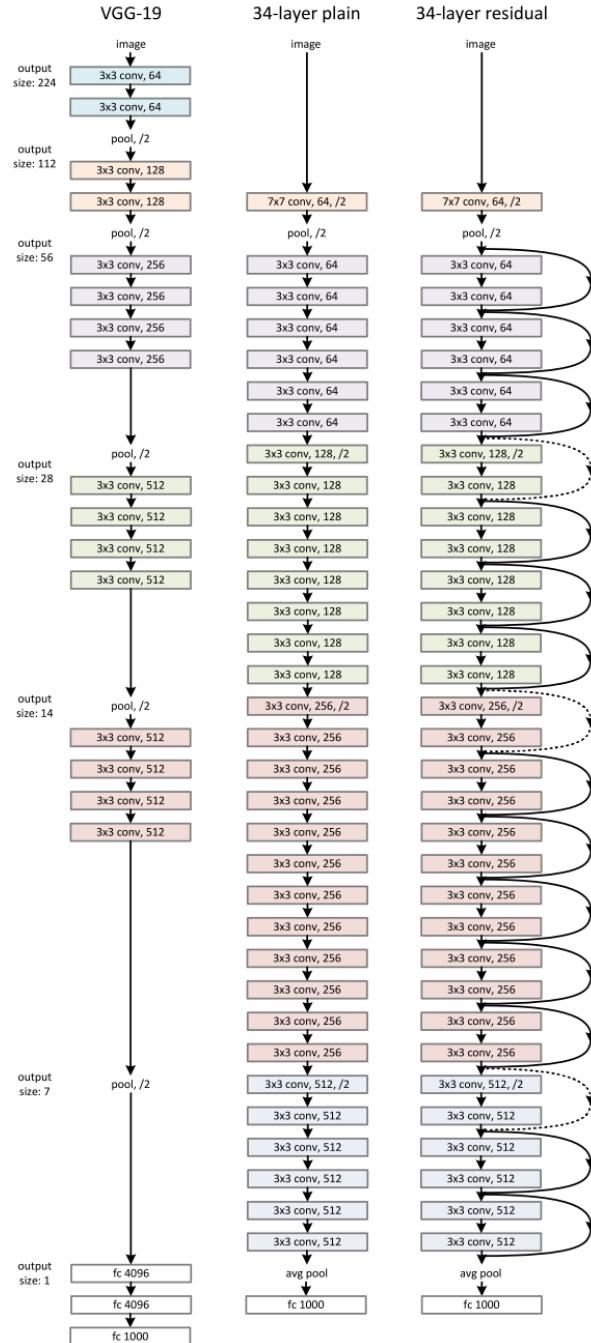


图 7: ResNet 网络结构

2.1.5 DenseNet[5]

回顾 ResNet 结构，对于每一层，ResNet 会与前面某层建立 shortcut 连接。DenseNet 可以说是 ResNet 的改进版，主要思想是每一层都与前面的所有层连接在一起，作为下一层的输入。对于 n 层网络，存在着 $O(n^2)$ 数量级的连接（这会导致内存消耗过大，之后实验中会讨论），因此这是一种密集连接。运用这种连接方式可以实现特征重用，提升效率。典型的 DenseNet 如图8所示。在 ImageNet 数据集上采用的 DenseNet 网络结构如图9所示。

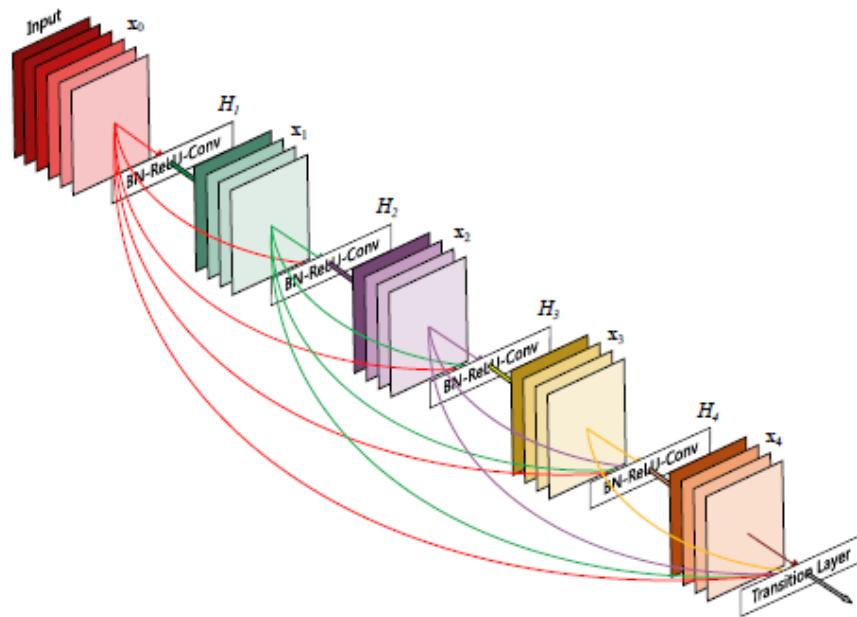


图 8: DenseNet

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112		7 × 7 conv, stride 2		
Pooling	56 × 56		3 × 3 max pool, stride 2		
Dense Block (1)	56 × 56	$\left[\begin{array}{l} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 6$	$\left[\begin{array}{l} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 6$	$\left[\begin{array}{l} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 6$	$\left[\begin{array}{l} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 6$
Transition Layer (1)	56 × 56		1 × 1 conv		
Dense Block (2)	28 × 28		2 × 2 average pool, stride 2		
Transition Layer (2)	28 × 28		1 × 1 conv		
Dense Block (3)	14 × 14	$\left[\begin{array}{l} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 12$	$\left[\begin{array}{l} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 12$	$\left[\begin{array}{l} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 12$	$\left[\begin{array}{l} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 12$
Transition Layer (3)	14 × 14		2 × 2 average pool, stride 2		
Dense Block (4)	7 × 7	$\left[\begin{array}{l} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 24$	$\left[\begin{array}{l} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 32$	$\left[\begin{array}{l} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 48$	$\left[\begin{array}{l} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 64$
Classification Layer	1 × 1		7 × 7 global average pool		1000D fully-connected, softmax

图 9: ImageNet 数据集上采用的 DenseNet 网络结构

下面用公式来比较传统的网络结构与两种新型的网络结构。

传统的网络在第 n 层的输出为:

$$x_l = H_l(x_{l-1})$$

ResNet 在第 n 层的输出为:

$$x_l = H_l(x_{l-1}) + x_{l-1}$$

DenseNet 在第 n 层的输出为:

$$x_l = H_l([x_0, x_1, \dots, x_{l-1}])$$

其中 H_l 表示非线性函数, 可能包括批量归一化、卷积、池化等操作。

DenseNet 实验结果如图10所示, 发现相比于 ResNet 网络, DenseNet 利用更少的参数得到了更低的错误率。

Method	Depth	Params	C10	C10+	C100	C100+	SVHN
Network in Network [22]	-	-	10.41	8.81	35.68	-	2.35
All-CNN [32]	-	-	9.08	7.25	-	33.71	-
Deeply Supervised Net [20]	-	-	9.69	7.97	-	34.57	1.92
Highway Network [34]	-	-	-	7.72	-	32.39	-
FractalNet [17] with Dropout/Drop-path	21 21	38.6M 38.6M	10.18 7.33	5.22 4.60	35.34 28.20	23.30 23.73	2.01 1.87
ResNet [11]	110	1.7M	-	6.61	-	-	-
ResNet (reported by [13])	110	1.7M	13.63	6.41	44.74	27.22	2.01
ResNet with Stochastic Depth [13]	110 1202	1.7M 10.2M	11.66 -	5.23 4.91	37.80 -	24.58 -	1.75 -
Wide ResNet [42] with Dropout	16 28 16	11.0M 36.5M 2.7M	- - -	4.81 4.17 -	- - -	22.07 20.50 -	- - 1.64
ResNet (pre-activation) [12]	164 1001	1.7M 10.2M	11.26* 10.56*	5.46 4.62	35.58* 33.47*	24.33 22.71	- -
DenseNet ($k = 12$) DenseNet ($k = 12$) DenseNet ($k = 24$)	40 100 100	1.0M 7.0M 27.2M	7.00 5.77 5.83	5.24 4.10 3.74	27.55 23.79 23.42	24.42 20.20 19.25	1.79 1.67 1.59
DenseNet-BC ($k = 12$) DenseNet-BC ($k = 24$) DenseNet-BC ($k = 40$)	100 250 190	0.8M 15.3M 25.6M	5.92 5.19 -	4.51 3.62 3.46	24.15 19.64 -	22.27 17.60 17.18	1.76 1.74 -

图 10: DenseNet 实验结果

DenseNet 当前存在的问题在于，当前的深度学习框架（如 pytorch）对 DenseNet 的密集连接没有很好的支持，只能通过拼接操作将当前层的输出与之前层的输出拼接在一起传递给下一层。这导致一个 n 层的网络需要消耗 $O(n^2)$ 量级的内存。这一点我在网络训练中有所体会，如果 Batchsize 设得过大，便会超出内存最大容量。因此我在训练中将 Batchsize 设为 8，可以正常训练。

2.1.6 Inception[6][7]

Inception 网络结构改进了多个版本，相比于 AlexNet、VGG16 等增加网络宽度的想法，其主要思想是通过横向增加网络的广度来增加模型的特征提取能力。下面介绍 Inception 的几种不同版本。

2.1.6.1 Inception V1

Inception V1 结构如图11所示。可见该结构包含 1×1 卷积， 3×3 卷积， 5×5 卷积， 3×3 池化层。通过 3×3 和 5×5 卷积层学习到数据中“稀疏”的特征，通过 1×1 卷积层学习到数据中“不稀疏”的结构，从而增强了模型的泛化能力。像这样增加网络广度所带来的一个主要问题

就是参数的数量会变得很大。为解决这个问题，该网络引入了 1×1 卷积进行降维来减小网络的参数量。

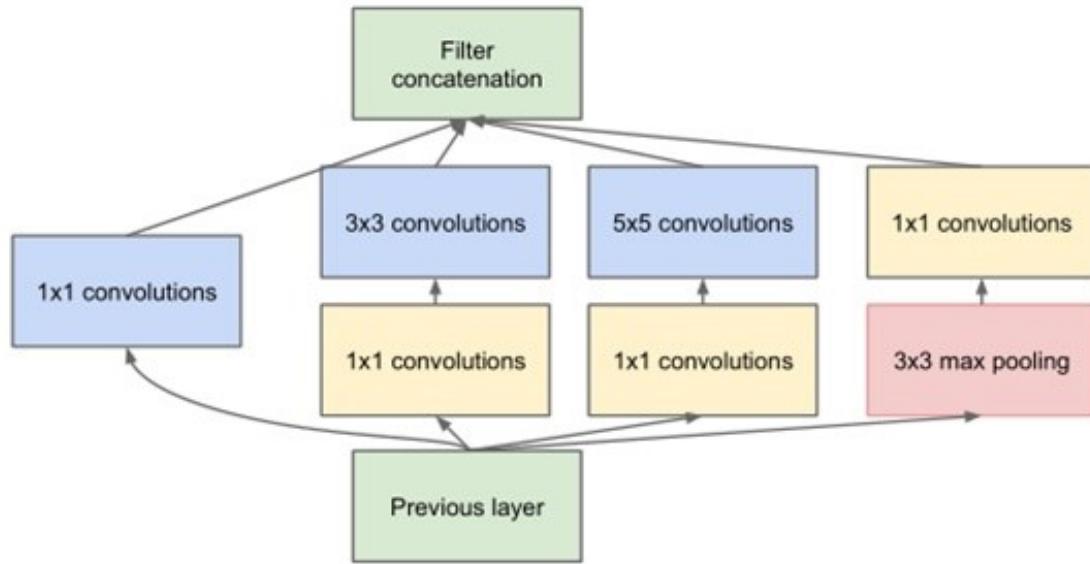


图 11: Inception V1 基础结构

2.1.6.2 Inception V2

Inception V2 在 Inception V1 的基础上做了许多改进：

1. 增加了批量归一化层，将每一层的输出都归一化到标准正态分布，这有助于训练。
2. 使用两个 3×3 的卷积层代替 1 个 5×5 的卷积层，这样既可以得到相同的特征图，又能减少参数的数量，还可以增加网络深度。替换后的网络结构如图12所示。

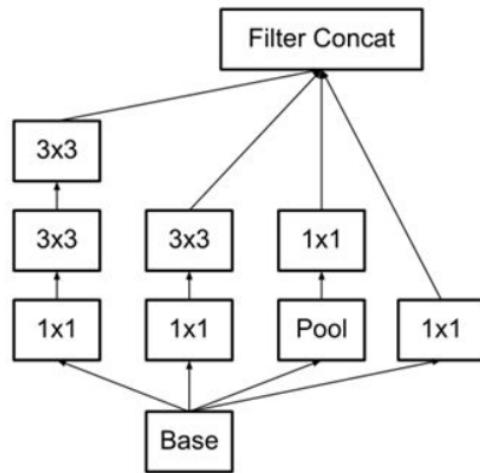


图 12: 将 5*5 卷积替换后的网络结构

3. 采用 $1 \times n$ 和 $n \times 1$ 的非对称卷积来代替 $n \times n$ 的对称卷积。这样增加了网络的深度，也降低了参数的数量，网络结构如图13所示。

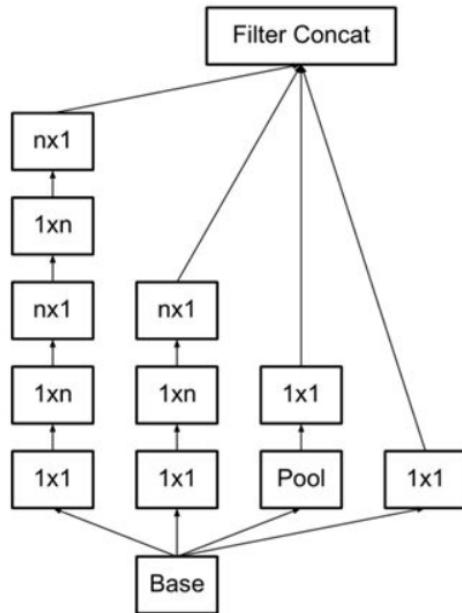


图 13: 将对称卷积替换为非对称卷积后的网络结构

4. 在梯度为 8×8 的情况下增加了滤波器输出的模块，从而产生高维特征，如图14所示。

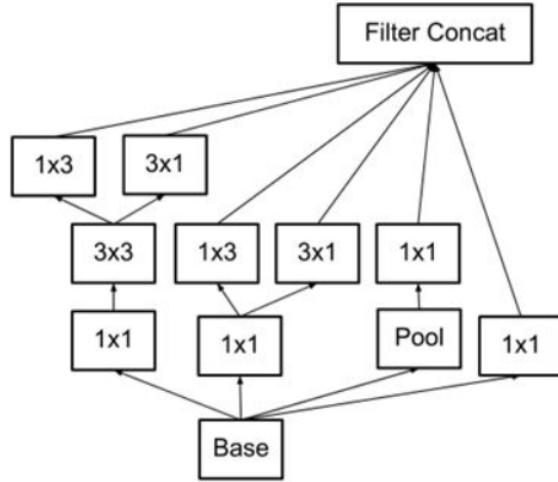


图 14: 加入滤波器输出后的网络结构

经过改进之后的网络与其他网络的误差率对比如图15，发现 Inception V2 明显优于其他网络。

Network	Top-1 Error	Top-5 Error	Cost Bn Ops
GoogLeNet [20]	29%	9.2%	1.5
BN-GoogLeNet	26.8%	-	1.5
BN-Inception [7]	25.2%	7.8	2.0
Inception-v2	23.4%	-	3.8
Inception-v2 RMSProp	23.1%	6.3	3.8
Inception-v2 Label Smoothing	22.8%	6.1	3.8
Inception-v2 Factorized 7×7	21.6%	5.8	4.8
Inception-v2 BN-auxiliary	21.2%	5.6%	4.8

图 15: Inception V2 与其他网络的误差率对比 (ImageNet 数据集)

2.1.6.3 Inception V3

Inception V2 存在着表达瓶颈的问题，即特征图大小在池化时出现了骤降，这将会丢失大量的信息，对模型训练造成困难。因此 Inception V3 设计了一种并行的降维结构，如图16所示。

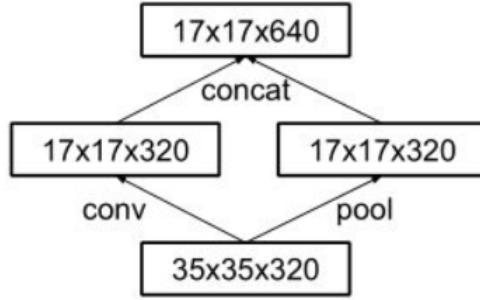


图 16: Inception V3 的并行结构

Inception V3 网络与其他网络的识别误差率对比如图17所示，相比于其他网络，Inception V3 的准确率进一步提升。

Network	Crops Evaluated	Top-5 Error	Top-1 Error
GoogLeNet [20]	10	-	9.15%
GoogLeNet [20]	144	-	7.89%
VGG [18]	-	24.4%	6.8%
BN-Inception [7]	144	22%	5.82%
PReLU [6]	10	24.27%	7.38%
PReLU [6]	-	21.59%	5.71%
Inception-v3	12	19.47%	4.48%
Inception-v3	144	18.77%	4.2%

图 17: Inception V3 网络与其他网络的识别误差率对比

2.1.7 匹配网络 [8] 与原型网络 [9]

这两种方法都是用以进行小样本 (few shot) 分类的。用于小样本分类任务的方法主要有基于 Fine-tune、基于度量与基于图神经网络等多类思路。本小组主要调研了基于度量的小样本分类方法，其中主要有匹配网络 (matching network) 与原型网络 (prototypical network)

两种方法。

匹配网络使用注意力机制，在少样本预测（或本实验中单样本预测）中，以简单的加权平均预测未见过的样本 $y = \sum_{i=1}^k a(\hat{x}, x_i) y_i$ ，其中 a 为一个注意力机制， $(x_i, y_i)_{i=1}^k$ 为 k 对样本-标签对组成的训练集（支持集）， \hat{x} 是待预测的样本。

模型的能力很大取决于注意力机制的选取。匹配网络中采用如下的 softmax 模型计算注意力：

$$a(\hat{x}, x_i) = e^{c(f(\hat{x}, g(x_i)))} / \sum_{j=1}^k e^{c(f(\hat{x}, g(x_j)))}$$

其中 c 是余弦距离，而 f 和 g 分别为查询集和测试集所嵌入的网络，这两个网络可以相同。图像分类任务中，这两个网络通常为 CNN。原论文中介绍了一些 f 和 g 网络的设计优化方法，但不影响匹配网络的核心思想。算法的流程如图18所示。

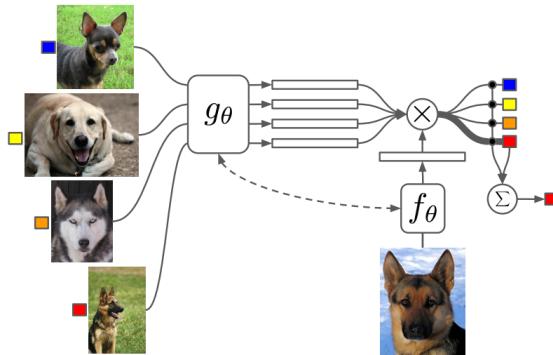


图 18：匹配网络算法流程

使用传统分类方法进行小样本分类时容易产生过拟合的问题。KNN 直接提取图片的特征向量并进行距离的比对，方法简捷直接，基本避免了过拟合的问题。但是在实际操作的过程中，提取图片特征向量的方法十分有限，比如直接以图片的像素值（或将其展平成一维向量）作为特征向量，或者利用 hog 方法进行提取。这些方法并没有利用到这一类图片的“特征”，因此正确率较低。另外，KNN 没有预训练的过程。而原型网络（Prototypical Networks）较好地解决了这一问题，示意图如图19所示。原型网络对某一类型问题提取特征向量（原型）的方法进行预训练；训练完成后使用训练好的矩阵进行特征向量的提取，正确率获得了较大的提升。

需要指出的是，原型网络下的小样本学习并不是指预训练时的“小样本”，而是指预训练之后遇到新的分类问题时给出的“小样本”，并且这些预训练和测试的样本对应的分类问题是同一类型的，比如交通标志的识别、手写体的识别等等。下面给出原型网络的具体原理。

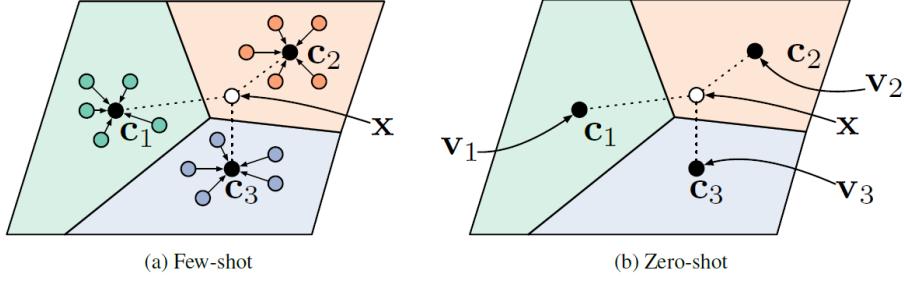


图 19: 原型网络示意

首先进行预训练，具体流程如图??所示。提取特征向量（原型）的方法为

$$c_k = \frac{1}{|S_k|} \sum_{x_i, y_i \in S_k} f_\phi(x_i)$$

其中 S_k 指的是支持集的数据，数目与 n-shot 对应。上式本质上是对这一类中所有支持集中的样本原型求平均的过程，从而获得这一类的原型。整个预训练过程就是训练 ϕ 。

假设预训练的数据集为 $D = (x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)$ 。数据集样本的总个数为 N ，样本标签空间为 $y_i \in 1, 2, \dots, K$ ，共有 K 类样本。其中 D_k 代表数据集标签为 k 的全部样本。训练过程分为若干个 epoch，每个 epoch 分为若干个 episode。训练的 episode 为随机从训练集中选择的一个类子集，一个类子集包含 N_C 个类：

$$V = \text{RandomSample}(1, 2, \dots, K, N_C)$$

对于每一个类，选择一些样例作为支持集 S_k ，个数为 N_S ，在其余的样本中选一些作为查询集 Q_k ，个数为 N_Q 。之后对支持集进行原型的提取，定义损失函数，并对类子集进行损失的更新

$$J = J + \frac{1}{N_C N_Q} [d(f_\phi(x), c_k) + \ln \sum_{k'} \exp(-d(f_\phi(x), c_{k'}))]$$

对损失函数进行随机梯度下降法，使同一个类的样本距离较近。

测试过程中，已知一个属于一个或很少几个新的类 k 的样本，计算类 k 的原型

$$c_k = \frac{1}{|S_k|} \sum_{x_i, y_i \in S_k} f_\phi(x_i)$$

给出测试样本后，定义如下 softmax 目标函数

$$p_\phi(y = k | \mathbf{x}) = \frac{\exp(-d(f_\phi(\mathbf{x}), \mathbf{c}_k))}{\sum_{k'} \exp(-d(f_\phi(\mathbf{x}), \mathbf{c}_{k'}))}$$

上式取最大值时，说明样本 x 属于类 k 的概率最大，因此可判定样本 x 属于类 k 。

论文中也考察了不同距离函数的选取（认为使用 Bregman 散度比余弦距离更为有效）以及训练时的样本数的影响。图20与图21分别为原型网络在 Omniglot 数据集与 miniImageNet 数据集上的结果。从结果可以看出，原型网络相比之前的匹配网络有提升，而且证实了欧几里得距离要优于余弦距离。总体而言，原型网络是一个简单有效，准确率较高的少样本分类算法。

Table 1: Few-shot classification accuracies on Omniglot.

Model	Dist.	Fine Tune	5-way Acc.		20-way Acc.	
			1-shot	5-shot	1-shot	5-shot
MATCHING NETWORKS [29]	Cosine	N	98.1%	98.9%	93.8%	98.5%
MATCHING NETWORKS [29]	Cosine	Y	97.9%	98.7%	93.5%	98.7%
NEURAL STATISTICIAN [6]	-	N	98.1%	99.5%	93.2%	98.1%
PROTOTYPICAL NETWORKS (OURS)	Euclid.	N	98.8%	99.7%	96.0%	98.9%

图 20: 原型网络在 Omniglot 数据集中的结果

Model	Dist.	Fine Tune	5-way Acc.	
			1-shot	5-shot
BASELINE NEAREST NEIGHBORS*	Cosine	N	$28.86 \pm 0.54\%$	$49.79 \pm 0.79\%$
MATCHING NETWORKS [29]*	Cosine	N	$43.40 \pm 0.78\%$	$51.09 \pm 0.71\%$
MATCHING NETWORKS FCE [29]*	Cosine	N	$43.56 \pm 0.84\%$	$55.31 \pm 0.73\%$
META-LEARNER LSTM [22]*	-	N	$43.44 \pm 0.77\%$	$60.60 \pm 0.71\%$
PROTOTYPICAL NETWORKS (OURS)	Euclid.	N	$49.42 \pm 0.78\%$	$68.20 \pm 0.66\%$

图 21: 原型网络在 miniImageNet 数据集中的结果

2.2 检测

2.2.1 R-CNN 与 Fast R-CNN[10]

R-CNN (region with CNN features) 在权威数据集 PASCAL 达到了 53.3% 的水平，将 mAP 在 VOC2012 的结果上提高了 30% 以上。

如图22所示，R-CNN 的物体检测系统由三部分构成。首先是进行区域推荐（region proposal），论文使用了产生类别无关的选择性搜索（selective search）；其次进行特征提取（feature extraction），具体是在图像转换的基础上，使用一个大型卷积神经网络，通过五个卷积层和两个全连接层进行前向传播，最终实现对每个推荐区域抽取 4096 维度的特征向量；第三个是一个指定类别的线性 SVM。

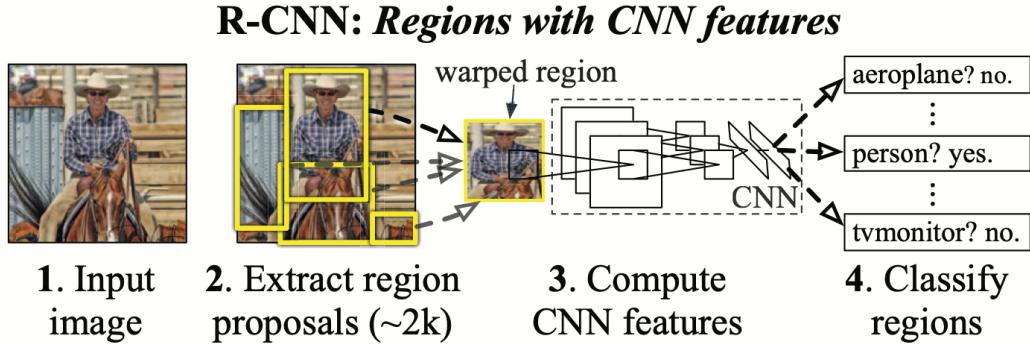


图 22: R-CNN 算法流程

作者认为，本方法取得较高性能的原因主要有两个方面。首先是应用自底向上与区域推荐结合的高容量卷积神经网络进行物体定位与分割，另外一个是标签匮乏情况下训练神经网络的一个方法。在有监督的情况下进行网络的预训练（supervised pre-training）是很有效的，之后采用稀少数据在特定领域进行定位任务的调优（domain-specific fine-tuning），从而获得了很高的检测精度，如图23所示。

VOC 2011 test	bg	aero	bike	bird	boat	bottle	bus	car	cat	chair	cow	table	dog	horse	mbike	person	plant	sheep	sofa	train	tv	mean
R&P [2]	83.4	46.8	18.9	36.6	31.2	42.7	57.3	47.4	44.1	8.1	39.4	36.1	36.3	49.5	48.3	50.7	26.3	47.2	22.1	42.0	43.2	40.8
O ₂ P [4]	85.4	69.7	22.3	45.2	44.4	46.9	66.7	57.8	56.2	13.5	46.1	32.3	41.2	59.1	55.3	51.0	36.2	50.4	27.8	46.9	44.6	47.6
ours (full+fg R-CNN fc ₆)	84.2	66.9	23.7	58.3	37.4	55.4	73.3	58.7	56.5	9.7	45.5	29.5	49.3	40.1	57.8	53.9	33.8	60.7	22.7	47.1	41.3	47.9

图 23: R-CNN 实验结果

然而，R-CNN 有着显著的缺陷。R-CNN 的训练过程是一个多级流水线，训练的时间较长，需要大量的磁盘空间缓存，目标检测速度很慢。而 Fast-RCNN 修正了 R-CNN 的不足，并提高速度与准确性。

Fast R-CNN 的结构如图24所示，输入图像与多个感兴趣区域（RoI）被输入到几个卷积层网络中，之后将每个 RoI 池化到特征图中，在此基础上抽取了固定长度的特征向量，送入一系列全连接层（fc）中。输出两个向量：softmax 概率和回归偏移量。

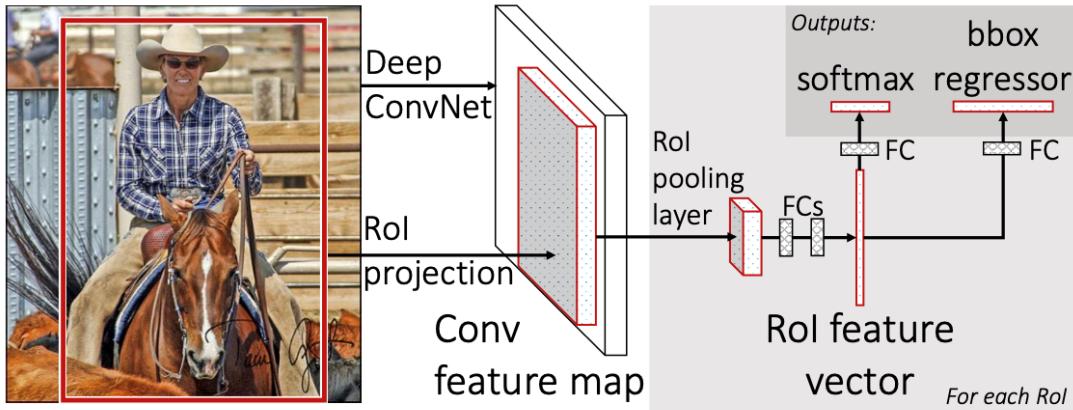


图 24: Fast R-CNN 算法流程

在训练的过程中使用反向传播算法将网络的所有权重进行微调是 Fast R-CNN 的重要能力，具体的过程包括多任务损失、小批量采样、RoI 池化层的反向传播和 SGD 超参数。在微调的同时，Fast R-CNN 还能联合优化 softmax 分类器和检测框回归。网络训练完毕，使用前向传播进行检测，由于计算全连接层花费时间较多，可以使用截断的 SVD 压缩来进行检测加速。

Fast R-CNN 在 VOC12 获得了最高精度；同时，相对于 R-CNN，Fast R-CNN 能进行更快速的训练和检测，实验结果如图25所示。

	Fast R-CNN			R-CNN			SPPnet	
	S	M	L	S	M	L	$\dagger L$	
train time (h)	1.2	2.0	9.5	22	28	84	25	
train speedup	18.3 ×	14.0×	8.8×	1×	1×	1×	3.4×	
test rate (s/im)	0.10	0.15	0.32	9.8	12.1	47.0	2.3	
▷ with SVD	0.06	0.08	0.22	-	-	-	-	
test speedup	98×	80×	146×	1×	1×	1×	20×	
▷ with SVD	169×	150×	213 ×	-	-	-	-	
VOC07 mAP	57.1	59.2	66.9	58.5	60.2	66.0	63.1	
▷ with SVD	56.5	58.7	66.6	-	-	-	-	

(a) 速度对比

method	train set	aero	bike	bird	boat	bottle	bus	car	cat	chair	cow	table	dog	horse	mbike	persn	plant	sheep	sofa	train	tv	mAP
BabyLearning	Prop.	78.0	74.2	61.3	45.7	42.7	68.2	66.8	80.2	40.6	70.0	49.8	79.0	74.5	77.9	64.0	35.3	67.9	55.7	68.7	62.6	63.2
NUS-NIN.c2000	Unk.	80.2	73.8	61.9	43.7	43.0	70.3	67.6	80.7	41.9	69.7	51.7	78.2	75.2	76.9	65.1	38.6	68.3	58.0	68.7	63.3	63.8
R-CNN BB [10]	12	79.6	72.7	61.9	41.2	41.9	65.9	66.4	84.6	38.5	67.2	46.7	82.0	74.8	76.0	65.2	35.6	65.4	54.2	67.4	60.3	62.4
FRCN [ours]	12	80.3	74.7	66.9	46.9	37.7	73.9	68.6	87.7	41.7	71.1	51.1	86.0	77.8	79.8	69.8	32.1	65.5	63.8	76.4	61.7	65.7
FRCN [ours]	07++12	82.3	78.4	70.8	52.3	38.7	77.8	71.6	89.3	44.2	73.0	55.0	87.5	80.5	80.8	72.0	35.1	68.3	65.7	80.4	64.2	68.4

(b) 精度对比

图 25: Fast R-CNN 实验结果

2.2.2 YOLO[11]

YOLO (You Only Look Once)，顾名思义“只看一次”，使用单个神经网络直接预测物品边界和类别概率。与 R-CNN 方法采用推荐区域生成 bounding box 进而用分类器进行识别相比，YOLO 使用单个神经网络直接提取多个 bounding boxes 和类别概率（如图26所示），流程简单，速度更快，适合用于实时检测。与分类器不同，YOLO 对直接影响检测性能的损失函数进行训练，并且对整个模型进行整体训练。

算法流程上，YOLO 首先对图像缩放到固定大小，然后输入卷积神经网络（24 层卷积层与 2 层全连接层），得到 bounding box 与类概率，最后通过非极大值压缩筛选 bbox。对于缩放后的图片，YOLO 会划分网格，每个网格负责检测中心落在其中的物体。网格所检测生成的 confidence score 定义为 $Pr(\text{Object}) \cdot IOU_{pred}^{truth}$ ，其中 $Pr(\text{Object})$ 表示 bounding box 中含有物体的概率，之后以生成的类条件概率 $Pr(\text{Class}_i | \text{Object})$ 进行分类。 $IOU = (A \cap B) / (A \cup B)$ 可以描述矩形框重叠的面积比例，这与 RCNN 使用的评价标准一致，但 YOLOv3 中进一步使用了更高级的 GIoU[12]，可以更好描述矩形框的对齐方式而不只是重叠比例。

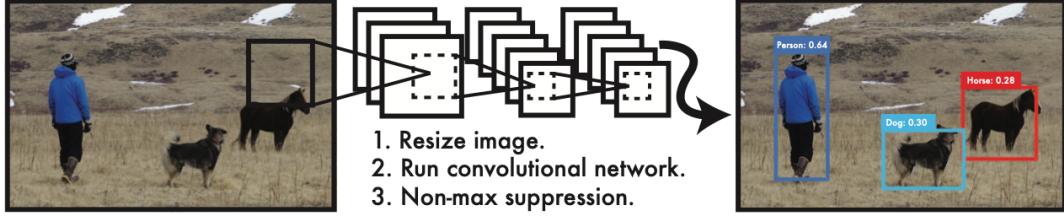


图 26: YOLO 算法流程

YOLO 的损失函数由 coordError、iouError 和 classError 三项累和构成，最终通过修正得到如图27的损失函数，其中平衡了 localization error、IOU error 和 classification error。

$$\begin{aligned}
 & \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
 & + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right] \\
 & + \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} (C_i - \hat{C}_i)^2 \\
 & + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{noobj}} (C_i - \hat{C}_i)^2 \\
 & + \sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
 \end{aligned}$$

图 27: YOLO 损失函数

考虑到本次实验的要求，与非实时检测系统（如 R-CNN）相比，YOLO 能获得更快的速度和精度（如图28所示），因而更加适合本实验。

YOLO 也存在某些方面的不足：每个网格只预测两个边界框和一个类别，距离较近的目标可能影响检测的准确度；在边界框较小的情况下产生的误差对准确度影响较大。

Real-Time Detectors	Train	mAP	FPS
100Hz DPM [30]	2007	16.0	100
30Hz DPM [30]	2007	26.1	30
Fast YOLO	2007+2012	52.7	155
YOLO	2007+2012	63.4	45
<hr/>			
Less Than Real-Time			
Fastest DPM [37]	2007	30.4	15
R-CNN Minus R [20]	2007	53.5	6
Fast R-CNN [14]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[27]	2007+2012	73.2	7
Faster R-CNN ZF [27]	2007+2012	62.1	18
YOLO VGG-16	2007+2012	66.4	21

图 28: YOLO 实验结果

3 本组工作

3.1 任务一

本小组任务一的模型在验证集上获得了 95% 的准确率，在测试集上获得了 87% 的准确率。我们采用 HOG+SVM 方式，其原理已在课程上详细说明。我们划分 20% 验证集，先在默认参数下对 HOG 参数进行调整，发现当输入图片调整至 64*64 像素，每个区域 4*4 时效果最好。这个归一化图像大小略大于训练图像的平均大小（约 47*42）。我们还尝试了不同色域下的特征提取，例如我们原先认为对于交通标志这类对比度突出的图像，HSV 色域可能较 RGB 色域特征更为明显，但在实际实验中提取特征训练表明，HSV 色域并不能达到更好的验证集准确度，因而我们仍采用 RGB 色域。随后使用 5-fold 网格搜索调整 SVM 参数，发现 C=100, gamma=0.1 时精度最高，达到 95%，具体如29所示。总体而言对于传统方法已经表现较好，但是我们没有对数据进行增强，使得特征提取有限，可能是可以未来进行改进以获得更高准确率的方法。

```
The parameters of the best model are:
{'C': 100, 'gamma': 0.1, 'kernel': 'rbf'}
Classification report for classifier GridSearchCV(cv=5, error_score=nan,
      estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                      class_weight=None, coef0=0.0,
                      decision_function_shape='ovr', degree=3,
                      gamma='scale', kernel='rbf', max_iter=-1,
                      probability=False, random_state=None, shrinking=True,
                      tol=0.001, verbose=False),
      iid='deprecated', n_jobs=12,
      param_grid={'C': [0.1, 1, 10, 100, 1000],
                  'gamma': [10, 1, 0.1, 0.01, 0.001],
                  'kernel': ('linear', 'rbf')},
      pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
      scoring='f1_weighted', verbose=1):
      precision    recall   f1-score   support

 i2      0.88      0.86      0.87      51
 i4      0.97      0.95      0.96     117
 i5      1.00      0.98      0.99     251
 io      0.94      0.95      0.94     139
 ip      0.96      1.00      0.98      44
 p11     0.94      0.97      0.96     270
 p23     0.96      0.96      0.96      26
 p26     0.95      0.93      0.94     136
 p5      0.98      0.95      0.96      42
 pl30     0.94      0.93      0.93      67
 pl40     0.96      0.98      0.97     211
 pl5      1.00      0.97      0.98      62
 pl50     0.94      0.94      0.94     160
 pl60     0.98      0.95      0.97     123
 pl80     0.97      0.95      0.96     132
 pn      0.93      0.98      0.95     480
 pne     0.99      0.98      0.99     341
 po      0.91      0.81      0.86     189
 w57      1.00      0.96      0.98      52

accuracy                           0.95      2893
macro avg       0.96      0.95      0.95      2893
weighted avg    0.95      0.95      0.95      2893
```

图 29: 任务一实验结果

在测试集上测试结果如下:

```
0 imgs missed

class : i2      recall : 0.933735
class : i4      recall : 0.936047
class : i5      recall : 0.924528
class : io      recall : 0.596667
class : ip      recall : 0.939560
class : p11     recall : 0.900943
```

```

class:p23      recall:0.987179
class:p26      recall:0.935294
class:p5       recall:0.988304
class:pl30     recall:0.906250
class:pl40     recall:0.915888
class:pl5      recall:0.988372
class:pl50     recall:0.879581
class:pl60     recall:0.930108
class:pl80     recall:0.954545
class:pn       recall:0.652318
class:pne      recall:0.956098
class:po       recall:0.626556
class:w57      recall:0.972222
Accuracy: 0.869211

```

训练代码 SVM_train.py 代码如下：

```

1 from skimage.feature import hog
2 from sklearn import svm, metrics
3 from sklearn.model_selection import GridSearchCV
4 from sklearn.model_selection import train_test_split
5 import joblib
6 from PIL import Image
7 import numpy as np
8 import os
9
10
11 def read_data(data_dir):
12     datas = []
13     labels = []
14     for label in os.listdir(data_dir):
15         print(label)
16         class_path = os.path.join(data_dir, label)
17         for img_name in os.listdir(class_path):
18             img = Image.open(os.path.join(class_path, img_name))
19             out = img.resize((64, 64), Image.ANTIALIAS)
20             fd = hog(out, orientations=9, pixels_per_cell=(4, 4), cells_per_block=(16, 16), block_norm='
21                 L2',
22                 feature_vector=True, multichannel=True)
23             datas.append(fd)
24             labels.append(label)
25     datas = np.array(datas)

```

```

26     labels = np.array(labels)
27     np.savez('hog', datas=datas, labels=labels)
28     return datas, labels
29
30
31 # DATA = NP.LOAD('../HOG.NPZ')
32 # DATAS = DATA['DATAS']
33 # LABELS = DATA['LABELS']
34 datas, labels = read_data('data/Classification/Data/Train')
35
36 svr = svm.SVC()
37 parameters = {'kernel': ('linear', 'rbf'), 'C': [0.1, 1, 10, 100, 1000], 'gamma': [10, 1, 0.1, 0.01,
38                 0.001]}
39 classifier = GridSearchCV(svr, parameters, scoring='f1_weighted', n_jobs=12, cv=5, verbose=1)
40 X_train, X_test, y_train, y_test = train_test_split(datas, labels, test_size=0.2)
41
42 classifier.fit(X_train, y_train)
43 print('The parameters of the best model are: ')
44 print(classifier.best_params_)
45 joblib.dump(classifier, '../svm.m')
46 predicted = classifier.predict(X_test)
47 print("Classification report for classifier %s:\n%s\n" % (classifier, metrics.classification_report(
    y_test, predicted)))

```

测试代码为 SVM_test.py，如下：

```

1 from skimage.feature import hog
2 import joblib
3 from PIL import Image
4 import numpy as np
5 import os
6 import json
7
8
9 def read_data(data_dir):
10     datas = []
11     names = []
12     for img_name in os.listdir(data_dir):
13         img = Image.open(os.path.join(data_dir, img_name))
14         out = img.resize((64, 64), Image.ANTIALIAS)
15         fd = hog(out, orientations=9, pixels_per_cell=(4, 4), cells_per_block=(16, 16), block_norm='L2',
16                     ,
17                     feature_vector=True, multichannel=True)
18         datas.append(fd)
19         names.append(img_name)
20         print(img_name)

```

```
21     datas = np.array(datas)
22     names = np.array(names)
23     return datas, names
24
25
26 datas, names = read_data('data/Classification/Data/Test')
27 print('Finish reading test images.')
28
29 classifier = joblib.load('../svm.m')
30 predicted = classifier.predict(datas)
31 test_labels = dict(zip(names, predicted))
32 test_json = json.dumps(test_labels)
33
34 file = open('pred1.json', 'w')
35 file.write(test_json)
36 file.close()
```

3.2 任务二

任务二基于 pytorch 实现的 ResNet18、ResNet50、Inception V3、DenseNet 网络模型实现交通标志检测任务，探寻提高检测准确率的最佳网络结构与训练方法。由于得到测试集结果的机会只有两次，所以先将数据集划分出验证集，在验证集上得到各个模型的准确率，然后选择表现最好的两个模型提交，得到最终的测试集准确率。

ResNet 原理详见 2.1.4 节，几种经典 ResNet 的网络结构如图30所示。从图中可以看出，ResNet 网络结构中的数字只包含了卷积层和全连接层，而并未包含池化层和 BN 层。以 ResNet18 为例，这里的 18=17 个卷积层 +1 个全连接层。同时可以看出，这里将较少数量的卷积层分成了不同的区块，以此建立 shortcut，达到降低学习难度，提高收敛速度的目的。我们目前主要训练了 ResNet18 和 ResNet50。

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112			7×7, 64, stride 2		
				3×3 max pool, stride 2		
conv2_x	56×56	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1			average pool, 1000-d fc, softmax		
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

图 30: ResNet 网络结构

DenseNet 原理详见 2.1.5 节, Inception 原理详见 2.1.6 节。

本实验选取交叉熵作为损失函数, 因为其数值表示各个类别出现的概率, 因此很适合作为该实验的损失函数。对于学习率而言, 只要不设得太大就不会影响模型最后的收敛情况, 只会影响其收敛速度。本实验由于硬件资源有限, 因此统一将学习率设为 0.001, 但是我们提供了可以修改学习率的命令行操作, 可以根据实际情况进行选择。另外对于损失函数的优化方法, 我们选择表现较好的 Adam 优化。

3.2.1 代码实现及实验结果

为了方便读取数据, 自定义了数据集类 my_dataset, 功能是将给定的数据转化为可以被 dataloader 调用的数据集。其中构造函数接受名为 mode 的参数, 若 mode=train 则用读取训练集的方式读取数据, 若 mode=test 则用读取测试集的方式读取数据。

相关代码位于 dataset.py 中, 如下所示。

```

1 import os
2 import torch
3 from PIL import Image
4 from torchvision import transforms
5 labels = {
6     'i2':0,
7     'i4':1,
8     'i5':2,
9     'io':3,
10    'ip':4,
```

```
11     'p5':5,
12     'p11':6,
13     'p23':7,
14     'p26':8,
15     'p15':9,
16     'p130':10,
17     'p140':11,
18     'p150':12,
19     'p160':13,
20     'p180':14,
21     'pn':15,
22     'pne':16,
23     'po':17,
24     'w57':18
25 }
26 class my_dataset(torch.utils.data.Dataset):
27     def __init__(self, mode, transforms = None):
28         self.num_class = 19
29         self.imgs = []
30         self.img_class = []
31         self.mode = mode
32         self.transforms = transforms
33         if self.mode == "train":
34             self.classes = os.listdir(os.path.join("data/Classification/Data/Train"))
35             for img_classes in self.classes:
36                 img_dir = os.listdir(os.path.join("data/Classification/Data/Train", img_classes))
37                 self.imgs += list(map(lambda x: img_classes + "/" + x, img_dir))
38                 c = [img_classes] * len(img_dir)
39                 self.img_class += c
40         elif self.mode == "test":
41             self.imgs += os.listdir(os.path.join("data/Classification/Data/Test"))
42         elif self.mode == "det-test":
43             self.imgs += os.listdir(os.path.join("data/Detection/crop"))
44
45     def __getitem__(self, idx):
46         if self.mode == "train":
47             img_path = os.path.join("data/Classification/Data/Train/", self.imgs[idx])
48         elif self.mode == "test":
49             img_path = os.path.join("data/Classification/Data/Test/", self.imgs[idx])
50         elif self.mode == "det-test":
51             img_path = os.path.join("data/Detection/crop/", self.imgs[idx])
52         img = Image.open(img_path).convert('RGB')
53         if self.transforms is not None:
54             img = self.transforms(img)
55         if self.mode == "train":
56             sample = {'image': img, 'label': labels[self.img_class[idx]]}
57         elif self.mode == "test" or self.mode == "det-test":
```

```
58         sample = {'image': img}
59     return sample
60     def __len__(self):
61         return len(self.imgs)
```

model.py 中自定义了用于分类的模型基类，派生了包括 ResNet18、ResNet50、DenseNet、Inception 等模型。代码如下。

```
1 import torch
2 import torch.nn as nn
3 import torchvision as tv
4
5 class Model(nn.Module):
6     def __init__(self, name):
7         super(Model, self).__init__()
8         self.name = name
9     def forward(self, images):
10        x = images
11        x = self.model(x)
12        return x
13
14 class ResNet18(Model):
15     def __init__(self):
16         super().__init__("ResNet18")
17         self.model = tv.models.resnet18(pretrained = True)
18         num_ftrs = self.model.fc.in_features
19         self.model.fc = nn.Linear(num_ftrs, 19)
20
21 class ResNet50(Model):
22     def __init__(self):
23         super().__init__("ResNet50")
24         self.model = tv.models.resnet50(pretrained=True)
25         num_ftrs = self.model.fc.in_features
26         self.model.fc = nn.Linear(num_ftrs, 19)
27
28 class DenseNet(Model):
29     def __init__(self):
30         super().__init__("DenseNet")
31         self.model = tv.models.densenet121(pretrained=True)
32         self.model.classifier = nn.Linear(1024, 19)
33
34 class Inception(Model):
35     def __init__(self):
36         super().__init__("inception")
37         self.model = tv.models.inception_v3(pretrained = True)
38
39         self.model.aux_logits = False
```

```

40     num_ftrs = self.model.fc.in_features
41     #NUM_AUXFTRS = SELF.MODEL.AUXLOGITS.FC.IN_FEATURES
42     self.model.fc = nn.Linear(num_ftrs, 19)
43     #SELF.MODEL.AUXLOGITS.FC = NN.LINEAR(NUM_AUXFTRS, 19)
44
45     def forward(self, x):
46         x = self.layer1(x)
47         x = self.layer2(x)
48         x = self.layer3(x)
49         x = x.view(x.size(0), -1)
50         x = self.layer4(x)
51         return x

```

train.py 中为训练函数，接受的参数如下：model 表示待训练的模型，num_epochs 表示训练的轮数，optimizer 表示使用的优化方法，loader 表示所用的训练集，val_loader 表示所用的验证集，save 表示是否保存模型，cuda 表示是否使用 GPU。

每训练一轮，会将训练的误差和现阶段的验证集准确率分别存入 results 和 accuracy 里，以便之后的结果可视化。代码如下。

```

1 import dataset
2 import model
3 import torch.nn as nn
4 import os
5 import time
6 from torch.autograd import Variable
7 import torchvision
8 import torch
9 import matplotlib.pyplot as plt
10 import pandas as pd
11 from test import test
12
13 def train(model, num_epochs, optimizer, loader, val_loader, modelname, save = True, cuda = False):
14     if cuda:
15         loss = nn.CrossEntropyLoss().cuda()
16     else:
17         loss = nn.CrossEntropyLoss()
18     results = []
19     accuracy = []
20     num = [i for i in range(num_epochs)]
21     best_acc = 0.0
22     f = open("classification.log", "w")
23     for epoch in range(num_epochs):
24         train_loss = 0.0
25         print("Epoch {}/{}".format(epoch, num_epochs - 1))
26         for data in loader:
27             model.train(True)

```

```

28         image, label = data['image'], data['label']
29         if cuda:
30             image = Variable(image.cuda())
31             label = Variable(label.cuda())
32
33         optimizer.zero_grad()
34
35         output = loss(model(image), label)
36         train_loss += output.item()
37         output.backward()
38         optimizer.step()
39         print("loss: {}".format(train_loss))
40         print("{} {}".format(epoch, train_loss), file = f)
41         results.append(train_loss)
42         acc = test(model, val_loader, cuda)
43         accuracy.append(acc.item())
44         #PRINT(ACC.ITEM())
45         if best_acc < acc:
46             torch.save(model.state_dict(), 'params'+modelname+'.pkl')
47             best_acc = max(best_acc, acc)
48             print('best accuracy is: {}'.format(best_acc))
49             dataframe = pd.DataFrame({'epoch': num, 'result': results})
50             dataframe.to_csv("result.csv")
51             dataframe = pd.DataFrame({'epoch': num, 'accuracy': accuracy})
52             dataframe.to_csv("accuracy.csv")
53
54     return model

```

test.py 为测试函数，接受的参数如下：model 表示待测试的模型，loader 表示所用的验证集，cuda 表示是否使用 GPU。该函数返回模型在验证集上的准确率。代码如下。

```

1 import dataset
2 import time
3 import torch.nn as nn
4 import torch
5 import model
6 from torch.autograd import Variable
7 import torchvision
8 import os
9
10 def test(model, loader, cuda = False):
11     model.eval()
12     if cuda:
13         correct = torch.zeros(1).squeeze().cuda()
14         total = torch.zeros(1).squeeze().cuda()
15     else:
16         correct = torch.zeros(1).squeeze()

```

```

17     total = torch.zeros(1).squeeze()
18
19     for i, data in enumerate(loader):
20         image, label = data['image'], data['label']
21         if cuda:
22             image = Variable(image.cuda())
23             label = Variable(label.cuda())
24
25         output = model(image)
26
27         pred = torch.argmax(output, 1)
28         correct += (pred == label).sum().float()
29         total += len(label)
30     print("accuracy: {}".format(correct / total))
31     return correct / total

```

main.py 为主函数，接受的主要命令行参数如下（其余详见说明文档）：

- --cuda，格式为 True 或 False，表示是否使用 GPU，默认为 True
- --epoch，一个整数，表示训练的轮数，默认为 100
- --model，格式为 ResNet18，ResNet50，Inception，DenseNet，表示使用的模型

先将图片进行归一化、调整大小等预处理，再将所给的数据集划分为训练集和验证集，根据使用的模型不同和是否使用 GPU 来进行训练。训练的过程中会输出训练误差和准确率，分别保存在了对应的 CSV 文件中。代码如下。

```

1 import model
2 from dataset import my_dataset
3 import torch
4 import torch.optim as optim
5 from train import train
6 from torchvision import transforms
7 from test import test
8 import argparse
9 def parse_args():
10     parser = argparse.ArgumentParser(description='Classification')
11     parser.add_argument('--cuda', default='True')
12     parser.add_argument('--epoch', type=int, default=100)
13     parser.add_argument('--model', default='ResNet18', help = 'You can choose: ResNet18, ResNet50,
14                         Inception, DenseNet')
15     parser.add_argument('--lr', type = float, default=0.001)
16     parser.add_argument('--BS', type = int, default=16, help = 'BatchSize')
17     parser.add_argument('--load', default='False', help='if you load an existing model or not')
18     return parser.parse_args()
19 def main():

```

```

19     args = parse_args()
20     save = True
21     use_gpu = args.cuda == 'True'
22     load = args.load == 'True'
23     train_tfs = transforms.Compose([
24         transforms.Resize(299),
25         transforms.RandomSizedCrop(299),
26         transforms.RandomHorizontalFlip(),
27         transforms.ToTensor(),
28         transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
29     ])
30     ds = my_dataset("train", train_tfs)
31     dataset_size = ds.__len__()
32     print(dataset_size)
33     train_ds, val_ds = torch.utils.data.random_split(ds, [13000, 1463])
34     train_loader = torch.utils.data.DataLoader(train_ds, args.BS, False, num_workers = 8)
35     val_loader = torch.utils.data.DataLoader(val_ds, args.BS, False, num_workers = 8)
36     print('train: ', len(train_ds))
37     print('validation: ', len(val_ds))
38     print(type(ds), type(train_ds))
39     if args.model == 'ResNet18':
40         test_model = model.ResNet18()
41     if args.model == 'ResNet50':
42         test_model = model.ResNet50()
43     if args.model == 'Inception':
44         test_model = model.Inception()
45     if args.model == 'DenseNet':
46         test_model = model.DenseNet()
47     if use_gpu:
48         test_model = test_model.cuda()
49     if load:
50         test_model.load_state_dict(torch.load('params' + args.model + '.pkl'))
51     optimizer = optim.Adam(test_model.parameters(), lr = args.lr)
52     print(use_gpu)
53     result = train(test_model, args.epoch, optimizer, train_loader, val_loader, args.model, save,
54                     use_gpu)
55     test(result, val_loader, use_gpu)
56 if __name__ == "__main__":
57     main()

```

预测文件 pred.py，代码如下。

```

1 import model
2 from dataset import my_dataset
3 import torch
4 import torch.optim as optim
5 from train import train

```

```
6 from torchvision import transforms
7 from test import test
8 from torch.autograd import Variable
9 import json
10 import argparse
11 def parse_args():
12     parser = argparse.ArgumentParser(description='Prediction')
13     parser.add_argument('--model', default='ResNet18', help = 'You can choose: ResNet18, ResNet50,
14                         Inception, DenseNet')
15     return parser.parse_args()
16 labels = [
17     'i2',
18     'i4',
19     'i5',
20     'io',
21     'ip',
22     'p5',
23     'p11',
24     'p23',
25     'p26',
26     'p15',
27     'p130',
28     'p140',
29     'p150',
30     'p160',
31     'p180',
32     'pn',
33     'pne',
34     'po',
35     'w57'
36 ]
37 def main():
38     args = parse_args()
39     name = open('pred2.json', 'w')
40     train_tfs = transforms.Compose([
41         transforms.Resize(299),
42         transforms.RandomSizedCrop(299),
43         transforms.RandomHorizontalFlip(),
44         transforms.ToTensor(),
45         transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
46     ])
47     ds = my_dataset("test", train_tfs)
48     dataset_size = ds.__len__()
49     print(dataset_size)
50     loader = torch.utils.data.DataLoader(ds, 16, False)
51     test_model = model.Inception()
```

```
52     test_model = test_model.cuda()
53     test_model.load_state_dict(torch.load('params' + args.model + '.pkl'))
54
55     preds = []
56     num = 0
57     for i, data in enumerate(loader):
58
59         image = data['image']
60         print(image.size())
61
62         image = Variable(image.cuda())
63         # LABEL = VARIABLE(LABEL.CUDA())
64
65         output = test_model(image)
66
67         pred = torch.argmax(output, 1)
68         # IF I < 10:
69         #   PRINT(PRED)
70         for j in pred:
71             if i < 10:
72                 print(j)
73             preds[ds.imgs[num]] = labels[j.int()]
74             num = num + 1
75         print(json.dumps(preds))
76         print(num)
77         name.write(json.dumps(preds))
78         name.close()
```

网络结构为 ResNet18, 学习率为 0.001, 损失函数为交叉熵, 更新策略为 Adam, BatchSize 为 32, 训练轮数为 1000, 所得损失函数输出与准确率如图31所示。ResNet18 在训练了 100 个 epoch 后获得了 95% 的准确率, 与传统方法持平。

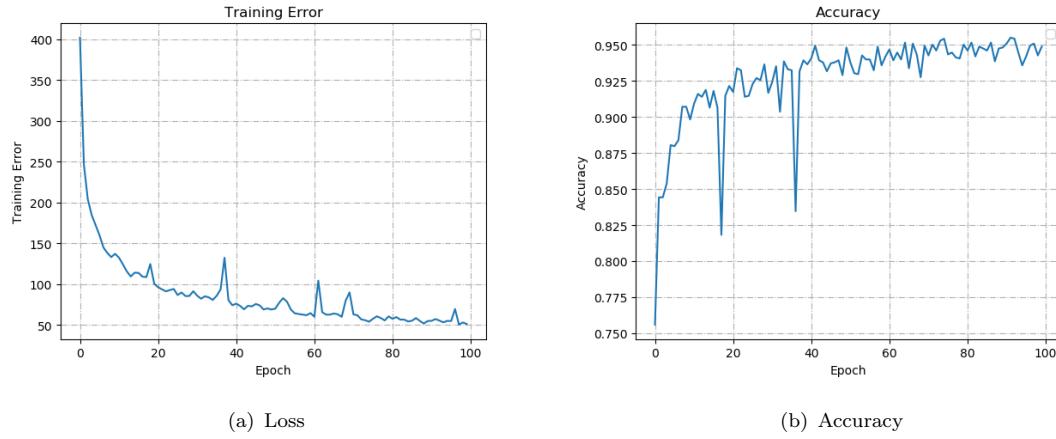


图 31: ResNet18 训练曲线

网络结构为 ResNet50, 学习率为 0.001, 损失函数为交叉熵, 更新策略为 Adam, BatchSize 为 32, 训练轮数为 1000, 所得损失函数输出与准确率如图32所示。对于规模更大的 ResNet50, 我们获得了 96% 的准确率, 略高于传统方法。1000 个 epoch 后, 二者已基本收敛, 这表明 ResNet 在当前参数下很难做到超过 96% 的准确率。

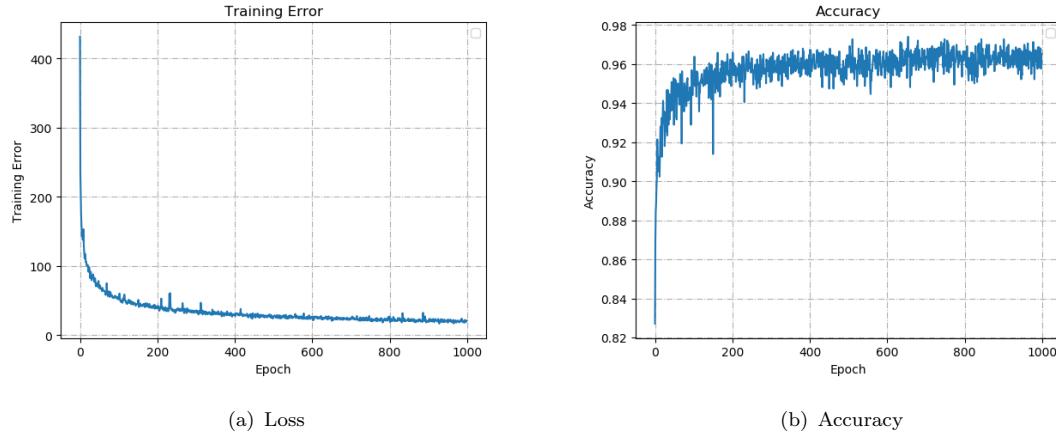


图 32: ResNet50 训练曲线

网络结构为 DenseNet, 学习率为 0.001, 损失函数为交叉熵, 更新策略为 Adam, BatchSize 为 8, 训练轮数为 100, 所得损失函数输出与准确率如图33所示。准确率基本收敛在 95%。

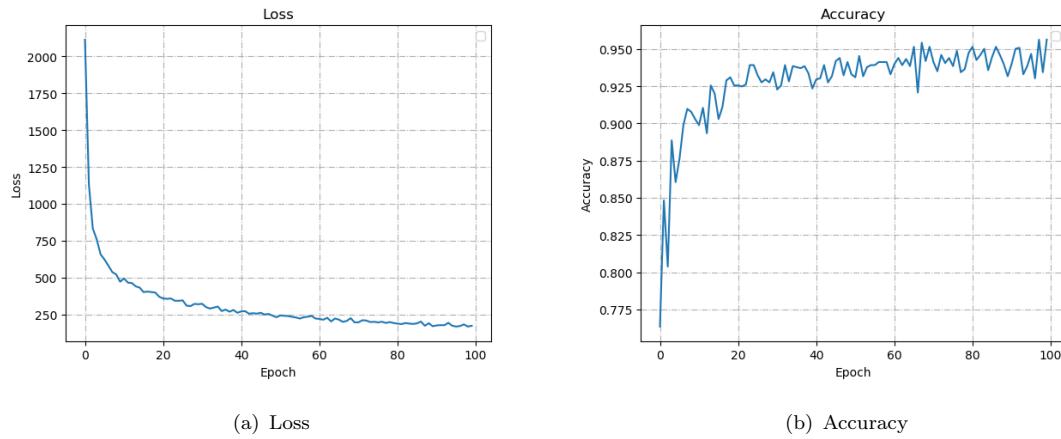


图 33: DenseNet 训练曲线

网络结构为 Inception, 学习率为 0.001, 损失函数为交叉熵, 更新策略为 Adam, BatchSize 为 16, 训练轮数为 250, 所得损失函数输出与准确率如图34所示。准确率大致收敛在 97.5%, 最高的验证集准确率可达到 98.2%。

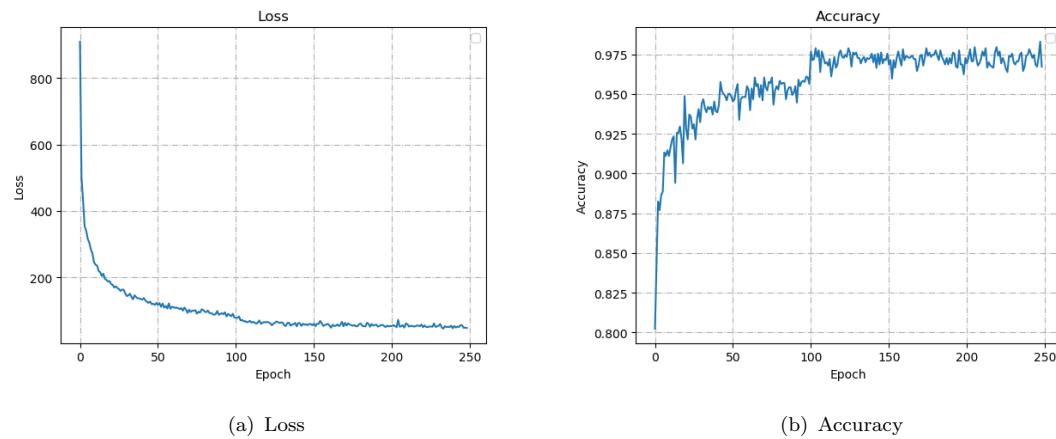


图 34: Inception 训练曲线

3.2.2 总结与反思

综上, 我们发现在验证集上表现最好的两个网络是 ResNet50 与 Inception, 因此分别将两个网络进行测试, 所得结果如下。

ResNet50 的测试集结果:

```
0 imgs missed
class : i2      recall : 0.935065
class : i4      recall : 0.934343
class : i5      recall : 0.928230
class : io      recall : 0.747899
class : ip      recall : 1.000000
class : p11     recall : 0.876712
class : p23     recall : 1.000000
class : p26     recall : 0.958580
class : p5      recall : 0.952632
class : pl30    recall : 0.906593
class : pl40    recall : 0.904762
class : pl5     recall : 0.983146
class : pl50    recall : 0.792035
class : pl60    recall : 0.902703
class : pl80    recall : 0.937500
class : pn      recall : 0.750943
class : pne     recall : 0.966019
class : po      recall : 0.652510
class : w57     recall : 0.994681
Accuracy: 0.888947
```

Inception 的测试集结果:

```
class : i2      recall : 0.981818
class : i4      recall : 0.984293
class : i5      recall : 0.900901
class : io      recall : 0.837838
class : ip      recall : 1.000000
class : p11     recall : 0.899543
class : p23     recall : 1.000000
class : p26     recall : 0.910000
class : p5      recall : 0.978378
class : pl30    recall : 1.000000
```

```
class : p140      recall : 0.937500
class : p15       recall : 0.988950
class : p150      recall : 0.908654
class : p160      recall : 0.927083
class : p180      recall : 0.964103
class : pn        recall : 0.802419
class : pne       recall : 0.929907
class : po        recall : 0.744589
class : w57       recall : 0.994737
Accuracy: 0.924474
```

虽然两个网络在验证集上表现的差距不大，只有 1% 左右，但这种差距在测试集上被放大了，说明增加训练轮数，即使只能使模型在验证集上的表现获得很小的提升，对于提高模型在测试集上的表现也是很有帮助的。另外二者的准确率均远低于在验证集上的准确率，这是因为由于测试次数只有两次，所以我们只能选择在验证集上表现最好的两个模型，多少会出现过拟合验证集的问题。如果能测试多次，我们会选择在验证集上表现最好的 5 个模型，分别测试其在测试集上的准确率，这样我们也许能选择出在测试集上表现更好的模型。

我们发现尽管在有些类别的表现上 ResNet50 要优于 Inception(如 i5, p26 等)，但是在几个较难分类的数据类别上(如 po, pn 等)Inception 的表现要明显优于 ResNet50。这说明相比于 ResNet50，Inception 的鲁棒性更好，更加适合多分类的任务。总体来讲，Inception 的准确率更高，可以在测试集上达到 92% 以上。

3.3 任务三

本实验采用了两种算法：KNN 算法与原型网络。

3.3.1 KNN

K 临近算法 (KNN, k-NearestNeighbor) 是机器学习中较为初级朴素的算法，所谓 K 临近，就是指每个样本都可以用最近的 K 个邻居代表。KNN 是一种惰性算法，不需要实际的训练，应该事先给出数据集的分类以及特征，当收到新的待分类样本之后直接进行处理。

KNN 是基于特征向量之间的距离进行分类的，通过参数 k 以及数据集特征提取进行调参，若与待分类样本距离最邻近 k 个样本中的大多数属于某一个类别，则待分类的样本就分到这一类别。

3.3.1.1 算法描述

- 计算训练数据和测试数据的特征值
- 计算测试数据和各个训练数据之间的距离
- 对距离进行排序
- 选取距离最小的 K 个点，确定出现频率最高的类别，返回该类别标签

注：由于这次实验用的是 one shot，因此 k=1，直接选取最近的即可。另外，我们采用 HOG 提取图片的特征向量。距离的定义可为曼哈顿距离或欧氏距离。

欧几里得距离：

$$E(x, y) = \sqrt{\sum_{i=1}^N (x_i - y_i)^2}$$

曼哈顿距离：

$$E(x, y) = \sum_{i=1}^N |x_i - y_i|$$

3.3.1.2 代码实现及实验结果

图片读取，灰度处理等预处理在 util.py 中完成，主要部分代码如下。

```

1 def read_image(task):
2     data = []
3     label = []
4     with open('data/Classification/Data/train.json', 'r') as fp:
5         js = json.load(fp)
6         v0 = 0
7         for f, v in js.items():
8             if v == 'po' or v == 'io':
9                 continue
10            if task == "train":
11                if v != v0:
12                    image = Image.open('data/Classification/Data/Train/' + v + '/' + f)
13                    v0 = v
14                else:
15                    continue
16            else:
17                image = Image.open('data/Classification/Data/Test/' + v + '/' + f)
18            imafter = image.resize((64, 64))
19            imafter = imafter.convert("L")
20            data.append(np.array(imafter))

```

```

21         label.append(v)
22     return data, label
23
24
25 def normalization(data):
26     data = np.array(data)
27     maxn = np.max(data, axis=0)
28     minn = np.min(data, axis=0)
29     return (data - minn) / (maxn - minn)

```

KNN 部分代码如下，位于 KNN.py 中。

```

1 class KNN:
2     def __init__(self, neighbors, data, label, method):
3         self.method = method
4         self.neighbors = neighbors
5         self.data = data
6         self.label = label
7         self.num = len(label)
8
9     def judge(self, test):
10        dist_arr = {}
11        for i in range(len(self.label)):
12            dist_arr[self.label[i]] = self.dist(self.data[i], test)
13            # SELF.DIST_ARR.APPEND(SELF.DIST(X, TEST))
14        dist_arr = sorted(dist_arr.items(), key=lambda kv: (kv[1], kv[0]))
15        return dist_arr[self.neighbors - 1][0]
16
17    def dist(self, x, y):
18        if self.method == "eu":
19            temp = np.sum((x - y) * (x - y))
20            if temp < 0.1:
21                return temp * 1000
22            else:
23                return temp
24        if self.method == "m":
25            return np.sum(np.abs(x - y))

```

HOG 特征提取位于 fewshot.py 中，主要代码如下。

```

1 train_raw_data, train_label = read_image("train")
2 test_raw_data, test_label = read_image("test")
3 print("finish reading raw image")
4 sumn = len(test_label)
5 train_hog = []
6 test_hog = []
7 for i, raw in enumerate(train_raw_data):
8     train_hog.append(

```

```

9      hog(raw,
10         orientations=9,
11         pixels_per_cell=(4, 4),
12         cells_per_block=(8, 8)))
13 for i, raw in enumerate(test_raw_data):
14     test_hog.append(
15         hog(raw,
16             orientations=9,
17             pixels_per_cell=(4, 4),
18             cells_per_block=(8, 8)))
19     if i % 1000 == 0:
20         print("finish hog process " + str(i / sumn))
21
22 print("finish hog process !!")

```

进行图像分类，并进行正确率的计算代码也在 fewshot.py 中，代码如下。

```

1 model = KNN(neighbors=1, data=train_hog, label=train_label, method="eu")
2 pridict = []
3 for i in range(len(test_label)):
4     pridict.append(model.judge(test_hog[i]))
5     if (i % 1000 == 0):
6         print("finish judge :" + str(i / sumn))
7 correct = 0
8 for i in range(len(pridict)):
9     if pridict[i] == test_label[i]:
10        correct += 1
11 print(correct / sumn)

```

实验结果如图35所示，我们采用分类任务的完整数据集中每类挑选一张图片用以训练，其余所有图片作为验证集得到 43% 的准确率。

```

finish judge :0.4676539360872954
finish judge :0.5455962587685113
finish judge :0.6235385814497272
finish judge :0.7014809041309431
finish judge :0.779423226812159
finish judge :0.857365549493375
finish judge :0.9353078721745908
0.4299298519095869
Terminated

```

图 35: 任务三实验结果

3.3.2 原型网络

原型网络原理见 2.1.7 节。我们使用<https://github.com/orobix/Prototypical-Networks-for-Few-shot-Learning-PyTorch>中的原型网络 pytorch 实现进行修改，从而实现我们的目标任务。

总体实现方案上，在读取图片的过程中，由于交通标志色彩单一，采取转换成灰度图像的方法，并统一压缩成 28*28 大小。在 protonet 模型构建方面，每一层网络采用一个卷积层，一个 batchnorm，一个激活函数层，一个池化层。模型可以通过 GPU 训练。对于每一个 episode 随机选择支持集和查询集通过 pytorch 自定义 batch_sampler 实现。对于损失函数的随机梯度下降方面采用 adam 算法。另外设置了学习率随 epoch 次数而下降。

验证集预训练中，我们使用前两个任务给出的数据集，每类划出一张图作为训练样本。

训练：i2,i4,i5,ip,p5,p11,p23,p26,pl5,pl30,pl40,pl50

验证：pl60,pl80,pn,pne,w57

训练曲线如图36所示。在验证集上的最优结果为 77%。

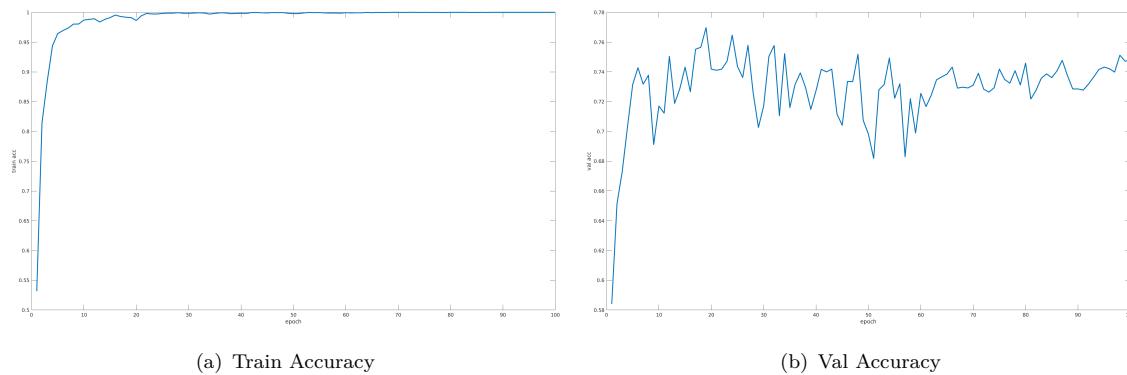


图 36: 原型网络训练曲线

3.3.3 总结与反思

原型网络训练后测试集准确度如下。

```
0 imgs missed
class : p1      recall :0.900000
class : p12     recall :0.888889
class : p14     recall :0.909091
class : p17     recall :0.703704
class : p19     recall :0.500000
```

```
class:p22      recall:0.739130
class:p25      recall:0.809524
class:p27      recall:0.875000
class:p3       recall:0.640000
class:p6       recall:0.733333
class:p9       recall:0.894737
Accuracy: 0.745455
```

KNN 实现起来较为简单，但是需要大量的存储空间，而且需要计算每个待分类样本与训练集的距离，其时间复杂度为 $O(mn)$ (m 、 n 分别为训练样本、验证样本数量)，较为耗时。

对于本实验而言，KNN 没有预训练和训练的过程，对现有的数据的利用不够充分，同时较为依赖特征的提取，不完全适宜本类任务，很难有较大提升空间。

而对于原型网络，经测试发现正确率浮动与 60%~80% 之间，存在较大的不确定性，而且对预训练给出的数据集依赖较大。其中一个明显的问题是，通过原型的距离来度量样本之间的相似度，这种方法是否合理。原型网络在 omnig 小数据集中取得了很好的结果，这与数据集有很大的关系，手写体白底黑字，特征提取比较明显。但是对于交通标志以及一般的图片，效果就不是很好了。因此，对于样本提取特征的方法、如何衡量样本之间的相似程度可以作为未来研究的方向。

3.4 任务四

本实验主要使用 YOLOv3 完成，YOLO 原理详见 2.2.2 节。我们使用<https://github.com/ultralytics/yolov3>中的 pytorch 实现。

3.4.1 实验过程及结果

我们将数据集中给出的标记文件转换为 YOLOv3 所需的标注格式（data 文件夹中的 images 与 labels 文件夹），同时区分验证集与训练集。这项任务编写脚本 makeLabel.py 完成，其读取 train_annotations.json，将其中的数据转化为 labels 文件夹中的标签（需要将坐标转为相对坐标），并划分出 10% 的训练集，写入 val.txt，训练数据写入 train.txt。标签每行以“class x_center y_center width height”的格式组成，以 60.jpg 为例，其有三个交通标志，标签文件形如：

```
13 0.7396737060546874 0.43244326171875 0.01933764648437497 0.01982109375000002
5 0.7570751708984376 0.4324431640625 0.016445263671875043 0.018854101562500003
15 0.77351728515625 0.43292666015625003 0.017405664062499993 0.01885429687500001
```

同时修改 yolov3.cfg 使之适应 19 个类别，将三个 yolo 层的 classes 改为 19，yolo 层之前的卷积层中 filters 改为 $3 \times (19 + 5) = 72$ 。

编写 traffic.data 作为训练所需的属性，traffic.names 存储类别名（略去）。

```
classes= 19
train   = data/train.txt
valid   = data/val.txt
names   = data/traffic.names
backup  = backup/
```

makeLabel.py 代码如下：

```
1 import os
2 import random
3 import json
4 from PIL import Image
5 import shutil
6
7 val_percent = 0.1
8 datapath = 'data/Detection'
9 imgsave = 'Detection/yolov3/data/images'
10 labelsave = 'Detection/yolov3/data/labels'
11 annotations = json.load(open(os.path.join(datapath, 'train_annotations.json')))[['imgs']]
12 labels = ['i2', 'i4', 'i5', 'io', 'ip', 'p11', 'p23', 'p26', 'p5', 'pl30', 'pl40', 'pl5', 'pl50', 'pl60', 'pl80', 'pn', 'pne', 'po', 'w57']
13 for img_data in annotations.values():
14     path = os.path.join(datapath, img_data['path'])
15     img = Image.open(path)
16     shutil.copy(path, imgsave)
17     label_file = open(os.path.join(labelsave, str(img_data['id']) + '.txt'), 'w')
18     for objects in img_data['objects']:
19         bbox = objects['bbox']
20         xmin = bbox['xmin']
21         xmax = bbox['xmax']
22         ymin = bbox['ymin']
23         ymax = bbox['ymax']
24         classnum = labels.index(objects['category'])
25         xceter = (xmin + xmax) / (2.0 * img.size[0])
26         yceter = (ymin + ymax) / (2.0 * img.size[1])
27         width = (xmax - xmin) / img.size[0]
28         height = (ymax - ymin) / img.size[1]
29         label_file.write(
30             str(classnum) + ' ' + str(xceter) + ' ' + str(yceter) + ' ' + str(width) + ' ' + str(
31             height) + '\n')
32 num = len(annotations)
```

```

33 tv = int(num * val_percent)
34 name = []
35 for img_data in annotations.values():
36     name.append(img_data['id'])
37 val = random.sample(name, tv)
38
39 ftrain = open('Detection/yolov3/data/train.txt', 'w')
40 fval = open('Detection/yolov3/data/val.txt', 'w')
41
42 for img_data in annotations.values():
43     name = 'data/images/' + str(img_data['id']) + '.jpg\n'
44     if img_data['id'] in val:
45         fval.write(name)
46     else:
47         ftrain.write(name)
48
49 ftrain.close()
50 fval.close()

```

python train.py -cfg cfg/yolov3.cfg -data data/traffic.data -batch-size 8 -weights "" 进行训练。训练 2500 个 epoch 后，在验证集上达到了 0.68 的准确率，0.83 的召回率，mAP (mean average precision) 达到 0.798，F1 score 达到 0.743 (即准确率与召回率的调和平均)。训练过程中的迭代曲线如图37所示，其中 GIoU (Generalized Intersection over Union) 表示物体框的对齐方式以及重叠面积，是检测任务的损失函数，而 objectness 反映了框中包含物体的可能性。

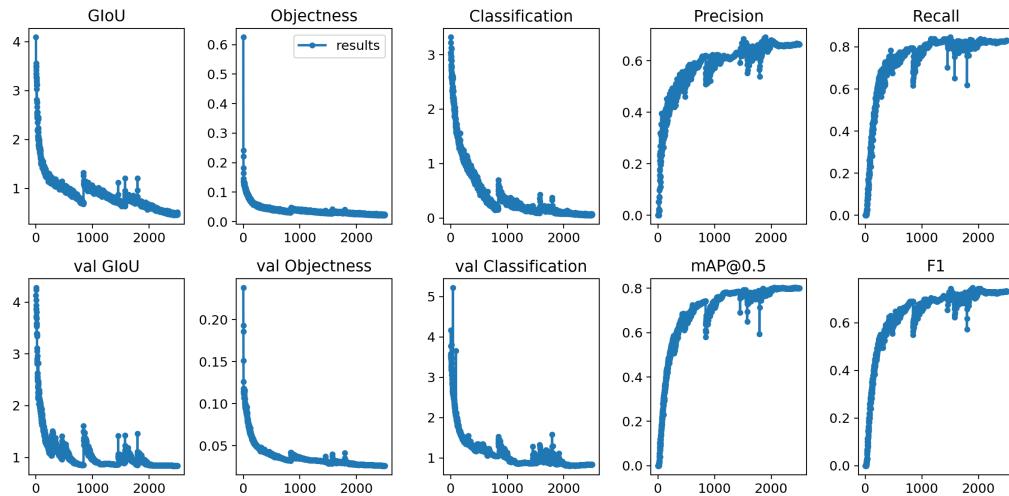


图 37: YOLOv3 训练曲线

以 `python detect.py -cfg cfg/yolov3.cfg -names data/traffic.names -weights weights/best.pt` 的方式进行检测，检测 `6026.jpg` 和 `82399.jpg` 两张（不在训练集中），结果如图38。这两张图片都正确检测出了内部所含的 2 或 4 个交通标志。



图 38: YOLOv3 检测结果

另一方面，可以将 YOLO 中的分类器部分换成任务二中训练得到的分类器。`predLabel.py` 中，我们根据 YOLO 检测到的标志，裁剪出后再使用任务二中分类器，裁剪出的标志如图39所示。代码如下，`-model` 参数可选择模型（默认为 YOLO 分类），`-cuda` 可选择是否使用 GPU。

```

1 import os
2 import json
3 import subprocess
4 from PIL import Image
5 import torch
6 from torchvision import transforms
7 from torch.autograd import Variable
8 import sys
9 import argparse
10
11 sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)), '..'))
12 from Classification import model
13 from Classification.dataset import my_dataset
14
15

```

```
16 def parse_args():
17     parser = argparse.ArgumentParser(description='Detection')
18     parser.add_argument('--cuda', default='True')
19     parser.add_argument('--model', default='YOLO')
20     return parser.parse_args()
21
22
23 args = parse_args()
24 use_gpu = args.cuda == 'True'
25 YOLO = args.model == 'YOLO'
26 # SUBPROCESS.CALL(['PYTHON', 'DETECT.PY', '--CFG', 'CFG/YOLOV3.CFG',
27 #                   '--NAMES', 'DATA/TRAFFIC.NAMES', '--WEIGHTS', 'WEIGHTS/BEST.PT', '--SOURCE',
28 #                   '../..../DATA/DETECTION/TEST', '--SAVE-TXT'], cwd='DETECTION/YOLOV3')
29
30 datapath = 'data/Detection/test'
31 predpath = 'Detection/yolov3/output'
32 croppath = 'data/Detection/crop'
33 if not os.path.exists(croppath):
34     os.makedirs(croppath)
35 labels = ['i2', 'i4', 'i5', 'io', 'ip', 'p11', 'p23', 'p26', 'p5', 'p130', 'p140', 'p15', 'p150', 'p160',
36             'p180', 'pn',
37             'pne', 'po', 'w57']
38
39 annotations = {}
40 if not YOLO:
41     print('Initializing model.')
42     if args.model == 'ResNet18':
43         test_model = model.ResNet18()
44     if args.model == 'ResNet50':
45         test_model = model.ResNet50()
46     if args.model == 'Inception':
47         test_model = model.Inception()
48     if args.model == 'DenseNet':
49         test_model = model.DenseNet()
50     if use_gpu:
51         test_model = test_model.cuda()
52     tfs = transforms.Compose([
53         transforms.Resize(299),
54         transforms.RandomSizedCrop(299),
55         transforms.RandomHorizontalFlip(),
56         transforms.ToTensor(),
57         transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
58     ])
59     if args.model == 'ResNet18':
60         test_model.load_state_dict(torch.load('params18.pkl'))
61     if args.model == 'ResNet50':
62         test_model.load_state_dict(torch.load('params50.pkl'))
```

```
62     if args.model == 'Inception':
63         test_model.load_state_dict(torch.load('paramsdnn.pkl'))
64     if args.model == 'DenseNet':
65         test_model.load_state_dict(torch.load('paramsInception.pkl'))
66     print('Initialization finished')
67 for img_name in os.listdir(datapath):
68     img_id = os.path.splitext(img_name)[0]
69     img = Image.open(os.path.join(datapath, img_name)).convert('RGB')
70     print(img_id)
71     path = os.path.join(datapath, img_name)
72     label_path = os.path.join(predpath, img_id + '.jpg.txt')
73     label_object = []
74     if os.path.exists(label_path):
75         label_file = open(label_path, 'r')
76         lines = label_file.readlines()
77         cnt = 0
78         for line in lines:
79             label = line.split()
80             bbox = {'xmax': int(label[2]), 'xmin': int(label[0]), 'ymax': int(label[3]), 'ymin': int(label[1])}
81             if not YOLO:
82                 crop = img.crop((bbox['xmin'], bbox['ymin'], bbox['xmax'], bbox['ymax']))
83                 crop.save(os.path.join(croppath, img_id + '_' + str(cnt) + '.jpg'))
84             label_object.append({'bbox': bbox, 'category': labels[int(label[4])], 'score': float(label[5])})
85             cnt = cnt + 1
86     annotations[img_id] = {'objects': label_object}
87
88 if not YOLO:
89     ds = my_dataset("det-test", tfs)
90     loader = torch.utils.data.DataLoader(ds, 16, False)
91     preds = {}
92     num = 0
93     for i, data in enumerate(loader):
94         image = data['image']
95         if use_gpu:
96             image = Variable(image.cuda())
97             output = test_model(image)
98             pred = torch.argmax(output, 1)
99             for j in pred:
100                 crop_name = ds.imgs[num]
101                 img_id = crop_name.split('_')[0]
102                 cnt = int(crop_name.split('_')[1].split('.')[0])
103                 annotations[img_id]['objects'][cnt]['category'] = labels[j.int()]
104                 num = num + 1
105
106 test_json = json.dumps({'imgs': annotations})
```

```

107 file = open('pred_annotations8.json', 'w')
108 file.write(test_json)
109 file.close()

```

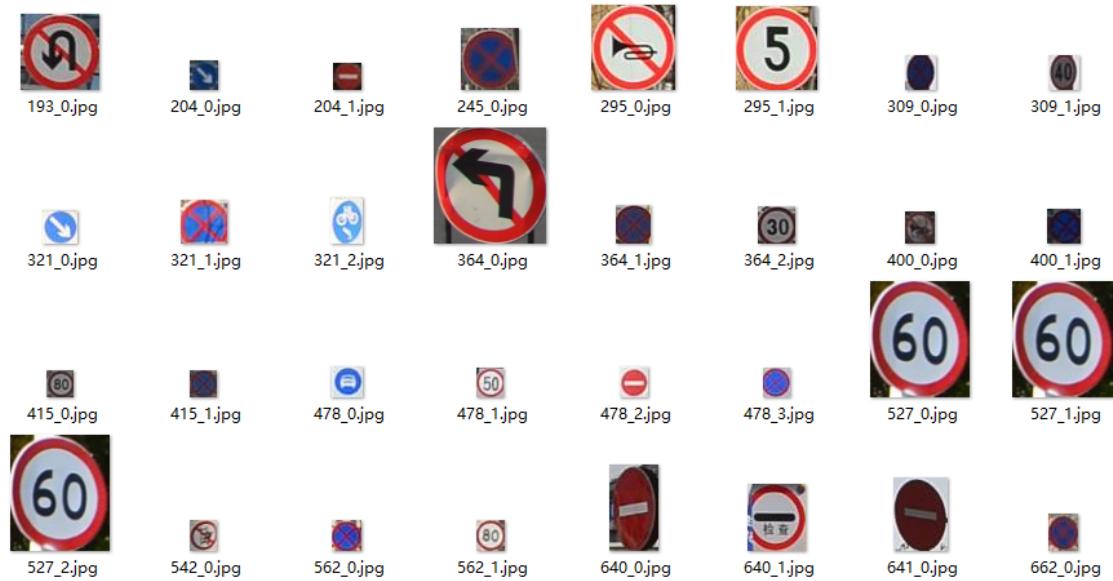


图 39: 裁剪交通标志

3.4.2 总结与反思

YOLOv3 测试集上准确率如下:

```

iou:0.5, size:[0,400), types:[ pl30, ... total 19...],
accuracy:0.8159812597605414, recall:0.6971314209472982

```

裁剪后使用 Inception 准确率如下:

```

iou:0.5, size:[0,400), types:[ pl30, ... total 19...],
accuracy:0.6470588235294118, recall:0.5528129864354013

```

本实验中我们采用了 YOLOv3 而非更新的 YOLOv4 是因为后者需要占用更大规模的显存，而超出了所能使用的极限。总体而言，YOLOv3 达到了较高的准确度。由于训练图片过大，而交通标志实际较小，而 YOLO 训练完整大小图片将占用过多计算资源而难以完成。因此我们将图片压缩到了 320*640，这使得交通标志过小而损失了较多信息，因此对准确度有一定影响。同时，检测需要较多计算资源，训练 2500 个 epoch 我们使用了共计五天服务器 GPU 资源，进一步利用计算资源提升精度有一定困难。

另一方面，实验要求使用任务二中的分类器，但 YOLO 实际上检测和分类是一起完成的，它们的损失函数是耦合的。强行将其拆开必然降低了准确度。但之所以 Inception 在裁剪出来后的交通标志上准确度不高的原因尚不清楚。这可能是因为预处理出现了不统一的情形。实际上，任务二训练集中图片是四通道的，而任务四中图片是三通道的。若要真正分离检测与分类，可能 Faster RCNN 更加合适。

4 代码运行方式

代码运行方式详见代码根目录下的 README.md。

5 成员的分工及贡献

刘圣禹：独立完成任务二、报告撰写；马啸阳：独立完成任务一、四，报告撰写、整理排版；袁健皓：独立完成任务三、报告撰写。

参考文献

- [1] N. Dalal and B. Triggs, “Histograms of oriented gradients for human detection,” in *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR’05)*, vol. 1. IEEE, 2005, pp. 886–893.
- [2] D. CireşAn, U. Meier, J. Masci, and J. Schmidhuber, “Multi-column deep neural network for traffic sign classification,” *Neural networks*, vol. 32, pp. 333–338, 2012.
- [3] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.
- [4] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [5] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

- [6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.
- [7] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.
- [8] O. Vinyals, C. Blundell, T. Lillicrap, D. Wierstra *et al.*, “Matching networks for one shot learning,” in *Advances in neural information processing systems*, 2016, pp. 3630–3638.
- [9] J. Snell, K. Swersky, and R. Zemel, “Prototypical networks for few-shot learning,” in *Advances in neural information processing systems*, 2017, pp. 4077–4087.
- [10] R. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.
- [11] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.
- [12] S. H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. D. Reid, and S. Savarese, “Generalized intersection over union: A metric and A loss for bounding box regression,” *CoRR*, vol. abs/1902.09630, 2019. [Online]. Available: <http://arxiv.org/abs/1902.09630>