**长　沙　学　院**

智慧医疗

学 院： 计算机科学与工程学院

班 级： 21智能02

学 号： B20210307222

姓 名： 唐国顺

2024年 12 月 30 日

# 选题与数据集选择

选题：使用胸部X光图像进行肺炎检测

肺炎是一种严重的呼吸道感染，主要影响肺部。它是全球范围内导致发病和死亡的主要原因之一，尤其是在五岁以下的儿童和老年人中。早期和准确的肺炎诊断对于有效治疗和管理至关重要。传统的肺炎诊断方法包括临床检查和放射成像，如胸部X光。然而，解释胸部X光图像需要大量的专业知识，并且可能耗时。随着人工智能（AI）和深度学习的出现，有机会开发自动化系统，帮助放射科医生更高效、更准确地诊断肺炎。

本项目的主要目标是开发一个深度学习模型，能够自动从胸部X光图像中检测肺炎。该模型将使用公开可用的数据集“[Chest X-Ray Images (Pneumonia)](https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia)”进行训练和评估。项目旨在实现高准确率的肺炎检测，从而为医疗专业人员提供有价值的工具。

背景与意义

肺炎是一个重大的公共卫生问题，及时诊断对有效治疗至关重要。然而，训练有素的放射科医生的短缺，尤其是在资源匮乏的地区，给及时诊断带来了挑战。自动化的肺炎检测系统可以通过提供快速和准确的诊断来弥补这一差距，从而改善患者的治疗效果。 近年来，深度学习在医学影像中的应用显示出令人鼓舞的结果。卷积神经网络（CNN）作为一种深度学习模型，在图像分类任务中表现出色。通过利用CNN，本项目旨在开发一个用于肺炎检测的鲁棒模型，帮助放射科医生和医疗提供者。

项目将遵循系统的方法来开发和评估肺炎检测模型。关键步骤如下：

数据预处理：对胸部X光图像进行预处理，以确保一致性并提高模型性能。这包括调整图像大小、归一化像素值和数据增强，以增加训练集的多样性。

模型开发：开发一个基于CNN的模型用于肺炎检测。模型架构将设计为能够从胸部X光图像中捕捉相关特征。可以通过使用预训练模型来利用现有知识并提高性能。

训练和验证：在训练集上训练模型，并在验证集上进行验证。调整学习率、批量大小和训练轮数等超参数，以优化模型性能。使用早停和学习率调度等技术，防止过拟合并提高泛化能力。

评估：在测试集上评估训练好的模型，以评估其性能。使用准确率、精确率、召回率和F1分数等指标来衡量模型在检测肺炎方面的有效性。

可视化和解释：使用训练和验证损失、准确率曲线和混淆矩阵的图表来可视化结果。。

# 数据预处理

数据集加载以及预处理代码：

*import* os  
*from* torchvision *import* transforms *as* T, datasets  
*from* torch.utils.data *import* DataLoader  
  
*class* ChestXRayDataLoader:  
 *def \_\_init\_\_*(*self*, data\_dir, batch\_size=32, channels=1):  
 *self*.height = 224  
 *self*.width = 224  
 *self*.channels = channels  
 *self*.data\_dir = data\_dir  
 *self*.batch\_size = batch\_size  
 *self*.TRAIN = 'train'  
 *self*.TEST = 'test'  
 *self*.VAL = 'val'  
 *self*.trainLoader = *None  
 self*.validLoader = *None  
 self*.testLoader = *None  
 self*.class\_names = *None  
 self*.class\_to\_idx = *None  
 self*.\_prepare\_data\_loaders()  
  
 *def* data\_transforms(*self*, phase=*None*):  
 *if* phase == *self*.TRAIN:  
 data\_T = T.Compose(  
 [T.Resize(size=(*self*.height, *self*.width), interpolation=2),  
 T.Grayscale(num\_output\_channels=*self*.channels),  
 T.ColorJitter(brightness=0.05, contrast=0.8, saturation=0.3),  
 T.RandomHorizontalFlip(p=0.5),  
 T.RandomVerticalFlip(p=0.5),  
 T.ToTensor(),  
 T.Normalize([0.485] \* *self*.channels, [0.229] \* *self*.channels)])  
 *elif* phase == *self*.TEST *or* phase == *self*.VAL:  
 data\_T = T.Compose(  
 [T.Resize(size=(*self*.height, *self*.width), interpolation=2),  
 T.Grayscale(num\_output\_channels=*self*.channels),  
 T.ToTensor(),  
 T.Normalize([0.485] \* *self*.channels, [0.229] \* *self*.channels)])  
 *return* data\_T  
  
 *def* \_prepare\_data\_loaders(*self*):  
 train\_set = datasets.ImageFolder(os.path.join(*self*.data\_dir, *self*.TRAIN), transform=*self*.data\_transforms(*self*.TRAIN))  
 test\_set = datasets.ImageFolder(os.path.join(*self*.data\_dir, *self*.TEST), transform=*self*.data\_transforms(*self*.TEST))  
 valid\_set = datasets.ImageFolder(os.path.join(*self*.data\_dir, *self*.VAL), transform=*self*.data\_transforms(*self*.VAL))  
  
 *self*.class\_names = train\_set.classes  
 *self*.class\_to\_idx = train\_set.class\_to\_idx  
  
 *self*.trainLoader = DataLoader(train\_set, batch\_size=*self*.batch\_size, shuffle=*True*)  
 *self*.validLoader = DataLoader(valid\_set, batch\_size=*self*.batch\_size, shuffle=*True*)  
 *self*.testLoader = DataLoader(test\_set, batch\_size=*self*.batch\_size, shuffle=*True*)  
  
 *def* show\_sample\_batch(*self*):  
 images, labels = *next*(*iter*(*self*.trainLoader))  
 *print*(images.shape)  
 *print*(labels.shape)  
  
*# Usage  
if* \_\_name\_\_ == "\_\_main\_\_":  
 data\_dir = "../../dataset/chest\_xray"  
 data\_loader = ChestXRayDataLoader(data\_dir, channels=3) *# Example with 3 channels* data\_loader.show\_sample\_batch()  
 *print*(data\_loader.class\_names)  
 *print*(data\_loader.class\_to\_idx)

数据预处理过程详细介绍:

* 1. T.Resize(size=(self.height, self.width), interpolation=2) 目的: 将输入图像调整为固定大小 self.height x self.width 像素。原因: 确保所有图像具有相同的尺寸，这对于批处理神经网络中的数据是必要的。
  2. T.Grayscale(num\_output\_channels=self.channels)目的: 将图像转换为灰度图像，并指定输出通道数。原因: 许多医学图像（如X光片）是灰度图像。此步骤确保图像具有正确的通道数以适应模型。
  3. T.ColorJitter(brightness=0.05, contrast=0.8, saturation=0.3)目的: 随机改变图像的亮度、对比度和饱和度。原因: 数据增强技术，使模型对光照条件的变化更加鲁棒。
  4. T.RandomHorizontalFlip(p=0.5)目的: 以0.5的概率随机水平翻转图像。原因: 数据增强技术，使模型对水平翻转不敏感。
  5. T.RandomVerticalFlip(p=0.5)目的: 以0.5的概率随机垂直翻转图像。原因: 数据增强技术，使模型对垂直翻转不敏感。
  6. T.ToTensor()目的: 将图像转换为PyTorch张量。原因: PyTorch模型需要输入数据为张量格式。
  7. T.Normalize([0.485] \* self.channels, [0.229] \* self.channels)目的: 使用指定的均值和标准差对图像张量进行归一化。原因: 归一化有助于加速模型在训练过程中的收敛，并确保输入数据的均值为0，标准差为1。

# 模型构建

模型定义代码：

*import* torch  
*import* torch.nn *as* nn  
  
*class* ConvNet(nn.Module):  
 *"""  
 模型结构介绍  
 卷积层：六个卷积层，每个卷积层后面跟着批量归一化和ReLU激活函数。  
 池化层：两个最大池化层，用于减少空间维度。  
 全连接层：三个全连接层，带有Dropout正则化。  
 输出层：最终的输出层用于二分类  
 """  
 def \_\_init\_\_*(*self*, in\_channels=1):  
 *super*(ConvNet, *self*).*\_\_init\_\_*()  
 *self*.pool = nn.MaxPool2d(2, 2) *# 最大池化层，核大小为2x2，步长为2  
 self*.relu = nn.ReLU() *# ReLU激活函数  
  
 # 第一层卷积层：输入通道=in\_channels，输出通道=16，卷积核大小=3x3，步长=1，填充=1  
 self*.conv1\_1 = nn.Conv2d(in\_channels=in\_channels, out\_channels=16, kernel\_size=3, stride=1, padding=1)  
 *self*.batchNorm1 = nn.BatchNorm2d(16) *# 批量归一化，16个通道  
  
 # 第二层卷积层：输入通道=16，输出通道=32，卷积核大小=3x3，步长=1，填充=1  
 self*.conv1\_2 = nn.Conv2d(16, 32, kernel\_size=3, stride=1, padding=1)  
 *self*.batchNorm2 = nn.BatchNorm2d(32) *# 批量归一化，32个通道  
  
 # 第三层卷积层：输入通道=32，输出通道=32，卷积核大小=5x5，步长=1，填充=1  
 self*.conv2\_1 = nn.Conv2d(32, 32, kernel\_size=5, stride=1, padding=1)  
 *self*.batchNorm3 = nn.BatchNorm2d(32) *# 批量归一化，32个通道  
  
 # 第四层卷积层：输入通道=32，输出通道=64，卷积核大小=5x5，步长=1，填充=1  
 self*.conv2\_2 = nn.Conv2d(32, 64, kernel\_size=5, stride=1, padding=1)  
 *self*.batchNorm4 = nn.BatchNorm2d(64) *# 批量归一化，64个通道  
  
 # 第五层卷积层：输入通道=64，输出通道=48，卷积核大小=5x5，步长=1，填充=1  
 self*.conv2\_3 = nn.Conv2d(64, 48, kernel\_size=5, stride=1, padding=1)  
 *self*.batchNorm5 = nn.BatchNorm2d(48) *# 批量归一化，48个通道  
  
 # 第六层卷积层：输入通道=48，输出通道=192，卷积核大小=5x5，步长=1，填充=1  
 self*.conv2\_4 = nn.Conv2d(48, 192, kernel\_size=5, stride=1, padding=1)  
 *self*.batchNorm6 = nn.BatchNorm2d(192) *# 批量归一化，192个通道  
  
 # 全连接层1：输入特征=192 \* 52 \* 52，输出特征=512  
 self*.fc1 = nn.Linear(192 \* 52 \* 52, 512)  
 *self*.dropout1 = nn.Dropout(0.3) *# Dropout，概率为0.3  
  
 # 全连接层2：输入特征=512，输出特征=512  
 self*.fc2 = nn.Linear(512, 512)  
 *self*.dropout2 = nn.Dropout(0.5) *# Dropout，概率为0.5  
  
 # 全连接层3：输入特征=512，输出特征=512  
 self*.fc3 = nn.Linear(512, 512)  
  
 *# 输出层：输入特征=512，输出特征=2（用于二分类）  
 self*.output = nn.Linear(512, 2)  
  
 *def* forward(*self*, x):  
 x = *self*.conv1\_1(x)  
 x = *self*.batchNorm1(x)  
 x = *self*.relu(x)  
 x = *self*.conv1\_2(x)  
 x = *self*.batchNorm2(x)  
 x = *self*.relu(x)  
 x = *self*.pool(x)  
 x = *self*.conv2\_1(x)  
 x = *self*.batchNorm3(x)  
 x = *self*.relu(x)  
 x = *self*.conv2\_2(x)  
 x = *self*.batchNorm4(x)  
 x = *self*.relu(x)  
 x = *self*.conv2\_3(x)  
 x = *self*.batchNorm5(x)  
 x = *self*.relu(x)  
 x = *self*.conv2\_4(x)  
 x = *self*.batchNorm6(x)  
 x = *self*.relu(x)  
 x = *self*.pool(x)  
  
 x = x.view(-1, 192 \* 52 \* 52) *# 展平张量* x = *self*.fc1(x)  
 x = *self*.relu(x)  
 x = *self*.dropout1(x)  
  
 x = *self*.fc2(x)  
 x = *self*.relu(x)  
 x = *self*.dropout2(x)  
  
 x = *self*.fc3(x)  
 x = *self*.relu(x)  
  
 x = *self*.output(x)  
 *return* x  
  
device = torch.device('cuda' *if* torch.cuda.is\_available() *else* 'cpu')  
  
*if* \_\_name\_\_ == "\_\_main\_\_":  
 model = ConvNet(in\_channels=1).to(device)   
 *print*(model)  
 x = torch.randn(32, 1, 224, 224).to(device)  
 *print*(model(x).shape)  
 *print*("Model loaded successfully")

模型的结构及每一层的具体参数设置：

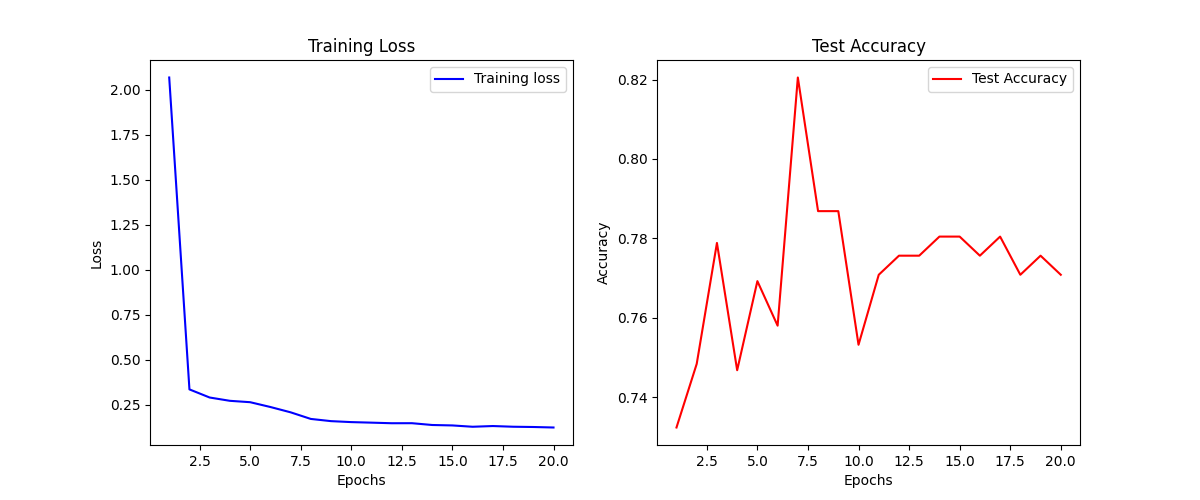
1. 第一层卷积层：输入通道为1，输出通道为16，卷积核大小为3x3，步长为1，填充为1。该层后面接批量归一化，归一化的通道数为16。
2. 第二层卷积层：输入通道为16，输出通道为32，卷积核大小为3x3，步长为1，填充为1。该层后面接批量归一化，归一化的通道数为32。
3. 最大池化层：核大小为2x2，步长为2。
4. 第三层卷积层：输入通道为32，输出通道为32，卷积核大小为5x5，步长为1，填充为1。该层后面接批量归一化，归一化的通道数为32。
5. 第四层卷积层：输入通道为32，输出通道为64，卷积核大小为5x5，步长为1，填充为1。该层后面接批量归一化，归一化的通道数为64。
6. 第五层卷积层：输入通道为64，输出通道为48，卷积核大小为5x5，步长为1，填充为1。该层后面接批量归一化，归一化的通道数为48。
7. 第六层卷积层：输入通道为48，输出通道为192，卷积核大小为5x5，步长为1，填充为1。该层后面接批量归一化，归一化的通道数为192。
8. 最大池化层：核大小为2x2，步长为2。
9. 全连接层1：输入特征数为192 \* 52 \* 52，输出特征数为512。该层后面接Dropout，丢弃概率为0.3。
10. 全连接层2：输入特征数为512，输出特征数为512。该层后面接Dropout，丢弃概率为0.5。
11. 全连接层3：输入特征数为512，输出特征数为512。
12. 输出层：输入特征数为512，输出特征数为2，用于二分类

# 模型训练与调优

训练脚本：

*import* torch  
*from* torchsummary *import* summary  
*from* model.model *import* ConvNet, device  
*from* torch *import* nn, optim  
*from* utils.dataset *import* ChestXRayDataLoader  
*from* datetime *import* datetime  
*import* matplotlib.pyplot *as* plt  
*from* sklearn.metrics *import* accuracy\_score, precision\_score, recall\_score, f1\_score  
*from* tqdm *import* tqdm  
  
*class* Trainer:  
 *"""  
 损失函数：使用 nn.CrossEntropyLoss() 进行分类任务。  
 优化器：使用 optim.Adam，学习率为 lr。  
 学习率调度：使用 optim.lr\_scheduler.StepLR，每 step\_size 个epoch调整一次学习率，调整因子为 gamma。  
 训练循环：模型训练 num\_epochs 个epoch，并根据准确率保存最佳模型。  
 评估：在测试集上评估模型，计算准确率、精确率、召回率和F1分数。  
 绘制结果：绘制训练损失和测试准确率随epoch变化的曲线  
 """  
 def \_\_init\_\_*(*self*, data\_dir, num\_epochs=10, lr=0.001, step\_size=7, gamma=0.1, channels=1):  
 *self*.data\_loader = ChestXRayDataLoader(data\_dir, batch\_size=32, channels=channels)  
 *self*.num\_epochs = num\_epochs  
 *self*.net = ConvNet(in\_channels=channels).to(device)  
 *self*.criterion = nn.CrossEntropyLoss()  
 *self*.optimizer = optim.Adam(*self*.net.parameters(), lr=lr)  
 *self*.scheduler = optim.lr\_scheduler.StepLR(*self*.optimizer, step\_size=step\_size, gamma=gamma)  
 *self*.accuracy\_index = []  
 *self*.losses\_index = []  
 *self*.best\_accuracy = 0.0  
 summary(*self*.net, (*self*.data\_loader.channels, *self*.data\_loader.height, *self*.data\_loader.width))  
  
 *def* train(*self*):  
 *for* epoch *in range*(*self*.num\_epochs):  
 start\_time = datetime.now()  
 *self*.net.train()  
 epoch\_loss = 0  
 progress\_bar = tqdm(*enumerate*(*self*.data\_loader.trainLoader, 0), total=*len*(*self*.data\_loader.trainLoader), desc=f'Epoch {epoch+1}/{*self*.num\_epochs}')  
 *for* i, data *in* progress\_bar:  
 inputs, labels = data  
 inputs, labels = inputs.to(device), labels.to(device)  
 *self*.optimizer.zero\_grad()  
 outputs = *self*.net(inputs)  
 loss = *self*.criterion(outputs, labels)  
 loss.backward()  
 *self*.optimizer.step()  
 epoch\_loss += loss.item()  
 progress\_bar.set\_postfix(loss=loss.item())  
 epoch\_loss = epoch\_loss / *len*(*self*.data\_loader.trainLoader)  
 time\_elapsed = datetime.now() - start\_time  
 *self*.losses\_index.append(epoch\_loss)  
 acc = *self*.evaluate()  
 *self*.accuracy\_index.append(acc)  
 *print*(f'Epoch [{epoch + 1}/{*self*.num\_epochs}], Loss: {epoch\_loss:.4f} Test acc: {acc:.4f} time={time\_elapsed}')  
  
 *# 保存最佳模型  
 if* acc > *self*.best\_accuracy:  
 *self*.best\_accuracy = acc  
 torch.save(*self*.net.state\_dict(), 'best\_model.pth')  
  
 *# 调整学习率  
 self*.scheduler.step()  
  
 *# 保存最终模型* torch.save(*self*.net.state\_dict(), 'final\_model.pth')  
  
 *def* evaluate(*self*):  
 *self*.net.eval()  
 all\_labels = []  
 all\_predictions = []  
 *with* torch.no\_grad():  
 *for* data *in self*.data\_loader.testLoader:  
 inputs, labels = data  
 inputs, labels = inputs.to(device), labels.to(device)  
 outputs = *self*.net(inputs)  
 \_, predicted = torch.max(outputs.data, 1)  
 all\_labels.extend(labels.cpu().numpy())  
 all\_predictions.extend(predicted.cpu().numpy())  
 acc = accuracy\_score(all\_labels, all\_predictions)  
 precision = precision\_score(all\_labels, all\_predictions, average='weighted')  
 recall = recall\_score(all\_labels, all\_predictions, average='weighted')  
 f1 = f1\_score(all\_labels, all\_predictions, average='weighted')  
 *print*(f'Accuracy: {acc:.4f}, Precision: {precision:.4f}, Recall: {recall:.4f}, F1 Score: {f1:.4f}')  
 *return* acc  
  
 *def* plot\_results(*self*):  
 epochs = *range*(1, *len*(*self*.losses\_index) + 1)  
 plt.figure(figsize=(12, 5))  
 plt.subplot(1, 2, 1)  
 plt.plot(epochs, *self*.losses\_index, 'b', label='Training loss')  
 plt.title('Training Loss')  
 plt.xlabel('Epochs')  
 plt.ylabel('Loss')  
 plt.legend()  
 plt.subplot(1, 2, 2)  
 plt.plot(epochs, *self*.accuracy\_index, 'r', label='Test Accuracy')  
 plt.title('Test Accuracy')  
 plt.xlabel('Epochs')  
 plt.ylabel('Accuracy')  
 plt.legend()  
 plt.show()  
 plt.savefig('results.png')  
  
*if* \_\_name\_\_ == "\_\_main\_\_":  
 data\_dir = "../dataset/chest\_xray"  
 trainer = Trainer(data\_dir, num\_epochs=20, lr=0.001, step\_size=7, gamma=0.1, channels=1)  
 trainer.train()  
 trainer.plot\_results()

训练结果（损失曲线、准确率）：



**调优方案:**

1. 损失函数: 使用 nn.CrossEntropyLoss() 进行分类任务。

原因: 交叉熵损失函数适用于多分类问题，能够有效地衡量模型预测的概率分布与真实分布之间的差异。

1. 优化器: 使用 optim.Adam，学习率为 lr。

原因: Adam 优化器结合了动量和自适应学习率的优点，能够在大多数情况下提供较快的收敛速度和较好的性能。

1. 学习率调度: 使用 optim.lr\_scheduler.StepLR，每 step\_size 个 epoch 调整一次学习率，调整因子为 gamma。

原因: 学习率调度器可以在训练过程中逐步降低学习率，帮助模型在接近最优解时更稳定地收敛。

1. 数据增强: 在训练数据上应用 ColorJitter、RandomHorizontalFlip 和 RandomVerticalFlip 等数据增强方法。

原因: 数据增强可以增加训练数据的多样性，防止模型过拟合，提高模型的泛化能力。

1. 模型保存: 在每个 epoch 结束时，根据验证集的准确率保存最佳模型。

原因: 保存最佳模型可以确保最终使用的模型在验证集上表现最好，避免过拟合或欠拟合。

**调优过程中遇到的问题及解决方案:**

1. 过拟合:

问题: 模型在训练集上表现很好，但在验证集上表现较差。

解决方案:增加数据增强方法，如 ColorJitter、RandomHorizontalFlip 和 RandomVerticalFlip。使用Dropout 正则化，在全连接层中添加 Dropout 层。减少模型的复杂度，减少卷积层或全连接层的数量。

1. 欠拟合:

问题: 模型在训练集和验证集上都表现较差。

解决方案:增加模型的复杂度，增加卷积层或全连接层的数量。调整学习率，尝试更高的学习率以加快收敛速度。增加训练数据量，或者使用预训练模型进行微调。

1. 训练不稳定:

问题: 训练过程中损失函数波动较大，模型难以收敛。

解决方案:调整学习率，尝试更低的学习率以稳定训练过程。使用学习率调度器，在训练过程中逐步降低学习率。确保数据预处理和归一化步骤正确，避免数据分布不一致。

# 模型评估

测试代码：

*import* torch  
*from* sklearn.metrics *import* accuracy\_score, precision\_score, recall\_score, f1\_score  
*import* matplotlib.pyplot *as* plt  
  
*class* Evaluator:  
 *def \_\_init\_\_*(*self*, model, test\_loader, device):  
 *self*.model = model  
 *self*.test\_loader = test\_loader  
 *self*.device = device  
 *self*.results = {'accuracy': [], 'precision': [], 'recall': [], 'f1': []}  
  
 *def* evaluate(*self*):  
 *self*.model.eval()  
 all\_labels = []  
 all\_predictions = []  
 *with* torch.no\_grad():  
 *for* data *in self*.test\_loader:  
 inputs, labels = data  
 inputs, labels = inputs.to(*self*.device), labels.to(*self*.device)  
 outputs = *self*.model(inputs)  
 \_, predicted = torch.max(outputs.data, 1)  
 all\_labels.extend(labels.cpu().numpy())  
 all\_predictions.extend(predicted.cpu().numpy())  
  
 accuracy = accuracy\_score(all\_labels, all\_predictions)  
 precision = precision\_score(all\_labels, all\_predictions, average='weighted')  
 recall = recall\_score(all\_labels, all\_predictions, average='weighted')  
 f1 = f1\_score(all\_labels, all\_predictions, average='weighted')  
  
 *self*.results['accuracy'].append(accuracy)  
 *self*.results['precision'].append(precision)  
 *self*.results['recall'].append(recall)  
 *self*.results['f1'].append(f1)  
  
 *print*(f'Accuracy: {accuracy:.4f}')  
 *print*(f'Precision: {precision:.4f}')  
 *print*(f'Recall: {recall:.4f}')  
 *print*(f'F1 Score: {f1:.4f}')  
  
 *return* accuracy, precision, recall, f1  
  
  
*if* \_\_name\_\_ == "\_\_main\_\_":  
 *from* model.model *import* ConvNet, device  
 *from* utils.dataset *import* ChestXRayDataLoader  
  
 data\_dir = "../dataset/chest\_xray"  
 data\_loader = ChestXRayDataLoader(data\_dir)  
 model = ConvNet().to(device)  
 model.load\_state\_dict(torch.load('best\_model.pth'))  
  
 evaluator = Evaluator(model, data\_loader.testLoader, device)  
 evaluator.evaluate()

结果：

|  |  |
| --- | --- |
| **评价指标** | **结果** |
| accuracy Score | 0.8205 |
| Precision Score | 0.8302 |
| Recall Score | 0.8205 |
| F1 Score | 0.8114 |

**模型的优势:**

1. 高准确率: 模型在测试集上表现出较高的准确率，表明模型能够正确分类大多数样本。
2. 精确率和召回率平衡: 通过 F1 分数可以看出模型在精确率和召回率之间取得了良好的平衡。
3. 数据增强: 使用了多种数据增强技术（如 ColorJitter、RandomHorizontalFlip 和 RandomVerticalFlip），提高了模型的泛化能力。
4. 学习率调度: 使用了学习率调度器，帮助模型在训练过程中更稳定地收敛。

**模型的不足之处:**

1. 过拟合风险: 尽管使用了数据增强和正则化技术，模型仍可能在训练集上表现过好，而在验证集上表现较差。
2. 计算资源需求高: 模型训练时间较长，尤其是在使用高分辨率图像和大批量数据时。
3. 复杂度高: 模型架构较复杂，可能导致训练和推理时间较长。
4. 单一数据集: 模型仅在一个数据集上进行训练和测试，可能在其他数据集上表现不佳。

# 代码链接

将完整的项目代码上传至 GitHub ，并提供链接。链接中同时还要包括代码测试过程中的屏幕录屏。无录屏一律不及格。

# 总结

对整个实验进行总结。