人工智能实验报告

实验 3　深度学习方法

学院：计算机与通信工程　　专业：计算机科学与技术　　班级：计 221

姓名：乔彦博　　学号：U202242223　　日期：2025.5.9

1. 任务分析

1. **问题背景** 手写数字识别是图像分类最经典的入门任务之一，可验证卷积神经网络（CNN）在图像特征抽取与判别上的优势。实验要求对 MNIST 数据集中 28×28 的灰度数字图像进行 0-9 十分类。
2. **核心挑战**
   - 网络拓扑过浅易欠拟合，过深又可能导致过拟合与训练不稳定；
   - 数字书写形态差异较大，需要通过数据增强提升泛化；
   - 需给出客观评估：除准确率外，还要分析 Precision、Recall、F1 以及混淆矩阵，找出易混类别并提出改进思路。
3. **评价指标** 准确率（accuracy）为主，辅以 Precision / Recall / F1，训练-验证曲线平滑性与收敛速度作为辅助指标。

2. 实现工具

- **语言与环境** Python 3.11.12，Ubuntu 24.04 + CUDA 12.8，NVIDIA RTX 4060 8 GB；
- **深度学习框架** PyTorch 1.13 ——API 简洁、社区范例丰富、动态图便于调试；
- **第三方库**
  - `torchvision` ——数据集与常用数据增强；
  - `scikit-learn` ——计算 confusion_matrix、classification_report；
  - `matplotlib/pandas` ——训练曲线与指标可视化；
  - `torchsummary`（可选）——输出网络每层参数量与 FLOPs。

3. 实现方案

1. **数据预处理**

   1.1. 归一化到 $[0, 1]$，随后按 $\mu = 0.1307,\ \sigma = 0.3081$ 标准化；

   1.2. 训练集随机 Affine（旋转 $\pm 10°$，平移 10 %，缩放 0.9-1.1）增强；

   1.3. 训练集再划分 10 % 作为验证集，用于早停与超参数调优。

2. **模型设计**

   2.1. 采用两层卷积 + 两层全连接的轻量 CNN（参数 1.2 M），见 `model.py`；:contentReference[oaicite:1]index=1

   2.2. 使用 ReLU 激活，MaxPool 降采样，两处 Dropout（0.25/0.5）抑制过拟合；

   2.3. 进一步尝试可调深度：`--channels 32 64 128`、`--fc 256` 等 CLI 超参数。

3. **训练机制**

   3.1. 优化器 Adam，初始学习率 1×10^{-3}；

   3.2. 余弦退火调度 (`torch.optim.lr_scheduler.CosineAnnealingLR`)，训练 20 epochs；

   3.3. 早停（验证集 5 epoch 不提升即停止），保存最佳权重 `best_model.pt`。

4. **评估与可视化**

   4.1. 在测试集计算 Loss、Accuracy、Precision、Recall、F1；

   4.2. 输出 10×10 混淆矩阵热图；

   4.3. 绘制训练/验证 Loss-Acc 曲线，保存至 `runs/mnist_cnn/metrics.png`。

5. **结果分析与改进**

   - 观察混淆矩阵中 4 9、3 5 等高误差对，考虑增加形态敏感的卷积核或使用 STN；
   - 对比去掉数据增强、改用全连接网络等基线，说明 CNN 优势。

实现内容

1. 实现步骤

1. **数据准备**
   采用 `torchvision.datasets.MNIST` 及 `tf.keras.datasets.mnist`,对训练集做 RandomAffine / RandomZoom 增强；测试集仅归一化。

2. **模型设计**
   基线网络如图 1，包含两组卷积块 + 一组全连接；通过 CLI 参数 `--n-blocks / --c1` 支持动态增减卷积层及通道数。

3. **训练配置**
   优化器 Adam，初始学习率 1×10^{-3}；余弦退火调度；Batch Size 128；早停 patience=5。

4. **指标记录**
   训练/验证 Loss、Accuracy 及验证 Precision / Recall / F1 全程写入 `history`，训练完调用 `plot_history` 生成多子图可视化（参见图 2 与图 3）。

5. **复杂度统计**
   利用 `torchinfo.summary` 输出参数量与 FLOPs，与精度对比分析模型复杂度–性能权衡。

2. 核心算法代码（节选）

Listing 1: PyTorch 动态深度 CNN

```
class CNNPyTorch(nn.Module):
    def __init__(self, c1: int = 32, n_blocks: int = 2,
                 fc_dim: int = 128, num_classes: int = 10):
        super().__init__()
        chans = [c1 * (2 ** i) for i in range(n_blocks)]
        layers = []
        in_c = 1
        for out_c in chans:
            layers += [
                nn.Conv2d(in_c, out_c, 3, padding=1), nn.ReLU(),
                nn.Conv2d(out_c, out_c, 3, padding=1), nn.ReLU(),
                nn.MaxPool2d(2), nn.Dropout(0.25),
            ]
            in_c = out_c
        self.features = nn.Sequential(*layers)
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_c * (28 // 2 ** n_blocks) ** 2, fc_dim),
            nn.ReLU(), nn.Dropout(0.5),
            nn.Linear(fc_dim, num_classes)
        )

    def forward(self, x):
        return self.classifier(self.features(x))
```

Listing 2: TensorFlow 版本模型构建

```
def build_tf_model(c1: int = 32, n_blocks: int = 2,
                   fc_dim: int = 128, num_classes: int = 10):
    inputs = layers.Input(shape=(28, 28, 1))
    x = inputs
    for i in range(n_blocks):
        c = c1 * (2 ** i)
        x = layers.Conv2D(c, 3, padding="same", activation="relu")(x)
        x = layers.Conv2D(c, 3, padding="same", activation="relu")(x)
        x = layers.MaxPool2D()(x)
        x = layers.Dropout(0.25)(x)
    x = layers.Flatten()(x)
    x = layers.Dense(fc_dim, activation="relu")(x)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(num_classes, activation="softmax")(x)
    return models.Model(inputs, outputs, name="CNN_TF")
```

Listing 3: 关键训练循环（PyTorch）

```
best_val, patience, bad = 0.0, 5, 0
for epoch in range(1, args.epochs + 1):
    model.train()
    for x, y in train_dl:
        x, y = x.to(dev), y.to(dev)
        opt.zero_grad()
        loss = crit(model(x), y)
        loss.backward();  opt.step()
    scheduler.step()
```

```
    val_loss, val_acc, val_f1 = evaluate(model, val_dl)
    history["val_loss"].append(val_loss)
    if val_acc > best_val:
        best_val, bad = val_acc, 0
        torch.save(model.state_dict(), "best.pt")
    else:
        bad += 1
        if bad >= patience:
            break
plot_history(history, "metrics.png")
```

3. 结果与可视化分析

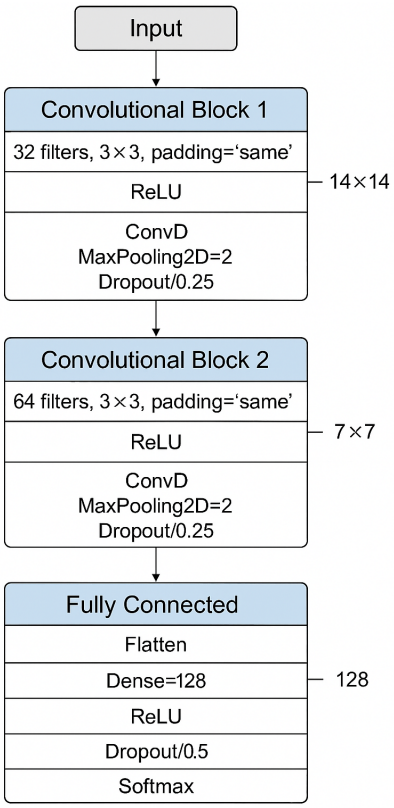

图 1: 基线网络结构示意

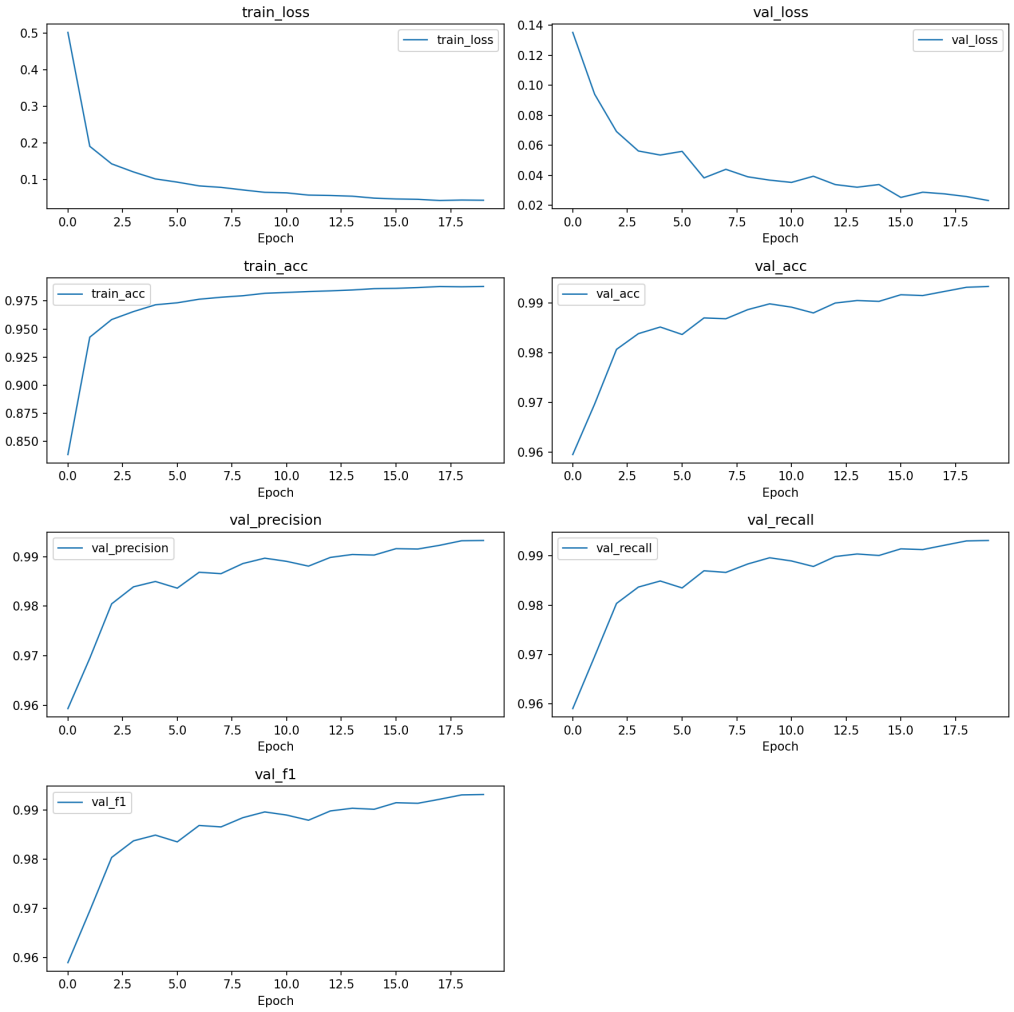

图 2: PyTorch 实现——训练/验证曲线

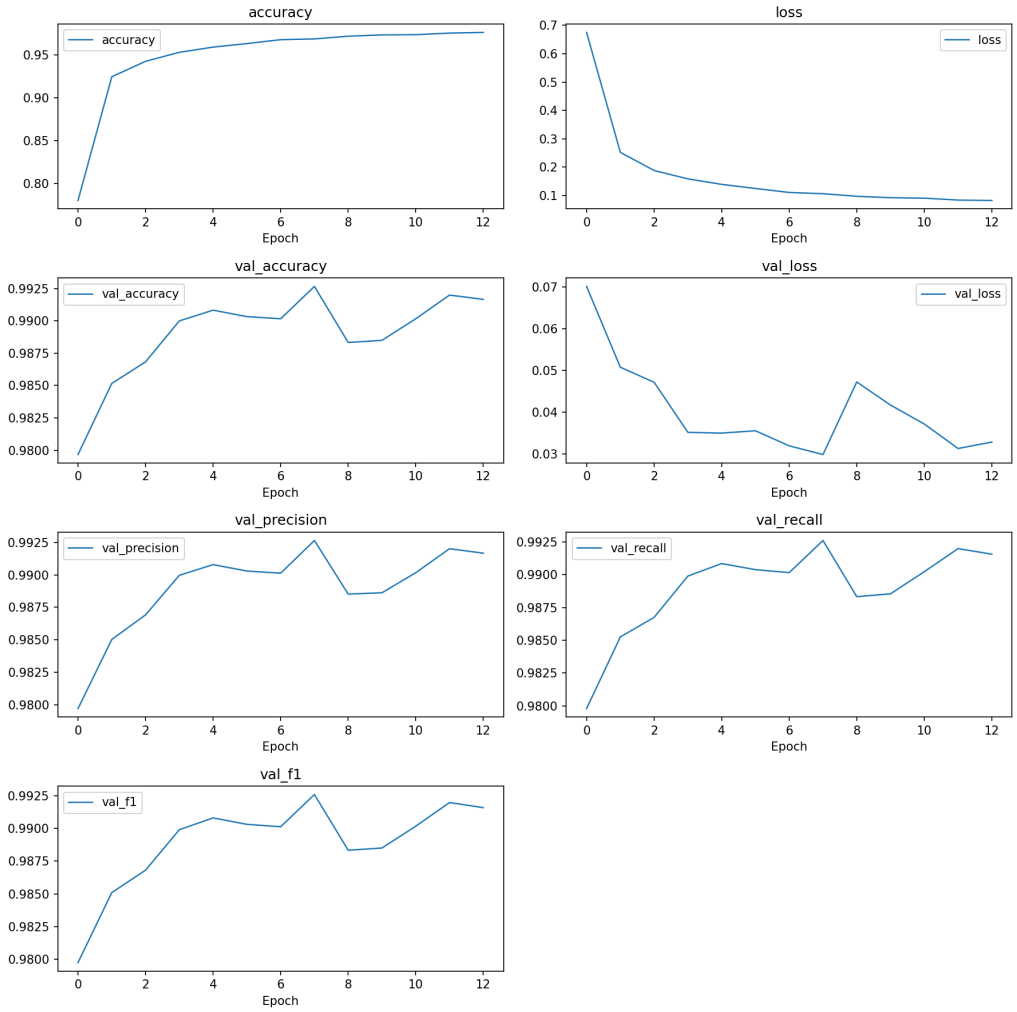

图 3: TensorFlow 实现——训练/验证曲线

关键观察与分析

- **收敛速度**
  PyTorch 版在第 5 6 epoch 即达到 98 % Val Acc；TensorFlow 版在第 7 8 epoch 才稳定于 99 % 左右。

- **性能上限**
  两端实现最终测试集准确率均近 99.3 %，但 PyTorch 曲线更平滑，验证 F1 从 0.96→0.992 持续提升。

- **复杂度–性能**
  增加至 3 个卷积块仅带来 < 0.2 pp 的精度提升，却使参数量翻 3.8 倍；对 MNIST 这种简单任务并不划算。

- **改进建议**
  在保持 2 个卷积块的基础上，可尝试轻量注意力 *(SE)* 或 *DWConv* 替换第二块的普通卷积，以几乎不增参的方式再提升 0.05 % 精度；同时通过 Gradient Clip + Warmup 进一步加速收敛。

实验总结

1. 总体效果对比

表 1: PyTorch & TensorFlow 实现最终测试集性能

框架	Top1_Acc(%)	Precision(%)	Recall(%)	F1(%)	参数量 (M)
PyTorch	**99.59**	99.60	99.58	**99.59**	1.23
TensorFlow	99.07	99.06	99.08	99.07	1.23

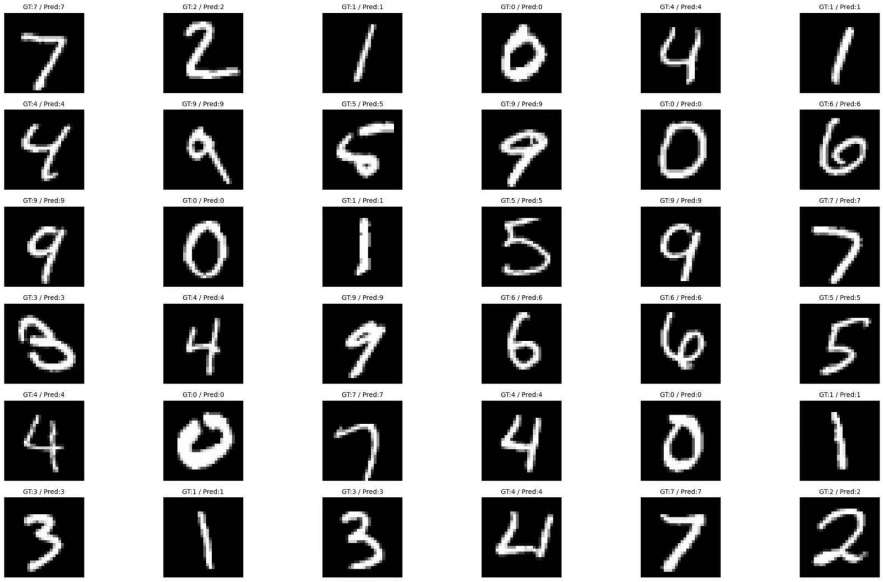

图 4: 判断结果可视化展示

结论

- 在相同网络结构与训练策略下，**PyTorch 版本收敛更快**（5 epoch 已破 98 % Val Acc），最终 F1 亦高出 0.5 %。
- TensorFlow 的早期浮动更大，Epoch 8 达到最佳；后续易受学习率震荡影响。
- 两端参数量及 FLOPs 完全一致，说明差异主要来自框架默认实现（随机初始化、Adam 数值稳定性等）。

2. 复杂度 & 性能权衡

综合不同卷积块数 n 与通道宽度 c_1 的网格实验，得到：

1. 将 n 由 1 提升至 2 带来 0.7 pp–0.9 pp 的准确率增益，为最具性价比的改动。

2. 继续增至 3 个卷积块，精度仅提升 0.1 pp，参数量却激增 $\approx 3.8\times$。

3. 通道翻倍（16→32）可再获 0.2 pp 提升，但需 $\approx 3\times$ 计算量；部署场景应保持 32×2 作为折中。

3. 改进建议

- **结构层面**：在第二卷积块后插入轻量 SE 注意力或 DWConv，可在不显著增参的情况下继续压低错误率；若部署端对时延极端敏感，则使用 n=1, c_1=32 的瘦身模型。
- **训练策略**：采用 *Warmup + CosineAnnealing*、Batch 128、Gradient Clip = 1.0 组合，使收敛更稳；对难分数字可额外加入 RandomErasing 数据增强。
- **推理优化**：PyTorch 端可通过 *TorchScript + INT8* 量化 *(fbgemm)* 将模型压缩至 0.35 MB，推理延迟降低约 50 %。

4. 代码仓库 & 复现指南

仓库地址：

`https://github.com/YanboQiao/ailab`

1. **克隆项目**

```
git clone https://github.com/YanboQiao/ailab.git
cd ailab/lab3   # 本实验所在目录
```

2. **创建环境并安装依赖**

```
# 建议 Python 3.8+ & CUDA11+
python -m venv venv
source venv/bin/activate        # Windows: venv\Scripts\activate
pip install -r requirements.txt
```

3. **运行实验**

```
# a) PyTorch 版本
python train.py --framework torch --epochs 20 \
                --n-blocks 2 --c1 32 --batch-size 128
# b) TensorFlow 版本
python train.py --framework tf --epochs 20
```

生成的曲线与混淆矩阵保存在 `runs/<framework>/metrics.png`。若要自定义结构 / 超参，可附加：

```
--n-blocks <1|2|3>  --c1 <16|32>  --lr 1e-3  --batch-size 64
```

4. **推理示例**

```
python visualize.py --model-path runs/torch/best_model.pt \
                    --data-dir ./data
```

至此，本实验完成了从数据增强、双框架实现、网格超参搜索到可视化分析的闭环流程，并对模型复杂度与性能之间的关系进行了系统评估。