人工智能 实验 1【街景字符识别】

实验目的

模型调优，以在测试集上达到更高的准确率。

日期: 2025-04-28 21:17:15
分数: 0.9095

日期: 2025-04-28 20:04:53
分数: 0.9038

日期: 2025-04-28 18:15:57
分数: 0.8939

日期: 2025-04-28 17:34:34
分数: 0.8900

日期: 2025-04-28 16:22:29
分数: 0.8272

日期: 2025-04-28 14:01:04
分数: 0.7185

日期: 2025-04-17 23:25:00
分数: 0.7995

日期: 2025-04-17 16:07:05
分数: 0.8873

实验结果

准确率**90.95%**。稍后论证做法和原因。

运行环境

本机运行。配置如下。

CPU
  12th Gen Intel(R) Core(TM) i9-12900H

GPU
  NVIDIA GeForce RTX 3070 Ti Laptop GPU
  驱动程序版本:　　32.0.15.6607
  驱动程序日期:　　2024/10/20
  专用 GPU 内存　　5.3/8.0 GB

Pytorch 2.7.0 + CUDA 12.6

模型设计

对比 baseline.py，尝试进行了以下架构优化：

1. **骨干网络升级：**

```
self.backbone = timm.create_model(model_name, pretrained=True, features_only=False)
# 支持 EfficientNetV2、ConvNext、Swin Transformer 等
```

2. **注意力机制：**添加 CBAM 双重注意力模块，同时关注通道和空间特征：

```
class CBAM(nn.Module):
    def __init__(self, channel, reduction=16):
        # ...通道注意力...
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.max_pool = nn.AdaptiveMaxPool2d(1)
        self.fc = nn.Sequential(...)

        # ...空间注意力...
        self.conv = nn.Sequential(
            nn.Conv2d(2, 1, kernel_size=7, padding=3, bias=False),
            nn.Sigmoid()
```

```
        )
```

3. **多层次特征融合**：使用更复杂的特征提取网络，尤其是 Transformer 模型：

```
# Transformer 特征融合网络
self.neck = nn.Sequential(
    nn.Linear(feature_dim, 2048),
    nn.LayerNorm(2048),  # 使用 LayerNorm 替代 BatchNorm
    nn.GELU(),  # 使用 GELU 激活函数
    nn.Dropout(dropout),
    nn.Linear(2048, 1536),
    nn.LayerNorm(1536),
    nn.GELU(),
    nn.Dropout(dropout)
)
```

4. **字符位置感知**：引入位置编码，增强模型对不同字符位置的感知：

```
# 位置编码参数
self.position_embedding = nn.Parameter(t.randn(4, 256))
# 应用位置编码
position_feat = common_feat + self.position_embedding[i]
```

损失函数设计

1. **组合损失函数**：结合标签平滑和 Focal Loss 的优点：

```
class CombinedLoss(nn.Module):
    def __init__(self, smooth=0.15, alpha=0.65, gamma=2):
        super(CombinedLoss, self).__init__()
        self.smooth_loss = LabelSmoothEntropy(smooth=smooth)
        self.focal_loss = FocalLoss(gamma=gamma)
        self.alpha = alpha

    def forward(self, preds, targets):
        return self.alpha * self.smooth_loss(preds, targets) + \
               (1 - self.alpha) * self.focal_loss(preds, targets)
```

2. **标签平滑和权重调整**：对空字符类别(10)给予更高权重：

```
def forward(self, preds, targets):
    # 为标签 10（空）提供更高的权重
    weights = t.ones(preds.shape[1], device=preds.device)
    weights[10] = 1.5  # 给空白类更高权重
```

3. Focal Loss：关注难样本，处理数据不平衡：

```
class FocalLoss(nn.Module):
    def __init__(self, alpha=1, gamma=2, reduction='mean'):
        # ...
    def forward(self, inputs, targets):
        ce_loss = F.cross_entropy(inputs, targets, reduction='none')
        pt = t.exp(-ce_loss)
        focal_loss = self.alpha * (1-pt)**self.gamma * ce_loss
```

4. **MixUp/CutMix 适应的损失计算**：

```
# 处理 MixUp/CutMix 的损失
if target_b is not None:
    loss = 0
    for j in range(4):
```

```
            loss += lam * self.criterion(pred[j], target_a[:, j]) + \
                    (1 - lam) * self.criterion(pred[j], target_b[:, j])
```

涨点工作

1. 数据增强

1. **CutMix/MixUp**：两种技术显著提高了模型泛化能力：
2. **自适应增强强度**：根据训练进度动态调整增强强度：

```
aug_intensity = min(1.0, 0.5 + self.epoch * 0.02)
```

3. **特定于字符识别的增强**：针对字符识别问题定制的增强策略，如：

```
transforms.RandomPerspective(distortion_scale=0.2 * aug_intensity, p=0.5),
transforms.RandomRotation(10 * aug_intensity),
# 保持小角度旋转以保留字符基本形状
```

2. 测试时增强（TTA）

实现了多种测试时增强策略，大幅提高推理准确率：

```
def tta_predict(self, img):
    # 原始预测
    pred_orig = self.model(img)
    preds = [p.clone() for p in pred_orig]

    # 水平翻转
    img_flip = t.flip(img, dims=[3])
    pred_flip = self.model(img_flip)

    # 亮度对比度变化
    brightness = transforms.ColorJitter(brightness=0.2)
    img_bright = brightness(img)

    # 随机裁剪缩放
    crop_resize = transforms.RandomResizedCrop(size=(128, 224), scale=(0.9, 1.0))

    # 高斯模糊或透视变换
    blur = transforms.GaussianBlur(kernel_size=3, sigma=(0.1, 0.5))

    # 平均所有预测结果
    return [p / 5.0 for p in preds]
```

3. 模型集成策略

通过训练多种不同架构的模型并结合它们的预测结果，获得更强大的预测能力：

```
def ensemble_predict(model_paths, csv_path):
    # 存储所有图像名称和预测 logits
    all_img_names = []
    all_logits = [[] for _ in range(4)]
```

```
    # 逐个模型预测
    for model_path in model_paths:
        # ...加载模型...
        # ...进行预测...
        # 将当前模型的预测添加到总预测中
        for pos in range(4):
            if len(all_logits[pos]) == 0:
                all_logits[pos] = current_logits[pos]
            else:
                all_logits[pos] += current_logits[pos]

    # 对所有模型预测取平均
    avg_logits = [logits / len(model_paths) for logits in all_logits]
```

4. 混合精度训练

使用 PyTorch 的 AMP (Automatic Mixed Precision)技术加速训练：

```
if config.mixed_precision:
    with t.amp.autocast('cuda'):
        pred = self.model(img)
        loss = self.calculate_loss(pred, label)

    self.scaler.scale(loss).backward()
    self.scaler.step(self.optimizer)
    self.scaler.update()
```

创新性

鉴于测试集有4万张图片，比训练集的3万张还要大，~~创新性地~~将验证集的一万张图片并入训练集以增大数据集规模，提高模型泛化能力。如此操作之后，在baseline上只需要改epoch为12，并稍微降低学习率为8e-4，便可以收获90.95%的准确率。

困难和解决

所尝试的前述各种复杂方案都太耗费性能，本机8GB显存只能支持同时运行1个训练。这严重拖慢了训练的进度。解决方案包括去云平台运行等，步骤也很繁琐，运行时长也没有明显的缩短。由此便很难“炼丹”式的调参。

按前述，简单地给baseline扩大数据集并稍降学习率，便可以提高两个百分点的准确率。预计再微调（fine-tune）各种参数，还可以在现在的90.95%的基础上更进一步，乃至到达92%。以及，由于随机数的影响，重新运行一次代码，同一批次的准确率也有所不同。固定随机数虽然结果可复现，也扼杀了更好结果出现的可能，因此没有指定种子。

由于不同批次验证集上的准确率高低，和最终交到网上得到的测试集准确率没有必然联系；所以还得多测几次。然而一天的提交次数为有限的5次，所以暂时只得到90.95%。