

华中科技大学

课程实验报告

课程名称： 计算机视觉

实验名称： 实验一

院 系： 计算机科学与技术学院

专业班级： 本硕博 2301 班

学 号： U202315763

姓 名： 王家乐

2025 年 10 月 11 日

1. 任务要求

设计一个前馈神经网络，对一组数据实现分类任务。下载“dataset.csv”数据集，其中包含四类二维高斯数据和它们的标签。设计至少含有一层隐藏层的前馈神经网络来预测二维高斯样本($data_1, data_2$)所属的分类 $label$ 。对数据集进行随机排序，然后选取 90% 用于训练，剩下的 10% 用于测试。

2. 实验设计

2.1 实验环境

项	参数
操作系统	Windows11
CPU	12th Gen Intel(R) Core(TM) i7-12700H
RAM	16GB
编程语言	Python3.10.18
平台	VScode & Jupyter Notebook

2.2 相关依赖

依赖项	版本
torch	2.8.0
pandas	2.3.1
numpy	2.2.6
matplotlib	3.10.5
scikit-learn	1.7.2
tqdm	4.67.1

2.3 数据集加载和处理

观察 dataset.csv 基本结构，一行表示一条数据，其中前两列为高斯样本($data_1, data_2$)，最后一列为所属的分类 $label$ 。根据该结构，进而构建 Pytorch 框架所需的数据集（Dataset）类。

```

class MyDataset(Dataset):
    """自定义数据集类"""
    def __init__(self, csv_file):
        self.data = pd.read_csv(csv_file, header=None)
        # 随机打乱数据
        self.data = self.data.sample(frac=1, random_state=42).reset_index(drop=True)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        features = self.data.iloc[idx, :-1].values.astype(np.float32)
        label = self.data.iloc[idx, -1].astype(np.int64)
        label = label - 1
        return torch.tensor(features), torch.tensor(label)

```

接下来创建完整的数据集对象并划分训练集与测试集。

```

dataset = MyDataset("dataset.csv")
train_size = int(0.9 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = random_split(dataset, [train_size, test_size])

```

对数据集的分布进行大致观察，使用 matplotlib 库绘制数据分布的散点。

```

def plot_dataset(dataset):
    """可视化数据集"""
    features = dataset.data.iloc[:, :-1].values
    labels = dataset.data.iloc[:, -1].values

    label_names = ['Class 1', 'Class 2', 'Class 3', 'Class 4']
    plt.figure(figsize=(10, 6))

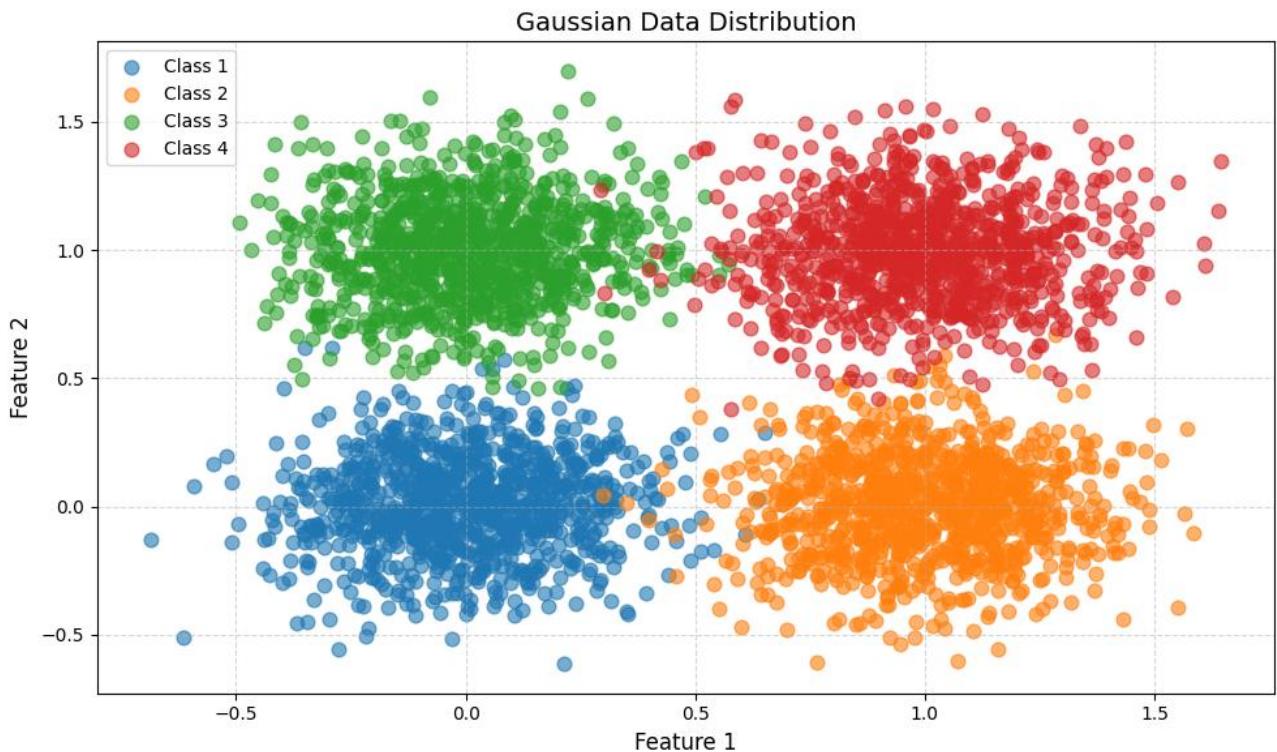
    for i, color in enumerate(label_names):
        class_features = features[labels == i + 1]
        plt.scatter(class_features[:, 0], class_features[:, 1],
                    label=label_names[i], alpha=0.6, s=60)

    plt.xlabel('Feature 1', fontsize=12)
    plt.ylabel('Feature 2', fontsize=12)
    plt.title('Gaussian Data Distribution', fontsize=14)
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.legend()
    plt.tight_layout()
    plt.show()

plot_dataset(dataset)

```

数据分布的散点图如下所示，没有明显的异常值和数据偏斜现象：



2.4 前馈神经网络设计

前馈神经网络是由若干全连接层组成的神经网络，每一层的输入维数必须与上一层的输出维数相等，层与层之间需要合适的激活函数进行非线性变换。本实验输入层的维数应与样本特征维数相等，输出层的维数应与分类数目相等。基本的网络只有一个大小为 64 的隐藏层，使用 ReLU 激活函数，不添加 dropout 层。

```
class FeedForwardNN(nn.Module):
    """前馈神经网络"""
    def __init__(
        self,
        input_size=2,
        hidden_sizes=[64],
        output_size=4,
        activation='relu',
        dropout_rate=0.0
    ):
        super(FeedForwardNN, self).__init__()
        # 构建网络
        layers = []
```

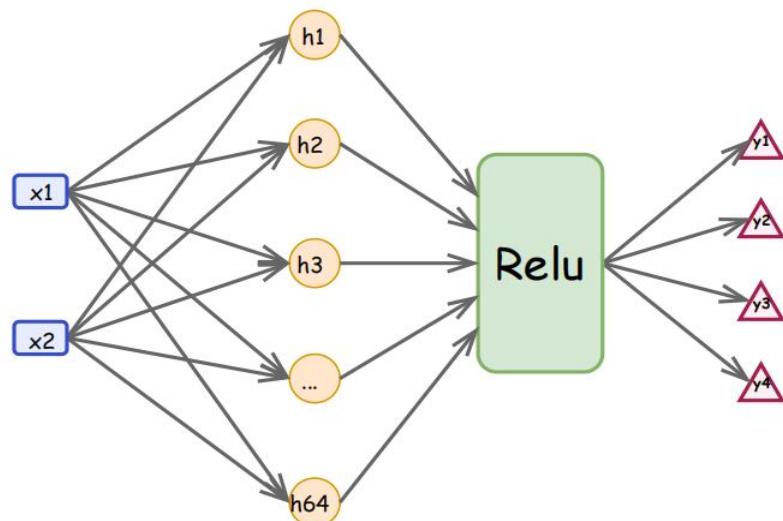
```

prev_size = input_size
for hidden_size in hidden_sizes:
    layers.append(nn.Linear(prev_size, hidden_size))
    # 激活函数
    if activation == 'relu':
        layers.append(nn.ReLU())
    elif activation == 'sigmoid':
        layers.append(nn.Sigmoid())
    elif activation == 'tanh':
        layers.append(nn.Tanh())
    elif activation == 'leaky_relu':
        layers.append(nn.LeakyReLU())
    else:
        layers.append(nn.ReLU())
    # 添加dropout
    if dropout_rate > 0.0:
        layers.append(nn.Dropout(dropout_rate))
    prev_size = hidden_size
layers.append(nn.Linear(prev_size, output_size))
self.network = nn.Sequential(*layers)
# 保存网络参数
self.hidden_sizes = hidden_sizes
self.activation = activation

def forward(self, x):
    return self.network(x)

```

本实验基本的神经网络结构如下图所示：



2.5 网络训练

首先根据配置信息（hidden_sizes、activation、dropout_rate）创建模型。

```
def create_model(model_config):
    """根据配置创建模型"""
    return FeedForwardNN(
        input_size=2,
        hidden_sizes=model_config.get('hidden_sizes', [64]),
        output_size=4,
        activation=model_config.get('activation', 'relu'),
        dropout_rate=model_config.get('dropout_rate', 0.0)
    )
```

接下来训练模型，最大迭代次数(num_epochs)默认为 100，模型名称(name)、批大小(batch_size)、和学习率(learning_rate)从配置信息 config 中获取。使用 DataLoader 创建数据对象，本实验选用交叉熵(Cross Entropy)作为损失函数，选用随机梯度下降(Stochastic Gradient Descent, SGD)作为优化器。训练过程中记录每个 epoch 在训练集和测试集上的损失和准确率，并将其写入 output 文件夹下的同名 log 文件中。

```
def train_model(model, config, num_epochs=100):
    """训练模型"""
    # 配置信息
    name = config['name']
    batch_size = config.get('batch_size', 32)
    learning_rate = config.get('learning_rate', 0.001)
    # 创建日志文件夹和文件
    log_dir = "output"
    os.makedirs(log_dir, exist_ok=True)
    log_file = os.path.join(log_dir, f"{name}.log")
    with open(log_file, "w") as log:
        log.write("Epoch,Train Loss,Train Accuracy,Test Loss,Test Accuracy\n")
    # 数据加载器
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
    # 定义损失函数和优化器
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)
    # 记录训练过程
```

```

train_losses = []
test_losses = []
train_accuracies = []
test_accuracies = []
# 训练循环
for epoch in tqdm(range(num_epochs), desc=f"Training {config['name']}:{<12}"):
    # 训练模式
    model.train()
    epoch_train_loss = 0.0
    train_preds = []
    train_labels = []
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        epoch_train_loss += loss.item()
    # 收集预测结果
    _, predicted = torch.max(output.data, 1)
    train_preds.extend(predicted.cpu().numpy())
    train_labels.extend(target.cpu().numpy())
# 计算训练损失和准确率
train_loss = epoch_train_loss / len(train_loader)
train_losses.append(train_loss)
train_acc = accuracy_score(train_labels, train_preds)
train_accuracies.append(train_acc)
# 测试模式
model.eval()
epoch_test_loss = 0.0
test_preds = []
test_labels = []
with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        loss = criterion(output, target)
        epoch_test_loss += loss.item()
        _, predicted = torch.max(output.data, 1)
        test_preds.extend(predicted.cpu().numpy())
        test_labels.extend(target.cpu().numpy())
# 计算测试损失和准确率
test_loss = epoch_test_loss / len(test_loader)
test_losses.append(test_loss)
test_acc = accuracy_score(test_labels, test_preds)
test_accuracies.append(test_acc)

```

```

# 写入日志文件
with open(log_file, "a") as log:
    log.write(f"{epoch+1},{train_loss:.4f},{train_acc:.4f},{test_loss:.4f},
{test_acc:.4f}\n")

# 最终评估
final_train_accuracy = train_accuracies[-1]
final_test_accuracy = test_accuracies[-1]

return {
    'model': model,
    'train_losses': train_losses,
    'test_losses': test_losses,
    'train_accuracies': train_accuracies,
    'test_accuracies': test_accuracies,
    'final_train_accuracy': final_train_accuracy,
    'final_test_accuracy': final_test_accuracy,
    'config': config
}

```

此外，为了更为直观地反映网络训练全过程的性能变化和分类效果，在训练结束后通过 matplotlib 库对训练集和测试集的损失、准确率进行可视化展示。

```

def plot_results(results):
    """绘制训练结果"""
    plt.figure(figsize=(20, 5))

    # 训练损失曲线
    ax1 = plt.subplot(141)
    for model_key, result in results.items():
        ax1.plot(result['train_losses'], label=f'{result["config"]["name"]}')
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Loss')
    ax1.set_title('Training Loss')
    ax1.legend()
    ax1.grid(True, linestyle='--', alpha=0.5)

    # 训练准确率曲线
    ax2 = plt.subplot(142)
    for model_key, result in results.items():
        ax2.plot(result['train_accuracies'], label=f'{result["config"]["name"]}')
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('Accuracy')
    ax2.set_title('Training Accuracy')
    ax2.legend()
    ax2.grid(True, linestyle='--', alpha=0.5)

```

```

# 测试损失曲线
ax3 = plt.subplot(143)
for model_key, result in results.items():
    ax3.plot(result['test_losses'], linestyle='--',
              label=f"{result['config']['name']}")

ax3.set_xlabel('Epoch')
ax3.set_ylabel('Loss')
ax3.set_title('Test Loss')
ax3.legend()
ax3.grid(True, linestyle='--', alpha=0.5)

# 测试准确率曲线
ax4 = plt.subplot(144)
for model_key, result in results.items():
    ax4.plot(result['test_accuracies'], linestyle='--',
              label=f"{result['config']['name']}")

ax4.set_xlabel('Epoch')
ax4.set_ylabel('Accuracy')
ax4.set_title('Test Accuracy')
ax4.legend()
ax4.grid(True, linestyle='--', alpha=0.5)

plt.tight_layout()
plt.show()

```

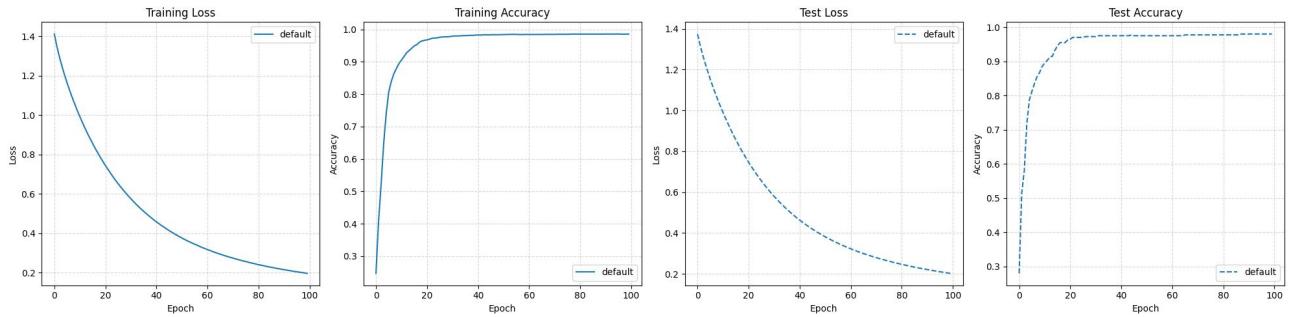
3. 实验结果与分析

3.1 基本网络结构

如 2.4 节所述, 基本网络结构为 $2(\text{input_size}) * 64(\text{hidden_size}) * 4(\text{output_size})$, 采用 ReLu 激活函数, 不添加 dropout 层, 训练条件如下表所示:

参数	值
num_epochs	100
batch_size	32
learning_rate	0.001

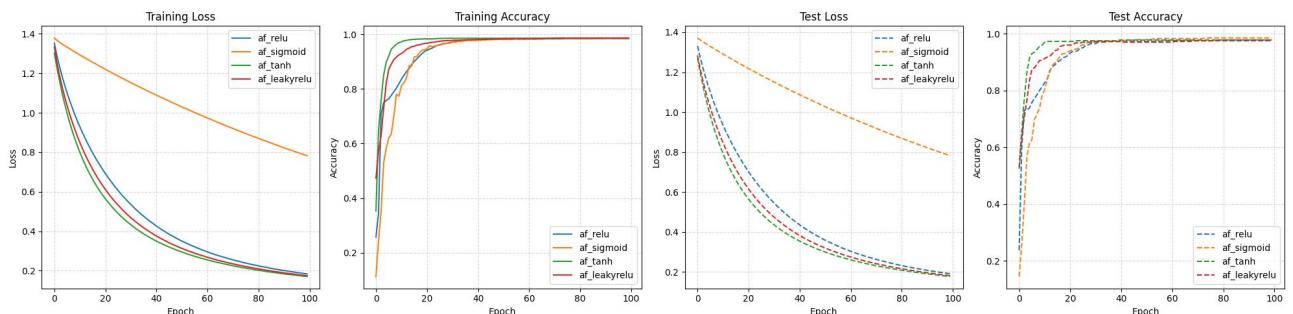
训练过程使用 tqdm 库通过进度条在控制台实时显示各项指标, 以 epoch 为单位更新。训练结束后, 得到训练过程的模型性能变化曲线如下图:



从测试结果可以看出，在基本的网络结构和训练条件下，模型的训练过程达到稳定且最终能收敛到较优的结果上（训练集准确率 98.56%，测试集准确率 98.00%），说明此前实验各模块的设计与实现正确，顺利地达成了实验目的。接下来依次改变网络结构和训练条件，对比性能。

3.2 不同激活函数

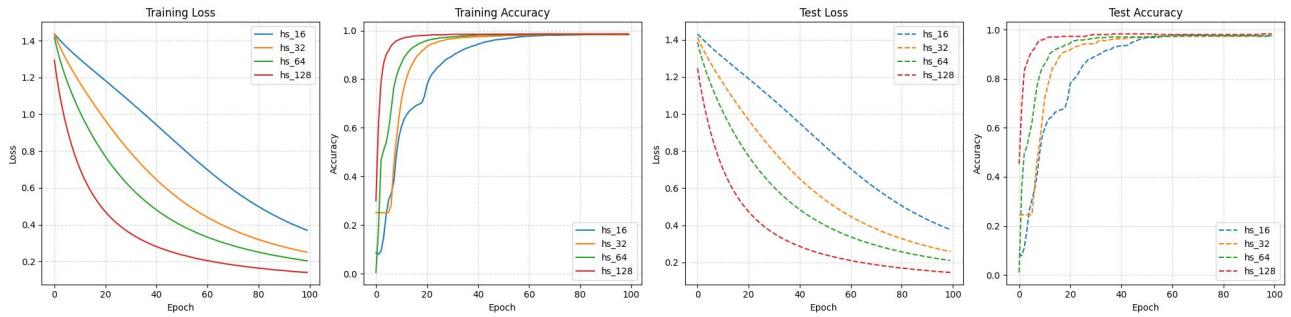
使用包含不同激活函数的网络进行训练，各网络的其他网络结构和训练条件全部相同。对于包含 ReLU（基本）、Sigmoid、Tanh、LeakyReLU 的网络，训练过程中的性能变化和分类效果如下图所示：



实验结果表明，在使用不同激活函数的网络中，ReLU、Tanh 和 LeakyReLU 激活函数在训练和测试过程中都表现出更快的收敛速度和更好的性能，训练损失和测试损失都迅速下降，训练准确率和测试准确率快速上升并稳定在较高水平；相比之下，Sigmoid 激活函数的收敛速度较慢，损失值较高，表明它在这种网络结构中可能由于梯度消失或饱和问题而性能较差。

3.3 不同隐藏层大小

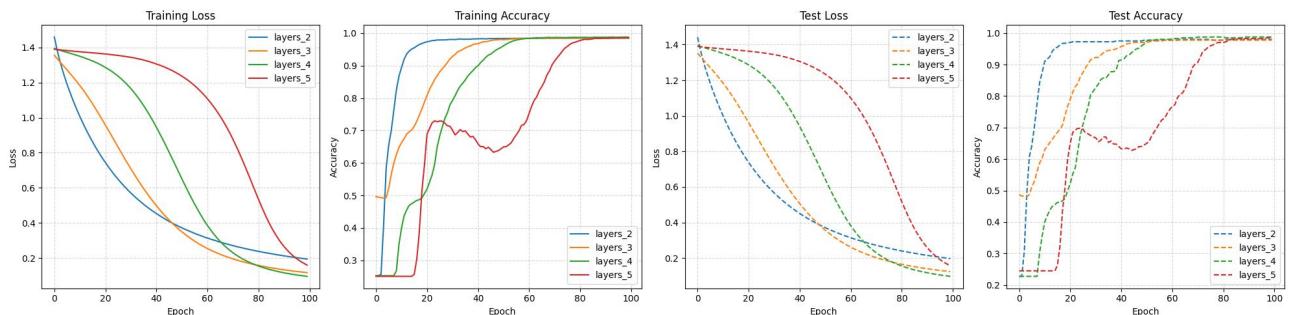
保持基本网络结构和训练条件不变，仅改变隐藏层大小，对隐藏层大小为 16、32、64（基本）、128 的网络，训练过程中的性能变化和分类效果如下图所示：



实验结果表明，随着隐藏层维数的增加，网络的收敛速度和分类效果逐渐提升。这表明隐藏层维数的增加有利于网络参数的学习，较大的隐藏层维数能有效增强网络的性能和分类效果；但在维数达到一定大小时，网络性能将收敛并不再随维度的增加显著变化。维数越大，模型参数越多，一味增大隐藏层维数，反而会徒增模型复杂程度，甚至造成网络收敛较慢、性能变差等不良后果。

3.4 不同网络层数

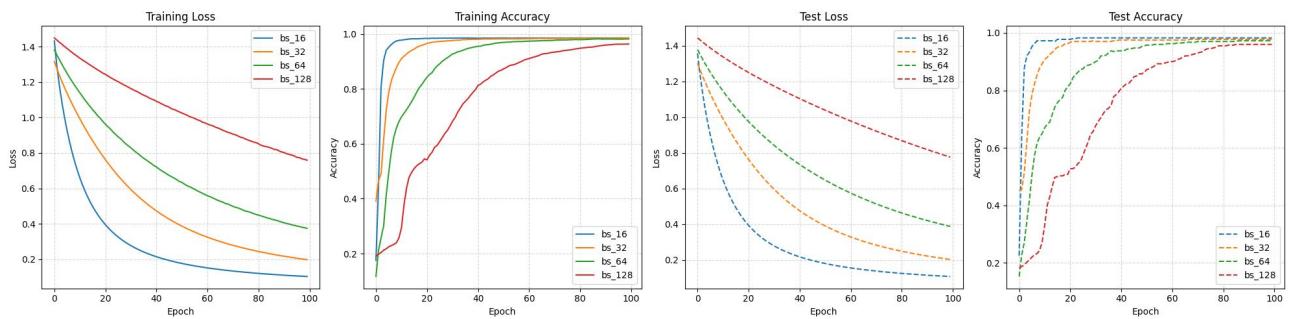
使用包含不同数目的全连接层的网络进行训练，其中，各网络在除输出层外的所有层后都使用了 ReLU 激活函数进行变换，且其他网络结构和训练条件均保持一致。对于 2 层全连接层（基本）、3 层全连接层、4 层全连接层和 5 层全连接层网络，训练过程中的性能变化和分类效果如下图所示：



实验结果表明，随着网络层数的增加，网络的收敛速度和分类效果逐渐下降，对于 5 层全连接层网络，在训练开始的一段时间内训练准确率没有发生任何提升。这说明了对于如本实验的简单任务，使用更为简单的网络结构有利于网络的收敛，在简化程序设计、缩短训练用时的同时，又能大幅增强网络性能。

3.5 不同批大小

使用不同批大小的数据加载器对相同结构的网络进行训练，其他训练参数保持一致。对于采用 16、32（基本）、64 和 128 批大小的数据加载器的网络，训练过程中的性能变化和分类效果如下图所示：

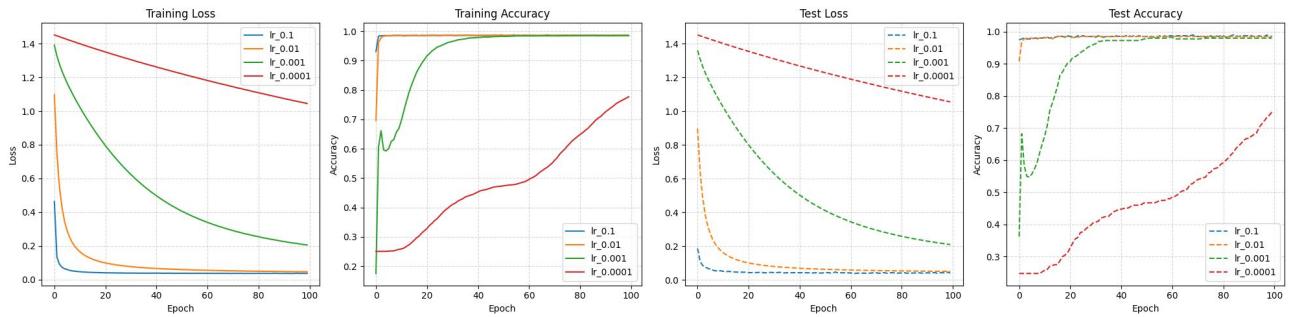


实验结果表明，在本任务中，批大小越小的网络收敛速度越快。这是由于本任务的数据分布较为均匀，批大小较小时每一个 epoch 中网络参数的更新频率更大，因此能够更快地收敛。而在更为复杂的任务中，批大小较小往往会使模型陷入局部最优，或者导致模型性能变化不稳定，所以一般使用较大的批大小来避免不利现象的发生。

3.6 不同学习率

学习率是控制神经网络在每次参数更新时调整步长大小的超参数，它决定了模型收敛的速度和最终性能。使用不同学习率的优化器对相同结构的网络进行训练，其他训练参数保持一致。对于采用学习率为 0.1、0.01、0.001（基本）和 0.0001

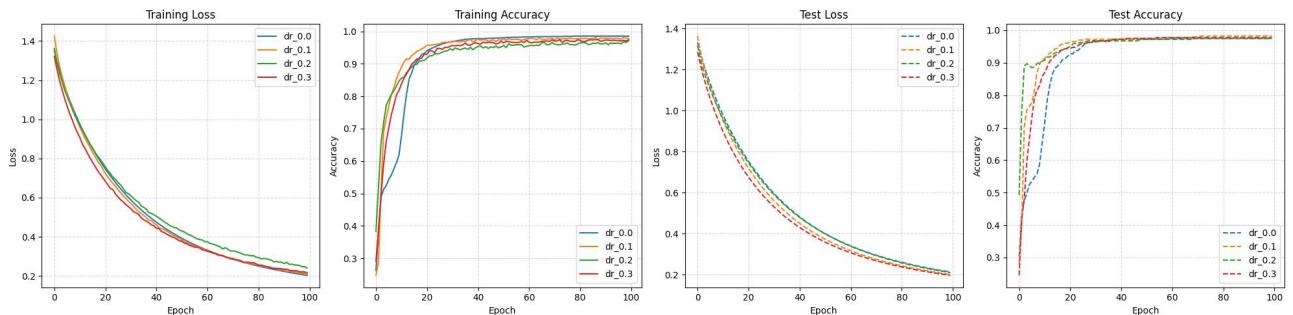
优化器的网络，训练过程中的性能变化和分类效果如下图所示：



实验结果表明，随着学习率的增加，网络的收敛速度逐渐提升。这是由于本任务较为简单，数据分布较为均匀，批大小较为均衡，从而无需使用细粒度较小（学习率较小）的训练方式就能取得较好的训练效果。然而，在某些大型机器学习或深度学习任务中，往往需要更小的学习率来调控网络的学习，一方面可以避免数据特征不显著时网络性能的大幅波动，另一方面能尽可能减缓网络的过拟合程度，从而最终达到均衡的训练效果。

3.7 不同随机失活（dropout）率

dropout 层是在训练时随机丢弃一部分神经元，防止神经网络过度依赖某些特定节点，从而减少过拟合。本实验设计随机失活率分别为 0.0（基本）、0.1、0.2、0.3 的 dropout 层的网络，训练过程中的性能变化和分类效果如下图所示：



实验结果表明，准确率在 dropout 率为 0.1 和 0.2 时明显高于无 dropout 的情况，说明适度的 dropout 有效缓解了过拟合。但当 dropout 率增至 0.3 时，性能开始下降，说明过高的丢弃率可能导致模型欠拟合。

4. 总结与体会

4.1 总结

此次实验成功设计并实现了一个包含隐藏层的前馈神经网络，完成了对二维高斯数据的四分类任务。在实验过程中，我系统探究了不同网络结构（包括激活函数类型、隐藏层维度、网络深度）和训练参数（如批大小、学习率、dropout 率）对模型性能的影响。实验结果表明：ReLU、Tanh 等激活函数在本任务中表现优异；适中的网络复杂度（如单隐藏层 64 维）既能保证性能又可避免过拟合；较小的批大小（16）和合适的学习率（0.001）有助于提升收敛效率；而适度的 dropout（0.1-0.2）能有效增强模型泛化能力。最终模型在测试集上取得了 98.00% 以上的准确率，验证了前馈神经网络处理此类分类问题的有效性。

4.2 体会

我深刻认识到，神经网络的设计需要充分考虑任务特性与模型复杂度之间的平衡。并非网络越深、参数越多效果就越好，相反，适合任务需求的简洁结构往往能取得更优效果。通过系统性的参数对比实验，我增强了调参的经验与直觉，同时也认识到可视化分析对理解模型训练过程的重要辅助作用。此外，在实验过程中，我进一步巩固了 PyTorch 框架的使用能力，提升了代码实现与实验设计的综合实践水平。