深度学习实践实验四

PB22151796 - 莫环欣

实验代码拆分与包装

这里先展示各部分的包装（不含`main`），后续也会再涉及，可直接从上面目录去看后文

训练器

将`test4.py`中与训练有关的代码包装到单独文件中，并新增了多个问题的控制开关

```python
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


def train_model(
    epochs=10,
    batch_size=64,
    learning_rate=0.001,
    momentum=0.9,
    test_2=False,
    test_3=False,
    test_3_1=False,
    test_4=False,
    test_8=False,
    model=Net(),
):
    trainloader, _, _ = get_dataloaders(
        batch_size=batch_size, test_2=test_2, test_8=test_8
    )
    net = model.to(device)
    criterion = nn.CrossEntropyLoss()
    if test_3_1:
        optimizer = optim.AdamW(net.parameters(), lr=learning_rate)
    elif test_3:
        optimizer = optim.Adam(net.parameters(), lr=learning_rate)
    else:
        optimizer = optim.SGD(net.parameters(), lr=learning_rate, momentum=momentum)

    # 问题4
    if test_4:
        scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
            optimizer, mode="min", factor=0.1, patience=5
        )

    for epoch in range(epochs):
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()

            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
```

```python
            running_loss += loss.item()
            if i % 200 == 199:  # 每 200 个 mini-batch 打印一次
                print(f"[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 200:.3f}")
                running_loss = 0.0

        # 任务4
        if test_4:
            scheduler.step(running_loss)

    print("Finished Training")

    # 保存模型
    PATH = "./cifar_net.pth"
    torch.save(net.state_dict(), PATH)
    print(f"Model saved to {PATH}")
```

模型

其中包含基础模型与问题要求引入的模型

```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)  # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


class Net1(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.bn1 = nn.BatchNorm2d(6)  # 1
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.bn2 = nn.BatchNorm2d(16)  # 2
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.bn3 = nn.BatchNorm1d(120)  # 3
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = torch.flatten(x, 1)  # flatten all dimensions except batch
        x = F.relu(self.bn3(self.fc1(x)))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


class Net2(nn.Module):
    def __init__(self):
        super().__init__()
```

```python
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

        # Kaiming
        for m in self.modules():
            if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
                nn.init.kaiming_normal_(m.weight, nonlinearity="relu")

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)  # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


class Net3(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 12, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(12, 32, 5)
        self.fc1 = nn.Linear(32 * 5 * 5, 240)
        self.fc2 = nn.Linear(240, 168)
        self.fc3 = nn.Linear(168, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)  # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

数据集

这里 `test_8` 本用于引入 `AutoAugment(policy=AutoAugmentPolicy.CIFAR10)` 和高斯噪声等模块，但发现效果不好后移除了，这里为保证接口不变就保留了

```python
def get_dataloaders(batch_size=64, test_2=False, test_8=False):
    if test_8:
        # 使用强数据增强策略
        transform = strong_transforms
    else:
        # 默认数据增强策略
        transform_list = [
            transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
        ]
        if test_2:
            transform_list.append(transforms.RandomHorizontalFlip())
        transform = transforms.Compose(transform_list)

    trainset = torchvision.datasets.CIFAR10(
        root="./", train=True, download=True, transform=transform
    )
    trainloader = torch.utils.data.DataLoader(
        trainset, batch_size=batch_size, shuffle=True, num_workers=2
    )

    testset = torchvision.datasets.CIFAR10(
        root="./", train=False, download=False, transform=transform
    )
    testloader = torch.utils.data.DataLoader(
        testset, batch_size=batch_size, shuffle=False, num_workers=2
    )

    classes = (
        "plane",
        "car",
        "bird",
        "cat",
        "deer",
        "dog",
        "frog",
        "horse",
        "ship",
        "truck",
    )

    print("训练集大小:", len(trainset))
    print("测试集大小:", len(testset))

    return trainloader, testloader, classes
```

测试

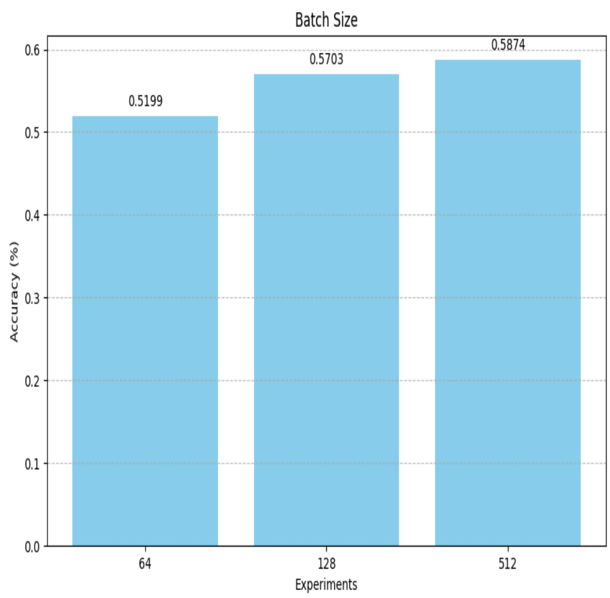
这部分基本无需改动，直接搬过来用即可，只额外加上了两个传参来进行自动化控制

```python
def test_model(batch_size=64, model=Net()):
    _, testloader, classes = get_dataloaders(batch_size)
    net = model
    PATH = "./cifar_net.pth"
    net.load_state_dict(torch.load(PATH))
    print(f"Model loaded from {PATH}")

    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(
        f"Accuracy of the network on the 10000 test images: {100 * correct // total} %"
    )

    # Accuracy for each class
    correct_pred = {classname: 0 for classname in classes}
    total_pred = {classname: 0 for classname in classes}

    with torch.no_grad():
        for data in testloader:
            images, labels = data
            outputs = net(images)
            _, predictions = torch.max(outputs, 1)
            for label, prediction in zip(labels, predictions):
                if label == prediction:
                    correct_pred[classes[label]] += 1
                total_pred[classes[label]] += 1

    for classname, correct_count in correct_pred.items():
        accuracy = 100 * float(correct_count) / total_pred[classname]
        print(f"Accuracy for class: {classname:5s} is {accuracy:.1f} %")

    return correct / total
```

1，实验测试 batch_size 是不是越大越好？可能原因有哪些？

> 不是

既然这么问了，那肯定不是的，具体可以参考经典研究。`batch_size` 是与学习率挂钩的，学习率不变的前提下，`batch_size` 在适当范围内升高可以提升训练效果，但太高了会导致学习率贡献不足、从而对样本的学习效果降低；太低了又会导致学习率对每部分样本来说太大，学到的都是噪声



大 `batch_size` 是对学习率进行了一定的缩放，对高学习率状况增大 `batch_size` 相当于对各部分减小了学习率，学习到噪声的概率降低，性能自然就上去了

这里的 `batch_size` 会传到数据提取器里的 `dataloader` 里，然后这里选中 `batch_size=64` 的情况作为后文基准，由于资源限制只测试了三组

```
batch_sizes = [64, 128, 512]
batch_size_results = []
for batch_size in batch_sizes:
    print(f"Testing with batch_size={batch_size}")
    train_model(batch_size=batch_size)
    accuracy = test_model()
    batch_size_results.append(accuracy)
    if batch_size == 64:
        base_line = accuracy

plot_and_save_results(
    batch_size_results, "Batch Size", "batch_size.png", is_batch_size=True
)
```

```
trainloader, _, _ = get_dataloaders(
    batch_size=batch_size, test_2=test_2, test_8=test_8
)
```

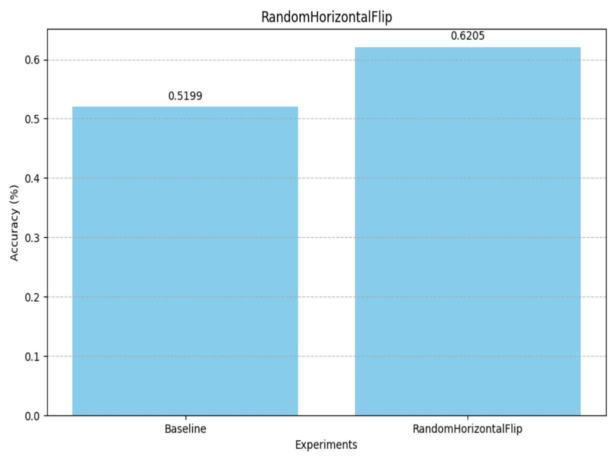
2，在训练集那里的 transform 试一下 RandomHorizontalFlip，效果会更好吗?

> 会的

这个数据增强相对来说还是比较轻量的，简单的翻转不会改变太多特征，同时还能使模型学习到更多样的训练数据。

不过由于任务要求不能改变训练轮次，那这里的数据增强也就不能弄得太强，否则还没收敛就结束了（模型太浅了），例如下图左侧是使用 `AutoAugment(policy=AutoAugmentPolicy.CIFAR10)` 数据增强的训练过程；下图右侧中的 `baseline` 则是基准 `batch_size=64` 以及未做任何处理时的准确率情况，下文同理

```
Epoch 1/10
Train Loss: 0.0043, Acc: 0.1753 | validation Loss: 0.0043, Acc: 0.2066
Epoch 2/10
Train Loss: 0.0040, Acc: 0.2369 | validation Loss: 0.0040, Acc: 0.2592
Epoch 3/10
Train Loss: 0.0039, Acc: 0.2720 | validation Loss: 0.0038, Acc: 0.2799
Epoch 4/10
Train Loss: 0.0038, Acc: 0.2944 | validation Loss: 0.0037, Acc: 0.3080
Epoch 5/10
Train Loss: 0.0037, Acc: 0.3131 | validation Loss: 0.0037, Acc: 0.3160
Epoch 6/10
Train Loss: 0.0036, Acc: 0.3286 | validation Loss: 0.0037, Acc: 0.3330
Epoch 7/10
Train Loss: 0.0035, Acc: 0.3452 | validation Loss: 0.0037, Acc: 0.3209
Epoch 8/10
Train Loss: 0.0035, Acc: 0.3496 | validation Loss: 0.0037, Acc: 0.3320
Epoch 9/10
Train Loss: 0.0035, Acc: 0.3617 | validation Loss: 0.0035, Acc: 0.3567
Epoch 10/10
Train Loss: 0.0034, Acc: 0.3645 | validation Loss: 0.0035, Acc: 0.3604
```



并且这一部分还受到模型结构的限制，本次使用的模型卷积层数较少，特征提取能力不足，数据增强做得太高也会影响性能（学不到东西）

```
train_model(test_2=True)
accuracy_flip = test_model()
```

这里在数据提取器中通过开关控制

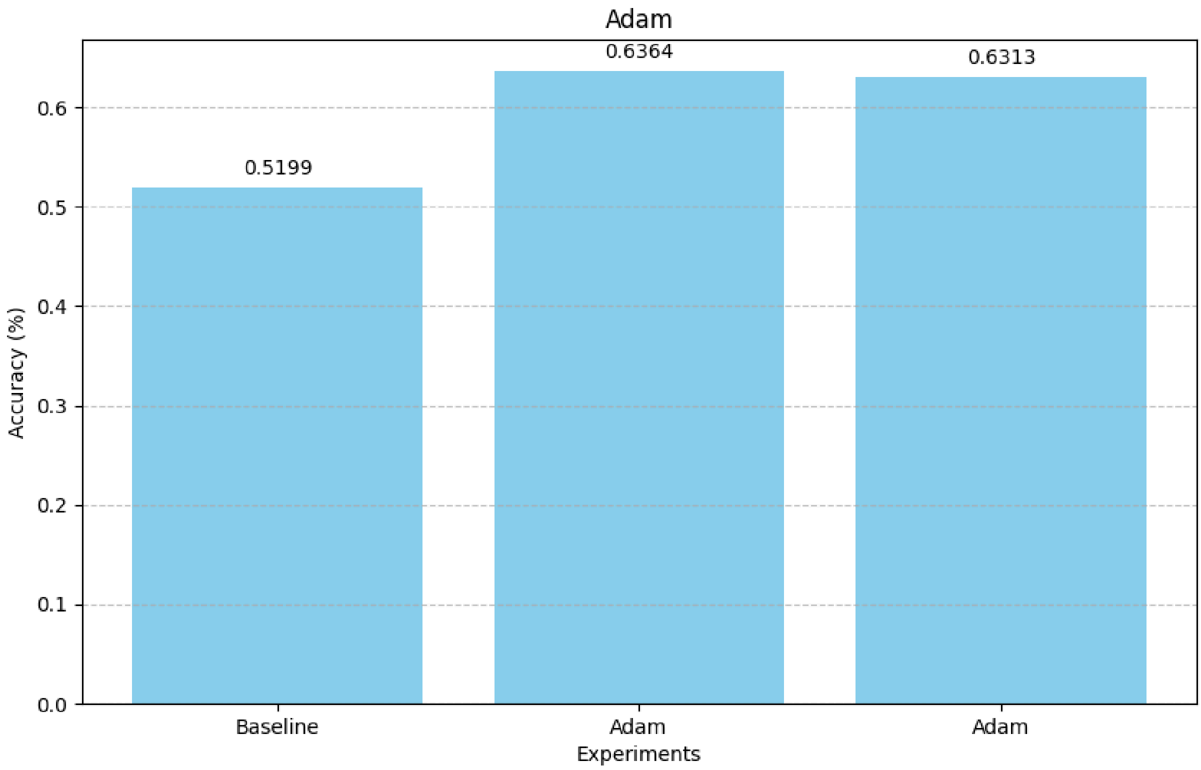
```
# 默认数据增强策略
transform_list = [
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
]
if test_2:
    transform_list.append(transforms.RandomHorizontalFlip())
transform = transforms.Compose(transform_list)
```

3，换一个 optimizer, 使效果更好一些？

> 换用了更适合本任务的 Adam，效果更好

下图中间是 `Adam` 优化器，右侧是 `AdamW` 优化器，可以看到两者都较 `SGD` 有了性能提升，而 `Adam` 的提升更大。

这两个都是自适应学习率优化算法，能动态调整学习率并加速收敛，从而在训练中更快找到较优参数； `AdamW` 则是加入了权重衰减版本的 `Adam` ，但由于当前模型太过简单，引入变成了负优化； `SGD` 虽然简单高效，但容易陷入局部最优



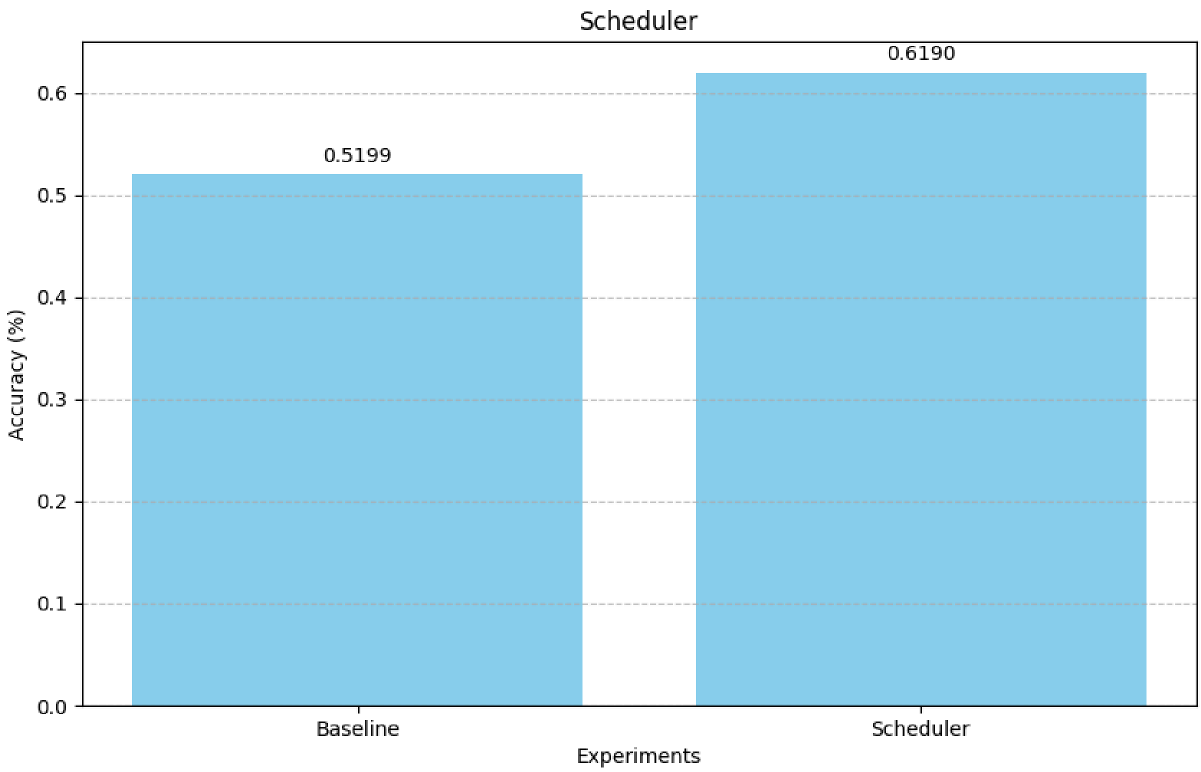
这里训练了两次之后一起画图

```
train_model(test_3=True)
accuracy_adam = test_model()
train_model(test_3_1=True)
accuracy_adamw = test_model()
```

```
if test_3_1:
    optimizer = optim.AdamW(net.parameters(), lr=learning_rate)
elif test_3:
    optimizer = optim.Adam(net.parameters(), lr=learning_rate)
else:
    optimizer = optim.SGD(net.parameters(), lr=learning_rate, momentum=momentum)
```

4，保持 epoch 数不变，加一个 scheduler，是否能让效果更好一些？

> 是



学习率过大可能会导致模型无法收敛、太小会导致训练太慢，模型前期正需要找到大方向，因此需要较高学习率；到后期权重都搞得差不多了，就需要更小的学习率去学习到更精细的东西（同时也是避免震荡）

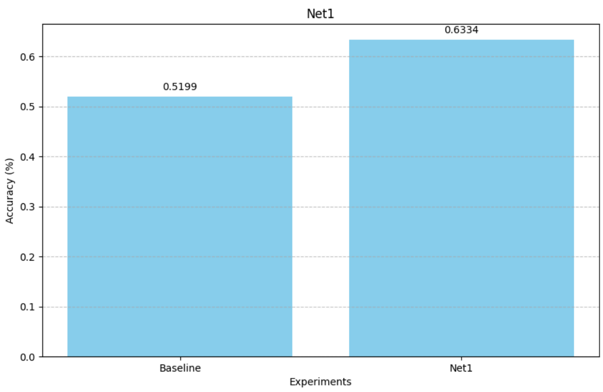
但是像在前期的时候由于都没学到什么东西、模型权重完全随机的情况下，理论上应该先弄两轮学习率预热（先把权重调整到一个比较合适的初值再大刀阔斧更新），然后再逐渐衰减，但此处由于限制训练轮次，就直接用了衰减没上预热

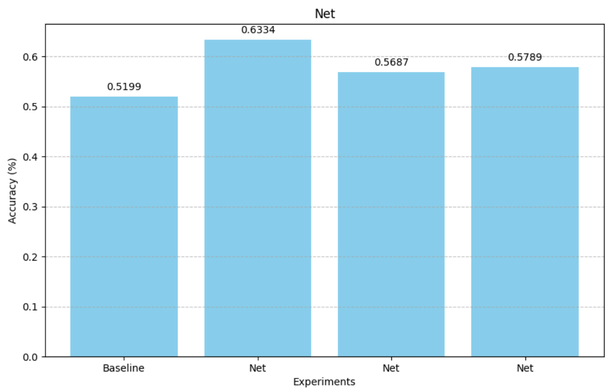
```
if test_4:
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, mode="min", factor=0.1, patience=5
    )
```

```
for epoch in range(epochs):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        if i % 200 == 199:  # 每 200 个 mini-batch 打印一次
            print(f"[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 200:.3f}")
            running_loss = 0.0
    # 任务4
    if test_4:
        scheduler.step(running_loss)
```

5，根据 Net() 生成 Net1(), 加入三个 batch_normalization 层，显示测试结果，效果是否更好?

> 是



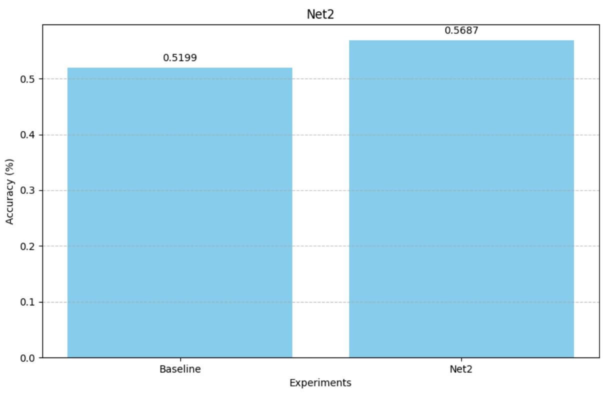


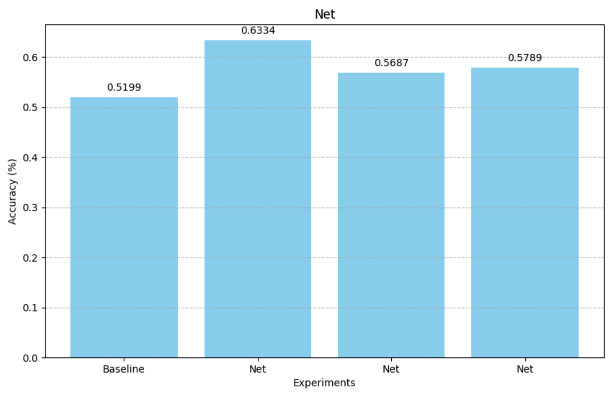
`Batch_norm`可以对每层的输入进行标准化，让训练更稳定、提升模型表达能力。同时因为模型本身比较简单，增加层数后表达能力也有所上升，也是三组里最好的

```python
class Net1(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.bn1 = nn.BatchNorm2d(6)  # 1
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.bn2 = nn.BatchNorm2d(16)  # 2
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.bn3 = nn.BatchNorm1d(120)  # 3
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = torch.flatten(x, 1)  # flatten all dimensions except batch
        x = F.relu(self.bn3(self.fc1(x)))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

6，根据 Net() 生成 Net2(), 使用 Kaiming 初始化卷积与全连接层，显示测试结果，效果是否更好?

> 是



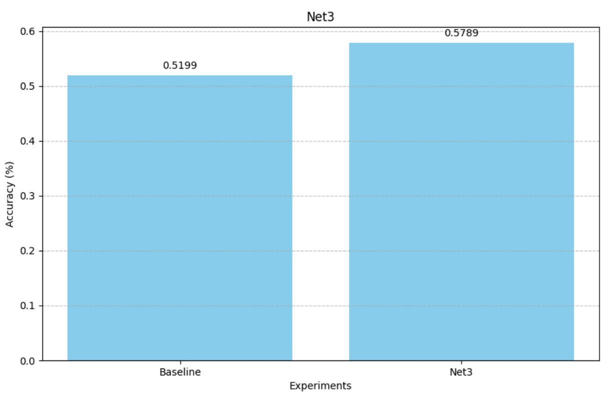


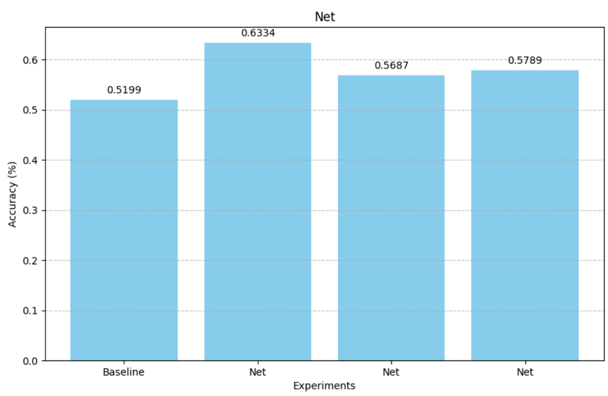
这里优化了初始权重分布，使得模型可以在一个比较好的位置进行更新，提升训练效率。但其缺乏其它复杂优化手段，相比起来不如其它两组

```python
class Net2(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

        # Kaiming
        for m in self.modules():
            if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
                nn.init.kaiming_normal_(m.weight, nonlinearity="relu")

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)  # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

7，根据 Net()生成 Net3(),将 Net()中的通道数加到原来的 2 倍，显示测试结果，效果是否更好？

> 是



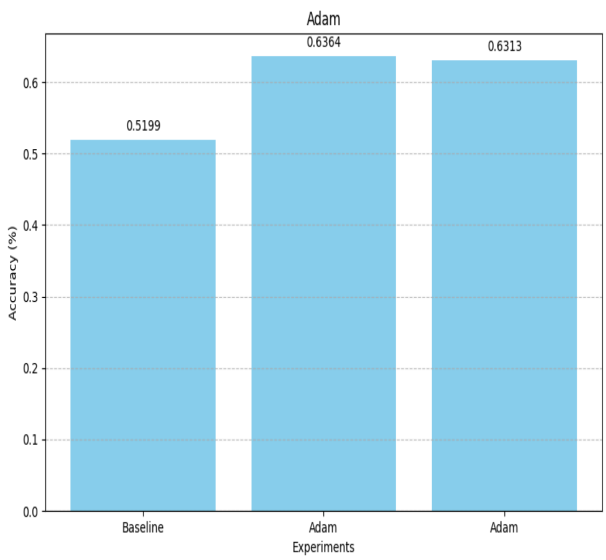


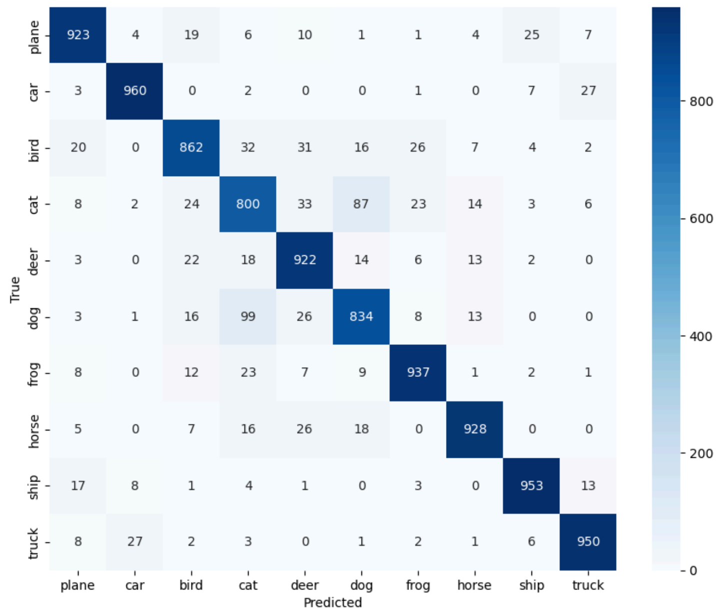
通道数加倍后模型的表达能力有所提升，但结构仍旧比较简单，对性能的提升相对有限，为三组里的中间者

```python
class Net3(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 12, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(12, 32, 5)
        self.fc1 = nn.Linear(32 * 5 * 5, 240)
        self.fc2 = nn.Linear(240, 168)
        self.fc3 = nn.Linear(168, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)  # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

8，在不改变 Net()的基础结构（卷积层数、全连接层数不变）和训练 epoch 数的前提下，你能得到最好的结果是多少?

> `0.6364` ，即上文 `Adam` 优化器跑出的结果，也下图左侧中间
> 下图右侧是我之前使用其它模型结构与训练策略得出的模型结果(0.9069)





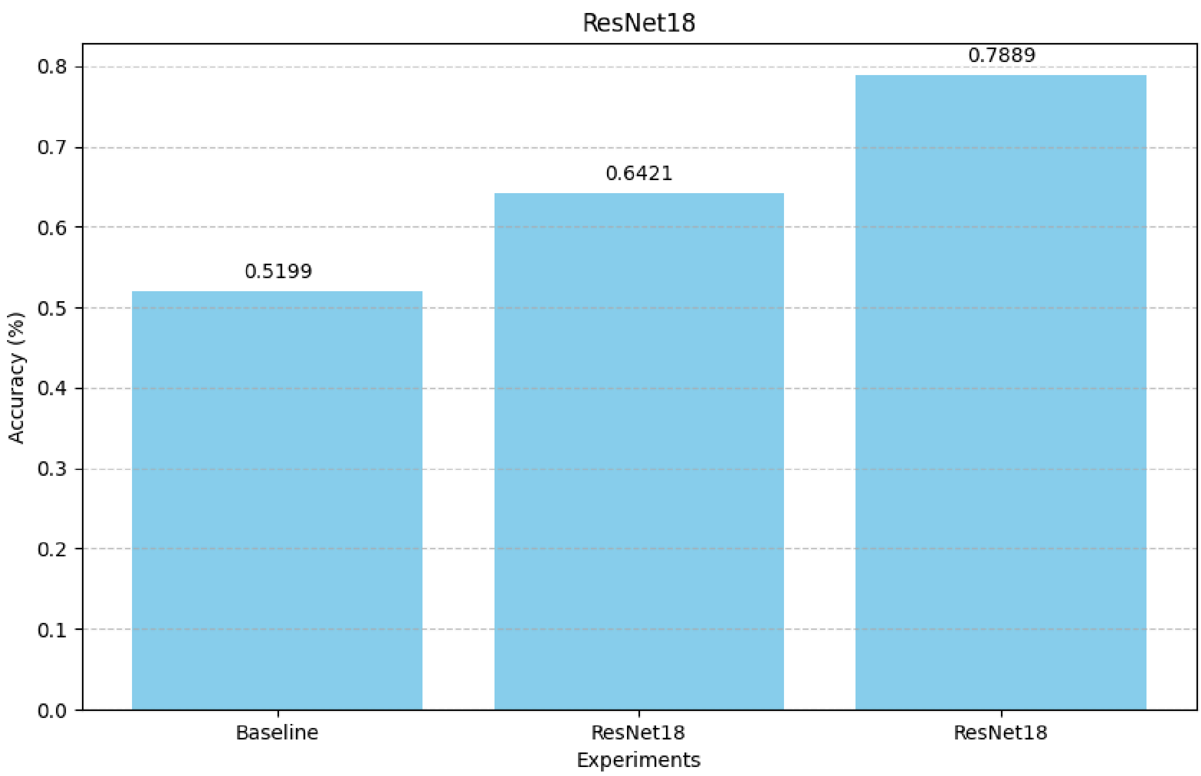
这里其实也有放很多组测试，但耗时太长没跑完，只能用上面的内容（跑了几个小时后的我运行了另一个实验的代码，由于 `batch_size` 没估计好内存爆了，上个厕所回来这个进程已经被 `kill` 掉了）。另外我还尝试过引入数据增强，除自动增强外高斯噪声和随机裁剪也都引入过，但效果都比较一般，模型结构太简单了而且这个训练轮数限制有点大

```python
net = Net()
# 定义每个参数的列表
base_line = 0.5199
batch_sizes = [8, 16, 32, 64, 128, 256, 512, 1024]
test_2_options = [True, False]
test_3_options = [True, False]
test_4_options = [True, False]
learning_rates = [0.05, 0.01, 0.005, 0.001, 0.0005]
best_results = []
# 测试所有参数组合
for batch_size in batch_sizes:
    for test_2 in test_2_options:
        for test_3 in test_3_options:
            for test_4 in test_4_options:
                for lr in learning_rates:
                    print(
                        f"Testing combination: batch_size={batch_size}, test_2={test_2}, test_
                    )
                    # 训练模型
                    train_model(
                        batch_size=batch_size,
                        test_2=test_2,
                        test_3=test_3,
                        test_4=test_4,
                        learning_rate=lr,
                        model=net,
                    )
                    # 测试模型并记录结果
                    accuracy = test_model(model=net)
                    best_results.append(
                        {
                            "batch_size": batch_size,
                            "test_2": test_2,
                            "test_3": test_3,
                            "test_4": test_4,
                            "lr": lr,
                            "accuracy": accuracy,
                        }
                    )
# 按准确率升序排序
best_results = sorted(best_results, key=lambda x: ["accuracy"])
# 打印所有测试结果（按升序）
print("\nAll Results (sorted by accuracy):")
for result in best_results:
    print(
        f"batch_size={result['batch_size']}, test_2={result['test_2']}, test_3={result['test_3
```

```
    )
# 取准确率最高的前五种组合
top_5_results = best_results[-5:]
# 准备绘图数据
labels = ["base_line"] + [f"Top {i+1}" for i in rang(len(top_5_results))]
accuracies = [base_line] + [result["accuracy"] forresult in top_5_results]
# 绘制结果
plot_and_save_results(
    accuracies,
    "Top 5 Parameter Combinations vs Base Line",
    "top_5_combinations.png",
    labels=labels,
)
```

9，使用 ResNet18(),显示测试结果

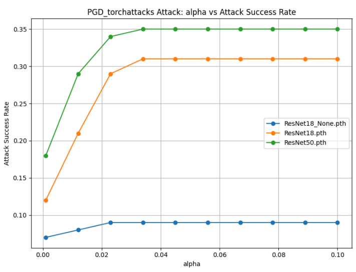
> 带预训练权重的（0.7889）比不带的强（0.6421），但两者都比基准模型强（0.5199）

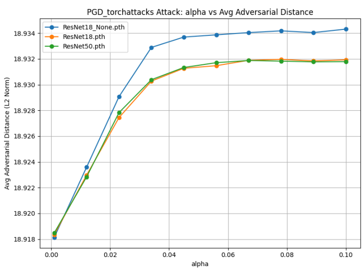


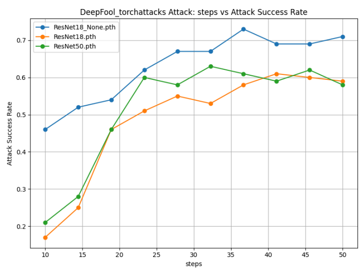
这个其实很好理解，`ResNet`系列权重是在`ImageNet`上预训练的，已经学习到了非常丰富的特征，放到这种经典简单分类任务上效果自然会好。带权重相当于迁移学习，那些已经学习到的特征同样适用，迁移后只需微调一小部分即可快速适应新任务（不过这里还是全部更新）

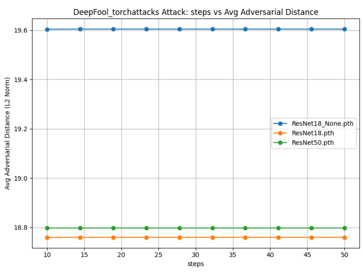
```python
resnet18 = models.resnet18(weights=None, num_classes=10)
train_model(model=resnet18)
accuracy_resnet18 = test_model(model=resnet18)

resnet18_1 = models.resnet18(weights=models.ResNet18_Weights.IMAGENET1K_V1)
resnet18_1.fc = nn.Linear(resnet18_1.fc.in_features, 10)
train_model(model=resnet18_1)
accuracy_resnet18_1 = test_model(model=resnet18_1)
plot_and_save_results(
    [base_line, accuracy_resnet18, accuracy_resnet18_1], "ResNet18", "resnet18.png"
)
```

另外我最近在做相关对抗攻击，发现在`不带权重从头预训练`与`带权重仅微调最后layer4与分类头`的干净样本准确率相同的前提下（`99%+`），前者对于对抗攻击表现得更鲁棒（`PGD`、`FGSM`都非常鲁棒（从头训练的模型在梯度分布上更为平滑，更新所有权重可以让模型更专注于对目标数据特征的学习（而不仅依赖于预训练权重中已有的特征），当前目标数据与预训练数据中的特征大概率存在偏差，而这部分偏差就很可能被对抗攻击利用到），而`DeepFool`（从头训练的模型数据集太小，决策边界太复杂从而更容易被找到弱点；而迁移微调模型继承了预训练权重的通用特征，在决策边界上更为平滑与稳定）的攻击太强了几种情况都不太鲁棒）









进而可以得出结论：若仅需在当前数据上表现更好、同时希望有更好的表达能力，最好使用预训练的权重进行微调；如果想要增强一些对于梯度攻击抵抗能力，可以从头进行训练（费时费力，但抵抗效果更好）；而如果想进一步再抵抗决策边界的攻击，可以尝试微调分层学习率（比如具体数值、应该怎么衰减、是否冻结前几层），可以预见其将结合两种策略的优点，同时对梯度与决策边界的攻击都有更好的鲁棒性